Mechanising Denotational Semantics in Agda

Peter D. Mosses[a,b,1] Jesper Cockx[a,2] Bernhard Reus[c,3]

[a] *Department of Software Technology*
Delft University of Technology
Delft, Netherlands

[b] *Faculty of Science and Engineering*
Swansea University
Swansea, United Kingdom

[c] *School of Engineering and Informatics*
University of Sussex
Brighton, United Kingdom

Abstract

Mechanisation of a mathematical definition (also referred to as formalisation) has many benefits. Here, we focus on mechanising denotational semantic definitions of programming languages by embedding them in the Agda language. The Agda type-checker detects and reports any issues with the wellformedness and type correctness of the embedded definitions.
To minimise the effort required, and to facilitate correlation of the original definition with its Agda embedding, mechanisation should not involve significant reformulation or extension. Here, we show how to embed conventional Scott–Strachey denotational definitions in Agda with almost no changes to λ-notation or domain equations.
Agda notation for definitions of types and functions corresponds closely to the conventional meta-notation of denotational semantics. We have developed a collection of Agda modules with postulated types for commonly used domain constructors and their associated operations. Some of our postulates are inconsistent with a classical set-theoretic interpretation of Agda; we conjecture that they would be consistent with an interpretation of Agda in the higher-order intuititionistic logic used by Simpson in his work on synthetic domain theory.
We illustrate our approach with mechanisations of three denotational definitions: Scott's D_∞ model of the untyped λ-calculus, Plotkin's denotational semantics of PCF, and a semantics of a sublanguage of Scheme. In previous work, similar mechanisations in Agda have revealed several unsuspected wellformedness issues in published denotational definitions.

Keywords: Denotational semantics, formalisation, mechanisation, Agda, postulates, synthetic domain theory

1 Introduction

The Scott–Strachey style of denotational semantics is based on Christopher Strachey's idea of defining the semantics of high-level programming languages by mapping phrases to pure higher-order functions [50], combined with the rigorous foundations for fixed points provided by Dana Scott's development of domain theory [42,43,44]. It has been widely adopted in theoretical studies of programming language semantics, and presented in many textbooks. However, its use in language reference manuals and standards has been quite limited, with the notable exception of the *Scheme* language reports [41].

[1] Email: p.d.mosses@tudelft.nl
[2] Email: j.g.h.cockx@tudelft.nl
[3] Email: bernhard@sussex.ac.uk

A significant drawback for potential users of the Scott–Strachey style is the lack of an IDE to support development and testing of denotational definitions. It is well known that manual development and review of large formal definitions are error-prone: mechanical checks are essential to ensure wellformedness [17].

We have developed a shallow embedding of Scott–Strachey style denotational definitions in Agda, to support their formulation and checking wellformedness. Crucially, definitions of denotations in λ-notation are embedded almost verbatim, with only minor lexical changes (such as inserting spaces between adjacent variables). If the embedding of a definition is rejected by Agda, the location of the error in the Agda code generally makes it easy to find the source of the wellformedness issue in the original definition.

Although the original motivation for developing an Agda embedding of denotational semantics was to check the proof steps in a semantics of inheritance [7,8], type-checking the embedding revealed some unsuspected wellformedness issues [26]. Using a subsequent version of our embedding, we have detected previously unreported wellformedness issues in the denotational semantics given in the Scheme reports [27].

Our current embedding of denotational semantics in Agda is based on declarations of postulated types for domain constructors and their associated operations. We abstract from the mathematical structure of domains, as it does not affect how conventional denotational semantic definitions are formulated. In future work, we aim to *define* the postulated domain constructors and operations based on postulated axioms for *synthetic domain theory* (SDT). As Alex Simpson hinted in [47, sect. 3]: "*it seems likely that, with appropriate reformulations, the development [of SDT] could be carried out in the (predicative) context of Martin-Löf's Type Theory*".

Contributions

After briefly recalling some features of denotational semantics and Agda (Section 2), we present the Agda code that supports our embedding of denotational definitions, and illustrate its use:

- We introduce our postulated types for domain constructors and their associated operations (Section 3).
- We give an Agda embedding of Scott's D_∞ model of the untyped λ-calculus (Section 4.1). Agda requires explicit use of the inverse functions between D_∞ and the domain of endofunctions on D_∞ when defining the denotations of abstraction and application, but otherwise the embedding is as simple as possible.
- Our Agda embedding of Plotkin's denotational semantics of the PCF language (Section 4.2) illustrates the use of dependent types in formalising the semantics of an intrinsically-typed programming language. This example exploits Agda's support for Unicode symbols to minimise notational differences from Plotkin's definition.
- The last illustrative example that we present here is an embedding of the denotational semantics of a simple sublanguage of *Scheme* (Section 4.3). It uses all the domain constructors introduced in Section 3.
- We postulate basic properties of operations associated with some domain constructors (Section 5), and use these properties as rewrite rules to support automatic equivalence proofs for term denotations (Section 6).

We then give an overview of mechanised domain theory and synthetic domain theory (SDT), and consider the possibility of mechanising SDT in Agda (Section 7).

Readers should preferably be familiar with the basic concepts of denotational semantics (e.g., in Scott–Strachey style [44,49,51]) and with dependently-typed languages such as Agda [52,55].

The Agda code shown in this paper elides various details. See the accompanying website [21] for hyperlinked, highlighted listings of the complete Agda code. The website and the LaTeX sources for the following sections were generated from the repository at github.com/pdmosses/mfps2026-agda/.

2 Background

2.1 Denotational Semantics

A conventional Scott–Strachey denotational semantics of a programming language consists of definitions of *abstract syntax*, *semantic domains*, and *semantic functions*. The abstract syntax uses a *context-free grammar* to define sets of abstract syntax trees (ASTs) that model the compositional structure of programs and their phrases. The semantic domains are defined by equations in terms of *domain constructors*, and can be mutually recursive. The semantic functions are defined *inductively*, also with mutual recursion,

and map ASTs to their *denotations*, which are elements of domains. The denotation of an AST models its contribution to the semantics of complete programs, and is defined *compositionally* in terms of the denotations of its direct components. The AST arguments of semantic functions are enclosed by $[\![\ldots]\!]$, to avoid potential confusion between abstract syntax terms and semantic notation.

Domains were originally required to be continuous lattices [42,43,49,51]. Various more general notions of domain have been suggested (see [1]) but for us it is only important that endomaps between domains have fixpoints, recursive domain equations have solutions, and each domain comes with a bottom element. The notation for domain constructors and their accompanying operations in a conventional Scott–Strachey semantics, summarised below, is independent of the choice of domains.

Let D, E be domains with elements $\delta : D$, $\epsilon : E$.

