An Operational Semantics for Scheme

John D. Ramsdell*
The MITRE Corporation

Abstract

In the informal description of Scheme, the order of evaluating the operands and the operator of each application is unspecified. This paper presents an operational semantics for Scheme which faithfully reflects this fact. Furthermore, when the semantics is restricted so as to assume there is one unspecified order used throughout a program, the semantics is shown to be sound with respect to the denotational semantics of Scheme.

1 Introduction

In the IEEE Standard [2], the formal semantics of the Scheme programming language is given in a denotational style [6, 5], i.e., as a map from a program to its meaning.[1] A denotational semantics can make certain proofs easier, such as proofs of properties of programs involving fixed points.

There are properties of programs that are not easily proved with a denotational semantics, but which are easily proved with an operational semantics. Furthermore, an operational semantics can describe some aspects of the language more smoothly than the denotational semantics. For example, in Scheme, the order of evaluating the operands and the operator of each application is unspecified, yet the formal semantics suggests there is one unspecified order used throughout a program.

What follows is an operational semantics for the Scheme programming language. Modulo a few caveats, each inference made within the operational semantics has an analog in the denotational semantics. For those who know little about the Lambda Calculus and Scott domains, the operational semantics provides an indirect way of reasoning within the denotational semantics.

e	expressions	
ℓ	locations	
i	identifiers	
v	expressed values	
s	stores	$\ell \rightarrow v$
u	environments	$i \rightarrow \ell$
k	continuations	
c	constants	
a	answers	
n	natural numbers	
p	permutations	$n \rightarrow n$

Figure 1: Types

2 Notation

The operational semantics for Scheme is specified by rules given in the relational style of natural semantics [3]. The types used in the natural semantics are in Figure 1. A type is a set of finite terms. Some of the types are defined by the grammar given in Figure 2. The one for expressed values v is incomplete.

Stores, environments, and permutations are finite maps—partial functions with finite domains. The term $\{x \mapsto y\}$ is the finite map which only maps x to y, i.e., its domain is the set $\{x\}$. The finite map g augment f, written $f + g$, has a domain of

*This work was supported by RADC Labs.
Author's address: The MITRE Corporation, Burlington Road, Bedford MA, 01730-0208.

[1]The denotational version of the semantics was written mostly by Will Clinger with the help of Jonathan Rees. It was based on the semantics given by Steven Muchnik and Uwe Pleban [4].

$$
\begin{array}{rcl}
e & ::= & c \mid i \mid (e\ e^*) \mid (\texttt{lambda}\ (i^*)\ e^*\ e) \\
 & \mid & (\texttt{lambda}\ (i^*\ .\ i)\ e^*\ e) \mid (\texttt{lambda}\ i\ e^*\ e) \\
 & \mid & (\texttt{if}\ e\ e\ e) \mid (\texttt{if}\ e\ e) \mid (\texttt{set!}\ i\ e). \\
v & ::= & \textsf{symbols} \mid \textsf{characters} \mid \textsf{numbers} \mid \textsf{strings} \\
 & \mid & \textsf{pair}(\ell,\ell) \mid \textsf{vector}(\langle \ell^* \rangle) \mid \textsf{false} \mid \textsf{true} \\
 & \mid & \textsf{null} \mid \textsf{undefined} \mid \textsf{unspecified} \\
 & \mid & \textsf{clsr}(\ell,\langle i^*\rangle,\langle e^*\rangle,e,u) \mid \textsf{cnt}(\ell,k) \\
 & \mid & \textsf{nclsr}(\ell,\langle i^*\rangle,i,\langle e^*\rangle,e,u) \\
 & \mid & \textsf{cons} \mid \textsf{car} \mid \textsf{setcar} \mid \textsf{cwcc} \mid \textsf{and others}. \\
k & ::= & \textsf{args}(\langle e^*\rangle,u,k) \mid \textsf{aug}(\langle v^*\rangle,k) \mid \textsf{perm}(p,k) \\
 & \mid & \textsf{app}(k) \mid \textsf{bind}(\langle i^*\rangle,\langle e^*\rangle,e,u,k) \\
 & \mid & \textsf{rest}(\langle i^*\rangle,\langle v^*\rangle,k) \mid \textsf{seq}(\langle e^*\rangle,e,u,k) \\
 & \mid & \textsf{switch}(e,e,u,k) \mid \textsf{test}(e,u,k) \\
 & \mid & \textsf{assign}(\ell,k) \mid \textsf{halt}.
\end{array}
$$

