The Revised VLISP PreScheme Front End

John D. Ramsdell[1]
M 93B0000095
The MITRE Corporation

August, 1993

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

Abstract

The Verified Programming Language Implementation Project developed a formally verified implementation of the Scheme programming language. It used a systems programming dialect of Scheme, called VLISP PreScheme, to program the VLISP Virtual Machine, a byte-code interpreter. The original compiler only accepted programs that specify iterative processes. This document describes a revision of the language and its compiler. The most important change is the compiler provides a stack to save control information for procedure calls so programs that specify recursive processes are accepted. The revision expands the systems programming tasks for which VLISP PreScheme can be used and simplifies the task of matching an algorithm with its code.

Acknowledgements

Leonard Monk and Vipin Swarup made important suggestions on justifying transformation rules. Jonathan Rees provided many helpful comments on an early draft of this paper. Joshua Guttman commented on the final version.

Contents

1 Introduction **1**

2 The PreScheme Language Family **4**
2.1 VLISP PreScheme 4
2.1.1 Syntax 5
2.1.2 Standard Procedures 6
2.1.3 Formal Semantics 8
2.1.4 Compiler Restrictions 12
2.2 Macro-Free PreScheme 13
2.2.1 Syntax 14
2.2.2 Semantics 14
2.2.3 Static Semantics 14
2.3 Simple PreScheme 15

3 The Transformational Compiler **18**
3.1 Transformation Rules 19
3.1.1 Syntactic Predicates 21
3.1.2 The List of Rules 23
3.2 Usage of the Rules 28
3.3 Justification of the Rules 29
3.3.1 `if` in the Consequence of an `if` 36
3.3.2 `lambda` Simplification 37
3.3.3 β-substitution 38
3.3.4 `letrec` Lifting 41
3.3.5 `letrec` Expression Merging 42
3.3.6 `letrec` Simplification 44
3.3.7 `letrec` Binding Merging 44

3.3.8 Combination in a Combination Rotation 46
3.3.9 Defined Constant Substitution 48

4 PreScheme Assembly Language 49
4.1 Abstract Syntax 51
4.2 Additional Domain Equation 51
4.3 Semantic Functions 52
4.4 Machine Instruction Auxiliary Functions 53
4.5 Additional Auxiliary Functions 55

5 The Syntax-Directed Compiler 56
5.1 Auxiliary Compiler Functions 60
5.2 Correctness 61
5.2.1 Code Linearity 61
5.2.2 Compile-Time Environments 63
5.2.3 Simple Expressions 66
5.2.4 Defined and `letrec`-Bound Variables 74

6 Results, Future Work, and Conclusions 77

A Operators in C 80

List of Figures

2.1 Predefined Procedures 6
2.2 Case Syntax 13
2.3 Typing Rules 16

3.1 Defined Constant Substitution 27
3.2 Example VLISP PreScheme Program 30
3.3 Expand into Macro-Free PreScheme 30
3.4 Name Anonymous Lambda Expressions 31
3.5 Closure Hoisting 31
3.6 Constant Folding 32
3.7 Inline Non-Tail-Recursive Procedure Calls 32
3.8 Eliminate Unused Lambda Expressions and Initializers 33

4.1 Machine Stack Layout 50

5.1 Code Generators 60

Chapter 1

Introduction

The VLISP Project developed an architecture for the specification and implementation of verified optimizing compilers. Source programs are translated into an intermediate representation based on the lambda calculus with an implicit store. The intermediate representation is transformed until it meets syntactic restrictions. These restrictions allow the generation of efficient code using a syntax-directed compiler.

The architecture assumes the source language and the intermediate representation have a formal denotational semantics. The correctness of each transformation of the program is justified relative to this semantics.

The abstract machine language which is produced by the syntax-directed compiler also has a denotational semantics expressed using the same domains as the source language. The correctness of the syntax-directed compiler is also justified relative to the denotational semantics.

At this stage, the abstract machine language is given an operational semantics which is faithful to the denotational semantics. The operational semantics is used to justify its translation into machine language.

The VLISP Project successfully used this architecture to produce a compiler for a systems programming language. It was used to compile a byte-code interpreter for a Scheme system. The VLISP Project constructed a verified design of this Scheme system, and then implemented that design [3]. The design is based on Scheme48 [5], whose byte-code interpreter is written in PreScheme.

PreScheme is a restricted dialect of Scheme intended for systems programming. It was invented by Jonathan Rees and Richard Kelsey [7]. The

language was defined so that programs can be executed using only a C-like run-time system. A compiler for this language will reject any program that requires run-time type checking, and the run-time system usually will not provide automatic storage reclamation.

PreScheme was carefully designed so that it syntactically looks like Scheme and has similar semantics. With a little care, PreScheme programs can be run and debugged as if they were ordinary Scheme programs.

This paper describes VLISP PreScheme, which is our systems programming oriented Scheme dialect inspired by PreScheme. The major syntactic difference between the two dialects of PreScheme is that VLISP PreScheme has no user-defined syntax (macros) or compiler directives. The only other major difference is that they each provide a different set of standard procedures.

The compiler for VLISP PreScheme does not accept the entire language. Compilation occurs in three stages. The first stage translates the program into a language called Macro-Free PreScheme by expanding most of VLISP PreScheme's derived syntax and changing bound variables. The second stage translates the Macro-Free program into an equivalent program by using meaning-refining transformations. If the translation succeeds, the program is in a very restricted subset called Simple PreScheme. This subset has properties that make it easy to compile. The final stage of compilation translates Simple PreScheme into machine language.

The Simple PreScheme language was inspired by Pure PreScheme, a language defined by Dino Oliva and Mitchell Wand. They created a verified compiler for Pure PreScheme [12]. A straightforward translation is all that is required to convert a Simple PreScheme program into a Pure PreScheme program, however, the current version of the Pure PreScheme compiler only compiles programs that specify iterative processes.

This paper describes the VLISP PreScheme language, the Macro-Free PreScheme language, and finally the Simple PreScheme language. The paper concludes by giving the design and verification of our program which attempts to translate Macro-Free PreScheme programs into Simple PreScheme ones, and a syntax-directed compiler for Simple PreScheme which produces a high-level assembly language called PreScheme Assembly Language. The syntax-directed compiler accepts Simple PreScheme programs that specify recursive processes.

This paper does not describe the translation of PreScheme Assembly Lan-

guage into machine language. There are two missing components: a proof that the control structures and variable environment of PreScheme Assembly Language can be correctly implemented using one machine stack and a register allocator along with its correctness proof. These topics have been addressed by Dino Oliva [11] for a closely related programming language.

This work is most closely related to the work of Richard Kelsey. The designs of both VLISP PreScheme and Pure PreScheme were strongly influenced by the design of PreScheme [7]. Furthermore, a major part of this compiler is a transformational compiler [6].

The differences between these two dialects of PreScheme reflect the differing goals of their designers. PreScheme is targeted to high-performance systems programming while VLISP PreScheme is targeted to verified compilation of state machines.

This paper revises the original paper on VLISP PreScheme [14] in several ways. The most important change is the presentation of the Simple PreScheme compiler which provides a stack to save control information for procedure calls. Procedure calls can now return to the location at which they were invoked.

The most important contribution of the previous report on VLISP PreScheme is the identification of a collection of transformation rules that can both be verified relative to the formal semantics of the source language, and form the basis of a practical optimizing compiler. This report includes a few new transformation rules, such as raising `begin`'s in an `if`'s test, which improve the quality of generated code.

The language has been changed in several small ways. The `abort` primitive has been renamed to `exit`, and the `assert` primitive has been removed. The most important change to the language is the addition of a new definition form `define-integrable` for naming lambda abstractions. This form directs the compiler to replace references to the defined variable with its definition when the variable is in the operator position of a combination. The algorithms that control the application of transformation rules have also been changed to make allowances for the syntax-directed compiler.

Chapter 2

The PreScheme Language Family

This chapter describes our systems programming oriented Scheme dialect VLISP PreScheme and various related dialects used in the VLISP project. Programs are written in VLISP PreScheme. 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.

2.1 VLISP PreScheme

VLISP PreScheme programs manipulate data objects that fit in machine words. The type of each data object is an integer, a character, a boolean, a string, a port, a pointer to an integer, or a procedure—really a pointer to a procedure. A compiler must ensure that operators are not applied to data of the wrong type without the use of run-time checks.

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

As with Scheme, implementations of PreScheme 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 PreScheme 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 in a procedure.

2.1.1 Syntax

The syntax of the VLISP PreScheme language is identical to the syntax of the language defined in the Scheme standard [4, 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 initializer for each variable bound by a `letrec` expression must be a `lambda` expression.
- Constants are restricted to integers, characters, booleans, and strings.
- Finally, a different set of the standard procedures has been specified.

```
(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))
```

Figure 2.1: Predefined Procedures

2.1.2 Standard Procedures

Most VLISP PreScheme standard procedures are listed following this paragraph. The text on the left of each entry gives the procedure's name and its type at one fixed arity. A pointer to an integer is $\star$Int; the other notation is standard. The text on the right describes the arity of the primitive. The semantics of each procedure is roughly defined by giving an implementation in C. Appendix A contains the header file that was once included into C code generated by the VLISP PreScheme Front End. The remaining primitives are defined in Figure 2.1.

`<` : Int $\times$ Int $\to$ Bool	2 or more
`<=` : Int $\times$ Int $\to$ Bool	2 or more
`=` : Int $\times$ Int $\to$ Bool	2 or more
`>=` : Int $\times$ Int $\to$ Bool	2 or more
`>` : Int $\times$ Int $\to$ Bool	2 or more
`abs` : Int $\to$ Int	1
`+` : Int $\times$ Int $\to$ Int	any
`-` : Int $\times$ Int $\to$ Int	1 or 2
`*` : Int $\times$ Int $\to$ Int	any
`quotient` : Int $\times$ Int $\to$ Int	2
`remainder` : Int $\times$ Int $\to$ Int	2

`ashl` : Int × Int → Int	2
`low-bits` : Int × Int → Int	2
`integer->char` : Int → Chr	1
`char->integer` : Chr → Int	1
`char=?` : Chr × Chr → Bool	2
`char<?` : Chr × Chr → Bool	2
`make-vector` : Int → ⋆Int	1
`vector-ref` : ⋆Int × Int → Int	2
`vector-set!` : ⋆Int × Int × Int → Int	3
`vector-byte-ref` : ⋆Int × Int → Int	2
`vector-byte-set!` : ⋆Int × Int × Int → Int	3
`addr<` : ⋆Int × ⋆Int → Bool	2
`addr=` : ⋆Int × ⋆Int → Bool	2
`addr+` : ⋆Int × Int → ⋆Int	2
`addr-` : ⋆Int × ⋆Int → Int	2
`addr->integer` : ⋆Int → Int	1
`integer->addr` : Int → ⋆Int	1
`addr->string` : ⋆Int → String	1
`string->addr` : String → ⋆Int	1
`port->integer` : Port → Int	1
`integer->port` : Int → Port	1
`read-char` : Port → Chr	0 or 1
`peek-char` : Port → Chr	0 or 1
`eof-object?` : Chr → Bool	1
`write-char` : Chr × Port → Int	1 or 2
`write-int` : Int × Port → Int	1 or 2
`write` : String × Port → Int	1 or 2
`newline` : Port → Int	0 or 1
`force-output` : Port → Int	0 or 1
`null-port?` : Port → Bool	1
`open-input-file` : String → Port	1
`close-input-port` : Port → Int	1
`open-output-file` : String → Port	1
`close-output-port` : Port → Int	1
`current-input-port` : → Port	0
`current-output-port` : → Port	0
`read-image` : ⋆Int × Int × Port → Int	3

`write-image` : $\star\text{Int} \times \text{Int} \times \text{Port} \to \text{Int}$	3
`bytes-per-word` : Int	–
`useful-bits-per-word` : Int	–
`exit` : $\forall\alpha, \text{Int} \to \alpha$	1
`err` : $\forall\alpha, \text{Int} \times \text{String} \to \alpha$	2

2.1.3 Formal Semantics

The formal semantics of VLISP PreScheme is presented in a form that very closely resembles Scheme's semantics [4, Appendix A]. It uses the same mathematical conventions and many of the standard's definitions without repeating them here. I strongly suggest you have a copy of the standard in hand as you read the rest of this section.

The VLISP PreScheme formal semantics differs from Scheme's in a small number of ways. All variables must be defined before they are referenced or assigned and no variable may be defined more than once. VLISP PreScheme 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. VLISP PreScheme `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. Finally, memory is assumed to be infinite so the storage allocator *new* always returns a location.

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

(`define` I_1 E_1) ... E_i ... (`define` I_n E_n) E_0

$\Longrightarrow$

(`define` I_1) ... (`define` I_n)
(`begin` (`set!` I_1 E_1) ... E_i ... (`set!` I_n E_n) E_0)

Abstract Syntax

K ∈ Con	constants
I ∈ Ide	variables
E ∈ Exp	expressions
B ∈ Bnd	bindings
P ∈ Pgm	programs

Pgm ⟶ (`define` I)* E
Bnd ⟶ (I (`lambda` (I*) E))*
Exp ⟶ K | I | (E E*) | (`lambda` (I*) E)
| (`begin` E* E) | (`letrec` (B) E)
| (`if` E E E) | (`if` E E) | (`set!` I E)

The variables bound by a `letrec` expression must be distinct.

Domain Equations

$\alpha \in L$		locations
$\nu \in N$		natural numbers
T	$= \{false, true\}$	booleans
H		characters
R		integers
E_v		vectors
E_s		strings
E_p		ports
M	$= \{unspecified, undefined\}$	miscellaneous
$\phi \in F$	$= E^* \to K \to C$	procedure values
$\epsilon \in E$	$= T + H + R + E_v + E_s + E_p + M + F$	expressed values
$\sigma \in S$	$= L \to (E \times T)$	stores
$\delta \in D$	$= L + E$	denoted values
$\rho \in U$	$=$ Ide $\to D$	environments
$\theta \in C$	$= S \to A$	command continuations
$\kappa \in K$	$= E \to C$	expression continuations
A	$= R$	answers
$\chi \in X$		errors

Semantic Functions

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

The definition of $\mathcal{K}$ is deliberately omitted.

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

$\mathcal{E}[\![\mathrm{K}]\!] =$ See [4, Appendix A]

$\mathcal{E}[\![\mathrm{I}]\!] =$ See [4, Appendix A]

$\mathcal{E}[\![(\mathrm{E}\ \mathrm{E}^*)]\!] =$ See [4, Appendix A]

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

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

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

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

$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}\ \mathrm{E}\ \mathrm{E})]\!] =$ See [4, Appendix A]

$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}\ \mathrm{E})]\!] =$ See [4, Appendix A]

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\kappa_0 = \lambda\epsilon\sigma.\, \epsilon \in R \to \epsilon \mid R, \mathit{wrong}\ \text{“result not an integer”}\sigma$$

The environment ρ_0 maps the name of each standard procedure to its value and maps all other identifiers to *undefined*. It maps no identifier to a location.

Auxiliary Functions

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

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

$$send : E \to K \to C$$
$$send = \lambda\epsilon\kappa.\,\kappa\epsilon$$

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

$$new : S \to L \qquad \text{[implementation-dependent]}$$
$$new \text{ satisfies } \forall\sigma, \sigma(new\,\sigma) \downarrow 2 = false$$

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

$$assign : D \to E \to C \to C$$
$$assign = \lambda\delta\epsilon\theta\sigma.\,\delta \in L \to \theta(update(\delta \mid L)\epsilon\sigma),$$
$$wrong \text{ “assignment of an immutable variable”}\sigma$$

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

$$applicate : E \to \mathrm{E}^* \to K \to C$$
$$applicate = \lambda\epsilon\epsilon^*\kappa.\,\epsilon \in F \to (\epsilon \mid F)\epsilon^*\kappa, wrong \text{ “bad procedure”}$$

2.1.4 Compiler Restrictions

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

(`case` ⟨key⟩
  ((`0`) ⟨sequence$_1$⟩)
  ⋮
  ((⟨$n-1$⟩) ⟨sequence$_n$⟩))

Figure 2.2: Case Syntax

- All references to standard procedures 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 2.2.

