The VLISP PreScheme Front End

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

The MITRE Corporation[1]

M92B098
September 1992

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

Abstract

The Verified Programming Language Implementation project has developed a formally verified implementation of the Scheme programming language. This report documents the VLISP PreScheme language, used to program the VLISP Virtual Machine (interpreter). It contains detailed proofs that a set of transformations preserve the meanings of VLISP PreScheme programs.

Contents

List of Figures

1 Introduction

It might seem desirable to implement systems programming tasks in Scheme. Scheme's expressivity, simplicity, and regularity draw programmers. Scheme implementations usually provide a highly interactive programming environment with dynamic linking and powerful debugging tools.

There is a problem with this use of Scheme. Scheme implementations provide an extensive run-time system which includes automatic storage reclamation, first class procedures, and many other advanced features. Most systems programming tasks cannot afford the overhead of this run-time system.

PreScheme is a restricted dialect of Scheme intended for systems programming. It was invented by Jonathan Rees and Richard Kelsey [6]. 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 need 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.

The VLISP project has constructed a verified design of a Scheme system, and then implemented that design [2]. The design is based on Scheme48 [4], whose byte-code interpreter is written in PreScheme.

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 or macros, or compiler directives. The only other difference is 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 produces a new program by expanding most of VLISP PreScheme's derived syntax. The second stage translates the macro-free program into an equivalent program by using meaning preserving 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 Mitch Wand. They are creating a verified compiler for Pure PreScheme [10]. A straightforward translation is all that is required to convert a Simple PreScheme program into a Pure PreScheme program. Note that the current version of the Pure PreScheme compiler allows only tail-recursive calls, that is, all calls are reguarded as goto's which pass parameters and which never return.

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.

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 [6]. Furthermore, this compiler is essentially a transformational compiler [5].

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.

The contribution reported within 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.

2 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.

2.1 Syntax

The syntax of the VLISP PreScheme language is identical to the syntax of the language defined in the Scheme standard [3, 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 `lambda` expression, variables can also be defined using the syntax

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

  When ⟨variable⟩ is 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.

