Lightweight Formalisation of Denotational Semantics in AGDA

Peter D Mosses

TU Delft (visitor)
Swansea University (emeritus)

FP Day, Nijmegen, Netherlands, 9 January 2026

How many of you are AGDA users?

Lightweight Formalisation of Denotational Semantics

– about the topic

Formalisation

- of (new or existing) ***mathematical*** definitions

Denotational semantics

- with ***recursively-defined Scott-domains***, ***fixed points***, ***λ-notation***

Lightweight

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

Lightweight Formalisation of Denotational Semantics

– about the talk

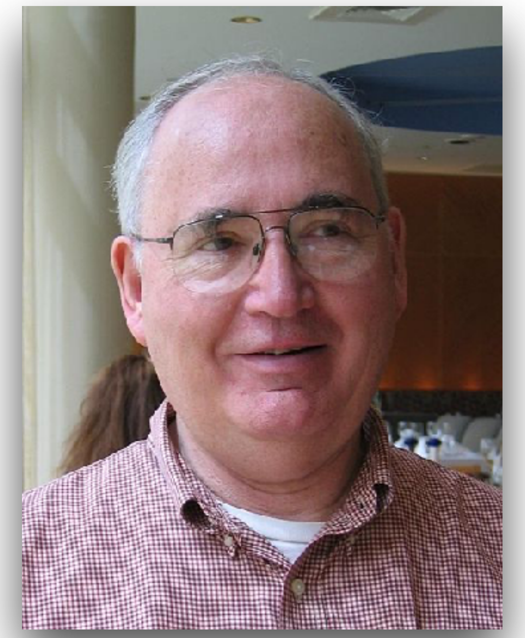
Examples

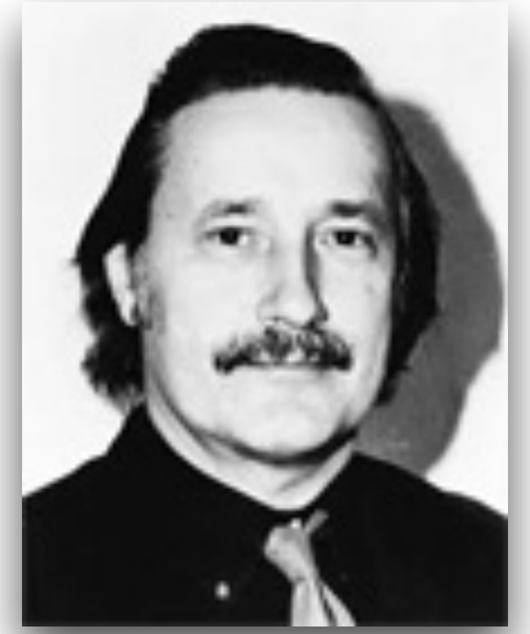
- inheritance
- the untyped λ-calculus
- SCHEME

Postulating domain theory

- lightweight
- synthetic

Denotational semantics
– Scott–Strachey style





Types of denotations are (Scott-)domains

- ***pointed cpos*** (e.g, ω-complete, directed-complete, continuous lattices)
- ***recursively defined*** (up to isomorphism)
- ***domain constructors*** (functions, products, sums, …)

Denotations are defined in typed λ-notation

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

Inheritance

Original motivation for lightweight formalisation

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

Formalisation of a Denotational Semantics of Inheritance

– AGDA code: GitHub repo pdmosses/jensfest-agda/

Quite clumsy

- my very first attempt to use AGDA (2024)
- domain equations: domains ***assumed isomorphic*** to their structure
- functions on domains: defined in λ-notation, ***assumed continuous***
- all assumptions declared as ***module parameters***