- Every domain D has a *bottom* element $\bot_D$ that represents absence of information (e.g., due to an error or nontermination of a computation), usually written just $\bot$.
- A function $\phi : D \to E$ is (Scott-) *continuous* if it is monotone and preserves limits of directed subsets of D [1]. The *function domain* $F = D \to E$ consists of all continuous (total) functions from D to E. The set of all functions from a set A to a domain also forms a domain, ordered pointwise. Functions between domains are defined using abstraction $\lambda\delta.\epsilon$, application $\phi\,\delta$, the operation fix that maps each endofunction $\phi : D \to D$ to its least fixed point, and the operations associated with other domain constructors.
- For any set A the *flat domain* $A_\bot$ is formed by adding $\bot$ as a fresh element. Functions on sets are implicitly extended to (continuous) functions on flat domains, returning $\bot$ when any argument is $\bot$. Elements τ of the domain of *truth-values* $\mathbf{T} = \{\mathrm{true}, \mathrm{false}\}_\bot$ are used in *conditionals* written $\tau \to \delta_1, \delta_2$, where $\mathrm{true} \to \delta_1, \delta_2$ is δ_1, $\mathrm{false} \to \delta_1, \delta_2$ is δ_2, and $\bot_\mathbf{T} \to \delta_1, \delta_2$ is $\bot_D$.
- The *separated sum domain* $X = \ldots + Y + \ldots$ consists of injected elements written 'v in X' (where $v : Y$ for some summand Y) together with $\bot_X$. The $\mathbf{T}$-valued operation $\chi\,\mathsf{E}\,Y$ (written $\chi \in Y$ in [41]) tests whether $\chi : X$ is the injection of some $v : Y$; if so, $\chi \mid Y$ projects χ to v, otherwise to $\bot_Y$.
- The *product domain* $P = D \times E$ consists of pairs $\langle\delta, \epsilon\rangle$ with $\bot_P = \langle\bot_D, \bot_E\rangle$. When $\pi : P$, the operations $\pi \downarrow 1$ and $\pi \downarrow 2$ select its components; similarly for domains of *n-tuples* D^n and finite *sequences* D^*. Further operations on D^* include the empty sequence $\langle\rangle$, concatenation $\delta_1^* \S \delta_2^*$, length $\#\,\delta^*$, nth component $\delta^* \downarrow n$, and nth tail $\delta^* \dagger n$ $(n \geq 1)$.

Robert Tennent's highly accessible tutorial on denotational semantics [51] includes illustrative examples using the above notation (partly in continuation-passing style) and further details of the conventional Scott–Strachey style.

2.2 Agda

Agda [52,55] is both a strongly typed functional *programming language* with support for first-class *dependent types* and a *proof assistant* based on the Curry–Howard correspondence between propositions and types. A number of features set Agda apart from a typical functional programming language:

- Types in Agda are first-class values of a *universe* Set_i where $\mathsf{Set} = \mathsf{Set}_0$, and each universe Set_i belongs to the next universe Set_{i+1}.
- Agda has *dependent function types* $(\mathsf{x} : \mathsf{A}) \to \mathsf{B}$ or $\forall\ \mathsf{x} \to \mathsf{B}$ where the type B can depend on the value x.
- All functions in Agda are *total*: evaluating a function call is guaranteed to return a value of the correct type in finite time. To ensure totality, Agda's type checker includes a termination check for recursive definitions, a positivity check for inductive datatypes, and a consistency check for universe levels.

Thanks to its dependent types and totality, Agda's type system can be used as a higher-order logic for writing mathematical statements and proofs, where types correspond to propositions and type checking corresponds to checking the validity of the proof.

Notes on some of Agda's syntax and features

Comments. End-of-line comments are initiated by `--`, while multi-line comments are between `{-` and `-}`.

Naming. Declared symbols, variables, and type constructors all share a single namespace. Names can be any sequence of non-whitespace ASCII and Unicode characters except `.;{}()@"`, excluding reserved

symbols and keywords such as → or where. Underscores in names play a special role for mixfix notation.

Mixfix notation. Agda supports *mixfix notation* for defining operators, with underscores in the name of a symbol indicating argument positions. For example, if we declare `_+_ : Nat → Nat → Nat`, we can use it as `1 + 1` (the spaces are required, since `1+1` is a valid Agda name!).

Anonymous functions. Lambda abstractions use the syntax `λ x → u` instead of `λ x. u`.

Implicit arguments. Arguments marked by single curly braces `{...}` in the type of a symbol are considered to be *implicit*. These arguments may be omitted, and are then inferred by Agda's type checker.

Type classes. Agda has no direct support for type classes, but they can be simulated using Agda's *instance arguments* to resolve type class instances. Instance arguments are marked by double curly braces `{{...}}` and are resolved automatically by using definitions marked as instance.

Rewrite rules. Agda has support for *rewrite rules* [5] that are applied automatically during type checking. Rewrite rules are declared by marking an equality proof `eq : a ≡ b` with a `{-# REWRITE eq #-}` pragma. Agda can optionally check confluence of rewrite rules, but currently does not check their termination. Since only *proven* (or postulated) equalities can be added as rewrite rules, non-confluence and non-termination cannot affect the soundness of Agda's type checker, only its completeness [6].

3 Postulated Domain Notation

This section postulates Agda notation for the domain constructors and associated functions used in the illustrative examples (Section 4 ⇑). See the accompanying website [21] for hyperlinked, highlighted listings of the complete Agda code with the details elided here (including module imports, fixity declarations, and declarations of the types of meta-variables). In the PDF, the symbol ⇑ following a reference to a numbered section is a link to the corresponding page on the website.

3.1 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
```

Some previous papers on embedding denotational semantics in Agda [27,28,29] defined domains to be types: `Domain = Set`. However, postulating `⊥ : D` for all `D : Domain` was then *inconsistent* with the existence of an empty type in Agda. Postulating `Domain : Set` avoids that inconsistency.

The notation for domains postulated here supports *type-checking* embeddings of denotational semantics in Agda such as those in the illustrative examples. It does *not* define or constrain the *mathematical structure* of domains, nor the algebraic and universal properties of the associated functions.

3.2 Function Domains

The conventional notation in denotational definitions for the domain of continuous functions from D to E is $D \to E$ or $[D \to E]$. However, Agda reserves the notation `D → E` for the *type* of *all* (total) functions from type D to type E; instead, we use the notation `D →ᶜ E` for embedding continuous function domains:

```
module Functions where
  postulate _→ᶜ_ : Domain → Domain → Domain
```

Both λ-abstraction and application preserve continuity. In conventional denotational semantics, functions between domains are defined using λ-abstraction and application from primitive continuous functions

associated with specific domain constructors, so they are *automatically* continuous. This motivates treating the carrier ⟪ D →ᶜ E ⟫ of the embedding of a function domain as a type of continuous functions. (*Proving* functions defined in λ-notation to be continuous in Agda requires pairing each λ-abstraction with an explicit proof of its continuity, which is quite impractical – especially when embedding denotations defined in continuation-passing style.)

However, to support type-checking the *direct* embedding of λ-notation from conventional denotational definitions in Agda, it appears to be necessary to *rewrite* the carrier types of function domains to ordinary function types:

```
postulate dom-cts : ⟪ D →ᶜ E ⟫ ≡ (⟪ D ⟫ → ⟪ E ⟫)
{-# REWRITE dom-cts #-}
```

Similarly, the notation A →ˢ D is the embedding of the domain of all functions from an ordinary type A to a domain D (which are trivially continuous, ordered pointwise):

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

Embeddings of *endofunctions* φ on a domain D should always have fixed points fix φ, with fix itself also being continuous:

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

3.3 Recursive Domains

Conventional denotational semantics often involves groups of mutually recursive domain definitions. In Agda, recursive type definitions lead to non-termination of the type-checker. To avoid non-termination, it is sufficient to break the recursion by leaving (one or more) domains as *postulated*. The following operations can then be used to map values from a postulated domain to its structure and *vice versa*.

