A Compositional Semantics for eval in Scheme

Mosses, Peter D.

DOI
10.1145/3759427.3760369

Publication date
2025

Document Version
Final published version

Published in
OLIVIERFEST 2025 - Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday

Citation (APA)
Mosses, P. D. (2025). A Compositional Semantics for eval in Scheme. In F. Henglein, J. Lawall, J. Palsberg, & S. Ilya (Eds.), *OLIVIERFEST 2025 - Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday* (pp. 72-81). ACM. https://doi.org/10.1145/3759427.3760369

A Compositional Semantics for `eval` in Scheme

Peter D. Mosses
Delft University of Technology
Delft, Netherlands
Swansea University
Swansea, United Kingdom
p.d.mosses@tudelft.nl

Abstract

The Scheme programming language was introduced in 1975 as a lexically-scoped dialect of Lisp. It has subsequently been improved and extended through many rounds of standardization, documented by the Scheme Reports.

In Lisp, `eval` is a function taking a single argument; when the value of the argument represents an expression, `eval` proceeds to evaluate it in the current environment. The Scheme procedure `eval` is similar, but takes an extra argument that determines the evaluation environment.

The so-called classic standards for Scheme and its latest modern standard all include a denotational semantics of core expressions and of some required procedures. However, the semantics does not cover `eval`. When Muchnick and Pleban compared the denotational semantics of Lisp and Scheme in 1980, they wrote that it was not possible to give a structural semantics of `eval` expressions. Similarly, in 1984, Clinger pointed out that the semantics of `eval` violates compositionality; he suggested to define `eval` using a non-compositional copy of the semantic function.

This paper adapts Clinger's suggestion by letting the semantic function take an extra argument, then defining that argument by an explicit fixed point. The resulting semantics of `eval` expressions is fully compositional. The wellformedness of the semantics has been tested using a lightweight Agda formalization; verifying that the denotations have expected properties is work in progress.

CCS Concepts: • **Theory of computation** → **Denotational semantics**; • **Software and its engineering** → **Semantics**; **Functional languages**.

Keywords: Scheme, denotational semantics, compositional semantics, quote and eval, Lisp, formalization, Agda

ACM Reference Format:
Peter D. Mosses. 2025. A Compositional Semantics for `eval` in Scheme. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3759427.3760369

1 Introduction

The Scheme programming language was introduced 50 years ago, in 1975 [25, 26]. See the Scheme language website[1] or Wikipedia page[2] for introductions to Scheme, and for information about the Scheme standards: the *Revised^n Reports on the Algorithmic Language Scheme* (abbreviated R^{n}RS).

Scheme is a lexically-scoped dialect of Lisp. In Lisp, `eval` is a function taking a single argument; when the value of the argument represents an expression, `eval` proceeds to evaluate it in the current environment. The Scheme procedure `eval` (introduced in 1998 in R^{5}RS [14]) is similar, but takes an extra argument that determines the evaluation environment.

The IEEE Scheme standard [13], the so-called classic standards [6, 14, 20], and the latest modern Scheme standard [21] all include a denotational semantics of core expressions, and of some required procedures. However, the denotational semantics in the Scheme standards does not cover `eval`. The small-step operational semantics given in R^{6}RS [23] covers a much larger subset of Scheme than the denotational semantics, but still omits `eval` (despite being based on [15], which includes it).

When Muchnick and Pleban [18] compared the denotational semantics of Lisp and Scheme in 1980, they wrote that it was not possible to give a structural semantics of `eval` expressions, because they involve evaluation of an expression that is not a subexpression. In 1984, Clinger [5] pointed out that "in order to preserve the principle of compositionality its semantics would have to be definable without reference to the semantic function $\mathcal{E}$ [for expressions]". He suggested to define `eval` using a non-compositional copy $\mathcal{E}'$ of the semantic function, or an infinite sequence of such functions, as in the semantics of reflection in Lisp and Scheme [10, 19, 28, 29]. And in 1987, Recanati and Deutsch [19] wrote "The introduction of `eval` is problematic in the denotational approach because it leads to the following equation:

$$\mathcal{E}[\![\texttt{(eval E)}]\!]\rho = \mathcal{E}[\![\mathcal{E}[\![\mathrm{E}]\!]\rho]\!]\rho$$

which is not structural".

[1] https://www.scheme.org
[2] https://en.wikipedia.org/wiki/Scheme_(programming_language)

However, it turns out that a compositional semantics of (eval E) can be given by adapting Clinger's cited suggestion so that $\mathcal{E}$ *takes* $\mathcal{E}'$ *as an extra argument.* The denotation $\mathcal{E}[\![\texttt{(eval E)}]\!]$ can then be defined compositionally:

$$\mathcal{E}[\![\texttt{(eval E)}]\!]\,\mathcal{E}'\rho = \mathcal{E}'[\![\mathcal{E}[\![\mathrm{E}]\!]\,\mathcal{E}'\rho]\!]\rho$$

To define the extra argument $\mathcal{E}'$, it is convenient to write it *before* the syntactic argument of $\mathcal{E}$, and rename $\mathcal{E}$ to $\mathcal{F}$:

$$\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(eval E)}]\!]\,\rho = \mathcal{E}'[\![\mathcal{F}\,\mathcal{E}'\,[\![\mathrm{E}]\!]\rho]\!]\rho$$

Then $\mathcal{E}'$ can be defined as the (least) fixed point of $\mathcal{F}$.

Related Work. As noted by the anonymous reviewers of the submitted version "the technique used here is a form of open recursion: rather than recursing directly to the (semantic) function being defined, an additional parameter is given to use for (problematic) recursive applications, and the loop is closed by a top-level fixed point". Wand and Friedman [28] use an explicit fixed point in their semantics of the reflective tower, but it appears that this approach has not been applied to the semantics of eval, despite the conceptual similarity to reflection. Cook and Palsberg [8, Fig. 16] also use an explicit fixed point to obtain the behavior of a method system from their denotational semantics of method expressions.

Outline. Section 2 briefly recalls the notation used in the Scheme reports for defining the denotational semantics of core expressions and procedures.

Section 3 presents a denotational semantics for an ad hoc sublanguage Scm of Scheme. Apart from literal constants (integers and truth values) the syntax of Scm includes only four kinds of expression: procedure calls, lambda-abstractions, conditionals, and assignments.

Section 4 extends Scm to a language ScmQ by adding literal quotations. The denotational semantics of core expressions in the Scheme standards omits quotations, although their semantics seems reasonably interesting, and not too complicated to define.

Section 5 defines an extension ScmQE of ScmQ by adding (eval E) as a primitive expression. For simplicity, the environment for evaluating the expression represented by the value of E is null, and left implicit. The addition of the extra argument of the semantic function for expressions requires reformulation of the denotations of ScmQ expressions.

The compositionality of the semantics of ScmQE has been tested using a lightweight formalization of denotational semantics in Agda. An archive of the Agda source code is available as supplemental material [17] accompanying this paper. Section 6 explains the relationship of the formalization to the semantic definitions in the paper.

Section 7 concludes by considering the possibility of including the presented denotational semantics of quotations, and of an eval procedure corresponding to that currently provided in a library, in some future revision of the Scheme standard.

Readers are assumed to be familiar with the Scheme language and with the basic concepts of denotational semantics. Familiarity with the denotational semantics of core Scheme expressions in the Scheme reports is not required.

Dedication. This paper is dedicated to Olivier Danvy, and published in his Festschrift at SPLASH 2025. Olivier was a colleague of mine at Aarhus for many years, and I recall the pleasure of collaborating with him on teaching concepts of programming languages and formal semantics.

I also recall the general astonishment when *Nature* [4] reported the great extent to which authors have explicitly acknowledged Olivier's help in their publications [12]. The Scheme reports are no exception: Olivier is among those thanked for their help with the $\mathrm{R^4RS}$ and $\mathrm{R^5RS}$ standards [6, 14]; he was also listed as a member of the Scheme Working Group of the IEEE Microprocessor and Microcomputer Standards Subcommittee, and he participated in the preparation of the IEEE standard [13].