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

2.2 Macro-Free PreScheme

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

2.2.1 Syntax

K	$\in$ Con	constants
I	$\in$ Ide	variables
O	$\in$ Op	primitive operators
E	$\in$ Exp	expressions
B	$\in$ Bnd	bindings
P	$\in$ Pgm	programs

Pgm $\longrightarrow$ (`define` I)* E
Bnd $\longrightarrow$ (I (`lambda` (I*) E))*
Exp $\longrightarrow$ K | I | (E E*) | (`lambda` (I*) E)
| (`begin` E* E) | (`letrec` (B) E)
| (`if` E E E) | (`if` `#f` `#f`) | (`set!` I E)
| (O E*) | (`case` E ((K) E)* ((K) E))

2.2.2 Semantics

The semantics of MFPS is given by the same equations as is VLISP PreScheme's. A MFPS program's abstract syntax is derived by expanding `case` expressions as described in the Scheme standard. As with VLISP PreScheme, the formal definition of each primitive has been deliberately omitted.

2.2.3 Static Semantics

Macro-Free PreScheme programs may be strongly typed. As in Standard ML [10], types are inferred, not declared, but unlike Standard ML, there are no polymorphic variables. All expressions are monomorphic except (`if` `#f` `#f`) and (`set!` I E).

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

Type $\star\tau$ is the type of a pointer, and $\tau_1 \times \cdots \times \tau_n \to \tau$ is the type of a procedure.

The rules used to assign types to MFPS abstract syntax expressions are given in Figure 2.3. When a type is unconstrained by the rules, the expression is assigned the integer type.

2.3 Simple PreScheme

Simple PreScheme programs are syntactically restricted, strongly typed Macro-Free PreScheme programs. The syntax is as follows:

K ∈ Con	constants
I ∈ Ide	variables
O ∈ Op	primitive operators
C ∈ Cls	case clauses
S ∈ Smpl	simple expressions
B ∈ Bnd	bindings
E ∈ Exp	top level expressions
P ∈ Pgm	programs

$$
\begin{aligned}
\text{Pgm} &\longrightarrow (\texttt{define}\ \text{I})^*\ \text{E} \\
\text{Exp} &\longrightarrow (\texttt{letrec}\ (\text{B})\ \text{S}) \\
\text{Bnd} &\longrightarrow (\text{I}\ (\texttt{lambda}\ (\text{I}^*)\ \text{S}))^* \\
\text{Smpl} &\longrightarrow \text{K} \mid \text{I} \mid (\text{S}\ \text{S}^*) \mid ((\texttt{lambda}\ (\text{I}^*)\ \text{S})\ \text{S}^*) \\
&\quad \mid (\texttt{begin}\ \text{S}^*\ \text{S}) \mid (\texttt{if}\ \text{S}\ \text{S}\ \text{S}) \mid (\texttt{if}\ \texttt{\#f}\ \texttt{\#f}) \\
&\quad \mid (\texttt{case}\ \text{S}\ \text{C}) \mid (\text{O}\ \text{S}^*) \mid (\texttt{set!}\ \text{I}\ \text{S}) \\
\text{Cls} &\longrightarrow ((\text{K})\ \text{S})^*\ ((\text{K})\ \text{S})
\end{aligned}
$$

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 2.2 on page 13.

The syntactic restrictions used to define Simple PreScheme imply that these programs will meet all of the run-time conditions of a VLISP PreScheme 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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.3: Typing Rules

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 are inherited from Macro-Free PreScheme's semantics.

Chapter 3

The Transformational Compiler

There are five phases in the translation of VLISP PreScheme into Simple PreScheme.

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

Inline standard procedures: A variable bound in a program is changed if it is also bound to a standard procedure. Consequently, every occurrence of a variable bound to a standard procedure refers to that procedure. The result is a Macro-Free PreScheme program.

Apply transformation rules: Translates a Macro-Free PreScheme program into a Simple PreScheme one 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 [9]. The unifier is based on a published program by Laurence Paulson [13, p. 381].

Linearize: Translates the internal representation of a strongly typed Simple PreScheme program into a PreScheme Assembly Language using a syntax-directed compiler as described in Chapter 5.

Compilers for VLISP PreScheme do not accept all syntactically correct programs because many Macro-Free PreScheme programs cannot be translated into Simple PreScheme. Furthermore, the set of Macro-Free 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 constitutes a VLISP PreScheme 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 most complex and error prone phase of the VLISP PreScheme Front End translates Macro-Free PreScheme programs into Simple PreScheme ones. The program is transformed by applying meaning-refining rules. The rules are meaning-refining in a sense made precise in Section 3.3.

The selection and application of rules is performed by an intricate set of procedures, however, the only way a program can be modified is by the application of some rules. For some inputs, the control procedures will run forever, but when these procedures terminate successfully, errors could have been introduced only by bad rules. Therefore, the verification effort focused solely on the transformation rules.

3.1 Transformation Rules

Each rule is a conditional rewrite rule. It has a pattern, a predicate, and a replacement. An expression matches a pattern if there is an assignment of pattern variables which makes the two expressions equal. The rewrite is performed if the matching expression satisfies the predicate. A rule with pattern E_0 and replacement E_1 is written $E_0 \Longrightarrow E_1$, 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 by instantiating its pattern variables using the assign-

ment of pattern variables produced during matching. This system avoids name conflicts by ensuring all expressions are α-converted.

Definition 1 *An expression is* α-converted *if no variable occurs both bound and free, and no variable is bound more than once.*

The system avoids name conflicts by a change of bound variables in each instantiation of a pattern variable during the construction of the replacing expression. Contexts are often used in rule presentations to help express the renaming requirement.

Definition 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*

  $$(\texttt{letrec}\ ((\mathrm{I}_1\ (\texttt{lambda}\ (\mathrm{I}^*)\ C_1[\ ])) \ldots)\ C_0[\ ]).$$

- *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 3 *A context substitution,* $C[E]$*, 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 both $C[\ ]$ and E are α-converted and they share no bound variables, then $C[E]$ 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.

3.1.1 Syntactic Predicates

Transformation rule applicability may be predicated on syntactic properties in addition to matching the rule's pattern. A common predicate tests if a variable is free in an expression. The definition of three other predicates follow.

A side-effect free expressions returns a value without modifying the store so it can be eliminated when its value is ignored.

Definition 4 *The* side-effect free *expressions are defined inductively by the following rules:*

- K *is side-effect free.*
- I *is side-effect free.*
- *If* O *is side-effect free, and* $\mathrm{E}_1 \ldots \mathrm{E}_n$ *are side-effect free, then so is* (O $\mathrm{E}_1 \ldots \mathrm{E}_n$).
- (`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).
- *If* E_0, E_1, *and* E_2 *are side-effect free, then so is* (`if` E_0 E_1 E_2).
- *If* $\mathrm{E}_0, \ldots, \mathrm{E}_n$ *are side-effect free, then so is* (`case` E_0 ((K) E_1)$\ldots$).

An expression is invariable if it is side-effect free and its value does not depend on modifiable variables. When an invariable expression is evaluated later during the execution of a program, its value remains the same.

[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.

Definition 5 *The* invariable *expressions are defined inductively by the following rules:*

- K *is invariable.*
- *If* I *is immutable, it is invariable.*
- *If* O *is invariable, and* $\mathrm{E}_1 \ldots \mathrm{E}_n$ *are invariable, then so is* (O $\mathrm{E}_1 \ldots \mathrm{E}_n$).
- (`lambda` (I^*) E) *is invariable.*
- *If* $\mathrm{E}_0, \ldots, \mathrm{E}_n$ *are invariable, then so is* (`begin` $\mathrm{E}_0 \ldots \mathrm{E}_n$).
- *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* $\mathrm{E}_0, \ldots, \mathrm{E}_n$ *are invariable, then so is* (`case` E_0 ((K) E_1)...).

An expression is almost side-effect free if it does not modify the store until its last action.

Definition 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* $\mathrm{E}_1 \ldots \mathrm{E}_n$ *are side-effect free, then* (O $\mathrm{E}_1 \ldots \mathrm{E}_n$) *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.*

3.1.2 The List of Rules

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

1. Primitive expression simplification.
   - Evaluate constant expressions.
   - Simplify operations applied to identity elements.

     $(+\ 0\ \mathrm{E}) \Longrightarrow \mathrm{E}$
     $(*\ 1\ \mathrm{E}) \Longrightarrow \mathrm{E}$
   - Move constants to first operand for commutative operators.

     $(\mathrm{O}\ \mathrm{E}\ \mathrm{K}) \Longrightarrow (\mathrm{O}\ \mathrm{K}\ \mathrm{E})$
   - Move associative operators to the first operand.

     $(\mathrm{O}\ \mathrm{E}_0\ (\mathrm{O}\ \mathrm{E}_1\ \mathrm{E}_2)) \Longrightarrow (\mathrm{O}\ (\mathrm{O}\ \mathrm{E}_0\ \mathrm{E}_1)\ \mathrm{E}_2)$
   - Use special rules for difference, arithmetic shift, and address arithmetic. The rule for `vector-set!` is not shown.

     $(-\ \mathrm{E}\ \mathrm{K}) \Longrightarrow (+\ \mathrm{E}\ -\mathrm{K})$
     (`ashl` (`ashl` E_0 E_1) E_2) $\Longrightarrow$ (`ashl` E_0 $(+\ \mathrm{E}_1\ \mathrm{E}_2)$)
     (`addr+` (`addr+` E_0 E_1) E_2) $\Longrightarrow$ (`addr+` E_0 $(+\ \mathrm{E}_1\ \mathrm{E}_2)$)
     (`vector-ref` (`addr+` E_0 E_1) E_2)
     $\Longrightarrow$ (`vector-ref` E_0 $(+\ \mathrm{E}_1\ \mathrm{E}_2)$)
     (`addr+` E 0) $\Longrightarrow$ E
   - Introduce a `let` for some primitives.

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

     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

   - Raise a combination in a test.

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

   - 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 consequence 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.

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

   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

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

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

8. `letrec` substitution.

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

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

9. `letrec` simplification.

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

   and

   $$(\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}_i\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_i))\ \mathrm{B}_1)\ \mathrm{E}) \Longrightarrow (\texttt{letrec}\ (\mathrm{B}_0\ \mathrm{B}_1)\ \mathrm{E})$$

   when I_i is referenced nowhere except in E_i.

10. `letrec` lifting.

    $$C[(\texttt{letrec}\ (\mathrm{B})\ \mathrm{E})] \Longrightarrow (\texttt{letrec}\ (\mathrm{B})\ C[\mathrm{E}])$$

    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.

    $$(\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ (\texttt{letrec}\ (\mathrm{B}_1)\ \mathrm{E})))\ \mathrm{B}_2)\ \mathrm{E}_0) \Longrightarrow (\texttt{letrec}\ (\mathrm{B}_0\ (\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}))\ \mathrm{B}_1\ \mathrm{B}_2)\ \mathrm{E}_0)$$

    when I^* are not free in B_1.

12. `letrec` expression merging.

    $$(\texttt{letrec}\ (\mathrm{B}_0)\ (\texttt{letrec}\ (\mathrm{B}_1)\ \mathrm{E})) \Longrightarrow (\texttt{letrec}\ (\mathrm{B}_0\ \mathrm{B}_1)\ \mathrm{E})$$

13. `letrec` elimination.

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

    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.

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

    when E_0 is invariable.

```
(define I_1) ... (define I_i) ... (define I_n)
(letrec ((I_{n+1} C_{n+1}[I_i]) ...)
  (begin
    (set! I_1 C_1[I_i])
      ⋮
    (set! I_i E_i)
      ⋮
    (set! I_n C_n[I_i])
    C_0[I_i]))
⟹
(define I_1) ... (define I_i) ... (define I_n)
(letrec ((I_{n+1} C_{n+1}[E_i]) ...)
  (begin
    (set! I_1 C_1[E_i])
      ⋮
    (set! I_i E_i)
      ⋮
    (set! I_n C_n[E_i])
    C_0[E_i]))
```

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

Figure 3.1: Defined Constant Substitution

15. `begin` in a combination rotation.

$$(\mathrm{E}_0\ (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E})) \Longrightarrow (\texttt{begin}\ \mathrm{E}^*\ (\mathrm{E}_0\ \mathrm{E}))$$

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 3.1.

17. Unused initializer elimination. If a defined immutable variable is never referenced and it is initialized with a side-effect free expression, the initialization can be eliminated.

3.2 Usage of the Rules

The rules result in program transformations similar to those produced by other compilers [8, 6]. 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 [15]. As a result, the rotate combinations rule was added. Here is a common example of its use.

(`let` ((I_0 (`let` ((I_1 E_1)...) E^* I_1))) E_0)
$\Longrightarrow$ (`let` ((I_1 E_1)...) (`let` ((I_0 I_1)) E^* E_0))

The rules are used as follows. With the exception of the `letrec` substitution rule and the defined constant substitution rule, 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 binding. 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 with 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 in which the arguments are either variables or constants.

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

```
(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. The loop in the following example must be unwound by all compilers.

```
(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 3.2. It specifies a simple iterative process. Figures 3.3 through 3.8 show how this program might be translated into Simple PreScheme using transformations implemented in the Front End.

3.3 Justification of the Rules

The application of a rule is justified if it transforms a Macro-Free PreScheme program into another and both programs have the same meaning as given by the semantics in Section 2.2. Some of the rules have another interesting property—they can transform a program which has bottom denotation into a program which produces a non-bottom answer. For example, the program

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

```
(define number 77)                      ; This program
(define (dec x) (- x 1))                ; produces zero
(define (even x)                        ; if NUMBER is even
  (if (zero? x)                         ; and non-negative,
      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 3.2: Example VLISP PreScheme Program

```
(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 3.3: Expand into Macro-Free PreScheme

```
(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 3.4: 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 3.5: 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 3.6: 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 3.7: Inline Non-Tail-Recursive 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 0) (set! dec 0)
    (set! even 0) (set! odd 0)
    (even-0 77)))
```

Figure 3.8: Eliminate Unused Lambda Expressions and Initializers

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 *undefined* should have been substituted. In summary, the application of a rule is justified if it does not affect non-bottom computational results.

Definition 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.*

Definition 8 *Assume* $\forall\chi\sigma$, *wrong* $\chi\sigma = \bot_A$. *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 proof that a rule is meaning-refining usually has the following form. Suppose E_1 is the expression that results from applying the rule to expression E_0. The proof shows that $\mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma \sqsubseteq \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\sigma$.

The reason the proof shows that a rule is meaning-refining follows. In VLISP PreScheme, the answer domain is the integers. Let P_1 be a program which results from the application of the rule to an expression in program P_0 and let $\mathcal{P}$ be the semantic function for programs. The proof implies that $\mathcal{P}[\![\mathrm{P}_0]\!] \sqsubseteq \mathcal{P}[\![\mathrm{P}_1]\!]$ because the semantic functions are compositional which implies the rule is meaning-refining, because the answer domain is flat.

The purpose of justifying a rule is to gain confidence in the correctness of the compiler. Justifications focus on aspects of rules which are likely to cause problems. For example, 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.