```
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 ⟫
```

The *instance parameter* {{D ≅ E}} of the above operations restricts them to domains D and E for which instance _ : D ≅ E has been declared.

3.4 Flat Domains

Lifting an ordinary set A by adding a ⊥ element gives a flat domain, usually written $A_\perp$. Our Agda embedding postulates a corresponding domain constructor A +⊥, together with a function ↑ for injecting elements of A into A +⊥, and an operator f ♯ for extending functions on A to arguments in A +⊥.

```
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 ⊥
```

3.4.1 Booleans

The McCarthy conditional operation $\beta \longrightarrow \delta_1 \, , \, \delta_2$ extends the usual ternary conditional choice to domains. It returns $\bot$ whenever its first argument is $\bot$.

```
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) ♯
```

This module also defines Eq A for use as an instance parameter, restricting operation definitions to types A such that _==_ : A → A → Bool, and postulates a Bool⊥-valued operation $\delta_1 ==\bot \delta_2$ on A +⊥.

3.4.2 Naturals

Agda allows decimal notation for natural numbers, as well as unary notation using zero and suc.

```
module Naturals where
  Nat⊥ = Nat +⊥
```

3.5 Sum Domains

The separated sum D + E of two domains corresponds to lifting the disjoint union of their carrier sets. The following operations can be used directly for binary sums, and iterated for domains with more than two summands.

```
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
```

Conventional denotational definitions of programming languages (e.g., in [41]) use domain names instead of numerical indices in operations associated with separated sums. The inherently *dependent* types of the Agda embedding of these operations are as follows.

```
  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
```

The operations are defined only for D and E where an instance of type $\mathsf{E} \gtrsim \mathsf{n} \mapsto \mathsf{D}$ is declared for some n. Instead of defining the summands D of a separated sum domain E by an equation E = ... + D + ..., the domain E is merely *postulated*, and each summand is declared separately by instance _ : $\mathsf{E} \gtrsim \mathsf{n} \mapsto \mathsf{D}$ (where n should be a different natural number for each summand).

3.6 Product Domains

The carrier of the binary product D × E of two domains consists of all pairs (d , e) of elements of D and E with the pair (⊥{D} , ⊥{E}) as the bottom element ⊥{D × E}. Neither the product nor pairing is associative. The following operations can be used directly for binary products, and iterated for products of more than two domains.

```
module Products where
  postulate
    _×_ : Domain → Domain → Domain      -- D × E is cartesian 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
```

3.6.1 Tuples

The domain D ^ n of n-tuples of elements of a domain D is conventionally written D^n, but Agda does not support the use of variables as superscripts.

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

3.6.2 Sequences

The domain D * of finite sequences of elements of a domain D is conventionally written D^*.

The following notation for the various operations on sequences was introduced in the early 1970s, and is used in the *Scheme* semantics [41]. (The single angle-brackets ⟨...⟩ used to form sequences are unrelated to the double angle-brackets ⟪ D ⟫ used for the carrier of domain D.)

```
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
```

3.7 Updates

When a type A has an equality operation _==_ : A → A → Bool, environments ρ : ⟪ A →ˢ D ⟫ can be 'updated' (i.e., extended or overridden) using the conventional notation ρ [δ / a], defined as follows.

```
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′
```

Similarly for stores σ : ⟪ (A +⊥) →ᶜ D ⟫:

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

Defining an operation m [x ← y] for extension or overriding of *dependent* maps m is less straightforward, as it involves an equality test that may return an *equivalence proof*.

4 Illustrative Examples

This section illustrates mechanisation of denotational semantics in Agda with three examples, all using the postulated notation (Section 3 ⇑) for domains and their associated operations: the Untyped Lambda-Calculus (Section 4.1 ⇑), PCF: A Programming Language for Computable Functions (Section 4.2 ⇑), and *Scm*: A Sublanguage of *Scheme*.

4.1 Untyped Lambda-Calculus

This section presents our Agda embedding of a denotational semantics of the untyped λ-calculus.

4.1.1 Abstract Syntax

Abstract syntax is *not* regarded as a domain. In conventional Scott–Strachey style denotational semantics, abstract syntax is generally first-order: terms are finite, totally-defined elements.

A variable is written x n. The argument n merely distinguishes between variables – it is *not* a De Bruin index. The term constructor var below includes variables in terms.

In Agda, mixfix notation requires argument positions _ to be separated by characters other than spaces. The term constructors for function abstraction and application use the Unicode character ␣ as a separator.

```
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
```

4.1.2 Domain Equations

Simply defining $D\infty = (D\infty \to^c D\infty)$ would lead to non-termination of the Agda type-checker. Instead, we postulate the domain $D\infty$, together with a bijection $D\infty \cong (D\infty \to^c D\infty)$. This declares unfold : $\langle\!\langle\ D\infty \to^c (D\infty \to^c D\infty)\ \rangle\!\rangle$ and fold : $\langle\!\langle\ (D\infty \to^c D\infty) \to^c D\infty\ \rangle\!\rangle$.

```
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
```

Use of the conventional notation ρ [δ / v] for updating an environment ρ to map v to d requires an equality test for variables, elided here.

4.1.3 Semantic Functions

The semantic equations below correspond closely to those found in textbooks on denotational semantics (e.g., [40]). In larger conventional definitions, fold and unfold are usually left implicit, but Agda does not support this.

```
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 )
```

4.2 PCF: A Programming Language for Computable Functions

PCF and its denotational semantics were orginally defined by Dana Scott in 1969 [46] with combinators (S, K) instead of λ-abstraction. Gordon Plotkin subsequently defined a denotational semantics for PCF including λ-abstraction [35]. The Agda modules presented below are an embedding of the latter definition.

PCF is an intrinsically typed language: every well-formed term has a unique type. The following grammar summarises the context-free abstract syntax of types σ, τ and terms M, N with variables α_i^σ ($i \geq 0$) and constants c. In Section 4.2.1 ⇑ we reflect Plotkin's presentation of PCF more accurately by exploiting Agda's support for dependent types.

$$\sigma, \tau ::= \iota \mid o \mid (\sigma \to \tau) \tag{1}$$
$$c ::= tt \mid ff \mid \supset \mid \mathbf{Y} \mid k_n \mid (+1) \mid (-1) \mid \mathbf{Z} \tag{2}$$
$$M, N ::= \alpha_i^\sigma \mid c \mid (M\ N) \mid (\lambda \alpha_i^\sigma M) \tag{3}$$

4.2.1 Abstract Syntax

The following abstract syntax of well-formed PCF terms in Agda uses indexed datatype definitions. PCF function types $\sigma \to \tau$ are written $\sigma \Rightarrow \tau$, and variables α_i^σ are written α i σ (where the argument i merely distinguishes between variables – it is *not* a De Bruin index).

