LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993
© 1993 Kluwer Academic Publishers – Manufactured in The Netherlands

A Verified Compiler for VLISP PreScheme

DINO P. OLIVA[*] (*oliva@cse.ogi.edu*)
College of Computer Science
161 Cullinane Hall
Northeastern University
Boston, MA 02115

JOHN D. RAMSDELL[†] (*ramsdell@mitre.org*)
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

MITCHELL WAND[‡] (*wand@ccs.neu.edu*)
College of Computer Science
161 Cullinane Hall
Northeastern University
Boston, MA 02115

Keywords: verified, compiler

Contents

[*]This work was supported by Rome Laboratories of the United States Air Force, contract No. F19628-89-C-0001, through the MITRE Corporation, and by NSF and DARPA under NSF grants CCR-9002253 and CCR-9014603. Author's current address: **??**

[†]This work was supported by Rome Laboratories of the United States Air Force, contract No. F19628-89-C-0001.

[‡]This work was supported by Rome Laboratories of the United States Air Force, contract No. F19628-89-C-0001, through the MITRE Corporation, and by NSF and DARPA under NSF grants CCR-9002253 and CCR-9014603.

Abstract. This paper describes a verified compiler for PreScheme, the implementation language for the VLISP run-time system. The compiler and proof were divided into three parts: A transformational front end that translates source text into a core language, a syntax-directed compiler that translates the core language into combinator-based tree-manipulation language, and a linearizer that translates combinator code into code for an abstract stored-program machine with linear memory for both data and code. This factorization enabled different proof techniques to be used for the different phases of the compiler, and also allowed the generation of good code. Finally, the whole process was made possible by carefully defining the semantics of VLISP PreScheme rather than just adopting Scheme's. We believe that the architecture of the compiler and its correctness proof can easily be applied to compilers for languages other than PreScheme.

1. Introduction

As part of the VLISP project, we have developed an architecture for the specification and implementation of verified optimizing compilers. We have used this architecture to develop a verified compiler for PreScheme.

This architecture divides the compiler into three components. In the case of PreScheme, these components are:

1. A transformational front end that translates source text into a core language called Pure PreScheme.

2. A syntax-directed compiler that translates Pure PreScheme into a

combinator-based tree-manipulation language. We call this language *combinator code.*

3. An assembler that translates combinator code into code for an abstract stored-program machine with linear memory for both data and code.

We believe that this is a good architecture for the specification and implementation of verified optimizing compilers. A smaller example of this proof architecture (without the front end) is given in [28]; another example, with a more elaborate assembler, is given in [17].

The correctness of each transformation of the program is justified relative to an appropriate semantics:

- The source language and its core subset (PreScheme and Pure PreScheme in our example) share a denotational semantics.
- The language of combinators has both a denotational semantics, expressed using the same domains as the source language, and an operational semantics. The latter may be derived from the former by general theorems of the λ-calculus.
- The stored-program machine is specified using an operational semantics.

This separation of semantics allows each transformation to be proved correct using a suitable proof technique:

1. The transformational front end is shown to preserve the denotational semantics of the source language, by induction on the number of transformation steps.
2. The syntax-directed compiler is shown correct by denotational reasoning, using structural induction on the phrases of the core language. The proof establishes a relation between the denotational semantics of the core language and the denotational semantics of the combinator language. The proof is made feasible by the fact that the core and combinator semantics are expressed using the same domains, and hence the same λ-theory.
3. The assembler is proved correct by operational reasoning, using the operational semantics of the combinator language and the operational semantics of the stored-program machine. We define a relation, called

Level	Language	Acronym	Section
Source Language	VLISP PreScheme	VLPS	2
Parsed Source Language	Macro-Free PreScheme	MFPS	2
Core Language	Pure PreScheme	PPS	2
Tree Structured Language	Combinator Code	CC	5
Linear Data Language	Linear-Data Code	LDC	6
Linear Program Language	Stored-Program Code	SPC	7
Target Language	Assembly Code	AC	9

Figure 1: Levels in the PreScheme compilation process

a *storage-layout relation*, that determines when the combinator machine and the stored-program machine are in corresponding states, and we show that operation of the machines preserves this relation.

The verification of the PreScheme compiler is at the same level as that of the VLISP system itself: the algorithms are verified with respect to a formal denotational semantics of PreScheme and a formal model of an abstract target machine. The implementation of the algorithms was carried out in Scheme, and code for the abstract target machine was translated into assembly language for two real target machines (Motorola 68000 and SPARC). While neither of these implementation steps was verified, both were straightforward; for a discussion of the trustworthiness of this approach, see [9].

In the case of the PreScheme compiler, the assembler proof is performed in two steps: we first relate the combinator machine to a machine, called the *linear-data machine*, in which the data store is linear, but the program is still a tree. We then relate the linear-data machine to the *stored-program machine*, in which both the data and code are kept in linear memories.

This gives a total of five language or machine levels and four major proofs. In general, we will present level n, followed by the proof or proofs relating level n to level $n-1$. Figure 1 tabulates the various levels.

The rest of the paper is organized as follows:

1.1. Introduction

Following [3], we will give a synopsis of each section in the paper. Section n will be summarized in section $1.n$.

1.2. VLISP PreScheme

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

However, PreScheme is particularly suitable for systems programming because every valid PreScheme program can be executed using only a C-like run-time system. A compiler for this language will reject any program that requires dynamic type checking or creation of closures at run-time. Furthermore, it is intended that PreScheme programs be runnable without the use of automatic storage reclamation.

There are a number of dialects of PreScheme that played a part in this project. VLISP PreScheme is the source language that our compiler translates. It is the least restrictive PreScheme dialect we consider. Macro-Free PreScheme programs result from VLISP PreScheme by expanding all derived syntax and performing a few simple transformations. Pure PreScheme programs are syntactically restricted, strongly typed Macro-Free PreScheme programs. The syntactic restrictions used to define Pure PreScheme imply that these programs will meet all of the run-time conditions for a valid VLISP PreScheme. Pure PreScheme programs are strongly typed, so no operator will be applied to data of the wrong type. `lambda` expressions in Pure PreScheme programs may occur only as initializers in top-level `letrec` bindings, or in the operator position of a procedure call. As a result, there is no need to represent closures at run-time. Finally, Pure PreScheme's syntax forbids non-tail-recursive procedure calls.

We present in more detail the various dialects of PreScheme used in the VLISP project and their design rationales. We also discuss the process by which the denotational semantics of PreScheme was derived.

1.3. The transformational front end and its denotational correctness

The transformational front end transforms VLISP PreScheme into Pure PreScheme. We give an overview of the transformation process and describe some of the transformations, including simplifiers, β-reduction, inlining, and lambda-lifting. We give an example to illustrate the operation of the compiler. Finally, we give an example of the correctness proof for a particular transform (letrec-lifting).

1.4. Compiler-Oriented Semantics

As discussed in Section 2, Pure PreScheme's semantics is inherited from the semantics of Macro-Free PreScheme. However, in order to justify the compiler presented in Section 5, we reframed the semantics to distinguish some special cases syntactically and split both the environment and the continuation into a compile-time and run-time components. We call this the *compiler semantics.* The compiler reorganizes these quantities so that all the static arguments are handled before all the dynamic ones. Thus the semantics can be thought of as taking all the compile-time data and producing a code-like object π that can be applied to the run-time data, including the local data stack (*cf* the discussion in [23, p. 253]). The code-like objects are built from combinators, which will play the role of instructions in the later operational semantics.

We present some pieces of the compiler-oriented semantics. For each valuation in the semantics, we present a formal specification showing the relation that must hold between the compiler-oriented valuation and the corresponding valuation in the original semantics. These specifications are based on the induction hypotheses of [6]. The proof that the compiler oriented semantics is the same as the original then becomes a straightforward structural induction. We present some sample cases in Section 4.6.

1.5. The Combinator Code Machine and the Compiler

The combinator machine manipulates tuples of rational trees built from the combinators. A machine state is the 4-tuple $\langle q, u, z, h\rangle$, where q is the code which operates on a runtime environment, u, a stack of stackable values, z, and a heap h. These rational trees are given two semantics: one denotational and one operational. The denotational semantics gives trees meanings in the same semantic domains as those used for the compiler-oriented semantics. Given the denotational semantics, the operational semantics is derived by performing β-conversion on the terms used in the compiler oriented valuations.

The Pure PreScheme compiler is similar to the compiler oriented semantics except the compiler produces syntax rather than a denotation. Given a program, the compiler is required to produce a code tree whose denotation is the same as that given by the compiler-oriented semantics. The compiler is produced by taking the definition of the valuations in the compiler-oriented semantics, and replacing domain transformations by operations that produce trees. An adequacy theorem relates the operational semantics of the combinator machine to the denotational semantics of the original program.

We conclude by discussing some of the choices made in the design of the compiler-oriented semantics and the combinator machine.

1.6. The Linear-Data Machine

The linear-data machine uses the same machine language as the combinator machine, but instead of manipulating rational trees, its programs manipulate representations of those trees in a linear data store. It also has a heap which corresponds to the heap of the combinator machine, except that it contains untagged quantities. The state of the linear-data machine is of the form $\langle q, up, sp, s, h\rangle$, where sp and up are pointers into the store representing the locations of the stack and the environment.

The representation of the stack and environment by these pointers is formalized. To aid in the intuition behind these formalisms, we first give an informal sketch of the representation. We then present the formal definitions. This is done by defining a *storage-layout relation* between states of the combinator-code machine and the linear-data machine. A pair of states is in this relation iff the linear-data-machine state represents the combinator-machine state. Some of the basic properties of this relation are sketched.

We define the operational semantics of the linear-data machine so that the linear-data machine simulates the behavior of the combinator machine. That is, if L_1 corresponds to C_1 and C_1 rewrites to C_2, then we want the linear-data machine to send L_1 to some state L_2 such that L_2 corresponds to C_2.

We prove that the linear-data machine satisfies the desired simulation theorem. The simulation property will need some refining, since the combinator machine keeps tag information and the linear-data machine does not. The proof relies on the global invariant that every validly compiled Pure PreScheme program runs in bounded control space.

1.7. The Stored-Program Machine

The stored-program machine is very similar to the linear-data machine, except that the code is represented in cells of a linear instruction store, much like the cells of the data store. The correspondence between states of the stored-program machine and states of the linear-data machine is defined, and a simulation theorem is proved.

1.8. The Linearizer and its Correctness

The assembler takes code for the combinator or linear-data machine and produces a state of the linear instruction store which corresponds to the original according to the definition in the preceding section. We therefore call it the *linearizer*. The major technical difficulty is the treatment of branches and joins, so that code is not duplicated. The proof is by induction on the structure of the combinator code.

1.9. Implementation

A compiler for PreScheme was developed in parallel with this specification. It generates code for the Motorola 68000. Representing the abstract machines with the 68000 requires a mapping similar to the ones just detailed. This section informally describes that mapping. Some performance results are presented.

1.10. Conclusions

We present some of the conclusions we have drawn from this effort. Some comparisons with other work are made. Some alternative design decisions are sketched. Directions for future work are suggested.

2. VLISP PreScheme

PreScheme is a dialect of Scheme intended for systems programming. PreScheme was carefully designed so that it syntactically looks like Scheme and has similar semantics. With a little care, PreScheme programs can be run and debugged as if they were ordinary Scheme programs. This section describes VLISP PreScheme and various related dialects used in the VLISP project.

VLISP PreScheme was inspired by Scheme48 PreScheme, but differs from it in that VLISP PreScheme has no user-defined syntax, macros, or compiler directives, and in that it provides a different set of standard procedures.

Because VLISP PreScheme is intended for systems programming, programs in the language are restricted so that they make as few assumptions as possible about the facilities available at run time. In particular, the computation model underlying VLISP PreScheme has the following properties:

- VLISP PreScheme programs manipulate data objects that fit in machine words. The type of each data object is an integer, a character, a boolean, a string, a port, a pointer to an integer, or a procedure (represented as a pointer). A PreScheme data object may be a full word, without room for run-time tags. Therefore no type predicates, like those of Scheme, are possible in the language. It becomes the compiler's responsibility to ensure statically that operators are never applied to data of the wrong type.

- A valid VLISP PreScheme program can run without creating closures at run time. Thus, if `p` is a pointer to a procedure, the free variables of that procedure must be allocatable at compile time. However, a VLISP PreScheme program may textually contain procedures with free variables that are lambda-bound. The compiler must transform these programs so that they meet the run-time restriction.

- Implementations of VLISP PreScheme are required to be tail-recursive, which means that iterative processes can be expressed by means of procedure calls. When the last action taken by a procedure is a call, a tail-recursive implementation is required to eliminate the control information of the calling procedure so that the order of space growth of iterative processes is constant. Such a call is said to be tail-recursive. The requirement that implementations be tail-recursive is inherited from Scheme [1].

- All procedure calls in a running VLISP PreScheme program must be tail-recursive. Thus, the VLISP PreScheme compiler used in the VLISP project can only be used to specify iterative processes. This restriction was made for historical reasons, as described in Section 2.5. One of the lessons learned was how to eliminate this restriction, and new versions of the compiler allow non-tail-recursive calls [**?**].

A PreScheme program is said to be *legal* only if the compiler can verify statically that these properties can be guaranteed at run time. Different implementations of PreScheme may accept different sets of programs, depending on how clever the compiler is in reasoning about these properties. The strategies used by the VLISP PreScheme compiler are discussed in Section 3.

2.1. VLISP PreScheme Syntax

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

The syntax of the VLISP PreScheme language is identical to the syntax of the language defined in the Scheme standard [1, Chapter 7] with the following exceptions:

- Every defined procedure takes a fixed number of arguments.

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

  (`define` ⟨variable⟩ ⟨expression⟩),

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

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

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

  When ⟨variable⟩ occurs in the operator position of a combination, compilers must replace it with ⟨expression⟩.
- No variable may be defined more than once.
- `letrec` is not a derived expression. The initializer for each variable bound by a `letrec` expression must be a `lambda` expression.
- Constants are restricted to integers, characters, booleans, and strings.
- Finally, a different set of primitive procedures is specified. These are listed in Figure 2

2.2. Macro-Free PreScheme

Macro-Free PreScheme programs result from VLISP PreScheme programs by expanding all derived syntax except the `case` expression, identifying which variables refer to standard procedures, and replacing single armed conditionals (`if` E E) with (`if` E E (`if` `#f` `#f`)) (while both these expressions give an unspecified result in the event that the test expression is false, the latter form allows later transformation rules to be applied more uniformly; see Section 3.1). A grammar for the resulting language is shown in Figure 3.

Not every string generated by the grammar in Figure 3 is a legal Macro-Free PreScheme program. A string is a legal program only if it is generated by the grammar and the compiler can verify that the program can be transformed to meet the runtime restrictions set forth at the beginning of this section. Macro-Free PreScheme programs must also satisfy the restriction that n-ary standard procedures must be used at one fixed arity.

VLISP PreScheme's complete formal denotational semantics is given in [21, Section 2.3]. Following the usual conventions, it assigns meanings to constructs of an abstract syntax. The abstract syntax is very similar to the

```
not, zero?, positive?, negative?, <, <=, =, >=, >, abs, +, -,
*, quotient, remainder, ashl, ashr, low-bits,
integer->char, char->integer, char=?, char<?, char<=, char>,
char>=,
make-vector, vector-ref, vector-set!, vector-byte-ref,
vector-byte-set!,
addr<, addr=, addr+, addr-, addr<=, addr>, addr>=,
addr->integer, integer->addr, addr->string,
port->integer, integer->port,
read-char, peek-char, eof-object?, write-char, write-int,
write, newline,
force-output, null-port?,
open-input-file, close-input-port, open-output-file,
close-output-port, current-input-port, current-output-port,
read-image, write-image,
bytes-per-word, useful-bits-per-word,
exit, err
```

Figure 2: Primitive Operators in VLISP PreScheme

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

$$
\begin{array}{rl}
\mathrm{P} ::= & (\texttt{define}\ \mathrm{I})^*\ \mathrm{E} \\
\mathrm{B} ::= & (\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}))^* \\
\mathrm{E} ::= & \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}\ \mathrm{E}^*) \mid (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}) \\
& \mid (\texttt{begin}\ \mathrm{E}^*\ \mathrm{E}) \mid (\texttt{letrec}\ (\mathrm{B})\ \mathrm{E}) \\
& \mid (\texttt{if}\ \mathrm{E}\ \mathrm{E}\ \mathrm{E}) \mid (\texttt{if}\ \texttt{\#f}\ \texttt{\#f}) \mid (\texttt{set!}\ \mathrm{I}\ \mathrm{E}) \\
& \mid (\mathrm{O}\ \mathrm{E}^*) \mid (\texttt{case}\ \mathrm{E}\ ((\mathrm{K})\ \mathrm{E})^*\ ((\mathrm{K})\ \mathrm{E}))
\end{array}
$$

Figure 3: Macro-Free PreScheme Syntax

syntax of Macro-Free PreScheme, so in the interest of space, it will be used as VLISP PreScheme's abstract syntax in what follows.