Unfortunately, Olivier and I have never co-authored any publications, despite our common interest in denotational semantics, structural operational semantics, and abstract machines, our joint participation in the BRICS research center at Aarhus, and our separate publications on topics of mutual interest (e.g., [1, 2, 11]). Nor did I ever follow his advice to start programming in Scheme, which he had exploited to great effect in his research on partial evaluation. I left Aarhus in the mid-2000s, and since then our paths have seldom crossed.

In 2009, however, Olivier kindly contributed a two-part major paper [3, 9] to my Festschrift, and presented it at the symposium. In Part I, he derives a functional representation of a denotational semantics corresponding to Clinger's abstract machine for Scheme [7], and compares it with the denotational semantics of Scheme in $\mathrm{R^3RS}$; the focus in Part II (with Małgorzata Biernacka) is on a reduction semantics with explicit substitutions that also corresponds to Clinger's abstract machine.

One of Olivier's earliest publications is *Intensions and Extensions in a Reflective Tower* [10] (with Karoline Malmkjær). It presents a model of the reflective tower based on the formal semantics of its levels, including a (structural) denotational semantics for a compositional subset of the model, and a Scheme implementation of a reflective tower. It appears that he hasn't published further papers on reflection, but I hope he may find at least some interest in the topic of this paper.

2 Notation

This section briefly recalls the notation used in the Scheme reports for defining domains and functions in denotational semantics. Stoy's textbook [24] explains the notation together with its original mathematical foundations. Tennent's introductory article [27] gives a concise introduction to the same notation, including illustrative examples.

Table 1. Notation for domains

$S_\perp$	flat domain with elements $S \cup \{\perp\}$
$D \to D'$	continuous function domain
$D \times D'$	cartesian product domain
$D + D'$	separated sum domain
D^*	possibly-empty sequence domain
D^+	nonempty sequence domain

Table 1 summarizes the notation used for basic domain constructors. Domains can be defined recursively in terms of these constructors (familiarity with the foundations of domain theory and the mathematical structure of domains is not required). The set $S \to D$ of all functions from a set S to a domain D can be regarded as a domain (e.g., the domain of environments defined in Section 3 consists of functions from a set of identifiers to a domain of locations).

Table 2. Notation for values

$\perp$	undefined element of any domain
$x = y$	equality in a flat domain $S_\perp$
$t \to a, b$	McCarthy conditional
$\lambda x. \cdots x \cdots$	function mapping x to $\cdots x \cdots$
fix	fixed point of functions in $D \to D$
$f[y/x]$	mapping x to y, otherwise as f
$\langle x, x' \rangle$	pair in product domain $D \times D'$
$x \text{ in } D$	injection into sum domain D
$x \in D$	test if in summand domain D
$x \mid D$	projection to summand domain D
$\langle \cdots, x, \cdots \rangle$	sequence in domain D^* or D^+
$\# s$	length of sequence s
$s \S t$	concatenation of sequences s and t
$s \downarrow k$	kth member of sequence or pair s
$s \dagger k$	drop first k members of sequence s

Table 2 summarizes the notation for values. Functions between domains are always continuous when defined in terms of this notation. The restriction of equality tests $x = y$ to flat domains is required for continuity.

3 Scm: A Simple Sublanguage of Scheme

This section presents a denotational semantics for a simple ad hoc sublanguage Scm of Scheme. Scm omits various features of the sublanguage defined in the Scheme reports:

- multiple return values;
- lambda-abstractions with fixed numbers of arguments;
- unspecified order of subexpression evaluation;
- command sequences and cond-expressions;
- immutable objects;
- characters, vectors, and strings;
- dynamic points; and
- error reports.

The definition of Scm has the same structure as the denotational semantics given in the Scheme reports: the abstract syntax and domain equations precede the definitions of the semantic functions, but the definitions of auxiliary functions are deferred to the end.

Table 3. Scm: Abstract syntax

$\mathrm{Z} \in \mathrm{Int}$	integers
$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{I} \in \mathrm{Ide}$	identifiers
$\mathrm{E} \in \mathrm{Exp}$	expressions

$$\mathrm{Con} \longrightarrow \mathrm{Z} \mid \texttt{\#t} \mid \texttt{\#f}$$
$$\mathrm{Exp} \longrightarrow \mathrm{K} \mid \mathrm{I} \mid \texttt{(E E}^*\texttt{)} \mid \texttt{(lambda I E)}$$
$$\mid \texttt{(if E E}_1\texttt{ E}_2\texttt{)} \mid \texttt{(set! I E)}$$

Scm: Abstract Syntax. The Scm constructs in Table 3 include literal integers and booleans, but not quotations, which are to be added in Section 4.

Table 4. Scm: Domain equations

$\alpha \in \mathbf{L}$	locations
$\nu \in \mathbf{N} = \mathrm{Nat}_\perp$	natural numbers
$\tau \in \mathbf{T} = \{\mathit{false}, \mathit{true}\}_\perp$	booleans
$\mathbf{R} = \mathrm{Int}_\perp$	integer numbers
$\mathbf{P} = \mathbf{L} \times \mathbf{L}$	pairs
$\mathbf{M} = \{\mathit{null}, \mathit{unallocated}, \mathit{undefined}, \mathit{unspecified}\}_\perp$	
$\mathbf{F} = \mathbf{E}^* \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$	procedures
$\epsilon \in \mathbf{E} = \mathbf{T} + \mathbf{R} + \mathbf{P} + \mathbf{M} + \mathbf{F}$	expressed values
$\sigma \in \mathbf{S} = \mathbf{L} \to \mathbf{E}$	stores
$\rho \in \mathbf{U} = \mathrm{Ide} \to \mathbf{L}$	environments
$\theta \in \mathbf{C} = \mathbf{S} \to \mathbf{A}$	continuations
$\mathbf{A}$	answers

Scm: Domain Equations. Table 4 defines domains for the Scm semantics. Some of them are simpler than the domains defined for the semantics in the Scheme standards:

- **P** corresponds to $\mathbf{E_P}$, but omits the truth-value that distinguishes between mutable and immutable pairs;
- **M** omits *false* and *true*, letting **T** be a summand of **E**;
- **F** omits the location component of procedures;
- **E** omits symbols, characters, vectors, and strings; and
- **S** avoids the need for pairing stored values with truth-values by adding the value *unallocated* to **E**.

Table 5. Scm: Semantic functions

$$\mathcal{K} : \text{Con} \to \mathbf{E}$$
$$\mathcal{E} : \text{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathcal{E}^* : \text{Exp}^* \to \mathbf{U} \to (\mathbf{E}^* \to \mathbf{C}) \to \mathbf{C}$$

$$\mathcal{K}[\![Z]\!] = Z \text{ in } \mathbf{E} \tag{1}$$
$$\mathcal{K}[\![\texttt{\#t}]\!] = \mathit{true} \text{ in } \mathbf{E} \tag{2}$$
$$\mathcal{K}[\![\texttt{\#f}]\!] = \mathit{false} \text{ in } \mathbf{E} \tag{3}$$

