A Compositional Semantics for `eval` in Scheme

Listing of a Lightweight Agda Formalization

Peter D. Mosses
Delft University of Technology
Delft, Netherlands
Swansea University
Swansea, United Kingdom
P.D.Mosses@tudelft.nl

Abstract

Scm is a simple sublanguage of Scheme; ScmQ adds quotations to Scm; and ScmQE adds `eval` expressions to ScmQ. An accompanying paper presents a denotational semantics of Scm, and the additions and changes to define the semantics of ScmQ and ScmQE.

This document provides a highlighted listing of the Agda source code of a lightweight formalization of the complete denotational semantics of ScmQE, and illustrates how soundness tests can be formulated and verified. For explanatory comments, see §6 of the accompanying paper.

Agda generated the LaTeX sources for the highlighted listing; a map from Unicode characters to similar-looking math symbols was manually coded. The LaTeX sources for the illustrative fragments presented in the body of the accompanying paper were copied from the Agda-generated sources, but may have been edited to adjust layout and alignment.

CCS Concepts: **• Theory of computation → Denotational semantics**; **• Software and its engineering → Semantics**; **Functional languages**.

Keywords: Scheme, denotational semantics, compositional semantics, quote and eval, Lisp, formalization, Agda

References

[1] Peter D. Mosses. 2025. A compositional semantics for `eval` in Scheme. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore* (Singapore, Singapore). ACM, New York, NY, USA, 10 pages. doi:10.1145/3759427.3760369

[2] Peter D. Mosses. 2025. Lightweight Agda formalization of denotational semantics in article 'A compositional semantics for `eval` in Scheme'. ACM. doi:10.1145/3747409

Contents

Abstract		1
References		1
Contents		1
1	Abstract Syntax	2
2	Domain Equations	4
3	Semantic Functions	6
4	Auxiliary Functions	9
A	Notation	13
B	Soundness Tests	15

Modules

```
module ScmQE.All where

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

1 Abstract Syntax

```
module ScmQE.Abstract-Syntax where

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

data     Con   : Set    -- constants, *excluding* quotations
variable K     : Con

data     Key   : Set    -- keywords
variable X     : Key
data     Dat   : Set    -- datum
variable Δ     : Dat
data     Dat*  : Set    -- datum sequences
variable Δ*    : Dat*
data     Dat+  : Set    -- non-empty datum sequences
variable Δ+    : Dat+

Ide            = String -- identifiers (variables)
variable I     : Ide
data     Exp   : Set    -- expressions
variable E     : Exp
data     Exp*  : Set    -- expression sequences
variable E*    : Exp*

data     Body  : Set    -- body expression or definition
variable B     : Body
data     Body+ : Set    -- body sequences
variable B+    : Body+

data     Prog  : Set    -- programs
variable Π     : Prog

------------------------------------------------------------------------
-- Literal constants

data Con where              -- basic constants
  int : Int → Con           -- integer numerals
  #t  : Con                 -- true
  #f  : Con                 -- false

------------------------------------------------------------------------
-- Quotations

data Key where              -- keyword symbols
  begin define eval if lambda quote′ set! : Key

data Dat where              -- datum syntax
  con    : Con → Dat        -- constants
  ide    : Ide → Dat        -- symbols
  key    : Key → Dat        -- keywords
  ′      : Dat → Dat        -- quotation 'Δ
  (|_|)  : Dat* → Dat       -- datum lists (Δ*)
  (|_·_|) : Dat+ → Dat → Dat -- datum pairs (Δ+.Δ)
  #proc : Dat               -- procedures
```

```
data Dat* where               -- datum sequences
  ␣   : Dat*                  -- empty sequence
  _␣_ : Dat → Dat* → Dat*     -- prefix sequence Δ Δ*

data Dat+ where               -- non-empty datum sequences
  ␣_  : Dat → Dat+            -- single datum sequence Δ
  _␣_ : Dat+ → Dat → Dat+     -- suffix sequence Δ+ Δ

-----------------------------------------------------------------------
-- Expressions

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

data Exp* where                              -- expression sequences
  ␣            : Exp*                        -- empty sequence
  _␣_          : Exp → Exp* → Exp*           -- prefix sequence E E*

