Checking a Denotational Semantics of Scheme in Agda

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

Abstract

The authoritative standards for the algorithmic language Scheme are the Scheme reports. Most of the revised reports include a denotational semantics for primitive Scheme expressions and selected procedures.

This paper first traces the history of the semantic definition, and summarizes its form and content. It then presents a shallow embedding of denotational semantics into the functional programming language Agda. The embedding is illustrated by showing how fragments of the denotational semantics given in the fifth revised Scheme report (R^5RS) are embedded into Agda.

Type-checking the Agda embedding of a semantics indirectly tests its wellformedness. Agda reported several issues with the embedding of the complete denotational semantics from R^5RS. The paper suggests changes to the semantics that would address the reported issues, as well as further changes that could improve the conciseness and perspicuity of the definitions.

This paper is dedicated to the memory of
Christopher Strachey (1916–1975)

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

Keywords: Denotational semantics, Scheme, wellformedness, Agda, shallow embedding

ACM Reference Format:
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 '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759537.3762694

Scheme '25, Singapore, Singapore

ACM ISBN 979-8-4007-2162-5/25/10
https://doi.org/10.1145/3759537.3762694

1 Introduction

The initial report on Scheme language was published in December 1975. Congratulations to the Scheme developers and users on the 50th anniversary!

1.1 The Scheme Reports

Quoting from the Scheme Standards web page [28]:

> The Scheme programming language was introduced in the 1975 paper, *Scheme: An Interpreter for Extended Lambda Calculus*. Since then it has been improved and extended through many rounds of standardization. The authoritative standards are the Scheme reports. Their names follow the convention *Revised^n Report on the Algorithmic Language Scheme*, abbreviated R^nRS.

In the abstract of the initial report, Sussman and Steele indicate how significantly Scheme extends the lambda calculus [38]:

> Inspired by ACTORS [...], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [...], but extended for side effects, multiprocessing, and process synchronization.

The reference manual in the first section is remarkably concise: just five pages! The programming examples in the second section raise issues of semantics, which the authors seek to clarify with reference to the lambda calculus and the fixed point operator in the third section.

The first Revised Report on Scheme (1978, [9]) has the subtitle "a dialect of LISP", and gives a complete "user manual" for the language (22 pages) accompanied by explanatory notes (10 pages). The second, R^2RS (1985, [5]) is about double the length, and promises that "formal definitions of the syntax and semantics of Scheme will be included in a separate report". Those definitions were provided in R^3RS (1986, [24]), which defines the lexical and context-free syntax of the complete language in an extended BNF, and "provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures". The denotational semantics was repeated with only minor changes in R^4RS (1991, [6]), the IEEE Standard (1991, [27]), R^5RS (1998, [10]), and R^7RS (2013, [31]).

R^{6}RS (2007, [34]) provides an operational semantics instead of the denotational semantics, and covers much more of the language. It is based on a paper by Matthews and Findler [11] where they state that "denotational semantics has fallen out of favor among programming language researchers", and "requires much more mathematical sophistication than operational semantics, making it less appropriate for a standard intended for use by working programmers".

However, lambda-abstractions have now become quite familiar to programmers in various mainstream languages; similarly for recursive type definitions, and defining functions by pattern-matching. And definitions of domains and semantic functions can easily be understood *without* knowing anything at all about their mathematical foundations (including the structure of domains and the notion of continuity). When formulating a denotational semantics, it can simply be *assumed* that domain equations always have well-defined solutions, and that when functions on a domain are defined in lambda-notation they always have (least) fixed points (as in so-called synthetic domain theory [25, 30, 32], which is based on constructive logic). See also Queinnec's motivation for denotational semantics [22, §5].

The R^{6}RS Rationale stresses the need for the formal semantics to specify precisely that the order of expression evaluation in procedure calls is implementation dependent [33, §12]:

> The denotational semantics in R^{5}RS has several problems, most seriously its incomplete treatment of the unspecific evaluation order of applications: the denotational semantics suggests that a single unspecified order is used. Modelling nondeterminism is generally difficult with denotational semantics, and an operational semantics allows specifying the unspecified evaluation order precisely.

Despite that issue, R^{7}RS reverted to giving a denotational semantics of core Scheme constructs, without changing the treatment of unspecified evaluation order from R^{5}RS.

1.2 Checking the Formal Semantics of Scheme

The operational semantics given in R^{6}RS is executable. As stated there in the introduction to the semantics [34, App. A]:

> To help understand the semantics and how it behaves, we have implemented it in PLT Redex. The implementation is available at the report's website.[1] All of the reduction rules and the meta-functions shown in the figures in this semantics were generated automatically from the source code.

[1] https://www.r6rs.org/

In fact the denotational semantics given in R^{3}RS was produced in much the same way. As stated in the introduction to the semantics [24, §7.2]:

> The semantics in this section was translated by machine from an executable version of the semantics written in Scheme itself.

Presumably its authors checked that the executable version computed the expected results for a sufficiently diverse suite of test programs. Van Straaten later implemented the semantics from R^{5}RS in Scheme [36], and checked its consistency with the original definition by translating the Scheme code back to denotational syntax using Queinnec's L2T tool [23]. However, reformulating a denotational semantics in a dynamically-typed language such as Scheme does not check that the semantic definition is actually *wellformed*.

1.3 Wellformedness of Denotational Definitions

A difficulty with checking the wellformedness of denotational definitions is the current lack of a precise description of the meta-notation used in the literature. In the 1970s, the author proposed a meta-notation MSL for denotational semantics [12]. MSL has an unambiguous grammar and a formal (meta-circular) definition of its semantics; it was a precursor of the meta-notation DSL of SIS (Semantics Implementation System) [14], which was used in courses on denotational semantics in several universities, although the implementation of SIS was too inefficient for execution of the denotational semantics of larger languages [2]. A DSL definition of a mini-language [8] was included in the successful bid for the contract to develop the Ada language. However, a more conventional mathematical meta-notation for denotational semantics is generally preferred in the scientific literature, including the Scheme reports.

Outline. This paper makes the following contributions:

- Section 2 traces the origin of the denotational semantics in R^{3}RS, and summarizes its structure and content.
- Section 3 presents a shallow embedding of denotational semantics into Agda, illustrated with fragments from R^{5}RS. Familiarity with the Agda language [41] would be helpful for this section.
- Section 4 discusses several issues with the wellformedness of the denotational semantics in R^{5}RS that were reported by Agda when checking its embedding, as well as some further issues noticed by the author.
- Section 5 suggests changes to improve the wellformedness, conciseness, and perspicuity of the denotational semantics given in the Scheme reports.
- Section 6 explains the relationship between the shallow Agda embedding of denotational semantics presented in Section 3 and previous work on defining denotational semantics in Agda.

Section 7 concludes, and mentions potential future work.

2 Denotational Semantics of Core Scheme

R^{3}RS (1986) defines the lexical and context-free syntax of the complete Scheme language in an extended BNF, and "provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures". The definitions given in R^{4}RS, the IEEE Standard, R^{5}RS, and R^{7}RS are based directly on the R^{3}RS version.

Section 2.1 describes the origin of the denotational semantics given in R^{3}RS. Section 2.2 summarizes the concepts and notation. Section 2.3 gives and overview of the contents and structure of the definition.

