

Towards Verification of a Denotational Semantics of Inheritance

Peter D Mosses

TU Delft (visitor)
Swansea University (emeritus)

JENSFEST, Pasadena, California, USA, 22 October 2024

Jens Palsberg

1988–1992: PhD studies, Aarhus University, Denmark

1989

- W. R. Cook, J. Palsberg:
A Denotational Semantics of
Inheritance and its Correctness.
OOPSLA 1989

1990

- J. Palsberg, M. I. Schwartzbach:
Type Substitution for Object-
Oriented Programming.
OOPSLA/ECOOP 1990

1991

- J. Palsberg, M. I. Schwartzbach:
What is Type-Safe Code Reuse?
ECOOP 1991
- J. Palsberg, M. I. Schwartzbach:
Object-Oriented Type Inference.
OOPSLA 1991

1992

- J. Palsberg: A Provably Correct
Compiler Generator. **ESOP** 1992
- J. Palsberg, M. I. Schwartzbach:
Three discussions on object-
oriented typing.
OOPS Messenger
- J. Palsberg: An automatically
generated and provably correct
compiler for a subset of Ada.
ICCL 1992

M. I. Schwartzbach, J. Palsberg:
Types for the language designer
(abstract).

OOPSLA Addendum 1992

D. Kozen, J. Palsberg,
M. I. Schwartzbach: Efficient
Inference of Partial Types.
FOCS 1992

J. Palsberg: Provably correct
compiler generation.
PhD thesis, Aarhus University

1993

- N. Oxhøj, J. Palsberg,
M. I. Schwartzbach: Making
Type Inference Practical.
ECOOP 1992
- J. Palsberg, M. I. Schwartzbach:
Safety Analysis Versus Type
Inference for Partial Types.
Inf. Process. Lett.
- J. Palsberg: Normal Forms Have
Partial Types. **Inf. Process. Lett.**
- J. Palsberg: Correctness of
Binding-Time Analysis.
J. Funct. Program.

D. Kozen, J. Palsberg,
M. I. Schwartzbach: Efficient
Recursive Subtyping.
POPL 1993

A. Bondorf, J. Palsberg:
Compiling Actions by Partial
Evaluation. **FPCA** 1993

O. Agesen, J. Palsberg,
M. I. Schwartzbach: Type
Inference of SELF. **ECOOP** 1993

M. Banahan, L. P. Deutsch,
B. Magnusson, J. Palsberg:
Panel: Aims, Means, and Future
of Object-Oriented Languages.
ECOOP 1993

A. Cheng, J. Esparza,
J. Palsberg: Complexity Results
for 1-safe Nets. **FSTTCS** 1993

OOPSLA '89

A Denotational Semantics of Inheritance and its Correctness



(1963–2021)

William Cook*
Department of Computer Science
Box 1910 Brown University

Jens Palsberg
Computer Science Department
Aarhus University



This paper presents a denotational model of inheritance. The model is based on an intuitive motivation of the purpose of inheritance. The correctness of the model is demonstrated by proving it equivalent to an operational semantics of inheritance based upon the method-lookup algorithm of object-oriented languages. . . .

Verification

OOPSLA '89 paper

- ▶ simple ***semantic definitions***
- ▶ elegant ***equivalence proof***, using ***step-indexing***

My initial aims for a JENSFEST submission

- ▶ ***verify*** the definitions and the proof
- ▶ experiment with an ***alternative form of step-indexing***
- ▶ use a ***proof assistant***



The rest of this talk

Denotational semantics

- fixed points

Semantics of inheritance

- operational (method-lookup)
- denotational
- in Agda

Towards verification of correctness

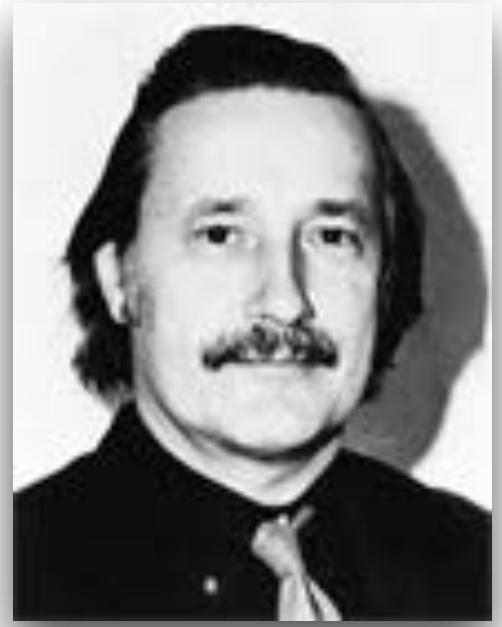
- intermediate semantics (step-indexed) and the proof of a lemma