2.2 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 $\star\,\mathrm{Int}$; the other notation

```
(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 1: Pre-defined Procedures

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 1.

`<` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$	2 or more
`<=` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$	2 or more
`=` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$	2 or more
`>=` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$	2 or more
`>` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$	2 or more
`abs` : $\mathrm{Int} \to \mathrm{Int}$	1
`+` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	any
`-` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	1 or 2
`*` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	any
`quotient` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	2
`remainder` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	2
`ashl` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	2
`low-bits` : $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	2
`integer->char` : $\mathrm{Int} \to \mathrm{Chr}$	1
`char->integer` : $\mathrm{Chr} \to \mathrm{Int}$	1

`char=?` : $\mathrm{Chr} \times \mathrm{Chr} \to \mathrm{Bool}$	2
`char<?` : $\mathrm{Chr} \times \mathrm{Chr} \to \mathrm{Bool}$	2
`make-vector` : $\mathrm{Int} \to \star\mathrm{Int}$	1
`vector-ref` : $\star\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	2
`vector-set!` : $\star\mathrm{Int} \times \mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	3
`vector-byte-ref` : $\star\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	2
`vector-byte-set!` : $\star\mathrm{Int} \times \mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$	3
`addr<` : $\star\mathrm{Int} \times \star\mathrm{Int} \to \mathrm{Bool}$	2
`addr=` : $\star\mathrm{Int} \times \star\mathrm{Int} \to \mathrm{Bool}$	2
`addr+` : $\star\mathrm{Int} \times \mathrm{Int} \to \star\mathrm{Int}$	2
`addr-` : $\star\mathrm{Int} \times \star\mathrm{Int} \to \mathrm{Int}$	2
`addr->integer` : $\star\mathrm{Int} \to \mathrm{Int}$	1
`integer->addr` : $\mathrm{Int} \to \star\mathrm{Int}$	1
`addr->string` : $\star\mathrm{Int} \to \mathrm{String}$	1
`port->integer` : $\mathrm{Port} \to \mathrm{Int}$	1
`integer->port` : $\mathrm{Int} \to \mathrm{Port}$	1
`read-char` : $\mathrm{Port} \to \mathrm{Chr}$	0 or 1
`peek-char` : $\mathrm{Port} \to \mathrm{Chr}$	0 or 1
`eof-object?` : $\mathrm{Chr} \to \mathrm{Bool}$	1
`write-char` : $\mathrm{Chr} \times \mathrm{Port} \to \mathrm{Int}$	1 or 2
`write-int` : $\mathrm{Int} \times \mathrm{Port} \to \mathrm{Int}$	1 or 2
`write` : $\mathrm{String} \times \mathrm{Port} \to \mathrm{Int}$	1 or 2
`newline` : $\mathrm{Port} \to \mathrm{Int}$	0 or 1
`force-output` : $\mathrm{Port} \to \mathrm{Int}$	0 or 1
`null-port?` : $\mathrm{Port} \to \mathrm{Bool}$	1
`open-input-file` : $\mathrm{String} \to \mathrm{Port}$	1
`close-input-port` : $\mathrm{Port} \to \mathrm{Int}$	1
`open-output-file` : $\mathrm{String} \to \mathrm{Port}$	1
`close-output-port` : $\mathrm{Port} \to \mathrm{Int}$	1
`current-input-port` : $\to \mathrm{Port}$	0
`current-output-port` : $\to \mathrm{Port}$	0
`read-image` : $\star\mathrm{Int} \times \mathrm{Int} \times \mathrm{Port} \to \mathrm{Int}$	3
`write-image` : $\star\mathrm{Int} \times \mathrm{Int} \times \mathrm{Port} \to \mathrm{Int}$	3
`bytes-per-word` : Int	–
`useful-bits-per-word` : Int	–
`exit` : $\forall\alpha, \mathrm{Int} \to \alpha$	1
`err` : $\forall\alpha, \mathrm{Int} \times \mathrm{String} \to \alpha$	2

2.3 Formal Semantics

The formal semantics of VLISP PreScheme is presented in a form that very closely resembles Scheme's semantics [3, Appendix A]. It uses the same mathematical conventions, and many of the standard's definitions without repeating them here. I strongly suggested you have a copy of the standard in hand while 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. 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.

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

2.3.1 Abstract Syntax

$$
\begin{array}{rl}
\mathrm{K} \in \mathrm{Con} & \text{constants}\\
\mathrm{I} \in \mathrm{Ide} & \text{variables}\\
\mathrm{E} \in \mathrm{Exp} & \text{expressions}\\
\mathrm{B} \in \mathrm{Bnd} & \text{bindings}\\
\mathrm{P} \in \mathrm{Pgm} & \text{programs}
\end{array}
$$

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

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

2.3.2 Domain Equations

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

2.3.3 Semantic Functions

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

Definition of $\mathcal{K}$ deliberately omitted.

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

$\mathcal{E}[\![\text{K}]\!] =$ See [3, Appendix A]

$\mathcal{E}[\![\text{I}]\!] =$ See [3, Appendix A]

$\mathcal{E}[\![(\text{E E}^*)]\!] =$ See [3, Appendix A]

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

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

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

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

$\mathcal{E}[\![(\texttt{if}\ \text{E E E})]\!] =$ See [3, Appendix A]

$\mathcal{E}[\![(\texttt{if}\ \text{E E})]\!] =$ See [3, Appendix A]

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

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

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 all other identifiers to *undefined*. It maps no identifier to a location.

2.3.4 Auxiliary Functions

$$\begin{array}{l}
\mathit{lookup} : U \to \mathrm{Ide} \to D \\
\mathit{lookup} = \lambda\rho\mathrm{I}.\, \rho\mathrm{I}
\end{array}$$

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

$$\begin{array}{l}
\mathit{send} : E \to K \to C \\
\mathit{send} = \lambda\epsilon\kappa.\, \kappa\epsilon
\end{array}$$

$$\begin{array}{l}
\mathit{single} : K \to K \\
\mathit{single} = \lambda\kappa.\, \kappa
\end{array}$$

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

$$\begin{array}{l}
\mathit{hold} : D \to K \to C \\
\mathit{hold} = \lambda\delta\kappa\sigma.\, \delta \in E \to \mathit{send}(\delta \mid E)\kappa\sigma, \mathit{send}(\sigma((\delta \mid L) \downarrow 1))\kappa\sigma
\end{array}$$

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

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

