

# 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  $\text{\LaTeX}$  sources for the highlighted listing; a map from UNICODE characters to similar-looking math symbols was manually coded. The  $\text{\LaTeX}$  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  $\rightarrow$  Denotational semantics; • Software and its engineering  $\rightarrow$  Semantics; Functional languages.

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

### CONTENTS

Abstract	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 ( $\mathbb{Z}$  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  $\Delta$  : Dat
data Dat* : Set -- datum sequences
variable  $\Delta^*$  : Dat*
data Dat+ : Set -- non-empty datum sequences
variable  $\Delta^+$  : 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  $\Pi$  : Prog

-----

-- Literal constants

data Con where
  int : Int  $\rightarrow$  Con -- integer numerals
  #t : Con -- true
  #f : Con -- false

-----

-- Quotations

data Key where
  begin define eval if lambda quote' set! : Key

data Dat where
  con : Con  $\rightarrow$  Dat -- constants
  ide : Ide  $\rightarrow$  Dat -- symbols
  key : Key  $\rightarrow$  Dat -- keywords
  ' : Dat  $\rightarrow$  Dat -- quotation ' $\Delta$ 
  ( $\_$ ) : Dat*  $\rightarrow$  Dat -- datum lists ( $\Delta^*$ )
  ( $\_$ · $\_$ ) : Dat+  $\rightarrow$  Dat  $\rightarrow$  Dat -- datum pairs ( $\Delta^+$ · $\Delta$ )
  #proc : Dat -- procedures

```

```

data Dat* where
  _ : Dat*
  _ : Dat → Dat* → Dat* -- prefix sequence  $\Delta \Delta^*$ 

data Dat+ where
  _ : Dat → Dat+
  _ : Dat+ → Dat → Dat+ -- suffix sequence  $\Delta^+ \Delta$ 

-----

-- Expressions

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

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

-----

-- Definitions and Programs

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

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

data Prog where
  _ : Prog
  _ : Body+ → Prog
  -- programs
  -- empty program
  -- 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  $\alpha$  : 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  $\epsilon$  : E
S          : Domain -- stores
variable  $\sigma$  : S
U          : Domain -- environments
variable  $\rho$  : U
C          : Domain -- command continuations
variable  $\theta$  : C
postulate A : Domain -- answers

E*          = E*
variable  $\epsilon^*$  : E*

-- Domain equations

data Misc : Set where null unallocated undefined unspecified : Misc

N = Nat $\perp$ 
T = Bool $\perp$ 
R = Int +  $\perp$ 
P = L  $\times$  L
M = Misc +  $\perp$ 
F = E*  $\rightarrow$  (E  $\rightarrow$  C)  $\rightarrow$  C
Q = Ide +  $\perp$ 
X = Key +  $\perp$ 
-- E = T + R + P + M + F + Q + X
S = L  $\rightarrow$  E
U = Ide  $\rightarrow$  L
C = S  $\rightarrow$  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

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

 $\mathcal{E}[\_]$  :  $\text{Exp} \rightarrow \mathbf{U} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{E}^*[\_]$  :  $\text{Exp}^* \rightarrow \mathbf{U} \rightarrow (\mathbf{E}^* \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{F}[\_]$  :  $(\text{Exp} \rightarrow \mathbf{U} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}) \rightarrow \text{Exp} \rightarrow \mathbf{U} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{F}^*[\_]$  :  $(\text{Exp} \rightarrow \mathbf{U} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}) \rightarrow \text{Exp}^* \rightarrow \mathbf{U} \rightarrow (\mathbf{E}^* \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 

 $\mathcal{B}[\_]$  :  $\text{Body} \rightarrow \mathbf{U} \rightarrow (\mathbf{U} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{B}^+[\_]$  :  $\text{Body}^+ \rightarrow \mathbf{U} \rightarrow (\mathbf{U} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{P}[\_]$  :  $\text{Prog} \rightarrow \mathbf{A}$ 

