A Verified Compiler for Pure PreScheme: Final Report for Contract Number F19628-89-C-001

Dino P. Oliva
oliva@corwin.ccs.northeastern.edu
Mitchell Wand
wand@flora.ccs.northeastern.edu
College Of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA

September 23, 1991

Contents

Abstract

This document gives a summary of activities under MITRE Contract Number F19628-89-C-001. It gives a detailed denotational specification of the language Pure PreScheme. A bytecode compiler, derived from the semantics, is presented, followed by proofs of correctness of the compiler with

respect to the semantics. Finally, an assembler from the bytecode to an actual machine architecture is shown.

1 Introduction

The goal of this project was to develop a verified compiler for the PreScheme programming language. PreScheme is a restricted dialect of Scheme. It is simpler than Scheme in the data types it supports, in its treatment of procedure objects, and because it makes very limited use of the run-time stack.

This report is divided into two parts. Sections 1 through 6 give a chronologically-organized summary of the workplan and the activities performed under this contract. Sections 7 through 11 then give the technical details and the proof of correctness of the Pure PreScheme compiler.

The workplan was divided into three parts:

1. Analysis of the syntax and semantics of PreScheme. Our activities here are discussed in Sections 2–3. The detailed syntax and semantics are presented in Sections 7–8.
2. Development and verification of a translator from Pure PreScheme into an abstract assembly language. This activity is discussed in Section 4. The compiler is presented in Section 9, and the proofs are presented in Section 10.
3. Development of a translator from the abstract assembly language to the final target language. This activity is discussed in Section 4, and the details of the assembler are presented in Section 11.

The executable versions of the compiler and assembler, coded in Scheme, have been electronically delivered and are now being tested by MITRE, Inc.

2 Obtaining an executable Scheme semantics

Our first undertaking was the translation of the denotational semantics of Scheme into SPS, a type-checked variant of Scheme [11]. The purpose of this exercise was to obtain a version of the Scheme semantics which could be checked and manipulated by machine.

To do this, we obtained from Jonathan Rees a copy of the executable version of the denotational semantics of Scheme given in Section 7.2 of [7]. We then modified it to fit in the semantic framework given by SPS. The resulting interpreter was run on a number of examples.

This exercise confirmed that in general, the Section 7.2 semantics was consistent (i.e., it contained no type errors). However it did reveal a few shortcomings of the Section 7.2 semantics and the Clinger-Rees executable semantics (CRES):

- concrete syntax: this is specified in the Report, but it is not included in the CRES. SPS requires that concrete syntax be fully specified, and we have done this.
- non-compositionality: the treatment of *permute* in the Report is somewhat non-compositional, as it works on lists of expressions, rather than lists of denotations. This is fixed in our version.
- storage allocation for constants: the report is silent on whether storage for constants is allocated at read time or at evaluation time. We have chosen the latter.

A more detailed discussion of our findings can be found in [6]. We also developed some tools to translate a grammar in SPS format to Latex and to preview the grammar in a window.

3 Refining the PreScheme semantics

Originally, PreScheme was defined as the set of programs which a particular program (the PreScheme compiler) could translate into C. This definition was necessarily imprecise. Therefore, our first major task was to produce a more precise syntax and semantics for PreScheme. We undertook this task by interviewing the participants (mostly Richard Kelsey) and by examining the corpus of PreScheme code in the Virtual Machine definition.

As a result of these discussions, we have refined PreScheme into three levels:

1. **Full PreScheme.** This includes the code in the VM, essentially unchanged, including macro definitions. This dialect is unmanageable because the language of macro definitions is full Scheme.

2. **Macro-Free PreScheme (MFPS).** This is obtained from Full PreScheme by expanding all macros.

3. **Pure PreScheme (PPS).** This is a tail-recursive, closure-free dialect of PreScheme, obtained from MFPS by hoisting lambda-expressions and beta-expansion. This dialect is semantically tractable and was the focus of our compiler development efforts.

We continued our discussions with Richard Kelsey, John Ramsdell, and Joshua Guttman to agree on the primitives and syntax of Pure PreScheme. Here we list the primary conclusions of those discussions, along with some of the implications of those decisions:

1. Pure PreScheme programs are tail-recursive. The original conception of PreScheme allowed bounded non-tail recursion; however, we agreed that the restriction to pure tail recursion was acceptable. This requires some minor recoding of the VM, since there are some calls to the garbage collector which are not tail recursive. However, since there are no *nested* calls, this recoding should be easy.

2. Pure PreScheme programs are closure-free: that is, closures are never created at run-time. All lambda-expressions can, at least in principle, be hoisted to a top-level scope. John Ramsdell has written a translator from MFPS to Pure PreScheme that performs this hoisting.

   The global structure of a PPS program consists of some global variables (including mutable globals, lexically distinguishable by names of the form `*`*name*`*`), followed by a large `letrec` containing all the mutually recursive procedures.

   In particular, this means that the primitive `goto` has been dropped. Both global procedures and those which are given by a local recursion will be called as ordinary procedures. This distinction was an artifact of the C implementation of PreScheme.

3. Local variables of procedures are not mutable.

4. Procedures will no longer be storable. The semantic overhead of this facility was considerable. It is used in only one place in the VM, to build a dispatch table for the opcode interpreter. This dispatch table, an array of procedures, was built once at VM initialization time, and never modified. This will be replaced by a computed-goto expression called `choose`. This also requires some recoding in the VM.

   Eliminating procedures as storables considerably simplifies the semantics of the heap, which will be approximately that of C. Heap primitives perform something like `malloc` to create vectors, and vector indices are pointers.

5. The following are the primitives of PreScheme:

```
not
unassigned
error

%+ %- %* %< %<= %= %>= %> %quotient %remainder %abs
%adjoin-bits %low-bits %high-bits
%bitwise-not %bitwise-and %bitwise-ior %bitwise-xor
%ashl %ashr
%ascii->char %char->ascii
%char=? %char<?
%useful-bits-per-word

%make-vector %vector-ref %vector-set! %vector-length
%vector-posq %vector-fill!
%make-string %{\bf string-set}!   ;for extract-string
%string-length %string-ref

%make-byte-vector
%byte-vector-fill!
%byte-vector-ref      %byte-vector-set!
%byte-vector-word-ref %byte-vector-word-set!

%make-byte-vector-pointer
%byte-vector-pointer-ref
%byte-vector-pointer-set!
%byte-vector-pointer-word-ref
```

```
%byte-vector-pointer-word-set!

;; I/O
%read-char %write-char %write-string
%newline
%eof-object?
%open-input-file    %open-output-file
%close-input-port   %close-output-port
%current-input-port %current-output-port
%force-output

;; Used only by READ-IMAGE and WRITE-IMAGE
%write-number %read-number
%write-page   %read-page
%write-byte   %read-byte    %bits-per-io-byte
%call-with-output-file %call-with-input-file ;ditto
```

These will be treated as keywords, so they will not be shadowable by lambda-bindings. The Pure PreScheme compiler will not necessarily deal with all these primitives, but will implement a subset large enough to handle the VM as recoded by the MITRE group.

The following were treated as PreScheme primitives in the original version of the VM, but were removed as primitives in light of our discussions:

```
goto computed-goto label run-machine halt-machine
make-dispatch-table define-dispatch! dispatch
```

The grammar and semantics for PPS programs is shown in Section 8. The semantics was developed by retrofitting the SPS Scheme semantics to the restrictions listed above.

4 Development of the PreScheme Byte-Code Compiler

The next stage was the development of a translator from Pure PreScheme to a combinator-based (byte-code) abstract assembly language. Byte code appeared to be a suitable target language for this compiler because the quantities manipulated by the byte code abstract machine were sufficiently close to ordinary machine quantities that the translation from byte code to machine code would be straightforward. This is in contrast with the situation for full Scheme, in which the representation of these quantities was one of the major difficulties.

The compiler was developed using the Wand-Clinger compiler development methodology [2, 8, 9, 10]. In this methodology, the denotational semantics of the source language is taken as a *concrete* semantics: a translation into some lambda-calculus. This translation is then modified to produce terms in a combinatory calculus. The operational semantics of the lambda-calculus, given by reduction, then yields an operational semantics for the target language.

This approach is made feasible by the observation that for suitable choices of combinators, the reduction sequences mimic precisely the behavior of ordinary machines. In the original papers [8, 9, 10], the correctness of the compiler was just equality between the combinator semantics and the original semantics. Clinger [2] showed how to produce a specification relating the combinator semantics and the original semantics in which the combinator terms clearly resembled ordinary byte codes; this avoided the linearization or distribution step in the original version.

The development of the Pure PreScheme Byte-Code Compiler had three main steps:

1. Development of the induction hypotheses. An induction hypothesis was developed for each non-terminal in the Pure PreScheme grammar. These were obtained by retrofitting the hypotheses in [2] to the restrictions of the Pure PreScheme architecture. In particular, we needed to eliminate the run-time recursion stack and to distinguish non-terminals that are executed for effect only. In our implementation, a small stack is still used for handling local variables, but this could be replaced by a register file.

2. Derivation of the byte codes. In this development method, the induction hypotheses serve as the specification for the compiler. One can then substitute the semantic equations for each production and essentially solve the equations to derive the behavior of the needed opcodes. This notion of derivation is not shown in this report, but the results are presented in the opcode definitions of Section 9.4.

3. Implementation of the translator proper. This was a straightforward programming exercise in Scheme.

We also implemented an interpreter for the byte-code language, using the definitions of the byte codes as Scheme definitions. Thus the interpreter came essentially for free from the theory.

5 Development of the Assembler

In order to make this compiler a usable tool, two more items were needed:

1. A hoister-expander to translate from MFPS to PPS. This task was accomplished by John Ramsdell.
2. A translator from byte code to real assembly language.

We implemented the byte code machine by designing a fairly straightforward representation scheme for the quantities in the byte code machine, and writing a translator that emitted MC68020 code to simulate its action. Because the quantities in Pure PreScheme, and therefore in the byte codes, were relatively simple, this was easy.

We used a tagged architecture to aid with debugging and because the PreScheme semantics might demand it. [The semantics does not completely specify the behavior of a program in the presence of a run-time type error. In most cases, run-time type errors invoke the continuation *wrong*, which is not specified in the semantics. If *wrong* is interpreted to return a don't-care value, then any behavior is acceptable, and a tagged architecture is unnecessary. If the value of *wrong* is deemed significant, then the implementation must test for run-time type errors. We chose to be conservative and take the latter interpretation.]

Another part of the translator was the implementation of the primitives. We chose to implement the following primitives, which were representative of the entire list:

```
%abort %zero? %= %> %< %<= %+ %- %*
%make-vector %vector-ref %vector-set!
```

For each primitive, it was necessary to emit assembly code that performed the operation.

The major complication was representing global procedures, which are the only closures in the system. This is discussed in Section 11.

6 Conclusions and Further Work

We have specified the semantics of Pure PreScheme and proved the correctness of a compiler from Pure PreScheme to a byte code. The correctness proof consists of a straightforward structural induction whose individual steps are accomplished in the lambda-calculus. The complete proof is given in this report. The byte-code to assembly-language translation is viewed as straightforward and self-evidently correct, though such a judgement is of course subject to debate.

As of this writing, the compiler is being tested using on a small virtual machine definition for the pure lambda-calculus. Once that is completed, work can progress on using it to compile the Scheme48 VM.

We currently envision extensions of this work in at least four directions, each of which will serve the overall goal of increasing the reliability and trustworthiness of compilers:

1. Machine support for checking or generating proofs of the form displayed here. We are working on this at the present time [1]

2. Formalizing the semantics of the target machine and the correctness of the representation strategies used to implement the byte-code machine. This was a major problem for the VM correctness proof. Some work is underway on this elsewhere [3], but we believe that our methods may lead to significant advances on this problem.

3. Extending the Wand-Clinger methodology to compilers that perform significant optimizations. Some first steps toward this are reported in [5] and [12].

4. Compiling Scheme by translating Scheme programs to Pure PreScheme programs.

7 Language Overview

This section begins the technical portion of this document. It begins with an overview of the Pure PreScheme language. Section 8 gives the syntax and denotational semantics of the language. Section 9 presents the compiler, and Section 10 presents the correctness proof. Last, Section 11 describes the assembler from byte codes to assembly language.

Pure PreScheme is a descendent of the language PreScheme, invented by Jonathan Rees and Richard Kelsey for the Scheme48 compiler. A Pure PreScheme program consists of an initial declaration of global variables, top level procedure declarations, and a body. All global variables are enclosed within asterisks (e.g. `*global*`) and are the only variables which can be affected imperatively. The syntax of PurePreScheme has been designed so that all procedures and the main body are tail-recursive. Nested procedure calls can occur in 'simple' expressions where it can be assured that no recursion will occur, that is, wherever it would be safe to expand the procedure call in-line. Procedures can be passed upwards and downwards but cannot be cannot be created dynamically. Other expressed values available in PurePreScheme are integers, characters, booleans, and byte pointers.

8 The Semantics

The specification of PurePreScheme is a straightforward exercise in denotational semantics.

The only unusual aspect has to do with the environments. Specifically, they can contain either pointers into the store for globals or expressible values for locals; this must be taken into account. We use the notation $\rho[-/-]$ for the simultaneous extension of an environment by a *list* of bindings; the usual one-point environment update is therefore written as $\rho[\langle\delta\rangle/\langle\mathtt{I}\rangle]$. The behavior of extension when the list of identifiers contains duplicates is unspecified. We also need a notion of sequential environment extension. This notion and notation are defined by the following axioms:

$$\rho[\langle\,\rangle/\langle\,\rangle]^* = \rho$$

$$\rho[(\delta :: \delta^*)/(\mathtt{I} :: \mathtt{I}^*)]^* = (\rho[\langle\delta\rangle/\langle\mathtt{I}\rangle])[\delta^*/\mathtt{I}^*]^*$$

Here $\langle\,\rangle$ denotes an empty sequence and $(\alpha :: \alpha^*)$ denotes adding α to a sequence of $\alpha's$, α^*. Similarly, $(\alpha_0^* : \alpha_1^*)$ denotes the concatenation of two sequences of $\alpha's$.

8.1 Syntax

The input to the language is assumed to be a list structure; therefore the lexical structure of the language is unspecified, except that global identifiers are Scheme symbols distinguished by being surrounded by asterisks (e.g. `*global-variable*`). The nonterminals of the language are as follows:

`L`	$\in$ `LocIde`	local identifiers
`G`	$\in$ `GloIde`	global identifiers
`I`	$\in$ `Ide`	local & global identifiers
`K`	$\in$ `Con`	constants
`P`	$\in$ `Prim`	primitives
`LSD`	$\in$ `LocSimDec`	local simple declarations
`GSD`	$\in$ `GloSimDec`	global simple declarations
`LPD`	$\in$ `LocProcDec`	local procedure declarations
`S`	$\in$ `Simple`	simple expressions & commands
`T`	$\in$ `Tre`	tail-recursive expressions
`E`	$\in$ `Pgm`	programs

The starting symbol is `Pgm`.