Encouraging results

- detected several (minor) issues – including the omission of a projection

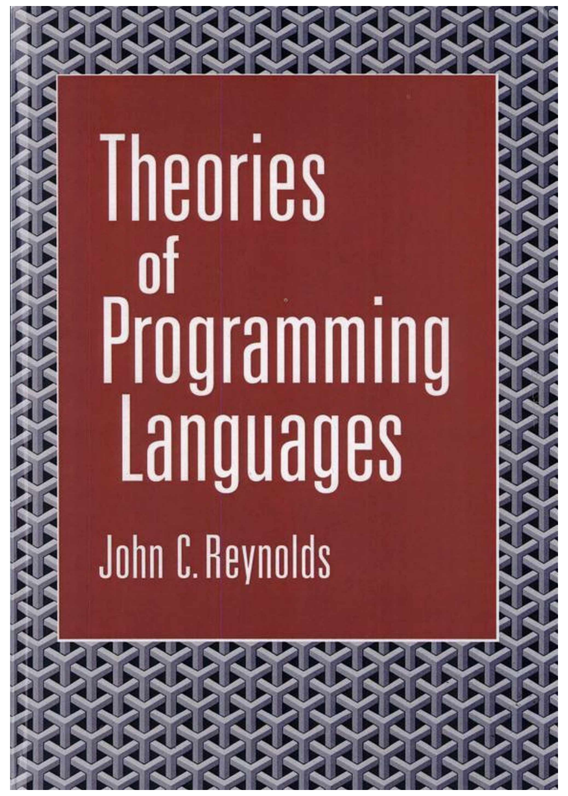
The untyped λ-calculus

Models of the untyped λ-calculus

Some mathematical presentations:

- Dana Scott (1970): *Outline of a Mathematical Theory of Computation*
  - complete lattices
- Samson Abramsky and Achim Jung (1994): *Domain Theory*
  - directed-complete posets (dcpos)
- John Reynolds (2009): *Theories of Programming Languages*
  - ω-complete posets (ω-cpos)

Denotational semantics of the untyped λ-calculus



isomorphism

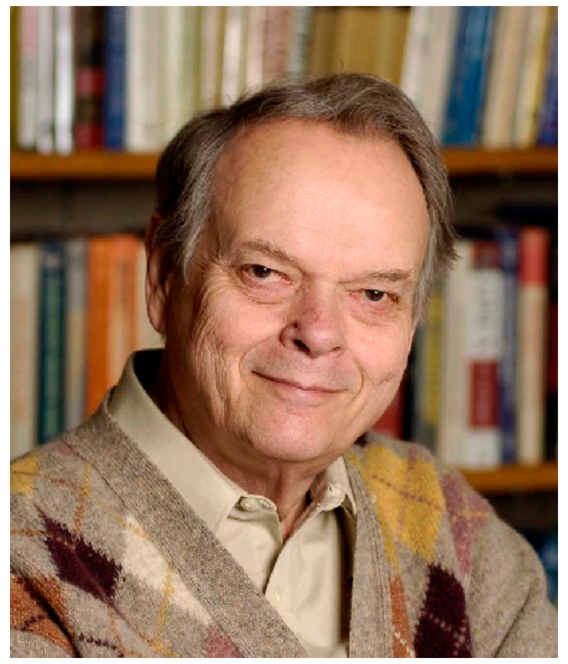
continuous maps

$$D_\infty \underset{\psi}{\overset{\phi}{\rightleftarrows}} [D_\infty \to D_\infty]$$

continuous maps

$$[\![-]\!] \in exp \to [(var \to D_\infty) \to D_\infty]$$

$$\begin{aligned}
[\![v]\!]\,\eta &= \eta\, v \\
[\![\lambda v.\, e]\!]\,\eta &= \boxed{\psi}(\lambda x \in D_\infty.\, [\![e]\!][\eta \mid v : x]) \\
[\![e\, e']\!]\,\eta &= \boxed{\phi}([\![e]\!]\,\eta)\,([\![e']\!]\,\eta)
\end{aligned}$$