```
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

data Terms : Types → Set where                          -- typed terms
  V_       : Vars σ → Terms σ                           -- variable
  L_       : 𝓛ᴬ σ → Terms σ                             -- constant
  (|_␣_|)  : Terms (σ ⇒ τ) → Terms σ → Terms τ          -- function application
  (|λ_␣_|) : Vars σ → Terms τ → Terms (σ ⇒ τ)           -- function abstraction
```

4.2.2 Domain Equations

The domains $\mathcal{D}\ \sigma$ form a "*standard collection of domains for arithmetic*" in PCF, written $\mathcal{D}_\sigma$ in [35]. As PCF is a simply-typed language, the domains $\mathcal{D}\ \sigma$ are not reflexive, so their embedding in Agda can use ordinary type definitions, not involving bijections.

```
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
```

Environments ρ are type-preserving maps from variables to values. They are naturally modeled by a dependent type: Env σ consists of type-preserving maps from variables in Vars σ to their values in the domain $\mathcal{D}\ \sigma$. The environment $\rho\bot$ maps all variables to $\bot$.

```
Env = (σ : Types) → ⟪ Vars σ →ˢ 𝒟 σ ⟫ -- typed environments
ρ⊥ : Env                              -- initial environment
ρ⊥ _ _ = ⊥
```

Extension or overriding typed environments, written ρ [v / x]′, requires instances of the equality tests for both variables and types. The definition of the latter is somewhat tedious.

4.2.3 Semantic functions

The notation ρ ⟦ α i σ ⟧ gives the value of the variable α i σ in ρ by applying $\rho\ \sigma$ to the variable.

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

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

The semantic function $\mathcal{A}$⟦ c ⟧ gives the standard interpretation of the constant c. The corresponding definitions in [35] use case analysis on the domain $\mathcal{D}\ \iota$, which our Agda embedding does not support (partly because it can express non-continuous functions).

```
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)) ♯
```

The semantic function $\mathcal{A}'$⟦ M ⟧ is written $\hat{\mathcal{A}}[\![M]\!]$ in [35]. It gives the denotation of the term M as a function of the environment ρ.

```
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 ]′)
```

Comparison with Plotkin's original definition of PCF [35] confirms the directness of our Agda embedding.

4.3 Scm*: A Sublanguage of* Scheme

Scm is a particularly basic sublanguage of the core *Scheme* expression language whose denotational semantics is defined in the *Scheme* reports [41]. The domains and auxiliary functions declared in this section are explained in the presentation of the conventional denotational semantics of *Scm* [28]; they involve the notation for sequence domains (Section 3.6.2 ⇑).

4.3.1 Abstract Syntax

The following grammar [28] summarises the abstract syntax of *Scm* expressions E : Exp with integers Z : Int, constants K : Con and identifiers I : Ide. The meta-variable E^* : Exp^* implicitly ranges over arbitrary sequences of expressions.

$$K ::= Z \mid \texttt{\#t} \mid \texttt{\#f} \tag{4}$$

$$E ::= K \mid I \mid (E_0\ E^*) \mid (\texttt{lambda}\ I\ E) \mid (\texttt{if}\ E_0\ E_1\ E_2) \mid (\texttt{set!}\ I\ E) \tag{5}$$

In the following Agda embedding of the above grammar, the abstract syntax of sequences E^* : Exp^* is made explicit: the empty sequence is represented by ␣␣␣ , and sequence prefixing by E ␣␣ E^*.

```
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
```

4.3.2 Domain Equations

The domains for *Scm* are somewhat simpler than for the denotational semantics in the Scheme standards [41], but still involve all our postulated domain constructors. Using definitional equations D = E instead of postulated bijections D ≅ E avoids the need for the functions fold and unfold.

```
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
```

The published denotational semantics of *Scm* [28] defines the domain $\mathbf{E}$ of expression values by the equation $\mathbf{E} = \mathbf{T} + \mathbf{R} + \mathbf{P} + \mathbf{M} + \mathbf{F}$, and the domain $\mathbf{F}$ of procedure values by $\mathbf{F} = \mathbf{E}^* \to (\mathbf{E} \to \mathbf{C}) \to \mathbf{C}$. Mutually recursive groups of domain equations have well-defined solutions, but in Agda, defining both the corresponding domains $\mathbf{E}$ and $\mathbf{F}$ by equations would cause the type checker to diverge. Postulating one (or both) of these domains avoids divergence; postulating $\mathbf{E}$ also has the benefit that the embeddings and projections for its summands subsume the bijection between $\mathbf{E}$ and its intended structure.

The following postulates instantiate injection (δ in⊥ $\mathbf{E}$), inspection (ε ∈⊥ D), and projection (ε |⊥ D) for each summand D of $\mathbf{E}$.

```
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
```

4.3.3 Auxiliary Functions

The λ-notation in the Agda definitions of auxiliary functions for *Scm* corresponds closely to that in its published denotational semantics [28].

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

assign : ⟪ L →ᶜ E →ᶜ C →ᶜ C ⟫    -- assign α ϵ stores ϵ at location α
assign α ϵ θ σ = θ (σ [ ϵ / α ]⊥)

hold : ⟪ L →ᶜ (E →ᶜ C) →ᶜ C ⟫    -- hold α gives the value stored at α
hold α κ σ = κ (σ α) σ
```

In the continuation-passing style used to define auxilary functions for *Scm*, giving explicit continuity proofs would be particulary tedious. For example, the function hold is simply a combination of λ-abstraction and application, which is wellknown to ensure continuity.

```
postulate new : ⟪ (L →ᶜ C) →ᶜ C ⟫ -- new gives an unallocated location

alloc : ⟪ E →ᶜ (L →ᶜ C) →ᶜ C ⟫    -- alloc ϵ allocates a location for ϵ
alloc ϵ κ = new (λ α → assign α ϵ (κ α))

postulate initial-store : ⟪ S ⟫       -- may have initialised locations
```

Conventional denotational definitions usually leave the injection function ↑ from sets into flat domains implicit, in contrast to the embedding of the definition of truish:

```
truish : ⟪ E →ᶜ T ⟫                    -- truish ε is true for all ε except false
truish ϵ = (ϵ ∈⊥ T) ⟶ (((ϵ |⊥ T) ==⊥ ↑ false) ⟶ ↑ false , ↑ true) ,
           ↑ true
```

The remaining auxiliary function definitions shown here involve the operations for (finite) sequences ϵ^* declared in the module Notation.Products.Sequences.

```
cons : ⟪ F ⟫                           -- cons ⟨ ϵ₁ , ϵ₂ ⟩ allocates and initialises a pair
cons ϵ* κ = (# ϵ* ==⊥ ↑ 2) ⟶
                alloc (ϵ* ↓ 1) (λ α₁ →
                  alloc (ϵ* ↓ 2) (λ α₂ → κ ((α₁ , α₂) in⊥ E))) ,
              ⊥
```

In [28] the auxiliary function *list* is defined by recursion on ϵ^*. Agda accepts recursive definitions only when it can mechanically prove that the recursion terminates, which is not the case for arguments in postulated types. The following definition uses the postulated operation fix to avoid recursion.

```
list : ⟪ F ⟫                           -- list ϵ* allocates and initialises a list
list = fix λ (list′ : ⟪ F ⟫) → λ ϵ* κ →
       (# ϵ* ==⊥ ↑ 0) ⟶ κ (↑ null in⊥ E) ,
       list′ (ϵ* † 1) (λ ϵ → cons ⟨ (ϵ* ↓ 1) , ϵ ⟩ κ)
```

4.3.4 Semantic Functions

The λ-notation in the Agda definitions of semantic functions for *Scm* corresponds closely to that in its published denotational semantics [28].

```
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*)))
```

5 Postulated Properties

The Properties module postulates basic properties of some of the operations of the postulated domain notation (Section 3⇑). These properties are expected to hold in various categories of domains [1] but they do *not* define the *mathematical structure* of domains.