$$\mathcal{E}[\![\mathrm{K}]\!]\rho\kappa = \kappa(\mathcal{K}[\![\mathrm{K}]\!]) \tag{4}$$
$$\mathcal{E}[\![\mathrm{I}]\!]\rho\kappa = \mathit{hold}\,(\rho\,\mathrm{I})\,\kappa \tag{5}$$
$$\begin{aligned}&\mathcal{E}[\![\texttt{(E E}^*\texttt{)}]\!]\rho\kappa = \\ &\quad\mathcal{E}[\![\mathrm{E}]\!]\rho(\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\rho(\lambda\epsilon^*.\,(\epsilon \mid \mathbf{F})\,\epsilon^*\kappa))\end{aligned} \tag{6}$$
$$\begin{aligned}&\mathcal{E}[\![\texttt{(lambda I E)}]\!]\rho\kappa = \\ &\quad\kappa\,((\lambda\epsilon^*\kappa'.\,\mathit{list}\,\epsilon^*\,(\lambda\epsilon. \\ &\qquad\qquad \mathit{alloc}\,\epsilon\,(\lambda\alpha.\,\mathcal{E}[\![\mathrm{E}]\!]\,(\rho[\alpha/\mathrm{I}])\,\kappa')) \\ &\qquad)\text{ in } \mathbf{E})\end{aligned} \tag{7}$$
$$\begin{aligned}&\mathcal{E}[\![\texttt{(if E E}_1\texttt{ E}_2\texttt{)}]\!]\rho\kappa = \\ &\quad\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\lambda\epsilon. \\ &\qquad \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\,\rho\,\kappa, \mathcal{E}[\![\mathrm{E}_2]\!]\,\rho\,\kappa)\end{aligned} \tag{8}$$
$$\begin{aligned}&\mathcal{E}[\![\texttt{(set! I E)}]\!]\rho\kappa = \\ &\quad\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\lambda\epsilon.\,\mathit{assign}\,(\rho\,\mathrm{I})\,\epsilon\,(\kappa\,\mathit{unspecified}))\end{aligned} \tag{9}$$

$$\mathcal{E}^*[\![\quad]\!]\rho\kappa = \kappa\,\langle\,\rangle \tag{10}$$
$$\begin{aligned}&\mathcal{E}^*[\![\mathrm{E}\ \mathrm{E}^*]\!]\rho\kappa = \\ &\quad\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*.\,\kappa\,(\langle\epsilon\rangle \,\S\, \epsilon^*)))\end{aligned} \tag{11}$$

Scm: Semantic Functions. The denotations of expressions and expression sequences in Table 5 are defined in continuation-passing style. The argument κ ranges over the domain $\mathbf{E} \to \mathbf{C}$ for expressions, and over $\mathbf{E}^* \to \mathbf{C}$ for expression sequences.

Most of the auxiliary functions used in the semantics of Scm have the same interpretation as in the Scheme reports:

- $\mathit{hold}\,\alpha\,\kappa$ gives the value stored in location α;
- $\mathit{list}\,\epsilon^*\,\kappa$ allocates a list of locations to store the components of ϵ^*;
- $\mathit{alloc}\,\epsilon\,\kappa$ allocates a location α and initializes it to ϵ, then applies κ to α;
- $\mathit{assign}\,\alpha\,\epsilon\,\theta$ stores ϵ in location α.

As usual in continuation-passing style, the current store σ is implicitly single-threaded by composing continuations.

Notice that, as in Scheme, a procedure `(lambda I E)` can be called with any number of arguments. For example, `((lambda x x) 3 4 5 6)` returns the list `(3 4 5 6)`; and `((lambda x x) 1)` returns the list `(1)`.

Table 6. Scm: Auxiliary functions

$$\mathit{assign} : \mathbf{L} \to \mathbf{E} \to \mathbf{C} \to \mathbf{C}$$
$$\mathit{assign}\,\alpha\epsilon\theta = \lambda\sigma.\,\theta\,(\sigma[\epsilon/\alpha])$$

$$\mathit{hold} : \mathbf{L} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathit{hold}\,\alpha\kappa = \lambda\sigma.\,\kappa\,(\sigma\,\alpha)\,\sigma$$

$$\mathit{new} : (\mathbf{L} \to \mathbf{C}) \to \mathbf{C} \quad \text{[implementation dependent]}$$

$$\mathit{alloc} : \mathbf{E} \to (\mathbf{L} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathit{alloc}\,\epsilon\kappa = \mathit{new}\,(\lambda\alpha.\,\mathit{assign}\,\alpha\,\epsilon\,(\kappa\,\alpha))$$

$$\mathit{truish} : \mathbf{E} \to \mathbf{T}$$
$$\begin{aligned}\mathit{truish}\,\epsilon = \epsilon \text{ in } \mathbf{T} \longrightarrow ((\epsilon \mid \mathbf{T}) = \mathit{false} \longrightarrow \mathit{false}, \mathit{true}), \\ \mathit{true}\end{aligned}$$

$$\mathit{cons} : \mathbf{F}$$
$$\begin{aligned}\mathit{cons}\,\epsilon^*\kappa = \#\,\epsilon^* = 2 \longrightarrow \mathit{alloc}\,(\epsilon^* \downarrow 1)\,(\lambda\alpha_1. \\ \mathit{alloc}\,(\epsilon^* \downarrow 2)\,(\lambda\alpha_2. \\ \kappa\,(\langle\alpha_1, \alpha_2\rangle \text{ in } \mathbf{E}))), \bot\end{aligned}$$

$$\mathit{list} : \mathbf{F}$$
$$\begin{aligned}\mathit{list}\,\epsilon^*\kappa = \#\,\epsilon^* = 0 \longrightarrow \kappa\,(\mathit{null} \text{ in } \mathbf{E}), \\ \mathit{list}\,(\epsilon^* \dagger 1)\,(\lambda\epsilon.\,\mathit{cons}\,\langle\epsilon^* \downarrow 1, \epsilon\rangle\,\kappa)\end{aligned}$$

$$\mathit{car} : \mathbf{F}$$
$$\mathit{car}\,\epsilon^*\kappa = \#\,\epsilon^* = 1 \longrightarrow \mathit{hold}\,((\epsilon^* \downarrow 1 \mid \mathbf{P}) \downarrow 1)\,\kappa, \bot$$

$$\mathit{cdr} : \mathbf{F}$$
$$\mathit{cdr}\,\epsilon^*\kappa = \#\,\epsilon^* = 1 \longrightarrow \mathit{hold}\,((\epsilon^* \downarrow 1 \mid \mathbf{P}) \downarrow 2)\,\kappa, \bot$$

$$\mathit{setcar} : \mathbf{F}$$
$$\begin{aligned}&\mathit{setcar}\,\epsilon^*\kappa = \\ &\quad \#\,\epsilon^* = 2 \longrightarrow \mathit{assign}\,((\epsilon^* \downarrow 1) \mid \mathbf{P}) \downarrow 1) \\ &\qquad\qquad (\epsilon^* \downarrow 2) \\ &\qquad\qquad (\kappa\,(\mathit{unspecified} \text{ in } \mathbf{E})), \bot\end{aligned}$$

$$\mathit{setcdr} : \mathbf{F}$$
$$\begin{aligned}&\mathit{setcdr}\,\epsilon^*\kappa = \\ &\quad \#\,\epsilon^* = 2 \longrightarrow \mathit{assign}\,((\epsilon^* \downarrow 1) \mid \mathbf{P}) \downarrow 2) \\ &\qquad\qquad (\epsilon^* \downarrow 2) \\ &\qquad\qquad (\kappa\,(\mathit{unspecified} \text{ in } \mathbf{E})), \bot\end{aligned}$$

Scm: Auxiliary Functions. The definitions of the auxiliary functions in Table 6 correspond closely to those in the Scheme reports. Here, however, the function *new* takes a continuation κ; it should either apply κ to a location currently storing *unallocated*, or ignore κ.

4 ScmQ: Adding Quotations to Scm

This section extends the language defined in Section 3 with literal quotations.