Justifications do not focus on all aspects of a rule. The compiler avoids name conflicts by using α-converted expressions. Therefore, issues arising from name conflicts are not addressed. A formal semantics for each primitive has not been provided, therefore, the rules specific to primitives have not been justified. When the justification of a rule is too obvious, it has been omitted, with the exception of the justification of the `if` in the consequence of an `if` rule.

The justification of many rules employs structural induction involving a large number of cases. The complete proof is sketched by providing a detailed analysis of the most interesting cases.

The formal semantics of VLISP PreScheme require that the order of evaluation within a call is constant throughout a program for any given number of arguments. Most proofs assume arguments are evaluated left-to-right, and then the operator is evaluated. The reader will observe that the order of evaluation is relevant only in the rotate combinations rule.

The justification of rules with non-trivial predicates requires associating semantics properties with syntactic ones. Side-effect free expressions have property Π.

Definition 9 $\Pi(\mathrm{E})$ *is* $\forall\rho\sigma, \exists\epsilon, \forall\kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma = (\epsilon = \mathit{undefined} \rightarrow \bot_A, \kappa\epsilon\sigma)$.

To understand this definition, ignore the case in which $\epsilon = \mathit{undefined}$. Notice the store given to the command continuation on the LHS is the same as the one given to the command continuation on the RHS. In other words, the command continuation on the LHS is the composition of the identity function and command continuation representing the rest of the computation. This characterizes the meaning of side-effect free expressions in a form useful for proofs.

Theorem 1 E *is side-effect free implies* $\Pi(\mathrm{E})$.

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

$$\mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma = (\epsilon = \mathit{undefined} \to \bot_A, \kappa\epsilon\sigma).$$

Notice ϵ is independent of κ so

$$\forall\kappa, \mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma = (\epsilon = \mathit{undefined} \to \bot_A, \kappa\epsilon\sigma).$$

Case E = (`if` E_0 E_1 E_2). By the induction hypothesis, there is at least one ϵ_0 such that

$$\forall\kappa, \mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma = (\epsilon_0 = \mathit{undefined} \to \bot_A, \kappa\epsilon_0\sigma).$$

If $\epsilon_0 = \mathit{undefined}$, the result is immediate, otherwise,

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

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

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

otherwise,

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

Use of the induction hypothesis verifies both alternatives. ∎

Invariable expressions have property Σ.

Definition 10 $\Sigma(\mathrm{E})$ *is* $\forall\rho\sigma, \exists\epsilon, \forall\sigma'$,

$$\begin{aligned}&(\forall \mathrm{I} \in FV(\mathrm{E}), \rho\mathrm{I} \in L \textit{ implies } \sigma(\rho\mathrm{I} \mid L)\downarrow 1 = \sigma'(\rho\mathrm{I} \mid L)\downarrow 1)\\ &\textit{implies } \forall\kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma' = (\epsilon = \mathit{undefined} \to \bot_A, \kappa\epsilon\sigma').\end{aligned}$$

Theorem 2 E *is invariable implies* $\Sigma(\mathrm{E})$.

Proof sketch: The proof is identical to that of Theorem 1, except for the case of variables. As before

$$\forall\kappa, \mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma = (\epsilon = \mathit{undefined} \to \bot_A, \kappa\epsilon\sigma),$$

with $\epsilon = \rho\mathrm{I} \in E \to \rho\mathrm{I} \mid E, \sigma(\rho\mathrm{I} \mid L)\downarrow 1$. For all σ' such that

$$\rho\mathrm{I} \in L \text{ implies } \sigma(\rho\mathrm{I} \mid L)\downarrow 1 = \sigma'(\rho\mathrm{I} \mid L)\downarrow 1,$$

$\epsilon = \rho\mathrm{I} \in E \to \rho\mathrm{I} \mid E, \sigma'(\rho\mathrm{I} \mid L)\downarrow 1$, so $\mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma = \mathcal{E}[\![\mathrm{I}]\!]\rho\kappa\sigma'$. ∎

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

Lemma 1 $\Sigma(\mathrm{E})$ *implies* $\Pi(\mathrm{E})$.

Lemma 2 *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 3

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

3.3.1 `if` in the Consequence of an `if`

Theorem 3 *When* E_0 *is side-effect free,*

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

Proof: By Theorem 1, there exists an ϵ_0 such that

$$\forall\kappa, \mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa\sigma = (\epsilon_0 = \mathit{undefined} \to \bot_A, \kappa\epsilon_0\sigma).$$

If $\epsilon_0 = \mathit{undefined}$, the proof is immediate, so assume $\epsilon_0 \neq \mathit{undefined}$.

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

∎

3.3.2 `lambda` Simplification

Theorem 4 *When* E_i *is side-effect free and* I_i *is not free in* E,

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

Proof: Pick a permutation for the application. Shown is the case in which the arguments are evaluated left-to-right and then the operator is evaluated.

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

Consider the case in which there exists κ' and σ' such that

$$\mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa'\sigma' = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ \mathrm{E})\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma,$$

and ϵ_i such that

$$\forall\kappa, \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma' = (\epsilon_i = \mathit{undefined} \rightarrow \bot_A, \kappa\epsilon_i\sigma').$$

If $\epsilon_i = \mathit{undefined}$, the proof is immediate, so assume that $\epsilon_i \neq \mathit{undefined}$. Also assume there exists $\epsilon_1 \ldots \epsilon_{i-1}\epsilon_{i+1} \ldots \epsilon_n$, σ'', and ψ such that,

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

This corresponds to the case in which the evaluation of each of $\mathrm{E}_1 \ldots \mathrm{E}_n$ invokes their continuation with a value; for when they do not, the proof is again immediate. Notice that the evaluation of E_i does not change the store so

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

In the case in which the computation continues, the proof is concluded by showing

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

Expanding the definitions gives

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

This is proved by structural induction on E assuming I_i is not free in E. ∎

3.3.3 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 5 *When* E_i *is invariable,*

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

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 and Lemma 5 which follow.

Lemma 4 *When* E_i *is invariable,*

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

Proof: For the same reasons employed in Theorem 4, consider only values κ' and σ' such that

$$\mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa'\sigma' = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n)\ C[\mathrm{I}_i])\ \mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n)]\!]\rho\kappa\sigma,$$

and $\epsilon_1 \ldots \epsilon_n$, σ'', and ψ such that $\epsilon_i \neq \mathit{undefined}$, and

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

Let $\rho' = \mathit{expand}\ \rho\langle\mathrm{I}_1 \ldots \mathrm{I}_i \ldots \mathrm{I}_n\rangle\langle\epsilon_1 \ldots \epsilon_i \ldots \epsilon_n\rangle$. Expanding definitions gives

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

because none of $\mathrm{I}_1 \ldots \mathrm{I}_n$ are free in E_i.

$\forall\kappa, \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma'' = \kappa\epsilon_i\sigma''$, for if not, then at least one of the free variables of E_i was initialized. Let I_0 be one. The semantics allow only a change from a value of *undefined*, so $\sigma'(\rho\mathrm{I}_0 \mid L) \downarrow 1 = \mathit{undefined}$. Therefore, E_i must ignore the value of I_0 and the values of variables referenced by E_i must agree in both stores.

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

because $\rho' = \mathit{extends}\ \rho'\langle\mathrm{I}_i\rangle\langle\epsilon_i\rangle$. ∎

Lemma 5 *When* E *is invariable,*

$$\begin{aligned}
&(\exists\epsilon, \epsilon \neq \mathit{undefined} \wedge \forall\kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma = \kappa\epsilon\sigma) \\
&\mathit{implies}\ \mathcal{E}[\![(\texttt{lambda}\ (\mathrm{I})\ C[\mathrm{I}])\ \mathrm{E})]\!]\rho\kappa\sigma = \mathcal{E}[\![C[\mathrm{E}]]\!]\rho\kappa\sigma.
\end{aligned}$$

Proof sketch: Proved by induction on contexts. The cases of $C[\ ]$ being $[\ ]$ and (`begin` $C_0[\ ]\ C_1[\ ]$) are shown. Assume there exists an $\epsilon_0 \neq \mathit{undefined}$ such that $\forall\kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma = \kappa\epsilon_0\sigma$. Pick a permutation for the application. Shown is the case in which the argument is evaluated before the operator.

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

Case of $C[\ ] = [\ ]$: $\mathcal{E}[\![C[\mathrm{I}]]\!] = \mathcal{E}[\![\mathrm{I}]\!]$. The proof follows from the semantics of variable reference.

Case of $C[\ ] =$ (`begin` $C_0[\ ]\ C_1[\ ]$): To be shown is

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

Expanding `begin`'s definition gives

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

and

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

using the induction hypothesis for the last equality.

Assume there exists a σ' such that

$$\mathcal{E}[\![C_1[\mathrm{E}]]\!]\rho\kappa\sigma' = \mathcal{E}[\![C_0[\mathrm{E}]]\!]\rho(\lambda\epsilon.\,\mathcal{E}[\![C_1[\mathrm{E}]]\!]\rho\kappa)\sigma$$

which corresponds to the case in which the evaluation of $C_0[\mathrm{E}]$ invokes its continuation with a value. $\forall\kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma' = \kappa\epsilon_0\sigma'$, for if not, then at least one of the free variables of E was initialized. Let I_0 be one. The semantics allow only a change from a value of *undefined*, so $\sigma(\rho\mathrm{I}_0 \mid L)\downarrow 1 = \mathit{undefined}$. Therefore, E must ignore the value of I_0 and the values of variables referenced by E must agree in both stores.

The proof is completed by use of the induction hypothesis to show

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

∎

β-substitution of side-effect free expressions

Theorem 6 *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,*

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

Note the RHS must be α-converted so I_i cannot be free in E_i. The theorem is proved by appealing to Lemma 6 and Lemma 7 which follow.

Lemma 6 *When* $\mathrm{E}_1 \ldots \mathrm{E}_i \ldots \mathrm{E}_n$ *are side-effect free,*

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

The proof is identical to that of Lemma 4 except that the store never changes, i.e., $\sigma'' = \sigma' = \sigma$.

Lemma 7 *When* E_i *is side-effect free and* $C[\mathrm{I}_i]$ *is almost side-effect free,*

$$(\exists \epsilon, \epsilon \neq \mathit{undefined} \wedge \forall \kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma = \kappa\epsilon\sigma)$$
$$\mathit{implies}\ \mathcal{E}[\![(\texttt{lambda}\ (\mathrm{I})\ C[\mathrm{I}])\ \mathrm{E})]\!]\rho\kappa\sigma = \mathcal{E}[\![C[\mathrm{E}]]\!]\rho\kappa\sigma.$$

Proof sketch: The proof is very similar to the proof of Lemma 5, except that the store remains the same. Consider the case of $C[\ ]$ being ($\texttt{begin}\ C_0[\ ]\ C_1[\ ]$) and all $\epsilon \neq \mathit{undefined}$ such that $\forall \kappa, \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma = \kappa\epsilon\sigma$. Because $C[\mathrm{E}]$ is almost side-effect free, $C_0[\mathrm{E}]$ is side-effect free. In the case in which the evaluation of $C_0[\mathrm{E}]$ invokes its continuation with a value,

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

The proof is completed by use of the induction hypothesis to show

$$\mathcal{E}[\![C_1[\mathrm{E}]]\!]\rho\kappa\sigma = \mathcal{E}[\![C_1[\mathrm{I}]]\!](\mathit{extends}\ \rho\langle \mathrm{I}\rangle\langle\epsilon\rangle)\kappa\sigma.$$

∎

3.3.4 `letrec` Lifting

Theorem 7 *When* $C[\ ]$ *has one hole,*

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

Note the RHS must be α-converted so $C[\ ]$ cannot bind any free variables bound by B.
Proof sketch: Shown is the case in which $C[\ ] = (\texttt{lambda}\ (\mathrm{I}^*)\ C_0[\ ])$.

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

by the induction hypothesis.

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

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 $C[\ ]$ binds no free variables of B, it binds none of I^* and

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

Furthermore, because all expressions are α-converted, $\rho'' = \mathit{extends}\,\rho'''\mathrm{I}^*\epsilon^*$, where $\rho''' = \mathit{extends}\,\rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}\,(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho))$.

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

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

∎

3.3.5 `letrec` Expression Merging

Theorem 8

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

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

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

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

Because expressions are α-converted,

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

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

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

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

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

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

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

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

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

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

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

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

3.3.6 `letrec` Simplification

Theorem 9 *When* I *is referenced nowhere except in* E,

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

Proof: By Theorem 7,

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

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

because I is not free in E_0. ∎

3.3.7 `letrec` Binding Merging

Theorem 10

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

Note the LHS must be α-converted so no binding in B_1 can reference a variable bound by B_0.
Proof: Define the bar operator on bindings as follows.

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

In words, it adds a `letrec` of B_0 into all the `lambda` expressions being bound. As was shown in Theorem 9,

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

therefore by use of Theorem 7,

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

Let

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

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

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

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

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

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

$\mathit{fix}\ f \sqsubseteq \mathit{fix}\ g$ 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 2,

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.

fix $g \sqsubseteq$ *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

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

∎

3.3.8 Combination in a Combination Rotation

Theorem 11 *When* E_0 *is invariable,*

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

Note the LHS must be α-converted so none of I^* can be free in E_0.
Proof: Pick a permutation for the applications. Shown is the case in which the arguments are evaluated left-to-right and then the operator is evaluated.

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

For the same reasons employed in Theorem 4, assume there exists ϵ^*, σ', and ψ such that $\mathcal{E}^*[\![\mathrm{E}^*]\!]\rho\psi\sigma = \psi\epsilon^*\sigma'$. Let $\rho' = \mathit{expand}\,\rho\mathrm{I}^*\epsilon^*$. Expanding definitions gives

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

because none of I^* are free in E_0. Let $\kappa' = \lambda\epsilon.\, \mathcal{E}[\![\mathrm{E}_0]\!]\rho(\lambda\epsilon'.\, \mathit{applicate}\,\epsilon'\langle\epsilon\rangle\kappa)$.

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

Now assume that the operator is evaluated before the arguments. The proof is much like the previous one except now there is the possibility of the rule transforming an erroneous program into one that produces an answer. The situation occurs when the evaluation of one of the arguments to the `lambda` expression invokes the `exit` primitive. ∎

The author was unable to write a VLISP PreScheme program with bottom denotation which is transformed by this rule into one which produces a non-bottom answer. Can you?

3.3.9 Defined Constant Substitution

Theorem 12 *The rule shown in Figure 3.1 is meaning-refining.*

Proof: The compiler rejects programs which contain assignments to immutable variables. The immutable variable I_i can be modified only during its initialization. Therefore, for ρ and σ in the program,

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

is either *undefined* or some other value ϵ_i.

When E_i is a constant, $\epsilon_i = \mathcal{K}[\![\mathrm{E}_i]\!]$, so $\mathcal{E}[\![\mathrm{I}_i]\!]\rho\kappa\sigma \sqsubseteq \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma$. When E_i is an immutable variable which is undefined at the time of I_i's initialization, the program is erroneous, therefore I_i must be defined whenever E_i is defined so $\mathcal{E}[\![\mathrm{I}_i]\!]\rho\kappa\sigma \sqsubseteq \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma$. ∎

Chapter 4

PreScheme Assembly Language

PreScheme Assembly Language is a linear 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 formal denotational semantics follows the introduction.

The abstract machine's state is captured by five terms: a code sequence, a value, a stack, a continuation, and a store. The stack is a sequence of values as is the store. A continuation stores a return point for a non-tail-recursive call. It contains a code sequence, a stack, and another continuation.