Denotational semantics and fixed points

– origins

Christopher Strachey: *Towards a Formal Semantics* (1966)

It seems that the imperative features of a programming language introduce some basically new ideas not easily incorporated into a descriptive system. In Landin's paper here on the semantics of ALGOL, he gets over this problem by an alteration of his evaluating mechanism, which now becomes an essential feature of the description of ALGOL.

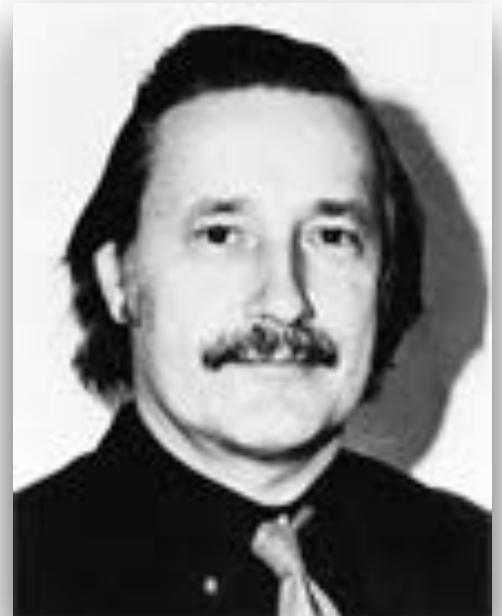


It is the aim of this paper to show that with the introduction of a few new basic concepts it is possible to describe even the imperative parts of a programming language in terms of applicative expressions, including even such "essentially computer-oriented" ideas as assignment and transfers of control. As a result, it is no longer an essential part of the basic semantic description of the language to postulate an "evaluating mechanism" or notional computer, however abstract.

Denotational semantics and fixed points

– origins

Christopher Strachey: *Towards a Formal Semantics* (1966)



- ▶ semantics : programs → mathematical ***functions*** ?
- ▶ ***fixed-point operator*** : $\mathbf{Y}(f) = (\lambda x. f(\mathbf{x}\ \mathbf{x})) (\lambda x. f(\mathbf{x}\ \mathbf{x})) = f(\mathbf{Y}(f)) = f(f(\mathbf{Y}(f))) = \dots$

Dana Scott: *Outline of a Mathematical Theory of Computation* (1970)



- ▶ ***domains*** : $D_\infty = (D_\infty \rightarrow D_\infty)$ – a model of the λ -calculus, *continuous* functions
- ▶ least ***fixed-point function*** $\mathbf{fix} : (D \rightarrow D) \rightarrow D$; $\mathbf{fix}(f) = \sqcup f^n(\perp) = f(\mathbf{fix}(f))$

Scott & Strachey: *Toward a Mathematical Semantics for Computer Languages* (1971)

- ▶ semantics $\llbracket _ \rrbracket$: programs → ***elements of domains*** (*lattices, cpos*)

Semantics of Inheritance

Method systems and semantic domains

– OOPSLA '89

Method System Domains				Semantic Domains			
Instances	ρ	\in	Instance			Number	
Classes	κ	\in	Class	α	\in	Value	= Behavior + Number
Messages	m	\in	Key	σ, π	\in	Behavior	= Key \rightarrow (Fun + ?)
Primitives	f	\in	Primitive	ϕ	\in	Fun	= Value \rightarrow Value
Methods	e	\in	Exp := self super arg $e_1 \ m \ e_2$ $f(e_1, \dots, e_q)$				
Method System Operations							
class	:	Instance	\rightarrow	Class	root : Class \rightarrow Boolean		
parent	:	Class	\rightarrow	(Class + ?)	$root(\kappa) = [\lambda \kappa' \in \text{Class} . \text{false},$ $\lambda v \in ? . \text{true}$](parent(\kappa))		
methods	:	Class	\rightarrow	Key \rightarrow (Exp + ?)			

