

## Lightweight Agda Formalization of Denotational Semantics

Peter D Mosses

TU Delft (visitor)  
Swansea University (emeritus)

TYPES 2025, Glasgow, Scotland, 9–13 June 2025

1

Thank you for still being here!

This is a lightweight talk about work in progress

### About the topic

– Lightweight Agda Formalization of Denotational Semantics

#### Lightweight Agda

- requiring *relatively little effort* or Agda *expertise*

#### Formalization

- of (new or existing) *mathematical* definitions

#### Denotational semantics

- with *recursively-defined Scott-domains, fixed points,  $\lambda$ -notation*

2

Let me start by clarifying the topic of the talk

### Original motivation

A Denotational Semantics of Inheritance and its Correctness



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 modification of the  
principle 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. . . .

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

3

This denotational semantics was published in 1989, and hadn't been mechanically checked.

I expected it to be quite straightforward to formulate the definitions and proofs in Agda...

### Denotational semantics

– Scott–Strachey style



Types of denotations are (Scott-)domains

- *pointed cpos* (e.g.,  $\omega$ -complete, directed-complete, continuous lattices)
- *recursively defined* – without guards, up to isomorphism

Denotations are defined in typed  $\lambda$ -notation

- functions on domains are *continuous maps*
- endofunctions on domains have least *fixed points*

4

Denotational semantics isn't as popular these days as it was in the 70s and 80s

Let me briefly recall the main features

## Models of the untyped $\lambda$ -calculus

- based on Scott's domain  $D_\infty$

Some mathematical presentations:

- Dana Scott (1970,1972): continuous lattices,  $D_\infty$
- Joseph Stoy (1977): universal domain  $\mathcal{P}\omega$
- Samson Abramsky and Achim Jung (1994): (pre)domain theory
- John Reynolds (2009): *Theories of Programming Languages*, cpos,  $D_\infty$

Some formalizations:

- Bernhard Reus (1994): using Extended Calculus of Constructions, in Lego
- Tom de Jong (2021): using Univalent Type Theory (TypeTopology), in Agda

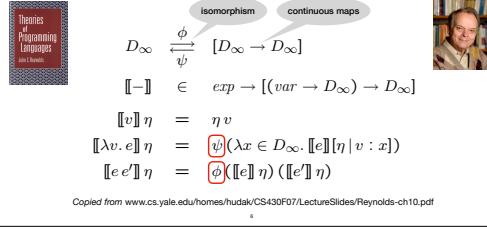
5

Dana Scott initially tried to prove that the untyped lambda-calculus has no set-theoretic models  
but then he discovered the bilimit construction of the D-infinity model...

Apart from mathematical presentations of the model, some formalizations have been developed

## Reynolds: Theories of Programming Languages

- denotational semantics of the untyped  $\lambda$ -calculus



Copied from [www.cs.yale.edu/homes/hudak/CS430F07/lectureSlides/Reynolds-ch10.pdf](http://www.cs.yale.edu/homes/hudak/CS430F07/lectureSlides/Reynolds-ch10.pdf)

6

Here is a mathematical presentation of a denotational semantics of the untyped lambda-calculus from 2009

In denotational semantics of larger programming languages, isomorphisms are usually left implicit

## Agda formalization

- using TypeTopology/DomainTheory (Tom de Jong)

We have the non-trivial domain  $D_\infty$  and isomorphism  $D_\infty \sim^{\text{dcpo}} (D_\infty \Rightarrow^{\text{dcpo}} D_\infty)$ .

$\text{abs} : (\mathcal{D}_\infty \Rightarrow^{\text{dcpo}} \mathcal{D}_\infty) \rightarrow (\mathcal{D}_\infty)$   
 $\text{abs} = [\mathcal{D}_\infty \Rightarrow^{\text{dcpo}} \mathcal{D}_\infty, \mathcal{D}_\infty](\pi\text{-}\exp_\infty')$

$\text{app} : (\mathcal{D}_\infty) \rightarrow (\mathcal{D}_\infty) \rightarrow (\mathcal{D}_\infty)$   
 $\text{app} = \text{underlying-function } \mathcal{D}_\infty \mathcal{D}_\infty \circ [\mathcal{D}_\infty, \mathcal{D}_\infty \Rightarrow^{\text{dcpo}} \mathcal{D}_\infty](\varepsilon\text{-}\exp_\infty')$

a continuous function is a pair:  
- an underlying function and  
- a proof of its continuity

7

Tom de Jong's formalization is based on directed-complete posets.