-- Constant denotations  $\mathcal{K}[\text{K}] : \mathbf{E}$ 
 $\mathcal{K}[\text{int } Z] = \eta \text{ } Z \text{ R-in-E}$ 
 $\mathcal{K}[\text{\#t}] = \eta \text{ true T-in-E}$ 
 $\mathcal{K}[\text{\#f}] = \eta \text{ false T-in-E}$ 

-- Datum denotations  $\mathcal{D}[\Delta] : (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{D}[\text{con } K] \kappa = \kappa(\mathcal{K}[K])$ 
 $\mathcal{D}[\text{ide } I] \kappa = \kappa(\eta \text{ } I \text{ Q-in-E})$ 
 $\mathcal{D}[\text{key } X] \kappa = \kappa(\eta \text{ } X \text{ X-in-E})$ 
 $\mathcal{D}[\Delta'] \kappa = \mathcal{D}[\Delta] \kappa$ 
 $\mathcal{D}[(\Delta^*)] \kappa = \mathcal{D}^*[\Delta^*] \kappa$ 
 $\mathcal{D}[(\Delta^+ \cdot \Delta)] \kappa = \mathcal{D}[\Delta] (\lambda \epsilon \rightarrow \mathcal{D}^+[\Delta^+] \epsilon \kappa)$ 
 $\mathcal{D}[\text{\#proc}] \kappa = \perp$ 

-- Datum sequence denotations  $\mathcal{D}^*[\Delta^*] : (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{D}^*[\text{\_}] \kappa = \kappa(\eta \text{ null M-in-E})$ 
 $\mathcal{D}^*[\Delta_1 \text{ \_ } \Delta^*] \kappa =$ 
 $\mathcal{D}[\Delta_1] (\lambda \epsilon_1 \rightarrow$ 
 $\mathcal{D}^*[\Delta^*] (\lambda \epsilon \rightarrow$ 
 $\text{cons } \langle \epsilon_1, \epsilon \rangle \kappa))$ 

-- Datum prefix sequence denotations  $\mathcal{D}^+[\Delta^+] : \mathbf{E} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{D}^+[\text{\_} \Delta_1] \epsilon \kappa =$ 
 $\mathcal{D}[\Delta_1] (\lambda \epsilon_1 \rightarrow$ 
 $\text{cons } \langle \epsilon_1, \epsilon \rangle \kappa)$ 
 $\mathcal{D}^+[\Delta^+ \text{ \_ } \Delta_1] \epsilon \kappa =$ 
 $\mathcal{D}[\Delta_1] (\lambda \epsilon_1 \rightarrow$ 
 $\text{cons } \langle \epsilon_1, \epsilon \rangle (\lambda \epsilon' \rightarrow$ 
 $\mathcal{D}^+[\Delta^+] \epsilon' \kappa))$ 

```

```

-- Fixed expression denotations  $\mathcal{E} \llbracket E \rrbracket : \mathbf{U} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{E} \llbracket E \rrbracket = \mathcal{F} (\text{fix } \mathcal{F}_- \llbracket \_ \rrbracket) \llbracket E \rrbracket$ 

-- Fixed expression sequence denotations  $\mathcal{E}^* \llbracket \_ \rrbracket : \text{Exp}^* \rightarrow \mathbf{U} \rightarrow (\mathbf{E}^* \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{E}^* \llbracket E^* \rrbracket = \mathcal{F}^* (\text{fix } \mathcal{F}_- \llbracket \_ \rrbracket) \llbracket E^* \rrbracket$ 

