Denotational Semantics of Scheme R^5 in Agda

DRAFT (March 15, 2025)

Peter D. Mosses

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

Abstract

In synthetic domain theory, all sets are predomains, domains are pointed sets, and functions are implicitly continuous. The denotational semantics of Scheme (R^5) presented here illustrates how it might look if synthetic domain theory can be implemented in Agda. As a work-around, the code presented here uses unsatisfiable postulates to allow Agda to type-check the definitions.

The (currently illiterate) Agda source code used to generate this document can be downloaded from https://github.com/pdmosses/xds-agda, and browsed with hyperlinks and highlighting at https://pdmosses.github.io/xds-agda/.

```
{-  Agda formalization of the denotational semantics of Scheme R5

    Based on a plain text copy of §7.2 in [R5RS]

    [R5RS]: https://standards.scheme.org/official/r5rs.pdf
-}

module Scheme.All where

import Scheme.Domain-Notation
import Scheme.Abstract-Syntax
import Scheme.Domain-Equations
import Scheme.Auxiliary-Functions
import Scheme.Semantic-Functions
```

```
module Scheme.Domain-Notation where

open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl) public

------------------------------------------------------------------------
-- Agda requires Predomain and Domain to be sorts

Predomain = Set
Domain    = Set
variable
  P   : Predomain
  D E : Domain

-- Domains are pointed
postulate
  ⊥        : {D : Domain} → D
  strict   : {D E : Domain} → (D → E) → (D → E)

  -- Properties
  strict-⊥ : ∀ {D E} → (f : D → E) →
               strict f ⊥ ≡ ⊥

------------------------------------------------------------------------
-- Fixed points of endofunctions on function domains

postulate
  fix      : {D : Domain} → (D → D) → D

  -- Properties
  fix-fix  : ∀ {D} (f : D → D) →
               fix f ≡ f (fix f)
  fix-app  : ∀ {P D} (f : (P → D) → (P → D)) (p : P) →
               fix f p ≡ f (fix f) p

------------------------------------------------------------------------
-- Lifted domains

postulate
  𝕃        : Predomain → Domain
  η        : {P : Predomain} → P → 𝕃 P
  _♯       : {P : Predomain} {D : Domain} → (P → D) → (𝕃 P → D)

  -- Properties
  elim-♯-η : ∀ {P D} (f : P → D) (p : P) →
               (f ♯) (η p) ≡ f p
  elim-♯-⊥ : ∀ {P D} (f : P → D) →
               (f ♯) ⊥ ≡ ⊥
```

```
------------------------------------------------------------------------
-- Flat domains

_+⊥ : Set → Domain
S +⊥ = 𝕃 S

-- Lifted operations on ℕ

open import Agda.Builtin.Nat
  using (_==_; _<_) public
open import Data.Nat.Base as Nat
  using (ℕ; suc; pred) public
open import Data.Bool.Base
  using (Bool) public

-- ν ==⊥ n : Bool +⊥

_==⊥_ : ℕ +⊥ → ℕ → Bool +⊥
ν ==⊥ n = ((λ m → η (m == n)) ♯) ν

-- ν >=⊥ n : Bool +⊥

_>=⊥_ : ℕ +⊥ → ℕ → Bool +⊥
ν >=⊥ n = ((λ m → η (n < m)) ♯) ν

------------------------------------------------------------------------
-- Products

-- Products of (pre)domains are Cartesian

open import Data.Product.Base
  using (_×_; _,_) renaming (proj₁ to _↓1; proj₂ to _↓2) public

-- (p₁ , ... , pₙ) : P₁ × ... × Pₙ  (n ≥ 2)
-- _↓1 : P₁ × P₂ → P₁
-- _↓2 : P₁ × P₂ → P₂

------------------------------------------------------------------------
-- Sum domains

-- Disjoint unions of (pre)domains are unpointed predomains
-- Lifted disjoint unions of domains are separated sum domains

open import Data.Sum.Base
  using (inj₁; inj₂) renaming (_⊎_ to _+_; [_,_]′ to [_,_]) public

-- inj₁ : P₁ → P₁ + P₂
-- inj₂ : P₂ → P₁ + P₂
-- [ f₁ , f₂ ] : (P₁ → P) → (P₂ → P) → (P₁ + P₂) → P
```

```
------------------------------------------------------------------------
-- Finite sequences

open import Data.Vec.Recursive
  using (_^_; []; append) public
open import Agda.Builtin.Sigma
  using (Σ)

-- Sequence predomains
-- P ^ n  = P × ... × P  (n ≥ 0)
-- P *    = (P ^ 0) + ... + (P ^ n) + ...
-- (n, p1 , ... , pn) : P *

_* : Set → Set
P * = Σ ℕ (P ^_)

-- #′ S * : ℕ

#′ : {S : Set} → S * → ℕ
#′ (n , _) = n

_::′_ : ∀ {P : Set} → P → P * → P *
p ::′ (0      , ps) = (1 , p)
p ::′ (suc n  , ps) = (suc (suc n) , p , ps)

_↓′_ : ∀ {P : Set} → P * → ℕ → 𝕃 P
(1             , p)      ↓′ 1             = η p
(suc (suc n) , p , ps) ↓′ 1             = η p
(suc (suc n) , p , ps) ↓′ suc (suc i) = (suc n , ps) ↓′ suc i
(_             , _)      ↓′ _             = ⊥

