The MacScheme Compiler: a denotational proof
of correctness

William D. Clinger
Northeastern University
will@ccs.neu.edu

December 27, 2009

Abstract

Denotational description of a simple code generator for Scheme supports an elementary proof of its correctness. The algorithm was used in a commercial product as the core of a just-in-time compiler that could generate either interpreted byte code or native machine code. This appears to have been the first proof of correctness for a commercial compiler’s code generator.

1 Introduction

Many authors have offered much good advice on structuring compilers and proving them correct [BL69, MW72, Mor73, TWW79, Wan80, Wan82b]. The correctness proof described here applies their advice to a commercial compiler for a real programming language.

This paper describes and proves the correctness of a simple code generation algorithm for Scheme, a statically scoped dialect of Lisp. The algorithm served as the core of an interactive (just-in-time) compiler that generated interpreted byte code or Motorola 68000 machine code.

Most of the correctness proof consists of trivial calculations. This is possible because the meanings of target code instructions are expressed in the same language used to express source meanings, and that language can be wielded as a calculus. The proof is complicated by what amounts to a static type distinction needed to compile primitive operators in line. Previous compiler correctness proofs using static type information (e.g. [Pol81]) assumed that separate presentations of static and dynamic semantics were available \textit{a priori}, but the proof in this paper works from a single standard semantics.

The proof is similar in spirit to that of [Wan82a], but the algorithm was designed and the compiler built before any thought was given to a formal correctness proof. The algorithm is superior to that of [Wan82a] in that it directly produces linear and properly tail-recursive object code of reasonable quality.
Abstract syntax

K ∈ Con   constants
I ∈ Ide   identifiers
E ∈ Exp   expressions

Productions

Exp ::= K
| I
| (set! I E)
| (lambda (I*) E)
| (if E₀ E₁ E₂)
| (E₀ E*)
| (begin E₀ E₁)

Figure 1: Syntax

2 Source Language

Scheme is a statically scoped dialect of Lisp [SS75, SS78, Cli85, RC86]. Parameters are passed by value (but most values are references). Procedures include full continuations, and are first class objects that may be returned as results, stored in variables, and so on.

As seen by the compiler, the core of Scheme comprises constants, identifier references, assignments, lambda expressions, conditional expressions, applications, and sequenced expressions (Figure 1). These language constructs compose well enough that most other syntactic constructs can be expressed as compile-time macros, and most operators can be considered to be predeclared procedures. Even the operator that captures continuations is considered to be a procedure rather than a syntactic construct.

The denotational semantics of Scheme was first described in [MP80]. A simplified and slightly updated version of that semantics is used in this paper; see Figures 2 through 5. The semantics presented in those figures was a predecessor of the semantics given by several reports on Scheme [RC86, CR91, KCR98].

In two respects, however, the semantics used here has been fudged a bit to reflect the compiler’s quirks. First of all, Scheme does not specify the order of evaluation, but the MacScheme compiler, for historical reasons, evaluates arguments right to left. Secondly, the compiler translates calls to primitive operators in line for the sake of efficiency. In most languages, this is easy because the primitive operators are constants. In Scheme, however, + is merely an identifier
Value domains

\[ \alpha \in L \text{ locations} \]
\[ \nu \in N \text{ natural numbers} \]
\[ T = \{ \text{false, true} \} \text{ booleans} \]
\[ \phi \in F = E^* \rightarrow K \rightarrow C \text{ procedure values} \]
\[ G \text{ other expressed values} \]
\[ \epsilon \in E = F + G \text{ expressed values} \]
\[ \delta \in D = F + L \text{ denoted values} \]
\[ \beta \in V = E \text{ stored values} \]
\[ \sigma \in S = L \rightarrow (V \times T) \text{ stores} \]
\[ \rho \in U = \text{Ide} \rightarrow D \text{ environments} \]
\[ \theta \in C = S \rightarrow A \text{ command continuations} \]
\[ \kappa \in K = E \rightarrow C \text{ expression continuations} \]

Figure 2: Domains

Notation

\[ \langle \ldots \rangle \text{ sequence formation} \]
\[ s \downarrow k \text{ } k \text{th member of the sequence } s \text{ (1-based)} \]
\[ \#s \text{ length of sequence } s \]
\[ s \sqcup t \text{ concatenation of sequences } s \text{ and } t \]
\[ s \triangledown k \text{ drop the first } k \text{ members of sequence } s \]
\[ \text{takefirst } k \text{ } s \text{ take the first } k \text{ members of sequence } s \]
\[ t \rightarrow a, b \text{ McCarthy conditional “if } t \text{ then } a \text{ else } b” \]
\[ \rho[x/i] \text{ substitution “} \rho \text{ with } x \text{ for } i” \]
\[ x \text{ in } D \text{ injection of } x \text{ into domain } D \]
\[ x \mid D \text{ projection of } x \text{ to domain } D \]

Figure 3: Notation
Semantic functions

\[ K : \text{Con} \rightarrow E \]
\[ E : \text{Exp} \rightarrow U \rightarrow K \rightarrow C \]
\[ E^* : \text{Exp}^* \rightarrow U \rightarrow (E^* \rightarrow C) \rightarrow C \]

\[ E[K] = \lambda \rho \kappa. \kappa (K[K]) \]
\[ E[I] = \lambda \rho \kappa. (\text{lookup } \rho \ I) \in F \rightarrow \kappa ((\text{lookup } \rho \ I) | F \text{ in } E), \]
\[ \text{hold } ((\text{lookup } \rho \ I) | L) \kappa \]

\[ E[(\text{set}! \ I \ E)] = \lambda \rho \kappa. E[E] \rho (\lambda \epsilon. (\text{lookup } \rho \ I) \in F \rightarrow \text{wrong}, \text{assign} (\text{lookup } \rho \ I) \epsilon (\kappa \epsilon)) \]

\[ E[(\text{lambda} (I^*) \ E)] = \lambda \rho \kappa. \kappa ((\lambda \epsilon^* \kappa'. \# \epsilon^* = \# I^* \rightarrow \text{tievals} (\lambda \delta^*. E[E] (\rho[\delta^*/I^*]) \kappa') \epsilon^*, \text{wrong}) \in E) \]

\[ E[(\text{if} \ E_0 \ E_1 \ E_2)] = \lambda \rho \kappa. E[E_0] \rho (\lambda \epsilon. \text{truth } \epsilon \rightarrow E[E_1] \rho \kappa, E[E_2] \rho \kappa) \]

\[ E[(E_0 \ E^*)] = \lambda \rho \kappa. E^*[E^*] \rho (\lambda \epsilon. E[E_0] \rho (\lambda \epsilon. \text{apply } \epsilon \epsilon^*) \rho (\lambda \epsilon. E[E_1] \rho \kappa) \]

\[ E^*[\text{[}] = \lambda \rho \psi. \psi \text{[]} \]

\[ E^*[E^* \ E] = \lambda \rho \psi. E[E] \rho (\lambda \epsilon. E^*[E^*] \rho (\lambda \epsilon. \psi (\epsilon^* \notin (\epsilon)))) \]

Figure 4: Standard semantics
Auxiliary functions

\(\text{wrong} : S \rightarrow A\)

\(\text{new} : S \rightarrow \{L + \text{error}\}\)

\(\text{hold} : L \rightarrow K \rightarrow C\)

\(\text{assign} : L \rightarrow E \rightarrow C \rightarrow C\)

\(\text{update} : L \rightarrow V \rightarrow S \rightarrow S\)

\(\text{tievals} : (D^* \rightarrow C) \rightarrow E^* \rightarrow C\)

\(\text{apply} : E \rightarrow E^* \rightarrow K \rightarrow C\)

\(\text{lookup} : U \rightarrow \text{Ide} \rightarrow D\)

[Definitions of \text{wrong} and \text{new} deliberately omitted.]