-- Expression denotations  $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket : \mathbf{U} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{F} \mathcal{E}' \llbracket \text{con } K \rrbracket \rho \kappa = \kappa (\mathcal{K} \llbracket K \rrbracket)$ 
 $\mathcal{F} \mathcal{E}' \llbracket \text{ide } I \rrbracket \rho \kappa = \text{hold } (\rho I) \kappa$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\lambda E \_ E^*) \rrbracket \rho \kappa =$ 
   $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow$ 
     $\mathcal{F}^* \mathcal{E}' \llbracket E^* \rrbracket \rho (\lambda \epsilon^* \rightarrow$ 
       $(\epsilon \mid \mathbf{F}) \epsilon^* \kappa))$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\text{lambda } I \_ E) \rrbracket \rho \kappa =$ 
   $\kappa (\lambda \epsilon^* \kappa' \rightarrow$ 
     $\text{list } \epsilon^* (\lambda \epsilon \rightarrow$ 
       $\text{alloc } \epsilon (\lambda \alpha \rightarrow$ 
         $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket (\rho [\alpha / I]) \kappa'))$ 
     $) \mathbf{F}\text{-in-}\mathbf{E})$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\text{if } E \_ E_1 \_ E_2) \rrbracket \rho \kappa =$ 
   $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow$ 
     $\text{truish } \epsilon \longrightarrow \mathcal{F} \mathcal{E}' \llbracket E_1 \rrbracket \rho \kappa, \mathcal{F} \mathcal{E}' \llbracket E_2 \rrbracket \rho \kappa)$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\text{set! } I \_ E) \rrbracket \rho \kappa =$ 
   $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow$ 
     $\text{assign } (\rho I) \epsilon ($ 
       $\kappa (\eta \text{ unspecified } \mathbf{M}\text{-in-}\mathbf{E}))$ 
     $)$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\text{quote } \Delta) \rrbracket \rho \kappa = \mathcal{D} \llbracket \Delta \rrbracket \kappa$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\text{eval } E) \rrbracket \rho \kappa =$ 
   $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow$ 
     $\text{datum } \epsilon (\lambda \Delta \rightarrow \mathcal{E}' (\text{exp} \llbracket \Delta \rrbracket) \text{nullenv } \kappa))$ 
 $\mathcal{F} \mathcal{E}' \llbracket (\_ \_ \_) \rrbracket \rho \kappa = \perp$ 

-- Expression sequence denotations  $\mathcal{F}^* \mathcal{E}' \llbracket E^* \rrbracket : \mathbf{U} \rightarrow (\mathbf{E}^* \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{F}^* \mathcal{E}' \llbracket \_ \_ \_ \rrbracket \rho \kappa = \kappa \langle \rangle$ 
 $\mathcal{F}^* \mathcal{E}' \llbracket E \_ E^* \rrbracket \rho \kappa =$ 
   $\mathcal{F} \mathcal{E}' \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow$ 
     $\mathcal{F}^* \mathcal{E}' \llbracket E^* \rrbracket \rho (\lambda \epsilon^* \rightarrow$ 
       $\kappa (\langle \epsilon \rangle \S \epsilon^*))$ 

```

```

-- Body denotations  $\mathcal{B} \llbracket B \rrbracket : \mathbf{U} \rightarrow (\mathbf{U} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{B} \llbracket \underline{\_} E \rrbracket \rho \kappa = \mathcal{E} \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow \kappa \rho)$ 
 $\mathcal{B} \llbracket (\text{define } I \underline{\_} E) \rrbracket \rho \kappa =$ 
   $\mathcal{E} \llbracket E \rrbracket \rho (\lambda \epsilon \rightarrow (\rho I ==^L \text{unknown}) \longrightarrow$ 
     $\text{alloc } \epsilon (\lambda \alpha \rightarrow \kappa (\rho [\alpha / I])),$ 
     $\text{assign } (\rho I) \epsilon (\kappa \rho))$ 
 $\mathcal{B} \llbracket (\text{begin } B^+) \rrbracket \rho \kappa = \mathcal{B}^+ \llbracket B^+ \rrbracket \rho \kappa$ 

-- Body sequence denotations  $\mathcal{B}^+ \llbracket B^+ \rrbracket : \mathbf{U} \rightarrow (\mathbf{U} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
 $\mathcal{B}^+ \llbracket \underline{\_} B \rrbracket \rho \kappa = \mathcal{B} \llbracket B \rrbracket \rho \kappa$ 
 $\mathcal{B}^+ \llbracket B \underline{\_} B^+ \rrbracket \rho \kappa = \mathcal{B} \llbracket B \rrbracket \rho (\lambda \rho' \rightarrow \mathcal{B}^+ \llbracket B^+ \rrbracket \rho' \kappa)$ 