2.1 Origin

Remarks by Sussman and Steele in their original presentation of Scheme in 1975 already foreshadowed the development of a denotational semantics for Scheme:

> Can we make these ideas more precise? One traditional approach is to model the computation with lambda calculus. [...] The "usual" lambda calculus construct for defining recursive functions is a rather obscure object called the "fixed point" operator.

In the first Revised Report, they also considered the impact of evaluation order in lambda calculus:

> {Normal Order Loses}
> Our definition of BLOCK exploits the fact that SCHEME is an applicative-order (call-by-value) language in order to enforce sequencing. Sussman has proved that one cannot do a similar thing in a normal-order (call-by-name) language:
> Theorem: Normal order, as such, is incapable of enforcing sequencing (whereas applicative order is) in the form of the BLOCK construct.
> (Informal) proof: [...]

Most of the details of the denotational semantics of primitive expressions and selected procedures in the Scheme reports originate in the work of Muchnick and Pleban [20]. (The treatment of environments in Pleban's dissertation [21] is somewhat different.) In personal correspondence, Clinger recalled the development of the formal semantics in R^{3}RS:

> Jonathan Rees wrote the executable version. [It] was based upon a draft of the denotational semantics I had written, which was itself based upon the Muchnick and Pleban semantics. That semantics appears within my paper with the title "The scheme 311 compiler an exercise in denotational semantics", which was published in the proceedings of the 1984 Lisp Conference and is cited by the R^{3}RS.
> [...] Jonathan Rees then wrote a small Scheme program that translated his Scheme code into the LaTeX that went into print.

Regarding the remark cited in Section 1.2, Clinger added:

> [...] the purpose of that remark in R^{3}RS §7.2 was to emphasize that the denotational semantics had been tested by creating an executable version of it, and that errors in the printed version should be unlikely because the typesetting had been done by machine translation from the executable semantics.

A note in the current LaTeX sources for R^{7}RS explains that for subsequent Scheme reports, the semantics was modified without going back to the executable version, so the remark was removed.

§7.2 of R^{4}RS differs from the same section of R^{3}RS mainly by distinguishing between mutable and immutable objects. Clinger spotted one other difference:

> In the R^{3}RS semantics, the empty list (which the semantics refers to as null) counts as false, mainly because Texas Instruments insisted upon that for compatibility with Common Lisp. In the R^{4}RS semantics, the empty list does not count as false. This difference shows up in the definition of the auxiliary function truish.

The denotational semantics included as an annex in the IEEE standard for Scheme [27] appears to be based on the definitions in R^{4}RS, but suffers from significant formatting issues. The version in R^{5}RS differs from R^{4}RS by adding the semantics of the procedures `values` and `call-with-values`. R^{7}RS added a domain `P` of dynamic points, and changed the denotations of expressions to be functions of `P`, to support the definition of the semantics of the `dynamic-wind` procedure; this required minor adjustments to most semantic equations and auxiliary function definitions.

2.2 Concepts and Notation

The standards refer to Stoy's 1977 textbook on denotational semantics [35] for a description of concepts and notation used in the definitions. They provide the following concise summary of the notation used for sequences, McCarthy conditional, environment construction, and injections and projections between sums of domains and their summands.

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth 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 in `D`	injection of x into domain `D`
$x \mid$ `D`	projection of x to domain `D`

Tennent's 1976 article in Comm. ACM [39] is an alternative reference for the concepts and notation used in the denotational semantics of Scheme.

2.3 Contents and Structure

The formal semantics in the Scheme standards defines the abstract syntax and denotations of identifiers, procedure calls, lambda-expressions, if-expressions, and assignment expressions; literals are omitted because "an accurate definition [...] would complicate the semantics without being very interesting".

It does not give an abstract syntax for programs and definitions, but explains that the meaning of a program P "in which all variables are defined before being referenced or assigned" is the denotation of an expression formed from I*, P′ and ⟨undefined⟩ "where I* is the sequence of variables defined in P, P′ is the sequence of expressions obtained by replacing every definition in P by an assignment, ⟨undefined⟩ is an expression that evaluates to *undefined*".

The formal semantics also defines auxiliary functions corresponding to the code of selected built-in procedures.

Abstract syntax. The formal semantics starts with a concise context-free grammar defining the abstract syntax of the core expressions; commands are defined to be the same as expressions.

Domain equations. The abstract syntax is followed by domain equations that define named domains recursively in terms of (continuous) function domains, (cartesian) product domains, (separated) sum domains, (finite) sequence domains, and flat domains. Some domains are declared but left undefined: locations, natural numbers, symbols, characters, numbers, answers, and errors.

Semantic functions. The definition of the semantic functions starts by declaring the domains of denotations for constants, expressions, expression sequences, and command sequences. The denotations of constants are omitted; the denotations of the other sorts of constructs are defined by semantic equations, mostly in continuation-passing style.

Auxiliary functions. The formal semantics concludes with the definitions of auxiliary functions. Some of these functions are used in the semantic equations; the rest of them correspond to built-in procedures. Several functions are considered implementation-dependent, and therefore not defined: the continuation *wrong* for error reports, the storage allocator *new*, and the arbitrary permutations *permute* and its inverse *unpermute*.

3 Embedding in Agda

This section presents a shallow embedding of denotational semantics into Agda. The embedding is illustrated by showing fragments of the denotational semantics in R^{5}RS and the corresponding Agda code. A listing of the embedding of the complete denotational semantics in R^{5}RS is available as supplemental material [19].

Readers are assumed to be familiar with the main concepts of denotational semantics, and with the meta-notation summarized in Section 2.2. Familiarity with the Agda language [40, 41] would also be helpful.

In denotational semantics, the type of a named function is usually declared before giving the definition of the function, and function definitions are implicitly recursive, but functions can be used before they have been declared; similarly for domain equations. R^{5}RS exploits this flexibility to present the definitions of the semantic functions before those of the auxiliary functions used in them.

A denotational definition is usually divided into the same four sections as in R^{5}RS. Its embedding into Agda naturally defines each section as a separate module, specifying which other modules it imports.

3.1 Notation

An additional module declares an Agda version of the conventional notation for domain constructors and their associated functions, including the operators summarized in Section 2.2. For brevity, the declarations and definitions presented below omit notation whose use is not illustrated in this paper; they have also been reformulated to ease their presentation.

```
Domain = Set
variable D E : Domain
```

Set is the fundamental universe in Agda. Its elements are types, including function types and empty types. For a shallow embedding of denotational semantics in Agda such that functions on domains can be defined in λ-notation, domains also need to be types.

Domains are always non-empty, and in principle, the universe Domain of domains should be a *new* universe, disjoint from Set. However, Agda does not support declaration of new universes: adding one would involve extending Agda.

So here, Domain is defined to be equal to Set, and the variables D and E range over arbitrary types. The main disadvantage is that applications of domain constructors to ordinary types in Set are not reported as errors by the typechecker.

```
postulate ⊥ : D
postulate fix : (D → D) → D
```

In Agda, a postulate simply declares a name to have a specified type, without defining its value. The postulate that all domains D have a distinguished element named ⊥ ensures that all domains are non-empty – but the lack of distinction between domains and ordinary types implies also that empty types are non-empty, so the postulate is obviously unsound. The postulated function type for fix is also unsound: when D is an empty type, fix must map the empty function on D to some element of D.

As stated in the Agda language reference [40], introducing postulates is in general not recommended: a preferable

way to work with assumptions is to declare them as module parameters. When using Agda as a proof checker, uninstantiated module parameters become explicit premises of proved propositions; in contrast, dependence of proved propositions on postulates is implicit, which can be misleading.

