Compositional Semantics for *eval* in Scheme

Lightweight Agda Formalization of ScmQE (Supplemental Material)

Peter D. Mosses
Delft University of Technology
Delft, Netherlands
Swansea University
Swansea, United Kingdom
P.D.Mosses@swansea.ac.uk

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 in the semantics of SCMQ and SCMQE. This document lists a lightweight AGDA formalization of the complete denotational semantics of SCMQE. For explanatory comments, see 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

ACM Reference Format:
Peter D. Mosses. 2025. Compositional Semantics for *eval* in Scheme: Lightweight Agda Formalization of ScmQE (Supplemental Material). In *Proceedings of OlivierFest '25*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

OlivierFest '25, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CONTENTS

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                            -- programs
  ␣␣_            : 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**$)$
$\mathcal{D}[\![$ ′ Δ $]\!]$ κ = $\mathcal{D}[\![$ Δ $]\!]$ κ
$\mathcal{D}[\![$ ⦇ Δ* ⦈ $]\!]$ κ = $\mathcal{D}^*[\![$ Δ* $]\!]$ κ
$\mathcal{D}[\![$ ⦇ Δ$^+$ · Δ ⦈ $]\!]$ κ = $\mathcal{D}[\![$ Δ $]\!]$ $(\lambda\ \epsilon \to \mathcal{D}^+[\![$ Δ$^+$ $]\!]$ $\epsilon\ \kappa)$
$\mathcal{D}[\![$ #proc $]\!]$ κ = ⊥

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

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

$\mathcal{D}^*[\![$ Δ$_1$ ␣ Δ* $]\!]$ κ =
  $\mathcal{D}[\![$ Δ$_1$ $]\!]$ $(\lambda\ \epsilon_1 \to$
    $\mathcal{D}^*[\![$ Δ* $]\!]$ $(\lambda\ \epsilon \to$
      cons ⟨ ϵ_1 , ϵ ⟩ κ))

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

$\mathcal{D}^+[\![$ ␣ Δ$_1$ $]\!]$ $\epsilon\ \kappa$ =
  $\mathcal{D}[\![$ Δ$_1$ $]\!]$ $(\lambda\ \epsilon_1 \to$
    cons ⟨ ϵ_1 , ϵ ⟩ κ)

$\mathcal{D}^+[\![$ Δ$^+$ ␣ Δ$_1$ $]\!]$ $\epsilon\ \kappa$ =
  $\mathcal{D}[\![$ Δ$_1$ $]\!]$ $(\lambda\ \epsilon_1 \to$
    cons ⟨ ϵ_1 , ϵ ⟩ $(\lambda\ \epsilon' \to$
    $\mathcal{D}^+[\![$ Δ$^+$ $]\!]$ $\epsilon'\ \kappa$))

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

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

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

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

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

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

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

$$\begin{aligned}
&\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\ \mathsf{E} \text{␣} \mathsf{E}^*\ |\!)\ ]\!]\ \rho\ \kappa = \\
&\quad \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow \\
&\qquad \mathcal{F}^*\ \mathcal{E}'\ [\![\ \mathsf{E}^*\ ]\!]\ \rho\ (\lambda\ \epsilon^* \longrightarrow \\
&\qquad\quad (\epsilon\ |\text{-}\mathbf{F})\ \epsilon^*\ \kappa))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\ \mathsf{lambda}\ \mathsf{I} \text{␣} \mathsf{E}\ |\!)\ ]\!]\ \rho\ \kappa = \\
&\quad \kappa\ (\ (\lambda\ \epsilon^*\ \kappa' \longrightarrow \\
&\qquad\qquad \mathsf{list}\ \epsilon^*\ (\lambda\ \epsilon \longrightarrow \\
&\qquad\qquad\quad \mathsf{alloc}\ \epsilon\ (\lambda\ \alpha \longrightarrow \\
&\qquad\qquad\qquad \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}\ ]\!]\ (\rho\ [\ \alpha\ /\ \mathsf{I}\ ])\ \kappa')) \\
&\qquad\ )\ \mathbf{F}\text{-in-}\mathbf{E})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\ \mathsf{if}\ \mathsf{E} \text{␣} \mathsf{E}_1 \text{␣} \mathsf{E}_2\ |\!)\ ]\!]\ \rho\ \kappa = \\
&\quad \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow \\
&\qquad \mathsf{truish}\ \epsilon \longrightarrow \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}_1\ ]\!]\ \rho\ \kappa\ ,\ \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}_2\ ]\!]\ \rho\ \kappa)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\ \mathsf{set!}\ \mathsf{I} \text{␣} \mathsf{E}\ |\!)\ ]\!]\ \rho\ \kappa = \\
&\quad \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow \\
&\qquad \mathsf{assign}\ (\rho\ \mathsf{I})\ \epsilon\ (\\
&\qquad\quad \kappa\ (\eta\ \mathsf{unspecified}\ \mathbf{M}\text{-in-}\mathbf{E})))
\end{aligned}$$

