PrimSchemeQED Agda Formalization

DRAFT

Abstract

This is a complete listing of an Agda formalization of the denotational semantics in the paper *Compositional Semantics for eval in Scheme* by Peter D. Mosses, submitted to OLIVIERFEST 2025.

The Agda code and its listing are unpolished initial versions, and will be replaced by improved versions before submission as supplementary material.

The Agda code type-checks with Agda version 2.7.0.1 and the standard Agda library. The PDF of this listing was produced by running the following command in the project root:

```
make DIR=. ROOT=PrimSchemeQED/All.lagda pdf
```


```
module PrimSchemeQED.All where

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

1 Domain Notation

```
module PrimSchemeQED.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 [_,_]) 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
```

2 Abstract Syntax

```
module PrimSchemeQED.Abstract-Syntax where

open import PrimSchemeQED.Domain-Notation
  using (_*′)

open import Data.Bool.Base
  using (Bool)
open import Data.Integer.Base
  renaming (ℤ to Int)
open import Data.String.Base
  using (String)

-- 7.2.1. Abstract syntax

data Con : Set     -- constants, *excluding* quotations
Ide      = String  -- identifiers (variables)
data Key : Set     -- keywords
data Dat : Set     -- external representations
data Exp : Set     -- expressions

data Con where
  int : Int → Con
  #t : Con
  #f : Con

data Key where
  quote′  : Key
  lambda : Key
  if     : Key
  set!   : Key
  eval   : Key

data Dat where
  con   : Con → Dat            -- constants
  ide   : Ide → Dat            -- symbols
  key   : Key → Dat            -- keyword
  ′     : Dat → Dat            -- ’Δ
  ⦅_⦆   : Dat *′ → Dat         -- lists (Δ*)
  ⦅_·_⦆ : Dat *′ → Dat → Dat -- pairs (Δ*.Δ)

data Exp where
  con              : Con → Exp                   -- K
  ide              : Ide → Exp                   -- I
  ⦅_␣_⦆            : Exp → Exp *′ → Exp          -- (E0 E*′)
  ⦅lambda␣⦅_⦆_⦆    : Ide *′ → Exp → Exp          -- (lambda (I*′) E0)
  ⦅if_␣_␣_⦆        : Exp → Exp → Exp → Exp -- (if E0 E1 E2)
  ⦅set!_␣_⦆        : Ide → Exp → Exp             -- (set! I E)
  ⦅quote_⦆         : Dat → Exp                   -- (quote Δ)
  ′                : Dat → Exp                   -- ′ Δ
  ⦅eval_⦆          : Exp → Exp                   -- (eval E)
  ⦅␣⦆              : Exp                         -- illegal
```

variable

$\mathsf{Z} \; : \; \mathsf{Int}$
$\mathsf{K} \; : \; \mathsf{Con}$
$\mathsf{I} \; : \; \mathsf{Ide}$
$\mathsf{I}^* \; : \; \mathsf{Ide}\ ^{*\prime}$
$\mathsf{X} \; : \; \mathsf{Key}$
$\mathsf{E} \; : \; \mathsf{Exp}$
$\mathsf{E}^* \; : \; \mathsf{Exp}\ ^{*\prime}$
$\Delta \; : \; \mathsf{Dat}$
$\Delta^* \; : \; \mathsf{Dat}\ ^{*\prime}$

3 Domain Equations

```
module PrimSchemeQED.Domain-Equations where

open import PrimSchemeQED.Domain-Notation
open import PrimSchemeQED.Abstract-Syntax
  using (Ide; Key; Dat; Exp)

open import Data.Integer.Base
  renaming (ℤ to Int)

-- 7.2.2. Domain equations

postulate
  Loc : Set     -- set of locations
  A   : Domain -- answers

data Misc : Set where
  null unspecified : Misc

-- Non-recursive domain definitions

L = Loc +⊥  -- locations
N = ℕ +⊥    -- natural numbers
T = Bool +⊥ -- booleans
Q = Ide +⊥  -- identifier symbols
R = Int +⊥  -- numbers
P = (L × L) -- pairs
M = Misc +⊥ -- miscellaneous
D = Dat +⊥  -- data ASTs
X = Key +⊥  -- keyword symbols

-- Recursive 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

postulate instance
  iso-F : F ↔ (E * → (E → C) → C)
  iso-E : E ↔ (𝕃 (Q ⊎ T ⊎ R ⊎ P ⊎ M ⊎ F ⊎ D ⊎ X))
  iso-S : S ↔ (L → E)
  iso-U : U ↔ (Ide → L)
  iso-C : C ↔ (S → A)