Here, however, Agda is primarily used as a *type* checker, and declaring unsound assumptions as postulates does not undermine the soundness of type correctness. Moreover, using module parameters instead of postulates would have significant disadvantages: module parameters cannot be used in rewrite rules; it is tedious to supply a large number of parameters when importing a module (as illustrated by the Agda code accompanying [15]); and module parameters may cause major performance problems in specific cases.

So in the shallow embedding of denotational semantics into Agda, all assumptions are formulated as postulates.

```
postulate _+⊥ : Set → Domain
postulate η : (S : Set) → S +⊥
data Bool : Set where true false : Bool
Bool⊥ = Bool +⊥
data Nat : Set where zero : Nat; suc : Nat → Nat
Nat⊥ = Nat +⊥
postulate _==⊥_   : Nat⊥ → Nat → Bool⊥
postulate _⟶_,_ : Bool⊥ → D → D → D
```

Underscores in function names indicate argument positions. The domain `S +⊥` consists of elements `η s` for `s : S` and the element `⊥`. The ternary McCarthy conditional operator `b ⟶ d , d′` uses a longer arrow than function types.

```
postulate _+_ : Domain → Domain → Domain
postulate _×_ : Domain → Domain → Domain
postulate _,_ : D → E → D × E
postulate _↓²1 : D × E → D
postulate _↓²2 : D × E → E
_^_           : Domain → Nat → Domain
```

The notation `D + E` for sum domains and `D × E` for product domains is as usual. For any `n : Nat`, `D ^ n` consists of `n`-tuples $d_1, \ldots, d_n$ of elements of `D`.

```
variable n : Nat
postulate _*    : Domain → Domain
postulate ⟨⟩    : D *
postulate ⟨_⟩   : (D ^ suc n) → D *
postulate #     : D * → Nat⊥
postulate _§_ : D * → D * → D *
postulate _↓_ : D * → Nat → D
postulate _†_ : D * → Nat → D *
```

The above notation for functions on domains `D *` of (finite, possibly-empty) sequences is as in the Scheme reports. Note that angle brackets `⟨ _ ⟩` form sequences from tuples; ordinary parentheses are used to group tuples, when needed.

3.2 Abstract Syntax

The definition of abstract syntax in R^{5}RS is as follows:

K ∈ Con	constants, including quotations
I ∈ Ide	identifiers (variables)
E ∈ Exp	expressions
Γ ∈ Com = Exp	commands

```
Exp ⟶ K | I | (E₀ E*)
    | (lambda (I*) Γ* E₀)
    | (lambda (I* . I) Γ* E₀)
    | (lambda I Γ* E₀)
    | (if E₀ E₁ E₂) | (if E₀ E₁)
    | (set! I E)
```

Abstract syntax definitions in Agda are less concise: sorts of terms are defined as inductive datatypes, and variables are declared separately from the types over which they range:

```
postulate Con  : Set  -- constants
variable  K    : Con
postulate Ide  : Set  -- identifiers
variable  I    : Ide
data      Exp  : Set  -- expressions
variable  E    : Exp
Com            = Exp -- commands
variable  Γ    : Com
data      Ide* : Set  -- identifier sequences
variable  I*   : Ide*
data      Exp* : Set  -- expression sequences
variable  E*   : Exp*
data      Com* : Set  -- command sequences
variable  Γ*   : Com*
```

Note that `Ide*` (with no space after `Ide`) is a simple type name, not an application of the sequence domain constructor.

The names of the constructors of the datatype use underscores to indicate argument positions, but adjacent underscores need to be separated by other characters, and ordinary parentheses are not allowed in names. Most of the constructor names used in the embedding of R^{5}RS are formed from banana-brackets and the Unicode character '␣' that represents a space. The types of the constructors have to be curried.

```
data Exp where
  con                    : Con → Exp
  ide                    : Ide → Exp
  ⦇ _␣_ ⦈                : Exp → Exp* → Exp
  ⦇ lambda␣⦇ _ ⦈ _␣_ ⦈    : Ide* → Com* → Exp → Exp
  ⦇ lambda␣⦇ _ · _ ⦈ _␣_ ⦈ : Ide* → Ide → Com* → Exp → Exp
  ⦇ lambda _␣_␣_ ⦈       : Ide → Com* → Exp → Exp
  ⦇ if _␣_␣_ ⦈           : Exp → Exp → Exp → Exp
  ⦇ if _␣_ ⦈             : Exp → Exp → Exp
  ⦇ set! _␣_ ⦈           : Ide → Exp → Exp
```

Syntactic sequence sorts are defined as datatypes in the same way, e.g.:

```
data Exp* where
  ␣     : Exp*
  _␣␣_ : Exp → Exp* → Exp*
```

3.3 Domain Equations

The only required properties of domains for defining a denotational semantics are that domains can be defined recursively (up to isomorphism), and that functions on domains have well-defined fixed-points. For the shallow embedding of denotational definitions into Agda, the usual mathematical structure of domains as cpos, and the restriction of function spaces to continuous functions, are simply ignored. Thus domains are embedded as ordinary Agda types; functions on these types are always defined in lambda-notation, and assumed to have well-defined fixed-points.

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

R^{5}RS specifies the following domain equations.

$\alpha \in \mathsf{L}$		locations
$\nu \in \mathsf{N}$		natural numbers
T	$= \{\mathit{false}, \mathit{true}\}$	booleans
Q		symbols
H		characters
R		numbers
$\mathsf{E_p}$	$= \mathsf{L} \times \mathsf{L} \times \mathsf{T}$	pairs
$\mathsf{E_v}$	$= \mathsf{L}^* \times \mathsf{T}$	vectors
$\mathsf{E_s}$	$= \mathsf{L}^* \times \mathsf{T}$	strings
M	$= \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}\}$	miscellaneous
$\phi \in \mathsf{F}$	$= \mathsf{L} \times (\mathsf{E}^* \to \mathsf{K} \to \mathsf{C})$	procedure values
$\epsilon \in \mathsf{E}$	$= \mathsf{Q} + \mathsf{H} + \mathsf{R} + \mathsf{E_p} + \mathsf{E_v} + \mathsf{E_s} + \mathsf{M} + \mathsf{F}$	expressed values
$\sigma \in \mathsf{S}$	$= \mathsf{L} \to (\mathsf{E} \times \mathsf{T})$	stores
$\rho \in \mathsf{U}$	$= \mathsf{Ide} \to \mathsf{L}$	environments
$\theta \in \mathsf{C}$	$= \mathsf{S} \to \mathsf{A}$	command conts
$\kappa \in \mathsf{K}$	$= \mathsf{E}^* \to \mathsf{C}$	expression conts
A		answers
X		errors

As with abstract syntax, the Agda embedding of domain equations is less concise. The following excerpts illustrate various possibilities:

```
postulate L : Domain
variable  α : L
N           : Domain
variable  ν : N
T           : Domain
Ep          : Domain
M           : Domain
F           : Domain
variable  φ : F
postulate E : Domain
variable  ϵ : E
S           : Domain
variable  σ : S
U           : Domain
variable  ρ : U
C           : Domain
variable  θ : C
K           : Domain
variable  κ : K
postulate A : Domain
E*          = E *
variable  ϵ* : E*
```

The following Agda type definitions correspond directly to recursive definitions of domains in R^{5}RS.

```
data Misc : Set where
  false true null undefined unspecified : Misc

N  = Nat⊥
T  = Bool⊥
Ep = L × L × T
M  = Misc +⊥
F  = L × (E* → K → C)
-- E  = ... + Ep + ... + M + F
S  = L → (E × T)
U  = Ide → L
C  = S → A
K  = E* → C
```

Note especially the following points:

- Unspecified domains such as **L** are simply postulated in the embedding as elements of the type Domain.
- The embeddings of almost all other domains are types defined directly using the standard notation for domain constructors (sums, products, functions) and flat domains (truth values, natural numbers) presented in Section 3.1.
- The flat domain **M** is defined by adding ⊥ to a Scheme-specific datatype of values Misc.
- The domains **F** and **E** are mutually recursive. Their embedding avoids recursive type definitions by postulating **E** as an undefined Agda type, together with injections and projections between **E** and its summands.
- The domain name **E*** abbreviates the application of the domain constructor _* to **E**.