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

$$\begin{aligned}
&\mathcal{F}\ \mathcal{E}'\ [\![\ (\!|\ \mathsf{eval}\ \mathsf{E}\ |\!)\ ]\!]\ \rho\ \kappa = \\
&\quad \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow \\
&\qquad \mathsf{datum}\ \epsilon\ (\lambda\ \Delta \longrightarrow \mathcal{E}'\ (\mathsf{exp}[\![\ \Delta\ ]\!])\ \mathsf{nullenv}\ \kappa))
\end{aligned}$$

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

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

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

$$\begin{aligned}
&\mathcal{F}^*\ \mathcal{E}'\ [\![\ \mathsf{E} \text{␣␣} \mathsf{E}^*\ ]\!]\ \rho\ \kappa = \\
&\quad \mathcal{F}\ \mathcal{E}'\ [\![\ \mathsf{E}\ ]\!]\ \rho\ (\lambda\ \epsilon \longrightarrow \\
&\qquad \mathcal{F}^*\ \mathcal{E}'\ [\![\ \mathsf{E}^*\ ]\!]\ \rho\ (\lambda\ \epsilon^* \longrightarrow \\
&\qquad\quad \kappa\ (\langle\ \epsilon\ \rangle\ \S\ \epsilon^*)))
\end{aligned}$$

`-- Body denotations` $\mathcal{B}[\![\ \mathsf{B}\ ]\!]$ `: U → (U → C) → C`

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

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

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

`-- Body sequence denotations` $\mathcal{B}^{+}[\![\ \mathsf{B}^{+}\ ]\!]$ `: U → (U → C) → C`

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

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

`-- Program denotations` $\mathcal{P}[\![\ \Pi\ ]\!]$ `: A`

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

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

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 =
$\lambda\ \epsilon^*\ \kappa \rightarrow$
$(\#\ \epsilon^*\ \texttt{==}\!\bot\ 2) \longrightarrow \text{alloc}\ (\epsilon^* \downarrow 1)\ (\lambda\ \alpha_1 \rightarrow$
$\quad \text{alloc}\ (\epsilon^* \downarrow 2)\ (\lambda\ \alpha_2 \rightarrow$
$\quad\quad \kappa\ ((\alpha_1\ ,\ \alpha_2)\ \textbf{P}\text{-in-}\textbf{E})))\ ,$
$\bot$

list : **F**
list = fix λ list′ $\rightarrow$
$\lambda\ \epsilon^*\ \kappa \rightarrow$
$(\#\ \epsilon^*\ \texttt{==}\!\bot\ 0) \longrightarrow \kappa\ (\eta\ \text{null}\ \textbf{M}\text{-in-}\textbf{E})\ ,$
$\text{list}'\ (\epsilon^*\ \dagger\ 1)\ (\lambda\ \epsilon \rightarrow \text{cons}\ \langle\ (\epsilon^* \downarrow 1)\ ,\ \epsilon\ \rangle\ \kappa)$

car : **F**
car =
$\lambda\ \epsilon^*\ \kappa \rightarrow (\#\ \epsilon^*\ \texttt{==}\!\bot\ 1) \longrightarrow \text{hold}\ ((\epsilon^* \downarrow 1)\ |\text{-}\textbf{P} \downarrow 1)\ \kappa\ ,\ \bot$