-----------------------------------------------------------------------
-- Definitions and Programs

data Body where
  ␣_           : Exp → Body                  -- side-effect expression E
  (|define_␣_|) : Ide → Exp → Body           -- definition (define I E)
  (|begin_|)   : Body+ → Body                -- block (begin B+)

data Body+ where                             -- body sequence
  ␣_           : Body → Body+                -- single body sequence B
  _␣_          : Body → Body+ → Body+        -- prefix body sequence B B+

data Prog where                              -- programs
  ␣            : Prog                        -- empty program
  ␣_           : Body+ → Prog                -- non-empty program B+

infix 30 ␣_
infixr 20 _␣_
```

2 Domain Equations

```
module ScmQE.Domain-Equations where

open import Notation
open import ScmQE.Abstract-Syntax using (Ide; Key; Dat; Int)

-- Domain declarations

postulate L : Domain -- locations
variable  α : L
N           : Domain -- natural numbers
T           : Domain -- booleans
R           : Domain -- numbers
P           : Domain -- pairs
M           : Domain -- miscellaneous
F           : Domain -- procedure values
Q           : Domain -- symbols
X           : Domain -- keyword values
postulate E : Domain -- expressed values
variable  ϵ : E
S           : Domain -- stores
variable  σ : S
U           : Domain -- environments
variable  ρ : U
C           : Domain -- command continuations
variable  θ : C
postulate A : Domain -- answers

E*          = E *
variable  ϵ* : E*

-- Domain equations

data Misc : Set where null unallocated undefined unspecified : Misc

N  = Nat⊥
T  = Bool⊥
R  = Int +⊥
P  = L × L
M  = Misc +⊥
F  = E* → (E → C) → C
Q  = Ide +⊥
X  = Key +⊥
-- E  =  T + R + P + M + F + Q + X
S  = L → E
U  = Ide → L
C  = S → A
```

```
-- Injections, tests, and projections
```

postulate
  _**T**-in-**E** : **T** → **E**
  _∈-**T** : **E** → Bool +⊥
  _|-**T** : **E** → **T**

  _**R**-in-**E** : **R** → **E**
  _∈-**R** : **E** → Bool +⊥
  _|-**R** : **E** → **R**

  _**P**-in-**E** : **P** → **E**
  _∈-**P** : **E** → Bool +⊥
  _|-**P** : **E** → **P**

  _**M**-in-**E** : **M** → **E**
  _∈-**M** : **E** → Bool +⊥
  _|-**M** : **E** → **M**

  _**F**-in-**E** : **F** → **E**
  _∈-**F** : **E** → Bool +⊥
  _|-**F** : **E** → **F**

  _**Q**-in-**E** : **Q** → **E**
  _∈-**Q** : **E** → Bool +⊥
  _|-**Q** : **E** → **Q**

  _**X**-in-**E** : **X** → **E**
  _∈-**X** : **E** → Bool +⊥
  _|-**X** : **E** → **X**

```
-- Operations on flat domains
```

postulate
  $\_{==}^{L}\_$: **L** → **L** → **T**
  $\_{==}^{M}\_$: **M** → **M** → **T**
  $\_{==}^{T}\_$: **T** → **T** → **T**

3 Semantic Functions

module ScmQE.Semantic-Functions where

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

$\mathcal{K}[\![\_]\!]$: Con → **E**
$\mathcal{D}[\![\_]\!]$: Dat → (**E** → **C**) → **C**
$\mathcal{D}^*[\![\_]\!]$: Dat* → (**E** → **C**) → **C**
$\mathcal{D}^+[\![\_]\!]$: Dat⁺ → **E** → (**E** → **C**) → **C**

$\mathcal{E}[\![\_]\!]$: Exp → **U** → (**E** → **C**) → **C**
$\mathcal{E}^*[\![\_]\!]$: Exp* → **U** → (**E*** → **C**) → **C**
$\mathcal{F}\_[\![\_]\!]$: (Exp → **U** → (**E** → **C**) → **C**) → Exp → **U** → (**E** → **C**) → **C**
$\mathcal{F}^*\_[\![\_]\!]$: (Exp → **U** → (**E** → **C**) → **C**) → Exp* → **U** → (**E** * → **C**) → **C**

$\mathcal{B}[\![\_]\!]$: Body → **U** → (**U** → **C**) → **C**
$\mathcal{B}^+[\![\_]\!]$: Body⁺ → **U** → (**U** → **C**) → **C**
$\mathcal{P}[\![\_]\!]$: Prog → **A**

```
-- Constant denotations 𝒦⟦ K ⟧ : E
```

$\mathcal{K}[\![$ int Z $]\!]$ = η Z **R**-in-**E**
$\mathcal{K}[\![$ #t $]\!]$ = η true **T**-in-**E**
$\mathcal{K}[\![$ #f $]\!]$ = η false **T**-in-**E**