\(\text{hold} = \lambda \alpha \kappa . \lambda \sigma . \kappa (\sigma \alpha \uparrow 1 | E) \sigma\)

\(\text{assign} = \lambda \alpha \epsilon \theta . \lambda \sigma . \theta (\text{update} \alpha (\epsilon \text{ in } V) \sigma)\)

\(\text{update} = \lambda \alpha \beta \sigma . \lambda \alpha' . \alpha' = \alpha \rightarrow \langle \beta, \text{true} \rangle, \sigma \alpha'\)

\(\text{tievals} =\)

\(\lambda \phi \epsilon^* \sigma . \# \epsilon^* = 0 \rightarrow \phi (\_ \sigma),\)

\(\text{new} \sigma \in L \rightarrow \text{tievals} (\lambda \delta^* . \phi ((\text{new} \sigma | L \text{ in } D) \triangledown \delta^*))\)

\(\langle \epsilon^* \uparrow 1 \rangle\)

\(\langle \text{update} (\text{new} \sigma | L) (\epsilon^* \downarrow 1 \text{ in } V) \sigma), \text{wrong} \sigma\)

\(\text{apply} =\)

\(\lambda \epsilon \epsilon^* \kappa . \epsilon \in F \rightarrow (\epsilon | F) \epsilon^* \kappa, \text{wrong}\)

\(\text{lookup} = \lambda \rho \text{I} . \rho \text{I}\)

Figure 5: Auxiliary functions
denoting a location that happens to hold the primitive procedure for addition. The MacScheme compiler gives programmers the option of using an initial environment that reflects that model, but defaults to an initial environment in which $+$ denotes the primitive procedure directly.

The compiler is indifferent to the primitive procedures and data types, so they are not described in Figure 2. The data types may be specified by fixing a domain equation for the domain $G$ of “other expressed values”. For example, a dialect of Scheme with generic numbers, symbols, mutable pairs and vectors, and immutable characters and strings could use the domain equation

$$G = R + Q + L \times L + L^* + H + H^*$$

where $R$, $Q$, and $H$ are independently specified domains of numbers, symbols, and characters. The primitive and predeclared procedures may be specified by describing the initial environment $\rho^{\text{init}}$ and the initial store $\sigma^{\text{init}}$.

2.1 Compositionality

In the early 1980s, when the MacScheme compiler was designed, implemented and proved correct, Scheme officially had no eval operator. In particular, neither eval nor load are described by the IEEE/ANSI standard for Scheme [IEE91]. That is significant because the compiler correctness theorem of this paper depends on the homomorphic nature of the semantics as codified by the principle of compositionality: the semantics of a program fragment is an independently definable function of the semantics of its syntactic constituents. eval violates the principle of compositionality because the semantics of an expression involving eval depends on the “semantics” of data structures constructed at run time.

Retrofitting eval to a compositional semantics or compiler is a delicate business. The usual approach is to ignore eval when describing the semantics or compiler, and to add eval later as a component of the environment or module system. That approach won’t work unless interactions between eval and the compositional semantics are sufficiently restricted, but it works for the restricted semantics of eval as described by the R5RS and R6RS [KCR98, SDF+09], and it also works for at least one semantics for load that is permitted by the R3RS, R4RS, and R5RS [RC86, CR91, KCR98].

Compositionality should be considered by anyone wishing to add eval to Scheme-like languages. An eval that evaluates in the current lexical environment cannot be added as a predeclared procedure, since procedures do not have access to such environments. An eval that evaluates in a fixed environment or takes a representation of an environment as an argument could be a predeclared procedure, but in order to preserve the principle of compositionality its semantics would have to be definable without reference to the semantic function $E$. One approach is to define such an eval not in terms of $E$ but in terms of a “copy” $E'$. (Think of $E$ as the compiler’s semantics and $E'$ as the interpreter’s semantics.) $E'$ need not be compositional, so the semantics of eval in
Compile-time and run-time environments

\[ \rho \in U = \text{Ide} \rightarrow D \] environments
\[ \rho_C \in U_C = \text{Ide} \rightarrow D_C \] compile-time environments
\[ \rho_R \in U_R = D_C \rightarrow D \] run-time environments
\[ \delta_C \in D_C = F + \text{Ide} + (N \times N) \] lexical addresses

Operations on environments

\[ \text{rep}_C : U \rightarrow U_C \]
\[ \text{rep}_R : U \rightarrow U_R \]
\[ \text{extends}_C : U_C \rightarrow \text{Ide}^* \rightarrow U_C \]
\[ \text{extends}_R : U_R \rightarrow D^* \rightarrow U_R \]

\[ (\text{rep}_R \rho) \circ (\text{rep}_C \rho) = \rho \]
\[ (\text{extends}_R \rho_R \delta^*) \circ (\text{extends}_C \rho_C \text{I}^*) = (\rho_R \circ \rho_C) [\delta^*/\text{I}^*] \]

A run-time environment \( \rho_R \) is reasonable if and only if

1. \( (\delta_C \in F) = (\rho_R \delta_C \in F) \)
2. If \( \delta_C \in F \) then \( \rho_R \delta_C = (\delta_C \mid F \in D) \).

Figure 6: Environments

\( \mathcal{E}' \) may without difficulty refer circularly to \( \mathcal{E}' \). Alternatively, one may imagine an infinite sequence of functions \( \mathcal{E}, \mathcal{E}', \mathcal{E}'', \mathcal{E}''', \ldots \) as in [Smi84].

3 Target Machine

The MacScheme compiler algorithm as described here produces code for a hypothetical and rather abstract Scheme machine. The machine’s architecture is guided but not determined by the semantics of Scheme.

If correctness were the only issue, the machine’s design would be a trivial matter. Its instructions could be taken to correspond to operations of the multi-sorted algebra generated by the semantics [Mos82]. The difficulty lies in designing a machine that can easily be proved correct and will also run acceptably fast.

An implementation’s correctness depends on the harmony and correctness of many parts operating at different levels of abstraction. Its designer must choose representations for each level that are sufficiently concrete to express what needs to be expressed at that level and at the same time sufficiently abstract to be
tractable and to preserve design flexibility at lower levels.

For programming languages the highest level of abstraction speaks of environments, continuations, stores, expressed values, and so on. The next highest level of abstraction begins to speak of implementation concepts. For example, compile-time and run-time environments may be introduced, and environments may be represented as compositions of compile-time and run-time environments. (See Figure 6.) At this level of abstraction there is no need to commit to any particular representation of these new objects, so they are better treated as functions than as data structures.

For the MacScheme compiler this level introduces code segments as well. Code segments are taken to be values in the domain

\[ P = E \rightarrow E^* \rightarrow U_R \rightarrow K \rightarrow C \]

where \( U_R \) is the domain of run-time environments (Figure 6). The domain \( P \) determines the register architecture of the Scheme machine, since it is interpreted as the domain of maps from register contents to command continuations. Thus the Scheme machine has an accumulator capable of holding an expressed value, a stack of expressed values, a register to hold the run-time environment, and a register to hold the expression continuation. This was an arbitrary design choice. Other register architectures could have been used instead, as discussed in Section 9.2.

Continuations can now be represented in terms of code segments, sequences of expressed values, run-time environments, and continuations. The general form of a continuation in this new representation is

\[ \lambda \epsilon \cdot \pi \epsilon \epsilon^* \rho_R \kappa \]

where \( \pi \) is a code segment. The representation chosen for continuations reflects the fact that during a computation the effective continuation may actually be made up of the contents of all registers, and only the most long-term portion is held in the so-called continuation register or control stack.

The instructions of the Scheme machine are specified by detailing a continuation semantics for each instruction. (See Figure 7.) The general form of an instruction is a mapping from zero or more parameters (or operands) to a mapping from register contents to command continuations.

This style of semantic description is superior to a register transfer language in two respects. The first is that proofs in continuation semantics tend to be more local, at a higher level, and hence more tractable, than proofs in a state transition semantics, and the second is that the exact effects of errors and exceptions are more easily accommodated within the framework of continuation semantics. At the same time a register transfer language description can easily be obtained from the continuation semantics should such a description be needed by a lower level of abstraction.