The postulated properties support proofs that terms have identical denotations. For example, some illustrative tests (Section 6⇑) declare that the denotation of a function application is equivalent to the denotation of a constant; other tests declare that particular instances of renaming do not affect denotations.

When postulated properties are declared as *rewrite rules*, Agda can use them *automatically* in proofs. Agda also has an option to check that the declared rewrite rules form a confluent system. Rewrite rules are safe to use with Agda.Builtin.Equality when that option is enabled. Confluent but non-terminating rewrite rules cannot break consistency, as shown by Cockx, Tabareau, and Winterhalter [6].

The rewrite rules declared below support *automatic* proof of identity for all the illustrative tests: the proof terms are simply refl (i.e., reflexivity).

```
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 #-}
```

The rewrite rule apply-fix does not cause the type-checker to diverge, despite the obvious non-termination. Agda's type checker uses *weak head evaluation*: it only unfolds expressions to the point where the top-level constructor becomes visible. In particular, it will not evaluate under a λ-abstraction unless it is being compared to another λ-abstraction and the bodies are not syntactically equal.

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

A rule for fold (unfold d) ≡ d could be added, but it is not needed for the current illustrative tests.

```
module Flat where
postulate
  elim-♯-↑ : (f ♯) (↑ a′) ≡ f a′
  elim-♯-⊥ : (f ♯) ⊥ ≡ ⊥
{-# REWRITE elim-♯-↑ elim-♯-⊥ #-}
```

Removing any of the above rewrite rules breaks the proof in at least one of the illustrative tests. In principle, all refl proof terms that rely on rewrite rules could be replaced by proofs that apply the postulated properties to specified subterms. However, we expect that it would be quite tedious to develop such proofs, and reading them is unlikely to provide new insights.

The remaining postulated properties are for domains that are not used in the semantics of the LC and PCF languages; they will be needed when tests for equivalence of denotations of *Scm* expressions are added. Postulates of properties for our operations on tuples and sequences have not yet been developed.

6 Illustrative Tests

The tests below illustrate *automatic* proof of equivalence of term denotations by the Agda type-checker: the type-correctness of the refl definitions of the declared equivalences confirms that they hold. Such tests can check that the denotation of a function call is equivalent to the denotation of a constant in the embedding of a denotational semantics. They can even check that a specific renaming preserves the denotation of a term.

Apart from confirming that a denotational semantics defines denotations which satisfy some expected equivalences at least for some terms, some of the tests also depend on rewrite rules for the postulated

properties (Section 5 ⇑). The success of those tests indirectly checks that the rewrite rules preserve denotations. (A more systematic approach would be to develop a suite of unit tests for consequences of postulated properties, independently of denotational definitions.)

6.1 LC Tests

```
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
```

A reviewer pointed out that the only interpretation of ⟪ D∞ ⟫ in Agda could be a singleton type, so that one should expect many equalities to hold. However, when the proof of an equality is simply by refl, Agda's type-checker does not automatically use such reasoning.

