Towards Verification of a Denotational Semantics of Inheritance

Peter D. Mosses
Delft University of Technology
Delft, Netherlands
Swansea University
Swansea, United Kingdom
P.D.Mosses@swansea.ac.uk

Abstract

Jens Palsberg's first research publication was an OOPSLA '89 paper, coauthored with William Cook. In that much-cited paper, the authors identify self-reference as a central feature of inheritance, and analyze it using fixed points. They then define both an operational and a denotational semantics of inheritance, and prove them equivalent. Their proof exploits an intermediate semantics, obtained by step-indexing the operational semantics – an early use of the so-called 'fuel pattern'.

This paper presents an Agda formulation of the definitions and lemmas from the OOPSLA '89 paper. The Agda proof assistant detected some minor issues when type-checking the definitions; after they had been fixed, Agda successfully checked all the steps in the proofs of the lemmas. The Agda definitions and proofs make the same assumptions as the OOPSLA '89 paper about the existence of recursively defined Scott domains, and about the continuity of the defined functions.

CCS Concepts: • **Theory of computation** → **Denotational semantics**; **Operational semantics**; **Object oriented constructs**; **Type theory**.

Keywords: Agda, proof assistant, dependent types, Scott domains, continuous functions

ACM Reference Format:
Peter D. Mosses. 2024. 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), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3694848.3694852

JENSFEST '24, October 22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1257-9/24/10
https://doi.org/10.1145/3694848.3694852

1 Introduction

In 1989, Jens Palsberg published a paper at OOPSLA, coauthored with William Cook [5]: *A Denotational Semantics of Inheritance and its Correctness.*[1] It was the first of Jens's many fine research publications, and has more than 500 citations on Google Scholar. It also includes an early practical application of *step-indexing* (a.k.a. *the fuel pattern* [12]) in formal semantics.

From 1988 to 1992, Jens was a PhD student in Computer Science at the University of Aarhus, Denmark, and I had the pleasure of being his supervisor. His thesis *Provably Correct Compiler Generation* [18] is based on action semantics (a hybrid of denotational, algebraic, and operational semantics [13, 14, 16]) and involves a compiler from action notation to RISC.

In practice, however, Jens had no need for any supervision, and it would have been futile to try to restrict his attention to his thesis topic. He took the opportunity to start a particularly fruitful collaboration with Dexter Kozen and Michael Schwartzbach on type inference and object-oriented programming languages [2, 10, 17, 20–22], and coauthored with them the first of his many POPL papers [11].

Fifteen years ago, Jens organized a symposium in connection with my 60th birthday, and edited the accompanying Festschrift [19]. I would now like to dedicate the present paper to Jens on the occasion of *his* 60th birthday, as a token of my admiration of his career and accomplishments, and of my gratitude for having had such a successful first PhD student.

The idea for this paper occurred to me when I first heard about the JensFest.[2] Although I had no previous experience of using proof assistants, it looked feasible to reformulate the elegant definitions from the OOPSLA '89 paper (hereafter referred to as CP89) in Agda, and check the proofs mechanically using the Agda proof assistant. I expected that it would be easy enough to define Agda types corresponding to Scott domains, and to exploit the continuity of functions defined by lambda-expressions in Agda. I soon realized that it wasn't

[1]The subsequent journal version [6] has the same title. William Cook also included much of the material in his PhD thesis [4, Ch. 5], acknowledging the benefit from Jens Palsberg's work in developing a rigorous proof of the correctness theorem.
[2]https://2024.splashcon.org/home/jensfest-2024

quite so straightforward – and that developing Agda proofs can also be somewhat challenging... However, I have now managed to check the proofs of the four main lemmas from CP89 in Agda, and it should be possible to check the proofs of the remaining results in the same way.

The rest of the present paper proceeds as follows. Section 2 recalls the origins of fixed-point semantics, and the main benefits of using Scott domains in semantic definitions. Section 3 considers how to define fixed-point semantics in Agda. Section 4 presents Agda definitions that correspond closely to the operational and denotational semantic definitions in CP89. Section 5 discusses the origins of step-indexing in programming language semantics, defines the intermediate semantics from CP89 in Agda, and presents the Agda proof of a lemma about its relationship to the denotational semantics; the proofs of the other lemmas from CP89 are available as auxiliary material accompanying this paper, together with the literate Agda source files used to generate Sections 4 and 5. Section 6 concludes, summarizing what has been achieved, and indicating what would be needed to complete the verification of the correctness of the denotational semantics of inheritance in Agda.

2 Fixed-Point Semantics

The denotational semantics of method systems in CP89 is based on fixed-point semantics. A fixed point of a function F is a value x such that $F(x) = x$. Functions on a set may have any number of fixed points: e.g., the successor function on the integers has none, a constant function has a unique fixed point, and all argument values are fixed points of the identity function. A fixed-point operator fix is a higher-order function which, when applied to an argument function F, returns a fixed-point of F, so $F(\mathsf{fix}(F)) = \mathsf{fix}(F)$.

The fixed-point approach to semantics of programming languages was developed initially by Christopher Strachey in the early 1960s [27]. Strachey aimed to define the semantics of program phrases (commands, declarations, expressions, etc.) as pure mathematical functions, obtaining the semantics of a compound phrase by composing the semantics of its subphrases. To define the semantics of loops compositionally, he used a fixed-point operator (Y). However, although $\mathsf{Y}(f)$ had a clear operational interpretation, based on its unfolding to $f(\mathsf{Y}(f))$, it could not be interpreted as a mathematical function in ordinary set theory.

