A Verified Compiler for Multithreaded PreScheme[1]

William M. Farmer John D. Ramsdell

January 1996

[1]This work was supported by the United States Army CECOM under contract DAAB07-94-C-H601, through MITRE's Technology Program.

Authors' address: The MITRE Corporation, 202 Burlington Road, Bedford MA, 01730-1420.

E-mail: {farmer,ramsdell}@mitre.org.

Abstract

Multithreaded PreScheme is a systems programming language for high-assurance systems. It is based on VLISP PreScheme, a systems programming language dialect of Scheme developed by the VLISP project. Multithreaded PreScheme is an extension of VLISP PreScheme which allows programs with more than one thread of control. This document describes the language and also a verified compiler which generates code for MIPS-based architectures. The two major contributions described in this report are a proof of the correctness of code generated for execution in the presence of multiple threads, and a new code generator that produces substantially faster code than that produced by the verified compiler for VLISP PreScheme.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

PreScheme, a restricted dialect of Scheme, was invented by J. Rees and R. Kelsey [11, Section 4] for systems programming. PreScheme was defined so that programs can be executed using only a C-like run-time system. A compiler for PreScheme will reject any program that requires run-time type checking, and the run-time system need not provide automatic storage reclamation. PreScheme was carefully designed so that syntactically it looks like Scheme and has a similar semantics. With a little care, PreScheme programs can be run and debugged as if they were ordinary Scheme programs.

Multithreaded PreScheme is an extension of VLISP PreScheme [16], a version of the PreScheme language developed under the VLISP project [7]. It is designed to be a systems programming language for high-assurance systems. Unlike VLISP PreScheme, it allows programs with more than one thread of control. This paper describes the Multithreaded PreScheme language and an optimizing compiler for the language that generates MIPS Assembly Language [9]. The language is given a formal denotational-style semantics and the compiler is verified against this semantics.

An overview of the formal semantics of Multithreaded PreScheme is presented in a separate paper [3]. It provides a good introduction to the motivation and key ideas behind semantics.

The compiler is a program written in Scheme consisting of five stages:

1. From Multithreaded PreScheme to Macro-Free PreScheme.

2. From Macro-Free PreScheme to Simple PreScheme.

3. From Simple PreScheme to Stack Assembly Language.

4. From Stack Assembly Language to PreScheme Assembly Language.

5. From PreScheme Assembly Language to MIPS Assembly Language.

The three languages Multithreaded PreScheme, Macro-Free PreScheme, and Simple PreScheme together with their denotational semantics are presented in Chapter 3. Stack Assembly Language and PreScheme Assembly Language are described in Chapter 5 and Chapter 7, respectively.

The first stage (described in Chapter 4) translates a Multithreaded PreScheme program into a Macro-Free PreScheme program by expanding derived syntax, changing bound variables, and inlining primitive operators. These transformations are straightforward and there is little to verify.

The second stage (also described in Chapter 4) attempts to translate a Macro-Free PreScheme program into an equivalent Simple PreScheme program by applying a certain set of transformations. We prove that each of these transformations is meaning refining. The syntactic restrictions of Simple PreScheme allow the generation of efficient code using a syntax-directed compiler (the third stage). The second stage does not always succeed; consequently, the compiler does not accept the entire Multithreaded PreScheme language.

The third stage (Chapter 6) is a syntax-directed compiler that translates Simple PreScheme into Stack Assembly Language, an abstract machine language. A denotational semantics is defined for Stack Assembly Language using the same domains as in the denotational semantics for Simple PreScheme. The correctness of the syntax-directed compiler is justified relative to the denotational semantics. An operational semantics is also defined for Stack Assembly Language which is faithful to its denotational semantics.

The fourth stage (Chapter 8) is a code generator that produces PreScheme Assembly Language from Stack Assembly Language. The operational semantics of the output of the code generator is shown to be a refinement of the operational semantics of its input.

Finally, the fifth stage (Chapter 9) is an unverified program that translates the PreScheme Assembly Language into MIPS Assembly Language.

The paper ends with a conclusion listing the major parts of the Multithreaded PreScheme system that are not described or verified in this paper.

$\langle x_1, \ldots, x_n \rangle$	sequence formation
$(x_1, \ldots, x_n)$	tuple formation
$s \downarrow k$	kth member of the sequence or tuple s (1-based)
$\#s$	length of the sequence or tuple s
$s \S\, t$	concatenation of sequences s and t
$s \dagger k$	the sequence s minus the first k members
$t \rightarrow a, b$	McCarthy conditional "if t then a else b"
$x \text{ in } D$	injection of x into domain D
$x \mid D$	projection of x to domain D
$x \in D$	x is an element of domain D or $(x \mid D) \neq \bot_D$
$A[y/x]$	substitution "A with y for x"

Table 1.1: Notation

1.1 Notation

The presentation of the formal denotational semantics uses the notation given in Table 1.1. It is similar to the notation used in the IEEE Standard for the Scheme Programming Language [8, Appendix A]. We will also use a notation like Scheme's `let*` notation: an expression of the form

$$\begin{array}{l} \text{let} \\ \quad x_1 = e_1 \\ \quad x_2 = e_2 \\ \qquad \vdots \\ \quad x_n = e_3 \\ \text{in} \\ \quad b \end{array}$$

is an abbreviation for

$$(\lambda x_1 .\, (\lambda x_2 .\, \ldots\, (\lambda x_n .\, b) e_n\, \ldots) e_2) e_1 .$$

1.2 Acknowledgments

The oracle-based semantics for Multithreaded PreScheme was developed jointly by the authors and Joshua Guttman. Guttman also contributed many valuable suggestions for improving the paper. Leonard Monk, Jonathan Rees, and Vipin Swarup provided helpful comments on earlier versions of the paper.

Chapter 2

Multithreaded Semantics for PreScheme

In the paper on VLISP PreScheme [16], a PreScheme program is intended to be executed in an address space with a single thread of control. In other words, a VLISP PreScheme program is intended to be executed by a "single-threaded process." In contrast, a Multithreaded PreScheme (MTPS) program is intended to be executed by a multithreaded process in which there are possibly several threads of control in a common address space, but with separate register sets, stacks, and program counters.

Consider a process which executes an MTPS program. When the process is started, it has a single thread of control called the *root thread*. The root thread may request the process to create new threads which may, in turn, request the creation of other threads, and so on. Each thread created by the process will have a unique thread ID. At any given moment, the process will possess a set of threads that are active (i.e., that have been created and have not yet terminated). This set will be some subset of the total set of threads created by the process (which may be potentially infinite if the process continuously creates new threads).

A portion of the process's memory, called its *shared memory*, is shared among the active threads of the process. For each thread, there is also a portion of the process's memory, called the *thread-local memory*. Even though the address space of the process is common to all the threads of the process, the MTPS language guarantees that a thread can access only shared memory and its own thread-local memory, but not the thread-local memory of another thread.

An MTPS program differs from a VLISP PreScheme program in two principal ways. First, an MTPS program may utilize special primitive operators that create new threads and that allow interaction between threads. Let us call these primitive operators *interthread operators.* The following are some of the operations that interthread operators might perform:

- Creation and initialization of a new thread.
- Suspension, resumption, and termination of a thread (possibly the calling thread itself).
- Manipulation of a shared data structure (i.e., a data structure which is stored in shared memory).
- Locking and unlocking access to a shared data structure.
- Synchronization with other threads.

Second, the formal semantics of an MTPS program is slightly more complicated than the formal semantics of a VLISP PreScheme program. The denotational semantics of VLISP PreScheme is derived from Scheme's semantics [8, Appendix A]. The denotational semantics of MTPS contains special machinery for specifying how the threads created during the execution of a program interact with each other. In the semantics of MTPS, the manipulation of data structures which are stored in the thread-local memory of a thread is modeled in exactly the same way as in the semantics of VLISP PreScheme. However, the manipulation of shared data structures and other kinds of interthread actions are modeled using the idea of an *oracle.*

Suppose P is an MTPS program. An oracle represents a prediction about the sequence of I/O and interthread actions performed by P. The meaning of P is characterized by its behavior with respect to all possible oracles. One virtue of this approach is that it allows the meaning of P to be decomposed into the meanings of the threads that are created during its execution. Another virtue of this approach is that, if P contains no I/O or interthread operators, its meaning is essentially the same as the meaning of a corresponding VLISP PreScheme program.

To highlight exactly how the semantics of MTPS differs from the semantics of VLISP PreScheme, we actually define two flavors of the MTPS semantics. The *thread-unaware semantics* of MTPS is essentially the same as the semantics of VLISP PreScheme; the thread-unaware meaning of an MTPS program does not utilize any of the special machinery for threads. The *thread-aware semantics* is an extension of the *thread-unaware semantics*; the thread-aware meaning of an MTPS program is composed of the meanings of its threads.

Since there are many different, often incompatible, approaches for controlling how threads interact with each other, we have not attempted to provide a complete set of interthread operators. Instead, we have added to MTPS interthread operators to create new threads, to wait on the termination of other threads, and to destroy threads:

```
create-thread
join-thread
exit-thread
exit
```

We have also added some interthread operators that manipulate shared vectors (i.e., vectors which are stored in shared memory and which thus may be manipulated by all of the threads of the process):

```
make-shared-vector
shared-vector-ref
shared-vector-set!
shared-vector-access
shared-vector-modify!
```

The semantics for these interthread operators and for MTPS programs is discussed below; a formal presentation of the semantics is given in the next chapter. The semantics illustrates how one would define the semantics for other interthread operators that manipulate shared data structures or perform other kinds of interthread actions. The semantics for the individual I/O operators is not considered in this report.

2.1 Oracles

As we mentioned above, an oracle is used to specify I/O and the interaction between the threads of a process executing an MTPS program. It represents the order and outcome of the I/O and interthread operator calls that take place during the execution of the program. Mathematically, an oracle is a pair consisting of a sequence of tuples (modeled as a function from the natural numbers to a cartesian product) plus a natural number. The tuples are called the *oracle entries* and the natural number is called the *oracle index*. Each oracle entry represents either a call to an I/O or interthread operator by one of the threads or the return of a thread's answer to the process. The oracle index designates which entry in the oracle is currently being considered.

Threads are referenced by their thread ID. The root thread has a distinguished ID named *root_tid*. The *thread* ID of an oracle entry determines the thread that is calling an operator or returning an answer. The *tag* of an entry determines what action is being performed. The tag `an` designates that an answer is returned; the other tags designate which of the I/O or interthread operators is called. Depending on its tag, an entry may also have a number of other meaningful fields. An entry with tag `an` has one meaningful field which designates the answer that is returned. An entry with a operator call tag has fields related to the arguments of the operator call and the results produced by the call.

In this paper, oracle entries are formalized as tuples, where components are shared among the different interthread operators. Other, more abstract, representations are equally possible.

In the MTPS semantics, a command continuation takes a store and a oracle and returns an answer. (It takes only a store in the VLISP PreScheme semantics.) The oracle is passed along unchanged except when an I/O or interthread operator O is called by the current thread (whose thread ID is, say, ι). Then the first entry in the oracle at or after the oracle's index with thread ID ι and the tag corresponding to O is retrieved and compared with the operator call. If the components of the entry correctly match the arguments of the operator, the oracle's index is incremented and the component of the entry that represents the result of the operator call, the oracle with this new index, and the store are passed to the current expression continuation; otherwise, an error is generated. If an entry with tag `ex` (which designates a call to the `exit` operator) is discovered in the process of looking for this entry, the argument to the call, the oracle, and the store are passed to the (thread-aware) initial expression continuation which results in the argument being returned as the answer of all the active threads.

The semantics of an MTPS program P is defined relative to an oracle. Fix some oracle o. Then, given o, the thread-unaware semantics of P is an answer which does not depend on o, but the thread-aware semantics of P is a function that maps thread IDs to answers which do depend on o.

2.2 Normal Oracles

Since the meaning of an MTPS program P in the thread-aware semantics is relative to an oracle, we will need to characterize which oracles are possible runs of P. We will say that a program *accepts* an oracle if the oracle represents the interthread activity of a possible execution of the program. One program may accept many oracles, for instance because the order in which different threads execute may vary from run to run. Moreover, one oracle may be accepted by more than one program. On the other hand, some oracles are not accepted by any program at all. An oracle is *acceptable* if some program accepts it.

To exclude a broad class of unacceptable oracles, we introduce the notion of a "normal oracle." An oracle o is *normal* if it satisfies the following conditions:

1. *Threads have distinct thread* IDs*.* For each thread ID ι, there is at most one entry in o that represents an interthread operator call that creates a thread with ID ι.

2. *Threads do not exist before they are created, or after they exit.* For all entries η in o, if η represents an interthread operator call by a thread T that is not the root thread, then there is an earlier entry in o representing an interthread operator call that creates T. Moreover, if η represents a thread T returning an answer, then there is no later entry in o representing an interthread operator call by T.

3. *Shared data structures do not exist before they are created.* For all entries η in o, if η represents an interthread operator call that manipulates a shared data structure S, then there is an earlier entry in o representing an interthread operator call that creates S.

4. *Shared data structures can only be modified by interthread operator calls.* For all entries η_1 and η_2 in o, if η_2 references a shared data structure S, and η_1 is the last previous interthread operator call that modifies S, then the value supplied in η_1 is the same value retrieved by η_2.

The semantics of an interthread operator can place further restrictions on the class of acceptable oracles. These restrictions can be incorporated into the definition of a normal oracle. For example, the following condition—which is derived from the semantics of `join-thread` given below—could be added to the list of conditions that normal oracles must satisfy:

(5) *Threads calling* ***`join-thread`*** *block until they receive an answer.* For all entries η in o, if η represents a `join-thread` call by a thread T whose argument is the thread ID of a thread T', then there is a entry η' in o after η which represents T' returning an answer such that:

  - The answer contained in η is equal to the answer contained in η'.
  - There are no entries in o between η and η' which have the thread ID of T.

2.3 Semantics of the Interthread Operators

This section describes the semantics of the interthread operators that are currently in MTPS.

2.3.1 Semantics of `create-thread`

The MTPS primitive operator `create-thread` is used to create new threads. The operator takes a procedure ϕ (which maps integers to integers) and an integer ζ, and returns a thread ID ι_0. As a side effect, a new thread is created whose thread ID is ι_0 and whose initial store is a copy of the current store of the calling thread (except at the location in the store which holds the ID of the thread). The new thread applies ϕ to ζ and returns the result of the application as its answer. An oracle entry representing a call of `create-thread` has the form

$$(\mathsf{ct}, \iota, x_1, \epsilon, \phi, \sigma, x_2, x_3)$$

where $(\phi \mid E)$ and ϵ (with $\epsilon \in R$) are the two arguments of the call, σ is the current store of the calling thread, and x_1, x_2, x_3 are not meaningful.

2.3.2 Semantics of `join-thread`

The primitive operator `join-thread` is used to wait on the termination of another thread. The operator takes a thread ID ι' and returns the answer produced by the thread whose thread ID is ι'. As a side effect, the calling thread blocks until the answer is returned. An oracle entry representing a call of `join-thread` has the form

$$(\mathsf{jt}, \iota, \iota', \epsilon, x_1, x_2, x_3, x_4)$$

where $(\iota' \mid E)$ is the argument of the call, ϵ (with $\epsilon \in A$) is the answer that is returned, and x_1, x_2, x_3, x_4 are not meaningful.

2.3.3 Semantics of `exit-thread` and `exit`

There are two primitive operators for terminating threads: `exit-thread` and `exit`.

The operator `exit-thread` takes an integer ζ and, as a side effect, the calling thread halts and returns ζ as its answer. An oracle entry representing a call of `exit-thread` has form

$$(\mathsf{et}, \iota, x_1, \epsilon, x_2, x_3, x_4, x_5)$$

where ϵ (with $\epsilon \in R$) is the argument of the call and x_1, x_2, x_3, x_4, x_5 are not meaningful.

The operator `exit` takes an integer ζ and, as a side effect, the calling thread halts and returns ζ as its answer and then each remaining active thread likewise halts and returns ζ as its answer. An oracle entry representing a call of `exit` has form

$$(\mathsf{ex}, \iota, x_1, \epsilon, x_2, x_3, x_4, x_5)$$

where ϵ (with $\epsilon \in R$) is the argument of the call and x_1, x_2, x_3, x_4, x_5 are not meaningful.

2.3.4 Semantics of Shared Vector Operators

MTPS has five interthread operators for manipulating shared vectors:

- `make-shared-vector` takes a nonnegative integer ζ, and returns a shared vector $\mathbf{V}$ of length ζ.

- `shared-vector-ref` takes a shared vector $\mathbf{V}$ and an integer ζ with $0 \leq \zeta < \#\mathbf{V}$, and returns the value stored in the ζ-th word of $\mathbf{V}$.

- `shared-vector-set!` takes a shared vector $\mathbf{V}$, an integer ζ with $0 \leq \zeta < \#\mathbf{V}$, and a value v, and returns an unspecified value after storing v in the ζ-th word of $\mathbf{V}$.

- `shared-vector-access` takes a shared vector $\mathbf{V}$ and a pointer to the ζ-th word of $\mathbf{V}$, and returns the value stored in the ζ-th word of $\mathbf{V}$.

- `shared-vector-modify!` takes a shared vector $\mathbf{V}$, a pointer to the ζ-th word of $\mathbf{V}$, and a value v, and returns an unspecified value after storing v in the ζ-th word of $\mathbf{V}$.

The latter two operators allow pointer arithmetic (as in the C language) to be used in conjunction with accessing and modifying shared vectors. They are related to `shared-vector-ref` and `shared-vector-set!` by the following two equations:

$$\begin{aligned}&(\texttt{shared-vector-access}\ \mathbf{V}\ p) = \\ &\quad(\texttt{shared-vector-ref}\ \mathbf{V}\ (\texttt{addr-}\ p\ (\texttt{shared-vector->addr}\ \mathbf{V})))\end{aligned}$$

$$\begin{aligned}&(\texttt{shared-vector-modify!}\ \mathbf{V}\ p\ v)\ = \\ &\quad(\texttt{shared-vector-set!}\ \mathbf{V}\ (\texttt{addr-}\ p\ (\texttt{shared-vector->addr}\ \mathbf{V}))\ v)\end{aligned}$$

An oracle entry representing a call of a shared vector operator has the form

$$(t, \iota, x_1, \epsilon, x_2, x_3, \beta, \mathbf{V})$$

where $t \in \{\mathsf{mv}, \mathsf{vr}, \mathsf{vs}, \mathsf{va}, \mathsf{vm}\}$ and x_1, x_2, x_3 are not meaningful. The roles of ϵ, β, and $\mathbf{V}$ depend on the value of t:

- If $t = \mathsf{mv}$, `make-shared-vector` is called. In this case, $\mathbf{V}$ must be a shared vector of length $(\epsilon_1 \mid R)$, where ϵ_1 is the argument of the call. (ϵ and β play no role.)

- If $t = \mathsf{vr}$, `shared-vector-ref` is called. In this case, β indicates the location of the word in $\mathbf{V}$ that is referenced and ϵ represents the contents of the referenced location. If ϵ_1 and ϵ_2 are the arguments of the call, then $\mathbf{V} = (\epsilon_1 \mid E_{sv})$ and $\beta = \mathbf{V} \downarrow ((\epsilon_2 \mid R) + 1)$.

- If t = `vs`, `shared-vector-set!` is called. In this case, β indicates the location of the word in $\mathbf{V}$ that is referenced and ϵ represents the contents to be stored in referenced word. If ϵ_1, ϵ_2, and ϵ_3 are the arguments of the call, then $\mathbf{V} = (\epsilon_1 \mid E_{sv})$, $\beta = \mathbf{V} \downarrow ((\epsilon_2 \mid R) + 1)$, and $\epsilon = \epsilon_3$.

- If t = `va`, `shared-vector-access` is called. In this case, β indicates the location of the word in $\mathbf{V}$ that is referenced and ϵ represents the contents of the referenced location. If ϵ_1 and ϵ_2 are the arguments of the call, then $\mathbf{V} = (\epsilon_1 \mid E_{sv})$ and β is the shared location pointed to by $(\epsilon_2 \mid R)$.

- If t = `vm`, `shared-vector-modify!` is called. In this case, β indicates the location of the word in $\mathbf{V}$ that is referenced and ϵ represents the contents to be stored in referenced word. If ϵ_1, ϵ_2, and ϵ_3 are the arguments of the call, then $\mathbf{V} = (\epsilon_1 \mid E_{sv})$, β is the shared location pointed to by $(\epsilon_2 \mid R)$, and $\epsilon = \epsilon_3$.

Chapter 3

The Multithreaded PreScheme Language Family

This chapter describes Multithreaded PreScheme (MTPS), a systems programming oriented Scheme dialect, and related dialects. MTPS is an extension of VLISP PreScheme [16], which was developed under the VLISP project [7].

There are three languages in MTPS language family: MTPS, Macro-Free PreScheme, and Simple PreScheme. Programs are written in MTPS. Macro-Free PreScheme is the language manipulated by the transformational compiler. Simple PreScheme is a restricted subset of Macro-Free PreScheme which is the source language of the syntax-directed compiler.

3.1 Multithreaded PreScheme

MTPS programs manipulate data objects that fit in machine words. A data object may be an integer, a character, a boolean, a string, a port, a thread ID, a pointer to an integer, an unshared vector of integers, a shared vector of integers, or a procedure (the last three are really pointers to the aforementioned objects). A compiler must ensure that operators are not applied to data of the wrong type since there are no run-time checks.

A running MTPS program can only manipulate pointers to a restricted class of procedures. The free variables of these procedures must be allocated at compile time. This restriction eliminates the need to represent closures at run-time. Since MTPS programs may contain procedures with free variables that are `lambda` bound, the compiler must transform these programs so that they meet the run-time restriction.

As with Scheme, implementations of MTPS are required to be tail-recursive, which means that iterative processes can be expressed by means of procedure calls. When the last action taken by a MTPS procedure is a call, tail-recursive implementations are required to eliminate the control information of the calling procedure so that the order of space growth of iterative processes is constant. Tail-recursive calls will be the name given to procedure calls which are the last action performed by a procedure.

3.1.1 Syntax

The syntax of the MTPS language is identical to the syntax of the language defined in the Scheme standard [8, Chapter 7] with the following exceptions:

- Every defined procedure takes a fixed number of arguments.

- The only variables that can be modified are those introduced at top level using the syntax

  (`define` ⟨variable⟩ ⟨expression⟩),

  and whose name begins and ends with an asterisk and is at least three characters long. Variables so defined are called *mutable variables*. Note that a variable introduced at other than top level may have a name which begins and ends with an asterisk, but this practice is discouraged.

- If ⟨expression⟩ is a `lambda` expression, variables can also be defined using the syntax

  (`define-integrable` ⟨variable⟩ ⟨expression⟩).

  When ⟨variable⟩ occurs in the operator position of a combination, compilers must replace it with ⟨expression⟩.

- No variable may be defined more than once.

- `letrec` is not a derived expression. The variables bound by a `letrec` expression must be distinct, and the initializer for each such variable must be a `lambda` expression.

Classification	Abbreviation
Invariable	inv
Side-effect free	sef
Changes store	chs
Accesses oracle	aco

Table 3.1: Primitive Operator Classifications

- Constants are restricted to integers, characters, booleans, and strings.
- Finally, a different set of standard operators has been specified.

A "variable" is simply an identifier; hence, there may be more than one *occurrence* of a variable in a program. A variable is *bound* in an MTPS (well-formed) program fragment if (1) it is occurs immediately after `define` or `define-integrable`, (2) it is in the list of identifiers after `lambda`, or (3) it is the first component of one of the pairs in the list after `letrec`. A variable is *free* in an MTPS program fragment if it is not bound.

3.1.2 Standard Operators

The MTPS standard primitive operators[1] are in the list below. The text on the left of each entry gives the operator's name and its type at one fixed arity. The type language is defined in Section 3.2.3, p. 40. (The types $\bullet\,\mathrm{Int}$, $\bullet_s\,\mathrm{Int}$, $\star\,\mathrm{Int}$ are like the pointer types in C.) The text on the right describes the arity and the classification of the primitive operator. The classification types are given in Table 3.1; they are explained in Chapter 4. An implementation of some of the primitive operators in the C programming language is given in Appendix A. The semantics of each primitive operator is intended to correspond to its implementation in C. The set of interthread operators in the language (see Chapter 2) is not meant to be complete; additional interthread operators will be added later.

The MTPS standard primitive operators are:

`<` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$	2 or more	inv

[1]The "standard primitive operators" of MTPS correspond to the "standard procedures" in the Scheme standard [8].

`<=` : Int × Int → Bool	2 or more	inv
`=` : Int × Int → Bool	2 or more	inv
`>=` : Int × Int → Bool	2 or more	inv
`>` : Int × Int → Bool	2 or more	inv
`abs` : Int → Int	1	inv
`+` : Int × Int → Int	any	inv
`-` : Int × Int → Int	1 or 2	inv
`*` : Int × Int → Int	any	inv
`quotient` : Int × Int → Int	2	inv
`remainder` : Int × Int → Int	2	inv
`ashl` : Int × Int → Int	2	inv
`low-bits` : Int × Int → Int	2	inv
`integer->char` : Int → Chr	1	inv
`char->integer` : Chr → Int	1	inv
`char=?` : Chr × Chr → Bool	2	inv
`char<?` : Chr × Chr → Bool	2	inv
`make-vector` : Int → ● Int	1	chs
`vector-ref` : ● Int × Int → Int	2	sef
`vector-set!` : ● Int × Int × Int → Int	3	chs
`vector-access` : ● Int × ⋆ Int → Int	2	sef
`vector-modify!` : ● Int × ⋆ Int × Int → Int	3	chs
`make-shared-vector` : Int → $\bullet_s$ Int	1	aco
`shared-vector-ref` : $\bullet_s$ Int × Int → Int	2	aco
`shared-vector-set!` : $\bullet_s$ Int × Int × Int → Int	3	aco
`shared-vector-access` : $\bullet_s$ Int × ⋆ Int → Int	2	aco
`shared-vector-modify!` : $\bullet_s$ Int × ⋆ Int × Int → Int	3	aco
`vector-byte-ref` : ● Int × Int → Int	2	sef
`vector-byte-set!` : ● Int × Int × Int → Int	3	chs
`addr<` : ⋆ Int × ⋆ Int → Bool	2	inv
`addr=` : ⋆ Int × ⋆ Int → Bool	2	inv
`addr+` : ⋆ Int × Int → ⋆ Int	2	inv
`addr-` : ⋆ Int × ⋆ Int → Int	2	inv
`vector->addr` : ● Int → ⋆ Int	1	inv
`shared-vector->addr` : $\bullet_s$ Int → ⋆ Int	1	inv
`addr->integer` : ⋆ Int → Int	1	inv
`integer->addr` : Int → ⋆ Int	1	inv
`addr->string` : ⋆ Int → String	1	sef
`string->addr` : String → ⋆ Int	1	sef

`port->integer` : Port $\to$ Int	1	inv
`integer->port` : Int $\to$ Port	1	inv
`read-char` : Port $\to$ Chr	0 or 1	aco
`peek-char` : Port $\to$ Chr	0 or 1	aco
`eof-object?` : Chr $\to$ Bool	1	inv
`write-char` : Chr $\times$ Port $\to$ Int	1 or 2	aco
`write-int` : Int $\times$ Port $\to$ Int	1 or 2	aco
`write` : String $\times$ Port $\to$ Int	1 or 2	aco
`newline` : Port $\to$ Int	0 or 1	aco
`force-output` : Port $\to$ Int	0 or 1	aco
`null-port?` : Port $\to$ Bool	1	inv
`open-input-file` : String $\to$ Port	1	aco
`close-input-port` : Port $\to$ Int	1	aco
`open-output-file` : String $\to$ Port	1	aco
`close-output-port` : Port $\to$ Int	1	aco
`current-input-port` : $\to$ Port	0	aco
`current-output-port` : $\to$ Port	0	aco
`read-image` : $\star$Int $\times$ Int $\times$ Port $\to$ Int	3	aco
`write-image` : $\star$Int $\times$ Int $\times$ Port $\to$ Int	3	aco
`create-thread` : (Int $\to$ Int) $\times$ Int $\to$ Tid	2	aco
`join-thread` : Tid $\to$ Int	1	aco
`exit-thread` : Int $\to$ Int	1	aco
`exit` : Int $\to$ Int	1	aco
`err` : Int $\times$ String $\to$ Int	2	aco

The remaining standard operators are defined below in terms of the standard primitive operators:

```
(define (not x) (if x #f #t))
(define (zero? x) (= 0 x))
(define (positive? x) (< 0 x))
(define (negative? x) (> 0 x))
(define (ashr x y) (ashl x (- y)))
(define (char<=? x y) (not (char<? y x)))
(define (char>? x y) (char<? y x))
(define (char>=? x y) (char<=? y x))
(define (addr<= x y) (not (addr< y x)))
(define (addr> x y) (addr< y x))
(define (addr>= x y) (addr<= y x))
```

3.1.3 Implementation Variables

There are a few immutable variables whose values depend on the implementation. They are in the following list which is similar to the list of primitive operators given above:

`bytes-per-word` : Int	inv
`useful-bits-per-word` : Int	inv
`max-int` : Int	inv
`min-int` : Int	inv
`root-tid` : Tid	inv

3.1.4 Formal Semantics

The formal semantics of MTPS is presented in a form that very closely resembles the semantics given in the IEEE Scheme standard [8, Appendix A]. It uses the same mathematical conventions and many of the standard's definitions without repeating them here. We strongly suggest you have a copy of the standard in hand as you read the rest of this section.

The MTPS formal semantics differs from Scheme's in a number of ways. All variables must be defined before they are referenced or assigned and no variable may be defined more than once. MTPS procedure values do not have a location associated with them because there is no comparison operator for procedures. `lambda` bound variables are immutable so a location need not be allocated for each actual parameter of an invoked procedure. Procedures always return exactly one value, so expression continuations map a single expressed value to a command continuation. MTPS `letrec` is no longer a derived expression because the immutability of `lambda` bound variables would make Scheme's definition of `letrec` useless. The answer domain is the flat domain of integers. Memory is assumed to be infinite so the storage allocator *new* always returns a location.

The semantics for MTPS specifies that the numerical operators, i.e, the primitive operators like `+` which manipulate integers, are defined only on machine integers. The set of machine integers is

$$\{\zeta \in R : \texttt{min-int} \leq \zeta \leq \texttt{max-int}\},$$

the finite subset of ordinary integers between the minimum machine integer `min-int` and the maximum integer `max-int`. (`min-int` and `max-int` are implementation specific.) In contrast, the semantics of Scheme specifies that numerical operators are defined on the entire (infinite) set of ordinary integers.

Finally and very importantly, the semantics for MTPS assumes that a program is executed by a multithreaded process. This aspect of the semantics is discussed in Chapter 2. The main idea is that I/O and interactions between the threads of a multithreaded process are modeled using the notion of an "oracle." Although the approach can be used for a variety of interthread operators, the semantics given here only handles the interthread operators that are currently in the language (see Chapter 2). As we mentioned in Chapter 2, the semantics of MTPS comes in two flavors: programs in the *thread-unaware* semantics can not invoke interthread operators and have just a single thread of control, while programs in the *thread-aware* semantics may invoke interthread operators and may have multiple threads of control.

An MTPS program is a sequence of definitions and expressions. The meaning of a program is defined via the following transformation into MTPS's abstract syntax.

$$\begin{array}{l}
(\texttt{define}\ \mathrm{I}_1\ \mathrm{E}_1)\ldots\mathrm{E}_i\ldots(\texttt{define}\ \mathrm{I}_n\ \mathrm{E}_n)\ \mathrm{E}_0 \\
\quad\Longrightarrow \\
(\texttt{define}\ \mathrm{I}_1)\ldots(\texttt{define}\ \mathrm{I}_n) \\
(\texttt{begin}\ (\texttt{set!}\ \mathrm{I}_1\ \mathrm{E}_1)\ldots\mathrm{E}_i\ldots(\texttt{set!}\ \mathrm{I}_n\ \mathrm{E}_n)\ \mathrm{E}_0)
\end{array}$$

The abstract syntax and formal semantics of MTPS is given in the remaining part of this section.

Abstract Syntax

$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{I} \in \mathrm{Ide}$	identifiers (variables and primitive operators)
$\mathrm{O} \in \mathrm{Op} \subseteq \mathrm{Ide}$	primitive operators
$\mathrm{E} \in \mathrm{Exp}$	expressions
$\mathrm{B} \in \mathrm{Bnd}$	bindings
$\mathrm{P} \in \mathrm{Pgm}$	programs

$$\begin{array}{rcl}
\mathrm{Pgm} & \longrightarrow & (\texttt{define}\ \mathrm{I})^*\ \mathrm{E} \\
\mathrm{Bnd} & \longrightarrow & (\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}))^* \\
\mathrm{Exp} & \longrightarrow & \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}\ \mathrm{E}^*) \mid (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}) \\
& & \mid (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}) \mid (\texttt{letrec}\ (\mathrm{B})\ \mathrm{E}) \\
& & \mid (\texttt{if}\ \mathrm{E}\ \mathrm{E}\ \mathrm{E}) \mid (\texttt{if}\ \mathrm{E}\ \mathrm{E}) \mid (\texttt{set!}\ \mathrm{I}\ \mathrm{E})
\end{array}$$

Op $\subseteq$ Ide since the same identifier may serve as both a variable and a primitive operator. Let $\text{Op}_0 \subseteq \text{Op}$ be the set consisting of the "thread-unaware" primitive operators in MTPS (i.e., the primitive operators in MTPS which do not access the oracle).

Semantic Domains

$\nu \in N$		natural numbers
$\alpha \in L$	$= N$	unshared locations
$\beta \in G$	$= N$	shared locations
T	$= \{\mathit{false}, \mathit{true}\}$	booleans
H		characters
$\zeta \in R$		integers
E_v	$= L^*$	unshared vectors
E_{sv}	$= G^*$	shared vectors
E_s		strings
E_p		ports
$\iota \in I$		thread IDs
M	$= \{\mathit{unspecified}, \mathit{undefined}\}$	miscellaneous
$\phi \in F$	$= E^* \to K \to C$	procedure values
$\epsilon \in E$	$= T + H + R + E_v + E_{sv} + E_s + E_p + I + M + F$	expressed values
$\sigma \in S$	$= L \to (E \times T)$	(unshared) stores
O_t	$= \{\mathsf{an}, \mathsf{ct}, \mathsf{jt}, \mathsf{et}, \mathsf{ex}, \mathsf{mv}, \mathsf{vr}, \mathsf{vs}, \mathsf{va}, \mathsf{vm}\}$	oracle tags
$\eta \in O_e$	$= O_t \times I \times I \times E \times F \times S \times G \times E_{sv}$	oracle entries
$o \in O$	$= (N \to O_e) \times N$	oracles
$\delta \in D$	$= L + E$	denoted values
$\rho \in U$	$= \text{Ide} \to D$	environments
$\theta \in C$	$= S \to O \to A$	command continuations
$\kappa \in K$	$= E \to C$	expression continuations
A	$= R$	answers
$\chi \in X$		errors

Note: O_t should contain a tag for every I/O and interthread operator. Since the semantics for the I/O operators are not considered in this report, O_t contains tags only for the interthread operators of MTPS for the sake of simplicity.

Semantic Functions

$$\begin{aligned}
\mathcal{K} &: \mathrm{Con} \to E \\
\mathcal{L} &: \mathrm{Exp} \to U \to E \\
\mathcal{I} &: \mathrm{Bnd} \to \mathrm{Ide}^* \\
\mathcal{B} &: \mathrm{Bnd} \to \mathrm{Ide}^* \to U \to E^* \to E^* \\
\mathcal{E} &: \mathrm{Exp} \to U \to K \to C \\
\mathcal{E}^* &: \mathrm{Exp}^* \to U \to (E^* \to C) \to C \\
\mathcal{D} &: \mathrm{Pgm} \to U \to K \to C \\
\mathcal{P}_0 &: \mathrm{Pgm} \to O \to A \\
\mathcal{P} &: \mathrm{Pgm} \to O \to I \to A \\
\mathcal{O} &: \mathrm{Op} \to F
\end{aligned}$$

The definition of $\mathcal{K}$ is deliberately omitted. The rest of the semantic functions are defined, in order, by the following equations.

$$\begin{aligned}
&\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{E})]\!] = \\
&\quad \lambda\rho.\,(\lambda\epsilon^*\kappa.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
&\qquad\quad \mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\,\rho\mathrm{I}^*\epsilon^*)\kappa, \\
&\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}) \\
&\qquad \text{in}\, E
\end{aligned}$$

$$\mathcal{I}[\![\ ]\!] = \langle\rangle$$

$$\mathcal{I}[\![(\mathrm{I}\ \texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{E}))\ \mathrm{B}]\!] = \langle \mathrm{I}\rangle \mathbin{\S} \mathcal{I}[\![\mathrm{B}]\!]$$

$$\mathcal{B}[\![\ ]\!] = \lambda\mathrm{I}^*\rho\epsilon^*.\,\langle\rangle$$

$$\begin{aligned}
&\mathcal{B}[\![(\mathrm{I}\ \texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{E}))\ \mathrm{B}]\!] = \\
&\quad \lambda\mathrm{I}_0^*\rho\epsilon^*.\,\langle\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{E})]\!](\mathit{extends}\,\rho\mathrm{I}_0^*\epsilon^*)\rangle \mathbin{\S} \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^*
\end{aligned}$$

$$\mathcal{E}[\![\mathrm{K}]\!] = \text{See [8, Appendix A]}$$

$$\mathcal{E}[\![\mathrm{I}]\!] = \text{See [8, Appendix A]}$$

$$\mathcal{E}[\![(\mathrm{E}\ \mathrm{E}^*)]\!] = \lambda\rho\kappa.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\rho(\lambda\epsilon^*.\,\mathcal{E}[\![\mathrm{E}]\!]\rho(\lambda\epsilon.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa))$$

Notice that, unlike in the Scheme semantics, the order of evaluation in a procedure call is specified: the arguments are evaluated left to right and then the operator is evaluated.

$$\mathcal{E}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{E})]\!] = \lambda\rho\kappa.\,\mathit{send}(\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{E})]\!]\rho)\kappa$$

$\mathcal{E}[\![(\texttt{begin}\ \mathrm{E})]\!] = \mathcal{E}[\![\mathrm{E}]\!]$

$\mathcal{E}[\![(\texttt{begin}\ \mathrm{E}\ \mathrm{E}^*\ \mathrm{E}_0)]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![\mathrm{E}]\!]\rho\lambda\epsilon.\,\mathcal{E}[\![(\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}_0)]\!]\rho\kappa$

$\mathcal{E}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{E})]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho)))\kappa$

$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}\ \mathrm{E}\ \mathrm{E})]\!] =$ See [8, Appendix A]

$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}\ \mathrm{E})]\!] =$ See [8, Appendix A]

Assignment for identifiers whose name begins and ends with an asterisk and is at least three characters long is defined using *assign*.

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

Assignment for all other identifiers is defined using *initialize*.

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

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

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

$\mathcal{D}[\![\mathrm{E}]\!] = \mathcal{E}[\![\mathrm{E}]\!]$

$\mathcal{D}[\![(\texttt{define}\ \mathrm{I})\ \mathrm{P}]\!] =$
$\quad\lambda\rho\kappa\sigma.\,\mathcal{D}[\![\mathrm{P}]\!](\rho[(\mathit{new}\ \sigma)\ \mathrm{in}\ D/\mathrm{I}])\kappa(\mathit{update}(\mathit{new}\ \sigma)(\mathit{undefined}\ \mathrm{in}\ E)\sigma)$

$\mathcal{P}_0$ gives the thread-unaware semantics for MTPS programs. ρ_0, κ_0, and σ_0 are the initial environment, expression continuation, and store for the thread-unaware semantics; they are defined below.

$\mathcal{P}_0[\![\mathrm{P}]\!] = \mathcal{D}[\![\mathrm{P}]\!]\rho_0\kappa_0\sigma_0$

$\mathcal{P}$ gives the thread-aware semantics for MTPS programs. ρ_1, κ_1, and σ_1 are the initial environment, expression continuation, and store for the thread-aware semantics; they are defined below.

$\mathcal{P}[\![\mathrm{P}]\!] = \lambda o\iota.\,\iota = \mathit{root\_tid} \rightarrow$
$\quad\mathcal{D}[\![\mathrm{P}]\!]\rho_1\kappa_1\sigma_1 o,$
$\quad\mathit{compute\_thread\_answer}\ o\iota$

The thread-unaware initial environment ρ_0 (respectively, the thread-aware initial environment ρ_1) maps each O in Op_0 (respectively, Op) to the value $((\mathcal{O}[\![\mathrm{O}]\!]\ \mathrm{in}\ E)\ \mathrm{in}\ D)$, each implementation variable to some value of the form $(\epsilon\ \mathrm{in}\ D)$, and all other identifiers to $((\mathit{undefined}\ \mathrm{in}\ E)\ \mathrm{in}\ D)$. (Thus, it maps no identifier to a location.) An identifier I will be treated as a primitive operator O in the context of an environment ρ if the name of I is the name of a primitive operator and $\rho\mathrm{I} = ((\mathcal{O}[\![\mathrm{O}]\!]\ \mathrm{in}\ E)\ \mathrm{in}\ D)$; otherwise, I will be treated as a variable.

The thread-unaware initial expression continuation κ_0 is defined by

```
$\kappa_0 = \lambda\epsilon\sigma o.\, \epsilon \in R \rightarrow$
  $\epsilon \mid R,$
  $\mathit{wrong}$ "result is not an integer"$\sigma o$
```

and the thread-aware initial expression continuation κ_1 is defined by

```
$\kappa_1 = \lambda\epsilon\sigma o.\, \epsilon \in R \rightarrow$
  let
    $o' = (\mathit{thread\_entry}\ o(\mathit{thread\_tid}\ \sigma))$
    $\eta = \mathit{oracle\_entry}\ o'$
  in
    $\mathit{entry\_tag}\ \eta = \mathsf{an} \rightarrow$
      $\mathit{entry\_val}\ \eta = \epsilon \rightarrow$
        $\epsilon \mid R,$
        $\mathit{wrong}$ "wrong value in entry"$\sigma o',$
      $\mathit{entry\_tag\_mismatch}\ \eta\sigma o',$
  $\mathit{wrong}$ "result is not an integer"$\sigma o.$
```

The thread-unaware initial store σ_0 is defined by

$$\sigma_0 = \lambda\alpha.\, \langle \bot_E, \mathit{false} \rangle$$

and the thread-aware initial store σ_1 is defined by

```
$\sigma_1 = \lambda\alpha.\, \alpha = \mathit{tid\_loc} \rightarrow$
  $\langle \mathit{root\_tid}\ \mathrm{in}\ E, \mathit{true} \rangle,$
  $\langle \bot_E, \mathit{false} \rangle$
```

Semantics of Primitive Operators

The definition of $\mathcal{O}$ is given only for a few representative primitive operators: `+`, `create-thread`, `join-thread`, `exit-thread`, `exit`, and all five of the shared vector operators.

```
$\mathcal{O}[\![\texttt{+}]\!] =$
  $twoarg(\lambda \epsilon_1 \epsilon_2 \kappa.$
    $\epsilon_1 \in R \;\; and \;\; \epsilon_2 \in R \rightarrow$
      let
        $\zeta_1 = \epsilon_1 \mid R$
        $\zeta_2 = \epsilon_2 \mid R$
      in
        $min\_int \leq \zeta_1 \;\; and \;\; \zeta_2 \leq max\_int \rightarrow$
          $min\_int \leq \zeta_1 + \zeta_2 \leq max\_int \rightarrow$
            $send((\zeta_1 + \zeta_2) \text{ in } E)\kappa,$
            $wrong$ “overflow of machine integer addition”,
          $wrong$ “arguments are not machine integers”,
      $wrong$ “arguments are not integers”)
```

```
$\mathcal{O}[\![$create-thread$]\!] =$
  $\mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa.$
    $\epsilon_1 \in F \rightarrow$
      $\epsilon_2 \in R \rightarrow$
        let
          $o' = (\mathit{thread\_entry}\, o(\mathit{thread\_tid}\, \sigma))$
          $\eta = \mathit{oracle\_entry}\, o'$
          $\sigma' = \mathit{entry\_store}\ \eta$
        in
          $\mathit{entry\_tag}\ \eta = \mathsf{ct} \rightarrow$
            $\mathit{entry\_proc}\ \eta = (\epsilon_1 \mid F) \rightarrow$
              $\mathit{entry\_val}\ \eta = \epsilon_2 \rightarrow$
                $\forall\alpha, \alpha \neq \mathit{tid\_loc}\ \ \mathit{implies}\ \ \sigma'\alpha = \sigma\alpha \rightarrow$
                  $\mathit{advance}((\mathit{entry\_child\_tid}\ \eta)\text{ in } E)\kappa\sigma o'$,
                  $\mathit{wrong}$ “wrong store in entry”$\sigma o'$,
                $\mathit{wrong}$ “wrong value in entry”$\sigma o'$,
              $\mathit{wrong}$ “wrong procedure in entry”$\sigma o'$,
            $\mathit{entry\_tag\_mismatch}\ \eta\sigma o'$,
        $\mathit{wrong}$ “second argument is not an integer”,
      $\mathit{wrong}$ “first argument is not a procedure”)
```

```
$\mathcal{O}[\![$join-thread$]\!] =$
  $\mathit{onearg}(\lambda\epsilon\kappa.$
    $\epsilon \in I \rightarrow$
      let
        $o' = (\mathit{thread\_entry}\, o(\mathit{thread\_tid}\, \sigma))$
        $\eta = \mathit{oracle\_entry}\, o'$
      in
        $\mathit{entry\_tag}\ \eta = \mathsf{jt} \rightarrow$
          $\mathit{entry\_aux\_tid}\ \eta = (\epsilon \mid I) \rightarrow$
            $\mathit{advance}(\mathit{entry\_val}\ \eta)\kappa\sigma o'$,
            $\mathit{wrong}$ “wrong auxiliary thread ID in entry”$\sigma o'$,
          $\mathit{entry\_tag\_mismatch}\ \eta\sigma o'$,
      $\mathit{wrong}$ “argument is not a thread ID”)
```

```
$\mathcal{O}[\![$exit-thread$]\!] =$
  $oneargs(\lambda\epsilon\kappa.$
    $\epsilon \in R \rightarrow$
      let
        $o' = (thread\_entry\ o(thread\_tid\ \sigma))$
        $\eta = oracle\_entry\ o'$
      in
        $entry\_tag\ \eta = \mathsf{et} \rightarrow$
          $entry\_val\ \eta = \epsilon \rightarrow$
            $halt\ \epsilon\sigma o'$,
            $wrong$ “wrong value in entry”$\sigma o'$,
          $entry\_tag\_mismatch\ \eta\sigma o'$,
      $wrong$ “argument is not an integer”)
```

```
$\mathcal{O}[\![$exit$]\!] =$
  $oneargs(\lambda\epsilon\kappa.$
    $\epsilon \in R \rightarrow$
      let
        $o' = (thread\_entry\ o(thread\_tid\ \sigma))$
        $\eta = oracle\_entry\ o'$
      in
        $entry\_tag\ \eta = \mathsf{ex} \rightarrow$
          $entry\_val\ \eta = \epsilon \rightarrow$
            $halt\ \epsilon\sigma o'$,
            $wrong$ “wrong value in entry”$\sigma o'$,
          $entry\_tag\_mismatch\ \eta\sigma o'$,
      $wrong$ “argument is not an integer”)
```

```
$\mathcal{O}[\![\texttt{make-shared-vector}]\!] =$
  $\mathit{onearg}(\lambda\epsilon\kappa.$
    $0 \leq (\epsilon \mid R) \rightarrow$
      let
        $o' = (\mathit{thread\_entry}\, o(\mathit{thread\_tid}\, \sigma))$
        $\eta = \mathit{oracle\_entry}\, o'$
      in
        $\mathit{entry\_tag}\ \eta = \mathsf{mv} \rightarrow$
          $\#(\mathit{entry\_vec}\ \eta) = (\epsilon \mid R) \rightarrow$
            $\mathit{advance}((\mathit{entry\_vec}\ \eta)\ \text{in}\ E)\kappa\sigma o'$,
            $\mathit{wrong}$ “wrong vector length in entry”$\sigma o'$,
          $\mathit{entry\_tag\_mismatch}\ \eta\sigma o'$,
      $\mathit{wrong}$ “argument is not a nonnegative integer”)

$\mathcal{O}[\![\texttt{shared-vector-ref}]\!] =$
  $\mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa.$
    $\epsilon_1 \in E_{sv} \rightarrow$
      $0 \leq (\epsilon_2 \mid R) < \#(\epsilon_1 \mid E_{sv}) \rightarrow$
        let
          $o' = (\mathit{thread\_entry}\, o(\mathit{thread\_tid}\, \sigma))$
          $\eta = \mathit{oracle\_entry}\, o'$
        in
          $\mathit{entry\_tag}\ \eta = \mathsf{vr} \rightarrow$
            $\mathit{entry\_vec}\ \eta = (\epsilon_1 \mid E_{sv}) \rightarrow$
              $\mathit{entry\_loc}\ \eta = (\epsilon_1 \mid E_{sv}) \downarrow ((\epsilon_2 \mid R) + 1) \rightarrow$
                $\mathit{entry\_val}\ \eta \neq (\mathit{undefined}\, \text{in}\, E) \rightarrow$
                  $\mathit{advance}(\mathit{entry\_val}\ \eta)\kappa\sigma o'$,
                  $\mathit{wrong}$ “value has not been set”$\sigma o'$,
                $\mathit{wrong}$ “wrong location in entry”$\sigma o'$,
              $\mathit{wrong}$ “wrong vector in entry”$\sigma o'$,
            $\mathit{entry\_tag\_mismatch}\ \eta\sigma o'$,
        $\mathit{wrong}$ “second argument is out of range”,
      $\mathit{wrong}$ “first argument is not a shared vector”)
```

```
$\mathcal{O}[\![$shared-vector-set!$]\!] =$
  $\mathit{threearg}(\lambda \epsilon_1 \epsilon_2 \epsilon_3 \kappa.$
    $\epsilon_1 \in E_{sv} \rightarrow$
      $0 \leq (\epsilon_2 \mid R) < \#(\epsilon_1 \mid E_{sv}) \rightarrow$
        let
          $o' = (\mathit{thread\_entry}\, o(\mathit{thread\_tid}\, \sigma))$
          $\eta = \mathit{oracle\_entry}\, o'$
        in
          $\mathit{entry\_tag}\ \eta = \mathsf{vs} \rightarrow$
            $\mathit{entry\_vec}\ \eta = (\epsilon_1 \mid E_{sv}) \rightarrow$
              $\mathit{entry\_loc}\ \eta = (\epsilon_1 \mid E_{sv}) \downarrow ((\epsilon_2 \mid R) + 1) \rightarrow$
                $\mathit{entry\_val}\ \eta = \epsilon_3 \rightarrow$
                  $\mathit{advance}(\mathit{unspecified}\,\mathrm{in}\,E)\kappa\sigma o'$,
                  $\mathit{wrong}$ “wrong value in entry”$\sigma o'$,
                $\mathit{wrong}$ “wrong location in entry”$\sigma o'$,
              $\mathit{wrong}$ “wrong vector in entry”$\sigma o'$,
            $\mathit{entry\_tag\_mismatch}\ \eta\sigma o'$,
        $\mathit{wrong}$ “second argument is out of range”,
      $\mathit{wrong}$ “first argument is not a shared vector”)
```

```
$\mathcal{O}[\![$shared-vector-access$]\!] =$
  $twoarg(\lambda\epsilon_1\epsilon_2\kappa.$
    $\epsilon_1 \in E_{sv} \rightarrow$
      $\exists\zeta, (\epsilon_1 \mid E_{sv}) \downarrow \zeta = shared\_loc(\epsilon_2 \mid R) \rightarrow$
        let
          $o' = (thread\_entry\, o(thread\_tid\, \sigma))$
          $\eta = oracle\_entry\, o'$
        in
          $entry\_tag\ \eta = \mathsf{va} \rightarrow$
            $entry\_vec\ \eta = (\epsilon_1 \mid E_{sv}) \rightarrow$
              $entry\_loc\ \eta = shared\_loc(\epsilon_2 \mid R) \rightarrow$
                $entry\_val\ \eta \neq (undefined\,\mathrm{in}\,E) \rightarrow$
                  $advance(entry\_val\ \eta)\kappa\sigma o'$,
                  $wrong$ “value has not been set”$\sigma o'$,
                $wrong$ “wrong location in entry”$\sigma o'$,
              $wrong$ “wrong vector in entry”$\sigma o'$,
            $entry\_tag\_mismatch\ \eta\sigma o'$,
        $wrong$ “second argument is out of range”,
      $wrong$ “first argument is not a shared vector”)
```

```
$\mathcal{O}[\![$shared-vector-modify!$]\!] =$
  $\mathit{threearg}(\lambda\epsilon_1\epsilon_2\epsilon_3\kappa.$
    $\epsilon_1 \in E_{sv} \rightarrow$
      $\exists\zeta, (\epsilon_1 \mid E_{sv}) \downarrow \zeta = \mathit{shared\_loc}(\epsilon_2 \mid R) \rightarrow$
        let
          $o' = (\mathit{thread\_entry}\, o(\mathit{thread\_tid}\, \sigma))$
          $\eta = \mathit{oracle\_entry}\, o'$
        in
          $\mathit{entry\_tag}\ \eta = \mathsf{vm} \rightarrow$
            $\mathit{entry\_vec}\ \eta = (\epsilon_1 \mid E_{sv}) \rightarrow$
              $\mathit{entry\_loc}\ \eta = \mathit{shared\_loc}(\epsilon_2 \mid R) \rightarrow$
                $\mathit{entry\_val}\ \eta = \epsilon_3 \rightarrow$
                  $\mathit{advance}(\mathit{unspecified}\,\mathrm{in}\,E)\kappa\sigma o'$,
                  $\mathit{wrong}$ “wrong value in entry”$\sigma o'$,
                $\mathit{wrong}$ “wrong location in entry”$\sigma o'$,
              $\mathit{wrong}$ “wrong vector in entry”$\sigma o'$,
            $\mathit{entry\_tag\_mismatch}\ \eta\sigma o'$,
        $\mathit{wrong}$ “second argument is out of range”,
      $\mathit{wrong}$ “first argument is not a shared vector”)
```