```
-- Datum denotations 𝒟⟦ Δ ⟧ : (E → C) → C
```

$\mathcal{D}[\![$ con K $]\!]$ κ = $\kappa(\mathcal{K}[\![K]\!])$
$\mathcal{D}[\![$ ide I $]\!]$ κ = $\kappa(\eta\ I$ **Q**-in-**E**$)$
$\mathcal{D}[\![$ key X $]\!]$ κ = $\kappa(\eta\ X$ **X**-in-**E**$)$
$\mathcal{D}[\![\ '\ \Delta\ ]\!]\ \kappa = \mathcal{D}[\![\ \Delta\ ]\!]\ \kappa$
$\mathcal{D}[\![\ (\!|\ \Delta^*\ |\!)\ ]\!]\ \kappa = \mathcal{D}^*[\![\ \Delta^*\ ]\!]\ \kappa$
$\mathcal{D}[\![\ (\!|\ \Delta^+ \cdot \Delta\ |\!)\ ]\!]\ \kappa = \mathcal{D}[\![\ \Delta\ ]\!]\ (\lambda\ \epsilon \to \mathcal{D}^+[\![\ \Delta^+\ ]\!]\ \epsilon\ \kappa)$
$\mathcal{D}[\![$ #proc $]\!]$ κ = $\bot$

```
-- Datum sequence denotations 𝒟*⟦ Δ* ⟧ : (E → C) → C
```

$\mathcal{D}^*[\![\ \text{␣}\ ]\!]\ \kappa = \kappa\ (\eta$ null **M**-in-**E**$)$

$\mathcal{D}^*[\![\ \Delta_1\ \text{␣}\ \Delta^*\ ]\!]\ \kappa =$
  $\mathcal{D}[\![\ \Delta_1\ ]\!]\ (\lambda\ \epsilon_1 \to$
    $\mathcal{D}^*[\![\ \Delta^*\ ]\!]\ (\lambda\ \epsilon \to$
      cons $\langle\ \epsilon_1\ ,\ \epsilon\ \rangle\ \kappa))$