Method lookup semantics

– OOPSLA '89

$\text{send} : \text{Instance} \rightarrow \text{Behavior}$
 $\text{send}(\rho) = \text{lookup}(\text{class}(\rho))\rho$

$\text{lookup} : \text{Class} \rightarrow \text{Instance} \rightarrow \text{Behavior}$
 $\text{lookup}(\kappa)\rho = \lambda m \in \text{Key} .$
 $[\lambda e \in \text{Exp} . \text{do}[e]\rho\kappa,$
 $\lambda v \in ? . \text{if } \text{root}(\kappa)$
 $\text{then } \perp ?$
 $\text{else } \text{lookup}(\text{parent}(\kappa))\rho m$

$](\text{methods}(\kappa)m)$

parent : Class → (Class + ?)

$\text{do} : \text{Exp} \rightarrow \text{Instance} \rightarrow \text{Class} \rightarrow \text{Fun}$
 $\text{do}[\text{self}]\rho\kappa = \lambda \alpha . \text{send}(\rho)$
 $\text{do}[\text{super}]\rho\kappa = \lambda \alpha . \text{lookup}(\text{parent}(\kappa))\rho$
 $\text{do}[\text{arg}]\rho\kappa = \lambda \alpha . \alpha$
 $\text{do}[e_1 \ m \ e_2]\rho\kappa = \lambda \alpha . (\text{do}[e_1]\rho\kappa\alpha) m (\text{do}[e_2]\rho\kappa\alpha)$
 $\text{do}[f(e_1, \dots, e_q)]\rho\kappa =$
 $\lambda \alpha . f(\text{do}[e_1]\rho\kappa\alpha, \dots, \text{do}[e_q]\rho\kappa\alpha)$

Acknowledgement. The authors would like to thank Peter Mosses for helpful comments on the second part of the paper, and John Mitchell for comments on an earlier draft.

Denotational semantics

– OOPSLA '89

Generator Semantics Domains

Generator	$=$	Behavior \rightarrow Behavior
Wrapper	$=$	Behavior \rightarrow Generator

behave : **Instance** \rightarrow **Behavior**
 $\text{behave}(\rho) = \text{fix}(\text{gen}(\text{class}(\rho)))$

gen : **Class** \rightarrow **Generator**
 $\text{gen}(\kappa) = \text{if } \text{root}(\kappa)$
 then $\lambda \sigma \in \text{Behavior}. \lambda m \in \text{Key}. \perp?$
 else $\text{wrap}(\kappa) \blacktriangleright \text{gen}(\text{parent}(\kappa))$

wrap : **Class** \rightarrow **Wrapper**
 $\text{wrap}(\kappa) = \lambda \sigma. \lambda \pi. \lambda m \in \text{Key}.$
 $[\lambda e \in \text{Exp}. \text{eval}[e]\sigma\pi$
 $\lambda v \in ?. \perp?$
 $] \text{methods}(\kappa)m$

eval : **Exp** \rightarrow **Behavior** \rightarrow **Behavior** \rightarrow **Fun**
 $\text{eval}[\text{self}]\sigma\pi = \lambda \alpha. \sigma$
 $\text{eval}[\text{super}]\sigma\pi = \lambda \alpha. \pi$
 $\text{eval}[\text{arg}]\sigma\pi = \lambda \alpha. \alpha$
 $\text{eval}[e_1 \ m \ e_2]\sigma\pi =$
 $\quad \lambda \alpha. (\text{eval}[e_1]\sigma\pi\alpha) m (\text{eval}[e_2]\sigma\pi\alpha)$
 $\text{eval}[f(e_1, \dots, e_q)]\sigma\pi =$
 $\quad \lambda \alpha. f(\text{eval}[e_1]\sigma\pi\alpha, \dots, \text{eval}[e_q]\sigma\pi\alpha)$

$$W \blacktriangleright P = \lambda \text{self}. (W(\text{self})(P(\text{self}))) \oplus P(\text{self})$$

Denotational semantics

– reformulation in Agda

Agda

- ▶ a pure functional *programming language*
- ▶ with *dependent types*
 - e.g., $\text{__} : \{A : \text{Set}\} \rightarrow \{n : \mathbb{N}\} \rightarrow \text{Vec } A \ n \rightarrow \text{Fin } n \rightarrow A$
- ▶ also a *proof assistant*
 - Curry–Howard correspondence: *propositions-as-types*
"a proof is a program, and the formula it proves is the type for the program"