Auxiliary Functions and Constants

$\mathit{lookup} : U \rightarrow \mathrm{Ide} \rightarrow D$
$\mathit{lookup} = \lambda\rho\mathrm{I}.\, \rho\mathrm{I}$

$\mathit{extends} : U \rightarrow \mathrm{Ide}^* \rightarrow \mathrm{E}^* \rightarrow U$
$\mathit{extends} = \lambda\rho\mathrm{I}^*\epsilon^*.\, \#\mathrm{I}^* = 0 \rightarrow$
  ρ,
  $\mathit{extends}(\rho[\epsilon^* \downarrow 1\,\mathrm{in}\,D/\mathrm{I}^* \downarrow 1])(\mathrm{I}^* \dagger 1)(\epsilon^* \dagger 1)$

$\mathit{wrong} : X \rightarrow C$
$\mathit{wrong} = \lambda\chi\sigma o.\, \bot_A$

$\mathit{send} : E \rightarrow K \rightarrow C$
$\mathit{send} = \lambda\epsilon\kappa.\, \kappa\epsilon$

$advance : E \to K \to S \to O \to A$
$advance = \lambda \epsilon \kappa \sigma o.\, send\, \epsilon \kappa \sigma (step\_oracle\, o)$

$halt : E \to S \to O \to A$
$halt = \lambda \epsilon \sigma o.\, advance\, \epsilon \kappa_1 \sigma o$

$single : K \to K$
$single = \lambda \kappa.\, \kappa$

$new : S \to L$ [implementation-dependent]
new satisfies $\forall \sigma, (\exists \alpha, \sigma(\alpha) \downarrow 2 = false)$ *implies* $\sigma(new\, \sigma) \downarrow 2 = false$

$hold : D \to K \to C$
$hold = \lambda \delta \kappa \sigma.\, \delta \in E \to$
$\quad send(\delta \mid E) \kappa \sigma,$
$\quad send(\sigma(\delta \mid L) \downarrow 1) \kappa \sigma$

$assign : D \to E \to C \to C$
$assign = \lambda \delta \epsilon \theta \sigma.\, \delta \in L \to$
$\quad \theta(update(\delta \mid L) \epsilon \sigma),$
$\quad wrong$ "assignment of an immutable variable"σ

$initialize : D \to E \to C \to C$
$initialize = \lambda \delta \epsilon \theta \sigma.\, \delta \in L \wedge \sigma(\delta \mid L) \downarrow 1 = (undefined\, \text{in}\, E) \to$
$\quad \theta(update(\delta \mid L) \epsilon \sigma),$
$\quad wrong$ "assignment of an immutable variable"σ

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

$unshared\_loc : R \to L$ [implementation-dependent]
$unshared\_loc$ satisfies $\zeta < 0$ *or* $\zeta = \bot_R$ *implies* $unshared\_loc\, \zeta = \bot_L$

$\mathit{shared\_loc} : R \to G$ [implementation-dependent]
$\mathit{shared\_loc}$ satisfies $\zeta < 0$ *or* $\zeta = \bot_R$ *implies* $\mathit{shared\_loc}\,\zeta = \bot_G$

The constants $\mathit{max\_int}$ and $\mathit{max\_int}$ are the minimum and maximum machine integers, respectively. The initial environments contain bindings for these values.

$\mathit{max\_int} : R$
$\rho_0[\![\texttt{max-int}]\!] = \rho_1[\![\texttt{max-int}]\!] = (\mathit{max\_int} \,\text{in}\, E)$ in D

$\mathit{min\_int} : R$
$\rho_0[\![\texttt{min-int}]\!] = \rho_1[\![\texttt{max-int}]\!] = (\mathit{min\_int} \,\text{in}\, E)$ in D

Given an oracle o, *oracle_index* returns the oracle index of o, *oracle_entry* returns the oracle entry indicated by the index of o, and *step_oracle* moves the index of o to the next entry, respectively.

$\mathit{oracle\_index} : O \to N$
$\mathit{oracle\_index} = \lambda o.\,(o \downarrow 2)$

$\mathit{oracle\_entry} : O \to O_e$
$\mathit{oracle\_index} = \lambda o.\,(o \downarrow 1)(o \downarrow 2)$

$\mathit{step\_oracle} : O \to I$
$\mathit{step\_oracle} = \lambda o.\,((o \downarrow 1), ((o \downarrow 2) + 1))$

The following nine auxiliary functions return certain information contained in an oracle entry.

$\mathit{entry\_tag} : O_e \to O_t$
$\mathit{entry\_tag} = \lambda \eta.\,(\eta \downarrow 1)$

$\mathit{entry\_tid} : O_e \to I$
$\mathit{entry\_tid} = \lambda \eta.\,(\eta \downarrow 2)$

$\mathit{entry\_aux\_tid} : O_e \to I$
$\mathit{entry\_aux\_tid} = \lambda \eta.\,(\eta \downarrow 3)$

$$\begin{aligned}&\mathit{entry\_val} : O_e \rightarrow E\\&\mathit{entry\_val} = \lambda\eta.\,(\eta \downarrow 4)\end{aligned}$$

$$\begin{aligned}&\mathit{entry\_proc} : O_e \rightarrow F\\&\mathit{entry\_proc} = \lambda\eta.\,(\eta \downarrow 5)\end{aligned}$$

$$\begin{aligned}&\mathit{entry\_store} : O_e \rightarrow S\\&\mathit{entry\_store} = \lambda\eta.\,(\eta \downarrow 6)\end{aligned}$$

$$\begin{aligned}&\mathit{entry\_loc} : O_e \rightarrow G\\&\mathit{entry\_loc} = \lambda\eta.\,(\eta \downarrow 7)\end{aligned}$$

$$\begin{aligned}&\mathit{entry\_vec} : O_e \rightarrow E_{sv}\\&\mathit{entry\_vec} = \lambda\eta.\,(\eta \downarrow 8)\end{aligned}$$

$$\begin{aligned}&\mathit{entry\_child\_tid} : O_e \rightarrow I\\&\mathit{entry\_child\_tid} = \lambda\eta.\,\mathit{thread\_tid}(\mathit{entry\_store}\ \eta)\end{aligned}$$

Given an oracle entry, a store, and an oracle *entry_tag_mismatch* halts the thread if the oracle entry is an `exit` entry; otherwise an error message is returned.

$$\begin{aligned}&\mathit{entry\_tag\_mismatch} : O_e \rightarrow S \rightarrow O \rightarrow A\\&\mathit{entry\_tag\_mismatch} = \lambda\eta\sigma o.\ \mathit{entry\_tag}\ \eta = \mathsf{ex} \rightarrow\\&\quad \mathit{halt}(\mathit{entry\_val}\,\eta)\sigma o,\\&\quad \mathit{wrong}\ \text{“wrong tag in entry”}\sigma o\end{aligned}$$

Given an oracle o and an thread ID ι, *thread_entry* moves the index of o to the first entry at or after the current entry of o whose thread ID is ι or whose tag is ex.

```
$thread\_entry : O \rightarrow I \rightarrow O$
$thread\_entry = \lambda o\iota.$
  let
    $\eta = oracle\_entry\, o$
  in
    $(entry\_tid\ \eta = \iota)\ \ or\ \ (entry\_tag\ \eta = \mathsf{ex}) \rightarrow$
      $o,$
      $thread\_entry\ (step\_oracle\ o)\iota$
```

Given an oracle o and an thread ID ι, *next_thread_entry* moves the index of o to the first entry after the current entry of o whose thread ID is ι.

```
$next\_thread\_entry : O \to I \to O$
$next\_thread\_entry = \lambda o \iota.$
  let
    $o' = step\_oracle\; o$
    $\eta = oracle\_entry\; o'$
  in
    $entry\_tid\;\; \eta = \iota \to$
      $o'$,
      $next\_thread\_entry\; o' \iota$
```

Given an oracle o and an thread ID ι, *thread_call* moves the index of o to the first entry at or after the current entry of o which corresponds to the creation of the thread with ID ι.

```
$thread\_call : O \to I \to O$
$thread\_call = \lambda o \iota.$
  let
    $\eta = oracle\_entry\; o$
  in
    $(entry\_tag\;\; \eta = \mathsf{ct})\;\; and\;\; (entry\_child\_tid\;\; \eta = \iota) \to$
      $o$,
      $thread\_call\; (step\_oracle\; o) \iota$
```

Given an oracle o and an thread ID ι, *thread_answer* moves the index of o to the first entry at or after the current entry of o which corresponds to the answer for the thread with ID ι.

```
$thread\_answer : O \to I \to O$
$thread\_answer = \lambda o \iota.$
  let
    $\eta = oracle\_entry\; o$
  in
    $(entry\_tag\;\; \eta = \mathsf{an})\;\; and\;\; (entry\_tid\;\; \eta = \iota) \to$
      $o$
      $thread\_answer\; (step\_oracle\; o) \iota$
```

Given an oracle o and an thread ID ι, *compute_thread_answer* returns the answer produced by the thread with ID ι.

$$
\begin{array}{l}
\mathit{compute\_thread\_answer} : O \to I \to A \\
\mathit{compute\_thread\_answer} = \lambda o \iota. \\
\quad \text{let} \\
\qquad o' = \mathit{thread\_call}\, o \iota \\
\qquad \eta = \mathit{oracle\_entry}\, o' \\
\quad \text{in} \\
\qquad (\mathit{entry\_proc}\, \eta)\langle(\mathit{entry\_val}\, \eta)\rangle \kappa_1 (\mathit{entry\_store}\, \eta) o'
\end{array}
$$

The constant *root_tid* is the thread ID of the root thread. The initial environments contain a binding for this value.

$$
\begin{array}{l}
\mathit{root\_tid} : I \\
\rho_0[\![\texttt{root-tid}]\!] = \rho_1[\![\texttt{root-tid}]\!] = (\mathit{root\_tid}\, \text{in}\, E)\, \text{in}\, D
\end{array}
$$

Given a store σ, *thread_tid* returns the thread ID stored in σ, which is intended to be the ID of the thread whose store is σ.

$$
\begin{array}{l}
\mathit{thread\_tid} : S \to I \\
\mathit{thread\_tid} = \lambda \sigma. ((\sigma\, \mathit{tid\_loc} \mid E) \mid I)
\end{array}
$$

$$
\begin{array}{l}
\mathit{tid\_loc} : L \\
\mathit{tid\_loc}\ \text{ is unspecified}
\end{array}
$$

Axioms

The axioms of the semantics are the assumptions that are implicitly given by the equations for the semantic domains plus the following axiom that limits the role of expression continuations.

Definition 3.1 *A store σ' is the* successor *of a store σ relative to an environment ρ, written $\sigma \preceq_\rho \sigma'$, if $\sigma((\rho \mathrm{I})\text{ in } L) \downarrow 1 \neq (\mathit{undefined}\,\text{in}\,E)$ implies $\sigma((\rho\mathrm{I})\text{ in } L) = \sigma'((\rho\mathrm{I})\text{ in } L)$ for all immutable variables* I.

Clearly, $\preceq_\rho$ is reflexive and transitive.

Axiom 3.2 *For all* $\mathrm{E}, \rho, \sigma, o$*, one of the following three statements is true:*

1. $\forall \kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma o = \bot_A$.

2. $\exists \epsilon\sigma' o', \forall \kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma o = \kappa_1 \epsilon\sigma' o'$.

3. $\exists \epsilon\sigma' o', \forall \kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma o = \kappa \epsilon\sigma' o'$.

Moreover, in the second and third statements, $\sigma \preceq_\rho \sigma'$, $o \downarrow 1 = o' \downarrow 1$ *and* $o \downarrow 2 \leq o' \downarrow 2$.

The first statement is true when the evaluation of E results in an error; the second statement is true when the evaluation of E is halted prematurely; and the third statement is true when the evaluation of E neither results in an error nor is halted prematurely. We will say that the evaluation of E *errs* relative to ρ, σ, o if the first statement is true and that the evaluation of E *halts* relative to ρ, σ, o if the second statement is true.

Axiom 3.2 makes explicit one of the defining characteristics of MTPS. It says that the evaluation of an expression generates an error or invokes either the (thread-aware) initial expression continuation or the current expression continuation on the resulting value. The axiom does not hold in Scheme due to the presence of `call-with-current-continuation`.

The theorem below, which is a consequence of Axiom 3.2, places an important restriction on what are allowed to be the primitive operators of MTPS. It rules out any primitive operator that invokes an expression continuation other than the (thread-aware) initial expression continuation or the current expression continuation.

Theorem 3.3 *For all* $\mathrm{O}, \rho, \epsilon^*, \sigma, o$, *one of the following three statements is true:*

1. $\forall \kappa, \mathcal{O}[\![\mathrm{O}]\!]\epsilon^*\kappa\sigma o = \bot_A$.

2. $\exists \epsilon\sigma' o', \forall \kappa, \mathcal{O}[\![\mathrm{O}]\!]\epsilon^*\kappa\sigma o = \kappa_1 \epsilon\sigma' o'$.

3. $\exists \epsilon\sigma' o', \forall \kappa, \mathcal{O}[\![\mathrm{O}]\!]\epsilon^*\kappa\sigma o = \kappa \epsilon\sigma' o'$.

Moreover, in the second and third statements, $\sigma \preceq_\rho \sigma'$, $o \downarrow 1 = o' \downarrow 1$ *and* $o \downarrow 2 \leq o' \downarrow 2$.

The second statement of the theorem is true only if O is `exit` or `exit-thread`.

Let a *premodel* for the MTPS semantics be any set of domains that satisfies all the axioms of the semantics except for possibly Axiom 3.2, and let a *model*

```
(case ⟨key⟩
   ((0) ⟨sequence₁⟩)
     ⋮
   (((n − 1)) ⟨sequenceₙ⟩)
   (else ⟨sequence₀⟩))
```

Figure 3.1: Case Syntax

for the MTPS semantics be any set of domains that satisfies all the axioms of the semantics including Axiom 3.2. By the theory of denotational semantics, there is a premodel for the MTPS semantics. We conjecture that, from any premodel $\mathcal{M}$ for the MTPS semantics, one can extract a model $\mathcal{M}'$ for the MTPS semantics such that, for all MTPS programs P, the value $\mathcal{P}[\![\mathrm{P}]\!]$ is the same in both $\mathcal{M}$ and $\mathcal{M}'$.

3.1.5 Compiler Restrictions

Our compiler places the following additional restrictions on the syntax of MTPS programs.

- All references to standard operators must be in the operator position of an application.
- The `case` derived expression is restricted so as to become essentially a computed goto. There must be exactly one integer given as the selection criterion for each clause. The first selection criteria must be zero and the selection criteria for other clauses must be the successor of the previous clause's selection criterion as shown in Figure 3.1.

  The `case` expression is normally expanded into a nested sequence of conditionals, and the semantics of `case` is derived from this expansion.

3.2 Macro-Free PreScheme

Macro-Free PreScheme (MFPS) programs result from MTPS programs by expanding all derived syntax except the `case` expression, changing each bound variable if its name is the name of a primitive operator, and replacing each single armed conditional (`if` E E) with (`if` E E (`if` #`f` #`f`)). These programs resemble an MTPS program after it has been translated into its abstract syntax. MFPS programs also satisfy the restriction that each multiary primitive operator must be used at one fixed arity. For example, calls to `+` must have exactly two operands so MTPS programs with calls to `+` using more than two operands must be translated into a combination of calls using only a binary version of `+`.

3.2.1 Abstract Syntax

$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{I} \in \mathrm{Ide}$	identifiers (variables and primitive operators)
$\mathrm{O} \in \mathrm{Op} \subseteq \mathrm{Ide}$	primitive operators
$\mathrm{E} \in \mathrm{Exp}$	expressions
$\mathrm{B} \in \mathrm{Bnd}$	bindings
$\mathrm{P} \in \mathrm{Pgm}$	programs

$$\begin{aligned}
\mathrm{Pgm} &\longrightarrow (\texttt{define}\ \mathrm{I})^*\ \mathrm{E} \\
\mathrm{Bnd} &\longrightarrow (\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}))^* \\
\mathrm{Exp} &\longrightarrow \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}\ \mathrm{E}^*) \mid (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}) \\
&\quad\ \mid (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}) \mid (\texttt{letrec}\ (\mathrm{B})\ \mathrm{E}) \\
&\quad\ \mid (\texttt{if}\ \mathrm{E}\ \mathrm{E}\ \mathrm{E}) \mid (\texttt{if}\ \#\texttt{f}\ \#\texttt{f}) \mid (\texttt{set!}\ \mathrm{I}\ \mathrm{E}) \\
&\quad\ \mid \langle \mathrm{O}\ \mathrm{E}^* \rangle \mid (\texttt{case}\ \mathrm{E}\ ((\mathrm{K})\ \mathrm{E})^*\ (\texttt{else}\ \mathrm{E}))
\end{aligned}$$

An expression (E E*) represents an ordinary procedure call, while an expression ⟨O E*⟩ represents an inlined primitive operator invocation. An MFPS program does not contain any procedure calls (E E*) where $\mathrm{E} \in \mathrm{Op}$. The user is not given access to the ⟨O E*⟩ syntax. This is because the compiler changes every procedure call of the form (O E*) to an inlined primitive operator invocation of the form ⟨O E*⟩.

3.2.2 Semantics

The semantics of MFPS is given by the same equations as MTPS's plus the following new equation for expressions of the form ⟨O E*⟩:

$$\mathcal{E}[\![\langle \mathrm{O}\ \mathrm{E}^* \rangle]\!] = \lambda\rho\kappa.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\rho(\lambda\epsilon^*.\,\mathcal{O}[\![\mathrm{O}]\!]\epsilon^*\kappa)$$

An MFPS program's abstract syntax is derived by expanding `case` expressions as described in the Scheme standard.

$$\rho \vdash \mathrm{K} : \mathit{type\_of}(\mathrm{K})$$

$$\rho \vdash \mathrm{O} : \mathit{type\_of}(\mathrm{O})$$

$$\rho, \mathrm{I} : \tau \vdash \mathrm{I} : \tau$$

Figure 3.2: Typing Axioms

3.2.3 Static Semantics

Some, but not all MFPS programs are strongly typed. A type checker makes this distinction. As in Standard ML [14], types are inferred, not declared, but unlike Standard ML, there are no polymorphic variables. All strongly typed expressions have a unique grounded type except (`if #f #f`) and expressions of the form (`set!` I E).

- The base types are Int, Chr, Bool, String, Port, and Tid.
- If τ is a type, then so are $\star\,\tau$, $\bullet\,\tau$, and $\bullet_s\,\tau$.
- If $\tau_1, \ldots, \tau_n$, and τ are types, then so is $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$.

The types Int, Chr, Bool, String, Port, and Tid are the types of an integer, a character, a boolean, a string, a port, and a thread ID, respectively. The type $\star\,\tau$ is the type of a pointer to an object of type τ; $\bullet\,\tau$ is the type of an unshared vector of objects of type τ; and $\bullet_s\,\tau$ is the type of a shared vector of objects of type τ.[2] Finally, $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ is the type of a procedure.

The axioms and rules used to assign types to MFPS abstract syntax expressions are given in Figure 3.2 and Figure 3.3, respectively. When a type is unconstrained by the rules, the expression is assigned the integer type.

[2]In C pointers and vectors have the same type, but in MTPS pointers and vectors have different types to guarantee that vectors can only be accessed within bounds.

$$\frac{\rho \vdash \mathrm{I} : \tau}{\rho, \mathrm{I}_0 : \tau_0 \vdash \mathrm{I} : \tau} \quad (\mathrm{I}_0 \neq \mathrm{I})$$

$$\frac{\rho \vdash \mathrm{E}_0 : \tau_1 \times \cdots \times \tau_n \to \tau \quad \rho \vdash \mathrm{E}_1 : \tau_1 \quad \ldots \quad \rho \vdash \mathrm{E}_n : \tau_n}{\rho \vdash (\mathrm{E}_0\ \mathrm{E}_1 \ldots \mathrm{E}_n) : \tau}$$

$$\frac{\rho \vdash \mathrm{O} : \tau_1 \times \cdots \times \tau_n \to \tau \quad \rho \vdash \mathrm{E}_1 : \tau_1 \quad \ldots \quad \rho \vdash \mathrm{E}_n : \tau_n}{\rho \vdash \langle \mathrm{O}\ \mathrm{E}_1 \ldots \mathrm{E}_n \rangle : \tau}$$

$$\frac{\rho, \mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n \vdash \mathrm{E} : \tau}{\rho \vdash (\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_n)\ \mathrm{E}) : \tau_1 \times \cdots \times \tau_n \to \tau}$$

$$\frac{\rho \vdash \mathrm{E} : \tau}{\rho \vdash (\texttt{begin}\ \mathrm{E}) : \tau}$$

$$\frac{\rho \vdash \mathrm{E}_0 : \tau_0 \quad \rho \vdash (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}) : \tau}{\rho \vdash (\texttt{begin}\ \mathrm{E}_0\ \mathrm{E}^*\ \mathrm{E}) : \tau}$$

$$\frac{\begin{array}{c} \rho, \mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n \vdash \mathrm{E}_1 : \tau_1 \\ \vdots \\ \rho, \mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n \vdash \mathrm{E}_n : \tau_n \\ \rho, \mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n \vdash \mathrm{E} : \tau \end{array}}{\rho \vdash (\texttt{letrec}\ ((\mathrm{I}_1\ \mathrm{E}_1) \ldots (\mathrm{I}_n\ \mathrm{E}_n))\ \mathrm{E}) : \tau}$$

$$\frac{\rho \vdash \mathrm{E}_0 : \mathrm{Bool} \quad \rho \vdash \mathrm{E}_1 : \tau \quad \rho \vdash \mathrm{E}_2 : \tau}{\rho \vdash (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2) : \tau}$$

$$\rho \vdash (\texttt{if \#f \#f}) : \tau$$

$$\frac{\rho \vdash \mathrm{I} : \tau_0 \quad \rho \vdash \mathrm{E} : \tau_0}{\rho \vdash (\texttt{set!}\ \mathrm{I}\ \mathrm{E}) : \tau}$$

$$\frac{\mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n \vdash \mathrm{E} : \mathrm{Int}}{\vdash (\texttt{define}\ \mathrm{I}_1) \ldots (\texttt{define}\ \mathrm{I}_n)\ \mathrm{E} : \mathrm{Int}}$$

Figure 3.3: Typing Rules

3.3 Simple PreScheme

Simple PreScheme programs are syntactically restricted, strongly typed MFPS programs. The abstract syntax is as follows:

$K \in \mathrm{Con}$	constants
$I \in \mathrm{Ide}$	identifiers (variables and primitive operators)
$O \in \mathrm{Op} \subseteq \mathrm{Ide}$	primitive operators
$C \in \mathrm{Cls}$	case clauses
$S \in \mathrm{Smpl}$	simple expressions
$B \in \mathrm{Bnd}$	bindings
$E \in \mathrm{Exp}$	top level expressions
$P \in \mathrm{Pgm}$	programs

$$\begin{array}{rl}
\mathrm{Pgm} \longrightarrow & (\texttt{define}\ \mathrm{I})^*\ \mathrm{E} \\
\mathrm{Exp} \longrightarrow & (\texttt{letrec}\ (\mathrm{B})\ \mathrm{S}) \\
\mathrm{Bnd} \longrightarrow & (\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S}))^* \\
\mathrm{Smpl} \longrightarrow & \mathrm{K} \mid \mathrm{I} \mid (\mathrm{S}\ \mathrm{S}^*) \mid ((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*) \\
& \mid (\texttt{begin}\ \mathrm{S}^*\ \mathrm{S}) \mid (\texttt{if}\ \mathrm{S}\ \mathrm{S}\ \mathrm{S}) \mid (\texttt{if}\ \texttt{\#f}\ \texttt{\#f}) \\
& \mid (\texttt{case}\ \mathrm{S}\ \mathrm{C}) \mid \langle \mathrm{O}\ \mathrm{S}^* \rangle \mid (\texttt{set!}\ \mathrm{I}\ \mathrm{S}) \\
\mathrm{Cls} \longrightarrow & ((\mathrm{K})\ \mathrm{S})^*\ (\texttt{else}\ \mathrm{S})
\end{array}$$

There is a further syntactic restriction. The first selection criteria of a `case` clause must be zero and the selection criteria for other clauses must be the successor of the previous clause's selection criterion. The `case` derived expression is restricted so as to allow it to be compiled into a computed goto. See Figure 3.1 on page 38.

The syntactic restrictions used to define Simple PreScheme imply that these programs will meet all of the run-time conditions of a MTPS program presented at the beginning of this section. Simple PreScheme programs are strongly typed so no operator will be applied to data of the wrong type. `lambda` expressions in Simple PreScheme programs may occur only as initializers in top-level `letrec` bindings or in the operator position of a procedure call. As a result, there is no need to represent closures at run-time.

Simple PreScheme's semantics is inherited from MFPS's semantics.

Chapter 4

The Transformational Compiler

There are six phases in the translation of MTPS into PreScheme Assembly Language.

Parse: Expands derived syntax using rules consistent with those presented in the Scheme standard. In addition, the program's bound variables are renamed so that no variable occurs both bound and free and no variable is bound more than once. Other syntactic checks are made.

Inline primitive operators: A variable bound in a program is changed if its name is also the name of a primitive operator. Consequently, every occurrence of an identifier whose name is the name of a primitive operator refers to that primitive operator. The result is an MFPS program.

Apply transformation rules: Translates an MFPS program into a Simple PreScheme program using meaning-refining transformations. More will be said about this phase later.

Type check: Ensures that a Simple PreScheme program is strongly typed. This phase implements Algorithm W. That algorithm's correctness was demonstrated by Robin Milner [13]. The unifier is based on a published program by Laurence Paulson [17, p. 381].

Compile: Translates the internal representation of a strongly typed Simple PreScheme program into Stack Assembly Language using a syntax-directed compiler as described in Chapter 6. Stack Assembly Language programs are tree structured and execute on a stack machine.

Linearize: Translates a Stack Assembly Language program into PreScheme Assembly Language using a code generator as described in Chapter 8. PreScheme Assembly Language programs are linear and run on a register machine.

The first four phases taken together constitute the MTPS Front End.

Compilers for the MTPS language do not accept all syntactically correct programs because many MFPS programs cannot be translated into Simple PreScheme. Furthermore, the set of MFPS programs that can be translated by a particular compiler depends on the transformation rules used by that compiler.

While there is no precise characterization of what is a compilable MTPS program, knowledgeable programmers know one when they see one. Their intuition is based on an understanding of the run-time conditions and the knowledge that existing compilers perform β-conversion, procedure inlining, and closure hoisting.

The actions of every phase of the MTPS Front End are straightforward with the exception of the actions of the phase that translates MFPS programs into Simple PreScheme ones. Programs are transformed by applying meaning-refining rules. The rules are meaning-refining in a sense made precise in Section 4.3.

The selection and application of transformation rules is performed by an intricate set of procedures. It is worth stressing that these control procedures modify the program only by applying transformation rules. Consequently, if all of the transformation rules are meaning-refining, a control procedure will be meaning-refining itself, unless it runs forever (which can happen for some inputs). Therefore, our verification effort focused solely on the transformation rules.

4.1 Transformation Rules

Each transformation rule is a conditional rewrite rule that operates on MFPS expressions. (The symbol E and the word "expression" will mean an MFPS expression throughout the rest of this section.) It has a pattern, a predicate, and a replacement. Patterns contain syntax variables. An expression matches a pattern if there is an assignment of terms to syntax variables which produces the expression from the pattern by substitution. Each syntax variable is associated with a syntactic category and terms assigned to it must be from that category. The rewrite is performed if the matching expression satisfies the predicate. A rule with pattern E_0 and replacement E_1 is written $\mathrm{E}_0 \Longrightarrow \mathrm{E}_1$, and its predicate is given in the text. The predicate for rules with no restrictions given in the text is always satisfied.

In many systems using rewrite rules, the replacing expression is derived from the replacement pattern by substitution using the assignment produced during matching. This system avoids name conflicts by ensuring that all expressions are α-converted.

Definition 4.1 *An expression is* α-converted *if no variable in it occurs both bound and free or is bound more than once.*

The system avoids name conflicts by making a change of bound variables in the term substituted for each occurrence of a syntax variable during the construction of the replacing expression. Contexts are often used in rule presentations to help express the renaming requirement.

Definition 4.2 *A context, $C[\,]$, is an expression with some holes.*

- $[\,]$ *is a context.*
- K *is a context.*
- I *is a context.*
- *If* $C_0[\,], \ldots, C_n[\,]$ *are contexts, then so is* $(C_0[\,] \ldots C_n[\,])$.
- *If* $C[\,]$ *is a context, then so is* (`lambda` (I^*) $C[\,]$).
- *If* $C_0[\,], \ldots, C_n[\,]$ *are contexts, then so is* (`begin` $C_0[\,] \ldots C_n[\,]$).
- *If* $C_0[\,], \ldots, C_n[\,]$ *are contexts, then so is*

  (`letrec` ((I_1 (`lambda` (I^*) $C_1[\,]$)) ...) $C_0[\,]$).

- *If* $C_0[\,]$, $C_1[\,]$, *and* $C_2[\,]$ *are contexts, then so is* (`if` $C_0[\,]$ $C_1[\,]$ $C_2[\,]$).
- *If* $C[\,]$ *is a context, then so is* (`set!` I $C[\,]$).

In this definition,the expression (`if #f #f`) is considered to be a constant. This slight abuse of the notation will be repeated in the remainder of the text.

Definition 4.3 *A context substitution, $C[\mathrm{E}]$, is a context in which each hole in $C[\ ]$ has been replaced with a copy of* E *in which every bound variable has been renamed using a fresh variable.*

As a consequence, if E is α-converted and E_1 and $C[\mathrm{E}_2]$ are contained in E, then $C[\mathrm{E}_1]$ is α-converted.[1]

Another way to avoid name conflicts is to use de Bruijn's nameless terms [1, Appendix C]. Their use was considered too late in the project to be taken seriously.

4.1.1 Syntactic Predicates

To be applicable to an expression, every rule has a pattern that must match that expression and a predicate that must be satisfied. For many rules, the predicate is always satisfied, and the matching process solely determines the rule's applicability. A common nontrivial predicate tests if a variable is free in an expression. The definition of three other predicates follow.

Intuitively, an expression is "side-effect free" if it returns a value without modifying the store or accessing the oracle. The importance of a side-effect free expression is that it can be eliminated when its value is ignored.

Definition 4.4 *The* side-effect free *expressions are defined inductively by the following rules:*

- K *is side-effect free.*
- I *is side-effect free.*
- *If the classification of* O *is either side-effect free or invariable (see the list of primitive operators in Chapter 3) and* $\mathrm{E}_1 \ldots \mathrm{E}_n$ *are side-effect free, then so is* $\langle \mathrm{O}\ \mathrm{E}_1 \ldots \mathrm{E}_n \rangle$.
- (`lambda` (I^*) E) *is side-effect free.*
- *If* $\mathrm{E}_0, \ldots, \mathrm{E}_n$ *are side-effect free, then so is* (`begin` $\mathrm{E}_0 \ldots \mathrm{E}_n$).
- *If* E *is side-effect free, then so is* (`letrec` (B) E).

[1]In the compiler, each variable is identified by a unique integer. Renaming a bound variable is implemented by reserving an unused integer for the new variable.

- *If* E_0*,* E_1*, and* E_2 *are side-effect free, then so is* (`if` E_0 E_1 E_2)*.*
- *If* $E_0, \ldots, E_n$ *are side-effect free, then so is* (`case` E_0 ((K) E_1) ...)*.*

Intuitively, an expression is "invariable" if it is side-effect free and its value does not depend on mutable variables. The importance of an invariable expression is that its value will remain the same if it is evaluated later during the execution of a program.

Definition 4.5 *The* invariable *expressions are defined inductively by the following rules:*

- K *is invariable.*
- I *is invariable if it is immutable.*
- *If the classification of* O *is invariable (see the list of primitive operators in Chapter 3) and* $E_1 \ldots E_n$ *are invariable, then so is* $\langle$O $E_1 \ldots E_n\rangle$*.*
- (`lambda` (I^*) E) *is invariable.*
- *If* $E_0, \ldots, E_n$ *are invariable, then so is* (`begin` $E_0 \ldots E_n$)*.*
- *If* E *is invariable, then so is* (`letrec` (B) E)*.*
- *If* E_0*,* E_1*, and* E_2 *are invariable, then so is* (`if` E_0 E_1 E_2)*.*
- *If* $E_0, \ldots, E_n$ *are invariable, then so is* (`case` E_0 ((K) E_1) ...)*.*

Obviously, every expression which is invariable is also side-effect free.

Intuitively, an expression is "almost side-effect free" if it does not modify the store until its last action.

Definition 4.6 *The* almost side-effect free *expressions are defined inductively by the following rules:*

- K *is almost side-effect free.*
- I *is almost side-effect free.*
- *If* $E_1 \ldots E_n$ *are side-effect free, then* $\langle$O $E_1 \ldots E_n\rangle$ *is almost side-effect free.*

- *If* E_0 *is almost side-effect free, and* $\mathrm{E}_1, \ldots, \mathrm{E}_n$ *are side-effect free, then* ((`lambda` ($\mathrm{I}_1 \ldots \mathrm{I}_n$) E_0) $\mathrm{E}_1 \ldots \mathrm{E}_n$) *is almost side-effect free.*

- (`lambda` (I^*) E) *is almost side-effect free.*

- *If* $\mathrm{E}_0, \ldots, \mathrm{E}_{n-1}$ *are side-effect free, and* E_n *is almost side-effect free, then* (`begin` $\mathrm{E}_0 \ldots \mathrm{E}_n$) *is almost side-effect free.*

- *If* E *is almost side-effect free, then so is* (`letrec` (B) E)*.*

- *If* E_0 *is side-effect free, and* E_1 *and* E_2 *are almost side-effect free, then* (`if` E_0 E_1 E_2) *is almost side-effect free.*

- *If* E_0 *is side-effect free, and* $\mathrm{E}_1, \ldots, \mathrm{E}_2$ *are almost side-effect free, then* (`case` E_0 ((K) E_1)...) *is almost side-effect free.*

- *If* E *is side-effect free, then* (`set!` I E) *is almost side-effect free.*

4.1.2 The List of Rules

Here is a list of the implemented rules. The rules assume that all expressions are α-converted.

1. Simplification of constants and primitive operator invocations.

   - Evaluate constant expressions.
   - Simplify operations applied to identity elements.

     $\langle + \ 0 \ \mathrm{E}\rangle \Longrightarrow \mathrm{E}$
     $\langle * \ 1 \ \mathrm{E}\rangle \Longrightarrow \mathrm{E}$

   - Move constants to first operand for commutative operators.

     $\langle \mathrm{O \ E \ K}\rangle \Longrightarrow \langle \mathrm{O \ K \ E}\rangle$

   - Move associative operators to the first operand.

     $\langle \mathrm{O} \ \mathrm{E}_0 \ \langle \mathrm{O} \ \mathrm{E}_1 \ \mathrm{E}_2\rangle\rangle \Longrightarrow \langle \mathrm{O} \ \langle \mathrm{O} \ \mathrm{E}_0 \ \mathrm{E}_1\rangle \ \mathrm{E}_2\rangle$

- Use special rules for difference, arithmetic shift, and address arithmetic. The rule for `vector-set!` is not shown.

  $\langle -\ \mathrm{E}\ \mathrm{K}\rangle \Longrightarrow \langle +\ \mathrm{E}\ {-\mathrm{K}}\rangle$
  $\langle \texttt{ashl}\ \langle \texttt{ashl}\ \mathrm{E}_0\ \mathrm{E}_1\rangle\ \mathrm{E}_2\rangle \Longrightarrow \langle \texttt{ashl}\ \mathrm{E}_0\ \langle +\ \mathrm{E}_1\ \mathrm{E}_2\rangle\rangle$
  $\langle \texttt{addr+}\ \langle \texttt{addr+}\ \mathrm{E}_0\ \mathrm{E}_1\rangle\ \mathrm{E}_2\rangle \Longrightarrow \langle \texttt{addr+}\ \mathrm{E}_0\ \langle +\ \mathrm{E}_1\ \mathrm{E}_2\rangle\rangle$
  $\langle \texttt{vector-ref}\ \langle \texttt{addr+}\ \mathrm{E}_0\ \mathrm{E}_1\rangle\ \mathrm{E}_2\rangle$
  $\quad\Longrightarrow \langle \texttt{vector-ref}\ \mathrm{E}_0\ \langle +\ \mathrm{E}_1\ \mathrm{E}_2\rangle\rangle$
  $\langle \texttt{addr+}\ \mathrm{E}\ 0\rangle \Longrightarrow \mathrm{E}$

- Introduce a `let` for some primitive operators.

  $\langle \mathrm{O}\ \mathrm{E}^*\rangle \Longrightarrow ((\texttt{lambda}\ (\mathrm{I}^*)\ \langle \mathrm{O}\ \mathrm{I}^*\rangle)\ \mathrm{E}^*)$

  when E^* contains a combination or a `begin` expression.

2. Conditional expression simplification (`if` and `case`).

- Select branch when the test is a constant.

  $(\texttt{if}\ \mathrm{K}\ \mathrm{E}_1\ \mathrm{E}_2) \Longrightarrow \mathrm{E}_2$ if K is false, else E_1
  $(\texttt{case}\ \mathrm{K}\ldots((i)\ \mathrm{E}_i)\ldots) \Longrightarrow \mathrm{E}_i$ if K is i

- Eliminate a conditional in a boolean expression or rotate conditional when unspecified is the consequent.

  $(\texttt{if}\ \langle \mathrm{O}_0\ \mathrm{E}^*\rangle\ \texttt{\#t}\ \texttt{\#f}) \Longrightarrow \langle \mathrm{O}_0\ \mathrm{E}^*\rangle$
  $(\texttt{if}\ \langle \mathrm{O}_0\ \mathrm{E}^*\rangle\ \texttt{\#f}\ \texttt{\#t}) \Longrightarrow \langle \mathrm{O}_1\ \mathrm{E}^*\rangle$
  $(\texttt{if}\ \langle \mathrm{O}_0\ \mathrm{E}^*\rangle\ (\texttt{if}\ \texttt{\#f}\ \texttt{\#f})\ \mathrm{E})$
  $\quad\Longrightarrow (\texttt{if}\ \langle \mathrm{O}_1\ \mathrm{E}^*\rangle\ \mathrm{E}\ (\texttt{if}\ \texttt{\#f}\ \texttt{\#f}))$

  when O_0 returns a boolean value, O_1 is its complement, and E is not `(if #f #f)`.

- Raise a combination in a test.

  $(\texttt{if}\ (\mathrm{E}_0\ \mathrm{E}^*)\ \mathrm{E}_1\ \mathrm{E}_2)$
  $\quad\Longrightarrow ((\texttt{lambda}\ (\mathrm{I})\ (\texttt{if}\ \mathrm{I}\ \mathrm{E}_1\ \mathrm{E}_2))\ (\mathrm{E}_0\ \mathrm{E}^*))$
  $(\texttt{case}\ (\mathrm{E}\ \mathrm{E}^*)\ \ldots)$
  $\quad\Longrightarrow ((\texttt{lambda}\ (\mathrm{I})\ (\texttt{case}\ \mathrm{I}\ \ldots))\ (\mathrm{E}\ \mathrm{E}^*))$

  Used when the operator in the combination is a `lambda` expression.

- Raise a `begin` in a test.

  $(\texttt{if}\ (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}_0)\ \mathrm{E}_1\ \mathrm{E}_2)$
  $\quad\Longrightarrow ((\texttt{lambda}\ (\mathrm{I})\ (\texttt{if}\ \mathrm{I}\ \mathrm{E}_1\ \mathrm{E}_2))\ (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}_0))$
  $(\texttt{case}\ (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E})\ \ldots)$
  $\quad\Longrightarrow ((\texttt{lambda}\ (\mathrm{I})\ (\texttt{case}\ \mathrm{I}\ \ldots))\ (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}))$

- Simplify an `if` in the test of an `if`.

  $(\texttt{if}\ (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{E}_3\ \mathrm{E}_4)$
  $\quad\Longrightarrow (\texttt{if}\ \mathrm{E}_0\ (\texttt{if}\ \mathrm{E}_1\ \mathrm{E}_3\ \mathrm{E}_4)\ (\texttt{if}\ \mathrm{E}_2\ \mathrm{E}_3\ \mathrm{E}_4))$

  Used when both E_3 and E_4 are constants or variables.

- Simplify an `if` in the consequent of an `if`.

  $(\texttt{if}\ \mathrm{E}_0\ (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{E}_3) \Longrightarrow (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_3)$
  $(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_2\ \mathrm{E}_3)) \Longrightarrow (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_3)$

  when E_0 is side-effect free.

3. `begin` introduction.

   $$((\texttt{lambda}\ (\mathrm{I})\ \mathrm{E}_1)\ \mathrm{E}_0) \Longrightarrow (\texttt{begin}\ \mathrm{E}_0\ \mathrm{E}_1)$$

   when I is not free in E_1.

4. `begin` simplification.

   $$(\texttt{begin}\ \mathrm{E}_0^*\ (\texttt{begin}\ \mathrm{E}_1^*)\ \mathrm{E}_2^*) \Longrightarrow (\texttt{begin}\ \mathrm{E}_0^*\ \mathrm{E}_1^*\ \mathrm{E}_2^*)$$
   $$(\texttt{begin}\ \mathrm{E}_0 \ldots \mathrm{E}_i \ldots \mathrm{E}_n) \Longrightarrow (\texttt{begin}\ \mathrm{E}_0 \ldots \mathrm{E}_{i-1}\ \mathrm{E}_{i+1} \ldots \mathrm{E}_n)$$

   when E_i is side-effect free and $i < n$.

5. `lambda` expression naming. Name anonymous `lambda` expressions which are not in the operator position of a combination.

   $$(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}) \Longrightarrow (\texttt{letrec}\ ((\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E})))\ \mathrm{I})$$

   where I is a fresh variable so it is not free in E.

6. β-substitution. Substitute for a variable when it is `lambda` bound to an invariable expression. Alternatively, substitute for a variable when it is `lambda` bound in a call in which all of the arguments are side-effect free and the body is almost side-effect free.

   The rule is used when the variable is bound to a constant, another variable, or when the variable is referenced at most once.

   $$\begin{array}{l}((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n) \\ \quad \Longrightarrow ((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{E}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)\end{array}$$

   when either E_i is invariable, or when $\mathrm{E}_1, \ldots, \mathrm{E}_n$ are side-effect free and $C[\mathrm{I}_i]$ is almost side-effect free.

7. `lambda` simplification.

   $$((\texttt{lambda}\ ()\ \mathrm{E})) \Longrightarrow \mathrm{E}$$

   and

   $$\begin{array}{l}((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n) \\ \quad \Longrightarrow ((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_{i-1}\ \mathrm{I}_{i+1} \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_{i-1}\ \mathrm{E}_{i+1} \ldots \mathrm{E}_n)\end{array}$$

   when E_i is side-effect free and I_i is not free in E.

8. `letrec` substitution.

   $$\begin{array}{l}(\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}\ \mathrm{E})\ \mathrm{B}_1)\ C[\mathrm{I}]) \\ \quad \Longrightarrow (\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}\ \mathrm{E})\ \mathrm{B}_1)\ C[\mathrm{E}])\end{array}$$

   $$\begin{array}{l}(\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}_i\ \mathrm{E}_i)\ \mathrm{B}_1\ (\mathrm{I}_j\ C[\mathrm{I}_i])\ \mathrm{B}_2)\ \mathrm{E}_0) \\ \quad \Longrightarrow (\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}_i\ \mathrm{E}_i)\ \mathrm{B}_1\ (\mathrm{I}_j\ C[\mathrm{E}_i])\ \mathrm{B}_2)\ \mathrm{E}_0)\end{array}$$

9. `letrec` simplification.

   $$(\texttt{letrec}\ ()\ \mathrm{E}) \Longrightarrow \mathrm{E}$$

   and

   $$\begin{array}{l}(\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}_i\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_i))\ \mathrm{B}_1)\ \mathrm{E}) \\ \quad \Longrightarrow (\texttt{letrec}\ (\mathrm{B}_0\ \mathrm{B}_1)\ \mathrm{E})\end{array}$$

   when I_i is referenced nowhere except in E_i.