The injections, tests, and projections for the Agda embedding of the sum domain E need to be declared individually, e.g.:

```
postulate
  _ Ep-in-E : Ep → E
  _ ∈-Ep    : E  → T
  _ |-Ep    : E  → Ep

  _ F-in-E  : F  → E
  _ ∈-F     : E  → T
  _ |-F     : E  → F
```

The Agda embedding of the domain equations in R^{5}RS is completed by postulating the operators used in the definitions of semantic and auxiliary functions, e.g.:

```
postulate
  _ ==L _ : L → L → T
  _ ==M _ : M → M → T
```

3.4 Semantic Functions

The Agda embedding of semantic function declarations and definitions is quite direct. Agda notation for λ-abstractions differs from the conventional notation used in R^{5}RS by using 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 and keywords.

R^{5}RS declares the following semantic functions:

$\mathcal{K}$: Con → E
$\mathcal{E}$: Exp → U → K → C
$\mathcal{E}^*$: Exp* → U → K → C
$\mathcal{C}$: Com* → U → C → C

Their Agda embedding incorporates the double-bracket notation $[\![\cdots]\!]$ in the names of the functions, using an underscore as a placeholder for the syntactic argument:

```
postulate 𝒦⟦ _ ⟧ : Con → E
          ℰ⟦ _ ⟧ : Exp → U → K → C
          ℰ*⟦ _ ⟧ : Exp* → U → (K → C)*
          C*⟦ _ ⟧ : Com* → U → C → C
```

(The reason for the difference between the type of denotations of expression sequences in R^{5}RS and in the embedding is explained in Section 5.2.)

The definition of $\mathcal{K}$ is deliberately omitted in R^{5}RS. This is indicated in the Agda embedding by declaring $\mathcal{K}[\![\_]\!]$ as a postulate.

The semantic equation for constant expressions K is:

$$\mathcal{E}[\![\,\mathrm{K}\,]\!] = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\,\mathrm{K}\,]\!])\,\kappa$$

In Agda, the only difference is the explicit construction of expressions from constants by con:

```
ℰ⟦ con K ⟧ = λ ρ κ → send (𝒦⟦ K ⟧) κ
```

The semantic equation for identifiers I in R^{5}RS is:

$$\begin{aligned}&\mathcal{E}[\![\,\mathrm{I}\,]\!] =\\ &\quad\lambda\rho\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\ &\qquad\qquad(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \rightarrow\\ &\qquad\qquad\qquad\mathit{wrong}\text{ “undefined variable”},\\ &\qquad\qquad\mathit{send}\,\epsilon\,\kappa))\end{aligned}$$

Formally, the test $\epsilon = \mathit{undefined}$ used above involves an implicit projection from E to the flat domain M, which has to be made explicit in the Agda embedding, as well as the injection η from Misc to **M**:

```
ℰ⟦ ide I ⟧ = λ ρ κ →
  hold (lookup ρ I)
       (single (λ ϵ → ((ϵ |-M) ==M η undefined) ⟶
                        wrong "undefined variable" ,
                       send ϵ κ))
```

The semantic equation for conditional expressions in R^{5}RS uses the auxiliary function *truish* : E → T, which is defined as the strict map taking the *false* value in the summand M of E to the value *false* in T, and all other non-⊥ arguments to *true* in T.

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

In the embedding, no projections or injections need to be added:

```
ℰ⟦ (| if E₀ ␣ E₁ ␣ E₂ |) ⟧ = λ ρ κ →
  ℰ⟦ E₀ ⟧ ρ (single (λ ϵ → truish ϵ ⟶ ℰ⟦ E₁ ⟧ ρ κ ,
                           ℰ⟦ E₂ ⟧ ρ κ))
```

Agda embeddings of the semantic equations for procedure call and lambda-abstraction are illustrated in Section 5.

3.5 Auxiliary Functions

The following examples of auxiliary functions from R^{5}RS and their Agda embeddings illustrate most of the additional notation required by the embedding.

R^{5}RS includes:

$$\begin{aligned}&\mathit{send} : \mathrm{E} \rightarrow \mathrm{K} \rightarrow \mathrm{C}\\ &\mathit{send} = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle\end{aligned}$$

Its Agda embedding is direct

```
send : E → K → C
send = λ ϵ κ → κ ⟨ ϵ ⟩
```

The function *single* is used to convert ordinary continuations to continuations that take one-element sequences:

$$
\begin{array}{l}
\mathit{single} : (\mathsf{E} \to \mathsf{C}) \to \mathsf{K} \\
\mathit{single} = \\
\quad \lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1), \\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of return values”}
\end{array}
$$

Its embedding uses the infix operator $\_\downarrow\_$ to select an element of **E** from a sequence in $\mathbf{E}^*$:

```
single : (E → C) → K
single =
  λ ψ ϵ* → (# ϵ* ==⊥ 1) ⟶ ψ (ϵ* ↓ 1) ,
             wrong "wrong number of return values"
```

The following definition in R^{5}RS involves also the store:

$$
\begin{array}{l}
\mathit{hold} : \mathsf{L} \to \mathsf{K} \to \mathsf{C} \\
\mathit{hold} = \lambda\alpha\kappa\sigma \,.\, \mathit{send}\,(\sigma\alpha \downarrow 1)\kappa\sigma
\end{array}
$$

The term $\sigma\alpha$ denotes a pair of a location and a truth-value. In the Agda embedding, pairs and sequences are different types, and the postfix operator $\_\downarrow^2 1 : \mathbf{E} \times \mathbf{T} \to \mathbf{E}$ needs to be used to select the first component of a pair:

```
hold : L → K → C
hold = λ α κ σ → send (σ α ↓²1) κ σ
```

The function *takefirst* is one of several that are defined recursively in R^{5}RS. Its type is not declared, but for well-formedness it needs to be $\mathsf{E}^* \to \mathsf{N} \to \mathsf{E}^*$, where N is the flat domain of natural numbers.

$$
\begin{array}{l}
\mathit{takefirst} = \\
\quad \lambda l n \,.\, n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle \,\S\, (\mathit{takefirst}\,(l \dagger 1)(n - 1))
\end{array}
$$

The Agda embedding defines takefirst inductively in its second argument. This is possible because that argument is the length of a sequence of identifiers, which is of type Nat.

```
takefirst : E* → Nat → E*
takefirst ϵ* 0       = ⟨⟩
takefirst ϵ* (suc n) = ⟨ ϵ* ↓ 1 ⟩ § takefirst (ϵ* † 1) n
```