The semantics is presented in a form that very closely resembles Scheme's semantics [1, Appendix A]. It uses the same mathematical conventions, and many of the standard's definitions. The VLISP PreScheme formal semantics differs from Scheme's in a small number of ways:

- All variables must be defined before they are referenced or assigned and no variable may be defined more than once.
- Lambda bound variables are immutable, so a location need not be allocated for each actual parameter of an invoked procedure. Consequently, variables may be bound to either locations or expressed values.
- VLISP PreScheme procedure values do not have a location associated with them because there is no comparison operator for procedures.
- VLISP PreScheme `letrec` is no longer a derived expression, because the immutability of lambda bound variables makes Scheme's definition of `letrec` inapplicable. Instead, `letrec` is defined by an explicit fixed point.
- Procedures always return exactly one value, so expression continuations map a single expressed value to a command continuation.
- The domain of answers, unspecified in Scheme, is specified to be the coalesced sum of a one-point domain (representing a run-time error) and the flat domain of the integers. Thus there are exactly three outcomes of a VLISP PreScheme computation: non-termination, termination in an error, or an integer. The initial continuation is specified to be $(\lambda\epsilon\sigma.(\epsilon \mid N) \text{ in} A)$.
- Finally, memory is assumed to be infinite, so the storage allocator *new* always returns a location.

The following fragments of the VLISP PreScheme semantics illustrate some of these distinctions. For example, the semantics of variable reference follows. The first two lines contains the domain equations for denoted values and environments.

$$\begin{array}{l}\delta \in D = E + L \\ \rho \in U = \mathrm{Ide} \to D\end{array}$$

$$\begin{array}{l}\mathcal{E} : \mathrm{Exp} \to U \to K \to C \\ \mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa.\, \mathit{hold}(\rho\mathrm{I})(\lambda\epsilon.\, \epsilon = \mathit{undefined} \to \mathit{wrong}\ \text{“undefined variable”}, \kappa\epsilon)\end{array}$$

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

These semantics differ from Scheme's in that some variables are bound to expressed values instead of locations.

The semantics of a `lambda` expression provide another example.

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

Notice that the meaning of a `lambda` expression depends only on its environment. In contrast, the meaning of a `lambda` expression in Scheme depends on the store as well as its environment. This difference makes it easier to justify changing the time at which a PreScheme `lambda` expression is evaluated.

The semantics of a `letrec` expression are quite different from Scheme's.

$$\begin{array}{l}\mathcal{B} : \mathrm{Bnd} \to \mathrm{Ide}^* \to U \to E^* \to E^* \\ \mathcal{E} : \mathrm{Exp} \to U \to K \to C \\ \mathcal{B}[\![\,]\!] = \lambda\mathrm{I}^*\rho\epsilon^*.\, \langle\rangle \\ \mathcal{B}[\![(\mathrm{I}\ (\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}))\ \mathrm{B}]\!] = \\ \quad \lambda\mathrm{I}_0^*\rho\epsilon^*.\, \langle\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{E})]\!](\mathit{extends}\ \rho\mathrm{I}_0^*\epsilon^*)\rangle \S\, \mathcal{B}[\![\mathrm{B}]\!]\mathrm{I}_0^*\rho\epsilon^* \\ \mathcal{E}[\![(\texttt{letrec}\ (\mathrm{B})\ \mathrm{E})]\!] = \lambda\rho\kappa.\, \mathcal{E}[\![\mathrm{E}]\!](\mathit{extends}\ \rho(\mathcal{I}[\![\mathrm{B}]\!])(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{I}[\![\mathrm{B}]\!])\rho)))\kappa\end{array}$$

The meaning of the body of a `letrec` expression is given with an environment extended by values derived from the `letrec` bindings. As with `lambda` expressions, the values depend only on the environment and not the store.

The specification of the meaning of a procedure call is textually unchanged from the Scheme semantics. However, the semantics is somewhat different, because the domains are different. Scheme's expression continuations map a sequence of expressed values into a command continuation, but VLISP PreScheme maps a single expressed value into a command continuation.

$$\begin{array}{l}\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \\ \quad \lambda\rho\kappa.\, \mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle \mathbin{\S} \mathrm{E}^*)) \\ \qquad\qquad \rho \\ \qquad\qquad (\lambda\epsilon^*.\, ((\lambda\epsilon^*.\, \mathit{applicate}(\epsilon^* \downarrow 1)(\epsilon^* \dagger 1)\kappa) \\ \qquad\qquad\qquad (\mathit{unpermute}\ \epsilon^*)))\end{array}$$

The motivation for all these changes was efficiency for the Scheme48 version of PreScheme. However, these changes profoundly impact the ability to prove interesting properties of programs based on their semantics. In particular, these changes greatly facilitate proofs that certain program transformations preserve the meanings of programs. The utility of the changed semantics will be demonstrated later in the paper.

2.3. Static Semantics

Macro-Free PreScheme programs may be strongly typed. As in Standard ML [15], types are inferred, not declared, but unlike Standard ML, there are no polymorphic variables. All expressions are monomorphic except (`if` #`f` #`f`) and (`set`! I E). The static semantics is given as a conventional set of inference rules, except that when an expression is unconstrained by the rules, the expression is assigned the integer type [21, Section 3.3].

The standard results from the theory of type inference imply that the operators in a strongly typed Macro-Free PreScheme program will never be applied to data of the wrong type.

2.4. Pure PreScheme

Pure PreScheme programs are syntactically restricted, strongly typed Macro-Free PreScheme programs. The syntax is given in Figure 4. The syntactic restrictions used to define Pure PreScheme imply that these programs will meet all of the run-time conditions of a VLISP PreScheme program presented at the beginning of this section. Pure PreScheme programs are strongly typed, so no operator will be applied to data of the wrong type. `lambda` expressions in Pure PreScheme programs may occur only as initializers in top-level `letrec` bindings, or in the operator position of a procedure call. As a result, there is no need to represent closures at run-time. Finally, Pure PreScheme's syntax forbids non-tail-recursive procedure calls.

There is a further syntactic restriction. The first selection criteria of a `case` clause must be zero and the selection criteria for other clauses must be the successor of the previous clause's selection criterion. This restriction is

K ∈ Con	constants
I ∈ Ide	variables
O ∈ Op	primitive operators
S ∈ Smpl	simple expressions
T ∈ Tail	tail recursive expressions
E ∈ Exp	top level expressions
B ∈ Bnd	bindings
P ∈ Pgm	programs

P ::= (`define` I)* E
E ::= (`letrec` (B) T)
B ::= (I (`lambda` (I*) T))*
T ::= S | (S S*) | (`if` S T T) | (`case` S ((K) T)*)
  | (`begin` S* T) | ((`lambda` (I*) T) S*)
S ::= K | I | (O S*) | (`if` S S S) | (`if` `#f` `#f`) | (`set!` I S)
  | (`begin` S* S) | ((`lambda` (I*) S) S*) | (`case` S ((K) S)*)

Figure 4: Pure PreScheme Syntax

imposed to allow an `case` expression to be compiled into a computed goto.

Pure PreScheme's semantics are inherited from Macro-Free PreScheme's semantics, but they assume particular values for *permute* and *unpermute*, that is, the arguments of a call are always evaluated left-to-right, and then the operator is evaluated.

2.5. Challenges in the Design of VLISP PreScheme

The presentation of the PreScheme dialects used in the VLISP project has been made as logical and tidy as possible. This presentation, however, deprives the reader of an appreciation of the complexities of the design process. At the beginning of the VLISP project, we knew that the architecture of VLISP would be modeled after Scheme48, but we did not know how to build a verified compiler for a language like PreScheme. We also knew that the VLISP byte-code interpreter would mostly be an iterative process.

We began by concentrating on the run-time conditions for VLISP PreScheme programs presented at the beginning of this section. Pure PreScheme was the first language defined, and its syntax and semantics directly reflected the run-time conditions. When it was defined, Pure PreScheme's syntax and semantics were given independently; only later did we work out the simple embedding of that semantics in the more general semantics of VLISP PreScheme discussed here.

Unfortunately, the VLISP byte-code interpreter is not conveniently described using only tail-recursive calls. While the byte-code interpreter is a finite state machine, it would have been more natural to allow non-tail-recursive calls to the interpreter's garbage collector. The implementation of a purely iterative byte-code interpreter is described in [8, Section 1.2.2]. In retrospect, we suffered from attempting to produce a language too finely tuned to our specific task. When we found our analysis of the task was wrong, the language was not general enough to conveniently handle the task when properly specified.

3. The Transformational Front End

The Front End translates VLISP PreScheme into Pure PreScheme. The translation occurs in three phases.

Parse: Expands usages of derived syntax by rules consistent with those presented in the Scheme standard. In addition, the program's bound variables are changed so that no variable occurs both bound and free, and no variable is bound more than once. This implies that a variable bound in the program is changed if it is also bound to a standard procedure. Other syntactic checks are made. The result is a Macro-Free PreScheme program.

Apply transformation rules: Translates a Macro-free PreScheme program into the syntax of Pure PreScheme using meaning-refining transformations. More will be said about this phase later.

Type check: Ensures that a syntactically restricted Macro-Free PreScheme program is strongly typed, and is therefore a Pure PreScheme program. This phase implements Algorithm W. That algorithm's correctness was demonstrated by Robin Milner [14]. The unifier is based on a published program by Laurence Paulson [19, p. 381].

A compiler for for VLISP PreScheme cannot accept all syntactically correct programs, because many Macro-Free PreScheme programs cannot be translated into Pure PreScheme. Furthermore, the set of Macro-Free programs that can be translated by a particular compiler depends on the transformation rules used by that compiler.

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

For example, the following VLISP PreScheme program cannot be translated into Pure PreScheme. The procedure `loop` must be unwound if there are to be only tail-recursive procedure calls, but the compiler cannot predict how many times it should be unwound because `*x*` is a mutable variable.

```
(define *x* 2)
(define-integrable (loop x)
  (if (positive? x) (loop (- x 1)) x))
(some-hairy-command-which-modifies-*x*!)
(+ (loop *x*) 3)
```

In contrast, the following loop will be unwound at compile-time.

```
(define bytes-per-word 4)
(define-integrable (floor-log2 x a)
  (if (<= x 1)
      a
      (floor-log2 (quotient x 2) (+ 1 a))))
(define log-bytes-per-word
  (floor-log2 bytes-per-word 0))
log-bytes-per-word
```

The most complex and error-prone phase of the VLISP PreScheme Front End translates Macro-free PreScheme programs into a syntactically restricted form. The program is transformed by applying a set of rules. Each rule can be shown to be *meaning-refining*, in a sense to be defined below.

The selection and application of rules is performed by an intricate set of procedures. However, since each rule is meaning-refining, the output of the transformation process will be a program that is a refinement of the original, regardless of the choice of transformation rules. Therefore, the verification effort focused solely on the transformation rules. The consequences of this strategy are discussed in Section 3.3.

3.1. Transformation Rules

Each rule is a conditional rewrite rule. It has a pattern, a predicate, and a replacement. An expression matches a pattern if there is an assignment of pattern variables which makes the two expressions equal. The rewrite is performed if the matching expression satisfies the predicate.

To avoid name conflicts, the matching system works only on expressions that are α-converted. An expression is *α-converted* if no variable occurs both bound and free, and no variable is bound more than once. The output of the parsing phase is already α-converted, and the matching system α-converts the output of each transformation step.

A rule with pattern E_0 and replacement E_1 is written $\mathrm{E}_0 \Longrightarrow \mathrm{E}_1$, and its predicate is given in the text. If no predicate is given, the predicate is assumed to be true of any α-converted expression.

The Front End implements a variety of rules [21, Section 5.1.2]. There are roughly five categories of rules. One category contains rules specific to primitive operators. These rules help implement constant folding. For example, the rules associated with the plus operator follow.

$(+\ \mathrm{K}_0\ \mathrm{K}_1) \Longrightarrow \mathrm{K}_0 + \mathrm{K}_1$	constant folding
$(+\ 0\ \mathrm{E}) \Longrightarrow \mathrm{E}$	idempotency
$(+\ \mathrm{E}\ \mathrm{K}) \Longrightarrow (+\ \mathrm{K}\ \mathrm{E})$	commutativity
$(+\ \mathrm{E}_0\ (+\ \mathrm{E}_1\ \mathrm{E}_2)) \Longrightarrow (+\ (+\ \mathrm{E}_0\ \mathrm{E}_1)\ \mathrm{E}_2)$	associativity

The next category of rules focus on conditional expressions. As with the previous category of rules, the form of the rules are quite conventional, as the following example shows.

Definition 1 *The* `if`*-in-an-*`if`*'s-Consequence Rule.*

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

when E_0 *is side-effect-free.*

The predicate for this rule requires that a certain expression is side-effect-free. Side-effect-free expressions are defined so as to allow them to be recognized by a simple recursive function on syntax.

Definition 2 *The* side-effect-free *expressions are defined inductively by the following rules:*

- K *is side-effect-free.*
- I *is side-effect-free.*
- *If* O *is side-effect-free, and* $\mathrm{E}_1 \ldots \mathrm{E}_n$ *are side-effect-free, then so is* $(\mathrm{O}\ \mathrm{E}_1 \ldots \mathrm{E}_n)$.
- (`lambda` (I^*) E) *is side-effect-free.*
- *If* $\mathrm{E}_0, \ldots, \mathrm{E}_n$ *are side-effect-free, then so is* (`begin` $\mathrm{E}_0 \ldots \mathrm{E}_n$).

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

Intuitively, if an expression is side-effect-free, it returns a value without modifying the store, as will be shown in Theorem 1. This expression can be eliminated when its value is ignored.

There are several other recursively defined predicates on expression syntax which are used by a variety of rules. There is a predicate which determines if a variable is free in an expression. Another predicate is true of expressions which are side-effect-free and have values which do not depend on modifiable values. Yet another one recognizes expressions that do not modify the store until their last action.

The predicates have been carefully chosen so as to work with the rule application mechanism. For example, the definition of side-effect-free expressions could have included the following clause.

- If $\mathrm{E}_0, \ldots, \mathrm{E}_n$ are side-effect-free, then ((`lambda` ($\mathrm{I}_1 \ldots \mathrm{I}_n$) E_0) $\mathrm{E}_1 \ldots \mathrm{E}_n$) is side-effect-free.

This clause was eliminated not because its presence adversely affected proofs, but because it caused looping of the rule application mechanism.

The remaining categories of rules implement β-conversion, closure hoisting, and defined constant substitution. Many of the rules in these categories look unusual. The forms of these rules was intended to facilitate correctness proofs. For example, two distinct rules produce the effect of β-conversion. Closures are hoisted by the use of several other rules, including the rule below.

Definition 3 *The* `letrec` *Expression Merging Rule.*

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

The rules result in program transformations similar to those produced by other compilers [11, 12]. Consider the even-odd program given in Figure 5. It specifies a simple iterative process. Figures 6–11 show how this program might be translated into Pure PreScheme using transformations implemented in the Front End. Notice that unlike [11, 12], this compiler does not convert the program into continuation-passing style [2, 22].

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

Figure 5: Example VLISP PreScheme Program

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

Figure 6: Expand into Macro-Free PreScheme

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

Figure 7: Name Anonymous Lambda Expressions

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

Figure 8: Closure Hoisting

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

Figure 9: Constant Folding

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

Figure 10: Inline Non-Tail-Recursive Procedure Calls

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

Figure 11: Eliminate Unused Lambda Expressions and Initializers

3.2. Justification of the Rules

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

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

is transformed into a program which produces the answer 2!

This odd behavior is tolerated so as to allow constant propagation without performing a dependency analysis. In the above example, `1` is substituted for the occurrence of the immutable variable `one` even though *undefined* should have been substituted.

Definition 4 *A rule is* meaning-refining *if its application does not affect non-bottom computational results.*

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

The reason the proof shows that a rule is meaning-refining follows. Let P_1 be a program which results from the application of the rule to an expression in program P_0, and let $\mathcal{P}$ be the semantic function for programs. The proof implies that $\mathcal{P}[\![\mathrm{P}_0]\!] \sqsubseteq \mathcal{P}[\![\mathrm{P}_1]\!]$, because the semantic functions are compositional. Hence, if $\mathcal{P}[\![\mathrm{P}_0]\!]$ is not bottom, we have $\perp \sqsubset \mathcal{P}[\![\mathrm{P}_0]\!] \sqsubseteq \mathcal{P}[\![\mathrm{P}_1]\!]$. But since the answer domain is flat, $\mathcal{P}[\![\mathrm{P}_0]\!] = \mathcal{P}[\![\mathrm{P}_1]\!]$.

The proof of each rule usually requires the consideration of many details. Many rules are proved using structural induction on the syntax of Macro-Free PreScheme. Some rules have predicates which oblige certain expressions not modify the store, others require certain expressions not modify the store or reference mutable data. Some of the proofs in [21] are incomplete as only the cases that appeared to us to be "interesting" were selected for detailed consideration.

As a result, there is no proof which is representative of all of the others, However, the proofs of the `if`-in-an-`if`'s-Consequence Rule and the `letrec` Expression Merging Rule give some flavor of the other proofs.

3.2.1. *The `if`-in-an-`if`'s-Consequence Rule*

This rule has a predicate which requires one of its expressions to be side-effect-free. Side-effect-free expressions have the following property Π.

