Denotational Semantics of PCF in Agda

DRAFT (March 10, 2025)

Peter D. Mosses

Delft University of Technology, The Netherlands
p.d.mosses@tudelft.nl
Swansea University, United Kingdom

Abstract

In synthetic domain theory, all sets are predomains, domains are pointed sets, and functions are implicitly continuous. The denotational semantics of PCF (Plotkin's version) presented here illustrates how it might look if synthetic domain theory can be implemented in Agda. As a work-around, the code presented here uses unsatisfiable postulates to allow Agda to type-check the definitions.

The (currently illiterate) Agda source code used to generate this document can be downloaded from https://github.com/pdmosses/xds-agda, and browsed with hyperlinks and highlighting at https://pdmosses.github.io/xds-agda/.

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

module PCF.All where

import PCF.Domain-Notation
import PCF.Types
import PCF.Constants
import PCF.Variables
import PCF.Environments
import PCF.Terms
import PCF.Checks
```

```
module PCF.Domain-Notation where

open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl) public

Domain = Set
variable D E : Domain

-- Domains are pointed
postulate
  ⊥        : {D : Domain} → D

-- Fixed points of endofunctions on function domains

postulate
  fix      : {D : Domain} → (D → D) → D

  -- Properties
  fix-fix  : ∀ {D} (f : D → D) →
               fix f ≡ f (fix f)
  fix-app : ∀ {P D} (f : (P → D) → (P → D)) (p : P) →
               fix f p ≡ f (fix f) p

-- Lifted domains

postulate
  𝕃        : Set → Domain
  η        : {P : Set} → P → 𝕃 P
  _♯       : {P : Set} {D : Domain} → (P → D) → (𝕃 P → D)

  -- Properties
  elim-♯-η  : ∀ {P D} (f : P → D) (p : P) →
              (f ♯) (η p) ≡ f p
  elim-♯-⊥ : ∀ {P D} (f : P → D) →
              (f ♯) ⊥ ≡ ⊥

-- Flat domains

_+⊥ : Set → Domain
S +⊥ = 𝕃 S

-- McCarthy conditional

-- t ⟶⊥ d₁ , d₂ : D  (t : Bool +⊥ ; d₁, d₂ : D)

open import Data.Bool.Base
  using (Bool; true; false; if_then_else_) public

postulate
  _⟶⊥_,_ : {D : Domain} → Bool +⊥ → D → D → D

  -- Properties
  true-cond     : ∀ {D} {d₁ d₂ : D} → (η true ⟶⊥ d₁ , d₂) ≡ d₁
  false-cond    : ∀ {D} {d₁ d₂ : D} → (η false ⟶⊥ d₁ , d₂) ≡ d₂
  bottom-cond : ∀ {D} {d₁ d₂ : D} → (⊥ ⟶⊥ d₁ , d₂)        ≡ ⊥
```

```
module PCF.Types where

open import Data.Bool.Base
  using (Bool)
open import Agda.Builtin.Nat
  using (Nat)

open import PCF.Domain-Notation
  using (Domain; _+⊥)

-- Syntax

data Types : Set where
  ι     : Types                    -- natural numbers
  o     : Types                    -- Boolean truthvalues
  _⇒_ : Types → Types → Types -- functions

variable σ τ : Types

infixr 1 _⇒_

-- Semantics 𝒟

𝒟 : Types → Domain

𝒟 ι         = Nat +⊥
𝒟 o         = Bool +⊥
𝒟 (σ ⇒ τ) = 𝒟 σ → 𝒟 τ

variable x y z : 𝒟 σ
```

```
module PCF.Constants where

open import Data.Bool.Base
  using (Bool; true; false; if_then_else_)
open import Agda.Builtin.Nat
  using (Nat; _+_; _-_; _==_)

open import PCF.Domain-Notation
  using (η; _♯; fix; ⊥; _⟶⊥_,_)
open import PCF.Types
  using (Types; o; ι; _⇒_; σ; 𝒟)