In 1969, Dana Scott became interested in Strachey's work, discovered how to construct a model of the (untyped) lambda-calculus [24], and started the development of domain theory. Domains can be defined recursively (up to isomorphism) using domain constructors including function spaces. The elements of function spaces are restricted to continuous functions, which always have a least fixed-point, thereby giving for any domain D an interpretation of Y as a (continuous) function from the function space $D \longrightarrow D$ to D.

Scott and Strachey presented their resulting framework in a joint paper in 1971 [25]; it is now referred to as denotational semantics. Extensive introductory explanations of denotational semantics are given in many textbooks; see [1, 24, 26] for presentations of the underlying domain theory.

3 Fixed-Point Semantics in Agda

Agda[3] is a pure functional language that can be used not only for programming, but also as a proof assistant for developing and checking proofs. It supports defining dependently-typed functions, and mutually-recursive groups of types and functions. I expected that it would be straightforward to define the denotational semantics of inheritance from CP89 in Agda, and check the proof of its equivalence to the operational semantics. One of my aims was for the semantic definitions in Agda to use the same notation as those in CP89, so the correspondence would be reasonably clear. Another aim was to avoid use of unsafe Agda features that might lead to inconsistencies when checking proofs. I encountered difficulties regarding the following points.

Types of continuous functions: The denotational semantics in CP89 uses a fixed-point operator *fix* on a domain of continuous functions. Agda does not support defining a corresponding type of continuous functions as a subtype of an Agda function type – it is necessary to encode it as a predicate or a sigma-type.

Recursive type definitions: The semantic domains in CP89 include $\mathbf{Fun} = \mathbf{Value} \rightarrow \mathbf{Value}$ where **Value** (indirectly) involves **Fun**. Termination checking of the corresponding Agda type definition failed.

Recursive function definitions: The method lookup semantics in CP89 involves mutually-recursive functions where a recursive call is non-structural. Termination checking of the function definitions failed.

Implicit function applications: Following standard practice in denotational semantics, both isomorphisms between domains and injections into sum domains are left implicit in CP89. Agda required them to be explicit.

Implicit function continuity: Functions defined on Scott domains using lambda-abstraction and application are always continuous. In Agda, it appears that the only implicit property of functions defined by lambda-expressions is totality.

The workarounds that have been found for some of the above difficulties are explained in Section 4. A more principled approach would be to represent Scott domains as products of Agda types, as in the *DomainTheory* modules[4] by Tom de Jong [7, 8], which are included in the *TypeTopology* library[5] developed by Martín Escardó et al. There, a domain is a

[3]https://github.com/agda/agda
[4]https://www.cs.bham.ac.uk/~mhe/TypeTopology/DomainTheory.index.html
[5]https://github.com/martinescardo/TypeTopology/

pointed directed-complete poset (dcpo), consisting of a carrier type D, a partial order on D, a distinguished element $\perp$ of D, and a proof that the partial order is directed-complete with $\perp$ being the least element. An element of the carrier of the domain of functions between domains D and E is an underlying function between the carriers of D and E, together with a proof of its continuity. Recursive definitions of domains follow Scott's original construction of a domain isomorphic to its own function space [24].

However, that approach appears to have a pragmatic drawback, which discouraged its adoption here: every lambda-abstraction used in the semantic definitions would have to to be accompanied by an explicit proof of its continuity; and every function application would have to discard the continuity proof and apply the underlying function. The extra notation involved would significantly clutter the Agda formulation of the definitions from CP89. Another Agda library that supports the use of Scott domains is *agda-unimath*,[6] but it appears to have the same drawback.

A further possibility might be to exploit the Agda support for synthetic denotational semantics in *guarded* dependent type theory [3], provided that the notational overhead (e.g., in the denotational semantics of PCF in [23]) could be eliminated. However, the correspondence between the semantic definitions in CP89 and the Agda definitions would then be less direct, and the proof steps required to prove the lemmas in CP89 might be different.

Caveat: As a novice Agda user, I found it difficult to read many of the definitions in the cited libraries, and I could certainly have overlooked some opportunities.

4 Semantic Definitions

This section presents an Agda formulation of the semantics of inheritance given by William Cook and Jens Palsberg in CP89 (their OOPSLA '89 paper) [5, §4.1–4.3]. The semantics is based on their conceptual analysis of inheritance and self-reference in terms of fixed points of generators and wrappers. See their paper for motivation of inheritance, and for their conceptual analysis of inheritance and self-reference using fixed points.

Apart from commenting on some differences from the original semantics, the explanations given together with the Agda definitions below focus on how various features are modeled as functions. Readers who are not familiar with Agda may find it helpful to browse the Agda Wikipedia page[7] before proceeding.

The LaTeX sources for this and the following section were generated by Agda from literate Agda specifications (included in the auxiliary material accompanying this paper). Some Unicode characters are displayed as corresponding LaTeX math symbols, and module indentation is omitted.