Definition 5 E *has property* Π *iff* $(\forall \rho, \sigma)(\exists \epsilon)(\forall \kappa)$

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

To understand this definition, ignore the case in which $\epsilon = \mathit{undefined}$. Notice the store given to the command continuation $\mathcal{E}[\![\mathrm{E}]\!]\rho\kappa$ on the left-hand-side is the same as the one given to the command continuation $\kappa\epsilon$ on the right-hand side. Thus, the meaning of E is either erroneous, or is the same as passing some value ϵ to κ, without changing the store. This characterizes the the meaning of side-effect-free expressions in a form useful for proofs.

Theorem 1 *If* E *is side-effect-free, then* E *has property* Π.

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

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

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

Notice ϵ is independent of κ so

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

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

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

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

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

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

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

otherwise

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

Use of the induction hypothesis verifies both alternatives. ▮

Theorem 2 *If* E_0 *is side-effect-free,*

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

Hence the `if`*-in-an-*`if`*'s-Consequence Rule is meaning-refining.*

Proof: By Theorem 1, for any ρ, κ, and σ, there exists an ϵ_0 such that

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

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

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

The proof of the other form of the rule is similar. ∎

3.2.2. The `letrec` *Expression Merging Rule*

The correctness proof of this rule is interesting because it demonstrates the use of fixed points in a denotational semantics to prove something that would be difficult using an operational semantics.

The proof makes use of the following lemmas which are presented without proofs. The *extends* auxiliary function is similar to Scheme's except that it associates identifiers with expressed values instead of locations. Recall from [1, Appendix A] that § denotes concatenation of sequences.

Lemma 1 *If* $\mathrm{I}_0^* \mathbin{\S} \mathrm{I}_1^*$ *is a sequence of distinct identifiers, then*

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

Lemma 2

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

The following theorem shows that the `letrec` expression merging rule is meaning-refining:

Theorem 3 *If $\mathcal{I}[\![\mathrm{B}_0\mathrm{B}_1]\!]$ is a sequence of distinct identifiers, then*

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

Proof: Let

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

Then we have

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

Because expressions are α-converted,

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

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

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

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

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

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

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

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

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

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

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

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

3.3. Selection and Proof of Rules

The most significant contribution of the work on the Front End is the identification of a collection of transformation rules that can both be verified relative to the formal semantics of the source language, and can also form the basis of a practical optimizing compiler.

It is difficult to quantify the quality of the rule set implemented in the Front End. The Front End implements many of the rules used in successful compilers, and so its performance should be comparable. We studied the result of translating the VLISP Byte-Code Interpreter into Pure PreScheme and could identify no other plausible general transformation that would improve the code. The general nature of each implemented rule suggests the rule set should be useful for a wide range of VLISP PreScheme programs.

There were two distinct prototypes of the Front End before the final version was implemented. Early guesses of the form the transformation rules were quite naïve, because we tried to use rules designed for unverified compilers. The first prototype explored various rule sets. It also became apparent that the mechanism used in the prototype to control rule application would not scale so as to allow the compilation of a program as large as the VLISP Byte-Code Interpreter. The second prototype had the control mechanism nearly right and its rule set seemed to generate reasonable code. Attempts to prove the correctness of its rules failed miserably. The exercise led to changes in both the rules and VLISP PreScheme's semantics. The prototypes gave us extremely valuable information and made the identification of the final, finely tuned rule set possible. Our motto is prototype but verify!

There are several unverified aspects of the Front End. The algorithm used to translate VLISP PreScheme into Macro-Free PreScheme was not studied rigorously. The lack of rigor is of little concern because the process is clearly specified in the Scheme Report section on derived expression types [1, Section 7.4].

The most unsatisfying part of the Front End is the fact that the algorithms describing the control procedures are unverified. All parts of the Front End were written in a simple and direct style. We studiously avoided tricky or obscure coding practices, partially so that someone reading the code which applies transformation rules will easily conclude that the control procedures only cause changes to the program which are consistent with the set of rules, but there is no formal verification of this fact. Further work is needed to design a formalism that when given a rule set produces an efficient and verified rule application mechanism.

4. Compiler-Oriented Semantics

As discussed in Section 2, Pure PreScheme's semantics is inherited from the semantics of Macro-Free PreScheme. However, in order to justify the compiler presented in Section 5, we reframed the semantics to distinguish some special cases syntactically and split both the environment and the continuation into a compile-time and run-time components. We call this the *compiler semantics.*

The architecture of this semantics is motivated by a binding-time analysis of the semantics:

- The program is always compile-time data (that is, it is available at compile time).
- The environment is split into a symbol table γ (available at compile time), and a run-time environment or display μ (available at run-time).
- The continuation is also split into static (compile-time) and dynamic (run-time) components.
- The heap h is always run-time data.

In addition, the compiler semantics introduces a data stack for the values of subexpressions. This stack is "partially static" in that its size is statically determinable, but its contents are dynamic.

The compiler semantics reorganizes these quantities so that all the static arguments are handled before all the dynamic ones. Thus the semantics can be thought of as taking all the compile-time data and producing a code-like object π that can be applied to the run-time data, including the local data stack (*cf* the discussion in [23, p. 253]).

In addition, the semantics distinguishes between simple expressions and tail-recursive expressions, following the grammar of Figure 4. An easy proof via structural induction shows that valuations of tail-recursive expressions always use the initial continuation κ_0, a fact exploited in the new semantics.

The compiler-oriented semantics also distinguishes commands (expressions executed only for effect) from expressions executed for their value. An easy proof via structural induction shows that valuations of commands always use a continuation which ignores its value. This separation allows the compiler to know that the value of a simple expression can be ignored. In principle, we could have preserved the semantic principle of "one nonterminal, one valuation" by creating a separate syntactic category for commands, but the programming principle of orthogonality ("a phrase should be usable anywhere it is meaningful") won out, since in Scheme in any (simple) expression can be evaluated for effect only.

As noted above, the new semantics relies on the usual division of environments into a compile-time and a run-time component. We introduce a new domain D_c of compile-time denotations (lexical addresses), and represent an environment $\rho : Ide \rightarrow D$ by a symbol table $\gamma : Ide \rightarrow D_c$ and a runtime environment $\mu : D_c \rightarrow D$ such that $\rho = (\mu \circ \gamma)$. 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:

$$\mathit{extends}(\mu \circ \gamma)\,\mathrm{L}^*\epsilon^* = (\mathit{extends}_{rl}\ \mu\,\epsilon^*) \circ (\mathit{extends}_{cl}\ \gamma\,\mathrm{L}^*)$$

with similar conditions for global variables.[1] Because the scoping structure of Pure PreScheme programs is very simple, our semantics uses integers for lexical addresses.

The domain of code-like objects π is

$$Q = U_r \rightarrow E^* \rightarrow S \rightarrow A$$

where U_r is the domain of run-time environments (displays), E is the domain of expressed values, S is the domain of heaps, and A is the domain of answers. These elements are the compile-time component of a continuation. In general, the elements of Q will be built out of certain combinators, which we call *instructions*. The justification for this terminology will become evident in Section 5.

[1]For historical reasons, this semantics assumes that the identifiers have been partitioned into local and global identifiers. Global identifiers can only be used for defined variables, and local identifiers can only be used for `letrec` and `lambda` bound variables.

We next present some pieces used to construct the compiler oriented semantics, $\mathcal{CP}[\![\mathrm{P}]\!]$. For each valuation we give the following data:

1. an informal specification of the valuation
2. the domain of the valuation
3. the induction hypothesis (formal specification) for the valuation
4. the rules for each production, and last
5. The semantics of typical instructions introduced in those rules

The proof that the compiler oriented semantics is the same as the original, i.e., $\mathcal{CP}[\![\mathrm{P}]\!] = \mathcal{P}[\![\mathrm{P}]\!]$, then becomes a straightforward structural induction. We will present some sample cases in Section 4.6.

4.1. Simple Expressions

Compile Simple Expressions. Generates code that evaluates the expression and pushes the value onto the runtime stack.[2]

type: $\mathcal{CSE} : \mathrm{Smpl} \to U_c \to Q \to Q$
specification: $(\mathcal{CSE}[\![\mathrm{S}]\!]\gamma\pi)\mu\zeta = \mathcal{E}[\![\mathrm{S}]\!](\mu \circ \gamma)(\lambda\epsilon.\, \pi\mu(\epsilon :: \zeta))$

The induction hypothesis states that running a simple expression compiled using symbol table γ with code to follow π in runtime environment μ with runtime stack ζ is the same as the semantics for that expression 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.

Note that $\mathcal{CSE}$ takes all the static arguments before all the dynamic arguments, as discussed above.

definition:
$\mathcal{CSE}[\![\mathrm{L}]\!] = \lambda\gamma\pi.\, \mathit{fetch}_l\ (\gamma\ \mathrm{L})\ \pi$
$\mathcal{CSE}[\![(\texttt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!] = \lambda\gamma\pi.\, \mathcal{CSE}[\![\mathrm{S}_0]\!]\gamma(\mathit{brf}\,(\mathcal{CSE}[\![\mathrm{S}_1]\!]\gamma(\mathit{label}\ \pi))$
$\qquad\qquad(\mathcal{CSE}[\![\mathrm{S}_2]\!]\gamma(\mathit{goto}\ \pi)))$
$\mathcal{CSE}[\![(\texttt{set!}\ \mathrm{G}\ \mathrm{S})]\!] = \lambda\gamma\pi.\, \mathcal{CSE}[\![\mathrm{S}]\!]\gamma(\mathit{update\text{-}store}\ (\gamma\ \mathrm{G})\ \pi)$
$\mathcal{CSE}[\![(\mathrm{O}\ \mathrm{S}^*)]\!] = \lambda\gamma\pi.\, \mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma(\mathit{prim\text{-}apply}\ \#\mathrm{S}^*\ \mathcal{O}[\![\mathrm{O}]\!]\ \pi)$

[2]The runtime stack in this semantics contains expressed values. The actual compiler allows environments as well [18]. The use of ζ instead of ϵ^* reminds us of the fact that the presentation has been simplified.

These clauses introduce six new instructions.

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

$$\begin{aligned}
&\mathit{brf} : \; Q \to Q \to Q \\
&\mathit{brf} = \lambda\pi_0\pi_1\lambda\mu\zeta.\,\mathit{truish}((\zeta \downarrow 1)) \to (\pi_0\mu(\zeta \dagger 1)),\; (\pi_1\mu(\zeta \dagger 1))
\end{aligned}$$

$$\begin{aligned}
&\mathit{label} : \; Q \to Q \\
&\mathit{label} = \lambda\pi\mu\zeta.\pi\mu\zeta
\end{aligned}$$

$$\begin{aligned}
&\mathit{goto} : \; Q \to Q \\
&\mathit{goto} = \lambda\pi\mu\zeta.\pi\mu\zeta
\end{aligned}$$

The *label* and *goto* instructions are used as directives to the linearizer (Section 8).

$$\begin{aligned}
&\mathit{update\text{-}store} : \; D_c \to Q \to Q \\
&\mathit{update\text{-}store} = \lambda\iota\pi.\,\lambda\mu\zeta.(\mu\;\iota) \;\mathrm{E}\; L \to \\
&\qquad (\mathit{assign}\;((\mu\;\iota) \mid L)\;(\zeta \downarrow 1)\;(\pi\mu\zeta)), \\
&\qquad (\mathit{wrong}\text{ “Can't assign to a local variable”})
\end{aligned}$$

$$\begin{aligned}
&\mathit{prim\text{-}apply} : \; N \to P \to Q \to Q \\
&\mathit{prim\text{-}apply} = \lambda\nu\upsilon\pi.\,\lambda\mu\zeta.(\mathit{apply\text{-}primitive}\;\upsilon\;(\mathit{take\text{-}first}\;\nu\;\zeta) \\
&\qquad (\lambda\epsilon.\,\pi\mu(\epsilon :: (\mathit{pop\text{-}first}\;\nu\;\zeta))))
\end{aligned}$$

4.2. Simple Expression Sequences

Compile Simple Expression Sequences. Generates code that evaluates the sequence of simple expressions and pushes the sequence of values onto the runtime stack.

type: $\mathcal{CSE}^* : \; \mathrm{Smpl}^* \to U_c \to Q \to Q$
specification: $(\mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma\pi)\mu\zeta = \mathcal{E}^*[\![\mathrm{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^*.\,\pi\mu(\epsilon^* \S\, \zeta))$

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.

$$\begin{aligned}
&\mathcal{CSE}^*[\![\;]\!] = \lambda\gamma\pi.\,\pi \\
&\mathcal{CSE}^*[\![\mathrm{S}\;\mathrm{S}^*]\!] = \lambda\gamma\pi.\,\mathcal{CSE}[\![\mathrm{S}]\!]\gamma(\mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma\pi)
\end{aligned}$$

4.3. Simple Commands

Compile Simple Commands. Generates code that evaluates the expression and but does not push the value onto the runtime stack.

type: $\mathcal{CSC} : \text{Smpl} \rightarrow U_c \rightarrow Q \rightarrow Q$
specification: $(\mathcal{CSC}[\![\text{S}]\!]\gamma\pi)\mu\zeta = \mathcal{E}[\![\text{S}]\!](\mu \circ \gamma)(\lambda\epsilon.\,\pi\mu\zeta)$

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).

definition:
$\mathcal{CSC}[\![\text{L}]\!] = \lambda\gamma\pi.\,\pi$
$\mathcal{CSC}[\![\texttt{(if } \text{S}_0\ \text{S}_1\ \text{S}_2\texttt{)}]\!] = \lambda\gamma\pi.\,\mathcal{CSE}[\![\text{S}_0]\!]\gamma(\mathit{brf}\,(\mathcal{CSC}[\![\text{S}_1]\!]\gamma\pi)\ (\mathcal{CSC}[\![\text{S}_2]\!]\gamma\pi))$
$\mathcal{CSC}[\![\texttt{(set! } \text{G S}\texttt{)}]\!] = \lambda\gamma\pi.\,\mathcal{CSE}[\![\text{S}]\!]\gamma(\mathit{update\text{-}store}_i\ (\gamma\ \text{G})\ \pi)$
$\mathcal{CSC}[\![\text{(O S}^*\text{)}]\!] = \lambda\gamma\pi.\,\mathcal{CSE}^*[\![\text{S}^*]\!]\gamma(\mathit{prim\text{-}apply}_i\ \#S^*\ \mathcal{O}[\![\text{O}]\!]\ \pi)$

This valuation introduces two new instructions, *update-store*$_i$ and *prim-apply*$_i$, which are like their expression counterparts but do not push a value onto the runtime stack.

4.4. Tail-Recursive Expressions

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

type: $\mathcal{CT} : \text{Tre} \rightarrow U_c \rightarrow Q$
specification: $(\mathcal{CT}[\![\text{T}]\!]\gamma)\mu\langle\rangle = \mathcal{E}[\![\text{T}]\!](\mu \circ \gamma)\kappa_0$

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.

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

$\mathcal{CT}[\![$`(begin` S* T)$]\!] = \lambda\gamma.\,\mathcal{CSC}[\![\mathrm{S}^*]\!]\gamma(\mathcal{CT}[\![\mathrm{T}]\!]\gamma)$
$\mathcal{CT}[\![$`((lambda` (L*) T) S*)$]\!] = \lambda\gamma.\mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma$
$\qquad(\mathit{add\text{-}to\text{-}env}\,(\mathcal{CT}[\![\mathrm{T}]\!](\mathit{extends}_{c\,l}\ \gamma\,\mathrm{L}^*)))$
$\mathcal{CT}[\![(\mathrm{S}\ \mathrm{S}^*)]\!] = \lambda\gamma.\,\mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma(\mathcal{CSE}[\![\mathrm{S}]\!]\gamma(\mathit{tail\text{-}call}))$

These clauses introduce two more new instructions, *add-to-env* and *tail-call*:

$\mathit{add\text{-}to\text{-}env}:\ Q \to Q$
$\mathit{add\text{-}to\text{-}env} = \lambda\pi.\,\lambda\mu\zeta.\,\pi(\mathit{extends}_{r\,l}\ \mu\ \zeta)\langle\rangle$

$\mathit{tail\text{-}call}:\ Q$
$\mathit{tail\text{-}call} = \lambda\mu\zeta.\,(\mathit{tail\text{-}apply}\ (\zeta\downarrow 1)\ (\zeta\dagger 1))$

where *tail-apply* is defined by

$\mathit{tail\text{-}apply}:\ E \to E^* \to C$
$\mathit{tail\text{-}apply} = \lambda\epsilon\epsilon^*.\,\epsilon\ \mathrm{E}\ F \to (\epsilon \mid F)\epsilon^*\kappa_0,\ (\mathit{wrong}$ "Non-function to apply"$)$

4.5. Recursive Procedures

Compile Recursive Declarations. Takes a set of local procedure declarations and generates code that extends the run-time environment by making the indicated mutually-recursive declarations.

type: $\mathcal{CE}:\ \mathrm{Exp} \to U_c \to Q$
specification:
$(\mathcal{CE}[\![$`(letrec` (B) T)$]\!]\gamma)\mu\langle\rangle = \mathcal{E}[\![$`(letrec` (B) T)$]\!](\mu\circ\gamma)\kappa_0$
definition:
$\mathcal{CE}[\![$`(letrec` (B) T)$]\!]\gamma = \mathit{closerecs}\,(\mathcal{CB}[\![\mathrm{B}]\!](\mathcal{GP}[\![\mathrm{B}]\!])\gamma)(\mathcal{CT}[\![\mathrm{T}]\!]\gamma)$

The induction hypothesis says that executing the code generated by the compiler is equivalent to executing the code for T in an appropriate recursively-extended environment.