-- Syntax

data ℒ : Types → Set where
  tt   : ℒ o
  ff   : ℒ o
  ⊃ᵢ   : ℒ (o ⇒ ι ⇒ ι ⇒ ι)
  ⊃ₒ   : ℒ (o ⇒ o ⇒ o ⇒ o)
  Y    : {σ : Types} → ℒ ((σ ⇒ σ) ⇒ σ)
  k    : (n : Nat) → ℒ ι
  +1′  : ℒ (ι ⇒ ι)
  -1′  : ℒ (ι ⇒ ι)
  Z    : ℒ (ι ⇒ o)

variable c : ℒ σ

-- Semantics

𝒜⟦_⟧ : ℒ σ → 𝒟 σ

𝒜⟦ tt  ⟧ = η true
𝒜⟦ ff  ⟧ = η false
𝒜⟦ ⊃ᵢ  ⟧ = _⟶⊥_,_
𝒜⟦ ⊃ₒ  ⟧ = _⟶⊥_,_
𝒜⟦ Y   ⟧ = fix
𝒜⟦ k n ⟧ = η n
𝒜⟦ +1′ ⟧ = (λ n → η (n + 1)) ♯
𝒜⟦ -1′ ⟧ = (λ n → if n == 0 then ⊥ else η (n - 1)) ♯
𝒜⟦ Z   ⟧ = (λ n → η (n == 0)) ♯
```

```
module PCF.Variables where

open import Agda.Builtin.Nat
  using (Nat)

open import PCF.Types
  using (Types; σ; 𝒟)

-- Syntax

data 𝒱 : Types → Set where
  var : Nat → (σ : Types) → 𝒱 σ

variable α : 𝒱 σ

-- Environments

Env = ∀ {σ} → 𝒱 σ → 𝒟 σ

variable ρ : Env

-- Semantics

_⟦_⟧ : Env → 𝒱 σ → 𝒟 σ

ρ ⟦ α ⟧ = ρ α
```

```
module PCF.Environments where

open import Data.Bool.Base
  using (Bool; if_then_else_)
open import Data.Maybe.Base
  using (Maybe; just; nothing)
open import Agda.Builtin.Nat
  using (Nat; _==_)
open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl; trans; cong)

open import PCF.Domain-Notation
  using (⊥)
open import PCF.Types
  using (Types; ι; o; _⇒_; 𝒟)
open import PCF.Variables
  using (𝒱; var; Env)

-- ρ⊥ is the initial environment

ρ⊥ : Env
ρ⊥ α = ⊥

-- (ρ [ x / α ]) α′ = x when α and α′ are identical, otherwise ρ α′

_[_/_] : {σ : Types} → Env → 𝒟 σ → 𝒱 σ → Env
ρ [ x / α ] = λ α′ → h ρ x α α′ (α ==V α′) where

  h : {σ τ : Types} → Env → 𝒟 σ → 𝒱 σ → 𝒱 τ → Maybe (σ ≡ τ) → 𝒟 τ
  h ρ x α α′ (just refl) = x
  h ρ x α α′ nothing    = ρ α′

  _==T_ : (σ τ : Types) → Maybe (σ ≡ τ)
  (σ ⇒ τ) ==T (σ′ ⇒ τ′) = f (σ ==T σ′) (τ ==T τ′) where
          f : Maybe (σ ≡ σ′) → Maybe (τ ≡ τ′) → Maybe ((σ ⇒ τ) ≡ (σ′ ⇒ τ′))
          f = λ { (just p) (just q) → just (trans (cong (_⇒ τ) p) (cong (σ′ ⇒_) q))
                ; _ _ → nothing }
  ι  ==T ι  = just refl
  o  ==T o  = just refl
  _  ==T _  = nothing

  _==V_ : {σ τ : Types} → 𝒱 σ → 𝒱 τ → Maybe (σ ≡ τ)
  var i σ ==V var i′ τ =
    if i == i′ then σ ==T τ else nothing
```

```
module PCF.Terms where

open import PCF.Types
  using (Types; _⇒_; σ; 𝒟)
open import PCF.Constants
  using (ℒ; 𝒜⟦_⟧; c)
open import PCF.Variables
  using (𝒱; Env; _⟦_⟧)
open import PCF.Environments
  using (_[_/_])

-- Syntax

data Terms : Types → Set where
  V     : {σ : Types} → 𝒱 σ → Terms σ                               -- variables
  L     : {σ : Types} → ℒ σ → Terms σ                               -- constants
  _˜_   : {σ τ : Types} → Terms (σ ⇒ τ) → Terms σ → Terms τ -- application
  ƛ_˜_  : {σ τ : Types} → 𝒱 σ → Terms τ → Terms (σ ⇒ τ)       -- λ-abstraction

variable M N : Terms σ
infixl 20 _˜_

-- Semantics

𝒜′⟦_⟧ : Terms σ → Env → 𝒟 σ

𝒜′⟦ V α     ⟧ ρ = ρ ⟦ α ⟧
𝒜′⟦ L c     ⟧ ρ = 𝒜⟦ c ⟧
𝒜′⟦ M ˜ N   ⟧ ρ = 𝒜′⟦ M ⟧ ρ (𝒜′⟦ N ⟧ ρ)
𝒜′⟦ ƛ α ˜ M ⟧ ρ = λ x → 𝒜′⟦ M ⟧ (ρ [ x / α ])
```