Agda requires recursive function definitions to be evidently terminating. Functions that recurse on sequences ϵ^* in the postulated type $\mathbf{E}^*$ cannot be defined by pattern-matching on the structure of ϵ^*, and their embedding requires explicit use of the fixed-point operator fix. For example, R^{5}RS defines the auxiliary function *list* as:

$$
\begin{array}{l}
\mathit{list} : \mathsf{E}^* \to \mathsf{K} \to \mathsf{C} \\
\mathit{list} = \\
\quad \lambda\epsilon^*\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\ \mathit{null}\ \kappa, \\
\qquad\qquad \mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{array}
$$

Its Agda embedding is:

```
list : E* → K → C
list = fix λ list′ →
  λ ϵ* κ → (# ϵ* ==⊥ 0) ⟶ send (η null M-in-E) κ ,
           list′ (ϵ* † 1)
                 (single (λ ϵ → cons ⟨ (ϵ* ↓ 1) , ϵ ⟩ κ))
```

4 Wellformedness Issues

This section discusses various issues with the wellformedness of the denotational semantics in R^{5}RS, mostly revealed by its embedding in Agda. The next section suggests changes that would address the issues.

The Agda embedding described in the previous section generally preserves the types of denotations. It also preserves the lambda-notation in the definitions of auxiliary functions and denotations, except that omitted injections and projections between E and its summands need to be inserted. The main changes to the abstract syntax notation used in the semantic equations concern the symbols in the names of abstract syntax constructors.

Type-checking the Agda embedding of R^{5}RS thus indirectly tests the definitions in R^{5}RS for wellformedness. The closeness of the notation used in the Agda embedding (largely due to exploiting the Agda support for Unicode) facilitates pinpointing the origin in R^{5}RS of reported issues.

However, the Agda type system has significant differences from the mathematical underpinnings of denotational semantics. For example, Agda does not accept recursive type definitions, whereas domains can be recursively defined (up to isomorphism) by domain equations in denotational semantics. And Agda accepts recursive function definitions only when it can prove that applications always terminate. On the other hand, dependent types, implicit arguments, and instance arguments make type-checking in Agda significantly more general than in domain theory.

Thus type-checking an Agda embedding can result in both false positives and false negatives. Nevertheless, it has turned out to be sufficiently precise in practice for testing the wellformedness of the formal semantics in R^{5}RS.

4.1 Non-compositionality

A denotational semantics is supposed to be *compositional*, where [26, §1.2]:

> A semantics is said to be compositional when the meaning of each phrase does not depend on any property of its immediate subphrases except the meanings of these subphrases.

As stated in the Agda language reference [40]:

> Agda accepts only these recursive schemas that it can mechanically prove 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 acceptance of a recursive definition of the Agda embedding of a semantic function does not check that it is actually compositional. On the other hand, Agda's rejection of a definition because it cannot prove it terminating shows that it cannot be compositional.

Agda rejects the embedding of the following two semantic equations from R^{5}RS due to potential non-termination, which implies that they are non-compositional (as can anyway be easily seen from the applications of $\mathcal{E}^*$ and $\mathcal{E}$ to arguments that are not simply variables):

$$\mathcal{E}[\![\ (\mathrm{E}_0\ \ \mathrm{E}^*)\ ]\!] =$$
$$\lambda\rho\kappa\ .\ \mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle \S \mathrm{E}^*))$$
$$\rho$$
$$(\lambda\epsilon^*\ .\ ((\lambda\epsilon^*\ .\ \mathit{applicate}\ (\epsilon^* \downarrow 1)\ (\epsilon^* \dagger 1)\ \kappa)$$
$$(\mathit{unpermute}\ \epsilon^*)))$$

$$\mathcal{E}[\![\ (\texttt{lambda I}\ \Gamma^*\ \mathrm{E}_0)\ ]\!] = \mathcal{E}[\![\ (\texttt{lambda (. I)}\ \Gamma^*\ \mathrm{E}_0)\ ]\!]$$

In both cases, it is straightforward to replace them by compositional equations, as suggested in Section 5.

4.2 Type Errors

Two significant type errors in R^{5}RS are detected by Agda when type-checking the embedding.

Ironically, one of them concerns the auxiliary function *wrong*: its type is declared to be $\mathtt{X} \to \mathtt{C}$, but it is applied to strings, and X is left unspecified. Agda reported the error that `String` is not a subtype of X when checking applications of *wrong*.

The other type error is due to omission of an argument. The very last auxiliary function definition in R^{5}RS is:

$$\mathit{cwv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad \texttt{[call-with-values]}$$
$$\mathit{cwv} =$$
$$\mathit{twoarg}\ (\lambda\epsilon_1\epsilon_2\kappa\ .\ \mathit{applicate}\ \epsilon_1\langle\ \rangle(\lambda\epsilon^*\ .\ \mathit{applicate}\ \epsilon_2\ \epsilon^*))$$

The type of *applicate* is $\mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$, so the second occurence of *applicate* above needs to be applied to a continuation. (This error was already noticed by Van Straaten in 2002, in connection with implementing the R^{5}RS semantics in Scheme [36]. The same error occurs also in R^{7}RS.)

As noted in Section 3, the types of the auxiliary functions *takefirst* and *dropfirst* are not declared in R^{5}RS. This was reported as an error when type-checking the Agda embedding.

Two wellformedness errors that were not detected by type-checking the Agda embedding concern the use of literal sets as domains in domain equations:

$$\mathtt{T} = \{\mathit{false}, \mathit{true}\}$$
$$\mathtt{M} = \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}\}$$

and in the type of an auxiliary function:

$$\mathit{new} : \mathtt{S} \to (\mathtt{L} + \{\mathit{error}\})$$

But the lack of detection of such errors is unsurprising, since domains are not distinguished from ordinary Agda types in the embedding, as explained in Section 3.1.

4.3 Overloaded Names

In R^{5}RS, the names *false* and *true* are overloaded as elements of the domains T and M. The definition of *truish* uses *false* in both domains:

$$\mathit{truish} : \mathtt{E} \to \mathtt{T}$$
$$\mathit{truish} = \lambda\epsilon\ .\ \epsilon = \mathit{false} \to \mathit{false}, \mathit{true}$$

The first occurrence of *false* can be seen to be in M, since $\epsilon \in \mathtt{E}$ and M is a summand of E; and the second occurrence is required to be in T by the type of *truish*. As Agda does not support implicit injections, explicit injections have to be added in the Agda embedding of R^{5}RS.

However, the lack of injections or projections in the condition $\epsilon = \mathit{false}$ was reported as an error – because the equality test is allowed only on flat domains, and $\epsilon \in \mathtt{E} = \cdots + \mathtt{F}$, which is not a flat domain. It is necessary to check first whether ϵ is in the summand M, then test its projection to M for equality to *false*. See Section 5 for the suggested definition, which is embedded straightforwardly in Agda.

In R^{5}RS the notation $s \downarrow k$ is used not only for selecting the kth member of the sequence s, but also for selecting components of elements of product domains. Similarly, _[_/_] is used as an operation both on environments ρ and on stores σ. Notational variants of these operators were introduced in their Agda embedding to eliminate such type-inconsistent overloading.

5 Suggestions

This section suggests changes for improving the wellformedness, conciseness, and perspicuity of the denotational semantics in the Scheme reports. Most of the suggestions are independent of the n in R^{n}RS; the fragments quoted below were all copied from R^{5}RS [10]. The suggestions are listed in order of occurrence in §7.2.