[6]https://unimath.github.io/agda-unimath/
[7]https://en.wikipedia.org/wiki/Agda_(programming_language)

4.1 Agda Standard Library Notation

The Agda definitions given below use the following modules from the standard library (v2.1).[8]

```
{-# OPTIONS –safe #-}                    – Agda       ~ CP89 notation
open import Data.Nat.Base
  using ( ℕ; zero; suc; _≤_ )            – ℕ          ~ Nat
open import Data.Maybe.Base
  renaming ( Maybe to _+?;               – A +?       ~ A + ?
             nothing to ??;              – ??         ~ ⊥?
             maybe′ to [_,_]? )          – [ f , x ]? ~ [f, λ⊥?.x]
open import Data.Product.Base
  using (_×_; _,_; proj₁; proj₂)         – A × B      ~ A × B
open import Function
  using (Inverse; _↔_; _∘_)              – A ↔ B      ~ implicit
open Inverse {{ ... }}
  using (to; from)                       – to from    ~ implicit
```

The declaration open Inverse {{ ... }} above introduces overloaded functions to and from for each parameter of the form {{ i : A ↔ B }}. The double braces specify so-called instance parameters, which are the Agda equivalent of Haskell type class constraints.

4.2 Domains

The types and functions declared below as module parameters correspond to assumptions about various features of Scott domains. They are used when defining the semantics of method systems in Agda.

An element D : Domain is an Agda type corresponding to a domain used in CP89. Such a type D has a type of elements 〈 D 〉 and a distinguished element ⊥. Further assumptions about domains will be made in Section 5, when proving results that involve the partial order on 〈 D 〉.

```
module Inheritance.Definitions
  ( Domain : Set₁ )
  ( 〈_〉    : Domain → Set )
  ( ⊥      : {D : Domain} → 〈 D 〉 )
  ( fix    : {D : Domain} → ( 〈 D 〉 → 〈 D 〉 ) → 〈 D 〉 )
```

The function fix is supposed to correspond to the least fixed point operator on the space of continuous functions on a domain. To ensure that fix can be applied only to continuous functions, it would need to take a proof of continuity as an extra argument.

In practice, however, we intend to apply fix only to functions on 〈 D 〉 that are defined by lambda-abstraction and application, and these are *assumed* to correspond to continuous functions on domains. It is superfluous to pass an *assumption* of continuity as an explicit argument – the same assumption can be made wherever it is needed – so we simply omit the extra argument from the type of fix.

[8]https://agda.github.io/agda-stdlib/v2.1/

```
( ?⊥       : Domain )
( _+⊥_     : Domain → Domain → Domain )
( inl      : {D E : Domain} → ⟨ D ⟩ → ⟨ D +⊥ E ⟩ )
( inr      : {D E : Domain} → ⟨ E ⟩ → ⟨ D +⊥ E ⟩ )
( [_,_]⊥ : {D E F : Domain} →
           ( ⟨ D ⟩ → ⟨ F ⟩ ) → ( ⟨ E ⟩ → ⟨ F ⟩ ) →
           ⟨ D +⊥ E ⟩ → ⟨ F ⟩ )
```

?⊥ here corresponds to the 1-point domain written '?' in CP89; its only element is ⊥ ($\bot_?$ in CP89). D +⊥ E corresponds to the notation $D + E$ for separated sums of domains in CP89. The injection functions inl and inr are left implicit in CP89. (Case analysis [f , g]⊥ on D +⊥ E is decorated with ⊥ to avoid confusion with the case analysis for ordinary disjoint union of Agda types.)

The Cartesian products of types provided by the standard Agda library support products of domains, regarding a pair (⊥ , ⊥) as the least element of the product of two domains.

4.3 Method Systems

The method systems defined in CP89 are a simple formalization of object-oriented programming. They abstract from aspects such as instance variables, assignment, and object creation. A method system corresponds to a snapshot in the execution of an object-oriented system.

In CP89, the ingredients of method systems are assumed to be elements of flat domains; however, the least elements of these domains are irrelevant, and it is simpler to declare them as ordinary Agda types instead of domains:

```
( Instance  : Set ) – objects
( Name      : Set ) – class names
( Key       : Set ) – method names
( Primitive : Set ) – function names
```

Both the operational and denotational semantics of method systems in CP89 involve the mutually-recursive domains Value, Behavior, and Fun:

```
( Number   : Domain ) – unspecified
( Value    : Domain ) – a value is a behavior or a number
( Behavior : Domain ) – a behavior maps keys to funs
( Fun      : Domain ) – a fun maps values to values
```

These domains cannot be defined (safely) as Agda types, due to the termination check on recursive type definitions. Scott domain theory ensures the existence of isomorphisms between the types of elements of these domains when the elements of ⟨ Value ⟩ → ⟨ Value ⟩ are restricted to continuous functions. However, this restriction is irrelevant for checking the types of functions on domains, so it is omitted.

```
{{ iso^v     : ⟨ Value ⟩    ↔ ⟨ Behavior +⊥ Number ⟩ }}
{{ iso^b     : ⟨ Behavior ⟩ ↔ ( Key → ⟨ Fun +⊥ ?⊥ ⟩ ) }}
{{ iso^f     : ⟨ Fun ⟩      ↔ ( ⟨ Value ⟩ → ⟨ Value ⟩ ) }}
( apply⟦_⟧ : Primitive → ⟨ Value ⟩ → ⟨ Value ⟩ )
where
```

```
variable ρ : Instance; m : Key; f : Primitive; ν : ⟨ Number ⟩
variable α : ⟨ Value ⟩; σ π : ⟨ Behavior ⟩; ϕ : ⟨ Fun ⟩
```

In the operational and denotational semantics of method systems in CP89, elements f of the flat domain **Primitive** are treated as if they are elements of the function domain **Fun**. When checking the corresponding part of the Agda formulation, the Agda type checker reported this as an error. The semantic function apply⟦_⟧ declared above is assumed to map elements of Primitive to functions on ⟨ Value ⟩, and using it fixed the error (as did the introduction of the function *id* in the journal version of CP89 [6]).