Table 7. ScmQ: Abstract syntax, extending Table 3

$\mathrm{I} \in \mathrm{Ide}$	identifiers
$\mathrm{X} \in \mathrm{Key}$	keywords
$\Delta \in \mathrm{Dat}$	datum
$\mathrm{E} \in \mathrm{Exp}$	expressions

$$\mathrm{Key} \longrightarrow \texttt{eval} \mid \texttt{if} \mid \texttt{lambda} \mid \texttt{quote} \mid \texttt{set!}$$
$$\mathrm{Dat} \longrightarrow \mathrm{K} \mid \mathrm{I} \mid \mathrm{X} \mid \texttt{'}\Delta \mid \texttt{(}\Delta^*\texttt{)} \mid \texttt{(}\Delta^+\,\texttt{.}\,\Delta\texttt{)} \mid \texttt{\#proc}$$
$$\mathrm{Exp} \longrightarrow \cdots \mid \texttt{(quote }\Delta\texttt{)}$$

ScmQ: Abstract Syntax. The denotational semantics of core Scheme expressions in the standards omits the abstract syntax and denotations of literal quotations. Table 7 extends the abstract syntax of Scm expressions given in Table 3 with constructs that correspond directly to a sublanguage of the concrete syntax of ⟨*datum*⟩ and ⟨*quotation*⟩ in the Scheme reports. For simplicity, the abstract syntax of a literal number is identified with the corresponding mathematical integer, and other forms of numbers are omitted, along with literal characters, strings, and vectors.

The final alternative form of Dat is a dummy construct `#proc` that represents procedures (which do not have standard representations in ⟨*datum*⟩). It is not supposed to occur in the abstract syntax of expressions, but is needed for the semantics of `eval` in Section 5.

Note that identifiers $\mathrm{I} \in \mathrm{Ide}$ in literal quotations are literal symbols, not variables. When adding `eval` expressions to ScmQ, keywords $\mathrm{X} \in \mathrm{Key}$ are assumed to be disjoint from identifiers.

Table 8. ScmQ: Domain equations, extending Table 4

$\mathbf{Q} = \mathrm{Ide}_\perp$	symbols
$\mathbf{X} = \mathrm{Key}_\perp$	keyword values
$\epsilon \in \mathbf{E} = \cdots + \mathbf{Q} + \mathbf{X}$	expressed values

ScmQ: Domain Equations. A quotation evaluates to an external representation of a Scheme object. For simplicity, the denotational semantics of ScmQ evaluates a quotation to an element of $\mathbf{E}$, the domain of expressed values defined in Table 4, disregarding its external representation.

Table 9. ScmQ: Semantic functions, extending Table 5

$$\mathcal{D} : \mathrm{Dat} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathcal{D}^* : \mathrm{Dat}^* \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathcal{D}^+ : \mathrm{Dat}^+ \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathcal{E} : \mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$

$$\mathcal{D}\,[\![\mathrm{K}]\!]\kappa = \kappa\,(\mathcal{K}\,[\![\mathrm{K}]\!]) \tag{12}$$
$$\mathcal{D}\,[\![\mathrm{I}]\!]\kappa = \kappa\,(\mathrm{I}\,\mathrm{in}\,\mathbf{E}) \tag{13}$$
$$\mathcal{D}\,[\![\mathrm{X}]\!]\kappa = \kappa\,(\mathrm{X}\,\mathrm{in}\,\mathbf{E}) \tag{14}$$
$$\mathcal{D}\,[\![\texttt{'}\Delta]\!]\kappa = \mathcal{D}\,[\![\Delta]\!]\,\kappa \tag{15}$$
$$\mathcal{D}\,[\![\texttt{(}\Delta^*\texttt{)}]\!]\kappa = \mathcal{D}^*\,[\![\Delta^*]\!]\,\kappa \tag{16}$$
$$\mathcal{D}\,[\![\texttt{(}\Delta^+\,\texttt{.}\,\Delta\texttt{)}]\!]\kappa = \mathcal{D}\,[\![\Delta]\!]\,(\lambda\epsilon.\,\mathcal{D}^+\,[\![\Delta^+]\!]\,\epsilon\,\kappa) \tag{17}$$
$$\mathcal{D}\,[\![\texttt{\#proc}]\!]\kappa = \perp \tag{18}$$

$$\mathcal{D}^*\,[\![\ \ ]\!]\kappa = \kappa\,(\mathit{null}\,\mathrm{in}\,\mathbf{E}) \tag{19}$$
$$\begin{aligned}&\mathcal{D}^*\,[\![\Delta_1\ \ \Delta^*]\!]\kappa = \\ &\quad \mathcal{D}\,[\![\Delta_1]\!]\,(\lambda\epsilon_1.\,\mathcal{D}^*\,[\![\Delta^*]\!]\,(\lambda\epsilon.\,\mathit{cons}\,\langle\epsilon_1,\epsilon\rangle\,\kappa))\end{aligned} \tag{20}$$
$$\mathcal{D}^+\,[\![\Delta_1]\!]\epsilon\kappa = \mathcal{D}\,[\![\Delta_1]\!]\,\epsilon\,(\lambda\epsilon_1.\,\mathit{cons}\,\langle\epsilon_1,\epsilon\rangle\,\kappa) \tag{21}$$
$$\begin{aligned}&\mathcal{D}^+\,[\![\Delta^+\ \ \Delta_1]\!]\epsilon\kappa = \\ &\quad \mathcal{D}\,[\![\Delta_1]\!]\,(\lambda\epsilon_1.\,\mathit{cons}\,\langle\epsilon_1,\epsilon\rangle\,(\lambda\epsilon'.\,\mathcal{D}^+\,[\![\Delta^+]\!]\,\epsilon'\,\kappa))\end{aligned} \tag{22}$$
$$\cdots$$
$$\mathcal{E}\,[\![\texttt{(quote }\Delta\texttt{)}]\!]\rho\kappa = \mathcal{D}\,[\![\Delta]\!]\,\kappa \tag{23}$$

ScmQ: Semantic Functions. In ScmQ, for simplicity, any required storage is allocated whenever quotations are evaluated, and no distinction is made between mutable and immutable objects.

The denotations of ScmQ quotations in Table 9 involve the auxiliary function $\mathit{cons} : \mathbf{E}^* \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$, which is defined in Table 6. When called with arguments ϵ_1 and ϵ_2, *cons* gives a so-called dotted pair. If ϵ_2 is a proper list, so is the result; otherwise the result is an improper list, which finishes with a non-list.

The denotation of $\texttt{(}\Delta^*\texttt{)}$ is straightforward: it applies its continuation to a proper list formed from the elements of Δ^*, finishing with the empty list *null*. As explained in the Scheme reports, `(a b c d e)` is equivalent notation for the proper list `(a . (b . (c . (d . (e . ())))))`.

The denotation of $\texttt{(}\Delta^+\,\texttt{.}\,\Delta\texttt{)}$ is a bit more complicated: it uses *cons* to prefix the elements of Δ^+ to the value given by evaluating Δ, starting from the last element of Δ^+. The denotation of Δ^+ takes an argument ϵ that accumulates the resulting list. This makes `(a b c . d)` equivalent to the improper list `(a . (b . (c . d)))`, as required by the Scheme reports. (It is possible to define the denotation of $\texttt{(}\Delta^+\,\texttt{.}\,\Delta\texttt{)}$ based on decomposing Δ^+ on the left, as with the denotation of Δ^*, but that turns out to be no simpler.)

5 ScmQE: Adding `eval` Expressions

The intended semantics of the expression `(eval E)` is to evaluate E, then evaluate the expression represented by the value of E. The latter expression is not a subexpression of `(eval E)`. As Clinger wrote [5]:

> Certain technical matters should be considered by anyone wishing to add `eval` as an extension. [...] in order to preserve the principle of compositionality its semantics would have to be definable without reference to the semantic function $\mathcal{E}$. One approach would be to define it not in terms of $\mathcal{E}$ but in terms of a "copy" $\mathcal{E}'$. (Think of $\mathcal{E}$ as the compiler's semantics and $\mathcal{E}'$ as an interpreter's semantics.) $\mathcal{E}'$ need not be compositional, so the semantics of `eval` in $\mathcal{E}'$ may without difficulty refer circularly to $\mathcal{E}'$. Alternatively one may imagine an infinite sequence of functions $\mathcal{E}, \mathcal{E}', \mathcal{E}'' \ldots$ as in [22].