_†′_ : ∀ {P : Set} → P * → ℕ → 𝕃 (P *)
(1             , p)      †′ 1             = η (0 , [])
(suc (suc n) , p , ps) †′ 1             = η (suc n , ps)
(suc (suc n) , p , ps) †′ suc (suc i) = (suc n , ps) †′ suc i
(_             , _)      †′ _             = ⊥

_§′_ : ∀ {P : Set} → P * → P * → P *
(m , pm) §′ (n , pn) = ((m Nat.+ n) , append m n pm pn)

-- Sequence domains
-- D * = 𝕃 ((D ^ 0) + ... + (D ^ n) + ...)

_* : Domain → Domain
D * = 𝕃 (Σ ℕ (D ^_))

-- ⟨⟩ : D *

⟨⟩ : ∀ {D} → D *
⟨⟩ = η (0 , [])

-- ⟨ d1 , ... , dn ⟩ : D *

⟨_⟩ : ∀ {n D} → D ^ suc n → D *
⟨_⟩ {n = n} ds = η (suc n , ds)
```

```
-- # D * : ℕ +⊥

# : ∀ {D} → D * → ℕ +⊥
# d* = ((λ p* → η (#′ p*)) ♯) d*

-- d*₁ § d*₂ : D *

_§_ : ∀ {D} → D * → D * → D *
d*₁ § d*₂ = ((λ p*₁ → ((λ p*₂ → η (p*₁ §′ p*₂)) ♯) d*₂) ♯) d*₁

open import Function
  using (id; _∘_) public

-- d* ↓ k : 𝕃 D  (k ≥ 1; k < # d*)

_↓_ : ∀ {D} → D * → ℕ → D
d* ↓ n = (id ♯) (((λ p* → p* ↓′ n) ♯) d*)

-- d* † k : D *  (k ≥ 1)

_†_ : ∀ {D} → D * → ℕ → D *
d* † n = (id ♯) (((λ p* → η (p* †′ n)) ♯) d*)

------------------------------------------------------------------------
-- McCarthy conditional

-- t ⟶ d₁ , d₂ : D  (t : Bool +⊥ ; d₁, d₂ : D)

open import Data.Bool.Base
  using (Bool; true; false; if_then_else_) public

postulate
  _⟶_,_ : {D : Domain} → Bool +⊥ → D → D → D

  -- Properties
  true-cond   : ∀ {D} {d₁ d₂ : D} → (η true ⟶ d₁ , d₂) ≡ d₁
  false-cond  : ∀ {D} {d₁ d₂ : D} → (η false ⟶ d₁ , d₂) ≡ d₂
  bottom-cond : ∀ {D} {d₁ d₂ : D} → (⊥ ⟶ d₁ , d₂)       ≡ ⊥

------------------------------------------------------------------------
-- Meta-Strings

open import Data.String.Base
  using (String) public
```

```
module Scheme.Abstract-Syntax where

open import Scheme.Domain-Notation using (_*)

-- 7.2.1. Abstract syntax

postulate Con : Set  -- constants, including quotations
postulate Ide  : Set  -- identifiers (variables)
data Exp       : Set  -- expressions
Com            = Exp -- commands

data Exp where
  con                     : Con → Exp                               -- K
  ide                     : Ide → Exp                               -- I
  ⦅_␣_⦆                   : Exp → Exp * → Exp                       -- (E₀ E*)
  ⦅lambda␣⦅_⦆_␣_⦆         : Ide * → Com * → Exp → Exp               -- (lambda (I*) Γ* E₀)
  ⦅lambda␣⦅_·_⦆_␣_⦆       : Ide * → Ide → Com * → Exp → Exp         -- (lambda (I* . I) Γ* E₀)
  ⦅lambda_␣_␣_⦆           : Ide → Com * → Exp → Exp                 -- (lambda I Γ* E₀)
  ⦅if_␣_␣_⦆               : Exp → Exp → Exp → Exp                   -- (if E₀ E₁ E₂)
  ⦅if_␣_⦆                 : Exp → Exp → Exp                         -- (if E₀ E₁)
  ⦅set!_␣_⦆               : Ide → Exp → Exp                         -- (set! I E)

variable
  K  : Con
  I  : Ide
  I* : Ide *
  E  : Exp
  E* : Exp *
  Γ  : Com
  Γ* : Com *
```

```
module Scheme.Domain-Equations where

open import Scheme.Domain-Notation
open import Scheme.Abstract-Syntax
  using (Ide)

-- 7.2.2. Domain equations

-- Domain definitions

postulate Loc :  Set
L              = Loc +⊥      -- locations
N              = ℕ +⊥        -- natural numbers
T              = Bool +⊥     -- booleans
postulate Q    :  Domain     -- symbols
postulate H    :  Domain     -- characters
postulate R    :  Domain     -- numbers
Ep             = (L × L × T) -- pairs
Ev             = (L * × T)   -- vectors
Es             = (L * × T)   -- strings
data Misc      :  Set where  false true null undefined unspecified : Misc
M              = Misc +⊥     -- miscellaneous
X              = String +⊥   -- errors

-- Domain isomorphisms

open import Function
  using (_↔_) public

postulate
  F            :  Domain     -- procedure values
  E            :  Domain     -- expressed values
  S            :  Domain     -- stores
  U            :  Domain     -- environments
  C            :  Domain     -- command continuations
  K            :  Domain     -- expression continuations
  A            :  Domain     -- answers

postulate instance
  iso-F        : F ↔ (L × (E * → K → C))
  iso-E        : E ↔ (𝕃 (Q + H + R + Ep + Ev + Es + M + F))
  iso-S        : S ↔ (L → E × T)
  iso-U        : U ↔ (Ide → L)
  iso-C        : C ↔ (S → A)
  iso-K        : K ↔ (E * → C)

open Function.Inverse {{ ... }}
  renaming (to to ▹ ; from to ◃ ) public
  -- iso-D : D ↔ D′ declares ▹ : D → D′ and ◃ : D′ → D
```