The productions are:

```
LSD ::=  (L S)*

GSD ::=  (G S)*

LPD ::=  (L (lambda (L*) T))*

S   ::=  K
     |   L
     |   G
     |   (if S0 S1 S2)
     |   (choose S (S*))
     |   (set! G S)
     |    (P S*)

T   ::=   S
     |    (if S T0 T1)
     |    (begin S* T)
     |    (let (LSD) T)
     |    (let* (LSD) T)
     |    (S S*)

E   ::=   (let* (GSD) (letrec (LPD) T))
```

8.2 Value Domains

$\alpha \in L$	locations/denoted values
$\nu \in N$	small integers
$T = \{\text{false, true}\}$	truth values
H	characters
$\phi \in F = E^* \to C$	procedures
$\beta \in B$	byte pointers
$\upsilon \in P = E^* \to K \to C$	primitives
$\omega \in O = U \to F$	openers
$\chi \in X$	error messages
$\epsilon \in E = N + T + H + F + B$	expressed/stored values
$\sigma \in S = L \to E$	stores
$\delta \in D = L + E$	denoted values
$\rho \in U = \texttt{Ide} \to D$	environments
$\theta \in C = S \to A$	command continuations
$\kappa \in K = E \to C$	expression continuations
A	answers

These domains are all standard, except for the domain of openers; these are procedures (closures) that are waiting to receive the environment in which they will be closed.

A distinguishing characteristic of these domains is that all the expressed values can naturally fit in one machine word, except for procedures, which must be represented by pointers. Note also that procedures do not take continuations, since they are tail-recursive.

8.3 Semantic Functions

The semantic functions include three auxiliary functions for analyzing declarations, and one main valuation for each syntactic category. Note that $\mathcal{T}$ does not take a continuation argument, since these are tail-recursive expressions, which do not use a run-time stack.

$\mathcal{GL}$:	$\texttt{LocSimDec} \to \texttt{LocIde}^*$
$\mathcal{GP}$:	$\texttt{LocProcDec} \to \texttt{LocIde}^*$
$\mathcal{GG}$:	$\texttt{GloSimDec} \to \texttt{GloIde}^*$
$\mathcal{K}$:	$\texttt{Con} \to E$
$\mathcal{P}$:	$\texttt{Prim} \to P$
$\mathcal{DL}$:	$\texttt{LocSimDec} \to U \to (E^* \to C) \to C$
$\mathcal{SDL}$:	$\texttt{LocSimDec} \to U \to (E^* \to C) \to C$
$\mathcal{SDG}$:	$\texttt{GloSimDec} \to U \to (L^* \to C) \to C$
$\mathcal{RP}$:	$\texttt{LocProcDec} \to \texttt{LocIde}^* \to U \to E^* \to E^*$
$\mathcal{RDP}$:	$\texttt{LocProcDec} \to U \to (E^* \to C) \to C$
$\mathcal{SE}$:	$\texttt{Simple} \to U \to K \to C$
$\mathcal{SE}^*$:	$\texttt{Simple}^* \to U \to (E^* \to C) \to C$
$\mathcal{SEC}^*$:	$\texttt{Simple}^* \to N \to U \to K \to C$
$\mathcal{SC}$:	$\texttt{Simple} \to U \to C \to C$
$\mathcal{SC}^*$:	$\texttt{Simple}^* \to U \to C \to C$
$\mathcal{SCC}^*$:	$\texttt{Simple}^* \to N \to U \to C \to C$
$\mathcal{T}$:	$\texttt{Tre} \to U \to C$
$\mathcal{E}$:	$\texttt{Pgm} \to U \to C$

8.3.1 Auxiliary Valuations

These valuations extract sequences of identifiers from various declarations.

$\mathcal{GL}$: $\texttt{LocSimDec} \to \texttt{LocIde}^*$

Get Locals. Extracts local identifiers from local simple declarations.

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

$\mathcal{GL}[\![(\texttt{L S})\ \texttt{LSD}]\!] = (\texttt{L} :: \mathcal{GL}[\![\texttt{LSD}]\!])$

$\mathcal{GP}$: $\texttt{LocProcDec} \to \texttt{LocIde}^*$

Get Procedures. Extracts procedure identifiers from local procedure declarations.

$\mathcal{GP}[\![\ ]\!] = \langle\,\rangle$

$\mathcal{GP}[\![\texttt{(L (lambda (L}^*\texttt{) T)) LPD}]\!] = (\texttt{L} :: \mathcal{GP}[\![\texttt{LPD}]\!])$

$\mathcal{GG}$: $\texttt{GloSimDec} \to \texttt{GloIde}^*$

Get Globals. Extracts global identifiers from global simple declarations.

$\mathcal{GG}[\![\ ]\!] = \langle\,\rangle$

$\mathcal{GG}[\![\texttt{(G S) GSD}]\!] = (\texttt{G} :: \mathcal{GG}[\![\texttt{GSD}]\!])$

8.3.2 Main Valuations

$\mathcal{K}$: $\texttt{Con} \to E$

Constants.

[definition deliberately omitted]

$\mathcal{P}$: $\texttt{Prim} \to P$

Primitives.

[definition deliberately omitted]

$\mathcal{DL}$: $\texttt{LocSimDec} \to U \to (E^* \to C) \to C$

Declare Locals. Takes a sequence of local simple declarations and returns a corresponding sequence of values of the simple expressions.

$\mathcal{DL}[\![\;]\!] = \lambda\rho\psi\,.\,\psi\,\langle\,\rangle$

$\mathcal{DL}[\![\texttt{(L S) LSD}]\!] = \lambda\rho\psi\,.\,\mathcal{DL}[\![\texttt{LSD}]\!]\rho(\lambda\epsilon^*\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,\psi\,(\epsilon :: \epsilon^*)))$

$\mathcal{SDL} : \texttt{LocSimDec} \rightarrow U \rightarrow (E^* \rightarrow C) \rightarrow C$

Sequentially Declare Locals. Takes a sequence of local simple declarations and returns a corresponding sequence of values of the simple expressions. The environment is updated as each simple expression is evaluated.

$\mathcal{SDL}[\![\;]\!] = \lambda\rho\psi\,.\,\psi\,\langle\,\rangle$

$$\begin{aligned}\mathcal{SDL}[\![\texttt{(L S) LSD}]\!] = \lambda\rho\psi\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,&\mathcal{SDL}[\![\texttt{LSD}]\!]\rho[\langle\epsilon\,\mathrm{in}\,D\rangle/\langle\texttt{L}\rangle]\\ &(\lambda\epsilon^*\,.\,\psi\,(\epsilon :: \epsilon^*)))\end{aligned}$$

$\mathcal{SDG} : \texttt{GloSimDec} \rightarrow U \rightarrow (L^* \rightarrow C) \rightarrow C$

Sequentially Declare Globals. Takes a sequence of global simple declarations and returns a corresponding sequence of locations in the store where values of the simple expressions are located.

$\mathcal{SDG}[\![\;]\!] = \lambda\rho\xi\,.\,\xi\,\langle\,\rangle$

$$\begin{aligned}\mathcal{SDG}[\![\texttt{(G S) GSD}]\!] = \lambda\rho\xi\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,(\mathit{tieval}\,(\lambda\alpha\,.\,&\mathcal{SDG}[\![\texttt{GSD}]\!]\rho[\langle\alpha\,\mathrm{in}\,D\rangle/\langle\texttt{G}\rangle]\\ &(\lambda\alpha^*\,.\,\xi\,(\alpha :: \alpha^*)))\\ \epsilon))\end{aligned}$$

$\mathcal{RP} : \texttt{LocProcDec} \rightarrow \texttt{LocIde}^* \rightarrow U \rightarrow E^* \rightarrow E^*$

Recursive Procedures. Takes a sequence of procedure declarations and returns a corresponding sequence of recursive procedures (when used in conjunction with $\mathcal{RDP}$). More precisely, given a sequence of procedure declarations, a list of local identifiers $\mathtt{L}_0^*$ (the names of the procedures in the recursive declaration), an environment ρ (the environment on entrance to the declaration), and a list of expressed values ϵ_0^* (the values of those procedures) , it returns a list of expressed values. Each expressed value is a procedure, obtained by closing the corresponding lambda expression in the environment obtained by extending ρ by binding the identifiers in $\mathtt{L}_0^*$ to the corresponding values in ϵ_0^*.

This is used in conjunction with $\mathcal{RDP}$, which uses a fixpoint operator to set up the appropriate fixpoint equation.

As noted above, the behavior of extension when the list of identifiers contains duplicates is unspecified. Therefore the behavior of $\mathcal{RP}$ is unspecified when either `L` or $\mathtt{L}_0^*$ contains duplicates.

$$\mathcal{RP}[\![\ ]\!] = \lambda \mathtt{L}^* \rho \epsilon^* \,.\, \langle\ \rangle$$

$$\begin{aligned}
&\mathcal{RP}[\![\mathtt{(L\ (lambda\ (L^*)\ T))\ LPD}]\!] = \\
&\qquad \lambda \mathtt{L}_0^* \rho \epsilon_0^* \,.\, (((\lambda \epsilon^* \,.\, \mathcal{T}[\![\mathtt{T}]\!](\rho[(map\ (\text{in}\ D)\ \epsilon_0^*)/\mathtt{L}_0^*]) \\
&\qquad\qquad\qquad [(map\ (\text{in}\ D)\ \epsilon^*)\ \text{in}(E)/(reverse\ \mathtt{L}^*)] \\
&\qquad\qquad\qquad :: \mathcal{RP}[\![\mathtt{LPD}]\!]\mathtt{L}_0^* \rho \epsilon_0^*)
\end{aligned}$$

$$\mathcal{RDP} : \mathtt{LocProcDec} \to U \to (E^* \to C) \to C$$

Recursively Declare Procedures. Takes a sequence of procedure declarations and returns a corresponding sequence recursive procedures. ($\mathcal{GP}[\![\mathtt{LPD}]\!]$ calculates the list of procedure names declared in `LPD`, corresponding to the argument $\mathtt{L}_0^*$ in the definition of $\mathcal{RP}$. ρ is the environment enclosing the declaration, as in $\mathcal{RP}$. Thus the argument to the fixpoint operator corresponds to $\lambda\epsilon^* \,.\, \ldots$; the value of the fixpoint is a sequence of expressed values, which are passed to the continuation.

$$\mathcal{RDP}[\![\mathtt{LPD}]\!] = \lambda \rho \psi \,.\, \psi(\mathtt{fix}\ (\mathcal{RP}[\![\mathtt{LPD}]\!](\mathcal{GP}[\![\mathtt{LPD}]\!])\rho))$$

$$\mathcal{SE} : \mathtt{Simple} \to U \to K \to C$$

Simple Expressions. Evaluates the expression and passes the value to the expression continuation.

$$\mathcal{SE}[\![\texttt{K}]\!] = \lambda\rho\kappa\,.\,\kappa(\mathcal{K}[\![\texttt{K}]\!])$$

$$\mathcal{SE}[\![\texttt{L}]\!] = \lambda\rho\kappa\,.\,(\rho\texttt{L})\ \text{E}\ E \rightarrow \kappa((\rho\texttt{L}) \mid E), \\ (\textit{wrong}\ \text{“Local variable given storage.”})$$

$$\mathcal{SE}[\![\texttt{G}]\!] = \lambda\rho\kappa\,.\,(\rho\texttt{G})\ \text{E}\ L \rightarrow (\textit{hold}\ ((\rho\texttt{G}) \mid L)\ \kappa), \\ (\textit{wrong}\ \text{“Global variable not given storage.”})$$

$$\mathcal{SE}[\![\texttt{(if S}_0\texttt{ S}_1\texttt{ S}_2\texttt{)}]\!] = \lambda\rho\kappa\,.\,\mathcal{SE}[\![\texttt{S}_0]\!]\rho(\lambda\epsilon\,.\,(\epsilon \mid T) = \texttt{false} \rightarrow \mathcal{SE}[\![\texttt{S}_2]\!]\rho\kappa, \\ \mathcal{SE}[\![\texttt{S}_1]\!]\rho\kappa)$$

$$\mathcal{SE}[\![\texttt{(choose S (S}^*\texttt{))}]\!] = \lambda\rho\kappa\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,\epsilon\ \text{E}\ N \rightarrow \mathcal{SEC}^*[\![\texttt{S}^*]\!](\epsilon \mid N)\rho\kappa \\ (\textit{wrong}\ \text{“Non-numeric argument.”}))$$

$$\mathcal{SE}[\![\texttt{(set! G S)}]\!] = \lambda\rho\kappa\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,(\rho\texttt{G})\ \text{E}\ L \rightarrow (\textit{assign}\ ((\rho\texttt{G}) \mid L)\ \epsilon\ (\kappa\ \epsilon)), \\ (\textit{wrong} \\ \text{“Can't assign to a local variable.”}))$$

$$\mathcal{SE}[\![\texttt{(P S}^*\texttt{)}]\!] = \lambda\rho\kappa\,.\,\mathcal{SE}^*[\![\texttt{S}^*]\!]\rho(\lambda\epsilon^*\,.\,\textit{apply-primitive}\ \mathcal{P}[\![\texttt{P}]\!]\ \epsilon^*\ \kappa)$$

$\mathcal{SE}^* : \texttt{Simple}^* \rightarrow U \rightarrow (E^* \rightarrow C) \rightarrow C$

Simple Expression sequences. Evaluates the sequence of simple expressions and passes the sequence of values to the continuation.

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

$$\mathcal{SE}^*[\![\texttt{S S}^*]\!] = \lambda\rho\psi\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,\mathcal{SE}^*[\![\texttt{S}^*]\!]\rho(\lambda\epsilon^*\,.\,\psi\ (\epsilon^* : \langle\epsilon\rangle)))$$

$\mathcal{SEC}^* : \texttt{Simple}^* \rightarrow N \rightarrow U \rightarrow K \rightarrow C$

Simple Expression Choose sequences. Returns the value of the simple expression in the sequence of simple expressions corresponding to the numeric choice, ν. If there is no corresponding choice an error is signaled.

$\mathcal{SEC}^*[\![\ ]\!] = \lambda\nu\rho\kappa\,.\,\mathit{wrong}$ "Choose: index out of bounds."

$\mathcal{SEC}^*[\![\texttt{S S}^*]\!] = \lambda\nu\rho\kappa\,.\,\nu = 0 \rightarrow \mathcal{SE}[\![\texttt{S}]\!]\rho\kappa,\ \mathcal{SEC}^*[\![\texttt{S}^*]\!](\nu - 1)\rho\kappa$

$\mathcal{SC} : \texttt{Simple} \rightarrow U \rightarrow C \rightarrow C$

Simple Commands. Evaluates the expression and but ignores the return value.