Copied from **www.cs.yale.edu/homes/hudak/CS430F07/LectureSlides/Reynolds-ch10.pdf**

Models of the untyped λ-calculus

Some formalisations:

- Bernhard Reus (1999): *Formalizing Synthetic Domain Theory*
  - using *Extended Calculus of Constructions*, defined in *LEGO*
- Tom de Jong (2021): *TypeTopology/DomainTheory*
  - using *Univalent Type Theory*, defined in *AGDA*

Analytic formalisation in AGDA

Analytic formalisation in AGDA

– using TypeTopology/DomainTheory

Definitions

- a ***domain*** D is a ***tuple*** ($\langle D \rangle$, $\sqsubseteq$, $\bot$, *proof*)
  - such that *proof* : "($\langle D \rangle$, $\sqsubseteq$, $\bot$) is a pointed dcpo"
- a ***continuous function*** between domains is a ***pair*** $(f : \langle D \rangle \to \langle E \rangle$, *proof*)
  - such that *proof* : "f preserves suprema of directed sets"
- collections of ***recursively-defined domains*** are ***bilimits*** of diagrams
  - e.g., $D_\infty = [D_\infty \to D_\infty]$, up to isomorphism

Analytic formalisation in AGDA
– using TypeTopology/DomainTheory

We have the non-trivial domain $\mathcal{D}_\infty$ and isomorphism $\mathcal{D}_\infty \sim^{\text{dcpo}} (\mathcal{D}_\infty \Longrightarrow^{\text{dcpo}} \mathcal{D}_\infty)$.

abs : $\langle\, \mathcal{D}_\infty \Longrightarrow^{\text{dcpo}} \mathcal{D}_\infty \,\rangle \to \langle\, \mathcal{D}_\infty \,\rangle$

app : $\langle\, \mathcal{D}_\infty \,\rangle \to \langle\, \mathcal{D}_\infty \,\rangle \to \langle\, \mathcal{D}_\infty \,\rangle$

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

Analytic formalisation in AGDA

– using TypeTopology/DomainTheory

We have the non-trivial domain $\mathcal{D}_\infty$ and isomorphism $\mathcal{D}_\infty \simeq^{\text{dcpo}} (\mathcal{D}_\infty \Longrightarrow^{\text{dcpo}} \mathcal{D}_\infty)$.

abs : $\langle\ \mathcal{D}_\infty \Longrightarrow^{\text{dcpo}} \mathcal{D}_\infty\ \rangle \to \langle\ \mathcal{D}_\infty\ \rangle$
abs = $[\ \mathcal{D}_\infty \Longrightarrow^{\text{dcpo}} \mathcal{D}_\infty\ ,\ \mathcal{D}_\infty\ ]\langle\ \pi\text{-exp}_\infty'\ \rangle$

app : $\langle\ \mathcal{D}_\infty\ \rangle \to \langle\ \mathcal{D}_\infty\ \rangle \to \langle\ \mathcal{D}_\infty\ \rangle$
app = underlying-function $\mathcal{D}_\infty\ \mathcal{D}_\infty \circ [\ \mathcal{D}_\infty\ ,\ \mathcal{D}_\infty \Longrightarrow^{\text{dcpo}} \mathcal{D}_\infty\ ]\langle\ \varepsilon\text{-exp}_\infty'\ \rangle$

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

Analytic formalisation in AGDA

– using TypeTopology/DomainTheory

$⟦\_⟧ : \mathsf{Exp} \to \mathsf{Env} \to \langle \mathcal{D}_\infty \rangle$

$ƛ\text{-is-continuous} : \forall\, e\, \rho\, v \to \text{is-continuous}\ \mathcal{D}_\infty\ \mathcal{D}_\infty\ (\lambda\, x \to ⟦ e ⟧\ (\rho\ [\, x\, /\, v\,]))$