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 from the stack bottom.

A tail-recursive call is translated into an instruction which simply shuffles the values on the stack and proceeds. A call which must return to the location at which is was invoked must save the return information as a continuation. The `make-cont` instructions 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 old code sequence,

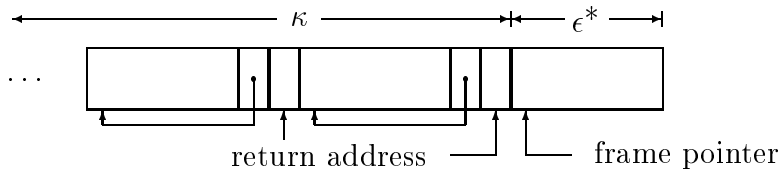

Figure 4.1: Machine Stack Layout

stack, and continuation are replaced by the information saved in the current continuation.

PreScheme Assembly Language was designed to allow translation into very efficient assembly language for a real machine. The defined variables and `letrec` bound variables can be allocated statically. The continuations and the stack can be allocated on a single machine stack. The machine stack consists of a sequence of frames adjoined to a sequence of values which make up the abstract machine's stack. A frame pointer marks the beginning of the sequence of values that make up the stack. These values represent the current bindings of `lambda` bound variables. A reference to a `lambda` bound variable is translated into a stack reference by giving an offset from the frame pointer as shown in Figure 4.1. In this diagram, the stack grows rightward. The syntactic restrictions of Simple PreScheme allow this simple variable reference scheme.

Each frame on the machine stack stores a continuation. A frame consists of a saved stack, the address of the beginning of the stack, and a return address. For the top continuation, these two addresses are at a fixed offset from the frame pointer. As a result, only one register is needed to maintain both the continuation and the stack. The syntactic restrictions of Simple PreScheme allow stack allocation of continuations.

PreScheme Assembly Language is fairly traditional in all aspects except one: the target of jump instructions are given as offsets rather than labels. The use of offsets easied the task of verifying the Simple PreScheme compiler.

4.1 Abstract Syntax

N	$\in$ Num	natural numbers (Nat $= N$)
K	$\in$ Con	constants
I	$\in$ Ide	variables
W	$\in$ Ref	variables or natural numbers
Q	$\in$ Code	machine instructions
B′	$\in$ Bnd′	bindings
E′	$\in$ Exp′	top level expressions
P′	$\in$ Pgm′	programs

$$
\begin{aligned}
\text{Pgm}' &\longrightarrow (\texttt{define}\ \text{I})^*\ \text{E}' \\
\text{Exp}' &\longrightarrow (\texttt{letrec}\ (\text{B}')\ \text{Q}^*) \\
\text{Bnd}' &\longrightarrow (\text{I}\ \text{Q}^*)^* \\
\text{Ref} &\longrightarrow \text{I} \mid \text{N} \\
\text{Code} &\longrightarrow (\texttt{literal}\ \text{K}) \mid (\texttt{unspecified}) \mid (\texttt{ignore}) \\
&\quad \mid (\texttt{fetch}\ \text{W}) \mid (\texttt{assign}\ \text{W}) \mid (\texttt{wrong}) \\
&\quad \mid (\texttt{jump}\ \text{N}) \mid (\texttt{jump-if-false}\ \text{N}) \mid (\texttt{dispatch}\ \text{N}^*) \\
&\quad \mid (\texttt{call}\ \text{W}^*) \mid (\texttt{return}) \mid (\texttt{make-cont}\ \text{N}\ \text{N}) \\
&\quad \mid (\texttt{entry}\ \text{N}) \mid (\texttt{reserve}\ \text{N}) \mid (\texttt{dispose}\ \text{N}) \\
&\quad \mid (\texttt{add}\ \text{W}\ \text{W}) \mid \cdots
\end{aligned}
$$

4.2 Additional Domain Equation

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

4.3 Semantic Functions

$$
\begin{aligned}
\mathcal{G} &: \mathrm{Code}^* \to N \to \mathrm{Code}^* \\
\mathcal{W} &: \mathrm{Ref} \to U \to W \\
\mathcal{F} &: \mathrm{Code}^* \to U \to F \\
\mathcal{Q} &: \mathrm{Code}^* \to U \to Q \\
\mathcal{I}' &: \mathrm{Bnd}' \to \mathrm{Ide}^* \\
\mathcal{B}' &: \mathrm{Bnd}' \to \mathrm{Ide}^* \to U \to E^* \to E^* \\
\mathcal{E}' &: \mathrm{Exp}' \to U \to F \\
\mathcal{D}' &: \mathrm{Pgm}' \to U \to K \to C \\
\mathcal{P}' &: \mathrm{Pgm}' \to A
\end{aligned}
$$

$$\mathcal{G}[\![\mathrm{Q}^*]\!] = \lambda\nu.\, \mathit{dropfirst}\ \mathrm{Q}^*\,\nu$$

$$
\begin{aligned}
\mathcal{W}[\![\mathrm{I}]\!] &= \lambda\rho.\, \mathit{lookup}\ \rho\mathrm{I}\ \mathrm{in}\ W \\
\mathcal{W}[\![\mathrm{N}]\!] &= \lambda\rho.\, \mathrm{N}\ \mathrm{in}\ W
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{F}[\![(\texttt{literal}\ \mathrm{K})\ \mathrm{Q}^*]\!] = \lambda\rho.\, \mathit{literal}(\mathcal{K}[\![\mathrm{K}]\!])(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho) \\
&\mathcal{F}[\![(\texttt{unspecified})\ \mathrm{Q}^*]\!] = \lambda\rho.\, \mathit{literal}\ \mathit{unspecified}(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho) \\
&\mathcal{F}[\![(\texttt{fetch}\ \mathrm{W})\ \mathrm{Q}^*]\!] = \lambda\rho.\, \mathit{fetch}(\mathcal{W}[\![\mathrm{W}]\!]\rho)(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho) \\
&\mathcal{F}[\![(\texttt{make-cont}\ \mathrm{N}_0\ \mathrm{N}_1)\ \mathrm{Q}^*]\!] \\
&\quad = \lambda\rho.\, \mathit{makecont}\,\mathrm{N}_0(\mathcal{Q}(\mathcal{G}[\![\mathrm{Q}^*]\!]\mathrm{N}_1)\rho)(\mathcal{F}[\![\mathrm{Q}^*]\!]\rho) \\
&\mathcal{F}[\![(\texttt{entry}\ \mathrm{N})\ \mathrm{Q}^*]\!] = \lambda\rho.\, \mathit{entry}\,\mathrm{N}(\mathcal{F}[\![\mathrm{Q}^*]\!]\rho) \\
&\mathcal{F}[\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{Q}^*]\!] = \lambda\rho.\, \mathit{reserve}\,\mathrm{N}(\mathcal{F}[\![\mathrm{Q}^*]\!]\rho) \\
&\mathcal{F}[\![(\texttt{add}\ \mathrm{W}_0\ \mathrm{W}_1)\ \mathrm{Q}^*]\!] = \lambda\rho.\, \mathit{add}(\mathcal{W}[\![\mathrm{W}_0]\!]\rho)(\mathcal{W}[\![\mathrm{W}_1]\!]\rho)(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)
\end{aligned}
$$

$\mathcal{Q}[\![(\texttt{ignore})\ \mathrm{Q}^*]\!] = \lambda\rho.\, ignore(\mathcal{F}[\![\mathrm{Q}^*]\!]\rho)$
$\mathcal{Q}[\![(\texttt{assign}\ \mathrm{N})\ \mathrm{Q}^*]\!] = \lambda\rho.\, setlocal\,\mathrm{N}(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)$
$\mathcal{Q}[\![(\texttt{assign}\ \mathrm{I})\ \mathrm{Q}^*]\!]$
$\quad = \lambda\rho.\, ismutable\,\mathrm{I} \to setglobal(lookup\,\rho\mathrm{I})(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho),$
$\qquad\qquad initglobal(lookup\,\rho\mathrm{I})(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)$
$\mathcal{Q}[\![(\texttt{wrong})\ \mathrm{Q}^*]\!]$
$\quad = \lambda\rho.\, \lambda\epsilon\epsilon^*\kappa.\, wrong$ "assignment of an immutable variable"
$\mathcal{Q}[\![(\texttt{jump}\ \mathrm{N})\ \mathrm{Q}^*]\!] = \mathcal{Q}(\mathcal{G}[\![\mathrm{Q}^*]\!]\mathrm{N})$
$\mathcal{Q}[\![(\texttt{jump-if-false}\ \mathrm{N})\ \mathrm{Q}^*]\!] = \lambda\rho.\, jumpfalse(\mathcal{F}[\![\mathrm{Q}^*]\!]\rho)(\mathcal{F}(\mathcal{G}[\![\mathrm{Q}^*]\!]\mathrm{N})\rho)$
$\mathcal{Q}[\![(\texttt{dispatch}\ \mathrm{N}^*)\ \mathrm{Q}^*]\!]$
$\quad = \lambda\rho.\, dispatch(map(\lambda\nu.\, \mathcal{F}(\mathcal{G}[\![\mathrm{Q}^*]\!]\nu)\rho)\mathrm{N}^*)(\mathcal{F}[\![\mathrm{Q}^*]\!]\rho)$
$\mathcal{Q}[\![(\texttt{call}\ \mathrm{W}^*)\ \mathrm{Q}^*]\!] = \lambda\rho.\, call(\mathcal{W}^*[\![\mathrm{W}^*]\!]\rho)$
$\mathcal{Q}[\![(\texttt{return})\ \mathrm{Q}^*]\!] = \lambda\rho.\, return$
$\mathcal{Q}[\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{Q}^*]\!] = \lambda\rho.\, dispose\,\mathrm{N}(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)$

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

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

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

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

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

4.4 Machine Instruction Auxiliary Functions

$literal : E \to Q \to F$
$literal = \lambda\epsilon\pi.\, \lambda\epsilon^*\kappa.\, \pi\epsilon\epsilon^*\kappa$

$ignore : F \to Q$
$ignore = \lambda\phi.\, \lambda\epsilon\epsilon^*\kappa.\, \phi\epsilon^*\kappa$

$fetch : W \to Q \to F$
$fetch = \lambda\omega\pi.\, \lambda\epsilon^*\kappa.\, get\, \omega\epsilon^*\lambda\epsilon.\, \pi\epsilon\epsilon^*\kappa$

$setlocal : N \to Q \to Q$
$setlocal =$
$\quad \lambda\nu\pi.\, \lambda\epsilon\epsilon^*\kappa.\, \pi\, unspecified(takefirst\, \epsilon^*\nu \S \langle\epsilon\rangle \S dropfirst\, \epsilon^*(\nu+1))\kappa$

$initglobal : D \to Q \to Q$
$initglobal = \lambda\delta\pi.\, \lambda\epsilon\epsilon^*\kappa.\, initialize\, \delta\epsilon(\pi\ unspecified\, \epsilon^*\kappa)$

$setglobal : D \to Q \to Q$
$setglobal = \lambda\delta\pi.\, \lambda\epsilon\epsilon^*\kappa.\, assign\, \delta\epsilon(\pi\ unspecified\, \epsilon^*\kappa)$

$jumpfalse : F \to F \to Q$
$jumpfalse = \lambda\phi_0\phi_1.\, \lambda\epsilon\epsilon^*\kappa.\, \epsilon = false \to \phi_1\epsilon^*\kappa, \phi_0\epsilon^*\kappa$

$dispatch : F^* \to F \to Q$
$dispatch =$
$\quad \lambda\phi^*\phi.\, \lambda\epsilon\epsilon^*\kappa.$
$\qquad (0 \le \epsilon \mid \mathrm{R}) \wedge (\epsilon \mid \mathrm{R} < \#\phi^*) \to (\phi^* \downarrow (1+\epsilon \mid \mathrm{R}))\epsilon^*\kappa, \phi\epsilon^*\kappa$

$call : W^* \to Q$
$call = \lambda\omega^*.\, \lambda\epsilon\epsilon^*\kappa.\, obtain\, \omega^*\epsilon^*\lambda\epsilon^*.\, applicate\, \epsilon\epsilon^*\kappa$

$return : Q$
$return = \lambda\epsilon\epsilon^*\kappa.\, \kappa\epsilon$

$makecont : N \to Q \to F \to F$
$makecont = \lambda\nu\pi\phi.\, \lambda\epsilon^*\kappa.\, \nu = \#\epsilon^* \to \phi\epsilon^*\lambda\epsilon.\, \pi\epsilon\epsilon^*\kappa,$
$\qquad\qquad\qquad\qquad\qquad\qquad wrong$ "bad stack"

$entry : N \to F \to F$
$entry = \lambda\nu\phi.\, \lambda\epsilon^*\kappa.\, \nu = \#\epsilon^* \to \phi\epsilon^*\kappa, wrong$ "bad stack"

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

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

$$
\begin{array}{l}
add : W \to W \to Q \to F \\
add = \\
\quad \lambda\omega_0\omega_1\pi. \\
\qquad \lambda\epsilon^*\kappa.\, get\,\omega_0\epsilon^*\lambda\epsilon_0.\, get\,\omega_1\epsilon^*\lambda\epsilon_1.\, \pi((\epsilon_0 \mid \mathrm{R} + \epsilon_1 \mid \mathrm{R})\ \mathrm{in}\ E)\epsilon^*\kappa
\end{array}
$$

4.5 Additional Auxiliary Functions

$$
\begin{array}{l}
stackref : E^* \to N \to E \\
stackref = \lambda\epsilon^*\nu.\, \epsilon^* \downarrow (\nu + 1)
\end{array}
$$

$$
\begin{array}{l}
get : W \to E^* \to K \to C \\
get = \lambda\omega\epsilon^*\kappa.\, \omega \in D \to hold(\omega \mid D)\kappa, send(stackref\ \epsilon^*(\omega \mid N))\kappa
\end{array}
$$

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

$$
\begin{array}{l}
unspecs : N \to E^* \\
unspecs = \lambda\nu.\, \nu = 0 \to \langle\rangle, \langle unspecified\rangle \S unspecs(\nu - 1)
\end{array}
$$

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

$$
\begin{array}{l}
ismutable : \mathrm{Ide} \to T \\
ismutable = \text{Definition follows.}
\end{array}
$$

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

Chapter 5

The Syntax-Directed Compiler

The syntax-directed compiler is given as a function which maps Simple PreScheme programs into PreScheme Assembly Language. The function is presented using the same notation used to map syntactic phrases into semantic values. A syntactic domain is viewed as a kind of semantic domain which always has a countable number of elements. The compiler maps syntactic phrases into a syntactic kind of semantic domain.

The BNF used to define abstract syntax is viewed as another notation for specifying denotational domains. It specifies domains that are constructed using the product and disjoint sum operators. For example, the following fragment from the VLISP PreScheme semantics

$$\mathrm{Exp} \longrightarrow \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}\ \mathrm{E}^*) \mid (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}) \mid \ldots$$

is interpreted as

$$\begin{aligned} \mathrm{Exp} &= \mathrm{Con} + \mathrm{Ide} + \mathrm{App} + \mathrm{Lam} + \cdots \\ \mathrm{App} &= \mathrm{Exp} \times \mathrm{Exp}^* \\ \mathrm{Lam} &= \mathrm{Ide}^* \times \mathrm{Exp} \\ &\vdots \end{aligned}$$

The semantic brackets $[\![\cdot]\!]$ allow a familiar BNF-like notation for specifying elements of syntactic domains. For example, $\mathcal{E}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E})]\!]$ stands for $\mathcal{E}((\mathrm{I}^*, \mathrm{E})\ \mathrm{in}\ \mathrm{Exp})$.