variable

$$\alpha : \mathbf{L}$$
$$\alpha^* : \mathbf{L}^*$$
$$\nu : \mathbf{N}$$
$$\mu : \mathbf{M}$$
$$\phi : \mathbf{F}$$
$$\epsilon : \mathbf{E}$$
$$\epsilon^* : \mathbf{E}^*$$
$$\sigma : \mathbf{S}$$
$$\rho : \mathbf{U}$$
$$\theta : \mathbf{C}$$
$$\kappa : \mathbf{K}$$

pattern

$$\text{inj-}\mathbf{Ep}\ \text{ep} = \text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_1\ \text{ep})))$$

pattern

$$\text{inj-}\mathbf{M}\ \mu = \text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_1\ \mu))))))$$

pattern

$$\text{inj-}\mathbf{F}\ \phi = \text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ (\text{inj}_2\ \phi))))))$$

$$\_ \in\!\mathbf{F} : \mathbf{E} \to \text{Bool} +\!\bot$$
$$\epsilon \in\!\mathbf{F} = ((\lambda \{ (\text{inj-}\mathbf{F}\ \_) \to \eta\ \text{true}\ ;\ \_ \to \eta\ \text{false} \})^{\sharp})\ (\triangleright\ \epsilon)$$

$$\_ |\mathbf{F} : \mathbf{E} \to \mathbf{F}$$
$$\epsilon\ |\mathbf{F} = ((\lambda \{ (\text{inj-}\mathbf{F}\ \phi) \to \phi\ ;\ \_ \to \bot \})^{\sharp})\ (\triangleright\ \epsilon)$$

$$\_ \in\!\mathbf{L} : \mathbb{L}\ (\mathbf{L} + \mathbf{X}) \to \text{Bool} +\!\bot$$
$$\_ \in\!\mathbf{L} = [\ (\lambda\ \_ \to \eta\ \text{true}), (\lambda\ \_ \to \eta\ \text{false})\ ]^{\sharp}$$

$$\_ |\mathbf{L} : \mathbb{L}\ (\mathbf{L} + \mathbf{X}) \to \mathbf{L}$$
$$\_ |\mathbf{L} = [\ \text{id}\ ,\ (\lambda\ \_ \to \bot)\ ]^{\sharp}$$

$$\_\mathbf{Ep}\text{-in-}\mathbf{E} : \mathbf{Ep} \to \mathbf{E}$$
$$\text{ep}\ \mathbf{Ep}\text{-in-}\mathbf{E} = \triangleleft\ (\eta\ (\text{inj-}\mathbf{Ep}\ \text{ep}))$$

$$\_\mathbf{F}\text{-in-}\mathbf{E} : \mathbf{F} \to \mathbf{E}$$
$$\phi\ \mathbf{F}\text{-in-}\mathbf{E} = \triangleleft\ (\eta\ (\text{inj-}\mathbf{F}\ \phi))$$