$⟦\ \mathsf{var}\ v\ ⟧\ \rho = \rho\ v$

$⟦\ ƛ\ v \cdot e\ ⟧\ \rho = \mathsf{abs}\ (\ (\lambda\, x \to ⟦ e ⟧\ (\rho\ [\, x\, /\, v\,]))\ ,\ ƛ\text{-is-continuous}\ e\ \rho\ v)$

$⟦\ e_1 \cdot e_2\ ⟧\ \rho = \mathsf{app}\ (\ ⟦\ e_1\ ⟧\ \rho\ )\ (\ ⟦\ e_2\ ⟧\ \rho\ )$

$ƛ\text{-is-continuous}\ e\ \rho\ v = \{!\ \ !\}$

The proof of the proposition isn't very deep –
but it takes 3 pages in John Reynolds's book…

λ-abstractions in continuation-passing style

– e.g., in the SCHEME language standards

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (I*)}\ \Gamma\texttt{*}\ \texttt{E}_0\texttt{)}]\!] = \\
\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,. \\
\qquad \mathit{new}\,\sigma \in \texttt{L} \rightarrow \\
\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \,|\, \texttt{L}, \\
\qquad\qquad\qquad \lambda\epsilon{*}\omega'\kappa'\,.\,\#\epsilon{*} = \#\mathrm{I}{*} \rightarrow \\
\qquad\qquad\qquad\quad \mathit{tievals}(\lambda\alpha{*}\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma{*}]\!]\rho'\omega'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\omega'\kappa')) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}{*}\,\alpha{*})) \\
\qquad\qquad\qquad\qquad\qquad \epsilon{*}, \\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}\rangle \\
\qquad\qquad\qquad \text{in}\ \texttt{E}) \\
\qquad\qquad \kappa \\
\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \,|\, \texttt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}
$$

Lightweight formalisation in AGDA

Lightweight formalisation in AGDA

Abstract syntax grammar

- inductive ***datatype definitions***

'Domain' definitions

- ***postulated bijections*** between ***type names*** and ***type terms***

Semantic functions

- defined ***inductively*** in ***λ-notation***

Auxiliary definitions

Lightweight formalisation in AGDA
– abstract syntax for the untyped λ-calculus

```
module LC.Terms where

open import LC.Variables

data Exp : Set where
  var_  : Var → Exp             -- variable value
  lam   : Var → Exp → Exp       -- lambda abstraction
  app   : Exp → Exp → Exp       -- application

variable e : Exp
```

Lightweight formalisation in AGDA
– postulating a domain for the untyped λ-calculus

```
module LC.Domains where

postulate
  Domain   : Set₁                  -- type of all domains
  《_》      : Domain → Set         -- carrier of a domain

open import Function    using (Inverse; _↔_)  public
open Inverse {{ ... }}  using (to; from)       public

postulate
  D∞   : Domain

postulate instance
  bi   : 《 D∞ 》 ↔ (《 D∞ 》 → 《 D∞ 》)

variable d : 《 D∞ 》
```

Lightweight formalisation in AGDA

– semantic function for the untyped λ-calculus

```
module LC.Semantics where

open import LC.Variables
open import LC.Terms
open import LC.Domains
open import LC.Environments

⟦_⟧ : Exp → Env → 《 D∞ 》
-- ⟦ e ⟧ ρ is the value of e with ρ giving the values of free variables

⟦ var   v       ⟧ ρ   = ρ v
⟦ lam   v e     ⟧ ρ   = from ( λ d → ⟦ e ⟧ (ρ [ d / v ]) )
⟦ app   e₁ e₂   ⟧ ρ   = to ( ⟦ e₁ ⟧ ρ ) ( ⟦ e₂ ⟧ ρ )
```

$$D_\infty \underset{\psi}{\overset{\phi}{\rightleftarrows}} [D_\infty \to D_\infty]$$

$$\begin{aligned}
[\![-]\!] &\in exp \to [(var \to D_\infty) \to D_\infty] \\
[\![v]\!]\,\eta &= \eta\, v \\
[\![\lambda v.\, e]\!]\,\eta &= \psi\,(\lambda x \in D_\infty.\, [\![e]\!][\eta \mid v : x]) \\
[\![e\, e']\!]\,\eta &= \phi\,([\![e]\!]\,\eta)\,([\![e']\!]\,\eta)
\end{aligned}$$