```
{-# OPTIONS --rewriting --confluence-check #-}
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite

module PCF.Checks where

open import Data.Bool.Base
open import Agda.Builtin.Nat
open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl)

open import PCF.Domain-Notation
open import PCF.Types
open import PCF.Constants
open import PCF.Variables
open import PCF.Environments
open import PCF.Terms

postulate
  {-# REWRITE fix-app elim-♯-η elim-♯-⊥ true-cond false-cond #-}

-- Constants
pattern N n    = L (k n)
pattern succ   = L +1′
pattern pred⊥ = L -1′
pattern if     = L ⊃ᵢ
pattern Y      = L Y
pattern Z = L Z

-- Variables
f = var 0 ι
g = var 1 (ι ⇒ ι)
h = var 2 (ι ⇒ ι ⇒ ι)
a = var 3 ι
b = var 4 ι

-- Arithmetic
check-41+1 : 𝒜′⟦ succ ˜ N 41 ⟧ ρ⊥ ≡ η 42
check-41+1 = refl

check-43-1 : 𝒜′⟦ pred⊥ ˜ N 43 ⟧ ρ⊥ ≡ η 42
check-43-1 = refl

-- Binding
check-id : 𝒜′⟦ (ƛ a ˜ V a) ˜ N 42 ⟧ ρ⊥ ≡ η 42
check-id = refl

check-k : 𝒜′⟦ (ƛ a ˜ ƛ b ˜ V a) ˜ N 42 ˜ N 41 ⟧ ρ⊥ ≡ η 42
check-k = refl

check-ki : 𝒜′⟦ (ƛ a ˜ ƛ b ˜ V b) ˜ N 41 ˜ N 42 ⟧ ρ⊥ ≡ η 42
check-ki = refl
```

```
check-suc-41 : 𝒜′⟦ (ƛ a ˜ (succ ˜ V a )) ˜ N 41 ⟧ ρ⊥ ≡ η 42
check-suc-41 = refl

check-pred-42 : 𝒜′⟦ (ƛ a ˜ (pred⊥ ˜ V a)) ˜ N 43 ⟧ ρ⊥ ≡ η 42
check-pred-42 = refl

check-if-zero : 𝒜′⟦ if ˜ (Z ˜ N 0) ˜ N 42 ˜ N 0 ⟧ ρ⊥ ≡ η 42
check-if-zero = refl

check-if-nonzero : 𝒜′⟦ if ˜ (Z ˜ N 42) ˜ N 0 ˜ N 42 ⟧ ρ⊥ ≡ η 42
check-if-nonzero = refl

-- fix (λf. 42) ≡ 42
check-fix-const :
  𝒜′⟦ Y ˜ (ƛ f ˜ N 42) ⟧ ρ⊥
  ≡ η 42
check-fix-const = fix-fix (λ x → η 42)

-- fix (λg. λa. 42) 2 ≡ 42
check-fix-lambda :
  𝒜′⟦ Y ˜ (ƛ g ˜ ƛ a ˜ N 42) ˜ N 2 ⟧ ρ⊥
  ≡ η 42
check-fix-lambda = refl

-- fix (λg. λa. ifz a then 42 else g (pred a)) 101 ≡ 42
check-countdown :
  𝒜′⟦ Y ˜ (ƛ g ˜ ƛ a ˜
                 (if ˜ (Z ˜ V a) ˜ N 42 ˜ (V g ˜ (pred⊥ ˜ V a))))
       ˜ N 101
    ⟧ ρ⊥
  ≡ η 42
check-countdown = refl

-- fix (λh. λa. λb. ifz a then b else h (pred a) (succ b)) 4 38 ≡ 42
check-sum-42 :
  𝒜′⟦ (Y ˜ (ƛ h ˜ ƛ a ˜ ƛ b ˜
                 (if ˜ (Z ˜ V a) ˜ V b ˜ (V h ˜ (pred⊥ ˜ V a) ˜ (succ ˜ V b)))))
       ˜ N 4 ˜ N 38
    ⟧ ρ⊥
  ≡ η 42
check-sum-42 = refl
-- Exponential in first arg?
```