$$\text{unspecified-in-}\mathbf{E} : \mathbf{E}$$
$$\text{unspecified-in-}\mathbf{E} = \triangleleft\ (\eta\ (\text{inj-}\mathbf{M}\ (\eta\ \text{unspecified})))$$

```
module Scheme.Auxiliary-Functions where

open import Scheme.Domain-Notation
open import Scheme.Domain-Equations
open import Scheme.Abstract-Syntax
  using (Ide)

-- 7.2.4. Auxiliary functions

postulate _==ᴵ_ : Ide → Ide → Bool

_[_/_] : U → L → Ide → U
-- ρ [ α / I ] overrides ρ with the binding of I to α
ρ [ α / I ] = ◂ λ I′ → if I ==ᴵ I′ then α else ▸ ρ I′

lookup : U → Ide → L
lookup = λ ρ I → ▸ ρ I

extends : U → Ide * → L * → U
extends = fix λ extends′ →
  λ ρ I* α* →
    η (#′ I* == 0) ⟶ ρ ,
      ( ( ( (λ I → λ I*′ →
              extends′ (ρ [ (α* ↓ 1) / I ]) I*′ (α* † 1)) ♯)
        (I* ↓′ 1)) ♯) (I* †′ 1)

postulate
  wrong : String → C
  -- wrong : X → C -- implementation-dependent

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

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

postulate
  new : S → 𝕃 (L + X)
-- new : S → (L + {error}) -- implementation-dependent
-- unclear why R5RS uses an undeclared value instead of X

hold : L → K → C
hold = λ α κ → ◂ λ σ → ▸ (send (▸ σ α ↓1) κ) σ

-- assign : L → E → C → C
-- assign = λ α ϵ θ σ → θ (update α ϵ σ)
-- forward reference to update
```

postulate
  $\_==^{\mathrm{L}}\_ : \mathbf{L} \to \mathbf{L} \to \mathbf{T}$

`-- R5RS and [Stoy] explain _[_/_] only in connection with environments`
$\_[\_/\_]' : \mathbf{S} \to (\mathbf{E} \times \mathbf{T}) \to \mathbf{L} \to \mathbf{S}$
$\sigma\ [\ \mathsf{z}\ /\ \alpha\ ]' = \triangleleft\ \lambda\ \alpha' \to (\alpha ==^{\mathrm{L}} \alpha') \longrightarrow \mathsf{z}\ ,\ \triangleright\ \sigma\ \alpha'$

$\mathsf{update} : \mathbf{L} \to \mathbf{E} \to \mathbf{S} \to \mathbf{S}$
$\mathsf{update} = \lambda\ \alpha\ \epsilon\ \sigma \to \sigma\ [\ (\epsilon\ ,\ \eta\ \mathsf{true})\ /\ \alpha\ ]'$

$\mathsf{assign} : \mathbf{L} \to \mathbf{E} \to \mathbf{C} \to \mathbf{C}$
$\mathsf{assign} = \lambda\ \alpha\ \epsilon\ \theta \to \triangleleft\ \lambda\ \sigma \to \triangleright\ \theta\ (\mathsf{update}\ \alpha\ \epsilon\ \sigma)$

$\mathsf{tievals} : (\mathbf{L}^{\ *} \to \mathbf{C}) \to \mathbf{E}^{\ *} \to \mathbf{C}$
$\mathsf{tievals} = \mathsf{fix}\ \lambda\ \mathsf{tievals}' \to$
  $\lambda\ \psi\ \epsilon^* \to \triangleleft\ \lambda\ \sigma \to$
    $(\#\ \epsilon^* ==\!\perp 0) \longrightarrow \triangleright\ (\psi\ \langle\rangle)\ \sigma\ ,$
      $((\mathsf{new}\ \sigma\ \in\!\mathbf{L}) \longrightarrow$
        $\triangleright\ (\mathsf{tievals}'\ (\lambda\ \alpha^* \to \psi\ (\langle\ \mathsf{new}\ \sigma\ |\mathbf{L}\ \rangle\ \S\ \alpha^*))\ (\epsilon^*\ \dagger\ 1))$
          $(\mathsf{update}\ (\mathsf{new}\ \sigma\ |\mathbf{L})\ (\epsilon^* \downarrow 1)\ \sigma)\ ,$
      $\triangleright\ (\mathsf{wrong}\ \text{"out of memory"})\ \sigma\ )$

$\mathsf{list} :\ \mathbf{E}^{\ *} \to \mathbf{K} \to \mathbf{C}$
`-- Add declarations:`
$\mathsf{dropfirst} :\ \mathbf{E}^{\ *} \to \mathbf{N} \to \mathbf{E}^{\ *}$
$\mathsf{takefirst} :\ \mathbf{E}^{\ *} \to \mathbf{N} \to \mathbf{E}^{\ *}$

