Denotational Semantics of Scheme R^5 in Agda

DRAFT (March 12, 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 ♯) ⊥ ≡ ⊥

2

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

$\_+\bot : \mathsf{Set} \to \mathsf{Domain}$
$\mathsf{S} +\bot = \mathbb{L}\,\mathsf{S}$

```
-- Lifted operations on ℕ
```

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

```
-- v ==⊥ n : Bool +⊥
```

$\_==\bot\_ : \mathbb{N} +\bot \to \mathbb{N} \to \mathsf{Bool} +\bot$
$v ==\bot\ n = ((\lambda\,m \to \eta\,(m == n))^{\sharp})\,v$

```
-- v >=⊥ n : Bool +⊥
```

$\_>=\bot\_ : \mathbb{N} +\bot \to \mathbb{N} \to \mathsf{Bool} +\bot$
$v >=\bot\ n = ((\lambda\,m \to \eta\,(n < m))^{\sharp})\,v$

```
---------------------------------------------------------------------
-- Products

-- Products of (pre)domains are Cartesian
```

open import Data.Product.Base
  using ($\_\times\_$; $\_,\_$) renaming (proj$_1$ to $\_\downarrow 1$; proj$_2$ to $\_\downarrow 2$) 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 (inj$_1$; inj$_2$) renaming ($\_\uplus\_$ 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)

4

```
-- # D * : ℕ +⊥
```

$\# : \forall \{D\} \to D^* \to \mathbb{N} +\bot$
$\# \, d^* = ((\lambda \, p^* \to \eta \, (\#' \, p^*))^\sharp) \, d^*$

```
-- d*₁ § d*₂ : D *
```

$\_\S\_ : \forall \{D\} \to D^* \to D^* \to D^*$
$d^*_1 \, \S \, d^*_2 = ((\lambda \, p^*_1 \to ((\lambda \, p^*_2 \to \eta \, (p^*_1 \, \S' \, p^*_2))^\sharp) \, d^*_2)^\sharp) \, d^*_1$

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

```
-- d* ↓ k : 𝕃 D   (k ≥ 1; k < # d*)
```

$\_\downarrow\_ : \forall \{D\} \to D^* \to \mathbb{N} \to D$
$d^* \downarrow n = (id^\sharp) \, (((\lambda \, p^* \to p^* \downarrow' n)^\sharp) \, d^*)$

```
-- d* † k : D *   (k ≥ 1)
```

$\_\dagger\_ : \forall \{D\} \to D^* \to \mathbb{N} \to D^*$
$d^* \dagger n = (id^\sharp) \, (((\lambda \, p^* \to \eta \, (p^* \dagger' n))^\sharp) \, d^*)$

```
----------------------------------------------------------------------
-- 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
  $\_\longrightarrow\_,\_ : \{D : Domain\} \to Bool +\bot \to D \to D \to D$

  -- Properties
  true-cond   : $\forall \{D\} \{d_1 \, d_2 : D\} \to (\eta \, true \longrightarrow d_1 , d_2) \equiv d_1$
  false-cond   : $\forall \{D\} \{d_1 \, d_2 : D\} \to (\eta \, false \longrightarrow d_1 , d_2) \equiv d_2$
  bottom-cond : $\forall \{D\} \{d_1 \, d_2 : D\} \to (\bot \longrightarrow d_1 , d_2) \qquad \equiv \bot$