The auxiliary valuation $\mathcal{CB}$ produces a sequence of code segments which will turn into the bodies of the procedures.

type: $\mathcal{CB}:\ \mathrm{Bnd} \to \mathrm{LocIde}^* \to U_c \to Q^*$
specification:
$\mathit{closerecs}\,(\mathcal{CB}[\![\mathrm{B}]\!](\mathcal{GP}[\![\mathrm{B}]\!])\gamma)\pi\mu = \pi(\mathit{extends}_{rl}\,\mu(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{GP}[\![\mathrm{B}]\!])(\mu\circ\gamma))))$
definition:
$\mathcal{CB}[\![\ ]\!] = \lambda\mathrm{L}^*\gamma.\,\langle\rangle$
$\mathcal{CB}[\![$(L `(lambda`(L*) T)) B$]\!] =$
$\quad\lambda\mathrm{L}_0^*\gamma.\,\mathcal{CT}[\![\mathrm{T}]\!](\mathit{extends}_{cl}(\mathit{extends}_{cl}\,\gamma\mathrm{L}_0^*)\mathrm{L}^*) :: (\mathcal{CB}[\![\mathrm{B}]\!]\mathrm{L}_0^*\gamma)$

This valuation introduces a single new instruction *closerecs*, defined by:

$$\mathit{closerecs}\,\pi^*\pi = \lambda\mu\zeta\,.\,\pi(\mathit{fix}(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu(\mathit{map}(\mathit{close}_r\,\mu')\pi^*)))\zeta$$

where

$$\mathit{close}_r = \lambda\mu\pi.\,(\lambda\epsilon^*.\,\pi(\mathit{extends}_{rl}\,\mu\epsilon^*)\langle\rangle)\,\mathrm{in}\,E$$

4.6. Correctness of the Compiler-Oriented Semantics

The main statement of correctness of the compiler is:

Theorem 4 (Correctness of the compiler oriented semantics) *For any program* P, $\mathcal{P}[\![\mathrm{P}]\!] = \mathcal{CP}[\![\mathrm{P}]\!]$.

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. There is one induction step for each clause in the definitions. Most of the induction steps are easy calculations: the only exceptions are the ones for recursive declarations.

Here we include a few sample cases to illustrate the calculations. We show a few simple cases and the cases for recursive declarations in more detail.

We begin with two cases for simple expressions. The induction hypothesis to be verified is

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

The following three cases are typical.

$(\mathcal{CSE}[\![\mathrm{L}]\!]\gamma\pi)\mu\zeta$
$=$ [by definition of $\mathcal{CSE}$]
$(\mathit{fetch}_l\;(\gamma\,\mathrm{L})\;\pi)\mu\zeta$
$=$ [by definition of fetch_l]
$((\mu\circ\gamma)\mathrm{L})\;\mathrm{E}\;E \to \pi\mu((((\mu\circ\gamma)\mathrm{L}) \mid E) :: \zeta),$
$\qquad(\mathit{wrong}$ “Local variable given storage.”$)$
$=$ [by definition of $\mathcal{E}$]
$\mathcal{E}[\![\mathrm{L}]\!](\mu\circ\gamma)(\lambda\epsilon.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![(\mathtt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]\gamma\pi)\mu\zeta$
$=$ [by definition of $\mathcal{CSE}$]

$(\mathcal{CSE}[\![\mathrm{S}_0]\!]\gamma(\mathit{brf}\,(\mathcal{CSE}[\![\mathrm{S}_1]\!]\gamma(\mathit{label}\,\pi))\ (\mathcal{CSE}[\![\mathrm{S}_2]\!]\gamma(\mathit{goto}\ \pi))))\mu\zeta$
$=$ [by induction hypothesis at S_0]
$\mathcal{E}[\![\mathrm{S}_0]\!](\mu\circ\gamma)(\lambda\epsilon.\,(\mathit{brf}\,(\mathcal{CSE}[\![\mathrm{S}_1]\!]\gamma(\mathit{label}\ \pi))\ (\mathcal{CSE}[\![\mathrm{S}_2]\!]\gamma(\mathit{goto}\ \pi)))\mu(\epsilon :: \zeta))$
$=$ [by definitions of *brf* , *label*, *goto*]
$\mathcal{E}[\![\mathrm{S}_0]\!](\mu\circ\gamma)(\lambda\epsilon.\,(\mathit{truish}(\epsilon) \to ((\mathcal{CSE}[\![\mathrm{S}_1]\!]\gamma\pi)\mu\zeta),$
$((\mathcal{CSE}[\![\mathrm{S}_2]\!]\gamma\pi)\mu\zeta)))$
$=$ [by induction hypothesis at S_1, S_2]
$\mathcal{E}[\![\mathrm{S}_0]\!](\mu\circ\gamma)(\lambda\epsilon.\,(\mathit{truish}(\epsilon) \to \mathcal{E}[\![\mathrm{S}_1]\!](\mu\circ\gamma)(\lambda\epsilon.\,\pi\mu(\epsilon :: \gamma))$
$\mathcal{E}[\![\mathrm{S}_2]\!](\mu\circ\gamma)(\lambda\epsilon.\,\pi\mu(\epsilon :: \zeta))))$
$=$ [by definition of $\mathcal{E}$]
$\mathcal{E}[\![(\texttt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!](\mu\circ\gamma)(\lambda\epsilon.\,\pi\mu(\epsilon :: \zeta))$

$(\mathcal{CSE}[\![(\texttt{set!}\ \mathrm{G}\ \mathrm{S})]\!]\gamma\pi)\mu\zeta$
$=$ [by definition of $\mathcal{CSE}$]
$(\mathcal{CSE}[\![\mathrm{S}]\!]\gamma(\mathit{update\text{-}store}\ (\gamma\ \mathrm{G})\ \pi))\mu\zeta$
$=$ [by induction hypothesis at S]
$\mathcal{E}[\![\mathrm{S}]\!](\mu\circ\gamma)(\lambda\epsilon.\,(\mathit{update\text{-}store}\ (\gamma\ \mathrm{G})\ \pi)\mu(\epsilon :: \zeta))$
$=$ [by definition of *update-store*]
$\mathcal{E}[\![\mathrm{S}]\!](\mu\circ\gamma)(\lambda\epsilon.\,((\mu\circ\gamma)\mathrm{G})\ \mathrm{E}\ L \to (\mathit{assign}\,(((\mu\circ\gamma)\mathrm{G})\mid L)\ \epsilon\ (\pi\mu(\epsilon :: \zeta))),$
$(\mathit{wrong}$ "Can't assign to a local variable"$))$
$=$ [by definition of $\mathcal{E}$]
$\mathcal{E}[\![(\texttt{set!}\ \mathrm{G}\ \mathrm{S})]\!](\mu\circ\gamma)(\lambda\epsilon.\,\pi\mu(\epsilon :: \zeta))$

Several valuations recur on sequences of syntactic elements; $\mathcal{CSE}^*$ is typical. The induction hypothesis is

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

There are two productions (clauses), which are proved as follows:

$(\mathcal{CSE}^*[\![\ ]\!]\gamma\pi)\mu\zeta$
$=$ [by definition of $\mathcal{CSE}^*$]
$\pi\mu\zeta$
$=$ [by definition of $\mathcal{E}^*$]
$\mathcal{E}^*[\![\ ]\!](\mu\circ\gamma)(\lambda\epsilon^*.\,\pi\mu(\epsilon^*\ \S\ \zeta))$

$(\mathcal{CSE}^*[\![\mathrm{S}\ \mathrm{S}^*]\!]\gamma\pi)\mu\zeta$
$=$ [by definition of $\mathcal{CSE}^*$]
$(\mathcal{CSE}[\![\mathrm{S}]\!]\gamma(\mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma\pi))\mu\zeta$
$=$ [by induction hypothesis at S]

$\mathcal{E}[\![\mathrm{S}]\!](\mu \circ \gamma)(\lambda\epsilon.\,(\mathcal{CSE}^*[\![\mathrm{S}^*]\!]\gamma\pi)\mu(\epsilon :: \zeta))$
$=$ [by induction hypothesis at S^*]
$\mathcal{E}[\![\mathrm{S}]\!](\mu \circ \gamma)(\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^*.\,\pi\mu(\epsilon^* \,\S\, (\epsilon :: \zeta))))$
$=$ [by definition of $\mathcal{E}^*$]
$\mathcal{E}^*[\![\mathrm{S}\ \mathrm{S}^*]\!](\mu \circ \gamma)(\lambda\epsilon^*.\,\pi\mu(\epsilon^* \,\S\, \zeta))$

The only difficult proof is the one for recursive procedure declarations. The induction hypothesis is

specification:
$\mathit{closerecs}\,(\mathcal{CB}[\![\mathrm{B}]\!](\mathcal{GP}[\![\mathrm{B}]\!])\gamma)\pi\mu = \pi(\mathit{extends}_{rl}\,\mu(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!](\mathcal{GP}[\![\mathrm{B}]\!])(\mu \circ \gamma))))$

This calculation is complicated by the necessity to reason both about sequences of values and about fixed points. To simplify this calculation, we write $\langle e \ldots \rangle$ for a sequence of values whose typical element is given by e, and "X . . . " for a sequence of syntactic elements whose typical element is given by X.

To verify the induction hypothesis, let

$$\begin{aligned} \mathrm{B} &= (\mathrm{L}\ (\texttt{lambda}\ (\mathrm{L}^*)\ \mathrm{T}))\ldots \\ \mathrm{L}_0^* &= \mathcal{GP}[\![\mathrm{B}]\!] \\ \gamma' &= \mathit{extends}_{cl}\,\gamma\mathrm{L}_0^* \end{aligned}$$

Then, calculating from the left-hand side of the induction hypothesis, we get:

$\mathit{closerecs}\,(\mathcal{CB}[\![\mathrm{B}]\!]\mathrm{L}_0^*\gamma)\pi\mu$
$=$ [by definition of $\mathit{closerecs}$]
$\pi(\mathit{fix}\,(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu(\mathit{map}\ (\mathit{close}\ _r\,\mu')(\mathcal{CB}[\![\mathrm{B}]\!]\mathrm{L}_0^*\gamma))))$
$=$ [by definition of $\mathcal{CB}$]
$\pi(\mathit{fix}\,(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu(\mathit{map}\ (\mathit{close}\ _r\,\mu')\langle\mathcal{CT}[\![\mathrm{T}]\!](\mathit{extends}_{cl}\,\gamma'\mathrm{L}^*)\ldots\rangle)))$
$=$ [by definition of map and $\mathit{close}\ _r$]
$\pi(\mathit{fix}\,(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu\langle(\lambda\epsilon^*.\,\mathcal{CT}[\![\mathrm{T}]\!](\mathit{extends}_{cl}\,\gamma'\mathrm{L}^*) \circ (\mathit{extends}_{rl}\,\mu'\epsilon^*)\langle\rangle)\ \mathrm{in}\,E$
$\ldots\rangle))$
$=$ [by induction hypothesis at T]
$\pi(\mathit{fix}\,(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu\langle(\lambda\epsilon^*.\,\mathcal{E}[\![\mathrm{T}]\!]((\mathit{extends}_{cl}\,\gamma'\mathrm{L}^*) \circ (\mathit{extends}_{rl}\,\mu'\epsilon^*))\kappa_0)\ \mathrm{in}\,E$
$\ldots\rangle))$
$=$ [by consistency of $\mathit{extends}_{rl}$, $\mathit{extends}_{cl}$]
$\pi(\mathit{fix}\,(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu\langle(\lambda\epsilon^*.\,\mathcal{E}[\![\mathrm{T}]\!](\mathit{extends}(\mu' \circ \gamma')\mathrm{L}^*\epsilon^*)\kappa_0)\ \mathrm{in}\,E\ldots\rangle))$
$=$ [by definition of $\mathcal{L}$]
$\pi(\mathit{fix}\,(\lambda\mu'.\,\mathit{extends}_{rl}\,\mu\langle\mathcal{L}[\![(\texttt{lambda}\ (\mathrm{L}^*)\ \mathrm{T})]\!](\mu' \circ \gamma')\ldots\rangle))$

Working from the right-hand side, we calculate:

$\pi(\mathit{extends}_{rl}\,\mu(\mathit{fix}(\mathcal{B}[\![\mathrm{B}]\!]\mathrm{L}_0^*(\mu\circ\gamma))))$
$=\quad$ [by definition of $\mathcal{B}$]
$\pi(\mathit{extends}_{rl}\,\mu(\mathit{fix}(\lambda\epsilon^*.\,\langle\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{L}^*)\ \mathrm{T})]\!](\mathit{extends}(\mu\circ\gamma)\mathrm{L}_0^*\epsilon^*)\ldots\rangle)))$

Let $f=\lambda\mu'.\,\mathit{extends}_{rl}\,\mu\langle\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{L}^*)\ \mathrm{T})]\!](\mu'\circ\gamma')\ldots\rangle$
and $g=\lambda\epsilon^*.\,\langle\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{L}^*)\ \mathrm{T})]\!](\mathit{extends}(\mu\circ\gamma)\mathrm{L}_0^*\epsilon^*)\ldots\rangle$

The theorem is proved once we show that:

$$\mathit{fix}\,f=\mathit{extends}_{rl}\,\mu(\mathit{fix}\,g)$$

The right-hand side is a fixed point of f because:

$$\begin{aligned}
f(\mathit{extends}_{rl}\,\mu\epsilon^*) &= \mathit{extends}_{rl}\,\mu(g\epsilon^*) && \text{[by substitution]}\\
f(\mathit{extends}_{rl}\,\mu(\mathit{fix}\,g)) &= \mathit{extends}_{rl}\,\mu(g(\mathit{fix}\,g)) && \text{[by use of previous line]}\\
&= \mathit{extends}_{rl}\,\mu(\mathit{fix}\,g)
\end{aligned}$$

The right-hand side is a least fixed point of f because:

$$\mathit{extends}_{rl}\,\mu(g^n\bot)\sqsubseteq f^{n+1}\bot$$

The proof is by induction on n.

$$\begin{aligned}
\mathit{extends}_{rl}\,\mu(g^0\bot) &= \mathit{extends}_{rl}\,\mu\bot\\
&\sqsubseteq \mathit{extends}_{rl}\,\mu\langle\mathcal{L}[\![\texttt{(lambda}\ (\mathrm{L}^*)\ \mathrm{T})]\!](\bot\circ\gamma')\ldots\rangle\\
&= f^1\bot
\end{aligned}$$

$$\begin{aligned}
\mathit{extends}_{rl}\,\mu(g^{n+1}\bot) &= \mathit{extends}_{rl}\,\mu(g(g^n\bot))\\
&= f(\mathit{extends}_{rl}\,\mu(g^n\bot))\\
&\sqsubseteq f(f^{n+1}\bot) && \text{[by induction hypothesis]}\\
&= f^{n+2}\bot
\end{aligned}$$

5. The Combinator Machine and the Compiler

5.1. The Combinator Machine

The combinator machine manipulates tuples of rational trees. Rational (or regular) trees [7] are finite or infinite trees that have only finitely many distinct subtrees. Such a tree can always be represented as a finite graph that unwinds to the possibly-infinite tree.

A machine state is the 4-tuple $\langle q,u,z,h\rangle$, where q is the code which operates on a runtime environment, u, a stack of stackable values, z, and a heap h. A portion of the grammar for permissible states $\langle q,u,z,h\rangle$ is

$$
\begin{array}{rcl}
q & ::= & (\mathit{halt}) \mid (\mathit{goto}\ q) \mid (\mathit{label}\ q) \mid (\mathit{fetch}_l\ i\, q) \mid (\mathit{brf}\ q_1\, q_2) \\
 & \mid & (\mathit{save\text{-}env}\ q) \mid (\mathit{restore\text{-}env}\ q) \mid (\mathit{update\text{-}store}\ i\, q) \\
 & \mid & (\mathit{prim\text{-}apply}\, n\, p\, q) \mid (\mathit{update\text{-}store}_i\ i\, q) \mid (\mathit{prim\text{-}apply}_i\ n\, p\, q) \\
 & \mid & (\mathit{add\text{-}to\text{-}env}\ n\, q) \mid (\mathit{add\text{-}to\text{-}env}_g\ q) \\
 & \mid & (\mathit{closerecs}\, (q_1 \ldots q_n)\, q) \mid (\mathit{tail\text{-}call}) \\
 & & \\
u & ::= & \mathit{emptydisplay} \mid (\mathit{extends}_{rl}\, u (v_1 \ldots v_n)) \mid (\mathit{extends}_{rg}\, u a) \\
 & & \\
z & ::= & v^* \\
 & & \\
v & ::= & \langle \mathbf{proc}, \langle q, u\rangle\rangle \mid \langle \mathbf{env}, u\rangle \mid \langle \mathbf{int}, n\rangle \mid \langle \mathbf{bool}, b\rangle \\
 & \mid & \langle \mathbf{char}, c\rangle \mid \langle \mathbf{string}, s\rangle \mid \langle \mathbf{hptr}, n\rangle \mid \langle \mathbf{quote}, d\rangle \\
 & & \\
a & ::= & \langle \mathbf{hptr}, n\rangle
\end{array}
$$

Figure 12: Grammar for states of the combinator machine

shown in Figure 12. The combinator machine has a total of 20 instructions, of which 15 are shown. The permissible states of the machine are those configurations $\langle q, u, z, h\rangle$ where q, u, z, and h are elements of the greatest fixed point defined by the operators of the grammar. In fact, the only infinite trees that arise are those generated for recursive environments.