cdr : **F**
cdr =
$\lambda\ \epsilon^*\ \kappa \rightarrow (\#\ \epsilon^*\ \texttt{==}\!\bot\ 1) \longrightarrow \text{hold}\ ((\epsilon^* \downarrow 1)\ |\text{-}\textbf{P} \downarrow 2)\ \kappa\ ,\ \bot$

setcar : **F**
setcar =
$\lambda\ \epsilon^*\ \kappa \rightarrow$
$(\#\ \epsilon^*\ \texttt{==}\!\bot\ 2) \longrightarrow \text{assign}\ ((\epsilon^* \downarrow 1)\ |\text{-}\textbf{P} \downarrow 1)$
$\quad (\epsilon^* \downarrow 2)$
$\quad (\kappa\ (\eta\ \text{unspecified}\ \textbf{M}\text{-in-}\textbf{E}))\ ,$
$\bot$

setcdr : **F**
setcdr =
$\lambda\ \epsilon^*\ \kappa \rightarrow$
$(\#\ \epsilon^*\ \texttt{==}\!\bot\ 2) \longrightarrow \text{assign}\ ((\epsilon^* \downarrow 1)\ |\text{-}\textbf{P} \downarrow 2)$
$\quad (\epsilon^* \downarrow 2)$
$\quad (\kappa\ (\eta\ \text{unspecified}\ \textbf{M}\text{-in-}\textbf{E}))\ ,$
$\bot$

```
-- datum prefix pre⟦ Δ ⟧ : Dat

pre⟦_⟧ : Dat → Dat

pre⟦ ⦇ ␣ Δ · ⦇ Δ* ⦈ ⦈ ⟧ = ⟦ ⦇ Δ ␣ Δ* ⦈ ⟧
-- otherwise:
pre⟦ Δ ⟧ = ⟦ Δ ⟧

-- datum ϵ κ applies κ to the datum represented by the value ϵ
datum : E → (Dat → C) → C
datum = fix λ datum′ →
  λ ϵ κ →
    (ϵ ∈-T) ⟶
      ((ϵ |-T) ⟶ κ ⟦ con #t ⟧ , κ ⟦ con #f ⟧) ,
    (ϵ ∈-R) ⟶
      ((λ Z → κ ⟦ con (int Z) ⟧) ♯) (ϵ |-R) ,
    (ϵ ∈-P) ⟶
      car ⟨ ϵ ⟩ (λ ϵ₁ → cdr ⟨ ϵ ⟩ (λ ϵ₂ →
        datum′ ϵ₁ (λ Δ₁ → datum′ ϵ₂ (λ Δ₂ →
          κ pre⟦ ⦇ ␣ Δ₁ · Δ₂ ⦈ ⟧)))) ,
    (ϵ ∈-M) ⟶
      (((ϵ |-M) ==ᴹ η null) ⟶ κ ⟦ ⦇ ␣␣ ⦈ ⟧ , ⊥) ,
    (ϵ ∈-F) ⟶
      κ ⟦ #proc ⟧ ,
    (ϵ ∈-Q) ⟶
      ((λ I → κ ⟦ ide I ⟧) ♯) (ϵ |-Q) ,
    (ϵ ∈-X) ⟶
      ((λ X → κ ⟦ key X ⟧) ♯) (ϵ |-X) ,
    ⊥
```