In CP89, the inheritance hierarchy is assumed to be a finite tree. Below, Class is defined as the datatype of all finite trees. Using a datatype avoids the need for the partial *parent* function, and for a predicate for testing whether a class is the root of the hierarchy.

```
data Class : Set where
  child  : Name → Class → Class – a subclass
  origin : Class                – the root class
variable κ : Class
```

The syntax of method expressions is defined by the inductive datatype Exp:

```
data Exp : Set where
  self  : Exp                       – current object behavior
  super : Exp                       – superclass behavior
  arg   : Exp                       – method argument value
  call  : Exp → Key → Exp → Exp     – call method with argument
  appl  : Primitive → Exp → Exp     – apply primitive to value
variable e : Exp
```

```
module Semantics
  ( class    : Instance → Class )          – the class of an object
  ( methods′ : Class → Key → (Exp +?) ) – the methods of a class
  where
  methods : Class → Key → (Exp +?)       – no root class methods
  methods (child c κ) m = methods′ (child c κ) m
  methods origin m      = ??
```

4.4 Method Lookup Semantics

The method lookup semantics uses mutually-recursive functions send, lookup, and do⟦_⟧, which can be non-terminating. They are therefore defined in Agda as the least fixed point of a non-recursive function g (as in the proof of Proposition 3 in CP89) on a domain G^g that is isomorphic to D^g:

```
D^g = ( Instance → ⟨ Behavior ⟩ ) ×
      ( Class → Instance → ⟨ Behavior ⟩ ) ×
      ( Exp → Instance → Class → ⟨ Fun ⟩ )
```

```
module _
  { G^g : Domain }
  {{ iso^g : ⟨ G^g ⟩ ↔ D^g }}
  where
  g : Dᵍ → Dᵍ
  g (s , l , d⟦_⟧) = (send , lookup , do⟦_⟧) where
```

The behavior of send ρ is to use lookup (to be supplied as the argument l of g above) to obtain the behavior of ρ using the class of ρ itself:

```
send : Instance → 〈 Behavior 〉
send ρ = l (class ρ) ρ
```

The behavior of lookup κ ρ for a subclass κ depends on whether it is called with a method m defined by κ: if so, it uses do⟦ e ⟧ (via argument d⟦_⟧ of g) to execute the corresponding method expression; if not, it recursively looks up m in the superclass of κ. The behavior is undefined when κ is the root of the inheritance hierarchy, which has been defined to have no methods:

```
lookup : Class → Instance → 〈 Behavior 〉
lookup (child c κ) ρ =
  from λ m → [ ( λ e → inl (d⟦ e ⟧ ρ (child c κ)) ) ,
                 ( to (l κ ρ) m )
               ]? (methods (child c κ) m)
lookup origin ρ = ⊥
```

When applied to a value α, the value returned by the function to (do⟦ e ⟧ ρ κ) may be a behavior, a number, or undefined ($\perp$):

```
do⟦_⟧ : Exp → Instance → Class → 〈 Fun 〉
do⟦ self      ⟧ ρ κ           = from λ α → from (inl (s ρ))
do⟦ super     ⟧ ρ (child c κ) = from λ α → from (inl (l κ ρ))
do⟦ super     ⟧ ρ origin      = from λ α → ⊥
do⟦ arg       ⟧ ρ κ           = from λ α → α
do⟦ call e₁ m e₂ ⟧ ρ κ =
  from λ α → [ ( λ σ → [ ( λ ϕ → to ϕ (to (d⟦ e₂ ⟧ ρ κ) α) ) ,
                           ( λ _ → ⊥ )
                         ]⊥ (to σ m) ) ,
                 ( λ ν → ⊥ )
               ]⊥ (to (to (d⟦ e₁ ⟧ ρ κ) α))
do⟦ appl f e₁    ⟧ ρ κ =
  from λ α → apply⟦ f ⟧ (to (d⟦ e₁ ⟧ ρ κ) α)
```

The only complicated case is for calling method m of object e_1 with argument e_2. When the value of e_1 is a behavior σ that maps m to a function ϕ, that function is applied to the value of e_2; otherwise the value of the call is undefined. The undefined cases are not explicit in CP89.

The use of fix below has the effect of making the above definitions mutually recursive:

```
γ : 〈 Gᵍ 〉 → 〈 Gᵍ 〉
γ = from ∘ g ∘ to
send    = proj₁ (to (fix γ))
lookup = proj₁ (proj₂ (to (fix γ)))
do⟦_⟧   = proj₂ (proj₂ (to (fix γ)))
```

That concludes the Agda definition of the method lookup semantics.

4.5 Denotational Semantics

The denotational semantics of method expressions takes the behavior of the expressions self (σ) and super (π) as arguments, so their evaluation is trivial. The evaluation of the other method expressions is similar to their method lookup semantics.

```
eval⟦_⟧ : Exp → 〈 Behavior 〉 → 〈 Behavior 〉 → 〈 Fun 〉
eval⟦ self        ⟧ σ π = from λ α → from (inl σ)
eval⟦ super       ⟧ σ π = from λ α → from (inl π )
eval⟦ arg         ⟧ σ π = from λ α → α
eval⟦ call e₁ m e₂ ⟧ σ π =
  from λ α → [ ( λ σ′ → [ ( λ ϕ → to ϕ (to (eval⟦ e₂ ⟧ σ π) α) ) ,
                            ( λ _ → ⊥ )
                          ]⊥ (to σ′ m) ) ,
                 ( λ ν → ⊥ )
               ]⊥ (to (to (eval⟦ e₁ ⟧ σ π) α))
eval⟦ appl f e₁    ⟧ σ π =
  from λ α → apply⟦ f ⟧ (to (eval⟦ e₁ ⟧ σ π) α)
```

The recursively-defined function eval⟦_⟧ is obviously total, so there is no need for an explicit fixed point.