Lightweight formalisation in AGDA

– testing the denotation of an untyped λ-term

```
open import Relation.Binary.PropositionalEquality using (refl)
open Inverse using (inverseˡ)

to-from-elim : ∀ {f}  →  to (from f)  ≡ f
to-from-elim  = inverseˡ bi refl

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

-- (λx1.x1)x42 = x42
check-id :
  ⟦ app (lam (x 1) (var x 1))
        (var x 42) ⟧ ≡ ⟦ var x 42 ⟧
check-id = refl
```

Lightweight formalisation in AGDA

– testing the denotation of an untyped λ-term

```
-- (λx0.x0 x0)(λx0.x0 x0) = ...
-- check-divergence :
--   ⟦ app (lam (x 0) (app (var x 0) (var x 0)))
--         (lam (x 0) (app (var x 0) (var x 0))) ⟧
--   ≡ ⟦ var x 42 ⟧
-- check-divergence = refl

-- (λx1.x42)((λx0.x0 x0)(λx0.x0 x0)) = x42
check-convergence :
  ⟦ app (lam (x 1) (var x 42))
        (app (lam (x 0) (app (var x 0) (var x 0)))
             (lam (x 0) (app (var x 0) (var x 0)))) ⟧
  ≡ ⟦ var x 42 ⟧
check-convergence = refl
```

SCHEME

Lightweight formalisation of SCHEME

– AGDA code: GitHub repo pdmosses/scheme25-agda/

Quite smooth

- my second attempt to use AGDA (2025)
- domains are ***arbitrary types***
- functions on domains: ***defined*** in λ-notation, ***assumed*** continuous
- all assumptions declared as (sometimes unsatisfiable!) ***postulates***

Encouraging results

- detected several ***wellformedness*** issues in the SCHEME standard

Lightweight formalisation of SCM

– AGDA code: GitHub repo pdmosses/xds-agda/

Quite smooth

- my ***current*** attempt to use AGDA (2026)
- ***carriers*** of domains are ***non-empty types***
- functions on domains: ***defined*** in λ-notation, ***assumed*** continuous
- all assumptions declared as (hopefully satisfiable!) ***postulates***

Safer notation

- ***consistent*** with the logical foundations of AGDA **?**

Lightweight formalisation of SCM

– postulated types and elements

```
postulate
  Domain   : Set₁              -- type of all domains
  《_》     : Domain → Set     -- carrier of a domain

variable
  A B C    : Set
  D E F    : Domain
  n        : Nat

postulate
  ⊥        : 《 D 》            -- bottom element
  𝟙        : Domain            -- trivial domain
```

Lightweight formalisation of SCM

– postulated types and elements

```
postulate
  _→ᶜ_      : Domain → Domain → Domain -- assume continuous
  dom-cts   : 《 D →ᶜ E 》 ≡ (《 D 》 → 《 E 》)

{-# REWRITE dom-cts #-}

infixr 0 _→ᶜ_

postulate
  fix : 《 (D →ᶜ D) →ᶜ D 》 -- fixed points of endofunctions
```

Lightweight formalisation of SCM
– abstract syntax

```
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)

data Exp⋆ where                                -- expression sequences
  ␣␣␣            : Exp⋆                        -- empty sequence
  _␣␣_           : Exp → Exp⋆ → Exp⋆           -- prefix sequence E E⋆
```

