

# Denotational Semantics of Scheme $R^5$ in Agda

DRAFT (March 10, 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 +⊥ =  $\mathbb{L}$  S

-- Lifted operations on  $\mathbb{N}$ 

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

--  $\nu ==_{\perp} n : \text{Bool } +_{\perp}$ 

_==⊥_ :  $\mathbb{N} +_{\perp} \rightarrow \mathbb{N} \rightarrow \text{Bool } +_{\perp}$ 
 $\nu ==_{\perp} n = ((\lambda m \rightarrow \eta (m == n)) \#) \nu$ 

--  $\nu >_{\perp} n : \text{Bool } +_{\perp}$ 

_>⊥_ :  $\mathbb{N} +_{\perp} \rightarrow \mathbb{N} \rightarrow \text{Bool } +_{\perp}$ 
 $\nu >_{\perp} n = ((\lambda m \rightarrow \eta (n < m)) \#) \nu$ 

-----
-- 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 -- (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 (Inverse; _↔_) 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 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

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

pattern

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

$\_ \in \mathbf{F} : \mathbf{E} \rightarrow \mathbf{Bool} + \perp$

$\epsilon \in \mathbf{F} = ((\lambda \{ (\text{inj-}\mathbf{F} \_) \rightarrow \eta \text{ true} ; \_ \rightarrow \eta \text{ false} \})^\#) (\triangleright \epsilon)$

$\_ | \mathbf{F} : \mathbf{E} \rightarrow \mathbf{F}$

$\epsilon | \mathbf{F} = ((\lambda \{ (\text{inj-}\mathbf{F} \phi) \rightarrow \phi ; \_ \rightarrow \perp \})^\#) (\triangleright \epsilon)$

$\_ \in \mathbf{L} : \mathbb{L} (\mathbf{L} + \mathbf{X}) \rightarrow \mathbf{Bool} + \perp$

$\_ \in \mathbf{L} = [ (\lambda \_ \rightarrow \eta \text{ true}), (\lambda \_ \rightarrow \eta \text{ false}) ]^\#$

$\_ | \mathbf{L} : \mathbb{L} (\mathbf{L} + \mathbf{X}) \rightarrow \mathbf{L}$

$\_ | \mathbf{L} = [ \text{id}, (\lambda \_ \rightarrow \perp) ]^\#$

$\_ \mathbf{Ep-in-E} : \mathbf{Ep} \rightarrow \mathbf{E}$

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

$\_ \mathbf{F-in-E} : \mathbf{F} \rightarrow \mathbf{E}$

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

$\text{unspecified-in-}\mathbf{E} : \mathbf{E}$

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



```

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
  _==L_ : L → L → T

-- R5RS and [Stoy] explain [_/_] only in connection with environments
_[_/_]': S → (E × T) → L → S
σ [z / α]' = ◁ λ α' → (α ==L α') → ⊥ z , ▷ σ α'

update : L → E → S → S
update = λ α ε σ → σ [ (ε , η true) / α ]'

assign : L → E → C → C
assign = λ α ε θ → ◁ λ σ → ▷ θ (update α ε σ)

tievals : (L* → C) → E* → C
tievals = fix λ tievals' →
  λ ψ ε* → ◁ λ σ →
    (# ε* == ⊥ 0) → ⊥ ▷ (ψ ⟨⟩) σ ,
    ((new σ ∈ L) → ⊥
     ▷ (tievals' (λ α* → ψ (⟨ new σ | L ⟩ § α*)) (ε* † 1))
     (update (new σ | L) (ε* ↓ 1) σ) ,
     ▷ (wrong "out of memory" σ)

list : E* → K → C
-- Add declarations:
dropfirst : E* → N → E*
takefirst : E* → N → E*

tievalsrest : (L* → C) → E* → N → C
tievalsrest =
  λ ψ ε* ν → list (dropfirst ε* ν)
    (single (λ ε → tievals ψ ((takefirst ε* ν) § ⟨ ε ⟩)))

dropfirst = fix λ dropfirst' →
  λ ε* ν →
    (ν == ⊥ 0) → ⊥ ε* ,
    dropfirst' (ε* † 1) (((η ∘ pred) #) ν)

takefirst = fix λ takefirst' →
  λ ε* ν →
    (ν == ⊥ 0) → ⊥ ⟨⟩ ,
    (⟨ ε* ↓ 1 ⟩ § (takefirst' (ε* † 1) (((η ∘ pred) #) ν)))

truish : E → T
-- truish = λ ε → ε = false → ⊥ false , true
truish = λ ε → (misc-false #) (▷ ε) → ⊥ (η false) , (η true) where
  misc-false : (Q + H + R + Ep + Ev + Es + M + F) → ℒ Bool
  misc-false (inj-M μ) = ((λ { false → η true ; _ → η false } ) #) (μ)
  misc-false (inj1 _) = η false
  misc-false (inj2 _) = η false