`-- mapping datum terms to expressions`

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

`-- datum expressions` $\mathrm{exp}[\![\ \Delta\ ]\!]$ `: Exp`

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

$$\mathrm{exp}[\![\ \mathrm{ide}\ I\ ]\!] \quad = [\![\ \mathrm{ide}\ I\ ]\!]$$

$$\mathrm{exp}[\![\ '\ \Delta\ ]\!] \qquad = [\![\ (\!|\,\mathrm{quote}\ \Delta\ |\!)\ ]\!]$$

$$\mathrm{exp}[\![\ (\!|\ \mathrm{key}\ \mathrm{quote}'\ \textvisiblespace\ \Delta\ \textvisiblespace\ \textvisiblespace\textvisiblespace\ |\!)\ ]\!] = [\![\ (\!|\,\mathrm{quote}\ \Delta\ |\!)\ ]\!]$$

$$\mathrm{exp}[\![\ (\!|\ \mathrm{key}\ \mathrm{lambda}\ \textvisiblespace\ \mathrm{ide}\ I\ \textvisiblespace\ \Delta\ \textvisiblespace\ \textvisiblespace\textvisiblespace\ |\!)\ ]\!] =$$
$$[\![\ (\!|\,\mathrm{lambda}\ I\ \textvisiblespace\ \mathrm{exp}[\![\ \Delta\ ]\!]\ |\!)\ ]\!]$$

$$\mathrm{exp}[\![\ (\!|\ \mathrm{key}\ \mathrm{if}\ \textvisiblespace\ \Delta\ \textvisiblespace\ \Delta_1\ \textvisiblespace\ \Delta_2\ \textvisiblespace\ \textvisiblespace\textvisiblespace\ |\!)\ ]\!] =$$
$$[\![\ (\!|\,\mathrm{if}\ \mathrm{exp}[\![\ \Delta\ ]\!]\ \textvisiblespace\ \mathrm{exp}[\![\ \Delta_1\ ]\!]\ \textvisiblespace\ \mathrm{exp}[\![\ \Delta_2\ ]\!]\ |\!)\ ]\!]$$

$$\mathrm{exp}[\![\ (\!|\ \mathrm{key}\ \mathrm{set!}\ \textvisiblespace\ \mathrm{ide}\ I\ \textvisiblespace\ \Delta\ \textvisiblespace\ \textvisiblespace\textvisiblespace\ |\!)\ ]\!] =$$
$$[\![\ (\!|\,\mathrm{set!}\ I\ \textvisiblespace\ \mathrm{exp}[\![\ \Delta\ ]\!]\ |\!)\ ]\!]$$

$$\mathrm{exp}[\![\ (\!|\ \mathrm{ide}\ I\ \textvisiblespace\ \Delta^*\ |\!)\ ]\!] =$$
$$[\![\ (\!|\ \mathrm{ide}\ I\ \textvisiblespace\ \mathrm{exp}^*[\![\ \Delta^*\ ]\!]\ |\!)\ ]\!]$$

$$\mathrm{exp}[\![\ \_\ ]\!] = [\![\ (\!|\textvisiblespace|\!)\ ]\!]$$

`-- datum sequence expressions` $\mathrm{exp}^*[\![\ \Delta^*$ `: Exp`*

$$\mathrm{exp}^*[\![\ \textvisiblespace\textvisiblespace\ ]\!] = [\![\ \textvisiblespace\textvisiblespace\ ]\!]$$

$$\mathrm{exp}^*[\![\ \Delta\ \textvisiblespace\ \Delta^*\ ]\!] = [\![\ \mathrm{exp}[\![\ \Delta\ ]\!]\ \textvisiblespace\ \mathrm{exp}^*[\![\ \Delta^*\ ]\!]\ ]\!]$$

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
  _♯      : (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 :
  𝒟⟦ ⦅ a ␣ b ␣ c ␣ d ␣ e ␣ ␣␣ ⦆ ⟧ ≡
  𝒟⟦ ⦅ ␣ a · ⦅ ␣ b · ⦅ ␣ c · ⦅ ␣ d · ⦅ ␣ e · ⦅ ␣␣ ⦆ ⦆ ⦆ ⦆ ⦆ ⦆ ⟧
test-proper-list = refl

-- (a b c . d) is equivalent to (a . (b . (c . d)))
test-improper-list :
  𝒟⟦ ⦅ (((␣ a) ␣ b) ␣ c) · d ⦆ ⟧ ≡
  𝒟⟦ ⦅ ␣ a · ⦅ ␣ b · ⦅ ␣ c · d ⦆ ⦆ ⦆ ⟧
test-improper-list = refl
```