$\mathcal{SC}[\![\texttt{K}]\!] = \lambda\rho\theta\,.\,\theta$

$\mathcal{SC}[\![\texttt{L}]\!] = \lambda\rho\theta\,.\,\theta$

$\mathcal{SC}[\![\texttt{G}]\!] = \lambda\rho\theta\,.\,\theta$

$$\begin{aligned}\mathcal{SC}[\![\texttt{(if S}_0\texttt{ S}_1\texttt{ S}_2\texttt{)}]\!] = \lambda\rho\theta\,.\,\mathcal{SE}[\![\texttt{S}_0]\!]\rho(\lambda\epsilon\,.\,(\epsilon \mid T) = \mathit{false} \rightarrow {}& \mathcal{SC}[\![\texttt{S}_2]\!]\rho\theta,\\ & \mathcal{SC}[\![\texttt{S}_1]\!]\rho\theta)\end{aligned}$$

$$\begin{aligned}\mathcal{SC}[\![\texttt{(choose S (S}^*\texttt{))}]\!] = \lambda\rho\theta\,.\,&\mathcal{SE}[\![\texttt{S}]\!]\rho\\ &(\lambda\epsilon\,.\,\epsilon\ \mathrm{E}\ N \rightarrow \mathcal{SCC}^*[\![\texttt{S}^*]\!](\epsilon \mid N)\rho\theta,\\ &\qquad(\mathit{wrong}\ \text{"Non-numeric argument."}))\end{aligned}$$

$$\begin{aligned}\mathcal{SC}[\![\texttt{(set! G S)}]\!] = \lambda\rho\theta\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,&(\rho\texttt{G})\ \mathrm{E}\ L \rightarrow (\mathit{assign}\ ((\rho\texttt{G}) \mid L)\ \epsilon\ \theta),\\ &(\mathit{wrong}\\ &\quad\text{"Can't assign to a local variable."}))\end{aligned}$$

$\mathcal{SC}[\![\texttt{(P S}^*\texttt{)}]\!] = \lambda\rho\theta\,.\,\mathcal{SE}^*[\![\texttt{S}^*]\!]\rho(\lambda\epsilon^*\,.\,\mathit{apply\text{-}primitive/ignore}\ \mathcal{P}[\![\texttt{P}]\!]\ \epsilon^*\ \theta)$

$\mathcal{SC}^* : \texttt{Simple}^* \rightarrow U \rightarrow C \rightarrow C$

Simple Command sequences. Evaluates the sequence of simple expressions ignoring the values.

$\mathcal{SC}^*[\![\ ]\!] = \lambda\rho\theta\,.\,\theta$

$\mathcal{SC}^*[\![\texttt{S S}^*]\!] = \lambda\rho\theta\,.\,\mathcal{SC}[\![\texttt{S}]\!]\rho(\mathcal{SC}^*[\![\texttt{S}^*]\!]\rho\theta)$

$\mathcal{SCC}^* :\ \texttt{Simple}^* \to N \to U \to C \to C$

Simple Command Choose sequences. Evaluates of the simple expression in the sequence of simple expressions corresponding to the numeric choice, ν, ignoring the value. If there is no corresponding choice an error is signaled.

$\mathcal{SCC}^*[\![\ ]\!] = \lambda\nu\rho\theta\,.\,\textit{wrong}$ "Choose: index out of bounds."

$\mathcal{SCC}^*[\![\texttt{S S}^*]\!] = \lambda\nu\rho\kappa\,.\,\nu = 0 \to \mathcal{SC}[\![\texttt{S}]\!]\rho\theta,\ \mathcal{SCC}^*[\![\texttt{S}^*]\!](\nu-1)\rho\theta$

Tail-recursive expressions. Evaluates the tail-recursive expression and returns the value. As the expressions are tail-recursive, no continuation is necessary. κ_0 is the initial continuation for the program. This is unspecified.

$\mathcal{T} :\ \texttt{Tre} \to U \to C$

$\mathcal{T}[\![\texttt{S}]\!] = \lambda\rho\,.\ \mathcal{SE}[\![\texttt{S}]\!]\rho\kappa_0$

$\mathcal{T}[\![\texttt{(if S T}_0\texttt{ T}_1\texttt{)}]\!] = \lambda\rho\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,(\epsilon \mid T) = \texttt{false} \to \mathcal{T}[\![\texttt{T}_1]\!]\rho,\ \mathcal{T}[\![\texttt{T}_0]\!]\rho)$

$\mathcal{T}[\![\texttt{(begin S}^*\texttt{ T)}]\!] = \lambda\rho\,.\,\mathcal{SC}[\![\texttt{S}^*]\!]\rho(\mathcal{T}[\![\texttt{T}]\!]\rho)$

$\mathcal{T}[\![\texttt{(let (LSD) T)}]\!] = \lambda\rho\,.\,\mathcal{DL}[\![\texttt{LSD}]\!]\rho(\lambda\epsilon^*\,.\,\mathcal{T}[\![\texttt{T}]\!]\rho[(\textit{map}\ (\text{in}\ D)\ \epsilon^*)/\mathcal{GL}[\![\texttt{LSD}]\!]])$

$$\begin{aligned}\mathcal{T}[\![\texttt{(let* (LSD) T)}]\!] = \lambda\rho\,.\,&\mathcal{SDL}[\![\texttt{LSD}]\!]\rho\\ &(\lambda\epsilon^*\,.\,\mathcal{T}[\![\texttt{T}]\!]\rho[(\textit{map}\ (\text{in}\ D)\ \epsilon^*)/\mathcal{GL}[\![\texttt{LSD}]\!]^*)\end{aligned}$$

$\mathcal{T}[\![\texttt{(S S}^*\texttt{)}]\!] = \lambda\rho\,.\,\mathcal{SE}^*[\![\texttt{S}^*]\!]\rho(\lambda\epsilon^*\,.\,\mathcal{SE}[\![\texttt{S}]\!]\rho(\lambda\epsilon\,.\,(\textit{tail-apply}\ \epsilon\ \epsilon^*)))$

$\mathcal{E} :\ \texttt{Pgm} \to U \to C$

Expressions. Sequentially declares any globals, then recursively defines any procedures in the updated environment, and finally evaluates the tail recursive body in the resulting environment.

$$\begin{aligned}&\mathcal{E}[\![\texttt{(let* (GSD) (letrec (LPD) T))}]\!] =\\ &\quad\lambda\rho\,.\,\mathcal{SDG}[\![\texttt{GSD}]\!]\rho\\ &\qquad(\lambda\alpha^*\,.\,\mathcal{RDP}[\![\texttt{LPD}]\!]\rho[(\textit{map}\ (\text{in}\ D)\ \alpha^*)/\mathcal{GG}[\![\texttt{GSD}]\!]]^*\\ &\qquad\quad(\lambda\epsilon^*\,.\,\mathcal{T}[\![\texttt{T}]\!](\rho[(\textit{map}\ (\text{in}\ D)\ \alpha^*)/\mathcal{GG}[\![\texttt{GSD}]\!]]^*)\\ &\qquad\qquad[(\textit{map}\ (\text{in}\ D)\ \epsilon^*)/\mathcal{GP}[\![\texttt{LPD}]\!]]))\end{aligned}$$

8.4 Auxiliary Functions

$\mathit{wrong} : \; X \to C$

[definition deliberately omitted]

$\mathit{new} : \; S \to L$

[definition deliberately omitted]

$\mathit{hold} : \; L \to K \to C$

$\mathit{hold} = \lambda\alpha\kappa \,.\, \lambda\sigma \,.\, \kappa(\sigma\;\alpha)\sigma$

$\mathit{update} : \; L \to E \to S \to S$

$\mathit{update} = \lambda\alpha\epsilon\sigma \,.\, \lambda\alpha_0 \,.\, \alpha_0 = \alpha \to \epsilon,\; \sigma\alpha_0$

$\mathit{assign} : \; L \to E \to C \to C$

$\mathit{assign} = \lambda\alpha\epsilon\theta \,.\, \lambda\sigma \,.\, \theta(\mathit{update}\;\alpha\;\epsilon\;\sigma)$

$\mathit{tieval} : \; (L \to C) \to E \to C$

$\mathit{tieval} = \lambda\xi\epsilon \,.\, \lambda\sigma \,.\, \xi(\mathit{new}\,\sigma)(\mathit{update}\;(\mathit{new}\,\sigma)\;\epsilon\;\sigma)$

$\mathit{apply\text{-}primitive} : \; P \to E^* \to K \to C$

$\mathit{apply\text{-}primitive} = \lambda\upsilon\epsilon^*\kappa \,.\, \upsilon\epsilon^*\kappa$

$\mathit{apply\text{-}primitive/ignore} : \; P \to E^* \to C \to C$

$\mathit{apply\text{-}primitive/ignore} = \lambda\upsilon\epsilon^*\theta \,.\, \upsilon\epsilon^*(\lambda\epsilon \,.\, \theta)$

$\mathit{tail\text{-}apply} : \; F \to E^* \to C$

$\mathit{tail\text{-}apply} = \lambda\epsilon\epsilon^* \,.\, \epsilon \;\mathrm{E}\; F \to (\epsilon \mid F)\epsilon^*,\; (\mathit{wrong}\;\text{“Non-function to apply”})$

9 The Compiler

The compiler for PurePreScheme produces code for an abstract machine consisting of a runtime environment, μ, and stack, ζ. The stack is used only for the storage of temporaries within an expression, and is not used for control purposes. A byte-code program is a term π of type $Q = U_r \to E^* \to C$. Hence a typical machine configuration is a term of the form $\pi\mu\zeta$.

Though the machine is defined denotationally, machine configurations are given an operational semantics by inheritance from the operational semantics (that is, the reduction behavior) of the lambda-calculus. The bytecode terms are carefully defined so that if π is a legal bytecode program, then $\pi\mu\zeta$ reduces to a term $\pi'\mu'\zeta'$ in some small number of reduction steps. Furthermore, since all these terms are in continuation-passing style, there is essentially only one such reduction that is possible. Therefore we can interpret this reduction sequence as a step in the operational semantics of the machine: $\pi\mu\zeta \Rightarrow \pi'\mu'\zeta'$. This notion of operational semantics is familiar to any Scheme programmer and in fact goes back at least 30 years to McCarthy, who first recognized the relation between tail-recursive program schemes and ordinary iteration [4].

For tail-recursive expressions, the compiler takes a symbol table γ and produces code. In the case of non tail-recursive constructs, the compiler takes not only γ but also a code continuation π, consisting of the code to follow the code generated.

We assume the existence of functions for extending run-time and compile-time environments. A runtime environment μ and a symbol table γ form a representation (distributed across different times!) of an environment $(\mu \circ \gamma)$. We need to ensure that the run-time and compile-time extension functions behave consistently. The behavior of these functions must satisfy the following for any μ and γ:

$$((\mu \circ \gamma))[(\mathit{map}\ (\mathrm{in}\ D)\ \epsilon^*)/\mathtt{L}^*] = ((\mathit{extends}_{r\ l}\ \mu\ \epsilon^*) \circ (\mathit{extends}_{c\ l}\ \gamma\ \mathtt{L}^*))$$

$$((\mu \circ \gamma))[(\mathit{map}\ (\mathrm{in}\ D)\ \epsilon^*)/\mathtt{L}^*]^* = ((\mathit{extends}^*_{r\ l}\ \mu\ \epsilon^*) \circ (\mathit{extends}^*_{c\ l}\ \gamma\ \mathtt{L}^*))$$

$$((\mu \circ \gamma))[(\mathit{map}\ (\mathrm{in}\ D)\ \alpha^*)/\mathtt{G}^*]^* = ((\mathit{extends}^*_{r\ g}\ \mu\ \alpha^*) \circ (\mathit{extends}^*_{c\ g}\ \gamma\ \mathtt{G}^*))$$

The standard implementation of lexical addresses satisfies these axioms.

In addition, the procedures *top*, *take-first*, and *pop-first* are used to manipulate the runtime stack. These are assumed to satisfy the following:

$$(\mathit{top}\ (\epsilon :: \zeta)) = \epsilon$$

$$(\textit{take-first}\, 0\, \zeta) = \langle\, \rangle$$
$$(\textit{take-first}\, (n+1+\ (\epsilon :: \zeta)) = (\epsilon :: (\textit{take-first}\, n\, \zeta))$$
$$(\textit{pop-first}\, 0\, \zeta) = \zeta$$
$$(\textit{pop-first}\, (n+1)\, (\epsilon :: \zeta)) = (\textit{pop-first}\, n\, \zeta)$$

The compiler itself is presented in terms of its semantics. Thus we write:

$\mathcal{CSDL}[\![\ ]\!] = \lambda\gamma\pi\,.\,\pi$

$$\begin{aligned}\mathcal{CSDL}[\![\texttt{(L S) LSD}]\!] = \lambda\gamma\pi\,.\,&\mathcal{CSE}[\![\texttt{S}]\!]\gamma\\ &(\textit{add-to-env*}\\ &\quad(\mathcal{CSDL}[\![\texttt{LSD}]\!](\textit{extends}^*_{cl}\,\gamma\,\langle\texttt{L}\rangle)\pi))\end{aligned}$$

However, as we did for the semantics, we envision this as a *concrete* semantics, yielding combinator terms of type Q rather than values in Q. For this translation, we assume that all arguments to compiler procedures (in this case, the syntactic argument, γ, and π) are compile-time values, and all results of compiler procedures (always of type Q) are run-time values, that is, at compile-time they appear as *terms* of type Q. Thus, using the back-quote convention of Lisp, we might translate the fragment above into code like the following:

```
(define csdl
  (lambda (locsimdec gamma pi)
    (record-case locsimdec
      (empty-locsimdec () pi)
      (composite-locsimdec (l s lsd)
(cse s gamma
     (add-to-env*
       (csdl lsd (extends*-cl gamma (list l)) pi)))))))
```

This code might be pure semantics, but we can specify a pass separation by revealing the definition of `add-to-env*` as:

```
(define add-to-env*
  (lambda (q)
    `(add-to-env* ,q)))
```

or equivalently

```
(define add-to-env*
  (lambda (q)
    (list 'add-to-env* q)))
```

This specifies that the output of this program is a term: a byte-code program. We obtain a compiler by assuming that each of the procedures listed as a machine instruction below is implemented by such a back-quote procedure. The semantics of the resulting byte-code program is given by replacing each byte-code symbol by its semantics.

9.1 Compiler Syntactic Domains

The syntactic domains for the compiler are the same as those for the semantics.