$\mathsf{tievalsrest} : (\mathbf{L}^{\ *} \to \mathbf{C}) \to \mathbf{E}^{\ *} \to \mathbf{N} \to \mathbf{C}$
$\mathsf{tievalsrest} =$
  $\lambda\ \psi\ \epsilon^*\ \nu \to \mathsf{list}\ (\mathsf{dropfirst}\ \epsilon^*\ \nu)$
    $(\mathsf{single}\ (\lambda\ \epsilon \to \mathsf{tievals}\ \psi\ ((\mathsf{takefirst}\ \epsilon^*\ \nu)\ \S\ \langle\ \epsilon\ \rangle)))$

$\mathsf{dropfirst} = \mathsf{fix}\ \lambda\ \mathsf{dropfirst}' \to$
  $\lambda\ \epsilon^*\ \nu \to$
    $(\nu ==\!\perp 0) \longrightarrow \epsilon^*\ ,$
      $\mathsf{dropfirst}'\ (\epsilon^*\ \dagger\ 1)\ (((\eta \circ \mathsf{pred})^{\ \sharp})\ \nu)$

$\mathsf{takefirst} = \mathsf{fix}\ \lambda\ \mathsf{takefirst}' \to$
  $\lambda\ \epsilon^*\ \nu \to$
    $(\nu ==\!\perp 0) \longrightarrow \langle\rangle\ ,$
      $(\ \langle\ \epsilon^* \downarrow 1\ \rangle\ \S\ (\mathsf{takefirst}'\ (\epsilon^*\ \dagger\ 1)\ (((\eta \circ \mathsf{pred})^{\ \sharp})\ \nu))\ )$

$\mathsf{truish} :\ \mathbf{E} \to \mathbf{T}$
`-- truish = λ ϵ → ϵ = false ⟶ false , true`
$\mathsf{truish} = \lambda\ \epsilon \to (\mathsf{misc\text{-}false}^{\ \sharp})\ (\triangleright\ \epsilon) \longrightarrow (\eta\ \mathsf{false})\ ,\ (\eta\ \mathsf{true})$ where
  $\mathsf{misc\text{-}false} :\ (\mathbf{Q} + \mathbf{H} + \mathbf{R} + \mathbf{Ep} + \mathbf{Ev} + \mathbf{Es} + \mathbf{M} + \mathbf{F}) \to \mathbb{L}\ \mathsf{Bool}$
  $\mathsf{misc\text{-}false}\ (\mathsf{inj\text{-}}\mathbf{M}\ \mu) = ((\lambda\ \{\ \mathsf{false} \to \eta\ \mathsf{true}\ ;\ \_ \to \eta\ \mathsf{false}\ \})^{\ \sharp})\ (\mu)$
  $\mathsf{misc\text{-}false}\ (\mathsf{inj}_1\ \_) \quad = \eta\ \mathsf{false}$
  $\mathsf{misc\text{-}false}\ (\mathsf{inj}_2\ \_) \quad = \eta\ \mathsf{false}$

`-- Added:`

$\mathsf{misc\text{-}undefined} : (\mathbf{Q} + \mathbf{H} + \mathbf{R} + \mathbf{Ep} + \mathbf{Ev} + \mathbf{Es} + \mathbf{M} + \mathbf{F}) \to \mathbb{L}\ \mathsf{Bool}$

$\mathsf{misc\text{-}undefined}\ (\mathsf{inj\text{-}}\mathbf{M}\ \mu) = ((\lambda\ \{\ \mathsf{undefined} \to \eta\ \mathsf{true}\ ;\ \_ \to \eta\ \mathsf{false}\ \})^{\sharp})\ (\mu)$

$\mathsf{misc\text{-}undefined}\ (\mathsf{inj}_1\ \_) = \eta\ \mathsf{false}$

$\mathsf{misc\text{-}undefined}\ (\mathsf{inj}_2\ \_) = \eta\ \mathsf{false}$

```
-- permute    : Exp * → Exp *  -- implementation-dependent
-- unpermute  : E * → E *      -- inverse of permute
```

$\mathsf{applicate} : \mathbf{E} \to \mathbf{E}^{\,*} \to \mathbf{K} \to \mathbf{C}$

$\mathsf{applicate} =$

$\quad \lambda\ \epsilon\ \epsilon^*\ \kappa \to$

$\qquad (\epsilon\ \in\!\mathbf{F}) \longrightarrow (\triangleright\ (\epsilon\ |\mathbf{F}) \downarrow\! 2)\ \epsilon^*\ \kappa\ ,$

$\qquad\quad \mathsf{wrong}$ "bad procedure"

$\mathsf{onearg} : (\mathbf{E} \to \mathbf{K} \to \mathbf{C}) \to (\mathbf{E}^{\,*} \to \mathbf{K} \to \mathbf{C})$

$\mathsf{onearg} =$

$\quad \lambda\ \zeta\ \epsilon^*\ \kappa \to$

$\qquad (\#\ \epsilon^*\ ==_{\perp} 1) \longrightarrow \zeta\ (\epsilon^* \downarrow 1)\ \kappa\ ,$

