Mechanising Denotational Semantics in Agda

Peter Mosses, Jesper Cockx, Bernhard Reus

MFPS 2026, Ljubljana, Slovenia, 1–3 June 2026

Mechanising denotational semantics in Agda

– motivation

Mechanisation?

- Support *practical use* of formal semantics, check wellformedness

Denotational semantics?

- Historical examples – e.g., semantics of inheritance (Cook & Palsberg 1989)
- Current interest – e.g., semantics of *Scheme* (R^7RS-Large ongoing)

Agda?

- *Ease of embedding* denotational definitions

Background

Denotational semantics
– Scott–Strachey style

Abstract syntax

- Sets of *abstract syntax trees* (ASTs) defined by context-free grammars

Domain equations

- Domains defined in terms of *domain constructors* – recursion *unrestricted*

Semantic functions

- *Compositional* maps from ASTs to denotations, defined by equations
- Denotations defined in typed λ*-notation*

Denotational semantics

– Scott–Strachey style

First let us recall the syntax of the λ-calculus. We assume a denumerable set **Var** of variables. Basically, the set **Exp** of *λ-terms* is taken to be the least such that:

- if x is in **Var** then x is in **Exp**;
- if x is in **Var** and M is in **Exp** then $(\lambda x\,.\,M)$ is in **Exp**;
- if M and N are in **Exp** then (MN) is in **Exp**.

Domains	Denotations
$\mathbf{D} \cong \mathbf{D} \to \mathbf{D}$ (see text) $\mathbf{Env} = \mathbf{D}^{\mathbf{Var}}$	$\mathcal{V} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{D}$ $\mathcal{V}[\![x]\!]\rho = \rho(x)$ $\mathcal{V}[\![\lambda x\,.\,M]\!]\rho = \lambda d \in \mathbf{D}\,.\,\mathcal{V}[\![M]\!](\rho[d/x])$ $\mathcal{V}[\![MN]\!]\rho = (\mathcal{V}[\![M]\!]\rho)(\mathcal{V}[\![N]\!]\rho)$

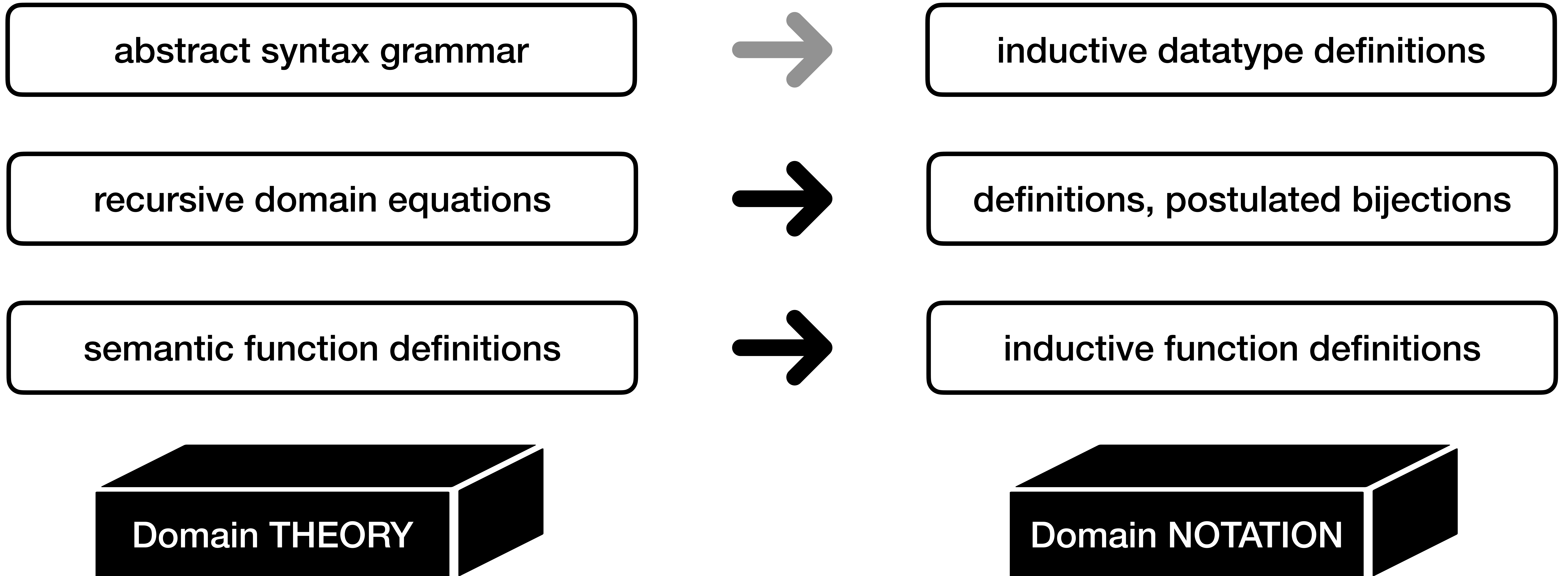
Peter D. Mosses, Gordon D. Plotkin: On Proving Limiting Completeness. SIAM J. Comput. 16(1): 179-194 (1987)