As already mentioned in the Introduction, $\mathcal{E}'$ can be made available to $\mathcal{E}$ by providing it as an *extra argument.* Letting $\mathcal{E}'$ be the *first* argument (before the syntactic argument $\mathrm{E} \in \mathrm{Exp}$) requires $\mathcal{E}$ to have the type

$$(\mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}) \to$$
$$(\mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}).$$

To avoid changing the type of $\mathcal{E}$, the semantic function that takes $\mathcal{E}'$ as an argument can be renamed to $\mathcal{F}$. All the previous semantic equations for $\mathcal{E}$ then have to be reformulated:

$$\mathcal{F}\,\mathcal{E}'\,[\![\cdots E \cdots]\!] = \cdots\ \mathcal{F}\,\mathcal{E}'\,[\![E]\!] \cdots$$

Such semantic equations admittedly look unconventional. Crucially, the equation for $\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(eval E)}]\!]$ can apply $\mathcal{E}'$ (rather than $\mathcal{F}\,\mathcal{E}'$) to the expression represented by the value of $\mathcal{F}\,\mathcal{E}'\,[\![\mathrm{E}]\!]$, thereby avoiding a violation of compositionality.

How to define the extra argument $\mathcal{E}'$ of $\mathcal{F}$ is now rather obvious: it is simply the (least) fixed point of $\mathcal{F}$, written *fix* $\mathcal{F}$. The denotation of an expression E is given by $\mathcal{F}\,(\mathit{fix}\,\mathcal{F})\,[\![\mathrm{E}]\!]$. Whenever needed, *fix* $\mathcal{F}$ can be unfolded to $\mathcal{F}\,(\mathit{fix}\,\mathcal{E})$, and thereby applied to any expression.

It could be objected that the above technique has merely *hidden* non-compositionality using an explicit fixed point, instead of avoiding it. But the definition of $\mathcal{F}$ *is* inductive.

The rest of this section extends the abstract syntax of ScmQ to a language ScmQE that includes `(eval E)`, and defines its denotation. This requires mapping the value of E to the abstract syntax of some expression that it represents, which naturally decomposes into a transformation from the value of E to a datum Δ, followed by a transformation of Δ to an expression E. The former transformation may fail to terminate, due to potential circularity of Scheme values; the latter may fail to produce a valid expression, since not all datum elements represent expressions.

Table 10. ScmQE: Abstract syntax, extending Table 7

$\mathrm{E} \in \mathrm{Exp}$	expressions
$\mathrm{Exp} \longrightarrow \cdots \mid$ `(eval E)` $\mid$ `()`	

ScmQE: Abstract Syntax. Table 10 extends the abstract syntax of ScmQ expressions with `(eval E)`. In Scheme, `eval` is a procedure, and takes a second argument that determines the environment to use when evaluating the returned expression. For simplicity, `eval` in ScmQE is a keyword, and the environment for evaluating the expression represented by the value of E is null, and left implicit.

The abstract syntax `()` does *not* correspond to a valid Scheme expression: it is used only as an error element when transforming $\Delta \in \mathrm{Dat}$ to an expression.

ScmQE: Domain Equations. The extension of ScmQ to ScmQE does not involve changes to previous definitions of domains, nor definition of new domains.

ScmQE: Semantic Functions. Table 11 defines the denotations of ScmQE expressions. The definitions of $\mathcal{E}\,[\![E]\!]$ and $\mathcal{E}^*\,[\![E^*]\!]$ (equations 24, 25) override all the previously-given semantic equations for expressions in ScmQ, which now need to be reformulated to use $\mathcal{F}\,\mathcal{E}'$ instead of $\mathcal{E}$, and $\mathcal{F}^*\,\mathcal{E}'$ instead of $\mathcal{E}^*$ (equations 26–32, 35, 36).

The definition of $\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(eval } E\texttt{)}]\!]$ (equation 33) uses the auxiliary functions *datum* and *exp*, which are defined in Table 12 and explained below. The application of $\mathcal{E}'$ to $\mathit{exp}\,[\![\Delta]\!]$ does not undermine the compositionality of $\mathcal{F}$.

When the datum $\Delta \in \mathrm{Dat}$ produced by $\mathit{datum}\,\epsilon$ from the result of evaluating E represents a legal ScmQE expression E′, the result of evaluating `(eval` E`)` is given by $\mathcal{E}'\ \mathrm{E}'\ \mathit{nullenv}\,\kappa$. Otherwise $\mathit{exp}\,[\![\Delta]\!]$ gives `()`, and $\mathcal{E}'\,[\![\texttt{()}]\!] = \mathit{fix}\,\mathcal{F}\,[\![\texttt{()}]\!] = \mathcal{F}\,(\mathit{fix}\,\mathcal{F})\,[\![\texttt{()}]\!] = \bot$. (Recall that for simplicity, the sublanguage of Scheme defined here uses ⊥ to represent erroneous expression evaluation, instead of reporting the cause of the error.)

ScmQE: Auxiliary Functions. Table 12 defines the functions *pre*, *datum*, *exp*, and exp^*. The definition of *datum* uses *pre* to convert `(`Δ`.(`Δ^*`))` to `(`$\Delta\ \Delta^*$`)`. This normalization simplifies the definition of *exp*, because improper lists of the form `(`Δ^+`.`Δ`)` do not represent ScmQE expressions.

$\mathit{datum}\,\epsilon\,\kappa$ applies κ to a datum Δ given by mapping constants in the domain $\mathbf{E}$ to the corresponding elements of the set Dat, and recursively transforming the values stored in pairs. The recursion might not terminate, due to potential circularity created by the procedures *setcar* and *setcar*. The tests in the definition cover all summands of $\mathbf{E}$. As mentioned in Section 4, procedures do not have standard representations in ⟨*datum*⟩, and the dummy element `#proc` of Dat is used for elements of $\mathbf{F}$.

Table 11. ScmQE: Semantic functions, overriding Tables 5, 9

$$\mathcal{E} : \mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathcal{E}^* : \mathrm{Exp} \to \mathbf{U} \to (\mathbf{E}^* \to \mathbf{C}) \to \mathbf{C}$$

$$\mathcal{F} : (\mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}) \to (\mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C})$$
$$\mathcal{F}^* : (\mathrm{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}) \to (\mathrm{Exp}^* \to \mathbf{U} \to (\mathbf{E}^* \to \mathbf{C}) \to \mathbf{C})$$

$$\mathcal{E}[\![E]\!]\rho\kappa = \mathcal{F}\,(\mathit{fix}\,\mathcal{F})\,[\![E]\!]\rho\kappa \tag{24}$$
$$\mathcal{E}^*[\![E^*]\!]\rho\kappa = \mathcal{F}^*\,(\mathit{fix}\,\mathcal{F})\,[\![E^*]\!]\rho\kappa \tag{25}$$