The stack z consists of a list of stackable values v. These values are procedures (closed in the representation u_0 of a global environment, to be described later), environments, and other values, which we call *immediate* values. These immediate values 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).

We have not included a grammar for the heap; it 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.

The operational semantics of each machine instruction is shown in Figure 13. The machine halts normally by executing a *halt* instruction, returning the value $\langle \mathbf{ok}, n\rangle$ for some integer n. The last rule in Figure 13 specifies that if the machine state does not match any of the preceding left-hand sides, then the machine goes into an error state and halts, returning the value $\langle\textbf{error}\rangle$. `fix` builds an infinite rational tree with a loop in it. For example, if $t = \texttt{fix}(\lambda u.\, \mathit{extends}_{rg}\, ua)$, then t is a tree such that $t = \mathit{extends}_{rg}\, ta$. It is easy to see that there is exactly one infinite tree that has this property, namely the one generated by the finite graph in which the left subtree of the $\mathit{extends}_{rg}$ loops back to the top of the tree [7]. The auxiliary `lookup` is an operation on syntax trees that simulates environment lookup. `truish` is a test on rational trees, simulating the operator *truish* of [1, Appendix A]. `apply-prim` handles the application of primitive procedures p.

Each set of terms manipulated has an analogous domain in the compiler oriented semantics for Pure PreScheme. The relation between syntactic and denotational variables is shown in Figure 14. Each set also has a natural valuation motivated by the form of the equations of the compiler oriented semantics. Some sample clauses of these valuations are:

$$\mathcal{U}[\![\mathit{emptydisplay}]\!] = \mu_0 \qquad \text{the initial runtime environment}$$

$$\mathcal{Q}[\![(\mathit{brf}\ q_0\ q_1)]\!] = \lambda\mu\zeta.\, \mathit{truish}((\zeta \downarrow 1)) \to \mathcal{Q}[\![q_0]\!]\mu(\zeta \dagger 1),\ \mathcal{Q}[\![q_1]\!]\mu(\zeta \dagger 1)$$

$$\mathcal{V}[\![\langle\textbf{proc}, \langle q, u\rangle\rangle]\!] = \lambda\epsilon^*.\, \mathcal{Q}[\![q]\!](\mathit{extends}_{rl}(\mathcal{U}[\![u]\!])\epsilon^*) \text{ in } E$$

The semantics for machine states is given by

$$\mathcal{M}[\![\langle q, u, z, h\rangle]\!] = (\mathcal{Q}[\![q]\!])(\mathcal{U}[\![u]\!])(\mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!])$$

We have given these valuations as if they were for finite trees. To accommodate infinite trees, we actually define a sequence of approximate valuations; for example, we write

$$\mathcal{V}_0\, [\![v]\!] = \bot$$
$$\mathcal{V}_{n+1}\, \langle\textbf{proc}, \langle q, u\rangle\rangle = \lambda\epsilon^*.\, \mathcal{Q}_n\, [\![q]\!](\mathit{extends}_{rl}(\mathcal{U}_n\, [\![u]\!])\epsilon^*) \text{ in } E$$

Given the valuations, the transitions of the machine were derived by performing β-conversion on the terms used in the compiler oriented valuations. The *brf* rule is derived by simple calculations:

$$\begin{array}{l}
\mathcal{M}[\![\langle(\mathit{brf}\, q_0\, q_1), u, (v :: z), h\rangle]\!] \\
\ = \quad \text{[by definition of } \mathcal{M}\text{]} \\
\mathcal{Q}[\![(\mathit{brf}\, q_0\, q_1)]\!](\mathcal{U}[\![u]\!])(\mathcal{Z}[\![(v :: z)]\!])(\mathcal{H}[\![h]\!]) \\
\ = \quad \text{[by definition of } \mathcal{Z}\text{]} \\
\mathcal{Q}[\![(\mathit{brf}\, q_0\, q_1)]\!](\mathcal{U}[\![u]\!])(\mathcal{V}[\![v]\!] :: \mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!])
\end{array}$$

Abstract machine:

$$
\begin{array}{lcl}
\langle (\mathit{halt}), u, (v :: z), h\rangle & \Longrightarrow & \langle \mathbf{ok}, v\rangle \\
\langle (\mathit{goto}\ q), u, z, h\rangle & \Longrightarrow & \langle q, u, z, h\rangle \\
\langle (\mathit{label}\ q), u, z, h\rangle & \Longrightarrow & \langle q, u, z, h\rangle \\
\langle (\mathit{fetch}_l\ i\, q), u, z, h\rangle & \Longrightarrow & \langle q, u, ((\texttt{lookup}\ u\, i) :: z), h\rangle \\
\langle (\mathit{brf}\ q_0\, q_1), u, (v :: z), h\rangle & \Longrightarrow & \langle (\texttt{truish}(v) \to q_0, q_1), u, z, h\rangle \\
\langle (\mathit{save\text{-}env}\ q), u, z, h\rangle & \Longrightarrow & \langle q, u, (\langle \mathbf{env}, u\rangle :: z), h\rangle \\
\langle (\mathit{restore\text{-}env}\ q), u, z, h\rangle & \Longrightarrow & \langle q, u', (v :: z'), h\rangle \\
\quad \text{where } z = (v :: \langle \mathbf{env}, u'\rangle :: z') & & \\
\langle (\mathit{update\text{-}store}\ i\, q), u, (v :: z), h\rangle & \Longrightarrow & \langle q, u, (v :: z), (\texttt{update}\, (\texttt{lookup}\ u\, i)\, v\, h')\rangle \\
\langle (\mathit{prim\text{-}apply}\ n\, p\, q), u, z, h\rangle & \Longrightarrow & \langle q, u, (w_1 :: z'), h_1\rangle \\
\quad \text{where } z = (v_1 :: \ldots :: v_n :: z') & & \\
\quad \text{and } (w_1, h_1) = (\texttt{apply-prim}\, p (v_1 \ldots v_n) h) & & \\
\langle (\mathit{add\text{-}to\text{-}env}\ n\, q), u, z, h\rangle & \Longrightarrow & \langle q, (\mathit{extends}_{rl}\ u\, (v_1 \ldots v_n)), z', h\rangle \\
\quad \text{where } z = (v_1 :: \ldots :: v_n :: z') & & \\
\langle (\mathit{add\text{-}to\text{-}env}_g\ q), u, (v :: z), h\rangle & \Longrightarrow & \\
\quad \langle q, (\mathit{extends}_{rg}\ u\, (\texttt{new}\ h)), z, (\texttt{update}\, (\texttt{new}\ h)\, v\, h)\rangle & & \\
\langle (\mathit{closerecs}\, (q_1 \ldots q_n)\, q), u, z, h\rangle & \Longrightarrow & \\
\quad \langle q, (\texttt{fix}\, (\lambda u'.\, (\mathit{extends}_{rl}\ u\, (\langle \mathbf{proc}, \langle q_1, u'\rangle\rangle \ldots \langle \mathbf{proc}, \langle q_n, u'\rangle\rangle)))), z, h\rangle & & \\
\langle (\mathit{tail\text{-}call}), u, z_1, h\rangle & \Longrightarrow & \langle q, (\mathit{extends}_{rl}\ u_1\, (v_1 \ldots v_n)), \langle\rangle, h\rangle \\
\quad \text{where } z_1 = (\langle \mathbf{proc}, \langle q, u_1\rangle\rangle :: v_1 :: \ldots :: v_n :: \langle\rangle) & & \\
\text{otherwise} & \Longrightarrow & \langle \mathbf{error}\rangle
\end{array}
$$

Auxiliaries:

$$
\texttt{lookup}\, (\mathit{extends}_{rl}\, u (v_0 \ldots v_{n-1}))\, (i) = \begin{cases} v_i & 0 \le i < n \\ \texttt{lookup}\ u (i - n) & n \le i \end{cases}
$$

$\texttt{lookup}\, (\mathit{extends}_{rg}\, u\, a)\ (0) = a$
$\texttt{lookup}\, (\mathit{extends}_{rg}\, u\, a)\ (i + 1) = \texttt{lookup}\ u\, i$
$\texttt{lookup}\ \mathit{emptydisplay}\ i$ is an error

Figure 13: Operational Semantics of the Combinator Machine

Syntactic	Denotational	Description
v	ϵ	expressed value
q	π	code sequence
u	μ	runtime environment
z	ζ	stack
h	σ	store, also called the heap
i	ι	global variable reference
a	α	location
g	γ	symbol table

Figure 14: Analogous syntactic and denotational variables

$= \quad$ [by definition of $\mathcal{Q}$ and *brf*]
$truish(\mathcal{V}[\![v]\!]) \to \mathcal{Q}[\![q_0]\!](\mathcal{U}[\![u]\!])(\mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!]), \mathcal{Q}[\![q_1]\!](\mathcal{U}[\![u]\!])(\mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!])$
$= \quad$ [by definition of `truish`]
$\texttt{truish}(v) \to \mathcal{Q}[\![q_0]\!](\mathcal{U}[\![u]\!])(\mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!]), \mathcal{Q}[\![q_1]\!](\mathcal{U}[\![u]\!])(\mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!])$
$= (\texttt{truish}(v) \to \mathcal{Q}[\![q_0]\!], \mathcal{Q}[\![q_0]\!])\ (\mathcal{U}[\![u]\!])\ (\mathcal{Z}[\![z]\!])\ (\mathcal{H}[\![h]\!])$
$= \mathcal{Q}[\![\texttt{truish}(v) \to q_0,\, q_1]\!](\mathcal{U}[\![u]\!])(\mathcal{Z}[\![z]\!])(\mathcal{H}[\![h]\!])$
$= \mathcal{M}[\![\langle(\texttt{truish}(v) \to q_0,\, q_1), u, z, h\rangle]\!]$

Observe that the reduction semantics of the instructions in the compiler oriented semantics determine the transitions of the combinator machine. One could regard each instruction as a combinator, so that each transition corresponds to the reduction of one combinator. This interpretation of the machine gives it its name.

The soundness of the machine can be stated as follows:

Theorem 5 (Soundness of the Combinator Machine) *If* $\langle q, u, z, h\rangle \Longrightarrow \langle q', u', z', h'\rangle$*, then* $\mathcal{M}[\![\langle q, u, z, h\rangle]\!] = \mathcal{M}[\![\langle q', u', z', h'\rangle]\!]$.

> **Proof:** By calculation for each transition in Figure 13. In the case of a *closerecs*, we unwind the fixed-point one time to see that u' is a legal runtime environment. The valuation $\mathcal{Q}$ is carefully written to check the length of the stack and the tags of the operands so that the error transition is sound. ∎

5.2. The Combinator-Code Compiler

The Pure PreScheme compiler is similar to the compiler oriented semantics except the compiler produces syntax rather than a denotation. The compiler is patterned after the semantics in that there is function for each valuation. For example, for the compilation of simple expressions, the analog of $\mathcal{CSE}$ is `cse`, which has the following specification:

$$\mathcal{Q}[\![\texttt{cse}\,[\![\mathrm{S}]\!]gq]\!] = \mathcal{CSE}[\![\mathrm{S}]\!](\mathcal{G}[\![g]\!])(\mathcal{Q}[\![q]\!])$$

where $\mathcal{G}$ is the valuation for symbol tables.

The function `cse` could be the lambda term constructed by replacing the denotational variables with their analogs in the definition of $\mathcal{CSE}$. Normal order reduction could be used to specify when `cse` is defined. This definition of `cse` obviously meets its specification. In practice, Scheme program text defined `cse`. For example, we could write

```
(define cse
  (lambda (exp symtab q)
    (record-case exp
      (local-var (l)
        (fetch-l (apply-symtab symtab l) q))
      ...)))
```

This Scheme code could be used to simulate $\mathcal{CSE}$ by defining, following Section 4,

```
(define fetch-l
  (lambda (lex-addr q)
    (lambda (mu zeta)
      (if (expressed-value? (mu lex-addr))
        (q mu (cons (restrict-to-E (mu lex-addr))
                    zeta))
        (wrong "Local variable given storage.")))))
```

or it could be used as part of the definition of `cse` by defining

```
(define fetch-l
  (lambda (lex-addr q)
    (list 'fetch-l lex-addr q)))
```

which emits a representation of a fetch_l instruction.

The syntactic structure of Pure PreScheme programs allows us to characterize the structure of a compiled program `cp` $[\![\mathrm{P}]\!]$. Every compiled program consists of a prelude that allocates globals with a sequence of $\mathit{add\text{-}to\text{-}env}_g$ instructions, followed by a *closerecs* instruction that creates all the procedures. This prelude is followed by the code for the body of the program. The effect of the prelude is to create an environment u_0 in which all the global procedures are closed.

We can combine this with the correctness of the compiler-oriented semantics (Theorem 4) to get the following:

Theorem 6 (Adequacy) *For any program* P *and answer* $a \neq \bot$, $\mathcal{P}[\![\mathrm{P}]\!] = a$ *if and only if* $\langle \texttt{cp}\,[\![\mathrm{P}]\!], \mathit{emptydisplay}, \langle\rangle, h_0\rangle \Longrightarrow a$

> **Proof:** The "if" follows immediately from the correctness of the compiler-oriented semantics, (Theorem 4), the soundness of the combinator machine, and the operational semantics of the *halt* instruction. Hence if the machine halts (either with a number or an error), the machine's answer and the denotational answer coincide.
>
> To prove the "only if", we must show that if the machine runs infinitely, then the denotation must be bottom. To do this, recall the definition of a reduction strategy from [3, Chapter 13]. A reduction strategy is a function F on λ-terms such that for all M, $M \rightarrow^* F(M)$. Then the soundness theorem above asserts that the combinator machine induces a reduction strategy on all terms of the form $quzh$. Furthermore, this strategy is quasi-leftmost, since each machine-induced reduction begins with a head reduction. Since any quasi-leftmost strategy is normalizing (Theorem 13.2.6 of [3]), if the machine runs infinitely long from state $\langle q, u, z, h\rangle$, then $quzh$ (thought of as a λ-term) has no normal form. In fact, since the reduction sequence has infinitely many head reductions, this term has no head normal form, and therefore denotes $\bot$ in the semantic domains. The desired result then follows from Theorem 4. ∎

An important property of the combinator machine is that the size of the terms (except for the heap) is statically bounded. To express this, we define $\mathit{dom}(u) = \{i \mid \texttt{lookup}\ u\, i \text{is not an error}\}$. Then we have:

Theorem 7 (Bounded Space) *If* P *is a valid Pure PreScheme program, then there exist integers* N_0 *and* N *such that for any state* $\langle q, u, z, h\rangle$ *in the computation sequence of* $\langle \texttt{cp}\,[\![\mathrm{P}]\!], \mathit{emptydisplay}, \langle\rangle, h_0\rangle$, $|\mathit{dom}(u)| \leq N$. *Furthermore, after the execution of the single closerecs instruction in* $\texttt{cp}\,[\![\mathrm{P}]\!]$,

$$N_0 \leq |\mathit{dom}(u)| \leq N$$

> **Proof:** (Sketch) A Pure PreScheme program must consist of a set of 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. ∎

5.3. Design Choices in the Compiler

The translation from Pure PreScheme to combinator code was patterned after [6]. In this section, we discuss some of the innovations and design choices that we made in the process of developing our translator.

An important change from [6] is the explicit separation of the syntax and semantics of the combinator code. This separation was already present in [24], but was dropped in [25, 26] in favor of punning between λ-terms and their denotations. At the time, the pun was useful in deriving the compiler-oriented semantics, but many people found it confusing. Furthermore, a clear separation between syntax trees and domain elements allowed us to understand when continuity arguments were permissible, and when they were not. This was even more important once rational trees became part of the structure.

We slightly modified the architecture of the machine. Clinger used a value register ϵ and a value stack ϵ^* where we use the value stack ζ. Clinger's value register serves to cache the top element of the stack. As a result, his induction hypotheses are asymmetrical because they need to keep track of when the cache is active: note that in Clinger's induction hypotheses A and B, ϵ occurs free on the left but not on the right, and in his induction hypothesis C, it appears free, but one has a precondition that says its value is ignored [6, page 360]. He also needs to generate *push* instructions to copy the value register onto the top of the stack. By replacing these with a single value stack, our induction hypotheses became more symmetrical and easier to work with; we did not need any pesky side conditions such as the one for Clinger's hypothesis C. We could rely on a later register allocation phase, such as that in [17], to deal with caching strategies.