$\qquad\quad \mathsf{wrong}$ "wrong number of arguments"

$\mathsf{twoarg} : (\mathbf{E} \to \mathbf{E} \to \mathbf{K} \to \mathbf{C}) \to (\mathbf{E}^{\,*} \to \mathbf{K} \to \mathbf{C})$

$\mathsf{twoarg} =$

$\quad \lambda\ \zeta\ \epsilon^*\ \kappa \to$

$\qquad (\#\ \epsilon^*\ ==_{\perp} 2) \longrightarrow \zeta\ (\epsilon^* \downarrow 1)\ (\epsilon^* \downarrow 2)\ \kappa\ ,$

$\qquad\quad \mathsf{wrong}$ "wrong number of arguments"

$\mathsf{cons} : \mathbf{E}^{\,*} \to \mathbf{K} \to \mathbf{C}$

```
-- list : E * → K → C
```

$\mathsf{list} = \mathsf{fix}\ \lambda\ \mathsf{list}' \to$

$\quad \lambda\ \epsilon^*\ \kappa \to$

$\qquad (\#\ \epsilon^*\ ==_{\perp} 0) \longrightarrow \mathsf{send}\ (\triangleleft\ (\eta\ (\mathsf{inj\text{-}}\mathbf{M}\ (\eta\ \mathsf{null}))))\ \kappa\ ,$

$\qquad\quad \mathsf{list}'\ (\epsilon^* \dagger 1)\ (\mathsf{single}\ (\lambda\ \epsilon \to \mathsf{cons}\ \langle\ (\epsilon^* \downarrow 1)\ ,\ \epsilon\ \rangle\ \kappa))$

```
-- cons : E * → K → C
```

$\mathsf{cons} = \mathsf{twoarg}$

$\quad \lambda\ \epsilon_1\ \epsilon_2\ \kappa \to \triangleleft\ \lambda\ \sigma \to$

$\qquad (\mathsf{new}\ \sigma\ \in\!\mathbf{L}) \longrightarrow$

$\qquad\quad (\lambda\ \sigma' \to (\mathsf{new}\ \sigma'\ \in\!\mathbf{L}) \longrightarrow$

$\qquad\qquad\qquad \triangleright\ (\mathsf{send}\ ((\mathsf{new}\ \sigma\ |\mathbf{L}\ ,\ \mathsf{new}\ \sigma'\ |\mathbf{L}\ ,\ (\eta\ \mathsf{true}))\ \mathbf{Ep}\text{-}\mathsf{in}\text{-}\mathbf{E})\ \kappa)$

$\qquad\qquad\qquad\ (\mathsf{update}\ (\mathsf{new}\ \sigma'\ |\mathbf{L})\ \epsilon_2\ \sigma')\ ,$

$\qquad\qquad\qquad \triangleright\ (\mathsf{wrong}$ "out of memory"$)\ \sigma')$

$\qquad\quad (\mathsf{update}\ (\mathsf{new}\ \sigma\ |\mathbf{L})\ \epsilon_1\ \sigma)\ ,$

$\qquad\quad \triangleright\ (\mathsf{wrong}$ "out of memory"$)\ \sigma$

```
{-# OPTIONS --allow-unsolved-metas #-}

module Scheme.Semantic-Functions where

open import Scheme.Domain-Notation
open import Scheme.Abstract-Syntax
open import Scheme.Domain-Equations
open import Scheme.Auxiliary-Functions

-- 7.2.3. Semantic functions

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

-- Definition of 𝒦 deliberately omitted.

ℰ⟦ con K ⟧ = λ ρ κ → send (𝒦⟦ K ⟧) κ

ℰ⟦ ide I ⟧ = λ ρ κ →
  hold (lookup ρ I) (single (λ ϵ →
      (misc-undefined ♯) (▹ ϵ) ⟶ wrong "undefined variable" ,
        send ϵ κ))

-- Non-compositional:
-- ℰ⟦ ⦅ E₀ ␣ E* ⦆ ⟧ =
--   λ ρ κ → ℰ*⟦ permute (⟨ E₀ ⟩ § E* ) ⟧
--              ρ
--              (λ ϵ* → ((λ ϵ* → applicate (ϵ* ↓ 1) (ϵ* † 1) κ)
--                          (unpermute ϵ*)))

ℰ⟦ ⦅ E₀ ␣ E* ⦆ ⟧ = λ ρ κ →
  ℰ⟦ E₀ ⟧ ρ (single (λ ϵ₀ →
      ℰ*⟦ E* ⟧ ρ (◃ λ ϵ* →
        applicate ϵ₀ ϵ* κ)))

ℰ⟦ ⦅lambda␣⦅ I* ⦆ Γ* ␣ E₀ ⦆ ⟧ = λ ρ κ → ◃ λ σ →
      (new σ ∈L) ⟶
        ▹ (send (◃ ( (new σ |L) ,
                    (λ ϵ* κ′ →
                      (# ϵ* ==⊥ #′ I*) ⟶
                          tievals
                            (λ α* → (λ ρ′ → 𝒞*⟦ Γ* ⟧ ρ′ (ℰ⟦ E₀ ⟧ ρ′ κ′))
                                    (extends ρ I* α*))
                            ϵ* ,
                        wrong "wrong number of arguments"
                    )
                  ) F-in-E)
                κ)
          (update (new σ |L) unspecified-in-E σ) ,
        ▹ (wrong "out of memory") σ
```

