An Accidental Simula User

Luca Cardelli

Microsoft Research

ECOOP 2007
Dahl-Nygaard Senior Prize
2007-08-02

http://LucaCardelli.name

Abstract

- It was a simple choice, really, on an IBM 370 in the 70's, between APL, Fortran, Lisp 1.5, PL/1, COBOL, and Simula'67. Nothing could come close to Simula's combination of strong typing, garbage collection, and proper string processing. Separate compilation (prefix classes) and coroutines were nice bonuses. And then there were these ... "objects" but, well, nothing is perfect. Hot topics in those days were the freshly invented denotational semantics (which Simula didn't have), formal type systems (which objects didn't have), and abstract data types (which seemed to have confusingly little to do with classes). Still, Simula was the obvious choice to get something done comfortably because, after all, it was an improved Algol. It even supported the functional programming feature of call-by-name. So, it became my first favorite language, for every reason other than it being object-oriented.
- The story I am going to tell is the very, very slow realization that Simula was the embodiment of a radically different philosophy of programming, and the gradual and difficult efforts to reconcile that philosophy with the formal methods that were being developed for procedural and functional programming. Along the way, domain theory helped rather unexpectedly, at least for a while. Type theory had to be recast for the task at hand. Landin's lambda-reductionism had to be partially abandoned. Always, there seemed to be a deep fundamental mismatch between objects and procedures, well described by Reynolds, that made any unification impossibly complicated. But in the end, both object-oriented and procedural programming have benefited from the clash of cultures. And the story is far from over yet, as witnessed by the still blooming area of program verification for both procedural and object-oriented languages.

Outline

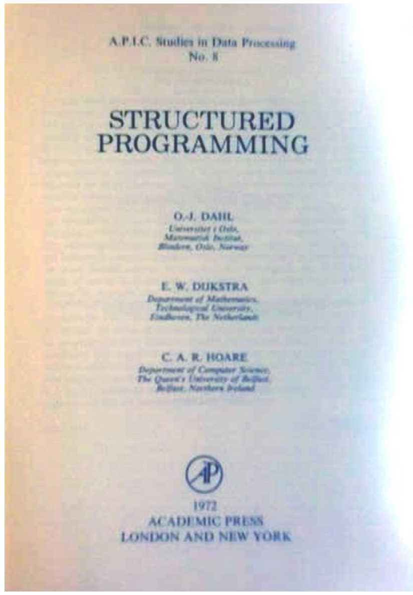
- I am under no pressure to present new results
  - (I understand.)
- Hence, a personal perspective:
  - How did I get here?
  - How did Ole-Johan Dahl and Kristen Nygaard influence my work?
- Only two things really *marked* me as an undergraduate:
  - λ-calculus
  - Simula 67

I: Functions

Things I Learned in Pisa

- Functional Programming
  - λ-calculus, Scheme, abstract machines, program transformations, denotation semantics, etc. etc.

- "Structured Programming" book [Dahl, Dijkstra, Hoare] (1972)
  - Dijkstra: "Structured Programming" (nested program structures)
  - Hoare: "Data structuring" (data types and type constructors)
  - Dahl & Hoare: "Hierarchical Program Structures" (Simula 67):
    - "A procedure which is capable of giving rise to block instances which survive its call will be known as a *class*, and the instances will be known as *objects* of that class" ... "Any variables or procedures declared local to the class body are called *attributes* of that class".
    - "Concatenation [subclassing] is an operation defined between two classes A and B, or a class A and a block C, and results in the formation of a new class or block. Concatenation consists in a merging of the attributes of both components, and the composition or their actions." Note: "this" is used in examples but never discussed.


A.P.I.C. Studies in Data Processing
No. 8

STRUCTURED PROGRAMMING

O.-J. DAHL
Universitet i Oslo,
Matematisk Institutt,
Blindern, Oslo, Norway

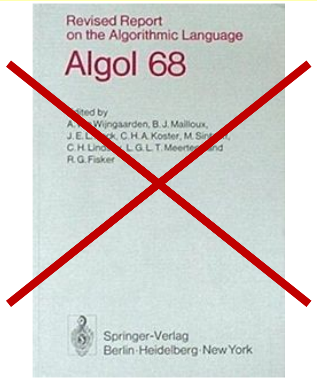
E. W. DIJKSTRA
Department of Mathematics,
Technological University,
Eindhoven, The Netherlands

C. A. R. HOARE
Department of Computer Science,
The Queen's University of Belfast,
Belfast, Northern Ireland

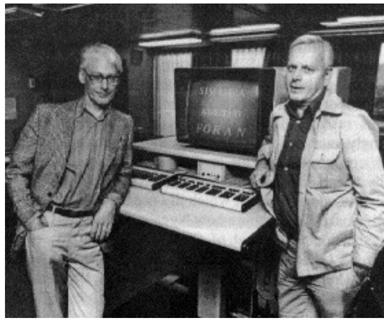
1972
ACADEMIC PRESS
LONDON AND NEW YORK

Things I Used in Pisa

- I wanted to use Algol'68!
  - State of the art (way overdesigned) algorithmic language.
  - But luckily, as it turned out, it was not available on IBM machines.


Revised Report on the Algorithmic Language
Algol 68
Springer-Verlag Berlin·Heidelberg·New York

- But Simula'67 was available (on IBM 370)
  - Other languages I tried: Fortran, APL, Lisp 1.5, PL/1. Yucc! gross!
  - So, I did most of my programming in Simula 67, mostly because of garbage collection and strong typing and strings!
  - But I did not buy the o-o propaganda; I just exploited it:
    - to obtain heap-allocated, garbage collected, data structures
    - for separate compilation (via a rather weird mode of class prefixing)
  - NO (reasonable) CHOICE BUT SIMULA


SIMULA Always First! Ole-Johan Dahl (left) and Kristen Nygaard, 1982. Courtesy of Rune Myhre, Dagbladet.

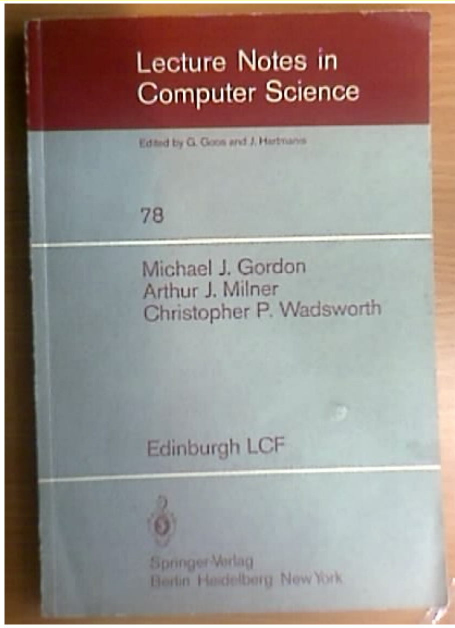
- Conversely, John Reynolds:
  - Wanted to use Simula but it was too expensive, and had to use Algol!

In the late sixties and the early seventies there were four main implementations of Simula:
UNIVAC 1100 by NCC
System/360 and System/370 by Swedish Research Institute for National Defence (FOA)
CDC 3000 by University of Oslo's Joint Computer Installation at Kjeller
TOPS-10 by ENEA AB
[Wikipedia]

The Goals of the Times

- To categorize all program constructions
  - And translate them to λ-calculus [Landin]
  - or give them denotational semantics [Scott-Strachey]
  - or something (which turned out later to be Operational Semantics [Plotkin]).
- My Master Thesis
  - On denotational semantics of programming languages
  - Tried to capture "all" program concepts in a single language
  - Mildly influenced by Simula:
    - Had a section on classes and subtyping by explicit injections.
    - But I realized there was a lot more to it.

Things I Learned in Edinburgh


Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

78

Michael J. Gordon
Arthur J. Milner
Christopher P. Wadsworth

Edinburgh LCF

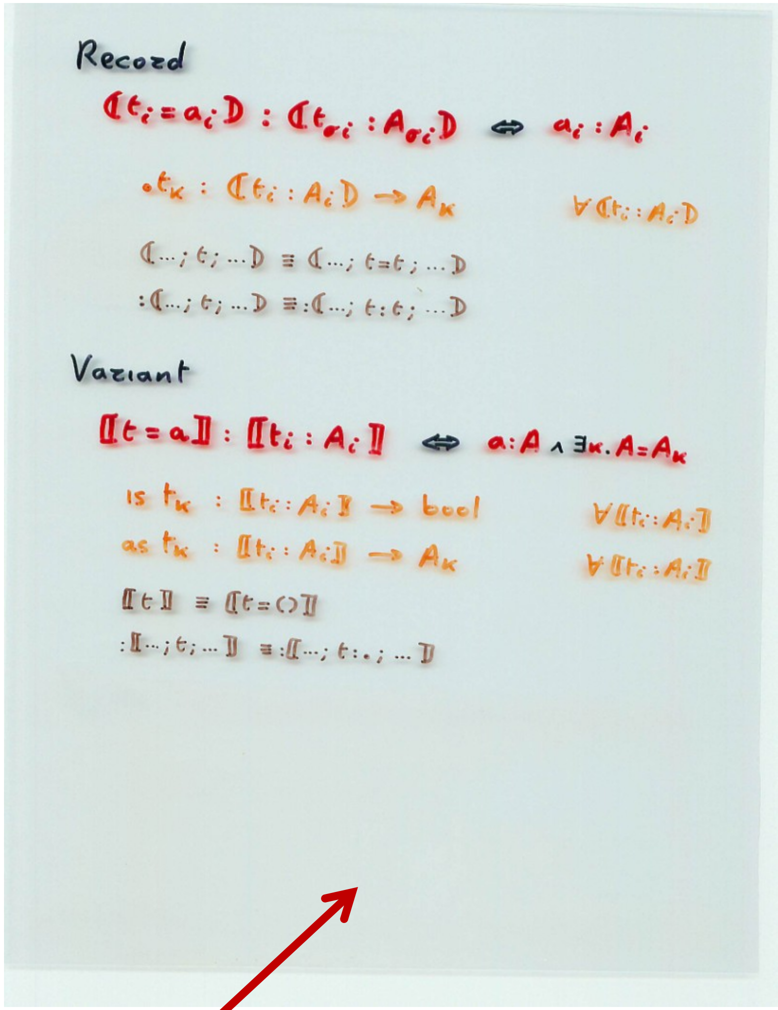
Springer-Verlag
Berlin Heidelberg New York

- Polymorphism
  - "A function that belongs to many types". Hmm... familiar??
- ML
  - Finally, a nice programming language!
  - No objects, and not even records (just binary products).
  - *Interactive* (in a way more "object-oriented" than batch Simula).
- Pascal
  - Finally, a nice *implementation* language! (To implement ML!)
  - No objects, but nice records and enumeration types.
  - So began a slippery slope away from functional languages.
- From Gordon Plotkin:
  - Contravariance from Category Theory
  - Product/sum duality (records/variants) from Category Theory
  - Operational semantics (on a slippery slope away from denotational)
- From Robin Milner (and David Park):
  - Calculus of Communicating Systems (how not to be a slave of λ-calculus)
  - Bisimulation (later, unlikely, useful for recursive types)

But Still Bugged by Simula

- ML extensions
  - Added records and variants to ML.
  - Tried to add record subtyping by type inference, but gave up. (This later became a little industry.)

- Galileo, with Albano and Orsini (and Ghelli as a student) back in Pisa.
  - A DataBase programming language (with records and variants)
  - With ML-inspired typing and aggregation/bulk constructs (map/filter).
  - With Simula-inspired subtyping (for OODB).
  - Ghelli later made important contributions to the theory of subtyping, providing the first example of undecidability of F<:.