10. `letrec` lifting.

    C[(`letrec` (B) E)] $\Longrightarrow$ (`letrec` (B) C[E])

    when C[ ] has one hole and binds no free variables of B. Since C[ ] has only one hole, there is no need to perform variable renaming.

11. `letrec` binding merging.

    (`letrec` (B_0 (I (`lambda` (I^*) (`letrec` (B_1) E))) B_2) E_0)
    $\Longrightarrow$ (`letrec` (B_0 (I (`lambda` (I^*) E)) B_1 B_2) E_0)

    when I^* are not free in B_1.

12. `letrec` expression merging.

    (`letrec` (B_0) (`letrec` (B_1) E)) $\Longrightarrow$ (`letrec` (B_0 B_1) E)

13. `letrec` elimination.

    (`letrec` ((I E)) I) $\Longrightarrow$ E

    when I is not free in E. Used when the expression is in the operator position of a combination.

14. Combination in a combination rotation.

    (E_0 ((`lambda` (I^*) E_1) E^*)) $\Longrightarrow$ ((`lambda` (I^*) (E_0 E_1)) E^*)

    when E_0 is invariable.

15. `begin` in a combination rotation.

    (E_0 (`begin` E^* E)) $\Longrightarrow$ (`begin` E^* (E_0 E))

    when E_0 is invariable.

16. Defined constant substitution. If an immutable variable is defined to be a constant or another immutable variable, the value is universally substituted. See Figure 4.1.

17. Unused initializer elimination. If a defined immutable variable is never referenced and it's initializer is a side-effect free expression, the initializer is replaced with (`if #f #f`).

```
(define $\mathrm{I}_1$) ... (define $\mathrm{I}_i$) ... (define $\mathrm{I}_n$)
(letrec (($\mathrm{I}_{n+1}$ $C_{n+1}[\mathrm{I}_i]$) ...)
  (begin
    (set! $\mathrm{I}_1$ $C_1[\mathrm{I}_i]$)
      ⋮
    (set! $\mathrm{I}_i$ $\mathrm{E}_i$)
      ⋮
    (set! $\mathrm{I}_n$ $C_n[\mathrm{I}_i]$)
    $C_0[\mathrm{I}_i]$))
$\Longrightarrow$
(define $\mathrm{I}_1$) ... (define $\mathrm{I}_i$) ... (define $\mathrm{I}_n$)
(letrec (($\mathrm{I}_{n+1}$ $C_{n+1}[\mathrm{E}_i]$) ...)
  (begin
    (set! $\mathrm{I}_1$ $C_1[\mathrm{E}_i]$)
      ⋮
    (set! $\mathrm{I}_i$ $\mathrm{E}_i$)
      ⋮
    (set! $\mathrm{I}_n$ $C_n[\mathrm{E}_i]$)
    $C_0[\mathrm{E}_i]$))
```

when I_i is immutable and E_i is a constant or an immutable variable reference.

Figure 4.1: Defined Constant Substitution

4.2 Usage of the Rules

The rules result in program transformations similar to those produced by other compilers [12, 10]. The rules for conditionals are the same, and the rules for β-conversion look different only to facilitate correctness proofs. The `letrec` rules implement the inlining of procedures and closure hoisting.

One major difference between this compiler and the others is it does not convert the program into continuation-passing style [18]. As a result, the rotate combinations rule was added. Here is a common example of its use.

$$\begin{array}{l}(\texttt{let}\ ((I_0\ (\texttt{let}\ ((I_1\ E_1)\ldots)\ E^*\ I_1)))\ E_0) \\ \quad \Longrightarrow (\texttt{let}\ ((I_1\ E_1)\ldots)\ (\texttt{let}\ ((I_0\ I_1))\ E^*\ E_0))\end{array}$$

The rules are used as follows. With the exception of the `letrec` substitution rule (Rule 8) and the defined constant substitution rule (Rule 16), all of the rules are applied by an expression simplifier. The simplifier always terminates. After the initial simplification, expressions are maintained in simplified form by the use of expression constructors that invoke the simplifier.

The next step is to repeatedly try defined constant substitution until there is no place it can be applied. This is followed by `letrec` substitution. If the `letrec` substitution phase applies no rules, the process terminates, otherwise, a new cycle of defined constant and `letrec` substitution is initiated.

`letrec` substitution replaces a `letrec` bound variable which occurs in the operator position of a combination with its corresponding initializer. The `letrec` substitution phase has two modes. It substitutes any binding which binds a `lambda` expression containing a `letrec` expression. In these bindings, the `letrec` lifting rule has failed to hoist a closure so the substitution is required.

In the second mode, it substitutes any binding which binds a `lambda` expression having a simple body or one which has been marked by the use of a `define-integrable` form. A simple `lambda` body is a constant, a variable, a combination in which the operator is a variable and the operands are either variables or constants, or a primitive operator in which the arguments are either variables or constants.

Programmers beware: the `letrec` substitution phase may never terminate as the following program demonstrates.

```
(define number 77)                        ; This program
(define (dec x) (- x 1))                  ; produces zero
(define (even x)                          ; if NUMBER is even
  (if (zero? x)                           ; and nonnegative,
      0                                   ; otherwise it
      (odd (dec x))))                     ; produces one.
(define (odd x)
  (if (zero? x)
      1
      (even (dec x))))
(if (negative? number)
    1
    (even number))
```

Figure 4.2: Example MTPS Program

```
(define *x* 2)
(define-integrable (loop x)
  (if (positive? x) (loop (- x 1)) x))
(loop *x*)
```

However, the `letrec` substitution phase may be used to force computations at compile time. All compilers must unwind the loop in the following example.

```
(define-integrable (floor-log2 x a)
  (if (<= x 1)
      a
      (floor-log2 (ashr x 1) (+ 1 a))))

(define log-bytes-per-word
  (floor-log2 bytes-per-word 0))

(if (not (= (ashl 1 log-bytes-per-word) bytes-per-word))
    (err 1 "Word size not a power of two"))
```

As an example, consider the even-odd program given in Figure 4.2. It specifies a simple iterative process. Figures 4.3 through 4.8 show how this

```
(define number)
(define dec)
(define even)
(define odd)
(begin
  (set! number 77)
  (set! dec (lambda (x) (- x 1)))
  (set! even (lambda (y) (if (= 0 y) 0 (odd (dec y)))))
  (set! odd (lambda (z) (if (= 0 z) 1 (even (dec z)))))
  (if (> 0 number) 1 (even number)))
```

Figure 4.3: Expand into Macro-Free PreScheme

program might be translated into Simple PreScheme using transformations implemented in the Front End.

4.3 Justification of the Rules

A rule is justified if each application of it transforms an MFPS program into another program which is syntactically well-formed and has the same meaning. Some of the rules have another interesting property—they can transform a program which has bottom denotation into a program which produces a nonbottom answer. For example, the program

```
(define two (+ 1 one))
(define one 1)
two
```

is transformed into a program which produces the answer 2!

This odd behavior is tolerated so as to allow constant propagation without performing a dependency analysis. In the above example, `1` is substituted for the occurrence of the immutable variable `one` even though ($\mathit{undefined}$ in E) should have been substituted. In summary, the application of a rule is justified if it does not affect nonbottom computational results.

Definition 4.7 *A P-context is a program with some holes. If* $C[\,]$ *is a context, then* (`define` I_1)...(`define` I_n) $C[\,]$ *is a P-context.*

```
(define number) (define dec) (define even) (define odd)
(begin
  (set! number 77)
  (set! dec (letrec ((dec-0 (lambda (x) (- x 1))) dec-0)))
  (set! even (letrec ((even-0 (lambda (y)
                                (if (= 0 y)
                                    0
                                    (odd (dec y))))))
                      even-0)))
  (set! odd (letrec ((odd-0 (lambda (z)
                              (if (= 0 z)
                                  1
                                  (even (dec z))))))
                     odd-0)))
  (if (> 0 number) 1 (even number)))
```

Figure 4.4: Name Anonymous Lambda Expressions

```
(define number) (define dec) (define even) (define odd)
(letrec
    ((dec-0 (lambda (x) (- x 1)))
     (even-0 (lambda (y) (if (= 0 y) 0 (odd (dec y)))))
     (odd-0 (lambda (z) (if (= 0 z) 1 (even (dec z))))))
  (begin
    (set! number 77)
    (set! dec dec-0)
    (set! even even-0)
    (set! odd odd-0)
    (if (> 0 number) 1 (even number))))
```

Figure 4.5: Closure Hoisting

```
(define number) (define dec) (define even) (define odd)
(letrec
    ((dec-0 (lambda (x) (- x 1)))
     (even-0 (lambda (y) (if (= 0 y) 0 (odd-0 (dec-0 y)))))
     (odd-0 (lambda (z) (if (= 0 z) 1 (even-0 (dec-0 z))))))
  (begin
    (set! number 77) (set! dec dec-0)
    (set! even even-0) (set! odd odd-0)
    (if (> 0 77)                        ; Which simplifies
        1                               ; to (even-0 77).
        (even-0 77))))
```

Figure 4.6: Constant Folding

```
(define number) (define dec) (define even) (define odd)
(letrec
    ((dec-0 (lambda (x) (- x 1)))
     (even-0 (lambda (y) (if (= 0 y) 0 (odd-0 (- y 1)))))
     (odd-0 (lambda (z) (if (= 0 z) 1 (even-0 (- z 1))))))
  (begin
    (set! number 77) (set! dec dec-0)
    (set! even even-0) (set! odd odd-0)
    (even-0 77)))
```

Figure 4.7: Inline Selected Procedure Calls

```
(define number) (define dec) (define even) (define odd)
(letrec
    ((even-0 (lambda (y) (if (= 0 y) 0 (odd-0 (- y 1)))))
     (odd-0 (lambda (z) (if (= 0 z) 1 (even-0 (- z 1))))))
  (begin
    (set! number (if #f #f)) (set! dec (if #f #f))
    (set! even (if #f #f)) (set! odd (if #f #f))
    (even-0 77)))
```

Figure 4.8: Eliminate Unused Lambda Expressions and Initializers

Definition 4.8 *A transformation rule is* meaning-refining *if for all P-contexts, $P[\ ]$, and for all expressions* E_0 *and* E_1*, if* $P[\mathrm{E}_0]$ *and* $P[\mathrm{E}_1]$ *are α-converted and the rule rewrites* E_0 *into* E_1*, then* $\mathcal{P}[\![P[\mathrm{E}_0]]\!] \sqsubseteq \mathcal{P}[\![P[\mathrm{E}_1]]\!]$.

The following lemma provides a convenient sufficient condition for a transformation rule to be meaning-refining.

Lemma 4.9 *Let $P[\,]$ be a P-context, $P[\mathrm{E}_0]$ and $P[\mathrm{E}_1]$ be α-converted, and* E_1 *be the result of applying a rule to* E_0*. Then* $\mathcal{P}[\![P[\mathrm{E}_0]]\!] \sqsubseteq \mathcal{P}[\![P[\mathrm{E}_1]]\!]$ *provided, for all ρ, κ, σ, o, $\mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o = \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o$ or $\mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o = \bot_A$.*

Proof: The condition of the lemma implies $\forall\rho\kappa\sigma o, \mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o \sqsubseteq \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o$ since the answer domain A is flat. Then $\forall\rho\kappa\sigma o, \mathcal{D}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o \sqsubseteq \mathcal{D}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o$ since $\mathcal{D}[\![\mathrm{E}]\!] = \mathcal{E}[\![\mathrm{E}]\!]$. This implies $\forall\rho\kappa\sigma o, \mathcal{D}[\![P[\mathrm{E}_0]]\!]\rho\kappa\sigma o \sqsubseteq \mathcal{D}[\![P[\mathrm{E}_1]]\!]\rho\kappa\sigma o$ since $P[\ ]$ is a P-context. Also, for all o and all $\iota \neq$ *root_tid*, *compute_thread_answer* $o\iota$ does not depend on $P[\mathrm{E}_0]$ or $P[\mathrm{E}_1]$. $\mathcal{P}[\![P[\mathrm{E}_0]]\!] \sqsubseteq \mathcal{P}[\![P[\mathrm{E}_1]]\!]$ follows from these last two facts. ∎

Ideally, a justification of the translation rules should be no less than a proof that each rule is meaning-refining. Since such a proof would be quite long and involve many tedious details, the proof we present in this section focuses on aspects of the rules which are likely to cause problems. After all, the purpose of justifying the rules is to gain confidence in the correctness of the compiler. While developing the compiler, several proposed rules were shown to have predicates which enable their application in contexts which did not refine the meaning of a program. These rules were modified or eliminated.

Our proof consists of a number of smaller proofs—each one shows that a particular rule satisfies the sufficient condition of Lemma 4.9. Proofs for rules that are obviously meaning-refining have been omitted with one exception. The justification of the `if` in the consequent of an `if` rule) has been included as an illustration. Proofs for the rules that simplify expressions involving primitive operators have also been omitted since a formal semantics has been provided for just a few of the primitive operators. Many of the proofs that are given employ structural induction involving a large number of cases. Only the most interesting cases are considered in these proofs. Since the compiler avoids name conflicts by using α-converted expressions, issues arising from name conflicts need not be addressed.

The proofs for rules with nontrivial predicates require associating semantics properties with syntactic ones, so we introduce the following definitions.

Definition 4.10 *An expression* E *is* evaluative *if*

$$\forall\rho\sigma, \exists\epsilon, \forall\kappa o,\ \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma o = \kappa\epsilon\sigma o \textit{ or } \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma o = \bot_A.$$

This definition says that an expression is evaluative if each evaluation of the expression invokes the current expression continuation on the resulting value without modifying the store or accessing the oracle, or generates an error. In particular, the dependency of $\mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma$ on E is captured completely by ϵ.

Definition 4.11 *An expression* E *is* uniformly evaluative *if*

$$\begin{aligned}&\forall\rho\sigma, \exists\epsilon, \forall\sigma',\ \sigma \preceq_\rho \sigma' \textit{ implies}\\ &\forall\kappa o, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma' o = \kappa\epsilon\sigma' o \textit{ or } \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma' o = \bot_A.\end{aligned}$$

Notice that E is evaluative if E is uniformly evaluative since every store is a successor of itself (relative to ρ).

Theorem 4.12 *If* E *is side-effect free, then* E *is evaluative.*

Proof sketch: This is proved by structural induction on side-effect free expressions. The cases of E being I and (`if` E_0 E_1 E_2) are shown.

Case E = I. Fix ρ and σ. Let

$$\epsilon = \rho\mathrm{I} \in E \to \rho\mathrm{I} \mid E, \sigma(\rho\mathrm{I} \mid L) \downarrow 1.$$

Expanding definitions gives

$$\mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma = \kappa\epsilon\sigma.$$

Notice ϵ is independent of κ and o so

$$\forall\kappa o,\ \mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma o = \kappa\epsilon\sigma o.$$

Case E = (`if` E_0 E_1 E_2). Fix ρ and σ. By the induction hypothesis, there is some ϵ_0 such that

$$\forall\kappa o,\ \mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o = \kappa\epsilon\sigma o \text{ or } \bot_A$$

Then

$$\begin{aligned}&\forall\kappa o,\ \mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![\mathrm{E}_0]\!]\rho(\lambda\epsilon.\, \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa)\sigma o\\ &\quad = \mathit{truish}\,\epsilon_0 \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o, \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa\sigma o \text{ or } \bot_A.\end{aligned}$$

When $\epsilon_0 = \mathit{false}$,

$$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\rho\kappa\sigma o = \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa\sigma o \text{ or } \bot_A;$$

otherwise,

$$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\rho\kappa\sigma o = \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o \text{ or } \bot_A.$$

Use of the induction hypothesis verifies both alternatives. ∎

Theorem 4.13 *If* E *is invariable, then* E *is uniformly evaluative.*

Proof sketch: This is proved by structural induction on invariable expressions; most of the proof is identical to the proof of the previous theorem. The cases of E being I and $\langle +\ \mathrm{E}_1\ \mathrm{E}_2\rangle$ are shown.

Case E = I. Fix ρ and σ. Since E is invariable, I must be immutable. Let

$$\epsilon = \rho\mathrm{I} \in E \to \rho\mathrm{I} \mid E, \sigma(\rho\mathrm{I} \mid L) \downarrow 1,$$

and let σ' be any store such that $\sigma \preceq_\rho \sigma'$. Reasoning as above,

$$\forall\kappa o,\ \mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma' o = \kappa\epsilon\sigma' o$$

since $\sigma \preceq_\rho \sigma'$.

Case E $= \langle +\ \mathrm{E}_1\ \mathrm{E}_2\rangle$. Fix ρ and σ. By the induction hypothesis, there are ϵ_1 and ϵ_2 such that, for all σ' with $\sigma \preceq_\rho \sigma'$,

$$\forall\kappa o,\ \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma' o = \kappa\epsilon_1\sigma' o \text{ or } \bot_A$$

and

$$\forall\kappa o,\ \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa\sigma' o = \kappa\epsilon_2\sigma' o \text{ or } \bot_A$$

Let $\epsilon = ((\epsilon_1 \mid R + \epsilon_2 \mid R) \text{ in } E)$, and let σ' be any store such that $\sigma \preceq_\rho \sigma'$. Expanding definitions and using the two equations above gives

$$\begin{aligned}
&\forall\kappa o,\ \mathcal{E}[\![\langle+\rangle\ \mathrm{E}_1\ \mathrm{E}_2]\!]\rho\kappa\sigma' o\\
&\quad = \mathcal{E}[\![+]\!]\rho\lambda\epsilon_0.\ \mathit{applicate}\,\epsilon_0\langle\epsilon_1,\epsilon_2\rangle\kappa\sigma' o \text{ or } \bot_A\\
&\quad = \mathit{applicate}\,(\rho + \mid E)\langle\epsilon_1,\epsilon_2\rangle\kappa\sigma' o \text{ or } \bot_A\\
&\quad = \mathit{applicate}\,(\mathcal{O}[\![+]\!] \text{ in } E)\langle\epsilon_1,\epsilon_2\rangle\kappa\sigma' o \text{ or } \bot_A\\
&\quad = \mathcal{O}[\![+]\!]\langle\epsilon_1,\epsilon_2\rangle\kappa\sigma' o \text{ or } \bot_A\\
&\quad = \kappa\epsilon\sigma' o \text{ or } \bot_A
\end{aligned}$$

∎

The following two obvious lemmas aid in the proofs of the rules.

Lemma 4.14 *When* $\mathrm{I}_0^* \S \mathrm{I}_1^*$ *are distinct,*

$$\mathit{extends}(\mathit{extends}\,\rho\mathrm{I}_0^*\epsilon_0^*)\mathrm{I}_1^*\epsilon_1^* = \mathit{extends}\,\rho(\mathrm{I}_0^* \S \mathrm{I}_1^*)(\epsilon_0^* \S \epsilon_1^*).$$

Lemma 4.15

$$\begin{aligned}
&\mathcal{B}[\![\mathrm{B}_0\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_0\mathrm{B}_1]\!])\rho\\
&\quad = \lambda\epsilon^*.\ \mathcal{B}[\![\mathrm{B}_0]\!](\mathcal{I}[\![\mathrm{B}_0]\!])\\
&\qquad\qquad\qquad (\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad\qquad (\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0)\\
&\qquad\qquad \S\ \mathcal{B}[\![\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_1]\!])\\
&\qquad\qquad\qquad\quad (\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0]\!])(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad\qquad\quad (\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0)
\end{aligned}$$

4.3.1 `if` in the Consequent of an `if`

Theorem 4.16 *When* E_0 *is side-effect free,*

$$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{E}_3)]\!]\rho\kappa\sigma o = \mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_3)]\!]\rho\kappa\sigma o.$$

Proof: By Theorem 4.12, there exists an ϵ_0 such that

$$\forall\kappa o,\ \mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o = \kappa\epsilon_0\sigma o \text{ or } \bot_A.$$

Then

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{E}_3)]\!]\rho\kappa\sigma o\\
&\quad = \mathcal{E}[\![\mathrm{E}_0]\!]\rho(\lambda\epsilon.\, \mathit{truish}\,\epsilon \to \mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\rho\kappa, \mathcal{E}[\![\mathrm{E}_3]\!]\rho\kappa)\sigma o\\
&\quad = \mathit{truish}\,\epsilon_0 \to \mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\rho\kappa\sigma o, \mathcal{E}[\![\mathrm{E}_3]\!]\rho\kappa\sigma o \text{ or } \bot_A\\
&\quad = \mathit{truish}\,\epsilon_0 \to \mathcal{E}[\![\mathrm{E}_0]\!]\rho(\lambda\epsilon.\, \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa)\sigma o,\\
&\qquad\quad \mathcal{E}[\![\mathrm{E}_3]\!]\rho\kappa\sigma o \text{ or } \bot_A\\
&\quad = \mathit{truish}\,\epsilon_0 \to (\mathit{truish}\,\epsilon_0 \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o, \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa\sigma o),\\
&\qquad\quad \mathcal{E}[\![\mathrm{E}_3]\!]\rho\kappa\sigma o \text{ or } \bot_A\\
&\quad = \mathit{truish}\,\epsilon_0 \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma o, \mathcal{E}[\![\mathrm{E}_3]\!]\rho\kappa\sigma o \text{ or } \bot_A\\
&\quad = \mathcal{E}[\![\mathrm{E}_0]\!]\rho(\lambda\epsilon.\, \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \mathcal{E}[\![\mathrm{E}_3]\!]\rho\kappa)\sigma o\\
&\quad = \mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_3)]\!]\rho\kappa\sigma o
\end{aligned}$$

∎

4.3.2 `lambda` Simplification

Theorem 4.17 *When* E_i *is side-effect free and* I_i *is not free in* E*,*

$$\begin{aligned}
&\mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\\
&\quad \sqsubseteq \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_{i-1}\ \mathrm{I}_{i+1} \ldots \mathrm{I}_n)\ \mathrm{E})\\
&\qquad\quad \mathrm{E}_1 \ldots \mathrm{E}_{i-1}\ \mathrm{E}_{i+1} \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o.
\end{aligned}$$

Proof: Clearly, the theorem holds if the LHS of the relation equals $\bot_A$, so assume otherwise. This means that we have to prove

$$\begin{aligned}
&\mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\\
&\quad = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_{i-1}\ \mathrm{I}_{i+1} \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_{i-1}\ \mathrm{E}_{i+1} \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o.
\end{aligned}$$

Expanding definitions we obtain

$$\begin{aligned}
&\mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\\
&\quad = \mathcal{E}^*[\![\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n]\!]\rho\psi\sigma o
\end{aligned}$$

where

$$\psi = \lambda\epsilon^*.\ \mathit{applicate}(\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})]\!]\rho)\epsilon^*\kappa.$$

The proof is easy to complete if the evaluation of one of $\mathrm{E}_1, \ldots, \mathrm{E}_i, \ldots, \mathrm{E}_n$ errs or halts relative to ρ, σ, o; so assume otherwise. By using Axiom 3.2 with $\mathrm{E}_1, \ldots, \mathrm{E}_{i-1}, \mathrm{E}_{i+1}, \ldots, \mathrm{E}_n$ and Theorem 4.12 with E_i, there exists $\epsilon_1, \ldots, \epsilon_i, \ldots, \epsilon_n, \sigma', o'$ such that

$$\mathcal{E}^*[\![\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n]\!]\rho\psi\sigma = \psi\langle\epsilon_1 \ldots \epsilon_i \ldots \epsilon_n\rangle\sigma' o'.$$

Since the evaluation of E_i does not change the store or access the oracle,

$$\mathcal{E}^*[\![\mathrm{E}_1 \ldots \mathrm{E}_{i-1}\ \mathrm{E}_{i+1} \ldots \mathrm{E}_n]\!]\rho\psi\sigma = \psi\langle\epsilon_1, \ldots, \epsilon_{i-1}, \epsilon_{i+1}, \ldots, \epsilon_n\rangle\sigma' o'.$$

By these equations and β-reduction, we have to only prove

$$\begin{aligned}&\mathit{applicate}(\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})]\!]\rho)\langle\epsilon_1, \ldots, \epsilon_i, \ldots, \epsilon_n\rangle\kappa\sigma' o' \\ &\quad = \mathit{applicate}\,(\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_{i-1}\ \mathrm{I}_{i+1} \ldots \mathrm{I}_n)\ \mathrm{E})]\!]\rho) \\ &\qquad\qquad \langle\epsilon_1, \ldots, \epsilon_{i-1}, \epsilon_{i+1}, \ldots, \epsilon_n\rangle\kappa\sigma' o',\end{aligned}$$

Expanding definitions reduces this equation to

$$\begin{aligned}&\mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\,\rho\langle\mathrm{I}_1, \ldots, \mathrm{I}_i, \ldots, \mathrm{I}_n\rangle\langle\epsilon_1, \ldots, \epsilon_i, \ldots, \epsilon_n\rangle)\kappa\sigma' o' \\ &\quad = \mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\,\rho\langle\mathrm{I}_1, \ldots, \mathrm{I}_{i-1}, \mathrm{I}_{i+1}, \ldots, \mathrm{I}_n\rangle \\ &\qquad \langle\epsilon_1, \ldots, \epsilon_{i-1}, \epsilon_{i+1}, \ldots, \epsilon_n\rangle)\kappa\sigma' o'.\end{aligned}$$

which is then proved by structural induction on E assuming I_i is not free in E. ▮

4.3.3 β-substitution

There are two cases for β-substitution. The expressions substituted can be invariable or side-effect free.

β-substitution of invariable expressions

Theorem 4.18 *When E_i is invariable,*

$$\begin{aligned}&\mathcal{E}[\![\texttt{((lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o \\ &\quad \sqsubseteq \mathcal{E}[\![\texttt{((lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{E}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o.\end{aligned}$$

Note the RHS must be α-converted so I_i cannot be free in E_i. The theorem is proved by appealing to Lemma 4.19 and Lemma 4.20 which follow.

Lemma 4.19 *When* E_i *is invariable,*

$$\begin{aligned}&\mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\\ &\quad\sqsubseteq \mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)(\texttt{(lambda}\ (\mathrm{I}_i)\ C[\mathrm{I}_i])\ \mathrm{E}_i))\\ &\qquad\quad \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o.\end{aligned}$$

Proof: Clearly, the theorem holds if the LHS of the relation equals $\perp_A$, so assume otherwise. This means that we have to prove

$$\begin{aligned}&\mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\\ &\quad= \mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)(\texttt{(lambda}\ (\mathrm{I}_i)\ C[\mathrm{I}_i])\ \mathrm{E}_i))\\ &\qquad\quad \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o.\end{aligned}$$

The proof is easy to complete if the evaluation of one of $\mathrm{E}_1, \ldots, \mathrm{E}_i, \ldots, \mathrm{E}_n$ errs or halts relative to ρ, σ, o; so assume otherwise. By using Axiom 3.2 with $\mathrm{E}_1, \ldots, \mathrm{E}_{i-1}, \mathrm{E}_{i+1}, \ldots, \mathrm{E}_n$ and Theorem 4.13 with E_i, there exists $\epsilon_1, \ldots, \epsilon_i, \ldots, \epsilon_n, \sigma_1, \ldots, \sigma_i, \ldots, \sigma_n, o_1, \ldots, o_i, \ldots, o_n$ such that:

- $\forall\psi, \mathcal{E}^*[\![\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n]\!]\rho\psi\sigma o = \psi\langle\epsilon_1 \ldots \epsilon_i \ldots \epsilon_n\rangle\sigma_n o_n$.
- $\sigma \preceq_\rho \sigma_1 \preceq_\rho \cdots \preceq_\rho \sigma_i \preceq_\rho \cdots \preceq_\rho \sigma_n$.
- $\forall\kappa o, \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma_n o = \kappa\epsilon_i\sigma_n o$.

Let

$$\rho' = \mathit{extends}\ \rho\langle\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n\rangle\langle\epsilon_1 \ldots \epsilon_i \ldots \epsilon_n\rangle.$$

Notice that $\rho' = \mathit{extends}\ \rho'\langle\mathrm{I}_i\rangle\langle\epsilon_i\rangle$ and none of $\mathrm{I}_1 \ldots \mathrm{I}_n$ are free in E_i.

By expanding definitions and using the facts given above, we obtain

$$\begin{aligned}&\mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\\ &\quad= \mathcal{E}^*[\![\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n]\!]\rho(\lambda\langle\epsilon_1 \ldots \epsilon_i \ldots \epsilon_n\rangle.\,\mathcal{E}[\![C[\mathrm{I}_i]]\!]\rho'\kappa)\sigma o\\ &\quad= \mathcal{E}[\![C[\mathrm{I}_i]]\!]\rho'\kappa\sigma_n o_n\\ &\quad= \mathcal{E}[\![C[\mathrm{I}_i]]\!](\mathit{extends}\ \rho'\langle\mathrm{I}_i\rangle\langle\epsilon_i\rangle)\kappa\sigma_n o_n\\ &\quad= \mathcal{E}[\![\mathrm{E}_i]\!]\rho(\lambda\epsilon_i.\,\mathcal{E}[\![C[\mathrm{I}_i]]\!](\mathit{extends}\ \rho'\langle\mathrm{I}_i\rangle\langle\epsilon_i\rangle)\kappa)\sigma_n o_n\\ &\quad= \mathcal{E}[\![\mathrm{E}_i]\!]\rho'(\lambda\epsilon_i.\,\mathcal{E}[\![C[\mathrm{I}_i]]\!](\mathit{extends}\ \rho'\langle\mathrm{I}_i\rangle\langle\epsilon_i\rangle)\kappa)\sigma_n o_n\\ &\quad= \mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_i)\ C[\mathrm{I}_i])\ \mathrm{E}_i)]\!]\rho'\kappa\sigma_n o_n\\ &\quad= \mathcal{E}^*[\![\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n]\!]\\ &\qquad\quad \rho(\lambda\langle\epsilon_1 \ldots \epsilon_i \ldots \epsilon_n\rangle.\,\mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_i)\ C[\mathrm{I}_i])\ \mathrm{E}_i)]\!]\rho'\kappa)\sigma o\\ &\quad= \mathcal{E}[\![(\texttt{(lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)(\texttt{(lambda}\ (\mathrm{I}_i)\ C[\mathrm{I}_i])\ \mathrm{E}_i))\\ &\qquad\quad \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma o\end{aligned}$$

▮

Lemma 4.20 *When* E *is invariable,*

$$\mathcal{E}[\![\texttt{((lambda}\ \text{(I)}\ C[\text{I}])\ \text{E)}]\!]\rho\kappa\sigma o = \mathcal{E}[\![C[\text{E}]]\!]\rho\kappa\sigma o.$$

Proof sketch: Since E is invariable, E is uniformly evaluative by Theorem 4.13, and thus there is some ϵ_0 such that

$$\begin{aligned}&\forall\sigma',\ \sigma \preceq_\rho \sigma' \text{ implies}\\ &\forall\kappa o, \mathcal{E}[\![\text{E}]\!]\rho\kappa\sigma' o = \kappa\epsilon_0\sigma' o \text{ or } \bot_A.\end{aligned}$$

Then

$$\begin{aligned}&\mathcal{E}[\![\texttt{((lambda}\ \text{(I)}\ C[\text{I}])\ \text{E)}]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![\text{E}]\!]\rho(\lambda\epsilon'.\,\mathcal{E}[\![\texttt{(lambda}\ \text{(I)}\ C[\text{I}])]\!]\rho(\lambda\epsilon.\ \mathit{applicate}\ \epsilon\langle\epsilon'\rangle\kappa))\sigma o\\ &\quad = \mathcal{E}[\![\texttt{(lambda}\ \text{(I)}\ C[\text{I}])]\!]\rho(\lambda\epsilon.\ \mathit{applicate}\ \epsilon\langle\epsilon_0\rangle\kappa)\sigma o \text{ or } \bot_A\\ &\quad = \mathit{applicate}(\mathcal{L}[\![\texttt{(lambda}\ \text{(I)}\ C[\text{I}])]\!]\rho)\langle\epsilon_0\rangle\kappa\sigma o \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![C[\text{I}]]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o \text{ or } \bot_A\end{aligned}$$

The rest of the proof is by induction on contexts. The cases of $C[\ ]$ being $[\ ]$ and (`begin` $C_0[\ ]\ C_1[\ ]$) are shown.

Case of $C[\ ] = [\ ]$.

$$\begin{aligned}&\mathcal{E}[\![C[\text{I}]]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![\text{I}]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o \text{ or } \bot_A\\ &\quad = \kappa\epsilon_0\sigma' o \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![\text{E}]\!]\rho\kappa\sigma' o\end{aligned}$$

Case of $C[\ ] =$ (`begin` $C_0[\ ]\ C_1[\ ]$). We must show that

$$\begin{aligned}&\mathcal{E}[\![\texttt{(begin}\ C_0[\text{E}]\ C_1[\text{E}])]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![\texttt{(begin}\ C_0[\text{I}]\ C_1[\text{I}])]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o \text{ or } \bot_A.\end{aligned}$$

Expanding `begin`'s definition gives

$$\mathcal{E}[\![\texttt{(begin}\ C_0[\text{E}]\ C_1[\text{E}])]\!]\rho\kappa\sigma o = \mathcal{E}[\![C_0[\text{E}]]\!]\rho(\lambda\epsilon.\,\mathcal{E}[\![C_1[\text{E}]]\!]\rho\kappa)\sigma o$$

and

$$\begin{aligned}&\mathcal{E}[\![\texttt{(begin}\ C_0[\text{I}]\ C_1[\text{I}])]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o\\ &\quad = \mathcal{E}[\![C_0[\text{I}]]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)(\lambda\epsilon.\,\mathcal{E}[\![C_1[\text{I}]]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa)\sigma o\\ &\quad = \mathcal{E}[\![C_0[\text{E}]]\!]\rho(\lambda\epsilon.\,\mathcal{E}[\![C_1[\text{I}]]\!](\mathit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa)\sigma o\end{aligned}$$

using the induction hypothesis for the last equality.

The proof is easy to complete if the evaluation of E errs or halts relative to ρ, σ, o; so assume otherwise. By Axiom 3.2, there exists ϵ_1, σ', and o' such that

$$\forall \kappa, \mathcal{E}[\![C_0[\mathrm{E}]]\!]\rho\kappa\sigma o = \kappa\epsilon_1\sigma' o'$$

and $\sigma \preceq_\rho \sigma'$. Hence

$$\mathcal{E}[\![\texttt{(begin } C_0[\mathrm{E}]\ C_1[\mathrm{E}]\texttt{)}]\!]\rho\kappa\sigma o = \mathcal{E}[\![C_1[\mathrm{E}]]\!]\rho\kappa\sigma' o' \text{ or } \bot_A$$

and

$$\begin{aligned}&\mathcal{E}[\![\texttt{(begin } C_0[\mathrm{I}]\ C_1[\mathrm{I}]\texttt{)}]\!](\mathit{extends}\ \rho\langle\mathrm{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o\\ &\quad = \mathcal{E}[\![C_1[\mathrm{I}]]\!](\mathit{extends}\ \rho\langle\mathrm{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma' o' \text{ or } \bot_A.\end{aligned}$$

The proof is completed by using the induction hypothesis to show

$$\mathcal{E}[\![C_1[\mathrm{E}]]\!]\rho\kappa\sigma' o' = \mathcal{E}[\![C_1[\mathrm{I}]]\!](\mathit{extends}\ \rho\langle\mathrm{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma' o' \text{ or } \bot_A.$$

∎

β-substitution of side-effect free expressions

Theorem 4.21 *When* $\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n$ *are side-effect free and* $C[\mathrm{I}_i]$ *is almost side-effect free,*

$$\begin{aligned}&\mathcal{E}[\![\texttt{((lambda } (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i]\texttt{)}\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n\texttt{)}]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![\texttt{((lambda } (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{E}_i]\texttt{)}\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n\texttt{)}]\!]\rho\kappa\sigma o.\end{aligned}$$

Note the RHS must be α-converted so I_i cannot be free in E_i. The theorem is proved by appealing to Lemma 4.22 and Lemma 4.23 which follow.

Lemma 4.22 *When* $\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n$ *are side-effect free,*

$$\begin{aligned}&\mathcal{E}[\![\texttt{((lambda } (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i]\texttt{)}\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n\texttt{)}]\!]\rho\kappa\sigma\\ &\quad = \mathcal{E}[\![\texttt{((lambda } (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\\ &\qquad\qquad \texttt{((lambda } (\mathrm{I}_i)\ C[\mathrm{I}_i]\texttt{)}\ \mathrm{E}_i\texttt{))}\\ &\qquad\quad \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n\texttt{)}]\!]\rho\kappa\sigma.\end{aligned}$$

The proof is similar to that of Lemma 4.19 except that the store and oracle never change, i.e., $\sigma' = \sigma$ and $o' = o$.

Lemma 4.23 *When* E *is side-effect free and* C[I] *is almost side-effect free,*

$$\mathcal{E}[\![\texttt{((lambda}\ \text{(I)}\ C[\text{I}])\ \text{E)}]\!]\rho\kappa\sigma o = \mathcal{E}[\![C[\text{E}]]\!]\rho\kappa\sigma o.$$

Proof sketch: The proof is very similar to the proof of Lemma 4.20, except that the store and oracle remain the same. Consider the case of $C[\ ]$ being (`begin` $C_0[\ ]$ $C_1[\ ]$). Because C[E] is almost side-effect free, C_0[E] is side-effect free. Then, by Theorem 4.12, C_0[E] is evaluative, and thus there is some ϵ such that

$$\forall\kappa, \mathcal{E}[\![C_0[\text{E}]]\!]\rho\kappa\sigma o = \kappa\epsilon\sigma o \text{ or } \bot_A.$$

Hence

$$\mathcal{E}[\![\texttt{(begin}\ C_0[\text{E}]\ C_1[\text{E}])]\!]\rho\kappa\sigma o = \mathcal{E}[\![C_1[\text{E}]]\!]\rho\kappa\sigma o \text{ or } \bot_A$$

and

$$\begin{aligned}&\mathcal{E}[\![\texttt{(begin}\ C_0[\text{I}]\ C_1[\text{I}])]\!](\textit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o\\ &\quad = \mathcal{E}[\![C_1[\text{I}]]\!](\textit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o \text{ or } \bot_A.\end{aligned}$$

The proof is completed by using the induction hypothesis to show

$$\mathcal{E}[\![C_1[\text{E}]]\!]\rho\kappa\sigma o = \mathcal{E}[\![C_1[\text{I}]]\!](\textit{extends}\ \rho\langle\text{I}\rangle\langle\epsilon_0\rangle)\kappa\sigma o \text{ or } \bot_A.$$

∎

4.3.4 `letrec` Lifting

Theorem 4.24 *When* $C[\ ]$ *has one hole,*

$$\mathcal{E}[\![C[\texttt{(letrec}\ \text{(B) E)}]]\!]\rho\kappa\sigma o = \mathcal{E}[\![\texttt{(letrec}\ \text{(B)}\ C[\text{E}])]\!]\rho\kappa\sigma o.$$

Note the RHS must be α-converted so there are no variables bound in $C[\ ]$ which are `letrec` bound in B.

Proof sketch: Shown is the case in which $C[\ ]$ = (`lambda` (I^*) $C_0[\ ]$).

$$\begin{aligned}&\mathcal{E}[\![C[\texttt{(letrec}\ \text{(B) E)}]]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![\texttt{(lambda}\ (\text{I}^*)\ C_0[\texttt{(letrec}\ \text{(B) E)}])]\!]\rho\kappa\sigma o\\ &\quad = \kappa(\mathcal{L}[\![\texttt{(lambda}\ (\text{I}^*)\ C_0[\texttt{(letrec}\ \text{(B) E)}])]\!]\rho)\sigma o\\ &\quad = \kappa(\mathcal{L}[\![\texttt{(lambda}\ (\text{I}^*)\ \texttt{(letrec}\ \text{(B)}\ C_0[\text{E}]))]\!]\rho)\sigma o\end{aligned}$$

by the induction hypothesis.

$$\begin{aligned}&\mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) (letrec (}\mathrm{B}\texttt{) }C_0[\mathrm{E}]\texttt{))}]\!]\rho\\ &\quad= (\lambda\epsilon^*\kappa.\,\#\epsilon^* = \#\mathrm{I}^* \rightarrow \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\texttt{) }C_0[\mathrm{E}]\texttt{)}]\!]\rho'\kappa, \lambda\sigma o.\,\bot_A) \text{ in } E\\ &\quad= (\lambda\epsilon^*\kappa.\,\#\epsilon^* = \#\mathrm{I}^* \rightarrow \mathcal{E}[\![C_0[\mathrm{E}]]\!]\rho''\kappa, \lambda\sigma o.\,\bot_A) \text{ in } E\end{aligned}$$

where $\rho' = \mathit{extends}\,\rho\mathrm{I}^*\epsilon^*$ and $\rho'' = \mathit{extends}\,\rho'(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho'))$.

Since there are no variables bound in $C[\,]$ which are `letrec` bound in B, none of I^* are `letrec` bound in B and so

$$\rho'' = \mathit{extends}\,\rho'(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho))$$

Furthermore, $\rho'' = \mathit{extends}\,\rho'''\mathrm{I}^*\epsilon^*$, where

$$\rho''' = \mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho)).$$

Therefore,

$$\begin{aligned}&\mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) (letrec (}\mathrm{B}\texttt{) }C_0[\mathrm{E}]\texttt{))}]\!]\rho\\ &\quad= \mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }C_0[\mathrm{E}]\texttt{)}]\!]\rho'''\end{aligned}$$

and

$$\begin{aligned}&\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }C_0[\mathrm{E}]\texttt{)}]\!]\rho'''\kappa\sigma o\\ &\quad= \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\texttt{) (lambda (}\mathrm{I}^*\texttt{) }C_0[\mathrm{E}]\texttt{))}]\!]\rho\kappa\sigma o.\end{aligned}$$

∎

4.3.5 `letrec` Expression Merging

Theorem 4.25

$$\begin{aligned}&\mathcal{E}[\![\texttt{(letrec (}\mathrm{B}_0\texttt{) (letrec (}\mathrm{B}_1\texttt{) }\mathrm{E}\texttt{))}]\!]\rho\kappa\sigma o\\ &\quad= \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}_0\ \mathrm{B}_1\texttt{) }\mathrm{E}\texttt{)}]\!]\rho\kappa\sigma o.\end{aligned}$$

Note the LHS must be α-converted so no binding in B_0 can reference a variable `letrec` bound in B_1.

Proof: Let $f_0 = \mathcal{B}[\![\mathrm{B}_0]\!](\mathcal{I}[\![\mathrm{B}_0]\!])\rho$,
$\rho' = \mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0]\!])(\mathit{fix}\,f_0)$,
$f_1 = \mathcal{B}[\![\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_1]\!])\rho'$.

$$\begin{aligned}&\mathcal{E}[\![\texttt{(letrec (}\mathrm{B}_0\texttt{) (letrec (}\mathrm{B}_1\texttt{) }\mathrm{E}\texttt{))}]\!]\rho\kappa\sigma o\\ &\quad= \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}_1\texttt{) }\mathrm{E}\texttt{)}]\!]\rho'\kappa\sigma o\\ &\quad= \mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\,\rho'(\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{fix}\,f_1))\kappa\sigma o\end{aligned}$$

Because expressions are α-converted,

$$\begin{aligned}
&\mathit{extends}\,\rho'(\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{fix}\,f_1)\\
&\quad= \mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0\mathrm{B}_1]\!])(\mathit{fix}\,f_0 \mathbin{\S} \mathit{fix}\,f_1).
\end{aligned}$$

Let $f_{01} = \mathcal{B}[\![\mathrm{B}_0\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_0\mathrm{B}_1]\!])\rho$.

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(letrec}\ (\mathrm{B}_0\ \mathrm{B}_1)\ \mathrm{E})]\!]\rho\kappa\sigma o\\
&\quad= \mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0\mathrm{B}_1]\!])(\mathit{fix}\,f_{01}))\kappa\sigma o
\end{aligned}$$

The proof is completed by showing $\mathit{fix}\,f_{01} = \mathit{fix}\,f_0 \mathbin{\S} \mathit{fix}\,f_1$.

$$\begin{aligned}
f_{01} &= \lambda\epsilon^*.\,\mathcal{B}[\![\mathrm{B}_0]\!](\mathcal{I}[\![\mathrm{B}_0]\!])\\
&\qquad\qquad(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0)\\
&\qquad\ \mathbin{\S}\,\mathcal{B}[\![\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_1]\!])\\
&\qquad\qquad\quad(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0]\!])(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad\quad(\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0)\\
&= \lambda\epsilon^*.\,\mathcal{B}[\![\mathrm{B}_0]\!](\mathcal{I}[\![\mathrm{B}_0]\!])\rho(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0)\\
&\qquad\ \mathbin{\S}\,\mathcal{B}[\![\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_1]\!])\\
&\qquad\qquad\quad(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0]\!])(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad\quad(\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0)\\
&= \lambda\epsilon^*.\,f_0(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0)\\
&\qquad\ \mathbin{\S}\,\mathcal{B}[\![\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}_1]\!])\\
&\qquad\qquad\quad(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}_0]\!])(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad\quad(\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0)
\end{aligned}$$

because no binding in B_0 references a variable bound by B_1.

Let $g = \lambda\epsilon^*.\,f_0(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0) \mathbin{\S} \mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0$. Superscripts will denote function iteration: $f^0 = \lambda\epsilon^*.\,\epsilon^*$ and $f^{n+1} = f \circ f^n$. Observe that $f_{01}^n(\mathit{fix}\,g) = \mathit{fix}\,f_0 \mathbin{\S} f_1^n\bot$, therefore, $\bigsqcup\{f_{01}^n(\mathit{fix}\,g)\} = \mathit{fix}\,f_0 \mathbin{\S} \mathit{fix}\,f_1$.

$\mathit{fix}\,f_0 \mathbin{\S} \mathit{fix}\,f_1$ is a fixed point of f_{01} because

$$\begin{aligned}
&f_{01}(\mathit{fix}\,f_0 \mathbin{\S} \mathit{fix}\,f_1)\\
&\quad= f_{01}(\textstyle\bigsqcup\{f_{01}^n(\mathit{fix}\,g)\})\\
&\quad= \textstyle\bigsqcup\{f_{01}^{n+1}(\mathit{fix}\,g)\} && \text{by continuity}\\
&\quad= \textstyle\bigsqcup\{f_{01}^n(\mathit{fix}\,g)\} && \text{as } \mathit{fix}\,g \sqsubseteq f_{01}(\mathit{fix}\,g)\\
&\quad= \mathit{fix}\,f_0 \mathbin{\S} \mathit{fix}\,f_1.
\end{aligned}$$

$\mathit{fix}\ f_0 \S \mathit{fix}\ f_1$ is the least fixed point of f_{01} because, by construction, $g^m\bot \sqsubseteq f_{01}^m\bot$ so $f_{01}^n(g^m\bot) \sqsubseteq f_{01}^{m+n}\bot$.

$$\begin{aligned} &f_{01}^n(\mathit{fix}\ g) = f_{01}^n(\bigsqcup\{g^m\bot\}) = \bigsqcup\{f_{01}^n(g^m\bot)\} \\ &\quad \sqsubseteq \bigsqcup\{f_{01}^n(f_{01}^m\bot)\} = f_{01}^n(\mathit{fix}\ f_{01}) = \mathit{fix}\ f_{01} \end{aligned}$$

Therefore $f_{01}^n(\mathit{fix}\ g) \sqsubseteq \mathit{fix}\ f_{01}$ and $\mathit{fix}\ f_{01} = \mathit{fix}\ f_0 \S \mathit{fix}\ f_1$. ∎

4.3.6 `letrec` Simplification

Theorem 4.26 *When* I *is referenced nowhere except in* E*,*

$$\begin{aligned} &\mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\ \texttt{(}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{))) }\mathrm{E}_0\texttt{)}]\!]\rho\kappa\sigma o \\ &\quad = \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\texttt{) }\mathrm{E}_0\texttt{)}]\!]\rho\kappa\sigma o. \end{aligned}$$

Proof: By Theorem 4.25,

$$\begin{aligned} &\mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\ \texttt{(}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{))) }\mathrm{E}_0\texttt{)}]\!]\rho\kappa\sigma o \\ &\quad = \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\texttt{) (letrec ((}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{))) }\mathrm{E}_0\texttt{))}]\!]\rho\kappa\sigma o. \end{aligned}$$

Then

$$\begin{aligned} &\mathcal{E}[\![\texttt{(letrec ((}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{))) }\mathrm{E}_0\texttt{)}]\!]\rho\kappa\sigma o \\ &\quad = \mathcal{E}[\![\mathrm{E}_0]\!](\mathit{extends}\ \rho\langle\mathrm{I}\rangle(\mathit{fix}\ \mathcal{B}[\![\texttt{(}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{))}]\!]\langle\mathrm{I}\rangle\rho))\kappa\sigma o \\ &\quad = \mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma o \end{aligned}$$

because I is not free in E_0. ∎

4.3.7 `letrec` Binding Merging

Theorem 4.27

$$\begin{aligned} &\mathcal{E}[\![\texttt{(letrec ((}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) (letrec (}\mathrm{B}_0\texttt{) }\mathrm{E}\texttt{))) }\mathrm{B}_1\texttt{) }\mathrm{E}_0\texttt{)}]\!]\rho\kappa\sigma o \\ &\quad = \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}_0\ \texttt{(}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{)) }\mathrm{B}_1\texttt{) }\mathrm{E}_0\texttt{)}]\!]\rho\kappa\sigma o. \end{aligned}$$

Note the LHS must be α-converted so no binding in B_1 can reference a variable `letrec` bound in B_0.