Agda

– some language features

- The universe of types Set
- Inductive datatypes **data** D **:** Set **where** …
- Dependent types (x **:** A) → B
- Anonymous functions λ x → y
- Postulated types **postulate** …
- Rewrite rules eq **:** a ≡ b**;** **{-#** REWRITE eq **#-}**
- Implicit **{** … **}** and instance **{{** … **}}** arguments

Embedding a denotational semantics in Agda
– shallow embedding



Notation

Notation
– domains

```
module Domains where
  postulate
    Domain : Set                       -- Domain is the type of all domains
    ⟪_⟫ : Domain → Set                 -- ⟪ D ⟫ is the carrier type of D
    ⊥ : {D : Domain} → ⟪ D ⟫           -- ⊥{D} is the 'bottom' element of D

module Functions where
  postulate _→ᶜ_ : Domain → Domain → Domain
  postulate dom-cts : ⟪ D →ᶜ E ⟫ ≡ (⟪ D ⟫ → ⟪ E ⟫)
  {-# REWRITE dom-cts #-}

  postulate _→ˢ_ : Set → Domain → Domain
  postulate set-cts : ⟪ A →ˢ D ⟫ ≡ (A → ⟪ D ⟫)
  {-# REWRITE set-cts #-}
```

Notation

– fixed points of functions; recursive domains

```
postulate fix : ⟪ (D →ᶜ D) →ᶜ D ⟫
```

```
module Recursion where
  postulate
    _≅_ : Domain → Domain → Set
    -- an instance of D ≅ E declares that the structure of D is the same as E
    unfold : {{D ≅ E}} → ⟪ D →ᶜ E ⟫
    fold   : {{D ≅ E}} → ⟪ E →ᶜ D ⟫
```

```
module Updates where
  _[_/_] : {{Eq A}} → ⟪ (A →ˢ D) →ᶜ D →ᶜ A →ˢ (A →ˢ D) ⟫
  ρ [ δ / a ] = λ a′ → if a == a′ then δ else ρ a′
```

Illustrative examples

Illustrative examples

Untyped λ-calculus

- simple example, embedding Scott's model D_∞

In the paper (and the appendix of these slides):

- **PCF** (*Programming Computable Functions*, Plotkin 1977)
  - intrinsically-typed syntax, dependent types of domains
- **Scm** (an ad hoc sublanguage of *Scheme*)
  - sum and product domains, denotations in continuation-passing style

Untyped λ-calculus

Denotational semantics

– Scott–Strachey style

First let us recall the syntax of the λ-calculus. We assume a denumerable set **Var** of variables. Basically, the set **Exp** of *λ-terms* is taken to be the least such that:

- if x is in **Var** then x is in **Exp**;
- if x is in **Var** and M is in **Exp** then $(\lambda x\,.\,M)$ is in **Exp**;
- if M and N are in **Exp** then (MN) is in **Exp**.

Domains	Denotations
$\mathbf{D} \cong \mathbf{D} \to \mathbf{D}$ (see text) $\mathbf{Env} = \mathbf{D}^{\mathbf{Var}}$	$\mathcal{V} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{D}$ $\mathcal{V}[\![x]\!]\rho = \rho(x)$ $\mathcal{V}[\![\lambda x\,.\,M]\!]\rho = \lambda d \in \mathbf{D}\,.\,\mathcal{V}[\![M]\!](\rho[d/x])$ $\mathcal{V}[\![MN]\!]\rho = (\mathcal{V}[\![M]\!]\rho)(\mathcal{V}[\![N]\!]\rho)$

Peter D. Mosses, Gordon D. Plotkin: On Proving Limiting Completeness. SIAM J. Comput. 16(1): 179-194 (1987)

Untyped λ-calculus
– embedding in Agda

```
module Examples.LC.Abstract-Syntax where
data Var : Set where
  x : Nat → Var
data Exp : Set where
  var_      : Var → Exp          -- variable reference
  (|λ_␣_|)  : Var → Exp → Exp    -- function abstraction
  (|_␣_|)   : Exp → Exp → Exp    -- function application
```

```
module Examples.LC.Semantic-Functions where
⟦_⟧ : Exp → ⟪ Env →ᶜ D∞ ⟫
⟦ var v ⟧ ρ          = ρ v
⟦ (|λ v ␣ e |) ⟧ ρ   = fold ( λ δ → ⟦ e ⟧ (ρ [ δ / v ]) )
⟦ (| e₁ ␣ e₂ |) ⟧ ρ  = unfold ( ⟦ e₁ ⟧ ρ ) ( ⟦ e₂ ⟧ ρ )
```

```
module Examples.LC.Domain-Equations where
postulate
  D∞ : Domain                          -- corresponds to Scott’s domain
  instance eqD∞ : D∞ ≅ (D∞ →ᶜ D∞)      -- bijection
Env = Var →ˢ D∞ -- environments
```