Record

$(\!| t_i = a_i |\!) : (\!| t_{\sigma i} : A_{\sigma i} |\!) \Leftrightarrow a_i : A_i$

$.t_k : (\!| t_i : A_i |\!) \to A_k \qquad \forall (\!| t_i : A_i |\!)$

$(\!| \ldots ; t ; \ldots |\!) \equiv (\!| \ldots ; t = t ; \ldots |\!)$

$:(\!| \ldots ; t ; \ldots |\!) \equiv :(\!| \ldots ; t : t ; \ldots |\!)$

Variant

$[\![t = a]\!] : [\![t_i : A_i]\!] \Leftrightarrow a : A \wedge \exists k. A = A_k$

$\text{is } t_k : [\![t_i : A_i]\!] \to \text{bool} \qquad \forall [\![t_i : A_i]\!]$

$\text{as } t_k : [\![t_i : A_i]\!] \to A_k \qquad \forall [\![t_i : A_i]\!]$

$[\![t]\!] \equiv [\![t = ()]\!]$

$:[\![\ldots ; t ; \ldots]\!] \equiv :[\![\ldots ; t : \cdot ; \ldots]\!]$

Contemporary slides

A Gaping Gap in the Literature

- Where was the Theory of Object-Oriented Languages ?!?
  - Logic languages: --> Predicate logic
  - Database languages: --> Relational calculus
  - Functional languages: --> λ-calculus
  - Imperative languages: --> Hoare Logic / Weakest Preconditions
  - Modular languages: --> Algebraic semantics

  - Object-oriented languages: --> ???

- Instead, a clash of cultures:
  - "Everything is an object" (Smalltalk)
  - "Everything is a function" (λ-calculus)
  - The latter had plenty of mathematical justification.

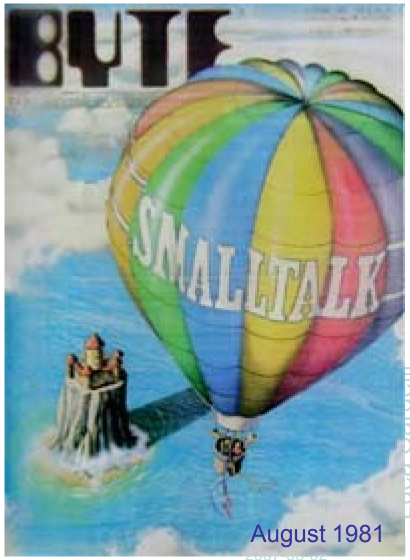
What Simula did not Have (or Need?)

- No denotational semantics
  - Not necessarily useful in itself
    but, e.g. useful to check that the type rules are correct.

- No formal type system
  - Simula was believed to be a safe language, like ML (and unlike Pascal), but (a subset of) ML had a formal type system and a proof of type safety via denotational semantics.

- No abstract data types, or modules
  - Are classes the same as abstract data types [Liskov]?
  - Are classes the same as modules [Parnas]?
  - The Norwegian Computing Center approach to software architecture was widely misunderstood.

- ... not to mention Smalltalk
  - Which did not even have a *syntax*, let alone a semantics!!
  - But had Super inheritance instead of Simula's Inner, indicating the need of a whole framework in which to study inheritance.



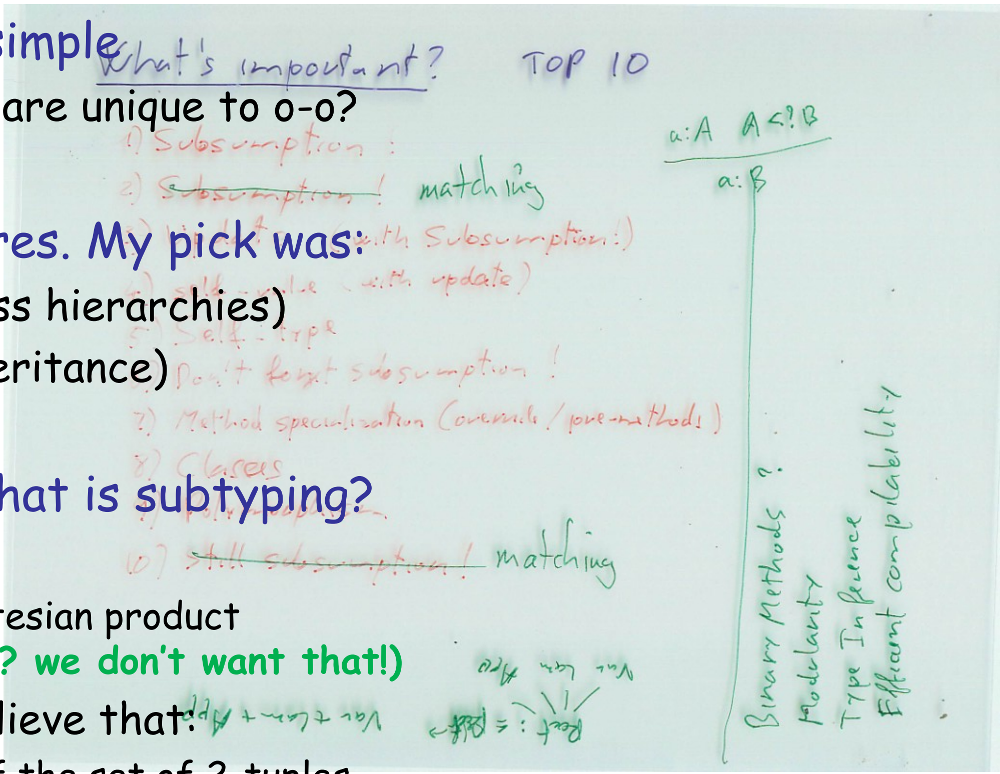
August 1981

But O-O too is "Higher-Order"

- Closer to functional programming than to imperative programming:
  - Need first-class function spaces (to model methods)
  - Need recursive types (to model self)
  - Need higher-order logics (to model class invariants)
- My Simula vs ML experience: O-o is a lot more "heavywheight"
  - Lots of work to set up simple inductive data types
    - A dummy virtual superclass with lots of subclasses that never inherit.
  - Lots of work to set up simple first-class functions
    - "Anonymous delegates" in C#.
- But is any of that essential or incidental?
  - Can we do functional programming in o-o?
  - Can we do o-o in functional programming?
- So many quasi-connections
  - Are polymorphism for objects and for functions related?
  - The Reynolds duality: extending code by adding functions or by adding objects.

What's "Important" about O-O?

- To build a theory you have to start simple
  - What are the simplest features that are unique to o-o?
- Simula had lots of interesting features. My pick was:
  - #1 subtyping (as a foundation for class hierarchies)
  - #2 this/self (as a foundation for inheritance)
- Hence, question #1 (at the time): What is subtyping?
  - Mathematicians, oddly believe that:
    - A function space is a subset of a cartesian product
      (Uh? functions subtypes of records? we don't want that!)
  - Conversely mathematicians, oddly, believe that:
    - The set of 3-tuples is not a subset of the set of 2-tuples.
      (Uh? but we want that!)
  - Dahl & Hoare clearly stated that
    - **"Any object of a subclass also belongs to the prefix class"**
  - So, what is subtyping?
    - <silence>
- In the mist of that confusion, I moved to Bell Labs...



A slide from a much later time
(apparently edited by Kim Bruce)

II: Records

Things I Learned at Bell Labs

Systems Programming in C

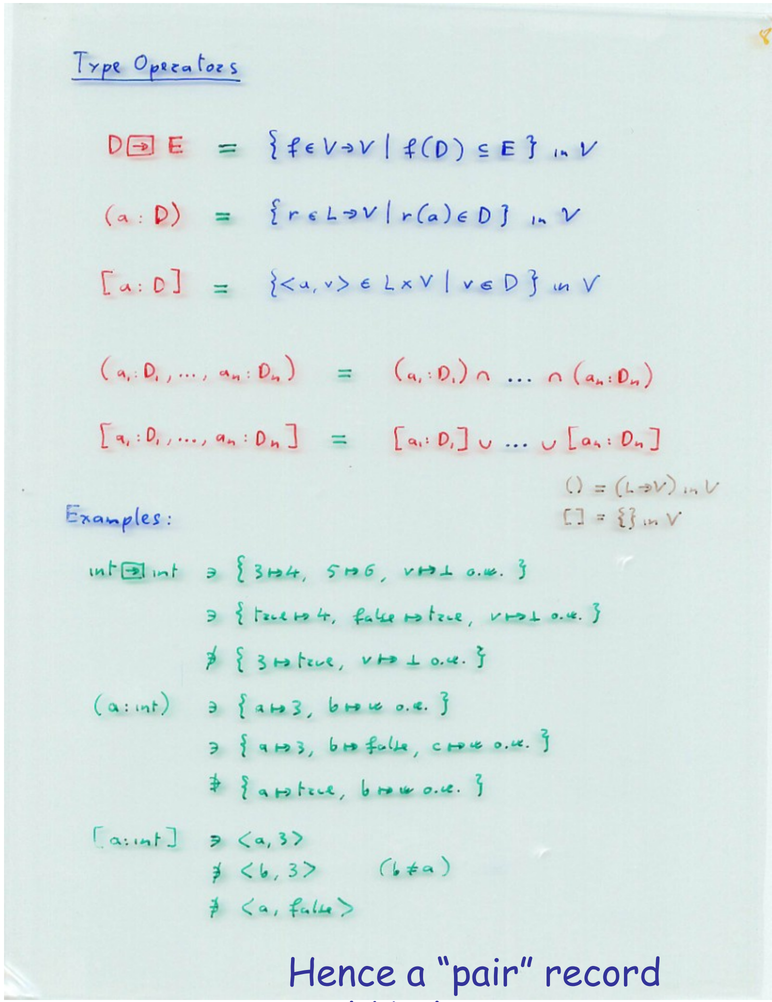
- Systems programming (Unix group)
  - ML not good "as such" for systems programming
    - Pointer arithmetic? What's that??
  - Using C (yucc! gross! but at least you can see the metal)
    - Had a pretty careful definition
  - But *never* C++ (mostly out of induced disgust with C)
    - N.B.: Bjorne just down the corridor.
    - The rumor was that he was trying to turn C into Simula.

- The MacQueen-Sethi-Plotkin Ideal Model
  - In early denotational models, types were "retracts" of the universal value set, which did not support subtyping.
  - The Ideal Model was designed as a semantics for polymorphism, which was modeled as a "big intersection" of domains.
  - So, it accidentally provided a subset-based denotational semantics of subtyping (via non-empty intersections between domains)
  - Therefore enabling:
    - records as functions (well-known lisp hack)
    - record types as domains (label-dependent function types)
    - record subtyping as set inclusion of function spaces

Semantics of Subtyping

- "A Semantics of Multiple Inheritance" paper
  - Subtyping for Record, Variant, Function types
  - It introduced "objects as recursive records" to model self/this.
  - The contravariance rule was introduced here.
  - The (later named) subsumption rule was a theorem in the system. (Ait-Kaci used that term in a different context.) *Any value of a subtype also belongs to the supertype.*

- Historical Footnotes
  - "Inheritance" here means "Subtyping": the famous paper "Inheritance is not subtyping" was written much later!
  - It was a "semantics", but it allowed me to justify the type rules, which were the real focus of the paper.
  - A question by Pavel Curtis about this paper led to bounded quantification.