$$\mathcal{F}\,\mathcal{E}'\,[\![K]\!]\rho\kappa = \kappa(\mathcal{K}[\![K]\!]) \tag{26}$$
$$\mathcal{F}\,\mathcal{E}'\,[\![I]\!]\rho\kappa = \mathit{hold}\,(\rho\, I)\,\kappa \tag{27}$$
$$\begin{aligned}&\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(}E\ \ E^*\texttt{)}]\!]\rho\kappa = \\ &\quad \mathcal{F}\,\mathcal{E}'\,[\![E]\!]\rho(\lambda\epsilon. \\ &\qquad \mathcal{F}^*\,\mathcal{E}'\,[\![E^*]\!]\rho(\lambda\epsilon^*.\,(\epsilon \mid \mathbf{F})\,\epsilon^*\kappa))\end{aligned} \tag{28}$$
$$\begin{aligned}&\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(lambda}\ \ I\ \ E\texttt{)}]\!]\rho\kappa = \\ &\quad \kappa\,((\lambda\epsilon^*\kappa'.\,\mathit{list}\,\epsilon^*\,(\lambda\epsilon. \\ &\qquad\qquad \mathit{alloc}\,\epsilon\,(\lambda\alpha. \\ &\qquad\qquad\quad \mathcal{F}\,\mathcal{E}'\,[\![E]\!]\,(\rho[\alpha/I])\,\kappa')) \\ &\qquad)\ \mathrm{in}\ \mathbf{E})\end{aligned} \tag{29}$$
$$\begin{aligned}&\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(if}\ \ E\ \ E_1\ \ E_2\texttt{)}]\!]\rho\kappa = \\ &\quad \mathcal{F}\,\mathcal{E}'\,[\![E]\!]\,\rho\,(\lambda\epsilon. \\ &\qquad \mathit{truish}\,\epsilon \to \mathcal{F}\,\mathcal{E}'\,[\![E_1]\!]\,\rho\,\kappa, \mathcal{F}\,\mathcal{E}'\,[\![E_2]\!]\,\rho\,\kappa)\end{aligned} \tag{30}$$
$$\begin{aligned}&\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(set!}\ \ I\ \ E\texttt{)}]\!]\rho\kappa = \\ &\quad \mathcal{F}\,\mathcal{E}'\,[\![E]\!]\,\rho\,(\lambda\epsilon. \\ &\qquad \mathit{assign}\,(\rho\, I)\,\epsilon\,(\kappa\ \mathit{unspecified}))\end{aligned} \tag{31}$$
$$\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(quote}\ \ \Delta\texttt{)}]\!]\rho\kappa = \mathcal{D}\,[\![\Delta]\!]\,\kappa \tag{32}$$
$$\begin{aligned}&\mathcal{F}\,\mathcal{E}'\,[\![\texttt{(eval}\ \ E\texttt{)}]\!]\rho\kappa = \\ &\quad \mathcal{F}\,\mathcal{E}'\,[\![E]\!]\,\rho\,(\lambda\epsilon. \\ &\qquad \mathit{datum}\,\epsilon\,(\lambda\Delta.\,\mathcal{E}'\,(\mathit{exp}\,[\![\Delta]\!])\ \mathit{nullenv}\,\kappa)\end{aligned} \tag{33}$$
$$\mathcal{F}\,\mathcal{E}'\,[\![\texttt{()}]\!]\rho\kappa = \bot \tag{34}$$

$$\mathcal{F}^*\,\mathcal{E}'\,[\![\ \ ]\!]\rho\kappa = \kappa\,\langle\,\rangle \tag{35}$$
$$\begin{aligned}&\mathcal{F}^*\,\mathcal{E}'\,[\![E\ \ E^*]\!]\rho\kappa = \\ &\quad \mathcal{F}\,\mathcal{E}'\,[\![E]\!]\,\rho\,(\lambda\epsilon. \\ &\qquad \mathcal{F}^*\,\mathcal{E}'\,[\![E^*]\!]\,\rho\,(\lambda\epsilon^*.\,\kappa\,(\langle\epsilon\rangle \,\S\, \epsilon^*)))\end{aligned} \tag{36}$$

The auxiliary functions *exp* and *exp** are defined inductively on abstract syntax trees, ensuring termination of their mutual recursion.

Note that in the left side of the case for $\mathit{exp}\,[\![\texttt{(quote}\ \ \Delta\texttt{)}]\!]$, the symbol quote is an element of Dat and (quote Δ) is

Table 12. ScmQE: Auxiliary functions, extending Table 6

$$\mathit{pre} : \mathrm{Dat} \to \mathrm{Dat}$$
$$\mathit{pre}\,[\![\texttt{(}\Delta\,\texttt{.}\,\texttt{(}\Delta^*\texttt{))}]\!] = [\![\texttt{(}\Delta\ \ \Delta^*\texttt{)}]\!]$$
$$\mathit{pre}\,[\![\Delta]\!] = [\![\Delta]\!]\ \text{otherwise}$$

$$\mathit{datum} : \mathbf{E} \to (\mathrm{Dat} \to \mathbf{C}) \to \mathbf{C}$$
$$\begin{aligned}\mathit{datum}\,\epsilon\kappa = {}& \epsilon \in \mathbf{T} \longrightarrow (\epsilon \mid \mathbf{T} \longrightarrow \kappa\,[\![\texttt{\#t}]\!], \kappa\,[\![\texttt{\#f}]\!]), \\ & \epsilon \in \mathbf{R} \longrightarrow (\lambda\,\mathrm{Z}.\,\kappa\,[\![\mathrm{Z}]\!])\,(\epsilon \mid \mathbf{R}), \\ & \epsilon \in \mathbf{P} \longrightarrow \\ &\quad \mathit{car}\,\langle\epsilon\rangle\,(\lambda\epsilon_1.\,\mathit{cdr}\,\langle\epsilon\rangle\,(\lambda\epsilon_2. \\ &\qquad \mathit{datum}\,\epsilon_1\,(\lambda\Delta_1.\,\mathit{datum}\,\epsilon_2\,(\lambda\Delta_2. \\ &\qquad\quad \kappa\,\mathit{pre}\,[\![\texttt{(}\Delta_1\,\texttt{.}\,\Delta_2\texttt{)}]\!])))), \\ & \epsilon \in \mathbf{M} \longrightarrow (\epsilon \mid \mathbf{M} = \mathit{null} \longrightarrow \kappa\,[\![\texttt{()}]\!], \bot), \\ & \epsilon \in \mathbf{F} \longrightarrow \kappa\,[\![\texttt{\#proc}]\!]), \\ & \epsilon \in \mathbf{Q} \longrightarrow (\lambda\,\mathrm{I}.\,\kappa\,[\![\mathrm{I}]\!])\,(\epsilon \mid \mathbf{T}), \\ & \epsilon \in \mathbf{X} \longrightarrow (\lambda\,\mathrm{X}.\,\kappa\,[\![\mathrm{X}]\!])\,(\epsilon \mid \mathbf{X}), \bot\end{aligned}$$

$$\mathit{exp} : \mathrm{Dat} \to \mathrm{Exp}$$
$$\mathit{exp}^* : \mathrm{Dat}^* \to \mathrm{Exp}^*$$

$$\begin{aligned}&\mathit{exp}\,[\![\mathrm{K}]\!] = [\![\mathrm{K}]\!] \\ &\mathit{exp}\,[\![\mathrm{I}]\!] = [\![\mathrm{I}]\!] \\ &\mathit{exp}\,[\![\texttt{'}\Delta]\!] = [\![\texttt{(quote}\,\Delta\texttt{)}]\!] \\ &\mathit{exp}\,[\![\texttt{(quote}\ \ \Delta\texttt{)}]\!] = [\![\texttt{(quote}\ \ \Delta\texttt{)}]\!] \\ &\mathit{exp}\,[\![\texttt{(lambda}\ \mathrm{I}\ \ \Delta\texttt{)}]\!] = [\![\texttt{(lambda}\ \mathrm{I}\ \ \mathit{exp}\,[\![\Delta]\!]\texttt{)}]\!] \\ &\mathit{exp}\,[\![\texttt{(if}\ \ \Delta\ \ \Delta_1\ \ \Delta_2\texttt{)}]\!] = \\ &\qquad [\![\texttt{(if}\ \ \mathit{exp}\,[\![\Delta]\!]\ \ \mathit{exp}\,[\![\Delta_1]\!]\ \ \mathit{exp}\,[\![\Delta_2]\!]\texttt{)}]\!] \\ &\mathit{exp}\,[\![\texttt{(set!}\ \mathrm{I}\ \ \Delta\texttt{)}]\!] = [\![\texttt{(set!}\ \mathrm{I}\ \ \mathit{exp}\,[\![\Delta]\!]\texttt{)}]\!] \\ &\mathit{exp}\,[\![\texttt{(}\mathrm{I}\ \ \Delta^*\texttt{)}]\!] = [\![\texttt{(}\mathrm{I}\ \ \mathit{exp}^*\,[\![\Delta^*]\!]\texttt{)}]\!] \\ &\mathit{exp}\,[\![\Delta]\!] = [\![\texttt{()}]\!]\ \text{otherwise}\end{aligned}$$

$$\begin{aligned}&\mathit{exp}^*\,[\![\ \ ]\!] = [\![\ \ ]\!] \\ &\mathit{exp}^*\,[\![\Delta\ \ \Delta^*]\!] = [\![\mathit{exp}\,[\![\Delta]\!]\ \ \mathit{exps}\,[\![\Delta^*]\!]]\!]\end{aligned}$$

a proper list of the form (Δ*), whereas in the right side, (quote Δ) is an element of Exp.[3]