Proof: Define the bar operator on bindings as follows.

$$\overline{\texttt{(}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{E}\texttt{)) }\mathrm{B}} = \texttt{(}\mathrm{I}\ \texttt{(lambda (}\mathrm{I}^*\texttt{) (letrec (}\mathrm{B}_0\texttt{) }\mathrm{E}\texttt{))) }\overline{\mathrm{B}}$$

In words, it adds a `letrec` of B_0 into all the `lambda` expressions being bound. As was shown in Theorem 4.26,

$$\mathcal{E}[\![\texttt{(letrec (}\overline{\mathrm{B}_0}\texttt{) }\mathrm{E}_0\texttt{)}]\!] = \mathcal{E}[\![\mathrm{E}_0]\!],$$

therefore by use of Theorem 4.24,

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(letrec ((I (lambda (I}^*\texttt{) (letrec (B}_0\texttt{) E))) B}_1\texttt{) E}_0\texttt{)}]\!] \\
&\quad = \mathcal{E}[\![\texttt{(letrec (}\overline{\mathrm{B}_0}\texttt{ (I (lambda (I}^*\texttt{) (letrec (B}_0\texttt{) E))) B}_1\texttt{) E}_0\texttt{)}]\!] \\
&\quad = \mathcal{E}[\![\texttt{(letrec (}\overline{\mathrm{B}_0\texttt{ (I (lambda (I}^*\texttt{) E)) B}_1}\texttt{) E}_0\texttt{)}]\!].
\end{aligned}$$

Let

$$\begin{aligned}
\mathrm{B} &= \mathrm{B}_0\ \texttt{(I (lambda (I}^*\texttt{) E)) }\mathrm{B}_1, \\
f &= \mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho, \\
g &= \mathcal{B}[\![\overline{\mathrm{B}}]\!](\mathcal{I}[\![\overline{\mathrm{B}}]\!])\rho.
\end{aligned}$$

The proof is completed by showing $\mathit{fix}\ f = \mathit{fix}\ g$.

$$\begin{aligned}
f &= \mathcal{B}[\![\mathrm{B}_0\ \texttt{(I (lambda (I}^*\texttt{) E)) }\mathrm{B}_1]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho \\
&= \lambda\epsilon^*.\,\langle\ldots\mathcal{L}[\![\texttt{(lambda (I}^*\texttt{) E)}]\!](\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*)\ldots\rangle
\end{aligned}$$

$$\begin{aligned}
g &= \mathcal{B}[\![\overline{\mathrm{B}_0}\ \texttt{(I (lambda (I}^*\texttt{) (letrec (B}_0\texttt{) E))) }\overline{\mathrm{B}_1}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho \\
&= \lambda\epsilon^*.\,\langle\ldots\mathcal{L}[\![\texttt{(lambda (I}^*\texttt{) (letrec (B}_0\texttt{) E))}]\!] \\
&\qquad\qquad (\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*) \\
&\qquad\ \ldots\rangle \\
&= \lambda\epsilon^*.\,\langle\ldots\mathcal{L}[\![\texttt{(lambda (I}^*\texttt{) E)}]\!] \\
&\qquad\qquad (\mathit{extends}\ (\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*) \\
&\qquad\qquad\qquad (\mathcal{I}[\![\mathrm{B}_0]\!]) \\
&\qquad\qquad\qquad (\mathit{fix}(\mathcal{B}[\![\mathrm{B}_0]\!](\mathcal{I}[\![\mathrm{B}]\!]_0)(\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*)))) \\
&\qquad\ \ldots\rangle
\end{aligned}$$

Define g_n as follows so that $g = \bigsqcup\{g_n\}$.

$$\begin{aligned}
h_n &= \lambda\rho.\,((\mathcal{B}[\![\overline{\mathrm{B}_0}]\!](\mathcal{I}[\![\overline{\mathrm{B}_0}]\!])\rho)^n\bot) \\
g_n &= \lambda\epsilon^*.\,\langle\ldots\mathcal{L}[\![\texttt{(lambda (I}^*\texttt{) E)}]\!] \\
&\qquad\qquad (\mathit{extends}\ (\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*) \\
&\qquad\qquad\qquad (\mathcal{I}[\![\mathrm{B}_0]\!]) \\
&\qquad\qquad\qquad (h_n(\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*))) \\
&\qquad\ \ldots\rangle
\end{aligned}$$

$\mathit{fix}\, f \sqsubseteq \mathit{fix}\, g$ is proved by showing $f^{n+1}\bot = g_n(f^n\bot)$ which implies $f^{n+1}\bot \sqsubseteq g_n^{n+1}\bot$. The definitions of f and g_n suggest that the environments used to evaluate the `lambda` expressions will be compared. Using Lemma 4.14, and the fact that h_n does not reference any of the variables in $\mathcal{I}[\![\mathrm{B}_0]\!]$ allows a simplification of g_n's environment.

$$\begin{aligned}
&\mathit{extends}\,(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*)\\
&\qquad(\mathcal{I}[\![\mathrm{B}_0]\!])\\
&\qquad(h_n(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*))\\
&\quad= \mathit{extends}\,\rho\\
&\qquad\qquad(\mathcal{I}[\![\mathrm{B}]\!])\\
&\qquad\qquad(h_n(\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])\epsilon^*)\,\S\,\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0)\\
&\quad= \mathit{extends}\,\rho\\
&\qquad\qquad(\mathcal{I}[\![\mathrm{B}]\!])\\
&\qquad\qquad(h_n(\mathit{extends}\,\rho(\langle\mathrm{I}\rangle\,\S\,\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0))\\
&\qquad\qquad\quad\S\,\mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0)
\end{aligned}$$

As a result, showing $f^{n+1}\bot = g_n(f^n\bot)$ is the same as showing

$$\begin{aligned}
f^n\bot = {}& h_n(\mathit{extends}\,\rho(\langle\mathrm{I}\rangle\,\S\,\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}(f^n\bot)\#\mathrm{B}_0))\\
&\S\,\mathit{dropfirst}(f^n\bot)\#\mathrm{B}_0,
\end{aligned}$$

which is proved by induction on n.

$\mathit{fix}\, g \sqsubseteq \mathit{fix}\, f$ is proved by showing $g_m^n\bot \sqsubseteq f^{n+m}\bot$. Following the same reasoning as before, the proof reduces to showing

$$\begin{aligned}
f^{n+m}\bot \sqsupseteq {}& h_m(\mathit{extends}\,\rho(\langle\mathrm{I}\rangle\,\S\,\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}(f^{n+m}\bot)\#\mathrm{B}_0))\\
&\S\,\mathit{dropfirst}(f^{n+m}\bot)\#\mathrm{B}_0.
\end{aligned}$$

Induction on m completes the proof because

$$\begin{aligned}
&f(h_m(\mathit{extends}\,\rho(\langle\mathrm{I}\rangle\,\S\,\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}(f^{n+m}\bot)\#\mathrm{B}_0))\\
&\qquad\S\,\mathit{dropfirst}(f^{n+m}\bot)\#\mathrm{B}_0)\\
&\quad\sqsupseteq h_{m+1}(\mathit{extends}\,\rho(\langle\mathrm{I}\rangle\,\S\,\mathcal{I}[\![\mathrm{B}_1]\!])(\mathit{dropfirst}(f^{n+m+1}\bot)\#\mathrm{B}_0))\\
&\qquad\quad\S\,\mathit{dropfirst}(f^{n+m+1}\bot)\#\mathrm{B}_0.
\end{aligned}$$

∎

4.3.8 Combination in a Combination Rotation

Theorem 4.28 *When* E_0 *is invariable,*

$$\begin{aligned}&\mathcal{E}[\![(\mathrm{E}_0\ ((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_1)\ \mathrm{E}^*))]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ (\mathrm{E}_0\ \mathrm{E}_1))\ \mathrm{E}^*)]\!]\rho\kappa\sigma o.\end{aligned}$$

Note the LHS must be α-converted so none of I^* can be free in E_0.

Proof: The proof is easy to complete if the evaluation of some member of E^* errs or halts relative to ρ, σ, o; so assume otherwise. By using Axiom 3.2 in succession with each member of E^*, there exists ϵ^*, σ', and o' such that

$$\forall\psi, \mathcal{E}^*[\![\mathrm{E}^*]\!]\rho\psi\sigma o = \psi\epsilon^*\sigma' o'$$

and $\sigma \preceq_\rho \sigma'$, Let $\kappa' = \lambda\epsilon.\, \mathcal{E}[\![\mathrm{E}_0]\!]\rho(\lambda\epsilon'.\, \mathit{applicate}\, \epsilon'\langle\epsilon\rangle\kappa)$ and $\rho' = \mathit{extends}\, \rho\mathrm{I}^*\epsilon^*$. Then

$$\begin{aligned}&\mathcal{E}[\![(\mathrm{E}_0\ ((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_1)\ \mathrm{E}^*))]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_1)\ \mathrm{E}^*)]\!]\rho\kappa'\sigma o\\ &\quad = \mathcal{E}^*[\![\mathrm{E}^*]\!]\rho(\lambda\epsilon^*.\, \mathit{applicate}(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_1)]\!]\rho)\epsilon^*\kappa')\sigma o\\ &\quad = \mathit{applicate}(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_1)]\!]\rho)\epsilon^*\kappa'\sigma' o' \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![\mathrm{E}_1]\!]\rho'\kappa'\sigma' o' \text{ or } \bot_A\end{aligned}$$

and

$$\begin{aligned}&\mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ (\mathrm{E}_0\ \mathrm{E}_1))\ \mathrm{E}^*)]\!]\rho\kappa\sigma o\\ &\quad = \mathcal{E}^*[\![\mathrm{E}^*]\!]\rho(\lambda\epsilon^*.\, \mathit{applicate}(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ (\mathrm{E}_0\ \mathrm{E}_1))]\!]\rho)\epsilon^*\kappa)\sigma o\\ &\quad = (\lambda\epsilon^*.\, \mathit{applicate}(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ (\mathrm{E}_0\ \mathrm{E}_1))]\!]\rho)\epsilon^*\kappa)\epsilon^*\sigma' o' \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}_1)]\!]\rho'\kappa\sigma' o' \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![\mathrm{E}_1]\!]\rho'(\lambda\epsilon.\, \mathcal{E}[\![\mathrm{E}_0]\!]\rho'(\lambda\epsilon'.\, \mathit{applicate}\, \epsilon'\langle\epsilon\rangle\kappa))\sigma' o' \text{ or } \bot_A\\ &\quad = \mathcal{E}[\![\mathrm{E}_1]\!]\rho'\kappa'\sigma' o' \text{ or } \bot_A\end{aligned}$$

because none of I^* are free in E_0. ∎

4.3.9 Defined Constant Substitution

Theorem 4.29 *The defined constant substitution rule shown in Figure 4.1 is meaning-refining.*

Proof: Let $P[\ ]$ be a P-context, I_i be an immutable variable in $P[\ ]$, E_i be a constant or an immutable variable in $P[\ ]$, and $P[\mathrm{E}_i]$ be the result of applying the defined constant substitution rule to $P[\mathrm{I}_i]$. Since I_i is immutable, it can be modified only during its initialization. Therefore, given ρ and σ at any point in the execution of the program,

$$\rho\mathrm{I}_i \in E \rightarrow \rho\mathrm{I}_i \mid E, \sigma(\rho\mathrm{I}_i \mid L) \downarrow 1$$

is either $(\mathit{undefined}\,\mathrm{in}\,E)$ or some other value ϵ_i.

Assume E_i is a constant. Then $\epsilon_i = \mathcal{K}[\![\mathrm{E}_i]\!]$, and so $\mathcal{E}[\![\mathrm{I}_i]\!]\rho\kappa\sigma o = \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma o$. Therefore, $\mathcal{P}[\![P[\mathrm{I}_i]]\!] \sqsubseteq \mathcal{P}[\![P[\mathrm{E}_i]]\!]$ by Lemma 4.9.

Now assume E_i is an immutable variable. If E_i is undefined at the time of I_i's initialization, then $\mathcal{D}[\![P[\mathrm{I}_i]]\!]\rho_0\kappa_0\sigma_0 o = \bot_A$, and so $\mathcal{P}[\![P[\mathrm{I}_i]]\!] \sqsubseteq \mathcal{P}[\![P[\mathrm{E}_i]]\!]$ since $\mathit{compute\_thread\_answer}\ o\iota$ does not depend on $P[\mathrm{E}_0]$ or $P[\mathrm{E}_1]$ for all o and all $\iota \neq \mathit{root\_tid}$. Otherwise, $\mathcal{E}[\![\mathrm{I}_i]\!]\rho\kappa\sigma o = \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma o$, so again $\mathcal{P}[\![P[\mathrm{I}_i]]\!] \sqsubseteq \mathcal{P}[\![P[\mathrm{E}_i]]\!]$ by Lemma 4.9. ∎

Chapter 5

Stack Assembly Language

Stack Assembly Language (SAL) is a tree structured assembly language for a stack machine. It is designed to be the target of a translation from Simple PreScheme, so many properties of Simple PreScheme are reflected in the language. The language is introduced by giving an informal description of an abstract machine which might execute the language. A thread-unaware formal denotational semantics and a thread-unaware operational semantics are given after the introduction.

The stack machine's state is captured by six terms: a code sequence, a value, a stack, a continuation, a store, and an environment. The stack is a sequence of values as is the store. A continuation stores a return point for a non-tail-recursive call or an indication that the computation should halt. Nonhalt continuations contain a code sequence, a stack, and another continuation.

Most machine instructions fall into two categories. Value producing instructions create a new term for the value portion of the state. For instance, the instruction which loads a constant is value producing. Value consuming instructions use the value portion of the state. For instance, the conditional branch instructions use the value portion of the state to decide the direction of the branch. The abstract syntax identifies instructions as value producing or value consuming. A mnemonic description for each value producing and consuming instruction is given in Figure 5.1.

There are three types of variables in Simple PreScheme programs: defined variables, `letrec`-bound variables, and `lambda`-bound variables, but only the values of `lambda`-bound variables are placed on the stack. A reference to a `lambda`-bound variable is translated into a stack reference by giving an offset

Description	Opcode	Operands
Value Producing Instructions		
Load literal	`lit`	immediate, code
Load unspecified value	`unspec`	code
Copy stack entry to value	`copy`	source, code
Fetch address	`fetch`	address, code
Load global	`load`	address, code
Unary operator O	O	src/imm, code
Binary operator O	O	src/imm, source, code
Make continuation	`mkcont`	stack size, code, code
Entry stack size declaration	`entry`	stack height, code
Reserve stack space	`reserve`	stack height, code
Comment	`cmt`	immediate list, code
Value Consuming Instructions		
Ignore value register	`ignore`	code
Move value to stack	`move`	destination, code
Store global	`store`	address, code
Branch always	`bra`	
Branch if false	`bif`	code, code
Branch when false	`bwf`	code, code, code
Select	`select`	code list
Pick	`pick`	code list, code
Call	`call`	src/imm list
Return	`return`	
Dispose of stack space	`dispose`	stack height, code
Wrong	`wrong`	code
Comment	`cmt`	immediate list, code

Figure 5.1: SAL Instruction Summary

from the stack bottom.

A tail-recursive call is translated into an instruction which simply shuffles the values on the stack, and proceeds using the code in the value portion of the state. A non-tail-recursive call, i.e., a call which must return to the location at which it was invoked, must save the return information as a continuation. The `mkcont` instruction saves the code sequence to be executed after the call, the stack, and the current continuation as a continuation. When a `return` occurs, the computed result is the value, and the current code sequence, stack, and continuation are replaced by the information saved as a continuation.

Conditionals are handled in an unusual fashion. During the execution of a code sequence, most instructions in the sequence are executed after exactly one other instruction. A join point is an instruction that can be executed after more than one instruction. The compilation of conditionals may lead to join points. For example, in the expression

```
(begin
  (if (null-port? *port*)
      (set! *flag* 0)
      (set! *flag* 4))
  (do-something))
```

the procedure call is a join point because the flow of control through both directions of the conditional lead to the call.

Stack Assembly Language has two conditional branch instructions. The `bif` instruction is used when there is no join point and the `bwf` instruction when there is one.

5.1 Abstract Syntax

$\mathrm{N} \in \mathrm{Num}$	natural numbers ($\mathrm{Num} = N$)
$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{I} \in \mathrm{Ide}$	identifiers
$\mathrm{R} \in \mathrm{Ref}$	local references (positive integers)
$\mathrm{J} \in \mathrm{Src}$	local references or quoted constants
$\mathrm{F} \in \mathrm{Form}$	value formation instructions
$\mathrm{G} \in \mathrm{User}$	value consuming instructions
$\mathrm{B}' \in \mathrm{Bnd}'$	bindings
$\mathrm{E}' \in \mathrm{Exp}'$	top level expressions
$\mathrm{P}' \in \mathrm{Pgm}'$	programs
$\mathrm{Q} \in \mathrm{Code}$	all instructions

$$\begin{aligned}
\mathrm{Pgm}' &\longrightarrow (\texttt{define}\ \mathrm{I})^*\ \mathrm{E}'\\
\mathrm{Exp}' &\longrightarrow (\texttt{letrec}\ (\mathrm{B}')\ \mathrm{F})\\
\mathrm{Bnd}' &\longrightarrow (\mathrm{I}\ \mathrm{F})^*\\
\mathrm{Ref} &\longrightarrow \mathrm{N}\\
\mathrm{Src} &\longrightarrow \mathrm{R} \mid {}'\mathrm{K}\\
\mathrm{Form} &\longrightarrow (\texttt{lit}\ {}'\mathrm{K})\ \mathrm{G} \mid (\texttt{unspec})\ \mathrm{G}\\
&\quad \mid (\texttt{copy}\ \mathrm{R})\ \mathrm{G} \mid (\texttt{fetch}\ \mathrm{I})\ \mathrm{G} \mid (\texttt{load}\ \mathrm{I})\ \mathrm{G}\\
&\quad \mid (\mathrm{O}_1\ \mathrm{J})\ \mathrm{G} \mid (\mathrm{O}_2\ \mathrm{J}\ \mathrm{R})\ \mathrm{G} \mid (\mathrm{O}_2\ \mathrm{R}\ \mathrm{J})\ \mathrm{G}\\
&\quad \mid (\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G} \mid (\texttt{entry}\ \mathrm{N})\ \mathrm{F} \mid (\texttt{reserve}\ \mathrm{N})\ \mathrm{F}\\
&\quad \mid (\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{F}\\
\mathrm{User} &\longrightarrow (\texttt{ignore})\ \mathrm{F} \mid (\texttt{move}\ \mathrm{R})\ \mathrm{G} \mid (\texttt{store}\ \mathrm{I})\ \mathrm{G}\\
&\quad \mid (\texttt{bra}) \mid (\texttt{bif}\ (\mathrm{F})\ (\mathrm{F})) \mid (\texttt{bwf}\ (\mathrm{F})\ (\mathrm{F}))\ \mathrm{G}\\
&\quad \mid (\texttt{select}\ (\mathrm{F})^*) \mid (\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G} \mid (\texttt{call}\ \mathrm{J}^*)\\
&\quad \mid (\texttt{return}) \mid (\texttt{dispose}\ \mathrm{N})\ \mathrm{F} \mid (\texttt{wrong})\ \mathrm{G}\\
&\quad \mid (\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{G}\\
\mathrm{Code} &\longrightarrow \mathrm{P}' \mid \mathrm{F} \mid \mathrm{G}
\end{aligned}$$

The SAL syntactic category associated with a nonterminal is the set of derivation trees specified for the nonterminal by the SAL BNF. For example, the SAL syntactic category Pgm′ is associated with nonterminal P′ and specified by the following production.

$$\mathrm{Pgm}' \longrightarrow (\texttt{define}\ \mathrm{I})^*\ \mathrm{E}'$$

The precise representation of the trees is not given, however, every tree associated with syntax of the form $[\![(\mathrm{F})^*]\!]$ must be a sequence of elements from the syntactic category Form. This fact is used when giving the meaning of the `select` and `pick` instructions.

5.2 SAL Denotational Semantics

Many definitions used in the MTPS semantics are used unchanged in the SAL semantics. Before studying this section closely, the reader is advised to read Section 5.3 on the operational semantics for SAL.

5.2.1 Additional Domain Equations

$\pi \in G$	$= E \to F$	expression instructions
$\omega \in W$	$= E + N$	instruction denoted values

5.2.2 Semantic Functions

$$
\begin{aligned}
\mathcal{W} &: \mathrm{Src} \to W\\
\mathcal{F} &: \mathrm{Form} \to U \to F\\
\mathcal{G} &: \mathrm{User} \to U \to G\\
\mathcal{I}' &: \mathrm{Bnd}' \to \mathrm{Ide}^*\\
\mathcal{B}' &: \mathrm{Bnd}' \to \mathrm{Ide}^* \to U \to E^* \to E^*\\
\mathcal{E}' &: \mathrm{Exp}' \to U \to F\\
\mathcal{D}' &: \mathrm{Pgm}' \to U \to K \to C\\
\mathcal{P}' &: \mathrm{Pgm}' \to O \to I \to A
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{W}[\![\mathrm{R}]\!] = \mathrm{R}\ \mathrm{in}\ W\\
&\mathcal{W}[\![{}'\mathrm{K}]\!] = \mathcal{K}[\![\mathrm{K}]\!]\ \mathrm{in}\ W
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{F}[\![(\texttt{lit}\ {}'\mathrm{K})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{literal}(\mathcal{K}[\![\mathrm{K}]\!])(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{unspec})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{literal}(\mathit{unspecified}\ \mathrm{in}\ E)(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{copy}\ \mathrm{R})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{copy}\, \mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{fetch}\ \mathrm{I})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{fetch}(\mathit{lookup}\, \rho\mathrm{I})(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{load}\ \mathrm{I})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{load}(\mathit{lookup}\, \rho\mathrm{I})(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(+\ \mathrm{J}_0\ \mathrm{J}_1)\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{add}(\mathcal{W}[\![\mathrm{J}_0]\!])(\mathcal{W}[\![\mathrm{J}_1]\!])(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{makecont}\, \mathrm{N}(\mathcal{F}[\![\mathrm{F}]\!]\rho)(\mathcal{G}[\![\mathrm{G}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{entry}\ \mathrm{N})\ \mathrm{F}]\!] = \lambda\rho.\, \mathit{entry}\, \mathrm{N}(\mathcal{F}[\![\mathrm{F}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}]\!] = \lambda\rho.\, \mathit{reserve}\, \mathrm{N}(\mathcal{F}[\![\mathrm{F}]\!]\rho)\\
&\mathcal{F}[\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{F}]\!] = \mathcal{F}[\![\mathrm{F}]\!]
\end{aligned}
$$

$\mathcal{G}[\![(\texttt{ignore})\ \mathrm{F}]\!] = \lambda\rho.\, \mathit{ignore}(\mathcal{F}[\![\mathrm{F}]\!]\rho)$
$\mathcal{G}[\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{move}\, \mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)$
$\mathcal{G}[\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!]$
$\quad = \lambda\rho.\, \mathit{ismutable}\, \mathrm{I} \to \mathit{setglobal}(\mathit{lookup}\, \rho \mathrm{I})(\mathcal{G}[\![\mathrm{G}]\!]\rho),$
$\qquad\qquad \mathit{initglobal}(\mathit{lookup}\, \rho \mathrm{I})(\mathcal{G}[\![\mathrm{G}]\!]\rho)$
$\mathcal{G}[\![(\texttt{bra})]\!] = \bot_G$
$\mathcal{G}[\![(\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!] = \lambda\rho.\, \mathit{jumpfalse}(\mathcal{F}[\![\mathrm{F}_0]\!]\rho)(\mathcal{F}[\![\mathrm{F}_1]\!]\rho)$
$\mathcal{G}[\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!]$
$\quad = \lambda\rho.\, \mathit{jumpfalse}(\mathcal{F}(\mathsf{join}\, \mathrm{F}_0\mathrm{G})\rho)(\mathcal{F}(\mathsf{join}\, \mathrm{F}_1\mathrm{G})\rho)$
$\mathcal{G}[\![(\texttt{select}\ (\mathrm{F})^*)]\!]$
$\quad = \lambda\rho.\, \mathit{select}(\mathit{map}(\lambda \mathrm{F}.\, \mathcal{F}[\![\mathrm{F}]\!]\rho)[\![(\mathrm{F})^*]\!])$
$\mathcal{G}[\![(\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G}]\!]$
$\quad = \lambda\rho.\, \mathit{select}(\mathit{map}(\lambda \mathrm{F}.\, \mathcal{F}(\mathsf{join}\, \mathrm{FG})\rho)[\![(\mathrm{F})^*]\!])$
$\mathcal{G}[\![(\texttt{call}\ \mathrm{J}^*)]\!] = \lambda\rho.\, \mathit{call}(\mathcal{W}^*[\![\mathrm{J}^*]\!])$
$\mathcal{G}[\![(\texttt{return})]\!] = \lambda\rho.\, \mathit{return}$
$\mathcal{G}[\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}]\!] = \lambda\rho.\, \mathit{dispose}\, \mathrm{N}(\mathcal{G}[\![\mathrm{G}]\!]\rho)$
$\mathcal{G}[\![(\texttt{wrong})\ \mathrm{G}]\!]$
$\quad = \lambda\rho.\, \lambda\epsilon\epsilon^*\kappa.\, \mathit{wrong}$ “assignment of an immutable variable”
$\mathcal{G}[\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{G}]\!] = \mathcal{G}[\![\mathrm{G}]\!]$

$\mathcal{I}'[\![\ ]\!] = \langle\rangle$
$\mathcal{I}'[\![(\mathrm{I}\ \mathrm{F})\ \mathrm{B}']\!] = \langle \mathrm{I}\rangle \S\, \mathcal{I}'[\![\mathrm{B}']\!]$

$\mathcal{B}'[\![\ ]\!] = \lambda \mathrm{I}^*\rho\epsilon^*.\, \langle\rangle$
$\mathcal{B}'[\![(\mathrm{I}\ \mathrm{F})\ \mathrm{B}']\!] =$
$\quad \lambda \mathrm{I}_0^*\rho\epsilon^*.\, \langle(\mathcal{F}[\![\mathrm{F}]\!](\mathit{extends}\, \rho \mathrm{I}_0^*\epsilon^*))\ \mathrm{in}\ E\rangle \S\, \mathcal{B}'[\![\mathrm{B}']\!]\mathrm{I}_0^*\rho\epsilon^*$

$\mathcal{E}'[\![(\texttt{letrec}\ (\mathrm{B}')\ \mathrm{F})]\!] =$
$\quad \lambda\rho.\, \mathcal{F}[\![\mathrm{F}]\!](\mathit{extends}\, \rho(\mathcal{I}[\![\mathrm{B}']\!])(\mathit{fix}(\mathcal{B}'[\![\mathrm{B}']\!](\mathcal{I}'[\![\mathrm{B}']\!])\rho)))$

$\mathcal{D}'[\![\mathrm{E}']\!] = \lambda\rho.\, \mathcal{E}'[\![\mathrm{E}']\!]\rho\langle\rangle$
$\mathcal{D}'[\![(\texttt{define}\ \mathrm{I})\ \mathrm{P}']\!] =$
$\quad \lambda\rho\kappa\sigma.\, \mathcal{D}'[\![\mathrm{P}']\!](\rho[(\mathit{new}\, \sigma)\ \mathrm{in}\ D/\mathrm{I}])\kappa(\mathit{update}(\mathit{new}\, \sigma)(\mathit{undefined}\, \mathrm{in}\, E)\sigma)$

$\mathcal{P}'[\![\mathrm{P}']\!] = \mathcal{D}'[\![\mathrm{P}']\!]\rho_0\kappa_0\sigma_0$

5.2.3 Machine Instruction Auxiliary Functions

$$\begin{aligned}
&\mathit{literal} : E \to G \to F\\
&\mathit{literal} = \lambda\epsilon\pi.\,\lambda\epsilon^*\kappa.\,\pi\epsilon\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathit{ignore} : F \to G\\
&\mathit{ignore} = \lambda\phi.\,\lambda\epsilon\epsilon^*\kappa.\,\phi\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathit{move} : N \to G \to G\\
&\mathit{move} =\\
&\quad \lambda\nu\pi.\,\lambda\epsilon\epsilon^*\kappa.\,\pi(\mathit{unspecified}\,\mathrm{in}\,E)\\
&\qquad (\mathit{takefirst}\,\epsilon^*(\nu-1)\;\S\;\langle\epsilon\rangle\;\S\;\mathit{dropfirst}\,\epsilon^*\nu)\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathit{copy} : N \to G \to F\\
&\mathit{copy} = \lambda\nu\pi.\,\lambda\epsilon^*\kappa.\,\pi(\mathit{stackref}\,\epsilon^*\nu)\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathit{fetch} : D \to G \to F\\
&\mathit{fetch} = \lambda\delta\pi.\,\lambda\epsilon^*\kappa.\,\delta \in E \to \pi(\delta \mid E)\epsilon^*\kappa,\\
&\qquad\qquad\qquad\qquad\quad \mathit{wrong}\text{ ``fetch given global''}
\end{aligned}$$

$$\begin{aligned}
&\mathit{load} : D \to G \to F\\
&\mathit{load} = \lambda\delta\pi.\,\lambda\epsilon^*\kappa.\,\delta \in L \to \mathit{hold}(\delta \mid L)\lambda\epsilon.\,\pi\epsilon\epsilon^*\kappa,\\
&\qquad\qquad\qquad\qquad\quad \mathit{wrong}\text{ ``load not given global''}
\end{aligned}$$

$$\begin{aligned}
&\mathit{initglobal} : D \to G \to G\\
&\mathit{initglobal} = \lambda\delta\pi.\,\lambda\epsilon\epsilon^*\kappa.\,\mathit{initialize}\,\delta\epsilon(\pi(\mathit{unspecified}\,\mathrm{in}\,E)\epsilon^*\kappa)
\end{aligned}$$

$$\begin{aligned}
&\mathit{setglobal} : D \to G \to G\\
&\mathit{setglobal} = \lambda\delta\pi.\,\lambda\epsilon\epsilon^*\kappa.\,\mathit{assign}\,\delta\epsilon(\pi(\mathit{unspecified}\,\mathrm{in}\,E)\epsilon^*\kappa)
\end{aligned}$$

$$\begin{aligned}
&\mathit{jumpfalse} : F \to F \to G\\
&\mathit{jumpfalse} = \lambda\phi_0\phi_1.\,\lambda\epsilon\epsilon^*\kappa.\,\epsilon = \mathit{false} \to \phi_1\epsilon^*\kappa, \phi_0\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathit{select} : F^* \to G\\
&\mathit{select} =\\
&\quad \lambda\phi^*.\,\lambda\epsilon\epsilon^*\kappa.\\
&\qquad (0 \le \epsilon \mid R) \wedge ((\epsilon \mid R) < \#\phi^* - 1) \to (\phi^* \downarrow (2 + \epsilon \mid R))\epsilon^*\kappa,\\
&\qquad\quad (\phi^* \downarrow 1)\epsilon^*\kappa
\end{aligned}$$

$$\mathit{call} : W^* \to G$$
$$\mathit{call} = \lambda\omega^*.\, \lambda\epsilon\epsilon^*\kappa.\, \mathit{obtain}\, \omega^*\epsilon^*\lambda\epsilon^*.\, \mathit{applicate}\, \epsilon\epsilon^*\kappa$$

$$\mathit{return} : G$$
$$\mathit{return} = \lambda\epsilon\epsilon^*\kappa.\, \kappa\epsilon$$

$$\mathit{makecont} : N \to F \to G \to F$$
$$\begin{aligned}\mathit{makecont} = \lambda\nu\phi\pi.\, \lambda\epsilon^*\kappa.\, \nu = \#\epsilon^* &\to \phi\epsilon^*\lambda\epsilon.\, \pi\epsilon\epsilon^*\kappa,\\ &\quad \mathit{wrong}\text{ “bad stack”}\end{aligned}$$

$$\mathit{entry} : N \to F \to F$$
$$\mathit{entry} = \lambda\nu\phi.\, \lambda\epsilon^*\kappa.\, \nu = \#\epsilon^* \to \phi\epsilon^*\kappa, \mathit{wrong}\text{ “bad stack”}$$

$$\mathit{reserve} : N \to F \to F$$
$$\mathit{reserve} = \lambda\nu\phi.\, \lambda\epsilon^*\kappa.\, \phi(\epsilon^* \mathbin{\S} \mathit{unspecs}\, \nu)\kappa$$

$$\mathit{dispose} : N \to G \to G$$
$$\mathit{dispose} = \lambda\nu\pi.\, \lambda\epsilon\epsilon^*\kappa.\, \pi\epsilon(\mathit{takefirst}\, \epsilon^*(\#\epsilon^* - \nu))\kappa$$

$$\mathit{add} : W \to W \to G \to F$$
$$\begin{aligned}&\mathit{add} =\\ &\quad \lambda\omega_0\omega_1\pi.\\ &\qquad \lambda\epsilon^*\kappa.\, \mathit{get}\, \omega_0\epsilon^*\lambda\epsilon_0.\, \mathit{get}\, \omega_1\epsilon^*\lambda\epsilon_1.\, \pi((\epsilon_0 \mid R + \epsilon_1 \mid R) \text{ in } E)\epsilon^*\kappa\end{aligned}$$

5.2.4 Additional Auxiliary Functions

$$\mathit{stackref} : E^* \to N \to E$$
$$\mathit{stackref} = \lambda\epsilon^*\nu.\, \epsilon^* \downarrow \nu$$

$$\mathit{get} : W \to E^* \to K \to C$$
$$\mathit{get} = \lambda\omega\epsilon^*\kappa.\, \mathit{send}(\omega \in E \to (\omega \mid E), \mathit{stackref}\, \epsilon^*(\omega \mid N))\kappa$$

$$\mathit{obtain} : W^* \to E^* \to (E^* \to C) \to C$$
$$\begin{aligned}&\mathit{obtain} =\\ &\quad \lambda\omega^*\epsilon^*\psi.\, \omega^* = \langle\rangle \to \psi\langle\rangle,\\ &\qquad\qquad \mathit{get}(\omega^* \downarrow 1)\epsilon^*\lambda\epsilon.\, \mathit{obtain}(\omega^* \dagger 1)\epsilon^*\lambda\epsilon^*.\, \psi(\langle\epsilon\rangle \mathbin{\S} \epsilon^*)\end{aligned}$$

$$\begin{aligned}&\mathit{unspecs} : N \to E^*\\&\mathit{unspecs} = \lambda\nu.\, \nu = 0 \to \langle\rangle, \langle(\mathit{unspecified}\, \mathrm{in}\, E)\rangle \mathbin{\S} \mathit{unspecs}(\nu - 1)\end{aligned}$$

$$\mathit{map} = \lambda\psi\nu^*.\, \nu^* = \langle\rangle \to \langle\rangle, \langle\psi(\nu^* \downarrow 1)\rangle \mathbin{\S} \mathit{map}\, \psi(\nu^* \dagger 1)$$

$$\begin{aligned}&\mathit{ismutable} : \mathrm{Ide} \to T\\&\mathit{ismutable} = \text{Definition follows.}\end{aligned}$$

$\mathit{ismutable}\,\mathrm{I} = \mathit{true}$ if and only if I is an identifier whose name begins and ends with an asterisk and is at least three characters long.

Definition 5.1

$$\begin{aligned}
&\mathsf{join} : \mathrm{Code} \to \mathrm{User} \to \mathrm{Code}\\
&\mathsf{join}[\![(\texttt{lit}\ {}'\mathrm{K})\ \mathrm{G}]\!] = [\![(\texttt{lit}\ {}'\mathrm{K})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{unspec})\ \mathrm{G}]\!] = [\![(\texttt{unspec})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{ignore})\ \mathrm{F}]\!] = [\![(\texttt{ignore})\ \mathrm{F}_0]\!] \text{ where } \mathrm{F}_0 = \mathsf{join}\, \mathrm{F}\\
&\mathsf{join}[\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!] = [\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{copy}\ \mathrm{R})\ \mathrm{G}]\!] = [\![(\texttt{copy}\ \mathrm{R})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{fetch}\ \mathrm{I})\ \mathrm{G}]\!] = [\![(\texttt{fetch}\ \mathrm{I})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{load}\ \mathrm{I})\ \mathrm{G}]\!] = [\![(\texttt{load}\ \mathrm{I})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!] = [\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{bra})]\!] = \lambda \mathrm{G}.\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!] = [\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G}]\!] = [\![(\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}]\!] = [\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}]\!] = [\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}_0]\!] \text{ where } \mathrm{F}_0 = \mathsf{join}\, \mathrm{F}\\
&\mathsf{join}[\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}]\!] = [\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{wrong})\ \mathrm{G}]\!] = [\![(\texttt{wrong})\ \mathrm{G}_0]\!] \text{ where } \mathrm{G}_0 = \mathsf{join}\, \mathrm{G}\\
&\mathsf{join}[\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{Q}]\!] = [\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{Q}_0]\!] \text{ where } \mathrm{Q}_0 = \mathsf{join}\, \mathrm{Q}
\end{aligned}$$

The function join concatenates a code sequence which ends with a branch always instruction with another sequence. The function is not defined for `bif`, `select`, `call`, and `return` because these instructions cannot occur on the path to a join point.

5.3 SAL Operational Semantics

In this section we present an operational semantics for SAL programs which do not contain I/O or interthread operators. That is, the operational semantics assumes "single-threaded" programs. This assumption greatly simplifies the arguments in the remaining part of the paper.

An operational semantics for multithreaded SAL programs can be easily defined from the operational semantics given here. A state of a multithreaded machine is just a function from thread IDs to states of single-threaded machines (see Section 5.3.3) plus an oracle. A transition of a multithreaded state consists of the application of a single-threaded transition to one of the constituent single-threaded states plus possibly the incrementing the index of the oracle. The single-threaded transition must correctly correspond to the entry in the oracle indicated by its index. Oracles are not actually needed to define a multithreaded operational semantics for SAL. However, they are needed for the proof of the full faithfulness theorem, that is, the multithreaded analogue of Theorem 5.7 on Page 95.

5.3.1 Values

The set of *values* manipulated by program P$'$, denoted by $V_{\mathrm{P}'}$, is defined as follows:

Ground types:

- Numbers: $n \in V_{\mathrm{P}'}$
- Booleans: `#t` $\in V_{\mathrm{P}'}$ and `#f` $\in V_{\mathrm{P}'}$
- Characters: `#\`$c \in V_{\mathrm{P}'}$
- Strings: `"`c^*`"` $\in V_{\mathrm{P}'}$
- Ports: $(\mathsf{port}, n) \in V_{\mathrm{P}'}$.
- Special constants: Unspecified and Undefined.

Pointer types: $(\star\,\tau, \ell) \in V_{\mathrm{P}'}$ where ℓ is a location and τ is a type.

Unshared Vector types: $(\bullet\,\tau, \ell^*) \in V_{\mathrm{P}'}$ where ℓ^* is a sequence of locations and τ is a type.

Shared Vector types: $(\bullet_s\,\tau, \ell^*) \in V_{\mathrm{P}'}$ where ℓ^* is a sequence of locations and τ is a type.

Procedure types: Each code sequence F which is bound to an identifier by the `letrec` expression in program P′, is a member of $V_{P'}$.

5.3.2 Continuations

A *continuation* is defined inductively by:

- The special constant Halt is a continuation.
- A triple (G, a, k) is a continuation if G is a value consuming instruction, a is a sequence of values, and k is a continuation.

5.3.3 States

A *state* of the machine has six components:

Q	the code
v	the value
a	the stack which is a sequence of values
k	the continuation
s	the store which maps locations to values
u	the environment which maps identifiers to either values or locations

The notation $[a \mapsto b]$ will be used for the term which denotes the finite map which only maps a to b, i.e., its domain is the set $\{a\}$ and $[a \mapsto b](a) = b$. The notation $[]$ denotes the empty map. The finite map g *augment* f, written $f \diamond g$, has a domain of $\mathrm{dom}(f) \cup \mathrm{dom}(g)$, and value

$$(f \diamond g)(x) = \begin{cases} g(x) & x \in \mathrm{dom}(g) \\ f(x) & \text{otherwise.} \end{cases}$$

The notation $\Sigma^S = (Q, v, a, k, s, u)$ will be used to denote a stack machine state, and $\Sigma_0^S \Longrightarrow \Sigma_1^S$ will denote one transition from Σ_0^S to Σ_1^S. Finally, $\Sigma_0^S \Longrightarrow^* \Sigma_1^S$ is written when zero or more transitions relate Σ_0^S to Σ_1^S.

5.3.4 Transition Rules

There is a small number of rules which determine the allowed state transitions. Following the conventions used in [6] and [4], for each transition rule a name is given, one or more conditions determining when the rule is applicable (and possibly introducing new locally bound variables for later use), and a specification of the new terms that make up the resulting state. The variables $\mathrm{Q}, v, a, k, s, u$ describe the starting state, and the variables $\mathrm{Q}', v', a', k', s', u'$ describe the state which results from the transition. Primed variables occur only as the left hand sides of equations specifying new terms. Primed variables for which no new term is given are implicitly asserted to remain unchanged.

A nonfinal state to which no transition rule applies is called *unproceedable.* For example, there is no transition rule for a state with code $\mathrm{Q} = [\![(\texttt{wrong})\ \mathrm{G}]\!]$, so such a state is unproceedable.

Rule: `lit`—load literal
Domain Conditions
$\mathrm{Q} = [\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = \mathrm{K}$
(5.1)

Rule: `unspec`—load unspecified value
Domain Conditions
$\mathrm{Q} = [\![(\texttt{unspec})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = \mathsf{Unspecified}$
(5.2)

Rule: `ignore`—ignore value register
Domain Conditions
$\mathrm{Q} = [\![(\texttt{ignore})\ \mathrm{F}]\!]$
Changes
$\mathrm{Q}' = \mathrm{F}$
$v' = \mathsf{Unspecified}$
(5.3)

Rule: `move`—move value to stack
Domain Conditions
$\mathrm{Q} = [\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$a' = \mathsf{takefirst}(\mathrm{R}-1)a \mathbin{\S} \langle v \rangle \mathbin{\S} a \dagger \mathrm{R}$
$v' = \mathsf{Unspecified}$
(5.4)

Rule: `copy`—copy stack entry to value
Domain Conditions
$\mathrm{Q} = [\![(\texttt{copy}\ \mathrm{R})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = a \downarrow \mathrm{R}$ (5.5)

Fetch is used on `letrec`-bound variables.

Rule: `fetch`—fetch address
Domain Conditions
$\mathrm{Q} = [\![(\texttt{fetch}\ \mathrm{I})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = u\mathrm{I}$ (5.6)

Load is used on defined variables.

Rule: `load`—load global
Domain Conditions
$\mathrm{Q} = [\![(\texttt{load}\ \mathrm{R}\ \mathrm{I})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = \mathsf{storeref}(u\mathrm{I})s$ (5.7)

Store is used on defined variables.

Rule: `store`—store global
Domain Conditions
$\mathrm{Q} = [\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = \mathsf{Unspecified}$
$s' = \mathsf{storeset}(u\mathrm{I})sv$ (5.8)

Rule: add
Domain Conditions
$\mathrm{Q} = [\![(+\ \mathrm{J}_0\ \mathrm{J}_1)\ \mathrm{G}]\!]$
$a_0 = \mathsf{shuffle}\langle \mathrm{J}_0, \mathrm{J}_1\rangle a$
$n_0 = a_0 \downarrow 1$ (5.9)
$n_1 = a_0 \downarrow 2$
Changes
$\mathrm{Q}' = \mathrm{G}$
$v' = n_0 + n_1$

A state with code $\mathrm{Q} = [\![(\texttt{bra}]\!]$ is unproceedable. The `bra` instruction is removed from instructions by the transition rule for `bwf` and `pick`.

Rule: `bif`—conditional branch true without join point
Domain Conditions
$\mathrm{Q} = [\![(\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$
$v = \texttt{\#t}$ (5.10)
Changes
$\mathrm{Q}' = \mathrm{F}_0$
$v' = \mathsf{Unspecified}$

Rule: `bif`—conditional branch false without join point
Domain Conditions
$\mathrm{Q} = [\![(\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$
$v = \texttt{\#f}$ (5.11)
Changes
$\mathrm{Q}' = \mathrm{F}_1$
$v' = \mathsf{Unspecified}$

Rule: `bwf`—conditional branch true with join point
Domain Conditions
$\mathrm{Q} = [\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!]$
$v = \texttt{\#t}$ (5.12)
Changes
$\mathrm{Q}' = \mathsf{join}\,\mathrm{F}_0\mathrm{G}$
$v' = \mathsf{Unspecified}$

Rule: `bwf`—conditional branch false with join point
Domain Conditions
$$\begin{array}{l} \mathrm{Q} = [\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!] \\ v = \texttt{\#f} \end{array} \tag{5.13}$$
Changes
$\mathrm{Q}' = \mathsf{join}\,\mathrm{F}_1\mathrm{G}$
$v' = \mathsf{Unspecified}$

Rule: `select`—select without join point
Domain Conditions
$$\begin{array}{l} \mathrm{Q} = [\![(\texttt{select}\ (\mathrm{F})^*)]\!] \\ v \text{ is an integer} \end{array} \tag{5.14}$$
Changes
$\mathrm{Q}' = \mathsf{select}\,v\mathrm{F}^*$
$v' = \mathsf{Unspecified}$

Rule: `pick`—select with join point
Domain Conditions
$$\begin{array}{l} \mathrm{Q} = [\![(\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G}]\!] \\ v \text{ is an integer} \end{array} \tag{5.15}$$
Changes
$\mathrm{Q}' = \mathsf{join}(\mathsf{select}\,v\mathrm{F}^*)\mathrm{G}$
$v' = \mathsf{Unspecified}$

Rule: `call`—jump with arguments
Domain Conditions
$$\begin{array}{l} \mathrm{Q} = [\![(\texttt{call}\ \mathrm{J}^*)]\!] \\ v = [\![(\texttt{entry}\ \mathrm{N})\ \mathrm{F}]\!] \\ \#\mathrm{J}^* = \mathrm{N} \end{array} \tag{5.16}$$
Changes
$\mathrm{Q}' = \mathrm{F}$
$v' = \mathsf{Unspecified}$
$a' = \mathsf{shuffle}\,\mathrm{J}^*a$

Rule: `return`
Domain Conditions
$\mathrm{Q} = [\![(\texttt{return})]\!]$
$k = (\mathrm{G}, a_0, k_0)$
Changes
$\mathrm{Q}' = \mathrm{G}$
$a' = a_0$
$k' = k_0$
(5.17)

Rule: `mkcont`—make continuation
Domain Conditions
$\mathrm{Q} = [\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}]\!]$
$\#a = \mathrm{N}$
Changes
$\mathrm{Q}' = \mathrm{F}$
$k' = (\mathrm{G}, a, k)$
(5.18)

A state with code $\mathrm{Q} = [\![(\texttt{entry}\ \mathrm{N})\ \mathrm{F}]\!]$ is unproceedable. The `entry` instruction is removed from instructions by the transition rule for `call` and `letrec`.

Rule: `reserve`—reserve stack space
Domain Conditions
$\mathrm{Q} = [\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}]\!]$
Changes
$\mathrm{Q}' = \mathrm{F}$
$a' = a \mathbin{\S} \mathsf{unspecs}\,\mathrm{N}$
(5.19)

Rule: `dispose`—dispose of stack space
Domain Conditions
$\mathrm{Q} = [\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}]\!]$
Changes
$\mathrm{Q}' = \mathrm{G}$
$a' = \mathsf{takefirst}(\#a - \mathrm{N})a$
(5.20)

A state with code $\mathrm{Q} = [\![(\texttt{wrong})\ \mathrm{G}]\!]$ is unproceedable.