Type Operators

$$D \boxed{\rightarrow} E = \{ f \in V \rightarrow V \mid f(D) \subseteq E \} \text{ in } V$$

$$(a : D) = \{ r \in L \rightarrow V \mid r(a) \in D \} \text{ in } V$$

$$[a : D] = \{ \langle a, v \rangle \in L \times V \mid v \in D \} \text{ in } V$$

$$(a_1 : D_1, \dots, a_n : D_n) = (a_1 : D_1) \cap \dots \cap (a_n : D_n)$$

$$[a_1 : D_1, \dots, a_n : D_n] = [a_1 : D_1] \cup \dots \cup [a_n : D_n]$$

$$() = (L \rightarrow V) \text{ in } V$$

$$[\,] = \{\} \text{ in } V$$

Examples:

$$\text{int} \boxed{\rightarrow} \text{int} \ni \{ 3 \mapsto 4,\ 5 \mapsto 6,\ v \mapsto \bot \text{ o.w.} \}$$

$$\ni \{ \text{true} \mapsto 4,\ \text{false} \mapsto \text{true},\ v \mapsto \bot \text{ o.w.} \}$$

$$\not\ni \{ 3 \mapsto \text{true},\ v \mapsto \bot \text{ o.w.} \}$$

$$(a : \text{int}) \ni \{ a \mapsto 3,\ b \mapsto w \text{ o.w.} \}$$

$$\ni \{ a \mapsto 3,\ b \mapsto \text{false},\ c \mapsto w \text{ o.w.} \}$$

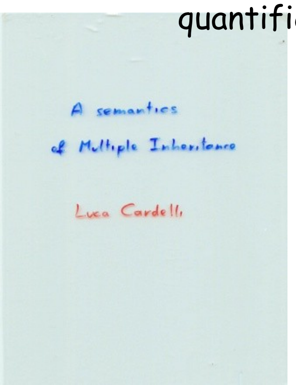
$$\not\ni \{ a \mapsto \text{true},\ b \mapsto w \text{ o.w.} \}$$

$$[a : \text{int}] \ni \langle a, 3 \rangle$$

$$\not\ni \langle b, 3 \rangle \quad (b \neq a)$$

$$\not\ni \langle a, \text{false} \rangle$$

Hence a "pair" record could belong to a "singleton" type.



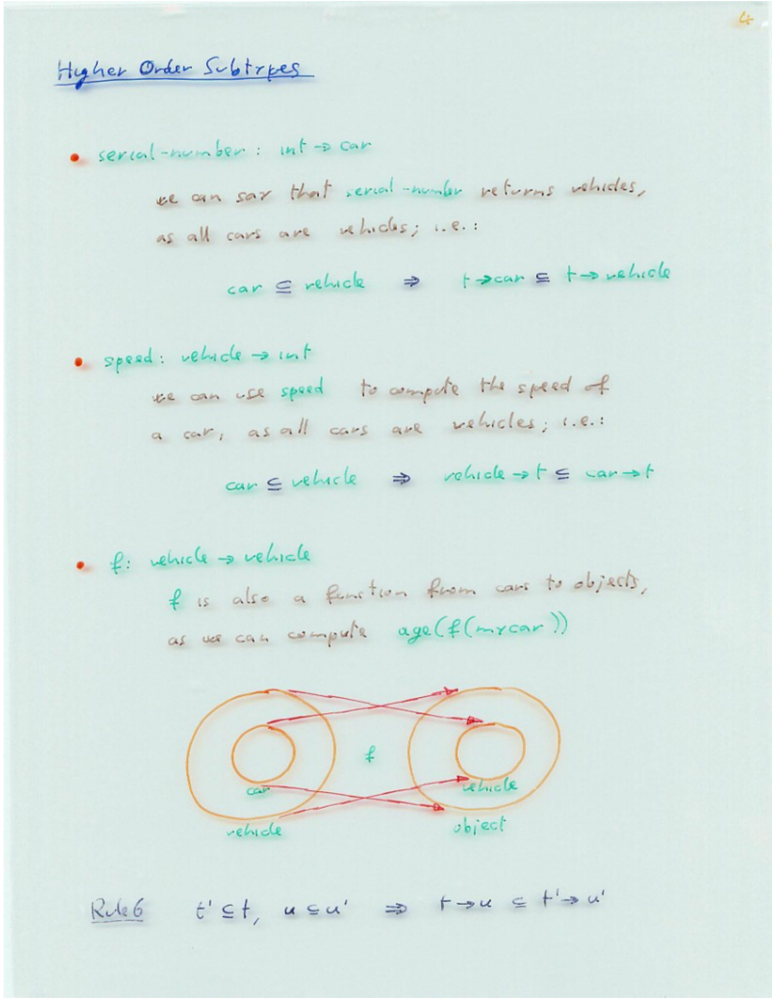
A semantics of Multiple Inheritance

Luca Cardelli

Contravariance Becomes (Un)Popular

- At my first OOPSLA
  - I was looking at an exhibition booth about a new French o-o programming language. A rather animated guy looked at my badge and said something like:
    "Luca Cardelli: Ah! Contravariance! I read your paper. I know I have a problem, but I promise you, I'll fix it, I'll fix it"
  - I had no idea what he was talking about, but I wrote his name on a piece of paper: Bertrand Meyer.
  - I am still waiting to hear back from him.

- Contravariance is a *fact* about functions
  - But not necessarily about objects/methods
  - Some languages were unsound because they failed to adopt it
  - But there are many different ways to deploy it in a type system



4

Higher Order Subtypes

- serial-number : int → car

  we can say that serial-number returns vehicles, as all cars are vehicles; i.e.:

  $\text{car} \leq \text{vehicle} \Rightarrow t \to \text{car} \leq t \to \text{vehicle}$

- speed: vehicle → int

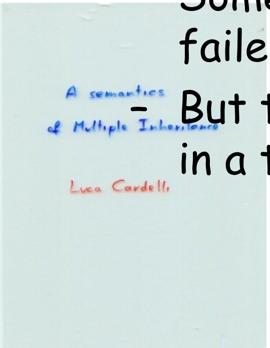
  we can use speed to compute the speed of a car, as all cars are vehicles; i.e.:

  $\text{car} \leq \text{vehicle} \Rightarrow \text{vehicle} \to t \leq \text{car} \to t$

- f: vehicle → vehicle

  f is also a function from cars to objects, as we can compute age(f(mycar))

Rule 6 $t' \leq t,\ u \leq u' \Rightarrow t \to u \leq t' \to u'$



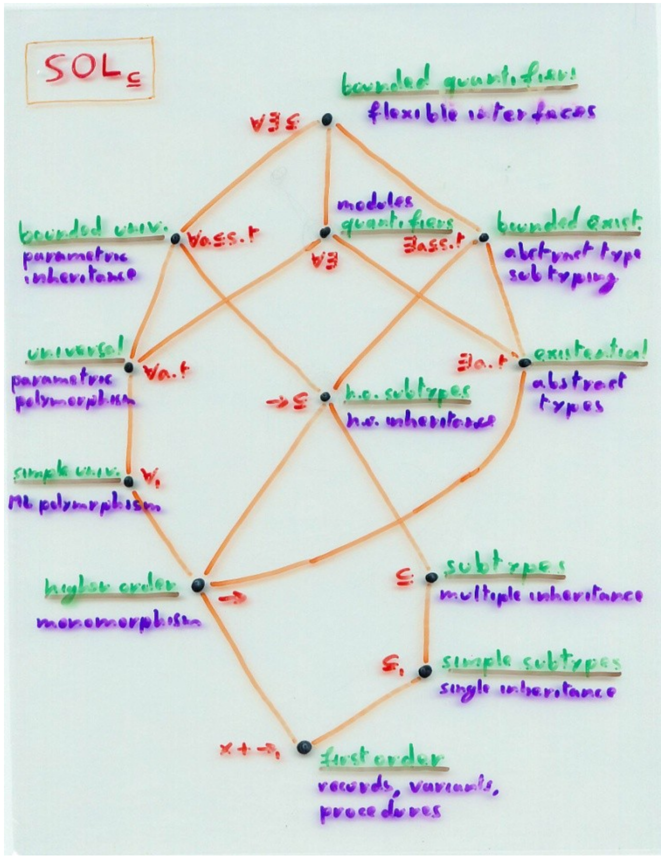
A semantics of Multiple Inheritance

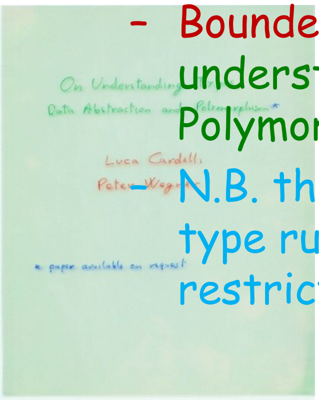
Luca Cardelli

Extending Subtyping to Quantifiers (SOL$_\subseteq$)

- After that, I had a "mission"
  - To boldly go and investigate subtyping for *all* type constructors.
  - E.g. to mix ML (polymorphism) with Simula (subtyping).
  - Next in line after records, variants, and functions were...

- Type quantifiers
  - Universal quantifiers captured polymorphism [Reynolds]
  - Existential quantifiers just shown to capture data abstraction [Mitchell-Plotkin]
  - Both together give you modules [MacQueen]
  - Bounded Quantification was born ("On understanding Types, Data Abstraction, and Polymorphism" with Peter Wegner).
  - N.B. this was a "survey" article. Appendix with type rules added at the last minute, with a restricted bounded quantification rule.

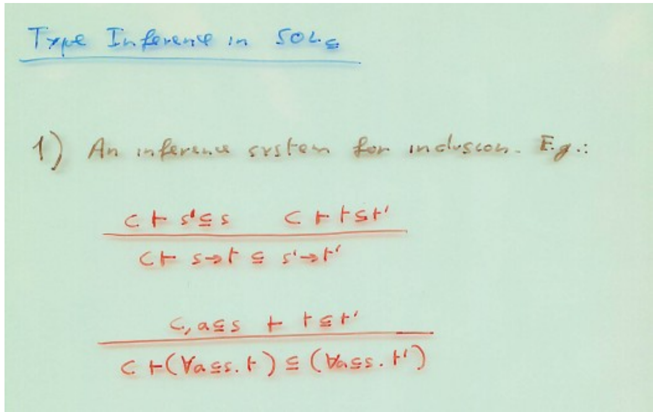




On Understanding Types, Data Abstraction and Polymorphism*

Luca Cardelli
Peter Wegner

* paper available on request



Type Inference in SOL$_\leq$

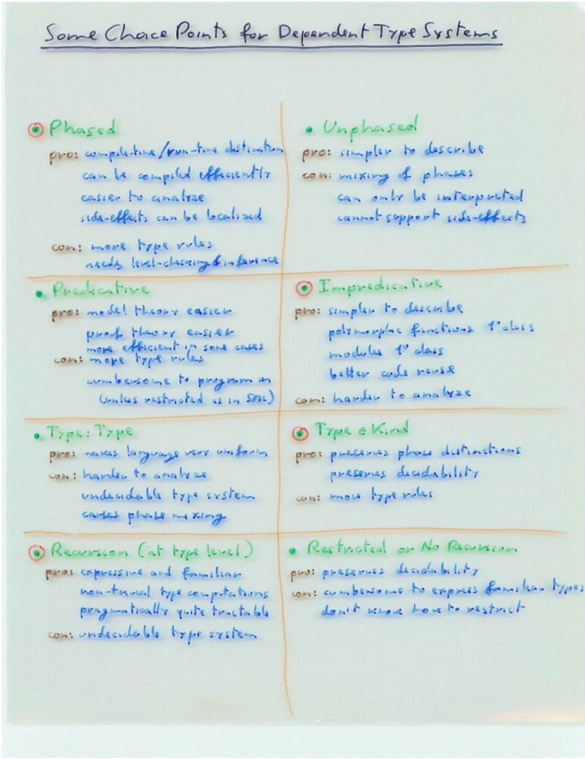
1) An inference system for inclusion. E.g.:

$$\frac{C \vdash s' \leq s \qquad C \vdash t \leq t'}{C \vdash s \to t \leq s' \to t'}$$

$$\frac{C, a \leq s \;\vdash\; t \leq t'}{C \vdash (\forall a \leq s.\, t) \leq (\forall a \leq s.\, t')}$$

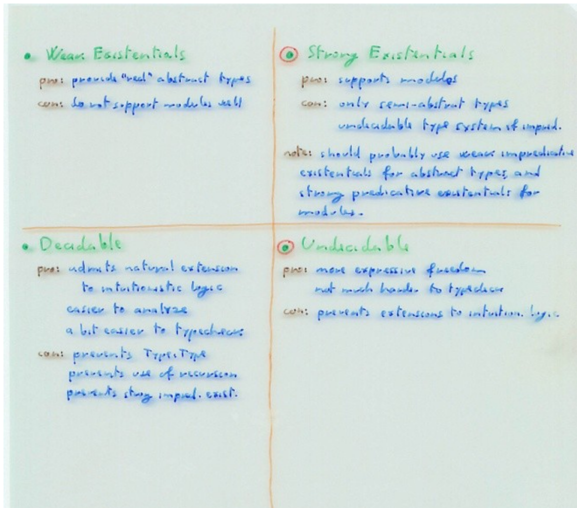
Things I Learned at DEC

- Distributed O-O Programming
  - Network Objects
- Seriously studying Type Theory
  - Denotational semantics was a start
  - Type theory was the real theory of types
    - Martin-Loef (dependent types)
    - Girard (second-order λ-calculus)
    - Reynolds (polymorphism and data abstraction)
  - Proper contravariant rule for quantifiers
- Around this time
  - I invented the (ASCII!!) subtyping symbol "<:"
  - Named the "subsumption" rule.

