Scheme.All

April 13, 2025

```
{-  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 Q : 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} → P → 𝕃 P
  _♯        : ∀ {P} {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
  using (ℕ; suc; NonZero; 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 (_^_; []) 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 *′

_*′ : Predomain → Predomain
P *′ = Σ ℕ (P ^_)

-- #′ P *′ : ℕ

#′ : ∀ {P} → P *′ → ℕ
#′ (n , _) = n

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

_↓′_ : ∀ {P} → P *′ → (n : ℕ) → .{{_ : NonZero 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} → P *′ → (n : ℕ) → .{{_ : NonZero 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} → P *′ → P *′ → P *′
(0              , _)      §′ p*′ = p*′
(1              , p)      §′ p*′ = p ::′ p*′
(suc (suc n) , p , ps) §′ p*′ = p ::′ ((suc n , ps) §′ p*′)

-- 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 : D  (k ≥ 1; k < # d*)

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

-- d* † k : D *  (k ≥ 1)

_†_ : ∀ {D} → D * → (n : ℕ) → .{{_ : NonZero 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
  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 = inj₂ (inj₂ (inj₂ (inj₁ ep)))
pattern
  inj-M μ   = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ μ))))))
pattern
  inj-F ϕ   = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ ϕ))))))

_∈F       : E → Bool +⊥
ϵ ∈F      = ((λ { (inj-F _) → η true ; _ → η false }) ♯) (▹ ϵ)

_|F       : E → F
ϵ |F      = ((λ { (inj-F ϕ) → ϕ ; _ → ⊥ }) ♯) (▹ ϵ)

_∈L       : 𝕃 (L + X) → Bool +⊥
_∈L       = [ (λ _ → η true), (λ _ → η false) ] ♯

_|L       : 𝕃 (L + X) → L
_|L       = [ id , (λ _ → ⊥) ] ♯

_Ep-in-E     : Ep → E
ep Ep-in-E   = ◃ (η (inj-Ep ep))

_F-in-E      : F → E
ϕ F-in-E     = ◃ (η (inj-F ϕ))

unspecified-in-E : E
unspecified-in-E = ◃ (η (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)

open import Data.Nat.Base
  using (NonZero; pred) public

-- 7.2.4. Auxiliary functions

postulate _==ᴵ_ : Ide → Ide → Bool

_[_/_] : U → L → Ide → U
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

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 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
```

```
ℰ⟦ ⦇lambda␣⦇ I* · I ⦈ Γ* ␣ E₀ ⦈ ⟧ = λ ρ κ → ◃ λ σ →
      (new σ ∈L) ⟶
        ▹ (send (◃ ( (new σ |L) ,
                     (λ ϵ* κ′ →
                        (# ϵ* >=⊥ #′ I*) ⟶
                          tievalsrest
                            (λ α* → (λ ρ′ → C*⟦ Γ* ⟧ ρ′ (ℰ⟦ E₀ ⟧ ρ′ κ′))
                            (extends ρ (I* §′ ( 1 , I )) α*))
                            ϵ*
                            (η (#′ I*)) ,
                        wrong "too few arguments"
                     )
                   ) F-in-E)
                κ)
          (update (new σ |L) unspecified-in-E σ) ,
        ▹ (wrong "out of memory") σ

-- Non-compositional:
-- ℰ⟦ ⦇lambda I ␣ Γ* ␣ E₀ ⦈ ⟧ = ℰ⟦ ⦇lambda ⦇ · I ⦈ Γ* ␣ E₀ ⦈ ⟧

ℰ⟦ ⦇lambda I ␣ Γ* ␣ E₀ ⦈ ⟧ = λ ρ κ → ◃ λ σ →
      (new σ ∈L) ⟶
        ▹ (send (◃ ( (new σ |L) ,
                     (λ ϵ* κ′ →
                        tievalsrest
                          (λ α* → (λ ρ′ → C*⟦ Γ* ⟧ ρ′ (ℰ⟦ E₀ ⟧ ρ′ κ′))
                                  (extends ρ (1 , I) α*))
                          ϵ*
                          (η 0))
                   ) F-in-E)
                κ)
          (update (new σ |L) unspecified-in-E σ) ,
        ▹ (wrong "out of memory") σ


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

ℰ⟦ ⦇if E₀ ␣ E₁ ⦈ ⟧ = λ ρ κ →
  ℰ⟦ E₀ ⟧ ρ (single (λ ϵ →
      truish ϵ ⟶ ℰ⟦ E₁ ⟧ ρ κ ,
        send unspecified-in-E κ))

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

```
𝓔⟦ ⦇set! I ␣ E ⦈ ⟧ = λ ρ κ →
  𝓔⟦ E ⟧ ρ (single (λ ϵ →
    assign (lookup ρ I) ϵ (send unspecified-in-E κ)))

-- 𝓔*⟦_⟧  : Exp *′ → U → K → C

𝓔*⟦ 0 , _ ⟧ = λ ρ κ → ▹ κ ⟨⟩

-- 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*))))

𝓔*⟦ 1 , E ⟧  = λ ρ κ →
  𝓔⟦ E ⟧ ρ (single (λ ϵ → ▹ κ ⟨ ϵ ⟩ ))

𝓔*⟦ suc (suc n) , E , Es ⟧ = λ ρ κ →
  𝓔⟦ E ⟧ ρ (single (λ ϵ₀ →
    𝓔*⟦ suc n , Es ⟧ ρ (◃ λ ϵ* →
      ▹ κ (⟨ ϵ₀ ⟩ § ϵ*))))

-- 𝓒*⟦_⟧  : Com *′ → U → C → C

𝓒*⟦ 0 , _ ⟧ = λ ρ θ → θ

𝓒*⟦ 1 , Γ ⟧ = λ ρ θ → 𝓔⟦ Γ ⟧ ρ (◃ λ ϵ* → θ)

𝓒*⟦ suc (suc n) , Γ , Γs ⟧ = λ ρ θ →
  𝓔⟦ Γ ⟧ ρ (◃ λ ϵ* →
    𝓒*⟦ suc n , Γs ⟧ ρ θ)
```