A Verified Run-Time Structure for Pure PreScheme*

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

September 18, 1992

Abstract

This document gives a summary of activities under MITRE Corporation Contract Number F19628-89-C-0001. It gives an operational semantics of an abstract machine for Pure PreScheme and of its implementation as a run-time structure on an Motorola 68000 microprocessor. The relationship between these two models is stated formally and proved.

1 Introduction

The goal of this project was to develop a verified compiler for the Vlisp PreScheme programming language[1]. 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.

In this compiler, PreScheme is first translated by a series of source-to-source transformations [9] into a kernel language called Pure PreScheme. In [8], we

*This report was sponsored by The MITRE Corporation, and is the Final Report for MITRE Contract Number F19628-89-C-001. The final preparation of the report was also supported in part by NSF and DARPA under NSF grants CCR-9002253 and CCR-9014603.

[1] Vlisp PreScheme is a descendent of the PreScheme programming language developed by Kelsey and Rees for the Scheme 48 run-time system[5]. While the two languages share many properties, they have developed independently since mid-1990.

presented a denotational semantics for Pure PreScheme, gave a compiler from Pure PreScheme to a byte-code like abstract machine, and proved the correctness of this compiler. We also presented a preliminary implementation of the compiler.

In the course of developing the compiler, it became clear that the most important issue was choosing a representation for the various quantities manipulated by the abstract machine. The abstract machine was essentially a tree-manipulating system, but the manipulations it performed were highly restricted, so that efficient representations were possible.

During 1991–92, we developed an attractive methodology, based on the concept of *storage layout relations*, for formalizing these representation decisions. A storage layout relation is an inductively-defined relation between states of the concrete and abstract machines. This idea was introduced by Hannan [4] in the case where the portion of the state that was represented in this way was static. We extended the techniques of [4] to deal with dynamic state, as this is by far the more common (and interesting) case. Some preliminary work on this methodology was presented in [12]. In this report, we apply this methodology to the Pure PreScheme compiler.

This report is organized as follows: We first present the abstract machine for Pure PreScheme and its operational semantics. The abstract machine manipulates combinator terms that are essentially trees. (For details on the source language, see [8].) In the concrete machine, the abstract structures of the abstract machine are represented in a linear memory model. In Section 3, we outline this architecture and present a formalization of the relationship between the abstract and concrete machine states. In Section 4 we prove some basic properties of these relations. In Section 5, we present the operational semantics of the concrete machine, and in Section 6, we state the connection between the concrete machine and the source language, and present the proof in detail. Finally, in Section 7 we discuss our implementation, and in Section 8 we present some conclusions.

We use the term "run-time structure" to suggest the package of representation decisions that allow an abstract machine to be implemented on a given target machine: how the resources of the abstract machine are laid out and represented in memory. Thus our proof is a verification of the correctness of this design. We do not prove the correctness of the full implementation, as that would require a semantics of the target machine instruction set, a much more difficult task. However, in practice this seems not to be so difficult, as the operations of our concrete machine can be easily implemented once the representation is chosen. Nor do we deal with the issues concerning the representation of tree-like abstract-machine code in a linear program store; some of the issues related to this are discussed in Section 3.2.5.

2 The Abstract Machine

The compiler for PurePreScheme produces code for an abstract machine consisting of a runtime environment, u, and a stack of stackable values, ζ, and a heap h.

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 $\pi u \zeta h$ reduces to a term $\pi' u' \zeta' h'$ 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: $\langle \pi, u, \zeta, h \rangle \Longrightarrow \langle \pi', u', \zeta', h' \rangle$.

The bytecodes π are given by the grammar:

$$
\begin{array}{lll}
\pi & ::= & (\textit{constant}\ \epsilon\pi) \\
& | & (\textit{fetch-local}\ \iota\pi) \\
& | & (\textit{fetch-global}\ \iota\pi) \\
& | & (\textit{goto}\ \pi) \\
& | & (\textit{label}\ \pi) \\
& | & (\textit{jmp}\ \nu\pi_1 \ldots \pi_\nu) \\
& | & (\textit{save-env}\ \pi) \\
& | & (\textit{restore-env}\ \pi) \\
& | & (\textit{update-store}\ \iota\pi) \\
& | & (\textit{prim-apply}\ \nu\upsilon\pi) \\
& | & (\textit{restore-env/ignore}\ \pi) \\
& | & (\textit{update-store/ignore}\ \iota\pi) \\
& | & (\textit{prim-apply/ignore}\ \nu\upsilon\pi) \\
& | & (\textit{brf}\ \pi_1\pi_2) \\
& | & (\textit{add-to-env}\ \nu\pi) \\
& | & (\textit{add-to-env}^*\ \pi) \\
& | & (\textit{tail-call}\ \nu) \\
& | & (\textit{halt}) \\
& | & (\textit{add-global-to-env}^*\ \pi) \\
& | & (\textit{closerecs}\,(\pi_1 \ldots \pi_n)\,\pi)
\end{array}
$$

The runtime display u is described by the following grammar:

$$
u \quad ::= \quad \textit{emptydisplay} \mid (\textit{extends}_{r\ l}(v_1 \ldots v_n)u') \mid (\textit{extends}_{r\ g}\ \alpha u')
$$

The stack consists of a list of stackable values given by the following grammar:

$$
\begin{array}{rcl}
v & ::= & \langle \mathbf{proc}, \langle \pi, u_0 \rangle \rangle \\
 & | & \langle \mathbf{env}, u \rangle \\
 & | & \langle \mathbf{int}, n \rangle \\
 & | & \langle \mathbf{bool}, b \rangle \\
 & | & \langle \mathbf{char}, c \rangle \\
 & | & \langle \mathbf{string}, s \rangle \\
 & | & \langle \mathbf{hptr}, n \rangle \\
 & | & \langle \mathbf{quote}, d \rangle
\end{array}
$$

These values are procedures (closed in a global environment u_0, to be described later), environments, and other values, which we call *immediate* values. These are integers, booleans, characters, strings, pointers into the heap (L-values), and quotations. L-values are tagged integers. Quotations represent data returned by primitives for use only with other primitives (primitives like *make-vector*, *vector-ref*, etc).

The heap for the abstract machine consists of three components: a map from heap pointers (as above) to immediate values, an integer (representing a free-location counter), and an unspecified third element. The first two components are used for ordinary mutable variables. The third component can be manipulated only by primitives (primitives like *make-vector*, *vector-ref*, etc); it plays a role analogous to that of the file system in a conventional language semantics. We use the notation $h.1$, $h.2$, and $h.3$ for the three components.

A program consists of a preamble, consisting of a sequence of *add-global-to-env** instructions followed by a *closerecs* instruction, followed by a bytecode term π that does not contain any *add-global-to-env** or *closerecs* instructions. The purpose of this preamble is to establish the invariants necessary for the proper operation of the rest of the machine. In this report we will be primarily concerned with the operation of the main program.

The operational semantics of each machine instruction is shown in Figure 1. If the machine state does not match any of the left-hand sides in Figure 1, then the machine goes into an error state and halts, returning the value $\langle \mathbf{error} \rangle$. Similarly, if `deref`$'$ is given a second argument that is not of the form $\langle \mathbf{hptr}, l \rangle$, the machine halts, returning $\langle \mathbf{error} \rangle$. In Figure 1, we have used h' as the variable ranging over heaps because this is the variable that will be used in our proofs, in which abstract-machine quantities typically appear on the right-hand sides of correspondence assertions, as in Section 3.

The machine halts normally by executing a *halt* instruction, returning the value $\langle \mathbf{ok}, v \rangle$ for some value v.

Abstract machine:

$$
\begin{array}{rcl}
\langle(\mathit{constant}\ \epsilon\pi), u, \zeta, h'\rangle & \Longrightarrow & \langle \pi, u, (\epsilon :: \zeta), h'\rangle \\
\langle(\mathit{fetch\text{-}local}\ \iota\pi), u, \zeta, h'\rangle & \Longrightarrow & \langle \pi, u, (u(\iota) :: \zeta), h'\rangle \\
\langle(\mathit{fetch\text{-}global}\ \iota\pi), u, \zeta, h'\rangle & \Longrightarrow & \langle \pi, u, ((\mathtt{deref}'\ h'\, u(\iota)) :: \zeta), h'\rangle \\
\langle(\mathit{goto}\ \pi), u, \zeta, h'\rangle & \Longrightarrow & \langle \pi, u, \zeta, h'\rangle \\
\langle(\mathit{label}\ \pi), u, \zeta, h'\rangle & \Longrightarrow & \langle \pi, u, \zeta, h'\rangle \\
\langle(\mathit{jmp}\ \nu\pi_1 \ldots \pi_\nu), u, \langle \mathbf{int}, n\rangle :: \zeta), h'\rangle & \Longrightarrow & \langle(\mathtt{list\text{-}ref}\ n\,(\pi_1 \ldots \pi_n u)), u, \zeta, h'\rangle \\
\langle(\mathit{save\text{-}env}\ \pi), u, \zeta, h'\rangle & \Longrightarrow & \langle \pi, u, (\langle \mathbf{env}, u\rangle :: \zeta), h'\rangle \\
\langle(\mathit{restore\text{-}env}\ \pi), u, (v :: \langle \mathbf{env}, u'\rangle :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, u', (v :: \zeta), h'\rangle \\
\langle(\mathit{update\text{-}store}\ \iota\pi), u, (v :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, u, (v :: \zeta), (\mathtt{update}\ u(i)\, v\, h')\rangle \\
\langle(\mathit{prim\text{-}apply}\, n\upsilon\pi), u, (v_1 :: \ldots :: v_n :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, u, (w_1 :: \zeta), h'_1\rangle \\
\multicolumn{3}{c}{\text{where } (w_1, h'_1) = (\mathtt{apply\text{-}prim}'\ \upsilon(v_1 \ldots v_n)h')} \\
\langle(\mathit{restore\text{-}env/ignore}\ \pi), u, (\langle \mathbf{env}, u'\rangle :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, u', \zeta, h'\rangle \\
\langle(\mathit{update\text{-}store/ignore}\ \iota\pi), u, (v :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, u, \zeta, (\mathtt{update}\ u(i)\, v\, h')\rangle \\
\langle(\mathit{prim\text{-}apply/ignore}\ \nu\upsilon\pi), u, \zeta_1, h'\rangle & \Longrightarrow & \langle \pi, u, \zeta, h'_1\rangle \\
\multicolumn{3}{c}{\text{where } \zeta_1 = (v_1 :: \ldots :: v_\nu :: \zeta)} \\
\multicolumn{3}{c}{\text{and } (w_1, h'_1) = (\mathtt{apply\text{-}prim}'\ \upsilon(v_1 \ldots v_n)h')} \\
\langle(\mathit{brf}\ \pi_1\pi_2), u, (\langle \mathbf{bool}, b\rangle :: \zeta), h'\rangle & \Longrightarrow & \langle(\mathtt{choose}\ b\,\pi_1\pi_2), u, \zeta, h'\rangle \\
\langle(\mathit{add\text{-}to\text{-}env}\ \nu\pi), u, (v_1 :: \ldots :: v_\nu :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, (\mathit{extends}_{r\,l}\ u\,(v_1 \ldots v_\nu)), \zeta, h'\rangle \\
\langle(\mathit{add\text{-}to\text{-}env}^*\ \pi), u, (v :: \zeta), h'\rangle & \Longrightarrow & \langle \pi, (\mathit{extends}_{r\,l}\ u\,(v)), \zeta, h'\rangle \\
\langle(\mathit{tail\text{-}call}\ \nu), u, \zeta_1, h'\rangle & \Longrightarrow & \langle \pi, (\mathit{extends}_{r\,l}\ u_0\,(v_1 \ldots v_\nu)), \langle\rangle, h'\rangle \\
\multicolumn{3}{c}{\text{where } \zeta_1 = (\langle \mathbf{proc}, \langle \pi, u_0\rangle\rangle :: v_1 :: \ldots :: v_\nu :: \zeta)} \\
\langle(\mathit{halt}), u, (v :: \zeta), h'\rangle & \Longrightarrow & \langle \mathbf{ok}, v\rangle \\
\langle(\mathit{add\text{-}global\text{-}to\text{-}env}^*\ \pi), u, (v :: \zeta), h'\rangle & \Longrightarrow & \\
\multicolumn{3}{c}{\langle \pi, (\mathit{extends}_{r\,g}\ u\,(\mathtt{new}\ h')), \zeta, (\mathtt{update}(\mathtt{new}\ h')\, v\, h')\rangle} \\
\langle(\mathit{closerecs}\,(\pi_1 \ldots \pi_n)\,\pi), u, \zeta), h'\rangle & \Longrightarrow & \\
\multicolumn{3}{c}{\langle \pi, (\mathtt{fix}(\lambda u' \cdot (\mathit{extends}_{r\,l}\ u\,(\langle \mathbf{proc}, \langle \pi_1, u'\rangle\rangle \ldots \langle \mathbf{proc}, \langle \pi_n, u'\rangle\rangle))))), \zeta, h'\rangle}
\end{array}
$$