9.2 Compiler Value Domains

$\alpha \in L$	locations/denoted values
$\nu \in N$	small integers
$T = \{\text{false, true}\}$	truth values
H	characters
$\phi \in F = E^* \to C$	procedures
$\beta \in B$	byte pointers
$\upsilon \in P = E^* \to K \to C$	primitives
$\omega \in O = U \to F$	openers
$\chi \in X$	error messages
$\epsilon \in E = N + T + H + F + B$	expressed/stored values
$\sigma \in S = L \to E$	stores
$\delta \in D = L + E$	denoted values
$\iota \in D_c$	lexical values
$\gamma \in U_c = \texttt{Ide} \to D_c$	compiler environments
$\mu \in U_r = D_c \to D$	runtime environments
$\rho \in U = \texttt{Ide} \to D$	environments
$\theta \in C = S \to A$	command continuations
$\kappa \in K = E \to C$	expression continuations
$\pi \in Q = U_r \to E^* \to C$	byte code

A	answers

These are all the same as in the semantics, except for the new domains of compiler environments (symbol tables), runtime environments (displays), lexical values (or lexical addresses)

9.3 Compiler Semantic Functions

$\mathcal{GL}$:	$\texttt{LocSimDec} \to \texttt{LocIde}^*$
$\mathcal{GP}$:	$\texttt{LocProcDec} \to \texttt{LocIde}^*$
$\mathcal{GG}$:	$\texttt{GloSimDec} \to \texttt{GloIde}^*$
$\mathcal{K}$:	$\texttt{Con} \to E$
$\mathcal{P}$:	$\texttt{Prim} \to P$
$\mathcal{CDL}$:	$\texttt{LocSimDec} \to U_c \to Q \to Q$
$\mathcal{CSDL}$:	$\texttt{LocSimDec} \to U_c \to Q \to Q$
$\mathcal{CSDG}$:	$\texttt{GloSimDec} \to U_c \to Q \to Q$
$\mathcal{CRP}$	$\texttt{LocProcDec} \to \texttt{LocIde}^* \to O$
$\mathcal{CRDP}$:	$\texttt{LocProcDec} \to U_c \to Q \to Q$
$\mathcal{CSE}$:	$\texttt{Simple} \to U_c \to Q \to Q$
$\mathcal{CSE}^*$:	$\texttt{Simple}^* \to U_c \to Q \to Q$
$\mathcal{CSEC}^*$:	$\texttt{Simple}^* \to U_c \to Q \to Q$
$\mathcal{CSC}$:	$\texttt{Simple} \to U_c \to Q \to Q$
$\mathcal{CSC}^*$:	$\texttt{Simple}^* \to U_c \to Q \to Q$
$\mathcal{CSCC}^*$:	$\texttt{Simple}^* \to U_c \to Q \to Q$
$\mathcal{CT}$:	$\texttt{Tre} \to U_c \to Q$
$\mathcal{CE}$:	$\texttt{Pgm} \to U_c \to Q$

9.3.1 Auxiliary Valuations

$\mathcal{GL}$: $\texttt{LocSimDec} \to \texttt{LocIde}^*$

[definition same as for semantics]

$\mathcal{GP}$: $\texttt{LocProcDec} \to \texttt{LocIde}^*$

[definition same as for semantics]

$\mathcal{GG}$: `GloSimDec` $\rightarrow$ `GloIde`*

[definition same as for semantics]

9.3.2 Main Valuations

$\mathcal{K}$: `Con` $\rightarrow E$

[definition same as for semantics]

$\mathcal{P}$: `Prim` $\rightarrow P$

[definition same as for semantics]

$\mathcal{CDL}$: `LocSimDec` $\rightarrow U_c \rightarrow Q \rightarrow Q$

Compile Declare Locals. Takes a sequence of local simple declarations and generates code which pushes onto the runtime stack the corresponding sequence of values of the simple expressions.

$\mathcal{CDL}[\![\ ]\!] = \lambda\gamma\pi\ .\ \pi$

$\mathcal{CDL}[\![\texttt{(L S) LSD}]\!] = \lambda\gamma\pi\ .\ \mathcal{CDL}[\![\texttt{LSD}]\!]\gamma(\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)$

$\mathcal{CSDL}$: `LocSimDec` $\rightarrow U_c \rightarrow Q \rightarrow Q$

Compile Sequentially Declare Locals. Takes a sequence of local simple declarations and generates code which pushes onto the runtime stack the corresponding sequence of values of the simple expressions. The environment is updated as each simple expression is evaluated.

$\mathcal{CSDL}[\![\ ]\!] = \lambda\gamma\pi\,.\,\pi$

$$\begin{aligned}\mathcal{CSDL}[\![\texttt{(L S) LSD}]\!] = \lambda\gamma\pi\,.\,&\mathcal{CSE}[\![\texttt{S}]\!]\gamma\\ &(\textit{add-to-env*}\\ &\quad(\mathcal{CSDL}[\![\texttt{LSD}]\!](\textit{extends}^*_{c\,l}\ \gamma\ \langle\texttt{L}\rangle)\pi))\end{aligned}$$

$\mathcal{CSDG} :$ `GloSimDec` $\rightarrow U_c \rightarrow Q \rightarrow Q$

Compile Sequentially Declare Globals. Takes a sequence of global simple declarations and generates code which pushes onto the runtime stack a corresponding sequence of locations in the store where values of the simple expressions are located.

$\mathcal{CSDG}[\![\ ]\!] = \lambda\gamma\pi\,.\,\pi$

$$\begin{aligned}\mathcal{CSDG}[\![\texttt{(G S) GSD}]\!] = \lambda\gamma\pi\,.\,&\mathcal{CSE}[\![\texttt{S}]\!]\gamma\\ &(\textit{add-global-to-env*}\\ &\quad(\mathcal{CSDG}[\![\texttt{GSD}]\!](\textit{extends}^*_{c\,g}\ \gamma\ \langle\texttt{G}\rangle)\ \pi))\end{aligned}$$

$\mathcal{CRP} :$ `LocProcDec` $\rightarrow$ `LocIde`* $\rightarrow U_c \rightarrow O$

Compile Recursive Procedures. Takes a sequence of procedure declarations and generates code which pushes onto the runtime stack the corresponding sequence recursive procedures (when used in conjunction with $\mathcal{CRDP}$).

$\mathcal{CRP}[\![\ ]\!] = \lambda\texttt{L}^*\gamma\,.\,\textit{empty-openers}$

$$\begin{aligned}&\mathcal{CRP}[\![\texttt{(L (lambda (L}^*\texttt{) T)) LPD}]\!] =\\ &\quad\lambda\texttt{L}^*_0\gamma\,.\,\textit{openers}\ (\#\texttt{L}^*)\ (\mathcal{CT}[\![\texttt{T}]\!](\textit{extends}_{c\,l}\ (\textit{extends}_{c\,l}\ \gamma\ (\textit{reverse}\,\texttt{L}^*))\ \texttt{L}^*_0))\\ &\qquad\qquad\qquad\qquad(\mathcal{CRP}[\![\texttt{LPD}]\!]\texttt{L}^*_0\gamma)\end{aligned}$$

$\mathcal{CRDP} :$ `LocProcDec` $\rightarrow U_c \rightarrow Q \rightarrow Q$

Compile Recursively Declare Procedures. Takes a sequence of procedure declarations and generates code which pushes onto the runtime stack a corresponding sequence recursive procedures.

$\mathcal{CRDP}[\![\texttt{LPD}]\!] = \lambda\gamma\pi.\mathit{closerecs}\,(\mathcal{CRP}[\![\texttt{LPD}]\!](\mathcal{GP}[\![\texttt{LPD}]\!])\gamma)\;\pi$

$\mathcal{CSE} : \texttt{Simple} \to U_c \to Q \to Q$

Compile Simple Expressions. Generates code which evaluates the expression and pushes the value onto the runtime stack.

$\mathcal{CSE}[\![\texttt{K}]\!] = \lambda\gamma\pi\;.\;\mathit{constant}\,(\mathcal{K}[\![\texttt{K}]\!])\;\pi$

$\mathcal{CSE}[\![\texttt{L}]\!] = \lambda\gamma\pi\;.\;\mathit{fetch\text{-}local}\,(\gamma\;\texttt{L})\;\pi$

$\mathcal{CSE}[\![\texttt{G}]\!] = \lambda\gamma\pi\;.\;\mathit{fetch\text{-}global}\,(\gamma\;\texttt{G})\;\pi$

$\mathcal{CSE}[\![\texttt{(if S}_0\texttt{ S}_1\texttt{ S}_2\texttt{)}]\!] = \lambda\gamma\pi\;.\;\mathcal{CSE}[\![\texttt{S}_0]\!]\gamma(\mathit{brf}\,(\mathcal{CSE}[\![\texttt{S}_1]\!]\gamma\pi)\;(\mathcal{CSE}[\![\texttt{S}_2]\!]\gamma\pi))$

$\mathcal{CSE}[\![\texttt{(choose S (S}^*\texttt{))}]\!] = \lambda\gamma\pi\;.\;\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{numeric?}\,(\mathcal{CSCE}^*[\![\texttt{S}^*]\!]\gamma\pi))$

$\mathcal{CSE}[\![\texttt{(set! G S)}]\!] = \lambda\gamma\pi\;.\;\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{update\text{-}store}\,(\gamma\;\texttt{G})\;\pi)$

$\mathcal{CSE}[\![\texttt{(P S}^*\texttt{)}]\!] = \lambda\gamma\pi\;.\;\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma(\mathit{prim\text{-}apply}\;\#\texttt{S}^*\;\mathcal{P}[\![\texttt{P}]\!]\;\pi)$

$\mathcal{CSE}^* : \texttt{Simple}^* \to U_c \to Q \to Q$

Compile Simple Expression sequences. Evaluates the sequence of simple expressions and pushes the sequence of values onto the runtime stack.

$\mathcal{CSE}^*[\![\;]\!] = \lambda\gamma\pi\;.\;\pi$

$\mathcal{CSE}^*[\![\texttt{S S}^*]\!] = \lambda\gamma\pi\;.\;\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma\pi)$

$\mathcal{CSEC}^* : \texttt{Simple}^* \to U_c \to Q \to Q$

Compile Simple Expression Choose sequences. Pushes the value of the simple expression in the sequence of simple expressions corresponding to the numeric choice, ν, onto the runtime stack. If there is no corresponding choice an error is signaled.

$\mathcal{CSEC}^*[\![\ ]\!] = \lambda\gamma\pi\ .\ \textit{out-of-bounds}$

$\mathcal{CSEC}^*[\![\texttt{S S}^*]\!] = \lambda\gamma\pi\ .\ \textit{pick}\,(\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)\ (\mathcal{CSEC}^*[\![\texttt{S}^*]\!]\gamma\pi)$

$\mathcal{CSC} : \texttt{Simple} \to U_c \to Q \to Q$

Compile Simple Commands. Evaluates the expression and but does not push the value onto the runtime stack.

$\mathcal{CSC}[\![\texttt{K}]\!] = \lambda\gamma\pi\ .\ \pi$

$\mathcal{CSC}[\![\texttt{L}]\!] = \lambda\gamma\pi\ .\ \pi$

$\mathcal{CSC}[\![\texttt{G}]\!] = \lambda\gamma\pi\ .\ \pi$

$\mathcal{CSC}[\![\texttt{(if S}_0\texttt{ S}_1\texttt{ S}_2\texttt{)}]\!] = \lambda\gamma\pi\ .\ \mathcal{CSE}[\![\texttt{S}_0]\!]\gamma(\textit{brf}\,(\mathcal{CSC}[\![\texttt{S}_1]\!]\gamma\pi)\ (\mathcal{CSC}[\![\texttt{S}_2]\!]\gamma\pi))$

$\mathcal{CSC}[\![\texttt{(choose S (S}^*\texttt{))}]\!] = \lambda\gamma\pi\ .\ \mathcal{CSE}[\![\texttt{S}]\!]\gamma(\textit{numeric?}\,(\mathcal{CSCC}^*[\![\texttt{S}^*]\!]\gamma\pi))$

$\mathcal{CSC}[\![\texttt{(set! G S)}]\!] = \lambda\gamma\pi\ .\ \mathcal{CSE}[\![\texttt{S}]\!]\gamma(\textit{update-store/ignore}\ (\gamma\ \texttt{G})\ \pi)$

$\mathcal{CSC}[\![\texttt{(P S}^*\texttt{)}]\!] = \lambda\gamma\pi\ .\ \mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma(\textit{prim-apply/ignore}\ \#S^*\ \mathcal{P}[\![\texttt{P}]\!]\ \pi)$

$\mathcal{CSC}^* : \texttt{Simple}^* \to U_c \to Q \to Q$

Compile Simple Command sequences. Evaluates the sequence of simple expressions ignoring the values.

$\mathcal{CSC}^*[\![\ ]\!] = \lambda\gamma\pi\ .\ \pi$

$\mathcal{CSC}^*[\![\texttt{S S}^*]\!] = \lambda\gamma\pi\ .\ \mathcal{CSC}[\![\texttt{S}]\!]\gamma(\mathcal{CSC}^*[\![\texttt{S}^*]\!]\gamma\pi)$

$\mathcal{CSCC}^* :\ \texttt{Simple}^* \to U_c \to Q \to Q$

Compile Simple Command Choose sequences. Evaluates of the simple expression in the sequence of simple expressions corresponding to the numeric choice, ν, ignoring the value. If there is no corresponding choice an error is signaled.

$\mathcal{CSCC}^*[\![\ ]\!] = \lambda\gamma\pi\ .\ \mathit{out\text{-}of\text{-}bounds}$

$\mathcal{CSCC}^*[\![\texttt{S S}^*]\!] = \lambda\gamma\pi\ .\ \mathit{pick}\,(\mathcal{CSC}[\![\texttt{S}]\!]\gamma\pi)\ (\mathcal{CSCC}^*[\![\texttt{S}^*]\!]\gamma\pi)$

$\mathcal{CT} :\ \texttt{Tre} \to U_c \to Q$

Compile Tail-recursive expressions. Generates code which evaluates the tail-recursive expression and pushes the value onto the runtime stack. As the expressions are tail-recursive, no code continuation is necessary for compilation.

$\mathcal{CT}[\![\texttt{S}]\!] = \lambda\gamma\ .\ \mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{halt})$

$\mathcal{CT}[\![\texttt{(if S T}_0\ \texttt{T}_1\texttt{)}]\!] = \lambda\gamma\ .\ \mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{brf}\,(\mathcal{CT}[\![\texttt{T}_0]\!]\gamma)\ (\mathcal{CT}[\![\texttt{T}_1]\!]\gamma))$

$\mathcal{CT}[\![\texttt{(begin S}^*\ \texttt{T)}]\!] = \lambda\gamma\ .\ \mathcal{CSC}[\![\texttt{S}^*]\!]\gamma(\mathcal{CT}[\![\texttt{T}]\!]\gamma)$

$$\mathcal{CT}[\![\texttt{(let (LSD) T)}]\!] = \lambda\gamma\ .\ \mathcal{CDL}[\![\texttt{LSD}]\!]\gamma \\ (\mathit{add\text{-}to\text{-}env}\,(\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\ \gamma\ (\mathcal{GL}[\![\texttt{LSD}]\!]))))$$

$\mathcal{CT}[\![\texttt{(let* (LSD) T)}]\!] = \lambda\gamma\ .\ \mathcal{CSDL}[\![\texttt{LSD}]\!]\gamma(\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}^*_{c\,l}\ \gamma\ (\mathcal{GL}[\![\texttt{LSD}]\!])))$

$\mathcal{CT}[\![\texttt{(S S}^*\texttt{)}]\!] = \lambda\gamma\ .\ \mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma(\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{tail\text{-}call}))$

$\mathcal{CE} : \ \texttt{Pgm} \to U_c \to Q$

Compile Expressions. Generates code which sequentially adds any globals to the runtime environment, then generates code using a properly updated symbol table which adds recursively defined procedures to the updated runtime environment, and finally generates code in the further updated symbol table which executes the tail recursive body in the resulting runtime environment.

$$\begin{aligned}&\mathcal{CE}[\![\texttt{(let* (GSD) (letrec (LPD) T))}]\!] = \\ &\quad \lambda\gamma\,.\,\mathcal{CSDG}[\![\texttt{GSD}]\!]\gamma(\mathcal{CRDP}[\![\texttt{LPD}]\!](\mathit{extends}^*_{c\,g}\ \gamma\ \mathcal{GG}[\![\texttt{GSD}]\!]) \\ &\qquad\qquad (\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\ (\mathit{extends}^*_{c\,g}\ \gamma\ \mathcal{GG}[\![\texttt{GSD}]\!])\ \mathcal{GP}[\![\texttt{LPD}]\!])))\end{aligned}$$

9.4 Machine Instructions

In this section, we specify the semantics of each machine instruction. As noted above, a compiler is obtained instead if we replace each of these procedures by a procedure that emits the appropriate code.