```
----------------------------------------------------------------------
-- 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	-- (E$_0$ E*)
(‖ lambda␣(‖ _ ‖)_␣_ ‖)	: Ide * → Com * → Exp → Exp	-- (lambda (I*) Γ* E$_0$)
(‖ lambda␣(‖ _·_ ‖)_␣_ ‖)	: Ide * → Ide → Com * → Exp → Exp	-- (lambda (I* . I) Γ* E$_0$)
(‖ lambda_␣_␣_ ‖)	: Ide → Com * → Exp → Exp	-- (lambda I Γ* E$_0$)
(‖ if_␣_␣_ ‖)	: Exp → Exp → Exp → Exp	-- (if E$_0$ E$_1$ E$_2$)
(‖ if_␣_ ‖)	: Exp → Exp → Exp	-- (if E$_0$ E$_1$)
(‖ 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 $= \text{Loc} + \bot$ `-- locations`
N $= \mathbb{N} + \bot$ `-- natural numbers`
T $= \text{Bool} + \bot$ `-- booleans`
postulate **Q** : Domain   `-- symbols`
postulate **H** : Domain   `-- characters`
postulate **R** : Domain   `-- numbers`
Ep $= (\textbf{L} \times \textbf{L} \times \textbf{T})$ `-- pairs`
Ev $= (\textbf{L}^* \times \textbf{T})$ `-- vectors`
Es $= (\textbf{L}^* \times \textbf{T})$ `-- strings`
data Misc   : Set where   false true null undefined unspecified : Misc
M $= \text{Misc} + \bot$ `-- miscellaneous`
X $= \text{String} + \bot$ `-- 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**   : $\textbf{F} \leftrightarrow (\textbf{L} \times (\textbf{E}^* \to \textbf{K} \to \textbf{C}))$
  iso-**E**   : $\textbf{E} \leftrightarrow (\mathbb{L}\,(\textbf{Q} + \textbf{H} + \textbf{R} + \textbf{Ep} + \textbf{Ev} + \textbf{Es} + \textbf{M} + \textbf{F}))$
  iso-**S**   : $\textbf{S} \leftrightarrow (\textbf{L} \to \textbf{E} \times \textbf{T})$
  iso-**U**   : $\textbf{U} \leftrightarrow (\text{Ide} \to \textbf{L})$
  iso-**C**   : $\textbf{C} \leftrightarrow (\textbf{S} \to \textbf{A})$
  iso-**K**   : $\textbf{K} \leftrightarrow (\textbf{E}^* \to \textbf{C})$

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

variable
  α : **L**
  α^* : **L** *
  ν : **N**
  μ : **M**
  ϕ : **F**
  ϵ : **E**
  ϵ^* : **E** *
  σ : **S**
  ρ : **U**
  θ : **C**
  κ : **K**

pattern
  inj-**Ep** ep = inj$_2$ (inj$_2$ (inj$_2$ (inj$_1$ ep)))
pattern
  inj-**M** μ = inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ (inj$_1$ μ))))))
pattern
  inj-**F** ϕ = inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ (inj$_2$ ϕ))))))

_$\in$**F**　　　 : **E** $\to$ Bool $+\bot$
ϵ $\in$**F**　　　 = $((\lambda \{$ (inj-**F** _) $\to \eta$ true ; _ $\to \eta$ false $\})$ $^\sharp)$ $(\rhd \epsilon)$

_|**F**　　　 : **E** $\to$ **F**
ϵ |**F**　　　 = $((\lambda \{$ (inj-**F** $\phi) \to \phi$; _ $\to \bot \})$ $^\sharp)$ $(\rhd \epsilon)$

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

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

_**Ep**-in-**E**　　 : **Ep** $\to$ **E**
ep **Ep**-in-**E**　　 = $\lhd (\eta$ (inj-**Ep** ep))

_**F**-in-**E**　　 : **F** $\to$ **E**
ϕ **F**-in-**E**　　 = $\lhd (\eta$ (inj-**F** ϕ))

unspecified-in-**E** : **E**
unspecified-in-**E** = $\lhd (\eta$ (inj-**M** (η 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)) $^\sharp$)
        (I* ↓′ 1)) $^\sharp$) (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
  $\_ ==^L \_ : \mathbf{L} \to \mathbf{L} \to \mathbf{T}$

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

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

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

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

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

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

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

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

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

```
-- Added:
```
misc-undefined : $(\mathbf{Q} + \mathbf{H} + \mathbf{R} + \mathbf{Ep} + \mathbf{Ev} + \mathbf{Es} + \mathbf{M} + \mathbf{F}) \to \mathbb{L} \text{ Bool}$
misc-undefined $(\text{inj-}\mathbf{M} \; \mu) = ((\lambda \{ \text{ undefined} \to \eta \text{ true} \; ; \; \_ \to \eta \text{ false} \})^{\sharp}) (\mu)$
misc-undefined $(\text{inj}_1 \; \_) = \eta \text{ false}$
misc-undefined $(\text{inj}_2 \; \_) = \eta \text{ false}$