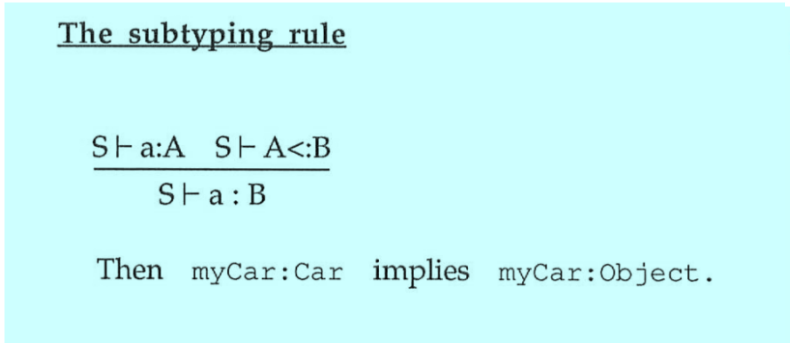


Some Choice Points for Dependent Type Systems

[✓ Phased] pro: compile-time/run-time distinction can be compiled efficiently easier to analyze side-effects can be localized con: more type rules needs level-checking & inference	• Unphased pro: simpler to describe con: mixing of phases can only be interpreted cannot support side-effects
• Predicative pro: model theory easier proof theory easier more efficient in some cases con: more type rules cumbersome to program in (unless restricted as in SML)	[✓ Impredicative] pro: simpler to describe polymorphic functions 1° class modules 1° class better code reuse con: harder to analyze
• Type : Type pro: makes language very uniform con: harder to analyze undecidable type system causes phase mixing	[✓ Type ∈ Kind] pro: preserves phase distinctions preserves decidability con: more type rules
[✓ Recursion (at type level)] pro: expressive and familiar non-trivial type computations pragmatically quite tractable con: undecidable type system	• Restricted or No Recursion pro: preserves decidability con: cumbersome to express familiar types don't know how to restrict



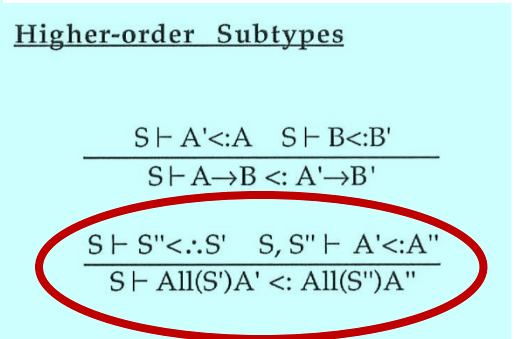
• Weak Existentials pro: provide "real" abstract types con: do not support modules well	[✓ Strong Existentials] pro: supports modules con: only semi-abstract types undecidable type system if impred. note: should probably use weak impredicative existentials for abstract types, and strong predicative existentials for modules.
• Decidable pro: admits natural extension to intuitionistic logic easier to analyze a bit easier to typecheck con: prevents Type:Type prevents use of recursion prevents strong impred. exist.	[✓ Undecidable] pro: more expressive freedom not much harder to typecheck con: prevents extensions to intuition. logic



The subtyping rule

$$\frac{S \vdash a:A \quad S \vdash A<:B}{S \vdash a:B}$$

Then `myCar:Car` implies `myCar:Object`.



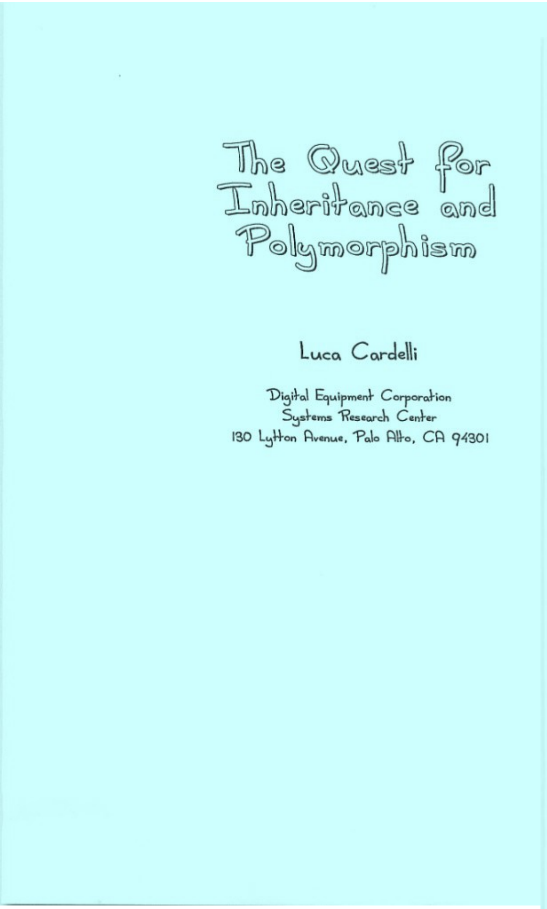
Higher-order Subtypes

$$\frac{S \vdash A'<:A \quad S \vdash B<:B'}{S \vdash A\to B <: A'\to B'}$$

$$\frac{S \vdash S''<:S' \quad S, S'' \vdash A'<:A''}{S \vdash All(S')A' <: All(S'')A''}$$

On A Long-Term Quest

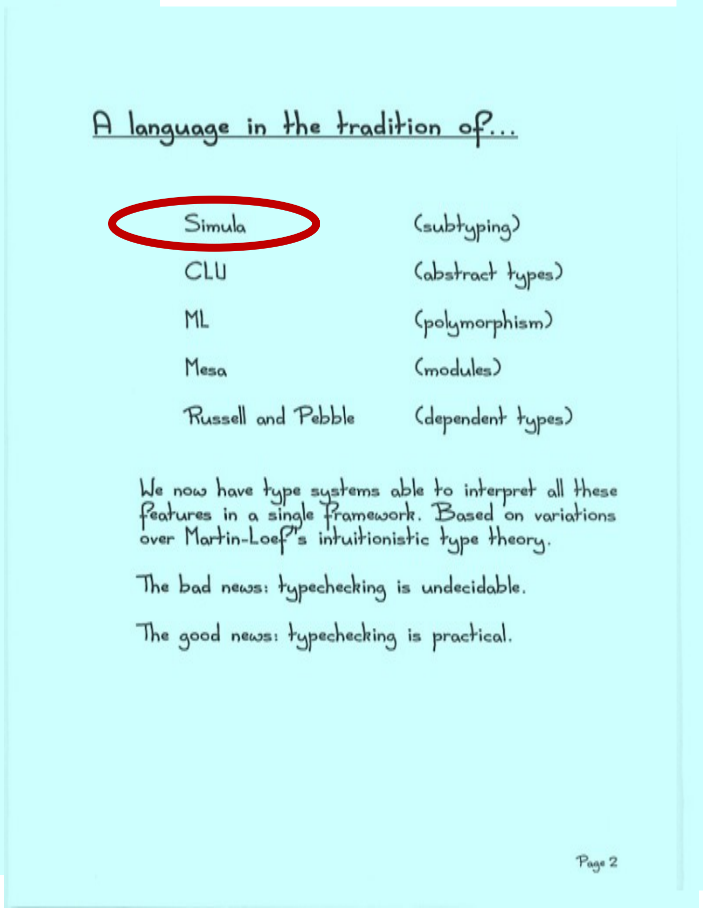
- Quest (Quantifiers & subtypes)
  - Setting out to design a language that would unify functional and object-oriented programming through subtyping.



The Quest For Inheritance and Polymorphism

Luca Cardelli

Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301



A language in the tradition of...

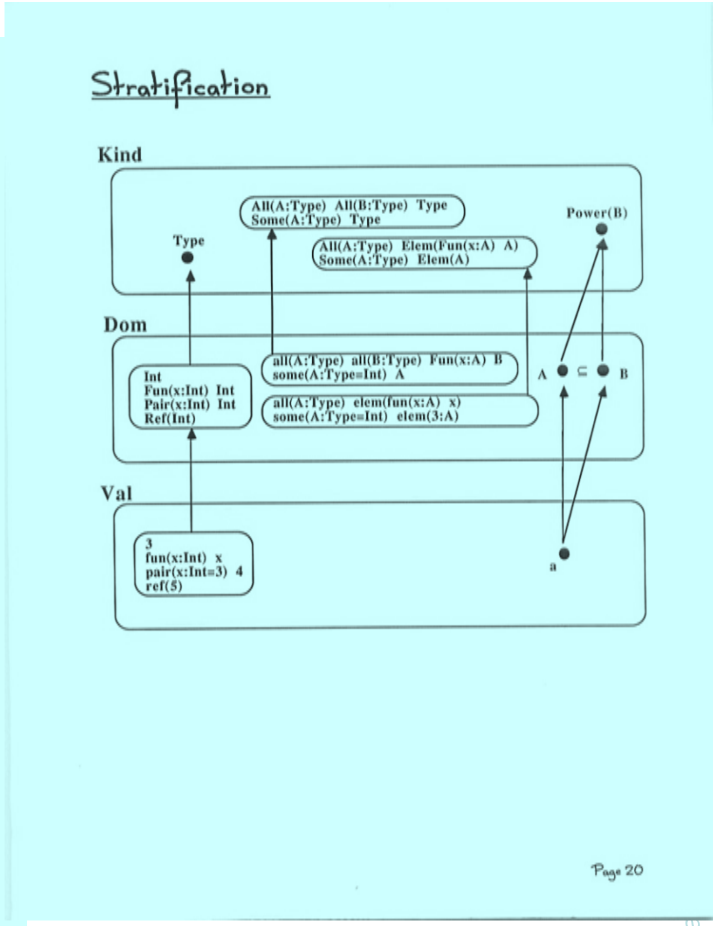
Simula	(subtyping)
CLU	(abstract types)
ML	(polymorphism)
Mesa	(modules)
Russell and Pebble	(dependent types)

We now have type systems able to interpret all these features in a single framework. Based on variations over Martin-Loef's intuitionistic type theory.

The bad news: typechecking is undecidable.

The good news: typechecking is practical.