Rule: `cmt`—comment
Domain Conditions
$\mathrm{Q} = [\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{Q}_0]\!]$
Changes
$\mathrm{Q}' = \mathrm{Q}_0$
(5.21)

Rule: `letrec`—let variables be recursively defined
Domain Conditions
$\mathrm{Q} = [\![(\texttt{letrec}\ (\mathrm{B}')\ (\texttt{entry}\ 0)\ \mathrm{F})]\!]$ (5.22)
Changes
$\mathrm{Q}' = \mathrm{F}$
$u' = \mathsf{bndextends}\,\mathrm{B}'u$

Rule: `define`—define variable
Domain Conditions
$\mathrm{Q} = [\![(\texttt{define}\ \mathrm{I})\ \mathrm{P}]\!]$
$\ell = \mathsf{new}\,s$ (5.23)
Changes
$\mathrm{Q}' = \mathrm{P}$
$s' = s \diamond [\ell \mapsto \mathsf{Undefined}]$
$u' = u \diamond [\mathrm{I} \mapsto \ell]$

5.3.5 Auxiliary Functions

$\mathsf{takefirst}\,0v^* = \langle\rangle$
$\mathsf{takefirst}(n+1)(\langle v\rangle \S v^*) = \langle v\rangle \S \mathsf{takefirst}\,nv^*$

$\mathsf{shuffle}\langle\rangle a = \langle\rangle$
$\mathsf{shuffle}(\mathrm{J}^* \S \langle '\mathrm{K}\rangle)a = (\mathsf{shuffle}\,\mathrm{J}^*a) \S \langle \mathrm{K}\rangle$
$\mathsf{shuffle}(\mathrm{J}^* \S \langle \mathrm{R}\rangle)a = (\mathsf{shuffle}\,\mathrm{J}^*a) \S \langle a \downarrow \mathrm{R}\rangle$

Lemma 5.2 *For* $1 \le \mathrm{R} \le \#\mathrm{J}^*$

$$(\mathsf{shuffle}\,\mathrm{J}^*a)\downarrow \mathrm{R} = \begin{cases} \mathrm{K} & \textit{if}\ \mathrm{J}^* \downarrow \mathrm{R} = '\mathrm{K} \\ a \downarrow \mathrm{R}_0 & \textit{if}\ \mathrm{J}^* \downarrow \mathrm{R} = \mathrm{R}_0 \end{cases}$$

Proof: Induction on the length of J^* proves the lemma. ∎

$\mathsf{select}\,n\mathrm{F}^* = n < 0 \vee \#\mathrm{F}^* < n+2 \to \mathrm{F}^* \downarrow 1, \mathrm{F}^* \downarrow (n+2)$

$\mathsf{unspec}\,0 = \langle\rangle$
$\mathsf{unspec}(\mathrm{N}+1) = \langle \mathsf{Unspecified}\rangle \S \mathsf{unspec}\,\mathrm{N}$

$\mathsf{storeref}\,\ell s = s\ell$

$$\mathsf{storeset}\,\ell s v = s \diamond [\ell \mapsto v]$$

$$\begin{aligned}\mathsf{bndextends}[\![\ ]\!]u &= u\\ \mathsf{bndextends}[\![(\mathrm{I}\ \mathrm{F})\ \mathrm{B}']\!]u &= \mathsf{bndextends}\,\mathrm{B}'(u \diamond [\mathrm{I} \mapsto \mathrm{F}])\end{aligned}$$

The function new has the property that $\mathsf{new}\,s \notin \mathrm{dom}(s)$.

5.3.6 Initial and Final States

An initial state for program P' has the form:

$$\begin{aligned}\mathrm{Q} &= \mathrm{P}'\\ a &= \langle\rangle\\ k &= \mathsf{Halt}\\ s &= [\,]\\ u &= [\,]\end{aligned}$$

An accepting final state with answer n has the form:

$$\begin{aligned}\mathrm{Q} &= [\![\texttt{(return)}]\!]\\ v &= n\\ k &= \mathsf{Halt}\end{aligned}$$

5.4 Properties of the Operational Semantics

The SAL operational semantics will be shown to be faithful to its denotational semantics in the sense that, for programs on which the operational semantics produces an accepting state with an answer, that answer will be the denotation of the program as given by the its denotational semantics.

While addressing faithfulness, the set of programs considered will be restricted to ones that are *stack safe.* Stack machine states that evolve from stack safe programs always have an appropriately sized stack. In particular, all stack references in code refer to stack locations that exist.

Stack safety is a syntactic property of code, and it will be shown that only stack safe code is generated by the compiler which produces SAL programs. In the definition of stack safe code, the cases for `call`, `mkcont`, `entry`, `reserve`, and `dispose` deserve special attention.

Definition 5.3 (Stack Safe Code) *Stack machine code* Q *is said to be stack safe for stack size* N*, written* $\mathrm{N} \rhd \mathrm{Q}$ *if it satisfies the following recursive conditions:*

1. $\mathrm{N} \rhd [\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$.

2. $\mathrm{N} \rhd [\![(\texttt{unspec})\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$.

3. $\mathrm{N} \rhd [\![(\texttt{ignore})\ \mathrm{F}]\!]$ if $\mathrm{N} \rhd \mathrm{F}$.

4. $\mathrm{N} \rhd [\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!]$ if $\mathrm{R} \le \mathrm{N}$ and $\mathrm{N} \rhd \mathrm{G}$.

5. $\mathrm{N} \rhd [\![(\texttt{copy}\ \mathrm{R})\ \mathrm{G}]\!]$ if $\mathrm{R} \le \mathrm{N}$ and $\mathrm{N} \rhd \mathrm{G}$.

6. $\mathrm{N} \rhd [\![(\texttt{fetch}\ \mathrm{I})\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$.

7. $\mathrm{N} \rhd [\![(\texttt{load}\ \mathrm{I})\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$.

8. $\mathrm{N} \rhd [\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$.

9. $\mathrm{N} \rhd [\![(\texttt{+}\ \mathrm{J}^*)\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$ and for $1 \le \nu \le \#\mathrm{J}^*$, when $\mathrm{J}^* \downarrow \nu = \mathrm{R}$ then $\mathrm{R} \le \mathrm{N}$.

10. $\mathrm{N} \rhd [\![(\texttt{brf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$ if $\mathrm{N} \rhd \mathrm{F}_0$ and $\mathrm{N} \rhd \mathrm{F}_1$.

11. $\mathrm{N} \rhd [\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathsf{join}\,\mathrm{F}_0\mathrm{G}$, $\mathrm{N} \rhd \mathsf{join}\,\mathrm{F}_1\mathrm{G}$, and $\mathrm{N} \rhd \mathrm{G}$.

12. $\mathrm{N} \rhd [\![(\texttt{select}\ (\mathrm{F})^*)]\!]$ if for $1 \le \nu \le \#\mathrm{F}^*$, $\mathrm{N} \rhd \mathrm{F}^* \downarrow \nu$.

13. $\mathrm{N} \rhd [\![(\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$ and for $1 \le \nu \le \#\mathrm{F}^*$, $\mathrm{N} \rhd \mathsf{join}(\mathrm{F}^* \downarrow \nu)\mathrm{G}$.

14. $\mathrm{N} \rhd [\![(\texttt{call}\ \mathrm{J}^*)]\!]$ if for $1 \le \nu \le \#\mathrm{J}^*$, when $\mathrm{J}^* \downarrow \nu = \mathrm{R}$ then $\mathrm{R} \le \mathrm{N}$.

15. $\mathrm{N} \rhd [\![(\texttt{return})]\!]$.

16. $\mathrm{N} \rhd [\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{F}$ and $\mathrm{N} \rhd \mathrm{G}$.

17. $\mathrm{N} \rhd [\![(\texttt{entry}\ \mathrm{N})\ \mathrm{F}]\!]$ if $\mathrm{N} \rhd \mathrm{F}$.

18. $\mathrm{N}_0 \rhd [\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}]\!]$ if $\mathrm{N}_0 + \mathrm{N} \rhd \mathrm{F}$.

19. $\mathrm{N}_0 \rhd [\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}]\!]$ if $\mathrm{N}_0 - \mathrm{N} \rhd \mathrm{G}$.

20. $\mathrm{N} \rhd [\![(\texttt{wrong}\ \mathrm{K}^*)\ \mathrm{G}]\!]$ if $\mathrm{N} \rhd \mathrm{G}$.

21. $\mathrm{N} \triangleright [\![$(`cmt` K^*) Q$]\!]$ if $\mathrm{N} \triangleright \mathrm{Q}$.

22. $0 \triangleright [\![$(`letrec` (B') F)$]\!]$ if $0 \triangleright \mathrm{F}$ and for each binding (I F_0) in B', $\mathrm{N} \triangleright \mathrm{F}_0$ for some N.

23. $0 \triangleright [\![$(`define` I) $\mathrm{P}']\!]$ if $0 \triangleright \mathrm{P}'$.

Definition 5.4 (Stack Safe Continuation) *The continuation* Halt *is stack safe, and the continuation* (G, a, k) *is stack safe iff* $\#a \triangleright \mathrm{G}$ *and* k *is stack safe.*

Definition 5.5 (Stack Safe State) *The stack machine state* $(\mathrm{Q}, v, a, k, s, u)$ *is stack safe iff* $\#a \triangleright \mathrm{Q}$ *and continuation* k *is stack safe.*

Lemma 5.6 (Stack Safety Preservation) *If state* Σ_0^S *is stack safe and* $\Sigma_0^S \Longrightarrow^* \Sigma_1^S$ *then state* Σ_1^S *is stack safe.*

Proof: The proof proceeds by analyzing each transition rule. The proof requires concluding that the code for the resulting state of a transition is stack safe given the that the code for the starting state is stack safe. The conclusion is warranted because the stack safe code relation is the least set which satisfies the given conditions, and there is only one condition for each SAL instruction. With this observation, the proof is straightforward. ∎

Theorem 5.7 (Faithfulness) *If* Σ_0^S *is an initial state for a stack safe program* P' *and there exists an accepting final state* Σ_1^S *with answer* ζ *such that* $\Sigma_0^S \Longrightarrow^* \Sigma_1^S$*, then* $\mathcal{P}'[\![\mathrm{P}']\!] = \zeta$ in R.

The faithfulness proof has been omitted. A complete proof would employ the well worked out procedure given in [2]. The operational semantics was derived from the denotational one—which is why a formal proof was deemed unnecessary.

Chapter 6

The Syntax-Directed Compiler

The syntax-directed compiler is given as a function which maps Simple PreScheme programs into Stack Assembly Language. The function is presented using the same notation used to map syntactic phrases into semantic values. A syntactic domain is a flat domain constructed from a syntactic category. The compiler maps syntactic phrases into the syntactic domain associated with SAL programs. Syntactic categories are described in the remarks at the end of section 5.1.

The functions which define the heart of the compiler take five arguments. The first argument is the expression being compiled. The second argument is the compile-time environment which records the variables that are allocated on the stack along with their offset. The third argument gives the destination of the value being computed. The fourth argument gives the number of locations on the stack that are already in use or have been reserved for future use. The fifth argument is the code to be appended after the code for the current expression.

The actual syntax-directed compiler is implemented to handle multiple threads. However, the definition of the compiler given in this chapter assumes that programs are single-threaded. Moreover, the correctness of the syntax-directed compiler is justified relative to only the thread-unaware semantics. To extend the compiler definition to the full multithreaded language and to prove the correctness of the compiler relative to the thread-aware semantics would be a straightforward, but somewhat tedious, exercise.

The Simple PreScheme compiler is very similar to the Pure PreScheme Compiler [16] and the VLISP byte-code compiler described in [6] and in detail in [5]; however, this compiler has an additional argument giving the destination of the computed value. When the argument is zero, the destination is the value register. When the argument is positive, the destination is the stack location given by the argument. When the argument is negative, the value will be ignored.

Recording the destination allows the production of more efficient code. SAL instructions usually either produce a value and place it in the value register or consume a value contained in that register. This compiler places most value consuming instructions immediately after their matching value producing instruction. Later passes of the compiler can often translate these pairs of instructions into one machine instruction.

6.1 Definition of the Compiler

$$\begin{aligned}
\mathcal{CL} &: \text{Smpl} \to U_c \to R \to N \to \text{User} \to \text{Form} \\
\mathcal{CS} &: \text{Smpl} \to U_c \to R \to N \to \text{User} \to \text{Form} \\
\mathcal{CC} &: \text{Cls} \to U_c \to R \to N \to \text{User} \to \text{Form}^* \\
\mathcal{CS}^* &: \text{Smpl}^* \to U_c \to N \to \text{Form} \to \text{Form} \\
\mathcal{CB} &: \text{Bnd} \to \text{Bnd}' \\
\mathcal{CE} &: \text{Exp} \to \text{Exp}' \\
\mathcal{CP} &: \text{Pgm} \to \text{Pgm}'
\end{aligned}$$

Note: all of the domains mentioned here except for U_c have been defined in previous sections; U_c is defined below in Section 6.2. Also, in this chapter the variable R will range over the integers greater than or equal to -1.

$$\begin{aligned}
&\mathcal{CL}[\![\texttt{(lambda}\ (\text{I}^*)\ \text{S})]\!]\gamma\text{R}\nu\text{G} = \\
&\quad \mathcal{CS}[\![\text{S}]\!](\mathsf{extends}\,\gamma\text{I}^*\nu) \\
&\qquad\quad \text{R} \\
&\qquad\quad (\nu + \#\text{I}^*) \\
&\qquad\quad (\mathsf{isreturn}\ \text{G} \to \text{G}, \mathsf{mkdispose}\,\#\text{I}^*\text{G})
\end{aligned}$$

$$\mathcal{CS}[\![\text{K}]\!]\gamma\text{R}\nu\text{G} = \mathsf{mkliteral}\,\text{K}(\mathsf{mkmove}\,\text{RG})$$

$$\mathcal{CS}[\![\text{I}]\!]\gamma\text{R}\nu\text{G} = \mathsf{mkvarref}\,\text{I}\gamma(\mathsf{mkmove}\,\text{RG})$$

When S* contains only `lambda`-bound variables or constants,

$$\begin{aligned}
&\mathcal{CS}[\![(\text{S S}^*)]\!]\gamma\text{R}\nu[\![\texttt{(return)}]\!] = \\
&\quad \mathcal{CS}[\![\text{S}]\!]\gamma\text{R}0(\mathsf{mkcall}(\mathsf{map}(\mathsf{local}\,\gamma)\text{S}^*))
\end{aligned}$$

When S* contains something other than `lambda`-bound variables or constants,

$\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma \mathrm{R}\nu[\![(\texttt{return})]\!] =$
$\quad\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ (\mathrm{S}\ \mathrm{I}^*))\ \mathrm{S}^*)]\!]\gamma \mathrm{R}\nu\, \mathsf{mkreturn}$

where the identifiers I^* are chosen so that none are free in S.[1]

$\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma \mathrm{R}\nu \mathrm{G} =$
$\quad\mathsf{mkmkcont}\,\nu(\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma 0\nu\, \mathsf{mkreturn})(\mathsf{mkmove}\,\mathrm{RG})$
$\quad\text{when } \mathsf{isreturn}\,\mathrm{G} = \mathit{false}$

$\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!]\gamma \mathrm{R}\nu \mathrm{G} =$
$\quad\mathsf{mkreserve}\,\#\mathrm{I}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\nu + \#\mathrm{I}^*)(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma \mathrm{R}\nu \mathrm{G}))$

$\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S})]\!] = \mathcal{CS}[\![\mathrm{S}]\!]$

$\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S}\ \mathrm{S}^*\ \mathrm{S}_0)]\!]\gamma \mathrm{R}\nu \mathrm{G} =$
$\quad\mathcal{CS}[\![\mathrm{S}]\!]\gamma(-1)\nu(\mathsf{mkignore}(\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S}^*\ \mathrm{S}_0)]\!]\gamma \mathrm{R}\nu \mathrm{G}))$

When S* contains something other than `lambda`-bound variables or constants,

$\mathcal{CS}[\![(\texttt{if}\ (\mathrm{O}\ \mathrm{S}^*)\ \mathrm{S}_1\ \mathrm{S}_2)]\!] =$
$\quad\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ (\texttt{if}\ (\mathrm{O}\ \mathrm{I}^*)\ \mathrm{S}_1\ \mathrm{S}_2))\ \mathrm{S}^*)]\!]$

where the identifiers I^* are chosen so that none are free in S_1 or S_2.[2]

```
$\mathcal{CS}[\![(\texttt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma \mathrm{R}\nu \mathrm{G} =$
  let
    $\mathrm{F}_1 = \mathcal{CS}[\![\mathrm{S}_1]\!]\gamma \mathrm{R}\nu(\mathsf{maybemkbranch}\,\mathrm{G})$
    $\mathrm{F}_2 = \mathcal{CS}[\![\mathrm{S}_2]\!]\gamma \mathrm{R}\nu(\mathsf{maybemkbranch}\,\mathrm{G})$
  in
    $\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma 0\nu(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G})$
```

$\mathcal{CS}[\![(\texttt{if}\ \texttt{\#f}\ \texttt{\#f})]\!]\gamma\nu \mathrm{RG} = \mathsf{mkunspecified}(\mathsf{mkmove}\,\mathrm{RG})$

[1]The actual compiler allocates a fresh variable only for a simple expression which is not a `lambda`-bound variable or a constant.

[2]Same as previous footnote.

$$
\begin{array}{l}
\mathcal{CS}[\![(\texttt{case}\ \mathrm{S}_0\ \mathrm{C}\ (\texttt{else}\ \mathrm{S}_1))]\!]\gamma\mathrm{R}\nu\mathrm{G} = \\
\quad \mathrm{let} \\
\qquad \mathrm{F} = \mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\mathrm{R}\nu(\mathsf{maybemkbranch}\,\mathrm{G}) \\
\qquad \mathrm{F}^* = \mathrm{F}\ \S\ \mathcal{CC}[\![\mathrm{C}]\!]\gamma\mathrm{R}\nu(\mathsf{maybemkbranch}\,\mathrm{G}) \\
\quad \mathrm{in} \\
\qquad \mathcal{CS}[\![\mathrm{S}_0]\!]\gamma 0\nu(\mathsf{mkselection}\,\mathrm{F}^*\mathrm{G})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CS}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!]\gamma\mathrm{R}\nu\mathrm{G} = \\
\quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma 0\nu(\mathsf{islocal}\,\gamma\mathrm{I} \vee \mathsf{isletrec}\,\mathrm{I} \rightarrow \mathsf{mkwrong}\,\mathrm{G}, \\
\qquad\qquad\qquad \mathsf{mkstore}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))
\end{array}
$$

For constants or `lambda`-bound variables S_0 and S_1,

$$
\mathcal{CS}[\![\langle +\ \mathrm{S}_0\ \mathrm{S}_1\rangle]\!]\gamma\mathrm{R}\nu\mathrm{G} = \mathsf{mkadd}(\mathsf{local}\,\gamma\mathrm{I}_0)(\mathsf{local}\,\gamma\mathrm{I}_1)(\mathsf{mkmove}\,\mathrm{RG})
$$

When either S_0 or S_1 is something other than a `lambda`-bound variable or a constant,[3]

$$
\begin{array}{l}
\mathcal{CS}[\![\langle +\ \mathrm{S}_0\ \mathrm{S}_1\rangle]\!]\gamma\mathrm{R}\nu\mathrm{G} = \\
\quad \mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}_0\ \mathrm{I}_1)\ \langle +\ \mathrm{I}_0\ \mathrm{I}_1\rangle)\ \mathrm{S}_0\ \mathrm{S}_1)]\!]\gamma\mathrm{R}\nu\mathrm{G}
\end{array}
$$

$$
\mathcal{CS}^*[\![\ ]\!]\gamma\nu\mathrm{F} = \mathrm{F}
$$

$$
\begin{array}{l}
\mathcal{CS}^*[\![\mathrm{S}\ \mathrm{S}^*]\!]\gamma\nu\mathrm{F} = \\
\quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma(\nu - \#\mathrm{S}^*)\nu(\mathsf{mkignore}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma\nu\mathrm{F}))
\end{array}
$$

$$
\mathcal{CC}[\![((\mathrm{K})\ \mathrm{S})]\!]\gamma\mathrm{R}\nu\mathrm{G} = \langle\mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu\mathrm{G}\rangle
$$

$$
\mathcal{CC}[\![((\mathrm{K})\ \mathrm{S})\ \mathrm{C}]\!]\gamma\mathrm{R}\nu\mathrm{G} = \langle\mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu\mathrm{G}\rangle\ \S\ \mathcal{CC}[\![\mathrm{C}]\!]\gamma\mathrm{R}\nu\mathrm{G}
$$

$$
\mathcal{CB}[\![\ ]\!] = \langle\rangle
$$

$$
\begin{array}{l}
\mathcal{CB}[\![(\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S}))\ \mathrm{B}]\!] = \\
\quad \mathrm{let} \\
\qquad \mathrm{F} = \mathsf{mkentry}\,\#\mathrm{I}^*(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma_0 00\,\mathsf{mkreturn}) \\
\quad \mathrm{in} \\
\qquad [\![(\mathrm{I}\ \mathrm{F})]\!]\ \S\ \mathcal{CB}[\![\mathrm{B}]\!]
\end{array}
$$

[3]Same as previous footnote.

$\mathcal{CE}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!] =$
```
let
   $\mathrm{B}' = \mathcal{CB}[\![\mathrm{B}]\!]$
   $\mathrm{F} = \mathsf{mkentry}\, 0(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 00\, \mathsf{mkreturn})$
in
   $[\![(\texttt{letrec}\ (\mathrm{B}')\ \mathrm{F})]\!]$
```

$\mathcal{CP}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!] = \mathcal{CE}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!]$

$\mathcal{CP}[\![(\texttt{define}\ \mathrm{I})\ \mathrm{P}]\!] = [\![(\texttt{define}\ \mathrm{I})]\!] \mathbin{\S} \mathcal{CP}[\![\mathrm{P}]\!]$

6.2 Auxiliary Compiler Functions

This section gives the properties of compile-time environments and defines the function isreturn. The definitions of the code generators are given in Figure 6.1.

Compile-time environments are represented as a sequence of pairs. Each pair associates a nonnegative integer with an identifier.

$\gamma \in U_c = (\mathrm{Ide} \times N)^*$

$\gamma_0 : U_c$
$\gamma_0 = \langle\rangle$

$\mathsf{extends} : U_c \to \mathrm{Ide}^* \to N \to U_c$
$\mathsf{extends}\,\gamma\langle\rangle\nu = \gamma$
$\mathsf{extends}\,\gamma(\langle \mathrm{I}\rangle \mathbin{\S} \mathrm{I}^*)\nu = \mathsf{extends}(\langle(\mathrm{I}, \nu+1)\rangle \mathbin{\S} \gamma)\mathrm{I}^*(\nu+1)$

$\mathsf{islocal} : U_c \to \mathrm{Ide} \to T$
$\mathsf{islocal}\,\gamma_0\mathrm{I} = \mathit{false}$
$\mathsf{islocal}(\langle(\mathrm{I}_0, \nu)\rangle \mathbin{\S} \gamma)\mathrm{I} = (\mathrm{I}_0 = \mathrm{I} \vee \mathsf{islocal}\,\gamma\mathrm{I})$

$\mathsf{local} : U_c \to \mathrm{Smpl} \to \mathrm{Src}$
$\mathsf{local}\,\gamma_0\mathrm{I} = \mathrm{I}$
$\mathsf{local}(\langle(\mathrm{I}_0, \nu)\rangle \mathbin{\S} \gamma)\mathrm{I} = (\mathrm{I}_0 = \mathrm{I} \to \nu, \mathsf{local}\,\gamma\mathrm{I})$
$\mathsf{local}\,\gamma\mathrm{K} = {}'\mathrm{K}$

$$
\begin{aligned}
&\mathsf{mkliteral}\,\mathrm{KG} &&= [\![\texttt{(lit }'\mathrm{K})\ \mathrm{G}]\!]\\
&\mathsf{mkunspecified}\,\mathrm{G} &&= [\![\texttt{(unspec)}\ \mathrm{G}]\!]\\
&\mathsf{mkignore}\,\mathrm{F} &&= [\![\texttt{(ignore)}\ \mathrm{F}]\!]\\
&\mathsf{mkmove}\,\mathrm{RG} &&= \mathrm{R} = 0 \vee \mathsf{isreturn}\,\mathrm{G} \to \mathrm{G},\\
& && \qquad \mathrm{R} > 0 \to [\![\texttt{(move }\mathrm{R})\ \mathrm{G}]\!],\\
& && \qquad\quad [\![\texttt{(ignore)}\ \texttt{(unspec)}\ \mathrm{G}]\!]\\
&\mathsf{mkcopy}\,\mathrm{RG} &&= [\![\texttt{(copy }\mathrm{R})\ \mathrm{G}]\!]\\
&\mathsf{mkfetch}\,\mathrm{IG} &&= [\![\texttt{(fetch }\mathrm{I})\ \mathrm{G}]\!]\\
&\mathsf{mkload}\,\mathrm{IG} &&= [\![\texttt{(load }\mathrm{I})\ \mathrm{G}]\!]\\
&\mathsf{mkstore}\,\mathrm{IG} &&= [\![\texttt{(store }\mathrm{I})\ \mathrm{G}]\!]\\
&\mathsf{mkadd}\,\mathrm{J_0 J_1 G} &&= [\![(+\ \mathrm{J_0}\ \mathrm{J_1})\ \mathrm{G}]\!]\\
&\mathsf{maybemkbranch}\,\mathrm{G} &&= \mathsf{isreturn}\,\mathrm{G} \to \mathrm{G}, [\![\texttt{(bra)}]\!]\\
&\mathsf{mkbranchfalse}\,\mathrm{F_0 F_1 G_2} &&= \mathsf{isreturn}\,\mathrm{G_2} \to [\![\texttt{(bif }(\mathrm{F_0})\ (\mathrm{F_1}))]\!],\\
& && \qquad [\![\texttt{(bwf }(\mathrm{F_0})\ (\mathrm{F_1}))\ \mathrm{G_2}]\!]\\
&\mathsf{mkselection}\,\mathrm{F^*G} &&= \mathsf{isreturn}\,\mathrm{G} \to [\![\texttt{(select }(\mathrm{F})^*)]\!],\\
& && \qquad [\![\texttt{(pick }(\mathrm{F})^*)\ \mathrm{G}]\!]\\
&\mathsf{mkcall}\,\mathrm{J^*} &&= [\![\texttt{(call }\mathrm{J^*})]\!]\\
&\mathsf{mkreturn} &&= [\![\texttt{(return)}]\!]\\
&\mathsf{mkmkcont}\,\mathrm{NF_0G_1} &&= [\![\texttt{(mkcont }\mathrm{N}\ \mathrm{F_0})\ \mathrm{G_1}]\!]\\
&\mathsf{mkentry}\,\mathrm{NF} &&= [\![\texttt{(entry }\mathrm{N})\ \mathrm{F}]\!]\\
&\mathsf{mkreserve}\,\mathrm{NF} &&= [\![\texttt{(reserve }\mathrm{N})\ \mathrm{F}]\!]\\
&\mathsf{mkdispose}\,\mathrm{NG} &&= [\![\texttt{(dispose }\mathrm{N})\ \mathrm{G}]\!]\\
&\mathsf{mkcmt}\,\mathrm{K^*Q} &&= [\![\texttt{(cmt }\mathrm{K^*})\ \mathrm{Q}]\!]\\
&\mathsf{mkwrong}\,\mathrm{IG} &&= [\![\texttt{(wrong)}\ \mathrm{G}]\!]
\end{aligned}
$$

Figure 6.1: Code Generators

$$\begin{aligned}&\mathsf{mkvarref} : \mathrm{Ide} \to U_c \to \mathrm{User} \to \mathrm{Form}\\&\mathsf{mkvarref} = \lambda \mathrm{I}\gamma\mathrm{G}.\, \mathsf{islocal}\,\gamma\mathrm{I} \to \mathsf{mkcopy}(\mathsf{local}\,\gamma\mathrm{I})\mathrm{G},\\&\qquad\qquad\qquad \mathsf{isletrec}\,\mathrm{I} \to \mathsf{mkfetch}\,\mathrm{IG}, \mathsf{mkload}\,\mathrm{IG}\end{aligned}$$

$$\begin{aligned}&\mathsf{isreturn} : \mathrm{Code} \to T\\&\mathsf{isreturn} = \lambda \mathrm{Q}.\, (\mathrm{Q} = \mathsf{mkreturn})\end{aligned}$$

6.3 Example

The results of applying the compiler to the Simple PreScheme program in Figure 4.8 is shown in Figure 6.2.

6.4 Correctness

The Simple PreScheme compiler is shown to be correct by proving that the denotation of a translated program is the same as the denotation of its source, i.e., $\mathcal{P}'(\mathcal{CP}[\![\mathrm{P}]\!]) = \mathcal{P}_0[\![\mathrm{P}]\!]$. The correctness of this theorem depends on the fact that Simple PreScheme programs are strongly typed. For example, in light of strong typing, one can infer that all procedures receive the correct number of arguments.

An overview of the theorems in this section follows. Theorem 6.2 shows that instructions cause a jump to the intended location. Theorem 6.7 shows that the compiler produces only stack safe code. Theorem 6.10 shows that the actions of shortening the stack and then updating its contents can be performed in the opposite order. Theorem 6.14 shows the correctness of the environment created by a `let` expression. Theorem 6.16 and Theorem 6.17 show the correctness of the compiler on simple, and sequences of simple, expressions. Theorems 6.19 through 6.22 show the correctness of the parts of the compiler which compose multiple sequences of instructions into a complete program.

6.4.1 Code Branching

Lemma 6.1 *If* $\mathrm{G}_0 \neq [\![\texttt{(return)}]\!]$ *and* $\mathrm{G}_1 \neq [\![\texttt{(return)}]\!]$ *then*

$$\mathsf{join}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu\mathrm{G}_0)\mathrm{G}_1 = \mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu(\mathsf{join}\,\mathrm{G}_0\mathrm{G}_1)$$

```
(define number) (define dec) (define even) (define odd)
(letrec
    ((even-0
        (entry 1)             ; (lambda (y)
        (= '0 1)              ;   (if (= 0 y)
        (bif
          ((lit '0)           ;       0
           (return))
          ((reserve 1)        ;       (odd-0 (- y 1))))
           (- 1 '1)
           (move 2)
           (ignore)
           (fetch odd-0)
           (call 2))))
     (odd-0 ...))
   (unspec)                   ; (set! number (if #f #f))
   (store number)
   (ignore)
   (unspec)
   (store dec)
   (ignore)
   (unspec)
   (store even)
   (ignore)
   (unspec)
   (store odd)
   (ignore)
   (fetch even-0)
   (call '77))                ; (even-0 77)
```

Figure 6.2: Even-odd in SAL

Proof sketch: The proof proceeds by tedious, but straightforward structural induction on simple expressions. The case of procedure call and conditional expressions are displayed.

Case of a non-tail-recursive call:

$$\begin{aligned}
&\text{Let } \mathrm{F} = \mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\mathrm{R}\nu\,\mathsf{mkreturn}\\
&\mathsf{join}(\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\mathrm{R}\nu\mathrm{G}_0)\mathrm{G}_1\\
&\quad = \mathsf{join}(\mathsf{mkmkcont}\,\nu\mathrm{F}(\mathsf{mkmove}\,\mathrm{R}\mathrm{G}_0))\mathrm{G}_1\\
&\quad = \mathsf{mkmkcont}\,\nu\mathrm{F}(\mathsf{join}(\mathsf{mkmove}\,\mathrm{R}\mathrm{G}_0)\mathrm{G}_1)\\
&\quad = \mathsf{mkmkcont}\,\nu\mathrm{F}(\mathsf{mkmove}\,\mathrm{R}(\mathsf{join}\,\mathrm{G}_0\mathrm{G}_1))\\
&\quad = \mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\mathrm{R}\nu(\mathsf{join}\,\mathrm{G}_0\mathrm{G}_1)
\end{aligned}$$

Case of an `if` expression:

$$\begin{aligned}
&\text{Let } \mathrm{F}_1 = \mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\mathrm{R}\#\epsilon^*(\mathsf{maybemkbranch}\,\mathrm{G}_0)\\
&\qquad \mathrm{F}_2 = \mathcal{CS}[\![\mathrm{S}_2]\!]\gamma\mathrm{R}\#\epsilon^*(\mathsf{maybemkbranch}\,\mathrm{G}_0)\\
&\mathsf{join}(\mathcal{CS}[\![(\mathtt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma\mathrm{R}\nu\mathrm{G}_0)\mathrm{G}_1\\
&\quad = \mathsf{join}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma 0\nu(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G}_0))\mathrm{G}_1\\
&\quad = \mathcal{CS}[\![\mathrm{S}_0]\!]\gamma 0\nu(\mathsf{join}(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G}_0)\mathrm{G}_1)\\
&\quad = \mathcal{CS}[\![\mathrm{S}_0]\!]\gamma 0\nu(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2(\mathsf{join}\,\mathrm{G}_0\mathrm{G}_1))\\
&\quad = \mathcal{CS}[\![(\mathtt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma\mathrm{R}\nu(\mathsf{join}\,\mathrm{G}_0\mathrm{G}_1)
\end{aligned}$$

∎

Theorem 6.2 *If* $\mathrm{G} \neq [\![\mathtt{(return)}]\!]$ *then*

$$\mathsf{join}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu[\![\mathtt{(bra)}]\!])\mathrm{G} = \mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu\mathrm{G}$$

Proof: Instantiate the preceding lemma with $\mathrm{G}_0 = [\![\mathtt{(bra)}]\!]$. ∎

The actual compiler generates assembly language comments so that one can trace instructions to their source. The compiler displayed does not generate comments; however, notice that Theorem 6.2 is valid in the presence of comments.

6.4.2 Stack Safety

This section shows that the compiler generates stack safe code. A lemma which relates compile-time environments to stack sizes precedes the main theorem on stack safety. The function size gives the minimum size of a stack which is compatible with a compile time environment.

Definition 6.3

$$\begin{array}{l}\mathsf{size} : U_c \to N \\ \mathsf{size}\,\gamma_0 = 0 \\ \mathsf{size}(\langle(\mathrm{I},\nu)\rangle \mathbin{\S} \gamma) = max(\nu, \mathsf{size}\,\gamma)\end{array}$$

Lemma 6.4 $\mathsf{size}\,\gamma \leq \nu$ *implies* $\mathsf{size}(\mathsf{extends}\,\gamma\mathrm{I}^*\nu) \leq \#\mathrm{I}^* + \nu$.

Proof: Proved by induction on I^*. When $\mathrm{I}^* = \langle\rangle$, the proof is trivial, so assume there is at least one identifier.

$$\begin{array}{l}\mathsf{size}(\mathsf{extends}\,\gamma(\langle\mathrm{I}\rangle \mathbin{\S} \mathrm{I}^*)\nu) \\ \quad = \mathsf{size}(\mathsf{extends}(\langle(\mathrm{I}, \nu+1)\rangle \mathbin{\S} \gamma)\mathrm{I}^*(\nu+1))\end{array}$$

The result follows from the use of the induction hypothesis given its assumption is satisfied:

$$\mathsf{size}(\langle(\mathrm{I}, \nu+1)\rangle \mathbin{\S} \gamma) \leq \nu + 1$$

The definition of size shows the assumption is satisfied. ▮

Consider the stack safety of simple expressions.

Lemma 6.5 *If* $\mathsf{size}\,\gamma \leq \nu$, $\mathrm{R} \leq \nu$, *and* $\nu \triangleright \mathrm{G}$, *then* $\nu \triangleright \mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\nu\mathrm{G}$.

Lemma 6.6 *If* $\mathsf{size}\,\gamma \leq \nu$ *and* $\nu \triangleright \mathrm{F}$, *then* $\nu \triangleright \mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma\nu\mathrm{F}$.

Proof: The previous two lemmas are proved using simultaneous structural induction on simple expressions. A complete proof would follow the pattern used to prove Theorems 6.16 and 6.17, the theorems that assert the correctness of simple expression compilation. Since it is much easier to show that the code generated by compiling simple expressions is stack safe, only the case of a `let` expression is displayed.

By assumption, $\mathsf{size}\,\gamma \leq \nu$, $\mathrm{R} \leq \nu$, and $\nu \triangleright \mathrm{G}$. We want to show

$$\nu \triangleright \mathcal{CS}[\![\texttt{((lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{) }\mathrm{S}^*\texttt{)}]\!]\gamma\mathrm{R}\nu\mathrm{G}.$$

Let

$$\begin{array}{l}\mathrm{F} = \mathcal{CL}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\gamma\mathrm{R}\nu\mathrm{G} \\ \quad = \mathcal{CS}[\![\mathrm{S}]\!](\mathsf{extends}\,\gamma\mathrm{I}^*\nu)\mathrm{R}(\nu + \#\mathrm{I}^*)\mathrm{G}_0\end{array}$$

where

$$\mathrm{G}_0 = (\mathsf{isreturn}\,\mathrm{G} \to \mathrm{G}, \mathsf{mkdispose}\,\#\mathrm{I}^*\mathrm{G}).$$

Also,

$$\begin{aligned}&\mathcal{CS}[\![(\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!]\gamma\mathrm{R}\nu\mathrm{G}\\ &\quad = \mathsf{mkreserve}\,\#\mathrm{I}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\nu + \#\mathrm{I}^*)\mathrm{F})\end{aligned}$$

Since $\nu + \#\mathrm{I}^* \triangleright \mathrm{G}_0$, the use of Lemma 6.4 and the induction hypothesis on simple expressions gives

$$\nu + \#\mathrm{I}^* \triangleright \mathrm{F}.$$

The use of the induction hypothesis on sequences of simple expressions gives

$$\nu + \#\mathrm{I}^* \triangleright \mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\nu + \#\mathrm{I}^*)\mathrm{F}.$$

The definition of stack safe code for the `reserve` instruction gives the desired result. ∎

Theorem 6.7 *If* $\mathrm{P}' = \mathcal{CP}[\![\mathrm{P}]\!]$ *for some program* P*, then* P' *is stack safe code.*

A proof of this theorem is straightforward and follows the pattern used to prove Theorem 6.22, and thus has been omitted.

6.4.3 Results Placement

As mentioned before, a unique feature of this compiler is that it records the destination of the value being computed. Semantically, when the destination is the value register, i.e. $\mathrm{R} = 0$, after computing a value, the computation continues by passing that value to $\lambda\epsilon.\,\pi\epsilon\epsilon^*\kappa$, where π is the meaning of the compiler's after code, ϵ^* is the stack, and κ is the continuation. In the general case, this computation continues by passing the value to $\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}\pi\epsilon\epsilon^*\kappa$, where *domove* is defined so as to update the stack when R is positive, and ignore the value when R is negative.

Definition 6.8

$$\begin{aligned}&\mathit{domove} : \mathrm{Ref} \to Q \to Q\\ &\mathit{domove} = \lambda\mathrm{R}\pi.\,\mathrm{R} = 0 \vee \pi = \mathit{return} \to \pi,\\ &\qquad\qquad\qquad \mathrm{R} > 0 \to \mathit{move}\,\mathrm{R}\pi,\\ &\qquad\qquad\qquad \lambda\epsilon.\,\pi(\mathit{unspecified}\,\mathrm{in}\,E)\end{aligned}$$

The function *domove* has the property that

Lemma 6.9 $\mathcal{G}(\mathsf{mkmove}\,\mathrm{R}\mathrm{G})\rho = \mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)$

Two useful identities follow:

$$\mathcal{G}(\mathsf{mkmove}\,0\mathrm{G}) = \mathcal{G}[\![\mathrm{G}]\!]$$
$$\mathcal{G}(\mathsf{mkmove}\,\mathrm{R}[\![(\mathtt{return})]\!])\rho = \mathit{return}$$

Placing value consuming instructions immediately after their matching value producing instruction poses only trivial additional proof obligations with one exception: the `dispose` instruction which shortens the stack. Theorem 6.10 shows that the actions of shortening the stack and then updating its contents can be performed in the opposite order.

Theorem 6.10 *Assume* $\mathrm{R} \leq \nu \leq \#\epsilon^*$.

$$\mathit{domove}\,\mathrm{R}(\mathit{dispose}(\#\epsilon^* - \nu)\pi)\epsilon\epsilon^* = \mathit{domove}\,\mathrm{R}\pi\epsilon(\mathit{takefirst}\,\epsilon^*\nu)$$

Proof: When R is nonpositive, the result is trivial because *domove* does nothing to the stack and so does not interfere with the shortening of the stack by *dispose*. Otherwise, the crux of the proof is showing that shortening the stack before updating it is the same as updating before shortening. After expanding definitions, the proof reduces to showing

$$\begin{aligned}&\mathit{takefirst}(\mathit{takefirst}\,\epsilon^*(\mathrm{R}-1)\mathbin{\S}\langle\epsilon\rangle\mathbin{\S}\mathit{dropfirst}\,\epsilon^*\mathrm{R})\nu\\ &\quad = \mathit{takefirst}(\mathit{takefirst}\,\epsilon^*\nu)(\mathrm{R}-1)\mathbin{\S}\langle\epsilon\rangle\mathbin{\S}\mathit{dropfirst}(\mathit{takefirst}\,\epsilon^*\nu)\mathrm{R}\end{aligned}$$

Since $\mathrm{R} \leq \nu$, the RHS becomes

$$\mathit{takefirst}\,\epsilon^*(\mathrm{R}-1)\mathbin{\S}\langle\epsilon\rangle\mathbin{\S}\mathit{dropfirst}(\mathit{takefirst}\,\epsilon^*\nu)\mathrm{R}$$

and the LHS becomes

$$\mathit{takefirst}\,\epsilon^*(\mathrm{R}-1)\mathbin{\S}\langle\epsilon\rangle\mathbin{\S}\mathit{takefirst}(\mathit{dropfirst}\,\epsilon^*\mathrm{R})(\nu-\mathrm{R})$$

Use of the following identity completes the proof. The identity

$$\mathit{dropfirst}(\mathit{takefirst}\,\epsilon^*(\nu_0+\nu_1))\nu_0 = \mathit{takefirst}(\mathit{dropfirst}\,\epsilon^*\nu_0)\nu_1$$

can be shown by induction on ν_0. ∎

6.4.4 Compile-Time Environments

Definition 6.11

$$\mathit{compose} = \lambda\rho\gamma\epsilon^*\mathrm{I}.\, \mathsf{islocal}\,\gamma\mathrm{I} \rightarrow \mathit{stackref}\,\epsilon^*(\mathsf{local}\,\gamma\mathrm{I})\ \mathrm{in}\ D, \rho\mathrm{I}$$

The *compose* function combines an environment for `lambda`-bound variables which are on the stack ϵ^* and in the compile-time environment γ with an environment for defined and `letrec`-bound variables ρ. Lemma 6.12 shows that the composition of extending a compile-time environment is equivalent to extending an environment with some items on the stack.

Lemma 6.12

$$\begin{aligned}&\mathit{compose}\,\rho(\mathsf{extends}\,\gamma\mathrm{I}^*\#\epsilon^*)(\epsilon^* \S\, \epsilon_0^*)\\ &\quad = \mathit{extends}(\mathit{compose}\,\rho\gamma(\epsilon^* \S\, \epsilon_0^*))\mathrm{I}^*\epsilon_0^*\end{aligned}$$

Proof: Proved by induction on I^*. The case in which $\mathrm{I}^* = \langle\rangle$ is trivial, so consider the case in which there is at least one identifier in I^* and one value in ϵ_0^*.

$$\begin{aligned}&\mathit{compose}\,\rho(\mathsf{extends}\,\gamma(\langle\mathrm{I}\rangle \S\, \mathrm{I}^*)\#\epsilon^*)(\epsilon^* \S\, \langle\epsilon\rangle \S\, \epsilon_0^*)\\ &\quad = \mathit{compose}\,\rho(\mathsf{extends}(\langle\mathrm{I}, \#\epsilon^* + 1\rangle \S\, \gamma)\mathrm{I}^*(\#\epsilon^* + 1))(\epsilon^* \S\, \langle\epsilon\rangle \S\, \epsilon_0^*)\\ &\quad = \mathit{extends}(\mathit{compose}\,\rho(\langle\mathrm{I}, \#\epsilon^* + 1\rangle \S\, \gamma)(\epsilon^* \S\, \langle\epsilon\rangle \S\, \epsilon_0^*))\mathrm{I}^*\epsilon_0^*\\ &\quad = \mathit{extends}(\mathit{compose}\,\rho\gamma(\epsilon^* \S\, \langle\epsilon\rangle \S\, \epsilon_0^*)[\epsilon\ \mathrm{in}\ D/\mathrm{I}])\mathrm{I}^*\epsilon_0^*\\ &\quad = \mathit{extends}(\mathit{compose}\,\rho\gamma(\epsilon^* \S\, \langle\epsilon\rangle \S\, \epsilon_0^*))(\langle\mathrm{I}\rangle \S\, \mathrm{I}^*)(\langle\epsilon\rangle \S\, \epsilon_0^*)\end{aligned}$$

where step one is by the definition of $\mathsf{extends}$, step two is by use of the induction hypothesis, step three is by use of the definition of *compose*, and the final step is a property of *extends*. ∎

The compile-time environments of interest during compilation contain bindings for a finite number of identifiers. The natural number $\mathsf{size}\,\gamma$ gives the largest stack that matters when composing an environment using the compile-time environment γ. A stack ϵ^* can be truncated to a length of $\mathsf{size}\,\gamma$ without changing the composed environment. Lemma 6.13 formalizes this idea.

Lemma 6.13 $\mathit{compose}\,\rho\gamma\epsilon^* = \mathit{compose}\,\rho\gamma(\mathit{takefirst}\,\epsilon^*(\mathsf{size}\,\gamma))$

Proof: Consider the expression $\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I}$. When $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{false}$, the lemma is trivial, so assume $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{true}$ and assume $\mathsf{local}\,\gamma\mathrm{I} = \nu$. By the definition of size, $\nu \leq \mathsf{size}\,\gamma$. Let $\nu_0 = \mathsf{size}\,\gamma - \nu$.

$$\begin{aligned}
&\mathit{compose}\,\rho\gamma(\mathit{takefirst}\,\epsilon^*(\mathsf{size}\,\gamma))\mathrm{I}\\
&\quad = \mathit{stackref}(\mathit{takefirst}\,\epsilon^*(\mathsf{size}\,\gamma))\nu\\
&\quad = \mathit{takefirst}\,\epsilon^*(\mathsf{size}\,\gamma) \downarrow \nu\\
&\quad = \mathit{takefirst}\,\epsilon^*(\nu_0 + \nu) \downarrow \nu\\
&\quad = (\langle \epsilon^* \downarrow 1\rangle \,\S\, \cdots \,\S\, \langle \epsilon^* \downarrow \nu\rangle \,\S\, \mathit{takefirst}(\mathit{dropfirst}\,\epsilon^*\nu)\nu_0) \downarrow \nu\\
&\quad = \epsilon^* \downarrow \nu\\
&\quad = \mathit{stackref}\,\epsilon^*\nu\\
&\quad = \mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I}
\end{aligned}$$

∎

Theorem 6.14 shows that the correct environment is produced when compiling a `let` expression.

Theorem 6.14 *Assume* $\#\mathrm{I}^* \leq \#\epsilon^*$. *Let* $\nu = \#\epsilon^* - \#\mathrm{I}^*$ *and assume* $\mathsf{size}\,\gamma \leq \nu$ *and* $\mathrm{R} \leq \nu$.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathsf{extends}\,\gamma\mathrm{I}^*\nu)\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho(\mathsf{extends}\,\gamma\mathrm{I}^*\nu)\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

implies

$$\begin{aligned}
&\mathcal{F}(\mathcal{CL}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathit{applicate}\,(\mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!](\mathit{compose}\,\rho\gamma(\mathit{takefirst}\,\epsilon^*\nu)))\\
&\qquad\qquad (\mathit{dropfirst}\,\epsilon^*\nu)\\
&\qquad\qquad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa
\end{aligned}$$

Proof: Expanding the LHS of the equation gives

$$\begin{aligned}
&\mathcal{F}(\mathcal{CL}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathsf{extends}\,\gamma\mathrm{I}^*\nu)\\
&\qquad\qquad \mathrm{R}\\
&\qquad\qquad (\nu + \#\mathrm{I}^*)\\
&\qquad\qquad (\mathsf{isreturn}\,\mathrm{G} \to \mathrm{G}, \mathsf{mkdispose}\,\#\mathrm{I}^*\mathrm{G}))\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho(\mathsf{extends}\,\gamma\mathrm{I}^*\nu)\epsilon^*)\\
&\qquad\quad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}(\mathsf{isreturn}\,\mathrm{G} \to \mathrm{G}, \mathsf{mkdispose}\,\#\mathrm{I}^*\mathrm{G})\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

by definition of $\mathcal{CL}$ and use of an assumption.

Expanding the RHS of the equation gives.

$$\begin{aligned}
&\mathit{applicate}\ (\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{S}\texttt{)}]\!](\mathit{compose}\,\rho\gamma(\mathit{takefirst}\,\nu)\epsilon^*))\\
&\qquad(\mathit{dropfirst}\,\epsilon^*\nu)\\
&\qquad\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa\\
&\quad= \mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}(\mathit{compose}\,\rho\gamma(\mathit{takefirst}\,\epsilon^*\nu))\mathrm{I}^*(\mathit{dropfirst}\,\epsilon^*\nu))\\
&\qquad\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa\\
&\quad= \mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I}^*(\mathit{dropfirst}\,\epsilon^*\nu))\\
&\qquad\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa
\end{aligned}$$

by definition of $\mathcal{L}$ and use of Lemma 6.13. Appealing to Lemma 6.12 shows that the environment on both sides of the equation are equal, so the proof is completed by showing that the continuations on both sides are the same.