Postulated properties

– for use in tests

```
module Functions where
postulate
  apply-fix : {φ : ⟪ D →ᶜ D ⟫} → fix φ ≡ φ (fix φ) -- apply-fix{φ} unfolds fix φ once
{-# REWRITE apply-fix #-}
```

```
module Recursion where
postulate
  elim-unfold-fold : {{_ : D ≅ E}} → {e : ⟪ E ⟫} → unfold (fold e) ≡ e
{-# REWRITE elim-unfold-fold #-}
```

Illustrative Tests

– for the untyped λ-calculus

```
check-convergence : -- (λx1.x42)((λx0.x0 x0)(λx0.x0 x0)) = x42
  ⟦ ⦇ ⦇λ x 1 ␣ var x 42 ⦈ ␣
    ⦇ ⦇λ x 0 ␣ ⦇ var x 0 ␣ var x 0 ⦈ ⦈ ␣ ⦇λ x 0 ␣ ⦇ var x 0 ␣ var x 0 ⦈ ⦈ ⦈ ⦈ ⟧ ≡ ⟦ var x 42 ⟧
check-convergence = refl

check-abs : -- (λx1.x1)(λx1.x42) = λx2.x42
  ⟦ ⦇ ⦇λ x 1 ␣ var x 1 ⦈ ␣ ⦇λ x 1 ␣ var x 42 ⦈ ⦈ ⟧ ≡ ⟦ ⦇λ x 1 ␣ var x 42 ⦈ ⟧
check-abs = refl

check-free : -- (λx1.(λx42.x1)x2)x42 = x42
  ⟦ ⦇ ⦇λ x 1 ␣ ⦇ ⦇λ x 42 ␣ var x 1 ⦈ ␣ var x 2 ⦈ ⦈ ␣ var x 42 ⦈ ⟧ ≡ ⟦ var x 42 ⟧
check-free = refl
```

Mechanising domain theory

Mechanised standard domain theory

– in proof assistants

HOLCF

- Regensburger (1995, 1999)

Coq/Rocq

- Paulin-Mohring (2009), Benton et al. (2009), Dockins (2014)

Agda

- De Jong & Escardó (2021)

Synthetic domain theory (SDT)

Aim:

- domains are sets
- functions between domains are automatically continuous

Dana Scott (1980, pp. 426–7):

- *“Now we cannot hope to embed the theory of a typed λ-calculus in a **classical** higher-order theory […]. Something else has to be tried, and the answer is higher-order **intuitionistic** logic.”*

Synthetic domain theory (SDT)

Various authors (1980s–90s):

- developed full subcategories of models of intuitionistic set theory

Alex Simpson (2004, p. 208):

- "*an axiomatic basis for [a single unifying] treatment*"
- "*start off with a category* **S** *of intuitionistic sets […] extract a full subcategory of* predomains, **P**, *whose associated category of partial maps,* **pP**, *is algebraically compact*"
- "*require that* **S** *have enough structure to model full Intuitionistic Zermelo–Fraenkel (IZF) set theory* […] *implemented by asking for* **S** *to be given as the full subcategory of* small *objects in a category* **C** *with* class(ic) structure *and* universal object"

Mechanising SDT

– an existing mechanisation

Bernhard Reus (1996, 1999) **& Thomas Streicher** (1999):

- domains as *complete extensional Σ-spaces*
- axiomatised in the *Extended Calculus of Constructions* (ECC)
- mechanised in ***Lego*** extended with an ***impredicative*** universe of sets
- appears to be ***impossible*** to translate into Agda