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

2.4 Compiler Restrictions

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

$$
\begin{array}{l}
(\texttt{case}\ \langle\text{key}\rangle \\
\quad ((\texttt{0})\ \langle\text{sequence}_1\rangle) \\
\qquad \vdots \\
\quad ((\langle n-1\rangle)\ \langle\text{sequence}_n\rangle))
\end{array}
$$

Figure 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. The effect of providing a key which is not one of the selection criteria is undefined. This allows the omission of run-time range checks.

3 Macro-free PreScheme

Macro-free PreScheme (MFPS) programs result from VLISP PreScheme programs by expanding all derived syntax except the `case` expression, identifying which variables refer to standard procedures, 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 N-ary standard procedures must be used at one fixed arity.

3.1 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

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

3.2 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.

3.3 Static Semantics

Macro-free PreScheme programs may be strongly typed. As in Standard ML [9], 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 3. When a type is unconstrained by the rules, the expression is assigned the integer type.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Typing Rules

4 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

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

Simple PreScheme's semantics are inherited from Macro-free PreScheme's semantics.

Appendix B contains a compiler for a subset of Simple PreScheme. All Simple PreScheme programs are easily transformed into this subset.

5 The Design

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

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

Inline standard procedures: Each reference to a VLISP PreScheme standard procedure is replaced by its code.

Apply transformation rules: Translates a Macro-free PreScheme program into a Simple PreScheme one using meaning preserving 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 [8]. The unifier is based on a published program by Laurence Paulson [11, p. 381].

Print: Translates the internal representation of a strongly typed Simple PreScheme program into Pure PreScheme syntax.

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 preserving rules. The rules are meaning preserving in a sense to be made precise in Section 5.3.

The selection and application of rules is performed by a complex 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 only been introduced by bad rules. Therefore, the verification effort focused solely on the transformation rules.

5.1 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 $\mathrm{E}_0 \Longrightarrow \mathrm{E}_1$, and its predicate is given in the text. The predicate for rules with no restrictions given in the text is always satisfied.

In many systems using rewrite rules, the replacing expression is derived from the replacement by instantiating its pattern variables using the assignment 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*

  (`letrec` ((I$_1$ (`lambda` (I*) $C_1[\ ]$)) ...) $C_0[\ ]$).
- *If* $C_0[\ ]$*,* $C_1[\ ]$*, and* $C_2[\ ]$ *are contexts, then so is* (`if` $C_0[\ ]$ $C_1[\ ]$ $C_2[\ ]$).
- *If* $C_0[\ ]$ *and* $C_1[\ ]$ *are contexts, then so is* (`if` $C_0[\ ]$ $C_1[\ ]$).
- *If* $C[\ ]$ *is a context, then so is* (`set!` I $C[\ ]$).

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.

[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.

5.1.1 Syntactic Predicates

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

Definition 4 *An expression is* side effect free *if it returns a value without modifying the store.*

- 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) ...).

A side effect free expression can be eliminated when its value is ignored.

Definition 5 *An expression is* invariable *if it is side effect free and its value does not depend on modifiable variables.*

- 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* $E_0, \ldots, E_n$ *are invariable, then so is* (`case` E_0 ((K) E_1) ...).

When an invariable expression is evaluated later during the execution of a program, its value remains the same.

Definition 6 *An expression is* almost side effect free *if it does not modify the store until its last action.*

- K *is almost side effect free.*
- I *is almost side effect free.*
- *If* $E_1 \ldots E_n$ *are side effect free, then* (O $E_1 \ldots E_n$) *is almost side effect free.*
- *If* E_0 *is almost side effect free, and* $E_1, \ldots, E_n$ *are side effect free, then* ((`lambda` ($I_1 \ldots I_n$) E_0) $E_1 \ldots E_n$) *is almost side effect free.*
- (`lambda` (I^*) E) *is almost side effect free.*
- *If* $E_0, \ldots, E_{n-1}$ *are side effect free, and* E_n *is almost side effect free, then* (`begin` $E_0 \ldots E_n$) *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* $E_1, \ldots, E_2$ *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.*

5.1.2 The List of Rules

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

1. Simplification of some primitive expressions.

   - 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.

     $(\mathrm{O\ E\ K}) \Longrightarrow (\mathrm{O\ K\ E})$

   - Associative operators are moved 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.

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

   - 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. Simplification of conditional expressions (`if` and `case`).

   - Evaluate when 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 combinations in tests.

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

- Raise `begin`'s in tests.

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

- Simplify `if` in an `if`'s test.

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

  Used when both E_3 and E_4 are constants or variables.

- Simplify `if` in an `if`'s consequence.

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

  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 and so 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 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. Rotate combinations.

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

when E_0 is invariable.