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

```
variable
  α  : L
  α* : L *
  ν  : N
  γ  : Q
  τ  : T
  ζ  : R
  π  : P
  μ  : M
  ϕ  : F
  δ  : D
  χ  : X
  ϵ  : E
  ϵ* : E *
  σ  : S
  ρ  : U
  θ  : C

pattern
  inj-Q γ  = inj₁ γ
pattern
  inj-T τ  = inj₂ (inj₁ τ)
pattern
  inj-R ζ  = inj₂ (inj₂ (inj₁ ζ))
pattern
  inj-P π  = inj₂ (inj₂ (inj₂ (inj₁ π)))
pattern
  inj-M μ = inj₂ (inj₂ (inj₂ (inj₂ (inj₁ μ))))
pattern
  inj-F ϕ  = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ ϕ)))))
pattern
  inj-D δ  = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ δ))))))
pattern
  inj-X χ  = inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ χ))))))

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

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

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

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

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

_D-in-E   : D → E
δ D-in-E  = ◃ (η (inj-D δ))

_Q-in-E   : Q → E
γ Q-in-E  = ◃ (η (inj-Q γ))
```

$$
\begin{aligned}
&\_\mathbf{T}\text{-in-}\mathbf{E} &&: \mathbf{T} \to \mathbf{E} \\
&\tau\ \mathbf{T}\text{-in-}\mathbf{E} &&= \triangleleft\ (\eta\ (\text{inj-}\mathbf{T}\ \tau))
\end{aligned}
$$

$$
\begin{aligned}
&\_\mathbf{R}\text{-in-}\mathbf{E} &&: \mathbf{R} \to \mathbf{E} \\
&\zeta\ \mathbf{R}\text{-in-}\mathbf{E} &&= \triangleleft\ (\eta\ (\text{inj-}\mathbf{R}\ \zeta))
\end{aligned}
$$

$$
\begin{aligned}
&\_\mathbf{P}\text{-in-}\mathbf{E} &&: \mathbf{P} \to \mathbf{E} \\
&\pi\ \mathbf{P}\text{-in-}\mathbf{E} &&= \triangleleft\ (\eta\ (\text{inj-}\mathbf{P}\ \pi))
\end{aligned}
$$

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

$$
\begin{aligned}
&\_\mathbf{X}\text{-in-}\mathbf{E} &&: \mathbf{X} \to \mathbf{E} \\
&\chi\ \mathbf{X}\text{-in-}\mathbf{E} &&= \triangleleft\ (\eta\ (\text{inj-}\mathbf{X}\ \chi))
\end{aligned}
$$

$$
\begin{aligned}
&\text{null-in-}\mathbf{E} &&: \mathbf{E} \\
&\text{null-in-}\mathbf{E} &&= \triangleleft\ (\eta\ (\text{inj-}\mathbf{M}\ (\eta\ \text{null})))
\end{aligned}
$$