Lightweight formalisation of SCM

– domain equations

```
data Misc : Set where
  null unallocated undefined unspecified : Misc

N      =  Nat⊥
T      =  Bool⊥
R      =  Int +⊥
P      =  L × L
M      =  Misc +⊥
F      =  E⋆ →ᶜ (E →ᶜ C) →ᶜ C
-- E   =  T + R + P + M + F
S      =  L →ᶜ E
U      =  Ide →ˢ L
C      =  S →ᶜ A
```

Lightweight formalisation of Scm

– injections, inspections, projections of summands

```
postulate
  _T-in-E     : ⟪ T   →ᶜ E ⟫
  _∈-T        : ⟪ E   →ᶜ Bool +⊥ ⟫
  _|-T        : ⟪ E   →ᶜ T ⟫

  _R-in-E     : ⟪ R   →ᶜ E ⟫
  _∈-R        : ⟪ E   →ᶜ Bool +⊥ ⟫
  _|-R        : ⟪ E   →ᶜ R ⟫

  _P-in-E     : ⟪ P  →ᶜ E ⟫
  _∈-P        : ⟪ E   →ᶜ Bool +⊥ ⟫
  _|-P        : ⟪ E   →ᶜ P ⟫

  _M-in-E     : ⟪ M   →ᶜ E ⟫
  _∈-M        : ⟪ E   →ᶜ Bool +⊥ ⟫
  _|-M        : ⟪ E   →ᶜ M ⟫

  _F-in-E     : ⟪ F   →ᶜ E ⟫
  _∈-F        : ⟪ E   →ᶜ Bool +⊥ ⟫
  _|-F        : ⟪ E   →ᶜ F ⟫
```

Lightweight formalisation of SCHEME
– semantic functions

R^5RS

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

$$\begin{aligned}&\mathcal{E}[\![\,\mathtt{(if\ E_0\ E_1\ E_2)}\,]\!] = \\ &\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\,\mathrm{E}_0\,]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\,\mathrm{E}_1\,]\!]\rho\kappa, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\,\mathrm{E}_2\,]\!]\rho\kappa))\end{aligned}$$

Agda

$$\mathcal{E}[\![\,\_\,]\!] \;:\; \mathrm{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{C}$$

$$\begin{aligned}&\mathcal{E}[\![\,(\!|\,\mathsf{if}\ \mathsf{E}_0 \,\llcorner\!\lrcorner\, \mathsf{E}_1 \,\llcorner\!\lrcorner\, \mathsf{E}_2\,|\!)\,]\!] = \\ &\quad \lambda\ \rho\ \kappa \to \mathcal{E}[\![\ \mathsf{E}_0\ ]\!]\ \rho\ (\mathsf{single}\ (\lambda\ \epsilon \to \mathsf{truish}\ \epsilon \longrightarrow \mathcal{E}[\![\ \mathsf{E}_1\ ]\!]\ \rho\ \kappa\ , \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\ \mathsf{E}_2\ ]\!]\ \rho\ \kappa))\end{aligned}$$

Postulating Domain Theory

Bernhard Reus (1999)

Formalizing Synthetic Domain Theory.

J. Autom. Reason. 23(3-4): 411-444

Abstract. Synthetic Domain Theory (SDT) is a constructive variant of Domain Theory where all functions are continuous following Dana Scott's idea of "domains as sets". Recently there have been suggested more abstract axiomatizations encompassing alternative notions of domain theory as, for example, stable domain theory.

In this article a logical and axiomatic version of SDT capturing the essence of Domain Theory à la Scott is presented. It is based on a sufficiently expressive version of constructive type theory and fully implemented in the proof checker LEGO. On top of this "core SDT" denotational semantics and program verification can be – and in fact has been – developed in a purely formal machine-checked way.

Alex Simpson (2004)

Computational adequacy for recursive types in models of intuitionistic set theory.