## Agda formalization

- using TypeTopology/DomainTheory (Tom de Jong)

$\llbracket \_ \rrbracket : \mathbb{E} \rightarrow \text{Env} \rightarrow (\mathcal{D}_\infty)$   
 $\lambda\text{-is-continuous} : \forall e p v \rightarrow \text{is-continuous } \mathcal{D}_\infty \mathcal{D}_\infty (\lambda x \rightarrow \llbracket e \rrbracket(p[x/v]))$

$\llbracket \text{var } v \rrbracket p = \rho v$   
 $\llbracket \lambda v \cdot e \rrbracket p = \text{abs}((\lambda x \rightarrow \llbracket e \rrbracket(p[x/v])), \lambda\text{-is-continuous } e p v)$   
 $\llbracket e_1 \cdot e_2 \rrbracket p = \text{app}(\llbracket e_1 \rrbracket p)(\llbracket e_2 \rrbracket p)$

$\lambda\text{-is-continuous } e p v = \{!\}$

8

The proof of the proposition that lambda is continuous isn't very deep, but takes 3 pages in John Reynolds book

I would personally find it an excessive amount of work to formalize the proof in Agda...

## Lightweight Agda formalization

- modules

### Abstract syntax grammar

- inductive **datatype definitions**
- 'Domain' definitions
  - *postulated isomorphisms* between *type names* and *type terms*
- Semantic functions
  - functions defined *inductively* in  $\lambda$ -*notation*
- Auxiliary definitions

9

... so I've developed a lightweight approach

The next few slides illustrate the approach. The Agda definitions are included in the abstract, and available online

## Lightweight Agda formalization

- abstract syntax

```
data Exp : Set where
  var_ : Var → Exp
  lam : Var → Exp → Exp
  app : Exp → Exp → Exp
```

## Lightweight Agda formalization

- a 'domain'

```
open import Function
using (Inverse; _ $\leftrightarrow$ _) public
open Inverse {{ ... }} public
using (to; from) public

postulate
  D $_{\infty}$  : Set
postulate
  instance iso : D $_{\infty}$   $\leftrightarrow$  (D $_{\infty}$   $\rightarrow$  D $_{\infty}$ )
```

10

In Scott–Strachey style, abstract syntax is defined by a context-free grammar. Its formalization in Agda is a corresponding inductive datatype.

For simplicity, this datatype uses ordinary functional notation for the constructors.

11

Here we assume D-infinity to be an ordinary Agda type with a bijection to the type of all Agda functions on D-infinity

In this example, the assumptions are satisfied when D-infinity has a single element.

In all other examples, the corresponding assumptions are unsatisfiable.

The highlighted Agda magic declares the inverse functions of the bijection, which are all we need...

## Lightweight Agda formalization

- semantic function

$\text{Env} = \text{Var} \rightarrow D_\infty$

$\llbracket \_ \rrbracket : \text{Exp} \rightarrow \text{Env} \rightarrow D_\infty$

$\begin{aligned} \llbracket \text{var } v \rrbracket \rho &= \rho v \\ \llbracket \text{lam } v e \rrbracket \rho &= \text{from}(\lambda d \rightarrow \llbracket e \rrbracket (\rho[d/v])) \\ \llbracket \text{app } e_1 e_2 \rrbracket \rho &= \text{to}(\llbracket e_1 \rrbracket \rho)(\llbracket e_2 \rrbracket \rho) \end{aligned}$

$$\begin{array}{c} D_\infty \xrightarrow{\phi} [D_\infty \rightarrow D_\infty] \\ \llbracket \_ \rrbracket \in \text{exp} \rightarrow [(var \rightarrow D_\infty) \rightarrow D_\infty] \\ \llbracket \_ \rrbracket \eta = \eta v \\ \llbracket \lambda x. \_ \rrbracket \eta = \psi(x \in D_\infty. \llbracket \_ \rrbracket \eta[v : x]) \\ \llbracket \_ \rrbracket \eta = \phi(\llbracket \_ \rrbracket \eta)(\llbracket \_ \rrbracket \eta) \end{array}$$

12

The semantic function corresponds directly to that defined by Reynolds!

The Agda type-checker insists on making the bijection explicit

## Lightweight Agda formalization

- testing denotations

$\text{to-from-elim} : \forall \{f\} \rightarrow \text{to}(\text{from } f) \equiv f$

$\text{to-from-elim} = \text{inverse}' \text{ iso refl}$

{-# REWRITE to-from-elim #-}

$\begin{aligned} \text{check-convergence} : & (\lambda x_1. x_{42})(\lambda x_0. x_0 x_0)(\lambda x_0. x_0 x_0) \equiv x_{42} \\ & \llbracket \text{app } (\text{lam } (x 1) (\text{var } x 42)) \\ & \quad (\text{app } (\text{lam } (x 0) (\text{app } (\text{var } x 0) (\text{var } x 0))) \\ & \quad (\text{lam } (x 0) (\text{app } (\text{var } x 0) (\text{var } x 0)))) \rrbracket \\ & \equiv \llbracket \text{var } x 42 \rrbracket \\ \text{check-convergence} & = \text{refl} \quad - \text{potentially unsafe!} \end{aligned}$

13

Agda can also check whether terms have equivalent denotations

## Other examples: PCF, Scheme

- pdmosses.github.io/xds-agda/



14

## Safe lightweight Agda formalization?

- future work

### Implement SDT (Synthetic Domain Theory)

- use *plain* Agda
- embed Agda types as *predomains*
- assume only properties **consistent** with MLTT
- make functions **implicitly** continuous
- allow **unrestricted** recursive domain definitions
- ...

15