-- Program denotations  $\mathcal{P} \llbracket \Pi \rrbracket : \mathbf{A}$ 
 $\mathcal{P} \llbracket \underline{\_} \rrbracket = \text{finished initial-store}$ 
 $\mathcal{P} \llbracket \underline{\_} B^+ \rrbracket = \mathcal{B}^+ \llbracket B^+ \rrbracket \text{nullenv } (\lambda \rho \rightarrow \text{finished}) \text{initial-store}$ 

```



## 4 Auxiliary Functions

```

module ScmQE.Auxiliary-Functions where

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

-- Environments  $\rho : \mathbf{U} = \text{Ide} \rightarrow \mathbf{L}$ 
postulate _==_ : Ide  $\rightarrow$  Ide  $\rightarrow$  Bool

_[_/_] :  $\mathbf{U} \rightarrow \mathbf{L} \rightarrow \text{Ide} \rightarrow \mathbf{U}$ 
 $\rho [\alpha / I] = \lambda I' \rightarrow \eta (I == I') \rightarrow \alpha, \rho I'$ 

postulate unknown :  $\mathbf{L}$ 
--  $\rho I = \text{unknown}$  represents the lack of a binding for I in  $\rho$ 

postulate nullenv :  $\mathbf{U}$ 
-- nullenv should include various procedures and values

-- Stores  $\sigma : \mathbf{S} = \mathbf{L} \rightarrow \mathbf{E}$ 

_[_/_]' :  $\mathbf{S} \rightarrow \mathbf{E} \rightarrow \mathbf{L} \rightarrow \mathbf{S}$ 
 $\sigma [\epsilon / \alpha]' = \lambda \alpha' \rightarrow (\alpha ==^L \alpha') \rightarrow \epsilon, \sigma \alpha'$ 

assign :  $\mathbf{L} \rightarrow \mathbf{E} \rightarrow \mathbf{C} \rightarrow \mathbf{C}$ 
assign =  $\lambda \alpha \epsilon \theta \sigma \rightarrow \theta (\sigma [\epsilon / \alpha]')$ 

hold :  $\mathbf{L} \rightarrow (\mathbf{E} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
hold =  $\lambda \alpha \kappa \sigma \rightarrow \kappa (\sigma \alpha) \sigma$ 

postulate new :  $(\mathbf{L} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
--  $\text{new } \kappa \sigma = \kappa \alpha \sigma'$  where  $\sigma \alpha = \text{unallocated}$ ,  $\sigma' \alpha \neq \text{unallocated}$ 

alloc :  $\mathbf{E} \rightarrow (\mathbf{L} \rightarrow \mathbf{C}) \rightarrow \mathbf{C}$ 
alloc =  $\lambda \epsilon \kappa \rightarrow \text{new } (\lambda \alpha \rightarrow \text{assign } \alpha \epsilon (\kappa \alpha))$ 
-- should be  $\perp$  when  $\epsilon \vdash \mathbf{M} == \text{unallocated}$ 

initial-store :  $\mathbf{S}$ 
initial-store =  $\lambda \alpha \rightarrow \eta \text{ unallocated } \mathbf{M-in-E}$ 

postulate finished :  $\mathbf{C}$ 
-- normal termination with answer depending on final store

truish :  $\mathbf{E} \rightarrow \mathbf{T}$ 
truish =
   $\lambda \epsilon \rightarrow (\epsilon \in \mathbf{T}) \rightarrow$ 
     $((\epsilon \vdash \mathbf{T}) ==^T \eta \text{ false}) \rightarrow \eta \text{ false}, \eta \text{ true},$ 
     $\eta \text{ true}$ 

```

