

The MacScheme Compiler: a denotational proof of correctness

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

August 22, 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 was the first proof of correctness for a commercial compiler. It was based on Mitch Wand's foundational research, and helped to inspire the VLISP project.

Notation

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

Abstract syntax

$K \in \text{Con}$ constants
 $I \in \text{Ide}$ identifiers
 $E \in \text{Exp}$ expressions

Productions

$\text{Exp} ::= K$
 | I
 | $(\text{set! } I E)$
 | $(\text{lambda } (I^*) E)$
 | $(\text{if } E_0 E_1 E_2)$
 | $(E_0 E^*)$
 | $(\text{begin } E_0 E_1)$

Value domains

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

Semantic functions

$$\begin{aligned}\mathcal{K} &: \text{Con} \rightarrow \mathbf{E} \\ \mathcal{E} &: \text{Exp} \rightarrow \mathbf{U} \rightarrow \mathbf{K} \rightarrow \mathbf{C} \\ \mathcal{E}^* &: \text{Exp}^* \rightarrow \mathbf{U} \rightarrow (\mathbf{E}^* \rightarrow \mathbf{C}) \rightarrow \mathbf{C}\end{aligned}$$

$$\mathcal{E}[\mathbf{K}] = \lambda\rho\kappa . \kappa (\mathcal{K}[\mathbf{K}])$$

$$\mathcal{E}[\mathbf{I}] = \lambda\rho\kappa . (\text{lookup } \rho \mathbf{I}) \in \mathbf{F} \rightarrow \kappa ((\text{lookup } \rho \mathbf{I}) \mid \mathbf{F} \text{ in } \mathbf{E}), \\ \text{hold } ((\text{lookup } \rho \mathbf{I}) \mid \mathbf{L}) \kappa$$

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

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

$$\mathcal{E}[(\text{if } \mathbf{E}_0 \ \mathbf{E}_1 \ \mathbf{E}_2)] = \lambda\rho\kappa . \mathcal{E}[\mathbf{E}_0] \rho (\lambda\epsilon . \text{truish } \epsilon \rightarrow \mathcal{E}[\mathbf{E}_1] \rho \kappa, \mathcal{E}[\mathbf{E}_2] \rho \kappa)$$

$$\mathcal{E}[(\mathbf{E}_0 \ \mathbf{E}^*)] = \lambda\rho\kappa . \mathcal{E}^*[\mathbf{E}^*] \rho (\lambda\epsilon^* . \mathcal{E}[\mathbf{E}_0] \rho (\lambda\epsilon . \text{apply } \epsilon \epsilon^* \kappa))$$

$$\mathcal{E}[(\text{begin } \mathbf{E}_0 \ \mathbf{E}_1)] = \lambda\rho\kappa . \mathcal{E}^*[\mathbf{E}_0] \rho (\lambda\epsilon . \mathcal{E}[\mathbf{E}_1] \rho \kappa)$$

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

$$\mathcal{E}^*[\mathbf{E}^* \ \mathbf{E}] = \lambda\rho\psi . \mathcal{E}[\mathbf{E}] \rho (\lambda\epsilon . \mathcal{E}^*[\mathbf{E}^*] \rho (\lambda\epsilon^* . \psi (\epsilon^* \S \langle \epsilon \rangle)))$$

Compile-time and run-time environments

$\rho \in \mathbf{U} = \mathbf{Ide} \rightarrow \mathbf{D}$	environments
$\rho_C \in \mathbf{U}_C = \mathbf{Ide} \rightarrow \mathbf{D}_C$	compile-time environments
$\rho_R \in \mathbf{U}_R = \mathbf{D}_C \rightarrow \mathbf{D}$	run-time environments
$\delta_C \in \mathbf{D}_C = \mathbf{F} + \mathbf{Ide} + (\mathbb{N} \times \mathbb{N})$	lexical addresses