To compile tail recursion properly it is important to distinguish between procedure invocation, continuation invocation, and continuation creation. The
MacScheme machine instructions

\[
\text{save} = \lambda \pi_1 \pi_2 . \lambda e \ast \rho R \kappa \cdot \pi_1 \epsilon (\rho R (\lambda e . \pi_2 \epsilon e \ast \rho R \kappa))
\]

\[
\text{restore} = \lambda e \ast \rho R \kappa . \kappa \epsilon
\]

\[
\text{const} = \lambda 0 \pi . \lambda e \ast \rho R \kappa . \pi e_0 \epsilon e \ast \rho R \kappa
\]

\[
\text{fetch} = \lambda \pi . \lambda e \ast \rho R \kappa \cdot \text{hold} (\rho R (I \text{ in } D_C) | L) (\lambda e . \pi e e \ast \rho R \kappa)
\]

\[
\text{lexical} = \lambda x \pi . \lambda e \ast \rho R \kappa \cdot \text{hold} (\rho R (x \text{ in } D_C) | L) (\lambda e . \pi e e \ast \rho R \kappa)
\]

\[
\text{set} = \lambda I \pi . \lambda e \ast \rho R \kappa \cdot \text{assign} (\rho R (I \text{ in } D_C) | L) e (\pi e e \ast \rho R \kappa)
\]

\[
\text{setlex} = \lambda x \pi . \lambda e \ast \rho R \kappa \cdot \text{assign} (\rho R (x \text{ in } D_C) | L) e (\pi e e \ast \rho R \kappa)
\]

\[
\text{lambda} = \lambda \nu \pi_1 \pi_2 . \lambda e \ast \rho R \kappa
\]

\[
\pi_2 (\langle \lambda e \ast \kappa' . \#e^* = \nu \rightarrow \text{tievals} (\lambda \delta^* . \pi_1 (\text{false in } E) \langle \rangle (\text{extendsR } \rho R \delta^*) \kappa') e^* , \text{wrong}) \text{ in } E \rangle ) e \ast \rho R \kappa
\]

\[
\text{if} = \lambda \pi_1 \pi_2 . \lambda e \ast \rho R \kappa \cdot \text{truish e} \rightarrow \pi_1 \epsilon e \ast \rho R \kappa , \pi_2 e e e \ast \rho R \kappa
\]

\[
\text{invoke} = \lambda e \ast \rho R \kappa \cdot \text{apply } e \ast e \ast \rho R \kappa
\]

\[
\text{push} = \lambda \pi . \lambda e \ast \rho R \kappa \cdot \pi e (\langle e \rangle \langle e \ast \rangle ) e \ast \rho R \kappa
\]

\[
\text{op} = \lambda \nu \phi \pi . \lambda e \ast \rho R \kappa \cdot \text{apply} (\phi \text{ in } E) (\langle e \rangle \langle e \rangle (\text{takefirst } \nu e \ast)) (\lambda e . \pi e (e \ast ) \nu e \ast \rho R \kappa)
\]

\[
\text{illegal} = \lambda e \ast \rho R \kappa . \text{wrong}
\]

Figure 7: Semantics of MacScheme machine instructions
Scheme machine instructions that correspond to those three notions are *invoke*, *restore*, and *save* respectively.

The *fetch* and *lexical* instructions are identical except for the types of their parameters, and likewise the *set* and *setlex* instructions. The difference in parameter types may later become important when these instructions are encoded as bit strings or as native machine instructions.

4 Two Lemmas

The following lemmas establish that evaluation of a sequence of operands pushes the values of those operands onto the expression stack while leaving the rest of the stack unchanged.

**Lemma 1** For all values of the free variables:

\[
\mathcal{E}^*[E^*] \rho (\lambda e^* \cdot f(\text{takefirst} \ (\#E^*) \ (e^* \ § x))
\]

\[
((e^* \ § x) \uparrow (\#E^*))
\]

\[
= \mathcal{E}^*[E^*] \rho (\lambda e^* \cdot f e^* \ x)
\]

**Proof.** By induction on (\#E^*). The base case:

\[
\mathcal{E}^*[E^*] \rho (\lambda e^* \cdot f(\text{takefirst} \ 0 \ (e^* \ § x))
\]

\[
((e^* \ § x) \uparrow 0))
\]

\[
= f(\text{takefirst} \ 0 \ ((\ ) \ § x)) \ ((\ ) \ § x) \uparrow 0))
\]

\[
= f(\ ) \ x
\]

\[
= \mathcal{E}^*[E^*] \rho (\lambda e^* \cdot f e^* \ x)
\]

For the induction case, let \#E^* = n + 1 and E^* = E_0^* E_0:

\[
\mathcal{E}^*[E_1^* \ E_0^*] \rho (\lambda e^* \cdot f(\text{takefirst} \ (n + 1) \ (e^* \ § x))
\]

\[
((e^* \ § x) \uparrow (n + 1)))
\]

\[
= \mathcal{E}[E_0^*] \rho (\lambda e^* \cdot \mathcal{E}^*[E_1^*] \rho (\lambda e_1^* \cdot (\lambda e^* \cdot f(\text{takefirst} \ (n + 1) \ (e^* \ § x))
\]

\[
((e^* \ § x) \uparrow (n + 1)))
\]

\[
= \mathcal{E}[E_0^*] \rho (\lambda e^* \cdot \mathcal{E}^*[E_1^*] \rho (\lambda e_1^* \cdot (\lambda e^* \cdot f(\text{takefirst} \ (n + 1) \ ((e_1^* \ § (e) \ § x))
\]

\[
(((e_1^* \ § (e) \ § x) \uparrow (n + 1)))
\]

\[
= \mathcal{E}[E_0^*] \rho (\lambda e^* \cdot \mathcal{E}^*[E_1^*] \rho (\lambda e_1^* \cdot (\lambda e^* \cdot f(\text{takefirst} \ n \ (e_1^* \ § (e) \ § x))
\]

\[
(((e_1^* \ § (e) \ § x) \uparrow n))
\]

\[
= \mathcal{E}[E_0^*] \rho (\lambda e^* \cdot \mathcal{E}^*[E_1^*] \rho (\lambda e_1^* \cdot (\lambda ab \cdot f(a \ § (\text{takefirst} \ b))
\]

\[
(b \uparrow 1))
\]

\[
= \mathcal{E}[E_0^*] \rho (\lambda e^* \cdot \mathcal{E}^*[E_1^*] \rho (\lambda e_1^* \cdot (\lambda ab \cdot f(a \ § (\text{takefirst} \ b))
\]

\[
(b \uparrow 1))
\]

\[
= \mathcal{E}[E_0^*] \rho (\lambda e^* \cdot \mathcal{E}^*[E_1^*] \rho (\lambda e_1^* \cdot (\lambda ab \cdot f(a \ § (\text{takefirst} \ b))
\]

\[
(b \uparrow 1))
\]
5 The Compiler

The MacScheme compiler algorithm introduces an embryonic notion of static type by separating compile-time denoted values into three subdomains: primitive procedures, global variables represented as identifiers for the sake of the
Compiling functions

\[ t : \text{Exp} \rightarrow U_C \rightarrow P \rightarrow P \quad \text{expression compiler, tidy} \]
\[ e : \text{Exp} \rightarrow U_C \rightarrow P \rightarrow P \quad \text{expression compiler} \]
\[ e^* : \text{Exp}^* \rightarrow U_C \rightarrow P \rightarrow P \quad \text{argument list compiler} \]

t[K] = e[K]

t[I] = e[I]

t[(\text{set! } I \ E)] = e[(\text{set! } I \ E)]

t[(\text{lambda } (I^*) \ E)] = e[(\text{lambda } (I^*) \ E)]

t[(\text{if } E_0 \ E_1 \ E_2)] = \lambda \rho_C \pi \cdot t[E_0] \rho_C (if (t[E_1] \rho_C \pi) (t[E_2] \rho_C \pi))

t[(E_0 \ E^*)] = \lambda \rho_C . \text{primop } [E_0] (\#E^*) \rho_C \rightarrow e[[E_0 \ E^*] \rho_C, \ \text{save} (e[[E_0 \ E^*] \rho_C \text{restore})

t[(\text{begin } E_0 \ E_1)] = \lambda \rho_C \pi \cdot t[E_0] \rho_C (t[E_1] \rho_C \pi)