Auxiliaries:

$$
\begin{array}{c}
(\mathtt{deref}'\ h\,\langle \mathbf{hptr}, l\rangle) = (h.1)(\langle \mathbf{hptr}, l\rangle) \\
(\mathtt{list\text{-}ref}\ \nu(\pi\pi^*)) = ((\nu = 0) \rightarrow \pi, (\mathtt{list\text{-}ref}\,(\nu - 1)\pi^*)) \\
(\mathtt{choose}\ b\,\pi\,\pi') = (b \rightarrow \pi, \pi')
\end{array}
$$

Figure 1: Operational Semantics of the Abstract Machine

3 Storage Layout

The concrete machine uses a *store* to represent the environment and stack of the abstract machine. It also has a heap which corresponds to the heap of the abstract machine. The concrete heap is like the abstract heap except that its L-values are *untagged* integers, and its first component contains immediate values in their concrete representation.

The concrete machine uses two pointers into the store, *sp* and *up*, to represent the stack and the environment; the way in which these pointers accomplish this task is the major subject of this section. Thus the state of the concrete machine is of the form $\langle \pi, up, sp, \sigma, h \rangle$.

In this section the representation of the abstract machine by the concrete machine is formalized. To aid in the intuition behind these formalisms, we will first give an informal sketch of the representation. We will then present the formal definitions.

The key to the representation is that any PreScheme program uses only a bounded amount of space. By analyzing the program, we may define two constants for that program. The first, N, is the total amount of environment space required by the program. The second, N_0, is the amount of space initially required by the environment for globals and procedure declarations. These constants are easily calculated (see Theorem 3) and are used for setting the initial parameters of the concrete machine.

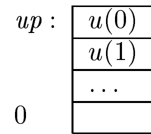
3.1 Informal Presentation

We can take advantage of this by using locations 0 through N of the concrete machine's store for the environment u, and the locations above N for the local stack ζ.

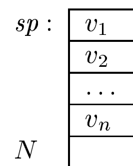
We can describe the representation pictorially as follows:

1. Stackable values (integers, booleans, characters, strings, procedures, environments). For efficiency purposes, runtime tags are not used. This restricts our correctness result but only for those programs that result in an error. All quantities are represented by a single machine word (i.e. one space in σ). Strings and procedures are represented by pointers into a static space; since all procedures in a PreScheme share the same environment, there is no need for a separate environment pointer in a procedure object. Environments need to be stacked, but they are not expressible; when stacked they are represented as pointers to the environment representation. As all other data is either immediate or a pointer to static space, these environment pointers are the only real "pointers" in the system (that is, they are the only references that can potentially dangle).

2. Runtime environment u. As all values are represented by 1 word in memory, this suggests that u be represented by locations 0 through up in σ:

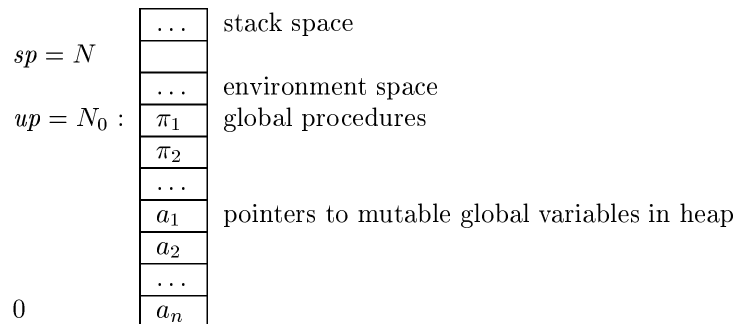


3. The local stack z. Since all values are represented by a single word in σ and that u ranges between 0 and N, a local stack $\zeta = (v_1 :: v_2 :: \ldots :: v_n :: \langle\rangle)$ can be represented using σ as a stack growing upward from position $N+1$:



In the initial startup of the machine, up is started at -1 and the stack pointer sp starts at N, so that both the environment and stack are empty. The prelude to the program allocates globals with a sequence of *add-global-to-env* instructions, followed by a *closerecs* instruction. The effect of these instructions is to create an environment u_0 in which all the global procedures are closed. At this point up has value N_0. This is the ground configuration which is used as a reference for all tail calls.

After the globals and procedures have been allocated, up will have value N_0 and will not drop below this value for the remainder of program execution. This configuration is the 'ground configuration' as it is the configuration restored at tail call:



3.2 Formal Presentation

To formalize this representation, we will have one storage layout relation for each kind of data manipulated by the abstract and concrete machines. These are: immediate data, stackable values, stacks, environments, heaps, and programs. These will be defined by simultaneous induction. The definition of the relations is a relatively straightforward transcription of the data in the diagrams above; the primary difficulty is in guarding against off-by-one errors.

3.2.1 Relating Immediate Data

$$\begin{array}{ll} \multicolumn{2}{l}{\epsilon \simeq \epsilon' \iff} \\ \text{either} & \epsilon' = \langle \mathbf{int}, n \rangle \text{ and } \epsilon = n \\ \text{or} & \epsilon' = \langle \mathbf{bool}, b \rangle \text{ and } \epsilon = b \\ \text{or} & \epsilon' = \langle \mathbf{char}, c \rangle \text{ and } \epsilon = c \\ \text{or} & \epsilon' = \langle \mathbf{string}, s \rangle \text{ and } \epsilon = s \\ \text{or} & \epsilon' = \langle \mathbf{hptr}, l \rangle \text{ and } \epsilon = l \\ \text{or} & \epsilon' = \langle \mathbf{quote}, d \rangle \text{ and } \epsilon = d \end{array}$$

3.2.2 Relating Stackable Values

$$\begin{array}{ll} \multicolumn{2}{l}{(\sigma, b) \models_V c \simeq v \iff} \\ \text{either} & v = \langle \mathbf{proc}, \langle \pi, u_0 \rangle \rangle \text{ and } c = \pi \\ \text{or} & v = \langle \mathbf{env}, u \rangle \text{ and } c \leq b \\ & \text{and } (\sigma, b) \models_U c \simeq u \\ \text{or} & c \simeq v \end{array}$$

Here u_0 denotes the environment at the end of the prelude; all procedures in a PreScheme program are closed in this environment, and are therefore mutually recursive.

The parameter b marks the upper boundary of environment space; the condition on environments ensures that no environment pointer is ever dangling. This parameter will almost always be *up*.

The last line refers to c and v as immediate data, as above.

3.2.3 Relating Pointers and Stacks

$$\begin{array}{ll} \multicolumn{2}{l}{(\sigma, b) \models_Z \langle p, p' \rangle \simeq \zeta \iff} \\ \text{either} & \zeta = \langle\rangle \text{ and } p = p' \\ \text{or} & \zeta = (z :: \zeta') \text{ and } p > p' \text{ and } (\sigma, b) \models_V \sigma(p) \simeq z \\ & \text{and } (\sigma, b) \models_Z \langle p-1, p' \rangle \simeq \zeta' \end{array}$$

Here the stack elements are in locations $p' + 1$ through p, so $p = p'$ marks an empty stack. In the proof, p' will always be N.

3.2.4 Relating Pointers and Environments

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\sigma, b) \models_U p \simeq u \iff} \\
\text{either} & u = \mathit{emptydisplay} \text{ and } p = -1 \\
\text{or} & u = (\mathit{extends}_{r\,l}(v_1 \ldots v_n)u') \text{ and } p \leq \mathit{up} \\
 & \text{and } (\sigma, b) \models_V \sigma(p) \simeq v_n \ldots (\sigma, b) \models_V \sigma(p - (n-1))) \simeq v_1 \\
 & \text{and } (\sigma, b) \models_U (p - n) \simeq u' \\
\text{or} & u = (\mathit{extends}_{r\,g}\, \alpha u') \text{ and } p \leq \mathit{up} \\
 & \text{and } \sigma(p) \simeq \alpha \\
 & \text{and } (\sigma, b) \models_U (p - 1) \simeq u'
\end{array}
$$

3.2.5 Relating Programs

Programs in the concrete and abstract machines differ only in the format of the literals embedded in them. The complete definition is shown in Figure 2.

We do not bother linearizing the code in the concrete machine. It would be relatively easy to extend the current proof to use a representation of the concrete-machine code in a linear store, along the lines of [12], since the program store is static. It would be a more substantial task to prove the correctness of the translator that converts from an abstract-machine program, which is tree-like in its structure, to a corresponding state of the program store. This program might be dubbed a linearizer or an assembler. Such an assembler-correctness proof would deal with the specifics of the translation process and would resemble in its specificity and level, the compiler-correctness proof of [8]. We have good ideas about how this proof should be structured, and we will report on this later, either in Oliva's forthcoming Ph.D. thesis or a separate technical report.