```
-- Datum prefix sequence denotations 𝒟⁺⟦ Δ⁺ ⟧ : E → (E → C) → C
```

$\mathcal{D}^+[\![\ \text{␣}\ \Delta_1\ ]\!]\ \epsilon\ \kappa =$
  $\mathcal{D}[\![\ \Delta_1\ ]\!]\ (\lambda\ \epsilon_1 \to$
    cons $\langle\ \epsilon_1\ ,\ \epsilon\ \rangle\ \kappa)$

$\mathcal{D}^+[\![\ \Delta^+\ \text{␣}\ \Delta_1\ ]\!]\ \epsilon\ \kappa =$
  $\mathcal{D}[\![\ \Delta_1\ ]\!]\ (\lambda\ \epsilon_1 \to$
    cons $\langle\ \epsilon_1\ ,\ \epsilon\ \rangle\ (\lambda\ \epsilon' \to$
    $\mathcal{D}^+[\![\ \Delta^+\ ]\!]\ \epsilon'\ \kappa))$

-- Fixed expression denotations $\mathcal{E}[\![\ \mathtt{E}\ ]\!] : \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$

$\mathcal{E}[\![\ E\ ]\!] = \mathcal{F}\ (\mathrm{fix}\ \mathcal{F}\_[\![\_]\!])\ [\![\ E\ ]\!]$

-- Fixed expression sequence denotations $\mathcal{E}^*[\![\_]\!] : \mathrm{Exp}^* \to \mathbf{U} \to (\mathbf{E}^* \to \mathbf{C}) \to \mathbf{C}$

$\mathcal{E}^*[\![\ E^*\ ]\!] = \mathcal{F}^*\ (\mathrm{fix}\ \mathcal{F}\_[\![\_]\!])\ [\![\ E^*\ ]\!]$

-- Expression denotations $\mathcal{F}\ \mathcal{E}'\ [\![\ \mathtt{E}\ ]\!] : \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$

$\mathcal{F}\ \mathcal{E}'\ [\![\ \mathrm{con}\ K\ ]\!]\ \rho\ \kappa = \kappa\ (\mathcal{K}[\![\ K\ ]\!])$

$\mathcal{F}\ \mathcal{E}'\ [\![\ \mathrm{ide}\ I\ ]\!]\ \rho\ \kappa = \mathrm{hold}\ (\rho\ I)\ \kappa$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\ E \,\llcorner\, E^*\ |\!)\ ]\!]\ \rho\ \kappa =$
$\quad \mathcal{F}\ \mathcal{E}'\ [\![\ E\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow$
$\quad\quad \mathcal{F}^*\ \mathcal{E}'\ [\![\ E^*\ ]\!]\ \rho\ (\lambda\ \epsilon^* \longrightarrow$
$\quad\quad\quad (\epsilon\ |\text{-}\mathbf{F})\ \epsilon^*\ \kappa))$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\mathrm{lambda}\ I \,\llcorner\, E\ |\!)\ ]\!]\ \rho\ \kappa =$
$\quad \kappa\ (\ (\lambda\ \epsilon^*\ \kappa' \longrightarrow$
$\quad\quad\quad \mathrm{list}\ \epsilon^*\ (\lambda\ \epsilon \longrightarrow$
$\quad\quad\quad\quad \mathrm{alloc}\ \epsilon\ (\lambda\ \alpha \longrightarrow$
$\quad\quad\quad\quad\quad \mathcal{F}\ \mathcal{E}'\ [\![\ E\ ]\!]\ (\rho\ [\ \alpha\ /\ I\ ])\ \kappa'))$
$\quad\quad)\ \mathbf{F}\text{-in-}\mathbf{E})$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\mathrm{if}\ E \,\llcorner\, E_1 \,\llcorner\, E_2\ |\!)\ ]\!]\ \rho\ \kappa =$
$\quad \mathcal{F}\ \mathcal{E}'\ [\![\ E\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow$
$\quad\quad \mathrm{truish}\ \epsilon \longrightarrow \mathcal{F}\ \mathcal{E}'\ [\![\ E_1\ ]\!]\ \rho\ \kappa\ ,\ \mathcal{F}\ \mathcal{E}'\ [\![\ E_2\ ]\!]\ \rho\ \kappa)$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\mathrm{set!}\ I \,\llcorner\, E\ |\!)\ ]\!]\ \rho\ \kappa =$
$\quad \mathcal{F}\ \mathcal{E}'\ [\![\ E\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow$
$\quad\quad \mathrm{assign}\ (\rho\ I)\ \epsilon\ ($
$\quad\quad\quad \kappa\ (\eta\ \mathrm{unspecified}\ \mathbf{M}\text{-in-}\mathbf{E})))$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\mathrm{quote}\ \Delta\ |\!)\ ]\!]\ \rho\ \kappa = \mathcal{D}[\![\ \Delta\ ]\!]\ \kappa$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\mathrm{eval}\ E\ |\!)\ ]\!]\ \rho\ \kappa =$
$\quad \mathcal{F}\ \mathcal{E}'\ [\![\ E\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow$
$\quad\quad \mathrm{datum}\ \epsilon\ (\lambda\ \Delta \longrightarrow \mathcal{E}'\ (\mathrm{exp}[\![\ \Delta\ ]\!])\ \mathrm{nullenv}\ \kappa))$

$\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\_|\!)\ ]\!]\ \rho\ \kappa = \bot$

-- Expression sequence denotations $\mathcal{F}^*\ \mathcal{E}'\ [\![\ \mathtt{E}^*\ ]\!] : \mathbf{U} \to (\mathbf{E}^* \to \mathbf{C}) \to \mathbf{C}$

$\mathcal{F}^*\ \mathcal{E}'\ [\![\ \llcorner\!\lrcorner\ ]\!]\ \rho\ \kappa = \kappa\ \langle\rangle$

$\mathcal{F}^*\ \mathcal{E}'\ [\![\ E \,\llcorner\!\lrcorner\, E^*\ ]\!]\ \rho\ \kappa =$
$\quad \mathcal{F}\ \mathcal{E}'\ [\![\ E\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow$
$\quad\quad \mathcal{F}^*\ \mathcal{E}'\ [\![\ E^*\ ]\!]\ \rho\ (\lambda\ \epsilon^* \longrightarrow$
$\quad\quad\quad \kappa\ (\langle\ \epsilon\ \rangle\ \S\ \epsilon^*)))$

```
-- Body denotations 𝓑⟦ B ⟧ : U → (U → C) → C
```

$\mathcal{B}[\![\,\llcorner\, E]\!]\ \rho\ \kappa = \mathcal{E}[\![E]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow \kappa\ \rho)$

$\mathcal{B}[\![(\!|\mathrm{define}\ I \llcorner E\ |\!)]\!]\ \rho\ \kappa =$
$\quad \mathcal{E}[\![E]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow (\rho\ I ==^{L} \mathrm{unknown}) \longrightarrow$
$\qquad\qquad \mathrm{alloc}\ \epsilon\ (\lambda\ \alpha \longrightarrow \kappa\ (\rho\ [\ \alpha\ /\ I\ ])),$
$\qquad\quad \mathrm{assign}\ (\rho\ I)\ \epsilon\ (\kappa\ \rho))$

$\mathcal{B}[\![(\!|\mathrm{begin}\ B^{+}\ |\!)]\!]\ \rho\ \kappa = \mathcal{B}^{+}[\![B^{+}]\!]\ \rho\ \kappa$