Page 2

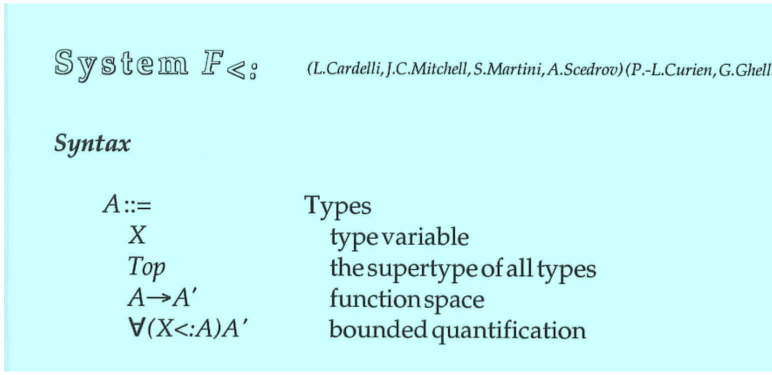


Stratification

Page 20

System F<: (Pure Bounded Quantification)

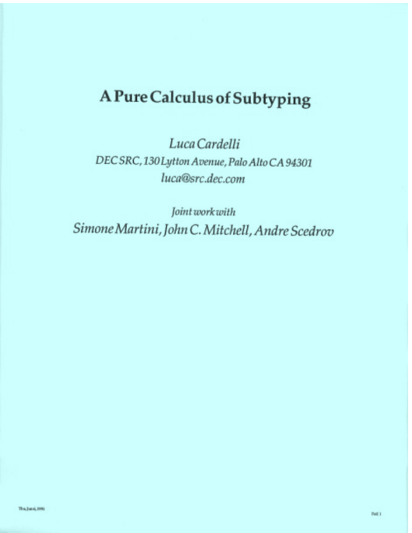
- While I was "moving to higher kinds"
  - POPL paper on Power Types
- Pierre-Louis Curien and Giorgio Ghelli were simplifying SOL⊆
  - Wanted a simpler system to study: F<:
  - No records: just functions.
  - They thought they were loosing power by cutting it down.



System $F_{<:}$ (L.Cardelli, J.C.Mitchell, S.Martini, A.Scedrov) (P.-L.Curien, G.Ghell

Syntax

$A ::=$	Types
X	type variable
Top	the supertype of all types
$A \to A'$	function space
$\forall(X<:A)A'$	bounded quantification



A Pure Calculus of Subtyping

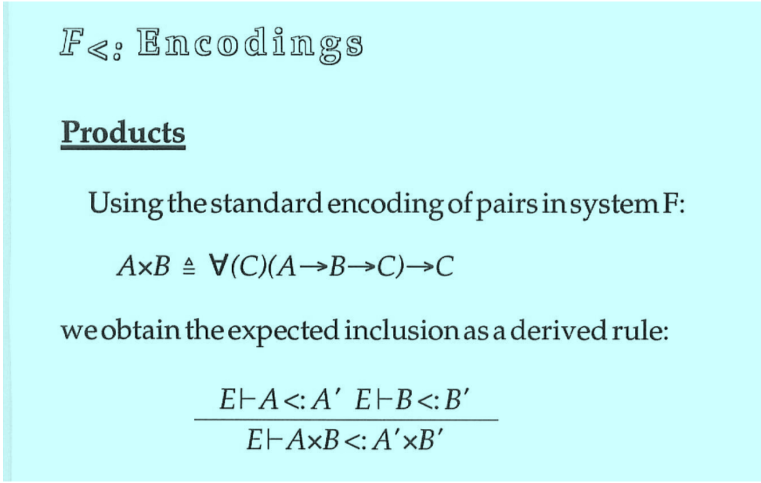
Luca Cardelli
DEC SRC, 130 Lytton Avenue, Palo Alto CA 94301
luca@src.dec.com

Joint work with
Simone Martini, John C. Mitchell, Andre Scedrov

Thu, Jun 6, 1991 Fol 1

System F<: was Enough

- But nothing was lost!
  - I showed how to encode record typing and subtyping in pure F<:.
  - That established F<: as a "core system" to study subtyping.
  - Much work followed on that basis [Cook, Mitchell et. al.: F-bounded quantification] [Pierce] Etc.



$F_{\leq}$: Encodings

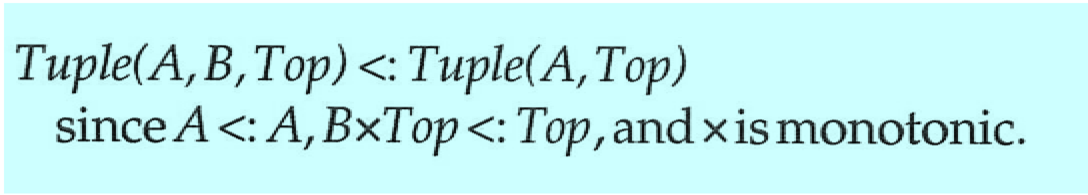
Products

Using the standard encoding of pairs in system F:

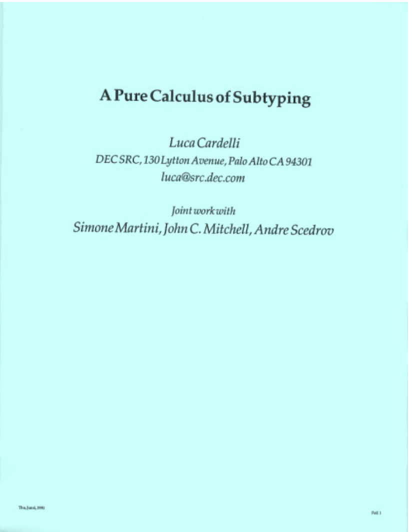
$$A{\times}B \triangleq \forall(C)(A{\to}B{\to}C){\to}C$$

we obtain the expected inclusion as a derived rule:

$$\frac{E\vdash A <: A' \quad E\vdash B <: B'}{E\vdash A{\times}B <: A'{\times}B'}$$



$Tuple(A,B,Top) <: Tuple(A,Top)$
since $A <: A$, $B{\times}Top <: Top$, and $\times$ is monotonic.



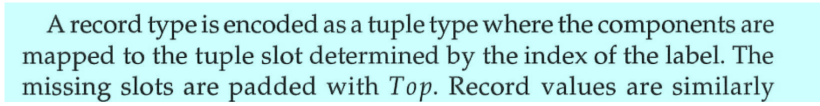
A Pure Calculus of Subtyping

Luca Cardelli
DEC SRC, 130 Lytton Avenue, Palo Alto CA 94301
luca@src.dec.com

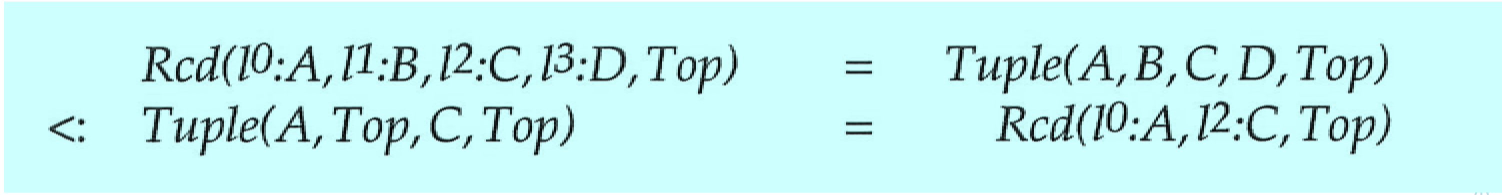
Joint work with
Simone Martini, John C. Mitchell, Andre Scedrov

Thu, Jun 6, 1991

Foil 1



A record type is encoded as a tuple type where the components are mapped to the tuple slot determined by the index of the label. The missing slots are padded with *Top*. Record values are similarly



$$\begin{array}{llcl} & Rcd(l^0{:}A, l^1{:}B, l^2{:}C, l^3{:}D, Top) & = & Tuple(A,B,C,D,Top) \\ <: & Tuple(A,Top,C,Top) & = & Rcd(l^0{:}A, l^2{:}C, Top) \end{array}$$

Finally: Subtyping Recursive Types

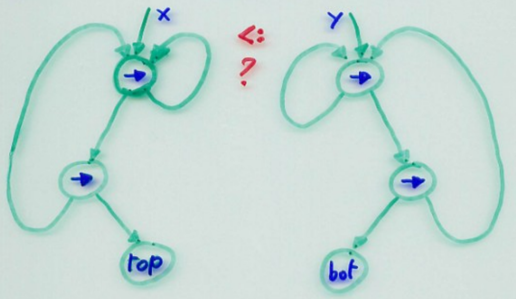
- Back to the "mission": Recursive Types
  - They should be necessary (?!?) to model Self.
  - Hence what are their subtyping rules?
- The Amber Rule
  - At Bell Labs, coding up the typechecker for Amber (a first-order language with subtyping) I arrived at the case for subtyping recursive types, and went: "uh-oh... now what?".
  - I made up a rule that seemed to work. It was inspired by a proof rule for bisimulation in CCS.

$$\frac{X <: Y \Rightarrow A <: B}{\mathrm{rec}\, X\, A <: \mathrm{rec}\, Y\, B} \quad + \quad \text{Type equivalence up to unfolding}$$

- Now, with Amadio, we finally proved its soundness and completeness
  - Later several people [Palsberg] studied its typechecking efficiency.
  - And still later it was set on proper coinductive grounds [Pierce], in the original bisimulation spirit.

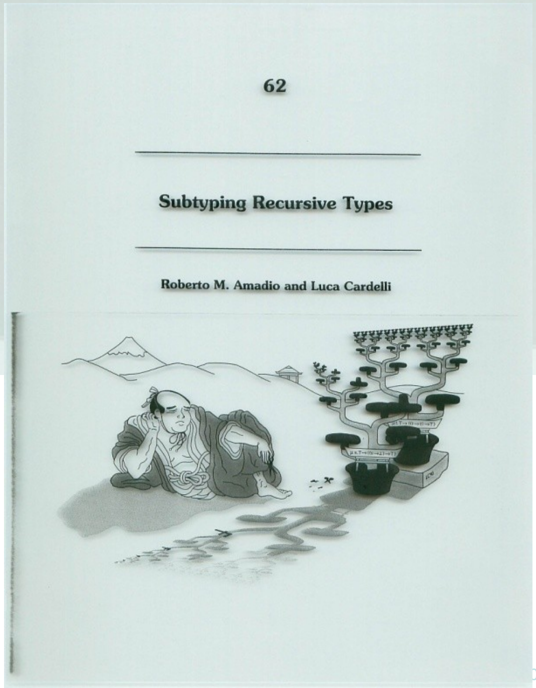
The problem

When are two recursively defined types in subtype relation?