3.2.6 Relating Heaps

As discussed above, the concrete machine's heap also consists of three components. The first component maps untagged integers to immediate values in their concrete representation, the second component is an integer-valued free-storage counter, and the third component is the same as that for the abstract machine. Heaps correspond if their free-storage counters and third components agree, and if the allocated portions of their first components correspond:

$$
\begin{array}{l}
h_c \simeq h_a \iff \\
\quad h_a.2 = h_c.2 \\
\quad \text{and } (\forall i : 0 \leq i \leq h_c.2)(h_c.1(i) \simeq h_a.1(\langle \mathbf{hptr}, i \rangle)) \\
\quad \text{and } h_a.3 = h_c.3
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\pi_c \simeq \pi_a \iff} \\
& \text{either} & \pi_a = (\textit{constant}\ \epsilon' \pi_a') \text{ and } \pi_c = (\textit{constant}\ \epsilon \pi_c') \\
& & \text{and } \epsilon \simeq \epsilon' \text{ and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{fetch-local}\ \iota \pi_a') \text{ and } \pi_c = (\textit{fetch-local}\ \iota \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{fetch-global}\ \iota \pi_a') \text{ and } \pi_c = (\textit{fetch-global}\ \iota \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{goto}\, \pi_a') \text{ and } \pi_c = (\textit{goto}\, \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{label}\, \pi_a') \text{ and } \pi_c = (\textit{label}\, \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{jmp}\ \nu \pi_{a1} \ldots \pi_{a\nu}) \text{ and } \pi_c = (\textit{jmp}\ \nu \pi_{c1} \ldots \pi_{c\nu}) \\
& & \text{and } \pi_{c1} \simeq \pi_{a1} \ldots \text{ and } \pi_{c\nu} \simeq \pi_{a\nu} \\
& \text{or} & \pi_a = (\textit{save-env}\ \pi_a') \text{ and } \pi_c = (\textit{save-env}\ \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{restore-env}\ \pi_a') \text{ and } \pi_c = (\textit{restore-env}\ \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{update-store}\ \iota \pi_a') \text{ and } \pi_c = (\textit{update-store}\ \iota \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{prim-apply}\ \nu \upsilon \pi_a') \text{ and } \pi_c = (\textit{prim-apply}\ \nu \upsilon \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{restore-env/ignore}\ \pi_a') \text{ and } \pi_c = (\textit{restore-env/ignore}\ \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{update-store/ignore}\ \iota \pi_a') \text{ and } \pi_c = (\textit{update-store/ignore}\ \iota \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{prim-apply/ignore}\ \nu \upsilon \pi_a') \text{ and } \pi_c = (\textit{prim-apply/ignore}\ \nu \upsilon \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{brf}\, \pi_{a1} \pi_{a2}) \text{ and } \pi_c = (\textit{brf}\, \pi_{c1} \pi_{c2}) \\
& & \text{and } \pi_{c1} \simeq \pi_{a1} \text{ and } \pi_{c2} \simeq \pi_{a2} \\
& \text{or} & \pi_a = (\textit{add-to-env}\ \nu \pi_a') \text{ and } \pi_c = (\textit{add-to-env}\ \nu \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{add-to-env}^*\ \pi_a') \text{ and } \pi_c = (\textit{add-to-env}^*\ \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{tail-call}\ \nu) \text{ and } \pi_c = (\textit{tail-call}\ \nu) \\
& \text{or} & \pi_a = (\textit{halt}) \text{ and } \pi_c = (\textit{halt}) \\
& \text{or} & \pi_a = (\textit{add-global-to-env}^*\ \pi_a') \text{ and } \pi_c = (\textit{add-global-to-env}^*\ \pi_c') \\
& & \text{and } \pi_c' \simeq \pi_a' \\
& \text{or} & \pi_a = (\textit{closerecs}\, (\pi_{a1} \ldots \pi_{an})\, \pi_a') \text{ and } \pi_c = (\textit{closerecs}\, (\pi_{c1} \ldots \pi_{cn})\, \pi_c') \\
& & \text{and } \pi_{c1} \simeq \pi_{a1} \ldots \text{ and } \pi_{cn} \simeq \pi_{an} \text{ and } \pi_c' \simeq \pi_a'
\end{array}
$$

Figure 2: Correspondence of concrete and abstract machine code

3.3 Relating the Abstract and Concrete Machines

We can now define the correspondence between abstract and concrete machine states.

Restating the discussion of initialization at the end of Section 3.1, the effect of the prelude is to establish the invariants

$$N_0 \leq up \leq N \leq sp$$

and

$$(\sigma, u_0) \models_U N_0 \simeq u_0$$

The first invariant expresses the disjointness of the environment and stack spaces, and the second establishes the correct representation of the globals.

Definition 1 *We say a concrete machine state* $\langle \pi, up, sp, \sigma, h\rangle$ corresponds *to an abstract machine state* $\langle \pi', u, \zeta, h'\rangle$ *(written* $\langle \pi, up, sp, \sigma, h\rangle \simeq \langle \pi', u, \zeta, h'\rangle$*) if and only if the following conditions are satisfied:*

1. $\pi \simeq \pi'$,
2. $(\sigma, up) \models_U up \simeq u$,
3. $(\sigma, up) \models_Z (sp, N) \simeq \zeta$,
4. $h \simeq h'$
5. $N_0 \leq up \leq N \leq sp$, *and*
6. $(\sigma, u_0) \models_U N_0 \simeq u_0$

4 Basic Properties of the Representation

Before proceeding, we state several lemmas that express the basic properties of these relations. These will be used continually in the proof.

Lemma 1 (Immediate Values) *If* $c \simeq v$ *as immediate values, then for all* σ *and* b, $(\sigma, b) \models_V c \simeq v$.

Proof: Immediate from the definition of $\models_V$.

Definition 2 *We say* $\sigma =_{[p,p']} \sigma'$ *if* $p \leq p'$ *and* $(\forall x : p \leq x \leq p')(\sigma(x) = \sigma'(x))$.

Lemma 2 (Free-Storage Lemma)

1. *If* $(\sigma, b) \models_V c \simeq v$ *and* $\sigma =_{[0,b]} \sigma'$ *then* $(\sigma', b) \models_V c \simeq v$.

2. *If* $(\sigma, b) \models_Z \langle p, p' \rangle \simeq \zeta$ *and* $\sigma =_{[0,b]} \sigma'$ *and* $\sigma =_{[p'+1,p]} \sigma'$ *then* $(\sigma', b) \models_Z \langle p, p' \rangle \simeq \zeta$.

3. *If* $(\sigma, b) \models_U p \simeq u$ *and* $\sigma =_{[0,p]} \sigma'$ *then* $(\sigma', b) \models_U p \simeq u$.

Proof: By induction on the definitions of $\models_V, \models_Z$, and $\models_U$.

1. Given $(\sigma, b) \models_V c \simeq v$ and $\sigma =_{[0,b]} \sigma'$, there are three cases we must consider:
   (a) $v = \langle \mathbf{proc}, \langle \pi, u \rangle \rangle$ and $c = \pi$
   If this is true, it is true regardless of σ hence $(\sigma', b) \models_V c \simeq v$.
   (b) $v = \langle \mathbf{env}, u \rangle$ and $c \leq b$ and $(\sigma, b) \models_U c \simeq u$
   Given this and $\sigma =_{[0,b]} \sigma'$ then by induction we can conclude $(\sigma', b) \models_U c \simeq u$ and hence $(\sigma', b) \models_V c \simeq v$.
   (c) $\sigma(p) = v.val$
   If this is true, it is true regardless of σ hence $(\sigma', b) \models_V c \simeq v$.

2. Given $(\sigma, b) \models_Z \langle p, p' \rangle \simeq \zeta$ and $\sigma =_{[0,b]} \sigma'$ and $\sigma =_{[p'+1,p]} \sigma'$ there are two cases to consider:
   (a) $\zeta = \langle\rangle$ and $p = p'$
   If this is true, it is true regardless of σ hence $(\sigma', b) \models_Z \langle p, p' \rangle \simeq \zeta$.
   (b) $\zeta = (z :: \zeta')$ and $p > p'$ and $(\sigma, b) \models_V \sigma(p) \simeq z$ and $(\sigma, b) \models_Z \langle p-1, p' \rangle \simeq \zeta'$
      - Given $(\sigma, b) \models_V \sigma(p) \simeq z$ and $\sigma =_{[0,b]} \sigma'$ and $\sigma(p) = \sigma'(p)$ then by induction we can conclude $(\sigma', b) \models_V \sigma'(p) \simeq z$.
      - Given $(\sigma, b) \models_Z \langle p-1, p' \rangle \simeq \zeta'$ then by induction we can conclude $(\sigma', b) \models_Z \langle p-1, p' \rangle \simeq \zeta'$.

   hence $(\sigma', b) \models_Z \langle p, p' \rangle \simeq \zeta$.

3. Given $(\sigma, b) \models_U p \simeq u$ and $\sigma =_{[0,b]} \sigma'$ there are three cases to consider:
   (a) $u = \mathit{emptydisplay}$ and $p = -1$
   If this is true, it is true regardless of σ hence $(\sigma', b) \models_U p \simeq u$.
   (b) $u = (\mathit{extends}_{r\,l}(v_1 \ldots v_n)u')$ and $p \leq b$ and $(\sigma, b) \models_V \sigma(p) \simeq v_n \ldots (\sigma, up) \models_V b \simeq \sigma(p-(n-1)))v_1$ and $(\sigma, b) \models_U (p-n) \simeq u'$
      - Given $(\sigma, b) \models_V \sigma(p) \simeq v_n \ldots (\sigma, b) \models_V \sigma(p-(n-1)) \simeq v_1$, $p \leq b$ and $\sigma =_{[0,b]} \sigma'$ then by induction we can conclude $(\sigma', b) \models_V \sigma'(p) \simeq v_n \ldots (\sigma', b) \models_V \sigma'(p-(n-1)) \simeq v_1$.
      - Given $(\sigma, b) \models_U (p-n) \simeq u'$ and $(p-n) < p \leq b$ then by induction we can conclude $(\sigma', b) \models_U (p-n) \simeq u'$.

hence $(\sigma', b) \models_U p \simeq u$.

(c) $u = (\mathit{extends}_{r\ g}\ \alpha u')$ and $p \leq b$ and $\sigma(p) \simeq \alpha$ and $(\sigma, b) \models_U (p-1) \simeq u'$

- Given $\sigma(p) \simeq \alpha$, $p \leq b$ and $\sigma =_{[0,b]} \sigma'$ we can conclude $\sigma'(p) \simeq \alpha$.
- Given $(\sigma, b) \models_U (p-1) \simeq u'$ and $(p-1) < p \leq b$ then by induction we can conclude $(\sigma', b) \models_U (p-1) \simeq u'$.

hence $(\sigma', b) \models_U p \simeq u$.

□

Lemma 3 *If* $(\sigma, b) \models_V c \simeq v$ *and* $b \leq b'$ *then* $(\sigma, b') \models_V c \simeq v$.

Proof: The parameter b is only used in the case $(\sigma, b) \models_V c \simeq \langle \mathbf{env}, u \rangle$. In that case, $c \leq b \leq b'$, so $(\sigma, b') \models_V c \simeq \langle \mathbf{env}, u \rangle$. The rest of the proof follows by tedious induction.