Mechanising SDT

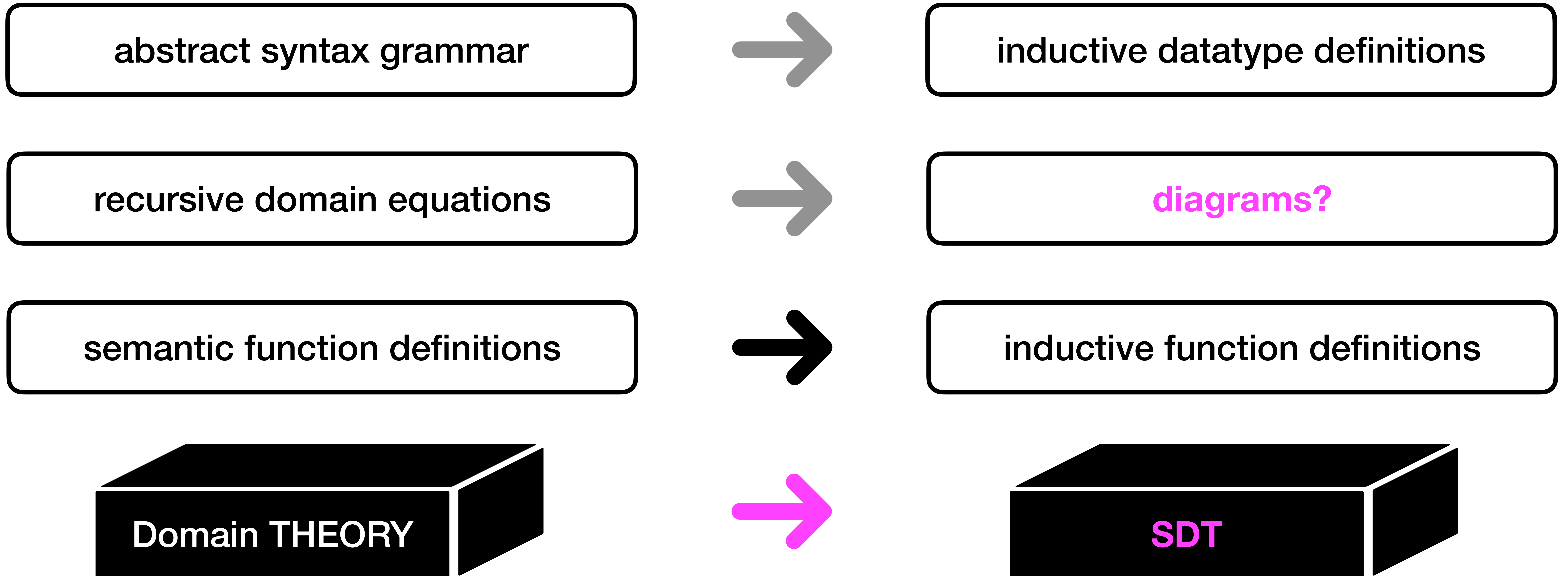
Alex Simpson (2004, p. 220):

- *"it seems likely that, with appropriate reformulations, the development [...] could be carried out in the (predicative) context of Martin-Löf's Type Theory"*
- *"Similarly, it appears that a predicative set theory could be used rather than IZF, for example Aczel's CZF"*

Are there major obstacles to doing it in Agda?

Embedding a denotational semantics in Agda

– shallow embedding



Conclusion

Currently:

- *Shallow embedding* of denotational definitions in Agda
- Mechanisation with *almost no changes* to λ-notation or domain equations

In future?

- Implement ***SDT in Agda*** – based on Alex Simpson's unifying framework
- Replace current postulates by ***definitions***

Help needed – new collaborators welcome!

pdmosses.github.io/mfps2026-agda

Postulated Domain Notation

Table of contents

- Domains
- Function Domains
- Recursive Domains
- Flat Domains
  - Booleans
  - Naturals
- Sum Domains
- Product Domains
  - Tuples
  - Sequences
- Updates

This section postulates Agda notation for the domain constructors and associated functions used in the illustrative examples.

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

module Notation where

open import Agda.Builtin.Equality public using (_≡_; refl)
open import Agda.Builtin.Equality.Rewrite using ()
open import Agda.Builtin.Nat public using (Nat) renaming (_==_ to _==ᴺ_)

variable A B C : Set
```

Domains

Domains are embedded in Agda as elements of the type `Domain`. A domain `D` is not itself a type, but it has a *carrier* type `⟪ D ⟫ : Set`, which always contains an element `⊥{D}` (written `⊥` when Agda can infer `D`).