Figure 2: Some terms

$\mathrm{dom}(f) \cup \mathrm{dom}(g)$, and value

$$(f+g)(x) = \begin{cases} g(x) & x \in \mathrm{dom}(g) \\ f(x) & \text{otherwise.} \end{cases}$$

A sequence is a finite map whose domain is the natural numbers less than the length of the sequence. $\langle v_0 v_1 \cdots v_{n-1}\rangle$ is notation for

$$\{0 \mapsto v_0\} + \{1 \mapsto v_1\} + \cdots + \{n-1 \mapsto v_{n-1}\},$$

$\vec{v}$ is a variable that ranges over sequences, and § is the sequence concatenation operator. A permutation is a sequence which is a one-to-one mapping to its domain. Most other notation follows that of the denotational semantics for Scheme.

3 Rules

The semantics is defined in terms of two judgement forms, $s,u,k \vdash e \Rightarrow a$ and $s,\vec{v} \vdash k \Rightarrow a$. They are patterned after the forms used to specify first-class continuations in ML [1]. Intuitively, the first one asserts that given store s, environment u, and continuation k, expression e evaluates to answer a. The second one asserts that given store s and a sequence of values $\vec{v}$, continuation k computes answer a.

Every rule has the following form. The conclusion is a judgement, and the premise contains one judgement and any number of conditions. A condition is a formula which is not a judgement. Conditions convey a restriction on the applicability of a rule. The form of the rules imply derivations are linear.

Logical variables that are only in the premise of a rule were introduced solely to make the rule intelligible. The right-hand side of the equation defining one of those variables could be substituted in place of each occurrence of the variable it defines.

The storage allocator new must obey the axiom $\textsf{new}(s) \notin \mathrm{dom}(s)$. It is a partial function if storage is finite.

3.1 Constant

$$\frac{s, \langle \textsf{const}(c)\rangle \vdash k \Rightarrow a}{s,u,k \vdash c \Rightarrow a} \tag{1}$$

The definition of const has been deliberately omitted.

3.2 Variable reference

$$\frac{\begin{array}{c} s(u(i)) \neq \textsf{undefined} \\ s, \langle s(u(i))\rangle \vdash k \Rightarrow a \end{array}}{s,u,k \vdash i \Rightarrow a} \tag{2}$$

3.3 Application

An inference using Rule 3 requires the selection of a permutation of the appropriate length.

$$\frac{\begin{array}{c} \#p = \#\langle ee^*\rangle \quad k' = \textsf{perm}(p, \textsf{app}(k)) \\ s, \langle\rangle \vdash \textsf{args}(\textsf{permute}(p^{-1}, \langle ee^*\rangle), u, k') \Rightarrow a \end{array}}{s,u,k \vdash (e\ e^*) \Rightarrow a} \tag{3}$$

$$\textsf{permute}(p,\vec{e}) = \langle \vec{e}(p(0))\, \vec{e}(p(1)) \cdots \vec{e}(p(\#p-1))\rangle$$

$$\frac{s,\vec{v} \vdash k \Rightarrow a}{s,\vec{v} \vdash \textsf{args}(\langle\rangle,u,k) \Rightarrow a} \tag{4}$$

$$\frac{s,u,\textsf{aug}(\vec{v},\textsf{args}(\vec{e},u,k)) \vdash e \Rightarrow a}{s,\vec{v} \vdash \textsf{args}(\langle e\rangle \S \vec{e},u,k) \Rightarrow a} \tag{5}$$

$$\frac{s,\vec{v} \S \langle v\rangle \vdash k \Rightarrow a}{s,\langle v\rangle \vdash \textsf{aug}(\vec{v},k) \Rightarrow a} \tag{6}$$