```
(define I₁) ... (define Iᵢ) ... (define Iₙ)
(letrec ((Iₙ₊₁ Cₙ₊₁[Iᵢ]) ...)
  (begin
    (set! I₁ C₁[Iᵢ])
      ⋮
    (set! Iᵢ Eᵢ)
      ⋮
    (set! Iₙ Cₙ[Iᵢ])
    C₀[Iᵢ]))
⟹
(define I₁) ... (define Iᵢ) ... (define Iₙ)
(letrec ((Iₙ₊₁ Cₙ₊₁[Eᵢ]) ...)
  (begin
    (set! I₁ C₁[Eᵢ])
      ⋮
    (set! Iᵢ Eᵢ)
      ⋮
    (set! Iₙ Cₙ[Eᵢ])
    C₀[Eᵢ]))
```

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

Figure 4: Defined Constant Substitution

15. Rotate `begin`'s.

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

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 4.

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.

5.2 Usage of the Rules

The rules result in program transformations similar to those produced by other compilers [7, 5]. The rules for conditionals are the same, and the rules for β-reduction 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 [12]. As a result, the rotate combinations rule was added. Here is a common example of its use.

$$\begin{array}{l}(\texttt{let}\ ((\mathrm{I}_0\ (\texttt{let}\ ((\mathrm{I}_1\ \mathrm{E}_1)\ldots)\ \mathrm{E}^*\ \mathrm{I}_1)))\ \mathrm{E}_0)\\ \quad\Longrightarrow (\texttt{let}\ ((\mathrm{I}_1\ \mathrm{E}_1)\ldots)\ (\texttt{let}\ ((\mathrm{I}_0\ \mathrm{I}_1))\ \mathrm{E}^*\ \mathrm{E}_0))\end{array}$$

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 (compute-log-bytes-per-word x a)
  (if (<= x 1)
      a
      (compute-log-bytes-per-word (ashr x 1) (+ 1 a))))

(define log-bytes-per-word
  (compute-log-bytes-per-word bytes-per-word 0))

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

5.3 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 3. 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
```