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

applicate : $\mathbf{E} \to \mathbf{E}^* \to \mathbf{K} \to \mathbf{C}$
applicate $=$
 $\lambda \; \epsilon \; \epsilon^* \; \kappa \to$
  $(\epsilon \in \mathbf{F}) \longrightarrow (\triangleright (\epsilon \,|\mathbf{F}) {\downarrow} 2) \; \epsilon^* \; \kappa \, ,$
   wrong "bad procedure"

onearg : $(\mathbf{E} \to \mathbf{K} \to \mathbf{C}) \to (\mathbf{E}^* \to \mathbf{K} \to \mathbf{C})$
onearg $=$
 $\lambda \; \zeta \; \epsilon^* \; \kappa \to$
  $(\# \; \epsilon^* == \bot \; 1) \longrightarrow \zeta \; (\epsilon^* {\downarrow} 1) \; \kappa \, ,$
   wrong "wrong number of arguments"

twoarg : $(\mathbf{E} \to \mathbf{E} \to \mathbf{K} \to \mathbf{C}) \to (\mathbf{E}^* \to \mathbf{K} \to \mathbf{C})$
twoarg $=$
 $\lambda \; \zeta \; \epsilon^* \; \kappa \to$
  $(\# \; \epsilon^* == \bot \; 2) \longrightarrow \zeta \; (\epsilon^* {\downarrow} 1) \; (\epsilon^* {\downarrow} 2) \; \kappa \, ,$
   wrong "wrong number of arguments"

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

```
-- list : E * ↛ K ↛ C
```
list $= \text{fix} \; \lambda \; \text{list}' \to$
 $\lambda \; \epsilon^* \; \kappa \to$
  $(\# \; \epsilon^* == \bot \; 0) \longrightarrow \text{send} \; (\triangleleft (\eta \; (\text{inj-}\mathbf{M} \; (\eta \; \text{null})))) \; \kappa \, ,$
   $\text{list}' \; (\epsilon^* \dagger 1) \; (\text{single} \; (\lambda \; \epsilon \to \text{cons} \; \langle \; (\epsilon^* {\downarrow} 1) \, , \; \epsilon \; \rangle \; \kappa))$

```
-- cons : E * ↛ K ↛ C
```
cons $=$ twoarg
 $\lambda \; \epsilon_1 \; \epsilon_2 \; \kappa \to \triangleleft \lambda \; \sigma \to$
  $(\text{new} \; \sigma \in \mathbf{L}) \longrightarrow$
    $(\lambda \; \sigma' \to (\text{new} \; \sigma' \in \mathbf{L}) \longrightarrow$
               $\triangleright (\text{send} \; ((\text{new} \; \sigma \,|\mathbf{L} \, , \; \text{new} \; \sigma' \,|\mathbf{L} \, , \; (\eta \; \text{true})) \; \mathbf{Ep}\text{-in-}\mathbf{E}) \; \kappa)$
               $(\text{update} \; (\text{new} \; \sigma' \,|\mathbf{L}) \; \epsilon_2 \; \sigma') \, ,$
               $\triangleright (\text{wrong} \; "\text{out of memory}") \; \sigma')$
         $(\text{update} \; (\text{new} \; \sigma \,|\mathbf{L}) \; \epsilon_1 \; \sigma) \, ,$
     $\triangleright (\text{wrong} \; "\text{out of memory}") \; \sigma$

```
{-# 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 $\mathcal{K}[\![\_]\!]$: Con $\rightarrow$ **E**
$\mathcal{E}[\![\_]\!]$: Exp $\rightarrow$ **U** $\rightarrow$ **K** $\rightarrow$ **C**
$\mathcal{E}*[\![\_]\!]$: Exp $*$ $\rightarrow$ **U** $\rightarrow$ **K** $\rightarrow$ **C**
$\mathcal{C}*[\![\_]\!]$: Com $*$ $\rightarrow$ **U** $\rightarrow$ **C** $\rightarrow$ **C**

```
-- Definition of 𝒦 deliberately omitted.
```

$\mathcal{E}[\![$ con K $]\!]$ $= \lambda \, \rho \, \kappa \rightarrow$ send $(\mathcal{K}[\![$ K $]\!]) \, \kappa$

$\mathcal{E}[\![$ ide I $]\!]$ $= \lambda \, \rho \, \kappa \rightarrow$
  hold (lookup ρ I) (single ($\lambda \, \epsilon \rightarrow$
    (misc-undefined $^\sharp$) ($\triangleright \epsilon$) $\longrightarrow$ wrong "undefined variable" ,
     send $\epsilon \, \kappa$))