$$\frac{s,\textsf{unpermute}(p,\vec{v}) \vdash k \Rightarrow a}{s,\vec{v} \vdash \textsf{perm}(p,k) \Rightarrow a} \tag{7}$$

$$\textsf{unpermute}(p,\vec{v}) = \langle \vec{v}(p(0))\, \vec{v}(p(1)) \cdots \vec{v}(p(\#p-1))\rangle$$

$$\frac{\#\vec{v} = \#\vec{\imath} \quad s,\vec{v} \vdash \textsf{bind}(\vec{\imath},\vec{e},e,u,k) \Rightarrow a}{s,\langle \textsf{clsr}(\ell,\vec{\imath},\vec{e},e,u)\rangle \S \vec{v} \vdash \textsf{app}(k) \Rightarrow a} \tag{8}$$

$$\frac{\begin{array}{c} \#\vec{v} \geq \#\vec{\imath} \\ k' = \textsf{bind}(\vec{\imath} \S \langle i\rangle, \vec{e}, e, u, k) \\ s, \langle \textsf{null}\rangle \vdash \textsf{rest}(\vec{\imath},\vec{v},k') \Rightarrow a \end{array}}{s, \langle \textsf{nclsr}(\ell,\vec{\imath},i,\vec{e},e,u)\rangle \S \vec{v} \vdash \textsf{app}(k) \Rightarrow a} \tag{9}$$

$$\frac{s,\vec{v} \vdash k \Rightarrow a}{s,\langle \textsf{cnt}(\ell,k)\rangle \S \vec{v} \vdash \textsf{app}(k') \Rightarrow a} \tag{10}$$

$$\frac{s,\vec{v} \vdash \textsf{seq}(\vec{e},e,u,k) \Rightarrow a}{s,\langle\rangle \vdash \textsf{bind}(\langle\rangle,\vec{e},e,u,k) \Rightarrow a} \tag{11}$$

$$\frac{\ell = \mathsf{new}(s) \quad s' = s + \{\ell \mapsto v\} \qquad s', \vec{v} \vdash \mathsf{bind}(\vec{\imath}, \vec{e}, e, u + \{i \mapsto \ell\}, k) \Rightarrow a}{s, \langle v \rangle \S \vec{v} \vdash \mathsf{bind}(\langle i \rangle \S \vec{\imath}, \vec{e}, e, u, k) \Rightarrow a} \tag{12}$$

$$\frac{\# \vec{v} = \# \vec{\imath} \quad s, \langle v \rangle \vdash \mathsf{aug}(\vec{v}, k) \Rightarrow a}{s, \langle v \rangle \vdash \mathsf{rest}(\vec{\imath}, \vec{v}, k) \Rightarrow a} \tag{13}$$

$$\frac{\# \vec{v} \geq \# \vec{\imath} \qquad s, \langle \mathsf{cons}, v, v' \rangle \vdash \mathsf{app}(\mathsf{rest}(\vec{\imath}, \vec{v}, k)) \Rightarrow a}{s, \langle v' \rangle \vdash \mathsf{rest}(\vec{\imath}, \vec{v} \S \langle v \rangle, k) \Rightarrow a} \tag{14}$$

$$\frac{s, u, k \vdash e \Rightarrow a}{s, \vec{v} \vdash \mathsf{seq}(\langle\rangle, e, u, k) \Rightarrow a} \tag{15}$$

$$\frac{s, u, \mathsf{seq}(\vec{e}, e', u, k) \vdash e \Rightarrow a}{s, \vec{v} \vdash \mathsf{seq}(\langle e \rangle \S \vec{e}, e', u, k) \Rightarrow a} \tag{16}$$