Figure 8: Code generation (part 1 of 2)
\[ e[K] = \lambda \rho C \cdot \text{const}(K[K]) \]
\[ e[I] = \lambda \rho C \cdot (\text{lookup}_C \rho C I) \in F \rightarrow \text{const}((\text{lookup}_C \rho C I) \mid F \text{ in } E), \]
\[ (\text{lookup}_C \rho C I) \in \text{Ide} \rightarrow \text{fetch}((\text{lookup}_C \rho C I) \mid \text{Ide}), \]
\[ \text{lexical}((\text{lookup}_C \rho C I) \mid (N \times N)) \]
\[ e[(\text{set}! I E)] = \lambda \rho C \pi \cdot t[E] \rho C \]
\[ ((\text{lookup}_C \rho C I) \in F \rightarrow \text{illegal}, \]
\[ (\text{lookup}_C \rho C I) \in \text{Ide} \rightarrow \text{set}((\text{lookup}_C \rho C I) \mid \text{Ide}) \pi, \]
\[ \text{setlex}((\text{lookup}_C \rho C I) \mid (N \times N)) \pi) \]
\[ e[(\text{lambda} (I^*) E)] = \lambda \rho C \cdot \text{lambda}(\#I^*)(e[E] \text{ (extends}_C \rho_C I^*) \text{ restore}) \]
\[ e[(\text{if} E_0 E_1 E_2)] = \lambda \rho C \pi \cdot t[E_0] \rho C (if(e[E_1] \rho C \pi) (e[E_2] \rho C \pi)) \]
\[ e[(E_0 E^*)] = \lambda \rho C \pi \cdot \text{primop}[E_0](\#E^*) \rho C \rightarrow p[[E_0 E^*]] \rho C \pi, \]
\[ e^*[E^*] \rho C (t[E_0] \rho C \text{ invoke}) \]
\[ e[(\text{begin} E_0 E_1 I)] = \lambda \rho C \pi \cdot t[E_0] \rho C (e[E_1] \rho C \pi) \]
\[ e^*[\text{[]}] = \lambda \rho C \pi \cdot \pi \]
\[ e^*[E^* E] = \lambda \rho C \pi \cdot t[E] \rho C (\text{push}(e^*[E^*] \rho C \pi)) \]
\[ p[([E E^*])] = \lambda \rho C \pi \cdot e^*[E^*] \rho C (t[E] \rho C (op(\#E^*)(\text{lookup}_C \rho C I \mid F))) \]
\[ \text{primop}[E] = \lambda \nu \rho C \cdot E \in \text{Ide} \rightarrow \nu \neq 0 \rightarrow \text{lookup}_C \rho_C (E \mid \text{Ide}) \in F, \text{false}, \text{false} \]

Figure 9: Code generation (part 2 of 2)
linker, and local variables represented as a block (or rib) number and offset. Three inference rules needed in the correctness proof require that environment components behave reasonably on these subdomains; for example, compile-time environments must be strict and run-time environments must be identities on the subdomain of primitive procedures. (See Figure 6.) The easiest way to enforce these conditions is to require that the operations on environment components preserve them.

These conditions are necessary because the static information has been extracted from a standard semantics that has no notion of compile time or run time. Similar conditions would be needed to prove the correctness of a compiler algorithm for a statically typed language. In practice, however, most work with statically typed languages assumes the separation into static and dynamic semantics has already been done correctly.

Figures 8 and 9 define the code generation algorithm in terms of three maps:

\[
\begin{align*}
t &: \text{Exp} \rightarrow \text{U} \rightarrow \text{P} \\
e &: \text{Exp} \rightarrow \text{U} \rightarrow \text{P} \\
e^* &: \text{Exp}^* \rightarrow \text{U} \rightarrow \text{P} \\
\end{align*}
\]

Intuitively \( e \) is a compiling function that takes an expression \( E \), a compile-time environment \( \rho_C \), and a code segment \( \pi \) to follow the code for \( E \), and produces a code segment consisting of the code for \( E \) followed by \( \pi \). The reason \( \pi \) is given as an argument to \( e \) is that in some cases \( \pi \) will represent dead code and can be ignored by \( e \); this can happen only when \( \pi \) is a single \text{restore} instruction or when the programmer attempts to assign to a procedure bound directly to a procedure.

### 6 Correctness Theorem

The code generated for a top-level expression \( E \) denotes

\[
e[[E]]^{\text{init}} \rho_C^{\text{init}} \text{restore}
\]

where \( \rho_C^{\text{init}} \) is the compile-time component of the initial environment. Part B of the following theorem asserts that this code, when applied to a random value \( \epsilon \) in the accumulator, an empty stack, and a reasonable run-time environment, will yield the standard meaning of the expression in the environment formed by composing the run-time and compile-time environments. This establishes the algorithm’s correctness.

The peculiar hypothesis of part C is needed because argument evaluation destroys the contents of the accumulator. It is easy to verify that the hypothesis is satisfied whenever part C is used in the induction; see Facts 1 and 2 in the proof. A similar device can be used to ignore other harmless side effects such as to condition code settings on many hardware architectures.
Theorem 3 If $\rho_C$ is strict, $\rho_R$ is a reasonable run-time environment, and the operations on compile-time and run-time environments preserve strictness and reasonableness, respectively, then

A. Tidiness.

$$t[E] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa = \mathcal{E}[E] (\rho_R \circ \rho_C)(\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa)$$

B. Correctness.

$$e[E] \rho_C \text{restore } \epsilon \langle \rangle \rho_R \kappa = \mathcal{E}[E] (\rho_R \circ \rho_C) \kappa$$

C. Evlis. If $\forall \epsilon \epsilon' . (\pi \epsilon = \pi \epsilon')$ then

$$e^*[E^*] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa = \mathcal{E}^*[E^*] (\rho_R \circ \rho_C)(\lambda \epsilon^* . \pi \epsilon (\epsilon \epsilon^* \rho_R \kappa))$$

Proof. The three parts of the theorem are proved by a simultaneous structural induction on $E$ and $E^*$. The proofs of all three parts of the theorem can rely on all three parts holding for smaller $E$ and $E^*$. The proof of part B can also rely on part A holding for the same $E$, but part A cannot rely on part B except at smaller sizes.

The proof of part A will be presented first, followed by the proof of part B, followed by the proof of part C. This is just a matter of presentation, but it reflects the ordering between parts A, B, and C.

Proof of part A:

$$t[K] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa = e[K] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa = \text{const} (K[K]) \pi \epsilon \epsilon^* \rho_R \kappa = \pi (K[K]) \epsilon^* \rho_R \kappa = (\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa) (K[K]) = \mathcal{E} [K] (\rho_R \circ \rho_C) (\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa)$$

$$t[I] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa = e[I] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa = (\rho_C I \in F \rightarrow \text{const} (\rho_C I | F \in E), \rho_C I \in \text{Ide} \rightarrow \text{fetch} (\rho_C I | \text{Ide}), \rho_C I \in \text{Ide} \rightarrow \text{hold} (\rho_R (\rho_C I | \text{Ide} \in \text{D}_C) | L) (\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa), \rho_C I \in \text{Ide}) \epsilon^* \rho_R \kappa = \rho_C I \in F \rightarrow \epsilon^* (\rho_C I | F \in E) \epsilon^* \rho_R \kappa, \rho_C I \in \text{Ide} \rightarrow \text{fetch} (\rho_R (\rho_C I | \text{Ide} \in \text{D}_C) | L) (\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa), \rho_C I \in \text{Ide} \rightarrow \text{fetch} (\rho_R (\rho_C I | \text{Ide} \in \text{D}_C) | L) (\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa), \rho_C I \in \text{Ide} \rightarrow \text{hold} (\rho_R (\rho_C I | \text{Ide} \in \text{D}_C) | L) (\lambda \epsilon . \pi \epsilon \epsilon^* \rho_R \kappa)$$