According to the conceptual analysis of inheritance in CP89, the behavior of an instance ρ is the fixed point of the generator associated with the class of ρ.

The generator for a subclass is obtained by modifying the generator of its parent class using a wrapper that provides the behavior of the methods defined by the subclass, given the behavior of the expressions self (σ) and super (π) as arguments.

The auxiliary operation $\sigma_1 \oplus \sigma_2$ combines its argument behaviors, letting the methods of σ_1 shadow those of σ_2. The operation w ⊳ p combines the wrapper of a subclass with the generator of its parent class. See Figure 9 of CP89 for an illustration of wrapper application.

```
Generator = 〈 Behavior 〉 → 〈 Behavior 〉
Wrapper = 〈 Behavior 〉 → 〈 Behavior 〉 → 〈 Behavior 〉

_⊕_ : 〈 Behavior 〉 → 〈 Behavior 〉 → 〈 Behavior 〉
σ₁ ⊕ σ₂ = from λ m →
  [ ( λ ϕ → inl ϕ ) , ( λ _ → to σ₂ m ) ]⊥ (to σ₁ m)
_⊳_ : Wrapper → Generator → Generator
w ⊳ p = λ σ → (w σ (p σ)) ⊕ (p σ)
wrap : Class → Wrapper
wrap κ = λ σ → λ π → from λ m →
  [ ( λ e → inl (eval⟦ e ⟧ σ π) ), ( inr ⊥ ) ]? (methods κ m)

gen : Class → Generator
gen (child c κ) = wrap (child c κ) ⊳ gen κ
gen origin      = λ σ → ⊥
behave : Instance → 〈 Behavior 〉
behave ρ = fix (gen (class ρ))
```

That concludes the Agda definition of the denotational semantics.

5 Equivalence

This section presents an Agda formulation of the intermediate semantics given in Section 4.4 of CP89. The intermediate semantics is a step-indexed version of the method lookup semantics. The section starts by recalling the origins of step-indexing, and concludes by showing the Agda proof of the first lemma in CP89; the proofs of the other three lemmas are available in the auxiliary material.

5.1 Step-Indexing

Theorem 1 in CP89 states the equivalence of the operational (method lookup) semantics and the denotational semantics: *send* = *behave*. Quoting from CP89:

> In the proof of the theorem we use an "intermediate semantics" … [15] … The semantics uses $n \in$ **Nat**, the flat domain of natural numbers. The intermediate semantics resembles the method lookup semantics but differs in that each of the syntactic domains of instances, classes, and expressions has a whole family of semantic equations, indexed by natural numbers.

The intermediate semantics defines the behavior of an instance by the following functions:

$$send' : \mathbf{Nat} \to \mathbf{Instance} \to \mathbf{Behavior} \quad (1)$$
$$lookup' : \mathbf{Nat} \to \mathbf{Class} \to \mathbf{Instance} \to \mathbf{Behavior} \quad (2)$$
$$do' : \mathbf{Nat} \to \mathbf{Exp} \to \mathbf{Instance} \to \mathbf{Class} \to \mathbf{Fun} \quad (3)$$

The intuition is that $send'_n\rho$ has 'fuel' for up to $n - 1$ uses of **self**, and the nth use gives $\perp$.

The cited intermediate semantics in [15] is for the lambda-calculus. There, the indices are taken from the *chain cpo* of natural numbers augmented by ∞, and continuity implies monotonicity. The intermediate semantics gives $\perp$ at 0, and corresponds to the standard (un-indexed) semantics at ∞. In CP89, the use of a *flat* domain of indices supports testing whether an index is zero, but monotonicity w.r.t. *numerical* order had to be proved.

Perhaps the earliest use of step-indexing was by Christopher Wadsworth, in 1976 [28]. Quoting from [15]:

> Wadsworth solved [a] problem in his study of the λ-calculus … by labelling expressions M (and their subexpressions) with integers n, so that $M^{(n)}$ denoted the nth projection of the denotation of M. Having introduced some extra syntax to make the levels visible, he then studied the operational properties of the $M^{(n)}$ induced by this semantics and also their relation to the operational properties of the original M. Thus one parameter – the labelling – was used both for inductions relating to the denotational semantics (the projection levels) and for inductions relating to the operational semantics.

More recently, abstract versions of step-indexed models of programming languages have been constructed in the internal logic of the topos of trees [3], avoiding explicit indices.

5.2 Intermediate Semantics

```
module Inheritance.Equivalence
```

The imports and parameters of this module are the same as those of Inheritance.Definitions, and elided here.

```
  where
open import Inheritance.Definitions
  ( Domain ) ( ⟨_⟩ ) ( ⊥ ) ( fix ) ( ?⊥ )
  ( _+⊥_ ) ( inl ) ( inr ) ( [_,_]⊥ )
  ( Instance ) ( Name ) ( Key ) ( Primitive )
  ( Number ) ( Value ) ( Behavior ) ( Fun )
  {{ isoᵛ }} {{ isoᵇ }} {{ isoᶠ }} ( apply⟦_⟧ )

module _
  ( class     : Instance → Class )
  ( methods′ : Class → Key → (Exp +?) )
  where
  open Semantics ( class ) ( methods′ )
```

The intermediate semantics of method expressions given in CP89 is a step-indexed variant of the method lookup semantics. It takes an extra argument n ranging over **Nat** (the set of natural numbers), which acts as sufficient 'fuel' for up to $n - 1$ uses of **self**: when n is zero, the intermediate semantics is the undefined behavior ($\perp$). One of the lemmas proved in CP89 shows that the intermediate semantics at n corresponds to the nth approximation to the denotational semantics.