```

```

-- Added:
misc-undefined : (Q + H + R + Ep + Ev + Es + M + F) → ℒ Bool
misc-undefined (inj-M μ) = ((λ { undefined → η true ; _ → η false })#) (μ)
misc-undefined (inj1 _) = η false
misc-undefined (inj2 _) = η false

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

applicate : E → E * → K → C
applicate =
  λ ε ε* κ →
    (ε ∈ F) → ⊥ (▷ (ε | F) ↓ 2) ε* κ ,
    wrong "bad procedure"

onearg : (E → K → C) → (E * → K → C)
onearg =
  λ ζ ε* κ →
    (# ε* == ⊥ 1) → ⊥ ζ (ε* ↓ 1) κ ,
    wrong "wrong number of arguments"

twoarg : (E → E → K → C) → (E * → K → C)
twoarg =
  λ ζ ε* κ →
    (# ε* == ⊥ 2) → ⊥ ζ (ε* ↓ 1) (ε* ↓ 2) κ ,
    wrong "wrong number of arguments"

cons : E * → K → C

-- list : E * → K → C
list = fix λ list' →
  λ ε* κ →
    (# ε* == ⊥ 0) → ⊥ send (◁ (η (inj-M (η null)))) κ ,
    list' (ε* ↑ 1) (single (λ ε → cons (ε* ↓ 1), ε) κ))

-- cons : E * → K → C
cons = twoarg
  λ ε1 ε2 κ → ◁ λ σ →
    (new σ ∈ L) → ⊥
    (λ σ' → (new σ' ∈ L) → ⊥
      ▷ (send ((new σ | L, new σ' | L, (η true)) Ep-in-E) κ)
      (update (new σ' | L) ε2 σ') ,
      ▷ (wrong "out of memory") σ')
    (update (new σ | L) ε1 σ) ,
    ▷ (wrong "out of memory") σ

```

```

{-# 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  $\mathcal{K}$  deliberately omitted.

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

 $\mathcal{E}[\text{ide } l] = \lambda \rho \kappa \rightarrow$ 
  hold (lookup  $\rho$  l) (single ( $\lambda \epsilon \rightarrow$ 
    (misc-undefined #) ( $\triangleright \epsilon$ )  $\rightarrow \perp$  wrong "undefined variable" ,
    send  $\epsilon \kappa$ ))

-- Non-compositional:
--  $\mathcal{E}[\langle E_0 \sqcup E^* \rangle] =$ 
--    $\lambda \rho \kappa \rightarrow \mathcal{E}^*[\text{permute } (\langle E_0 \rangle \S E^*)]$ 
--    $\rho$ 
--   ( $\lambda \epsilon^* \rightarrow ((\lambda \epsilon^* \rightarrow \text{applicate } (\epsilon^* \downarrow 1) (\epsilon^* \uparrow 1) \kappa)$ 
--     (unpermute  $\epsilon^*$ )))

 $\mathcal{E}[\langle E^* \rangle] = \lambda \rho \kappa \rightarrow$ 
   $\mathcal{E}^*[\langle E^* \rangle] \rho (\triangleleft \lambda \epsilon^* \rightarrow$ 
    applicate ( $\epsilon^* \downarrow 1$ ) ( $\epsilon^* \uparrow 1$ )  $\kappa$ )

 $\mathcal{E}[\langle \text{lambda}_{\sqcup} \langle l^* \rangle \Gamma^* \sqcup E_0 \rangle] = \lambda \rho \kappa \rightarrow \triangleleft \lambda \sigma \rightarrow$ 
  (new  $\sigma \in \mathbf{L}$ )  $\rightarrow \perp$ 
   $\triangleright$  (send ( $\triangleleft$  (new  $\sigma \mid \mathbf{L}$ ) ,
    ( $\lambda \epsilon^* \kappa' \rightarrow$ 
      ( $\# \epsilon^* == \perp \# l^*$ )  $\rightarrow \perp$ 
      tievals
      ( $\lambda \alpha^* \rightarrow (\lambda \rho' \rightarrow \mathcal{C}^*[\Gamma^*] \rho' (\mathcal{E}[E_0] \rho' \kappa'))$ 
        (extends  $\rho \mid \alpha^*$ ))
       $\epsilon^*$  ,
      wrong "wrong number of arguments"
    )
  )  $\mathbf{F}$ -in-E)
   $\kappa$ )
  (update (new  $\sigma \mid \mathbf{L}$ ) unspecified-in-E  $\sigma$ ) ,
   $\triangleright$  (wrong "out of memory")  $\sigma$ 