Another change was in the separation of the different compiler modes. Clinger's compiler used two different compiling valuations, t (for general expressions) and e (for expressions in tail position). The choice of which valuation to use was embedded in the code of the compiler [6, Figure 5]. We chose instead to separate tail-recursive expressions from simple expressions in the syntax. This made it clearer which valuation should be used at any point in the compiler.

Another distinction in the compiler was between simple expressions evaluated for value or for effect. It is advantageous for the compiler to distinguish between these cases so that it need not worry about handling the value of an expression evaluated for effect. There were a number of different ways of organizing the compiler to treat this distinction. We could have made a separate syntactic category for commands. This would be the obvious approach for an ordinary procedural language. However, in our language, the set of commands would have been exactly the same as the set of simple expressions, so introducing a new syntactic category would have been redundant. We could have had commands leave their values on the top of the stack, and introduced an instruction that would drop the top value of the stack when it was to be ignored. This would have led to unacceptably inefficient code. In the end, we simply used a separate compiler valuation for expressions in command position, and we introduced a few extra instructions, such as *update-store*$_i$, to perform primitive operations without pushing a value on the stack. This is also consistent with modern ideas about machine architecture, which condone and even encourage new instructions to handle common special cases.

A more subtle change was in the role of the specifications. In [6], induction hypotheses were introduced only as part of the internals of the proof. Indeed, Clinger writes, "... the algorithm was designed and a compiler built before any thought was given to a formal correctness proof." [6, page 356]. Instead, in keeping with the ideas of [27], we viewed the induction hypotheses as *specifications* for the compiler: they described what the code generated by the compiler was supposed to do. This idea, along with our simplified machine design, allowed us to write down the specifications for the different valuations in a relatively systematic way. Given the specifications and a few general ideas about the form of the code, it was easy to write the clauses for the various productions. Thus the design of the specifications preceded the writing of the compiler. Most (though not all) of the proofs were easy, as was shown in Section 4.6. We believe that the induction hypotheses were a major work product, and should be considered a major work product of future compiler verification projects.

6. The Linear-Data Machine

The linear-data machine differs from the combinator machine in that it uses a linearly-addressed data store to represent the environment and stack of the combinator machine. It uses two pointers into the store, *sp* and *up*, to represent the stack and the environment. It also has a heap which corresponds to the heap of the combinator machine. Thus the state of the linear-data machine is of the form $\langle q, up, sp, s, h\rangle$, where *sp* and *up* are integers, and s is a map from integers to tree structured values.

In this section we will formalize the way in which the *sp* and *up* pointers represent the stack and the environment. 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 representation utilizes the fact that any PreScheme program uses only a bounded, statically-determinable amount of space in the store. 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 7) and are used for setting the initial parameters of the linear-data machine.

6.1. Informal Presentation

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

We can describe the representation pictorially as follows:

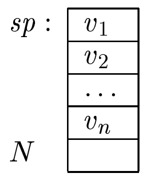
1. Stackable values (integers, booleans, characters, strings, procedures, environments). In keeping with the assumptions of the runtime model discussed in Section 2, runtime tags are not used. This restricts our correctness result but only for those programs that result in an error. All quantities are represented by a single machine word (i.e. one location in s). Strings and procedures are represented by pointers into a static space; since all procedures in a PreScheme program 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. Since 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 a single word in memory, this suggests that u be represented by locations 0 through up in s:

$$\begin{array}{r|c|}\cline{2-2} up: & v_0 \\ \cline{2-2} & v_1 \\ \cline{2-2} & \ldots \\ \cline{2-2} 0 & \\ \cline{2-2}\end{array}$$

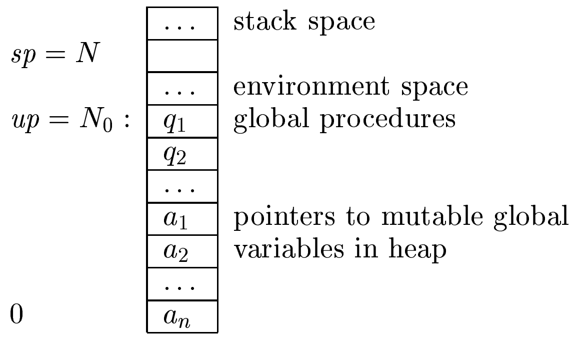
   where $v_0 = \texttt{lookup}\, u\, 0$, $v_1 = \texttt{lookup}\, u\, 1$, etc. By Theorem 7, $0 \leq up \leq N$.

3. The local stack z. Since all values are represented by a single word in s and that up ranges between 0 and N, a local stack $z = (v_1 :: v_2 :: \ldots :: v_n :: \langle\rangle)$ can be represented using s 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-to-env*$_g$ 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 configuration restored at tail call:



6.2. Formal Presentation

To formalize this representation, we will introduce the notion of *storage layout relation* [10] which is defined in [8, Section 3.3]. A storage layout relation is a predicate asserting that some concrete values (that is, values in the linear-data machine) represent some abstract term (that is, a term in the language of the combinator machine). A typical example is the relation for environments, which is a judgement of the form $(s, b) \models_U p \simeq u$. This predicate asserts that pointer p into store s represents run-time environment u. (Here the parameter b marks the upper boundary of environment space). Such a relation is typically defined by a least fixed-point induction on a powerset $P(X \times Y)$ for suitable sets X and Y.

We will have one storage layout relation for each kind of data manipulated by the combinator and linear-data machines. These are: immediate data, stackable values, stacks, environments, heaps, and programs. These will be defined as the least fixed point of a simultaneous induction. There is no difficulty in defining these relations on infinite trees; certain infinite trees will be in the relation because one of the base cases in the induction (the one for recursive environments) will require it. 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.

6.2.1. Relating Immediate Data

If i and i' are immediate data, then $i \simeq_{imm} i'$ (read "i represents i'") iff i is the untagged version of i'. This is a degenerate storage-layout relation, since it does not involve the store.

6.2.2. Relating Stackable Values

$$\begin{array}{lll}
\multicolumn{3}{l}{(s,b) \models_V c \simeq v \iff} \\
& \text{either} & v = \langle \mathbf{proc}, \langle q, u_0 \rangle \rangle \text{ and } c = q \\
& \text{or} & v = \langle \mathbf{env}, u \rangle \\
& & \text{and } (s,b) \models_U c \simeq u \\
& & \text{and } c \leq b \\
& \text{or} & c \simeq_{imm} v
\end{array}$$

Here u_0 denotes the environment tree at the end of the prelude. This rule is noteworthy for two reasons: First, in this representation scheme only procedures closed in this environment are representable. However, all procedures in a PreScheme program are closed in this environment, so this restriction is compatible with the semantics of PreScheme. Second, u_0 may in general be an infinite rational tree, so any solution to this equation, including its least fixed point, will have infinite trees in the relation. We could have made u_0 a parameter to the definition of $\models$, as we did with s and b, but that would have made our notation even more cumbersome with little gain in clarity.

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*.

We use c to range over concrete values (the values in the linear-data machine), and v to range over rational trees as in Section 5.

6.2.3. Relating Pointers and Stacks

$$
\begin{array}{lll}
\multicolumn{3}{l}{(s,b) \models_Z \langle p,p'\rangle \simeq z \iff} \\
& \text{either} & z = \langle\rangle \text{ and } p = p' \\
& \text{or} & z = (v :: z') \text{ and } p > p' \text{ and } (s,b) \models_V s(p) \simeq v \\
& & \text{and } (s,b) \models_Z \langle p-1,p'\rangle \simeq z'
\end{array}
$$

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.

6.2.4. Relating Pointers and Environments

$$
\begin{array}{lll}
\multicolumn{3}{l}{(s,b) \models_U p \simeq u \iff} \\
& \text{either} & u = \mathit{emptydisplay} \text{ and } p = -1 \\
& \text{or} & u = (\mathit{extends}_{rl}(v_1 \ldots v_n)u') \text{ and } p \leq \mathit{up} \\
& & \text{and } (s,b) \models_V s(p) \simeq v_n \ldots (s,b) \models_V s(p-(n-1)) \simeq v_1 \\
& & \text{and } (s,b) \models_U (p-n) \simeq u' \\
& \text{or} & u = (\mathit{extends}_{rg}\, au') \text{ and } p \leq \mathit{up} \\
& & \text{and } s(p) \simeq a \\
& & \text{and } (s,b) \models_U (p-1) \simeq u'
\end{array}
$$

6.2.5. Relating Programs

Programs in the linear-data and combinator machines differ only in the format of the literals embedded in them. Thus we define

$$
\begin{array}{lll}
\multicolumn{3}{l}{q_l \simeq q_c \iff} \\
& \text{either} & q_c = (\mathit{constant}\; i'q_c') \text{ and } q_l = (\mathit{constant}\; iq_l') \\
& & \text{and } i \simeq i' \text{ and } q_l' \simeq q_c' \\
& \text{or} & q_c = (\mathit{fetch}_l\; iq_c') \text{ and } q_l = (\mathit{fetch}_l\; iq_l') \\
& & \text{and } q_l' \simeq q_c' \\
& \text{or} & q_c = (\mathit{fetch}_g\, iq_c') \text{ and } q_l = (\mathit{fetch}_g\, iq_l') \\
& & \text{and } q_l' \simeq q_c' \\
& \text{or} & q_c = (\mathit{goto}\, q_c') \text{ and } q_l = (\mathit{goto}\, q_l') \\
& & \text{and } q_l' \simeq q_c' \\
& & \text{etc.}
\end{array}
$$

6.2.6. Relating Heaps

The linear-data heap is like the combinator-machine heap except that its L-values are *untagged* integers, and its first component contains immediate values in their linear-data representation. Hence the first component maps untagged integers to immediate values in their linear-data representation, the second component is an integer-valued free-storage counter, and the third component is the same as that for the combinator machine. Heaps correspond if their free-storage counters and third components agree, and if the allocated portions of their first components correspond:

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

6.3. Relating the Combinator and Linear-Data Machines

We can now define the correspondence between combinator and linear-data machine states.

The effect of the prelude is to establish the invariants

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

and

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

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

Definition 6 *We say a linear-data machine state* $\langle q, up, sp, s, h \rangle$ corresponds *to an combinator machine state* $\langle q', u, z, h' \rangle$*, written*

$$\langle q, up, sp, s, h \rangle \simeq \langle q', u, z, h' \rangle$$

if and only if the following conditions are satisfied:

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

6.4. Basic Properties of the Representation

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

Lemma 3 (Immediate Values) *If $c \simeq_{imm} v$, then for all s and b, $(s, b) \models_V c \simeq v$.*

Definition 7 *We say $s =_{[p,p']} s'$ if $p \leq p'$ and $(\forall x : p \leq x \leq p')(s(x) = s'(x))$.*

Lemma 4 (Free-Storage Lemma) *1. If $(s, b) \models_V c \simeq v$ and $s =_{[0,b]} s'$ then $(s', b) \models_V c \simeq v$.*

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

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

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

Lemma 5 *If $(s, b) \models_V c \simeq v$ and $b \leq b'$ then $(s, b') \models_V c \simeq v$.*

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

Lemma 6 *If $(s, b) \models_U p \simeq u$ then $(s, b) \models_V s(p - n) \simeq$ `lookup` $u\, n$.*

6.5. Definition of the Linear-Data Machine

We define the operational semantics of the linear-data machine by considering the representation of each possible combinator machine state, so that the linear-data machine simulates the behavior of the combinator machine. That is, if $L_1 \simeq C_1$ and $C_1 \Longrightarrow C_2$, then we want the linear-data machine to send L_1 to some state L_2 such that $L_2 \simeq C_2$.

In general, we will only be able to perform this simulation when C_1 is a state arising from reducing a properly compiled Pure PreScheme program. There are a number of things that might go wrong:

- The linear-data machine does not have tags to distinguish different data types. The combinator machine uses these tags to distinguish an error state. Since the linear-data machine cannot mimic this behavior, we will require the linear-data machine to behave correctly only if A_2 is not an error state. If A_2 is an error state, the behavior of the linear-data machine is unspecified. For type-checked PreScheme programs, such as the Scheme 48 Virtual Machine, this restriction is moot, because the typing rules guarantee that the program will never reach an error state.

- Similarly, if A_1 is an combinator-machine state with some arbitrary program q, executing it may cause other errors, such as underflowing the stack or overflowing the environment. However, this will not happen if A_1 is a state that is reached when the abstract machine is started with a correctly compiled Pure PreScheme program. The simulation theorem will have to take this into account.

The operational semantics of the linear-data machine is shown in Figure 15. The various auxiliaries used by the machine are shown in Figure 16 `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.

6.6. Correctness of the Linear-Data Machine

The combinator and linear-data machines are related by a property like the following:

Let C_1 and C_2 be states of the combinator machine and L_1 and L_2 be states of the linear-data machine. If $L_1 \simeq C_1$, $C_1 \Longrightarrow C_2$, and $L_1 \Longrightarrow L_2$, then $L_2 \simeq C_2$.

As suggested above, this property does not hold in general. However this result does hold where C_1 is a state that arises when the abstract machine is started with a correctly compiled Pure PreScheme program.

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 q, $\texttt{conc}(q) \simeq q$. Now we can state the simulation theorem:

Theorem 8 (Correctness of the Linear-Data Machine) *If* P *is a Pure PreScheme program, and $\mathcal{P}[\![\mathrm{P}]\!] = \langle \mathbf{ok}, \langle \mathbf{int}, n\rangle\rangle$, then there are values of N_0 and N such that*

$$\langle \texttt{conc}(\texttt{cp}\,[\![\mathrm{P}]\!]), -1, N, s_0, h_0\rangle \overset{*}{\Longrightarrow} \langle \mathbf{ok}, n\rangle$$

$$
\begin{array}{lcl}
\langle (\mathit{halt}), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle \mathbf{ok}, s(\mathit{sp})\rangle \\
\langle (\mathit{goto}\ q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, \mathit{up}, \mathit{sp}, s, h\rangle \\
\langle (\mathit{label}\ q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, \mathit{up}, \mathit{sp}, s, h\rangle \\
\langle (\mathit{fetch}_l\ i\, q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, \mathit{up}, \mathit{sp} + 1, s', h\rangle \\
\quad \text{where } s' = s[(\texttt{lookup}\ i\ \mathit{up}\, s)/\mathit{sp}+1] & & \\
\langle (\mathit{brf}\ q_1 q_2), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle (\texttt{truish}(s(\mathit{sp})) \to q_1, q_2), \mathit{up}, \mathit{sp} - 1, s, h\rangle \\
\langle (\mathit{save\text{-}env}\ q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, \mathit{up}, \mathit{sp} + 1, s[\mathit{up}\,/\mathit{sp}+1], h\rangle \\
\langle (\mathit{restore\text{-}env}\ q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, s(\mathit{sp} - 1), \mathit{sp} - 1, s_1, h\rangle \\
\quad \text{where } s_1 = s[s(\mathit{sp})/\mathit{sp}-1] & & \\
\langle (\mathit{update\text{-}store}\ iq), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, \mathit{up}, \mathit{sp}, s, h_1\rangle \\
\quad \text{where } h_1 = (\texttt{update}\ s(\mathit{up} - i)\ s(\mathit{sp})\ h) & & \\
\langle (\mathit{prim\text{-}apply}\, n\, p\, q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, \mathit{up}, \mathit{sp} - (n-1), s_1, h_1\rangle \\
\quad \text{where } (w_1, h_1) = (\texttt{apply-prim}'\ p\, (\texttt{collect}\ s\ \mathit{sp}\ n)\, h) & & \\
\quad \text{and } s_1 = s[w_1/\ \mathit{sp} - (n-1)] & & \\
\langle (\mathit{add\text{-}to\text{-}env}\ n\, q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle q, (\mathit{up} + n), (\mathit{sp} - n), s_1, h\rangle \\
\quad \text{where } s_1 = (\texttt{copy}\ s\, n\, (\mathit{up} + 1)\ \ \mathit{sp}) & & \\
\langle (\mathit{add\text{-}to\text{-}env}_g\ \ q), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \\
\quad \langle q, \mathit{up} + 1, \mathit{sp} - 1, (s[(\texttt{new}\ h)/(\mathit{up} + 1)]), (\texttt{update}\, (\texttt{new}\ h)\ s(\mathit{sp})\ h)\rangle & & \\
\langle (\mathit{closerecs}\, (q_1 \ldots q_n)\ q), \mathit{up}, \mathit{sp}, s, h\rangle & & \\
 & \Longrightarrow & \langle q, \mathit{up} + n, \mathit{sp}, s_1, h\rangle \\
\quad \text{where } s_1 = (\texttt{spread}\ s(\mathit{up} + n)\ (q_1 \ldots q_n)) & & \\
\langle (\mathit{tail\text{-}call}), \mathit{up}, \mathit{sp}, s, h\rangle & \Longrightarrow & \langle s(\mathit{sp}), (N_0 + n), (N-1), s_1, h\rangle \\
\quad \text{where } n = \mathit{sp} - N - 1 & & \\
\quad \text{and } s_1 = (\texttt{copy}\ s\, n\, (N_0 + 1)\ (\mathit{sp} - 1)) & & \\
\text{otherwise} & \Longrightarrow & \langle \mathbf{error}\rangle
\end{array}
$$

Figure 15: Operational Semantics of the Linear-Data Machine

$(\texttt{new}\ h) = h.2 + 1$

$(\texttt{update}\ l\, v\, h) = \langle h.1[v/l], l > h.2 \to l, h.2, h.3\rangle$

$(\texttt{lookup}\ i\ \mathit{up}\ s) = s(\mathit{up} - i)$

$(\texttt{copy}\ s\, n\, d\, s\,) = (n = 0) \to s, (\texttt{copy}\ s[s(s)/d]\,(n-1)\,(d+1)\,(s-1))$

$(\texttt{spread}\ s\, n\, \langle\rangle) = s$
$(\texttt{spread}\ s\, n\, v :: (v_0 \ldots v_n)) = (\texttt{spread}\ s[v/n]\,(n-1)\,(v_0 \ldots v_n))$

$(\texttt{collect}\ s\, i\, 0) = \langle\rangle$
$(\texttt{collect}\ s\, i\, n) = s(i) :: (\texttt{collect}\ s\,(i-1)\,(n-1))$

Figure 16: Auxiliaries for the Linear-Data Machine

Proof: Choose N_0 and N as in Theorem 7. Let C_k and L_k denote the states of the combinator and linear-data machines, respectively, after k steps. We will show that for all sufficiently large k, either $L_k \simeq C_k$ or both machines halt with corresponding answers.