The functions used to specify the intermediate semantics are mutually recursive, in the same way as in the method lookup semantics. Here, however, the finiteness of the fuel argument ensures that the functions are total, so they can be defined in Agda without an explicit least fixed-point:

```
send′    : ℕ → Instance → ⟨ Behavior ⟩
lookup′ : ℕ → Class → Instance → ⟨ Behavior ⟩
do′_⟦_⟧ : ℕ → Exp → Instance → Class → ⟨ Fun ⟩

send′ n ρ = lookup′ n (class ρ) ρ

lookup′ zero κ ρ = ⊥
lookup′ n (child c κ) ρ =
  from λ m → [ ( λ e → inl (do′ n ⟦ e ⟧ ρ (child c κ)) ) ,
               ( to (lookup′ n κ ρ ) m )
             ]? (methods (child c κ) m)
lookup′ n origin ρ = ⊥
```

Cases of Agda definitions are sequential: putting a case for zero before the corresponding case for n implies that n is positive in the latter.

```
do′ zero       ⟦ e    ⟧ ρ κ = ⊥

do′ (suc n)    ⟦ self ⟧ ρ κ = from λ α → from (inl (send′ n ρ))
```

```
do′ n ⟦ super ⟧ ρ (child c κ) =
  from λ α → from (inl (lookup′ n κ ρ))
do′ n ⟦ super ⟧ ρ origin = from λ α → ⊥
do′ n ⟦ arg ⟧ ρ κ = from λ α → α
do′ n ⟦ call e₁ m e₂ ⟧ ρ κ =
  from λ α → [ ( λ σ → [ ( λ ϕ → to ϕ (to (do′ n ⟦ e₂ ⟧ ρ κ) α) ) ,
                          ( λ _ → ⊥ )
                        ]⊥ (to σ m) ) ,
               ( λ ν → ⊥ )
             ]⊥ (to (to (do′ n ⟦ e₁ ⟧ ρ κ ) α))
do′ n ⟦ appl f e₁ ⟧ ρ κ =
  from λ α → apply⟦ f ⟧ (to (do′ n ⟦ e₁ ⟧ ρ κ) α)
```

5.3 Proofs of Lemmas in Agda

The proofs of the lemmas use the following additional modules from the standard library:

```
open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl; cong; sym)
open import Relation.Binary.PropositionalEquality.Properties
open import Relation.Binary.Reasoning.Syntax
open ≡-Reasoning
open import Axiom.Extensionality.Propositional
  using (Extensionality)
open import Level
  renaming (zero to lzero) hiding (suc)

module _ ( ext : Extensionality lzero lzero )
  where
```

Lemma 1 establishes a significant fact about the relationship between the denotational semantics and the intermediate semantics of method systems. Its Agda proof exhibits the equational reasoning steps of the original proof in CP89. This checks the correctness not only of the stated result, but also of the steps themselves.

The Agda standard library defines the following notation for equational reasoning:

- x ≡ y asserts the equality of x and y;
- begin starts a proof;
- ≡⟨⟩ adds a step that Agda can check automatically;
- ≡⟨ t ⟩ adds a step with an explicit proof term t; and
- ◻ concludes a proof.

```
lemma-1 : ∀ n e ρ c κ →
  do′ (suc n) ⟦ e ⟧ ρ (child c κ) ≡
  eval⟦ e ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
```

The restriction to the class child c κ ensures that it is not the root class; in CP89, the use of *parent*(κ) as an argument of type **Class** in the statement of Lemma 1 leaves this restriction implicit.

The proof of this lemma is a straightforward structural induction:

```
lemma-1 n self ρ c κ =
  begin do′ (suc n) ⟦ self ⟧ ρ (child c κ)
  ≡⟨⟩    ( from λ α → from (inl (send′ n ρ)) )
  ≡⟨⟩    eval⟦ self ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
  ◻
lemma-1 n super ρ c (child c′ κ) =
  begin do′ (suc n) ⟦ super ⟧ ρ (child c (child c′ κ))
  ≡⟨⟩    ( from λ α → from (inl (lookup′ (suc n) (child c′ κ) ρ)) )
  ≡⟨⟩    eval⟦ super ⟧ (send′ n ρ) (lookup′ (suc n) (child c′ κ) ρ)
  ◻
lemma-1 n super ρ c origin =
  begin do′ (suc n) ⟦ super ⟧ ρ (child c origin)
  ≡⟨⟩    ( from λ α → from (inl ⊥ ) )
  ≡⟨⟩    eval⟦ super ⟧ (send′ n ρ) (lookup′ (suc n) origin ρ)
  ◻
lemma-1 n arg ρ c κ =
  begin do′ (suc n) ⟦ arg ⟧ ρ (child c κ)
  ≡⟨⟩    ( from λ α → α )
  ≡⟨⟩    eval⟦ arg ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
  ◻
```