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

4 Auxiliary Functions

```
module PrimSchemeQED.Auxiliary-Functions where

open import PrimSchemeQED.Domain-Notation
open import PrimSchemeQED.Domain-Equations
open import PrimSchemeQED.Abstract-Syntax -- using (Dat; Ide; Exp)

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′

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
  new : S → L

postulate
  _==ᴸ_ : L → L → T

_[_/_]′ : S → E → L → S
σ [ z / α ]′ = ◃ λ α′ → (α ==ᴸ α′) ⟶ z , ▹ σ α′

tievals : (L * → C) → E * → C
tievals = fix λ tievals′ →
  λ ψ ϵ* → ◃ λ σ →
    (# ϵ* ==⊥ 0) ⟶ ▹ (ψ ⟨⟩) σ ,
      (▹ (tievals′ (λ α* → ψ (⟨ new σ ⟩ § α*)) (ϵ* † 1))
      (σ [ (ϵ* ↓ 1) / new σ ]′))

truish : E → T
-- truish = λ ϵ → ϵ = false ⟶ false , true
truish = λ ϵ → (is-not-false ♯) (▹ ϵ) where
  is-not-false : (Q ⊎ T ⊎ R ⊎ P ⊎ M ⊎ F ⊎ D ⊎ X) → T
  is-not-false (inj-T τ) = ((λ { false → η false ; _ → η true }) ♯) (τ)
  is-not-false (inj₁ _)  = η true
  is-not-false (inj₂ _)  = η true
```

```
cons : E * → (E → C) → C
cons =
  λ ϵ* κ → ◁ λ σ →
    (λ σ′ → ▷ (κ ((new σ , new σ′) P-in-E))
                      (σ′ [ (ϵ* ↓ 2)/ new σ′ ]′))
    (σ [ (ϵ* ↓ 1) / new σ ]′)

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

-- For use in the denotation of (eval expression ...):

-- datum ϵ κ maps the object ϵ representing the Dat Δ to Δ

datum : E → (E → C) → C
datum = fix λ datum′ →
  λ ϵ κ → ◁ λ σ → ▷ (
    (ϵ ∈P) ⟶
      datum′ (▷ σ (ϵ |P ↓1)) (λ ϵ₁ →
        datum′ (▷ σ (ϵ |P ↓2)) (λ ϵ₂ →
          κ (η (dat-cons ((id ♯) (ϵ₁ |D)) ((id ♯) (ϵ₂ |D))) D-in-E))) ,
      κ (f ((id ♯) (▷ ϵ)) D-in-E)
    ) σ
  where
    dat-cons : Dat → Dat → Dat
    dat-cons Δ₀ ⦇ Δ* ⦈ = ⦇ (Δ₀ ::′ Δ*) ⦈
    dat-cons Δ₀ Δ₁ = ⦇ (1 , Δ₀) · Δ₁ ⦈
    f : (Q ⊎ T ⊎ R ⊎ P ⊎ M ⊎ F ⊎ D ⊎ X) → D
    f (inj-Q γ)  = η (ide I′) where I′ = (id ♯) γ
    f (inj-T τ)  = η (con (if b then #t else #f)) where b = (id ♯) τ
    f (inj-R ζ)  = η (con (int Z′)) where Z′ = (id ♯) ζ
    f (inj-P π)  = ⊥
    f (inj-M μ) with (id ♯) μ
    f (inj-M μ) | null = η ( ⦇ 0 , [] ⦈ )
    f (inj-M μ) | _    = ⊥
    f (inj-F ϕ)  = ⊥
    f (inj-D δ)  = δ
    f (inj-X χ)  = η (key X′) where X′ = (id ♯) χ
```

```
-- exp Δ maps Δ : Dat to an expression, returning the illegal ⦇␣⦈
-- when Δ does not represent a valid expression

exp : Dat → Exp

exps : ∀ {n} → Dat ^ n → Exp ^ n

ides : ∀ {n} → Dat ^ n → Ide ^ n

-- exp : Dat → Exp

exp (con K) = con K

exp (ide I) = ide I

exp ( ′ Δ ) =
  ⦇quote Δ ⦈

exp ⦇ 2 , key quote′ , Δ ⦈ =
  ⦇quote Δ ⦈

exp ⦇ 3 , key lambda , ⦇ m , I* ⦈ , Δ₀ ⦈ =
  ⦇lambda␣⦇ m , ides I* ⦈ exp Δ₀ ⦈

exp ⦇ 4 , key if , Δ₀ , Δ₁ , Δ₂ ⦈ =
  ⦇if exp Δ₀ ␣ exp Δ₁ ␣ exp Δ₂ ⦈

exp ⦇ 3 , key set! , ide I , Δ ⦈ =
  ⦇set! I ␣ exp Δ ⦈

exp ⦇ suc (suc n) , ide I , Δ* ⦈ =
  ⦇ ide I ␣ (suc n , exps Δ*) ⦈

exp _ = ⦇␣⦈

-- exps : ∀ {n} → Dat ^ n → Exp ^ n

exps {0} _ = []

exps {1} Δ = exp Δ

exps {suc (suc n)} (Δ , Δ*) = (exp Δ , exps Δ*)

-- ides : ∀ {n} → Dat ^ n → Ide ^ n

ides {0} _ = []

ides {1} (ide I) = I

ides {1} _ = "?"

ides {suc (suc n)} (ide I , Δ*) = (I , ides Δ*)

ides {suc (suc n)} ( _ , Δ*) = ("?" , ides Δ*)
```