15
\[ t[(\text{set} \ I \ E)] \rho C \pi \in \epsilon^* \rho R \kappa \]

\[ = \rho R(\rho C I) \in F \rightarrow \pi (\rho R(\rho C I) | F \text{ in } E) \epsilon^* \rho R \kappa, \]

\[ \rho C I \in \text{Idc} \rightarrow \text{hold}(\rho R(\rho C I) | L) (\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa), \]

\[ \text{hold}(\rho R(\rho C I) | L) (\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa) \]

(by reasonableness of \( \rho_R \))

\[ = (\rho R \circ \rho C) I \in F \rightarrow \pi ((\rho R \circ \rho C) I | F \text{ in } E) \epsilon^* \rho R \kappa, \]

\[ \text{hold}((\rho R \circ \rho C) I | L) (\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa) \]

(by strictness of \( \rho C \))

\[ = \text{lookup}(\rho R \circ \rho C) I \in F \rightarrow (\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa)(\text{lookup}(\rho R \circ \rho C) I | F \text{ in } E), \]

\[ \text{hold}(\text{lookup}(\rho R \circ \rho C) I | L) (\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa) \]

\[ = \mathcal{E} [I] (\rho R \circ \rho C)(\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa) \]

\[ t[I \ E] \rho C \pi \in \epsilon^* \rho R \kappa \]

\[ = t[I] \rho C (\rho C I \in F \rightarrow \text{illegal}, \]

\[ \rho C I \in \text{Idc} \rightarrow \text{set}(\rho C I | \text{Idc}) \pi, \]

\[ \text{setlex}(\rho C I | (N \times N)) \pi) \epsilon^* \rho R \kappa \]

\[ = \mathcal{E} [I] (\rho R \circ \rho C) \]

\[ (\lambda \epsilon . (\rho C I \in F \rightarrow \text{illegal}, \]

\[ \rho C I \in \text{Idc} \rightarrow \text{set}(\rho C I | \text{Idc}) \pi, \]

\[ \text{setlex}(\rho C I | (N \times N)) \pi) \epsilon^* \rho R \kappa \]

(by induction hypothesis)

\[ = \mathcal{E} [I] (\rho R \circ \rho C) \]

\[ (\lambda \epsilon . \rho C I \in F \rightarrow \text{wrong}, \]

\[ \rho C I \in \text{Idc} \rightarrow \text{assign}(\rho R(\rho C I | \text{Idc} in D C) | L) (\pi \epsilon \in \epsilon^* \rho R \kappa), \]

\[ \text{assign}(\rho R(\rho C I | (N \times N) in D C) | L) (\pi \epsilon \in \epsilon^* \rho R \kappa)) \]

(by strictness of \( \rho C \))

\[ = \mathcal{E} [I] (\rho R \circ \rho C) \]

\[ (\lambda \epsilon . \rho C I \in F \rightarrow \text{wrong}, \]

\[ \text{assign}(\text{lookup}(\rho R \circ \rho C) I | L) (\pi \epsilon \in \epsilon^* \rho R \kappa)) \]

(by strictness of \( \rho C \))

\[ = \mathcal{E} [I] (\rho R \circ \rho C) \]

\[ (\lambda \epsilon . \text{lookup}(\rho R \circ \rho C) I \in F \rightarrow \text{wrong}, \]

\[ \text{assign}(\text{lookup}(\rho R \circ \rho C) I | L) (\pi \epsilon \in \epsilon^* \rho R \kappa)) \]

(by reasonableness of \( \rho_R \))

\[ = \mathcal{E} [set ! I E]) (\rho R \circ \rho C)(\lambda \epsilon \cdot \pi \in \epsilon^* \rho R \kappa) \]

16
\[ t\left[ \lambda (I^*) \ E \right] \rho_C \pi \epsilon^* \rho_R \kappa \]
\[ = \epsilon \left[ \lambda (I^*) \ E \right] \rho_C \pi \epsilon^* \rho_R \kappa \]
\[ = \lambda (#I^*) \left( e \left[ E \right] \left( \text{extends}_C \rho_C I^* \right) \text{restore} \right) \pi \epsilon^* \rho_R \kappa \]
\[ = \pi((\lambda^*\kappa'). \# \epsilon^* = \# I^* \rightarrow \text{tievals}(\lambda \delta^* \cdot (e \left[ E \right] \left( \text{extends}_C \rho_C I^* \right) \text{restore})) \]
\[ \quad \left( \text{false in } E \right) \]
\[ \quad \left( \text{extends}_R \rho_R \delta^* \right) \]
\[ \quad \epsilon^*, \quad \text{(by induction hypothesis)} \]
\[ \quad \text{wrong in } E) \]
\[ e^* \rho_R \kappa \]
\[ = \pi((\lambda^*\kappa'). \# \epsilon^* = \# I^* \rightarrow \text{tievals}(\lambda \delta^* \cdot E \left[ E \right] \left( (\rho_R \circ \rho_C) \left( \rho_R \circ \rho_C \delta^* / 1^* \right) \right) \kappa') \]
\[ \quad \epsilon^*, \quad \text{(by induction hypothesis)} \]
\[ \quad \text{wrong in } E) \]
\[ e^* \rho_R \kappa \]
\[ = (\lambda . \pi \epsilon^* \rho_R \kappa) \]
\[ (\lambda^*\kappa'). \# \epsilon^* = \# I^* \rightarrow \text{tievals}(\lambda \delta^* \cdot E \left[ E \right] ((\rho_R \circ \rho_C) \left( \rho_R \circ \rho_C \delta^* / 1^* \right) \) \kappa') \]
\[ \quad \epsilon^*, \quad \text{by induction hypothesis)} \]
\[ \quad \text{wrong in } E) \]
\[ e^* \rho_R \kappa \]
\[ = E \left[ (\lambda^* \ E) \right] (\rho_R \circ \rho_C) (\lambda . \pi \epsilon^* \rho_R \kappa) \]

\[ t\left[ \text{if } E_0 \ E_1 \ E_2 \right] \rho_C \pi \epsilon^* \rho_R \kappa \]
\[ = t[\left[ E_0 \right] \rho_C \left( \text{if} \left( t[E_1] \rho_C \pi \right) \left( t[E_2] \rho_C \pi \right) \right) \epsilon^* \rho_R \kappa \]
\[ = E \left[ [E_0] \left( \rho_R \circ \rho_C \right) \left( \lambda . \left( \text{if} \left( t[E_1] \rho_C \pi \right) \left( t[E_2] \rho_C \pi \right) \right) \epsilon^* \rho_R \kappa \right) \right) \]
\[ \text{by induction hypothesis) \]
\[ = E \left[ E_0 \right] \left( \rho_R \circ \rho_C \right) \left( \lambda . \epsilon = \text{false} \rightarrow t[E_2] \rho_C \pi \epsilon^* \rho_R \kappa \right), \]
\[ \quad t[E_1] \rho_C \pi \epsilon^* \rho_R \kappa \right) \]
\[ = E \left[ E_0 \right] \left( \rho_R \circ \rho_C \right) \left( \lambda . \epsilon = \text{false} \rightarrow E \left[ E_2 \right] \left( \rho_R \circ \rho_C \right) \left( \lambda . \pi \epsilon^* \rho_R \kappa \right), \right) \]
\[ \quad E \left[ E_1 \right] \left( \rho_R \circ \rho_C \right) \left( \lambda . \pi \epsilon^* \rho_R \kappa \right) \right) \]
\[ \text{(by induction hypothesis)} \]
\[ = E \left[ \text{if } E_0 \ E_1 \ E_2 \right] \left( \rho_R \circ \rho_C \right) \left( \lambda . \pi \epsilon^* \rho_R \kappa \right) \]