The equational reasoning proof steps in the inductive case for calling a method are quite complicated in Agda. This is mainly due to the case analysis required in the semantics of method calls to make the semantics type-correct (these cases are left implicit in CP89). The use of rewrite below succinctly verifies the correctness of the case analysis:

```
lemma-1 n (call e₁ m e₂) ρ c κ
  rewrite (lemma-1 n e₁ ρ c κ)
  rewrite (lemma-1 n e₂ ρ c κ) = refl
```

The inductive case for applying a primitive function is relatively simple, and concludes the proof of Lemma 1.

```
lemma-1 n (appl f e₁) ρ c κ =
  begin
    do′ (suc n) ⟦ appl f e₁ ⟧ ρ (child c κ)
  ≡⟨⟩
    ( from λ α →
        apply⟦ f ⟧
          (to (do′ (suc n) ⟦ e₁ ⟧ ρ (child c κ)) α) )
  ≡⟨ use-induction ⟩
    ( from λ α →
        apply⟦ f ⟧
          (to (eval⟦ e₁ ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)) α) )
  ≡⟨⟩
    eval⟦ appl f e₁ ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
  ◻
  where
    use-induction =
      cong from (ext λ α →
        cong (λ X →
          apply⟦ f ⟧ ((to X) α)) (lemma-1 n e₁ ρ c κ))
```

The Agda proofs of the remaining lemmas are available in the accompanying auxiliary material.

```
lemma-2 : ∀ κ n ρ → gen κ (send′ n ρ) ≡ lookup′ (suc n) κ ρ
```

```
iter : {D : Domain} → ℕ → ( 〈 D 〉 → 〈 D 〉 ) → 〈 D 〉
iter zero g    = ⊥
iter (suc n) g = g (iter n g)

lemma-3 : ∀ n ρ → iter n (gen (class ρ)) ≡ send′ n ρ

lemma-4-send′ : ∀ n ρ →
  send′ n ρ ⊑ send′ (suc n) ρ

lemma-4-lookup′ : ∀ n κ ρ →
  lookup′ n κ ρ ⊑ lookup′ (suc n) κ ρ

lemma-4-do′ : ∀ n e ρ c κ →
  do′ (suc n) ⟦ e ⟧ ρ (child c κ) ⊑
  do′ (suc (suc n)) ⟦ e ⟧ ρ (child c κ)
```

5.4 Remaining Results

When Agda proofs of the remaining propositions and correctness theorem from CP89 have been developed, they are to be made available at https://github.com/pdmosses/jensfest-agda.

```
module _
  { Gᵍ : Domain }
  {{ isoᵍ : 〈 Gᵍ 〉 ↔ Dᵍ }}
  ( lub : {D : Domain} → (δ : ℕ → 〈 D 〉) → 〈 D 〉 )
  where

  interpret : Instance → 〈 Behavior 〉
  interpret ρ = lub (λ n → send′ n ρ)

  proposition-1 : ∀ ρ → interpret ρ ≡ behave ρ
  proposition-2 : ∀ ρ → behave ρ ⊑ send ρ
  proposition-3 : ∀ ρ → send ρ ⊑ interpret ρ
  theorem-1 :     ∀ ρ → send ρ ≡ behave ρ
```

6 Conclusion and Future Work

The definitions and proofs in the OOPSLA '89 paper (CP89) [5] are elegant, clear, and convincing. However, I started wondering whether it might simplify the proofs to use the *chain cpo* of natural numbers (augmented by ∞) for step-indexing, instead of the flat domain of natural numbers. I realized that I would need to validate any claims using a proof assistant; I had no previous practical experience with any, but this seemed a good opportunity to learn Agda. The present paper is essentially a progress report; completion of the Agda verification of the remaining results is future work.

In published presentations of semantic definitions, it is common practice to leave some notational details implicit. CP89 is no exception: it leaves injections into sum domains implicit, and doesn't mention the isomorphisms between recursively defined domains. To type-check the semantics in Agda, however, all injections and isomorphisms had to be made explicit.

The proofs in CP89 naturally focus on the important steps, and skip over tedious low-level details. Moreover, some assumptions in CP89 are stated informally, e.g., "*parent* defines the inheritance hierarchy, which is required to be a tree", and "the root of the inheritance hierarchy doesn't define any methods" [5, §4.1]. Agda provides plenty of assistance by automating low-level details in proofs, but to check the proofs, these assumptions had to be formally specified.

It is well known [9] that *running* a specification can reveal unsuspected errors and omissions in presentations of semantic definitions and proofs. The Agda type-checker revealed the following minor issues, which stemmed from Figs. 14, 16, and 17 of CP89:

- The primitive f in the syntax of method expressions is used as a function in the semantics.[9]
- The value of e_1 in a message-passing expression $e_1\ m\ e_2$ might be either a behavior or a number, but it is applied to m in both cases.
- The value of $parent(\kappa)$ is in **Class + ?**, but it is used as an argument supposed to be in **Class**. The semantics of **super** should check that $root(\kappa)$ isn't true.

Checking the reformulated proofs of Lemmas 1–4 using Agda revealed only the missing cases related to the type mismatches mentioned above. Reformulating the proofs of the remaining results and checking them using Agda is future work; but even when that has been completed, it will verify only that the results stated in CP89 are sound *relative to the various assumptions made in the Agda modules.* To verify that the results are unconditionally sound, it would be necessary to *discharge* the assumptions by providing definitions for all the declared symbols, which might require the use of guarded type theory.

Section 3 discusses issues with exploiting some available Agda libraries for Scott domain theory. Scott domains have some significant advantages over Agda types: domains can be recursively defined (up to isomorphism) without the need for guards, and functions defined on domains using lambda-expressions are automatically continuous. Perhaps some future version of Agda could support declaring particular types to be Scott domains, with built-in notation for the partial order on their elements, their bottom elements, and least fixed-points?