$\mathit{constant} : \ E \to Q \to Q$

$\mathit{constant} = \lambda\epsilon\pi\ .\ \lambda\mu\zeta\ .\ (\pi\mu(\epsilon :: \zeta))$

$\mathit{fetch\text{-}local} : \ D_c \to Q \to Q$

$$\begin{aligned}\mathit{fetch\text{-}local} = \lambda\iota\pi\ .\ \lambda\mu\zeta\ .\ (\mu\ \iota)\ \mathrm{E}\ E \to &\\ &\pi\mu(((\mu\ \iota) \mid E) :: \zeta)), \\ &(\mathit{wrong}\ \text{“Local variable given storage.”})\end{aligned}$$

$\mathit{fetch\text{-}global} : \ D_c \to Q \to Q$

$$\begin{aligned}\mathit{fetch\text{-}global} = \lambda\iota\pi\ .\ \lambda\mu\zeta\ .\ (\mu\ \iota)\ \mathrm{E}\ L \to &\\ &(\mathit{hold}\ ((\mu\ \iota) \mid L)\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta))), \\ &(\mathit{wrong}\ \text{“Global variable not given storage.”})\end{aligned}$$

$brf : Q \to Q \to Q$

$brf = \lambda\pi_0\pi_1\lambda\mu\zeta \,.\, ((top\ \zeta) \mid T) = false \to (\pi_1\mu(pop\text{-}first\ 1\ \zeta)),\ (\pi_0\mu(pop\text{-}first\ 1\ \zeta))$

$numeric? : Q \to Q$

$numeric? = \lambda\mu\zeta \,.\, (top\ \zeta)\ \mathrm{E}\ N \to (\pi\mu\zeta),\ (wrong$ “Non-numeric argument.”$))$

$update\text{-}store : D_c \to Q \to Q$

$update\text{-}store = \lambda\iota\pi \,.\, \lambda\mu\zeta \,.\, (\mu\ \iota)\ \mathrm{E}\ L \to$
$(assign\ (\mu\ \iota) \mid L)\ (top\ \zeta)\ (\pi\mu\zeta)),$
$(wrong$ “Can’t assign to a local variable”$)$

$prim\text{-}apply : N \to P \to Q \to Q$

$prim\text{-}apply = \lambda\nu\upsilon\pi \,.\, \lambda\mu\zeta \,.\, (apply\text{-}primitive\ \upsilon\ (take\text{-}first\ \nu\ \zeta)\ (\lambda\epsilon \,.\, \pi\mu(\epsilon :: (pop\text{-}first\ \nu\ \zeta))))$

$update\text{-}store/ignore : D_c \to Q \to Q$

$update\text{-}store/ignore =$
$\lambda\iota\pi \,.\, \lambda\mu\zeta \,.\, (\mu\ \iota)\ \mathrm{E}\ L \to (assign\ ((\mu\ \iota) \mid L)(top\ \zeta)\ (\pi\mu(pop\text{-}first\ 1\ \zeta))),$
$(wrong$ “Can’t assign to a local variable”$)$

$out\text{-}of\text{-}bounds : Q$

$out\text{-}of\text{-}bounds = \lambda\mu\zeta \,.\, wrong$ “Choose: index out of bounds.”

$pick : Q \to Q \to Q$

$pick = \lambda\pi_0\pi_1 \,.\, \lambda\mu\zeta \,.\, ((top\ \zeta) \mid N) = 0 \to$
$(\pi_0\mu(pop\text{-}first\ 1\ \zeta)),$
$(\pi_1\mu((((top\ \zeta) \mid N - 1) \mid E) :: (pop\text{-}first\ 1\ \zeta)))$

$prim\text{-}apply/ignore : N \to P \to Q \to Q$

$prim\text{-}apply/ignore = \lambda\nu\upsilon\pi\,.\,\lambda\mu\zeta\,.\,(apply\text{-}primitive\,\upsilon\;(take\text{-}first\,\nu\,\zeta)\;(\lambda\epsilon\,.\,\pi\mu(pop\text{-}first\,\nu\,\zeta)))$

$halt : Q$

$halt = \lambda\mu\zeta\,.\,\kappa_0(top\,\zeta)$

$add\text{-}to\text{-}env : Q \to Q$

$add\text{-}to\text{-}env = \lambda\pi\,.\,\lambda\mu\zeta\,.\,\pi(extends_{r\,l}\,\mu\,\zeta)\langle\,\rangle$

$add\text{-}to\text{-}env^* : Q \to Q$

$add\text{-}to\text{-}env^* = \lambda\pi\,.\,\lambda\mu\zeta\,.\,\pi(extends^*_{r\,l}\,\mu\,\langle(top\,\zeta)\rangle)(pop\text{-}first\,1\,\zeta)$

$add\text{-}global\text{-}to\text{-}env^* : Q \to Q$

$add\text{-}global\text{-}to\text{-}env^* =$
$\quad\lambda\pi\,.\,\lambda\mu\zeta\,.\,tieval\,(\lambda\alpha\,.\,\pi(extends^*_{r\,g}\,\mu\,\langle\alpha\rangle)(pop\text{-}first\,1\,\zeta))\;(top\,\zeta)$

$empty\text{-}openers : O$

$empty\text{-}openers = \lambda\mu\,.\,\lambda\epsilon^*\,.\,\langle\,\rangle$

$openers : N \to Q \to O \to O$

$openers =$
$\quad\lambda\nu\pi\omega\,.\,\lambda\mu\,.\,\lambda\epsilon^*_0\,.\,((\lambda\epsilon^*\,.\,\pi(extends_{r\,l}\,(extends_{r\,l}\,\mu\,\epsilon^*)\;\epsilon^*_0)\langle\,\rangle)\text{ in }E :: (\omega\mu\epsilon^*))$

$tail\text{-}call : Q$

$tail\text{-}call = \lambda\mu\zeta\,.\,(tail\text{-}apply\,(top\,\zeta)\;(pop\text{-}first\,1\,\zeta))$

$closerecs : O \to Q \to Q$

$closerecs = \lambda\omega\pi\,.\,\lambda\mu\zeta\,.\,\pi(extends_{r\,l}\,\mu\,(fix\,(\omega\mu)))\zeta$

10 Correctness of the PurePreScheme Compiler

The main statement of the correctness proof is

$$(\mathcal{CE}[\![\mathtt{PGM}]\!]\gamma)\mu\langle\,\rangle = \mathcal{E}[\![\mathtt{PGM}]\!](\mu \circ \gamma)$$

This asserts that when the code $\mathcal{CE}[\![\mathtt{PGM}]\!]\gamma$ produced by compiling `PGM` in symbol table γ is run on the machine, starting with run-time environment μ and empty stack, the result will be the same as that of applying the semantics of $\mathcal{E}[\![\mathtt{PGM}]\!]$ of `PGM` to the environment $(\mu \circ \gamma)$ obtained by composing μ and γ.

The proof is a tedious but straightforward structural induction. There are a total of 13 simultaneous induction hypotheses, one for each function in the compiler. Each induction hypothesis specifies the behavior of one compiler function relative to a suitable semantic valuation.

These specifications generally fall into two categories. The first flavor relates semantic valuations which do not take expression continuations to their corresponding compiler valuations. The main statement above falls into this category. The corresponding compiler functions do not take a continuation argument π. Furthermore, these program phrases run only when the run-time stack ζ is empty.

The second of induction hypothesis deals with semantic valuations which do take expression continuations. In this case code is compiled using a symbol table and the code to follow the given code, π. A typical induction hypothesis in this category is

$$(\mathcal{CSE}[\![\mathtt{S}]\!]\gamma\pi)\mu\zeta = \mathcal{SE}[\![\mathtt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$$

This asserts that when the code $\mathcal{CSE}[\![\mathtt{S}]\!]\gamma\pi$, produced by compiling `S` in symbol table γ and code continuation π, is run on the machine, starting with run-time environment μ and stack ζ, the result is the same as evaluating the semantics of `S` with environment $(\mu \circ \gamma)$ and a continuation which places the value ϵ on the stack ζ and then runs the code continuation π. Informally, this is read as "the code for `S` puts its result on the stack and continues with π."

When sequences are involved, the push operation :: is typically replaced by an append operation : .

Induction Hypothesis 0

$(\mathcal{CDL}[\![\texttt{LSD}]\!]\gamma\pi)\mu\zeta = \mathcal{DL}[\![\texttt{LSD}]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \pi\mu(\epsilon^* : \zeta))$

The induction hypothesis states that running local declarations compiled using symbol table γ with the code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation which executes π in the same runtime environment but with all the values pushed onto the runtime stack.

$(\mathcal{CDL}[\![\ ]\!]\gamma\pi)\mu\zeta$

$= \pi\mu\zeta$

$= \mathcal{DL}[\![\ ]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \pi\mu(\epsilon^* : \zeta))$

$(\mathcal{CDL}[\![\texttt{(L S) LSD}]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CDL}[\![\texttt{LSD}]\!]\gamma(\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)\mu\zeta$

$= \mathcal{DL}[\![\texttt{LSD}]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, (\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)\mu(\epsilon^* : \zeta))$

$= \mathcal{DL}[\![\texttt{LSD}]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon \,.\, \pi\mu((\epsilon :: \epsilon^*) : \zeta)))$

$= \mathcal{DL}[\![\texttt{(L S) LSD}]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \pi\mu(\epsilon^* : \zeta))$

Induction Hypothesis 1

$(\mathcal{CSDL}[\![\texttt{LSD}]\!]\gamma\pi)\mu\langle\,\rangle = \mathcal{SDL}[\![\texttt{LSD}]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \pi(\mathit{extends}^*_{r\,l}\ \mu\ \epsilon^*)\langle\,\rangle)$

The induction hypothesis states that running local sequential declarations compiled using symbol table γ with the code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation which executes π in with the runtime environment extended with the values and the same runtime stack.

$(\mathcal{CSDL}[\![\ ]\!]\gamma\pi)\mu\langle\ \rangle$

$= \pi\mu\langle\ \rangle$

$= \mathcal{SDL}[\![\ ]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \pi(\mathit{extends}^*_{r\,l}\ \mu\ \epsilon^*)\langle\ \rangle)$

$(\mathcal{CSDL}[\![\texttt{(L S) LSD}]\!]\gamma\pi)\mu\langle\ \rangle$

$= (\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{add\text{-}to\text{-}env^*}\,(\mathcal{CSDL}[\![\texttt{LSD}]\!](\mathit{extends}^*_{c\,l}\ \gamma\ \langle\texttt{L}\rangle)\pi))\mu\langle\ \rangle$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\mathit{add\text{-}to\text{-}env^*}\,(\mathcal{CSDL}[\![\texttt{LSD}]\!](\mathit{extends}^*_{c\,l}\ \gamma\ \langle\texttt{L}\rangle)\pi))\mu(\epsilon :: \langle\ \rangle))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\mathit{add\text{-}to\text{-}env^*}\,(\mathcal{CSDL}[\![\texttt{LSD}]\!](\mathit{extends}^*_{c\,l}\ \gamma\ \langle\texttt{L}\rangle)\pi))\mu(\epsilon :: \langle\ \rangle))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\mathcal{CSDL}[\![\texttt{LSD}]\!](\mathit{extends}^*_{c\,l}\ \gamma\ \langle\texttt{L}\rangle)\pi)(\mathit{extends}^*_{r\,l}\ \mu\ \langle\epsilon\rangle)\langle\ \rangle)$

$$\begin{aligned}= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,&\mathcal{SDL}[\![\texttt{LSD}]\!](\mu \circ \gamma)[\epsilon \,\mathrm{in}\, D/\texttt{L}]\\ &(\lambda\epsilon^* \,.\, \pi(\mathit{extends}^*_{r\,l}\ \mu\ (\epsilon :: \epsilon^*))\langle\ \rangle))\end{aligned}$$

$= \mathcal{SDL}[\![\texttt{LSD}]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \pi(\mathit{extends}_{r\,l}\ \mu\ \epsilon^*)\langle\ \rangle)$

Induction Hypothesis 2

$(\mathcal{CSDG}[\![\texttt{GSD}]\!]\gamma\pi)\mu\langle\ \rangle = \mathcal{SDG}[\![\texttt{GSD}]\!](\mu \circ \gamma)(\lambda\alpha^* \,.\, \pi(\mathit{extends}^*_{r\,g}\ \mu\ \alpha^*)\langle\ \rangle)$

The induction hypothesis states that running global declarations compiled using symbol table γ with code to follow π in runtime environment μ with an empty runtime stack is the same as the semantics using composition of μ and γ for an environment with a continuation which executes π in with the runtime environment extended with the new locations and the same empty runtime stack.

$(\mathcal{CSDG}[\![\ ]\!]\gamma\pi)\mu\langle\ \rangle$

$= \pi\mu\langle\ \rangle$

$= \mathcal{SDG}[\![\ ]\!](\mu \circ \gamma)(\lambda\alpha^* \,.\, \pi(\mathit{extends}^*_{r\,g}\ \mu\ \alpha^*)\langle\ \rangle)$

$(\mathcal{CSDG}[\![\texttt{(G S) GSD}]\!]\gamma\pi)\mu\langle\,\rangle$

$= (\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\textit{add-global-to-env*}\,(\mathcal{CSDG}[\![\texttt{GSD}]\!](\textit{extends}^*_{c\,g}\,\gamma\,\langle\texttt{G}\rangle)\,\pi)))\mu\langle\,\rangle$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)$
$\quad(\lambda\epsilon\,.\,(\textit{add-global-to-env*}\,(\mathcal{CSDG}[\![\texttt{GSD}]\!](\textit{extends}^*_{c\,g}\,\gamma\,\langle\texttt{G}\rangle)\,\pi))\mu(\epsilon::\langle\,\rangle))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)$
$\quad(\lambda\epsilon\,.\,\textit{tieval}\,(\lambda\alpha\,.\,(\mathcal{CSDG}[\![\texttt{GSD}]\!](\textit{extends}^*_{c\,g}\,\gamma\,\langle\texttt{G}\rangle)\,\pi)(\textit{extends}^*_{r\,g}\,\mu\,\langle\alpha\rangle)\langle\,\rangle)\,\epsilon)$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)$
$\quad(\lambda\epsilon\,.\,\textit{tieval}\,(\lambda\alpha\,.\,\mathcal{SDG}[\![\texttt{GSD}]\!](\mu\circ\gamma)[\alpha\,\mathrm{in}\,D/\texttt{G}]$
$\qquad\qquad(\lambda\alpha^*\,.\,\pi(\textit{extends}^*_{r\,g}\,\mu\,(\alpha::\alpha^*))\langle\,\rangle))\,\epsilon)$

$= \mathcal{SDG}[\![\texttt{(G S)GSD}]\!](\mu\circ\gamma)(\lambda\alpha^*\,.\,\pi(\textit{extends}^*_{r\,g}\,\mu\,\alpha^*)\langle\,\rangle)$

Induction Hypothesis 3

$(\mathcal{CRP}[\![\texttt{LPD}]\!]\texttt{L}^*\gamma)\mu\epsilon^* = \mathcal{RP}[\![\texttt{LPD}]\!]\texttt{L}^*(\mu\circ\gamma)\epsilon^*$

The induction hypothesis states that recursive procedure declarations compiled using symbol table γ running in runtime environment μ with is the same as the semantics using composition of μ and γ for an environment.