For a procedure call \( (E_0 \ E^*) \), the generated code depends upon whether \( E \) is a primitive operation whose code is generated inline. If \( \text{primop}[E_0] \left( \# E^* \right) \rho_C = \text{true} \), then

- \( E_0 \in \text{Ide} \)
- \( \# E^* \neq 0 \)
Fact 1. $\forall \epsilon \in \epsilon'$:

$$(t[I_1] \rho_C (op (#E^*) (lookup_{C} \rho_C I | F) \pi)) \epsilon
= \mathcal{E}[I_1] (\rho_R \circ \rho_C) (\lambda \epsilon \cdot (op (#E^*) (lookup_{C} \rho_C I | F) \pi) \epsilon \epsilon^* \rho_R \kappa)$$

(by induction hypothesis)

$$= (t[I_1] \rho_C (op (#E^*) (\rho_C I | F) \pi)) \epsilon'$$

Fact 2. $\forall \epsilon \epsilon'$:

$t[E_0] \rho_C invoke \epsilon \epsilon^* \rho_R \kappa$

$$= \mathcal{E}[E_0] (\rho_R \circ \rho_C) (\lambda \epsilon \cdot invoke \epsilon \epsilon^* \rho_R \kappa)$

$$= t[E_0] \rho_C invoke \epsilon' \epsilon^* \rho_R \kappa$$

Case 1: \textit{primop}[E_0] (#E*) \rho_C = true

$$t[E_0 E^*] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa$$

$$(\text{primop}[E_0] (#E^*) \rho_C \rightarrow e[[E_0 E^*]] \rho_C, \quad \text{save (e[[E_0 E^*]] \rho_C restore))}$$

$$\pi \epsilon \epsilon^* \rho_R \kappa$$

$$= e[[E_0 E^*]] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa$$

$$= e[[I_1 E^*]] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa$$

$$= p[[I_1 E^*]] \rho_C \pi \epsilon \epsilon^* \rho_R \kappa$$

$$= e^* [E^*] \rho_C (t[I_1] \rho_C (op (#E^*) (lookup_{C} \rho_C I | F) \pi)) \epsilon \epsilon^* \rho_R \kappa$$

$$= \mathcal{E}^* [E^*] \rho_R \circ \rho_C (\lambda \epsilon^* \cdot t[I_1] \rho_C (op (#E^*) (lookup_{C} \rho_C I | F) \pi))$$

$$\quad \epsilon \epsilon^* \rho_R \kappa$$

(by induction hypothesis and Fact 1 proved above)

$$= \mathcal{E}^* [E^*] \rho_R \circ \rho_C (\lambda \epsilon^* \cdot \mathcal{E}[E_1] (\rho_R \circ \rho_C) (\lambda \epsilon \cdot op (#E^*) (lookup_{C} \rho_C I | F) \pi \epsilon (e^*_2 \hat{\epsilon}^*) \rho_R \kappa))$$

(by induction hypothesis)

$$= \mathcal{E}^* [E^*] \rho_R \circ \rho_C (\lambda \epsilon^* \cdot \mathcal{E}[E_1] (\rho_R \circ \rho_C) (\lambda \epsilon \cdot apply (lookup_{C} \rho_C I | F in E)$$

$$\quad ((\epsilon) \hat{\epsilon}^* (takefirst (#E^*_2) (e^*_2 \hat{\epsilon}^*)))))$$

$$\quad (\lambda \epsilon \cdot \pi \epsilon ((e^*_2 \hat{\epsilon}^*) \uparrow (#E^*_2)) \rho_R \kappa)))$$

18
\[
\begin{align*}
\mathcal{E}^*[E_2'] (\rho_R \circ \rho_C) \\
= (\lambda_2' \cdot (\text{lab} \cdot \mathcal{E}[E_1]) (\rho_R \circ \rho_C) \\
(\lambda_e \cdot \text{apply} (\text{lookup}_C \rho_C I | F \text{ in } E) \\
((e) \triangleright a) \\
(\lambda_e \cdot \pi \in b \rho_R \kappa)) \\
(takefirst (#E_2') (e_2' \triangleright e^*)) \\
((e_2' \triangleright e^*) \dagger (#E_2')) \\
= \mathcal{E}^*[E_2'] (\rho_R \circ \rho_C) \\
(\lambda_2' \cdot (\text{lab} \cdot \mathcal{E}[E_1]) (\rho_R \circ \rho_C) \\
(\lambda_e \cdot \text{apply} (\text{lookup}_C \rho_C I | F \text{ in } E) \\
((e) \triangleright a) \\
(\lambda_e \cdot \pi \in b \rho_R \kappa)) \\
\epsilon_2^* \\
e^*
\end{align*}
\]
(\lambda \epsilon \cdot \pi \epsilon e^* \rho_R \kappa)\})

(by reasonableness of \rho_R)

= \mathcal{E}^* [E_2] (\rho_R \circ \rho_C)
(\lambda_2 \cdot \mathcal{E} [E_1] (\rho_R \circ \rho_C)
(\lambda_1 \cdot \langle \lambda_0 \cdot \text{apply } \epsilon_0\rangle
(\langle \epsilon \rangle \circ \epsilon_2\rangle
(\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa))

((\lambda_0 \cdot \text{apply } \epsilon_0)
(\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa)))

= \mathcal{E}^* [E_1, E_2] (\rho_R \circ \rho_C)
(\lambda_1 \cdot \mathcal{E} [I] (\rho_R \circ \rho_C)
(\lambda_0 \cdot \text{apply } \epsilon_0 \epsilon_1^* (\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa))

(by Lemma 2)

= \mathcal{E} [I, E_1, E_2] (\rho_R \circ \rho_C)
(\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa)

= \mathcal{E} [E_0, E^*] (\rho_R \circ \rho_C)
(\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa)

Case 2: \text{primop} [E_0] (#E^*) \rho_C = \text{false}

t \in (E_0, E^*) \rho_C \pi \epsilon e^* \rho_R \kappa

= (\text{primop} [E_0] (#E^*) \rho_C \rightarrow e \left[ (E_0, E^*) \right] \rho_C,
\text{save} (e \left[ (E_0, E^*) \right] \rho_C \text{restore}))

\pi \epsilon e^* \rho_R \kappa

= \text{save} (e \left[ (E_0, E^*) \right] \rho_C \text{restore}) \pi \epsilon e^* \rho_R \kappa

= \text{save} (e^* \left[ E^* \right] \rho_C \left( t [E_0] \rho_C \text{ invoke} \right)) \pi \epsilon e^* \rho_R \kappa

= e^* \left[ E^* \right] \rho_C \left( t [E_0] \rho_C \text{ invoke} \right) \epsilon (\epsilon_1 \circ \epsilon_2 \circ \epsilon_3) \rho_R (\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa)

(by induction hypothesis
and Fact 2 proved above)

= \mathcal{E}^* [E^*] (\rho_R \circ \rho_C)
(\lambda_1 \cdot \mathcal{E} [E_0] (\rho_R \circ \rho_C)
(\lambda_1 \cdot \text{invoke } \epsilon (\epsilon_1 \circ \epsilon_2 \circ \epsilon_3) \rho_R (\lambda_1 \cdot \pi \epsilon e^* \rho_R \kappa))

(by induction hypothesis)

20
That completes the proof of part A for procedure calls.

Proof of part B:

When \( t[E] \rho_C = e[E] \rho_C \),

\[
e[E] \rho_C \text{ restore } \epsilon \langle \rangle \rho_R \kappa
= t[E] \rho_C \text{ restore } \epsilon \langle \rangle \rho_R \kappa
= \mathcal{E} [E] (\rho_R \circ \rho_C) (\lambda \epsilon . \text{ restore } \epsilon \langle \rangle \rho_R \kappa)
= \mathcal{E} [E] (\rho_R \circ \rho_C) (\lambda \epsilon . \kappa \epsilon)
= \mathcal{E} [E] (\rho_R \circ \rho_C) \kappa
\]