```
module Domains where
  postulate
    Domain : Set                  -- Domain is the type of all domains
    ⟪_⟫ : Domain → Set            -- ⟪ D ⟫ is the carrier type of D
    ⊥ : {D : Domain} → ⟪ D ⟫      -- ⊥{D} is the 'bottom' element of D
    𝟙 : Domain                    -- 𝟙 is a unit domain
```

Appendix

Original motivation

A Denotational Semantics of Inheritance and its Correctness

(1963–2021)

William Cook*
Department of Computer Science
Box 1910 Brown University

Jens Palsberg
Computer Science Department
Aarhus University

This paper presents a denotational model of inheritance. The model is based on an intuitive motivation of the purpose of inheritance. The correctness of the model is demonstrated by proving it equivalent to an operational semantics of inheritance based upon the method-lookup algorithm of object-oriented languages. ...

OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications

Scheme denotational semantics

– excerpt:

7.2.2. Domain equations

$\alpha \in \mathtt{L}$		locations
$\nu \in \mathtt{N}$		natural numbers
$\mathtt{T}$	$= \{false,\ true\}$	booleans
$\mathtt{Q}$		symbols
$\mathtt{H}$		characters
$\mathtt{R}$		numbers
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	pairs
$\mathtt{E_v}$	$= \mathtt{L}^* \times \mathtt{T}$	vectors
$\mathtt{E_s}$	$= \mathtt{L}^* \times \mathtt{T}$	strings
$\mathtt{M}$	$= \{false,\ true,\ null,\ undefined,\ unspecified\}$	miscellaneous
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	procedure values
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	stores
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	environments
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	command conts
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	expression conts
$\mathtt{A}$		answers
$\mathtt{X}$		errors
$\omega \in \mathtt{P}$	$= (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{root\}$	dynamic points

7.2.3. Semantic functions

$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$

$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (I* . I) }\Gamma^*\ \mathrm{E}_0\texttt{)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathtt{L} \to \\
&\qquad\quad send\,(\langle new\,\sigma \,|\, \mathtt{L}, \\
&\qquad\qquad \lambda\epsilon^*\omega'\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to \\
&\qquad\qquad\qquad tievalsrest \\
&\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\omega'\kappa')) \\
&\qquad\qquad\qquad\qquad (extends\,\rho\,(\mathrm{I}^*\ \S\ \langle \mathrm{I}\rangle)\ \alpha^*)) \\
&\qquad\qquad\qquad\quad \epsilon^* \\
&\qquad\qquad\qquad\quad (\#\mathrm{I}^*), \\
&\qquad\qquad\qquad wrong\ \text{“too few arguments”}\rangle \text{ in } \mathtt{E}) \\
&\qquad\qquad \kappa \\
&\qquad\qquad (update\,(new\,\sigma \,|\, \mathtt{L})\ unspecified\,\sigma), \\
&\qquad\quad wrong\ \text{“out of memory”}\ \sigma
\end{aligned}
$$

$$
\mathcal{E}[\![\texttt{(lambda I }\Gamma^*\ \mathrm{E}_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (. I) }\Gamma^*\ \mathrm{E}_0\texttt{)}]\!]
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(if }\mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\omega\kappa))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(if }\mathrm{E}_0\ \mathrm{E}_1\texttt{)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa, \\
&\qquad\qquad\qquad\qquad\qquad\qquad send\ unspecified\,\kappa))
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(set! I E)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,\omega\,(single(\lambda\epsilon\,.\,assign\,(lookup\,\rho\,\mathrm{I}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \epsilon \\
&\qquad\qquad\qquad\qquad\qquad\qquad (send\ unspecified\,\kappa)))
\end{aligned}
$$

PCF

Notation

– flat domains, Booleans

```
module Flat where
  postulate
    _+⊥ : Set → Domain                             -- A +⊥ constructs a flat domain
    ↑   : ⟪ A →ˢ (A +⊥) ⟫                          -- (↑ a) injects a into A +⊥
    _♯  : ⟪ (A →ˢ D) →ᶜ (A +⊥) →ᶜ D ⟫             -- f ♯ extends f to map ⊥ to ⊥
```

```
module Booleans where
  Bool⊥ = Bool +⊥
  _⟶_,_ : ⟪ Bool⊥ →ᶜ D →ᶜ D →ᶜ D ⟫ -- β ⟶ δ₁ , δ₂ is conditional choice
  _⟶_,_ = (λ b δ₁ δ₂ → if b then δ₁ else δ₂) ♯
```

PCF
– abstract syntax: types, variables, constants

```
module Examples.PCF.Abstract-Syntax where

data Types : Set where                  -- type terms
  ι        : Types                      -- individuals
  o        : Types                      -- truth-values
  _⇒_      : Types → Types → Types      -- functions

data Vars  : Types → Set where          -- typed variables
  α        : Nat → (σ : Types) → Vars σ -- α i σ is a variable of type σ

data ℒᴬ    : Types → Set where          -- typed constants
  tt       : ℒᴬ o                       -- true
  ff       : ℒᴬ o                       -- false
  ⊃        : ℒᴬ (o ⇒ σ ⇒ σ ⇒ σ)         -- conditional
  Y        : ℒᴬ ((σ ⇒ σ) ⇒ σ)           -- fixed point
  k        : Nat → ℒᴬ ι                 -- numerals
  (|+1|)   : ℒᴬ (ι ⇒ ι)                 -- successor
  (|−1|)   : ℒᴬ (ι ⇒ ι)                 -- predecessor
  Z        : ℒᴬ (ι ⇒ o)                 -- zero test
```

PCF

– abstract syntax: terms; domain equations

```
data Terms : Types → Set where                          -- typed terms
  V _       : Vars σ → Terms σ                          -- variable
  L _       : ℒ^A σ → Terms σ                           -- constant
  ⦇_⊔_⦈     : Terms (σ ⇒ τ) → Terms σ → Terms τ        -- function application
  ⦇λ_⊔_⦈    : Vars σ → Terms τ → Terms (σ ⇒ τ)         -- function abstraction
```

```
module Examples.PCF.Domain-Equations where
```

```
𝒟 : Types → Domain        -- standard domains
𝒟 ι         = Nat⊥        -- natural numbers
𝒟 o         = Bool⊥       -- truth-values
𝒟 (σ ⇒ τ) = 𝒟 σ →ᶜ 𝒟 τ -- functions
```

```
Env = (σ : Types) → ⟪ Vars σ →ˢ 𝒟 σ ⟫ -- typed environments
```

PCF

– semantic functions

```
module Examples.PCF.Semantic-Functions where

_⟦_⟧ : Env → Vars σ → ⟪ 𝒟 σ ⟫      -- typed variable denotations
ρ ⟦ α i σ ⟧ = ρ σ (α i σ)