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* $\chi\sigma = \perp_A$. *A transformation rule is* meaning preserving *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 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 preserve 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 an `if`'s consequence 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.

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

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

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

Notice ϵ is independent of κ so

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

5.3.1 `if` in an `if`'s Consequence

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

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

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 $\epsilon_0 \neq \mathit{undefined}$.

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

5.3.2 `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

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

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.

5.3.3 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,*

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

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

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

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,*

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

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,*

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

The proof is identical to that of Lemma 4, except 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,*

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

Proof sketch. The proof is very similar to the proof of Lemma 5, except that the store remains the same. Consider the case of $C[\,]$ being (`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

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

5.3.4 `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.

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

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

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

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

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

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

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

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

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

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

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

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

Let $g = \lambda\epsilon^*.\,f_0(\mathit{takefirst}\,\epsilon^*\#\mathrm{B}_0)\ \S\ \mathit{dropfirst}\,\epsilon^*\#\mathrm{B}_0$. 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

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

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

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

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

5.3.6 `letrec` Simplification

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

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

Proof. By Theorem 7,

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

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

because I is not free in E_0.

5.3.7 `letrec` Binding Merging

Theorem 10

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

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.

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

In words, it adds `letrec`'s 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,

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

Let

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

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

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

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

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

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

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

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

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

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

which is proved by induction on n.

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

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

Induction on m completes the proof because

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

5.3.8 Rotate Combinations

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 first 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?

5.3.9 Defined Constant Substitution

Theorem 12 *The rule shown in Figure 4 is meaning preserving.*

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,

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

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 $\mathcal{E}[\![\mathrm{I}_i]\!]\rho\kappa\sigma \sqsubseteq \mathcal{E}[\![\mathrm{E}_i]\!]\rho\kappa\sigma$.

6 Results

The transformation rules provide a foundation for a Scheme program called VPS which translates VLISP PreScheme into Pure PreScheme.[2] VPS has been used to translate the VLISP Virtual Machine (VVM) [13], 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.

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

[2] For debugging purposes, VPS can also translate VLISP PreScheme into C.

is not of production quality. It has yet to be subjected to any code review by peers or independent referees.

7 Conclusion

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 that perform the first translation, and Dino Oliva and Mitchell Wand [10] have produced a verified compiler which performs the second translation. The result is a verified compiler for VLISP PreScheme. One can build transformational compilers which use rules whose justification is based firmly in the formal semantics of the programming language being compiled.

Acknowledgements

John Ramsdell wrote this paper. 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. Vipin Swarup proofread the final version.

