# 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
- 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.**
- 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

- 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**

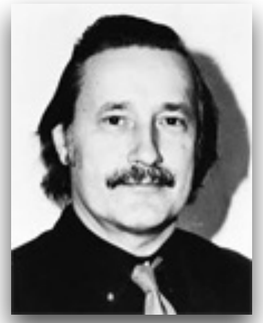
- operational (method-lookup)
- denotational
- in Agda

**Towards verification of correctness**

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

# Denotational semantics and fixed points

## – origins



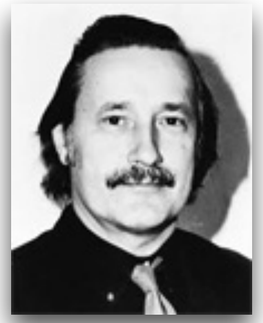
### Christopher Strachey: *Towards a Formal Semantics* (1966)

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.

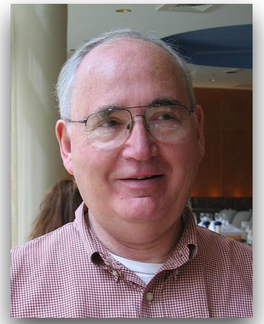
# Denotational semantics and fixed points

## – origins



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

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



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

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

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

- semantics $[\![\ \_\ ]\!]$ : programs → ***elements of domains*** (*lattices*, *cpos*)

# Semantics of Inheritance

# Method systems and semantic domains
## – OOPSLA '89

Method System Domains

| | | | |
|---|---|---|---|
| Instances | $\rho$ | $\in$ | **Instance** |
| Classes | $\kappa$ | $\in$ | **Class** |
| Messages | $m$ | $\in$ | **Key** |
| Primitives | $f$ | $\in$ | **Primitive** |
| Methods | $e$ | $\in$ | **Exp**:= **self** \| **super** \| **arg** |
| | | | $\mid e_1\ m\ e_2 \mid f(e_1, \ldots, e_q)$ |

Semantic Domains

| | | | |
|---|---|---|---|
| | | **Number** | |
| $\alpha$ | $\in$ | **Value** | = **Behavior** + **Number** |
| $\sigma, \pi$ | $\in$ | **Behavior** | = **Key** $\rightarrow$ (**Fun** + ?) |
| $\phi$ | $\in$ | **Fun** | = **Value** $\rightarrow$ **Value** |

Method System Operations

| | | |
|---|---|---|
| *class* | : | **Instance** $\rightarrow$ **Class** |
| *parent* | : | **Class** $\rightarrow$ (**Class** + ?) |
| *methods* | : | **Class** $\rightarrow$ **Key** $\rightarrow$ (**Exp** + ?) |

$root$ : **Class** $\rightarrow$ **Boolean**

$root(\kappa) = [\lambda\, \kappa' \in \textbf{Class}\,.\, false,$
$\lambda\, v \in ?\,.\, true$
$](parent(\kappa))$

# Method lookup semantics

## – OOPSLA '89

$$send : \mathbf{Instance} \to \mathbf{Behavior}$$
$$send(\rho) = lookup(class(\rho))\rho$$

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

$$parent : \mathbf{Class} \to (\mathbf{Class} + ?)$$

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

*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

$$\begin{aligned} \textbf{Generator} &= \textbf{Behavior} \to \textbf{Behavior} \\ \textbf{Wrapper} &= \textbf{Behavior} \to \textbf{Generator} \end{aligned}$$

$$\begin{aligned} &\mathit{behave} : \textbf{Instance} \to \textbf{Behavior} \\ &\mathit{behave}(\rho) = \mathrm{fix}(\mathit{gen}(\mathit{class}(\rho))) \end{aligned}$$

$$\begin{aligned} &\mathit{gen} : \textbf{Class} \to \textbf{Generator} \\ &\mathit{gen}(\kappa) = \textbf{if } \mathit{root}(\kappa) \\ &\qquad \textbf{then } \lambda\sigma \in \textbf{Behavior} \,.\, \lambda m \in \textbf{Key} \,.\, \bot_? \\ &\qquad \textbf{else } \mathit{wrap}(\kappa) \,\boxed{\triangleright}\, \mathit{gen}(\mathit{parent}(\kappa)) \end{aligned}$$