Operations on environments

$rep_C : \mathbf{U} \rightarrow \mathbf{U}_C$
$rep_R : \mathbf{U} \rightarrow \mathbf{U}_R$
$extends_C : \mathbf{U}_C \rightarrow \mathbf{Ide}^* \rightarrow \mathbf{U}_C$
$extends_R : \mathbf{U}_R \rightarrow \mathbf{D}^* \rightarrow \mathbf{U}_R$

$$\begin{aligned} (rep_R \rho) \circ (rep_C \rho) &= \rho \\ (extends_R \rho_R \delta^*) \circ (extends_C \rho_C \mathbf{I}^*) &= (\rho_R \circ \rho_C) [\delta^* / \mathbf{I}^*] \end{aligned}$$

A run-time environment ρ_R is reasonable if and only if

1. $(\delta_C \in \mathbf{F}) = (\rho_R \delta_C \in \mathbf{F})$
2. If $\delta_C \in \mathbf{F}$ then $\rho_R \delta_C = (\delta_C \mid \mathbf{F} \text{ in } \mathbf{D})$.

MacScheme machine instructions

$$save = \lambda\pi_1\pi_2 . \lambda\epsilon\epsilon^*\rho_R\kappa . \pi_1 \epsilon \langle \rangle \rho_R (\lambda\epsilon . \pi_2 \epsilon \epsilon^* \rho_R \kappa)$$

$$restore = \lambda\epsilon\epsilon^*\rho_R\kappa . \kappa \epsilon$$

$$const = \lambda\epsilon_0\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . \pi \epsilon_0 \epsilon^* \rho_R \kappa$$

$$fetch = \lambda I\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . hold (\rho_R (I \text{ in } D_C) | L) (\lambda\epsilon . \pi \epsilon \epsilon^* \rho_R \kappa)$$

$$lexical = \lambda x\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . hold (\rho_R (x \text{ in } D_C) | L) (\lambda\epsilon . \pi \epsilon \epsilon^* \rho_R \kappa)$$

$$set = \lambda I\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . assign (\rho_R (I \text{ in } D_C) | L) \epsilon (\pi \epsilon \epsilon^* \rho_R \kappa)$$

$$setlex = \lambda x\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . assign (\rho_R (x \text{ in } D_C) | L) \epsilon (\pi \epsilon \epsilon^* \rho_R \kappa)$$

$$lambda = \lambda\nu\pi_1\pi_2 . \lambda\epsilon\epsilon^*\rho_R\kappa .$$

$$\pi_2 ((\lambda\epsilon^*\kappa' . \#\epsilon^* = \nu \rightarrow tievals (\lambda\delta^* . \pi_1 (false \text{ in } E) \langle \rangle (extends_R \rho_R \delta^*) \kappa') \epsilon^*, \\ wrong) \text{ in } E) \epsilon^* \rho_R \kappa$$

$$if = \lambda\pi_1\pi_2 . \lambda\epsilon\epsilon^*\rho_R\kappa . truish \epsilon \rightarrow \pi_1 \epsilon \epsilon^* \rho_R \kappa, \pi_2 \epsilon \epsilon^* \rho_R \kappa$$

$$invoke = \lambda\epsilon\epsilon^*\rho_R\kappa . apply \epsilon \epsilon^* \rho_R \kappa$$

$$push = \lambda\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . \pi \epsilon (\langle \epsilon \rangle \S \epsilon^*) \rho_R \kappa$$

$$op = \lambda\nu\phi\pi . \lambda\epsilon\epsilon^*\rho_R\kappa . apply (\phi \text{ in } E) (\langle \epsilon \rangle \S (takefirst \nu \epsilon^*)) (\lambda\epsilon . \pi \epsilon (\epsilon^* \dagger \nu) \rho_R \kappa)$$

$$illegal = \lambda\epsilon\epsilon^*\rho_R\kappa . wrong$$

Compiling functions

$t : \text{Exp} \rightarrow \text{U}_C \rightarrow \text{P} \rightarrow \text{P}$ expression compiler, tidy
 $e : \text{Exp} \rightarrow \text{U}_C \rightarrow \text{P} \rightarrow \text{P}$ expression compiler
 $e^* : \text{Exp}^* \rightarrow \text{U}_C \rightarrow \text{P} \rightarrow \text{P}$ argument list compiler