5.1 Domain Equations

$$\mathtt{T} = \{\mathit{false}, \mathit{true}\} \quad \text{booleans}$$
$$\mathtt{M} = \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}\}$$

It is confusing to use the same names for values in different domains. If T were to be a summand of E, *false* and *true* could be removed from M.

$$\mathtt{X} \quad \text{errors}$$

As discussed in Section 4.2, the type of the auxiliary function *wrong* is $\mathtt{X} \to \mathtt{C}$, but *wrong* is applied to strings, and X is left undefined. This wellformedness issue could be addressed by defining X to be a flat domain containing the literal strings used as error messages. However, the Agda embedding of applications of *wrong* to literal strings would then need explicit injections of strings into the flat domain; to avoid these tedious injections, the embedding of the argument type of *wrong* is an Agda datatype of strings, rather than a domain.

5.2 Semantic Functions

See Section 4.1 for discussion of non-compositionality issues. The following compositional semantics of procedure calls permutes thunks instead of raw expressions:

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

where

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

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

$$\mathcal{E}^*[\![\ \mathrm{E}_0\ \mathrm{E}^*\ ]\!] = \lambda\rho\ .\ \langle \mathcal{E}[\![\ \mathrm{E}_0\ ]\!]\ \rho\rangle \S \mathcal{E}^*[\![\ \mathrm{E}^*\ ]\!]\ \rho$$

$$\mathit{permute} : (\mathsf{K} \to \mathsf{C})^* \to (\mathsf{K} \to \mathsf{C})^*$$
[implementation-dependent]

$$\mathit{unpermute} : \mathsf{E}^* \to \mathsf{E}^* \quad \text{[inverse of } \mathit{permute}\text{]}$$

$$\mathit{forces} : (\mathsf{K} \to \mathsf{C})^* \to \mathsf{K} \to \mathsf{C}$$
$$\mathit{forces} =$$
$$\quad \lambda\zeta^*\kappa\ .\ \#\zeta^* = 0 \to \kappa\ \langle\ \rangle,$$
$$\qquad (\zeta^* \downarrow 1)$$
$$\qquad (\mathit{single}(\lambda\epsilon\ .\ \mathit{forces}\ (\zeta^* \dagger 1)$$
$$\qquad\qquad (\lambda\epsilon^*\ .\ \kappa\ (\langle\epsilon\rangle \S \epsilon^*))))$$

The Agda embedding is:

```
ℰ⟦ ( E₀ ␣ E* ) ⟧ =
  λ ρ κ →
    forces (permute ( ⟨ ℰ⟦ E₀ ⟧ ρ ⟩ § ℰ*⟦ E* ⟧ ρ))
           (λ ϵ* →
             ( (λ ϵ*′ → applicate (ϵ*′ ↓ 1) (ϵ*′ † 1) κ)
               (unpermute ϵ*) ))

ℰ*⟦ _ ⟧ : Exp* → U → (K → C)*
ℰ*⟦ ␣ ⟧ = λ ρ → ⟨⟩
ℰ*⟦ E ␣ E* ⟧ = λ ρ → ⟨ ℰ⟦ E ⟧ ρ ⟩ § ℰ*⟦ E* ⟧ ρ

postulate permute   : (K → C)* → (K → C)*

postulate unpermute : E* → E*

forces : (K → C)* → K → C
forces = fix λ forces′ →
  λ ζ* κ → (# ζ* ==⊥ 0) ⟶ κ ⟨⟩ ,
           (ζ* ↓ 1)
           (single (λ ϵ → forces′ (ζ* † 1)
                                  (λ ϵ* → κ (⟨ ϵ ⟩ § ϵ*))))
```

Anglade, Lacrampe, and Queinnec [1] suggest denotations of procedure calls that generate all possible permutations of their subexpression denotations, and collect sequences of the results obtained by composing them sequentially.

The right-hand side of the non-compositional semantic equation for lambda-abstractions could simply be expanded using $\mathcal{E}[\![$ `(lambda (I* . I) Γ* E0)` $]\!]$ with I^* empty:

$$\mathcal{E}[\![\ \texttt{(lambda I}\ \Gamma^*\ \mathrm{E}_0\texttt{)}\ ]\!] =$$
$$\quad \lambda\rho\kappa\ .\ \lambda\sigma\ .$$
$$\qquad \mathit{new}\ \sigma \in \mathsf{L} \to$$
$$\qquad\quad \mathit{send}\ (\langle \mathit{new}\ \sigma \mid \mathsf{L},$$
$$\qquad\qquad\quad \lambda\epsilon^*\kappa'\ .\ \mathit{tievalsrest}$$
$$\qquad\qquad\qquad\quad (\lambda\alpha^*\ .\ (\lambda\rho'\ .\ \mathcal{C}[\![\ \Gamma^*\ ]\!]\rho'(\mathcal{E}[\![\ \mathrm{E}_0\ ]\!]\rho'\kappa'))$$
$$\qquad\qquad\qquad\qquad (\mathit{extends}\ \rho\ \langle \mathrm{I}\rangle\ \alpha^*))$$
$$\qquad\qquad\qquad\quad \epsilon^*$$
$$\qquad\qquad\qquad\quad 0\rangle\ \text{in}\ \mathsf{E})$$
$$\qquad\qquad \kappa$$
$$\qquad\qquad (\mathit{update}\ (\mathit{new}\ \sigma \mid \mathsf{L})\ \mathit{unspecified}\ \sigma),$$
$$\qquad\quad \mathit{wrong}\ \text{“out of memory”}\ \sigma$$

5.3 Auxiliary Functions

The correction for the type error reported in the definition of *cwv*, discussed in Section 4.2, appears to be as follows:

$$\mathit{cwv} : \mathsf{E}^* \to \mathsf{K} \to \mathsf{C} \qquad \texttt{[call-with-values]}$$
$$\mathit{cwv} =$$
$$\quad \mathit{twoarg}\ (\lambda\epsilon_1\epsilon_2\kappa\ .\ \mathit{applicate}\ \epsilon_1\langle\ \rangle(\lambda\epsilon^*\ .\ \mathit{applicate}\ \epsilon_2\ \epsilon^*\ \kappa))$$

The following declarations should be added before the corresponding definitions:

$$\mathit{takefirst} : \mathsf{E}^* \to \mathsf{N} \to \mathsf{E}^*$$
$$\mathit{dropfirst} : \mathsf{E}^* \to \mathsf{N} \to \mathsf{E}^*$$

The auxiliary function $\mathit{new} : \mathsf{S} \to (\mathsf{L} + \{\mathit{error}\})$ is rather awkward to use, because it requires explicit application to the current store σ, which is usually left implicit in the continuation-passing style adopted by the Scheme semantics. Moreover, it is tedious to test $\mathit{new}\ \sigma \in \mathsf{L}$, then use $\mathit{new}\ \sigma \mid \mathsf{L}$ when true or continue with *wrong* “out of memory” σ when false. And care is needed in subsequent applications of *new* to refer to an updated store where the new location is marked as in use.