(rec x ((x→top)→x))

(rec y (y→(bot→y)))


62

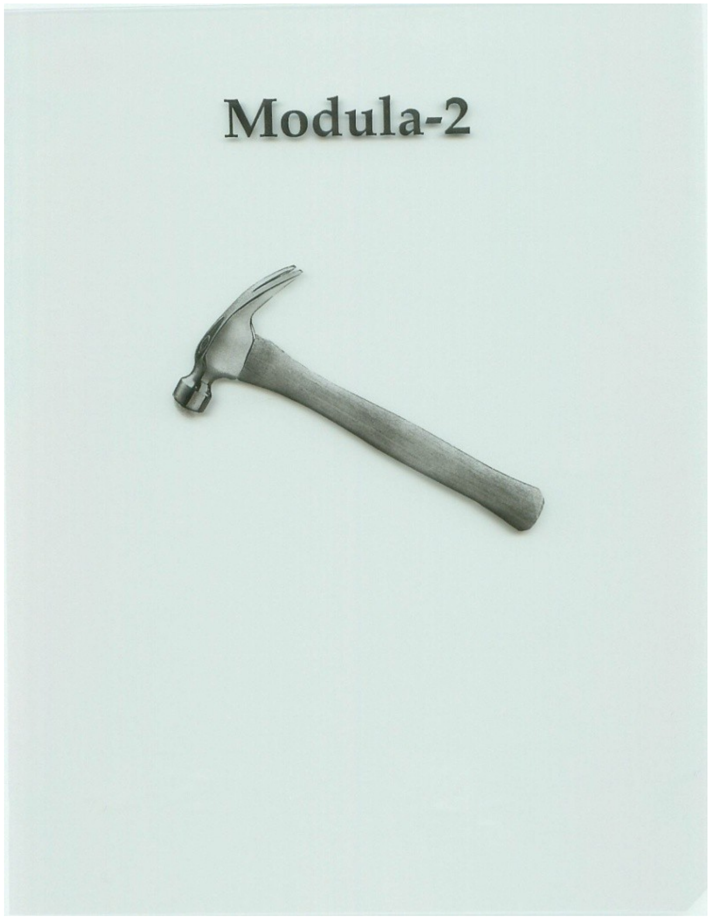
Subtyping Recursive Types

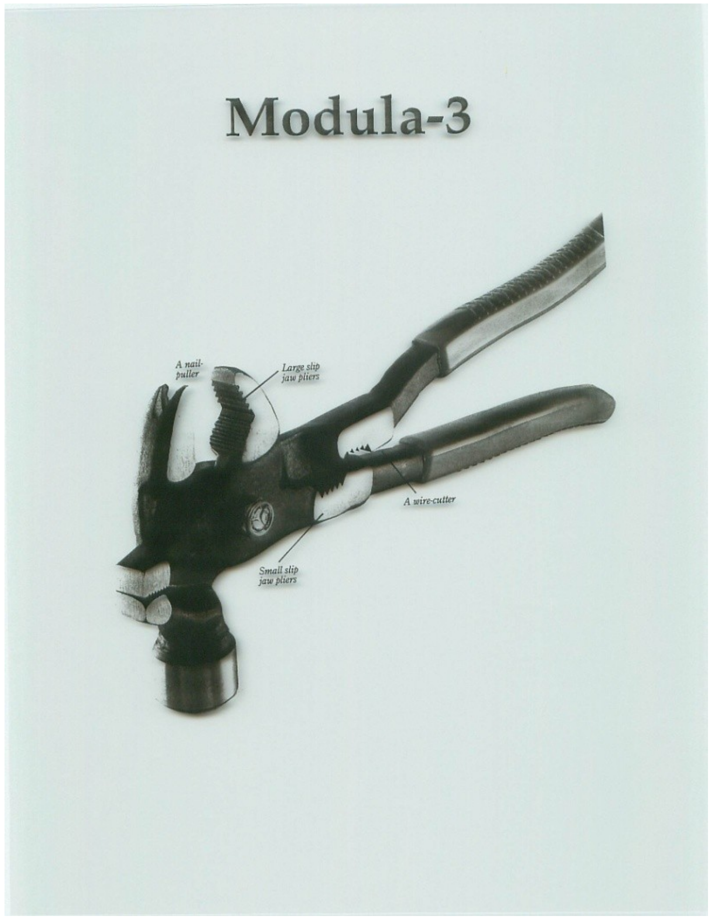
Roberto M. Amadio and Luca Cardelli

III: Real Languages

Yes, But What About O-O?

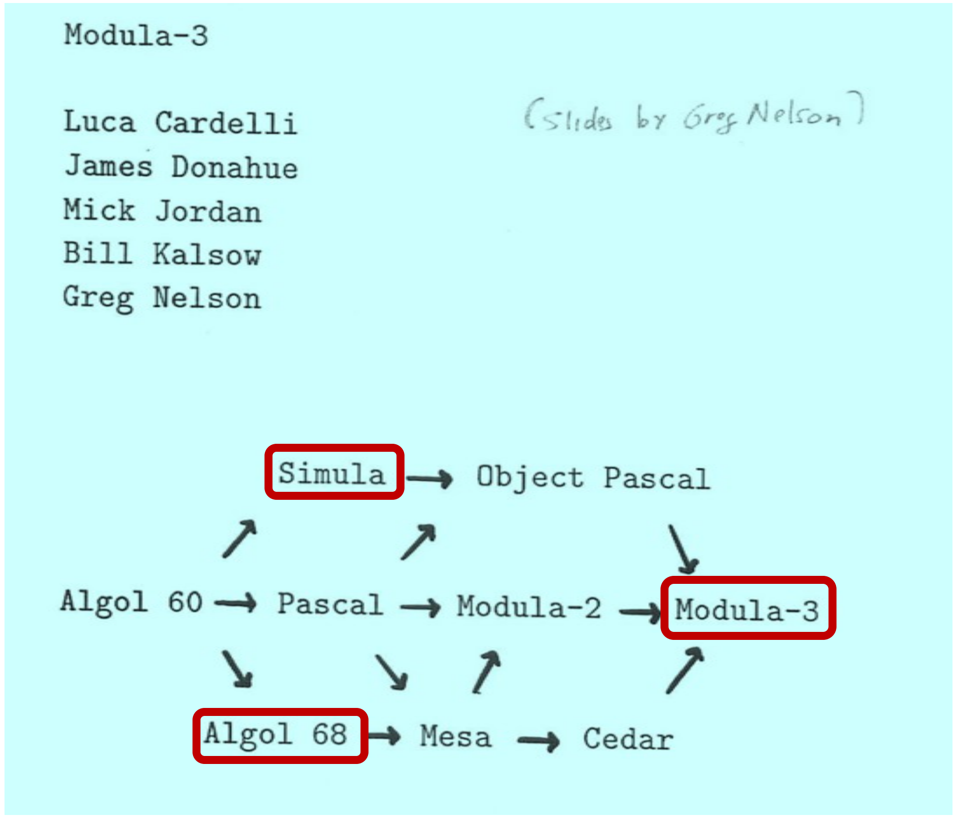
- At DEC I got involved in the design of Modula-3
  - Industrial-strength Modula-2 for systems (o-o) programming





Particularly, Modula-3 Types

- Driven by extensive discussions on the "value set" rule (subsumption) and when it was ok to use it.
- Innovating in "partial revelation" types (somewhat akin to bounded existentials).
- Reuniting Simula67 and Algol68 !

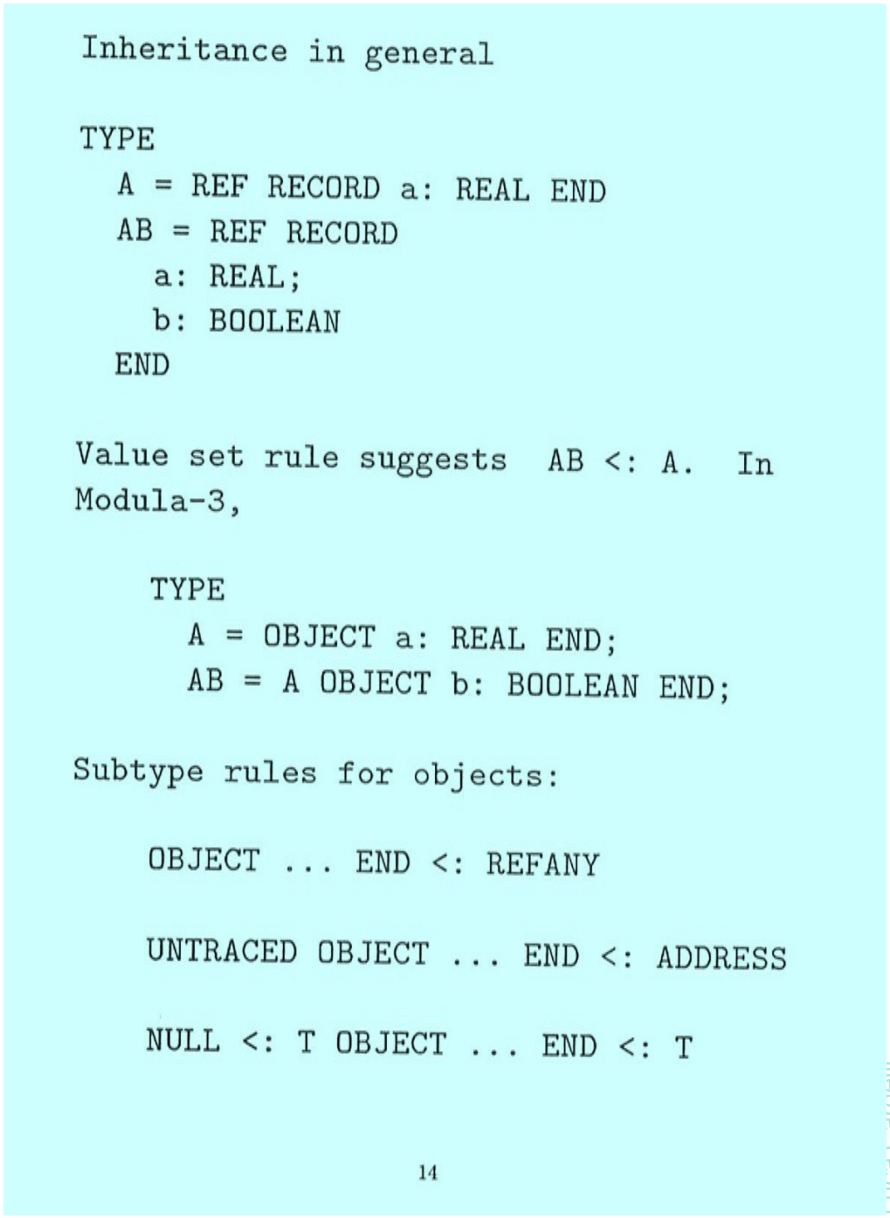


Modula-3

Luca Cardelli
James Donahue
Mick Jordan
Bill Kalsow
Greg Nelson

(slides by Greg Nelson)

$$\begin{tikzcd}
 & \boxed{\text{Simula}} \arrow[r] & \text{Object Pascal} \arrow[dr] & \\
\text{Algol 60} \arrow[ur] \arrow[r] \arrow[dr] & \text{Pascal} \arrow[ur] \arrow[r] \arrow[dr] & \text{Modula-2} \arrow[r] & \boxed{\text{Modula-3}} \\
 & \boxed{\text{Algol 68}} \arrow[r] & \text{Mesa} \arrow[u] \arrow[r] & \text{Cedar} \arrow[u]
\end{tikzcd}$$



Inheritance in general

```
TYPE
  A = REF RECORD a: REAL END
  AB = REF RECORD
    a: REAL;
    b: BOOLEAN
  END
```

Value set rule suggests AB <: A. In Modula-3,

```
    TYPE
      A = OBJECT a: REAL END;
      AB = A OBJECT b: BOOLEAN END;
```

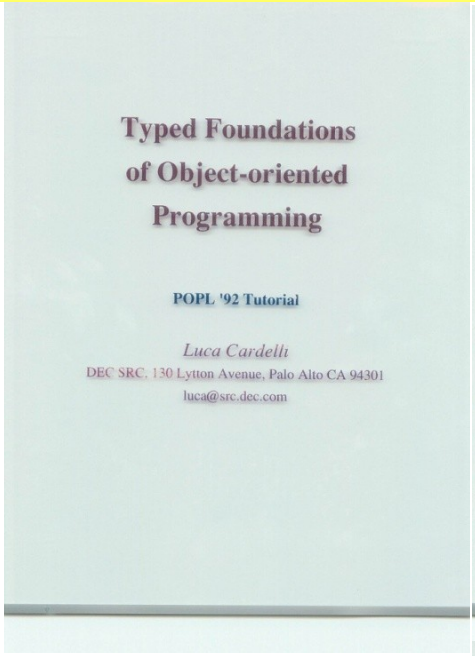
Subtype rules for objects:

```
    OBJECT ... END <: REFANY

    UNTRACED OBJECT ... END <: ADDRESS

    NULL <: T OBJECT ... END <: T
```

14

Signs of Trouble

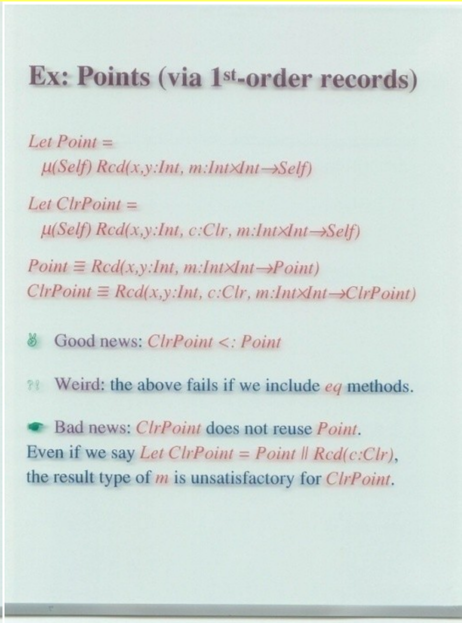
- Mission accomplished!
  - We had subtyping rules for all (classical) type constructors.
- But lots more was now happening
  - “Inheritance is not Subtyping” [Cook Hill Canning ’90]
  - POPL’92 Tutorial: puzzling out recursive ColorPoints.
  - Polymorphic Row Variables [Wand]
  - Operations on Records [with Mitchell] to find a way towards objects.
- Basically, something wasn’t clicking.