Ann. Pure Appl. Log. 130(1-3): 207-275

Categories that model recursive types have nontrivial fixed-point operators and thus, by a simple argument using classical logic, cannot be full subcategories of the category of sets. In [37], Dana Scott showed that such categories can nonetheless live as full subcategories of models of *intuitionistic* set theory, an observation that led to the subsequent development of *synthetic domain theory* [7,14,22,27,34,35,38,45,47]. In this paper, we exploit this idea to obtain algebraically compact categories in a uniform way. Roughly speaking, we start off with a category **S** of intuitionistic sets that satisfies one simple axiom, Axiom **1** of Section 2. From any such category **S**, we extract a full subcategory of *predomains*, $\mathbf{P} \hookrightarrow \mathbf{S}$, whose associated category of partial maps, **pP**, is algebraically compact.

[37] D.S. Scott, Relating theories of the λ-calculus, in: To H.B. Curry, Academic Press, 1980, pp. 403–450.

Safe lightweight formalization in AGDA?

– future work (help welcome!)

Implement *Synthetic Domain Theory* in *plain* AGDA

- add a type of ***predomains***
- allow ***unrestricted*** recursive domain definitions (?)
- prove that all postulated properties are ***consistent*** with MLTT
- prove that functions defined in λ-notation are ***always*** continuous
- …

Lightweight Formalisation of Denotational Semantics

– summary

Examples

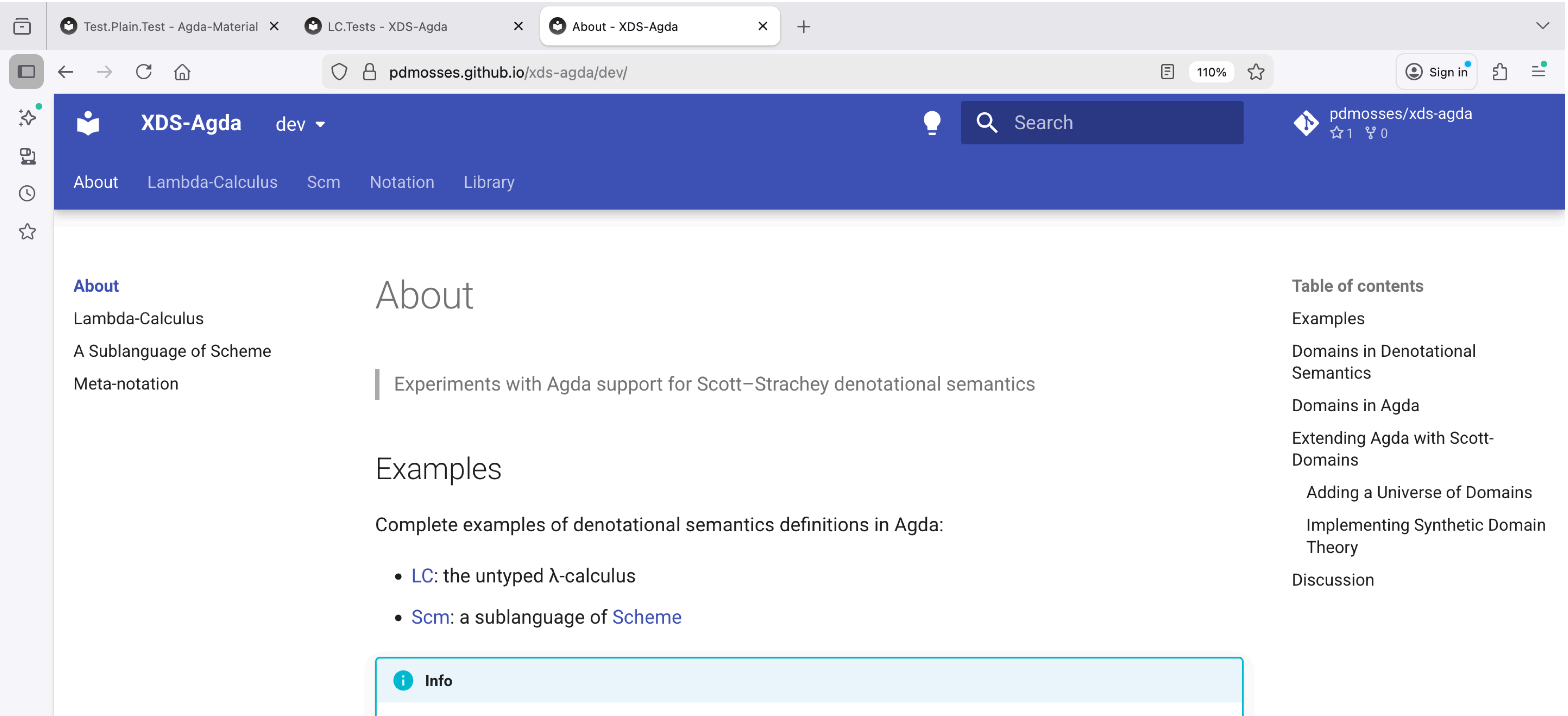
- inheritance
- the untyped λ-calculus (analytic, lightweight)
- SCHEME sublanguage SCM