3.4 Abstraction

$$\frac{\ell = \mathsf{new}(s) \quad s' = s + \{\ell \mapsto \mathsf{unspecified}\} \qquad s', \langle \mathsf{clsr}(\ell, \langle i^* \rangle, \langle e^* \rangle, e, u) \rangle \vdash k \Rightarrow a}{s, u, k \vdash (\texttt{lambda}\ (i^*)\ e^*\ e) \Rightarrow a} \tag{17}$$

$$\frac{\ell = \mathsf{new}(s) \quad s' = s + \{\ell \mapsto \mathsf{unspecified}\} \qquad s', \langle \mathsf{nclsr}(\ell, \langle i^* \rangle, i, \langle e^* \rangle, e, u) \rangle \vdash k \Rightarrow a}{s, u, k \vdash (\texttt{lambda}\ (i^*\ .\ i)\ e^*\ e) \Rightarrow a} \tag{18}$$

$$\frac{s, u, k \vdash (\texttt{lambda}\ (.\ i)\ e^*\ e) \Rightarrow a}{s, u, k \vdash (\texttt{lambda}\ i\ e^*\ e) \Rightarrow a} \tag{19}$$

3.5 Conditional

$$\frac{s, u, \mathsf{switch}(e', e'', u, k) \vdash e \Rightarrow a}{s, u, k \vdash (\texttt{if}\ e\ e'\ e'') \Rightarrow a} \tag{20}$$

$$\frac{v \neq \mathsf{false} \quad s, u, k \vdash e \Rightarrow a}{s, \langle v \rangle \vdash \mathsf{switch}(e, e', u, k) \Rightarrow a} \tag{21}$$

$$\frac{s, u, k \vdash e' \Rightarrow a}{s, \langle \mathsf{false} \rangle \vdash \mathsf{switch}(e, e', u, k) \Rightarrow a} \tag{22}$$

$$\frac{s, u, \mathsf{test}(e', u, k) \vdash e \Rightarrow a}{s, u, k \vdash (\texttt{if}\ e\ e') \Rightarrow a} \tag{23}$$

$$\frac{v \neq \mathsf{false} \quad s, u, k \vdash e \Rightarrow a}{s, \langle v \rangle \vdash \mathsf{test}(e, u, k) \Rightarrow a} \tag{24}$$

$$\frac{s, \langle \mathsf{unspecified} \rangle \vdash k \Rightarrow a}{s, \langle \mathsf{false} \rangle \vdash \mathsf{test}(e, u, k) \Rightarrow a} \tag{25}$$

3.6 Assignment

$$\frac{s, u, \mathsf{assign}(u(i), k) \vdash e \Rightarrow a}{s, u, k \vdash (\texttt{set!}\ i\ e) \Rightarrow a} \tag{26}$$

$$\frac{s + \{\ell \mapsto v\}, \langle \mathsf{unspecified} \rangle \vdash k \Rightarrow a}{s, \langle v \rangle \vdash \mathsf{assign}(\ell, k) \Rightarrow a} \tag{27}$$

3.7 Primitives

$$\frac{\ell = \mathsf{new}(s) \quad s' = s + \{\ell \mapsto v\} \qquad \ell' = \mathsf{new}(s') \quad s'' = s' + \{\ell' \mapsto v'\} \qquad s'', \langle \mathsf{pair}(\ell, \ell') \rangle \vdash k \Rightarrow a}{s, \langle \mathsf{cons}, v, v' \rangle \vdash \mathsf{app}(k) \Rightarrow a} \tag{28}$$

$$\frac{s, \langle s(\ell) \rangle \vdash k \Rightarrow a}{s, \langle \mathsf{car}, \mathsf{pair}(\ell, \ell') \rangle \vdash \mathsf{app}(k) \Rightarrow a} \tag{29}$$

$$\frac{s + \{\ell \mapsto v\}, \langle \mathsf{unspecified} \rangle \vdash k \Rightarrow a}{s, \langle \mathsf{setcar}, \mathsf{pair}(\ell, \ell'), v \rangle \vdash \mathsf{app}(k) \Rightarrow a} \tag{30}$$

$$\frac{\ell = \mathsf{new}(s) \quad s' = s + \{\ell \mapsto \mathsf{unspecified}\} \qquad s', \langle v, \mathsf{cnt}(\ell, k) \rangle \vdash \mathsf{app}(k) \Rightarrow a}{s, \langle \mathsf{cwcc}, v \rangle \vdash \mathsf{app}(k) \Rightarrow a} \tag{31}$$