A The C Header File vps.h

This is vps.h version 4.1 of 92/19/13. 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
  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);
}
```

B The Facile PreScheme Compiler

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

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

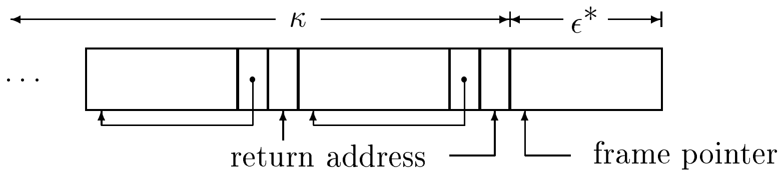

Figure 5: Stack Layout

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

There are further syntactic restrictions. Notice that only variables may be the elements of combinations and the arguments of primitive invocations. Each of these variables must be bound by a `lambda` expression. Furthermore, 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. See Figure 2 on page 11.

Facile PreScheme's semantics are inherited from Macro-free PreScheme's semantics.

B.1 Compilation

Facile PreScheme was designed to execute on an abstract machine which has only one stack for temporary storage. Both the local environment, temporary values, and the continuation are placed on the same stack. Facile PreScheme's syntactic restrictions on combinations and primitive invocations make this possible.

Figure 5 shows a potential layout for the stack. The stack grows rightward in the figure. The compiler translates a reference to a `lambda` bound variable into an offset used to reference the variable's value on the stack relative to the frame pointer.

The compiler is presented as an alternate semantics for Facile PreScheme.

B.1.1 Additional Domain Equations

$$\begin{array}{rcll} & D_c & = N + D & \text{compiler denoted values} \\ \gamma \in & U_c & = \text{Ide} \to D_c & \text{compiler environments} \\ \pi \in & Q & = E \to F & \text{expression instructions} \end{array}$$

B.1.2 Compiler Semantic Functions

$$\begin{array}{rl} \mathcal{CL} & : \text{Exp} \to U_c \to N \to Q \to F \\ \mathcal{CB} & : \text{Bnd} \to \text{Ide}^* \to U \to E^* \to E^* \\ \mathcal{CS} & : \text{Smpl} \to U_c \to N \to Q \to F \\ \mathcal{CC} & : \text{Cls} \to U_c \to N \to Q \to F^* \\ \mathcal{CS}^* & : \text{Smpl}^* \to U_c \to N \to F \to F \\ \mathcal{CE} & : \text{Exp} \to U \to F \\ \mathcal{CD} & : \text{Pgm} \to U \to K \to C \\ \mathcal{CP} & : \text{Pgm} \to A \end{array}$$

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

$$\mathcal{CS}[\![\mathrm{K}]\!] = \lambda\gamma\nu\pi.\, \mathit{literal}(\mathcal{K}[\![\mathrm{K}]\!])\pi$$

$$\mathcal{CS}[\![\mathrm{I}]\!] = \lambda\gamma\nu\pi.\, \gamma\mathrm{I} \in N \to \mathit{local}(\gamma\mathrm{I} \mid N)\pi, \mathit{global}(\gamma\mathrm{I} \mid D)\pi$$

$$\begin{aligned} &\mathcal{CS}[\![(\mathrm{I}\ \mathrm{I}^*)]\!] = \\ &\quad \lambda\gamma\nu\pi.\, \pi = \mathit{return} \to \mathit{call}(\gamma\mathrm{I} \mid N)(\mathit{map}(\lambda\mathrm{I}.\, \gamma\mathrm{I} \mid N)\mathrm{I}^*), \\ &\qquad\qquad \mathit{makecont}\, \nu\pi(\mathcal{CS}[\![(\mathrm{I}\ \mathrm{I}^*)]\!]\gamma\nu\, \mathit{return}) \end{aligned}$$

$$\begin{aligned} &\mathcal{CS}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})\ \mathrm{S}^*)]\!] = \\ &\quad \lambda\gamma\nu\pi.\, \mathit{reserve}\, \#\mathrm{I}^*(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma(\nu + \#\mathrm{I}^*)(\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\gamma\nu\pi)) \end{aligned}$$

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

$$\begin{aligned} &\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S}\ \mathrm{S}^*\ \mathrm{S}_0)]\!] = \\ &\quad \lambda\gamma\nu\pi.\, \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{ignore}(\mathcal{CS}[\![(\texttt{begin}\ \mathrm{S}^*\ \mathrm{S}_0)]\!]\gamma\nu\pi)) \end{aligned}$$

$$\begin{aligned} &\mathcal{CS}[\![(\texttt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!] = \\ &\quad \lambda\gamma\nu\pi.\, \mathcal{CS}[\![\mathrm{S}_0]\!]\gamma\nu(\mathit{jumpfalse}(\mathcal{CS}[\![\mathrm{S}_1]\!]\gamma\nu\pi)(\mathcal{CS}[\![\mathrm{S}_2]\!]\gamma\nu\pi)) \end{aligned}$$