(by induction hypothesis)

The three remaining cases are for conditional expressions, procedure calls, and sequenced expressions.

\[
e[(\text{if } E_0 \ E_1 \ E_2)] \rho_C \text{ restore } \epsilon \langle \rangle \rho_R \kappa
= t[E_0] \rho_C (\text{ if } (e[E_1] \rho_C \text{ restore}) (e[E_2] \rho_C \text{ restore}) \epsilon \langle \rangle \rho_R \kappa
= \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon . \text{ if } (e[E_1] \rho_C \text{ restore}) (e[E_2] \rho_C \text{ restore}) \epsilon \langle \rangle \rho_R \kappa)
= \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon . \epsilon = \text{ false } \rightarrow e[E_2] \rho_C \text{ restore } \epsilon \langle \rangle \rho_R \kappa, e[E_1] \rho_C \text{ restore } \epsilon \langle \rangle \rho_R \kappa)
\]

(by induction hypothesis)
\[ = \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon. \epsilon = \text{false} \rightarrow \mathcal{E} [E_1] (\rho_R \circ \rho_C) \kappa, \mathcal{E} [E_2] (\rho_R \circ \rho_C) \kappa) \]

(by induction hypothesis)

\[ = \mathcal{E} [(\text{if } E_0 \ E_1 \ E_2)] (\rho_R \circ \rho_C) \kappa \]

For procedure calls, there are two cases.

**Case 1:** \( \text{primop } [E_0] (\#E^*) \rho_C = \text{true} \)

For this case, \( t [(E_0 \ E^*)] \rho_C = e [(E_0 \ E^*)] \rho_C \) so there is nothing left to prove.

**Case 2:** \( \text{primop } [E_0] (\#E^*) \rho_C = \text{false} \)

\[ e [(E_0 \ E^*)] \rho_C \text{ restore } \epsilon ( \rho_R \kappa \]

\[ = (\text{primop } [E_0] (\#E^*) \rho_C \rightarrow p [(E_0 \ E^*)] \rho_C \text{ restore, e} (E^*) \rho_C (t [E_0] \rho_C \text{ invoke})) \]

\[ = e (E^*) \rho_C (t [E_0] \rho_C \text{ invoke}) \epsilon ( \rho_R \kappa \]

\[ = \mathcal{E}^* [E^*] (\rho_R \circ \rho_C) (\lambda \epsilon^1. \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon. \text{invoke } \epsilon \epsilon^1 \rho_R \kappa)) \]

(by induction hypothesis
and Fact 2 proved above)

\[ = \mathcal{E}^* [E^*] (\rho_R \circ \rho_C) (\lambda \epsilon^1. \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon. \text{apply } \epsilon \epsilon^1 \kappa)) \]

(by induction hypothesis)

\[ = \mathcal{E}^* [E^*] (\rho_R \circ \rho_C) (\lambda \epsilon^1. \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon. \text{invoke } \epsilon \epsilon^1 \rho_R \kappa)) \]

That completes the proof of part B for procedure calls.

\[ e [(\text{begin } E_0 \ E_1)] \rho_C \text{ restore } \epsilon ( \rho_R \kappa \]

\[ = t [E_0] \rho_C (e [E_1] \rho_C \text{ restore}) \epsilon ( \rho_R \kappa \]

\[ = \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon. e [E_1] \rho_C \text{ restore } \epsilon ( \rho_R \kappa) \]

(by induction hypothesis)

\[ = \mathcal{E} [E_0] (\rho_R \circ \rho_C) (\lambda \epsilon. \mathcal{E} [E_1] (\rho_R \circ \rho_C) \kappa) \]

(by induction hypothesis)

\[ = \mathcal{E} [(\text{begin } E_0 \ E_1)] (\rho_R \circ \rho_C) \kappa \]

**Proof of part C:**

22
Assuming $\forall \epsilon, \epsilon' . \pi \epsilon = \pi \epsilon'$:

$$e^* [ \rho_C \pi \epsilon e^* \rho_R \kappa = \pi \epsilon e^* \rho_R \kappa = (\lambda \epsilon_1 . \pi \epsilon (\epsilon_1 \epsilon^* \rho_R \kappa) \langle \rangle)$$

$$E^* [ (\rho_R \circ \rho_C) (\lambda \epsilon_1 . \pi \epsilon (\epsilon_1 \epsilon^* \rho_R \kappa) \rangle$$

$$e^* [E^* E] \rho_C \pi \epsilon e^* \rho_R \kappa = t[E] \rho_C (\rho_R \circ \rho_C) (\lambda \epsilon . \pi \epsilon (\epsilon \epsilon^* \rho_R \kappa)$$

(by induction hypothesis)

$$= E [E] (\rho_R \circ \rho_C) (\lambda \epsilon_1 . \pi \epsilon (\epsilon_1 \epsilon^* \rho_R \kappa)$$

(by induction hypothesis)

$$= E [E] (\rho_R \circ \rho_C) (\lambda \epsilon_1 . E^* [E^*] (\rho_R \circ \rho_C) (\lambda \epsilon_1 . \pi \epsilon (\epsilon_1 \epsilon^* \rho_R \kappa)$$

(by induction hypothesis)

$$= E [E] (\rho_R \circ \rho_C) (\lambda \epsilon_1 . \pi \epsilon (\epsilon_1 \epsilon^* \rho_R \kappa)$$

7 Tail recursion

The map $e$ is used only in tail-recursive positions where registers need not be preserved. In non-tail-recursive positions a tidy version of $e$ called $t$ is used, which in effect wraps a save/restore sequence around code that would otherwise destroy registers. Actually the caller saves registers and a return address before evaluating untidy arguments. The callee restores registers when it returns. This calling discipline is based on [Ste77], and is similar to calling conventions used in most other implementations of Scheme.

Thus the body of a tail-recursive procedure such as

(define (add x y)
  (if (zero? x)
      y
      (add (pred x) (succ y))))

compiles into a sequence of machine instructions whose semantics is
In the MacScheme machine’s assembly language, the numerical operand of the `op` instruction is 1 greater than the corresponding operand of its `op` semantics, so that sequence of machine instructions looks like

```
(lexical (0, 0)
(op 0 zero?)
(if
(lexical (0, 1)
(restore))
(lexical (0, 1)
(op 0 succ
(push
(lexical (0, 0)
(op 0 pred
(push
(fetch add
:invoke)))))))))
```

whereas the body of a non-tail-recursive procedure such as

```
(define (add x y)
  (if (zero? x)
    y
    (succ (add (pred x) y))))
```