```
-- Body sequence denotations 𝓑⁺⟦ B⁺ ⟧ : U → (U → C) → C
```

$\mathcal{B}^{+}[\![\,\llcorner\, B]\!]\ \rho\ \kappa = \mathcal{B}[\![B]\!]\ \rho\ \kappa$

$\mathcal{B}^{+}[\![B \llcorner B^{+}]\!]\ \rho\ \kappa = \mathcal{B}[\![B]\!]\ \rho\ (\lambda\ \rho' \longrightarrow \mathcal{B}^{+}[\![B^{+}]\!]\ \rho'\ \kappa)$

```
-- Program denotations 𝓟⟦ Π ⟧ : A
```

$\mathcal{P}[\![\,\llcorner\llcorner\,]\!] = \mathrm{finished\ initial\text{-}store}$

$\mathcal{P}[\![\,\llcorner\, B^{+}]\!] = \mathcal{B}^{+}[\![B^{+}]\!]\ \mathrm{nullenv}\ (\lambda\ \rho \longrightarrow \mathrm{finished})\ \mathrm{initial\text{-}store}$

4 Auxiliary Functions

```
module ScmQE.Auxiliary-Functions where

open import Notation
open import ScmQE.Abstract-Syntax
open import ScmQE.Domain-Equations

-- Environments ρ : U = Ide → L

postulate _==_ : Ide → Ide → Bool

_[_/_] : U → L → Ide → U
ρ [ α / I ] = λ I′ → η (I == I′) ⟶ α , ρ I′

postulate unknown : L
-- ρ I = unknown represents the lack of a binding for I in ρ

postulate nullenv : U
-- nullenv shoud include various procedures and values

-- Stores σ : S = L → E

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

assign : L → E → C → C
assign = λ α ϵ θ σ → θ (σ [ ϵ / α ]′)

hold : L → (E → C) → C
hold = λ α κ σ → κ (σ α) σ

postulate new : (L → C) → C
-- new κ σ = κ α σ′ where σ α = unallocated, σ′ α ≠ unallocated

alloc : E → (L → C) → C
alloc = λ ϵ κ → new (λ α → assign α ϵ (κ α))
-- should be ⊥ when ϵ |-M == unallocated

initial-store : S
initial-store = λ α → η unallocated M-in-E

postulate finished : C
-- normal termination with answer depending on final store

truish : E → T
truish =
  λ ϵ → (ϵ ∈-T) ⟶
      (((ϵ |-T) ==ᵀ η false) ⟶ η false , η true) ,
    η true
```

```
-- Lists
```

cons : **F**
cons =
  $\lambda\ \epsilon^*\ \kappa \rightarrow$
    $(\#\ \epsilon^*\ ==_\perp\ 2) \longrightarrow$ alloc $(\epsilon^* \downarrow 1)\ (\lambda\ \alpha_1 \rightarrow$
      alloc $(\epsilon^* \downarrow 2)\ (\lambda\ \alpha_2 \rightarrow$
        $\kappa\ ((\alpha_1\ ,\ \alpha_2)$ **P**-in-**E**$)))$,
  $\perp$

list : **F**
list = fix λ *list′* $\rightarrow$
  $\lambda\ \epsilon^*\ \kappa \rightarrow$
    $(\#\ \epsilon^*\ ==_\perp\ 0) \longrightarrow \kappa\ (\eta$ null **M**-in-**E**$)$,
      list′ $(\epsilon^* \dagger 1)\ (\lambda\ \epsilon \rightarrow$ cons $\langle\ (\epsilon^* \downarrow 1)\ ,\ \epsilon\ \rangle\ \kappa)$

car : **F**
car =
  $\lambda\ \epsilon^*\ \kappa \rightarrow (\#\ \epsilon^*\ ==_\perp\ 1) \longrightarrow$ hold $((\epsilon^* \downarrow 1)\ |$-**P** $\downarrow 1)\ \kappa\ ,\ \perp$

cdr : **F**
cdr =
  $\lambda\ \epsilon^*\ \kappa \rightarrow (\#\ \epsilon^*\ ==_\perp\ 1) \longrightarrow$ hold $((\epsilon^* \downarrow 1)\ |$-**P** $\downarrow 2)\ \kappa\ ,\ \perp$

setcar : **F**
setcar =
  $\lambda\ \epsilon^*\ \kappa \rightarrow$
    $(\#\ \epsilon^*\ ==_\perp\ 2) \longrightarrow$ assign $((\epsilon^* \downarrow 1)\ |$-**P** $\downarrow 1)$
      $(\epsilon^* \downarrow 2)$
      $(\kappa\ (\eta$ unspecified **M**-in-**E**$))$,
  $\perp$

setcdr : **F**
setcdr =
  $\lambda\ \epsilon^*\ \kappa \rightarrow$
    $(\#\ \epsilon^*\ ==_\perp\ 2) \longrightarrow$ assign $((\epsilon^* \downarrow 1)\ |$-**P** $\downarrow 2)$
      $(\epsilon^* \downarrow 2)$
      $(\kappa\ (\eta$ unspecified **M**-in-**E**$))$,
  $\perp$