$$\mathcal{CS}[\![(\texttt{if \#f \#f})]\!] = \lambda\gamma\nu\pi.\, \mathit{literal}\ \mathit{unspecified}\ \pi$$

$$\mathcal{CS}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{S})]\!] = \lambda\gamma\nu\pi.\, \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{setglobal}(\gamma\mathrm{I} \mid D)\pi)$$

$$\mathcal{CS}[\![(+\ \mathrm{I}_0\ \mathrm{I}_1)]\!] = \lambda\gamma\nu\pi.\, \mathit{add}(\gamma\mathrm{I}_0 \mid N)(\gamma\mathrm{I}_1 \mid N)\pi$$

$$\mathcal{CS}[\![(\texttt{case}\ \mathrm{S}\ \mathrm{C})]\!] = \lambda\gamma\nu\pi.\, \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{dispatch}(\mathcal{CC}[\![\mathrm{C}]\!]\gamma\nu\pi))$$

$$\mathcal{CC}[\![\ ]\!] = \lambda\gamma\nu\pi.\, \langle\rangle$$

$$\mathcal{CC}[\![((\mathrm{K})\ \mathrm{S})\ \mathrm{C}]\!] = \lambda\gamma\nu\pi.\, \langle\mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu\pi\rangle \ \S\ \mathcal{CC}[\![\mathrm{C}]\!]\gamma\nu\pi$$

$$\mathcal{CS}^*[\![\ ]\!] = \lambda\gamma\nu\phi.\, \phi$$

$$\begin{aligned}&\mathcal{CS}^*[\![\mathrm{S}\ \mathrm{S}^*]\!] = \\ &\quad \lambda\gamma\nu\phi.\, \mathcal{CS}[\![\mathrm{S}]\!]\gamma\nu(\mathit{setlocal}(\nu - \#\mathrm{S}^* - 1)(\mathcal{CS}^*[\![\mathrm{S}^*]\!]\gamma\nu\phi))\end{aligned}$$

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

$$\begin{aligned}&\mathcal{CB}[\![(\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S}))\ \mathrm{B}]\!] = \\ &\quad \lambda\mathrm{I}_0^*\rho\epsilon^*. \\ &\qquad \langle\mathcal{CL}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](\lambda\delta.\, \delta \text{ in } D_c \circ \mathit{extends}\ \rho\mathrm{I}_0^*\epsilon^*)0\ \mathit{return}\ \mathrm{in}E\rangle \\ &\qquad\quad \S\ \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^*\end{aligned}$$

$$\begin{aligned}&\mathcal{CE}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{S})]\!] = \\ &\quad \lambda\rho.\, \mathcal{CS}[\![\mathrm{S}]\!](\lambda\delta.\, \delta \text{ in } D_c \circ \mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{CB}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho))) \\ &\qquad\qquad 0 \\ &\qquad\qquad \mathit{return}\end{aligned}$$

$$\mathcal{CD}[\![\mathrm{E}]\!] = \lambda\rho\kappa\sigma.\, \mathcal{CE}[\![\mathrm{E}]\!]\rho\langle\rangle\kappa\sigma$$

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

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

B.1.3 Machine Instruction Auxiliary Functions

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

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

$$\begin{aligned}
&\mathit{setlocal} : N \to F \to Q\\
&\mathit{setlocal} = \lambda\nu\phi.\, \lambda\epsilon\epsilon^*\kappa.\, \phi(\mathit{takefirst}\ \epsilon^*\nu \ \S\ \langle\epsilon\rangle \ \S\ \mathit{dropfirst}\ \epsilon^*(\nu+1))\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathit{global} : D \to Q \to F\\
&\mathit{global} = \lambda\delta\pi.\, \lambda\epsilon^*\kappa.\, \mathit{hold}\ \delta\lambda\epsilon.\, \pi\epsilon\epsilon^*\kappa
\end{aligned}$$

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

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

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

$$\begin{aligned}
&\mathit{dispatch} : F^* \to Q\\
&\mathit{dispatch} = \lambda\phi^*.\, \lambda\epsilon\epsilon^*\kappa.\, (\phi^* \downarrow (1 + \epsilon \mid \mathrm{R}))\epsilon^*\kappa
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{return} : Q\\
&\mathit{return} = \lambda\epsilon\epsilon^*\kappa.\, \kappa\epsilon
\end{aligned}$$

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

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