```
-- 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*)))
```

$\mathcal{E}[\![$ (E$*$) $]\!]$ $= \lambda \, \rho \, \kappa \rightarrow$
  $\mathcal{E}*[\![$ E$*$ $]\!]$ ρ ($\triangleleft \lambda \, \epsilon^* \rightarrow$
    applicate ($\epsilon^* \downarrow 1$) ($\epsilon^* \dagger 1$) κ)

$\mathcal{E}[\![$ (lambda$_\sqcup$(I$*$) $\Gamma^* \sqcup$ E$_0$) $]\!]$ $= \lambda \, \rho \, \kappa \rightarrow \triangleleft \lambda \, \sigma \rightarrow$
    (new $\sigma \in$**L**) $\longrightarrow$
      $\triangleright$ (send ($\triangleleft$ ((new σ |**L**) ,
            ($\lambda \, \epsilon^* \, \kappa' \rightarrow$
              ($\#\, \epsilon^* ==\!\perp \#'$ I$*$) $\longrightarrow$
                 tievals
                   ($\lambda \, \alpha^* \rightarrow (\lambda \, \rho' \rightarrow \mathcal{C}*[\![$ Γ^* $]\!]$ ρ' $(\mathcal{E}[\![$ E$_0$ $]\!]$ $\rho' \, \kappa'$))
                     (extends ρ I$*$ α^*))
                ϵ^* ,
              wrong "wrong number of arguments"
)
) **F**-in-**E**)
        κ)
      (update (new σ |**L**) unspecified-in-**E** σ) ,
     $\triangleright$ (wrong "out of memory") σ

$\mathcal{E}[\![\ (\!|\text{lambda}_\sqcup (\!| \ \text{I}^* \cdot \text{I} \ |\!) \ \Gamma^* \ _\sqcup \ \text{E}_0 \ |\!) \ ]\!] = \lambda \ \rho \ \kappa \rightarrow \triangleleft \lambda \ \sigma \rightarrow$

     $(\text{new} \ \sigma \in \mathbf{L}) \longrightarrow$

       $\triangleright (\text{send} \ (\triangleleft \ (\ (\text{new} \ \sigma \ | \mathbf{L}) \ ,$

                 $(\lambda \ \epsilon^* \ \kappa' \rightarrow$

                     $(\# \ \epsilon^* >= \perp \#' \ \text{I}^*) \longrightarrow$

                         tievalsrest

                           $(\lambda \ \alpha^* \rightarrow (\lambda \ \rho' \rightarrow C^*[\![\ \Gamma^* \ ]\!] \ \rho' \ (\mathcal{E}[\![\ \text{E}_0 \ ]\!] \ \rho' \ \kappa'))$

                           $(\text{extends} \ \rho \ (\text{I}^* \ \S' \ (\ 1 \ , \text{I} \ )) \ \alpha^*))$

                           ϵ^*

                           $(\eta \ (\#' \ \text{I}^*)) \ ,$

                     wrong "too few arguments"

                 $)$

               $) \ \mathbf{F}\text{-in-}\mathbf{E})$

           $\kappa)$

       $(\text{update} \ (\text{new} \ \sigma \ | \mathbf{L}) \ \text{unspecified-in-}\mathbf{E} \ \sigma) \ ,$

       $\triangleright (\text{wrong} \ \text{"out of memory"}) \ \sigma$

-- Non-compositional:
-- $\mathcal{E}[\![\ (\!|\text{lambda} \ \text{I} \ _\sqcup \ \Gamma^* \ _\sqcup \ \text{E}_0 \ |\!) \ ]\!] = \mathcal{E}[\![\ (\!|\text{lambda} \ (\!| \ \cdot \ \text{I} \ |\!) \ \Gamma^* \ _\sqcup \ \text{E}_0 \ |\!) \ ]\!]$

$\mathcal{E}[\![\ (\!|\text{lambda} \ \text{I} \ _\sqcup \ \Gamma^* \ _\sqcup \ \text{E}_0 \ |\!) \ ]\!] = \lambda \ \rho \ \kappa \rightarrow \triangleleft \lambda \ \sigma \rightarrow$

     $(\text{new} \ \sigma \in \mathbf{L}) \longrightarrow$

       $\triangleright (\text{send} \ (\triangleleft \ (\ (\text{new} \ \sigma \ | \mathbf{L}) \ ,$