4 Programs

If e^* is a program in which all variables are defined before being referenced or assigned, s is an initial store, and u an initial environment, then the program computes store s' and values $\vec{v}$ if one can show

$$\begin{array}{c} s', \vec{v} \vdash \mathsf{halt} \Rightarrow a \\ \vdots \\ s, u, \mathsf{halt} \vdash e \Rightarrow a, \end{array}$$

where $e = ((\texttt{lambda}\ (i^*)\ e'^*)\ \mathit{undefined}\ \ldots)$, i^* is the sequence of distinct variables defined in e^*, e'^* is the sequence of expressions obtained by replacing every definition in e^* by an assignment, and *undefined* is an expression that evaluates to undefined. A program whose results depend on the selection of a particular set of permutations in Rule 3 is invalid.

5 Operational Semantics

An operational semantics uses an abstract machine to define the semantics of a language. The terms of this operational semantics map identically from the natural semantics, and the machine states are in one of two forms: $\mathsf{e}(e, u, k, s)$ or $\mathsf{k}(k, \vec{v}, s)$. When the abstract machine is executing a program, there is the obvious correspondence between the states of the abstract machine, and the steps of the program's derivation. Allowed machine transitions correspond to inferences in the natural semantics read backwards from conclusion to premise.

The abstract machine so defined is nondeterministic. The choice points occur at transitions that correspond to an instance of Rule 3.

6 Soundness

The Scheme semantics as given is not sound with respect to the denotational definition. The reason is the natural semantics allows the evaluation order of different applications to differ within the same program. In this section, assume that for each natural number, there is one permutation of that length which is chosen when using Rule 3. This is the same assumption made in the denotational semantics.

The natural semantics of Scheme was designed to allow a translation of each rule into a rule valid in the denotational semantics. The translation allows one to view the possible deductions in the natural semantics as a subset of the possible deductions in the denotational semantics.

Semantic functions assign meanings to terms in the natural semantics. The signature of each semantic function used is given in Figure 3 along with a partial definition of some of the semantic functions. A logical variable is translated by replacing it with a variable from the domain corresponding to the logical variable's type.

For the purposes of the translation, it is convenient to consider a modified set of rules. Observe there are two kinds of conditions in the rules. Some define variables by equations. These can be eliminated by substitution. The remaining conditions are of the form $\#\vec{v} = \#\vec{\imath}$, $\#\vec{v} \geq \#\vec{\imath}$, $v \neq \mathsf{undefined}$, or $v \neq \mathsf{false}$. Each rule has no more than one of these conditions. Let r be the condition if it exists, otherwise $\mathsf{false} = \mathsf{false}$. Let $\Sigma \Rightarrow a$ be the judgement in the conclusion after defined variables have been eliminated by substitution, and $\Sigma' \Rightarrow a$ be the major premise. All modified rules have a simple form.

$$\frac{r \quad \Sigma' \Rightarrow a}{\Sigma \Rightarrow a}$$

A modified rule is translated into a rule involving equations.

$$\frac{\mathcal{R}\llbracket r\rrbracket = \mathit{true} \quad \mathcal{J}\llbracket\Sigma'\rrbracket = \mathcal{A}\llbracket a\rrbracket}{\mathcal{J}\llbracket\Sigma\rrbracket = \mathcal{A}\llbracket a\rrbracket}$$

The definition of $\mathcal{J}$ is given in Figure 3, the definition of $\mathcal{R}$ is obvious, and the definition of $\mathcal{A}$ is arbitrary. Each rule is justified by a valid equation.

$$(\mathcal{R}\llbracket r\rrbracket \rightarrow \mathcal{J}\llbracket\Sigma'\rrbracket, \bot) = (\mathcal{R}\llbracket r\rrbracket \rightarrow \mathcal{J}\llbracket\Sigma\rrbracket, \bot)$$

Since each inference must satisfy its rule's condition, a derivation is justified by a single closed equation.