6.2 PCF 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
```

7 Mechanising Domain Theory

In order to check ones understanding of a programming language or to develop new features one wishes to use a theorem prover. To model denotational semantics of any interesting language, however, one must employ domain theory to model recursive functions and recursive types. The prover of choice must thus provide libraries that implement a sufficiently rich domain theory. Alas, there are not many libraries (that are easy enough to use), one can make repeated use of postulates, like in the presented Agda development. In the literature the following mechanisations of domain theory are known:

7.1 Overview of Mechanised Standard Domain Theory

There have been previous successful attempts to mechanise classical domain theory and Milner's Logic for Computable Functions [22]. Franz Regensburger [36] developed a corresponding formalisation in Isabelle/HOL [31] which was further extended as described in [30]. The authors of the latter use a special tactic that "*reduces the continuity of an LCF term to the continuity of its basic components by structural rules*" [30]. In Agda, there exists the possibility for users to define tactics through reflection [54]. It might

be possible to define tactics for automatically proving continuity for functions between domains, but it is unclear how well that would work in practice.

There have also been attempts to mechanise domain theory in a constructive setting using intuitionistic logic. There are, for instance, developments of constructive domain theory in Coq/Rocq [2,11,32]. A version of ω-cpos has been developed in [2] where the lifting is defined via a coinductive definition of infinite streams. It is based on Paulin-Mohring's mechanisation [32] of the semantics of Kahn networks where she developed a general library for cpos. Dockins developed effective algebraic domains (more precisely: constructive models of profinite domains) in Coq/Rocq [11].

Another formalisation of domain theory has been carried out in Agda based on constructive and predicative univalent foundations [10]. In this work, Scott's model of PCF is developed in Agda with the aim to "*explore the development of domain theory from the univalent point of view*" [10]. The authors use universes to allow for algebraic directed complete partial orders to be large but locally small. Their axiomatisation is based on propositional and functional extensionality. This is an excellent example of formalising mathematics and semantics in univalent foundations but it requires a rather well trained person. One must keep track of universe levels and manage the required proofs that types are propositions and that λ-abstractions are continuous. The proof of continuity of combinator S in `PCF.Combinatory.PCFCombinators.html` [9] is an example of how detailed such continuity proofs must be.

7.2 Synthetic Domain Theory

All of the formalisations considered above have in common that the continuity of functions between domains needs to be proven for each function involved (including anonymous functions expressed via λ). This is extremely tedious, and discourages formalisation of denotational semantics. Developments like the one presented in this paper would be much easier to carry out if domains were sets and functions between domains automatically continuous – as originally suggested by Dana Scott [45] – such that the category of domains becomes a full subcategory of the category of sets. Based on this idea, several different flavours of domains have been defined over time in specific realisability/topos models or axiomatically: replete objects [16], well-complete objects [19], extensional Σ-spaces [34,37]. As domains are "synthesized" as sets with properties, rather than defined as sets with structure, i.e. partial order, the corresponding theory was called *Synthetic Domain Theory* (SDT) in analogy with Synthetic Differential Geometry. Since "domains as sets" is only consistent with intuitionistic logic, any such approach needs to be developed in a version of intuitionistic set theory. Domain Theory requires the notion of partiality which in SDT is expressed via Σ, the subset of those propositions that express termination of programs. The set Σ contains the proposition $\top$ and is closed under composition (i.e., it is a dominance [14]). Based on Σ one can define a lifting functor with final coalgebra $\mathbf{F}$ and initial algebra $\mathbf{I}$. This $\mathbf{I}$ is the generic ω-chain such that elements in $X^{\mathbf{I}}$ can be regarded as ω-chains in X. And $\mathbf{F}$ can be viewed as $\mathbf{I}$'s chain completion.

7.3 Mechanisations of SDT

There are only very few mechanisations of SDT in theorem provers.

The formalisation of SDT in [38], based on [39], was carried out in the *Lego* proof assistant developed in Edinburgh and used in the 1990s. It implements the Extended Calculus of Constructions [20] and uses the impredicative universe of propositions. For its mechanisation an additional impredicative universe of sets had been built into the Lego system. Complete extensional Σ-spaces serve as notion of domains which can be represented as double-negation closed subsets of powers of Σ. Domain constructors are defined as endofunctors on this category of domains and existence of fixpoints for those functors was proven using the standard inverse limit construction. A correctness proof of the Sieve of Eratosthenes on the domain of streams of natural numbers was used to test the theory. Due to its use of two impredicative universes (Set and Prop), this mechanisation appears to be impossible to translate into Agda. The justification for this axiomatisation is the extensional Σ-per model [34] which is impredicative. A more general approach that does not require impredicativity is Alex Simpson's wonderfully unifying account of SDT [47].

Simpson's purely axiomatic model appears to be the only one that "*applies uniformly across the different types of model*" [47, p. 209]. "*This is not trivial, however, ... The problem is that elementary toposes, although models of intuitionistic higher-order logic, are not, in general, models of a sufficiently powerful set theory. Thus, ... we shall require that* ***S*** *having enough structure to model full intuitionstic Zermelo-Fraenkel*

(IZF) set theory asking for $\boldsymbol{S}$ *to be given as the full subcategory of small objects in a category* $\boldsymbol{C}$ *with class(ic) structure and universal object.*" [47, p. 208] Within $\mathbf{C}$ the category $\mathbf{P}$ of predomains is the full subcategory of $\mathbf{S}$ with small well-complete sets as objects. An object is said to be well-complete if its lifting is complete w.r.t the embedding $\iota : \mathbf{I} \to \mathbf{F}$ or, in other words, X is complete if X^ι is an isomorphism. Here, $\mathbf{I}$ represents a generic ω chain and $\mathbf{F}$ its completion via ι. An ω-chain in X is simply represented then as an element of $X^{\mathbf{I}}$.

If one stipulates that the single-element type 1 is well-complete then many useful results hold, among them the following: the category of predomains is cartesian closed, well-complete objects are automatically complete, lifting preserves well-complete objects, and products of families of well-complete objects indexed by a small set are well-complete. Simpson managed to prove that the category of Σ-partial maps is algebraically compact (for internal endofunctors). This is shown by an adaptation of the classic limit-colimit coincidence theorem [48]. Instead of $\mathbf{N}$-indexed families of embedding-projection pairs, one uses general $\mathbf{I}$-indexed diagrams that meet some equational properties. Actually, Simpson shows more generally that certain kinds of suitable categories are algebraically compact and that this property is closed under products and duality, which is important to define fixpoints of mixed-variant internal bifunctors, and thus to model recursive type in FPC [13].

Note that the algebraic compactness result requires that $\mathbf{S}$, $\mathbf{P}$ and the category of partial maps between objects in $\mathbf{P}$ must be *internal* subcategories of $\mathbf{C}$. The axiom that the type 2 of Booleans is well-complete is required if one wishes to model sum types and also implies that the empty type 0 is well-complete and thus that $\bot \in \Sigma$.

It remains future work to attempt to mechanise Simpson's setup of constructive set theory with a few axioms in a prover like Agda. As Simpson himself, we speculate that such a mechanisation should be possible. Unlike the one presented in this paper, this would prove that all major notions and results of domain theory, like fixpoints of functions and functors between domains, can be developed in Agda from a very small number of axioms and postulates. However, this would not require any change to the Agda embedding of the definitions of denotational semantics presented in this paper.

7.4 Other Flavours of SDT