Typed Foundations of Object-oriented Programming

POPL '92 Tutorial

Luca Cardelli
DEC SRC, 130 Lytton Avenue, Palo Alto CA 94301
luca@src.dec.com



Ex: Points (via 1st-order records)

Let Point =
μ(Self) Rcd(x,y:Int, m:Int×Int→Self)

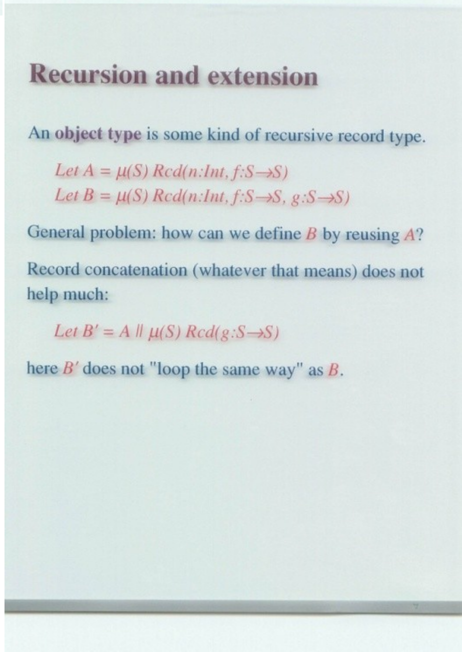
Let ClrPoint =
μ(Self) Rcd(x,y:Int, c:Clr, m:Int×Int→Self)

Point ≡ Rcd(x,y:Int, m:Int×Int→Point)
ClrPoint ≡ Rcd(x,y:Int, c:Clr, m:Int×Int→ClrPoint)

Good news: *ClrPoint <: Point*

?! Weird: the above fails if we include *eq* methods.

Bad news: *ClrPoint* does not reuse *Point*.
Even if we say *Let ClrPoint = Point || Rcd(c:Clr)*,
the result type of *m* is unsatisfactory for *ClrPoint*.



Recursion and extension

An **object type** is some kind of recursive record type.

Let A = μ(S) Rcd(n:Int, f:S→S)
Let B = μ(S) Rcd(n:Int, f:S→S, g:S→S)

General problem: how can we define *B* by reusing *A*?

Record concatenation (whatever that means) does not help much:

Let B′ = A || μ(S) Rcd(g:S→S)

here *B′* does not "loop the same way" as *B*.

Something Fishy About Self

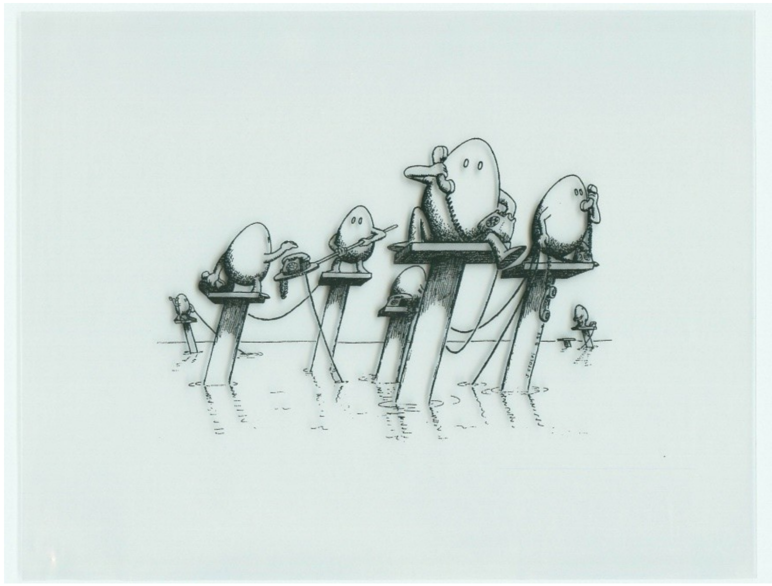
- In the 80's...
  - We had been focusing on "getting the foundations right".
  - But once we got those, we still had difficulties modeling objects and classes.

- By the early 90's...
  - The typed semantics of objects and classes was a widespread and increasingly baroque activity.
  - ColorPoints became the factorial of O-O Type Theory.
  - Handling "self" in a typed calculus was still a major puzzle.

- Reductionism in trouble
  - We could reduce untyped objects to untyped λ-calculus in many ways.
  - We were hoping that reducing object types to typed λ-calculus would help understand the type rules for objects and classes.
  - But that approach, despite great activity and great progress [FOOL workshops], was becoming "heavy".

Meanwhile, back in the Real World

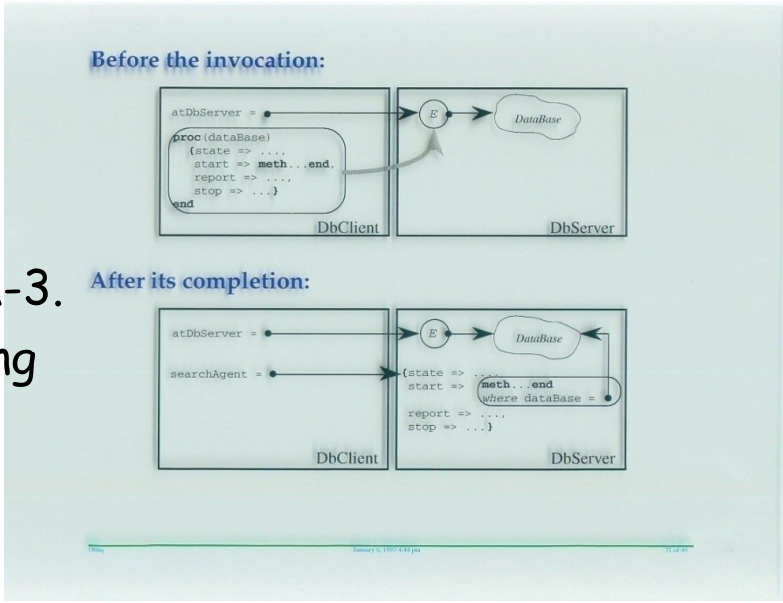
- Apparently...
  - I was not the only one to suspect that I was getting lost in type theory
  - My bosses thought so as well: “Why don’t you do something useful?”



- Ok, I will do something useful
  - It will have objects (objects are useful)
  - It will have no types whatsoever (types are not useful)

Obliq

- *An object-based language*
  - (I was still scared of classes.)
  - For distributed (LAN) programming.
  - A *scripting language* to complement Modula-3.
  - Using the λ-calculus notion of lexical scoping to manage network objects.



```
atDbServer = •
proc(dataBase)
  {state => ...,
   start => meth...end,
   report => ...,
   stop => ...}
end
```

```
atDbServer = •
searchAgent = •
{state => ...,
 start => meth...end
          where dataBase = •
 report => ...,
 stop => ...}
```

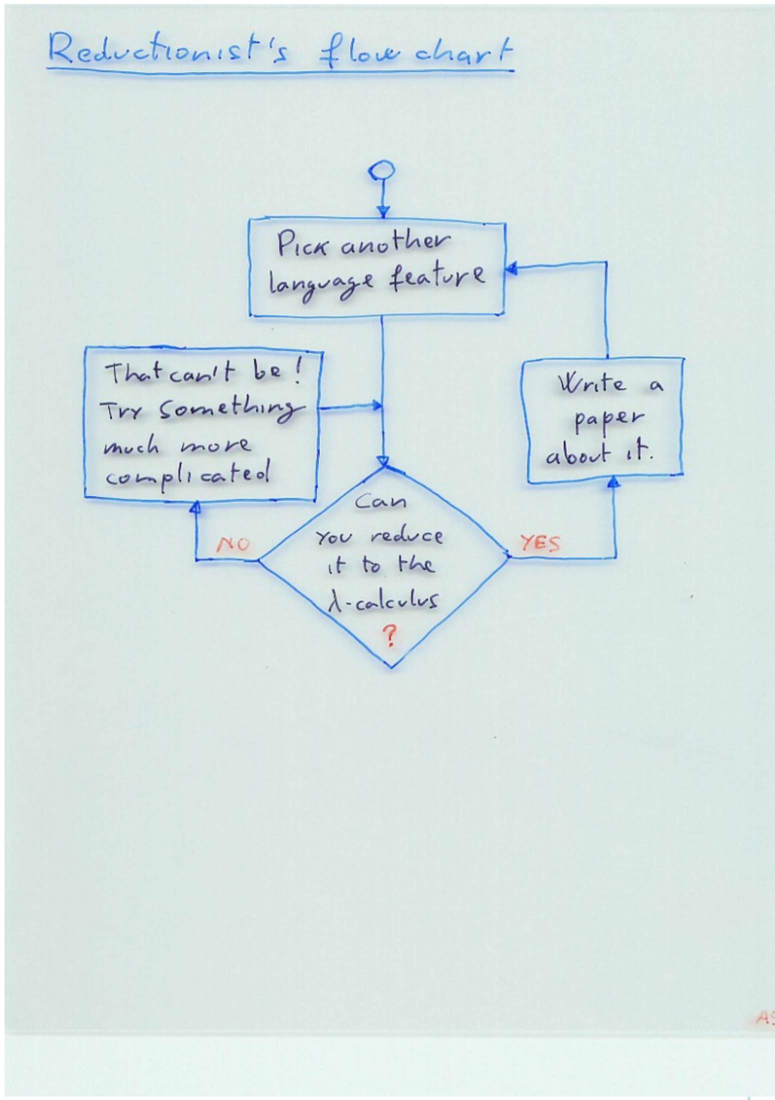
- Born out of schizophrenia
  - Strikingly similar to the object calculus (done at the same time). But:
    - I refused to give semantics or type theory to Obliq (others did).
    - I refused to present Obliq as an implementation of the object calculus.
    - Because this was my "useful" work. Not to be confused!
  - *"Most Influential POPL Paper Award"* 2005 (for 1995)

IV: Objects

A Break with Tradition

- Enough already with λ-calculus!
  - Martin Abadi had invented "Baby Modula-3", as a simplified o-o language to study.
  - We had a conversation about the pain of encodings objects in λ-calculus and we went:
  - This is too hard. Why don't we just invent an object calculus, with *its own* rules, the rules *it should have*, and then *later* we figure out how it relates to lambda-calculus?
  - We were inspired by π-calculus in the sense that there was more out there than λ-calculus!