The precise mapping from the BNF notation into domain elements is not given, however, for PreScheme Assembly Language, the semantic brackets $[\![\cdot]\!]$ usually denote elements in Code* so the following makes sense.

$$[\![(\texttt{return})\ \mathrm{Q}^*]\!] = [\![(\texttt{return})]\!] \S\ \mathrm{Q}^*$$

The functions which define the heart of the compiler[1] take four 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 number of locations on the stack that are already in use or have been reserved for future use. The fourth argument is the code to be appended after the code for the current expression.

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

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

$$\mathcal{CS}[\![\mathrm{K}]\!]\gamma\nu\mathrm{Q}^* = \mathit{mkliteral}\ \mathrm{KQ}^*$$

$$\mathcal{CS}[\![\mathrm{I}]\!]\gamma\nu\mathrm{Q}^* = \mathit{mkfetch}(\mathit{varref}\ \gamma\mathrm{I})\mathrm{Q}^*$$

When I^* contains only `lambda` bound variables,

$$\begin{aligned}
&\mathcal{CS}[\![(\mathrm{S}\ \mathrm{I}^*)]\!]\gamma\nu[\![(\texttt{return})\ \mathrm{Q}^*]\!] = \\
&\quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{mkcall}(\mathit{map}(\mathit{varref}\ \gamma)\mathrm{I}^*)\mathrm{Q}^*)
\end{aligned}$$

[1] The Simple PreScheme compiler is very similar to the VLISP byte-code compiler described in [2]. Both compilers have nearly the same architecture, however, their correctness proofs are different.

When S^* contains something other than `lambda` bound variables,

$$\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\nu[\![(\texttt{return})\ \mathrm{Q}^*]\!] = \mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ (\mathrm{S}\ \mathrm{I}^*))\ \mathrm{S}^*)]\!]\gamma\nu[\![(\texttt{return})\ \mathrm{Q}^*]\!]$$

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

$$\begin{array}{l}\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\nu\mathrm{Q}^* = \\ \quad \text{let} \\ \quad\quad \mathrm{Q}_0^* = \mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\nu[\![(\texttt{return})\ \mathrm{Q}^*]\!] \\ \quad \text{in} \\ \quad\quad mkmakecont\ \nu(\#\mathrm{Q}_0^* - \#\mathrm{Q}^*)\mathrm{Q}_0^* \\ \quad \text{when}\ \ isreturn\ \mathrm{Q}^* = false\end{array}$$

$$\begin{array}{l}\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!]\gamma\nu\mathrm{Q}^* = \\ \quad mkreserve\ \#\mathrm{I}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\nu + \#\mathrm{I}^*)(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma\nu\mathrm{Q}^*))\end{array}$$

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

$$\begin{array}{l}\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S}\ \mathrm{S}^*\ \mathrm{S}_0)]\!]\gamma\nu\mathrm{Q}^* = \\ \quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(mkignore(\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S}^*\ \mathrm{S}_0)]\!]\gamma\nu\mathrm{Q}^*))\end{array}$$

$$\begin{array}{l}\mathcal{CS}[\![(\texttt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma\nu\mathrm{Q}^* = \\ \quad \text{let} \\ \quad\quad \mathrm{Q}_2^* = \mathcal{CS}[\![\mathrm{S}_2]\!]\gamma\nu\mathrm{Q}^* \\ \quad\quad \mathrm{Q}_1^* = \mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\nu(maybemkjump\ \mathrm{Q}_2^*\mathrm{Q}^*) \\ \quad \text{in} \\ \quad\quad \mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\nu(mkjumpfalse(\#\mathrm{Q}_1^* - \#\mathrm{Q}_2^*)\mathrm{Q}_1^*)\end{array}$$

$$\mathcal{CS}[\![(\texttt{if}\ \#\texttt{f}\ \#\texttt{f})]\!]\gamma\nu\mathrm{Q}^* = mkunspecfied\ \mathrm{Q}^*$$

$$\begin{array}{l}\mathcal{CS}[\![(\texttt{case}\ \mathrm{S}\ \mathrm{C})]\!]\gamma\nu\mathrm{Q}^* = \\ \quad \text{let} \\ \quad\quad \mathrm{Q}^{**} = \mathcal{CC}[\![\mathrm{C}]\!]\gamma\nu\mathrm{Q}^* \\ \quad\quad \mathrm{Q}_1^* = \mathrm{Q}^{**} \downarrow 1 \\ \quad\quad \mathrm{Q}_0^* = mkunspecified(maybemkjump\ \mathrm{Q}_1^*\mathrm{Q}^*) \\ \quad \text{in} \\ \quad\quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(mkdispatch(map(\lambda\mathrm{Q}^*.\ \#\mathrm{Q}_0^* - \#\mathrm{Q}^*)\mathrm{Q}^{**})\mathrm{Q}_0^*)\end{array}$$

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

$$\mathcal{CS}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!]\gamma\nu\mathrm{Q}^* =$$
$$\quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{islocal}\,\gamma\mathrm{I} \to \mathit{mkwrong}\,\mathrm{Q}^*, \mathit{mkassign}\,\mathrm{I}\mathrm{Q}^*)$$

For `lambda` bound variables I_0 and I_1,

$$\mathcal{CS}[\![(+\ \mathrm{I}_0\ \mathrm{I}_1)]\!]\gamma\nu\mathrm{Q}^* = \mathit{mkadd}(\mathit{varref}\ \gamma\mathrm{I}_0)(\mathit{varref}\ \gamma\mathrm{I}_1)\mathrm{Q}^*$$

When S_0 or S_1 is something other than a `lambda` bound variable,[2]

$$\mathcal{CS}[\![(+\ \mathrm{S}_0\ \mathrm{S}_1)]\!]\gamma\nu\mathrm{Q}^* =$$
$$\quad \mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}_0\ \mathrm{I}_1)\ (+\ \mathrm{I}_0\ \mathrm{I}_1))\ \mathrm{S}_0\ \mathrm{S}_1)]\!]\gamma\mathrm{Q}^*$$

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

$$\mathcal{CS}^*[\![\mathrm{S}\ \mathrm{S}^*]\!]\gamma\nu\mathrm{Q}^* =$$
$$\quad \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{mkassign}(\nu - \#\mathrm{S} - 1)(\mathit{mkignore}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma\nu\mathrm{Q}^*)))$$

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

$$\mathcal{CC}[\![((\mathrm{K})\ \mathrm{S})\ \mathrm{C}]\!]\gamma\nu\mathrm{Q}^* =$$
$$\quad \text{let}$$
$$\quad\quad \mathrm{Q}^{**} = \mathcal{CC}[\![\mathrm{C}]\!]\gamma\nu\mathrm{Q}^*$$
$$\quad\quad \mathrm{Q}_1^* = \mathrm{Q}^{**} \downarrow 1$$
$$\quad \text{in}$$
$$\quad\quad \langle\mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{maybemkjump}\,\mathrm{Q}_1^*\mathrm{Q}^*)\rangle \S\ \mathrm{Q}^{**}$$

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

$$\mathcal{CB}[\![(\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S}))\ \mathrm{B}]\!] =$$
$$\quad \text{let}$$
$$\quad\quad \mathrm{Q}^* = \mathit{mkentry}\,\#\mathrm{I}^*(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma_0 0[\![(\texttt{return})]\!])$$
$$\quad \text{in}$$
$$\quad\quad [\![(\mathrm{I}\ \mathrm{Q}^*)]\!] \S\, \mathcal{CB}[\![\mathrm{B}]\!]$$

$$\mathcal{CE}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!] =$$
$$\quad \text{let}$$
$$\quad\quad \mathrm{B}' = \mathcal{CB}[\![\mathrm{B}]\!]$$
$$\quad\quad \mathrm{Q}^* = \mathit{mkentry}\,0(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 0[\![(\texttt{return})]\!])$$
$$\quad \text{in}$$
$$\quad\quad [\![(\texttt{letrec}\ (\mathrm{B}')\ \mathrm{Q}^*)]\!]$$

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

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

$$
\begin{array}{ll}
\mathit{mkliteral}\,\mathrm{KQ}^* & = [\![(\texttt{literal}\ \mathrm{K})\ \mathrm{Q}^*]\!] \\
\mathit{mkunspecified}\,\mathrm{Q}^* & = [\![(\texttt{unspecified})\ \mathrm{Q}^*]\!] \\
\mathit{mkignore}\,\mathrm{Q}^* & = [\![(\texttt{ignore})\ \mathrm{Q}^*]\!] \\
\mathit{mkfetch}\,\mathrm{WQ}^* & = [\![(\texttt{fetch}\ \mathrm{W})\ \mathrm{Q}^*]\!] \\
\mathit{mkassign}\,\mathrm{WQ}^* & = [\![(\texttt{assign}\ \mathrm{W})\ \mathrm{Q}^*]\!] \\
\mathit{mkwrong}\,\mathrm{IQ}^* & = [\![(\texttt{wrong})\ \mathrm{Q}^*]\!] \\
\mathit{mkjump}\,\mathrm{NQ}^* & = [\![(\texttt{jump}\ \mathrm{N})\ \mathrm{Q}^*]\!] \\
\mathit{mkjumpfalse}\,\mathrm{NQ}^* & = [\![(\texttt{jump-if-false}\ \mathrm{N})\ \mathrm{Q}^*]\!] \\
\mathit{mkdispatch}\,\mathrm{N}^*\mathrm{Q}^* & = [\![(\texttt{dispatch}\ \mathrm{N}^*)\ \mathrm{Q}^*]\!] \\
\mathit{mkcall}\,\mathrm{W}^*\mathrm{Q}^* & = [\![(\texttt{call}\ \mathrm{W}^*)\ \mathrm{Q}^*]\!] \\
\mathit{mkreturn}\,\mathrm{Q}^* & = [\![(\texttt{return})\ \mathrm{Q}^*]\!] \\
\mathit{mkmakecont}\,\mathrm{N}_0\mathrm{N}_1\mathrm{Q}^* & = [\![(\texttt{make-cont}\ \mathrm{N}_0\ \mathrm{N}_1)\ \mathrm{Q}^*]\!] \\
\mathit{mkentry}\,\mathrm{NQ}^* & = [\![(\texttt{entry}\ \mathrm{N})\ \mathrm{Q}^*]\!] \\
\mathit{mkreserve}\,\mathrm{NQ}^* & = [\![(\texttt{reserve}\ \mathrm{N})\ \mathrm{Q}^*]\!] \\
\mathit{mkdispose}\,\mathrm{NQ}^* & = [\![(\texttt{dispose}\ \mathrm{N})\ \mathrm{Q}^*]\!] \\
\mathit{mkadd}\,\mathrm{W}_0\mathrm{W}_1\mathrm{Q}^* & = [\![(\texttt{add}\ \mathrm{W}_0\ \mathrm{W}_1)\ \mathrm{Q}^*]\!]
\end{array}
$$

Figure 5.1: Code Generators

5.1 Auxiliary Compiler Functions

This section gives the properties of compile-time environments and defines the function *isreturn* and *maybemkjump*. The definition of code generators are given in Figure 5.1.

$\gamma_0 : U_c$

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

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

$$
\begin{array}{l}
\mathit{varref} : U_c \to \mathrm{Ide} \to \mathrm{Ref} \\
\mathit{varref}\ \gamma_0 \mathrm{I}_0 = \mathrm{I}_0 \\
\mathit{varref}\,(\mathit{extends}_c\ \gamma\langle \mathrm{I} \rangle \nu)\mathrm{I}_0 = (\mathrm{I} = \mathrm{I}_0 \to \nu, \mathit{varref}\ \gamma \mathrm{I}_0)
\end{array}
$$

$$
\begin{array}{l}
\mathit{isreturn} : \mathrm{Code}^* \to T \\
\mathit{isreturn} = \lambda \mathrm{Q}^*.\, \langle \mathrm{Q}^* \downarrow 1 \rangle = [\![(\texttt{return})]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathit{maybemkjump} : \mathrm{Code}^* \to \mathrm{Code}^* \to \mathrm{Code}^* \\
\mathit{maybemkjump} = \lambda \mathrm{Q}_1^* \mathrm{Q}_0^*.\, \mathit{isreturn}\, \mathrm{Q}_0^* \to \mathit{mkreturn}\, \mathrm{Q}_1^*, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mkjump}(\#\mathrm{Q}_1^* - \#\mathrm{Q}_0^*)\mathrm{Q}_1^*
\end{array}
$$

5.2 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., $\mathcal{P}'(\mathcal{CP}[\![\mathrm{P}]\!]) = \mathcal{P}[\![\mathrm{P}]\!]$. 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 assume that all procedures receive the correct number of arguments.

An overview of the theorems in this section follows. Theorems 13 through 15 show that the offset computed for branch instructions will cause a jump to the intended location. Theorem 16 shows the correctness of the environment created by a `let` expression. Theorem 17 and Theorem 18 show the correctness of the compiler on simple and sequences of simple expressions. Theorems 19 through 22 show the correctness of the parts of the compiler which compose multiple sequences of instructions into a complete program.

5.2.1 Code Linearity

Theorem 13 *If* $\mathit{isreturn}\, \mathrm{Q}^* = \mathit{false}$ *then* Q^* *is a suffix of* $\mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu\mathrm{Q}^*$.

Proof: Inspection of each function involved with the compilation of a simple expression shows that all append their results to the after code Q^*, with one exception: the case of compiling a tail-recursive call. By assumption, the after code does not begin with a `return` instruction, so the result of compiling a combination is a `make-cont` instruction appended to a sequence of instructions followed by a `call` instruction appended to Q^*. ∎

Theorem 14 Q^* *is a suffix of* $\mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu[\![(\texttt{return})\ \mathrm{Q}^*]\!]$.

Proof: Inspection of each function involved with the compilation of a simple expression shows that all append their results to the after code $[\![(\texttt{return})\ \mathrm{Q}^*]\!]$, with one exception: the case of compiling a tail-recursive call. In this case, the `return` instruction is replaced by a `call` instruction and Q^* is still a suffix. ∎

Theorem 15 $isreturn(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu\mathrm{Q}^*) = false$.

Proof: Inspection of each function involved with the compilation of a simple expression shows that the first instruction produced is never a `return` instruction. ∎

The following lemma is used to show the correctness of the code generated for conditional expressions. For example, the alternative or else part of an `if` expression is preceded by the code generated by *maybemkjump*.

Lemma 8

$$\lambda\epsilon.\ \mathcal{Q}(maybemkjump(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu\mathrm{Q}^*)\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa = \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa$$

Proof: Let $\mathrm{Q}_0^* = \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu\mathrm{Q}^*$. Assume $isreturn\,\mathrm{Q}^* = true$.

$$\begin{aligned}
&\lambda\epsilon.\ \mathcal{Q}(maybemkjump\,\mathrm{Q}_0^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\ \mathcal{Q}(mkreturn\,\mathrm{Q}_0^*)\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\ return\,\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

Assume $isreturn\,\mathrm{Q}^* = false$.

$$\begin{aligned}
&\lambda\epsilon.\ \mathcal{Q}(maybemkjump\,\mathrm{Q}_0^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\ \mathcal{Q}(mkjump(\#\mathrm{Q}_0^* - \#\mathrm{Q}^*)\mathrm{Q}_0^*)\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\ \mathcal{Q}(\mathcal{G}[\![\mathrm{Q}_0^*]\!](\#\mathrm{Q}_0^* - \#\mathrm{Q}^*))\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where step two is by use of Theorem 13, using the fact that when Q^* is a suffix of Q_0^*, $dropfirst\,\mathrm{Q}_0^*(\#\mathrm{Q}_0^* - \#\mathrm{Q}^*) = \mathrm{Q}^*$. ∎

5.2.2 Compile-Time Environments

Definition 11

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

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 9 shows that the composition of extending a compile time environment is equivalent to extending an environment with some items on the stack.