$$\begin{aligned} &\mathit{eval} : \textbf{Exp} \to \textbf{Behavior} \to \textbf{Behavior} \to \textbf{Fun} \\ &\mathit{eval}[\![\,\textbf{self}\,]\!]\sigma\pi = \lambda\alpha \,.\, \sigma \\ &\mathit{eval}[\![\,\textbf{super}\,]\!]\sigma\pi = \lambda\alpha \,.\, \pi \\ &\mathit{eval}[\![\,\textbf{arg}\,]\!]\sigma\pi = \lambda\alpha \,.\, \alpha \\ &\mathit{eval}[\![\,e_1\ m\ e_2\,]\!]\sigma\pi = \\ &\qquad \lambda\alpha \,.\, \mathit{eval}[\![\,e_1\,]\!]\sigma\pi\alpha\, m(\mathit{eval}[\![\,e_2\,]\!]\sigma\pi\alpha) \\ &\mathit{eval}[\![\,f(e_1, \ldots, e_q)\,]\!]\sigma\pi = \\ &\qquad \lambda\alpha \,.\, f(\mathit{eval}[\![\,e_1\,]\!]\sigma\pi\alpha, \ldots, \mathit{eval}[\![\,e_q\,]\!]\sigma\pi\alpha) \end{aligned}$$

$$\begin{aligned} &\mathit{wrap} : \textbf{Class} \to \textbf{Wrapper} \\ &\mathit{wrap}(\kappa) = \lambda\sigma \,.\, \lambda\pi \,.\, \lambda m \in \textbf{Key} \,. \\ &\qquad [\lambda e \in \textbf{Exp} \,.\, \mathit{eval}[\![\,e\,]\!]\sigma\pi \\ &\qquad\ \lambda v \in ? \,.\, \bot_? \\ &\qquad ]\mathit{methods}(\kappa)m \end{aligned}$$

$$W \,\boxed{\triangleright}\, P = \lambda\,\mathsf{self} \,.\, (W(\mathsf{self})(P(\mathsf{self}))) \oplus P(\mathsf{self})$$

# Denotational semantics
## – reformulation in Agda

**Agda**

- a pure functional ***programming language***
- with ***dependent types***
  - e.g., `_[_] : {A : Set} → {n : ℕ} → Vec A n → Fin n → 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 ⟩    ↔ ⟨ Behavior +⊥ Number ⟩ }}
  {{ iso^b  : ⟨ Behavior ⟩ ↔ ( Key → ⟨ Fun +⊥ ?⊥ ⟩ ) }}
  {{ iso^f  : ⟨ Fun ⟩      ↔ ( ⟨ 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

$$send' : \mathbf{Nat} \to \mathbf{Instance} \to \mathbf{Behavior}$$
$$send'_0(\rho) = \bot$$
$$send'_n(\rho) = lookup'_n(class(\rho))\rho \qquad n > 0$$

$$lookup' : \mathbf{Nat} \to \mathbf{Class} \to \mathbf{Instance} \to \mathbf{Behavior}$$
$$lookup'_0 \kappa\rho = \bot$$
$$\textit{if } n > 0 \textit{ then}$$
$$lookup'_n \kappa\rho = \lambda\, m \in \mathbf{Key}\,.$$
$$[\lambda\, e \in \mathbf{Exp}\,.\, do'_n[\![\, e \,]\!]\rho\kappa,$$
$$\lambda\, v \in ?\,.\, \mathbf{if}\ root(\kappa)$$
$$\mathbf{then}\ \bot_?$$
$$\mathbf{else}\ lookup'_n(parent(\kappa))\rho m$$
$$](methods(\kappa)m)$$

$$do' : \mathbf{Nat} \to \mathbf{Exp} \to \mathbf{Instance} \to \mathbf{Class} \to \mathbf{Fun}$$
$$do'_0[\![\, e \,]\!]\rho\kappa = \lambda\, \alpha\,.\, \bot$$
$$\textit{if } n > 0 \textit{ then}$$
$$do'_n[\![\, \mathbf{self} \,]\!]\rho\kappa = \lambda\, \alpha\,.\, send'_{n-1}\rho$$
$$do'_n[\![\, \mathbf{super} \,]\!]\rho\kappa = \lambda\, \alpha\,.\, lookup'_n(parent(\kappa))\rho$$
$$do'_n[\![\, \mathbf{arg} \,]\!]\rho\kappa = \lambda\, \alpha\,.\, \alpha$$
$$do'_n[\![\, e_1\ m\ e_2 \,]\!]\rho\kappa =$$
$$\lambda\, \alpha\,.\, (do'_n[\![\, e_1 \,]\!]\rho\kappa\alpha)m(do'_n[\![\, e_2 \,]\!]\rho\kappa\alpha)$$
$$do'_n[\![\, f(e_1, \ldots, e_q) \,]\!]\rho\kappa =$$
$$\lambda\, \alpha\,.\, f(do'_n[\![\, e_1 \,]\!]\rho\kappa\alpha, \ldots, do'_n[\![\, e_q \,]\!]\rho\kappa\alpha)$$