Assume $\mathsf{isreturn}\,\mathrm{G} = \mathit{true}$. Let $\mathrm{G} = [\![\texttt{(return)}]\!]$.

$$\begin{aligned}
&\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}(\mathsf{isreturn}\,\mathrm{G} \to \mathrm{G}, \mathsf{mkdispose}\,\#\mathrm{I}^*\mathrm{G})\rho)\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\texttt{(return)}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}\,\mathit{return}\,\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{return}\,\epsilon\epsilon^*\kappa\\
&\quad= \kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}\,\mathit{return}\,\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa
\end{aligned}$$

Assume $\mathsf{isreturn}\,\mathrm{G} = \mathit{false}$.

$$\begin{aligned}
&\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}(\mathsf{isreturn}\,\mathrm{G} \to \mathrm{G}, \mathsf{mkdispose}\,\#\mathrm{I}^*\mathrm{G})\rho)\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}(\mathsf{mkdispose}\,\#\mathrm{I}^*\mathrm{G})\rho)\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathit{dispose}\,\#\mathrm{I}^*(\mathcal{G}[\![\mathrm{G}]\!]\rho))\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*(\#\epsilon^* - \#\mathrm{I}^*))\kappa\\
&\quad= \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon(\mathit{takefirst}\,\epsilon^*\nu)\kappa
\end{aligned}$$

where step three is by use of Theorem 6.10. ▮

6.4.5 Simple Expressions

Before the main theorems are presented, a useful lemma about variable reference is given.

Lemma 6.15

$$get(\mathcal{W}(\mathsf{local}\,\gamma\mathrm{K})\rho)\epsilon^*\kappa = \mathcal{E}[\![\mathrm{K}]\!](compose\,\rho\gamma\epsilon^*)\kappa$$

and

$$\mathsf{islocal}\,\gamma\mathrm{I}\ \textit{implies}\ \ get(\mathcal{W}(\mathsf{local}\,\gamma\mathrm{I})\rho)\epsilon^*\kappa = \mathcal{E}[\![\mathrm{I}]\!](compose\,\rho\gamma\epsilon^*)\kappa$$

Proof:

$$\begin{aligned} & get(\mathcal{W}(\mathsf{local}\,\gamma\mathrm{K})\rho)\epsilon^*\kappa \\ & \quad = get(\mathcal{W}[\![{'}\mathrm{K}]\!])\epsilon^*\kappa \\ & \quad = get(\mathcal{K}[\![\mathrm{K}]\!])\epsilon^*\kappa \\ & \quad = send(\mathcal{K}[\![\mathrm{K}]\!])\kappa \\ & \quad = \mathcal{E}[\![\mathrm{K}]\!](compose\,\rho\gamma\epsilon^*)\kappa \end{aligned}$$

Assume $\mathsf{islocal}\,\gamma\mathrm{I} = true$, and $\mathsf{local}\,\gamma\mathrm{I} = \nu$.

$$\begin{aligned} & get(\mathcal{W}(\mathsf{local}\,\gamma\mathrm{I})\rho)\epsilon^*\kappa \\ & \quad = get(\nu \text{ in } W)\epsilon^*\kappa \\ & \quad = send(stackref\ \epsilon^*\nu)\kappa \\ & \quad = hold(stackref\ \epsilon^*\nu \text{ in } D)\kappa \\ & \quad = hold(compose\,\rho\gamma\epsilon^*\mathrm{I})\kappa \\ & \quad = \mathcal{E}[\![\mathrm{I}]\!](compose\,\rho\gamma\epsilon^*)\kappa \end{aligned}$$

∎

Theorem 6.16 *Assume* $\mathsf{size}\,\gamma \leq \#\epsilon^*$ *and* $\mathrm{R} \leq \#\epsilon^*$.

$$\begin{aligned} & \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa \\ & \quad = \mathcal{E}[\![\mathrm{S}]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon.\,domove\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa \end{aligned}$$

Theorem 6.17 *Assume* $\mathsf{size}\,\gamma \leq \#\epsilon^*$ *and* $\mathrm{R} \leq \#\epsilon^*$.

$$\begin{aligned} & \mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{F})\rho(\epsilon^*\ \S\ unspecs\,\#\mathrm{S}^*)\kappa \\ & \quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{F}]\!]\rho(\epsilon^*\ \S\ \epsilon_0^*)\kappa \end{aligned}$$

Proof: The previous two theorems are proved using simultaneous structural induction on simple expressions.

The case of sequences of simple expressions is proved by induction on the sequence of expressions. When there are no expressions, the result is obvious, so consider the case in which there is at least one expression:

$$\begin{aligned}
&\text{Let F} = \mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + 1 + \#\mathrm{S}^*)\mathrm{F}\\
&\mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}\ \mathrm{S}^*]\!]\gamma(\#\epsilon^* + 1 + \#\mathrm{S}^*)\mathrm{F})\rho(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma(\#\epsilon^* + 1)(\#\epsilon^* + 1 + \#\mathrm{S}^*)(\mathsf{mkignore}\,\mathrm{F}))\\
&\qquad\quad \rho(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\\
&\qquad\qquad \lambda\epsilon.\, \mathit{domove}\,(\#\epsilon^* + 1)\\
&\qquad\qquad\qquad\qquad (\mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho)\\
&\qquad\qquad\qquad\qquad \epsilon(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\, \mathit{domove}\,(\#\epsilon^* + 1)\\
&\qquad\qquad\qquad\qquad (\mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho)\\
&\qquad\qquad\qquad\qquad \epsilon(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa
\end{aligned}$$

where step one is by definition of $\mathcal{CS}^*$, step two is by use of Theorem 6.16 as an induction hypothesis, and step three is by use of Lemma 6.13.

By use of the definition of $\mathcal{E}^*$, the RHS of Theorem 6.17 becomes

$$\begin{aligned}
&\mathcal{E}^*[\![\mathrm{S}\ \mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)(\lambda\epsilon_0^*.\, \mathcal{F}[\![\mathrm{F}]\!]\rho(\epsilon^* \mathbin{\S} \epsilon_0^*)\kappa)\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\, \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\, \mathcal{F}[\![\mathrm{F}]\!]\rho(\epsilon^* \mathbin{\S} \langle\epsilon\rangle \mathbin{\S} \epsilon_0^*)\kappa)
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same.

$$\begin{aligned}
&\lambda\epsilon.\, \mathit{domove}(\#\epsilon^* + 1)(\mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho)\epsilon(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa\\
&\quad = \lambda\epsilon.\, \mathit{move}(\#\epsilon^* + 1)(\mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho)\epsilon(\epsilon^* \mathbin{\S} \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa\\
&\quad = \lambda\epsilon.\, \mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho(\mathit{unspecified}\,\mathrm{in}\,E)(\epsilon^* \mathbin{\S} \langle\epsilon\rangle \mathbin{\S} \mathit{unspecs}\,\#\mathrm{S}^*)\kappa\\
&\quad = \lambda\epsilon.\, \mathit{ignore}(\mathcal{F}[\![\mathrm{F}]\!]\rho)(\mathit{unspecified}\,\mathrm{in}\,E)(\epsilon^* \mathbin{\S} \langle\epsilon\rangle \mathbin{\S} \mathit{unspecs}\,\#\mathrm{S}^*)\kappa\\
&\quad = \lambda\epsilon.\, \mathcal{F}[\![\mathrm{F}]\!]\rho(\mathit{unspecified}\,\mathrm{in}\,E)(\epsilon^* \mathbin{\S} \langle\epsilon\rangle \mathbin{\S} \mathit{unspecs}\,\#\mathrm{S}^*)\kappa\\
&\quad = \lambda\epsilon.\, \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma(\epsilon^* \mathbin{\S} \langle\epsilon\rangle))\lambda\epsilon_0^*.\, \mathcal{F}[\![\mathrm{F}]\!]\rho(\epsilon^* \mathbin{\S} \langle\epsilon\rangle \mathbin{\S} \epsilon_0^*)\kappa\\
&\quad = \lambda\epsilon.\, \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\, \mathcal{F}[\![\mathrm{F}]\!]\rho(\epsilon^* \mathbin{\S} \langle\epsilon\rangle \mathbin{\S} \epsilon_0^*)\kappa
\end{aligned}$$

where step one is by definition of *domove*, step two is by definition of *move*, step three is by definition of mkignore, step four is by definition of *ignore*, step five is by use of Theorem 6.17 as an induction hypothesis, and step six is by use of Lemma 6.13.

Theorem 6.16 is demonstrated by performing an exhaustive case analysis on each type of simple expression.

Case of constants:

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{K}]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad= \mathcal{F}(\mathsf{mkliteral}\,\mathrm{K}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa\\
&\quad= \mathit{literal}(\mathcal{K}[\![\mathrm{K}]\!])(\mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho)\epsilon^*\kappa\\
&\quad= \mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho(\mathcal{K}[\![\mathrm{K}]\!])\epsilon^*\kappa\\
&\quad= \mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)(\mathcal{K}[\![\mathrm{K}]\!])\epsilon^*\kappa\\
&\quad= \mathit{send}(\mathcal{K}[\![\mathrm{K}]\!])\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathcal{E}[\![\mathrm{K}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

Case of variables: Assume $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{true}$. A `lambda`-bound variable has the property that $\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I} \in E$.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{I}]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad= \mathcal{F}(\mathsf{mkcopy}(\mathsf{local}\,\gamma\mathrm{I})(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa\\
&\quad= \mathit{copy}(\mathsf{local}\,\gamma\mathrm{I})(\mathcal{G}(\mathsf{mkmove}\,\mathrm{RG}))\epsilon^*\kappa\\
&\quad= \mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho(\mathit{stackref}\,\epsilon^*(\mathsf{local}\,\gamma\mathrm{I}))\epsilon^*\kappa\\
&\quad= \mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)(\mathit{stackref}\,\epsilon^*(\mathsf{local}\,\gamma\mathrm{I}))\epsilon^*\kappa\\
&\quad= \mathit{send}(\mathit{stackref}\,\epsilon^*(\mathsf{local}\,\gamma\mathrm{I}))\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathit{hold}(\mathit{stackref}\,\epsilon^*(\mathsf{local}\,\gamma\mathrm{I})\ \mathrm{in}\ D)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathit{hold}(\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I})\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathcal{E}[\![\mathrm{I}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

Assume $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{false}$ and $\mathsf{isletrec}\,\mathrm{I} = \mathit{true}$. A `letrec`-bound variable has the property that $\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I} \in E$.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{I}]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad= \mathcal{F}(\mathsf{mkfetch}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa\\
&\quad= \mathit{fetch}(\mathit{lookup}\,\rho\mathrm{I})(\mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho)\epsilon^*\kappa\\
&\quad= \mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho(\rho\mathrm{I} \mid E)\epsilon^*\kappa\\
&\quad= \mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)(\rho\mathrm{I} \mid E)\epsilon^*\kappa\\
&\quad= \mathit{send}(\rho\mathrm{I} \mid E)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathit{hold}(\rho\mathrm{I})\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathit{hold}(\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I})\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathcal{E}[\![\mathrm{I}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

Assume $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{false}$ and $\mathsf{isletrec}\,\mathrm{I} = \mathit{false}$. A defined variable has the property that $\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I} \in L$.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{I}]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathsf{mkload}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa\\
&\quad = \mathit{load}(\mathit{lookup}\,\rho\mathrm{I})(\mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho)\epsilon^*\kappa\\
&\quad = \mathit{load}(\rho\mathrm{I})(\mathit{domove}\,\mathrm{I}(\mathcal{G}[\![\mathrm{G}]\!]\rho))\epsilon^*\kappa\\
&\quad = \mathit{hold}(\rho\mathrm{I})\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad = \mathit{hold}(\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I})\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{I}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

Case of a tail-recursive call: The operands of a combination are evaluated left-to-right and then the operator is evaluated, which means that:

$$\mathcal{E}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\rho\kappa = \mathcal{E}^*[\![\mathrm{S}^*]\!]\rho\lambda\epsilon^*.\,\mathcal{E}[\![\mathrm{S}]\!]\rho\lambda\epsilon.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa$$

The order in which the compiler evaluates the operands is determined by the case of a `let` expression, so for now, assume the operands are all `lambda`-bound variables or constants.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\mathrm{R}\#\epsilon^*[\![\texttt{(return)}]\!])\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma 0\#\epsilon^*(\mathsf{mkcall}(\mathsf{map}(\mathsf{local}\,\gamma)\mathrm{S}^*)))\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{G}(\mathsf{mkcall}(\mathsf{map}(\mathsf{local}\,\gamma)\mathrm{S}^*))\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![(\mathrm{S}\ \mathrm{S}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\texttt{(return)}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S}\ \mathrm{S}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}\,\mathit{return}\,\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S}\ \mathrm{S}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{return}\,\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S}\ \mathrm{S}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon^*.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon^*.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step is justified because `lambda`-bound variables and constants are invariable.

The LHS and the RHS are equal when their continuations are the same.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{G}(\mathsf{mkcall}(\mathsf{map}(\mathsf{local}\,\gamma)\mathrm{S}^*))\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathit{call}(\mathcal{W}^*(\mathsf{map}(\mathsf{local}\,\gamma)\mathrm{S}^*)\rho)\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathit{obtain}(\mathcal{W}^*(\mathsf{map}(\mathsf{local}\,\gamma)\mathrm{S}^*)\rho)\epsilon^*\lambda\epsilon^*.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon^*.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa
\end{aligned}$$

where step three is by an-easy-to-prove lemma that is analogous to Lemma 6.15.

Case of a non-tail-recursive call ($\mathsf{isreturn}\,\mathrm{G} = \mathit{false}$):

$$\begin{aligned}
&\text{Let } \mathrm{F} = \mathcal{CS}[\![(\mathrm{S\ S}^*)]\!]\gamma 0\#\epsilon^*[\![(\texttt{return})]\!]\\
&\mathcal{F}(\mathcal{CS}[\![(\mathrm{S\ S}^*)]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathsf{mkmkcont}\,\#\epsilon^*\mathrm{F}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa\\
&\quad = \mathit{makecont}\,\#\epsilon^*(\mathcal{F}[\![\mathrm{F}]\!]\rho)(\mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho)\epsilon^*\kappa\\
&\quad = \mathit{makecont}\,\#\epsilon^*(\mathcal{F}[\![\mathrm{F}]\!]\rho)(\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho))\epsilon^*\kappa\\
&\quad = \mathcal{F}[\![\mathrm{F}]\!]\rho\epsilon^*\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S\ S}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

where step three is by use of Lemma 6.9.

Case of a `let` expression:

$$\begin{aligned}
&\text{Let } \ \mathrm{F} = \mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G}\\
&\qquad \rho_0 = \mathit{compose}\,\rho\gamma\epsilon^*\\
&\qquad \kappa_0 = \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\mathcal{F}(\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathsf{mkreserve}\,\#\mathrm{I}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{I}^*)\mathrm{F}))\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathsf{mkreserve}\,\#\mathrm{S}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{F}))\rho\epsilon^*\kappa\\
&\quad = \mathit{reserve}\,\#\mathrm{S}^*(\mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{F})\rho)\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{F})\rho(\epsilon^*\ \S\ \mathit{unspec}\,\#\mathrm{S}^*)\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{F}]\!]\rho(\epsilon^*\ \S\ \epsilon_0^*)\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad\qquad \lambda\epsilon_0^*.\,\mathit{applicate}\ (\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](\mathit{compose}\,\rho\gamma\epsilon^*))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \epsilon_0^*\\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!]\rho_0\lambda\epsilon_0^*.\,\mathit{applicate}(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\rho_0)\epsilon_0^*\kappa_0\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!]\rho_0\lambda\epsilon_0^*.\,\mathcal{E}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\rho_0\lambda\epsilon.\,\mathit{applicate}\,\epsilon\epsilon_0^*\kappa_0\\
&\quad = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!]\rho_0\kappa_0\\
&\quad = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

where step one is by the definition of $\mathcal{CS}$, step two uses the fact that a Simple PreScheme program is strongly typed to conclude that $\#\mathrm{I}^* = \#\mathrm{S}^*$, step three is by definition of $\mathsf{mkreserve}$, step five is by use of Theorem 6.17 as an induction hypothesis, step six is by use of Theorem 6.14 and Lemma 6.4 to prove one of the assumptions of Theorem 6.14, step eight uses the definition `lambda` abstraction, and step nine uses the definition of application.

The case of a `begin` expression with one expression is trivial so consider only the case in which there is more than one expression:

$$
\begin{aligned}
&\text{Let } \mathrm{F} = \mathcal{CS}[\![\texttt{(begin } \mathrm{S}^* \ \mathrm{S}_0)]\!]\gamma \mathrm{R}\#\epsilon^* \mathrm{G} \\
&\mathcal{F}(\mathcal{CS}[\![\texttt{(begin } \mathrm{S}\ \mathrm{S}^* \ \mathrm{S}_0)]\!]\gamma \mathrm{R}\#\epsilon^* \mathrm{G})\rho\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma(-1)\#\epsilon^*(\mathsf{mkignore}\,\mathrm{F}))\rho\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}(-1)(\mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho)\epsilon\epsilon^*\kappa
\end{aligned}
$$

The RHS of Theorem 6.16 becomes

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(begin } \mathrm{S}\ \mathrm{S}^* \ \mathrm{S}_0)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\qquad \lambda\epsilon.\,\mathcal{E}[\![\texttt{(begin } \mathrm{S}^* \ \mathrm{S}_0)]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\qquad\qquad\qquad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}
$$

The LHS and the RHS are equal when their continuations are the same.

$$
\begin{aligned}
&\lambda\epsilon.\,\mathit{domove}(-1)(\mathcal{G}(\mathsf{mkignore}\,\mathrm{F})\rho)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathit{domove}(-1)(\mathit{ignore}(\mathcal{F}[\![\mathrm{F}]\!]\rho))\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathit{ignore}(\mathcal{F}[\![\mathrm{F}]\!]\rho)(\mathit{unspecified}\,\mathrm{in}\,E)\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{F}[\![\mathrm{F}]\!]\rho\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{E}[\![\texttt{(begin } \mathrm{S}^* \ \mathrm{S}_0)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}
$$

where the last step is another use of an induction hypothesis.

Case of an `if` expression:

$$
\begin{aligned}
&\text{Let } \mathrm{F}_1 = \mathcal{CS}[\![\mathrm{S}_1]\!]\gamma \mathrm{R}\#\epsilon^*(\mathsf{maybemkbranch}\,\mathrm{G}) \\
&\qquad \mathrm{F}_2 = \mathcal{CS}[\![\mathrm{S}_2]\!]\gamma \mathrm{R}\#\epsilon^*(\mathsf{maybemkbranch}\,\mathrm{G}) \\
&\mathcal{F}(\mathcal{CS}[\![\texttt{(if } \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma \mathrm{R}\#\epsilon^* \mathrm{G})\rho\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma 0\#\epsilon^*(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G}))\rho\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}_0]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{G}(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G})\rho\epsilon\epsilon^*\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(if } \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}_0]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\qquad \lambda\epsilon.\,\epsilon = \mathit{false} \rightarrow \mathcal{E}[\![\mathrm{S}_2]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\qquad\qquad\qquad\qquad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa, \\
&\qquad\qquad\qquad \mathcal{E}[\![\mathrm{S}_1]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}
$$

The LHS and the RHS are equal when their continuations are the same. Assume $\mathrm{G} = [\![\texttt{(return)}]\!]$.

$$
\begin{aligned}
&\lambda\epsilon.\,\mathcal{G}(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G})\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{G}[\![\texttt{(bif } \mathrm{F}_1\ \mathrm{F}_2)]\!]\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathit{jumpfalse}(\mathcal{F}[\![\mathrm{F}_1]\!]\rho)(\mathcal{F}[\![\mathrm{F}_2]\!]\rho)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\epsilon = \mathit{false} \rightarrow \mathcal{F}[\![\mathrm{F}_2]\!]\rho\epsilon^*\kappa, \mathcal{F}[\![\mathrm{F}_1]\!]\rho\epsilon^*\kappa
\end{aligned}
$$

Assume $\mathrm{G} \neq [\![\texttt{(return)}]\!]$.

$$
\begin{aligned}
&\lambda\epsilon.\,\mathcal{G}(\mathsf{mkbranchfalse}\,\mathrm{F}_1\mathrm{F}_2\mathrm{G})\rho\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathcal{G}[\![\texttt{(bwf}\ \mathrm{F}_1\ \mathrm{F}_2\texttt{)}\ \mathrm{G}]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{jumpfalse}(\mathcal{F}(\mathsf{join}\,\mathrm{F}_1\mathrm{G})\rho)(\mathcal{F}(\mathsf{join}\,\mathrm{F}_2\mathrm{G})\rho)\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\epsilon = \mathit{false} \to \mathcal{F}(\mathsf{join}\,\mathrm{F}_2\mathrm{G})\rho\epsilon^*\kappa, \mathcal{F}(\mathsf{join}\,\mathrm{F}_1\mathrm{G})\rho\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\epsilon = \mathit{false} \to \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_2]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa,\\
&\qquad\qquad \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa
\end{aligned}
$$

where the last step is justified by use of Theorem 6.2. Use of the induction hypothesis completes the proof.

Case of unspecified:

$$
\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\texttt{(if \#f \#f)}]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad= \mathcal{F}(\mathsf{mkunspecified}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa\\
&\quad= \mathit{literal}(\mathit{unspecified}\,\mathrm{in}\,E)(\mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho)\epsilon^*\kappa\\
&\quad= \mathcal{G}(\mathsf{mkmove}\,\mathrm{RG})\rho(\mathit{unspecified}\,\mathrm{in}\,E)\epsilon^*\kappa\\
&\quad= \mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)(\mathit{unspecified}\,\mathrm{in}\,E)\epsilon^*\kappa\\
&\quad= \mathit{send}(\mathit{unspecified}\,\mathrm{in}\,E)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathcal{E}[\![\texttt{\#f}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\,\epsilon = \mathit{false} \to \mathit{send}\,(\mathit{unspecified}\,\mathrm{in}\,E)\\
&\qquad\qquad\qquad\qquad\qquad \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa,\\
&\qquad\qquad\quad \mathcal{E}[\![\texttt{\#f}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad= \mathcal{E}[\![\texttt{(if \#f \#f)}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}
$$

Case of `case` expressions:

$$
\begin{aligned}
&\text{Let } \mathrm{F} = \mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu(\mathsf{maybemkbranch}\,\mathrm{G})\\
&\qquad \mathrm{F}^* = \mathrm{F}\,\S\,\mathcal{CC}[\![\texttt{((0)}\ \mathrm{S}_1\texttt{)}\ldots\texttt{((}n-1\texttt{)}\ \mathrm{S}_n\texttt{)}]\!]\gamma\mathrm{R}\nu(\mathsf{maybemkbranch}\,\mathrm{G})\\
&\mathcal{F}(\mathcal{CS}[\![\texttt{(case}\ \mathrm{S}\ \texttt{((0)}\ \mathrm{S}_1\texttt{)}\ldots\texttt{((}n-1\texttt{)}\ \mathrm{S}_n\texttt{)}\ \texttt{(else}\ \mathrm{S}_0\texttt{))}]\!]\gamma\mathrm{R}\nu\mathrm{G})\epsilon^*\kappa\\
&\quad= \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma0\nu(\mathsf{mkselection}\,\mathrm{F}^*\mathrm{G}))\epsilon^*\kappa\\
&\quad= \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{G}(\mathsf{mkselection}\,\mathrm{F}^*\mathrm{G})\rho\epsilon\epsilon^*\kappa
\end{aligned}
$$

The continuation simplifies to

$$
\begin{aligned}
&\lambda\epsilon.\,\mathit{select}(\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle\\
&\qquad\qquad \S\,\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle\\
&\qquad\qquad\quad \vdots\\
&\qquad\qquad \S\,\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle)\epsilon\epsilon^*\kappa
\end{aligned}
$$

by considering two cases for G. Assume $\mathrm{G} = [\![\texttt{(return)}]\!]$.

$$\begin{aligned}
&\lambda\epsilon.\, \mathcal{G}(\mathsf{mkselection}\ \mathrm{F}^*\mathrm{G})\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathit{select}(\mathsf{map}(\lambda\mathrm{F}.\, \mathcal{F}[\![\mathrm{F}]\!]\rho)\mathrm{F}^*)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathit{select}(\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle \S \cdots \S \langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle)\epsilon\epsilon^*\kappa
\end{aligned}$$

Assume $\mathrm{G} \neq [\![\texttt{(return)}]\!]$.

$$\begin{aligned}
&\lambda\epsilon.\, \mathcal{G}(\mathsf{mkselection}\ \mathrm{F}^*\mathrm{G})\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathit{select}(\mathsf{map}(\lambda\mathrm{F}.\, \mathcal{F}(\mathsf{join}\,\mathrm{FG})\rho)\mathrm{F}^*)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathit{select}(\langle\mathcal{F}(\mathsf{join}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu[\![\texttt{(bra)}]\!])\mathrm{G})\rho\rangle \S \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad \S \langle\mathcal{F}(\mathsf{join}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu[\![\texttt{(bra)}]\!])\mathrm{G})\rho\rangle)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathit{select}(\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle \S \cdots \S \langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle)\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step is by use of Theorem 6.2.

Consider applying the continuation to various expressed values. Choose n_0 such that $0 \leq n_0 < n$.

$$\begin{aligned}
&\mathit{select}(\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle \S \cdots \S \langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle)(n_0 \text{ in } E)\epsilon^*\kappa \\
&\quad = (((\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle \S \cdots \S \langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle) \downarrow (2+n_0))\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_{n_0+1}]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa
\end{aligned}$$

For n_0 such that $n_0 < 0$ or $n \leq n_0$,

$$\begin{aligned}
&\mathit{select}(\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle \S \cdots \S \langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle)(n_0 \text{ in } E)\epsilon^*\kappa \\
&\quad = (((\langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle \S \cdots \S \langle\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\rangle) \downarrow 1)\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa
\end{aligned}$$

As a result the continuation can also be written as follows.

$$\begin{aligned}
&\lambda\epsilon.\, \epsilon \mid R = 0 \rightarrow \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa, \\
&\qquad\quad \vdots \\
&\quad \epsilon \mid R = n-1 \rightarrow \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_n]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa, \\
&\quad \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\mathrm{R}\nu\mathrm{G})\rho\epsilon^*\kappa
\end{aligned}$$

A `case` expression is a derived expression which is defined in terms of a sequence of `if` expressions so that

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{case}\ \mathrm{S}\ ((0)\ \mathrm{S}_1)\ldots((n-1)\ \mathrm{S}_n)\ (\texttt{else}\ \mathrm{S}_0))]\!]\rho_0\kappa_0\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!]\rho_0\lambda\epsilon.\,\epsilon \mid R = 0 \to \mathcal{E}[\![\mathrm{S}_1]\!]\rho_0\kappa_0,\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\ \ \epsilon \mid R = n-1 \to \mathcal{E}[\![\mathrm{S}_n]\!]\rho_0\kappa_0,\\
&\qquad\qquad\qquad\ \ \mathcal{E}[\![\mathrm{S}_0]\!]\rho_0\kappa_0\\
&\text{where } \rho_0 = \mathit{compose}\,\rho\gamma\epsilon^*\\
&\qquad\quad \kappa_0 = \lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa
\end{aligned}$$

The use of the induction hypothesis completes the proof of this case.

Case of assignment for identifiers whose name begins and ends with an asterisk and is at least three characters long ($\mathit{ismutable}\,\mathrm{I} = \mathit{true}$):

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!]\gamma\mathrm{R}\#\epsilon^*\mathrm{G})\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma 0\#\epsilon^*(\mathsf{islocal}\,\gamma\mathrm{I} \vee \mathsf{isletrec}\,\mathrm{I} \to \mathsf{mkwrong}\,\mathrm{G},\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{mkstore}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG})))\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\,\mathcal{G}(\mathsf{islocal}\,\gamma\mathrm{I} \vee \mathsf{isletrec}\,\mathrm{I} \to \mathsf{mkwrong}\,\mathrm{G},\\
&\qquad\qquad\qquad\quad \mathsf{mkstore}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{G}[\![\mathrm{G}]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\,\mathit{assign}\,(\mathit{lookup}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I})\\
&\qquad\qquad\qquad\qquad \epsilon\\
&\qquad\qquad\qquad\qquad (\mathit{send}(\mathit{unspecified}\,\mathrm{in}\,E)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa)
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same. Assume $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{true}$ or $\mathsf{isletrec}\,\mathrm{I} = \mathit{true}$ so that $\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I} \in E$.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{G}(\mathsf{islocal}\,\gamma\mathrm{I} \to \mathsf{mkwrong}\,\mathrm{G}, \mathsf{mkstore}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathcal{G}(\mathsf{mkwrong}\,\mathrm{G})\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathit{wrong}\ \text{“assignment of an immutable variable”}\\
&\quad = \lambda\epsilon.\,\mathit{assign}\,(\mathit{lookup}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I})\\
&\qquad\qquad\quad \epsilon\\
&\qquad\qquad\quad (\mathit{send}(\mathit{unspecified}\,\mathrm{in}\,E)\lambda\epsilon.\,\mathcal{G}[\![\mathrm{G}]\!]\rho\epsilon\epsilon^*\kappa)
\end{aligned}$$

Assume $\mathsf{islocal}\,\gamma\mathrm{I} = \mathit{false}$ and $\mathsf{isletrec}\,\mathrm{I} = \mathit{false}$ so that $\mathit{compose}\,\rho\gamma\epsilon^*\mathrm{I} \in L$.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{G}(\mathsf{islocal}\,\gamma\mathrm{I} \to \mathsf{mkwrong}\,\mathrm{G}, \mathsf{mkstore}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathcal{G}(\mathsf{mkstore}\,\mathrm{I}(\mathsf{mkmove}\,\mathrm{RG}))\rho\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{setglobal}(\mathit{lookup}\,\rho\mathrm{I})(\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho))\epsilon\epsilon^*\kappa\\
&\quad= \lambda\epsilon.\,\mathit{assign}(\mathit{lookup}\,\rho\mathrm{I})\epsilon(\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)(\mathit{unspecified}\,\mathrm{in}\,E)\epsilon^*\kappa)\\
&\quad= \lambda\epsilon.\,\mathit{assign}\;(\mathit{lookup}\,\rho\mathrm{I})\\
&\qquad\qquad\qquad\epsilon\\
&\qquad\qquad\qquad(\mathit{send}(\mathit{unspecified}\,\mathrm{in}\,E)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa)\\
&\quad= \lambda\epsilon.\,\mathit{assign}\;(\mathit{lookup}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I})\\
&\qquad\qquad\qquad\epsilon\\
&\qquad\qquad\qquad(\mathit{send}(\mathit{unspecified}\,\mathrm{in}\,E)\lambda\epsilon.\,\mathit{domove}\,\mathrm{R}(\mathcal{G}[\![\mathrm{G}]\!]\rho)\epsilon\epsilon^*\kappa)
\end{aligned}$$

The case of $\mathit{ismutable}\,\mathrm{I} = \mathit{false}$ is similar. ∎

6.4.6 Defined and Letrec-Bound Variables

Lemma 6.18

$$\begin{aligned}
&\mathcal{F}(\mathcal{CL}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\gamma_0 00[\![\texttt{(return)}]\!])\rho\,\mathrm{in}\,E\\
&\quad= \mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\rho
\end{aligned}$$

Proof:

$$\begin{aligned}
&\mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\rho \mid F\\
&\quad= \lambda\epsilon^*\kappa.\,\#\epsilon^* = \#\mathrm{I}^* \to \mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}\,\rho\mathrm{I}^*\epsilon^*)\kappa,\\
&\qquad\qquad\quad \mathit{wrong}\text{ “wrong number of arguments”}\\
&\quad= \lambda\epsilon^*\kappa.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}\,\rho\mathrm{I}^*\epsilon^*)\kappa\\
&\quad= \lambda\epsilon^*\kappa.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}(\mathit{compose}\,\rho\gamma_0\epsilon^*)\mathrm{I}^*\epsilon^*)\kappa\\
&\quad= \lambda\epsilon^*\kappa.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho(\mathsf{extends}\,\gamma_0\mathrm{I}^*0)\epsilon^*)\kappa\\
&\quad= \lambda\epsilon^*\kappa.\,\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathsf{extends}\,\gamma_0\mathrm{I}^*0)0\#\mathrm{I}^*[\![\texttt{(return)}]\!])\rho\epsilon^*\kappa\\
&\quad= \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathsf{extends}\,\gamma_0\mathrm{I}^*0)0\#\mathrm{I}^*[\![\texttt{(return)}]\!])\rho\\
&\quad= \mathcal{F}(\mathcal{CL}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\gamma_0 00[\![\texttt{(return)}]\!])\rho
\end{aligned}$$

where step two is by use of Lemma 6.18 and the fact that Simple PreScheme programs are strongly typed so procedures will always receive the correct number of arguments, step four is by Lemma 6.12, and step five is by Theorem 6.16. ∎

Theorem 6.19 $\mathcal{B}'(\mathcal{CB}[\![\mathrm{B}]\!]) = \mathcal{B}[\![\mathrm{B}]\!]$

Proof: Induction on the length of B proves the theorem. The case when $[\![\mathrm{B}]\!] = \langle\rangle$ is trivial, so consider the following.

$$\begin{aligned}
&\text{Let } \mathrm{F} = \mathsf{mkentry}\,\#\mathrm{I}^*(\mathcal{CL}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!]\gamma_0 00[\![\texttt{(return)}]\!])\\
&\mathcal{B}'(\mathcal{CB}[\![\texttt{(}\mathrm{I}\texttt{ (lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)) }\mathrm{B}]\!])\\
&\quad = \mathcal{B}'([\![\texttt{(}\mathrm{I}\ \mathrm{F}\texttt{)}]\!]\ \S\ \mathcal{CB}[\![\mathrm{B}]\!])\\
&\quad = \lambda\mathrm{I}_0^*\rho\epsilon^*.\,\langle(\mathcal{F}[\![\mathrm{F}]\!](\mathit{extends}\,\rho\mathrm{I}_0^*\epsilon^*))\text{ in }E\rangle\ \S\ \mathcal{B}'(\mathcal{CB}[\![\mathrm{B}]\!])\mathrm{I}_0^*\rho\epsilon^*\\
&\quad = \lambda\mathrm{I}_0^*\rho\epsilon^*.\,\langle(\mathcal{F}[\![\mathrm{F}]\!](\mathit{extends}\,\rho\mathrm{I}_0^*\epsilon^*))\text{ in }E\rangle\ \S\ \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^*\\
&\quad = \lambda\mathrm{I}_0^*\rho\epsilon^*.\,\langle(\mathcal{L}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)}]\!](\mathit{extends}\,\rho\mathrm{I}_0^*\epsilon^*))\text{ in }E\rangle\ \S\ \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^*\\
&\quad = \mathcal{B}[\![\texttt{(}\mathrm{I}\texttt{ (lambda (}\mathrm{I}^*\texttt{) }\mathrm{S}\texttt{)) }\mathrm{B}]\!]
\end{aligned}$$

where step three is by use of the induction hypothesis and step four is by use of Lemma 6.18 and the fact that Simple PreScheme programs are strongly typed so that one can ignore the `entry` instruction. ∎

Theorem 6.20 $\mathcal{E}'(\mathcal{CE}[\![\mathrm{E}]\!])\rho\langle\rangle\kappa = \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa$

$$\begin{aligned}
&\text{Let } \mathrm{B}' = \mathcal{CB}[\![\mathrm{B}]\!]\\
&\qquad \mathrm{F} = \mathsf{mkentry}\,0(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 00[\![\texttt{(return)}]\!])\\
&\mathcal{E}'(\mathcal{CE}[\![\texttt{(letrec (}\mathrm{B}\texttt{) }\mathrm{S}\texttt{)}]\!])\rho\langle\rangle\kappa\\
&\quad = \mathcal{E}'[\![\texttt{(letrec (}\mathrm{B}'\texttt{) }\mathrm{F}\texttt{)}]\!]\rho\langle\rangle\kappa\\
&\quad = \mathcal{F}[\![\mathrm{F}]\!](\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}']\!])(\mathit{fix}(\mathcal{B}'[\![\mathrm{B}']\!](\mathcal{I}'[\![\mathrm{B}']\!])\rho)))\langle\rangle\kappa\\
&\quad = \mathcal{F}[\![\mathrm{F}]\!](\mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho)))\langle\rangle\kappa
\end{aligned}$$

where the last step is justified by Theorem 6.19.

$$\begin{aligned}
&\text{Let } \rho' = \mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho))\\
&\mathcal{F}[\![\mathrm{F}]\!]\rho'\langle\rangle\kappa\\
&\quad = \mathcal{F}(\mathsf{mkentry}\,0(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 00[\![\texttt{(return)}]\!]))\rho'\langle\rangle\kappa\\
&\quad = \mathit{entry}\,0(\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 00[\![\texttt{(return)}]\!])\rho')\langle\rangle\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 00[\![\texttt{(return)}]\!])\rho'\langle\rangle\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho'\gamma_0\langle\rangle)\lambda\epsilon.\,\mathcal{G}[\![\texttt{(return)}]\!]\rho'\epsilon\langle\rangle\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!]\rho'\lambda\epsilon.\,\mathcal{G}[\![\texttt{(return)}]\!]\rho'\epsilon\langle\rangle\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!]\rho'\kappa\\
&\quad = \mathcal{E}[\![\texttt{(letrec (}\mathrm{B}\texttt{) }\mathrm{S}\texttt{)}]\!]\rho\kappa
\end{aligned}$$

where step four is by use of Theorem 6.16.

Theorem 6.21 $\mathcal{D}'(\mathcal{CP}[\![\mathrm{P}]\!]) = \mathcal{D}[\![\mathrm{P}]\!]$

Proof:

$$\begin{aligned}
&\mathcal{D}'(\mathcal{CP}[\![\texttt{(letrec}\ \text{(B) S)}]\!])\rho\kappa \\
&\quad = \mathcal{E}'(\mathcal{CP}[\![\texttt{(letrec}\ \text{(B) S)}]\!])\rho\langle\rangle\kappa \\
&\quad = \mathcal{E}[\![\texttt{(letrec}\ \text{(B) S)}]\!]\rho\kappa \\
&\quad = \mathcal{D}[\![\texttt{(letrec}\ \text{(B) S)}]\!]\rho\kappa
\end{aligned}$$

where step two is by Theorem 6.20.

$$\begin{aligned}
&\mathcal{D}'(\mathcal{CP}[\![\texttt{(define}\ \text{I) P}]\!])\rho\kappa\sigma \\
&\quad = \mathcal{D}'([\![\texttt{(define}\ \text{I)}]\!]\ \S\ \mathcal{CP}[\![\text{P}]\!])\rho\kappa\sigma \\
&\quad = \mathcal{D}'(\mathcal{CP}[\![\text{P}]\!])(\rho[(\textit{new}\ \sigma)\ \text{in}\ D/\text{I}])\kappa(\textit{update}(\textit{new}\ \sigma)(\textit{undefined}\ \text{in}\ E)\sigma) \\
&\quad = \mathcal{D}[\![\text{P}]\!](\rho[(\textit{new}\ \sigma)\ \text{in}\ D/\text{I}])\kappa(\textit{update}(\textit{new}\ \sigma)(\textit{undefined}\ \text{in}\ E)\sigma) \\
&\quad = \mathcal{D}[\![\texttt{(define}\ \text{I) P}]\!]
\end{aligned}$$

where step three is by use of the induction hypothesis. ▮

Theorem 6.22 $\mathcal{P}'(\mathcal{CP}[\![\text{P}]\!]) = \mathcal{P}_0[\![\text{P}]\!]$

Proof: By use of Theorem 6.21.

$$\mathcal{P}'(\mathcal{CP}[\![\text{P}]\!]) = \mathcal{D}'(\mathcal{CP}[\![\text{P}]\!])\rho_0\kappa_0\sigma_0 = \mathcal{D}[\![\text{P}]\!]\rho_0\kappa_0\sigma_0 = \mathcal{P}_0[\![\text{P}]\!]$$

▮

Chapter 7

PreScheme Assembly Language

PreScheme Assembly Language (PAL) is a linear assembly language for a register machine. It is designed to be easily translated into MIPS assembly language so many properties of that machine's assembly language are reflected in PAL [9]. The language is introduced by giving an informal description of an abstract machine which might execute the language. A formal operational semantics follows the introduction.

A mnemonic description for each PAL instruction is given in Figure 7.1. The syntax for specifying an address is given in Figure 7.2; however, this work will assume that addresses are identifiers that refer to relocatable addresses. The PreScheme assembler assumes that registers in the register bank are identified using the integers greater than or equal to -3. The register named by -1 is special in that its value is always Unspecified, and attempts to change its value are silently ignored.

The PAL register machine's state is captured by five terms: a code sequence, a register bank, a control stack, a store, and an environment. The register bank is a sequence of values as is the control stack and the store. There are three types of variables in MTPS programs: defined variables, `letrec`-bound variables, and `lambda`-bound variables. Global variables are bound to locations and all other variables are bound to values. The environment is a function. Given a global or `letrec`-bound identifier, it returns the corresponding location or value (respectively); both of these are allocated statically, leading to more efficient code. The `lambda`-bound variables are in the register bank or in the control stack.

The control stack consists of a sequence of frames each containing a sequence of values. The control stack is designed to grow downward in the

Description	Opcode	Operands
Load Immediate	`li`	destination, immediate
Load Unspecified value	`lu`	destination
Load Address	`la`	destination, address
Load Continuation	`lc`	destination, address
Load Word	`lw`	destination, address
Store Word	`sw`	source, address
Move	`mv`	destination, source
Wrong	`wng`	
Unary operator O	O	destination, src/imm
Binary operator O	O	destination, src/imm, source
		destination, source, src/imm
Branch	`b`	label
Branch on False	`brf`	source, label
Select	`sel`	source, label list
Jump	`j`	address
		source
Push frame	`psh`	frame size
Pop frame	`pop`	frame size
Get from frame	`get`	destination, frame offset
Put into frame	`put`	source, frame offset
Declare labels	`dcl`	label list
Declare Data	`def`	label
Comment	`cmt`	immediate list

Figure 7.1: PAL Instruction Summary

Address	Description
immediate	Specifies an absolute address.
label	Specifies a relocatable address.
(immediate source)	Specifies a based address.
(label source)	Specifies an indexed relocatable address.

Figure 7.2: PAL Addresses

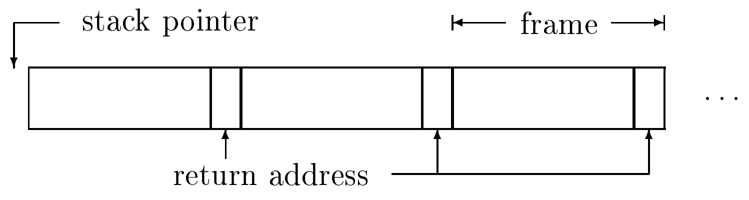

Figure 7.3: Register Machine Stack Layout

direction of smaller addresses as is common in most real machine stacks. Thus, the addresses in the control stack will be represented by negative integers. The stack pointer contains the address one below the newest (most negative) address used by stack. A stack frame using $n+1$ locations is used to store n `lambda`-bound variables. The other location in the frame is used to store a return address. There is no information stored on the stack which records the size of any frame. A control stack is shown in Figure 7.3. In this diagram, the stack grows leftward.

In [16], the arguments to a procedure call are pushed onto the argument stack, and then copied *en masse* to the bottom of the argument stack. In the MTPS compiler, the transformational front end ensures that all arguments to a procedure call are in the register bank by introducing local variables if need be.[1] The arguments are arranged for the call by shuffling register values using code that implements a parallel assignment. Arguments that happen to be in the right place are not moved. Variables added to enable closure hoisting are always in the right place for calls used to implement tight loops.

[1]What if there are more arguments than registers available in the register bank? In our current implementation, the unverified translator PALAS raises an exception if any call requires more registers than are actually available on the real MIPS processor. This is called a PALAS revolt. In practice, however, the MIPS hardware offers enough registers that we have not encountered PALAS revolts. Naturally, a more elaborate mechanism to "spill" registers out into the stack could be added were it to be needed in practice.

7.1 Abstract Syntax

$\mathrm{N} \in \mathrm{Num}$	natural numbers ($\mathrm{Num} = N$)
$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{I} \in \mathrm{Ide}$	identifiers
$\mathrm{R} \in \mathrm{Ref}$	local references (integers)
$\mathrm{J} \in \mathrm{Src}$	local references or quoted constants
$\iota \in \mathrm{Inst}$	machine instructions
$\pi \in \mathrm{Seq}$	machine instruction sequences
$\mathrm{P}'' \in \mathrm{Pgm}''$	PAL programs

Note: in this chapter the variable R will range over the integers greater than or equal to -4.

$$\begin{aligned}
\mathrm{Pgm}'' &\longrightarrow \pi \\
\mathrm{Seq} &\longrightarrow \iota^* \\
\mathrm{Src} &\longrightarrow \mathrm{R} \mid {}'\mathrm{K} \\
\mathrm{Inst} &\longrightarrow \mathrm{I} \mid (\texttt{li}\ \mathrm{R}\ {}'\mathrm{K}) \mid (\texttt{lu}\ \mathrm{R}) \mid (\texttt{la}\ \mathrm{R\ I}) \mid (\texttt{lc}\ \mathrm{R\ I}) \\
&\quad \mid (\texttt{lw}\ \mathrm{R\ I}) \mid (\texttt{sw}\ \mathrm{R\ I}) \mid (\texttt{mv}\ \mathrm{R\ R}) \mid (\texttt{wng}) \\
&\quad \mid (\mathrm{O}_1\ \mathrm{R\ J}) \mid (\mathrm{O}_2\ \mathrm{R\ J\ R}) \mid (\mathrm{O}_2\ \mathrm{R\ R\ J}) \\
&\quad \mid (\texttt{b}\ \mathrm{I}) \mid (\texttt{brf}\ \mathrm{R\ I}) \mid (\texttt{sel}\ \mathrm{R}\ \mathrm{I}^*) \mid (\texttt{j}\ \mathrm{R}) \mid (\texttt{j}\ \mathrm{I}) \\
&\quad \mid (\texttt{psh}\ \mathrm{N}) \mid (\texttt{pop}\ \mathrm{N}) \mid (\texttt{get}\ \mathrm{R\ N}) \mid (\texttt{put}\ \mathrm{R\ N}) \\
&\quad \mid (\texttt{dcl}\ \mathrm{I}^*) \mid (\texttt{def}\ \mathrm{I}) \mid (\texttt{cmt}\ \mathrm{K}^*)
\end{aligned}$$

7.2 PAL Operational Semantics

The set of values manipulated by program P″, denoted by $V_{\mathrm{P}''}$, is defined to be the same as the values manipulated by Stack Assembly Language program (see Section 5.3) except SAL code sequences are replaced by their equivalent PAL code sequences.

The state of the machine has four components:

π	the code register sequence
η	the register bank which maps register names to values
κ	the control stack which maps nonpositive integers to values
σ	the store which maps locations to values
ρ	the environment which maps identifiers to locations or values

For the operational semantics, there is one additional register name: the stack pointer is stored in the register named -4.

7.2.1 Transition Rules

Rule: label
Domain Conditions
$\pi = [\![\mathrm{I}\ \pi_0]\!]$ (7.1)
Changes

Rule: `li`—load literal
Domain Conditions
$\pi = [\![(\texttt{li}\ \mathrm{R}\ '\mathrm{K})\ \pi_0]\!]$
Changes (7.2)
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R} \mapsto \mathrm{K}]$

Rule: `lu`—load unspecified value
Domain Conditions
$\pi = [\![(\texttt{lu}\ \mathrm{R})\ \pi_0]\!]$
Changes (7.3)
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R} \mapsto \mathsf{Unspecified}]$

Rule: `la`—load address
Domain Conditions
$\pi = [\![(\texttt{la}\ \mathrm{R}\ \mathrm{I})\ \pi_0]\!]$
$\epsilon = \rho\mathrm{I}$ (7.4)
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R} \mapsto \epsilon]$

Rule: `lc`—load continuation
Domain Conditions
$\pi = [\![(\texttt{lc}\ \mathrm{R}\ \mathrm{I})\ \pi_0]\!]$
$\epsilon = \mathsf{goto}\,\mathrm{I}\pi_0$ (7.5)
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R} \mapsto \epsilon]$

Rule: `lw`—load word from store
Domain Conditions
$$\begin{aligned}&\pi = [\![(\texttt{lw}\ \mathrm{R}\ \mathrm{I})\ \pi_0]\!]\\&\epsilon = \sigma(\rho\mathrm{I})\end{aligned}\tag{7.6}$$
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R} \mapsto \epsilon]$

Rule: `sw`—store word
Domain Conditions
$$\begin{aligned}&\pi = [\![(\texttt{sw}\ \mathrm{R}\ \mathrm{I})\ \pi_0]\!]\\&\ell = \rho\mathrm{I}\\&\epsilon = \mathsf{rr}\,\eta\mathrm{R}\end{aligned}\tag{7.7}$$
Changes
$\pi' = \pi_0$
$\sigma' = \sigma \diamond [\ell \mapsto \epsilon]$

Rule: `mv`—move between registers
Domain Conditions
$$\begin{aligned}&\pi = [\![(\texttt{mv}\ \mathrm{R}_0\ \mathrm{R}_1)\ \pi_0]\!]\\&\epsilon = \mathsf{rr}\,\eta\mathrm{R}_1\end{aligned}\tag{7.8}$$
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R}_0 \mapsto \epsilon]$

A state with code $\pi = [\![(\texttt{wng})]\!]$ is unproceedable.