Method systems and semantic domains

– reformulation in Agda

module Inheritance.Definitions

```
( Domain : Set1 )
( ⟨_⟩      : Domain → Set )
( ⊥       : {D : Domain} → ⟨ D ⟩ )
( fix     : {D : Domain} → ( ⟨ D ⟩ → ⟨ D ⟩ ) → ⟨ D ⟩ )
( Instance : Set ) – objects
( Name    : Set ) – class names
( Key     : Set ) – method names
( Primitive : Set ) – function names
```

data Class : Set where

```
child : Name → Class → Class – a subclass
origin : Class                         – the root class
```

(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
{ iso ^v : ⟨ Value ⟩	\leftrightarrow ⟨ Behavior + _⊥ Number ⟩ }
{ iso ^b : ⟨ Behavior ⟩	\leftrightarrow (Key → ⟨ Fun + _⊥ ? _⊥ ⟩) }
{ iso ^f : ⟨ Fun ⟩	\leftrightarrow (⟨ Value ⟩ → ⟨ Value ⟩) }

to, from (inverse)

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

Denotational semantics

– reformulation in Agda

```

eval[] : Exp → ⟨ Behavior ⟩ → ⟨ Behavior ⟩ → ⟨ Fun ⟩
eval[] self      ] σ π = λ α → (inl σ)
eval[] super     ] σ π = λ α → (inl π)
eval[] arg       ] σ π = λ α → α
eval[] call e₁ m e₂ ] σ π =
    λ α → [ ( λ σ' → [ ( λ φ → φ ( (eval[] e₂] σ π) α) ) ,
        ( λ _ → ⊥ )
    ]⊥ ( σ' m) ) ,
    ( λ ν → _ )
]⊥ ( ( (eval[] e₁] σ π) α))
eval[] appl f e₁ ] σ π =
    λ α → apply[] f ( (eval[] e₁] σ π) α)

```

```

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))

```

Towards Verification ...

Intermediate semantics (step-indexed)

– OOPSLA '89

$\text{send}' : \boxed{\text{Nat}} \rightarrow \text{Instance} \rightarrow \text{Behavior}$

$\text{send}'_0(\rho) = \perp$

$\text{send}'_n(\rho) = \text{lookup}'_n(\text{class}(\rho))\rho \quad n > 0$

$\text{lookup}' : \boxed{\text{Nat}} \rightarrow \text{Class} \rightarrow \text{Instance} \rightarrow \text{Behavior}$

$\text{lookup}'_0 \kappa \rho = \perp$

if $n > 0$ *then*

$\text{lookup}'_n \kappa \rho = \lambda m \in \text{Key}.$

$[\lambda e \in \text{Exp}. \text{do}'_n[e]\rho\kappa,$

$\lambda v \in ?. \text{if } \text{root}(\kappa)$

then $\perp?$

else $\text{lookup}'_n(\text{parent}(\kappa))\rho m$

](*methods*(κ) m)

$\text{do}' : \boxed{\text{Nat}} \rightarrow \text{Exp} \rightarrow \text{Instance} \rightarrow \text{Class} \rightarrow \text{Fun}$

$\text{do}'_0[e]\rho\kappa = \lambda \alpha. \perp$

if $n > 0$ *then*

$\text{do}'_n[\text{self}]\rho\kappa = \lambda \alpha. \text{send}'_{n-1}\rho$

$\text{do}'_n[\text{super}]\rho\kappa = \lambda \alpha. \text{lookup}'_n(\text{parent}(\kappa))\rho$

$\text{do}'_n[\text{arg}]\rho\kappa = \lambda \alpha. \alpha$

$\text{do}'_n[e_1 \ m \ e_2]\rho\kappa =$

$\lambda \alpha. (\text{do}'_n[e_1]\rho\kappa\alpha)m(\text{do}'_n[e_2]\rho\kappa\alpha)$

$\text{do}'_n[f(e_1, \dots, e_q)]\rho\kappa =$

$\lambda \alpha. f(\text{do}'_n[e_1]\rho\kappa\alpha, \dots, \text{do}'_n[e_q]\rho\kappa\alpha)$