# 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.

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

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

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

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

$$\begin{aligned}
&do'_n[\![ e_1\ m\ e_2 ]\!]\rho\kappa\alpha \\
&\quad = do'_n[\![ e_1 ]\!]\rho\kappa\alpha m(do'_n[\![ e_2 ]\!]\rho\kappa\alpha) \\
&\quad = eval[\![ e_1 ]\!](send'_{n-1}\rho)\pi\alpha m(eval[\![ e_2 ]\!](send'_{n-1}\rho)\pi\alpha) \\
&\quad = eval[\![ e_1\ m\ e_2 ]\!](send'_{n-1}\rho)\pi\alpha
\end{aligned}$$

$$\begin{aligned}
&do'_n[\![ f(e_1, \ldots, e_q) ]\!]\rho\kappa\alpha \\
&\quad = f(do'_n[\![ e_1 ]\!]\rho\kappa\alpha, \ldots, do'_n[\![ e_q ]\!]\rho\kappa\alpha) \\
&\quad = f(eval[\![ e_1 ]\!](send'_{n-1}\rho)\pi\alpha, \ldots, \\
&\qquad\qquad eval[\![ e_q ]\!](send'_{n-1}\rho)\pi\alpha) \\
&\quad = eval[\![ f(e_1, \ldots, e_q) ]\!](send'_{n-1}\rho)\pi\alpha
\end{aligned}$$

QED

# Lemma 1

## – OOPSLA '89 reformulation in Agda: statement

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

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

```
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.

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

```
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

## – 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

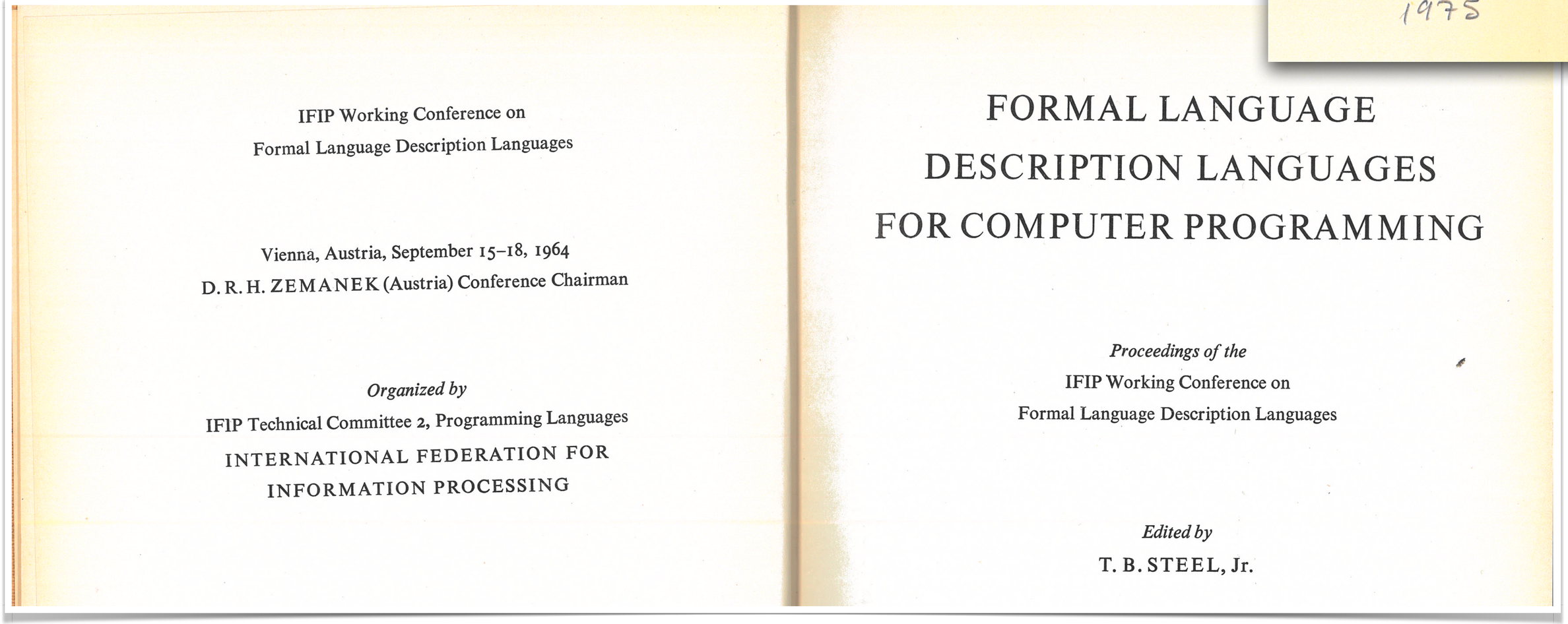
## – OOPSLA '89 reformulation in Agda: induction step

```
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



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