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

$$\begin{aligned}
&\mathit{add} : N \to N \to Q \to F\\
&\mathit{add} = \lambda\nu_0\nu_1\pi.\, \lambda\epsilon^*\kappa.\, \pi(\mathit{stackref}\ \epsilon^*\nu_0 \mid \mathrm{R} + \mathit{stackref}\ \epsilon^*\nu_1 \mid \mathrm{R})\epsilon^*\kappa
\end{aligned}$$

B.1.4 Additional Auxiliary Functions

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

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

$$\mathit{map} = \lambda\psi\nu^*.\, \#\nu^* = 0 \to \langle\rangle, \langle\psi(\nu^* \downarrow 1)\rangle \ \S\ \mathit{map}\,\psi(\nu^* \dagger 1)$$

$$\begin{aligned}&\mathit{extends}_c : D_c \to \mathrm{Ide}^* \to N \to D_c\\ &\mathit{extends}_c =\\ &\quad \lambda\gamma \mathrm{I}^*\nu.\, \#\mathrm{I}^* = 0 \to \gamma,\\ &\qquad\qquad \mathit{extends}_c(\lambda \mathrm{I}.\, \mathrm{I} = \mathrm{I}^* \downarrow 1 \to \nu \text{ in } D_c, \gamma\mathrm{I})(\mathrm{I}^* \dagger 1)(\nu + 1)\end{aligned}$$

B.2 Correctness

The correctness of the Facile PreScheme compiler has not yet been demonstrated. It would involve proving $\mathcal{CP}[\![\mathrm{P}]\!] = \mathcal{P}[\![\mathrm{P}]\!]$ by appealing to the following conjectures. The correctness of the conjectures depends on the fact that Facile PreScheme programs are strongly typed.

Definition 11 $\mathit{compose} = \lambda\gamma\epsilon^*\mathrm{I}.\, \gamma\mathrm{I} \in N \to \mathit{stackref}\ \epsilon^*(\gamma\mathrm{I} \mid N) \text{ in } D, \gamma\mathrm{I} \mid D$

Conjecture 1 *Assume* $\mathit{compose}\,\gamma\epsilon^* = \mathit{compose}\,\gamma(\epsilon^* \ \S\ \epsilon_0^*)$.

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

Conjecture 2 *Assume* $\mathit{compose}\,\gamma\epsilon^* = \mathit{compose}\,\gamma(\epsilon^* \ \S\ \epsilon_0^*)$.

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

Conjecture 3 *Assume* $\#\epsilon^* \geq \#\mathrm{I}^*$ *and let* $\nu = \#\epsilon^* - \#\mathrm{I}^*$.

$$\mathit{compose}\,\gamma(\mathit{takefirst}\ \epsilon^*\nu) = \mathit{compose}\,\gamma((\mathit{takefirst}\ \epsilon^*\nu) \ \S\ \epsilon_0^*)$$

implies

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

The previous three conjectures could be proved using simultaneous structural induction on simple expressions.

Conjecture 4

$$\begin{aligned}&\mathcal{CL}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!](\lambda\delta.\,\delta \text{ in } D_c \circ \rho)0\ \mathit{return} \,\mathrm{in} E\\ &\quad = \mathcal{L}[\![\texttt{(lambda}\ (\mathrm{I}^*)\ \mathrm{S})]\!]\rho\end{aligned}$$

Conjecture 5 $\mathcal{BP}[\![\mathrm{B}]\!] = \mathcal{B}[\![\mathrm{B}]\!]$

Conjecture 6 $\mathcal{CE}[\![\mathrm{E}]\!]\rho\langle\rangle\kappa\sigma = \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma$

Conjecture 7 $\mathcal{CD}[\![\mathrm{P}]\!] = \mathcal{D}[\![\mathrm{P}]\!]$

Conjecture 8 $\mathcal{CP}[\![\mathrm{P}]\!] = \mathcal{P}[\![\mathrm{P}]\!]$

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, revised edition, 1984.

[2] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to VLISP, a verified programming language implementation. M 92B091, The MITRE Corporation, 1992.

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

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

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

[6] Richard A. Kelsey. PreScheme: A Scheme dialect for systems programming. Submitted for publication, 1992.

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

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

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

[10] 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.

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

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

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