Lightweight Agda Formalization of Denotational Semantics

Peter D. Mosses[1][2]

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

This paper introduces a lightweight approach to formalization of Scott–Strachey style denotational semantics in Agda. In contrast to previous approaches, it allows definitions of denotations in λ-notation to be embedded straightforwardly in Agda without significant changes. A lightweight Agda formalization of the standard denotational semantics of the untyped λ-calculus is given to illustrate the simplicity of the approach; lightweight Agda formalizations of denotational definitions of PCF and core Scheme are available online [11].

The lightweight approach presented here simply assumes that *all Agda types are Scott-domains, and that all Agda functions have fixed points.* Such assumptions are obviously unsound, but the Agda proof assistant accepts them, and their unsoundness does not affect checking definitions for well-formedness.

Motivation. The author's original motivation for formalizing denotational semantics was to validate a semantics of inheritance [10]. The aim was to check the soundness not only of the results stated in [2], but also of the individual proof-steps, and the Agda proof assistant seemed particularly well-suited for that. The resulting formalization [9] is reasonably lightweight. Type-checking it with Agda revealed several subtle flaws in the original denotational definition; after they had been fixed, Agda successfully checked the proof-steps of various lemmas.

Scott domain theory. In conventional Scott–Strachey denotational semantics [20, 21, 22, 27, 29], the denotation of a program phrase is an element of a Scott-domain: an ω-complete poset with a least element, possibly with further properties. Flat domains are defined as lifted sets, and domains are closed under domain constructors including lifting, cartesian product, and separated sum. Domains can also be defined recursively (up to isomorphism) without restrictions. Functions on domains defined in λ-notation are continuous, ensuring that endo-functions have least fixed points. More recent presentations of Scott domain theory [1, e.g.] define also predomains, which need not have a least element, and thus include ordinary sets (discretely ordered) as well as domains.

Agda formalization of Scott domain theory. The Agda *TypeTopology* library is based on univalent foundations. It includes modules for Scott domain theory, and illustrates their use in denotational definitions of PCF and the untyped λ-calculus [5].

This Agda formalization of domain theory corresponds directly to the usual set-theoretic definitions: a domain consists of a carrier type together with a partial order relation, its least element, and proofs of the required completeness properties; a continuous function between domains is an underlying function between their carrier types, paired with a proof of its continuity. Similarly for predomains.

Currently, the formalization requires definitions of denotations in λ-notation to include explicit continuity proofs, and subsequently discard the proof terms when applying functions. This prevents direct embedding of λ-notation from conventional denotational definitions, and seems quite impractical for formalizing the denotational semantics of larger languages (especially in continuation-passing style, e.g., for Scheme [19]).

Agda formalization of synthetic domain theory. Instead of formalizing the standard set-theoretic definitions of domains and continuous functions, synthetic domain theory (SDT) axiomatizes domains as kinds of sets in intuitionistic set-theory. Endofunctions on such sets have fixed points, and recursive set equations have solutions. SDT was suggested by Dana Scott [23], about 10 years after his initial development of domain theory.

Most of the published theoretical work on SDT [3, 7, 12, 13, 16, 18, 24, 25, 28] concentrated mainly on sorting out the underlying mathematical framework of what properties domains have, and on studying models of such domains. However, Bernhard Reus [14] also formalized SDT in the *Lego* proof assistant. The formalization relies on impredicativity and proof-irrelevance [15], which prevents porting it straightforwardly to Agda.

Alex Simpson's development of SDT [24] is based on intuitionistic ZF set theory. The generality of the approach is illustrated by a denotational semantics of FPC, a recursively-typed λ-calculus with sum and product types. In op. cit. (§3) he wrote: "it seems likely that, with appropriate reformulations, the development of this paper could be carried out in the (predicative) context of Martin-Löf's Type Theory", but apparently its formalization in Agda has not yet been attempted.

It appears that the only formalizations of SDT so far developed in Agda are based on guarded domain definitions in clocked cubical type theory [4, 6, 26]. However, denotations then involve step-indexing, so they are generally more intensional than in conventional Scott domain theory.

Lightweight Agda formalization. The Agda code presented below is a lightweight formalization of a standard denotational semantics of the untyped call-by-name λ-calculus, following [17, §10.5]. The complete source code is available online [11].

Abstract syntax. Denotational semantics conventionally defines the abstract syntax of a language by a context-free grammar. Agda doesn't include grammars, but it is quite straightforward to transform a grammar to inductive datatype definitions with the same interpretation. The following datatype uses ordinary functional notation for term constructors (partly because Agda's mixfix notation doesn't allow the usual terminal symbols of the λ-calculus) but it is otherwise a reasonably direct formalization of the original definition.

```
data Var : Set where
  x : ℕ → Var -- variables
_==_ : Var → Var → Bool
x n == x n′ = (n ≡ᵇ n′)
```

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

Domains. The standard denotational semantics of the λ-calculus is based on a domain D_∞ isomorphic to the domain of all continuous functions from D_∞ to D_∞.[1] Its lightweight formalization postulates[2] the existence of an Agda type D∞ with a bijection _↔_ to the type D∞ → D∞ of *all* Agda functions on D∞.