𝒜⟦_⟧ : ℒᴬ σ → ⟪ 𝒟 σ ⟫              -- typed constant denotations
𝒜⟦ tt ⟧      = ↑ true
𝒜⟦ ff ⟧      = ↑ false
𝒜⟦ ⊃ ⟧       = λ β δ₁ δ₂ → (β ⟶ δ₁ , δ₂)
𝒜⟦ Y ⟧       = fix
𝒜⟦ k n ⟧     = ↑ n
𝒜⟦ ⦅+1⦆ ⟧  = (λ n → ↑ (n + 1)) ♯
𝒜⟦ ⦅−1⦆ ⟧  = (λ n → ↑ (n ==ᴺ 0) ⟶ ⊥ , ↑ (n - 1)) ♯
𝒜⟦ Z ⟧       = (λ n → ↑ (n ==ᴺ 0)) ♯

𝒜′⟦_⟧ : Terms σ → ⟪ Env →ˢ 𝒟 σ ⟫ -- typed term denotations
𝒜′⟦ V α i σ ⟧ ρ            = ρ ⟦ α i σ ⟧
𝒜′⟦ L c ⟧ ρ                = 𝒜⟦ c ⟧
𝒜′⟦ ⦅ M ␣ N ⦆ ⟧ ρ          = 𝒜′⟦ M ⟧ ρ (𝒜′⟦ N ⟧ ρ)
𝒜′⟦ ⦅λ α i σ ␣ M ⦆ ⟧ ρ x = 𝒜′⟦ M ⟧ (ρ [ x / α i σ ]′)
```

PCF

– illustrative tests

```
check-fix-lambda : -- fix (λg. λa. 42) 2 ≡ 42
  𝒜′⟦ ⦇ ⦇ L Y ␣ ⦇λ g ␣ ⦇λ a ␣ L k 42 ⦈ ⦈ ⦈ ␣ L k 2 ⦈ ⟧ ρ⊥ ≡ ↑ 42
check-fix-lambda = refl

check-countdown : -- fix (λg. λa. ifz a then 42 else g (pred a)) 5 ≡ 42
  𝒜′⟦ ⦇ ⦇ L Y ␣ ⦇λ g ␣ ⦇λ a ␣
                ⦇ ⦇ ⦇ L ⊃ ␣ ⦇ L Z ␣ V a ⦈ ⦈ ␣ L k 42 ⦈ ␣
                      ⦇ V g ␣ ⦇ L ⦇−1⦈ ␣ V a ⦈ ⦈ ⦈ ⦈ ⦈ ⦈ ␣ L k 5 ⦈ ⟧ ρ⊥ ≡ ↑ 42
check-countdown = refl
```

Scm

Notation

– sum domains

```
module Sums where
  postulate
    _+_      : Domain → Domain → Domain -- D + E is separated sum
    inj₁     : ⟪ D →ᶜ (D + E) ⟫              -- inj₁ δ is injection from D
    inj₂     : ⟪ E →ᶜ (D + E) ⟫              -- inj₂ ε is injection from E
    [_,_]    : ⟪ (D →ᶜ F) →ᶜ (E →ᶜ F) →ᶜ ((D + E) →ᶜ F) ⟫
    -- [ φ , ψ ] applies φ to arguments in D, and ψ to arguments in E

  postulate
    _≳_↦_   : Domain → Nat → Domain → Set
    _in⊥_   : ⟪ D ⟫ → (E : Domain) → {{E ≳ n ↦ D}} → ⟪ E ⟫       -- δ in⊥ E injection
    _|⊥_    : ⟪ E ⟫ → (D : Domain) → {{E ≳ n ↦ D}} → ⟪ D ⟫       -- ε |⊥ D  projection
    _∈⊥_    : ⟪ E ⟫ → (D : Domain) → {{E ≳ n ↦ D}} → ⟪ Bool⊥ ⟫ -- ε ∈⊥ D  inspection
```

Notation

– product domains, function updates

```
module Products where
  postulate
    _×_  : Domain → Domain → Domain           -- D × E is the categorical product
    _,_  : ⟪ D →ᶜ E →ᶜ (D × E) ⟫              -- (δ , ε) is a pair of elements
    _↓1  : ⟪ (D × E) →ᶜ D ⟫                   -- (δ , ε)↓1 is δ
    _↓2  : ⟪ (D × E) →ᶜ E ⟫                   -- (δ , ε)↓2 is ε

module Updates where
  _[_/_] : {{Eq A}} → ⟪ (A →ˢ D) →ᶜ D →ᶜ A →ˢ (A →ˢ D) ⟫
  ρ [ δ / a ] = λ a′ → if a == a′ then δ else ρ a′

  _[_/_]⊥ : {{Eq A}} → ⟪ ((A +⊥) →ᶜ D) →ᶜ D →ᶜ (A +⊥) →ᶜ ((A +⊥) →ᶜ D) ⟫
  σ [ δ / α ]⊥ = λ α′ → (α ==⊥ α′) ⟶ δ , σ α′
```

Notation

– tuple and sequence domains