A simple way to express storage allocation in continuation-passing style is to define an auxiliary function *alloc* to allocate a new location and update it to an initial value, as follows:

$$\mathit{alloc} : \mathsf{E} \to (\mathsf{L} \to \mathsf{C}) \to \mathsf{C}$$
$$\mathit{alloc} =$$
$$\quad \lambda\epsilon\chi\sigma\ .\ \mathit{new}\ \sigma \in \mathsf{L} \to$$
$$\qquad\qquad \chi\ (\mathit{new}\ \sigma \mid \mathsf{L})\ (\mathit{update}\ (\mathit{new}\ \sigma \mid \mathsf{L})\ \epsilon\ \sigma),$$
$$\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma$$

The definition of the auxiliary function *cons* in R^{5}RS is:

```
cons : E* → K → C
cons =
  twoarg (λε₁ε₂κσ . new σ ∈ L →
                     (λσ′ . new σ′ ∈ L →
                       send (⟨new σ | L, new σ′ | L, true⟩
                              in E)
                            κ
                            (update(new σ′ | L)ε₂σ′),
                       wrong "out of memory"σ′)
                     (update(new σ | L)ε₁σ),
                     wrong "out of memory"σ)
```

Using *alloc* would significantly simplify the definition:

```
cons =
  twoarg (λε₁ε₂κ . alloc ε₁ (λα₁ .
                    alloc ε₂ (λα₂ .
                      send (⟨α₁, α₂, true⟩ in E) κ)))
```

Similarly, using *alloc unspecified* would simplify the definitions of the denotations of Scheme lambda-abstractions.

Tennent [39, §7.5] suggests defining all primitive store functions to take continuations. Apart from notational convenience, this makes it possible for the store functions to terminate program execution simply by discarding their continuation argument. In fact the functions *hold* and *assign* in R^{5}RS are already defined to take continuations; the declaration of *new* could easily be adjusted to do the same.

6 Related Work

The Agda *TypeTopology* library is based on univalent foundations. It includes modules for Scott domain theory, and illustrates their use in denotational definitions of PCF and the untyped λ-calculus [7]. This Agda formalization of domain theory corresponds directly to the usual set-theoretic definitions: a domain consists of a carrier type together with a partial order relation, its least element, and proofs of the required completeness properties; a continuous function between domains is an underlying function between their carrier types, paired with a proof of its continuity. Currently, the formalization requires definitions of denotations in λ-notation to include explicit continuity proofs, and subsequently discard the proof terms when applying functions. This prevents direct embedding of λ-notation from conventional denotational definitions, and seems quite impractical for formalizing the denotational semantics of larger languages (especially in the continuation-passing style used in the Scheme reports).

The author previously developed a relatively lightweight formalization of denotational semantics in Agda, and used it to detect (minor) wellformedness issues in a semantics of inheritance in object-oriented systems [15]. Recursive domain definitions were embedded as (unsatisfiable) assumptions of isomorphisms between Agda types, circumventing the issues with recursive definitions of Agda types.

A major drawback in [15] was the widespread insertion of isomorphisms in function definitions. In the present paper, almost all domain equations are embedded as Agda type definitions, and no isomorphisms between domains are needed. However, when embedding the denotational semantics of the untyped λ-calculus in Agda [17], the domain D_∞ is required to be isomorphic to the domain $D_\infty \to D_\infty$, and there the isomorphism needs to be explicit in the embedding.

The author has recently completed a shallow Agda embedding of a denotational semantics for a simple form of `eval` expressions, and presented it in a paper [16] in the proceedings of OlivierFest '25. The semantics is based on an adaptation of a suggestion made by Clinger [4] more than 40 years ago. A language that includes `eval` is defined incrementally, starting from a particularly basic sublanguage Scm of the core Scheme expressions whose denotational semantics is defined in the Scheme reports. ScmQ extends Scm with literal quotations, then ScmQE adds `eval`. An archive of the Agda source code is available as supplemental material [18] accompanying the cited paper. The current version of the Agda embedding of ScmQE includes also tentative abstract syntax and semantics for Scheme programs and definitions.

7 Conclusion

After recalling the history of the Scheme reports, this paper focused on the denotational semantics of primitive Scheme constructs and selected procedures. It introduced a shallow embedding of the semantics in Agda, developed systematically from the definitions given in §7.2 of R^{5}RS. Checking the embedding using the Agda proof assistant detected various issues with the wellformedness of the original definitions. Suggestions for how to address those issues have been made, as well as for further changes that could improve the perspicuity and conciseness of the semantics.

Regardless of whether there will ever be any further revisions of the current Scheme language and its report, it would surely be worthwhile to replace the denotational semantics in R^{7}RS by a corrected version that addresses at least the non-compositionality and the omitted argument in the definition of *cwv*. Systematic use of continuation-passing style would significantly simplify some of the semantic equations with only minor changes to the definitions of the auxiliary functions. An extension to include the semantics of programs, definitions, and literal truth-values and quotations could be based on their Agda embedding in [18]

Clearly, any update of the denotational semantics in R^{7}RS should be carefully checked – not only for wellformedness, but also for soundness of the specified semantics relative to reference implementations of Scheme. The shallow embedding of denotational semantics in Agda has been useful for detecting wellformedness issues, and might be used as a basis for proving properties of denotations, but it seems unsuitable for executing programs to test their behavior.

Various systems were developed from the late 1970s to the 1990s for implementing the denotational semantics of programming languages, based on formal specifications for the syntax and wellformedness of the meta-notation, but the author is not aware of any that are still in use. It could be interesting to develop a deep embedding of denotational semantics in some modern language workbench, and use it to make the current Scheme semantics truly executable.

To some extent, the shallow embedding of denotational semantics in Agda presented in this paper corresponds to Strachey's original use of untyped λ-calculus to formally specify the semantics of programming languages [37]: in the absence of a model of the untyped λ-calculus, he relied on reasoning about λ-expressions using their laws, from which the unfolding property of the fixed point combinator can be derived. He subsequently embraced the models provided by Scott's theory of domains, where the fixed point operator is defined as the limit of the Kleene sequence, and proved to satisfy unfolding. Synthetic domain theory (SDT) [25, 30, 32] reconciles postulated axioms with domain-theoretic models and constructive logic; a formalization of SDT in Agda would avoid the unsound postulates used in the current shallow embedding of denotational semantics.

Acknowledgments

This paper is dedicated to the memory of Christopher Strachey, who passed away 50 years ago, aged 58. He accomplished a great deal as a computing pioneer in the 1950s and 1960s [3]. By 1964, he was already using the lambda calculus to specify the semantics of programming constructs [37]. From 1969, in collaboration with Dana Scott, he developed the denotational style of formal semantics [29, 35, 39]. He was a hugely inspiring advisor for my doctoral studies [13].

I would like to thank William Clinger, John D. Ramsdell, Anton van Straaten, and Mitchell Wand for sharing their recollections of the origin and evolution of the denotational semantics included in the Scheme reports, as summarized in Section 2.1.

Thanks also to Jesper Cockx, who provided helpful comments on a draft of the submitted paper, and subsequently on a draft of the discussion in Section 3.1 of the advantages of using postulates instead of module parameters.

I am grateful to the anonymous reviewers for their perceptive comments on the submitted paper and their constructive suggestions for its improvement.

Data-Availability Statement

The Agda code in the accompanying artifact [19] is a shallow embedding of the denotational semantics presented in R^{5}RS [10]. The relationship of the embedding to the definitions in the Scheme report is explained in Section 3 of the present paper.

The artifact includes a literate version of the embedding, where the Agda code is interspersed with explanatory prose (mostly copied from the paper).

The literate version also interleaves the original definitions from R^{5}RS with their embedding into Agda, to facilitate comparison.