For example, the translation of the rule for constants (Rule 1) is justified by the following valid equation.

$$\mathcal{E}\llbracket c\rrbracket\rho\kappa\sigma = \kappa\langle\mathcal{K}\llbracket c\rrbracket\rangle\sigma$$

$\mathcal{E}$	expression	$e \rightarrow \mathrm{U} \rightarrow \mathrm{K} \rightarrow \mathrm{C}$
$\mathcal{L}$	location	$\ell \rightarrow \mathrm{L}$
$\mathcal{V}$	expressed value	$v \rightarrow \mathrm{E}$
$\mathcal{V}^*$	value sequence	$v^* \rightarrow \mathrm{E}^*$
$\mathcal{S}$	store	$s \rightarrow \mathrm{S}$
$\mathcal{U}$	environment	$u \rightarrow \mathrm{U}$
$\mathcal{K}'$	continuation	$k \rightarrow \mathrm{K}$
$\mathcal{A}$	answer	$a \rightarrow \mathrm{A}$

$$\begin{array}{l}
\mathcal{J}\llbracket s, u, k \vdash e\rrbracket = \mathcal{E}\llbracket e\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\mathcal{K}'\llbracket k\rrbracket)(\mathcal{S}\llbracket s\rrbracket) \\
\mathcal{J}\llbracket s, \langle v^*\rangle \vdash k\rrbracket = \mathcal{K}'\llbracket k\rrbracket(\mathcal{V}^*\llbracket v^*\rrbracket)(\mathcal{S}\llbracket s\rrbracket) \\
\mathcal{V}^*\llbracket\,\rrbracket = \langle\rangle \\
\mathcal{V}^*\llbracket vv^*\rrbracket = \langle\mathcal{V}\llbracket v\rrbracket\rangle \mathbin{\S} \mathcal{V}^*\llbracket v^*\rrbracket \\
\mathcal{V}\llbracket\mathsf{clsr}(\ell, \langle i^*\rangle, \langle e^*\rangle, e, u)\rrbracket = \langle\mathcal{L}\llbracket\ell\rrbracket, \lambda\epsilon^*\kappa.\cdots\rangle \text{ in E} \\
\mathcal{V}\llbracket\mathsf{cnt}(\ell, k)\rrbracket = \langle\mathcal{L}\llbracket\ell\rrbracket, \lambda\epsilon^*\kappa.\,\mathcal{K}'\llbracket k\rrbracket\epsilon^*\rangle \text{ in E} \\
\mathcal{V}\llbracket\mathsf{cons}\rrbracket = \langle\alpha, \mathit{cons}\rangle \text{ in E} \\
\quad \text{where } \alpha \text{ is as given in the initial store.} \\
\mathcal{V}\llbracket\cdots\rrbracket = \text{and others}\ldots \\
\mathcal{K}'\llbracket\mathsf{args}(\langle e^*\rangle, u, k)\rrbracket = \\
\quad \lambda\epsilon^*.\,\mathcal{E}^*\llbracket e^*\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\lambda\epsilon'^*.\,\mathcal{K}'\llbracket k\rrbracket(\epsilon^* \mathbin{\S} \epsilon'^*)) \\
\mathcal{K}'\llbracket\mathsf{aug}(\langle v^*\rangle, k)\rrbracket = \mathit{single}\,\lambda\epsilon.\,\mathcal{K}'\llbracket k\rrbracket(\mathcal{V}^*\llbracket v^*\rrbracket \mathbin{\S} \langle\epsilon\rangle) \\
\mathcal{K}'\llbracket\mathsf{perm}(p, k)\rrbracket = \lambda\epsilon^*.\,\mathcal{K}'\llbracket k\rrbracket(\mathit{unpermute}\,\epsilon^*) \\
\mathcal{K}'\llbracket\mathsf{app}(k)\rrbracket = \\
\quad \lambda\epsilon^*.\,\mathit{applicate}(\epsilon^* \downarrow 1)(\epsilon^* \dagger 1)(\mathcal{K}'\llbracket k\rrbracket) \\
\mathcal{K}'\llbracket\mathsf{bind}(\langle i^*\rangle, \langle e^*\rangle, e, u, k)\rrbracket = \\
\quad \lambda\epsilon^*. \\
\qquad \#\epsilon^* = \#i^* \rightarrow \\