Lemma 9

$$compose\,\rho(extends_c\,\gamma\mathrm{I}^*\#\epsilon^*)(\epsilon^* \S \epsilon_0^*)$$
$$= extends(compose\,\rho\gamma(\epsilon^* \S \epsilon_0^*))\mathrm{I}^*\epsilon_0^*$$

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^*.

$$compose\,\rho(extends_c\,\gamma(\langle\mathrm{I}\rangle \S \mathrm{I}^*)\#\epsilon^*)(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*)$$
$$= compose\,\rho(extends_c(extends_c\,\gamma\langle\mathrm{I}\rangle\#\epsilon^*)\mathrm{I}^*(\#\epsilon^* + 1))(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*)$$
$$= extends(compose\,\rho(extends_c\,\gamma\langle\mathrm{I}\rangle\#\epsilon^*)(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*))\mathrm{I}^*\epsilon_0^*$$
$$= extends(compose\,\rho\gamma(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*)[\epsilon\ \mathrm{in}\ D/\mathrm{I}])\mathrm{I}^*\epsilon_0^*$$
$$= extends(compose\,\rho\gamma(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*))(\langle\mathrm{I}\rangle \S \mathrm{I}^*)(\langle\epsilon\rangle \S \epsilon_0^*)$$

where step one is by the definition of $extends_c$, 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. There is a natural number, call it $size\,\gamma$, which 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 $size\,\gamma$ without changing the composed environment. Lemma 10 formalizes this idea.

Definition 12

$$size : U_c \rightarrow N$$
$$size\,\gamma_0 = 0$$
$$size(extends_c\,\gamma\langle\mathrm{I}\rangle\nu) = max(\nu + 1, size\,\gamma)$$

Lemma 10 *compose* $\rho\gamma\epsilon^* = compose\,\rho\gamma(takefirst\,\epsilon^*(size\,\gamma))$

Proof: Consider the expression *compose* $\rho\gamma\epsilon^*\mathrm{I}$. When $islocal\,\gamma\mathrm{I} = false$, the lemma is trivial, so assume $islocal\,\gamma\mathrm{I} = true$ and assume $varref\,\gamma\mathrm{I} = \nu$. By the definition of *size*, $\nu < size\,\gamma$. Let $\nu_0 = size\,\gamma - \nu - 1$.

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

∎

Lemma 11 $size\,\gamma \le \nu$ *implies* $size(extends_c\,\gamma\mathrm{I}^*\nu) \le \#\mathrm{I}^* + \nu$.

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

$$\begin{aligned}
&size(extends_c\,\gamma(\langle\mathrm{I}\rangle\,\S\,\mathrm{I}^*)\nu)\\
&\quad = size(extends_c(extends_c\,\gamma\langle\mathrm{I}\rangle\nu)\mathrm{I}^*(\nu+1))
\end{aligned}$$

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

$$size(extends_c\,\gamma\langle\mathrm{I}\rangle\nu) \le \nu + 1$$

The definition of *size* shows the assumption is satisfied. ∎

Theorem 16 shows that the correct environment is produced when compiling a `let` expression.

Theorem 16 *Assume* $\#\epsilon^* \ge \#\mathrm{I}^*$. *Let* $\nu = \#\epsilon^* - \#\mathrm{I}^*$ *and assume* $size\,\gamma \le \nu$.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](extends_c\,\gamma\mathrm{I}^*\nu)\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](compose\,\rho(extends_c\,\gamma\mathrm{I}^*\nu)\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

implies

$$\begin{aligned}
&\mathcal{F}(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma\nu\mathrm{Q}^*)\rho\epsilon^*\kappa \\
&\quad = \mathit{applicate}\ (\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](\mathit{compose}\ \rho\gamma(\mathit{takefirst}\ \epsilon^*\nu))) \\
&\qquad\qquad (\mathit{dropfirst}\ \epsilon^*\nu) \\
&\qquad\qquad \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(\mathit{takefirst}\ \epsilon^*\nu)\kappa
\end{aligned}$$

Proof: Expanding the LHS of the equation gives

$$\begin{aligned}
&\mathcal{F}(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma\nu\mathrm{Q}^*)\rho\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathit{extends}_c\ \gamma\mathrm{I}^*\nu) \\
&\qquad\qquad (\nu + \#\mathrm{I}^*) \\
&\qquad\qquad (\mathit{isreturn}\ \mathrm{Q}^* \to \mathrm{Q}^*, \mathit{mkdispose}\ \#\mathrm{I}^*\mathrm{Q}^*))\rho\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\ \rho(\mathit{extends}_c\ \gamma\mathrm{I}^*\nu)\epsilon^*) \\
&\qquad\quad \lambda\epsilon.\ \mathcal{Q}(\mathit{isreturn}\ \mathrm{Q}^* \to \mathrm{Q}^*, \mathit{mkdispose}\ \#\mathrm{I}^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

by definition of $\mathcal{CL}$ and use of an assumption.

Expanding the RHS of the equation gives.

$$\begin{aligned}
&\mathit{applicate}\ (\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](\mathit{compose}\ \rho\gamma(\mathit{takefirst}\ \nu)\epsilon^*)) \\
&\qquad\quad (\mathit{dropfirst}\ \epsilon^*\nu) \\
&\qquad\quad \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(\mathit{takefirst}\ \epsilon^*\nu)\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}(\mathit{compose}\ \rho\gamma(\mathit{takefirst}\ \epsilon^*\nu))\mathrm{I}^*(\mathit{dropfirst}\ \epsilon^*\nu)) \\
&\qquad\quad \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(\mathit{takefirst}\ \epsilon^*\nu)\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}(\mathit{compose}\ \rho\gamma\epsilon^*)\mathrm{I}^*(\mathit{dropfirst}\ \epsilon^*\nu)) \\
&\qquad\quad \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(\mathit{takefirst}\ \epsilon^*\nu)\kappa
\end{aligned}$$

by definition of $\mathcal{L}$ and use of Lemma 10. Appealing to Lemma 9 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 $\mathit{isreturn}\ \mathrm{Q}^* = \mathit{true}$. Let $\mathrm{Q}^* = [\![(\texttt{return})\ \mathrm{Q}_0^*]\!]$.

$$\begin{aligned}
&\lambda\epsilon.\ \mathcal{Q}(\mathit{isreturn}\ \mathrm{Q}^* \to \mathrm{Q}^*, \mathit{mkdispose}\ \#\mathrm{I}^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\ \mathcal{Q}[\![(\texttt{return})\ \mathrm{Q}_0^*]\!]\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\ \mathit{return}\ \epsilon\epsilon^*\kappa \\
&\quad = \kappa \\
&\quad = \lambda\epsilon.\ \mathit{return}\ \epsilon(\mathit{takefirst}\ \epsilon^*\nu)\kappa \\
&\quad = \lambda\epsilon.\ \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(\mathit{takefirst}\ \epsilon^*\nu)\kappa
\end{aligned}$$

Assume $isreturn\,\mathrm{Q}^* = false$.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{Q}(isreturn\,\mathrm{Q}^* \to \mathrm{Q}^*, mkdispose\,\#\mathrm{I}^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{Q}(mkdispose\,\#\mathrm{I}^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,dispose\,\#\mathrm{I}^*(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(takefirst\,\epsilon^*(\#\epsilon^* - \#\mathrm{I}^*))\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon(takefirst\,\epsilon^*\nu)\kappa
\end{aligned}$$

∎

5.2.3 Simple Expressions

Before the main theorems are presented, a useful lemma about variable reference is given.

Lemma 12

$$get(\mathcal{W}(varref\,\gamma\mathrm{I})\rho)\epsilon^*\kappa = \mathcal{E}[\![\mathrm{I}]\!](compose\,\rho\gamma\epsilon^*)\kappa$$

Proof: Assume $islocal\,\gamma\mathrm{I} = true$, and $varref\,\gamma\mathrm{I} = \nu$.

$$\begin{aligned}
&get(\nu\ \mathrm{in}\ W)\epsilon^*\kappa \\
&\quad = send(stackref\,\epsilon^*\nu)\kappa \\
&\quad = hold(stackref\,\epsilon^*\nu\ \mathrm{in}\ D)\kappa \\
&\quad = hold(compose\,\rho\gamma\epsilon^*\mathrm{I})\kappa \\
&\quad = \mathcal{E}[\![\mathrm{I}]\!](compose\,\rho\gamma\epsilon^*)\kappa
\end{aligned}$$

Assume $islocal\,\gamma\mathrm{I} = false$, and $varref\,\gamma\mathrm{I} = \mathrm{I}$.

$$\begin{aligned}
&get(lookup\,\rho\mathrm{I}\ \mathrm{in}\ W)\epsilon^*\kappa \\
&\quad = get(\rho\mathrm{I}\ \mathrm{in}\ W)\epsilon^*\kappa \\
&\quad = hold(\rho\mathrm{I})\kappa \\
&\quad = hold(compose\,\rho\gamma\epsilon^*\mathrm{I})\kappa \\
&\quad = \mathcal{E}[\![\mathrm{I}]\!](compose\,\rho\gamma\epsilon^*)\kappa
\end{aligned}$$

∎

Theorem 17 *Assume* $size\,\gamma \le \#\epsilon^*$.

$$\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa = \mathcal{E}[\![\mathrm{S}]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa$$

Theorem 18 *Assume size* $\gamma \leq \#\epsilon^*$.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{Q}^*)\rho(\epsilon^* \S \mathit{unspecs}\,\#\mathrm{S}^*)\kappa \\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{Q}^*]\!]\rho(\epsilon^* \S \epsilon_0^*)\kappa
\end{aligned}$$

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:

$$\begin{aligned}
&\text{Let } \mathrm{Q}_0^* = \mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + 1 + \#\mathrm{S}^*)\mathrm{Q}^* \\
&\mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}\ \mathrm{S}^*]\!]\gamma(\#\epsilon^* + 1 + \#\mathrm{S}^*)\mathrm{Q}^*)\rho(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma(\#\epsilon^* + 1 + \#\mathrm{S}^*)(\mathit{mkassign}\,\#\epsilon^*(\mathit{mkignore}\,\mathrm{Q}_0^*))) \\
&\qquad \rho(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*)) \\
&\qquad\quad \lambda\epsilon.\,\mathcal{Q}(\mathit{mkassign}\,\#\epsilon^*(\mathit{mkignore}\,\mathrm{Q}_0^*)) \\
&\qquad\qquad \rho\epsilon(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\quad \lambda\epsilon.\,\mathcal{Q}(\mathit{mkassign}\,\#\epsilon^*(\mathit{mkignore}\,\mathrm{Q}_0^*)) \\
&\qquad\qquad \rho\epsilon(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa
\end{aligned}$$

where step one is by definition of $\mathcal{CS}^*$, step two is by use of Theorem 17 as an induction hypothesis, and step three is by use of Lemma 10.

By use of the definition of $\mathcal{E}^*$, the RHS of Theorem 18 becomes

$$\begin{aligned}
&\mathcal{E}^*[\![\mathrm{S}\ \mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)(\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{Q}^*]\!]\rho(\epsilon^* \S \epsilon_0^*)\kappa) \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\quad \lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{Q}^*]\!]\rho(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*)\kappa)
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{Q}(\mathit{mkassign}\,\#\epsilon^*(\mathit{mkignore}\,\mathrm{Q}_0^*))\rho\epsilon(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa \\
&\quad = \lambda\epsilon.\,\mathit{setlocal}\,\#\epsilon^*(\mathcal{Q}(\mathit{mkignore}\,\mathrm{Q}_0^*)\rho)\epsilon(\epsilon^* \S \mathit{unspecs}(1 + \#\mathrm{S}^*))\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{Q}(\mathit{mkignore}\,\mathrm{Q}_0^*)\rho\,\mathit{unspecified}(\epsilon^* \S \langle\epsilon\rangle \S \mathit{unspecs}\,\#\mathrm{S}^*)\kappa \\
&\quad = \lambda\epsilon.\,\mathit{ignore}(\mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho)\,\mathit{unspecified}(\epsilon^* \S \langle\epsilon\rangle \S \mathit{unspecs}\,\#\mathrm{S}^*)\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho(\epsilon^* \S \langle\epsilon\rangle \S \mathit{unspecs}\,\#\mathrm{S}^*)\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma(\epsilon^* \S \langle\epsilon\rangle))\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{Q}^*]\!]\rho(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*)\kappa \\
&\quad = \lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{S}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\,\mathcal{F}[\![\mathrm{Q}^*]\!]\rho(\epsilon^* \S \langle\epsilon\rangle \S \epsilon_0^*)\kappa
\end{aligned}$$

where step one is by definition of *mkassign*, step two is by definition of *setlocal*, step three is by definition of *mkignore*, step four is by definition of *ignore*, step five is by use of Theorem 18 as an induction hypothesis, and step six is by use of Lemma 10.

Theorem 17 is demonstrated by performing an exhaustive case analysis on each type of simple expression.

Case of constants:

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{K}]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathit{mkliteral}\,\mathrm{KQ}^*)\rho\epsilon^*\kappa\\
&\quad = \mathit{literal}(\mathcal{K}[\![\mathrm{K}]\!])(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)\epsilon^*\kappa\\
&\quad = \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho(\mathcal{K}[\![\mathrm{K}]\!])\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{K}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

Case of variables:

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![\mathrm{I}]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathit{mkfetch}(\mathit{varref}\ \gamma\mathrm{I})\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathit{fetch}(\mathcal{W}(\mathit{varref}\ \gamma\mathrm{I})\rho)(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)\epsilon^*\kappa\\
&\quad = \mathit{get}(\mathcal{W}(\mathit{varref}\ \gamma\mathrm{I})\rho)\epsilon^*\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{I}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step is by Lemma 12.

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.

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![(\mathrm{S}\ \mathrm{I}^*)]\!]\gamma\#\epsilon^*[\![(\texttt{return})\ \mathrm{Q}^*]\!])\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\#\epsilon^*(\mathit{mkcall}(\mathit{map}(\mathit{varref}\ \gamma)\mathrm{I}^*)\mathrm{Q}^*))\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathit{mkcall}(\mathit{map}(\mathit{varref}\ \gamma)\mathrm{I}^*)\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![(\mathrm{S}\ \mathrm{I}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![(\texttt{return})\ \mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S}\ \mathrm{I}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{return}\,\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S}\ \mathrm{I}^*)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{I}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon^*.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{I}^*]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon^*.\,\mathit{applicate}\,\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step is justified because `lambda` bound variables are invariable.

The LHS and the RHS are equal when their continuations are the same.

$$\begin{aligned}
&\lambda\epsilon.\, \mathcal{Q}[\![mkcall(map(varref\ \gamma)\mathrm{I}^*)\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\, call(\mathcal{W}^*(map(varref\ \gamma)\mathrm{I}^*)\rho)\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\, obtain(\mathcal{W}^*(map(varref\ \gamma)\mathrm{I}^*)\rho)\epsilon^*\lambda\epsilon^*.\, applicate\,\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\, \mathcal{E}^*[\![\mathrm{I}^*]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon^*.\, applicate\,\epsilon\epsilon^*\kappa
\end{aligned}$$

where step three is by an easy to prove lemma which is analogous to Lemma 12.