$(\mathcal{CRP}[\![\;]\!]\texttt{L}^*\gamma)\mu\epsilon^*$

$= \textit{empty-openers}\,\mu\,\epsilon^*$

$= \langle\,\rangle$

$= \mathcal{RP}[\![\;]\!]\texttt{L}^*(\mu\circ\gamma)\epsilon^*$

$(\mathcal{CRP}[\![\texttt{(L (lambda (L}^*\texttt{) T)) LPD}]\!]\texttt{L}^*_0\gamma)\mu\epsilon^*_0$

$=(\textit{openers}$
$\quad(\mathcal{CT}[\![\texttt{T}]\!](\textit{extends}_{c\,l}\,(\textit{extends}_{c\,l}\,\gamma\,(\textit{reverse}\,\texttt{L}^*))\,\texttt{L}^*_0))$

$$(\mathcal{CRP}[\![\texttt{LPD}]\!]\texttt{L}_0^*\gamma))\mu\epsilon^*$$

$$\begin{aligned}=&(((\lambda\epsilon^* \,.\, (\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\ (\mathit{extends}_{c\,l}\ \gamma\ (\mathit{reverse}\ \texttt{L}^*))\ \texttt{L}_0^*))\\ &\qquad(\mathit{extends}_{r\,l}\ (\mathit{extends}_{r\,l}\ \mu\ \epsilon^*)\ \epsilon_0^*)\langle\ \rangle)\ \text{in}\ E)\\ &\ :: (\mathcal{CRP}[\![\texttt{LPD}]\!]\texttt{L}_0^*\gamma)\mu\epsilon^*)\end{aligned}$$

$$\begin{aligned}=&(((\lambda\epsilon^* \,.\, \mathcal{T}[\![\texttt{T}]\!]((\mu\circ\gamma)[(\mathit{map}\ (\text{in}\ D)\ \epsilon^*)/\mathit{reverse}\ \texttt{L}^*])[(\mathit{map}\ (\text{in}\ D)\ \epsilon_0^*)/\texttt{L}_0^*])\ \text{in}\ E)\\ &:: \mathcal{RP}[\![\texttt{LPD}]\!]\texttt{L}_0^*(\mu\circ\gamma)\epsilon^*\end{aligned}$$

$$= \mathcal{RP}[\![\texttt{(L (lambda (L}^*\texttt{) T)) LPD}]\!]\texttt{L}_0^*(\mu\circ\gamma)\epsilon^*$$

Induction Hypothesis 4

$$(\mathcal{CRDP}[\![\texttt{LPD}]\!]\gamma\pi)\mu\langle\ \rangle = \mathcal{RDP}[\![\texttt{LPD}]\!](\mu\circ\gamma)(\lambda\epsilon^* \,.\, \pi(\mathit{extends}_{r\,l}\ \mu\ \epsilon^*)\langle\ \rangle)$$

The induction hypothesis states that running recursive procedure declarations compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the runtime environment extended with the procedure and the same runtime stack.

$$(\mathcal{CRDP}[\![\texttt{LPD}]\!]\gamma\pi)\mu\langle\ \rangle$$

$$= (\mathit{closerecs}\ (\mathcal{CRP}[\![\texttt{LPD}]\!](\mathcal{GP}[\![\texttt{LPD}]\!])\gamma)\ \pi)\mu\langle\ \rangle$$

$$= \pi(\mathit{extends}_{r\,l}\ \mu\ (\mathit{fix}\ ((\mathcal{CRP}[\![\texttt{LPD}]\!](\mathcal{GP}[\![\texttt{LPD}]\!])\gamma)\mu)))\langle\ \rangle$$

$$= \pi(\mathit{extends}_{r\,l}\ \mu\ (\mathit{fix}\ (\mathcal{RP}[\![\texttt{LPD}]\!](\mathcal{GP}[\![\texttt{LPD}]\!])(\mu\circ\gamma))))\langle\ \rangle$$

$$= \mathcal{RDP}[\![\texttt{LPD}]\!](\mu\circ\gamma)(\lambda\epsilon^* \,.\, \pi(\mathit{extends}_{r\,l}\ \mu\ \epsilon^*)\langle\ \rangle)$$

Induction Hypothesis 5

$$(\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)\mu\zeta = \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon \,.\, \pi\mu(\epsilon :: \zeta))$$

The induction hypothesis states that running simple expressions compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the same runtime environment but with the result pushed onto the runtime stack.

$(\mathcal{CSE}[\![\mathtt{K}]\!]\gamma\pi)\mu\zeta$

$= (\mathit{constant}\,(\mathcal{K}[\![\mathtt{K}]\!])\ \pi)\mu\zeta$

$= \pi\mu((\mathcal{K}[\![\mathtt{K}]\!]) :: \zeta)$

$= \mathcal{SE}[\![\mathtt{K}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![\mathtt{L}]\!]\gamma\pi)\mu\zeta$

$= (\mathit{fetch\text{-}local}\,(\gamma\ \mathtt{L})\ \pi)\mu\zeta$

$= ((\mu \circ \gamma)\mathtt{L})\ \mathrm{E}\ E \to \pi\mu((((\mu \circ \gamma)\mathtt{L}) \mid E) :: \zeta),$
$(\mathit{wrong}$ “Local variable given storage.”$)$

$= \mathcal{SE}[\![\mathtt{L}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![\mathtt{G}]\!]\gamma\pi)\mu\zeta$

$= (\mathit{fetch\text{-}global}\,(\gamma\ \mathtt{G})\ \pi)\mu\zeta$

$= ((\mu \circ \gamma)\mathtt{G})\ \mathrm{E}\ L \to (\mathit{hold}\,(((\mu \circ \gamma)\mathtt{G}) \mid L)\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta))),$
$(\mathit{wrong}$ “Global variable not given storage.”$)$

$= \mathcal{SE}[\![\mathtt{G}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![\mathtt{(if\ S_0\ S_1\ S_2)}]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CSE}[\![\mathtt{S}_0]\!]\gamma(\mathit{brf}\,(\mathcal{CSE}[\![\mathtt{S}_1]\!]\gamma\pi)\ (\mathcal{CSE}[\![\mathtt{S}_2]\!]\gamma\pi)))\mu\zeta$

$= \mathcal{SE}[\![\mathtt{S}_0]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\mathit{brf}\,(\mathcal{CSE}[\![\mathtt{S}_1]\!]\gamma\pi)\ \ (\mathcal{CSE}[\![\mathtt{S}_2]\!]\gamma\pi))\mu(\epsilon :: \zeta))$

$$\begin{aligned}= \mathcal{SE}[\![\mathtt{S}_0]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\epsilon \mid T) = \mathit{false} \to &((\mathcal{CSE}[\![\mathtt{S}_2]\!]\gamma\pi)\mu\zeta),\\ &((\mathcal{CSE}[\![\mathtt{S}_1]\!]\gamma\pi)\mu\zeta))\end{aligned}$$

$$\begin{aligned}= \mathcal{SE}[\![\mathtt{S}_0]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\epsilon \mid T) = \mathit{false} \to &\mathcal{SE}[\![\mathtt{S}_2]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \gamma))\\ &\mathcal{SE}[\![\mathtt{S}_1]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta)))\end{aligned}$$

$= \mathcal{SE}[\![\mathtt{(if\ S_0\ S_1\ S_2)}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![\mathtt{(choose\ S\ (S^*))}]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CSE}[\![\mathtt{S}]\!]\gamma(\mathit{numeric?}\,(\mathcal{CSCE}^*[\![\mathtt{S}^*]\!]\gamma\pi))\mu\zeta)$

$= \mathcal{SE}[\![\mathtt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\mathit{numeric?}\,(\mathcal{CSCE}^*[\![\mathtt{S}^*]\!]\gamma\pi))\mu(\epsilon :: \zeta))$

$$\begin{aligned}= \mathcal{SE}[\![\mathtt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\epsilon \mathrel{\mathrm{E}} N \to\ &(\mathcal{CSCE}^*[\![\mathtt{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta),\\ &(\mathit{wrong}\ \text{“Non-numeric argument.”}))\end{aligned}$$

$$\begin{aligned}= \mathcal{SE}[\![\mathtt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\epsilon \mathrel{\mathrm{E}} N \to\ &\mathcal{SCE}^*[\![\mathtt{S}^*]\!](\epsilon \mid N)(\mu\circ\gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta)),\\ &(\mathit{wrong}\ \text{“Non-numeric argument.”}))\end{aligned}$$

$= \mathcal{SE}[\![\mathtt{(choose\ S\ (S^*))}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![\mathtt{(set!\ G\ S)}]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CSE}[\![\mathtt{S}]\!]\gamma(\mathit{update\text{-}store}\ (\gamma\,\mathtt{G})\ \pi))\mu\zeta$

$= \mathcal{SE}[\![\mathtt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\mathit{update\text{-}store}\ (\gamma\,\mathtt{G})\ \pi)\mu(\epsilon :: \zeta))$

$$\begin{aligned}= \mathcal{SE}[\![\mathtt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,((\mu\circ\gamma)\mathtt{G}) \mathrel{\mathrm{E}} L \to &(\mathit{assign}\ (((\mu\circ\gamma)\mathtt{G}) \mid L)\ \epsilon\ (\pi\mu(\epsilon :: \zeta))),\\ &(\mathit{wrong}\ \text{“Can't assign to a local variable”}))\end{aligned}$$

$= \mathcal{SE}[\![\mathtt{(set!\ G\ S)}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![\texttt{(P S}^*\texttt{)}]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma(\mathit{prim\text{-}apply}\ \#\texttt{S}^*\ \mathcal{P}[\![\texttt{P}]\!]\ \pi))\mu\zeta$

$= \mathcal{SE}^*[\![\texttt{S}^*]\!](\mu\circ\gamma)(\lambda\epsilon^*\ .\ (\mathit{prim\text{-}apply}\ \#\texttt{S}^*\ \mathcal{P}[\![\texttt{P}]\!]\ \pi)\mu(\epsilon^* : \zeta))$

$= \mathcal{SE}^*[\![\texttt{S}^*]\!](\mu\circ\gamma)(\lambda\epsilon^*\ .\ \mathit{apply\text{-}primitive}\ \mathcal{P}[\![\texttt{P}]\!]\ (\mathit{take\text{-}first}\ \#\texttt{S}^*\ (\epsilon^* : \zeta))$
$\qquad (\lambda\epsilon\ .\ \pi\mu(\epsilon :: (\mathit{pop\text{-}first}\ \#\texttt{S}^*\ (\epsilon^* : \zeta)))))$

$= \mathcal{SE}^*[\![\texttt{S}^*]\!](\mu\circ\gamma)(\lambda\epsilon^*\ .\ \mathit{apply\text{-}primitive}\ \mathcal{P}[\![\texttt{P}]\!]\ \epsilon^*\ (\lambda\epsilon\ .\ \pi\mu(\epsilon :: \zeta)))$

$= \mathcal{SE}[\![\texttt{(P S}^*\texttt{)}]\!](\mu\circ\gamma)(\lambda\epsilon\ .\ \pi\mu(\epsilon :: \zeta))$

Induction Hypothesis 6

$(\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma\pi)\mu\zeta = \mathcal{SE}^*[\![\texttt{S}^*]\!](\mu\circ\gamma)(\lambda\epsilon^*\ .\ \pi\mu(\epsilon^* : \zeta))$

The induction hypothesis states that running simple expression sequences compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the same runtime environment but with the resulting values pushed onto the runtime stack.

$(\mathcal{CSE}^*[\![\ ]\!]\gamma\pi)\mu\zeta$

$= \pi\mu\zeta$

$= \mathcal{SE}^*[\![\ ]\!](\mu\circ\gamma)(\lambda\epsilon^*\ .\ \pi\mu(\epsilon^* : \zeta))$

$(\mathcal{CSE}^*[\![\texttt{S S}^*]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma\pi))\mu\zeta$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\ .\ (\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\mathcal{SE}^*[\![\texttt{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^*\,.\,\pi\mu(\epsilon^* : (\epsilon :: \zeta))))$

$= \mathcal{SE}^*[\![\texttt{S S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^*\,.\,\pi\mu(\epsilon^* : \zeta))$

Induction Hypothesis 7

$(\mathcal{CSEC}^*[\![\texttt{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta) = \mathcal{SEC}^*[\![\texttt{S}^*]\!](\epsilon \mid N)(\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

The induction hypothesis states that running simple expressions choose sequences compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the same runtime environment but with the result pushed onto the runtime stack.

$(\mathcal{CSEC}^*[\![\ ]\!]\gamma\pi)\mu(\epsilon :: \zeta)$

$= (\textit{out-of-bounds})\mu(\epsilon :: \zeta)$

$=$ *wrong* "Choose: index out of bounds."

$= \mathcal{SEC}^*[\![\ ]\!](\epsilon \mid N)(\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSEC}^*[\![\texttt{S S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta)$

$= (\textit{pick}\,(\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)\ (\mathcal{CSEC}^*[\![\texttt{S}^*]\!]\gamma\pi))\mu(\epsilon :: \zeta)$

$= ((\epsilon \mid N) = 0) \rightarrow (\mathcal{CSE}[\![\texttt{S}]\!]\gamma\pi)\mu\zeta,\ (\mathcal{CSEC}^*[\![\texttt{S}^*]\!]\gamma\pi)\mu(((\epsilon \mid N-1) \mid E) :: \zeta)$

$$\begin{aligned}= ((\epsilon \mid N) = 0) \rightarrow\ & \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta)),\\ & (\mathcal{CSEC}^*[\![\texttt{S}^*]\!]\gamma\pi)\mu(((\epsilon \mid N-1) \mid E) :: \zeta)\end{aligned}$$

$$\begin{aligned}= ((\epsilon \mid N) = 0) \rightarrow\ & \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta)),\\ & \mathcal{SEC}^*[\![\texttt{S}^*]\!](\epsilon \mid N-1)(\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))\end{aligned}$$

$= \mathcal{SEC}^*[\![\texttt{S S}^*]\!](\epsilon \mid N)(\mu \circ \gamma)(\lambda\epsilon\,.\,\pi\mu(\epsilon :: \zeta))$

Induction Hypothesis 8

$$(\mathcal{CSC}[\![\texttt{S}]\!]\gamma\pi)\mu\zeta = \mathcal{SC}[\![\texttt{S}]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

The induction hypothesis states that running simple commands compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the same runtime environment and stack (i.e. the value is ignored).