\qquad\quad \mathit{tievals}(\lambda\alpha^*.\,(\lambda\rho.\,\mathcal{C}\llbracket e^*\rrbracket\rho(\mathcal{E}\llbracket e\rrbracket\rho(\mathcal{K}'\llbracket k\rrbracket))) \\
\qquad\qquad\qquad\qquad (\mathit{extends}(\mathcal{U}\llbracket u\rrbracket)i^*\alpha^*)) \\
\qquad\qquad\quad \epsilon^*, \\
\qquad \mathit{wrong} \text{ “wrong number of arguments”} \\
\mathcal{K}'\llbracket\mathsf{rest}(\langle i^*\rangle, \langle v^*\rangle, k)\rrbracket = \\
\quad \mathit{rest}(\#v^* - \#i^*)(\mathcal{V}^*\llbracket v^*\rrbracket)(\mathcal{K}'\llbracket k\rrbracket) \\
\mathit{rest} = \lambda\nu\epsilon^*\kappa.\,\mathit{single}\,\lambda\epsilon. \\
\quad \nu = 0 \rightarrow \kappa(\epsilon^* \mathbin{\S} \langle\epsilon\rangle), \\
\quad \mathit{cons}\langle\epsilon^* \downarrow \#\epsilon^*, \epsilon\rangle \\
\qquad\quad (\mathit{rest}(\nu - 1)(\mathit{takefirst}(\#\epsilon^* - 1)\epsilon^*)\kappa) \\
\mathcal{K}'\llbracket\mathsf{seq}(\langle e^*\rangle, e, u, k)\rrbracket = \\
\quad \lambda\epsilon^*.\,\mathcal{C}\llbracket e^*\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\mathcal{E}\llbracket e\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\mathcal{K}'\llbracket k\rrbracket)) \\
\mathcal{K}'\llbracket\mathsf{switch}(e, e', u, k)\rrbracket = \\
\quad \mathit{single}\,\lambda\epsilon.\,\epsilon = \mathit{false} \rightarrow \mathcal{E}\llbracket e'\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\mathcal{K}'\llbracket k\rrbracket), \\
\qquad\qquad\quad \mathcal{E}\llbracket e\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\mathcal{K}'\llbracket k\rrbracket) \\
\mathcal{K}'\llbracket\mathsf{test}(e, u, k)\rrbracket = \\
\quad \mathit{single}\,\lambda\epsilon.\,\epsilon = \mathit{false} \rightarrow \mathcal{K}'\llbracket k\rrbracket\langle\mathit{unspecified}\rangle, \\
\qquad\qquad\quad \mathcal{E}\llbracket e\rrbracket(\mathcal{U}\llbracket u\rrbracket)(\mathcal{K}'\llbracket k\rrbracket) \\
\mathcal{K}'\llbracket\mathsf{assign}(\ell, k)\rrbracket = \\
\quad \mathit{single}\,\lambda\epsilon.\,\mathit{assign}(\mathcal{L}\llbracket\ell\rrbracket) \\
\qquad\qquad\qquad\qquad \epsilon \\
\qquad\qquad\qquad\qquad (\mathcal{K}'\llbracket k\rrbracket\langle\mathit{unspecified}\rangle)) \\
\mathcal{K}'\llbracket\mathsf{halt}\rrbracket = \text{the initial continuation.}
\end{array}$$

Figure 3: Semantic functions

The case of lambda expressions (Rule 17) shows how memory allocation is handled.

$$\begin{aligned}&\mathcal{E}[\![(\texttt{lambda}\ (i^*)\ e^*\ e)]\!]\rho\kappa\sigma = \\ &\kappa \\ &\langle\langle \mathit{new}\,\sigma, \lambda\epsilon^*\kappa.\cdots\rangle \text{ in E}\rangle \\ &(\mathit{update}(\mathit{new}\,\sigma \mid \text{L})\ \mathit{unspecified}\,\sigma)\end{aligned}$$

The case of evaluating a sequence (Rule 16) is justified by using the semantic function for both expressions and commands.