```
module Tuples where
  _^_ : Domain → Nat → Domain            -- D ^ n is the domain of n-tuples (n ≥ 0)


module Sequences where
  postulate
    _*    : Domain → Domain                -- D * is the finite sequence domain
    ⟨⟩    : ⟪ D * ⟫                         -- ⟨⟩ is the empty sequence
    ⟨_⟩   : ⟪ (D ^ suc n) →ᶜ D * ⟫          -- ⟨ δ₁ , ... ⟩ is a non-empty sequence
    #     : ⟪ D * →ᶜ Nat⊥ ⟫                 -- # δ* is the length of sequence δ*
    _§_   : ⟪ D * →ᶜ D * →ᶜ D * ⟫           -- δ*₁ § δ*₂ is sequence concatenation
    _↓_   : ⟪ D * →ᶜ Nat →ˢ D ⟫             -- δ* ↓ n is the nth element
    _†_   : ⟪ D * →ᶜ Nat →ˢ D * ⟫           -- δ* † n is the nth tail
```

Scm

– from a published paper

Table 3. Scm: Abstract syntax

$\mathrm{Z} \in \mathrm{Int}$	integers
$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{I} \in \mathrm{Ide}$	identifiers
$\mathrm{E} \in \mathrm{Exp}$	expressions

$$\mathrm{Con} \longrightarrow \mathrm{Z} \mid \texttt{\#t} \mid \texttt{\#f}$$

$$\mathrm{Exp} \longrightarrow \mathrm{K} \mid \mathrm{I} \mid \texttt{(E E}^{*}\texttt{)} \mid \texttt{(lambda I E)} \mid \texttt{(if E E}_1\texttt{ E}_2\texttt{)} \mid \texttt{(set! I E)}$$

Table 4. Scm: Domain equations

$\alpha \in \mathbf{L}$	locations
$\nu \in \mathbf{N} = \mathrm{Nat}_{\perp}$	natural numbers
$\tau \in \mathbf{T} = \{\mathit{false}, \mathit{true}\}_{\perp}$	booleans
$\mathbf{R} = \mathrm{Int}_{\perp}$	integer numbers
$\mathbf{P} = \mathbf{L} \times \mathbf{L}$	pairs
$\mathbf{M} = \{\mathit{null}, \mathit{unallocated}, \mathit{undefined}, \mathit{unspecified}\}_{\perp}$	
$\mathbf{F} = \mathbf{E}^{*} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$	procedures
$\epsilon \in \mathbf{E} = \mathbf{T} + \mathbf{R} + \mathbf{P} + \mathbf{M} + \mathbf{F}$	expressed values
$\sigma \in \mathbf{S} = \mathbf{L} \to \mathbf{E}$	stores
$\rho \in \mathbf{U} = \mathrm{Ide} \to \mathbf{L}$	environments
$\theta \in \mathbf{C} = \mathbf{S} \to \mathbf{A}$	continuations
$\mathbf{A}$	answers

Scm

– in Agda

```
module Examples.Scm.Abstract-Syntax where

Ide = String       -- identifiers

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

mutual
  data Exp          : Set where             -- expressions
    con             : Con → Exp             -- constants
    ide             : Ide → Exp             -- identifiers
    (|_␣_|)         : Exp → Exp* → Exp      -- procedure application
    (|lambda_␣_|)   : Ide → Exp → Exp       -- procedure abstraction
    (|if_␣_␣_|)     : Exp → Exp → Exp → Exp -- conditional choice
    (|set!_␣_|)     : Ide → Exp → Exp       -- assignment
  data Exp*         : Set where             -- expression sequences
    ␣␣␣             : Exp*                  -- empty sequence
    _␣␣_            : Exp → Exp* → Exp*     -- sequence prefix
```

```
module Examples.Scm.Domain-Equations where

postulate Loc : Set
L  = Loc +⊥                -- locations
N  = Nat⊥                  -- natural numbers
T  = Bool⊥                 -- booleans
R  = Int +⊥                -- numbers
P  = L × L                 -- pairs
U  = Ide →ˢ L              -- environments
data Misc : Set where null unallocated undefined unspecified : Misc
M  = Misc +⊥               -- miscellaneous
postulate E : Domain       -- expressed values
S  = L →ᶜ E                -- stores
postulate A : Domain       -- answers
C  = S →ᶜ A                -- command continuations
F  = E * →ᶜ (E →ᶜ C) →ᶜ C  -- procedure values