□

Lemma 4 *If* $(\sigma, b) \models_U p \simeq u$ *then* $(\sigma, b) \models_V \sigma(p - \iota) \simeq u(\iota)$.

Proof: by induction on ι.

1. Given $\iota = 0$ and $(\sigma, b) \models_U p \simeq u$ there are three cases we must consider:
   - $u = \mathit{emptydisplay}$ and $p = -1$
     hence an error occurs and nothing is guaranteed.
   - $u = (\mathit{extends}_{r\ l}(v_1 \ldots v_n)u')$ and $(\sigma, b) \models_V \sigma(p) \simeq v_n \ldots (b, \sigma) \models_V \sigma(p - (n-1))) \simeq v_1$ and $(\sigma, b) \models_U (p - n) \simeq u'$ hence $u(0) = v_n$ and $(\sigma, b) \models_V \sigma(p - 0) \simeq u(0)$
   - $u = (\mathit{extends}_{r\ g}\ \alpha u')$ and $\sigma(p) \simeq \alpha$ and $(\sigma, b) \models_U (p-1) \simeq u'$ hence $u(0) = \alpha$ and $(\sigma, b) \models_V \sigma(p - 0) \simeq u(0)$
2. Assume induction for $0 \leq \iota < N$ and prove for N. Again there are three cases we must consider:
   - $u = \mathit{emptydisplay}$ and $p = -1$
     hence an error occurs and nothing is guaranteed.
   - $u = (\mathit{extends}_{r\ l}(v_1 \ldots v_n)u')$ and $(\sigma, b) \models_V \sigma(p) \simeq v_n \ldots (\sigma, b) \models_V \sigma(p - (n-1))) \simeq v_1$ and $(\sigma, \mathit{up}) \models_U (p - n) \simeq u'$ hence if $N \leq n$ then $u(N) = v_N$ and $(\sigma, b) \models_V \sigma(p - (N-1)) \simeq u(N)$ otherwise if $N > n$ then by induction we have $(\sigma, b) \models_V \sigma(p - (N - n)) \simeq u'(N - n)$.
   - $u = (\mathit{extends}_{r\ g}\ \alpha u')$ and $\sigma(p) \simeq \alpha$ and $(\sigma, b) \models_U (p-1) \simeq u'$ hence by induction we have $(\sigma, b) \models_V \sigma(p-1) \simeq u'(N-1)$.

□

5 The Concrete Machine

We define the operational semantics of the concrete machine by considering the representation of each possible abstract machine state, so that the concrete machine simulates the behavior of the abstract machine. That is, if $C_1 \simeq A_1$ and $A_1 \Longrightarrow A_2$, then we want the concrete machine to send C_1 to some state C_2 such that $C_2 \simeq A_2$.

In general, this will not be possible, since the concrete machine does not have tags to distinguish different data types. The abstract machine uses these tags to distinguish an error state. Since the concrete machine cannot mimic this behavior, we will require the concrete machine to behave correctly only if A_2 is not an error state. If A_2 is an error state, the behavior of the concrete machine is unspecified. Similarly, if A_1 is an abstract-machine state with some arbitrary program π, executing it may cause other errors, such as underflowing the stack or overflowing the environment. In general, we will only be able to perform this simulation when A_1 is a state arising from reducing a properly compiled Pure PreScheme program.

It seems possible to *derive* the behavior of the concrete machine from this specification, but we have no formal way of doing this at present. Instead, we developed the operational semantics of the concrete machine informally, using the specification for guidance, and proved its correctness *post hoc*. This proof will constitute the main part of this report.

The operational semantics of the concrete machine is shown in Figure 3. `apply-prim` is unspecified, but it must satisfy the constraint that when `apply-prim` and `apply-prim`′ are applied to congruent arguments, they return congruent immediate values and congruent heaps.

6 Main Theorem

As suggested previously, the concrete and abstract machines are related by a property like the following:

> *Let A_1 and A_2 be states of the abstract machine and C_1 and C_2 be states of the concrete machine. If $C_1 \simeq A_1$, $A_1 \Longrightarrow A_2$, and $C_1 \Longrightarrow C_2$, then $C_2 \simeq A_2$.*

Unfortunately, this is impossible, since arbitrary states A_1 of the abstract machine may break the concrete machine by making the environment grow too large, by putting environments in places where the values are required to be expressible, etc. However this result does hold where A_1 is a state that arises when the abstract machine is started with a correctly compiled Pure PreScheme program.

Concrete machine:

$$
\begin{array}{rcl}
\langle(\mathit{constant}\ \epsilon\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp+1, \sigma[\epsilon/\ sp+1], h\rangle \\
\langle(\mathit{fetch\text{-}local}\ \iota\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp+1, \sigma[(\texttt{lookup}\ \iota\ up\,\sigma)/sp+1], h\rangle \\
\langle(\mathit{fetch\text{-}global}\ \iota\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp+1, \sigma[(\texttt{deref}\ h\,(\texttt{lookup}\ \iota\ up\,\sigma))/sp+1], h\rangle \\
\langle(\mathit{goto}\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp, \sigma, h\rangle \\
\langle(\mathit{label}\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp, \sigma, h\rangle \\
\langle(\mathit{jmp}\ \nu\pi_1 \ldots \pi_\nu), up, sp, \sigma, h\rangle & \Longrightarrow & \langle(\texttt{list-ref}\ \sigma(sp)\,(\pi_1 \ldots \pi_\nu)), up, sp, \sigma, h\rangle \\
\langle(\mathit{save\text{-}env}\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp+1, \sigma[up\,/sp+1], h\rangle \\
\langle(\mathit{restore\text{-}env}\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, \sigma(sp-1), sp-1, \sigma[\sigma(sp)/sp-1], h\rangle \\
\langle(\mathit{update\text{-}store}\ \iota\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp, \sigma, (\texttt{update}\ \sigma(up-\iota)\,\sigma(sp)\,h)\rangle \\
\langle(\mathit{prim\text{-}apply}\,\nu\upsilon\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp-(\nu-1), \sigma[w_1/\ sp-(\nu-1)], h_1\rangle \\
 & & \text{where } (w_1, h_1) = (\texttt{apply-prim}\ \upsilon\,(\texttt{collect}\ \sigma\ sp\,\nu)\,h) \\
\langle(\mathit{restore\text{-}env/ignore}\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, \sigma(sp), sp-1, \sigma, h\rangle \\
\langle(\mathit{update\text{-}store/ignore}\ \iota\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp-1, \sigma, (\texttt{update}\ \sigma(up-\iota)\,\sigma(sp))\rangle \\
\langle(\mathit{prim\text{-}apply/ignore}\ \nu\upsilon\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up, sp-\nu, \sigma, h_1\rangle \\
 & & \text{where } (w_1, h_1) = (\texttt{apply-prim}\ \upsilon\,(\texttt{collect}\ \sigma\ sp\,\nu)\,h) \\
\langle(\mathit{brf}\ \pi_1\pi_2), up, sp, \sigma, h\rangle & \Longrightarrow & \langle(\texttt{choose}\ \sigma(sp)\,\pi_1\pi_2), up, sp-1, \sigma, h\rangle \\
\langle(\mathit{add\text{-}to\text{-}env}\ \nu\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, (up+\nu), (sp-\nu), (\texttt{copy}\ \sigma\nu(up+1)\ sp), h\rangle \\
\langle(\mathit{add\text{-}to\text{-}env}^*\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \\
 & & \langle\pi, up+1, sp-1, (\sigma[\sigma(sp)/\ up+1]), h\rangle \\
\langle(\mathit{tail\text{-}call}\ \nu), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\sigma(sp), (N_0+\nu), (N-1), \sigma_1, h\rangle \\
 & & \text{where } \sigma_1 = (\texttt{copy}\ \sigma\,\nu\,(N_0+1)\,(sp-1)) \\
\langle(\mathit{halt}), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\mathbf{ok}, \sigma(sp)\rangle \\
\langle(\mathit{add\text{-}global\text{-}to\text{-}env}^*\ \pi), up, sp, \sigma, h\rangle & \Longrightarrow & \\
 & & \langle\pi, up+1, sp-1, (\sigma[(\texttt{new}\ h)/(up+1)]), (\texttt{update}\,(\texttt{new}\ h)\,\sigma(sp)\,h)\rangle \\
\langle(\mathit{closerecs}\,(\pi_1 \ldots \pi_n)\,\pi), up, sp, \sigma, h\rangle & \Longrightarrow & \langle\pi, up+n, sp, \sigma_1, h\rangle \\
 & & \text{where } \sigma_1 = (\texttt{spread}\ \sigma(up+n)\,(\pi_1 \ldots \pi_n))
\end{array}
$$

Auxiliaries:

$$
\begin{array}{c}
(\texttt{deref}\ h\,l) = (h.1)(l) \\
(\texttt{list-ref}\ \nu(\pi\pi^*)) = (\nu = 0) \to \pi, (\texttt{list-ref}\,(\nu-1)\pi^*) \\
(\texttt{choose}\ b\,\pi\pi') = (b \to \pi, \pi')(\texttt{lookup}\ \iota\ up\,\sigma) = \sigma(up-\iota) \\
(\texttt{copy}\ \sigma\,\nu\,d\,s) = (\nu = 0) \to \sigma, (\texttt{copy}\ \sigma[\sigma(s)/d]\,(\nu-1)\,(d+1)\,(s-1)) \\
(\texttt{spread}\ \sigma\,\nu\,\langle\rangle) = \sigma \\
(\texttt{spread}\ \sigma\,\nu\,(v :: v^*)) = (\texttt{spread}\ \sigma[v/\nu]\,(\nu-1)\,v^*) \\
(\texttt{collect}\ \sigma i\,0) = \langle\rangle \\
(\texttt{collect}\ \sigma i\,\nu) = \sigma(i) :: (\texttt{collect}\ \sigma\,(i-1)\,(\nu-1))
\end{array}
$$

Figure 3: Operational Semantics of the Concrete Machine

To make this connection, we need to recall the main theorem from [8]. That report introduced a compiler function $\mathcal{CE}$, which took a Pure PreScheme program, and a symbol table, and produced an abstract machine program π which took a display u, a stack ζ, and a heap h and produced an answer. The semantics of Pure PreScheme programs was expressed with a valuation $\mathcal{E}$ which took a program p, an environment ρ, and a heap h to an answer. These valuations were related by the main theorem of [8]:

Theorem 1 (Compiler Correctness) *If p is a Pure PreScheme program and h_0 the empty heap, then*

$$(\mathcal{CE}[\![p]\!]\emptyset)\ \mathit{emptydisplay}\ h_0 = \mathcal{E}[\![p]\!]\emptyset h_0$$

Pure Prescheme has evolved somewhat since the proof in [8] was done. These changes are summarized as follows:

- The simple form `choose` has been modified to reflect the same syntax as the Scheme `case` statement. While the semantics remain the same, this change allows Pure PreScheme programs to be executed directly by Scheme implementations.
- The tail recursive forms `let` and `let*` have been added as simple forms. The semantics of these two new syntactic forms is based upon the semantics of the pre-existing tail recursive forms and are straightforward.
- The simple form `case`, formerly `choose`, has been added as a tail recursive form. Similarly, the semantics of this new form is based upon the semantics of the pre-existing simple form and is straightforward.