Rule: add register
Domain Conditions
$$\begin{aligned}&\pi = [\![(+\ \mathrm{R}_0\ \mathrm{R}_1\ \mathrm{R}_2)\ \pi_0]\!]\\&n = \mathsf{rr}\,\eta\mathrm{R}_1 + \mathsf{rr}\,\eta\mathrm{R}_2\end{aligned}\tag{7.9}$$
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R}_0 \mapsto n]$

Rule: add immediate
Domain Conditions
$\pi = [\![(+\ \mathrm{R}_0\ '\mathrm{K}\ \mathrm{R}_1)\ \pi_0]\!]$
$n = \mathrm{K} + \mathsf{rr}\,\eta\mathrm{R}_1$ (7.10)
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R}_0 \mapsto n]$

Rule: `b`—branch always
Domain Conditions
$\pi = [\![(\mathtt{b}\ \mathrm{I})\ \pi_0]\!]$
$\pi_1 = \mathsf{goto}\,\mathrm{I}\pi_0$ (7.11)
Changes
$\pi' = \pi_1$

Rule: `brf`—conditional branch: true case
Domain Conditions
$\pi = [\![(\mathtt{brf}\ \mathrm{R}\ \mathrm{I})\ \pi_0]\!]$
$\mathsf{rr}\,\eta\mathrm{R} = \mathtt{\#t}$ (7.12)
Changes
$\pi' = \pi_0$

Rule: `brf`—conditional branch: false case
Domain Conditions
$\pi = [\![(\mathtt{brf}\ \mathrm{R}\ \mathrm{I})\ \pi_0]\!]$
$\mathsf{rr}\,\eta\mathrm{R} = \mathtt{\#f}$ (7.13)
Changes
$\pi' = \mathsf{goto}\,\mathrm{I}\pi_0$

Rule: `sel`—select with number in range
Domain Conditions
$\pi = [\![(\mathtt{sel}\ \mathrm{R}\ \mathrm{I}^*)\ \pi_0]\!]$
$n = \mathsf{rr}\,\eta\mathrm{R}$ (7.14)
$0 \leq n < \#\mathrm{I}^*$
Changes
$\pi' = \mathsf{goto}(\mathrm{I}^* \downarrow (n+1))\pi_0$

Rule: `sel`—select with number out of range
Domain Conditions
$\pi = [\![(\texttt{sel}\ \mathrm{R}\ \mathrm{I}^*)\ \pi_0]\!]$
$n = \mathsf{rr}\,\eta\mathrm{R}$ (7.15)
$n < 0 \vee \#\mathrm{I}^* \leq n$
Changes
$\pi' = \pi_0$

Rule: `j`—jump to an address
Domain Conditions
$\pi = [\![(\texttt{j}\ \mathrm{I})\ \pi_0]\!]$ (7.16)
Changes
$\pi' = \rho\mathrm{I}$

Rule: `j`—jump from register
Domain Conditions
$\pi = [\![(\texttt{j}\ \mathrm{R})\ \pi_0]\!]$ (7.17)
Changes
$\pi' = \mathsf{rr}\,\eta\mathrm{R}$

Rule: `psh`—allocate a stack frame
Domain Conditions
$\pi = [\![(\texttt{psh}\ \mathrm{N})\ \pi_0]\!]$
$n = \eta(-4) - \mathrm{N}$ (7.18)
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [-4 \mapsto n]$

where -4 is the stack pointer register.

Rule: `pop`—deallocate a stack frame
Domain Conditions
$\pi = [\![(\texttt{pop}\ \mathrm{N})\ \pi_0]\!]$
$n = \eta(-4) + \mathrm{N}$ (7.19)
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [-4 \mapsto n]$

Rule: `get`—load value from stack
Domain Conditions
$\pi = [\![(\texttt{get}\ \mathrm{R}\ \mathrm{N})\ \pi_0]\!]$
$\epsilon = \kappa(\eta(-4) + \mathrm{N})$ (7.20)
Changes
$\pi' = \pi_0$
$\eta' = \eta \diamond [\mathrm{R} \mapsto \epsilon]$

Rule: `put`—store value in stack
Domain Conditions
$\pi = [\![(\texttt{put}\ \mathrm{R}\ \mathrm{N})\ \pi_0]\!]$
$n = \eta(-4) + \mathrm{N}$
$n \leq 0$ (7.21)
$\epsilon = \mathsf{rr}\,\eta\mathrm{R}$
Changes
$\pi' = \pi_0$
$\kappa = \kappa \diamond [n \mapsto \epsilon]$

Rule: `dcl`—add `letrec` bindings to the environment
Domain Conditions
$\pi = [\![(\texttt{dcl}\ \mathrm{I}^*)\ \pi_0]\!]$ (7.22)
Changes
$\pi' = \pi_0$
$\rho' = \mathsf{augment}\,\rho\mathrm{I}^*\pi_0$

Rule: `def`—add defined variables to the environment
Domain Conditions
$\pi = [\![(\texttt{def}\ \mathrm{I})\ \pi_0]\!]$
$\ell = \mathsf{new}\,\sigma$ (7.23)
Changes
$\pi' = \pi_0$
$\rho' = \rho \diamond [\mathrm{I} \mapsto \ell]$
$\sigma' = \sigma \diamond [\ell \mapsto \mathsf{Undefined}]$

Rule: `cmt`—comment
Domain Conditions
$\pi = [\![(\texttt{cmt}\ \mathrm{K}^*)\ \pi_0]\!]$ (7.24)
Changes
$\pi' = \pi_0$

Register	Description
0	Value register
-1	Unspecified register
-2	Return address register
-3	Spare register for register swapping

Figure 7.4: Dedicated Registers

7.2.2 Auxiliary Functions

$$\mathsf{goto} : \mathrm{Ide} \to \mathrm{Seq} \to \mathrm{Seq}$$
$$\mathsf{goto} = \lambda \mathrm{I}\pi.\, \mathrm{I} = \pi \downarrow 1 \to \pi \dagger 1, \mathsf{goto}\, \mathrm{I}(\pi \dagger 1)$$

The name rr stands for read register.

$$\mathsf{rr} = \lambda \eta \mathrm{R}.\, \mathrm{R} = -1 \to \mathsf{Unspecified}, \eta \mathrm{R}$$

$$\mathsf{augment}\, \rho\langle\rangle\pi = \rho$$
$$\mathsf{augment}\, \rho(\langle \mathrm{I}\rangle \mathbin{\S} \mathrm{I}^*)\pi =$$
$$\quad \mathsf{augment}(\rho \diamond [\mathrm{I} \mapsto \mathsf{goto}\, \mathrm{I}\pi])\mathrm{I}^*\pi$$

7.2.3 Initial and Final States

Registers usage in executing programs will follow a convention. `lambda`-bound variables will be placed in registers identified by positive numbers. Each nonpositive register is dedicated for a specific purpose as summarized in Figure 7.4. The spare register is used while shuffling values for a procedure call. It's function will be revealed in Chapter 8.

An initial state for program P″ has the form:

$$\pi = \mathrm{P}''$$
$$\eta = [-4 \mapsto 0]$$
$$\kappa = [\,]$$
$$\sigma = [\,]$$
$$\rho = [\,]$$

An accepting final state with answer n has the form:

$$\pi = [\![(\mathsf{j}\ -2)\ \pi_0]\!]$$
$$\eta 0 = n$$
$$\eta(-4) = 0$$

where $\eta(-2)$ is not code.

7.3 Relation to the Stack Machine

There are many small differences between the stack machine and the register machine, but two major differences stand out. The most obvious is stack machine code is tree structured and register machine code is linear with segments linked by branch instructions.

The other major difference between the two machines is how they store control information required by a return from a procedure call. In the stack machine, continuations are tree structured. The register machine stores the equivalent information in a control stack, which is a sequence of values, and a stack pointer, which records the location of the stack top.

The stack machine stores and restores all of the elements of the local variable stack when it stores or restores a continuation, however, the register machine may omit storing and restoring some of the registers which contain local variables. A register need not be saved in the control stack when the code to be executed upon return from a procedure call ignores its value.

A stack location R is called *semantically dead* in code Q if the set of accepting stack machine states that are accessible from a state with code Q is not influenced by the value of the stack at location R. Semantically dead stack locations need not be saved in continuations.

Figure 7.5 defines the function $\mathcal{R}$ which is used to approximate from above the set of stack locations which are not semantically dead. A stack location R is called *live* in code Q if $\mathrm{R} \in \mathcal{R}[\![\mathrm{Q}]\!]$. Live locations are those potentially accessed by the code. In addition, the function $\mathcal{R}$ also determines if the value register is live. If $0 \notin \mathcal{R}[\![\mathrm{Q}]\!]$ than code Q is said to ignore the value register. From the definition of $\mathcal{R}$, observe that all value formation instructions ignore the value register, i.e., $\forall \mathrm{F}, 0 \notin \mathcal{R}[\![\mathrm{F}]\!]$.

The live stack location analysis is central to the definition of the relation between the stack machine and the register machine. The relation requires that $\mathcal{R}$ produces sets that are not too small. It is a by-product of the proof of Theorem 7.11 that, if R is not semantically dead in Q, then R is live in Q. The relation also requires that $\mathcal{R}$ produces sets that are not too big.

Lemma 7.1 $\mathrm{N} \rhd \mathrm{Q}$ *implies* $\forall \mathrm{R} \in \mathcal{R}[\![\mathrm{Q}]\!], \mathrm{R} \leq \mathrm{N}$.

Proof sketch: The proof is by induction on the syntax of SAL programs. The proof of each case is straightforward. ∎

$\mathcal{R}[\![(\texttt{lit}\ 'K)\ G]\!] = \mathcal{R}[\![G]\!] \setminus \{0\}$
$\mathcal{R}[\![(\texttt{unspec})\ G]\!] = \mathcal{R}[\![G]\!] \setminus \{0\}$
$\mathcal{R}[\![(\texttt{ignore})\ F]\!] = \mathcal{R}[\![F]\!]$
$\mathcal{R}[\![(\texttt{move}\ R)\ G]\!] = \mathcal{R}[\![G]\!] \cup \{0\} \setminus \{R\}$
$\mathcal{R}[\![(\texttt{copy}\ R)\ G]\!] = \mathcal{R}[\![G]\!] \cup \{R\} \setminus \{0\}$
$\mathcal{R}[\![(\texttt{fetch}\ I)\ G]\!] = \mathcal{R}[\![G]\!] \setminus \{0\}$
$\mathcal{R}[\![(\texttt{load}\ I)\ G]\!] = \mathcal{R}[\![G]\!] \setminus \{0\}$
$\mathcal{R}[\![(\texttt{store}\ I)\ G]\!] = \mathcal{R}[\![G]\!] \cup \{0\}$
$\mathcal{R}[\![(+\ J^*)\ G]\!] = \textsf{regs}\,J^* \cup \mathcal{R}[\![G]\!] \setminus \{0\}$
$\mathcal{R}[\![(\texttt{bif}\ (F_0)\ (F_1))]\!] = \mathcal{R}[\![F_0]\!] \cup \mathcal{R}[\![F_1]\!] \cup \{0\}$
$\mathcal{R}[\![(\texttt{bwf}\ (F_0)\ (F_1))\ G]\!] = \mathcal{R}(\textsf{join}\,F_0G) \cup \mathcal{R}(\textsf{join}\,F_1G) \cup \{0\}$
$\mathcal{R}[\![(\texttt{select}\ (F)^*)]\!] = \{0\} \cup \bigcup_{1\le\nu\le\#F^*} \mathcal{R}(F^* \downarrow \nu)$
$\mathcal{R}[\![(\texttt{pick}\ (F)^*)\ G]\!] = \{0\} \cup \bigcup_{1\le\nu\le\#F^*} \mathcal{R}(\textsf{join}(F^* \downarrow \nu)G)$
$\mathcal{R}[\![(\texttt{call}\ J^*)]\!] = \textsf{regs}\,J^* \cup \{0\}$
$\mathcal{R}[\![(\texttt{return})]\!] = \{0\}$
$\mathcal{R}[\![(\texttt{mkcont}\ N\ F)\ G]\!] = \mathcal{R}[\![F]\!] \cup \mathcal{R}[\![G]\!] \setminus \{0\}$
$\mathcal{R}[\![(\texttt{entry}\ N)\ F]\!] = \mathcal{R}[\![F]\!]$
$\mathcal{R}[\![(\texttt{reserve}\ N)\ F]\!] = \mathcal{R}[\![F]\!]$
$\mathcal{R}[\![(\texttt{dispose}\ N)\ G]\!] = \mathcal{R}[\![G]\!]$
$\mathcal{R}[\![(\texttt{cmt}\ K^*)\ Q]\!] = \mathcal{R}[\![Q]\!]$
$\mathcal{R}[\![(\texttt{letrec}\ (B')\ F)]\!] = \mathcal{R}[\![F]\!]$
$\mathcal{R}[\![(\texttt{define}\ I)\ P']\!] = \mathcal{R}[\![P']\!]$

$\textsf{regs}\langle\rangle = \{\}$
$\textsf{regs}(\langle R\rangle \,\S\, J^*) = \{R\} \cup \textsf{regs}\,J^*$
$\textsf{regs}(\langle 'K\rangle \,\S\, J^*) = \textsf{regs}\,J^*$

Figure 7.5: The Live Stack Location Analysis Function

Stack safe code has the property that stack references in code refer to stack locations that exist in every transition between stack safe states. The lemma states that live stack locations have the same property.

The remaining text in this chapter formally define a storage layout relation between the register machine and the stack machine and present a proof that the register machine simulates the stack machine. The first definition presents the idea of a parallel assignment which is used to refine the notion of procedure call.

Definition 7.2 (Parallel Assignment) *Code segment π implements a parallel assignment of* J^* *iff when* $(\pi \mathbin{\S} \pi', \eta, \kappa, \sigma) \Longrightarrow^* (\pi', \eta', \kappa', \sigma')$ *then*

- *The control stack and the store remain unchanged, i.e.* $\kappa = \kappa'$ *and* $\sigma = \sigma'$.
- *The contents of the value, return address, and stack pointer register remain unchanged, i.e.* $\eta(0) = \eta'(0)$, $\eta(-2) = \eta'(-2)$ *and* $\eta(-4) = \eta'(-4)$.
- *For* $1 \le \mathrm{R} \le \#\mathrm{J}^*$,

$$\eta'\mathrm{R} = \begin{cases} \mathrm{K} & \textit{if } \mathrm{J}^* \downarrow \mathrm{R} = {}'\mathrm{K} \\ \eta\mathrm{R}_0 & \textit{if } \mathrm{J}^* \downarrow \mathrm{R} = \mathrm{R}_0 \end{cases}$$

Code refinement is defined in terms of a relation, denoted by $\simeq_q$, between PAL code and SAL code. Built into the relation is the removal of useless loads of the unspecified value. For instance, a naïve refinement of the `store` instruction is

$$[\![(\texttt{sw}\ 0\ \mathrm{I})\ (\texttt{lu}\ 0)\ \pi]\!] \simeq_q [\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!]$$

when $\pi \simeq_q \mathrm{G}$. However, if G ignores the value register, the load unspecified instruction can be elided without changing the behavior of the code. Therefore, the refinement relation is defined so that the load of the unspecified value appears only when necessary, with the help of the following notation:

$$\pi^{\mathrm{G}} = \begin{cases} \pi & \text{if } 0 \notin \mathcal{R}[\![\mathrm{G}]\!] \\ [\![(\texttt{lu}\ 0)\ \pi]\!] & \text{otherwise} \end{cases}$$

Assuming $\pi \simeq_q \mathrm{G}$, the refinement of the `store` instruction is

$$[\![(\texttt{sw}\ 0\ \mathrm{I})\ \pi^{\mathrm{G}}]\!] \simeq_q [\![(\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!]$$

SAL	PAL
`lit`	`li`
`unspec`	`lu`
`fetch`	`la`
`load`	`lw`
`copy`	`mv`
+	+

Table 7.1: Corresponding SAL and PAL Instructions

For the sake of compactness, we will employ the following notational shorthand that underlines certain stack machine instructions which may or may not be present. The case of

$$[\![(\texttt{brf}\ \mathrm{R\ I})\ \pi]\!] \simeq_q [\![\underline{(\texttt{copy}\ \mathrm{R})}\ (\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$$

is shorthand for the two cases of

$$[\![(\texttt{brf}\ \mathrm{R\ I})\ \pi]\!] \simeq_q [\![(\texttt{copy}\ \mathrm{R})\ (\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$$
$$[\![(\texttt{brf}\ 0\ \mathrm{I})\ \pi]\!] \simeq_q [\![(\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$$

(Recall that, when R occurs in SAL code, $0 < \mathrm{R}$.)

Definition 7.3 (Code Refinement) *The relation $\simeq_q$ is defined recursively by the following conditions:*

1. $[\![\mathrm{I}\ \pi]\!] \simeq_q \mathrm{Q}$ if $\pi \simeq_q \mathrm{Q}$.

2. $[\![(z_P\ 0\ \ldots)\ \pi]\!] \simeq_q [\![(z_S\ \ldots)\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$ where z_S and z_P are corresponding SAL and PAL instructions as given in Table 7.1.

3. $[\![(z_P\ \mathrm{R}\ \ldots)\ \pi^{\mathrm{G}}]\!] \simeq_q [\![(z_S\ \ldots)\ (\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$ where z_S and z_P are as above.

4. $\pi \simeq_q [\![(\texttt{unspec})\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$ and $0 \notin \mathcal{R}[\![\mathrm{G}]\!]$.

5. $\pi \simeq_q [\![(\texttt{ignore})\ \mathrm{F}]\!]$ if $\pi \simeq_q \mathrm{F}$.

6. $[\![(\texttt{sw}\ \mathrm{R\ I})\ \pi^{\mathrm{G}}]\!] \simeq_q [\![\underline{(\texttt{copy}\ \mathrm{R})}\ (\texttt{store}\ \mathrm{I})\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$ where $\mathrm{R} = 0$ when $[\![(\texttt{copy}\ \mathrm{R})]\!]$ is absent.

7. $[\![$`(mv` $\mathrm{R}\ 0)\ \pi^{\mathrm{G}}]\!] \simeq_q [\![$`(move` $\mathrm{R})\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$.

8. $\pi^{\mathrm{G}} \simeq_q [\![$`(move` $\mathrm{R})\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$ and $\mathrm{R} \notin \mathcal{R}[\![\mathrm{G}]\!]$.

9. $[\![$`(b` $\mathrm{I})\ \pi]\!] \simeq_q \mathrm{Q}$ if $\pi_0 \simeq_q \mathrm{Q}$ and $\pi_0 = \mathsf{goto}\,\mathrm{I}\pi$.

10. $[\![$`(brf` $\mathrm{R\ I})\ \pi]\!] \simeq_q [\![\underline{\texttt{(copy}\ \mathrm{R})}$ `(bif` $(\mathrm{F}_0)\ (\mathrm{F}_1))]\!]$ if:

    - $\pi \simeq_q \mathrm{F}_0$.
    - $\pi_1 \simeq_q \mathrm{F}_1$ and $\pi_1 = \mathsf{goto}\,\mathrm{I}\pi$.
    - $\mathrm{R} = 0$ when $[\![$`(copy` $\mathrm{R})]\!]$ is absent.

11. $[\![$`(brf` $\mathrm{R\ I})\ \pi]\!] \simeq_q [\![\underline{\texttt{(copy}\ \mathrm{R})}$ `(bwf` $(\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!]$ if:

    - $\pi \simeq_q \mathrm{F}_2$ and $\mathrm{F}_2 = \mathsf{join}\,\mathrm{F}_0\mathrm{G}$.
    - $\pi_1 \simeq_q \mathrm{F}_3$ and $\mathrm{F}_3 = \mathsf{join}\,\mathrm{F}_1\mathrm{G}$ and $\pi_1 = \mathsf{goto}\,\mathrm{I}\pi$.
    - $\mathrm{R} = 0$ when $[\![$`(copy` $\mathrm{R})]\!]$ is absent.

12. $[\![$`(sel` $\mathrm{R\ I}^*)\ \pi]\!] \simeq_q [\![\underline{\texttt{(copy}\ \mathrm{R})}$ `(select` $(\mathrm{F})^*)]\!]$ if:

    - $\#\mathrm{I}^* + 1 = \#\mathrm{F}^*$.
    - $\pi \simeq_q \mathrm{F}_0$ and $\mathrm{F}_0 = \mathrm{F}^* \downarrow 1$.
    - For $1 \leq \nu \leq \#\mathrm{I}^*$, $\pi_\nu \simeq_q \mathrm{F}_\nu$ where $\mathrm{F}_\nu = \mathrm{F}^* \downarrow (\nu + 1)$ and $\pi_\nu = \mathsf{goto}(\mathrm{I}^* \downarrow \nu)\pi$.
    - $\mathrm{R} = 0$ when $[\![$`(copy` $\mathrm{R})]\!]$ is absent.

13. $[\![$`(sel` $\mathrm{R\ I}^*)\ \pi]\!] \simeq_q [\![\underline{\texttt{(copy}\ \mathrm{R})}$ `(pick` $(\mathrm{F})^*)\ \mathrm{G}]\!]$ if:

    - $\#\mathrm{I}^* + 1 = \#\mathrm{F}^*$.
    - $\pi \simeq_q \mathrm{F}_0$ and $\mathrm{F}_0 = \mathsf{join}(\mathrm{F}^* \downarrow 1)\mathrm{G}$.
    - For $1 \leq \nu \leq \#\mathrm{I}^*$, $\pi_\nu \simeq_q \mathrm{F}_\nu$ where $\mathrm{F}_\nu = \mathsf{join}(\mathrm{F}^* \downarrow (\nu + 1))\mathrm{G}$ and $\pi_\nu = \mathsf{goto}(\mathrm{I}^* \downarrow \nu)\pi$.
    - $\mathrm{R} = 0$ when $[\![$`(copy` $\mathrm{R})]\!]$ is absent.

14. $\pi \,\S\, \pi_0 \simeq_q [\![\underline{\texttt{(fetch}\ \mathrm{I})}$ `(call` $\mathrm{J}^*)]\!]$ if:

    - π implements a parallel assignment of J^* (See Definition 7.2).

- For any π_1, $\pi_0 = [\![(\texttt{j}\ \mathrm{I})\ \pi_1]\!]$ when ($\texttt{fetch}$ I) is present; otherwise, $\pi_0 = [\![(\texttt{j}\ 0)\ \pi_1]\!]$.

15. $[\![(\texttt{j}\ -2)\ \pi]\!] \simeq_q [\![(\texttt{return})]\!]$ for any π.

16. $[\![(\texttt{put}\ -2\ 0)\ (\texttt{lc}\ -2\ \mathrm{I})\ (\texttt{psh}\ \mathrm{N}_0)\ \pi]\!] \simeq_q [\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}]\!]$ if:

    - $\mathrm{R}^* = \mathsf{layout}(\mathcal{R}[\![\mathrm{G}]\!] \setminus \{0\})$, where layout is defined so that R^* is the sorted sequence of registers names that make up the set $\mathcal{R}[\![\mathrm{G}]\!]\setminus\{0\}$.
    - $\mathrm{N}_0 = \#\mathrm{R}^* + 1$.
    - $\pi = \mathsf{saveregs}\,\mathrm{R}^*\pi_0$ and

```
$\mathsf{saveregs}\langle\rangle\pi = \pi$
$\mathsf{saveregs}(\mathrm{R}^* \S \langle\mathrm{R}\rangle)\pi =$
  let
    $\mathrm{N} = 1 + \#\mathrm{R}^*$
  in
    $[\![(\texttt{put}\ \mathrm{R}\ \mathrm{N})]\!] \S \mathsf{saveregs}\,\mathrm{R}^*\pi$
```

    - $\pi_0 \simeq_q \mathrm{F}$.
    - $\pi_1 \simeq_q \mathrm{G}$.
    - $\mathsf{goto}\,\mathrm{I}\pi = \mathsf{restoreregs}\,\mathrm{R}^*[\![(\texttt{pop}\ \mathrm{N}_0)\ (\texttt{get}\ -2\ 0)\ \pi_1]\!]$ and

```
$\mathsf{restoreregs}\langle\rangle\pi = \pi$
$\mathsf{restoreregs}(\mathrm{R}^* \S \langle\mathrm{R}\rangle)\pi =$
  let
    $\mathrm{N} = 1 + \#\mathrm{R}^*$
  in
    $\mathsf{restoreregs}\,\mathrm{R}^*[\![(\texttt{get}\ \mathrm{R}\ \mathrm{N})\ \pi]\!]$
```

17. $\pi \simeq_q [\![(\texttt{entry}\ \mathrm{N})\ \mathrm{F}]\!]$ if $\pi \simeq_q \mathrm{F}$.

18. $\pi \simeq_q [\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}]\!]$ if $\pi \simeq_q \mathrm{F}$.

19. $\pi \simeq_q [\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}]\!]$ if $\pi \simeq_q \mathrm{G}$.

20. $[\![(\texttt{cmt}\ \mathrm{K}^*)\ \pi]\!] \simeq_q [\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{Q}]\!]$ if $\pi \simeq_q \mathrm{Q}$.

21. $[\![(\texttt{wrg})\ \pi]\!] \simeq_q [\![(\texttt{wrong})\ \mathrm{Q}]\!]$ if $\pi \simeq_q \mathrm{Q}$.

22. $[\![(\texttt{dcl}\ \mathrm{I}^*)\ \pi]\!] \simeq_q [\![(\texttt{letrec}\ (\mathrm{B}')\ \mathrm{F})]\!]$ if:

- $\pi \simeq_q \mathrm{F}$ and $\mathrm{I}^* = \mathcal{I}[\![\mathrm{B}']\!]$.
- For each binding $(\mathrm{I}\ \mathrm{F}_0)$ in B', $\pi_0 \simeq_q \mathrm{F}_0$ where $\pi_0 = \mathsf{goto}\,\mathrm{I}\pi$.

23. $[\![(\texttt{def}\ \mathrm{I})\ \pi]\!] \simeq_q [\![(\texttt{define}\ \mathrm{I})\ \mathrm{P}']\!]$ if $\pi \simeq_q \mathrm{P}'$.

Definition 7.4 (Value Refinement) *In a register machine value ϵ refines a stack machine value v, written $\epsilon \simeq_v v$, iff both ϵ and v are not code and are equal, or both both ϵ and v are code, and $\epsilon \simeq_q v$.*

Definition 7.5 (Continuation Refinement) *A register machine value return address, stack pointer, and control stack (ϵ, n, κ) refines a stack safe continuation k, written $(\epsilon, n, \kappa) \simeq_k k$, iff the following conditions hold:*

- *If ϵ is not code, then $(\epsilon, 0, \kappa) \simeq_k \mathsf{Halt}$.*
- $(\pi, n, \kappa) \simeq_k (\mathrm{G}, a, k)$ *iff*
  - $\mathrm{R}^* = \mathsf{layout}(\mathcal{R}[\![\mathrm{G}]\!] \setminus \{0\})$, *where* layout *is defined so that* R^* *is the sorted sequence of registers names that make up the set* $\mathcal{R}[\![\mathrm{G}]\!] \setminus \{0\}$.
  - $\mathrm{N} = 1 + \#\mathrm{R}^*$.
  - $\pi = \mathsf{restoreregs}\,\mathrm{R}^*[\![(\texttt{pop}\ \mathrm{N})\ (\texttt{get}\ {-2}\ 0)\ \pi_0]\!]$.
  - $\pi_0 \simeq_q \mathrm{G}$.
  - $\kappa(n + \mathrm{R}) \simeq_v a \downarrow (\mathrm{R}^* \downarrow \mathrm{R})$ *for* $1 \leq \mathrm{R} \leq \#\mathrm{R}^*$
  - $(\kappa(n + \mathrm{N}), n + \mathrm{N}, \kappa) \simeq_k k$.

Definition 7.6 (Environment Refinement) *A register machine environment ρ refines a stack machine environment u, written $\rho \simeq_u u$ iff the following conditions hold:*

- *The domains of ρ and u are the same.*
- *For all $\ell \in \mathrm{dom}(\rho)$, $\rho\mathrm{I} = u\mathrm{I}$ or $\rho\mathrm{I} \simeq_q u\mathrm{I}$.*

Definition 7.7 (State Refinement) *A register machine state Σ^R refines a stack safe state Σ^S, written $\Sigma^R \simeq \Sigma^S$, iff when $\Sigma^R = (\pi, \eta, \kappa, \sigma, \rho)$ and $\Sigma^S = (\mathrm{Q}, v, a, k, s, u)$ then the following conditions hold:*

- *$\pi \simeq_q \mathrm{Q}$.*

- $\eta(0) \simeq_v v$ *if* $0 \in \mathcal{R}[\![\mathrm{Q}]\!]$.
- $\eta(\mathrm{R}) \simeq_v a \downarrow \mathrm{R}$ *for all* $\mathrm{R} \in \mathcal{R}[\![\mathrm{Q}]\!] \setminus \{0\}$.
- $(\eta(-2), \eta(-4), \kappa) \simeq_k k$.
- $\mathrm{dom}(\sigma) = \mathrm{dom}(s)$ *and for all* $\ell \in \mathrm{dom}(\sigma)$, $\sigma\ell \simeq_v s\ell$.
- $\rho \simeq_u u$.

Theorem 7.8 (Transition Simulation) *For stack safe state* Σ_0^S *assume that:*

- $\Sigma_0^S \Longrightarrow^* \Sigma_1^S$.
- $\Sigma_0^R \simeq \Sigma_0^S$.

Then there exists an Σ_1^R *such that:*

- $\Sigma_0^R \Longrightarrow^* \Sigma_1^R$.
- $\Sigma_1^R \simeq \Sigma_1^S$.

Proof sketch: A complete proof of this lemma would show the result for each clause defining the code refinement relation. Five representative cases are show below.

Case of $[\![\texttt{(li}\ 0\ \text{'K)}\ \pi]\!] \simeq_q [\![\texttt{(lit}\ \text{'K)}\ \mathrm{G}]\!]$:

$$\begin{aligned} \text{Let } \Sigma_0^S &= ([\![\texttt{(lit}\ \text{'K)}\ \mathrm{G}]\!], v, a, k, s, u) \\ \Sigma_0^R &= ([\![\texttt{(li}\ 0\ \text{'K)}\ \pi]\!], \eta, \kappa, \sigma, \rho) \end{aligned}$$

By the respective operational semantics:

$$\begin{aligned} \Sigma_1^S &= (\mathrm{G}, \mathrm{K}, a, k, s, u) \\ \Sigma_1^R &= (\pi, \eta \diamond [0 \mapsto \mathrm{K}], \kappa, \sigma, \rho) \end{aligned}$$

Since $\Sigma_0^R \simeq \Sigma_0^S$, then $[\![\texttt{(li}\ 0\ \text{'K)}\ \pi]\!] \simeq_q [\![\texttt{(lit}\ \text{'K)}\ \mathrm{G}]\!]$. Because $\simeq_q$ is defined to be the least set which satisfies the conditions, one can infer that $\pi \simeq_q \mathrm{G}$. The operational semantics preserves stack safety, so $\#a \triangleright \mathrm{G}$. Furthermore, given that $\eta \diamond [0 \mapsto \mathrm{K}](0) = \mathrm{K}$ and other items remain unchanged, $\Sigma_1^R \simeq \Sigma_1^S$.

Case of $[\![\texttt{(+}\ \mathrm{R}_0\ \mathrm{R}_1\ \mathrm{R}_2\texttt{)}\ \pi^{\mathrm{G}}]\!] \simeq_q [\![\texttt{(+}\ \mathrm{R}_1\ \mathrm{R}_2\texttt{)}\ \texttt{(move}\ \mathrm{R}_0\texttt{)}\ \mathrm{G}]\!]$:

$$\begin{aligned} \text{Let } \Sigma_0^S &= ([\![\texttt{(+}\ \mathrm{R}_1\ \mathrm{R}_2\texttt{)}\ \texttt{(move}\ \mathrm{R}_0\texttt{)}\ \mathrm{G}]\!], v, a, k, s, u) \\ \Sigma_0^R &= ([\![\texttt{(+}\ \mathrm{R}_0\ \mathrm{R}_1\ \mathrm{R}_2\texttt{)}\ \pi^{\mathrm{G}}]\!], \eta, \kappa, \sigma, \rho) \end{aligned}$$

Since Σ_0^S is stack safe, $\mathrm{R}_0 \leq \#a$, $\mathrm{R}_1 \leq \#a$, and $\mathrm{R}_2 \leq \#a$. The only transition allowed for `+` requires that $a\mathrm{R}_1$ and $a\mathrm{R}_2$ be numbers, so

$$\Sigma_1^S = (\mathrm{G}, \mathsf{Unspecified}, a_0, k, s, u)$$

where $a\mathrm{R}_1 = n_1$, $a\mathrm{R}_2 = n_2$, $n_0 = n_1 + n_2$, and a_0 is the same as a except at R_0, $a \downarrow \mathrm{R}_0 = n_0$. Since $\Sigma_0^R \simeq \Sigma_0^S$, the code refinement relation allows one to conclude that $\pi \simeq_q \mathrm{G}$ and that $\eta\mathrm{R}_1 \simeq_v n_1$ and $\eta\mathrm{R}_2 \simeq_v n_2$. The register machine operational semantics gives:

$$\Sigma_1^R = (\pi^{\mathrm{G}}, \eta \diamond [\mathrm{R}_0 \mapsto n_0], \kappa, \sigma, \rho)$$

If G ignores the value register, then simple calculation shows that $\Sigma_1^R \simeq \Sigma_1^S$. If G uses the value register, register 0 of state Σ_1^R may not refine the value register, however, in this case, the first instruction in π^{G} must be $[\![$`(lu` 0)$]\!]$, so the next state of the register machine will refine Σ_1^S.

Case of $[\![$(`brf` R I) $\pi]\!] \simeq_q [\![$(`copy` R) (`bwf` (F_0) (F_1)) G$]\!]$:

Let $\Sigma_0^S = ([\![$(`copy` R) (`bwf` (F_0) (F_1)) G$]\!], v, a, k, s, u)$
$\Sigma_0^R = ([\![$(`brf` R I) $\pi]\!], \eta, \kappa, \sigma, \rho)$

Since Σ_0^S is stack safe, $\mathrm{R} \leq \#a$. The only transition allowed by `bwf` requires that $a\mathrm{R}$ is a boolean. Assume $a\mathrm{R} =$ `#f`. The proof for the case of $a\mathrm{R} =$ `#t` easily follows. Since $\mathrm{R} \in \mathcal{R}[\![$(`copy` R) $\ldots]\!]$ and $\Sigma_0^R \simeq \Sigma_0^S$, $\eta\mathrm{R} =$ `#f` and $\pi_0 \simeq_q \mathrm{F}_2$ where $\mathrm{F}_2 = \mathsf{join}\,\mathrm{F}_1\mathrm{G}$ and $\pi_0 = \mathsf{goto}\,\mathrm{I}\pi$. By the respective operational semantics:

$$\Sigma_1^S = (\mathrm{F}_2, \mathsf{Unspecified}, a, k, s, u)$$
$$\Sigma_1^R = (\pi_0, \eta, \kappa, \sigma, \rho)$$

Since F_2 ignores the value register, $\Sigma_1^R \simeq \Sigma_1^S$.

Case of $[\![\pi$ (`j` I) $\ldots]\!] \simeq_q [\![$(`fetch` I) (`call` J*)$]\!]$:

Let $\Sigma_0^S = ([\![$(`fetch` I) (`call` J*)$]\!], v, a, k, s, u)$
$\Sigma_0^R = ([\![\pi$ (`j` I) $\ldots]\!], \eta, \kappa, \sigma, \rho)$

where code segment π implements a parallel assignment of J^*. No transition occurs unless $\mathrm{I} \in \mathrm{dom}(u)$, $u\mathrm{I} = [\![$(`entry` N) F$]\!]$, and $\#\mathrm{J}^* = \mathrm{N}$. The operational semantics gives:

$$\Sigma_1^S = (\mathrm{F}, \mathsf{Unspecified}, a_0, k, s, u)$$

where $a_0 = \mathsf{shuffle}\,\mathrm{J}^* a$. Since $\Sigma_0^R \simeq \Sigma_0^S$, $\rho \simeq_u u$, implying that $\rho\mathrm{I} = \pi_0$ with $\pi_0 \simeq_q \mathrm{F}$. The register machine operational semantics gives:

$$\Sigma_1^R = (\pi_0, \eta_0, \kappa, \sigma, \rho)$$

where η_0 is the result of the parallel assignment. Since Σ_1^S is stack safe, $\forall \mathrm{R} \in \mathcal{R}[\![\mathrm{F}]\!], \mathrm{R} \leq \#a_0$. Comparison of Lemma 5.2 for $\mathsf{shuffle}$ and Definition 7.2 of parallel assignment allows one to conclude that for $1 \leq \mathrm{R} \leq \#a_0$, $a_0 \downarrow \mathrm{R} \simeq_v \eta_0\mathrm{R}$. Since F ignores the value register, $\Sigma_1^R \simeq \Sigma_1^S$.

Case of $[\![\texttt{(j } -2\texttt{)}\ \ldots]\!] \simeq_q [\![\texttt{(return)}]\!]$:

$$\begin{aligned} \text{Let } \Sigma_0^S &= ([\![\texttt{(return)}]\!], v, a_0, (\mathrm{G}, a, k), s, u) \\ \Sigma_0^R &= ([\![\texttt{(j } -2\texttt{)}\ \ldots]\!], \eta, \kappa, \sigma, \rho) \\ \Sigma_1^S &= (\mathrm{G}, v, a, k, s, u) \end{aligned}$$

where Σ_1^S is derived from the operational semantics so that $\Sigma_0^S \Longrightarrow^* \Sigma_1^S$. Since $\Sigma_0^R \simeq \Sigma_0^S$, $(\eta(-2), \eta(-4), \kappa) \simeq_k (\mathrm{G}, a, k)$, and therefore,

- $\mathrm{R}^* = \mathsf{layout}(\mathcal{R}[\![\mathrm{G}]\!] \setminus \{0\})$.
- $\eta(-4) = n$, $\mathrm{N} = 1 + \#\mathrm{R}^*$, and $n + \mathrm{N} \leq 0$.
- $\eta(-2) = \mathsf{restoreregs}\,\mathrm{R}^*[\![\texttt{(pop N) (get } -2\ 0\texttt{)}\ \pi]\!]$.
- $\pi \simeq_q \mathrm{G}$.
- $\kappa(n + \mathrm{R}) \simeq_v a \downarrow (\mathrm{R}^* \downarrow \mathrm{R})$ for $1 \leq \mathrm{R} \leq \#\mathrm{R}^*$
- $(\kappa(n + \mathrm{N}), n + \mathrm{N}, \kappa) \simeq_k k$.

The restore register code sequence produces an η_0 which is the same as η except for $1 \leq \mathrm{R} \leq \#\mathrm{R}^*$, $\eta_0(\mathrm{R}^* \downarrow \mathrm{R}) = \kappa(n + \mathrm{R})$. Executing the `pop` and `get` instructions yield:

$$\Sigma_1^R = (\pi, \eta_0 \diamond [-4 \mapsto n + \mathrm{N}] \diamond [-2 \mapsto \kappa(n + \mathrm{N})], \kappa, \sigma, \rho)$$

Calculation shows that $\Sigma_1^R \simeq \Sigma_1^S$. ∎

Lemma 7.9 *If Σ^S is an initial state for syntax directed compiler generated code, then there exists an initial register machine state Σ^R such that $\Sigma^R \simeq \Sigma^S$.*

Proof: A register machine which satisfies the desired conditions can be produced as follows. The register machine code can be produced from stack machine code using the function $\mathcal{GP}$ defined in Chapter 8. Theorem 8.6 shows this code refines the stack machine code. Pick a noncode value for the return code register, say $\eta(-2) = 0$. Then the desired initial state is obvious from its definition. ▮

Lemma 7.10 *If Σ^S is an accepting state and $\Sigma^R \simeq \Sigma^S$ then Σ^R is an accepting state.*

Proof: Obvious from the definitions of accepting states. ▮

Theorem 7.11 (Simulation) *Assume that:*

- $\Sigma_0^S \Longrightarrow^* \Sigma_1^S$.
- Σ_0^S *is an initial state for syntax directed compiler generated code.*
- Σ_1^S *is a final stack machine state with answer n.*

Then there exists register machine states Σ_0^R and Σ_1^R such that:

- $\Sigma_0^R \simeq \Sigma_0^S$.
- $\Sigma_1^R \simeq \Sigma_1^S$.
- $\Sigma_0^R \Longrightarrow^* \Sigma_1^R$.
- Σ_0^R *is an initial register machine state.*
- Σ_1^R *is a final register machine state with answer n.*

Proof: Lemma 7.9 shows the existence of the initial register machine state, Theorem 7.8 shows that the register machine simulates transitions, and Lemma 7.10 shows the accepting states match. ▮

Chapter 8

Code Generator

The code generator produces Pre-Scheme Assembly Language from Stack Assembly Language. In this chapter, the translation algorithm is displayed and the output of the generator is shown to refine its input.

The code generator shows the value of implementing a prototype before attempting to formally specify and verify the algorithm. The code which implements the algorithm drove the form of the specification. The code is imperative in that it outputs instructions as it constructs them. A global counter records the number of instructions emitted. Requests to create a label use the emitted instruction count to ensure the generation of unique labels. The instruction count is incremented as a side-effect of emitting an instruction.

The code generator algorithm is specified as a function which takes SAL code to be translated and a sequence of PAL code already generated, and appends newly generated code onto that sequence. The previously generated PAL code is only used to obtain emitted instruction counts for the purposes generating labels.

8.1 Definition of the Code Generator

$$
\begin{aligned}
\mathcal{GU} &: \mathrm{User} \to \mathrm{Src} \to \mathrm{Seq} \to \mathrm{Seq}\\
\mathcal{GQ} &: \mathrm{Code} \to \mathrm{Seq} \to \mathrm{Seq}\\
\mathcal{GC} &: \mathrm{Cls} \to \mathrm{Ide}^{*} \to \mathrm{Seq} \to \mathrm{Seq}\\
\mathcal{GA} &: \mathrm{Cls} \to \mathrm{Ide}^{*} \to \mathrm{Ide} \to \mathrm{Seq} \to \mathrm{Seq}\\
\mathcal{GB} &: \mathrm{Bnd}' \to \mathrm{Seq} \to \mathrm{Seq}\\
\mathcal{GP} &: \mathrm{Pgm}' \to \mathrm{Seq}
\end{aligned}
$$

The function $\mathcal{GU}$ is used to insert the unspecified value into the value register after `move` instructions.

$$\mathcal{GU}[\![\mathrm{G}]\!] = \lambda \mathrm{R}\pi.\, \mathcal{GQ}[\![\mathrm{G}]\!](\mathrm{R} = 0 \vee 0 \notin \mathcal{R}[\![\mathrm{G}]\!] \rightarrow \pi, [\![\pi\ (\texttt{lu}\ 0)]\!])$$

The function $\mathcal{GQ}$ is the main function. The code generation algorithm is specified by showing how patterns of SAL instructions translate into PAL instructions. Underlined instructions in patterns refer to optional instructions. Optional instructions usually contain a local reference R. When an optional instruction is absent, the local reference is taken to be zero (R = 0). Written without the use of this convention, the two cases for the `lit` instruction are:

$$\mathcal{GQ}[\![(\texttt{lit}\ '\mathrm{K})\ (\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}[\![\pi\ (\texttt{li}\ \mathrm{R}\ '\mathrm{K})]\!]$$
$$\mathcal{GQ}[\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]0[\![\pi\ (\texttt{li}\ 0\ '\mathrm{K})]\!] \text{ otherwise.}$$

The algorithm specified using the convention follows.

$$\mathcal{GQ}[\![(\texttt{lit}\ '\mathrm{K})\ \underline{(\texttt{move}\ \mathrm{R})}\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}[\![\pi\ (\texttt{li}\ \mathrm{R}\ '\mathrm{K})]\!]$$

$$\begin{aligned}
&\mathcal{GQ}[\![(\texttt{unspec})\ \underline{(\texttt{move}\ \mathrm{R})}\ \mathrm{G}]\!] \\
&\quad = \lambda\pi.\, \mathrm{let} \\
&\qquad\qquad \pi_0 = \mathrm{R} > 0 \rightarrow [\![\pi\ (\texttt{lu}\ \mathrm{R})]\!], \pi \\
&\qquad\quad \mathrm{in} \\
&\qquad\qquad \mathcal{GQ}[\![\mathrm{G}]\!](0 \notin \mathcal{R}[\![\mathrm{G}]\!] \rightarrow \pi_0, [\![\pi_0\ (\texttt{lu}\ 0)]\!])
\end{aligned}$$

$$\mathcal{GQ}[\![(\texttt{ignore})\ \mathrm{F}]\!] = \mathcal{GQ}[\![\mathrm{F}]\!]$$

$$\mathcal{GQ}[\![(\texttt{copy}\ \mathrm{R}_0)\ \underline{(\texttt{move}\ \mathrm{R}_1)}\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}_1[\![\pi\ (\texttt{mv}\ \mathrm{R}_1\ \mathrm{R}_0)]\!]$$

$$\mathcal{GQ}[\![(\texttt{fetch}\ \mathrm{I})\ \underline{(\texttt{move}\ \mathrm{R})}\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}[\![\pi\ (\texttt{la}\ \mathrm{R}\ \mathrm{I})]\!]$$

$$\mathcal{GQ}[\![(\texttt{load}\ \mathrm{I})\ \underline{(\texttt{move}\ \mathrm{R})}\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}[\![\pi\ (\texttt{lw}\ \mathrm{R}\ \mathrm{I})]\!]$$

$$\begin{aligned}
&\mathcal{GQ}[\![\underline{(\texttt{copy}\ \mathrm{R}_0)}\ (\texttt{store}\ \mathrm{I})\ \underline{(\texttt{move}\ \mathrm{R}_1)}\ \mathrm{G}]\!] = \\
&\quad \lambda\pi.\, \mathcal{GQ}[\![(\texttt{unspec})\ \underline{(\texttt{move}\ \mathrm{R}_1)}\ \mathrm{G}]\!][\![\pi\ (\texttt{sw}\ \mathrm{R}_0\ \mathrm{I})]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathcal{GQ}[\![(+\ \mathrm{J}_0\ \mathrm{J}_1)\ \underline{(\texttt{move}\ \mathrm{R})}\ \mathrm{G}]\!] = \\
&\quad \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}[\![\pi\ (+\ \mathrm{R}\ \mathrm{J}_0\ \mathrm{J}_1)]\!]
\end{aligned}$$

$$\mathcal{GQ}[\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!] = \lambda\pi.\, \mathcal{GU}[\![\mathrm{G}]\!]\mathrm{R}[\![\pi\ (\texttt{mv}\ \mathrm{R}\ 0)]\!]$$

$\mathcal{GQ}[\![(\texttt{bra})]\!] = \lambda\pi.\,\pi$

```
$\mathcal{GQ}[\![\underline{(\texttt{copy}\ \mathrm{R})}\ (\texttt{bif}\ (\mathrm{F}_0)\ (\mathrm{F}_1))]\!] =$
  $\lambda\pi.$ let
        $\mathrm{I} = \mathsf{mklabel}\,\#\pi$
      in
        $\mathcal{GQ}[\![\mathrm{F}_1]\!](\mathcal{GQ}[\![\mathrm{F}_0]\!][\![\pi\ (\texttt{brf}\ \mathrm{R}\ \mathrm{I})]\!]\ \S\ [\![\mathrm{I}]\!])$
```

```
$\mathcal{GQ}[\![\underline{(\texttt{copy}\ \mathrm{R})}\ (\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!] =$
  $\lambda\pi.$ let
        $\mathrm{I}_0 = \mathsf{mklabel}\,\#\pi$
        $\pi_0 = \mathcal{GQ}[\![\mathrm{F}_0]\!][\![\pi\ (\texttt{brf}\ \mathrm{R}\ \mathrm{I}_0)]\!]$
        $\mathrm{I}_1 = \mathsf{mklabel}\,\#\pi_0$
      in
        $\mathcal{GQ}[\![\mathrm{G}]\!](\mathcal{GQ}[\![\mathrm{F}_1]\!][\![\pi_0\ (\texttt{b}\ \mathrm{I}_1)\ \mathrm{I}_0]\!]\ \S\ [\![\mathrm{I}_1]\!])$
```

```
$\mathcal{GQ}[\![\underline{(\texttt{copy}\ \mathrm{R})}\ (\texttt{select}\ (\mathrm{F})^*)]\!] =$
  $\lambda\pi.$ let
        $\mathrm{I}^* = \mathsf{mkmanylabels}(\#\mathrm{F}^* - 1)\#\pi$
      in
        $\mathcal{GC}(\mathrm{F}^* \dagger 1)\mathrm{I}^*(\mathcal{GQ}(\mathrm{F}^* \downarrow 1)[\![\pi\ (\texttt{sel}\ \mathrm{R}\ \mathrm{I}^*)]\!])$
```

```
$\mathcal{GQ}[\![\underline{(\texttt{copy}\ \mathrm{R})}\ (\texttt{pick}\ (\mathrm{F})^*)\ \mathrm{G}]\!] =$
  $\lambda\pi.$ let
        $\mathrm{I}^* = \mathsf{mkmanylabels}(\#\mathrm{F}^* - 1)\#\pi$
        $\pi_0 = \mathcal{GQ}(\mathrm{F}^* \downarrow 1)[\![\pi\ (\texttt{sel}\ \mathrm{R}\ \mathrm{I}^*)]\!]$
        $\mathrm{I} = \mathsf{mklabel}\,\#\pi_0$
        $\pi_1 = \mathcal{GA}(\mathrm{F}^* \dagger 1)\mathrm{I}^*\mathrm{I}\pi_0$
      in
        $\mathcal{GQ}[\![\mathrm{G}]\!][\![\pi_1\ \mathrm{I}]\!]$
```

$\mathcal{GQ}[\![\underline{(\texttt{fetch}\ \mathrm{I})}\ (\texttt{call}\ \mathrm{J}^*)]\!] = \lambda\pi.\,\mathsf{mkcallconsts}\,\mathrm{J}^*(\mathsf{mkcallrefs}\,\mathrm{J}^*\pi)\ \S\ \pi_0$

where $\pi_0 = [\![(\texttt{j}\ \mathrm{I})]\!]$ when `fetch` is present, otherwise $\pi_0 = [\![(\texttt{j}\ 0)]\!]$.