$(\mathcal{CSC}[\![\texttt{K}]\!]\gamma\pi)\mu\zeta$

$= \pi\mu\zeta$

$= \mathcal{SC}[\![\texttt{K}]\!](\mu \circ \gamma)(\pi\mu\zeta)$

$(\mathcal{CSC}[\![\texttt{L}]\!]\gamma\pi)\mu\zeta$

$= \pi\mu\zeta$

$= \mathcal{SC}[\![\texttt{L}]\!](\mu \circ \gamma)(\pi\mu\zeta)$

$(\mathcal{CSC}[\![\texttt{G}]\!]\gamma\pi)\mu\zeta$

$= \pi\mu\zeta$

$= \mathcal{SC}[\![\texttt{G}]\!](\mu \circ \gamma)(\pi\mu\zeta)$

$(\mathcal{CSC}[\![\texttt{(if S}_0\texttt{ S}_1\texttt{ S}_2\texttt{)}]\!]\gamma\pi)\mu\zeta$

$= (\mathcal{CSE}[\![\texttt{S}_0]\!]\gamma(\mathit{brf}(\mathcal{CSC}[\![\texttt{S}_1]\!]\gamma\pi)\ (\mathcal{CSC}[\![\texttt{S}_2]\!]\gamma\pi)))\mu\zeta$

$= \mathcal{SE}[\![\texttt{S}_0]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\mathit{brf}(\mathcal{CSC}[\![\texttt{S}_1]\!]\gamma\pi)\ \ (\mathcal{CSC}[\![\texttt{S}_2]\!]\gamma\pi))\mu(\epsilon :: \zeta))$

$= \mathcal{SE}[\![\texttt{S}_0]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\epsilon\mu T) = \mathit{false} \to ((\mathcal{CSC}[\![\texttt{S}_2]\!]\gamma\pi)\mu\zeta),\ ((\mathcal{CSC}[\![\texttt{S}_1]\!]\gamma\pi)\mu\zeta))$

$$= \mathcal{SE}[\![\texttt{S}_0]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\epsilon\mu T) = \mathit{false} \rightarrow \mathcal{SC}[\![\texttt{S}_2]\!](\mu \circ \gamma)(\pi\mu\zeta),\; \mathcal{SC}[\![\texttt{S}_1]\!](\mu \circ \gamma)(\pi\mu\zeta))$$

$$= \mathcal{SC}[\![\texttt{(if S}_0\texttt{ S}_1\texttt{ S}_2\texttt{)}]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

$$(\mathcal{CSC}[\![\texttt{(choose S (S}^*\texttt{))}]\!]\gamma\pi)\mu\zeta$$

$$= (\mathcal{CSC}[\![\texttt{S}]\!]\gamma(\mathit{numeric?}\,(\mathcal{CSCC}^*[\![\texttt{S}^*]\!]\gamma\pi))\mu\zeta)$$

$$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\mathit{numeric?}\,(\mathcal{CSCC}^*[\![\texttt{S}^*]\!]\gamma\pi))\mu(\epsilon :: \zeta))$$

$$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\epsilon \mathrel{\mathrm{E}} N \rightarrow (\mathcal{CSCC}^*[\![\texttt{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta),\; (\mathit{wrong}\text{ “Non-numeric argument.”}))$$

$$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,\epsilon \mathrel{\mathrm{E}} N \rightarrow \mathcal{SCC}^*[\![\texttt{S}^*]\!](\epsilon \mid N)(\mu \circ \gamma)(\pi\mu\zeta),\; (\mathit{wrong}\text{ “Non-numeric argument.”}))$$

$$= \mathcal{SC}[\![\texttt{(choose S (S}^*\texttt{))}]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

$$(\mathcal{CSC}[\![\texttt{(set! G S)}]\!]\gamma\pi)\mu\zeta$$

$$= (\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{update\text{-}store/ignore}\;(\gamma\,\texttt{G})\;\pi))\mu\zeta$$

$$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,(\mathit{update\text{-}store}\;(\gamma\,\texttt{G})\;\pi)\mu(\epsilon :: \zeta))$$

$$= \mathcal{SE}[\![\texttt{S}]\!](\mu \circ \gamma)(\lambda\epsilon\,.\,((\mu \circ \gamma)\texttt{G}) \mathrel{\mathrm{E}} L \rightarrow (\mathit{assign}\;(((\mu \circ \gamma)\texttt{G}) \mid L)\;\epsilon\;(\pi\mu\zeta)),\; (\mathit{wrong}\text{ “Can't assign to a local variable”}))$$

$$= \mathcal{SE}[\![\texttt{(set! G S)}]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

$$(\mathcal{CSC}[\![\texttt{(P S}^*\texttt{)}]\!]\gamma\pi)\mu\zeta$$

$$= (\mathcal{CSE}^*[\![\texttt{S}^*]\!]\gamma(\mathit{prim\text{-}apply/ignore}\;\#\texttt{S}^*\;\mathcal{P}[\![\texttt{P}]\!]\;\pi))\mu\zeta$$

$$= \mathcal{SE}^*[\![\texttt{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^*\,.\,(\mathit{prim\text{-}apply/ignore}\;\#\texttt{S}^*\;\mathcal{P}[\![\texttt{P}]\!]\;\pi)\mu(\epsilon^* : \zeta))$$

$$= \mathcal{SE}^*[\![\mathtt{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \textit{apply-primitive/ignore}\, \mathcal{P}[\![\mathtt{P}]\!]\ (\textit{take-first}\, \#\mathtt{S}^*\ (\epsilon^* : \zeta))$$
$$(\pi\mu(\textit{pop-first}\, \#\mathtt{S}^*\ (\epsilon^* : \zeta))))$$

$$= \mathcal{SE}^*[\![\mathtt{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^* \,.\, \textit{apply-primitive/ignore}\, \mathcal{P}[\![\mathtt{P}]\!]\ \epsilon^*\ (\pi\mu\zeta))$$

$$= \mathcal{SC}[\![\mathtt{(P\ S^*)}]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

Induction Hypothesis 9

$$(\mathcal{CSC}^*[\![\mathtt{S}^*]\!]\gamma\pi)\mu\zeta = \mathcal{SC}^*[\![\mathtt{S}^*]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

The induction hypothesis states that running simple command sequences compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the same runtime environment and stack (i.e. the value is ignored).

$$(\mathcal{CSC}^*[\![\ ]\!]\gamma\pi)\mu\zeta$$

$$= \pi\mu\zeta$$

$$= \mathcal{SC}^*[\![\ ]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

$$(\mathcal{CSC}^*[\![\mathtt{S\ S^*}]\!]\gamma\pi)\mu\zeta$$

$$= (\mathcal{CSC}[\![\mathtt{S}]\!]\gamma(\mathcal{CSC}^*[\![\mathtt{S}^*]\!]\gamma\pi))\mu\zeta$$

$$= \mathcal{SC}[\![\mathtt{S}]\!](\mu \circ \gamma)((\mathcal{CSC}^*[\![\mathtt{S}^*]\!]\gamma\pi)\mu\zeta)$$

$$= \mathcal{SC}[\![\mathtt{S}]\!](\mu \circ \gamma)(\mathcal{SC}^*[\![\mathtt{S}^*]\!](\mu \circ \gamma)(\pi\mu\zeta))$$

$$= \mathcal{SC}^*[\![\mathtt{S\ S^*}]\!](\mu \circ \gamma)(\pi\mu\zeta)$$

Induction Hypothesis 10

$$(\mathcal{CSCC}^*[\![\mathtt{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta) = \mathcal{SCC}^*[\![\mathtt{S}^*]\!](\epsilon \mid N)(\mu \circ \gamma)(\pi\mu\zeta)$$

The induction hypothesis states that running simple command choose sequences compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics using composition of μ and γ for an environment with a continuation executing π in with the same runtime environment and stack (i.e. the value is ignored).

$(\mathcal{CSCC}^*[\![\ ]\!]\gamma\pi)\mu(\epsilon :: \zeta)$

$= (\textit{out-of-bounds})\mu(\epsilon :: \zeta)$

$= \textit{wrong}$ "Choose: index out of bounds."

$= \mathcal{SCC}^*[\![\ ]\!](\epsilon \mid N)(\mu \circ \gamma)(\pi\mu\zeta)$

$(\mathcal{CSCC}^*[\![\mathtt{S}\ \mathtt{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta)$

$= (\textit{pick}\,(\mathcal{CSC}[\![\mathtt{S}]\!]\gamma\pi)\ (\mathcal{CSCC}^*[\![\mathtt{S}^*]\!]\gamma\pi))\mu(\epsilon :: \zeta)$

$= (\epsilon \mid N) = 0 \rightarrow (\mathcal{CSC}[\![\mathtt{S}]\!]\gamma\pi)\mu\zeta,\ (\mathcal{CSCC}^*[\![\mathtt{S}^*]\!]\gamma\pi)\mu(((\epsilon \mid N-1) \mid E) :: \zeta)$

$= (\epsilon \mid N) = 0 \rightarrow \mathcal{SC}[\![\mathtt{S}]\!](\mu \circ \gamma)(\pi\mu\zeta),\ (\mathcal{CSCC}^*[\![\mathtt{S}^*]\!]\gamma\pi)\mu(((\epsilon \mid N-1) \mid E) :: \zeta)$

$= (\epsilon \mid N) = 0 \rightarrow \mathcal{SC}[\![\mathtt{S}]\!](\mu \circ \gamma)(\pi\mu\zeta),\ \mathcal{SCC}^*[\![\mathtt{S}^*]\!](\epsilon \mid N-1)(\mu \circ \gamma)(\pi\mu\zeta)$

$= \mathcal{SCC}^*[\![\mathtt{S}\ \mathtt{S}^*]\!](\epsilon \mid N)(\mu \circ \gamma)(\pi\mu\zeta)$

Induction Hypothesis 11

$(\mathcal{CT}[\![\mathtt{T}]\!]\gamma)\mu\langle\,\rangle = \mathcal{T}[\![\mathtt{T}]\!](\mu \circ \gamma)$

The induction hypothesis states that running tail recursive expressions compiled using symbol table γ in runtime environment μ with an empty runtime stack is the same as the semantics using composition of μ and γ for an environment.

$(\mathcal{CT}[\![\mathtt{S}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CSE}[\![\texttt{S}]\!]\gamma(\mathit{halt}))\mu\langle\,\rangle$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\mathit{halt})\mu(\epsilon :: \langle\,\rangle))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,\kappa_0\epsilon)$

$= \mathcal{T}[\![\texttt{S}]\!](\mu\circ\gamma)$

$(\mathcal{CT}[\![\texttt{(if S T}_0\texttt{ T}_1\texttt{)}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CSE}[\![\mathrm{S}]\!]\gamma(\mathit{brf}\,(\mathcal{CT}[\![\texttt{T}_0]\!]\gamma)\ (\mathcal{CT}[\![\texttt{T}_1]\!]\gamma)))\mu\langle\,\rangle$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\mathit{brf}\,(\mathcal{CT}[\![\texttt{T}_0]\!]\gamma)\ (\mathcal{CT}[\![\texttt{T}_1]\!]\gamma))\mu(\epsilon :: \langle\,\rangle))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\epsilon \mid T) = \mathit{false} \rightarrow ((\mathcal{CT}[\![\texttt{T}_1]\!]\gamma)\mu\langle\,\rangle),\ ((\mathcal{CT}[\![\texttt{T}_0]\!]\gamma)\mu\langle\,\rangle))$

$= \mathcal{SE}[\![\texttt{S}]\!](\mu\circ\gamma)(\lambda\epsilon\,.\,(\epsilon \mid T) = \mathit{false} \rightarrow \mathcal{T}[\![\texttt{T}_1]\!](\mu\circ\gamma),\ \mathcal{T}[\![\texttt{T}_0]\!](\mu\circ\gamma))$

$= \mathcal{T}[\![\texttt{(if S T}_0\texttt{ T}_0\texttt{)}]\!](\mu\circ\gamma)$

$(\mathcal{CT}[\![\texttt{(begin S}^*\texttt{ T)}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CSC}[\![\texttt{S}^*]\!]\gamma(\mathcal{CT}[\![\texttt{T}]\!]\gamma))\mu\langle\,\rangle$

$= \mathcal{SC}[\![\texttt{S}^*]\!](\mu\circ\gamma)((\mathcal{CT}[\![\texttt{T}]\!]\gamma)\mu\langle\,\rangle)$

$= \mathcal{SC}[\![\texttt{S}^*]\!](\mu\circ\gamma)(\mathcal{T}[\![\texttt{T}]\!](\mu\circ\gamma))$

$= \mathcal{T}[\![\texttt{(begin S}^*\texttt{ T)}]\!](\mu\circ\gamma)$

$(\mathcal{CT}[\![\texttt{(let (LSD) T)}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CDL}[\![\texttt{LSD}]\!]\gamma(\mathit{add\text{-}to\text{-}env}\,(\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\,\gamma\ (\mathcal{GL}[\![\texttt{LSD}]\!])))))\mu\langle\,\rangle$

$= \mathcal{DL}[\![\mathtt{LSD}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, (\mathit{add\text{-}to\text{-}env}\,(\mathcal{CT}[\![\mathtt{T}]\!](\mathit{extends}_{c\,l}\,\gamma\,(\mathcal{GL}[\![\mathtt{LSD}]\!]))))\mu(\epsilon^* : \langle\,\rangle))$

$= \mathcal{DL}[\![\mathtt{LSD}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, (\mathcal{CT}[\![\mathtt{T}]\!](\mathit{extends}_{c\,l}\,\gamma\,(\mathcal{GL}[\![\mathtt{LSD}]\!])))(\mathit{extends}_{r\,l}\,\mu\,\epsilon^*)\langle\,\rangle)$

$= \mathcal{DL}[\![\mathtt{LSD}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, \mathcal{T}[\![\mathtt{T}]\!](\mu \circ \gamma)[(\mathit{map}\,(\mathrm{in}\,D)\,\epsilon^*)/\mathcal{GL}[\![\mathtt{LSD}]\!]])$

$= \mathcal{T}[\![\mathtt{(let\ (LSD)\ T)}]\!](\mu \circ \gamma)$

$(\mathcal{CT}[\![\mathtt{(let*\ (LSD)\ T)}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CSDL}[\![\mathtt{LSD}]\!]\gamma(\mathcal{CT}[\![\mathtt{T}]\!](\mathit{extends}^*_{c\,l}\,\gamma\,(\mathcal{GL}[\![\mathtt{LSD}]\!]))))\mu\langle\,\rangle$

$= \mathcal{SDL}[\![\mathtt{LSD}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, (\mathcal{CT}[\![\mathtt{T}]\!](\mathit{extends}^*_{c\,l}\,\gamma\,(\mathcal{GL}[\![\mathtt{LSD}]\!])))(\mathit{extends}^*_{r\,l}\,\mu\,\epsilon^*)\langle\,\rangle)$

$= \mathcal{SDL}[\![\mathtt{LSD}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, \mathcal{T}[\![\mathtt{T}]\!](\mu \circ \gamma)[(\mathit{map}\,(\mathrm{in}\,D)\,\epsilon^*)/\mathcal{GL}[\![\mathtt{LSD}]\!]^*)$

$= \mathcal{T}[\![\mathtt{(let*\ (LSD)\ T)}]\!](\mu \circ \gamma)$

$(\mathcal{CT}[\![\mathtt{(S\ S^*)}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CSE}^*[\![\mathtt{S^*}]\!]\gamma(\mathcal{CSE}[\![\mathtt{S}]\!]\gamma(\mathit{tail\text{-}call})))\mu\langle\,\rangle$

$= \mathcal{SE}^*[\![\mathtt{S^*}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, (\mathcal{CSE}[\![\mathtt{S}]\!]\gamma(\mathit{tail\text{-}call}))\mu(\epsilon^* : \langle\,\rangle))$

$= \mathcal{SE}^*[\![\mathtt{S^*}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, \mathcal{SE}[\![\mathtt{S}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, (\mathit{tail\text{-}call})\mu(\epsilon :: (\epsilon^* : \langle\,\rangle))))$

$= \mathcal{SE}^*[\![\mathtt{S^*}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, \mathcal{SE}[\![\mathtt{S}]\!](\mu \circ \gamma)(\lambda \epsilon^* \,.\, \mathit{tail\text{-}apply}\,\epsilon\,\epsilon^*))$

$= \mathcal{T}[\![\mathtt{(S\ S^*)}]\!](\mu \circ \gamma)$

Induction Hypothesis 12

$(\mathcal{CE}[\![\mathtt{PGM}]\!]\gamma)\mu\langle\,\rangle = \mathcal{E}[\![\mathtt{PGM}]\!](\mu \circ \gamma)$

The induction hypothesis states that running a program compiled using symbol table γ in runtime environment μ with an empty runtime stack is the same as the semantics using composition of μ and γ for an environment.