Case of a non-tail-recursive call ($isreturn\ \mathrm{Q}^* = false$):

$$\begin{aligned}
&\text{Let } \mathrm{Q}_0^* = \mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\#\epsilon^*[\![(\texttt{return})\ \mathrm{Q}^*]\!]\\
&\mathcal{F}(\mathcal{CS}[\![(\mathrm{S}\ \mathrm{S}^*)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(mkmakecont\,\#\epsilon^*(\#\mathrm{Q}_0^* - \#\mathrm{Q}^*)\mathrm{Q}_0^*)\rho\epsilon^*\kappa\\
&\quad = makecont\,\#\epsilon^*(\mathcal{Q}(\mathcal{G}[\![\mathrm{Q}_0^*]\!](\#\mathrm{Q}_0^* - \#\mathrm{Q}^*))\rho)(\mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho)\epsilon^*\kappa\\
&\quad = makecont\,\#\epsilon^*(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)(\mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho)\epsilon^*\kappa\\
&\quad = \mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho\epsilon^*\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![(\mathrm{S}\ \mathrm{S}^*)]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where step three is true because Q_0^* is the result of appending a tail-recursive call to Q^*.

Case of a `let` expression:

$$\begin{aligned}
&\text{Let } \mathrm{Q}_0^* = \mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma\#\epsilon^*\mathrm{Q}^*\\
&\mathcal{F}(\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(mkreserve\,\#\mathrm{I}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{I}^*)\mathrm{Q}_0^*))\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(mkreserve\,\#\mathrm{S}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{Q}_0^*))\rho\epsilon^*\kappa\\
&\quad = reserve\,\#\mathrm{S}^*(\mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{Q}_0^*)\rho)\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\#\epsilon^* + \#\mathrm{S}^*)\mathrm{Q}_0^*)\rho(\epsilon^* \S\ unspec\,\#\mathrm{S}^*)\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon_0^*.\, \mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho(\epsilon^* \S\ \epsilon_0^*)\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](compose\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon_0^*.\, applicate\,(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](compose\,\rho\gamma\epsilon^*))\\
&\qquad\qquad\qquad\qquad \epsilon_0^*\\
&\qquad\qquad\qquad\qquad \lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}^*[\![\mathrm{S}^*]\!](compose\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon_0^*.\, \mathcal{E}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](compose\,\rho\gamma\epsilon^*)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda\epsilon.\, applicate\,\epsilon\epsilon_0^*\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!](compose\,\rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\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 *mkreserve*, step four is by use of Theorem 18 as an induction hypothesis, step five is by use of Theorem 16 and Lemma 11 to prove one of the assumptions of Theorem 16, step six uses the definition `lambda` abstraction, and step seven 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:

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![(\texttt{begin S S}^*\ \mathrm{S}_0)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\#\epsilon^*(\mathit{mkignore}(\mathcal{CS}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)))\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\,\mathcal{Q}(\mathit{mkignore}(\mathcal{CS}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!]\gamma\#\epsilon^*\mathrm{Q}^*))\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

The RHS of Theorem 17 becomes

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{begin S S}^*\ \mathrm{S}_0)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\,\mathcal{E}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{Q}(\mathit{mkignore}(\mathcal{CS}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!]\gamma\#\epsilon^*\mathrm{Q}^*))\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathit{ignore}(\mathcal{F}(\mathcal{CS}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho)\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathcal{F}(\mathcal{CS}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathcal{E}[\![(\texttt{begin S}^*\ \mathrm{S}_0)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step is another use of an induction hypothesis.

Case of an `if` expression:

$$\begin{aligned}
&\text{Let } \mathrm{Q}_2^* = \mathcal{CS}[\![\mathrm{S}_2]\!]\gamma\#\epsilon^*\mathrm{Q}^*\\
&\qquad \mathrm{Q}_1^* = \mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\#\epsilon^*(\mathit{maybemkjump}\,\mathrm{Q}_2^*\mathrm{Q}^*)\\
&\mathcal{F}(\mathcal{CS}[\![(\texttt{if S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\#\epsilon^*(\mathit{mkjumpfalse}(\#\mathrm{Q}_1^* - \#\mathrm{Q}_2^*)\mathrm{Q}_1^*))\rho\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}_0]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}(\mathit{mkjumpfalse}(\#\mathrm{Q}_1^* - \#\mathrm{Q}_2^*)\mathrm{Q}_1^*)\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{if S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad = \mathcal{E}[\![\mathrm{S}_0]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad \lambda\epsilon.\,\epsilon = \mathit{false} \rightarrow \mathcal{E}[\![\mathrm{S}_2]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa,\\
&\qquad\qquad\qquad \mathcal{E}[\![\mathrm{S}_1]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same.

$$\begin{aligned}
&\lambda\epsilon.\, \mathcal{Q}(mkjumpfalse(\#\mathrm{Q}_1^* - \#\mathrm{Q}_2^*)\mathrm{Q}_1^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, jumpfalse(\mathcal{F}[\![\mathrm{Q}_1^*]\!]\rho)(\mathcal{F}(\mathcal{G}[\![\mathrm{Q}_1^*]\!](\#\mathrm{Q}_1^* - \#\mathrm{Q}_2^*))\rho)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, jumpfalse(\mathcal{F}[\![\mathrm{Q}_1^*]\!]\rho)(\mathcal{F}[\![\mathrm{Q}_2^*]\!]\rho)\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \epsilon = false \to \mathcal{F}[\![\mathrm{Q}_2^*]\!]\rho\epsilon^*\kappa, \mathcal{F}[\![\mathrm{Q}_1^*]\!]\rho\epsilon^*\kappa
\end{aligned}$$

where step two is justified by use of Theorem 13. It is applicable because by Theorem 15, the compiler for simple expressions $\mathcal{CS}$ can never generate a code sequence which begins with the `return` instruction.

The fact that

$$\mathcal{F}[\![\mathrm{Q}_2^*]\!]\rho\epsilon^*\kappa = \mathcal{E}[\![\mathrm{Q}_2^*]\!](compose\ \rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa$$

follows from an induction hypothesis as does

$$\begin{aligned}
&\mathcal{F}[\![\mathrm{Q}_1^*]\!]\rho\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{Q}_1^*]\!](compose\ \rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}(maybemkjump\ \mathrm{Q}_2^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{Q}_1^*]\!](compose\ \rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step is by use of Lemma 8.

Case of unspecified:

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![(\texttt{if\ \#f\ \#f})]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa \\
&\quad = \mathcal{F}(mkunspecified\ \mathrm{Q}^*)\rho\epsilon^*\kappa \\
&\quad = literal\ unspecified(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)\epsilon^*\kappa \\
&\quad = \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\ unspecified\ \epsilon^*\kappa \\
&\quad = send\ unspecified\ \lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\#\texttt{f}]\!](compose\ \rho\gamma\epsilon^*) \\
&\qquad\qquad \lambda\epsilon.\, \epsilon = false \to send\ unspecified\ \lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa, \\
&\qquad\qquad\qquad \mathcal{E}[\![\#\texttt{f}]\!](compose\ \rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![(\texttt{if\ \#f\ \#f})]\!](compose\ \rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

Case of `case` expressions:

$$\begin{aligned}
&\text{Let } \mathrm{Q}^{**} = \mathcal{CC}[\![((0)\ \mathrm{S}_0)\ldots((\nu)\ \mathrm{S}_\nu)]\!]\gamma\#\epsilon^*\mathrm{Q}^* \\
&\qquad \mathrm{Q}_1^* = \mathrm{Q}^{**}\downarrow 1 \\
&\qquad \mathrm{Q}_0^* = mkunspecified(maybemkjump\ \mathrm{Q}_1^*\mathrm{Q}^*) \\
&\mathcal{F}(\mathcal{CS}[\![(\texttt{case}\ \mathrm{S}\ ((0)\ \mathrm{S}_0)\ldots((\nu)\ \mathrm{S}_\nu))]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\#\epsilon^*(mkdispatch(map(\lambda\mathrm{Q}^*.\, \#\mathrm{Q}_0^* - \#\mathrm{Q}^*)\mathrm{Q}^{**})\mathrm{Q}_0^*)) \\
&\qquad \rho\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](compose\ \rho\gamma\epsilon^*) \\
&\qquad\quad \lambda\epsilon.\, \mathcal{Q}(mkdispatch(map(\lambda\mathrm{Q}^*.\, \#\mathrm{Q}_0^* - \#\mathrm{Q}^*)\mathrm{Q}^{**})\mathrm{Q}_0^*)\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

A `case` expression is a derived expression which is defined in terms of a sequence of `if` expressions so that

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{case S } ((0)\ \mathrm{S}_0)\ldots((\nu)\ \mathrm{S}_\nu))]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa\\
&\quad=\mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad\lambda\epsilon.\,\epsilon \mid R=0\to\mathcal{E}[\![\mathrm{S}_0]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa,\\
&\qquad\qquad\qquad\vdots\\
&\qquad\qquad\epsilon \mid R=\nu\to\mathcal{E}[\![\mathrm{S}_\nu]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa,\\
&\qquad\qquad\mathit{send\ unspecified}\,\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same, however, before investigating the continuations, the `case` clauses are considered.

$$\begin{aligned}
&\mathcal{F}(\mathrm{Q}^{**}\downarrow(\nu+1))\rho\epsilon^*\kappa\\
&\quad=\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_\nu]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa\\
&\quad=\mathcal{E}[\![\mathrm{S}_\nu]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

For $\nu_0<\nu$

$$\begin{aligned}
&\mathcal{F}(\mathrm{Q}^{**}\downarrow(\nu_0+1))\rho\epsilon^*\kappa\\
&\quad=\mathcal{F}(\mathcal{CS}[\![\mathrm{S}_{\nu_0}]\!]\gamma\#\epsilon^*(\mathit{maybemkjump}(\mathrm{Q}^{**}\downarrow(\nu_0+2))\mathrm{Q}^*))\rho\epsilon^*\kappa\\
&\quad=\mathcal{E}[\![\mathrm{S}_{\nu_0}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\\
&\qquad\qquad\lambda\epsilon.\,\mathcal{Q}(\mathit{maybemkjump}(\mathrm{Q}^{**}\downarrow(\nu_0+2))\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa\\
&\quad=\mathcal{E}[\![\mathrm{S}_{\nu_0}]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step follows directly from Theorem 13 only when $\nu_0=\nu-1$. When there are at least three clauses, $\mathrm{Q}^{**}\downarrow(\nu+1)$ is a suffix of $\mathrm{Q}^{**}\downarrow\nu$, but since Q^* is a suffix of $\mathrm{Q}^{**}\downarrow(\nu+1)$, it is also a suffix of $\mathrm{Q}^{**}\downarrow\nu$ as the suffix relation is transitive. This argument can be repeated for each $\nu_0<\nu$ justifying the last proof step.

$$\begin{aligned}
&\mathcal{F}[\![\mathrm{Q}_0^*]\!]\rho\epsilon^*\kappa\\
&\quad=\mathcal{F}(\mathit{mkunspecified}(\mathit{maybemkjump}\,\mathrm{Q}_1^*\mathrm{Q}^*))\rho\epsilon^*\kappa\\
&\quad=\mathit{literal\ unspecified}(\mathcal{Q}(\mathit{maybemkjump}\,\mathrm{Q}_1^*\mathrm{Q}^*)\rho)\epsilon^*\kappa\\
&\quad=\mathcal{Q}(\mathit{maybemkjump}\,\mathrm{Q}_1^*\mathrm{Q}^*)\rho\ \mathit{unspecified}\,\epsilon^*\kappa\\
&\quad=\mathit{send\ unspecified}\,\lambda\epsilon.\,\mathcal{Q}(\mathit{maybemkjump}\,\mathrm{Q}_1^*\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa\\
&\quad=\mathit{send\ unspecified}\,\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

where the last step follows from an argument similar to the one used for the clauses.

The next step is to show that the computed offsets are correct, that is, for $\nu_0 \leq \nu$

$$\mathcal{G}[\![\mathrm{Q}_0^*]\!](\#\mathrm{Q}_0^* - \#(\mathrm{Q}^{**} \downarrow (\nu_0 + 1))) = \mathrm{Q}^{**} \downarrow (\nu_0 + 1).$$

For $\nu_0 = 0$, it is obvious that $\mathrm{Q}^{**} \downarrow 1$ is a suffix of Q_0^*. For $0 < \nu_0 \leq \nu$, $\mathcal{CC}$ precedes $\mathrm{Q}^{**} \downarrow (\nu_0 + 1)$ with either a `return`, `call`, or a `jump` instruction and $\mathrm{Q}^{**} \downarrow (\nu_0 + 1)$ itself cannot begin with a `return` instruction by Theorem 15. Therefore, $\mathrm{Q}^{**} \downarrow (\nu_0 + 1)$ is a suffix of Q_0^*.

Case of assignment for identifiers whose name begins and ends with an asterisk and is at least three characters long ($\mathit{ismutable}\,\mathrm{I} = \mathit{true}$):

$$\begin{aligned}
&\mathcal{F}(\mathcal{CS}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!]\gamma\#\epsilon^*\mathrm{Q}^*)\rho\epsilon^*\kappa \\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma\#\epsilon^*(\mathit{islocal}\,\gamma\mathrm{I} \to \mathit{mkwrong}\,\mathrm{Q}^*, \mathit{mkassign}\,\mathrm{IQ}^*))\rho\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\qquad \lambda\epsilon.\, \mathcal{Q}(\mathit{islocal}\,\gamma\mathrm{I} \to \mathit{mkwrong}\,\mathrm{Q}^*, \mathit{mkassign}\,\mathrm{IQ}^*)\rho\epsilon\epsilon^*\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!](\mathit{compose}\,\rho\gamma\epsilon^*)\lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa \\
&\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho\gamma\epsilon^*) \\
&\qquad\qquad \lambda\epsilon.\, \mathit{assign}\,(\mathit{lookup}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I}) \\
&\qquad\qquad\qquad\qquad \epsilon \\
&\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\ \lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa)
\end{aligned}$$

The LHS and the RHS are equal when their continuations are the same. Assume $\mathit{islocal}\,\gamma\mathrm{I} = \mathit{true}$ and $\mathit{varref}\ \gamma\mathrm{I} = \nu$.

$$\begin{aligned}
&\lambda\epsilon.\, \mathcal{Q}(\mathit{islocal}\,\gamma\mathrm{I} \to \mathit{mkwrong}\,\mathrm{Q}^*, \mathit{mkassign}\,\mathrm{IQ}^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathcal{Q}(\mathit{mkwrong}\,\mathrm{Q}^*)\rho\epsilon\epsilon^*\kappa \\
&\quad = \lambda\epsilon.\, \mathit{wrong}\ \text{“assignment of an immutable variable”} \\
&\quad = \lambda\epsilon.\, \mathit{assign}(\mathit{stackref}\ \epsilon^*\nu\ \mathrm{in}\ D)\epsilon(\mathit{send}\ \mathit{unspecified}\ \lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa) \\
&\quad = \lambda\epsilon.\, \mathit{assign}\,(\mathit{lookup}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I}) \\
&\qquad\qquad\qquad \epsilon \\
&\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\ \lambda\epsilon.\, \mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa)
\end{aligned}$$

where step two is justified by the fact that $(\mathit{stackref}\ \epsilon^*\nu\ \mathrm{in}\ D) \notin L$.