The changes to the language are minor but greatly enhance its utility. The added symmetry in the syntax makes it more intuitive for programmers to work with while the ability to execute Pure PreScheme programs in Scheme greatly facilitates program development.

The proof of Theorem 1 requires only minor changes to account for these modifications. The inductive hypotheses used in the proof remain the same. The only changes in the proof are the addition of two cases in the proof of equivalence of simple forms and compiled simple forms, and one case in the equivalence of tail-recursive forms and compiled tail recursive forms. These new cases follow the same form as the previous cases and are easy to show; the proof for the simple-expression `let` may be found in Appendix A. We now use the main theorem with confidence that it holds for the existing version of Pure PreScheme.

Recall that answers are either of the form $\langle \mathbf{ok}, v\rangle$ or $\langle \mathbf{error}\rangle$. We can state the following corollary to the Compiler Correctness Theorem:

Theorem 2 (Correctness of the Abstract Machine) *If* $\mathcal{E}[\![p]\!]\emptyset h_0 = \langle \mathbf{ok}, v\rangle$, *then in the abstract machine*

$$\langle \mathcal{CE}[\![p]\!]\emptyset, \mathit{emptydisplay}, \langle\rangle, h_0\rangle \stackrel{*}{\Longrightarrow} \langle \mathbf{ok}, v\rangle$$

Proof: If $\mathcal{E}[\![p]\!]\emptyset h_0 = \langle \mathbf{ok}, v\rangle$, then $(\mathcal{CE}[\![p]\!]\emptyset)$ $\mathit{emptydisplay}$ h_0. But the abstract machine simulates a quasi leftmost reduction sequence, which is always normalizing[1]. So the abstract machine must eventually reduce to $\langle \mathbf{ok}, v\rangle$. □

Note that for type-checked PreScheme programs, such as the Scheme 48 Virtual Machine, these restrictions are moot, because the typing rules guarantee that the program will never reach an error state.

In addition to this basic result, we will need the following lemma about the behavior of the abstract machine on validly compiled Pure PreScheme programs:

Theorem 3 (Bounded Space) *If* p *is a valid Pure PreScheme program, then there exist integers* N_0 *and* N *such that for any state* $\langle \pi, u, \zeta, h\rangle$ *in the computation sequence of* $\langle \mathcal{CE}[\![p]\!]\emptyset, \mathit{emptydisplay}, \langle\rangle, h_0\rangle$, $N_0 \leq |\mathit{dom}(u)| \leq N$.

Proof: (Sketch) According to the grammar in [8], a Pure PreScheme program p must consist of a setof global simple declarations followed by a set of local procedure declarations followed by a tail-recursive expression. Let N_0 be total number of global simple declarations and local procedure declarations, and let N be N_0 plus the deepest lexical depth in the program. Since all procedures are closed in the global environment, which has N_0 elements, it must be that $N_0 \leq |\mathit{dom}(u)|$; the usual argument about static scoping establishes $|\mathit{dom}(u)| \leq N$. A completely formal proof would have to consider the details of the syntax of Pure PreScheme and the compiler. This would be exhaustive but probably unenlightening [3]. □

Let **conc** be the operation that converts an abstract machine program to a concrete machine program by removing the tags on the operands of all the *constant* instructions, so for all abstract-machine programs π, $\mathbf{conc}(\pi) \simeq \pi$. Now we can state the main theorem:

Theorem 4 (Main Theorem) *If* p *is a Pure PreScheme program, and* $\mathcal{E}[\![p]\!]\emptyset h_0 = \langle \mathbf{ok}, v\rangle$, *where* v *is immediate data, then there exist values of* N_0 *and* N *such that*

$$\langle \mathbf{conc}(\mathcal{CE}[\![p]\!]\emptyset), -1, N, \sigma_0, h_0\rangle \stackrel{*}{\Longrightarrow} \langle \mathbf{ok}, v_c\rangle$$

and $v_c \simeq v$.

Proof: Choose N_0 and N as in Theorem 3. Let A_k and C_k denote the states of the abstract and concrete machines, respectively, after k steps. We will show that for all sufficiently large k, either $C_k \simeq A_k$ or both machines halt with corresponding answers.

We observe that the programs for both the concrete and abstract machines begin with a sequence of *add-global-to-env** instructions, followed by a *closerecs* instruction. These instructions establish the invariant that $C_k \simeq A_k$. Let k_0 be the number of steps to execute these instructions. We then show the following:

For all $k \geq k_0$, $C_k \simeq A_k$.

For the basis, we must establish that $C_{k_0} \simeq A_{k_0}$. We defer this to the end of the proof.

For the induction case, we must show that if $C_k \simeq A_k$, then $C_{k+1} \simeq A_{k+1}$. We do this by analysis of each instruction. In each case, we assume that $\langle \pi, up, sp, \sigma, h\rangle \simeq \langle \pi', u, \zeta, h'\rangle$, and therefore that

1. $\pi \simeq \pi'$,
2. $(\sigma, up) \models_U up \simeq u$,
3. $(\sigma, up) \models_Z (sp, N) \simeq \zeta$,
4. $h \simeq h'$
5. $N_0 \leq up \leq N \leq sp$, and
6. $(\sigma, u_0) \models_U N_0 \simeq u_0$

We proceed by cases on the form of the conclusion that $\pi \simeq \pi'$.

1. Assume we have

$$\langle (\mathit{constant}\ \epsilon\pi), up, sp, \sigma, h\rangle \simeq \langle (\mathit{constant}\ \epsilon'\pi'), u, \zeta, h'\rangle$$

   We must show that

$$\langle \pi, up, sp+1, \sigma', h\rangle \simeq \langle \pi', u, \epsilon :: \zeta, h'\rangle$$

   where $\sigma' = \sigma[\epsilon/\ sp+1]$. We consider each of the required conditions for $\simeq$ in turn:

   (a) $\pi \simeq \pi'$ by the definition of $\simeq$ for programs.

   (b) We have $(\sigma, up) \models_U up \simeq u$. Since $sp > up$, we have by lemma 2.2 that $(\sigma', up) \models_U up \simeq u$.

   (c) Since ϵ and ϵ' are immediate data, we have by lemma 1 that $(\sigma', up) \models_V \epsilon \simeq \epsilon'$. Since $(\sigma, up) \models_Z \langle sp, N\rangle \simeq \zeta$, then by lemma 2.3, $(\sigma', up) \models_Z \langle sp, N\rangle \simeq \zeta$. Therefore $(\sigma', up) \models_Z \langle sp+1, N\rangle \simeq (\epsilon' :: \zeta)$

   (d) $h \simeq h'$ by assumption.

(e) $N_0 \leq up \leq N \leq sp$, so $N_0 \leq up \leq N \leq sp+1$

(f) $(\sigma', u_0) \models_U N_0 \simeq u_0$ is true by lemma 2.2.

Thus, for each case, we will have six conditions to verify. For the rest of the proof, we will omit those conditions that are verified by arguments like those here, and include the verification only of the conditions pertaining to changed components of the abstract machine (e.g. ζ here), or of those conditions that are dangerous (i.e., those that require more delicate arguments). In most cases that will leave only one or two conditions to verify.

2. Assume we have $\langle(\textit{fetch-local}\ \iota\pi), up, sp, \sigma, h\rangle \simeq \langle(\textit{fetch-local}\ \iota\pi'), u, \zeta, h'\rangle$

   We must show that

   $$\langle\pi, up, sp+1, \sigma', h\rangle \simeq \langle\pi', u, (u(\iota) :: \zeta), h'\rangle$$

   where $\sigma' = \sigma[(\texttt{lookup}\ \iota\ up\,\sigma)/\ sp+1]$.

   We need to show that $(\sigma', up) \models_Z \langle sp+1, N\rangle \simeq (u(\iota) :: \zeta)$. We know from the assumption that $(\sigma, up) \models_Z \langle sp, N\rangle \simeq \zeta$, and therefore $(\sigma', up) \models_Z \langle sp, N\rangle \simeq \zeta$. By the definition of $\models_Z$, it will suffice to show that $(\sigma', up) \models_V \sigma'(sp+1) \simeq u(\iota)$. Now, $\sigma'(sp+1) = (\texttt{lookup}\ \iota\ up\,\sigma) = \sigma(up-\iota) = \sigma'(up-\iota)$, so it will suffice to show that $(\sigma', up) \models_V \sigma'(up-\iota) \simeq u(\iota)$. But this follows by lemma 2.2 from that fact that $(\sigma', up) \models_U up \simeq u$.

3. Assume we have $\langle(\textit{fetch-global}\ \iota\pi), up, sp, \sigma, h\rangle \simeq \langle(\textit{fetch-global}\ \iota\pi'), u, \zeta, h'\rangle$

   We must show that

   $$\langle\pi, up, sp+1, \sigma', h\rangle \simeq \langle\pi, u, ((\texttt{deref}'\ h'\,(u(\iota))) :: \zeta), h'\rangle$$

   where $\sigma' = \sigma[(\texttt{deref}\ h\,(\texttt{lookup}\ \iota\ up\,\sigma))/sp+1]$.

   We have $(\sigma, up) \models_Z \langle sp, N\rangle \simeq \zeta$, and therefore $(\sigma', up) \models_Z \langle sp, N\rangle \simeq \zeta$. So we need only show that $(\sigma', up) \models_V (\texttt{deref}\ h\,(\texttt{lookup}\ \iota\ up\,\sigma)) \simeq (\texttt{deref}'\ h'\,(u(\iota)))$ But this follows immediately from the definition of congruent heaps.

4. Assume we have $\langle(\textit{goto}\ \pi), up, sp, \sigma, h\rangle \simeq \langle(\textit{goto}\ \pi'), u, \zeta, h'\rangle$

   We must show that

   $$\langle\pi, up, sp, \sigma, h\rangle \simeq \langle\pi', u, \zeta, h'\rangle$$

   All the conditions follow immediately.

5. The case of *label* is the same as *goto*.

6. Assume we have

$$\langle (jmp\ \nu\pi_1\ \ldots\ \pi_\nu), up, sp, \sigma, h\rangle \simeq \langle (jmp\ \nu\pi'_1\ \ldots\ \pi'_\nu), u, (\langle \mathbf{int}, n\rangle :: \zeta), h'\rangle$$

We must show that

$$\langle (\texttt{list-ref}\ \sigma(sp)\,(\pi_1\ \ldots\ \pi_\nu), up, sp, \sigma, h\rangle \simeq \langle (\texttt{list-ref}\ n\,(\pi'_1\ \ldots\ \pi'_\nu)), u, \zeta, h'\rangle$$

We have $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (\langle \mathbf{int}, n\rangle :: \zeta)$, so $(\sigma, up) \models_V \sigma(sp) \simeq \langle \mathbf{int}, n\rangle$. Therefore, by the definition of $\models_V$, we have $\sigma(sp) = n$. Hence if $(\texttt{list-ref}\ \sigma(sp)\,(\pi_1\ \ldots\ \pi_\nu)) = \pi_j$, then $(\texttt{list-ref}\ n\,(\pi'_1\ \ldots\ \pi'_\nu)) = \pi'_j$, and $\pi_j \simeq \pi'_j$ by assumption.