compiles as something like

```
(lexical 0,0 ; x
  op1 zero?
  branchf L1
  lexical 0,1 ; y
  restore
L1: lexical 0,1 ; y
  op1 succ
  push
  lexical 0,0 ; x
  op1 pred
  push
  fetch add
  invoke
```

```
Loops in Scheme can be expressed only through tail recursion. At optimization level 0, the MacScheme compiler does not detect loops, because loop detection requires an analysis of side effects. For example, if \texttt{add} is global then the tail-recursive call in the first \texttt{add} procedure cannot safely be compiled as a backward branch. Hence it is very important that procedure invocation be fast.

The code generation algorithm appears to duplicate the code that follows an if expression in non-tail position. This is because the instruction algebra is treelike. In practice, however, code can be represented as dags rather than as trees, so the subtrees can share one data structure. At optimization level 0, the MacScheme compiler does not generate code with cycles, but similar remarks would apply if it did [Wan83].

## 8 Optimizations

Problems arise when the compiler algorithm to be proved correct includes optimizations that upset the pattern of storage allocation or depend on invariants difficult to express using standard semantics. This section illustrates these problems by considering several optimizations that are performed by the MacScheme compiler at optimization level 1 but not at optimization level 0.

Some source level optimizations can be proved correct simply by proving equalities such as

\[
E \llbracket (\text{if} \ (\text{not} \ E_0) \ E_1 \ E_2) \rrbracket \rho_{\text{init}} = E \llbracket (\text{if} \ E_0 \ E_2 \ E_1) \rrbracket \rho_{\text{init}}
\]

and

\[
E \llbracket ((\lambda \ () \ E)) \rrbracket = E \llbracket E \rrbracket
\]

On the other hand it is impossible to prove that

\[
E \llbracket ((\lambda \ (x) \ x) \ 3) \rrbracket = E \llbracket 3 \rrbracket
\]

because \((\lambda \ (x) \ x) \ 3\) allocates storage for the variable \(x\) whereas the second form doesn’t, and this difference can affect the final answer. For example, the initial continuation might print statistics on storage allocation. Perhaps we don’t care about such continuations, but the semantics has no way of knowing
that. Furthermore it is not immediately obvious how to express our insouciance in terms the semantics can understand.

It is possible to deal with this problem by descending to a lower level of abstraction and constructing a more complex proof, but that would sacrifice the simplicity of the denotational approach.

Exactly the same problem appears at the target level. It is possible to prove the correctness of such target level peephole optimizations as

\[ \text{fetch} \ I (\text{if} (\text{fetch} \ I \ \pi_1) \ \pi_2) = \text{fetch} \ I (\text{if} \ \pi_1 \ \pi_2) \]

and

\[ \text{fetch} \ I (\text{push} (\text{fetch} \ I \ \pi)) = \text{fetch} \ I (\text{push} \ \pi) \]

Most optimizations of the sort described in [MD80] are of this nature. It so happens that these optimizations are relatively unimportant when compared with optimizations that disturb the pattern of storage allocation, as for example when a variable is kept in a register rather than on a heap. Such target level optimizations present the same obstacles to denotational correctness proofs as source level optimizations that alter storage allocation.

More global optimizations depend on more global analysis, such as a side effects or liveness analysis. It is not clear whether such analysis can easily be performed a standard denotational semantics. Once more, the solution is to use a less abstract semantics.

With stack-based languages a deletion strategy can be moved into the semantics of the language. For example, the semantics might specify that on return from a procedure the storage allocated by that procedure is reclaimed. Such a tactic is only a partial solution to the problems that have been outlined in this section, and absent a proof it begs the question of whether the deletion semantics is equivalent to the original semantics.

In light of these problems, it may be reasonable for compiler designers to rely on conjectured equivalences. Designers concerned to verify their compilers will prove the conjectures, using a lower level of abstraction as necessary, but even if left unproved a careful statement of the conjectures will serve to warn of thin ice.

The MacScheme compiler follows that strategy. At optimization level 0, the MacScheme compiler uses the code generator whose correctness is proved by Theorem 3. Optimizations that cannot easily be proved correct using the denotational approach are performed only at optimization level 1 and higher.

9 Related work

The problem of proving a compiler algorithm correct relative to a high level specification such as a standard denotational semantics is very different from the problem of generating a compiler from the specification. For example, [Ras82]
describes an automatic system that can often transform a denotational semantics into a reasonably efficient code generator, but the correctness of the code generators so obtained is a conjecture rather than a theorem. As is pointed out in [BBK82, CJ82], automatic systems that can guarantee correctness relative to a standard semantics may also generate inefficient code.

The correctness proof reported here falls into a tradition of simple, rigorous compiler correctness proofs [BL69, MW72, Mor73, TWW79, Wan82b]. When first published, this proof was the first in that line to prove the correctness of a usable compiler for a real programming language. (It helps of course that Scheme is simpler even than many toy languages.) The importance of compiler correctness proofs for real programming languages is that they try the ability of the proof techniques to adapt to unanticipated difficulties.

For example, the MacScheme compiler performs several optimizations at optimization levels 1 and above that could not be included in this paper because there was no simple denotational proof of their correctness. Examples include keeping variables in registers rather than on the heap, and compiling calls to known procedures (notably self-tail calls) as simple branch instructions instead of the more complex `invoke` instruction described in this paper.

This paper is based directly on [Wan82a]. One difference is that in this paper the target machine instructions perform their own sequencing, while [Wan82a] uses special sequencing combinators to paste actions together. This difference is not in itself terribly important, since the target machines in [Wan82a] are reasonable only if pairs consisting of a sequencing combinator and an action are viewed as single instructions. The real advantage of the special sequencing combinators is that they allow separate inductions to be used to prove certain associativities satisfied by the combinators and to prove the correctness of algorithms for linearizing the target code, “optimizing” tail recursion, and distributing compile-time environment information. In this paper all those inductions are performed at once, mainly because it was easier to work with the compiler algorithm as it existed than to tear it apart into distinct phases.

In turn, the proof in this paper inspired the VLISP project, which remains one of the most complete proofs of compiler and implementation correctness [GW95, GRW95, ORW95].

Difficulties in reasoning about storage allocation within denotational semantics have been noted before, giving rise to some nonstandard treatments of storage allocation [HMT84]. The notion of storage allocation used in the semantics of Scheme in this paper is the more traditional notion in which the `new` operator is continuous. (See sections 3.3.2 and 4.2.6 of [MS76].)

In the years that have passed since the conference version of this paper, some progress has been made toward almost-denotational proofs of optimizations that change the pattern of storage allocation [WS97]. In general, however, those optimizations are still proved correct by descending to a lower level of abstraction and using an operational semantics [WC01, KW06].

Scheme was first described in 1975 and 1978 [SS75, SS78]. The denotational semantics given in this paper was the basis for a semantics given in several revisions of those reports and as an appendix to the IEEE standard for Scheme.
[RC86, CR91, KCR98, IEE91].

9.1 MacScheme

The code generation algorithm of this paper was used at optimization level 0 by the compiler for MacScheme, which was a commercial implementation of Scheme for the Apple Macintosh [Lig88]. Optimization level 1 added several optimizations that had not been proved correct, and optimization level 2 generated native machine code instead of interpreted byte code.

MacScheme was sold commercially for about 10 years. During that time, only 5 compiler bugs were discovered. None of those bugs involved the part of the code generator whose correctness is proved here. A couple of the compiler bugs involved code generation for rest arguments, which were left out of the semantics and proof. The other compiler bugs were in the macro expander.

9.2 Twobit

Experience with the MacScheme compiler led to a more ambitious optimizing compiler named Twobit, which is used by the Larceny implementation of Scheme [CH94]. Twobit relies on incremental lambda lifting and closure conversion to keep most variables and temporaries in registers or stack frames, and performs a variety of other optimizations that were not performed by the MacScheme compiler. No part of Twobit has been proved correct.

10 Conclusions

The MacScheme compiler’s code generation algorithm has been stated and proved correct relative to a denotational semantics for Scheme. This appears to have been the first proof of correctness for a commercial compiler’s code generator.

The main benefits of the proof are that it makes explicit the global invariants on which the algorithm depends and provides the semantic framework needed to explore new optimizations and architectures. The proof also contributed to the extraordinary reliability of the MacScheme compiler.

The design of the target machine may have been the hardest part of the compiler construction and proof. The target machine itself is just a special purpose algebra. The reason it is difficult to extract such an algebra from the source language is that the desire for efficiency leads the target machine’s architecture away from the obvious algebra. The process by which the target machine is designed should be seen as a descent to lower levels of abstraction in which higher level data types are implemented in terms of more machine-like concepts. It is important to describe the target machine at as high a level as possible, however, both to simplify the compiler correctness proof and to preserve design flexibility for lower levels of implementation. Continuation semantics provides a nice meeting ground between the source language and the target machine.
The idea of factoring environments through a domain of compile-time environments works well for Scheme and should be used more often with statically typed languages.

The main problem with the semantic framework used in this paper is that it has trouble dealing with many important optimizations. It may be possible to prove correctness of the basic code generator using a denotational framework, while descending to a lower level of abstraction to prove the correctness of optimizations that change the pattern of storage allocation.

References


[Wan82a] ———, *Deriving target code as a representation of continuation semantics*, ACM Transactions on Programming Languages and Systems **4** (1982), no. 3, 496–517.