After downloading the artifact, the type-correctness of all the Agda code can be checked by executing a single command. The development of soundness tests for the embedding is work in progress.

Highlighted listings of both the plain and literate versions of the Agda embedding are available in PDF as supplemental material accompanying this paper. They were generated using the artifact.

The artifact is an archive of a release of the public GitHub repository pdmosses/scheme25-agda.

References

[1] Sophie Anglade, Jean-Jacques Lacrampe, and Christian Queinnec. 1994. Semantics of combinations in Scheme. *SIGPLAN Lisp Pointers* VII, 4 (Oct. 1994), 15–20. doi:10.1145/382109.382669

[2] James Bodwin, Laurette Bradley, Kohji Kanda, Diane Litle, and Uwe Pleban. 1982. Experience with an experimental compiler generator based on denotational semantics. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 216–229. doi:10.1145/800230.806997

[3] Martin Campbell-Kelly. 1985. Christopher Strachey, 1916-1975: A Biographical Note . *IEEE Annals of the History of Computing* 7, 01 (Jan. 1985), 19–42. doi:10.1109/MAHC.1985.10001

[4] 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

[5] William Clinger. 1985. *The revised revised report on Scheme, or an uncommon Lisp.* Technical Report MIT Artificial Intelligence Memo 848. MIT. https://standards.scheme.org/official/r2rs.pdf Also published as Computer Science Department Technical Report 174, Indiana University, June 1985..

[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] Tom de Jong. since 2019. *TypeTopology/DomainTheory (Agda modules).* Retrieved Augest 25, 2025 from https://martinescardo.github.io/TypeTopology/DomainTheory.index.html

[8] Véronique Donzeau-Gouge, Gilles Kahn, and Bernard Lang. 1978. *A complete machine-checked definition of a simple programming language using denotational semantics.* Technical report RR-330. IRIA, Rocquencourt. https://inria.hal.science/hal-04716568/

[9] Guy Lewis Steele Jr. and Gerald Jay Sussman. 1978. *The revised report on Scheme: A dialect of Lisp.* Technical Report MIT Artificial Intelligence Memo 452. MIT. https://standards.scheme.org/official/r1rs.pdf

[10] 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

[11] 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

[12] Peter D. Mosses. 1974. The semantics of semantic equations. In *Mathematical Foundations of Computer Science, 3rd Symposium at Jadwisin near Warsaw, Poland, June 17-22, 1974, Proceedings (Lecture Notes in Computer Science, Vol. 28)*, Andrzej Blikle (Ed.). Springer, Berlin, Heidelberg, 409–422. doi:10.1007/3-540-07162-8_701

[13] Peter D. Mosses. 1975. *Mathematical Semantics and Compiler Generation.* DPhil dissertation. University of Oxford.

[14] Peter D. Mosses. 2024. *SIS.* Retrieved August 25, 2025 from https://pdmosses.github.io/software/sis/

[15] Peter D. Mosses. 2024. Towards Verification of a Denotational Semantics of Inheritance. In *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday* (Pasadena, CA, USA) *(JENSFEST '24)*. ACM, New York, NY, USA, 5–13. doi:10.1145/3694848.3694852

[16] 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* (Singapore, Singapore). ACM, New York, NY, USA, 10 pages. doi:10.1145/3759427.3760369

[17] Peter D. Mosses. 2025. Lightweight Agda formalization of denotational semantics. In *Proceedings of the 31st International Conference on Types for Proofs and Programs (TYPES 2025)*, Fredrik Nordvall Forsberg (Ed.). University of Strathclyde, Glasgow, Scotland, 286–290. https://msp.cis.strath.ac.uk/types2025/TYPES2025-book-of-abstracts.pdf

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

[19] Peter D. Mosses. 2025. Shallow Agda embedding of denotational semantics in article 'Checking a denotational semantics of Scheme in Agda'. ACM. doi:10.1145/3747410

[20] 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

[21] Uwe F. Pleban. 1981. *Preexecution Analysis Based on Denotational Semantics.* Ph. D. Dissertation. University of Kansas.

[22] Christian Queinnec. 1996. *Lisp in Small Pieces.* Cambridge University Press, Cambridge, UK. doi:10.1017/CBO9781139172974

[23] Christian Queinnec. 2003. *L2T: Literate Programming Utility.* LIP6, Université Pierre et Marie Curie. Retrieved August 25, 2025 from https://christian.queinnec.org/WWW/l2t.html

[24] 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

[25] Bernhard Reus. 1999. Formalizing Synthetic Domain Theory. *Journal of Automated Reasoning* 23 (1999), 411–444. doi:10.1023/A:1006258506401

[26] John C. Reynolds. 1998. *Theories of Programming Languages.* Cambridge Univ. Press, Cambridge, UK. doi:10.1017/CBO9780511626364

[27] Scheme Working Group, Microprocessor and Microcomputer Standards Subcommittee. 1991. *IEEE Standard 1178-1990 for the Scheme Programming Language.* Technical Report IEEE1178. IEEE, New York, NY, USA.

[28] Scheme.org [n. d.]. *Scheme Standards.* Retrieved August 25, 2025 from https://standards.scheme.org

[29] Dana Scott and Christopher Strachey. 1971. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata (Microwave Research Inst. Symposia Series, Vol. 21)*. Polytechnic Inst. of Brooklyn, New York, NY, USA, 19–46. Also: Tech. Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971). URL https://www.cs.ox.ac.uk/files/3228/PRG06.pdf.

[30] Dana S. Scott. 1980. Relating Theories of the Lambda-calculus: Dedicated to Professor H. B. Curry on the Occasion of His 80th Birthday. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley (Eds.). Academic Press, London, UK, 403–450. https://prl.khoury.northeastern.edu/blog/static/scott-80-relating-theories.pdf

[31] Alex Shinn, John Cowan, and Arthur A. Gleckler. 2021. *Revised^7 Report on the Algorithmic Language Scheme: Small Edition.* R7RS Working Group 1. Retrieved August 25, 2025 from https://standards.scheme.org/official/r7rs.pdf

[32] Alex Simpson. 2004. Computational Adequacy for Recursive Types in Models of Intuitionistic Set Theory. *Annals of Pure and Applied Logic* 130, 1 (2004), 207–275. doi:10.1016/j.apal.2003.12.005 Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS).

[33] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. 2007. *Revised^7 Report on the Algorithmic Language Scheme: Rationale.* R7RS Working Group 1. Retrieved August 25, 2025 from https://standards.scheme.org/official/r6rs-rationale.pdf

[34] 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, Cambridge, UK. https://standards.scheme.org/official/r6rs.pdf

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

[36] Anton van Straaten. 2002. *An Executable Denotational Semantics for Scheme.* AppSolutions. Retrieved August 25, 2025 from https://web.archive.org/web/20130531130945/http://www.appsolutions.com/SchemeDS/ds.html

[37] Christopher Strachey. 1966. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming, Proc. IFIP Working Conference, 1964.* North-Holland, Amsterdam, Netherlands, 198–220.

[38] 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

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

[40] The Agda Team. 2025. *Agda Language Reference.* Retrieved August 25, 2025 from https://agda.readthedocs.io/en/v2.8.0/language/

[41] Wikipedia. 2025. *Agda.* Retrieved August 25, 2025 from https://en.wikipedia.org/wiki/Agda_(programming_language)

Received 2025-07-25; accepted 2025-08-14