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

```
postulate
  D∞ : Set
postulate
  instance iso : D∞ ↔ (D∞ → D∞)
```

[1]In fact the least solution of the domain equation $D_\infty = D_\infty \to D_\infty$ is a 1-element domain; the intended solution includes some arbitrary non-trivial domain.

[2]Assumptions are specified as postulates to avoid module parameters that would need to be repeated in importing modules, and to allow assumed properties to be added as rewrite rules.

The special module application Inverse {{ ... }} above has the effect of declaring the functions to : D∞ → (D∞ → D∞) and from : (D∞ → D∞) → D∞ to be inverse.

Domain equations in the denotational semantics of other languages generally involve also some flat domains, and domain constructors for cartesian product and separated sum. Their lightweight formalizations import standard Agda library modules for the corresponding datatypes and type constructors, and postulate groups of types with bijections to Agda type terms, as illustrated for PCF and Scheme in [11].

Environments are functions from the abstract syntax of variables to values in the domain D_∞. Ordering them pointwise defines a domain of environments. The lightweight formalization of this non-recursive domain in Agda is a simple type definition, together with the definition of the conventional notation for extending an environment with a single binding:

```
Env = Var → D∞                    _[_/_] : Env → D∞ → Var → Env
                                  ρ [ d / v ] = λ v′ → if v == v′ then d else ρ v′
```

Semantic functions. A conventional denotational semantics declares semantic functions from abstract syntax to domains of denotations, and defines the functions compositionally by semantic equations. Agda formalization of semantic functions is straightforward, as semantic equations can be written directly in Agda, and the type-checker reports any missing or overlapping cases. Some minor lexical adjustments to λ-notation are needed: $\lambda x.fx$ becomes λ x → f x, adjacent names have to be separated by spaces, and sub- and superscript terms are not supported.

```
⟦_⟧ : Exp → Env → D∞              ⟦ lam v e    ⟧ ρ = from ( λ d → ⟦ e ⟧ (ρ [ d / v ]) )
⟦ var v ⟧ ρ = ρ v                 ⟦ app e₁ e₂ ⟧ ρ = to ( ⟦ e₁ ⟧ ρ ) ( ⟦ e₂ ⟧ ρ )
```

Conventional denotational definitions usually elide the isomorphisms between domains and their definitions, but Agda requires explicit use of to and from in the formalization (cf. [17, §10.5]). The type-checker reports where elided isomorphisms need to be inserted.

Checking computed values. The following rewrite rule allows Agda to automatically evaluate the denotations of terms in the untyped λ-calculus, thereby supporting trivial proofs of equivalence. (Caveat: The proofs could be unsound, as the rewrite rule involves postulates.)

```
open Inverse using (inverseˡ)                    to-from-elim = inverseˡ iso refl
to-from-elim : ∀ {f} → to (from f) ≡ f           {-# REWRITE to-from-elim #-}

check-convergence :                              check-free :
  ⟦ app (lam (x 1) (var x 42))                     ⟦ app (lam (x 1)
        (app (lam (x 0) (app (var x 0) (var x 0)))          (app (lam (x 42) (var x 1))
             (lam (x 0) (app (var x 0) (var x 0)))) ⟦            (var x 2)))
  ≡ ⟦ var x 42 ⟧                                          (var x 42) ⟧ ≡ ⟦ var x 42 ⟧
check-convergence = refl                         check-free = refl
```

The denotational semantics of PCF involves explicit use of the fixed-point function fix. Its lightweight Agda formalization postulates fix f ≡ f (fix f). To use that property directly as a rewrite rule would lead to non-termination; however, the following derived property can be used, as it unfolds fix f only when f needs to be applied (as in *SIS* [8]): fix f p ≡ f (fix f) p.

Future work. It is clearly an *abuse* of Agda to type-check definitions based on potentially-unsound postulates. An implementation of some framework for (unguarded) SDT in Agda would presumably require a significant effort, but might contribute to increased interest in SDT, as well as providing proper foundations for lightweight formalization of denotational semantics.

Acknowledgments. Thanks to Benedikt Ahrens, Jesper Cockx, Tom de Jong, Bernhard Reus, Alex Simpson, and the 24th IFIP WG 2.11 meeting participants for helpful advice regarding formalization of denotational semantics and Agda, and to the anonymous reviewers for pertinent comments and suggestions for improvement.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 3: Semantic Structures*. Clarendon Press, 1994. URL `https://achimjungbham.github.io/pub/papers/handy1.pdf`.

[2] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA 1989*, pages 433–443. ACM, 1989. doi:10.1145/74877.74922.