$$t \llbracket \text{K} \rrbracket = e \llbracket \text{K} \rrbracket$$

$$t \llbracket \text{I} \rrbracket = e \llbracket \text{I} \rrbracket$$

$$t \llbracket (\text{set! } \text{I } \text{E}) \rrbracket = e \llbracket (\text{set! } \text{I } \text{E}) \rrbracket$$

$$t \llbracket (\text{lambda } (\text{I}^*) \text{E}) \rrbracket = e \llbracket (\text{lambda } (\text{I}^*) \text{E}) \rrbracket$$

$$t \llbracket (\text{if } \text{E}_0 \text{E}_1 \text{E}_2) \rrbracket = \lambda \rho_C \pi . t \llbracket \text{E}_0 \rrbracket \rho_C (\text{if } (t \llbracket \text{E}_1 \rrbracket \rho_C \pi) (t \llbracket \text{E}_2 \rrbracket \rho_C \pi))$$

$$t \llbracket (\text{E}_0 \text{E}^*) \rrbracket = \lambda \rho_C . \text{primop} \llbracket \text{E}_0 \rrbracket (\# \text{E}^*) \rho_C \rightarrow e \llbracket (\text{E}_0 \text{E}^*) \rrbracket \rho_C , \\ \text{save } (e \llbracket (\text{E}_0 \text{E}^*) \rrbracket \rho_C \text{restore})$$

$$t \llbracket (\text{begin } \text{E}_0 \text{E}_1) \rrbracket = \lambda \rho_C \pi . t \llbracket \text{E}_0 \rrbracket \rho_C (t \llbracket \text{E}_1 \rrbracket \rho_C \pi)$$

$$e \llbracket \mathbf{K} \rrbracket = \lambda \rho_C . \text{const } (\mathcal{K} \llbracket \mathbf{K} \rrbracket)$$

$$e \llbracket \mathbf{I} \rrbracket = \lambda \rho_C . (\text{lookup } \rho_C \mathbf{I}) \in \mathbf{F} \rightarrow \text{const } ((\text{lookup } \rho_C \mathbf{I}) \mid \mathbf{F} \text{ in } \mathbf{E}), \\ (\text{lookup } \rho_C \mathbf{I}) \in \mathbf{Ide} \rightarrow \text{fetch } ((\text{lookup } \rho_C \mathbf{I}) \mid \mathbf{Ide}), \\ \text{lexical } ((\text{lookup } \rho_C \mathbf{I}) \mid (\mathbf{N} \times \mathbf{N}))$$

$$e \llbracket (\text{set! } \mathbf{I} \ \mathbf{E}) \rrbracket = \\ \lambda \rho_C \pi . t \llbracket \mathbf{E} \rrbracket \rho_C \\ ((\text{lookup } \rho_C \mathbf{I}) \in \mathbf{F} \rightarrow \text{illegal}, \\ (\text{lookup } \rho_C \mathbf{I}) \in \mathbf{Ide} \rightarrow \text{set } ((\text{lookup } \rho_C \mathbf{I}) \mid \mathbf{Ide}) \pi, \\ \text{setlex } ((\text{lookup } \rho_C \mathbf{I}) \mid (\mathbf{N} \times \mathbf{N})) \pi)$$

$$e \llbracket (\text{lambda } (\mathbf{I}^*) \ \mathbf{E}) \rrbracket = \lambda \rho_C . \text{lambda } (\#\mathbf{I}^*) (e \llbracket \mathbf{E} \rrbracket (\text{extends}_C \rho_C \mathbf{I}^*) \text{restore})$$

$$e \llbracket (\text{if } \mathbf{E}_0 \ \mathbf{E}_1 \ \mathbf{E}_2) \rrbracket = \lambda \rho_C \pi . t \llbracket \mathbf{E}_0 \rrbracket \rho_C (\text{if } (e \llbracket \mathbf{E}_1 \rrbracket \rho_C \pi) (e \llbracket \mathbf{E}_2 \rrbracket \rho_C \pi))$$