Lemma 1

– OOPSLA '89

PROOF of Lemma 1: Recall that we want to prove, for $n > 0$, that

$$do'_n[e]\rho\kappa = eval[e](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)$$

by induction in the structure of e . The base case is proved as follows.

$$do'_n[\text{self}]\rho\kappa\alpha$$

$$= send'_{n-1}\rho$$

$$= eval[\text{self}](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)\alpha$$

$$do'_n[\text{super}]\rho\kappa\alpha$$

$$= lookup'_n(parent(\kappa))\rho$$

$$= eval[\text{super}](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)\alpha$$

$$do'_n[\text{arg}]\rho\kappa\alpha$$

$$= \alpha$$

$$= eval[\text{arg}](send'_{n-1}\rho)(lookup'_n(parent(\kappa))\rho)\alpha$$

The induction step is proven below using the abbreviation $\pi = lookup'_n(parent(\kappa))\rho$.

$$do'_n[e_1 \ m \ e_2]\rho\kappa\alpha$$

$$= do'_n[e_1]\rho\kappa\alpha m(do'_n[e_2]\rho\kappa\alpha)$$

$$= eval[e_1](send'_{n-1}\rho)\pi\alpha m(eval[e_2](send'_{n-1}\rho)\pi\alpha)$$

$$= eval[e_1 \ m \ e_2](send'_{n-1}\rho)\pi\alpha$$

$$do'_n[f(e_1, \dots, e_q)]\rho\kappa\alpha$$

$$= f(do'_n[e_1]\rho\kappa\alpha, \dots, do'_n[e_q]\rho\kappa\alpha)$$

$$= f(eval[e_1](send'_{n-1}\rho)\pi\alpha, \dots,$$

$$eval[e_q](send'_{n-1}\rho)\pi\alpha)$$

$$= eval[f(e_1, \dots, e_q)](send'_{n-1}\rho)\pi\alpha$$

QED

Lemma 1

– OOPSLA '89 reformulation in Agda: statement

PROOF of Lemma 1: Recall that we want to prove, for $n > 0$, that

$$\text{do}'_n[e]\rho\kappa = \text{eval}[e](\text{send}'_{n-1}\rho)(\text{lookup}'_n(\text{parent}(\kappa))\rho)$$

```
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)
```

Lemma 1

– OOPSLA '89 reformulation in Agda: a base case

by induction in the structure of e . The base case is proved as follows.

$$\begin{aligned} do'_n[\text{self}] \rho \kappa \alpha \\ = & send'_{n-1} \rho \\ = & eval[\text{self}](send'_{n-1} \rho)(lookup'_n(\text{parent}(\kappa)) \rho) \alpha \end{aligned}$$

lemma-1 n self ρ c κ =

```
begin  do' (suc n) [[ self ]]  $\rho$  (child c  $\kappa$ )
≡⟨⟩  ( from  $\lambda \alpha \rightarrow \text{from}(\text{inl}(\text{send}' n \rho))$  )
≡⟨⟩  eval[[ self ]](send' n  $\rho$ )(lookup' (suc n)  $\kappa$   $\rho$ )
```

□

Lemma 1

– OOPSLA '89 reformulation in Agda: remaining base cases

```
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)  
□
```

Lemma 1

– OOPSLA '89 reformulation in Agda: induction step

```
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
```

```
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))
```

Towards a Conclusion ...

A 60th Birthday Present for Jens

C. STRACHEY
1966

P.D. MOSSES
1975

IFIP Working Conference on
Formal Language Description Languages

Vienna, Austria, September 15-18, 1964

D. R. H. ZEMANEK (Austria) Conference Chairman

Organized by
IFIP Technical Committee 2, Programming Languages
INTERNATIONAL FEDERATION FOR
INFORMATION PROCESSING

FORMAL LANGUAGE DESCRIPTION LANGUAGES FOR COMPUTER PROGRAMMING

Proceedings of the
IFIP Working Conference on
Formal Language Description Languages

Edited by
T. B. STEEL, Jr.

Conclusion

– of this progress report ...

Semantics of inheritance

- definitions checked, revealing a couple of missing cases

Towards verification of correctness

- proofs of lemmas checked

Denotational semantics in Agda – issues 😬

- no implicit continuity of λ -expressions
- no implicit injections and isomorphisms
- restrictions on recursive type definitions