```

```

 $\mathcal{E}[\![ \text{lambda}_{\sqcup} (l^* \cdot l) \Gamma^* \sqcup E_0 ]\!] = \lambda \rho \kappa \rightarrow \triangleleft \lambda \sigma \rightarrow$ 
  (new  $\sigma \in \mathbf{L}$ )  $\longrightarrow \perp$ 
   $\triangleright$  (send ( $\triangleleft$  (new  $\sigma \mid \mathbf{L}$ ),
    ( $\lambda \epsilon^* \kappa' \rightarrow$ 
      ( $\# \epsilon^* \geq \perp \# l^*$ )  $\longrightarrow \perp$ 
      tievalsrest
      ( $\lambda \alpha^* \rightarrow (\lambda \rho' \rightarrow C^*[\![ \Gamma^* ]\!] \rho' (\mathcal{E}[\![ E_0 ]\!] \rho' \kappa'))$ 
      (extends  $\rho (l^* \S' (1, l)) \alpha^*$ )
       $\epsilon^*$ 
      ( $\eta (\# l^*)$ ),
      wrong "too few arguments"
    )
  )  $\mathbf{F-in-E}$ 
   $\kappa$ )
  (update (new  $\sigma \mid \mathbf{L}$ ) unspecified-in- $\mathbf{E} \sigma$ ),
   $\triangleright$  (wrong "out of memory")  $\sigma$ 

-- Non-compositional:
--  $\mathcal{E}[\![ \text{lambda } l \sqcup \Gamma^* \sqcup E_0 ]\!] = \mathcal{E}[\![ \text{lambda } ( \cdot \cdot l ) \Gamma^* \sqcup E_0 ]\!]$ 

 $\mathcal{E}[\![ \text{lambda } l \sqcup \Gamma^* \sqcup E_0 ]\!] = \lambda \rho \kappa \rightarrow \triangleleft \lambda \sigma \rightarrow$ 
  (new  $\sigma \in \mathbf{L}$ )  $\longrightarrow \perp$ 
   $\triangleright$  (send ( $\triangleleft$  (new  $\sigma \mid \mathbf{L}$ ),
    ( $\lambda \epsilon^* \kappa' \rightarrow$ 
      tievalsrest
      ( $\lambda \alpha^* \rightarrow (\lambda \rho' \rightarrow C^*[\![ \Gamma^* ]\!] \rho' (\mathcal{E}[\![ E_0 ]\!] \rho' \kappa'))$ 
      (extends  $\rho (1, l) \alpha^*$ )
       $\epsilon^*$ 
      ( $\eta 0$ ))
    )  $\mathbf{F-in-E}$ 
     $\kappa$ )
  (update (new  $\sigma \mid \mathbf{L}$ ) unspecified-in- $\mathbf{E} \sigma$ ),
   $\triangleright$  (wrong "out of memory")  $\sigma$ 

 $\mathcal{E}[\![ \text{if } E_0 \sqcup E_1 \sqcup E_2 ]\!] = \lambda \rho \kappa \rightarrow$ 
   $\mathcal{E}[\![ E_0 ]\!] \rho$  (single ( $\lambda \epsilon \rightarrow$ 
    truish  $\epsilon \longrightarrow \perp \mathcal{E}[\![ E_1 ]\!] \rho \kappa$ ,
     $\mathcal{E}[\![ E_2 ]\!] \rho \kappa$ ))

 $\mathcal{E}[\![ \text{if } E_0 \sqcup E_1 ]\!] = \lambda \rho \kappa \rightarrow$ 
   $\mathcal{E}[\![ E_0 ]\!] \rho$  (single ( $\lambda \epsilon \rightarrow$ 
    truish  $\epsilon \longrightarrow \perp \mathcal{E}[\![ E_1 ]\!] \rho \kappa$ ,
    send unspecified-in- $\mathbf{E} \kappa$ ))

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

```

```

 $\mathcal{E}[\![ \text{set! } l \sqcup E ]\!] = \lambda \rho \kappa \rightarrow$ 
 $\mathcal{E}[\![ E ]\!] \rho (\text{single } (\lambda \epsilon \rightarrow$ 
 $\text{assign } (\text{lookup } \rho l) \epsilon (\text{send unspecified-in-} E \kappa)))$ 

--  $\mathcal{E}^*[\![\_]\!] : \text{Exp} * \rightarrow \mathbf{U} \rightarrow \mathbf{K} \rightarrow \mathbf{C}$ 

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

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

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

--  $C^*[\![\_]\!] : \text{Com} * \rightarrow \mathbf{U} \rightarrow \mathbf{C} \rightarrow \mathbf{C}$ 

 $C^*[\![ 0, \_ ]\!] = \lambda \rho \theta \rightarrow \theta$ 

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

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

```