postulate instance
  E+=T  : E ≳ 1 ↦ T
  E+=R  : E ≳ 2 ↦ R
  E+=P  : E ≳ 3 ↦ P
  E+=M  : E ≳ 4 ↦ M
  E+=F  : E ≳ 5 ↦ F
```

Scm

– from a published paper

$$\mathcal{K} : \text{Con} \to \mathbf{E}$$
$$\mathcal{E} : \text{Exp} \to \mathbf{U} \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$$
$$\mathcal{E}^* : \text{Exp}^* \to \mathbf{U} \to (\mathbf{E}^* \to \mathbf{C}) \to \mathbf{C}$$

$$\mathcal{K}[\![Z]\!] = Z \text{ in } \mathbf{E} \tag{1}$$
$$\mathcal{K}[\![\texttt{\#t}]\!] = \mathit{true} \text{ in } \mathbf{E} \tag{2}$$
$$\mathcal{K}[\![\texttt{\#f}]\!] = \mathit{false} \text{ in } \mathbf{E} \tag{3}$$

$$\mathcal{E}[\![\mathrm{K}]\!]\rho\kappa = \kappa(\mathcal{K}[\![\mathrm{K}]\!]) \tag{4}$$
$$\mathcal{E}[\![\mathrm{I}]\!]\rho\kappa = \mathit{hold}\,(\rho\,\mathrm{I})\,\kappa \tag{5}$$
$$\mathcal{E}[\![\texttt{(E E}^*\texttt{)}]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}]\!]\rho(\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\rho(\lambda\epsilon^*.\,(\epsilon \mid \mathbf{F})\,\epsilon^*\kappa)) \tag{6}$$
$$\mathcal{E}[\![\texttt{(lambda I E)}]\!]\rho\kappa = \kappa\,((\lambda\epsilon^*\kappa'.\,\mathit{list}\,\epsilon^*\,(\lambda\epsilon.\,\mathit{alloc}\,\epsilon\,(\lambda\alpha.\,\mathcal{E}[\![\mathrm{E}]\!]\,(\rho[\alpha/\mathrm{I}])\,\kappa'))\,)\text{ in }\mathbf{E}) \tag{7}$$

$$\mathcal{E}[\![\texttt{(if E E}_1\texttt{ E}_2\texttt{)}]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\lambda\epsilon.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\,\rho\,\kappa, \mathcal{E}[\![\mathrm{E}_2]\!]\,\rho\,\kappa) \tag{8}$$
$$\mathcal{E}[\![\texttt{(set! I E)}]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\lambda\epsilon.\,\mathit{assign}\,(\rho\,\mathrm{I})\,\epsilon\,(\kappa\,\mathit{unspecified})) \tag{9}$$

$$\mathcal{E}^*[\![\;\;]\!]\rho\kappa = \kappa\,\langle\,\rangle \tag{10}$$
$$\mathcal{E}^*[\![\mathrm{E}\;\;\mathrm{E}^*]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\lambda\epsilon.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*.\,\kappa\,(\langle\epsilon\rangle\;\S\;\epsilon^*))) \tag{11}$$

Scm

– in Agda

```
module Examples.Scm.Semantic-Functions where

𝒦⟦_⟧  :  ⟪ Con →ˢ E ⟫                        -- constant denotations
ℰ⟦_⟧  :  ⟪ Exp →ˢ U →ᶜ (E →ᶜ C) →ᶜ C ⟫       -- expression denotations
ℰ*⟦_⟧ :  ⟪ Exp* →ˢ U →ᶜ (E * →ᶜ C) →ᶜ C ⟫    -- sequence denotations

𝒦⟦ int Z ⟧ = ↑ Z in⊥ E
𝒦⟦ #t ⟧    = ↑ true in⊥ E
𝒦⟦ #f ⟧    = ↑ false in⊥ E

ℰ⟦ con K ⟧ ρ κ              = κ (𝒦⟦ K ⟧)
ℰ⟦ ide I ⟧ ρ κ              = hold (ρ I) κ
ℰ⟦ ⦇ E ␣ E* ⦈ ⟧ ρ κ          = ℰ⟦ E ⟧ ρ (λ ϵ → ℰ*⟦ E* ⟧ ρ (λ ϵ* → (ϵ |⊥ F) ϵ* κ))
ℰ⟦ ⦇lambda I ␣ E ⦈ ⟧ ρ κ    = κ ( (λ ϵ* κ′ →
                                   list ϵ* (λ ϵ → alloc ϵ (λ α → ℰ⟦ E ⟧ (ρ [ α / I ]) κ′))
                                 ) in⊥ E )
ℰ⟦ ⦇if E ␣ E₁ ␣ E₂ ⦈ ⟧ ρ κ = ℰ⟦ E ⟧ ρ (λ ϵ → truish ϵ ⟶ ℰ⟦ E₁ ⟧ ρ κ , ℰ⟦ E₂ ⟧ ρ κ)
ℰ⟦ ⦇set! I ␣ E ⦈ ⟧ ρ κ      = ℰ⟦ E ⟧ ρ (λ ϵ → assign (ρ I) ϵ (κ (↑ unspecified in⊥ E)))

ℰ*⟦ ␣␣␣ ⟧ ρ κ       = κ ⟨⟩
ℰ*⟦ E ␣␣ E* ⟧ ρ κ = ℰ⟦ E ⟧ ρ (λ ϵ → ℰ*⟦ E* ⟧ ρ (λ ϵ* → κ (⟨ ϵ ⟩ § ϵ*)))
```