7. Assume we have $\langle (save\text{-}env\ \pi), up, sp, \sigma, h\rangle \simeq \langle (save\text{-}env\ \pi'), u, \zeta, h'\rangle$

We must show that

$$\langle \pi, up, sp+1, \sigma', h\rangle \simeq \langle \pi', u, (\langle \mathbf{env}, u\rangle :: \zeta), h'\rangle$$

where $\sigma' = \sigma[up\,/sp+1]$. We check the requirements:

We know by that $(\sigma, up) \models_Z \langle sp, N\rangle \simeq \zeta$, so $(\sigma', up) \models_Z \langle sp, N\rangle \simeq \zeta$. Similarly, $(\sigma, up) \models_U up \simeq u$, so $(\sigma', up) \models_U up \simeq u$ as well. Hence, by the definition of $\models_V$, $(\sigma', up) \models_V up \simeq \langle \mathbf{env}, \upsilon\rangle$. Therefore, since $\sigma'(sp+1) = up$, $(\sigma', up) \models_Z \langle sp+1, N\rangle \simeq (\langle \mathbf{env}, u\rangle :: \zeta)$, as desired.

8. Assume we have

$$\langle (restore\text{-}env\ \pi), up, sp, \sigma, h\rangle \simeq \langle (restore\text{-}env\ \pi'), u, (v :: \langle \mathbf{env}, u'\rangle :: \zeta), h'\rangle$$

We must show that

$$\langle \pi, \sigma(sp-1), sp-1, \sigma', h\rangle \simeq \langle \pi', u', (v :: \zeta), h'\rangle$$

where $\sigma' = \sigma[\sigma(sp)/sp-1]$

$(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v :: \langle \mathbf{env}, u'\rangle :: \zeta)$. Hence $(\sigma, up) \models_V \sigma(sp-1) \simeq \langle \mathbf{env}, u'\rangle$, and therefore $(\sigma, up) \models_U \sigma(sp-1) \simeq u'$.

$(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v :: \langle \mathbf{env}, u'\rangle :: \zeta)$. Therefore $(\sigma, up) \models_V \sigma(sp) \simeq v$. Since $\sigma'(sp-1) = \sigma(sp)$, it follows that $(\sigma, up) \models_V \sigma'(sp-1) \simeq v$, so by lemma 2.1, $(\sigma', up) \models_V \sigma'(sp-1) \simeq v$. Also, by lemma 2.3, we have $(\sigma', up) \models_Z \langle sp-2, N\rangle \simeq \zeta$. Therefore, by the definition of $\models_Z$, we have $(\sigma', up) \models_Z \langle sp-1, N\rangle \simeq (v :: \zeta)$

9. Assume we have

$$\langle (update\text{-}store\ \iota\pi), up, sp, \sigma, h\rangle \simeq \langle (update\text{-}store\ \iota\pi'), u, (v :: \zeta), h'\rangle$$

We must show that

$$\langle \pi, up, sp, \sigma, (\texttt{update}\ \sigma(up - \iota)\, \sigma(sp)\, h)\rangle \simeq \langle \pi', u, (v :: \zeta), (\texttt{update}\ u(i)\, v\, h')\rangle$$

By assumption, we have $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v :: \zeta)$, so $(\sigma, up) \models_V sp \simeq v$. Since $(\sigma, up) \models_U up \simeq u$, we have $(\sigma, up) \models_V \sigma(up - i) \simeq u(i)$. Also by assumption we have $h \simeq h'$. Therefore it follows that $(\texttt{update}\ \sigma(up - \iota)\, \sigma(sp)\, h) \simeq (\texttt{update}\ u(i)\, v\, h)$.

10. Assume we have

$$\langle (\textit{prim-apply}\ n\, v\, \pi), up, sp, \sigma, h\rangle \simeq \langle (\textit{prim-apply}\ n\, v'\, \pi'), u, \zeta_1, h'\rangle$$

where $\zeta_1 = (v_1 :: \ldots :: v_n :: \zeta)$. We must show that

$$\langle \pi, up, sp - n + 1), \sigma', h_1\rangle \simeq \langle \pi', u, (w_1' :: \zeta), h_1'\rangle$$

where

$$\begin{aligned}
&(w_1, h_1) = \texttt{apply-prim}\ v\, (\texttt{collect}\ \sigma\ sp\ n) h \\
&\sigma' = \sigma[w_1 /\ sp - n + 1] \\
&(w_1', h_1') = \texttt{apply-prim}'\ v'\, (v_1 :: \ldots :: v_n :: \langle\rangle)
\end{aligned}$$

$(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v_1 :: \ldots :: v_n :: \zeta)$, so by the definition of $\models_Z$ we have

$$\begin{aligned}
&(\sigma, up) \models_V \sigma(sp) \simeq v_1 \\
&(\sigma, up) \models_V \sigma(sp - 1) \simeq v_2 \\
&\ldots \\
&(\sigma, up) \models_V \sigma(sp - n + 1) \simeq v_n \\
&(\sigma, up) \models_Z \langle sp - n, N\rangle \simeq \zeta
\end{aligned}$$

Furthermore, $(\texttt{collect}\ \sigma\ sp\ n) = (\sigma(sp)\ \ldots\ \sigma(sp - n + 1))$, so the arguments to apply-prim and $\texttt{apply-prim}'$ are componentwise congruent. Therefore we have $h_1 \simeq h_1'$ and $w_1 \simeq w_1'$ (as immediate data). So $(\sigma', up) \models_V w_1 \simeq w_1'$.

Now, $\sigma'(sp - n + 1) = w_1$, so $(\sigma', up) \models_V \sigma'(sp - n + 1) \simeq w_1'$, and furthermore $(\sigma', up) \models_Z \langle sp - n, N\rangle \simeq \zeta$, so we conclude that $(\sigma', up) \models_Z \langle sp - n + 1, N\rangle \simeq (w_1 :: \zeta)$, as desired.

11. The cases for *restore-env/ignore*, *update-store/ignore*, and *prim-apply/ignore* are similar to the corresponding value-returning instructions.

12. Assume we have

$$\langle (\textit{brf}\ \pi_1\, \pi_2), up, sp, \sigma, h\rangle \simeq \langle (\textit{brf}\ \pi_1'\, \pi_2'), u, (\langle \mathbf{bool}, b\rangle :: \zeta), h'\rangle$$

We must show that

$$\langle (\texttt{choose}\ \sigma(sp)\, \pi_1\, \pi_2), up, sp - 1, \sigma, h\rangle \simeq \langle (\texttt{choose}\ b\, \pi_1'\, \pi_2'), u, \zeta, h'\rangle$$

$\pi_1 \simeq \pi_1'$ and $\pi_2 \simeq \pi_2'$ and $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (\langle \mathbf{bool}, b\rangle :: \zeta)$. Therefore $(\sigma, up) \models_V sp \simeq \langle \mathbf{bool}, b\rangle$, so $\sigma(sp) = b$. Hence $(\mathit{choose}\,\sigma(sp)\,\pi_1\pi_2) \simeq (\mathtt{choose}\; b\,\pi_1'\pi_2')$

$(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v :: \zeta)$. Therefore $(\sigma, up) \models_Z \langle sp-1, N\rangle \simeq \zeta$

13. Assume we have $\langle (\mathit{add\text{-}to\text{-}env}\ \nu\pi), up, sp, \sigma, h\rangle \simeq \langle (\mathit{add\text{-}to\text{-}env}\ \nu\pi'), u, (v_1 :: \ldots :: v_\nu :: \zeta), h'\rangle$

    We must show that

$$\langle \pi, (up+\nu), (sp-\nu), \sigma', h\rangle \simeq \langle \pi', u', \zeta, h'\rangle$$

    where $\sigma' = (\mathtt{copy}\ \sigma\,\nu\,(up+1)\ sp)$ and $u' = (\mathit{extends}_{r\,l}\ u\,(v_1 \ldots v_\nu))$

    $(\sigma, up) \models_U up \simeq v$ and $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v_1 :: \ldots :: v_\nu :: \zeta)$. Therefore for $1 \leq j \leq \nu$, we have $(\sigma, up) \models_V \sigma(sp-j+1) \simeq v_j$. By the definition of copy we have $\sigma'(up+j) = \sigma(sp-j+1)$, so $(\sigma, up) \models_V \sigma'(up+j) \simeq v_j$. Since σ and σ' agree up to up, we get $(\sigma', up) \models_V \sigma'(up+j) \simeq v_j$, so by the definition of $\models_V$ we have $(\sigma', up+\nu) \models_V \sigma'(up+j) \simeq v_j$. So, by the definition of $\models_U$, $(\sigma', up+\nu) \models_U (up+\nu) \simeq (\mathit{extends}_{r\,l}\ u\,(v_1 \ldots v_\nu))$

    $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v_1 :: \ldots :: v_\nu :: \zeta)$. Hence $(\sigma', up) \models_Z \langle sp-\nu, N\rangle \simeq \zeta$, and therefore $(\sigma', up+\nu) \models_V \langle sp-\nu, N\rangle \simeq \zeta$.

    Last, we must show that $up+\nu \leq N$. By Theorem 3, we know that $|\mathit{dom}(\mathit{extends}_{r\,l}\ u\,(v_1 \ldots v_\nu))| \leq N$. But the left-hand side of this inequality is equal to $up+\nu$.

14. The case of *add-to-env** is similar to *add-to-env*, but simpler since it adds only a single item to the environment.

15. Assume we have $\langle (\mathit{tail\text{-}call}\ \nu), up, sp, \sigma, h\rangle \simeq \langle (\mathit{tail\text{-}call}\ \nu), u, (\langle \mathbf{proc}, \langle \pi, u_0\rangle\rangle :: v_1 :: \ldots :: v_\nu :: \zeta), h'\rangle$

    We must show that

$$\langle \sigma(sp), (N_0+\nu), (N-1), s_1, h\rangle \simeq \langle \pi, (\mathit{extends}_{r\,l}\ u_0\,(v1 \ldots v_\nu)), \langle\rangle, h'\rangle$$

    where $\sigma_1 = (\mathtt{copy}\ \sigma\,\nu\,(N_0+1)\,(sp-1))$

    (a) $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (\langle \mathbf{proc}, \langle \pi, u\rangle\rangle :: v_1 :: \ldots :: v_\nu :: \zeta)$. Therefore $\sigma(sp) = \pi$, by the definition of $\models_Z$.