Acknowledgments

The anonymous reviewers of the submitted version made helpful comments and suggestions for improvement. Casper Bach Poulsen, Cas van der Rest, and Jesper Cockx gave me expert advice on how to use Agda; Casper also provided the two most complicated proof steps for Lemma 2, and Jesper drew my attention to the Agda support for guarded type theory and synthetic denotational semantics.

[9]The journal version [6] of CP89 already fixed this issue.

References

[1] Samson Abramsky and Achim Jung. 1995. Domain Theory. In *Handbook of Logic in Computer Science*. Oxford University Press. https://doi.org/10.1093/oso/9780198537625.003.0001

[2] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. 1993. Type inference of SELF. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 707)*, Oscar Nierstrasz (Ed.). Springer, 247–267. https://doi.org/10.1007/3-540-47910-4_14

[3] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science* Volume 8, Issue 4 (Oct. 2012), 45 pages. https://doi.org/10.2168/lmcs-8(4:1)2012

[4] William R. Cook. 1989. *A Denotational Semantics of Inheritance.* Ph. D. Dissertation. Brown University. https://cs.brown.edu/research/pubs/theses/phd/1989/cook.pdf

[5] William R. Cook and Jens Palsberg. 1989. A denotational semantics of inheritance and its correctness. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA 1989, New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings*, George Bosworth (Ed.). ACM, 433–443. https://doi.org/10.1145/74877.74922

[6] William R. Cook and Jens Palsberg. 1994. A denotational semantics of inheritance and its correctness. *Inf. Comput.* 114, 2 (1994), 329–350. https://doi.org/10.1006/INCO.1994.1090

[7] Tom de Jong. 2021. The Scott model of PCF in Univalent Type Theory. *Math. Struct. Comput. Sci.* 31, 10 (2021), 1270–1300. https://doi.org/10.1017/S0960129521000153

[8] Tom de Jong. 2022. *Domain Theory in Constructive and Predicative Univalent Foundations.* Ph. D. Dissertation. University of Birmingham. https://etheses.bham.ac.uk/id/eprint/13401/

[9] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 285–296. https://doi.org/10.1145/2103656.2103691

[10] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1992. Efficient inference of partial types. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*. IEEE Computer Society, 363–371. https://doi.org/10.1109/SFCS.1992.267754

[11] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1993. Efficient recursive subtyping. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 419–428. https://doi.org/10.1145/158511.158700

[12] Conor McBride. 2015. Turing-completeness totally free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

[13] Peter D. Mosses. 1992. *Action Semantics.* Cambridge University Press. https://doi.org/10.1017/CBO9780511569869

[14] Peter D. Mosses. 1996. Theory and practice of action semantics. In *Mathematical Foundations of Computer Science 1996, 21st International Symposium, MFCS'96, Cracow, Poland, September 2-6, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1113)*, Wojciech Penczek and Andrzej Szalas (Eds.). Springer, 37–61. https://doi.org/10.1007/3-540-61550-4_139

[15] Peter D. Mosses and Gordon D. Plotkin. 1987. On proving limiting completeness. *SIAM J. Comput.* 16, 1 (1987), 179–194. https://doi.org/10.1137/0216015

[16] Peter D. Mosses and David A. Watt. 1987. The use of action semantics. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 135–166.

[17] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. 1992. Making type inference practical. In *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands, June 29 - July 3, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 615)*, Ole Lehrmann Madsen (Ed.). Springer, 329–349. https://doi.org/10.1007/BFB0053045

[18] Jens Palsberg. 1992. *Provably Correct Compiler Generation.* Ph. D. Dissertation. University of Aarhus. https://doi.org/10.7146/dpb.v21i422.6736 DAIMI Report Series PB-422.

[19] Jens Palsberg (Ed.). 2009. *Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday.* Lecture Notes in Computer Science, Vol. 5700. Springer. https://doi.org/10.1007/978-3-642-04164-8

[20] Jens Palsberg and Michael I. Schwartzbach. 1991. Object-oriented type inference. In *Proceedings of the Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1991, Phoenix, Arizona, USA, October 6-11, 1991*, Andreas Paepcke (Ed.). ACM, 146–161. https://doi.org/10.1145/117954.117965

[21] Jens Palsberg and Michael I. Schwartzbach. 1991. What is type-safe code reuse?. In *ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 512)*, Pierre America (Ed.). Springer, 325–341. https://doi.org/10.1007/BFB0057030

[22] Jens Palsberg and Michael I. Schwartzbach. 1992. Safety analysis versus type inference for partial types. *Inf. Process. Lett.* 43, 4 (1992), 175–180. https://doi.org/10.1016/0020-0190(92)90196-3

[23] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A model of PCF in Guarded Type Theory. *Electronic Notes in Theoretical Computer Science* 319 (2015), 333–349. https://doi.org/10.1016/j.entcs.2015.12.020 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)..

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

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

[26] Michael B. Smyth and Gordon D. Plotkin. 1982. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.* 11, 4 (1982), 761–783. https://doi.org/10.1137/0211062

[27] Christopher Strachey. 1966. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming, Proc. IFIP Working Conference, 1964.* North-Holland, 198–220.

[28] Christopher P. Wadsworth. 1976. The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM J. Comput.* 5, 3 (1976), 488–521. https://doi.org/10.1137/0205036

Received 2024-06-09; accepted 2024-07-12