$(\mathcal{CE}[\![\texttt{(let* (GSD) (letrec (LPD) T))}]\!]\gamma)\mu\langle\,\rangle$

$= (\mathcal{CSDG}[\![\texttt{GSD}]\!]\gamma$
$\quad(\mathcal{CRDP}[\![\texttt{LPD}]\!](\mathit{extends}^*_{c\,g}\;\gamma\;\mathcal{GG}[\![\texttt{GSD}]\!])$
$\quad\quad(\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\;(\mathit{extends}^*_{c\,g}\;\gamma\;\mathcal{GG}[\![\texttt{GSD}]\!])\;\mathcal{GP}[\![\texttt{LPD}]\!]))))\mu\langle\,\rangle$

$= \mathcal{SDG}[\![\texttt{GSD}]\!](\mu\circ\gamma)$
$\quad(\lambda\alpha^*\,.\,(\mathcal{CRDP}[\![\texttt{LPD}]\!](\mathit{extends}^*_{c\,g}\;\gamma\;\mathcal{GG}[\![\texttt{GSD}]\!])$
$\quad\quad(\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\;(\mathit{extends}^*_{c\,g}\;\gamma\;\mathcal{GG}[\![\texttt{GSD}]\!])\;\mathcal{GP}[\![\texttt{LPD}]\!])))(\mathit{extends}^*_{r\,g}\;\mu\;\alpha^*)\langle\,\rangle)$

$= \mathcal{SDG}[\![\texttt{GSD}]\!](\mu\circ\gamma)$
$\quad(\lambda\alpha^*\,.\,\mathcal{RDP}[\![\texttt{LPD}]\!](\mu\circ\gamma)[(\mathit{map}\;(\mathrm{in}\,D)\;\alpha^*)/\mathcal{GG}[\![\texttt{GSD}]\!]^*)$
$\quad\quad(\lambda\epsilon^*\,.\,(\mathcal{CT}[\![\texttt{T}]\!](\mathit{extends}_{c\,l}\;(\mathit{extends}^*_{c\,g}\;\gamma\;\mathcal{GG}[\![\texttt{GSD}]\!])\;\mathcal{GP}[\![\texttt{LPD}]\!]))$
$\quad\quad\quad(\mathit{extends}_{r\,l}\;(\mathit{extends}^*_{r\,g}\;\mu\;\alpha^*)\;\epsilon^*)\langle\,\rangle))$

$= \mathcal{SDG}[\![\texttt{GSD}]\!](\mu\circ\gamma)$
$\quad(\lambda\alpha^*\,.\,\mathcal{RDP}[\![\texttt{LPD}]\!](\mu\circ\gamma)[(\mathit{map}\;(\mathrm{in}\,D)\;\alpha^*)/\mathcal{GG}[\![\texttt{GSD}]\!]^*)$
$\quad\quad(\lambda\epsilon^*\,.\,\mathcal{T}[\![\texttt{T}]\!]((\mu\circ\gamma)[(\mathit{map}\;(\mathrm{in}\,D)\;\alpha^*)/\mathcal{GG}[\![\texttt{GSD}]\!]]^*[(\mathit{map}\;(\mathrm{in}\,D)\;\epsilon^*)/\mathcal{GP}[\![\texttt{LPD}]\!]]))$

$= \mathcal{E}[\![\texttt{(let* (GSD) (letrec (LPD) T))}]\!](\mu\circ\gamma)$

11 The Assembler

A method for translating PurePreScheme bytecode into assembly language for a 32 bit microprocessor is now described. This methodology has been used to generate code for the Motorola MC68020 and is being used to generate code for the Motorola MC88100 risc chip.

11.1 Representation of the Abstract Machine

The abstract machine uses storage for global procedures and a small stack for temporaries inside an expression. For simplicity, we have chosen to represent these using a single small stack. Note that the size of this stack is linear in the size of the input program, and therefore all stack references could have been statically allocated. This is not the usual run-time recursion stack, which may grow unboundedly depending on the input to the program.

The initial environment, containing the globals and procedure declarations, is pushed onto the stack at program startup and remains the same throughout. The runtime environment environment, π, is built on top of the initial environment, and the runtime stack, ζ, is built on top of μ. Three global registers are used in this scheme. The first is a pointer to the initial environment, the second is a pointer to top of the runtime environment μ, and the third is a pointer to the top of the runtime stack ζ. Some bytecodes also use additional temporary registers for some intermediate values but no assumptions are about the values in these registers.

11.2 The Initial Environment

11.2.1 Globals

Globals are just represented as tagged data which is pushed onto the stack at program instantiation.

11.2.2 Procedures

Procedures are represented by a code pointer, an argument count, and an offset from the initial environment of telling where the procedure representation exists on the stack. This offset is tagged and used as the runtime representation of the procedure.

Tagged Offset
Argument Count
Code Pointer

11.3 Representation Of Data

For simplicity, all data is represented by 4 bytes. The current semantics of PurePreScheme necessitates runtime type tags. Tags are kept in the lower order bits:

1.	Tag B = 1	byte pointers
2.	Tag N = 00	small integers
3.	Tag T = 0010	truth values
4.	Tag H = 0110	characters
5.	Tag F = 1010	global procedures

There are several advantages to this scheme. First, byte pointers only need one bit for the tag field, allowing the other 31 bits for address space. Second, numeric operations can be done with little or no tag coding/decoding. Third, as all tag bits are in the low order bits, most operations on them can be done quickly. The actual representation of data with these tags given by:

1. Representation B = B * 2 + 1
2. Representation N = N * 4
3. Representation T = true = 18, T = false = 2
4. Representation H = ascii(H) * 16 + 6
5. Representation F = (offset from top of initial env) * 16 + 10

11.4 Assembling Bytecode

Shown here is each bytecode and the corresponding translation into assembly. For basic functions are necessary for this translation. The first, generate labels, is assumed to generate labels in some sequential manner. The second, emit-i, emits assembly instructions. Similarly, the third, emit-l, emits labels, and finally, the assembler can make recursive calls to itself. Two auxiliary functions, assemble-proc-code and assemble-proc-rep, are also used for assembling the top level procedures and placing their representation in the initial environment.

The generic assembly language instructions used here are straightforward. The stack is assumed to grow in a negative fashion, i.e. the stack pointer is decremented when something is pushed on the stack. The three global registers for the initial environment, runtime environment, and local stack are referred to as μ_0, μ, and ζ respectively. Additional registers are refered to as r_0 ... r_N. Register values can be accessed directly by naming the register, r_0, or can be dereferenced as pointer using the @ operator, r_0@(4). Constant decimal values are prefixed by a # such as #18.

11.4.1 Assemble

assemble((constant ϵ π))

```
    ;; Push value on the stack.
    emit-i(push $\epsilon$,$\zeta$)
    assemble($\pi$)
```

assemble((fetch-local ι π**))**

```
    ;; Find the local in the environment $\mu$
    ;; and push its value on the stack.
    emit-i(move $\mu$@($\iota$),$r_0$)
    emit-i(push $r_0$,$\zeta$)
    assemble($\pi$)
```

assemble((fetch-global ι π**))**

```
    ;; Find the global in the environment $\mu$
    ;; and push its value on the stack.
    emit-i(move $\mu$@($\iota$),$r_0$)
    emit-i(push $r_0$,$\zeta$)
    assemble($\pi$)
```

assemble((brf π_0 π_1**))**

```
    generate labels $l_0$, $l_1$
    emit-i(pop $\zeta$,$r_0$)
    ;; Test against false.
    emit-i(cmp #18,$r_1$)
    emit-i(jeq $l_0$)
    ;; Generate code for the true branch.
    assemble($\pi_0$)
    emit-i(goto $l_1$)
    ;; Label & generate code for the false branch.
    emit-l($l_0$)
    assemble($\pi_1$)
    emit-l($l_1$)
```

assemble((pick π^***))**

```
    generate $\#\pi^* + 1$ labels - wlog assume $l_n$ is first.
    emit-i(pop $\zeta$,$r_0$)
    check-numeric($r_0$)
    ;; Pick a label to transfer program control.
    emit-i(computed-goto $r_0$,labels)
    ;; Label and generate code for 0th option.
    emit-l($l_n$)
    assemble($\pi^*$.0)
    emit-i(jmp $l_{\#\pi^*}$)
              ⋮
    ;; Label and generate code the last option.
    emit-l($l_{n+\#\pi^*-1}$)
    assemble($\pi^*$.($\#\pi^*$-1))
    emit-l($l_{n+\#\pi^*}$)
```

assemble((update-store ι π**))**

```
    ;; Get value and move to global in environment.
    emit-i(move $\zeta$@(0),$\mu$@($\iota$))
    assemble($\pi$)
```

assemble((prim-apply ν υ π**))**

```
    ;; Generate code for primitive.
    inline-primitive($\nu$,$\upsilon$)
    assemble($\pi$)
```

assemble((update-store/ignore ι π**))**

```
    ;; Get value and move to global in environment,
    ;; pop off stack so value is not returned.
    emit-i(pop $\zeta$,$r_0$)
    emit-i(move $r_0$,$\mu$@($\iota$))
    assemble($\pi$)
```

assemble((prim-apply/ignore ν υ π**))**

```
    ;; Generate code for primitive such that
    ;; value is not returned.
    inline-primitive/ignore($\nu$,$\upsilon$)
    assemble($\pi$)
```

assemble((halt))

```
    emit-i(jmp exit)
```

assemble((add-to-env π))

```
    ;; Add stuff in $\zeta$ to $\mu$.
    emit-i(move $\zeta$,$\mu$)
    assemble($\pi$)
```

assemble((add-to-env* π))

```
    ;; Add one item in $\zeta$ to $\mu$.
    emit-i(sub #4,$\mu$)
    assemble($\pi$)
```

assemble((add-global-to-env* π))

```
    ;; Add one item in $\zeta$ to $\mu$.
    emit-i(sub #4,$\mu$)
    assemble($\pi$)
```

assemble((tail-call))

```
    generate labels $l_0$ & $l_1$
    emit-i(move $\mu_0$,$r_3$)
    emit-i(pop $\zeta$,$r_0$)
    check-function($r_0$)
    ;; Get rid of type flag.
    emit-i(shr #4,$r_0$)
    ;; Find function in init env.
    emit-i(add $r_0$,$r_3$)
    emit-i(move $\zeta$,$r_4$)
    ;; Get # of args.
    emit-i(move $r_3$@(4),$r_0$)
    ;; Check # of args.
    emit-i(shl #2,$r_0$)
    emit-i(move $\mu$,$r_1$)
    emit-i(move $\zeta$,$r_2$)
    emit-i(sub $r_2$,$r_1$)
    emit-i(cmp $r_1$,$r_0$)
    error(tail-call-error1)
    emit-l($l_0$)
    ;; Reset initial env for call.
    emit-i(move $\mu_0$,$\mu$)
    emit-i(move $\mu_0$,$\zeta$)
    emit-i(add $r_0$,$r_4$)
    ;; Get code address.
    emit-i(move $r_3$@(8),$r_3$)
    emit-i(sub $r_0$,$\zeta$)
    ;; Copy args and go.
    emit-l($l_1$)
    emit-i(cmp #0,$r_0$)
    emit-i(jeq $r_3$@(0))
    emit-i(move $r_4$@(0),$\mu$@)
    emit-i(sub #4,$\mu$)
    emit-i(sub #4,$r_4$)
    emit-i(sub #4,$r_0$)
    emit-i(jmp $l_1$)
```

assemble((closerecs ω π))

```
    generate #$\omega$+1 labels - wlog assume $l_n$ is first.
    ;; jump over generated code for procs.
    emit-i(jmp $l_{n+\#\omega}$)
    Generate code for procs.
    assemble-proc-code($\omega$ $l_n$ $l_{n+\#\omega-1}$)
    emit-l($l_{n+\#\omega}$)
    emit-i(push $l_n$,$\zeta$)
    ;; Push procedure representations on stack.
    assemble-proc-rep($\omega$ $l_n$ $l_{n+\#\omega-1}$)
    ;; Set up initial environment.
    emit-i(move $\zeta$,$\mu_0$)
    emit-i(move $\zeta$,$\mu$)
    assemble($\pi$)
```

11.4.2 Assemble-proc-code

```
**assemble-proc-code((empty-openers) $l_n$ $l_m$)**
    ;; Do nothing.

**assemble-proc-code((openers $\nu$ $\pi$ $\omega$) $l_n$ $l_m$)**
    ;; Emit label for proc to be assembled.
    emit-l($l_n$)
    ;; Assemble proc.
    assemble($\pi$)
    ;; Assemble remaining procedures.
    assemble-proc-code($\omega$ $l_{n+1}$ $l_m$)
```

11.4.3 Assemble-proc-rep

```
**assemble-proc-rep((empty-openers) $l_n$ $l_m$)**
    ;; Do nothing.

**assemble-proc-rep((openers $\nu$ $\pi$ $\omega$) $l_n$ $l_m$)**
    ;; Push code address.
    emit-i(push $l_n$,$\zeta$)
    ;; Push arg count.
    emit-i(push $\nu$,$\zeta$)
    ;; Push offset from top.
    emit-i(push m-n,$\zeta$)
    assemble-proc-rep($\omega$ $l_{n+1}$ $l_m$)
```

References

[1] Cieselski, B., and Wand, M. “Using Isabelle to Prove the Correctness of a Compiler,” in preparation.

[2] Clinger, W. "The Scheme 311 Compiler: An Exercise in Denotational Semantics," *Conf. Rec. 1984 ACM Symposium on Lisp and Functional Programming* (August, 1984), 356–364.

[3] Hannan, J. "Making Abstract Machines Less Abstract," *Proc. ACM Symp. on Functional Programming, Languages, and Architecture* (1991), to appear.

[4] McCarthy, J. "Towards a Mathematical Science of Computation," *Information Processing 62* (Popplewell, ed.) Amsterdam:North Holland, 1962, 21–28.

[5] Montneyohl, M., and Wand, M. "Incorporating Static Analysis in a Semantics-Based Compiler," *Information and Computation 82* (1989) 151–184.

[6] Oliva, Dino P., and Wand, M. "The Semantic Specification of Scheme via SPS," NU CCS Technical Report, to appear.

[7] Rees, J., and Clinger, W., eds. "Revised^3.99 Report on the Algorithmic Language Scheme," electronic manuscript on `altdorf.ai.mit.edu`.

[8] Wand, M. "Semantics-Directed Machine Architecture" *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.* (1982), 234–241.

[9] Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems 4*, 3 (July, 1982) 496–517.

[10] Wand, M. "Loops in Combinator-Based Compilers," *Info. and Control 57*,2–3 (May/June, 1983), 148–164.

[11] Wand, M. "A Semantic Prototyping System," *Proc. ACM SIGPLAN '84 Compiler Construction Conference* (1984) 213–221.

[12] Wand, M., and Wang, Z.-Y. "Conditional Lambda-Theories and the Verification of Static Properties of Programs," *Proc. 5th IEEE Symposium on Logic in Computer Science* (1990), 321–332