Assume $\mathit{islocal}\,\gamma\mathrm{I} = \mathit{false}$.

$$\begin{aligned}
&\lambda\epsilon.\,\mathcal{Q}(\mathit{islocal}\,\gamma\mathrm{I} \to \mathit{mkwrong}\,\mathrm{Q}^*, \mathit{mkassign}\,\mathrm{IQ}^*)\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathcal{Q}(\mathit{mkassign}\,\mathrm{IQ}^*)\rho\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathit{setglobal}(\mathit{lookup}\,\rho\mathrm{I})(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho)\epsilon\epsilon^*\kappa\\
&\quad = \lambda\epsilon.\,\mathit{assign}(\mathit{lookup}\,\rho\mathrm{I})\epsilon(\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\;\mathit{unspecified}\,\epsilon^*\kappa)\\
&\quad = \lambda\epsilon.\,\mathit{assign}(\mathit{lookup}\,\rho\mathrm{I})\epsilon(\mathit{send}\;\mathit{unspecified}\,\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa)\\
&\quad = \lambda\epsilon.\,\mathit{assign}\,(\mathit{lookup}(\mathit{compose}\,\rho\gamma\epsilon^*)\mathrm{I})\\
&\qquad\qquad\qquad \epsilon\\
&\qquad\qquad\qquad (\mathit{send}\;\mathit{unspecified}\,\lambda\epsilon.\,\mathcal{Q}[\![\mathrm{Q}^*]\!]\rho\epsilon\epsilon^*\kappa)
\end{aligned}$$

The case of $\mathit{ismutable}\,\mathrm{I} = \mathit{false}$ is similar. ∎

5.2.4 Defined and `letrec`-Bound Variables

Lemma 13

$$\begin{aligned}
&\mathcal{F}(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma_0 0[\![(\texttt{return})]\!])\rho\ \text{in}\ E\\
&\quad = \mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\rho
\end{aligned}$$

Proof:

$$\begin{aligned}
&\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\rho \mid F\\
&\quad = \lambda\epsilon^*\kappa.\,\#\epsilon^* = \#\mathrm{I}^* \to \mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}\,\rho\mathrm{I}^*\epsilon^*)\kappa,\\
&\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}\\
&\quad = \lambda\epsilon^*\kappa.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}\,\rho\mathrm{I}^*\epsilon^*)\kappa\\
&\quad = \lambda\epsilon^*\kappa.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{extends}(\mathit{compose}\,\rho\gamma_0\epsilon^*)\mathrm{I}^*\epsilon^*)\kappa\\
&\quad = \lambda\epsilon^*\kappa.\,\mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\,\rho(\mathit{extends}_c\,\gamma_0\mathrm{I}^*0)\epsilon^*)\kappa\\
&\quad = \lambda\epsilon^*\kappa.\,\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathit{extends}_c\,\gamma_0\mathrm{I}^*0)\#\mathrm{I}^*[\![(\texttt{return})]\!])\rho\epsilon^*\kappa\\
&\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!](\mathit{extends}_c\,\gamma_0\mathrm{I}^*0)\#\mathrm{I}^*[\![(\texttt{return})]\!])\rho\\
&\quad = \mathcal{F}(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma_0 0[\![(\texttt{return})]\!])\rho
\end{aligned}$$

where step two is by use of Lemma 13 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 9, and step five is by Theorem 17. ∎

Theorem 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.

$$
\begin{array}{l}
\text{Let } \mathrm{Q}^* = \mathit{mkentry}\,\#\mathrm{I}^*(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma_0 0[\![(\texttt{return})]\!]) \\
\mathcal{B}'(\mathcal{CB}[\![(\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S}))\ \mathrm{B}]\!]) \\
\quad = \mathcal{B}'([\![(\mathrm{I}\ \mathrm{Q}^*)]\!] \S\, \mathcal{CB}[\![\mathrm{B}]\!]) \\
\quad = \lambda \mathrm{I}_0^*\rho\epsilon^*.\langle(\mathcal{F}[\![\mathrm{Q}^*]\!](\mathit{extends}\ \rho\mathrm{I}_0^*\epsilon^*))\ \mathrm{in}\ E\rangle \S\, \mathcal{B}'(\mathcal{CB}[\![\mathrm{B}]\!])\mathrm{I}_0^*\rho\epsilon^* \\
\quad = \lambda \mathrm{I}_0^*\rho\epsilon^*.\langle(\mathcal{F}[\![\mathrm{Q}^*]\!](\mathit{extends}\ \rho\mathrm{I}_0^*\epsilon^*))\ \mathrm{in}\ E\rangle \S\, \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^* \\
\quad = \lambda \mathrm{I}_0^*\rho\epsilon^*.\langle(\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](\mathit{extends}\ \rho\mathrm{I}_0^*\epsilon^*))\ \mathrm{in}\ E\rangle \S\, \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^* \\
\quad = \mathcal{B}[\![(\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S}))\ \mathrm{B}]\!]
\end{array}
$$

where step three is by use of the induction hypothesis and step four is by use of Lemma 13 and the fact that Simple PreScheme programs are strongly typed so that one can ignore the `entry` instruction. ∎

Theorem 20 $\mathcal{E}'(\mathcal{CE}[\![\mathrm{E}]\!])\rho\langle\rangle\kappa = \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa$

$$
\begin{array}{l}
\text{Let } \mathrm{B}' = \mathcal{CB}[\![\mathrm{B}]\!] \\
\qquad \mathrm{Q}^* = \mathit{mkentry}\,0(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 0[\![(\texttt{return})]\!]) \\
\mathcal{E}'(\mathcal{CE}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!])\rho\langle\rangle\kappa \\
\quad = \mathcal{E}'[\![(\texttt{letrec}\ (\mathrm{B}')\ \mathrm{Q}^*)]\!]\rho\langle\rangle\kappa \\
\quad = \mathcal{F}[\![\mathrm{Q}^*]\!](\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}']\!])(\mathit{fix}(\mathcal{B}'[\![\mathrm{B}']\!](\mathcal{I}'[\![\mathrm{B}']\!])\rho)))\langle\rangle\kappa \\
\quad = \mathcal{F}[\![\mathrm{Q}^*]\!](\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho)))\langle\rangle\kappa
\end{array}
$$

where the last step is justified by Theorem 19.

$$
\begin{array}{l}
\text{Let } \rho' = \mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho)) \\
\mathcal{F}[\![\mathrm{Q}^*]\!]\rho'\langle\rangle\kappa \\
\quad = \mathcal{F}(\mathit{mkentry}\,0(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 0[\![(\texttt{return})]\!]))\rho'\langle\rangle\kappa \\
\quad = \mathit{entry}\,0(\mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 0[\![(\texttt{return})]\!])\rho')\langle\rangle\kappa \\
\quad = \mathcal{F}(\mathcal{CS}[\![\mathrm{S}]\!]\gamma_0 0[\![(\texttt{return})]\!])\rho'\langle\rangle\kappa \\
\quad = \mathcal{E}[\![\mathrm{S}]\!](\mathit{compose}\ \rho'\gamma_0\langle\rangle)\lambda\epsilon.\,\mathcal{Q}[\![(\texttt{return})]\!]\rho'\epsilon\langle\rangle\kappa \\
\quad = \mathcal{E}[\![\mathrm{S}]\!]\rho'\lambda\epsilon.\,\mathcal{Q}[\![(\texttt{return})]\!]\rho'\epsilon\langle\rangle\kappa \\
\quad = \mathcal{E}[\![\mathrm{S}]\!]\rho'\kappa \\
\quad = \mathcal{E}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!]\rho\kappa
\end{array}
$$

where step four is by use of Theorem 17.

Theorem 21 $\mathcal{D}'(\mathcal{CP}[\![\mathrm{P}]\!]) = \mathcal{D}[\![\mathrm{P}]\!]$

Proof:

$$\begin{aligned}
&\mathcal{D}'(\mathcal{CP}[\![(\texttt{letrec (B) S})]\!])\rho\kappa \\
&\quad = \mathcal{E}'(\mathcal{CP}[\![(\texttt{letrec (B) S})]\!])\rho\langle\rangle\kappa \\
&\quad = \mathcal{E}[\![(\texttt{letrec (B) S})]\!]\rho\kappa \\
&\quad = \mathcal{D}[\![(\texttt{letrec (B) S})]\!]\rho\kappa
\end{aligned}$$

where step two is by Theorem 20.

$$\begin{aligned}
&\mathcal{D}'(\mathcal{CP}[\![(\texttt{define I) P}]\!])\rho\kappa\sigma \\
&\quad = \mathcal{D}'([\![(\texttt{define I})]\!] \S\, \mathcal{CP}[\![\mathrm{P}]\!])\rho\kappa\sigma \\
&\quad = \mathcal{D}'(\mathcal{CP}[\![\mathrm{P}]\!])(\rho[(new\,\sigma)\text{ in }D/\mathrm{I}])\kappa(update(new\,\sigma)\,undefined\,\sigma) \\
&\quad = \mathcal{D}[\![\mathrm{P}]\!](\rho[(new\,\sigma)\text{ in }D/\mathrm{I}])\kappa(update(new\,\sigma)\,undefined\,\sigma) \\
&\quad = \mathcal{D}[\![(\texttt{define I) P}]\!]
\end{aligned}$$

where step three is by use of the induction hypothesis. ∎

Theorem 22 $\mathcal{P}'(\mathcal{CP}[\![\mathrm{P}]\!]) = \mathcal{P}[\![\mathrm{P}]\!]$

Proof: By use of Theorem 21.

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

∎

Chapter 6

Results, Future Work, and Conclusions

The transformation rules presented in Chapter 3 provide a foundation for a Scheme program called VPS which translates VLISP PreScheme into Pure PreScheme. VPS was used to compile the VLISP Virtual Machine (VVM) [16], which is a byte-code interpreter. The VVM source is about 2200 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 VVM.

The original source for the VVM specified an iterative process because of a limitation of the Pure PreScheme compiler. 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 revised VPS which translates VLISP PreScheme into PreScheme Assembly Language or, for debugging purposes, C. While the C code is unverified as is its compiler, practice has shown that it provides a useful gauge of run-time performance. Measurements showed the recursive version of the VVM was slightly faster than the iterative one.

John Ramsdell and Vipin Swarup spent a considerable amount of time studying the VVM and its translation into Pure PreScheme. Our subjective analysis concluded the optimizations performed were comparable to the ones

performed by other optimizing transformational compilers. There were several times in which we thought the generated PreScheme was in error, only to realize later that VPS had simply performed more optimizations than we expected which obscured the correctness of the translation!

While VPS has been used to compile only one substantial program, it should perform well on most other VLISP PreScheme programs because its transformations are similar to those used by other successful compilers. There may be a few missing rules, such as some rules about arithmetic comparisons, but these rules can easily be added when identified. One note of caution: VPS is not of production quality. It has yet to be subjected to any code review by peers or independent referees.

Future Work

There are three tasks which would improve our verified compiler VPS. First, a verified assembler for PreScheme Assembly Language must be written. As mentioned before, its construction should be straightforward due to the work of Mitchell Wand and Dino Oliva [17, 11]. Second, confidence in the compiler could be increased by formally specifying the primitive operators and verifying their translation. Finally, the VLISP PreScheme language would be more useful as a systems programming language if facilities were provided to allow separate compilation. The first task is by far the most important one, and we plan to produce a verified assembler in FY94.

Conclusions

This paper defines the VLISP PreScheme language, a Scheme dialect useful for systems programming. The definition includes a formal denotational semantics.

The language can be compiled by transforming the program into a syntactically restricted subset of VLISP PreScheme, which then can be compiled using a syntax-directed compiler. This combination seems to produce reasonably good assembly code.

This paper gives a proof of the correctness of a set of transformation rules which perform the first translation and of a syntax-directed compiler

which performs the second translation. The result is a verified compiler that translates VLISP PreScheme into PreScheme Assembly Language. One can build transformational compilers which use rules whose justification is based firmly in the formal semantics of the programming language being compiled along with syntax-direct compilers justified relative to those same formal semantics.

Appendix A

Operators in C

This is vps.h version 5.5 of 93/09/28. It is the header file which is included into C code generated from Vlisp PreScheme by the Vlisp PreScheme Front End.

```
#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 */
{
  Int n2 = n1 >= 0 ? mag(n0) << n1 : mag(n0) >> -n1;
  return n0 >= 0 ? n2 : -n2;
}
```

```
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;
}

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. */
}

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)             /* changes store */
{
  if (feof(p)) return EOF;
  else return getc(p);
}

static Chr
  peek_char(Port p)             /* changes store */
{
  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)     /* changes store */
{
  return fputc(c, p) == EOF ? -1 : 0;
}
```

```
static inline Int
  write_int(Int n, Port p)      /* changes store */
{
  return fprintf(p, "%ld",  n) == EOF ? -1 : 0;
}

static inline Int
  write(String s, Port p)       /* changes store */
{
  return fputs(s, p) == EOF ? -1 : 0;
}

static inline Int
  newline(Port p)               /* changes store */
{
  return write_char('\n', p);
}

static inline Int
  force_output(Port p)          /* changes store */
{
  return fflush(p);
}

static inline Bool
  is_null_port(Port p)          /* invariable */
{
  return p == NULL;
}

static inline Port
  open_input_file(String s)     /* changes store */
{
  return fopen(s, "r");
}
```

```
static inline Int
  close_input_port(Port p)      /* changes store */
{
  return fclose(p);
}

static inline Port
  open_output_file(String s)    /* changes store */
{
  return fopen(s, "w");
}

static inline Int
  close_output_port(Port p)     /* changes store */
{
  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) /* changes store */
{
  n = fread(v, sizeof(Int), n, p);
  if (ferror(p)) {
    fprintf(stderr, "Error in read_image.\n");
    abort(1);
  }
  return n;
}
```

```
static Int
  write_image(Int *v, Int n, Port p) /* changes store */
{
  n = fwrite(v, sizeof(Int), n, p);
  if (ferror(p)) {
    fprintf(stderr, "Error in write_image.\n");
    abort(1);
  }
  return n;
}

#define bytes_per_word (sizeof(Int))

#define useful_bits_per_word (CHAR_BIT * sizeof(Int))

static inline Int               /* The type really is */
  quit(Int n)                   /* ∀α, Int → α. */
{                               /* changes store */
  exit(n);
}

#define err(n, s) \
  report_err((n), (s), __FILE__, __LINE__))

static Int
  report_err(Int n, String s, char *f, int l) /* changes store */
{
  (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] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[2] J. D. Guttman, L. G. Monk, W. M. Farmer, J. D. Ramsdell, and V. Swarup. The VLISP byte-code compiler. Technical Report M92B092, The MITRE Corporation, 1992.

[3] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to VLISP, a verified programming language implementation. Technical Report M92B091, The MITRE Corporation, 1992.

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

[5] Richard Kelsey and Jonathan Rees. Scheme48 progress report. Manuscript in preparation, 1992.

[6] Richard A. Kelsey. Realistic compilation by program transformation. In *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages*. ACM, 1989.

[7] Richard A. Kelsey. PreScheme: a scheme dialect for systems programming. Submitted for publication, 1992.

[8] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for scheme. *SIGPLAN Notices*, 21:219–233, 1986. Proceedings of the '86 Symposium on Compiler Construction.

[9] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[10] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[11] Dino P. Oliva. Advice on structuring compiler back ends and proving them correct. Technical Report NU-CCS-93, Northeastern University College of Computer Science, June 1993.

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

[13] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, Great Britain, 1991.

[14] J. D. Ramsdell, W. M. Farmer, J. D. Guttman, L. G. Monk, and V. Swarup. The VLISP PreScheme front end. Technical Report M92B098, The MITRE Corporation, 1992.

[15] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, MIT AI Laboratory, 1978.

[16] V. Swarup, W. M. Farmer, J. D. Guttman, L. G. Monk, and J. D. Ramsdell. The VLISP byte-code interpreter. Technical Report M92B097, The MITRE Corporation, 1992.

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