    (b) We calculate:

$(\sigma, up) \models_Z \langle sp, N\rangle \simeq (\langle \mathbf{proc}, \langle \pi, u\rangle\rangle :: v_1 :: \ldots :: v_\nu :: \zeta)$	assumption
for $1 \leq j \leq \nu$, $(\sigma, up) \models_V \sigma(sp - j) \simeq v_j$	definition of $\models_Z$
for $1 \leq j \leq \nu$, $\sigma_1(N_0 + 1 + j) = \sigma(sp - j)$	definition of `copy`
for $1 \leq j \leq \nu$, $(\sigma, up) \models_V \sigma_1(N_0 + 1 + j) \simeq v_j$	substituting equals for equals
for $1 \leq j \leq \nu$, $(\sigma, N_0) \models_V \sigma_1(N_0 + 1 + j) \simeq v_j$	v_j are all expressed values
for $1 \leq j \leq \nu$, $(\sigma_1, N_0) \models_V \sigma_1(N_0 + 1 + j) \simeq v_j$	lemma 2.1
$(\sigma, N_0) \models_U N_0 \simeq u_0$	assumption
$(\sigma_1, N_0) \models_U N_0 \simeq u_0$	lemma 2.2
$(\sigma_1, N_0) \models_U N_0 + \nu \simeq (extends_{r\,l}\ u_0\,(v1 \ldots v_\nu))$	definition of $\models_U$
$(\sigma_1, N_0 + \nu) \models_U N_0 + \nu \simeq (extends_{r\,l}\ u_0\,(v1 \ldots v_\nu))$	padding lemma

We know that the v_j are all expressed values because otherwise the state $A_{k+1} = \langle \pi, (extends_{r\,l}\ u_0\,(v1 \ldots v_\nu)), \langle\rangle, h'\rangle$ would not be a legal abstract-machine state, which would contradict the assumption that the A_k are all legal states.

(c) $(\sigma', up) \models_Z \langle N - 1, N\rangle \simeq \langle\rangle$

16. Assume we have $\langle (halt), up, sp, \sigma, h\rangle \simeq \langle (halt), u, (v :: \zeta), h'\rangle$

    We must show that $\sigma(sp) \simeq v$. Now, from the statement of the theorem, v must be immediate data, so the required conclusion follows immediately from $(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v :: \zeta)$.

17. Last, if the state $\langle \pi, u, \zeta, h'\rangle$ of the abstract machine does not match the left-hand side of any of the rules of section 2, then the abstract machine goes to an error state. This contradicts the assumptions of the theorem, so this case can never arise.

This completes the induction step. We now return to the base case. The machines start in states

$$\langle \pi, -1, N, \sigma_0, h_0\rangle$$

for the concrete machine and

$$\langle \pi', emptydisplay, \langle\rangle, h'_0\rangle$$

for the abstract machine. Comparing these states with the definition of $\simeq$ for states, we see that all of the conditions for correspondence hold, except for $N_0 \leq up$ and $(\sigma, u_0) \models_U N_0 \simeq u_0$ Let $\simeq_0$ denote this weakened relation.

The programs π and π' begin with a sequence of *add-global-to-env** instructions. We first show that these instructions preserve $\simeq_0$. Assume we have

$$\langle (add\text{-}global\text{-}to\text{-}env^*\ \pi), up, sp, \sigma, h\rangle \simeq_0 \langle (add\text{-}global\text{-}to\text{-}env^*\ \pi), u, \zeta, h'\rangle$$

We must show that

$$\langle \pi, up + 1, sp, \sigma_1, h_1\rangle \simeq_0 \langle \pi, u_1, \zeta, h'_1\rangle$$

where

$$\begin{aligned} \sigma_1 &= (\sigma[(\texttt{new}\ h)/(up+1)]) \\ h_1 &= (\texttt{update}\,(\texttt{new}\ h)\,0\,h) \\ u_1 &= (extends_{r\ g}\ u\,(\texttt{new}\ h')) \\ h_1' &= (\texttt{update}\,(\texttt{new}\ h')\,\langle \textbf{int}, 0\rangle\, h') \end{aligned}$$

For the interesting components, we calculate as follows:

$(\sigma, up) \models_U up \simeq u$	assumption
$(\sigma_1, up+1) \models_U up \simeq u$	lemma 2.2
$(\texttt{new}\ h) \simeq (\texttt{new}\ h')$	
$(\sigma_1, up+1) \models_U up+1 \simeq (extends_{r\ g}\ u\,(\texttt{new}\ h'))$	definition of $\models_U$.
$(\sigma, up) \models_Z \langle sp, N\rangle \simeq (v :: \zeta)$	assumption
$(\sigma, up) \models_Z \langle (sp-1), N\rangle \simeq \zeta$	definition of $\models_Z$
$(\sigma_1, up+1) \models_Z \langle (sp-1), N\rangle \simeq \zeta$	
$h \simeq h'$	assumption
$(\texttt{new}\ h) \simeq (\texttt{new'}\ h')$	
$(\texttt{update}\,(\texttt{new}\ h)\,0\,h) \simeq (\texttt{update}\,(\texttt{new}\ h')\,\langle \textbf{int}, 0\rangle\, h')$	
$h_1 \simeq h_1'$	

$up+1 \leq N$ because $\langle \pi, u_1, \zeta, h_1'\rangle$ is a valid abstract-machine state, and therefore $up+1 = |dom(u_1)| \leq N$.

So the *add-global-to-env** instructions preserve $\simeq_0$. Last, we turn to the *closerecs* instruction that concludes the prelude. Assume we have

$$\langle (closerecs\,(\pi_1 \ldots \pi_n)\,\pi), up, sp, \sigma, h\rangle \simeq_0 \langle (closerecs\,(\pi'_1 \ldots \pi'_n)\,\pi'), u, \zeta, h'\rangle$$

We must show that

$$\langle \pi, up+n, sp, \sigma', h\rangle \simeq \langle \pi', u_0, \zeta, h'\rangle$$

where

$$\begin{aligned} \sigma' &= (\texttt{spread}\ \sigma(up+n)\,(\pi_1 \ldots \pi_n)) \\ u_0 &= (\texttt{fix}(\lambda u' \cdot (extends_{r\ l}\ u\,(\langle \textbf{proc}, \langle \pi'_1, u'\rangle\rangle \ldots \langle \textbf{proc}, \langle \pi'_n, u'\rangle\rangle))))) \end{aligned}$$

Note that here we need to establish $\simeq$, not $\simeq_0$.

At the end of this instruction, we have $up = N_0$, by the definition of N_0.

By the definition of σ', for $1 \leq i \leq n$ we have $\sigma'(up+n-i+1) = \pi_{n-i}$, so $(\sigma', up) \models_V \sigma'(up+n-i+1) \simeq \langle \textbf{proc}, \langle \pi_i', u_0\rangle\rangle$, and $(\sigma', up) \models_U up \simeq u$, so by the definition of $\models_U$ we have $(\sigma', up) \models_U up+n \simeq u_1$, where

$$u_1 = (extends_{r\ l}\ u\,(\langle \textbf{proc}, \langle \pi'_1, u_0\rangle\rangle \ldots \langle \textbf{proc}, \langle \pi'_n, u_0\rangle\rangle))$$

But u_1 is just one unwinding of u_0, so $u_1 = u_0$.

All the other conditions are established trivially. This completes the establishment of the invariant, and therefore establishes the base step. □

7 Implementation

An assembler for PreScheme was developed in parallel with this specification. It generates code for the Motorola 68000. Representing the concrete model with the 68000 requires a mapping similar to one between the abstract and concrete machine just detailed. This section informally describes that mapping.

Immediate data is modeled via a 32 bit quantity (4 bytes). Thus immediate data can be stored directly in any of the 68000 registers (`D0`–`D7` and `A0`–`A7`) or in 4 consecutive bytes of user memory.

The concrete machine's store, σ, is modeled by 68000 with the user stack and its stack pointer, *sp*, is modeled by the 68000 stack pointer `SP` (a.k.a. address register `A7`). The environment register, *up*, is modeled by the address register `A1`. For efficiency, the initial environment pointer, N_0, and initial stack pointer, N, are also kept in address registers for quick reset on tail call.

Two details must be noted in using the 68000 user stack as described. First, 68000 user stack grows downward rather than upward so rather than increment the stack pointer on a push, it must be decremented (similarly for popping et al). Second, to decrement the stack pointer by one unit of immediate data, SP must be decremented by 4, the number of bytes necessary to represent the immediate data.

Procedures are represented by their effective addresses. When the initial environment u_0 is being created, these effective addresses (which are 32 bit quantities) are pushed on the stack. On tail-call, the effective address is loaded from the stack into the program counter.

The environment representation is almost completely analogous to that of the concrete machine, with environment values kept in the first $4 * N$ bytes of the stack. The local stack is handled almost completely analogously as well. The only difference is that for efficiency, the top 8 values of the local stack are cached in in the data registers.

Representation of the concrete machine's heap is done differently for the different components of the heap. The first component of the heap, the store for mutable variables, is kept directly in the first $4 * j$ bytes of the user stack (where j is the number of global variables in the program being compiled). The second component of the heap, which tells how many objects are stored in the first component, remains constant during the execution of a given Pure PreScheme program after the initial environment has been established (in particular, it is also j). It is modeled implicitly through the correctness of the instructions manipulating mutable data. The third component of the heap, that which the primitives manipulate, are handled by the primitives (usually via operating system calls).

Performance of the compiler is difficult to measure as there is no other Pure PreScheme compiler available. As an alternative, we decided to test our compiler

Fibonacci	Pure PreScheme	Chez Scheme	gcc
20	0.4s	0.7s	0.3s
22	0.9s	1.8s	0.8s
24	2.4s	4.8s	2.0s
26	6.3s	12.5s	5.3s
28	16.4s	32.9s	13.9s
29	26.5s	53.5s	22.4s

Figure 4: Pure PreScheme vs. Chez Scheme vs. C

against Chez Scheme, an optimizing native code Scheme compiler, and against gcc, a standard C compiler, on an unloaded Sun 3/160.

The test case we used was a Pure PreScheme version of Fibonacci, using an explicit array to model the recursion stack. This test duplicates the general structure of the Vlisp VM. The results summarized in Figure 4 indicate that our compiler produces code executing about twice as fast as the code produced by Chez Scheme (Chez/PPS = 2.01). To make the test as fair as possible, we used fixnum operators in the Chez Scheme version, and set Chez Scheme's optimization level 3 to produce maximum inlining.

We conjecture that most of the difference between the PPS compiler and Chez Scheme is accounted for by the extra operations that Scheme must perform in order to do garbage collection. For example, Chez sets a byte in a "dirty vector" for each `vector-set!` to support the generational garbage collector. According to Kent Dybvig [personal communication] this costs 4–5 instructions and some extra memory traffic for each assignment. Our benchmark and the VM are both quite vector-intensive.

This data supports Kelsey's thesis about PreScheme: that by restricting the language, we can obtain much better performance. We therefore tested our benchmark against a straightforward transcription of the fibonacci program into C. In the C version, tail-recursion is replaced by explicit goto's. The C version also uses `malloc` to allocate the array dynamically, just as the PPS version does. We were pleased to see that our code ran within 20% of the speed of the code produced by gcc (PPS/C = 1.18).

Thus, the code produced by this verified compiler was competitive with conventionally produced code.

8 Conclusions

We have presented a verified storage layout for a Pure PreScheme compiler. The proof verifies that an abstract machine that manipulates trees and a concrete machine that manipulates realistic data structures always produce corresponding results. From this experience we can draw a number of conclusions.