Lists (Δ Δ*) where Δ is a keyword symbol X : Key return valid primitive expressions or the illegal expression (); other proper lists where Δ is an identifier I : Ide return procedure call expressions.

That concludes the definition of the denotational semantics of ScmQE.

[3]The potential confusion here is due to the fact that expressions are also data in Scheme.

6 Lightweight Formalization in Agda

The compositionality of the semantics of ScmQE has been tested using a lightweight formalization of denotational semantics in Agda. An archive including the Agda source code of the formalization is available as supplemental material [17]. This section explains the relationship of the formalization to the definitions in the paper.

Abstract Syntax. The Agda formalization of abstract syntax is not so direct, but systematic. Syntactic sorts are defined as inductive datatypes. Meta-variables are declared separately from the types over which they range.

Agda supports mixfix notation for syntactic constructors, but not the use of ordinary parentheses and dots as tokens in names; the formalization of ScmQE replaces them by Unicode characters for so-called banana-brackets and raised dots.

Domain Equations. Domains are formalized as Agda types using the same notation as listed in Table 1, except that Agda does not support writing $S_\perp$ with a subscript $\perp$ (the formalization uses $S + \perp$).

In Agda, however, the type-checker does not terminate when type constants are recursively defined. The lightweight Agda formalization of recursively defined domains avoids nontermination by leaving one or more types undefined, then postulating functions between these types and their intended structure.

For ScmQE, the type corresponding to the sum domain **E** is related to the types for its summands by the injection and projection functions; all the other domains are defined by type equations corresponding directly to the domain equations given in the paper.

Semantic Functions. The Agda formalization of semantic function declarations and definitions is quite direct. Agda notation for lambda abstractions differs from the conventional notation used here by requiring an arrow instead of a dot between the bound variable(s) and the body; Agda also requires names to be separated (by layout or parentheses) from adjacent names. Otherwise, the notation for values listed in Table 2 can be used without significant changes.

Agda accepts recursive semantic function definitions only when it can mechanically prove them terminating. The simplest case of this is primitive recursion, which corresponds directly to compositionality when defining semantic functions. However, Agda also accepts functions defined by structural recursion, which allows recursive application to arbitrarily deeper subphrases. So the acceptance of the formalization of the semantic functions for ScmQE does not check that they are strictly compositional.

Note that the argument $\mathcal{E}'$ of the semantic function $\mathcal{F}$ for ScmQE is defined by an explicit fixed point, and thereby circumvents Agda's termination checks. Using $\mathcal{F}\,\mathcal{E}'$ for recursive applications checks that those applications are compositional; using just $\mathcal{E}'$ would lose those checks.

The semantic equations used to define semantic functions are formalized directly as equations in Agda. The type-checker requires the patterns of constructors in the syntactic argument to be not only well-typed but also exhaustive. Agda warns about overlapping patterns, and about equations that are shadowed by preceding equations.

Auxiliary Functions. Agda allows auxiliary functions to be defined recursively, but requires the recursion to be evidently terminating. This excludes functions that recurse on components of computed values; their definitions require explicit use of the fixed-point operator *fix*. For ScmQE, this affects only the definitions of the functions *list* and *datum*.

Caveats. The lightweight Agda formalization of denotational semantics currently *abuses* the Agda language by using arbitrary Agda types as domains, and postulating declarations that are inconsistent with the underlying logic of Agda. For example, it postulates that each domain has an element named $\perp$, but Agda allows declaration of an empty type. Similarly, the formalization treats ordinary Agda function types as domains, and postulates a function from $(D \to D) \to D$, but there is no such function when D is the empty type.[4]

Nevertheless, the impact of such inconsistent postulates when using the Agda type-checker on the semantics of ScmQE appears to be insignificant. Type errors reported during the development of the formalization always indicated genuine mistakes. Explicit injections and projections between sum domains and their summands can be tedious to write in denotational semantics, and are sometimes elided; currently, Agda does not support implicit coercions between types, and reported missing injections and projections as errors.

The Agda source code for an initial version of the formalization was produced by systematic editing of a plain-text preview of the PDF of R^{5}RS. The preview preserved all the Unicode characters from the PDF. However, the LaTeX code for the denotational definitions in the present paper was subsequently produced manually, which may have led to (hopefully minor) differences and mistakes. It should be possible to detect and correct these by re-developing the formalization from a preview of the PDF of the present paper.

Finally, it should be stressed that although checking a lightweight Agda formalization can ensure wellformedness and reveal non-compositionality, this does not directly verify that denotations of programs compute the *intended* values. However, it is possible to formulate equality of denotations as Agda types; checking that some term has such a type then verifies that the equality holds. For ScmQE, the required equivalences for list notation mentioned in Section 4 have been verified automatically by Agda; verifying equivalences involving explicit fixed points is currently work in progress.

[4]All functions in Agda are total.

7 Conclusion

This paper presented a denotational semantics for a simple form of `eval` expressions. The semantics is based on an adaptation of a suggestion made by Clinger [5] more than 40 years ago.

A language that includes `eval` was defined incrementally, starting from a particularly basic sublanguage Scm of the core Scheme expressions whose denotational semantics is defined in the Scheme reports. ScmQ extended Scm with literal quotations, then ScmQE added `eval`.

The compositionality of the semantics of ScmQE has been tested using a lightweight formalization of denotational semantics in Agda. An archive of the Agda source code is available as supplemental material [17] accompanying this paper. Using Agda to verify that the semantics satisfies expected program equivalences is work in progress.

It should be straightforward to add all the remaining core expressions covered by the denotational semantics in the Scheme reports to ScmQE and its formalization. The use of continuation-passing style for storage allocation functions, as illustrated here in the definition of Scm, would significantly simplify the original definitions. The current version of the formalization of ScmQE includes also tentative abstract syntax and semantics for Scheme programs and definitions.

The author has recently completed a lightweight Agda formalization of the denotational semantics defined in R^{5}RS, and presented it in a paper [16] in the proceedings of SCHEME '25. The paper discusses some wellformedness issues with the denotational definitions in the Scheme reports, and suggests how they could be addressed. The formalization of the R^{5}RS semantics uses the same notation as the formalization of the ScmQE semantics, and could easily be extended to include the semantics of literal quotations, programs, and definitions; extending it to include `eval` expressions would require reformulation of the definition of the semantic function $\mathcal{E}$ for expressions as illustrated here for ScmQE in Section 5.

In principle, extending the denotational semantics defined in R^{7}RS with literal quotations and `eval` expressions is as straightforward as extending Scm to ScmQ and ScmQE. A definition of the Scheme *procedure* `eval` would involve adding auxiliary functions corresponding to evaluation environments, and mutual recursion between the definitions of the semantic functions and the auxiliary functions.