$$e \llbracket (\mathbf{E}_0 \ \mathbf{E}^*) \rrbracket = \lambda \rho_C \pi . \text{primop } \llbracket \mathbf{E}_0 \rrbracket (\#\mathbf{E}^*) \rho_C \rightarrow p \llbracket (\mathbf{E}_0 \ \mathbf{E}^*) \rrbracket \rho_C \pi, \\ e^* \llbracket \mathbf{E}^* \rrbracket \rho_C (t \llbracket \mathbf{E}_0 \rrbracket \rho_C \text{invoke})$$

$$e \llbracket (\text{begin } \mathbf{E}_0 \ \mathbf{E}_1) \rrbracket = \lambda \rho_C \pi . t \llbracket \mathbf{E}_0 \rrbracket \rho_C (e \llbracket \mathbf{E}_1 \rrbracket \rho_C \pi)$$

$$e^* [] = \lambda \rho_C \pi . \pi$$

$$e^* [E^* E] = \lambda \rho_C \pi . t [E] \rho_C (push (e^* [E^*] \rho_C \pi))$$

$$p^* [(I E E^*)] = \lambda \rho_C \pi . e^* [E^*] \rho_C (t [E] \rho_C (op (\#E^*) (lookup \rho_C I | F) \pi))$$

$$primop [E] = \lambda \nu \rho_C . E \in \mathbf{Ide} \rightarrow \nu \neq 0 \rightarrow lookup \rho_C (E | \mathbf{Ide}) \in \mathbf{F}, false, false$$

Theorem 1 *If ρ_C is strict, ρ_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.

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

B. Correctness.

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

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

$$e^*[\mathbb{E}^*] \rho_C \pi \in \epsilon^* \rho_R \kappa = \mathcal{E}^*[\mathbb{E}^*] (\rho_R \circ \rho_C) (\lambda \epsilon_1^* . \pi \in (\epsilon_1^* \S \epsilon^*) \rho_R \kappa)$$

Sketch of proof. By simultaneous structural induction on \mathbb{E} and \mathbb{E}^* . ■

Example: tail call

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

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

Example: tail call

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

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
```

Example: non-tail call

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

lexical 0,0          ; x
op1    zero?
branchf L1
lexical 0,1          ; y
restore
L1:  save    L2          ; save registers, L2
     lexical 0,1        ; y
     push
     lexical 0,0        ; x
     op1    pred
     push
     fetch  add
     invoke
L2:  op1    succ
     restore
```

Optimizations

$$\mathcal{E}[(\text{if } (\text{not } E_0) E_1 E_2)] \rho^{init} = \mathcal{E}[(\text{if } E_0 E_2 E_1)] \rho^{init}$$

$$\mathcal{E}[(\text{lambda } () E)] = \mathcal{E}[E]$$

$$\mathcal{E}[(\text{lambda } (x) x) 3] \neq \mathcal{E}[3]$$

Related work

- 1980 Wand. Different advice on structuring compilers and proving them correct.
- 1982 Wand. Deriving target code as a representation of continuation semantics.
- 1982 Wand. Semantics-directed machine architecture.
- 1983 Wand. Loops in combinator-based compilers.
- 1984 Clinger. The Scheme 311 compiler: an exercise in denotational semantics.
- 1991 Ciesielski and Wand. Using the theorem prover Isabelle-91 to verify a simple proof of compiler correctness.
- 1995 Guttman and Wand. VLISP: a verified implementation of Scheme.
- 1995 Oliva, Ramsdell, and Wand. The VLISP verified PreScheme compiler.
- 1995 Wand. Compiler correctness for parallel languages.
- 1996 Gladstein and Wand. Compiler correctness for concurrent languages.
- 1997 Steckler and Wand. Lightweight closure conversion.
- 1997 Wand and Sullivan. Denotational semantics using an operationally-based term model.
- 1998,2001 Wand and Clinger. Set constraints for destructive array update optimization.
- 1999 Wand and Siveroni. Constraint systems for useless variable elimination.
- 2002 Wand and Williamson. A modular, extensible proof method for small-step flow analyses.
- 2006 Koutavas and Wand. Small bisimulations for reasoning about higher-order imperative programs.
- 2009 Dimoulas and Wand. The higher-order aggregate update problem.

Conclusions

MacScheme was sold commercially for about 10 years.

5 compiler bugs were discovered.

No bugs were ever discovered in the part that was proved correct.