Postulating domain theory

- lightweight
- synthetic

pdmosses.github.io/xds-agda/dev/

– examples: untyped λ-calculus, Scm



Appendix

Postulating Domain Theory

Postulating Domains

```
module Lifted where

  postulate
    _+⊥   : Set → Domain                   -- lifted set
    η     : 《 A →ˢ A +⊥ 》                 -- inclusion
    _♯    : 《 (A →ˢ D) →ᶜ A +⊥ →ᶜ D 》     -- Kleisli extension
```

```
module Sums where

  postulate
    _+_     : Domain → Domain → Domain                          -- coalesced sum
    inj₁    : 《 D →ᶜ D + E 》                                   -- injection
    inj₂    : 《 E →ᶜ D + E 》                                   -- injection
    [_,_]   : 《 (D →ᶜ F) →ᶜ (E →ᶜ F) →ᶜ (D + E →ᶜ F) 》         -- case analysis
```

Postulating Domains

```
module Products where

  postulate
    _×_    : Domain → Domain → Domain    -- cartesian product
    _,_    : ⟪ D →ᶜ E →ᶜ D × E ⟫          -- pairing
    _↓²1   : ⟪ D × E →ᶜ D ⟫               -- 1st projection
    _↓²2   : ⟪ D × E →ᶜ E ⟫               -- 2nd projection
    _↓³1   : ⟪ D × E × F →ᶜ D ⟫           -- 1st projection
    _↓³2   : ⟪ D × E × F →ᶜ E ⟫           -- 2nd projection
```

```
module Tuples where

  _^_ : Domain → Nat → Domain
  D ^ 0             = 𝟙
  D ^ 1             = D
  D ^ suc (suc n)   = D × (D ^ suc n)
```

Postulating Domains

```
module Sequences where

  open Lifted.Naturals
  open Tuples

  postulate
    _⋆       : Domain → Domain             -- D ⋆              finite sequences
    ⟨⟩       : ⟪ D ⋆ ⟫                     -- ⟨⟩               empty sequence
    ⟨_⟩      : ⟪ (D ^ suc n) →ᶜ D ⋆ ⟫      -- ⟨ d₁ , ... ⟩     non-empty sequence
    #        : ⟪ D ⋆ →ᶜ Nat⊥ ⟫             -- # d⋆             sequence length
    _§_      : ⟪ D ⋆ →ᶜ D ⋆ →ᶜ D ⋆ ⟫       -- d⋆ § d⋆          concatenation
    _↓_      : ⟪ D ⋆ →ᶜ Nat →ˢ D ⟫         -- d⋆ ↓ n           nth component
    _†_      : ⟪ D ⋆ →ᶜ Nat →ˢ D ⋆ ⟫       -- d⋆ † n           nth tail
```