$$
\begin{array}{l}
\mathcal{E}[\![\ (\!|\,\mathsf{lambda}_{\sqcup}(\!|\ \mathsf{I}^* \cdot \mathsf{I}\ |\!)\ \Gamma^* \sqcup \mathsf{E}_0\ |\!)\ ]\!] = \lambda\, \rho\, \kappa \to \triangleleft\, \lambda\, \sigma \to \\
\qquad (\mathsf{new}\ \sigma \in \mathbf{L}) \longrightarrow \\
\qquad\quad \triangleright\, (\mathsf{send}\ (\triangleleft\ (\ (\mathsf{new}\ \sigma\ |\mathbf{L})\ , \\
\qquad\qquad\qquad\qquad (\lambda\, \epsilon^*\, \kappa' \to \\
\qquad\qquad\qquad\qquad\quad (\#\, \epsilon^* >=\!\perp \#'\ \mathsf{I}^*) \longrightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathsf{tievalsrest} \\
\qquad\qquad\qquad\qquad\qquad\quad (\lambda\, \alpha^* \to (\lambda\, \rho' \to \mathcal{C}^*[\![\ \Gamma^*\ ]\!]\ \rho'\ (\mathcal{E}[\![\ \mathsf{E}_0\ ]\!]\ \rho'\ \kappa')) \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{extends}\ \rho\ (\mathsf{I}^*\ \S'\ (\ 1\ ,\ \mathsf{I}\ ))\ \alpha^*)) \\
\qquad\qquad\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\qquad\qquad\quad (\eta\ (\#'\ \mathsf{I}^*))\ , \\
\qquad\qquad\qquad\qquad\quad \mathsf{wrong}\ \texttt{"too few arguments"} \\
\qquad\qquad\qquad\qquad) \\
\qquad\qquad\qquad\quad)\ \mathbf{F}\text{-in-}\mathbf{E}) \\
\qquad\qquad\qquad \kappa) \\
\qquad\quad (\mathsf{update}\ (\mathsf{new}\ \sigma\ |\mathbf{L})\ \mathsf{unspecified\text{-}in\text{-}}\mathbf{E}\ \sigma)\ , \\
\qquad\quad \triangleright\, (\mathsf{wrong}\ \texttt{"out of memory"})\ \sigma
\end{array}
$$

-- Non-compositional:

-- $\mathcal{E}[\![\ (\!|\,\texttt{lambda I} \sqcup \Gamma* \sqcup \mathsf{E}_0\ |\!)\ ]\!] = \mathcal{E}[\![\ (\!|\,\texttt{lambda}\ (\!|\ \cdot\ \texttt{I}\ |\!)\ \Gamma* \sqcup \mathsf{E}_0\ |\!)\ ]\!]$