However, the denotational semantics in the Scheme reports has remained remarkably stable since its introduction in R^{3}RS, almost four decades ago. It is unclear to the present author whether there would be any enthusiasm or support in the Scheme community for revising it at all – especially independently of the development of a future version of the Scheme language and its standard. The unorthodox appearance of the semantic equations when the semantic function for expressions takes $\mathcal{E}'$ as an extra argument might also be a significant disincentive to include a definition of `eval`.

Acknowledgments

I am extremely grateful to Olivier Danvy for crucial help with completing this paper. When he saw the abstract of the accepted submission on the track website, he very kindly emailed me to say that he would like a preview of the Agda formalization. I sent it to him together with an incomplete draft of the paper, less than a week before the submission deadline; within 24 hours he responded with many helpful queries and constructive suggestions. In particular, he pointed out a significant discrepancy between my original definition of the auxiliary function *datum* (Table 12) and how Scheme implementations transform values to expressions when calling `eval`; I hope that I have now managed to fix it.

I am also very grateful to the anonymous reviewers for their perceptive expert comments on the submitted paper, and for their constructive suggestions for improvement. In particular, the original definition of $\mathcal{D}\llbracket\Delta\rrbracket$ (Table 9) used the same semantic function for sequences Δ^* and Δ^+; one of the reviewers pointed out that the denotations failed to satisfy an equivalence required by the Scheme standards. The Agda formalization of the current definition includes a proof of that equivalence. Any remaining issues with the text and the formal definitions in the paper are of course entirely my own responsibility.

Finally, I would like to thank the organizers of the OlivierFest for their initiative, and for the invitation to submit a paper to the Festschrift.

Data-Availability Statement

The Agda code in the accompanying artifact [17] is a lightweight formalization of the denotational semantics of the language ScmQE presented in Section 5. The artifact includes a PDF of a highlighted listing of the Agda code, generated using Agda. The relationship of the formalization to the definitions in the paper is explained in Section 6.

After downloading the artifact, the type-correctness of the formalization can be checked by loading `ScmQE/All.lagda` in Agda. The current soundness tests can be verified by loading `ScmQE/Soundness-Tests.lagda`.

The artifact is an archive of version 1.0.2 of the public repository pdmosses/olivierfest-agda. The development of further soundness tests is work in progress, to appear in future updates of the same repository.

References

[1] Casper Bach Poulsen and Peter D. Mosses. 2013. Generating specialized interpreters for modular structural operational semantics. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8901)*, Gopal Gupta and Ricardo Peña (Eds.). Springer, Berlin, Heidelberg, 220–236. doi:10.1007/978-3-319-14125-1_13

[2] Casper Bach Poulsen and Peter D. Mosses. 2014. Deriving Pretty-Big-Step Semantics from Small-Step Semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014,*

Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410), Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 270–289. doi:10.1007/978-3-642-54833-8_15

[3] Małgorzata Biernacka and Olivier Danvy. 2009. Towards compatible and interderivable semantic specifications for the Scheme programming language, part II: Reduction semantics and abstract machines. In *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, Jens Palsberg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–206. doi:10.1007/978-3-642-04164-8_10

[4] Declan Butler. 2004. Frenchman is most thanked computer scientist. *Nature* 432 (2004), 790. doi:10.1038/432790b

[5] William Clinger. 1984. The Scheme 311 compiler: An exercise in denotational semantics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 356–364. doi:10.1145/800055.802052

[6] William Clinger and Jonathan Rees. 1991. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers* IV, 3 (July–September 1991), 1–55. https://standards.scheme.org/official/r4rs.pdf

[7] William D. Clinger. 1998. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 174–185. doi:10.1145/277650.277719

[8] William Cook and Jens Palsberg. 1989. A denotational semantics of inheritance and its correctness. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (New Orleans, Louisiana, USA) *(OOPSLA '89)*. Association for Computing Machinery, New York, NY, USA, 433–443. doi:10.1145/74877.74922

[9] Olivier Danvy. 2009. Towards compatible and interderivable semantic specifications for the Scheme programming language, part I: Denotational semantics, natural semantics, and abstract machines. In *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, Jens Palsberg (Ed.). Springer, Berlin, Heidelberg, 162–185. doi:10.1007/978-3-642-04164-8_9

[10] Olivier Danvy and Karoline Malmkjær. 1988. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, USA) *(LFP '88)*. Association for Computing Machinery, New York, NY, USA, 327–341. doi:10.1145/62678.62725

[11] Olivier Danvy and Morten Rhiger. 1998. *Compiling actions by partial evaluation, revisited.* Research Report BRICS RS-98-13. Department of Computer Science, Aarhus University, Aarhus, Denmark.

[12] C. Lee Giles and Isaac G. Councill. 2004. Who gets acknowledged: Measuring scientific contributions through automatic acknowledgment indexing. *Proc. Natl. Acad. Sci. U.S.A.* 101, 51 (2004), 17599–17604. doi:10.1073/pnas.0407743101

[13] IEEE1178 1991. IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.

[14] Richard Kelsey, William Clinger, and Jonathan Rees. 1998. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (1998), 7–105. https://standards.scheme.org/official/r5rs.pdf

[15] Jacob Matthews and Robert Bruce Findler. 2008. An operational semantics for Scheme. *J. Funct. Program.* 18, 1 (Jan. 2008), 47–86. doi:10.1017/S0956796807006478

[16] Peter D. Mosses. 2025. Checking a denotational semantics of Scheme in Agda. In *Proceedings of the 26th ACM SIGPLAN International Workshop on Scheme and Functional Programming (Scheme 2025)* (Singapore, Singapore). ACM, New York, NY, USA. doi:10.1145/3759537.3762694

[17] Peter D. Mosses. 2025. Lightweight Agda formalization of denotational semantics in article 'A compositional semantics for `eval` in Scheme'. ACM. doi:10.1145/3747409

[18] Steven S. Muchnick and Uwe F. Pleban. 1980. A semantic comparison of LISP and SCHEME. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming* (Stanford University, California, USA) *(LFP '80)*. Association for Computing Machinery, New York, NY, USA, 56–64. doi:10.1145/800087.802790

[19] Catherine Recanati and Alain Deutsch. 1987. *Towards a denotational semantics for a reflective Scheme: An implementation of the towerless model.* Technical report 87/36. Esprit project No 1228 (Chameleon). https://hal.science/hal-00165673

[20] Jonathan Rees and William Clinger. 1986. Revised^3 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices* 21, 12 (December 1986), 37–79. https://standards.scheme.org/official/r3rs.pdf

[21] Alex Shinn, John Cowan, and Arthur A. Gleckler. 2021. Revised^7 Report on the Algorithmic Language Scheme. https://standards.scheme.org/official/r7rs.pdf

[22] Brian Cantwell Smith. 1984. Reflection and semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 23–35. doi:10.1145/800017.800513

[23] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. 2010. *Revised^6 Report on the Algorithmic Language Scheme.* Cambridge University Press. https://standards.scheme.org/official/r6rs.pdf

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

[25] Gerald Jay Sussman and Guy Lewis Steele Jr. 1975. *Scheme: An interpreter for extended lambda calculus.* Technical Report MIT Artificial Intelligence Memo 349. MIT. https://standards.scheme.org/official/r0rs.pdf

[26] Gerald J. Sussman and Guy L. Steele Jr. 1998. The First Report on Scheme Revisited. *Higher-Order and Symbolic Computation* 11, 4 (1998), 399–404.

[27] Robert D. Tennent. 1976. The denotational semantics of programming languages. *Commun. ACM* 19, 8 (Aug. 1976), 437–453. doi:10.1145/360303.360308

[28] Mitchell Wand and Daniel P. Friedman. 1986. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 298–307. doi:10.1145/319838.319871

[29] Mitchell Wand and Daniel P. Friedman. 1988. The mystery of the tower revealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation* 1 (1988), 11–37. doi:10.1007/BF01806174

Received 2025-06-10; accepted 2025-07-31