```

-- Lists

cons : F
cons =
  λ  $\epsilon^*$   $\kappa$  →
    (#  $\epsilon^*$  == ⊥ 2) → alloc ( $\epsilon^*$  ↓ 1) (λ  $\alpha_1$  →
      alloc ( $\epsilon^*$  ↓ 2) (λ  $\alpha_2$  →
         $\kappa$  (( $\alpha_1$ ,  $\alpha_2$ ) P-in-E))),
    ⊥

list : F
list = fix λ  $list'$  →
  λ  $\epsilon^*$   $\kappa$  →
    (#  $\epsilon^*$  == ⊥ 0) →  $\kappa$  ( $\eta$  null M-in-E),
     $list'$  ( $\epsilon^*$  † 1) (λ  $\epsilon$  → cons ⟨ ( $\epsilon^*$  ↓ 1),  $\epsilon$  ⟩  $\kappa$ )

car : F
car =
  λ  $\epsilon^*$   $\kappa$  → (#  $\epsilon^*$  == ⊥ 1) → hold (( $\epsilon^*$  ↓ 1) |-P ↓ 1)  $\kappa$ , ⊥

cdr : F
cdr =
  λ  $\epsilon^*$   $\kappa$  → (#  $\epsilon^*$  == ⊥ 1) → hold (( $\epsilon^*$  ↓ 1) |-P ↓ 2)  $\kappa$ , ⊥

setcar : F
setcar =
  λ  $\epsilon^*$   $\kappa$  →
    (#  $\epsilon^*$  == ⊥ 2) → assign (( $\epsilon^*$  ↓ 1) |-P ↓ 1)
      ( $\epsilon^*$  ↓ 2)
      ( $\kappa$  ( $\eta$  unspecified M-in-E)),
    ⊥

setcdr : F
setcdr =
  λ  $\epsilon^*$   $\kappa$  →
    (#  $\epsilon^*$  == ⊥ 2) → assign (( $\epsilon^*$  ↓ 1) |-P ↓ 2)
      ( $\epsilon^*$  ↓ 2)
      ( $\kappa$  ( $\eta$  unspecified M-in-E)),
    ⊥

```

```

-- 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) ]) SHARP) (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 ]) SHARP) (ε |-Q),
    (ε ∈-X) →
      ((λ X → κ [ key X ]) SHARP) (ε |-X),
    ⊥

```

```

-- mapping datum terms to expressions
exp[ ] : Dat → Exp
exp*[ ] : Dat* → Exp*

-- datum expressions exp[ Δ ] : Exp
exp[ con K ] = [ con K ]
exp[ ide I ] = [ ide I ]
exp[ ' Δ ] = [ (quote Δ) ]
exp[ (key quote' Δ Δ Δ Δ) ] = [ (quote Δ) ]
exp[ (key lambda Δ ide I Δ Δ Δ Δ) ] =
  [ (lambda I Δ exp[ Δ ] Δ) ]
exp[ (key if Δ Δ Δ1 Δ2 Δ Δ Δ Δ) ] =
  [ (if exp[ Δ ] Δ exp[ Δ1 ] Δ exp[ Δ2 ] Δ) ]
exp[ (key set! Δ ide I Δ Δ Δ Δ) ] =
  [ (set! I Δ exp[ Δ ] Δ) ]
exp[ (ide I Δ*) ] =
  [ (ide I Δ* exp*[ Δ* ] Δ) ]
exp[ _ ] = [ (Δ) ]

-- datum sequence expressions exp*[ Δ* : Exp*
exp*[ Δ Δ ] = [ Δ Δ ]
exp*[ Δ Δ* ] = [ exp[ Δ ] Δ exp*[ Δ* ] ]

```

## 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      = T
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 :
 $\mathcal{D}[(\lambda (a \_ b \_ c \_ d \_ e \_ ) ())] \equiv$ 
 $\mathcal{D}[(\lambda (a \_ ) (\lambda (b \_ ) (\lambda (c \_ ) (\lambda (d \_ ) (\lambda (e \_ ) ())) (d \_ )) (c \_ )) (b \_ )) (a \_ ))]$ 
test-proper-list = refl

-- (a b c . d) is equivalent to (a . (b . (c . d)))
test-improper-list :
 $\mathcal{D}[(\lambda ((a \_) (b \_) (c \_) (d \_)) ())] \equiv$ 
 $\mathcal{D}[(\lambda (a \_) (\lambda (b \_) (\lambda (c \_) (d \_)) (c \_)) (b \_)) (a \_)]$ 
test-improper-list = refl
```