$\mathcal{GQ}[\![(\texttt{return})]\!] = \lambda\pi.\,[\![\pi\ (\texttt{j}\ -2)]\!]$

$\mathcal{GQ}[\![(\texttt{mkcont}\ \mathrm{N}\ \mathrm{F})\ \mathrm{G}]\!] =$

```
$\lambda\pi.\,\mathrm{let}$
        $\mathrm{I} = \mathsf{mklabel}\,\#\pi$
        $\phi = \mathcal{R}[\![\mathrm{G}]\!] \setminus \{0\}$
        $\mathrm{N}_0 = 1 + |\phi|$
        $\pi_0 = [\![\pi\ (\texttt{put}\ {-2}\ 0)\ (\texttt{lc}\ {-2}\ \mathrm{I})]\!]$
        $\pi_1 = \mathcal{GQ}[\![\mathrm{F}]\!](\mathsf{saveregs}\,\phi[\![\pi_0\ (\texttt{psh}\ \mathrm{N}_0)]\!])$
        $\pi_2 = \mathsf{restoreregs}\,\phi[\![\pi_1\ \mathrm{I}]\!]\ \S\ [\![(\texttt{pop}\ \mathrm{N}_0)]\!]$
      in
        $\mathcal{GQ}[\![\mathrm{G}]\!][\![\pi_2\ (\texttt{get}\ {-2}\ 0)]\!]$
```

$\mathcal{GQ}[\![(\texttt{entry}\ \mathrm{N})\ \mathrm{F}]\!] = \mathcal{GQ}[\![\mathrm{F}]\!]$

$\mathcal{GQ}[\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{F}]\!] = \mathcal{GQ}[\![\mathrm{F}]\!]$

$\mathcal{GQ}[\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{G}]\!] = \mathcal{GQ}[\![\mathrm{G}]\!]$

$\mathcal{GQ}[\![(\texttt{cmt}\ \mathrm{K}^*)\ \mathrm{Q}]\!] = \lambda\pi.\,\mathcal{GQ}[\![\mathrm{Q}]\!][\![\pi\ (\texttt{cmt}\ \mathrm{K}^*)]\!]$

$\mathcal{GQ}[\![(\texttt{letrec}\ (\mathrm{B}')\ \mathrm{F})]\!] =$

```
$\lambda\pi.\,\mathrm{let}$
        $\mathrm{I}^* = \mathcal{I}[\![\mathrm{B}']\!]$
      in
        $\mathcal{GB}[\![\mathrm{B}']\!](\mathcal{GQ}[\![\mathrm{F}]\!][\![\pi\ (\texttt{dcl}\ \mathrm{I}^*)]\!])$
```

$\mathcal{GQ}[\![(\texttt{define}\ \mathrm{I})\ \mathrm{P}']\!] = \lambda\pi.\,\mathcal{GQ}[\![\mathrm{P}']\!][\![\pi\ (\texttt{def}\ \mathrm{I})]\!]$

The function $\mathcal{GC}$ processes clauses from the `select` instruction.

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

$\mathcal{GC}[\![(\mathrm{F})\ (\mathrm{F})^*]\!](\langle\mathrm{I}\rangle\ \S\ \mathrm{I}^*) = \lambda\pi.\,\mathcal{GC}[\![(\mathrm{F})^*]\!]\mathrm{I}^*(\mathcal{GQ}[\![\mathrm{F}]\!][\![\pi\ \mathrm{I}]\!])$

The function $\mathcal{GA}$ processes clauses from the `pick` instruction.

$\mathcal{GA}[\![\ ]\!]\langle\rangle\mathrm{I} = \lambda\pi.\,\pi$

$\mathcal{GA}[\![(\mathrm{F})\ (\mathrm{F})^*]\!](\langle\mathrm{I}\rangle\ \S\ \mathrm{I}^*)\mathrm{I}_0 = \lambda\pi.\,\mathcal{GA}[\![(\mathrm{F})^*]\!]\mathrm{I}^*\mathrm{I}_0(\mathcal{GQ}[\![\mathrm{F}]\!][\![\pi\ (\texttt{b}\ \mathrm{I}_0)\ \mathrm{I}]\!])$

The function $\mathcal{GB}$ processes bindings in the `letrec` instruction.

$\mathcal{GB}[\![\ ]\!] = \lambda\pi.\,\pi$

$\mathcal{GB}[\![(\mathrm{I}\ \mathrm{F})\ \mathrm{B}']\!] = \lambda\pi.\,\mathcal{GB}[\![\mathrm{B}']\!](\mathcal{GQ}[\![\mathrm{F}]\!][\![\pi\ \mathrm{I}]\!])$

The function $\mathcal{GP}$ processes programs.

$\mathcal{GP}[\![\mathrm{P}']\!] = \mathcal{GQ}[\![\mathrm{P}']\!]\langle\rangle$

8.2 Auxiliary Code Generator Functions

The function mklabel generates identifiers from PAL line numbers. It must be an injection and it must not generate an identifier which is either a defined variable or a `letrec`-bound variable. The function mkmanylabels generates a sequence of unique identifiers and has the same properties as mklabel.

The `call` instruction is translated into a sequence of register move instructions, load literal instructions, followed by a jump instruction. The function mkcallconsts generates the load literal instructions.

```
$\mathsf{mkcallconsts}\langle\rangle\pi = \pi$
$\mathsf{mkcallconsts}(\mathrm{J}^* \S \langle\mathrm{R}\rangle)\pi = \mathsf{mkcallconsts}\,\mathrm{J}^*\pi$
$\mathsf{mkcallconsts}(\mathrm{J}^* \S \langle{'}\mathrm{K}\rangle)\pi =$
  let
    $\mathrm{R} = 1 + \#\mathrm{J}^*$
  in
    $\mathsf{mkcallconsts}\,\mathrm{J}^*[\![\pi\ (\mathtt{li}\ \mathrm{R}\ {'}\mathrm{K})]\!]$
```

The function mkcallrefs generates the register move instructions.

$$\mathsf{makecallrefs}\,\mathrm{J}^*\pi = \mathsf{simplemoves}(\mathsf{assigngraph}\,\mathrm{J}^*)\pi$$

Register moves are generated by analyzing something called an assignment graph. Each vertex in the graph is a local reference (index into the register bank). Given a sequence of constants and local references J^*, the set of edges in an assignment graph is

$$\mathsf{assigngraph}\,\mathrm{J}^* = \{(\mathrm{R}_0, \mathrm{R}_1) \mid \mathrm{J}^* \downarrow \mathrm{R}_0 = \mathrm{R}_1 \wedge \mathrm{R}_0 \neq \mathrm{R}_1\}$$

The ordered pair $(\mathrm{R}_0, \mathrm{R}_1)$ signifies that data will be moved from R_1 to R_0. Assignment graphs always have two properties. First, an assignment graph is disjoint from the identity relation, in the sense that it contains no pairs of the form (x, x). Second, an assignment graph ϕ is always a function, in the sense that $(\mathrm{R}_0, \mathrm{R}_1) \in \phi \wedge (\mathrm{R}_0, \mathrm{R}_2) \in \phi \rightarrow \mathrm{R}_1 = \mathrm{R}_2$. These two properties of assignment graphs are preserved by graph manipulations in what follows.

Simple moves are generated whenever there are pairs in the assignment graph that are not members of cycles; otherwise, a spare register will be used to break a cycle. More formally, simple moves are generated when the assignment graph has local references in its domain that are not in the image of its domain. The existence of such a local reference means the stack location to which it refers is semantically dead. An implementation of a parallel assignment can safely overwrite this location.

```
simplemoves $\phi\pi$ =
  $\phi = \{\} \to \pi$,
  $\exists(\mathrm{R}_0, \mathrm{R}_1) \in \phi, \mathrm{R}_0 \notin \mathrm{img}\,\phi \to$
    simplemoves$(\phi \setminus \{(\mathrm{R}_0, \mathrm{R}_1)\})[\![\pi$ (mv $\mathrm{R}_0$ $\mathrm{R}_1$)$]\!]$,
    breakcycle $\phi\pi$
```

While the algorithm presented nondeterministically picks an edge from the set ϕ, the implementing code deterministically picks one based on a canonical ordering of edges.

The function breakcycle is used only when the image of the domain of the assignment graph is its domain. Since the assignment graph is a surjection, by the pigeonhole principle, it as also a bijection, and therefore a permutation.

In this situation, the stack location to which each local reference refers is not known to be semantically dead. One of the cycles of dependency is broken by moving the value in stack location R to a spare register, and eliminating references to the value via local reference R in favor of using the spare. In the register machine, the register $\mathrm{R}_s = -3$ is dedicated solely to the task of being the spare.

In these graphs, when one cycle of dependency is broken, all vertices in the cycle become linearly ordered. The function breakcycle linearizes one cycle and lets the function simplemoves construct the sequences of move instructions indicated by the linear chain.

```
breakcycle $\phi\pi$ =
  let
    $(\mathrm{R}_0, \mathrm{R}_1) \in \phi$
  in
    simplemoves$(\phi \cup \{(\mathrm{R}_0, \mathrm{R}_s)\} \setminus \{(\mathrm{R}_0, \mathrm{R}_1)\})[\![\pi$ (mv $\mathrm{R}_s$ $\mathrm{R}_1$)$]\!]$
```

Notice that R_s is never in the domain of an assignment graph, so R_s will never be a vertex in any graph given to breakcycle.

8.3 Example

The results of applying the code generator to the SAL program in Figure 6.2 is shown in Figures 8.1 and 8.2.

```
(cmt "Declaration of" number)
(def d20)
(cmt "Declaration of" dec)
(def d21)
(cmt "Declaration of" even)
(def d22)
(cmt "Declaration of" odd)
(def d23)
(dcl p31 p32)
(lu 0)
(sw 0 d20)                ; (set! number (if #f #f))
(lu 0)
(sw 0 d21)
(lu 0)
(sw 0 d22)
(lu 0)
(sw 0 d23)
(li 1 '77)
(j p31)                   ; (even-0 77)
```

Figure 8.1: Even-odd declarations in PAL

```
p31                             ; (lambda (y)
    (cmt "Code defined by" even)
    (= 0 '0 1)                  ;   (if (= 0 y)
    (brf 0 l23)
    (li 0 '0)                   ;       0
    (j -2)
l23
    (+ 2 '-1 1)
    (mv 1 2)                    ;       (odd-0 (- y 1))))
    (j p32)
p32
    (cmt "Code defined by" odd)
    (= 0 '0 1)
    (brf 0 l33)
    (li 0 '1)
    (j -2)
l33
    (+ 2 '-1 1)
    (mv 1 2)
    (j p31)
```

Figure 8.2: Even-odd code in PAL

8.4 Correctness

Lemma 7.9 assumed the existence of a function which produces PAL code that refines syntax directed compiler generated SAL code given as input. The goal of this section is to show that $\mathcal{GP}$ is such a function.

There are many details involved in showing the correctness of the code generator, but by far, the two most interesting and intricate aspects are associated with the code generated for parallel assignments and conditional expressions containing join points. They will be the focus of the presentation.

The notion of a parallel assignment is given in Definition 7.2 and used in the specification of the refinement relation for the `call` instruction (Case 14 in Definition 7.3).

Lemma 8.1 *The code* $\mathsf{mkcallconsts}\,\mathrm{J}^*(\mathsf{mkcallrefs}\,\mathrm{J}^*\pi)\dagger\#\pi$ *implements a parallel assignment of* J^**.*

Proof: The proof is by induction on the size of assignment graphs. Consider a sequence of local references and quoted constants J^* with an empty assignment graph. This means that for each local reference R in J^*, $\mathrm{J}^* \downarrow \mathrm{R} = \mathrm{R}$. Inspection of function $\mathsf{mkcallconsts}$ shows it generates the required code.

For the induction step, assume the lemma is true when assignment graphs have n edges, and the assignment graph ϕ of J^* has $n+1$ edges. There are two cases. First assume there exists an R_0 which is in the domain of ϕ but not in its image. Let R_1 be the local reference associated with R_0 by the function ϕ. Let

$$\begin{aligned}
\mathrm{J}_0^* &= \mathsf{takefirst}\,\mathrm{J}^*(\mathrm{R}_0 - 1)\ \S\ \langle \mathrm{R}_0\rangle\ \S\ \mathsf{dropfirst}\,\mathrm{J}^*\mathrm{R}_0\\
\phi_0 &= \mathsf{assigngraph}\,\mathrm{J}_0^*\\
\pi_0 &= \mathsf{mkcallconsts}\,\mathrm{J}_0^*(\mathsf{mkcallrefs}\,\mathrm{J}_0^*\pi)\dagger\#\pi
\end{aligned}$$

so that

$$\begin{aligned}
\mathrm{J}_0^* \downarrow \mathrm{R} &= \begin{cases} \mathrm{R}_0 & \mathrm{R} = \mathrm{R}_0 \\ \mathrm{J}^* \downarrow \mathrm{R} & \text{otherwise} \end{cases}\\
\phi_0 &= \phi \setminus \{(\mathrm{R}_0, \mathrm{R}_1)\}
\end{aligned}$$

Since ϕ_0 has n edges, by hypothesis, the code π_0 implements a parallel assignment of J_0^*. The use of the definition of $\mathsf{simplemoves}$ gives:

$$\pi_1 = \mathsf{mkcallconsts}\,\mathrm{J}^*(\mathsf{mkcallrefs}\,\mathrm{J}^*\pi)\dagger\#\pi = [\![(\mathtt{mv}\ \mathrm{R}_0\ \mathrm{R}_1)\ \pi_0]\!]$$

Definition 7.2 shows that the additional move is what is required to make the parallel assignment for J_0^* into one for J^* because R_0 is semantically dead, i.e. $\mathrm{R}_0 \notin \mathcal{R}[\![\texttt{(call J}^*\texttt{)}]\!]$.

Finally, assume the image of the domain of ϕ is its domain. Following the method implemented in breakcycle, pick $(\mathrm{R}_0, \mathrm{R}_1) \in \phi$. Let

$$\begin{aligned}&\mathrm{J}_0^* = \mathsf{takefirst}\,\mathrm{J}^*(\mathrm{R}_0 - 1)\ \S\ \langle \mathrm{R}_s \rangle\ \S\ \mathsf{dropfirst}\,\mathrm{J}^*\mathrm{R}_0\\ &\phi_0 = \mathsf{assigngraph}\,\mathrm{J}_0^*\\ &\pi_0 = \mathsf{mkcallconsts}\,\mathrm{J}_0^*(\mathsf{mkcallrefs}\,\mathrm{J}_0^*\pi) \dagger \#\pi\end{aligned}$$

so that

$$\begin{aligned}&\mathrm{J}_0^* \downarrow \mathrm{R} = \begin{cases} \mathrm{R}_s & \mathrm{R} = \mathrm{R}_0 \\ \mathrm{J}^* \downarrow \mathrm{R} & \text{otherwise} \end{cases}\\ &\phi_0 = \phi \cup \{(\mathrm{R}_0, \mathrm{R}_s)\} \setminus \{(\mathrm{R}_0, \mathrm{R}_1)\}\end{aligned}$$

Since R_s is not in the domain of ϕ_0, there exists a local reference which is in the domain of ϕ_0 but not in its image. The set ϕ_0 has $n + 1$ edges, so by the previous case, the code π_0 implements a parallel assignment of J_0^*. The use of the definition of breakcycle gives:

$$\pi_1 = \mathsf{mkcallconsts}\,\mathrm{J}^*(\mathsf{mkcallrefs}\,\mathrm{J}^*\pi) \dagger \#\pi = [\![\texttt{(mv}\ \mathrm{R}_s\ \mathrm{R}_1\texttt{)}\ \pi_0]\!]$$

Definition 7.2 shows that the additional move is what is required to make the parallel assignment for J_0^* into one for J^*. ∎

The code generator produces linear code from tree structured code. The linear code contains join points when the source contains `bwf` or `pick` instructions. One could eliminate the need to generate join points by duplicating code. The instruction

$$[\![\texttt{(bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1)\texttt{)}\ \mathrm{G}]\!]$$

could be compiled as

$$[\![\texttt{(bif}\ (\mathrm{F}_2)\ (\mathrm{F}_3)\texttt{)}]\!]$$

where $\mathrm{F}_2 = \mathsf{join}\,\mathrm{F}_0\mathrm{G}$ and $\mathrm{F}_3 = \mathsf{join}\,\mathrm{F}_1\mathrm{G}$. The resulting code would be far too large. However, the method of code duplication motivates the form of the correctness proof of a code generator which produces join points. In particular, the correctness proof inductively assumes that a code generator is correct when using the code duplication method to conclude that it is also correct when producing join points. The induction is well-founded when the code being compiled is stack safe. There are several preliminary lemmas that are required before these notions can be made precise.

As stated in the introduction, the code generator appends newly generated code onto previously generated code.

Lemma 8.2 (Prefix Lemma) *If code* $\pi_0 = \mathcal{GQ}[\![\mathrm{Q}]\!]\pi$ *then* π *is a prefix of* π_0.

Proof: Inspection of the definition of $\mathcal{GQ}$ shows the result. ∎

Definition 8.3 (Overline Notation) *Let* $\overline{\mathcal{GQ}}[\![\mathrm{Q}]\!]\pi = \mathcal{GQ}[\![\mathrm{Q}]\!]\pi \dagger \#\pi$.

Lemma 8.4 (Join Lemma) *Assume* $\pi' \simeq_q \mathrm{G}$ *and*

$$\overline{\mathcal{GQ}}[\![\mathrm{Q}]\!]\pi \mathbin{\S} \pi'' \simeq_q \mathsf{join}\,\mathrm{QG}$$

Then

$$\overline{\mathcal{GQ}}[\![\mathrm{Q}]\!]\pi \mathbin{\S} \pi' \simeq_q \mathsf{join}\,\mathrm{QG}$$

Proof: The correctness proof for code linearization depends on properties of the function join. The domain of join has four noncode producing instructions, `ignore`, `bra`, `restore`, and `dispose`. The code producing instructions have the form:

$$\mathrm{Q} = [\![(\ldots)\ \mathrm{G}]\!]$$

and can be rewritten as:

$$\mathrm{Q} = \mathsf{join}[\![(\ldots)\ (\mathtt{bra})]\!]\mathrm{G}$$

Therefore, for each join expression, it is possible to construct an equivalent sequence of of join expressions in which every code producing instruction is followed by `bra`.

The proof of this lemma uses the following inductive scheme. Consider code Q_0 with no code-producing instructions. Usually the proof for this case is immediate. Let code Q_{n+1} contain $n+1$ code-producing instructions. It can be written as the join of code Q_n containing n code-producing instructions and code Q_1 containing one code-producing instruction. The case analysis focuses on code sequences with one code-producing instruction, and induction proceeds in a back-to-front direction in a code sequence.

A complete proof would display the result for each instruction in the domain of the function join. A few interesting cases are shown.

Case of $\mathrm{Q} = [\![(\texttt{bra})]\!]$:

$$\overline{\mathcal{GQ}}[\![(\texttt{bra})]\!]\pi \mathbin{\S} \pi' = \langle\rangle \mathbin{\S} \pi' = \pi' \simeq_q \mathrm{G} = \mathsf{join}\,\mathrm{QG}$$

Case of $\mathrm{Q} = [\![(\texttt{lit}\ '\mathrm{K})\ (\texttt{bra})]\!]$:

$$\mathsf{join}\,\mathrm{QG} = [\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]$$

Let $\pi_0 = \overline{\mathcal{GQ}}[\![\mathrm{Q}]\!]\pi = [\![(\texttt{li}\ 0\ '\mathrm{K})]\!]$. Since $\pi' \simeq_q \mathrm{G}$, by Case 2 of Definition 7.3, $\pi_0 \mathbin{\S} \pi' \simeq_q \mathsf{join}\,\mathrm{QG}$.

Case of $\mathrm{Q} = [\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ (\texttt{bra})]\!]$:

$$\mathsf{join}\,\mathrm{QG} = [\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!]$$

Let

$$\begin{aligned}
&\mathrm{I}_0 = \mathsf{mklabel}\,\#\pi\\
&\pi_0 = \mathcal{GQ}[\![\mathrm{F}_0]\!][\![\pi\ (\texttt{brf}\ 0\ \mathrm{I}_0)]\!]\\
&\mathrm{I}_1 = \mathsf{mklabel}\,\#\pi_0\\
&\pi_1 = \mathcal{GQ}[\![\mathrm{F}_1]\!][\![\pi_0\ (\texttt{b}\ \mathrm{I}_1)\ \mathrm{I}_0]\!]\\
&\pi_2 = [\![\pi_1\ \mathrm{I}_1]\!]\\
&\pi_3 = \pi_2 \mathbin{\S} \pi''\\
&\pi_4 = \pi_2 \mathbin{\S} \pi'
\end{aligned}$$

so by definition of the $\texttt{bwf}$ case for $\mathcal{GQ}$,

$$\pi_2 = \mathcal{GQ}[\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ (\texttt{bra})]\!]\pi = \mathcal{GQ}[\![\mathrm{Q}]\!]\pi$$

By assumption $\pi_3 \dagger \#\pi \simeq_q \mathsf{join}\,\mathrm{QG}$ so by Case 11 of Definition 7.3,

$$\pi_3 \dagger \#\pi + 1 \simeq_q \mathsf{join}\,\mathrm{F}_0\mathrm{G} \tag{8.1}$$
$$\pi_3 \dagger \#\pi_0 + 2 \simeq_q \mathsf{join}\,\mathrm{F}_1\mathrm{G} \tag{8.2}$$
$$\mathsf{goto}\,\mathrm{I}_0(\pi_3 \dagger \#\pi + 1) = \pi_3 \dagger \#\pi_0 + 2 \tag{8.3}$$

Since the prefix π_2 is the same for both π_3 and π_4, Equation 8.3 is true when π_4 is substituted for π_3.

$$\pi_3 \dagger \#\pi' + 1 = [\![\mathrm{I}_1]\!] \mathbin{\S} \pi'$$

so $\pi' \simeq_q \mathrm{G}$ implies that $[\![\mathrm{I}_1]\!] \mathbin{\S} \pi' \simeq_q \mathrm{G}$. Use of the induction hypothesis shows that Equation 8.2 is true when π_4 is substituted for π_3. Following the branch instructions, $\pi' \simeq_q \mathrm{G}$ implies that $\pi_2 \dagger \#\pi_0 \simeq_q \mathrm{G}$ and use of the induction hypothesis shows that Equation 8.1 is true when π_4 is substituted for π_3. ∎

The Join Lemma formalizes the intuition presented on code duplication, because

$$\mathcal{GQ}(\mathsf{join}\,\mathrm{QG})\pi = \mathcal{GQ}[\![\mathrm{G}]\!](\mathcal{GQ}[\![\mathrm{Q}]\!]\pi) = \mathcal{GQ}[\![\mathrm{Q}]\!]\pi \,\S\, \overline{\mathcal{GQ}}[\![\mathrm{G}]\!](\mathcal{GQ}[\![\mathrm{Q}]\!]\pi)$$

Lemma 8.5 *If code* Q *is stack safe for some stack size* N, $\overline{\mathcal{GQ}}[\![\mathrm{Q}]\!]\pi \simeq_q \mathrm{Q}$.

Proof: The proof follows the inductive structure of Definition 5.3 for stack safe code.

Case of `return`:

$$\mathcal{GQ}[\![(\texttt{return})]\!]\pi \dagger \#\pi = [\![(\mathtt{j}\ {-2})]\!] \simeq_q [\![(\texttt{return})]\!]$$

Case of `lit`: Let $\pi_0 = \mathcal{GQ}[\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]\pi \dagger \#\pi + 1$. The induction hypothesis is $\pi_0 \simeq_q \mathrm{G}$.

$$\begin{aligned}
&\overline{\mathcal{GQ}}[\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]\pi \\
&\quad = \mathcal{GQ}[\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]\pi \dagger \#\pi \\
&\quad = \mathcal{GQ}[\![\mathrm{G}]\!][\![\pi\ (\texttt{li}\ 0\ '\mathrm{K})]\!] \dagger \#\pi \\
&\quad = [\![\pi\ (\texttt{li}\ 0\ '\mathrm{K})\ \pi_0]\!] \dagger \#\pi \\
&\quad = [\![(\texttt{li}\ 0\ '\mathrm{K})\ \pi_0]\!]
\end{aligned}$$

The definition of code refinement implies that $[\![(\texttt{li}\ 0\ '\mathrm{K})\ \pi_0]\!] \simeq_q [\![(\texttt{lit}\ '\mathrm{K})\ \mathrm{G}]\!]$.

Case of `bwf`: Since stack safe code is assumed, the induction hypotheses are

$$\begin{aligned}
&\overline{\mathcal{GQ}}(\mathsf{join}\,\mathrm{F}_0\mathrm{G})\pi_0 \simeq_q \mathsf{join}\,\mathrm{F}_0\mathrm{G} \\
&\overline{\mathcal{GQ}}(\mathsf{join}\,\mathrm{F}_1\mathrm{G})\pi_1 \simeq_q \mathsf{join}\,\mathrm{F}_1\mathrm{G} \\
&\overline{\mathcal{GQ}}[\![\mathrm{G}]\!]\pi_2 \simeq_q \mathrm{G}
\end{aligned}$$

Let

$$\begin{aligned}
&\mathrm{I}_0 = \mathsf{mklabel}\,\#\pi_3 \\
&\pi_0 = \mathcal{GQ}[\![\mathrm{F}_0]\!][\![\pi_3\ (\texttt{brf}\ 0\ \mathrm{I}_0)]\!] \\
&\mathrm{I}_1 = \mathsf{mklabel}\,\#\pi_0 \\
&\pi_1 = \mathcal{GQ}[\![\mathrm{F}_1]\!][\![\pi_0\ (\texttt{b}\ \mathrm{I}_1)\ \mathrm{I}_0]\!] \\
&\pi_2 = \mathcal{GQ}[\![\mathrm{G}]\!][\![\pi_1\ \mathrm{I}_1]\!]
\end{aligned}$$

so by definition, $\mathcal{GQ}[\![(\texttt{bwf}\ (\mathrm{F}_0)\ (\mathrm{F}_1))\ \mathrm{G}]\!]\pi_3 = \pi_2$.

Referring to the definition of the refinement relation (Case 11 of Definition 7.3), the proof is complete when the following is shown:

$$\pi_2 \dagger \#\pi_3 + 1 \simeq_q \mathsf{join}\,\mathrm{F}_0\mathrm{G} \tag{8.4}$$
$$\pi_2 \dagger \#\pi_0 + 2 \simeq_q \mathsf{join}\,\mathrm{F}_1\mathrm{G} \tag{8.5}$$
$$\mathsf{goto}\,\mathrm{I}_0(\pi_2 \dagger \#\pi_3 + 1) = \pi_2 \dagger \#\pi_0 + 2 \tag{8.6}$$

By the Prefix Lemma,

$$\begin{aligned}\pi_2 &= \pi_3 \S \pi_0 \dagger \#\pi_3 \S \pi_1 \dagger \#\pi_0 \S \pi_2 \dagger \#\pi_1 \\ &= \pi_3 \S [\![(\texttt{brf}\ 0\ \mathrm{I}_0)]\!] \S \pi_0 \dagger \#\pi_3 + 1 \\ &\qquad \S [\![(\mathsf{b}\ \mathrm{I}_1)\ \mathrm{I}_0]\!] \S \pi_1 \dagger \#\pi_0 + 2 \\ &\qquad \S [\![\mathrm{I}_1]\!] \S \pi_2 \dagger \#\pi_1 + 1\end{aligned}$$

Equation 8.6 is true because the code generator produces unique labels. The proof of Equation 8.4 uses the Join Lemma.

$$\overline{\mathcal{GQ}}[\![\mathrm{G}]\!][\![\pi_1\ \mathrm{I}_1]\!] = \mathcal{GQ}[\![\mathrm{G}]\!][\![\pi_1\ \mathrm{I}_1]\!] \dagger \#\pi_1 + 1 = \pi_2 \dagger \#\pi_1 + 1$$

so $\pi_2 \dagger \#\pi_1 + 1 \simeq_q \mathrm{G}$. Following the branch instructions gives $\pi_2 \dagger \#\pi_0 \simeq_q \mathrm{G}$. Use of the Join Lemma and an induction hypothesis gives the desired result.

$$\begin{aligned}&\pi_2 = \pi_0 \S \pi_2 \dagger \#\pi_0 = \mathcal{GQ}[\![\mathrm{F}_0]\!][\![\pi_3\ (\texttt{brf}\ 0\ \mathrm{I}_0)]\!] \S (\pi_2 \dagger \#\pi_0) \\ &\pi_2 \dagger \#\pi_3 + 1 \simeq_q \mathsf{join}\,\mathrm{F}_0\mathrm{G}\end{aligned}$$

The case for Equation 8.5 is similar to the one for Equation 8.4. ▮

Theorem 8.6 (Code Generator Correctness) *If* P' *is syntax directed compiler generated code, then* $\mathcal{GP}[\![\mathrm{P}']\!] \simeq_q \mathrm{P}'$.

Proof: By Theorem 6.7, P' is stack safe code. Expanding the definition of $\mathcal{GP}$ and using Lemma 8.5 gives the result. ▮

8.5 Code Correspondence

The correspondence between code and algorithm differs in the three phases of the compiler. In the transformational front end, code rewriting rules are verified, not the algorithms used to apply the rules. In the syntax directed compiler, the code faithfully reflects the algorithm presented in Chapter 6.

As stated at the beginning of this chapter, the implementation of the code generator is imperative. Nevertheless, the algorithm is directly reflected in

```
(define (pal-bwf out then else after)
  (pal-bwf-aux val then else after out))

(define (pal-bwf-aux src then else after out)
  (let ((l0 (make-label)))
    (emit-brf src l0 out)
    (pal-code then out)
    (let ((l1 (make-label)))
      (emit-b l1 out)
      (emit-label l0 out)
      (pal-code else out)
      (emit-label l1 out)
      (pal-code after out))))
```

Figure 8.3: Code Generator for `bwf`

the code. Figure 8.3 contains the procedure used to translate a `bwf` instruction. The function `pal-code` corresponds to $\mathcal{GQ}$. The function `pal-bwf-aux` is invoked with its first argument specifying something other than the value register when a `copy` instruction is immediately followed by a `bwf` instruction.

The implementation has a feature not directly reflected in the algorithm. The live stack location analysis is performed as SAL instructions are formed, before the code generation is attempted. The join point associated with each instance of a `bra` instruction is saved, and the live stack location analysis of a `bra` instruction is defined to be that of its associated join point. As a result, there is no need to implement function join.

Slightly better code could have been produced had the following source-to-source translation been implemented for SAL code. When $\mathrm{R} \notin \mathcal{R}[\![\mathrm{G}]\!]$

$$[\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!] \Longrightarrow [\![(\texttt{ignore})\ (\texttt{unspec})\ \mathrm{G}]\!]$$

This transformation could have been implemented by changing the definition of mkmove in Figure 6.1 to:

$$\begin{aligned}\mathsf{mkmove}\,\mathrm{R}\mathrm{G} = {} & \mathrm{R} = 0 \vee \mathsf{isreturn}\,\mathrm{G} \to \mathrm{G},\\ & \mathrm{R} \in \mathcal{R}[\![\mathrm{G}]\!] \to [\![(\texttt{move}\ \mathrm{R})\ \mathrm{G}]\!],\\ & [\![(\texttt{ignore})\ (\texttt{unspec})\ \mathrm{G}]\!]\end{aligned}$$

Experiments showed this definition results in only a marginal improvement in performance.

Chapter 9

The Implementation of the Compiler

The MTPS compiler was coded in Scheme. To test the performance of the MTPS compiler, a small change was made to the front end so as to allow it to compile pre-existing programs in a single-threaded dialect of PreScheme called VLISP PreScheme. The modified compiler was called VPS, and uses the verified algorithms given here to translate VLISP PreScheme programs into PreScheme Assembly Language. An unverified Scheme program called PALAS was constructed which translates PAL source into MIPS Assembly Language [9]. For testing and debugging purposes, VPS can produce C code. While the C code is unverified as is its compiler, practice has shown that it provides a useful gauge of run-time performance.

Since PAL is so close to MIPS Assembly Language, the function of PALAS is limited. The PALAS program is a simple finite state machine. For each small number of PAL instructions read, it produces a small number of MIPS Assembly Language instructions. The program statically allocates memory for each defined variable. Each `letrec` bound variable is translated into a code label. The most complex task performed by PALAS is the translation of primitives into sequences of MIPS instructions. In addition, sometimes a predicate followed by a conditional branch instruction is translated into a single conditional branch instruction. For example, a conditional branch predicated on the equality of two values can be implemented using one conditional branch instruction on the MIPS architecture.

The program VPS was used to compile the VLISP Virtual Machine (VVM) [6], which is a byte-code interpreter. The VVM source is about 2200

Program	Time (Sec.)	Machine	Description
VVM	1414.9	MIPS	VVM translated into PAL
CVM	533.8	MIPS	CVM compiled using GCC

Table 9.1: Performance of Generated Code

lines of code and makes extensive use of the various features of the VPS program. Indeed, the features provided by VPS were mostly motivated by the VLISP PreScheme language.

The original source for the VVM specified an iterative process because the original back-end of the compiler was unable to handle non-tail-recursive procedure calls. The interpreter's garbage collector is naturally written as a subroutine which returns to the point at which it was called. The interpreter was expressed as an iterative process by writing code which explicitly saved the information required by the garbage collector to restart the interpreter upon completion of garbage collection.

The source for the VVM was rewritten as a recursive process. This version more closely matched the algorithmic specification of the VVM and was accepted by the current version of VPS. Measurements based on generating C code showed the recursive version of the VVM is slightly faster than the iterative one.

The code written by VPS was compared as follows. One of the authors wrote a C version of the VLISP Byte-Code Interpreter, called CVM, which was designed only for speed and not structured for verification. It was compiled on a Sun 4 using GCC version 2.1 with the -O2 switch, and the generated assembly code was studied. Any inefficiency observed in the code resulted in a modification to the C source. When the process was completed, all the variables used to describe the state of the byte-code interpreter were placed into machine registers, and the code sequences for the most used byte-code instructions appeared to be optimal.

Table 9.1 shows that it took 2.65 times longer to run programs on a MIPS-based machine using the VPS compiled verified PreScheme version of the VLISP Byte-Code Interpreter as compared with the C version compiled with GCC version 2.1 using the -O switch. We consider this a significant accomplishment given the extensive and intricate optimization techniques used by the GNU C compiler, and the fact that the VLISP PreScheme compiler does not place the variables used to describe the state of the byte-code interpreter into machine registers.

This test suggests the current back-end produces significantly faster code than that produced by the previous back-end, called the Pure PreScheme Compiler [16]. Running the same tests showed that it took 3.9 times longer to run programs on a Sun 3/60 using the verified PreScheme version of the VLISP Byte-Code Interpreter compiled with the Pure PreScheme Compiler as compared with the C version compiled with GCC version 1.37 using the -O switch. Of course, caution is advised when comparing results produced on differing machine architectures.

Both the older Pure PreScheme compiler and our compiler have a syntax directed compilation phase. A major difference between the two back-ends is our compilation function takes an additional parameter which gives the destination of the value being computed by a code fragment. The use of the information given by this parameter allows a significant reduction in register shuffling. We suspect this difference accounts for most of the performance improvement.

For non-tail-recursive calls, the current code generator stores only the values associated with live stack locations in the control stack, rather than storing all allocated stack locations. Experiments with the VVM showed this reduction in load and store operations was insignificant for this particular program. A further reduction in these operations is possible. The current code generator will store a value in the control stack when in is already there, and load a value from the control stack when it is easy to show that that value will be never be used. These situations occur when there are multiple non-tail-recursive procedure calls. Consider the following code.

```
(lambda (a b)
  (do-this a)
  (do-that)
  (do-the-next-thing a)
  b)
```

The current code generator will produce code which stores the value of variable `b` into the control stack three times when once will do. Furthermore, the value of variable `b` needs to be loaded from the control stack only once, but the generated code will do it three times. The needless stores could have been eliminated by maintaining a model of the stack as was done for the ν-Lisp compiler [15]. Following Oliva's method, an extra argument to the main code generator function could have recorded which stack locations had already been stored into the control stack. The reward for such an analysis is unclear.

Chapter 10

Conclusion

This paper presents the Multithreaded PreScheme (MTPS) language (including its denotational semantics) and a verified compiler for the language which generates code for MIPS-based architectures. The paper does not describe a complete, fully verified MIPS system; some components of the system and parts of the verification are missing. These are principally the following:

- MTPS does not include all the interthread operators that would be needed for real multithreaded programming.
- Formal denotational semantics has not been given for the great majority of the primitive operators, and the correctness of the translation of primitive operators has not been verified.
- The existence of a model for the denotational semantics of MTPS has not be proven (see section 3.1.4).
- The operational semantics for Stack Assembly Language (SAL) is defined only for thread-unaware programs (i.e., programs that do not contain I/O and interthread operators), and the proof that the operational semantics for (thread-unaware) SAL is faithful to its (thread-unaware) denotational semantics has been omitted (because the operational semantics was derived directly from the denotational semantics).

- Although the syntactic-directed compiler is implemented to handle multiple threads, the definition of the compiler assumes that programs are single-threaded and the correctness of the compiler is justified relative to only the thread-unaware semantics.
- The PALAS translator from PAL to MIPS Assembly Language is unverified.

Appendix A

Operators in C

This chapter contains C functions that were once used to implement some of the MTPS operators. Since the formal semantics of most operators has not been defined, each C function attempts to suggest the intended behavior of the associated operator.

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

typedef long Int;                 /* At least a 32-bit number. */
                                  /* Make sure the format used */
                                  /* in write_int agrees. */
typedef int Chr;                  /* The result type of getc. */
typedef int Bool;                 /* The result type of ==. */
typedef char *String;             /* Arg type of open_input_file. */
typedef FILE *Port;               /* The arg type of getc. */
typedef unsigned char Byte;       /* The type of an element in */
                                  /* a byte array. */

#define TRUE ((Bool) 1)
#define FALSE ((Bool) 0)

#if !defined __GNUC__             /* When not using GCC, scratch */
#define inline                    /* inline directives. The */
#endif                            /* resulting code is ANSI C. */
```

A function is marked invariable if its use is side-effect free and its value does not depend on modifiable values.

```
static inline Bool
  less(Int n0, Int n1)           /* invariable */
{
  return n0 < n1;
}

static inline Bool
  leq(Int n0, Int n1)            /* invariable */
{
  return n0 <= n1;
}

static inline Bool
  eq(Int n0, Int n1)             /* invariable */
{
  return n0 == n1;
}

static inline Bool
  geq(Int n0, Int n1)            /* invariable */
{
  return n0 >= n1;
}

static inline Bool
  greater(Int n0, Int n1)        /* invariable */
{
  return n0 > n1;
}

static inline Int                /* abs or magnitude. */
  mag(Int n)                     /* invariable */
{
  return n >= 0 ? n : -n;
}
```

```
static inline Int
  plus(Int n0, Int n1)          /* invariable */
{
  return n0 + n1;
}

static inline Int
  difference(Int n0, Int n1)    /* invariable */
{
  return n0 - n1;
}

static inline Int
  times(Int n0, Int n1)         /* invariable */
{
  return n0 * n1;
}

static inline Int               /* Note: the result of / */
  quotient(Int n0, Int n1)      /* has unpredictable sign. */
{                               /* invariable */
  Int n2 = mag(n0) / mag(n1);
  return (n0 >= 0) == (n1 >= 0) ? n2 : -n2;
}

static inline Int               /* Note: the result of % */
  remainder(Int n0, Int n1)     /* has unpredictable sign. */
{                               /* invariable */
  Int n2 = mag(n0) % mag(n1);
  return n0 >= 0 ? n2 : -n2;
}

static inline Int
  ashl(Int n0, Int n1)          /* invariable */
{                               /* Arithmetic shift right */
  if (n1 >= 0) return n0 << n1;  /* on negative numbers */
  else if (n0 >= 0) return n0 >> -n1; /* requires special */
  else return -1 - ((-1 - n0) >> -n1); /* attention in C. */
}
```

```
static inline Int
  low_bits(Int n0, Int n1)      /* invariable */
{
  return n0 & ~(~0 << n1);
}

static inline Chr
  int2chr(Int n)                /* invariable */
{
  return (Chr) n;
}

static inline Int
  chr2int(Chr c)                /* invariable */
{
  return (Int) c;
}

static inline Bool
  is_char_eq(Chr c0, Chr c1)    /* invariable */
{
  return c0 == c1;
}

static inline Bool
  is_char_less(Chr c0, Chr c1)  /* invariable */
{
  return c0 < c1;
}
```

No check is made by this implemention to see if unshared vectors are mistakenly shared.

```
static Int *
  make_vector(Int n)            /* changes store */
{
  void *vec = malloc(sizeof(Int) * n);
  if (vec == NULL) {
    fprintf(stderr, "Could not allocate %ld words.\n", n);
    abort();
  }
  return (Int *) vec;
}

static inline Int
  vector_ref(Int *v, Int n)     /* side-effect free */
{
  return v[n];
}

static inline Int
  do_vector_set(Int *v, Int n, Int o) /* changes store */
{
  v[n] = o;
  return 0;                     /* Result unspecified. */
}
```

The implementation of shared vector operators is similar to the implementation of unshared vector operators.

```
static inline Int
  vector_byte_ref(Int *v, Int n) /* side-effect free */
{
  return ((Int) ((Byte *) v)[n]);
}

static inline Int
  do_vector_byte_set(Int *v, Int n, Int o) /* changes store */
{
  ((Byte *) v)[n] = (Byte) o;
  return 0;                     /* Result unspecified. */
}
```

```
static inline Bool
  addr_less(Int *v0, Int *v1)   /* invariable */
{
  return v0 < v1;
}

static inline Bool
  addr_eq(Int *v0, Int *v1)     /* invariable */
{
  return v0 == v1;
}

static inline Int *
  addr_plus(Int *v, Int n)      /* invariable */
{
  return v + n;
}

static inline Int
  addr_difference(Int *v0, Int *v1) /* invariable */
{
  return v0 - v1;
}

static inline Int
  addr2int(Int *v)              /* invariable */
{
  return (Int) v;
}

static inline Int *
  int2addr(Int n)               /* invariable */
{
  return (Int *) n;
}

static inline String
  addr2string(Int *v)           /* side-effect free */
{
  return (String) v;
}
```

```
static inline Int *
  string2addr(String s)          /* side-effect free */
{
  return (Int *) s;
}

static inline Int
  port2int(Port p)               /* invariable */
{
  return (Int) p;
}

static inline Port
  int2port(Int n)                /* invariable */
{
  return (Port) n;
}

static Chr
  read_char(Port p)              /* accesses oracle */
{
  if (feof(p)) return EOF;
  else return getc(p);
}

static Chr
  peek_char(Port p)              /* accesses oracle */
{
  if (feof(p)) return EOF;
  else return ungetc(getc(p), p);
}

static inline Bool
  is_eof_object(Chr c)           /* invariable */
{
  return c == EOF;
}
```

```
static inline Int
  write_char(Chr c, Port p)      /* accesses oracle */
{
  return fputc(c, p) == EOF ? -1 : 0;
}

static inline Int
  write_int(Int n, Port p)       /* accesses oracle */
{
  return fprintf(p, "%ld",  n) == EOF ? -1 : 0;
}

static inline Int
  write(String s, Port p)        /* accesses oracle */
{
  return fputs(s, p) == EOF ? -1 : 0;
}

static inline Int
  newline(Port p)                /* accesses oracle */
{
  return write_char('\n', p);
}

static inline Int
  force_output(Port p)           /* accesses oracle */
{
  return fflush(p);
}

static inline Bool
  is_null_port(Port p)           /* invariable */
{
  return p == NULL;
}

static inline Port
  open_input_file(String s)      /* accesses oracle */
{
  return fopen(s, "r");
}
```

```
static inline Int
  close_input_port(Port p)      /* accesses oracle */
{
  return fclose(p);
}

static inline Port
  open_output_file(String s)    /* accesses oracle */
{
  return fopen(s, "w");
}

static inline Int
  close_output_port(Port p)     /* accesses oracle */
{
  return fclose(p);
}

static inline Port
  current_input_port(void)      /* side-effect free */
{
  return stdin;
}

static inline Port
  current_output_port(void)     /* side-effect free */
{
  return stdout;
}

static Int
  read_image(Int *v, Int n, Port p) /* accesses oracle */
{
  n = fread(v, sizeof(Int), n, p);
  if (ferror(p)) {
    fprintf(stderr, "Error in read_image.\n");
    abort();
  }
  return n;
}
```

```
static Int
  write_image(Int *v, Int n, Port p) /* accesses oracle */
{
  n = fwrite(v, sizeof(Int), n, p);
  if (ferror(p)) {
    fprintf(stderr, "Error in write_image.\n");
    abort();
  }
  return n;
}
```

An implementation of the thread operators is not presented. When invoked, the operator `create-thread` copies the defined variables to thread private memory and obtains space for a new control stack. It then invokes the given procedure using this stack, in a fashion such that a return from the given procedure deallocates thread specific memory. The `exit-thread` operator also deallocates thread specific memory.

```
#define bytes_per_word (sizeof(Int))

#define useful_bits_per_word (CHAR_BIT * sizeof(Int))

static inline Int
  quit(Int n)
{                              /* accesses oracle */
  exit(n);
}

#define err(n, s) \
  report_err((n), (s), __FILE__, __LINE__))

static Int
  report_err(Int n, String s, char *f, int l)
{                              /* accesses oracle */
  (void) write(s, stderr);
  (void) newline(stderr);
  (void) fprintf(stderr,
                 "Error detected in file %s on line %d.\n",
                 f, l);
  exit(n);
}
```

Bibliography

[1] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, revised edition, 1984.

[2] William M. Farmer, Joshua D. Guttman, Leonard G. Monk, John D. Ramsdell, and Vipin Swarup. The faithfulness of the VLISP operational semantics. M 92B093, The MITRE Corporation, September 1992.

[3] William M. Farmer, Joshua D. Guttman, and John D. Ramsdell. A denotational semantics for a multithreaded language. Technical report, The MITRE Corporation, 1995.

[4] Joshua D. Guttman, Leonard G. Monk, William M. Farmer, John D. Ramsdell, and Vipin Swarup. The VLISP flattener. M 92B094, The MITRE Corporation, 1992.

[5] Joshua D. Guttman, John D. Ramsdell, Leonard G. Monk, William M. Farmer, and Vipin Swarup. The VLISP byte-code compiler. M 92B092, The MITRE Corporation, September 1992.

[6] Joshua D. Guttman, John D. Ramsdell, and Vipin Swarup. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8(1/2):33–110, 1995.

[7] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

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

[9] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[10] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages*, pages 281–292. ACM, 1989.

[11] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[12] David Kranz, Richard A. Kelsey, Jonathan A. Rees, Paul Hudak, Jim Philbin, and Norman I. Adams. Orbit: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233, June 1986. Proceedings of the '86 Symposium on Compiler Construction.

[13] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[14] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[15] Dino P. Oliva. Advice on structuring compiler back ends and proving them correct. Technical Report NU-CCS-93, Northeastern University College of Computer Science, December 1993.

[16] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[17] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, Great Britain, 1991.

[18] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, MIT AI Laboratory, 1978.

Index