[3] M. P. Fiore and G. Rosolini. Domains in H. *Theor. Comput. Sci.*, 264(2):171–193, 2001. doi:10.1016/S0304-3975(00)00221-8.

[4] E. Giovannini, T. Ding, and M. S. New. Denotational semantics of gradual typing using synthetic guarded domain theory. *Proc. ACM Program. Lang.*, 9 (POPL), 2025. doi:10.1145/3704863.

[5] T. de Jong. TypeTopology/DomainTheory (Agda modules), since 2019. URL `https://martinescardo.github.io/TypeTopology/DomainTheory.index.html`.

[6] M. B. Kristensen, R. E. Møgelberg, and A. Vezzosi. Greatest HITs: Higher inductive types in coinductive definitions via induction under clocks. In *Proc. 37th Annual IEEE Symposium on Logic in Computer Science (LICS 2022)*, pages 42:1–42:13. IEEE Computer Society Press, 2022.

[7] J. R. Longley and A. K. Simpson. A uniform approach to domain theory in realizability models. *Math. Struct. Comput. Sci.*, 7(5):469–505, 1997. doi:10.1017/S0960129597002387.

[8] P. D. Mosses. SIS, 1979. URL `https://pdmosses.github.io/software/sis/`.

[9] P. D. Mosses. Agda code corresponding to a semantics of inheritance, 2024. URL `https://github.com/pdmosses/jensfest-agda/`.

[10] P. D. Mosses. Towards verification of a denotational semantics of inheritance. In *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday*, JENSFEST '24, page 5–13, New York, NY, USA, 2024. ACM. doi:10.1145/3694848.3694852.

[11] P. D. Mosses. Denotational semantics in Agda, 2025. URL `https://github.com/pdmosses/xds-agda/`. Experiments with lightweight formalization in Agda of existing language definitions.

[12] J. van Oosten and A. K. Simpson. Axioms and (counter) examples in synthetic domain theory. *Ann. Pure Appl. Log.*, 104(1-3):233–278, 2000. doi:10.1016/S0168-0072(00)00014-2.

[13] W. Phoa. Effective domains and intrinsic structure. In *LICS '90*, pages 366–377. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113762.

[14] B. Reus. Formalizing synthetic domain theory. *Journal of Automated Reasoning*, 23:411–444, 1999. doi:10.1023/A:1006258506401.

[15] B. Reus. Re: SDT in Agda. Personal communication, 2025.

[16] B. Reus and T. Streicher. General synthetic domain theory – a logical approach. *Math. Struct. Comput. Sci.*, 9(2):177–223, 1999. doi:10.1017/S096012959900273X.

[17] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[18] G. Rosolini. *Continuity and Effectivity in Topoi*. PhD thesis, Univ. of Oxford, 1986.

[19] Revised^5 report on the algorithmic language Scheme, 1998. URL `https://standards.scheme.org/official/r5rs.pdf`.

[20] D. Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970. URL `https://ncatlab.org/nlab/files/Scott-TheoryOfComputation.pdf`. Also: Tech. Monograph PRG-2, Oxford Univ.

Computing Lab., Programming Research Group (1970). URL https://www.cs.ox.ac.uk/files/3222/PRG02.pdf.

[21] D. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136, Berlin, Heidelberg, 1972. Springer. doi:10.1007/BFb0073967. Also: Tech. Monograph PRG-7, Oxford Univ. Computing Lab., Programming Research Group (1971). URL https://www.cs.ox.ac.uk/files/3229/PRG07.pdf.

[22] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia Series*, pages 19–46. Polytechnic Inst. of Brooklyn, 1971. Also: Tech. Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971). URL https://www.cs.ox.ac.uk/files/3228/PRG06.pdf.

[23] D. S. Scott. Relating theories of the lambda-calculus: Dedicated to Professor H. B. Curry on the occasion of his 80th birthday. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980. URL https://prl.khoury.northeastern.edu/blog/static/scott-80-relating-theories.pdf.

[24] A. Simpson. Computational adequacy for recursive types in models of intuitionistic set theory. *Annals of Pure and Applied Logic*, 130(1):207–275, 2004. doi:10.1016/j.apal.2003.12.005. Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS).

[25] A. K. Simpson. Computational adequacy in an elementary topos. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *CSL '98*, volume 1584 of *Lecture Notes in Computer Science*, pages 323–342. Springer, 1998. doi:10.1007/10703163_22.

[26] P. Stassen, R. E. Møgelberg, M. A. Zwart, A. Aguirre, and L. Birkedal. Modelling recursion and probabilistic choice in guarded type theory. *Proc. ACM Program. Lang.*, 9 (POPL), 2025. doi:10.1145/3704884.

[27] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, 1977.

[28] P. Taylor. The fixed point property in synthetic domain theory. In *LICS '91*, pages 152–160. IEEE Computer Society, 1991. doi:10.1109/LICS.1991.151640.

[29] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, Aug. 1976. doi:10.1145/360303.360308.