$$\mathcal{C}[\![e\ e^*]\!]\rho(\mathcal{E}[\![e']\!]\rho\kappa) = \mathcal{E}[\![e]\!]\rho(\lambda\epsilon^*.\,\mathcal{C}[\![e^*]\!]\rho(\mathcal{E}[\![e']\!]\rho\kappa))$$

A case in which the justification is not obvious is the case of evaluating a sequence of expressions in preparation for applying a function (Rules 4–6). Their justification requires the definition of an auxiliary function.

$$\mathcal{F}[\![e^*]\!]\epsilon^*\rho\kappa = \mathcal{E}^*[\![e^*]\!]\rho(\lambda\epsilon'^*.\,\kappa(\epsilon^*\ \S\ \epsilon'^*))$$

Occurrences of $\mathcal{E}^*[\![e]\!]$ can be replaced by $\mathcal{F}[\![e]\!]\langle\rangle$ in the denotational semantics.

The justification for Rules 5–6 comes from the use of the following equation which has been derived from the definition of $\mathcal{F}$.

$$\mathcal{F}[\![e\ e^*]\!]\epsilon^*\rho\kappa = \mathcal{E}[\![e]\!]\rho(\mathit{single}\,\lambda\epsilon.\,\mathcal{F}[\![e^*]\!](\epsilon^*\ \S\ \langle\epsilon\rangle)\rho\kappa)$$

The justification for Rule 4 is also derived from the definition of $\mathcal{F}$.

$$\mathcal{F}[\![\ ]\!]\epsilon^*\rho\kappa = \kappa\epsilon^*$$

Another non-obvious case is the case of allocating a list for rest arguments (Rules 13–14). Their justifications can be derived from the following three identities.

$$\begin{aligned}&\mathit{list}(\epsilon^*\ \S\ \epsilon'^*)\kappa = \mathit{list}\,\epsilon'^*(\mathit{rest}\,\#\epsilon^*\epsilon^*\kappa) \\ &\mathit{rest}\,\#\epsilon^*(\epsilon'^*\ \S\ \epsilon^*)\kappa = \\ &\quad \mathit{rest}\,\#\epsilon^*\epsilon^*(\mathit{single}\,\lambda\epsilon.\,\kappa(\epsilon'^*\ \S\ \langle\epsilon\rangle)) \\ &\mathit{tievalsrest}\,\psi\epsilon^*\nu = \mathit{rest}\,\nu\epsilon^*(\mathit{tievals}\,\psi)\langle\mathit{null}\rangle\end{aligned}$$

7 Conclusion

An operational semantics of Scheme has been defined by describing an abstract machine. Its allowed transition sequences are specified by the set of derivations in the natural semantics of Scheme. A translation establishes that each derivation in Scheme's natural semantics is justified by a valid equation in Scheme's denotational semantics, when the natural semantics is restricted so as to assume that throughout a program, there is one unspecified order used to evaluate applications.

Acknowledgement: Joshua Guttman, Leonard Monk, and Vipin Swarup made many useful suggestions.

References

[1] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, New York, NY, 1991. ACM Press.

[2] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[3] Giles Kahn. Natural semantics. In F. J. Bandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Berlin, 1987. Springer-Verlag.

[4] Steven S. Muchnik and Uwe Pleban. A semantic comparison of Lisp and Scheme. In *Conference Record of the 1980 Lisp Conference*, pages 54–64, 1980.

[5] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Wm. C. Brown, Dubuque, IO, 1986.

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

Fundamental Rules of
Writing, Editing, and Publishing

1. Don't use no double negatives.
2. Make each pronoun agree with their antecedent.
3. Join clauses good, like a conjunction should.
4. About them sentence fragments.
5. When dangling, watch them participles.
6. Verbs has to agree their subjects.
7. Just between you and I, case is important too.
8. Don't write run-on sentences they are hard to read.
9. Don't use commas, which aren't necessary.
10. Try to not ever split infinitives.
11. Its important to use your apostrophe's correctly.
12. Proofread your writing to see if you any words left out.
13. Correct spelling is absolutcley essential.
14. Don't abbr.
15. You've heard it a million times: avoid hyperbole.