There are also some mechanisations of a different kind of Synthetic Domain Theory. Those would require some rewriting of the ways domains are defined.

Synthetic Guarded Domain Theory (SGDT) [3] is a variant where Nakano's later modality is used to type the fixpoint operator. Fixpoints exist for functions that factor through the unit of the later monad. These principles can be applied to define (productive) recursive types. The underlying model is the category of presheaves over ω which contains the category of complete bisected ultrametric spaces as coreflective subcategory. The latter provides the semantic justification for the later monad and existence of fixpoints for contractive maps.

Guarded Dependent Type Theory (GDTT) [4] is an extensional type theory based on the later monad as type constructor. Paviotti et al. [33] have developed PCF in GDTT but no Agda implementation is available.

Clocked Cubical Type Theory [18] is an intensional type theory extending cubical type theory and thus permits developments in homotopy type theory. Agda includes an implementation of some of clocked cubical type theory under the name Guarded Cubical Agda. CCS and the π-calculus have been implemented in this version of Agda [53] as well as gradual typing [15]. Those mechanisations clearly involve some denotational semantics but a dedicated user friendly library for domain theory or denotational semantics appears to be lacking.

8 Conclusion

We have shown how to formalise conventional Scott–Strachey denotational definitions by embedding them in Agda with almost no changes to λ-notation or domain equations. This supports mechanisation of semantic definitions of programming languages with relatively little extra effort. Mechanisation is essential for checking the wellformedness of larger definitions.

We have presented three illustrative examples of our shallow embedding of denotational semantics

in Agda: Scott's D_∞ model of the untyped λ-calculus, Plotkin's denotational semantics of PCF, and a semantics of a sublanguage *Scm* of *Scheme*. Each example exhibited different benefits of our approach:

- The λ-calculus semantics was particularly simple to mechanise in Agda, thanks to the ease of postulating recursive domain equations, and to our support for the conventional notation for updating environments.
- The PCF semantics demonstrated how the mathematical meta-notation in Plotkin's article could be represented in Agda by exploiting its support for Unicode symbols. It also showed how easily dependent types are able to formalise a verbal explanation of the syntax of an intrinsically typed language.
- The semantics of *Scm* was developed by one of the authors from the denotational semantics of the full *Scheme* expression language, which has been included in the *Scheme* standards since 1986 [41]. The Agda formalisation of the *Scheme* semantics [27] was produced from a plain (Unicode) text preview of the PDF of a published *Scheme* report, followed by some simple transformations in an interactive editor, and finally insertion of layout to recover the original alignment. The Agda type-checker reported several wellformedness issues; some of them stemmed from informally elided injections and projections, but some were trivial type errors that had not previously been reported. The presentation of the results at a *Scheme* workshop led to interest in Agda mechanisation of future denotational definitions of *Scheme*.

Although the main aim of our mechanisation of denotational semantics in Agda is to check wellformedness, we have discovered that we can also use it to check equivalence of denotations of terms, as demonstrated in Section 6. The proof terms are trivial (refl) thanks to the automatic use of our specified rewrite rules.

Previous versions of the Agda mechanisation of the λ-calculus model and the *Scm* semantics have been published [28,29]. They both treated ordinary Agda types as domains, which is inconsistent with the existence of the empty type in Agda. Moreover, the previous version of the *Scm* semantics did not make use of instance parameters, so the postulated operations on separated sums had to be defined separately for each summand. Those drawbacks have been eliminated in the current mechanisation.

We have chosen to use a *shallow* embedding of denotational semantics in Agda. This has several disadvantages compared to a deep embedding. For example, the Agda type-checker is far more general than required for checking wellformedness of a denotational semantics, but does not allow domain definitions to be recursive; the embedding of abstract syntax grammars as datatype definitions is undesirable; and there are minor differences at the lexical level, such as the need to replace conventional λ-abstraction notation $\lambda x.fx$ by $\lambda\ \mathsf{x} \to \mathsf{f}\ \mathsf{x}$ (with symbols also separated by layout). However, implementing a deep embedding of denotational semantics in Agda can take a significantly greater effort, as for example with the development of SIS (Semantics Implementation System) in the 1970s [12,23,24,25]; use of a modern language workbench would reduce the effort, but also introduce an extra level of meta-notation.

The plethora of postulates in our Agda code is unconventional: module parameters are generally preferred to postulates, as they document dependence of proofs on undischarged assumptions. However, postulates have significant advantages over module parameters in connection with rewrite rules, as well as being notationally simpler to declare. As discussed in detail in Section 7.2, an axiomatisation of synthetic domain theory in Agda could lead not only to a much smaller number of postulates, but also to proof of the existence of the various domain constructors declared in Section 3. This should establish solid mathematical foundations for our mechanisation of denotational semantics in Agda, without undermining the directness of the shallow embedding of domain equations and λ-notation.

Acknowledgements.

The inspiration for the Agda development reported in this paper stems from Alex Simpson's helpful response to our enquiry about the possibility of formalising his development of SDT [47] in Agda: he wrote "*All published theoretical work on SDT I'm aware of (including my own) concentrated mainly on sorting out the underlying mathematical framework [. . .] It would be really nice to have a more black-box, modular and applicable approach. For example, one might simply assume in Agda that there is a 'universe' of (pre)domains closed under many type constructions, and with a suitably well-behaved fixed-point operator, and use that to do semantics*". We have also benefited from the advice of participants at an IFIP WG 2.11 meeting (Edinburgh, December 2024) and at an Agda Implementors' Meeting (Angers, November 2025).

We are very grateful to the anonymous reviewers of our submitted paper for their perceptive comments and suggestions for its improvement.

References

[1] Abramsky, S. and A. Jung, *Domain theory*, pages 1–168, Oxford University Press, Inc., USA (1995), ISBN 019853762X. https://achimjungbham.github.io/pub/papers/handy1.pdf

[2] Benton, N., A. Kennedy and C. Varming, *Some domain theory and denotational semantics in Coq*, in: S. Berghofer, T. Nipkow, C. Urban and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 115–130, Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_10

[3] Birkedal, L., R. E. Møgelberg, J. Schwinghammer and K. Støvring, *First steps in synthetic guarded domain theory: step-indexing in the topos of trees*, Log. Methods Comput. Sci. **8** (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

[4] Bizjak, A., H. B. Grathwohl, R. Clouston, R. E. Møgelberg and L. Birkedal, *Guarded dependent type theory with coinductive types*, CoRR **abs/1601.01586** (2016). 1601.01586. http://arxiv.org/abs/1601.01586

[5] Cockx, J., *Type theory unchained: Extending Agda with user-defined rewrite rules*, in: M. Bezem and A. Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:27, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020), ISBN 978-3-95977-158-0, ISSN 1868-8969. https://doi.org/10.4230/LIPIcs.TYPES.2019.2

[6] Cockx, J., N. Tabareau and T. Winterhalter, *The taming of the rew: a type theory with computational assumptions*, Proc. ACM Program. Lang. **5** (2021). https://doi.org/10.1145/3434341

[7] Cook, W. R. and J. Palsberg, *A denotational semantics of inheritance and its correctness*, in: G. Bosworth, editor, *Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA 1989, New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings*, pages 433–443, ACM (1989). https://doi.org/10.1145/74877.74922

[8] Cook, W. R. and J. Palsberg, *A denotational semantics of inheritance and its correctness*, Inf. Comput. **114**, pages 329–350 (1994). https://doi.org/10.1006/INCO.1994.1090

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

[10] de Jong, T. and M. H. Escardó, *Domain Theory in Constructive and Predicative Univalent Foundations*, in: C. Baier and J. Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021), ISBN 978-3-95977-175-7, ISSN 1868-8969. https://doi.org/10.4230/LIPIcs.CSL.2021.28