```
-- datum prefix pre⟦ Δ ⟧ : Dat
```

$\mathrm{pre}[\![\_]\!] : \mathrm{Dat} \to \mathrm{Dat}$

$\mathrm{pre}[\![\ (\!|\ \sqcup\ \Delta \cdot (\!|\ \Delta^{*}\ |\!)\ |\!)\ ]\!] = [\![\ (\!|\ \Delta \sqcup \Delta^{*}\ |\!)\ ]\!]$

```
-- otherwise:
```

$\mathrm{pre}[\![\ \Delta\ ]\!] = [\![\ \Delta\ ]\!]$

```
-- datum ϵ κ applies κ to the datum represented by the value ϵ
```

$\mathrm{datum} : \mathbf{E} \to (\mathrm{Dat} \to \mathbf{C}) \to \mathbf{C}$

$\mathrm{datum} = \mathrm{fix}\ \lambda\ \mathit{datum'} \to$

$\quad \lambda\ \epsilon\ \kappa \to$

$\quad\quad (\epsilon \in\text{-}\mathbf{T}) \longrightarrow$

$\quad\quad\quad ((\epsilon\ |\text{-}\mathbf{T}) \longrightarrow \kappa\ [\![\ \mathrm{con}\ \#\mathrm{t}\ ]\!]\ ,\ \kappa\ [\![\ \mathrm{con}\ \#\mathrm{f}\ ]\!])\ ,$

$\quad\quad (\epsilon \in\text{-}\mathbf{R}) \longrightarrow$

$\quad\quad\quad ((\lambda\ Z \to \kappa\ [\![\ \mathrm{con}\ (\mathrm{int}\ Z)\ ]\!])\ \mathit{SHARP})\ (\epsilon\ |\text{-}\mathbf{R})\ ,$

$\quad\quad (\epsilon \in\text{-}\mathbf{P}) \longrightarrow$

$\quad\quad\quad \mathrm{car}\ \langle\ \epsilon\ \rangle\ (\lambda\ \epsilon_1 \to \mathrm{cdr}\ \langle\ \epsilon\ \rangle\ (\lambda\ \epsilon_2 \to$

$\quad\quad\quad\quad \mathit{datum'}\ \epsilon_1\ (\lambda\ \Delta_1 \to \mathit{datum'}\ \epsilon_2\ (\lambda\ \Delta_2 \to$

$\quad\quad\quad\quad\quad \kappa\ \mathrm{pre}[\![\ (\!|\ \sqcup\ \Delta_1 \cdot \Delta_2\ |\!)\ ]\!]))))\ ,$

$\quad\quad (\epsilon \in\text{-}\mathbf{M}) \longrightarrow$

$\quad\quad\quad (((\epsilon\ |\text{-}\mathbf{M}) ==^{M} \eta\ \mathrm{null}) \longrightarrow \kappa\ [\![\ (\!|\ \sqcup\sqcup\ |\!)\ ]\!]\ ,\ \bot)\ ,$

$\quad\quad (\epsilon \in\text{-}\mathbf{F}) \longrightarrow$

$\quad\quad\quad \kappa\ [\![\ \#\mathrm{proc}\ ]\!]\ ,$

$\quad\quad (\epsilon \in\text{-}\mathbf{Q}) \longrightarrow$

$\quad\quad\quad ((\lambda\ I \to \kappa\ [\![\ \mathrm{ide}\ I\ ]\!])\ \mathit{SHARP})\ (\epsilon\ |\text{-}\mathbf{Q})\ ,$

$\quad\quad (\epsilon \in\text{-}\mathbf{X}) \longrightarrow$

$\quad\quad\quad ((\lambda\ X \to \kappa\ [\![\ \mathrm{key}\ X\ ]\!])\ \mathit{SHARP})\ (\epsilon\ |\text{-}\mathbf{X})\ ,$

$\quad\quad \bot$