We observe that the programs for both the linear-data and combinator machines begin with a sequence of *add-to-env*$_g$ instructions, followed by a *closerecs* instruction. Let k_0 be the number of steps to execute these instructions.

The machines start in states

$$\langle q, -1, N, s_0, h_0\rangle$$

for the linear-data machine and

$$\langle q', \mathit{emptydisplay}, \langle\rangle, h'_0\rangle$$

for the combinator machine. Comparing these states with the definition of $\simeq$ for states, we see that all of the conditions for correspondence hold, except for $N_0 \leq \mathit{up}$ and $(s, N) \models_U N_0 \simeq u_0$ Let $\simeq_0$ denote this weakened relation. Since there are no procedures represented during the preamble, this relation does not depend on u_0.

We then show the following:

For all $k < k_0$, $L_k \simeq_0 C_k$.
For all $k \geq k_0$, $L_k \simeq C_k$.

The programs q and q' begin with a sequence of *add-to-env*$_g$ instructions. A straightforward calculation, using the definitions of the storage layout relations and Lemma 4, shows that these instructions preserve $\simeq_0$. This establishes the first proposition.

Next, we turn to the *closerecs* instruction that ends the prelude. Assume we have

$$\begin{aligned}&\langle(\mathit{closerecs}\,(q_1 \ldots q_n)\,q), \mathit{up}, \mathit{sp}, s, h\rangle\\ &\quad\simeq_0 \langle(\mathit{closerecs}\,(q'_1 \ldots q'_n)\,q'), u, z, h'\rangle\end{aligned}$$

We must show that

$$\langle q, \mathit{up}+n, \mathit{sp}, s', h\rangle \simeq \langle q', u_0, z, h'\rangle$$

where

$$\begin{aligned}s' &= (\texttt{spread}\ s(\mathit{up}+n)\,(q_1 \ldots q_n))\\ u_0 &= (\texttt{fix}(\lambda u'.\,(\mathit{extends}_{rl}\ u\,(\langle\mathbf{proc}, \langle q'_1, u'\rangle\rangle \ldots \langle\mathbf{proc}, \langle q'_n, u'\rangle\rangle))))\end{aligned}$$

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

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

By the definition of s', for $1 \leq i \leq n$ we have $s'(\mathit{up}+n-i+1) = q_i$, so by the definition of $\models_V$ we have $(s', \mathit{up}) \models_V s'(\mathit{up}+n-i+1) \simeq \langle\mathbf{proc}, \langle q'_i, u_0\rangle\rangle$, and $(s', \mathit{up}) \models_U \mathit{up} \simeq u$, so by the definition of $\models_U$ we have $(s', \mathit{up}) \models_U \mathit{up}+n \simeq u_1$, where

$$u_1 = (\mathit{extends}_{rl}\ u\,(\langle\mathbf{proc}, \langle q'_1, u_0\rangle\rangle \ldots \langle\mathbf{proc}, \langle q'_n, u_0\rangle\rangle))$$

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

All the other conditions are established trivially. This establishes $L_{k_0} \simeq C_{k_0}$.

We next turn to the case of $k_0 < k$ We must show that if $k_0 \leq k$ and $L_k \simeq C_k$, then $L_{k+1} \simeq C_{k+1}$. We do this by a tedious analysis of each instruction, using the definitions of the storage layout relations and Lemma 4.

∎

6.7. Development of Storage-Layout Relations

Storage-layout relations, as we have described them, allow us to structure a proof by induction on computation steps. Many early compiler-correctness proofs used congruences between domains as a way of structuring such proofs, e.g. [23]. Such proofs were complicated because the relations were formulated between elements of domains, and inverse-limit constructions acted as a surrogate for operational reasoning.

Instead, relying on the development of natural semantics [20], our predicates relate pairs of trees, thus avoiding most of these complications. Our definitions are patterned on those of Hannan [10]. In [28], we introduced these ideas, showing how the relations of [10] could be generalized and the proofs simplified.

7. The Stored-Program Machine

The stored-program machine is identical to the linear-data machine, except that the code is represented in cells of a linear instruction store M, much like the cells of the data store.

We use storage-layout relations to specify the correspondence between the two machines. We first show how linear-data machine code is represented in the linear program store. For example the representation of bytecode $(\mathit{fetch}_l\ i\, q_0)$ at location p in M is:

$$p: \begin{array}{|c|}\hline \mathit{fetch}_l \\ \hline i \\ \hline \\ \hline \\ \end{array}$$

where $p+2$ represents q_0. This notion is formalized by the storage-layout relation $\models_P$:

$$\begin{array}{ll}
\multicolumn{2}{l}{M \models_P p \simeq q \iff} \\
\text{either} & M(p) = \mathit{halt} \text{ and } q = (\mathit{halt}) \\
\text{or} & M(p) = \mathit{goto} \text{ and } q = (\mathit{goto}\ q_0) \\
 & \text{and } M \models_P M(p+1) \simeq q_0 \\
\text{or} & q = (\mathit{label}\ q_0) \\
 & \text{and } M \models_P p \simeq q_0 \\
\text{or} & M(p) = \mathit{fetch}_l \text{ and } q = (\mathit{fetch}_l\ i\, q_0) \\
 & \text{and } M(p+1) = i \text{ and } M \models_P p+2 \simeq q_0 \\
\text{or} & M(p) = \mathit{brf} \text{ and } q = (\mathit{brf}\ q_0\, q_1) \\
 & \text{and } M \models_P p+2 \simeq q_0 \text{ and } M \models_P M(p+1) \simeq q_1 \\
\text{etc.} &
\end{array}$$

This change propagates through the other classes of data that are represented in the machine. Values correspond if they are alike except for the representation of programs. Data stores correspond up to pointer p if they correspond location by location for all smaller locations. Heaps correspond if they both have the same number of used cells and the values of corresponding cells are related. All these can be written easily using the language of storage-layout relations in the style of Section 6.

Given these relations, we can define the correspondence between states of the linear-data machine and fetch states of the stored-program machine. For correspondence, the stored-program machine's instruction pointer must correspond to the linear-data machine's program. Furthermore the dedicated registers must have the same values in both machines and data in stores and heaps must correspond.

Definition 8 *We say that under linear code store M, stored-program machine state $F(ip, up, sp, s, h)$ corresponds to a linear-data machine state $\langle q, up', sp', s', h'\rangle$ (written $M \models F(ip, up, sp, s, h) \simeq \langle q, up', sp', s', h'\rangle$) if and only if:*

1. $M \models_P ip \simeq q$
2. $up = up'$
3. $sp = sp'$
4. $M \models_{sp} s \simeq s'$
5. $M \models_H h \simeq h'$

The formal definition of the stored-program machine appears in figure 17. The program resides in the linear program store M, which is indexed by register *ip*. The machine uses a fetch-execute cycle. In a fetch state $F(ip, up, sp, s, h)$, the machine retrieves the contents of the location pointed to by *ip* and goes into the execute state $E(ip, M(ip), up, sp, s, h)$. In the execute state, the machine decodes the current instruction and behaves accordingly, going into a fetch state. The machine's goal is to preserve correspondence with the behavior of the linear-data machine. The program store is treated as a global since it remains constant throughout execution.

The simulation theorem says that machines in corresponding states remain in corresponding states as they execute. Therefore, upon termination they will return corresponding values.

Theorem 9 *Let L_1 and L_2 be states of the linear-data machine and P_1 and P_2 be fetch states of the stored-program machine. If $M \models P_1 \simeq L_1$, $L_1 \Longrightarrow L_2$, and $P_1 \Longrightarrow P_2$, then $M \models P_2 \simeq L_2$.*

Proof:

By analysis of each instruction. ▮

8. The Linearizer and its Correctness

In Theorem 9, we showed that if L and P were corresponding states of the linear-data machine and the stored-program machine, then L and P would always compute in corresponding states. In particular, if L halted with an integer n, then so would P. Theorem 8 specified how to get from a Pure PreScheme program P to a suitable linear-data machine state by using `cp`

$$
\begin{array}{lll}
\text{Fetch state :} & & \\
F(ip, up, sp, s, h) & \Longrightarrow & E(ip, M(ip), up, sp, s, h) \\
& & \\
\text{Execute state :} & & \\
E(ip, \mathit{halt}, up, sp, s, h) & \Longrightarrow & \langle \mathbf{ok}, s(sp) \rangle \\
E(ip, \mathit{goto}, up, sp, s, h) & \Longrightarrow & F(M(ip+1), up, sp, s, h) \\
E(ip, \mathit{fetch}_l, up, sp, s, h) & \Longrightarrow & F(ip+2, up, sp+1, s_1, h) \\
\quad \text{where } s_1 = s[(\texttt{lookup}\ M(ip+1)\ up\ s)/(sp+1)] & & \\
E(ip, \mathit{brf}, up, sp, s, h) & \Longrightarrow & F(ip_1, up, sp-1, s, h) \\
\quad \text{where } ip_1 = (\texttt{truish}(s(sp)) \rightarrow (ip+2), M(ip+1)) & & \\
E(ip, \mathit{save\text{-}env}, up, sp, s, h) & \Longrightarrow & F(ip+1, up, sp+1, s[up/sp+1], h) \\
E(ip, \mathit{restore\text{-}env}, up, sp, s, h) & \Longrightarrow & F(ip+1, s(sp-1), sp-1, s_1, h) \\
\quad \text{where } s_1 = s[s(sp)/sp-1] & & \\
E(ip, \mathit{update\text{-}store}, up, sp, s, h) & \Longrightarrow & F(ip+2, up, sp, s, h_1) \\
\quad \text{where } h_1 = (\texttt{update}\ s(up - M(ip+1))\ s(sp)\ h) & & \\
E(ip, \mathit{prim\text{-}apply}, up, sp, s, h) & \Longrightarrow & F(ip+3, up, sp_1, s_1, h_1) \\
\quad \text{where } sp_1 = sp - (M(ip+1) - 1) & & \\
\quad \text{and } (w_1, h_1) = (\texttt{apply-prim}\ M(ip+2)\ (\texttt{collect}\ s\ sp\ M(ip+1))\ h) & & \\
\quad \text{and } s_1 = s[w_1/\ sp - (M(ip+1) - 1)] & & \\
E(ip, \mathit{update\text{-}store}_i, up, sp, s, h) & \Longrightarrow & F(ip+2, up, sp-1, s, h_1) \\
\quad \text{where } h_1 = (\texttt{update}\ s(up - M(ip+1))\ s(sp)\ h) & & \\
E(ip, \mathit{prim\text{-}apply}_i, up, sp, s, h) & \Longrightarrow & F(ip+3, up, sp - M(ip+1), s, h_1) \\
\quad \text{where } (w_1, h_1) = (\texttt{apply-prim}\ M(ip+2)\ (\texttt{collect}\ s\ sp\ M(ip+1))\ h) & & \\
E(ip, \mathit{add\text{-}to\text{-}env}, up, sp, s, h) & \Longrightarrow & F(ip+2, up_1, sp_1, s_1, h) \\
\quad \text{where } up_1 = (up + M(ip+1)) & & \\
\quad \text{and } sp_1 = (sp - M(ip+1)) & & \\
\quad \text{and } s_1 = (\texttt{copy}\ s\ M(ip+1)\ (up+1)\ \ sp) & & \\
E(ip, \mathit{add\text{-}to\text{-}env}_g, up, sp, s, h) & \Longrightarrow & F(ip+1, up+1, sp-1, s_1, h_1) \\
\quad \text{where } s_1 = (s[(\texttt{new}\ h)/(up+1)]) & & \\
\quad \text{and } h_1 = (\texttt{update}\ (\texttt{new}\ h)\ s(sp)\ h) & & \\
E(ip, \mathit{closerecs}, up, sp, s, h) & \Longrightarrow & F(M(ip+1), up + M(ip+2), sp, s_1, h) \\
\text{where } s_1 = (\texttt{spread}\ s(up + M(ip+2))\ (ip+3 \ldots ip + M(ip+2) + 2)) & & \\
E(ip, \mathit{tail\text{-}call}, up, sp, s, h) & \Longrightarrow & F(s(sp), N_0 + n, (N-1), s_1, h) \\
\quad \text{where } n = sp - N - 1 & & \\
\quad \text{and } s_1 = (\texttt{copy}\ s\ M(ip+1)\ (N_0 + 1)\ (sp-1)) & & \\
\text{otherwise} & \Longrightarrow & \mathbf{error}
\end{array}
$$

Figure 17: The fetch/execute cycle of the stored-program machine.

and `conc`, but Theorem 9 gives no corresponding specification for how to get to a stored-program machine state.

In this section, we describe an algorithm that takes the combinator code for a program q and an initial code address p, and produces a linear instruction store M such that $M \models_P p \simeq q$, that is, M contains a program, starting at p, that corresponds to q.

We call this algorithm the *linearizer*. This would be a straightforward tree traversal, except that splits and joins must be accounted for. Splits occur at conditionals. For example, consider the combinator code emitted for a simple conditional expression. The syntax-directed compiler gives:

$$\begin{aligned}\texttt{cse}\,[\![(\texttt{if}\ \mathrm{S}_0\ \mathrm{S}_1\ \mathrm{S}_2)]\!]gq = \texttt{cse}\,[\![\mathrm{S}_0]\!]g(\mathit{brf}\,&(\texttt{cse}\,[\![\mathrm{S}_1]\!]g(\mathit{label}\ \ q))\\ &(\texttt{cse}\,[\![\mathrm{S}_2]\!]g(\mathit{goto}\ \ q)))\end{aligned}$$

The linearizer must avoid making two copies of q. It does this by relying on the syntax-directed compiler to mark the join point with *label* and *goto* instructions. More precisely, the linearizer assumes that the left path into a join is marked by a *label* and the right path is marked by a *goto*. This requirement is formalized by the predicate *legal*(q) in Figure 18, which also defines the join point *join*(q) of a combinator program q. It is easy to confirm that code generated by the compiler satisfies these restrictions:

Theorem 10 *For any program* P, `conc`(`cp`$[\![\mathrm{P}]\!]$) *is legal.*

Proof: Routine induction. ∎

The linearizer is a set of three valuations on combinator-code programs. We present it in a style similar to that employed for the compiler. For each valuation, we present an informal description of its intended purpose, its formal specification, and some excerpts from its definition. As in Section 4, formulating the specification was the most difficult part of the process; once that was done, writing the linearizer and its proof was relatively straightforward. We therefore give the specifications in some detail, and the algorithm and proof in less detail.

We begin with some preliminary definitions needed for the specification.

Definition 9 *For code stores* M *and* M', $M =_p^{p'} M'$ *if and only if* $(\forall i : p \le i < p')\ M(i) = M'(i)$.