[11] Dockins, R., *Formalized, effective domain theory in Coq*, in: G. Klein and R. Gamboa, editors, *Interactive Theorem Proving*, pages 209–225, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_14

[12] Donzeau-Gouge, V., G. Kahn and B. Lang, *A complete machine-checked definition of a simple programming language using denotational semantics*, Technical report RR-330, IRIA, Rocquencourt (1978). https://inria.hal.science/hal-04716568/

[13] Fiore, M. P. and G. D. Plotkin, *An axiomatization of computationally adequate domain theoretic models of FPC*, in: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 92–102, IEEE Computer Society (1994). https://doi.org/10.1109/LICS.1994.316083

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

[15] Giovannini, E., T. Ding and M. S. New, *Denotational semantics of gradual typing using synthetic guarded domain theory*, Proceedings of the ACM on Programming Languages **9**, pages 772–801 (2025), ISSN 2475-1421. https://doi.org/10.1145/3704863

[16] Hyland, J. M. E. and E. Moggi, *The S-replete construction*, in: *CTCS 1995*, volume 953 of *LNCS*, Springer Verlag (1995). https://doi.org/10.1007/3-540-60164-3_22

[17] Klein, C., J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt and R. B. Findler, *Run your research: on the effectiveness of lightweight mechanization*, in: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296, Association for Computing Machinery, New York, NY, USA (2012), ISBN 9781450310833.
https://doi.org/10.1145/2103656.2103691

[18] Kristensen, M. B., R. E. Møgelberg and A. Vezzosi, *A model of clocked cubical type theory*, ArXiv **abs/2102.01969** (2021).
https://api.semanticscholar.org/CorpusID:231786671

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

[20] Luo, Z., *A higher-order calculus and theory abstraction*, Information and Computation **90**, pages 107–137 (1991), ISSN 0890-5401.
https://doi.org/https://doi.org/10.1016/0890-5401(91)90062-7

[21] *Mechanising denotational semantics in Agda: Code* (2026).
https://pdmosses.github.io/mfps2026-agda/

[22] Milner, R., *Implementation and application of Scott's logic of continuous functions*, in: *Conference on Proving Assertions About Programs*, pages 1–6, SIGPLAN 1 (1972).
https://doi.org/https://doi.org/10.1145/942578.80706

[23] Mosses, P. D., *The semantics of semantic equations*, in: A. Blikle, editor, *Mathematical Foundations of Computer Science, 3rd Symposium at Jadwisin near Warsaw, Poland, June 17-22, 1974, Proceedings*, volume 28 of *Lecture Notes in Computer Science*, pages 409–422, Springer, Berlin, Heidelberg (1974).
https://doi.org/10.1007/3-540-07162-8_701

[24] Mosses, P. D., *Mathematical Semantics and Compiler Generation*, DPhil dissertation, University of Oxford (1975).

[25] Mosses, P. D., *SIS: Semantics implementation system* (2024).
https://pdmosses.github.io/software/sis/

[26] Mosses, P. D., *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, pages 5–13, Association for Computing Machinery, New York, NY, USA (2024), ISBN 9798400712579.
https://doi.org/10.1145/3694848.3694852

[27] Mosses, P. D., *Checking a denotational semantics of Scheme in Agda*, in: *Proceedings of the 26th ACM SIGPLAN International Workshop on Scheme and Functional Programming*, Scheme '25, pages 3–15, Association for Computing Machinery, New York, NY, USA (2025), ISBN 9798400721625.
https://doi.org/10.1145/3759537.3762694

[28] Mosses, P. D., *A compositional semantics for* `eval` *in Scheme*, in: *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday*, OLIVIERFEST '25, pages 72–81, Association for Computing Machinery, New York, NY, USA (2025), ISBN 9798400721502.
https://doi.org/10.1145/3759427.3760369

[29] Mosses, P. D., *Lightweight Agda formalization of denotational semantics*, in: F. Nordvall Forsberg, editor, *Proceedings of the 31st International Conference on Types for Proofs and Programs*, TYPES 2025, pages 286–290, University of Strathclyde, Glasgow, Scotland (2025).
https://msp.cis.strath.ac.uk/types2025/abstracts/TYPES2025_paper11.pdf

[30] Müller, O., T. Nipkow, D. Von Oheimb and O. Slotosch, *HOLCF = HOL + LCF*, Journal of Functional Programming **9**, pages 191–223 (1999).
https://doi.org/10.1017/S095679689900341X

[31] Nipkow, T., L. C. Paulson and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*, Springer (2002).
https://doi.org/https://doi.org/10.1007/3-540-45949-9

[32] Paulin-Mohring, C., *A constructive denotational semantics for Kahn networks in Coq*, in: Y. Bertot, G. Huet, J.-J. Lévy and G. Plotkin, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 383–414, Cambridge University Press (2009).
https://doi.org/10.1017/CBO9780511770524.018

[33] Paviotti, M., R. E. Møgelberg and L. Birkedal, *A model of PCF in guarded type theory*, Electronic Notes in Theoretical Computer Science **319**, pages 333–349 (2015), ISSN 1571-0661. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).
https://doi.org/https://doi.org/10.1016/j.entcs.2015.12.020

[34] Phoa, W., *Effective domains and intrinsic structure*, in: *LICS '90*, pages 366–377, IEEE Computer Society (1990). https://doi.org/10.1109/LICS.1990.113762

[35] Plotkin, G. D., *LCF considered as a programming language*, Theoretical Computer Science **5**, pages 223–255 (1977), ISSN 0304-3975. https://doi.org/10.1016/0304-3975(77)90044-5

[36] Regensburger, F., *HOLCF: higher order logic of computable functions*, in: E. T. Schubert, P. J. Windley and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307, Springer (1995). https://doi.org/10.1007/3-540-60275-5_72

[37] Reus, B., *Extensional Σ-spaces in type theory*, Applied Categorical Structures **7**, pages 159–183 (1999). https://doi.org/10.1023/A:1008600521659

[38] Reus, B., *Formalizing synthetic domain theory*, Journal of Automated Reasoning **23**, pages 411–444 (1999). https://doi.org/10.1023/A:1006258506401

[39] Reus, B. and T. Streicher, *General Synthetic Domain Theory – a logical approach*, Math. Struct. in Comp. Sci. **9**, pages 177–223 (1999). https://doi.org/https://doi.org/10.1017/S096012959900273X

[40] Reynolds, J. C., *Theories of Programming Languages*, Cambridge Univ. Press, Cambridge, UK (1998), ISBN 9780521106979. https://doi.org/10.1017/CBO9780511626364

[41] *Scheme standards*. https://standards.scheme.org

[42] Scott, D., *Outline of a mathematical theory of computation*, in: *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, IEEE, Princeton, NJ, USA (1970). Also: Tech. Monograph PRG-2, Oxford Univ. Computing Lab., Programming Research Group (1970). URL https://www.cs.ox.ac.uk/files/3222/PRG02.pdf. https://ncatlab.org/nlab/files/Scott-TheoryOfComputation.pdf

[43] Scott, D., *Continuous lattices*, in: F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136, Springer Berlin Heidelberg, Berlin, Heidelberg (1972), ISBN 978-3-540-37609-5. https://doi.org/10.1007/BFb0073967

[44] Scott, D. 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, New York, NY, USA (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.

[45] Scott, D. S., *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*, pages 403–450, Academic Press, London, UK (1980). https://prl.khoury.northeastern.edu/blog/static/scott-80-relating-theories.pdf

[46] Scott, D. S., *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Theoretical Computer Science **121**, pages 411–440 (1993), ISSN 0304-3975. https://doi.org/10.1016/0304-3975(93)90095-B

[47] Simpson, A., *Computational adequacy for recursive types in models of intuitionistic set theory*, Annals of Pure and Applied Logic **130**, pages 207–275 (2004), ISSN 0168-0072. Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS). https://doi.org/10.1016/j.apal.2003.12.005

[48] Smyth, M. B. and G. D. Plotkin, *The category-theoretic solution of recursive domain equations*, SIAM Journal on Computing **11**, pages 761–783 (1982). https://doi.org/10.1137/0211062

[49] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, MIT Press, Cambridge, MA, USA (1977), ISBN 978-0-262-19147-0.

[50] Strachey, C., *Towards a formal semantics*, in: *Formal Language Description Languages for Computer Programming, Proc. IFIP Working Conference, 1964*, pages 198–220, North-Holland, Amsterdam, Netherlands (1966).

[51] Tennent, R. D., *The denotational semantics of programming languages*, Commun. ACM **19**, pages 437–453 (1976), ISSN 0001-0782. https://doi.org/10.1145/360303.360308

[52] The Agda Team, *Agda Language Reference* (2025).
https://agda.readthedocs.io/en/v2.8.0/language/

[53] Vezzosi, A., A. Mörtberg and A. Abel, *Cubical Agda: A dependently typed programming language with univalence and higher inductive types*, Journal of Functional Programming **31**, page e8 (2021).
https://doi.org/10.1017/S0956796821000034

[54] Walt, P. and W. Swierstra, *Engineering proof by reflection in Agda*, volume 8241 (2012), ISBN 978-3-642-41581-4.
https://doi.org/10.1007/978-3-642-41582-1_10

[55] Wikipedia, *Agda* (2026).
https://en.wikipedia.org/wiki/Agda_(programming_language)