```
-- mapping datum terms to expressions
```

$\mathsf{exp}[\![\_]\!] : \mathsf{Dat} \longrightarrow \mathsf{Exp}$
$\mathsf{exp}^*[\![\_]\!] : \mathsf{Dat}^* \longrightarrow \mathsf{Exp}^*$

```
-- datum expressions exp⟦ Δ ⟧ : Exp
```

$\mathsf{exp}[\![\ \mathsf{con}\ K\ ]\!] = [\![\ \mathsf{con}\ K\ ]\!]$

$\mathsf{exp}[\![\ \mathsf{ide}\ I\ ]\!] = [\![\ \mathsf{ide}\ I\ ]\!]$

$\mathsf{exp}[\![\ '\ \Delta\ ]\!] = [\![\ (\!|\mathsf{quote}\ \Delta\ |\!)\ ]\!]$

$\mathsf{exp}[\![\ (\!|\ \mathsf{key}\ \mathsf{quote}'\ \text{␣}\ \Delta\ \text{␣}\ \text{␣␣}\ |\!)\ ]\!] = [\![\ (\!|\mathsf{quote}\ \Delta\ |\!)\ ]\!]$

$\mathsf{exp}[\![\ (\!|\ \mathsf{key}\ \mathsf{lambda}\ \text{␣}\ \mathsf{ide}\ I\ \text{␣}\ \Delta\ \text{␣}\ \text{␣␣}\ |\!)\ ]\!] =$
$\quad [\![\ (\!|\mathsf{lambda}\ I\ \text{˽}\ \mathsf{exp}[\![\ \Delta\ ]\!]\ |\!)\ ]\!]$

$\mathsf{exp}[\![\ (\!|\ \mathsf{key}\ \mathsf{if}\ \text{␣}\ \Delta\ \text{␣}\ \Delta_1\ \text{␣}\ \Delta_2\ \text{␣}\ \text{␣␣}\ |\!)\ ]\!] =$
$\quad [\![\ (\!|\mathsf{if}\ \mathsf{exp}[\![\ \Delta\ ]\!]\ \text{˽}\ \mathsf{exp}[\![\ \Delta_1\ ]\!]\ \text{˽}\ \mathsf{exp}[\![\ \Delta_2\ ]\!]\ |\!)\ ]\!]$

$\mathsf{exp}[\![\ (\!|\ \mathsf{key}\ \mathsf{set!}\ \text{␣}\ \mathsf{ide}\ I\ \text{␣}\ \Delta\ \text{␣}\ \text{␣␣}\ |\!)\ ]\!] =$
$\quad [\![\ (\!|\mathsf{set!}\ I\ \text{˽}\ \mathsf{exp}[\![\ \Delta\ ]\!]\ |\!)\ ]\!]$

$\mathsf{exp}[\![\ (\!|\ \mathsf{ide}\ I\ \text{␣}\ \Delta^*\ |\!)\ ]\!] =$
$\quad [\![\ (\!|\ \mathsf{ide}\ I\ \text{˽}\ \mathsf{exp}^*[\![\ \Delta^*\ ]\!]\ |\!)\ ]\!]$

$\mathsf{exp}[\![\ \_\ ]\!] = [\![\ (\!|\text{˽}|\!)\ ]\!]$

```
-- datum sequence expressions exp*⟦ Δ* : Exp*
```

$\mathsf{exp}^*[\![\ \text{␣␣}\ ]\!] = [\![\ \text{␣␣}\ ]\!]$

$\mathsf{exp}^*[\![\ \Delta\ \text{␣}\ \Delta^*\ ]\!] = [\![\ \mathsf{exp}[\![\ \Delta\ ]\!]\ \text{␣}\ \mathsf{exp}^*[\![\ \Delta^*\ ]\!]\ ]\!]$

A Notation

```
module Notation where

open import Data.Bool.Base     using (Bool; false; true) public
open import Data.Nat.Base      renaming (ℕ to Nat) using (suc) public
open import Data.String.Base   using (String) public
open import Data.Unit.Base     using (⊤)
open import Function           using (id; _∘_) public

Domain = Set -- unsound!

variable
  A B C   : Set
  D E F   : Domain
  n       : Nat

------------------------------------------------------------------------
-- Domains

postulate
  ⊥ : D              -- bottom element
  fix : (D → D) → D -- fixed point of endofunction