5 Semantic Functions

```
module PrimSchemeQED.Semantic-Functions where

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

-- 7.2.3. Semantic functions

-- Constant denotations

𝒦⟦_⟧ : Con → E

𝒦⟦ int Z ⟧ = (η Z) R-in-E
𝒦⟦ #t    ⟧ = (η true) T-in-E
𝒦⟦ #f    ⟧ = (η false) T-in-E

-- Datum denotations

𝒟⟦_⟧  : Dat → (E → C) → C
𝒟*⟦_⟧ : Dat *′ → (E → C) → C

-- 𝒟⟦_⟧   : Dat → (E → C) → C

𝒟⟦ con K        ⟧ = λ κ → κ(𝒦⟦ K ⟧)
𝒟⟦ ide I        ⟧ = λ κ → κ((η I) Q-in-E)
𝒟⟦ key X        ⟧ = λ κ → κ((η X) X-in-E)
𝒟⟦ ′ Δ          ⟧ = 𝒟⟦ Δ ⟧
𝒟⟦ ⦇ Δ* ⦈       ⟧ = 𝒟*⟦ Δ* ⟧
𝒟⟦ ⦇ Δ* · Δ ⦈   ⟧ = λ κ →
  𝒟*⟦ Δ* ⟧ (λ ϵ₀ →
    𝒟⟦ Δ ⟧ (λ ϵ₁ →
        cons ⟨ ϵ₀ , ϵ₁ ⟩ κ))

-- 𝒟*⟦_⟧  : Dat *′ → (E → C) → C

𝒟*⟦ 0 , _ ⟧ = λ κ → κ null-in-E

𝒟*⟦ 1 , Δ ⟧ = λ κ →
  𝒟⟦ Δ ⟧ (λ ϵ →
        cons ⟨ ϵ , null-in-E ⟩ κ)

𝒟*⟦ suc (suc n) , Δ , Δ* ⟧ = λ κ →
  𝒟⟦ Δ ⟧ (λ ϵ₀ →
    𝒟*⟦ suc n , Δ* ⟧ (λ ϵ₁ →
        cons ⟨ ϵ₀ , ϵ₁ ⟩ κ))
```

```
-- Expression denotations

𝓔_⟦_⟧  : (Exp → U → (E → C) → C) → Exp → U → (E → C) → C
𝓔*_⟦_⟧ : (Exp → U → (E → C) → C) → Exp *′ → U → (E * → C) → C

-- 𝓔_⟦_⟧    : Exp → U → (E → C) → C

𝓔 𝓔′ ⟦ con K ⟧ = λ ρ κ → κ (𝒦⟦ K ⟧)

𝓔 𝓔′ ⟦ ide I ⟧ = λ ρ κ →
  ◃ λ σ → ▹ (κ (▹ σ (▹ ρ I))) σ

𝓔 𝓔′ ⟦ ⦇ E0 ␣ E* ⦈ ⟧ = λ ρ κ →
  𝓔 𝓔′ ⟦ E0 ⟧ ρ (λ ε0 →
    𝓔* 𝓔′ ⟦ E* ⟧ ρ (λ ε* →
      ▹ (ε0 |F) ε* κ))

𝓔 𝓔′ ⟦ ⦇lambda␣⦇ I* ⦈ E0 ⦈ ⟧ = λ ρ κ →
  κ (◃ ( λ ε* κ′ →
          tievals
            (λ α* → 𝓔 𝓔′ ⟦ E0 ⟧ (extends ρ I* α*) κ′)
            ε*
        ) F-in-E)

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

𝓔 𝓔′ ⟦ ⦇set! I ␣ E ⦈ ⟧ = λ ρ κ →
  𝓔 𝓔′ ⟦ E ⟧ ρ (λ ε →
    ◃ λ σ → ▹ (κ unspecified-in-E) (σ [ ε / (▹ ρ I) ]′))

𝓔 𝓔′ ⟦ ⦇quote Δ ⦈ ⟧ = λ ρ κ → 𝒟⟦ Δ ⟧ κ

𝓔 𝓔′ ⟦ ′ Δ ⟧ = λ ρ κ → 𝒟⟦ Δ ⟧ κ

𝓔 𝓔′ ⟦ ⦇eval E ⦈ ⟧ = λ ρ κ →
  𝓔 𝓔′ ⟦ E ⟧ ρ (λ ε →
    datum ε (λ ε′ →
      (λ E′ → 𝓔′ E′ ρ κ) ((exp ♯) (ε′ |D))))

𝓔 𝓔′ ⟦ ⦇␣⦈ ⟧ = λ ρ κ → ⊥

-- 𝓔*_⟦_⟧   : Exp *′ → U → (E * → C) → C

𝓔* 𝓔′ ⟦ 0 , _ ⟧ = λ ρ κ → κ ⟨⟩

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

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

-- Program denotations

𝒫⟦_⟧ : Exp → U → (E → C) → C

𝒫⟦ E ⟧ = 𝓔 (fix 𝓔_⟦_⟧) ⟦ E ⟧
```