Definition 10 *The instructions tail-call and halt are called* leaves *and leaf* (q) *is true if and only if* q *is a leaf.*

q	$join(q)$	$legal(q)$
$(halt)$	$(halt)$	$true$
$(goto\ q_0)$	$(goto\ q_0)$	$legal(q_0)$
$(label\ q_0)$	$(label\ q_0)$	$legal(q_0)$
$(fetch_l\ i\ q_0)$	$join(q_0)$	$legal(q_0)$
$(brf\ q_0\ q_1)$	$join(join(q_0))$	$join(q_0) = (label\ q')$ and $join(q_1) = (goto\ q')$ and $legal(q')$
$(save\text{-}env\ q_0)$	$join(q_0)$	$legal(q_0)$
$(restore\text{-}env\ q_0)$	$join(q_0)$	$legal(q_0)$
$(update\text{-}store\ i\ q_0)$	$join(q_0)$	$legal(q_0)$
$(prim\text{-}apply\ n\ p\ q_0)$	$join(q_0)$	$legal(q_0)$
$(update\text{-}store_i\ i\ q_0)$	$join(q_0)$	$legal(q_0)$
$(prim\text{-}apply_i\ n\ p\ q_0)$	$join(q_0)$	$legal(q_0)$
$(add\text{-}to\text{-}env\ nu\ q_0)$	$join(q_0)$	$legal(q_0)$
$(add\text{-}to\text{-}env_g\ q_0)$	$join(q_0)$	$legal(q_0)$
$(closerecs\ (q_1 \ldots q_n)\ q_0)$	$join(q_0)$	$(\forall i : 0 \leq i \leq n)$ $legal(q_i)$ and $leaf\ (join(q_i))$
$(tail\text{-}call\ n)$	$(tail\text{-}call\ n)$	$true$

Figure 18: Specification of legal combinator code.

We are now ready to describe the linearizer.

$\mathcal{L}$ is the initial linearization function. It takes as input a combinator-code program q representing an entire program or procedure, a code store M, and a pointer p to the next free position in M. It returns two values $\langle M', p'\rangle$, where M' is a code store just like M except that $M' \models_P p \simeq q$, and p' is the next free position in M'.

This informal specification must be refined to clarify the meaning of phrases like "just like" and "next free position." This must be done in a way that will support the eventual induction proof. After considerable experimentation, we arrived at the following formal specification of $\mathcal{L}$:

If q is legal and the join point of q is a leaf then $\mathcal{L}[\![q]\!]Mp = \langle M', p'\rangle$ such that

1. $M =_0^p M'$
2. $p' \geq p$
3. $(\forall\ M'')$ if $M'' =_p^{p'} M'$, then $M \models_P p \simeq q$

The first condition guarantees that the linearizer does not change the locations in M below p, corresponding to the informal statement that p is the first free location in M and M' is "just like" M. The third condition guarantees that the linearization of q lies between locations p and $p' - 1$.

$\mathcal{L}$ is defined by:

$$\mathcal{L}[\![(\mathit{halt})]\!]Mp = \langle M[\mathit{halt}/p], p+1\rangle$$

$$\mathcal{L}[\![(\mathit{tail\text{-}call}\ n)]\!]Mp = \langle M[\mathit{tail\text{-}call}/p, n/p+1], p+2\rangle$$

otherwise

$$\begin{aligned}\mathcal{L}[\![q]\!]Mp = \ & \texttt{let}\ \langle M', p', q'\rangle = \mathcal{L}'[\![q]\!]M(p+2) \\ & \texttt{in}\ \mathcal{L}[\![q']\!]M'p'\end{aligned}$$

The idiom $M[\mathit{tail\text{-}call}/p, n/p+1]$ corresponds to the emission of code: Here location p is loaded with *tail-call* and location $p+1$ is loaded with the number n.

$\mathcal{L}'$ does the major work. It takes as input a combinator-code program q, a code store M, and a code pointer p (the next free position in M). It returns three values $\langle M', p', q'\rangle$, where M' is a modified code store, p' is a pointer into the code store, and q' is a combinator-code program.

The formal specification for $\mathcal{L}'$ is: If q is legal then $\mathcal{L}'[\![q]\!]Mp = \langle M', p', q'\rangle$ such that

1. $M =_0^p M'$
2. $p' \geq p$
3. $q' = join(q)$
4. $(\forall\ M'')$ if $M'' =_p^{p'} M'$ and $M'' \models_P p' \simeq q'$, $M'' \models_P p \simeq q$.

The first and second conditions are similar to those for $\mathcal{L}$. The third condition states that $\mathcal{L}'$ computes $join(q)$. The fourth condition says that if M'' has a linearization of q' beginning at p', and is identical with M' for positions between p and p', then M'' has a linearization of q starting at p. So p' is a position in the code store at which the linearizer expects an image of q' to be placed.

When both sides of the conditional are linearized, the linearizer chooses which copy of q' to linearize (they are guaranteed to be the same because q is legal), and it inserts a goto at the place in the code store where the other branch wanted to put q. This is illustrated in the clause for linearizing a *brf*:

$$
\begin{array}{l}
\mathcal{L}'[\![(brf\ q_0\ q_1)]\!]Mp \\
= \texttt{let*}\ \langle M', p', (goto\ q')\rangle = \mathcal{L}'[\![q_0]\!]M(p+2) \\
\qquad\quad \langle M'', p'', (label\ q')\rangle = \mathcal{L}'[\![q_1]\!]M'(p'+2) \\
\qquad \texttt{in}\ \mathcal{L}'[\![q']\!]M''[brf/p, (p'+2)/(p+1), goto\,/p', p''/(p'+1)]p''
\end{array}
$$

Here q' represents the common code after q_0 and q_1 rejoin each other. Though q' is bound twice in this definition, the two bindings are guaranteed to coincide because of the assumption that $(brf\ q_0\ q_1)$ is legal. If $\mathcal{L}'[\![(brf\ q_0\ q_1)]\!]Mp = \langle M''', p''', q''\rangle$, then M''' will look like

p :	*brf*	
	$p' + 2$	
	...	linearization of q_0
p' :	*goto*	
	p''	
	...	linearization of q_1
p'' :	...	linearization of q'
p''' :		starting address for q''

We give some additional excerpts from the definition of $\mathcal{L}'$ to suggest the other cases.

$\mathcal{L}'[\![(halt)]\!]Mp = \langle M, p, (halt)\rangle$

$$\mathcal{L}'[\![(goto\ q_0)]\!]Mp = \langle M, p, (goto\ q_0)\rangle$$

$$\mathcal{L}'[\![(label\ q_0)]\!]Mp = \langle M, p, (label\ q_0)\rangle$$

$$\mathcal{L}'[\![(fetch_l\ i\, q_0)]\!]Mp = \mathcal{L}'[\![q_0]\!]M[fetch_l\,/p, i/p+1](p+2)$$

$$\begin{array}{l}\mathcal{L}'[\![(closerecs\,(q_1 \ldots q_n)\, q_0)]\!]Mp = \\ \quad \texttt{let}\ \langle M', (p_1 \ldots p_n), p'\rangle = \mathcal{L}^*[\![(q_1 \ldots q_n)]\!]M(p+n+3) \\ \qquad \texttt{in}\ \mathcal{L}'[\![q_0]\!]r'[closerecs\,/p, p'/p+1, n/p+2, \\ \qquad\qquad\qquad p_1/p+3, \ldots, p_n/(p+n+2)]p'\end{array}$$

$$\mathcal{L}'[\![(tail\text{-}call\ n)]\!]Mp = \langle M, p, (tail\text{-}call\ n)\rangle$$

The last valuation, $\mathcal{L}^*$, is used for linearizing a sequence of combinator-code programs. It takes as input a list of combinator-code programs $(q_1 \ldots q_n)$, a code store M, and a code pointer p (the next free position in M) and returns three values $\langle M', (p_1 \ldots p_n), p'\rangle$. M' is a code store just like M except that $(\forall i : 1 \leq i \leq n)\ M' \models_P p_i \simeq q_i$. Again, p' is the next free position in M'. $\mathcal{L}^*$ is used to linearize the procedure definitions in a *closerecs.*

The formal specification of $\mathcal{L}^*$ is: if $(\forall i : 1 \leq i \leq n)$ q_i is legal and the join point of q_i is a leaf then $\mathcal{L}^*[\![(q_1 \ldots q_n)]\!]Mp = \langle M', (p_1 \ldots p_n), p'\rangle$ such that

1. $M =_0^p M'$
2. $p' \geq p$
3. $(\forall\ M : M =_p^{p'} M')$, $(\forall i : 1 \leq i \leq n)\ M \models_P p_i \simeq q_i$

$\mathcal{L}^*$ is defined by:

$$\mathcal{L}^*[\![\langle\rangle]\!]Mp = \langle M, \langle\rangle, p\rangle$$

$$\begin{array}{rl}\mathcal{L}^*[\![q :: (q_1\ \ldots q_n)]\!]Mp = \texttt{let}^* & \langle M', p'\rangle = \mathcal{L}[\![q]\!]Mp \\ & \langle r'', (p_1 \ldots p_n), p''\rangle = \mathcal{L}^*[\![(q_1 \ldots q_n)]\!]M'p' \\ \texttt{in} & \langle r'', p :: (p_1 \ldots p_n), p''\rangle\end{array}$$

The correctness of the linearizer is given as follows:

Theorem 11 *$\mathcal{L}$, $\mathcal{L}'$ and $\mathcal{L}^*$ satisfy their specifications.*

Proof: By induction on the definitions of $\mathcal{L}$, $\mathcal{L}'$ and $\mathcal{L}^*$.

∎

9. Implementation

The VLISP PreScheme compiler was implemented as two separate programs. The front end translates VLISP PreScheme into Pure PreScheme and the back end translates Pure PreScheme into code for the Motorola 68000.[3] The front end was implemented using about 4300 lines of code and the back end used 2300 lines. To facilitate bootstrapping as described in [8, Section 8], they were both written in the dialect of Scheme accepted by the VLISP Scheme byte-code compiler.

The back end used the 68000 as a target because it was the architecture most readily available to us at the start of the project. Later, the compiler was retargeted to produce SPARC code with very little effort.

Representing our abstract machines with the 68000 requires a mapping similar to the ones just detailed. This section informally describes that mapping and then describes the code generated for the VLISP Byte-Code Interpreter by the compiler.

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 stack s is modeled by the 68000 with the user stack, and its stack pointer, sp, is modeled by the 68000 stack pointer `SP` (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, the 68000 user stack grows downward rather than upward. Furthermore, one unit of immediate data is represented by four bytes of 68000 memory. Therefore the stack pointer manipulations and the linearization algorithm needed to be altered accordingly.

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 stored-program 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. The correctness of such

[3]The front end and the back end were developed at two different sites. The two programs were not merged solely because we did not take the time to resolve conflicting usages of Scheme program variables.

caching strategies can be performed as part of the linearization phase; see [17] for details. This simple minded register allocation technique was quite effective. Program execution speed increased by 35% when the technique was implemented.

Representation of the stored-program 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, is simulated by the 68000 heap, primarily via operating system calls.

We used a program that computes Fibonacci numbers to give an initial benchmark for the back end of the compiler on an unloaded Sun 3/60. The Pure PreScheme program uses an explicit array to model the recursion stack. It was compared with an implementation written in C and compiled with gcc version 1.37 using the -O switch. In the C version, tail-recursion is replaced by explicit goto's, and arguments are passed using automatic variables, which gcc allocates in machine registers. The C version also uses `malloc` to allocate the array dynamically, just as the Pure PreScheme version does. We found that that our code used about three times as much CPU time as the C version.

A more realistic test of the VLISP software was constructed as follows. One of the authors wrote a C version of the VLISP Byte-Code Interpreter which was designed only for speed and not structured for verification. It was compiled on a Sun 4 using gcc version 2.1 with the -O2 switch, and the generated assembly code was studied. Any inefficiency observed in the code resulted in a modification to the C source. When the process was completed, all the variables used to describe the state of the byte-code interpreter were placed into machine registers, and the code sequences for the most used byte-code instructions appeared to be optimal.

Tests showed that it took 3.9 times longer to run programs on a Sun 3/60 using the verified PreScheme version of the VLISP Byte-Code Interpreter as compared with the C version compiled with gcc version 1.37 using the -O switch. We consider this a significant accomplishment given the extensive and intricate optimization techniques used by the GNU C compiler.

10. Conclusions

We have presented a verified compiler for PreScheme. Our architecture divided the compiler into three components:

1. A transformational front end that translates source text into a core language called Pure PreScheme.
2. A syntax-directed compiler that translates Pure PreScheme into a combinator-based tree-manipulation language. We call this language *combinator code.*
3. An assembler that translates combinator code into code for an abstract stored-program machine with linear memory for both data and code.

This factorization proved to be a successful decomposition. It allowed 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 [13, 23], were highly complex because they used induction on the construction of reflexive domains as a denotational analog of induction on length of computation. Our proof would likely be just as complex had we adopted this strategy (see [29] 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 between the various phases of the compiler, including errors in syntax, etc. These were artifacts of the circumstance that the various portions of the compiler, including the interface between the front and back ends, 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. [16, 5]). 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 [28].

The proofs of the different stages had rather different styles. The proofs of the transformational front end (Section 3) were most like "traditional" semantics proofs, such as those in [13]. The proofs of the syntax-directed translator (Section 4.6) tended to be more like exercises in λ-calculus. We believe that a fairly simple simplification-based theorem prover could automate most of the cases in this induction. The back-end proofs (Sections 6.6–8) resemble traditional Hoare-style verification proofs, but seem quite stylized and may be amenable to mechanization by systems like the Boyer-Moore theorem prover [4]. Most of the individual proof cases were routine calculations. We believe this is a mark of success for this effort.

Use of a verified compiler need not entail catastrophic performance penalties. On a realistic example, we showed that a verified program compiled by our verified compiler used only about four times more CPU time compared with an unverified version compiled by the optimizing gcc.

We believe that this proof architecture is applicable to other languages as well. There are, however, a number of ways in which this might be done. One might use a modification of the transformational front end to translate the source language into a variant of Pure PreScheme. One could then modify the Pure PreScheme compiler appropriately. Another approach would be to modify the syntax-directed compiler of Sections 4–5 to use the denotational semantics of the source language instead of the semantics of Section 2. In this approach, the front end would be replaced by a source-to-source optimizer.

A key issue in either of these approaches is the incorporation of optimization into the compiler. In real compilers, optimization is performed on many levels. From the point of view of a compiler proof, the job of an optimizer is to derive some non-local invariants that allow the program to be transformed safely. For example, 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.

References

1. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY (1991).
2. Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge and New York (1992).
3. Barendregt, Henk P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam (1981).
4. Boyer, R. and Moore, J. *A Computational Logic*. Academic Press (1979).
5. Boyer, R. S. and Yu, Y. Automated correctness proofs of machine code programs for a commercial microprocessor. In Kapur, D., editor, *Automated Deduction — CADE-11*, 11th International Conference on Automated Deduction, Springer Verlag (1992) 416–430.
6. Clinger, Will. The Scheme 311 compiler: An exercise in denotational semantics. In *1984 ACM Symposium on Lisp and Functional Programming*, The Association for Computing Machinery, Inc., New York (August 1984) 356–364.
7. Courcelle, Bruno. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25 (1983) 95–169.
8. Guttman, J. D. and Swarup, V. VLISP: A verified implementation of scheme. Submitted to LASC.
9. Guttman, Joshua D., Wand, Mitchell, and Ramsdell, John D. Results and conclusions from the vlisp project. *Lisp and Symbolic Computation* (??).
10. Hannan, John. Making abstract machines less abstract. In Hughes, J., editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, Springer-Verlag, Berlin, Heidelberg, and New York (1991) 618–635.

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

12. Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *SIGPLAN Notices* (1986) 219–233. Proceedings of the '86 Symposium on Compiler Construction.

13. Milne, R. and Strachey, Christopher. *A Theory of Programming Language Semantics.* Chapman and Hall, London (1976). Also Wiley, New York.

14. Milner, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 (1978) 348–375.

15. Milner, Robin, Tofte, Mads, and Harper, Robert. *The Definition of Standard ML.* The MIT Press, Cambridge, MA (1990).

16. Moore, J S. *Piton: A Verified Assembly-Level Language.* Technical Report 22, Computational Logic, Inc., Austin, Texas (1988).

17. Oliva, Dino P. *Advice on Structuring Compiler Back Ends and Proving Them Correct.* PhD thesis, Northeastern University (1993).

18. Oliva, Dino P. and Wand, Mitchell. *A Verified Run-Time Structure for Pure PreScheme.* Technical Report NU-CCS-92-27, Northeastern University College of Computer Science (September 1992).

19. Paulson, Laurence C. *ML for the Working Programmer.* Cambridge University Press, Cambridge, Great Britain (1991).

20. Plotkin, Gordon D. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Aarhus University (1981).

21. Ramsdell, J. D., Farmer, W. M., Guttman, J. D., Monk, L. G., and Swarup, V. *The VLISP PreScheme Front End.* M 92B098, The MITRE Corporation (September 1992).

22. Steele, Guy L. *Rabbit: A Compiler for Scheme.* Technical Report 474, MIT AI Laboratory (1978).

23. Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA (1977).

24. Wand, Mitchell. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4, 3 (July 1982) 496–517.

25. Wand, Mitchell. Semantics-directed machine architecture. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages* (1982) 234–241.

26. Wand, Mitchell. Loops in combinator-based compilers. *Information and Control*, 57, 2–3 (May/June 1983) 148–164.

27. Wand, Mitchell. Correctness of procedure representations in higher-order assembly language. In Brookes, S., editor, *Proceedings Mathematical Foundations of Programming Semantics '91*, Springer-Verlag, Berlin, Heidelberg, and New York (1992) 294–311.

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

29. Wand, Mitchell and Wang, Zheng-Yu. Conditional lambda-theories and the verification of static properties of programs. In *Proc. 5th IEEE Symposium on Logic in Computer Science* (1990) 321–332.