                 $(\lambda \ \epsilon^* \ \kappa' \rightarrow$

                     tievalsrest

                       $(\lambda \ \alpha^* \rightarrow (\lambda \ \rho' \rightarrow C^*[\![\ \Gamma^* \ ]\!] \ \rho' \ (\mathcal{E}[\![\ \text{E}_0 \ ]\!] \ \rho' \ \kappa'))$

                           $(\text{extends} \ \rho \ (1 \ , \text{I}) \ \alpha^*))$

                     ϵ^*

                     $(\eta \ 0))$

               $) \ \mathbf{F}\text{-in-}\mathbf{E})$

           $\kappa)$

       $(\text{update} \ (\text{new} \ \sigma \ | \mathbf{L}) \ \text{unspecified-in-}\mathbf{E} \ \sigma) \ ,$

       $\triangleright (\text{wrong} \ \text{"out of memory"}) \ \sigma$

$\mathcal{E}[\![\ (\!|\text{if} \ \text{E}_0 \ _\sqcup \ \text{E}_1 \ _\sqcup \ \text{E}_2 \ |\!) \ ]\!] = \lambda \ \rho \ \kappa \rightarrow$

   $\mathcal{E}[\![\ \text{E}_0 \ ]\!] \ \rho \ (\text{single} \ (\lambda \ \epsilon \rightarrow$

     $\text{truish} \ \epsilon \longrightarrow \mathcal{E}[\![\ \text{E}_1 \ ]\!] \ \rho \ \kappa \ ,$

     $\mathcal{E}[\![\ \text{E}_2 \ ]\!] \ \rho \ \kappa))$

$\mathcal{E}[\![\ (\!|\text{if} \ \text{E}_0 \ _\sqcup \ \text{E}_1 \ |\!) \ ]\!] = \lambda \ \rho \ \kappa \rightarrow$

   $\mathcal{E}[\![\ \text{E}_0 \ ]\!] \ \rho \ (\text{single} \ (\lambda \ \epsilon \rightarrow$

     $\text{truish} \ \epsilon \longrightarrow \mathcal{E}[\![\ \text{E}_1 \ ]\!] \ \rho \ \kappa \ ,$

     $\text{send} \ \text{unspecified-in-}\mathbf{E} \ \kappa))$

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

$\mathcal{E}[\![\;(\!|\text{set! } I \sqcup E \;)\!] \;]\!] = \lambda\, \rho\, \kappa \to$
  $\mathcal{E}[\![\; E \;]\!]\; \rho\; (\text{single } (\lambda\, \epsilon \to$
    $\text{assign } (\text{lookup } \rho\; I)\; \epsilon\; (\text{send unspecified-in-}\mathbf{E}\; \kappa)))$

$--\;\; \mathcal{E}*[\![\_]\!] \quad : \; \text{Exp} * \to \text{U} \to \text{K} \to \text{C}$

$\mathcal{E}*[\![\; 0 \;,\; \_ \;]\!] = \lambda\, \rho\, \kappa \to \rhd\, \kappa\; \langle\rangle$

$\mathcal{E}*[\![\; 1 \;,\; E \;]\!] = \mathcal{E}[\![\; E \;]\!]$

$\mathcal{E}*[\![\; \text{suc (suc n)} \;,\; E \;,\; Es \;]\!] = \lambda\, \rho\, \kappa \to$
  $\mathcal{E}[\![\; E \;]\!]\; \rho\; (\text{single } (\lambda\, \epsilon_0 \to$
    $\mathcal{E}*[\![\; \text{suc n} \;,\; Es \;]\!]\; \rho\; (\lhd\, \lambda\, \epsilon^* \to$
      $\rhd\, \kappa\; (\langle\, \epsilon_0 \,\rangle\; \S\; \epsilon^*))))$

$--\;\; C*[\![\_]\!] \quad : \; \text{Com} * \to \text{U} \to \text{C} \to \text{C}$

$C*[\![\; 0 \;,\; \_ \;]\!] = \lambda\, \rho\, \theta \to \theta$

$C*[\![\; 1 \;,\; \Gamma \;]\!] = \lambda\, \rho\, \theta \to \mathcal{E}[\![\; \Gamma \;]\!]\; \rho\; (\lhd\, \lambda\, \epsilon^* \to \theta)$

$C*[\![\; \text{suc (suc n)} \;,\; \Gamma \;,\; \Gamma s \;]\!] = \lambda\, \rho\, \theta \to$
  $\mathcal{E}[\![\; \Gamma \;]\!]\; \rho\; (\lhd\, \lambda\, \epsilon^* \to$
    $C*[\![\; \text{suc n} \;,\; \Gamma s \;]\!]\; \rho\; \theta)$