$$
\begin{array}{l}
\mathcal{E}[\![\ (\!|\,\mathsf{lambda}\ \mathsf{I} \sqcup \Gamma^* \sqcup \mathsf{E}_0\ |\!)\ ]\!] = \lambda\, \rho\, \kappa \to \triangleleft\, \lambda\, \sigma \to \\
\qquad (\mathsf{new}\ \sigma \in \mathbf{L}) \longrightarrow \\
\qquad\quad \triangleright\, (\mathsf{send}\ (\triangleleft\ (\ (\mathsf{new}\ \sigma\ |\mathbf{L})\ , \\
\qquad\qquad\qquad\qquad (\lambda\, \epsilon^*\, \kappa' \to \\
\qquad\qquad\qquad\qquad\quad \mathsf{tievalsrest} \\
\qquad\qquad\qquad\qquad\quad (\lambda\, \alpha^* \to (\lambda\, \rho' \to \mathcal{C}^*[\![\ \Gamma^*\ ]\!]\ \rho'\ (\mathcal{E}[\![\ \mathsf{E}_0\ ]\!]\ \rho'\ \kappa')) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{extends}\ \rho\ (1\ ,\ \mathsf{I})\ \alpha^*)) \\
\qquad\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\qquad\quad (\eta\ 0)) \\
\qquad\qquad\qquad\quad)\ \mathbf{F}\text{-in-}\mathbf{E}) \\
\qquad\qquad\qquad \kappa) \\
\qquad\quad (\mathsf{update}\ (\mathsf{new}\ \sigma\ |\mathbf{L})\ \mathsf{unspecified\text{-}in\text{-}}\mathbf{E}\ \sigma)\ , \\
\qquad\quad \triangleright\, (\mathsf{wrong}\ \texttt{"out of memory"})\ \sigma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\ (\!|\,\mathsf{if}\ \mathsf{E}_0 \sqcup \mathsf{E}_1 \sqcup \mathsf{E}_2\ |\!)\ ]\!] = \lambda\, \rho\, \kappa \to \\
\quad \mathcal{E}[\![\ \mathsf{E}_0\ ]\!]\ \rho\ (\mathsf{single}\ (\lambda\, \epsilon \to \\
\qquad \mathsf{truish}\ \epsilon \longrightarrow \mathcal{E}[\![\ \mathsf{E}_1\ ]\!]\ \rho\ \kappa\ , \\
\qquad\quad \mathcal{E}[\![\ \mathsf{E}_2\ ]\!]\ \rho\ \kappa))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\ (\!|\,\mathsf{if}\ \mathsf{E}_0 \sqcup \mathsf{E}_1\ |\!)\ ]\!] = \lambda\, \rho\, \kappa \to \\
\quad \mathcal{E}[\![\ \mathsf{E}_0\ ]\!]\ \rho\ (\mathsf{single}\ (\lambda\, \epsilon \to \\
\qquad \mathsf{truish}\ \epsilon \longrightarrow \mathcal{E}[\![\ \mathsf{E}_1\ ]\!]\ \rho\ \kappa\ , \\
\qquad\quad \mathsf{send}\ \mathsf{unspecified\text{-}in\text{-}}\mathbf{E}\ \kappa))
\end{array}
$$

-- Here and elsewhere, any expressed value other than ‘undefined‘
-- may be used in place of ‘unspecified‘.

$\mathcal{E}[\![$ ⦇set! I ␣ E ⦈ $]\!] = \lambda\, \rho\, \kappa \rightarrow$
  $\mathcal{E}[\![$ E $]\!]\, \rho$ (single ($\lambda\, \epsilon \rightarrow$
    assign (lookup ρ I) ϵ (send unspecified-in-**E** κ)))

```
-- ℰ*⟦_⟧  : Exp * → U → K → C
```

$\mathcal{E}*[\![$ 0 , _ $]\!] = \lambda\, \rho\, \kappa \rightarrow \triangleright\, \kappa\, \langle\rangle$

```
-- Cannot split on argument of non-datatype Exp ^ suc n:
-- ℰ*⟦ suc n , E , Es ⟧ = λ ρ κ →
--   ℰ⟦ E ⟧ ρ (single (λ ϵ0 →
--     ℰ*⟦ n , Es ⟧ ρ (◃ λ ϵ* →
--       ▹ κ (⟨ ϵ0 ⟩ § ϵ*))))
```

$\mathcal{E}*[\![$ 1 , E $]\!] = \lambda\, \rho\, \kappa \rightarrow$
  $\mathcal{E}[\![$ E $]\!]\, \rho$ (single ($\lambda\, \epsilon \rightarrow \triangleright\, \kappa\, \langle\, \epsilon\, \rangle$))

$\mathcal{E}*[\![$ suc (suc n) , E , Es $]\!] = \lambda\, \rho\, \kappa \rightarrow$
  $\mathcal{E}[\![$ E $]\!]\, \rho$ (single ($\lambda\, \epsilon_0 \rightarrow$
    $\mathcal{E}*[\![$ suc n , Es $]\!]\, \rho$ ($\triangleleft\, \lambda\, \epsilon^* \rightarrow$
      $\triangleright\, \kappa$ ($\langle\, \epsilon_0\, \rangle$ § ϵ^*))))

```
-- C*⟦_⟧  : Com * → U → C → C
```

$\mathcal{C}*[\![$ 0 , _ $]\!] = \lambda\, \rho\, \theta \rightarrow \theta$

$\mathcal{C}*[\![$ 1 , Γ $]\!] = \lambda\, \rho\, \theta \rightarrow \mathcal{E}[\![\, \Gamma\,]\!]\, \rho$ ($\triangleleft\, \lambda\, \epsilon^* \rightarrow \theta$)

$\mathcal{C}*[\![$ suc (suc n) , Γ , Γs $]\!] = \lambda\, \rho\, \theta \rightarrow$
  $\mathcal{E}[\![\, \Gamma\,]\!]\, \rho$ ($\triangleleft\, \lambda\, \epsilon^* \rightarrow$
    $\mathcal{C}*[\![$ suc n , Γs $]\!]\, \rho\, \theta$)