Our approach factored the correctness proof into two basic layers:

1. Proof of correctness of the compiler, which translates from the source language to abstract-machine code. This was the part reported in [8].
2. Proof of correctness of the implementation of the abstract machine. In this report we prove the correctness of the run-time structure of our implementation; the issue of translation from abstract to concrete machine code will be reported separately.

This factorization proved to be a successful decomposition. The factoring of the proof into the correctness of the compiler and the correctness of the implementation of the abstract machine allows us to use both denotational and operational reasoning to their best advantage. Furthermore, each portion of the proof gave us confidence in a different part of the implementation.

The proof of correctness of the run-time structure is operational, rather than denotational, in its structure: that is, it proceeds by induction on the number of steps taken by the machine, rather than on the size of the program. We believe this is a fundamental improvement. Previous proofs of compiler correctness, such as those in [6, 10] were highly complex because they used induction on the construction of reflexive domains as a denotational analog to induction on length of computation. Our proof would likely be just as complex had we adopted this strategy (see [13] for an attempt to do a much smaller problem in a purely denotational style). By using induction on computation length directly, we avoid this indirection. Furthermore, since we deal directly with terms (trees) rather than with their denotations, we avoid the complication of "inclusive predicates," with their attendant complexity.

The implementation experience showed that having a validated compiler and run-time structure eliminated most bugs in the areas covered by the proofs. The various delivered versions of the compilers had a number of bugs, but these were almost entirely in one of two categories:

1. Incompatibilities with the PreScheme front end [9] (errors in syntax, missing primitives, etc.). These were artifacts of the circumstance that the various portions of the compiler, including the interface with the front end, were developed concurrently.
2. Problems with the assembly code sequences generated for the concrete-machine instructions. These were mostly minor in nature (registers not

being saved across routine calls, etc.), and were below the grain of the proof. Extending the proof to reach this level would require an extremely detailed model of the behavior of the machine and operating system (see, e.g. [7]). In practice, however, the concrete machine was at a sufficiently low level that implementation of the primitives was easy.

The method of formalizing storage layout relations seems to be flexible enough to model standard representation strategies. More of these are presented in [12].

The proofs resemble traditional Hoare-style verification proofs, but seem quite stylized and may be amenable to mechanical proof-checking. This work extends the realm of "dull" proofs from the byte-code compiler level [2, 11] to something much closer to an ordinary "compiler-writer's abstract machine." We believe this is a mark of success for this effort.

The performance of the PPS compiler was competitive with conventionally produced code. On a small benchmark, the code produced by the PPS compiler ran within 20% of the code produced by gcc for a simple-minded transcription of the benchmark into C. Therefore, use of a verified compiler need not entail catastrophic performance penalties.

In this report, we did not deal with the correctness of the translation from abstract to concrete machine code. For the machines in this report, this is trivial, consisting simply of translation of literals from tagged to untagged form. The real compiler, however, translates the tree-like abstract-machine code to a linear representation. This more complex representation requires two additional portions in the proof:

1. The storage layout relation for programs would have to be modified to formalize the linear representation, and the proof of correctness of the concrete machine (the main theorem of this report) would have to be modified accordingly. This should be easy, since, unlike the other components of the machines, the program store remains constant during execution. The outlines of the proof are clear in [4, 12].

2. We would then have to show that our translator converts an abstract-machine program to corresponding state of the program store. Such an assembler-correctness proof would deal with the specifics of the translation process and would resemble in its specificity and level, the compiler-correctness proof of [8]. We have good ideas about how this proof should be structured, and we will report on this later, either in Oliva's forthcoming Ph.D. thesis or a separate technical report.

One surprise in the development of the proof was the necessity to import information from higher levels in the compilation process. We were unable

to prove the congruence between abstract and concrete machines except for programs that were in the image of the compiler.

For the Scheme VM, this restriction was not important, because the VM is type-correct. This guarantees that it will never reach an error state, so all the error tests were guaranteed to succeed. However, it would be desirable to formulate a cleaner version of the correctness result.

One way to do this might be to factor the proof as follows: Call an abstract-machine program (N_0, N)-*bounded* iff the conclusion of the Bounded Space Theorem holds for it. Then we believe that the Main Theorem holds for any (N_0, N)-bounded abstract-machine program. However, we did not have time to verify that this is the only information needed from the correctness proof of [8].

We believe the problem of needing to import information from higher levels of the compiler is related to the issue of incorporating flow analyses, which similarly verify the maintenance of non-local invariants, into compiler correctness proofs. In this case, we needed to rely on the invariance of $N_0 \leq |dom(u)| \leq N$, which is not possible to verify locally without additional information. The operational type-soundness of the program is another such invariant. We believe that developing a systematic way of incorporating such information is the most important outstanding problem in semantics-based compiler correctness.

A Compiling Simple Let Expressions

In this appendix we show the correctness of compiling simple `let` expressions. Correctness of simple `let*` expressions and the analogous simple commands follow similarly.

We begin by showing the denotational definition of the simple `let`:

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

Using our standard methodology, we derive the compiler semantics to be:

$$\begin{aligned}\mathcal{CSE}[\![(\texttt{let}\,(\texttt{LSD})\ \texttt{S})]\!] = \lambda\gamma\pi.(&\mathit{save\text{-}env}\,(\mathcal{CDL}[\![\texttt{LSD}]\!]\gamma\\ &(\mathit{add\text{-}to\text{-}env}\,(\mathcal{CSE}[\![\texttt{S}]\!]\,(\mathit{extends}_{c\,l}\ \gamma\,\mathcal{GL}[\![\texttt{LSD}]\!])\,(\mathit{restore\text{-}env}\ \pi)))))\end{aligned}$$

where the new machine instructions are defined as:

$$\mathit{save\text{-}env} = \lambda\pi.\lambda u\zeta.\pi u(u :: \zeta)$$

$$\mathit{restore\text{-}env} = \lambda\pi.\lambda u\zeta.\pi(\texttt{top}\,(\texttt{pop-first}\ 1\,\zeta))((\texttt{top}\ \zeta) :: (\texttt{pop-first}\ 2\,\zeta))$$

Recall that to prove correctness of the compiler we must prove that execution of the compiled simple expression, `let`, results in same value computed by the denotational definition. Formally stated, the induction hypothesis for simple expressions was:

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

Proof of correctness for the `let` progresses as the other alternatives for simple expressions:

$(\mathcal{CSE}[\![\mathtt{(let\ (LSD)\ S)}]\!])u\zeta$

$= (\mathit{save\text{-}env}\,(\mathcal{CDL}[\![\mathtt{LSD}]\!]\gamma(\mathit{add\text{-}to\text{-}env}\,(\mathcal{CSE}[\![\mathtt{S}]\!]\,(\mathit{extends}_{c\,l}\ \gamma\,\mathcal{GL}[\![\mathtt{LSD}]\!])\,(\mathit{restore\text{-}env}\ \pi)))))u\zeta$

$= (\mathcal{CDL}[\![\mathtt{LSD}]\!]\gamma(\mathit{add\text{-}to\text{-}env}\,(\mathcal{CSE}[\![\mathtt{S}]\!]\,(\mathit{extends}_{c\,l}\ \gamma\,\mathcal{GL}[\![\mathtt{LSD}]\!])\,(\mathit{restore\text{-}env}\ \pi))))u(u :: \zeta)$

$= \mathcal{DL}[\![\mathtt{LSD}]\!](u \circ \gamma)(\lambda\epsilon^*.(\mathit{add\text{-}to\text{-}env}\,(\mathcal{CSE}[\![\mathtt{S}]\!]\,(\mathit{extends}_{c\,l}\ \gamma\,\mathcal{GL}[\![\mathtt{LSD}]\!])\,(\mathit{restore\text{-}env}\ \pi)))u(\epsilon^* : u :: \zeta))$

$= \mathcal{DL}[\![\mathtt{LSD}]\!](u \circ \gamma)(\lambda\epsilon^*.(\mathcal{CSE}[\![\mathtt{S}]\!]\,(\mathit{extends}_{c\,l}\ \gamma\,\mathcal{GL}[\![\mathtt{LSD}]\!])\,(\mathit{restore\text{-}env}\ \pi))(\mathit{extends}_{r\,l}\,u\epsilon^*)(u :: \zeta))$

$$= \mathcal{DL}[\![\mathtt{LSD}]\!](u \circ \gamma)(\lambda\epsilon^*.\mathcal{SE}[\![\mathtt{S}]\!]\,(u \circ \gamma)[(\mathtt{map}\,(\mathit{in}\,D)\,\epsilon^*)/\mathcal{GL}[\![\mathtt{LSD}]\!]]$$
$$(\lambda\epsilon.(\mathit{restore\text{-}env}\ \pi)(\mathit{extends}_{r\,l}\,u\epsilon^*)(\epsilon :: u :: \zeta)))$$

$= \mathcal{DL}[\![\mathtt{LSD}]\!](u \circ \gamma)(\lambda\epsilon^*.\mathcal{SE}[\![\mathtt{S}]\!]\,(u \circ \gamma)[(\mathtt{map}\,(\mathit{in}\,D)\,\epsilon^*)/\mathcal{GL}[\![\mathtt{LSD}]\!]](\lambda\epsilon.\pi u(\epsilon :: \zeta)))$

$= \mathcal{SE}[\![\mathtt{(let\ (LSD)\ S)}]\!](u \circ \gamma)(\lambda\epsilon.\pi u(\epsilon :: \zeta))$

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[2] William Clinger. The Scheme 311 Compiler: An Exercise in Denotational Semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.

[3] Edsger W. Dijkstra. Speech at the Occasion of an Anniversary. In *Selected Writings On Computing: A Personal Perspective*, pages 50–53. Springer-Verlag, 1982.

[4] John Hannan. Making Abstract Machines Less Abstract. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523 of *Lecture Notes in Computer Science*, pages 618–635. Springer-Verlag, Berlin, Heidelberg, and New York, 1991.

[5] Richard Kelsey. PreScheme: A dialect of Scheme for Systems Programming. in preparation, 1993.

[6] R. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, 1976. Also Wiley, New York.

[7] J Strother Moore. A Mechanically Verified Language Implementation. Technical Report 30, Computational Logic Inc., Austin, TX, September 1988. also appeared in J. of Computer-Aided Deduction.

[8] Dino P. Oliva and Mitchell Wand. A Verified Compiler for Pure PreScheme. Technical Report NU-CCS-92-5, Northeastern University College of Computer Science, February 1992.

[9] J. D. Ramsdell, W. M. Farmer, J. D. Guttman, L. G. Monk, and V. Swarup. The VLISP PreScheme Front End. M 92B098, The MITRE Corporation, 1992.

[10] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[11] Mitchell Wand. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

[12] Mitchell Wand and Dino P. Oliva. Proving the Correctness of Storage Representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.

[13] Mitchell Wand and Zheng-Yu Wang. Conditional Lambda-Theories and the Verification of Static Properties of Programs. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, pages 321–332, 1990.