------------------------------------------------------------------------
-- Flat domains

postulate
  _+⊥     : Set → Domain            -- lifted set
  η       : A → A +⊥                -- inclusion
  _SHARP  : (A → D) → (A +⊥ → D) -- Kleisli extension

Bool⊥     = Bool +⊥                 -- truth value domain
Nat⊥      = Nat +⊥                  -- natural number domain
String⊥   = String +⊥               -- meta-string domain

postulate
  _==⊥_   : Nat⊥ → Nat → Bool⊥     -- strict numerical equality
  _⟶_,_   : Bool⊥ → D → D → D      -- McCarthy conditional

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

postulate
  _+_     : Domain → Domain → Domain      -- separated sum
  inj₁    : D → D + E                     -- injection
  inj₂    : E → D + E                     -- injection
  [_,_]   : (D → F) → (E → F) → (D + E → F) -- case analysis

------------------------------------------------------------------------
-- Product domains

postulate
  _×_ : Domain → Domain → Domain -- cartesian product
  _,_ : D → E → D × E             -- pairing
  _↓1 : D × E → D                 -- projection
  _↓2 : D × E → E                 -- projection
```

```
--------------------------------------------------------------------------
-- Tuple domains

_^_ : Domain → Nat → Domain -- D ^ n                n-tuples
D ^ 0            = ⊤
D ^ 1            = D
D ^ suc (suc n) = D × (D ^ suc n)

--------------------------------------------------------------------------
-- Finite sequence domains

postulate
  _*       : Domain → Domain -- D * domain of finite sequences
  ⟨⟩       : D *               -- empty sequence
  ⟨_⟩      : (D ^ suc n) → D *  -- ⟨ d₁ , ... , dₙ₊₁ ⟩ non-empty sequence
  #        : D * → Nat⊥        -- # d*                sequence length
  _§_      : D * → D * → D *   -- d* § d*             concatenation
  _↓_      : D * → Nat → D     -- d* ↓ n              nth component
  _†_      : D * → Nat → D *   -- d* † n              nth tail

--------------------------------------------------------------------------
-- Grouping precedence

infixr 1  _+_
infixr 2  _×_
infixr 4  _,_
infix  8  _^_
infixr 20 _⟶_,_

⟦_⟧ = id
```

B Soundness Tests

```
{-# OPTIONS --rewriting --confluence-check #-}

open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite

module ScmQE.Soundness-Tests where

open import Notation
open import ScmQE.Abstract-Syntax
open import ScmQE.Domain-Equations
open import ScmQE.Auxiliary-Functions
open import ScmQE.Semantic-Functions

open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl; cong-app)

postulate
  fix-fix : (f : D → D) → fix f ≡ f (fix f)

fix-app : (f : (A → D) → (A → D)) (a : A) →
          fix f a ≡ f (fix f) a
fix-app f = cong-app (fix-fix f)

{-# REWRITE fix-app #-}

test-1 : ∀ {K ρ κ} →
  𝓔⟦ con K ⟧ ρ κ ≡ κ (𝒦⟦ K ⟧)
test-1 = refl

test-2 : ∀ {ρ κ} →
  𝓔⟦ ⦅eval con #t ⦆ ⟧ ρ κ ≡
    datum (η true T-in-E) (λ Δ → (fix 𝓕_⟦_⟧) exp⟦ Δ ⟧ nullenv κ)
test-2 = refl
```

a b c d e : Dat
a = ide "a"
b = ide "b"
c = ide "c"
d = ide "d"
e = ide "e"

```
-- R7RS §6.4
```

```
-- (a b c d e) and (a . (b . (c . (d . (e . ()))))) are equivalent
```
test-proper-list :
  $\mathcal{D}⟦$ ⦇ a ␣ b ␣ c ␣ d ␣ e ␣ ␣␣ ⦈ ⟧ ≡
  $\mathcal{D}⟦$ ⦇ ␣ a · ⦇ ␣ b · ⦇ ␣ c · ⦇ ␣ d · ⦇ ␣ e · ⦇ ␣␣ ⦈ ⦈ ⦈ ⦈ ⦈ ⦈ ⟧
test-proper-list = refl

```
-- (a b c . d) is equivalent to (a . (b . (c . d)))
```
test-improper-list :
  $\mathcal{D}⟦$ ⦇ (((␣ a) ␣ b) ␣ c) · d ⦈ ⟧ ≡
  $\mathcal{D}⟦$ ⦇ ␣ a · ⦇ ␣ b · ⦇ ␣ c · d ⦈ ⦈ ⦈ ⟧
test-improper-list = refl