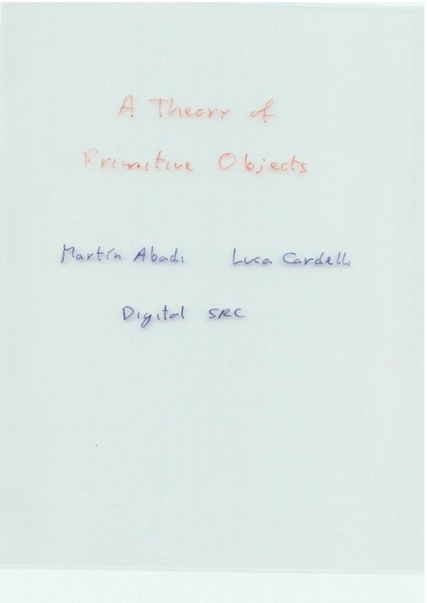
- The Object Calculus
  - Within ½ hour, as I recall, we had the original untyped object calculus on the whiteboard.
  - Very soon we figure out how to encode λ-calculus in it, and then we knew we "had it".
  - Despite the credo of "everything is an object" nobody had bothered to show that "functions are objects too".


Reductionist's flow chart
Pick another language feature
That can't be! Try something much more complicated
Can you reduce it to the λ-calculus ?
NO
YES
Write a paper about it.

A Theory of Objects

- Rapid progress [with Martin Abadi]
  - 1992 POPL Tutorial: lost in type theory.
  - 1994 ESOP: “A Theory of Primitive Objects” (first of many papers)
  - 1996: Book “A Theory of Objects”

A very early web search (1995!) revealed that there were no other “theories of objects”, except in philosophy.

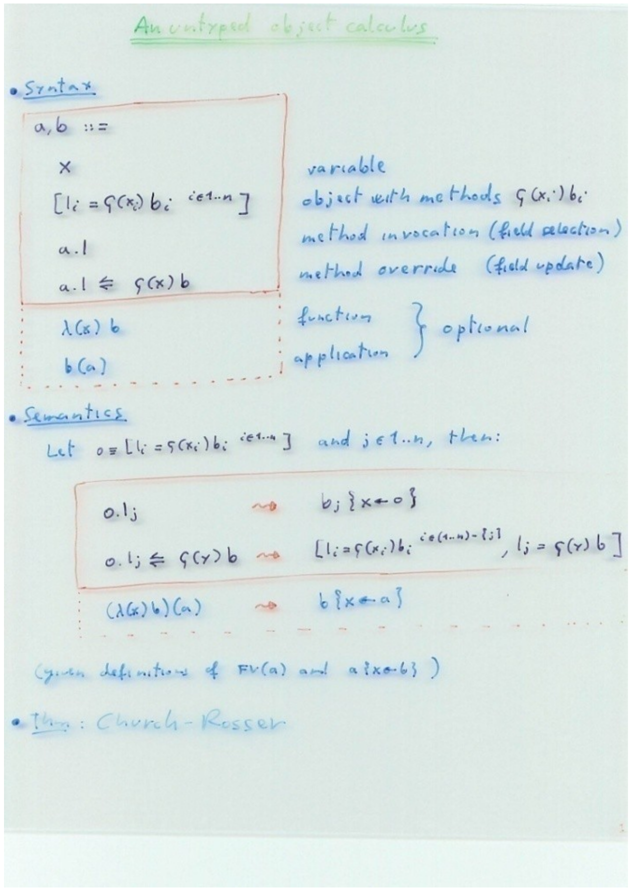


A Theory of

Primitive Objects

Martin Abadi Luca Cardelli

Digital SRC



An untyped object calculus

- Syntax

$a, b ::=$	
x	variable
$[l_i = \varsigma(x_i) b_i \; ^{i \in 1..n}]$	object with methods $\varsigma(x_i) b_i$
$a.l$	method invocation (field selection)
$a.l \Leftarrow \varsigma(x) b$	method override (field update)
$\lambda(x) b$	function } optional
$b(a)$	application }

- Semantics

Let $o \equiv [l_i = \varsigma(x_i) b_i \; ^{i \in 1..n}]$ and $j \in 1..n$, then:

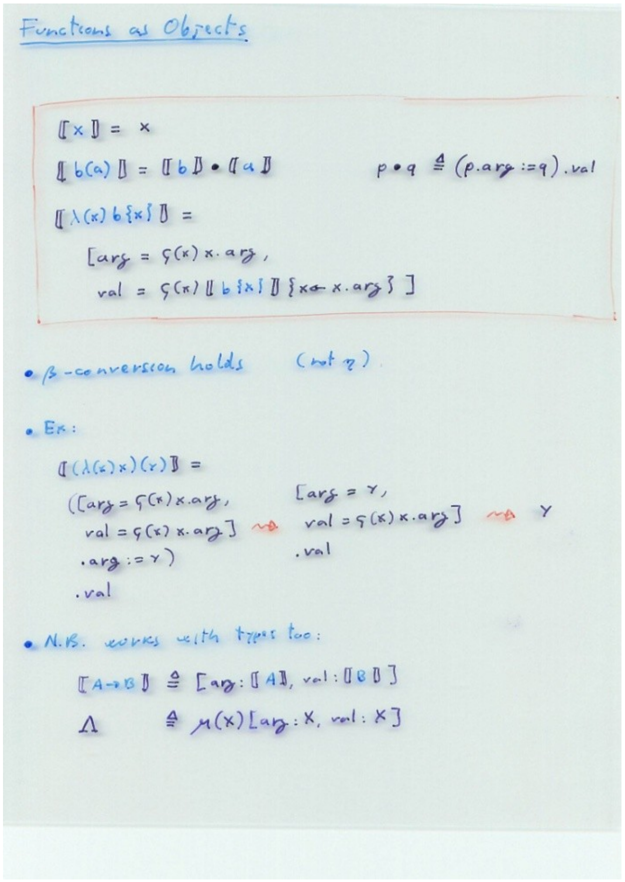
$$o.l_j \rightsquigarrow b_j\{x \leftarrow o\}$$

$$o.l_j \Leftarrow \varsigma(y) b \rightsquigarrow [l_i = \varsigma(x_i) b_i \; ^{i \in (1..n) - \{j\}}, l_j = \varsigma(y) b]$$

$$(\lambda(x) b)(a) \rightsquigarrow b\{x \leftarrow a\}$$

(given definitions of $FV(a)$ and $a\{x \leftarrow b\}$)

- Thm: Church-Rosser



Functions as Objects

$$[\![x]\!] = x$$

$$[\![b(a)]\!] = [\![b]\!] \bullet [\![a]\!] \qquad p \bullet q \triangleq (p.arg := q).val$$

$$[\![\lambda(x) b\{x\}]\!] = [arg = \varsigma(x) x.arg, \; val = \varsigma(x) [\![b\{x\}]\!]\{x \leftarrow x.arg\}]$$

- β-conversion holds (not η)
- Ex:

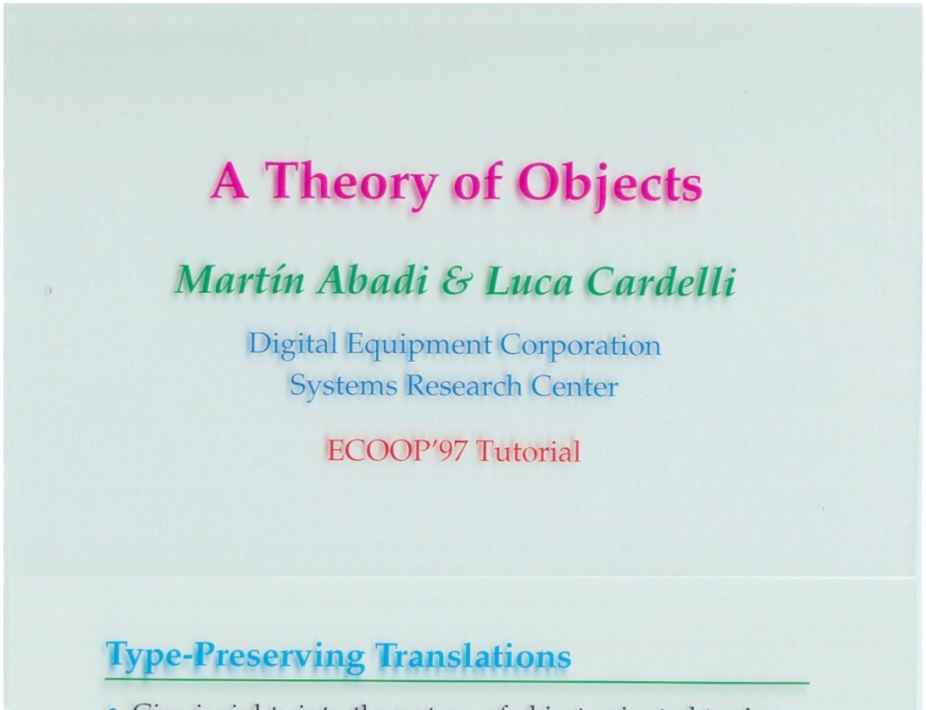
$$[\![(\lambda(x)x)(y)]\!] = ([arg = \varsigma(x) x.arg, \; val = \varsigma(x) x.arg] .arg := y) .val \rightsquigarrow [arg = y, \; val = \varsigma(x) x.arg] .val \rightsquigarrow y$$

- N.B. works with types too:

$$[\![A \to B]\!] \triangleq [arg : [\![A]\!], val : [\![B]\!]]$$

$$\Lambda \triangleq \mu(X)[arg : X, val : X]$$

... the problem in hindsight

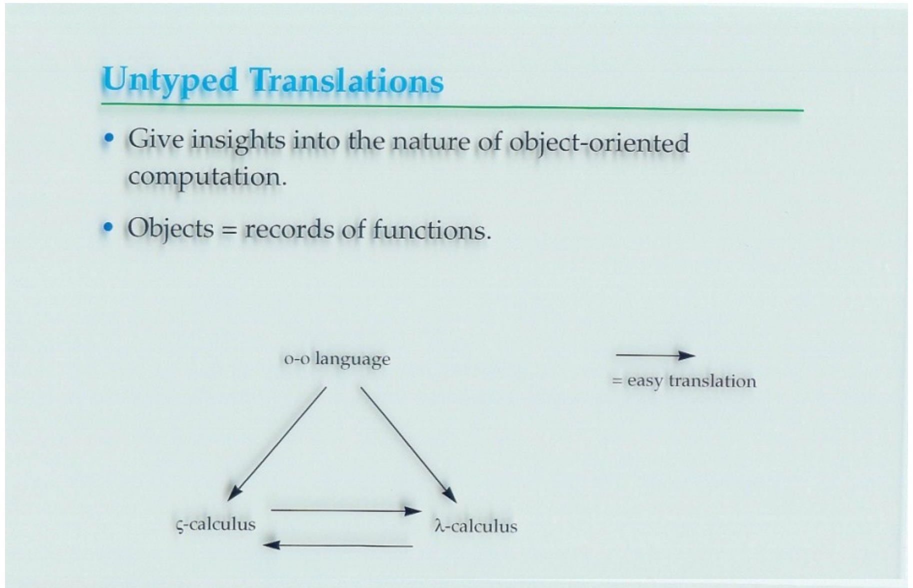


A Theory of Objects

Martín Abadi & Luca Cardelli

Digital Equipment Corporation
Systems Research Center

ECOOP'97 Tutorial

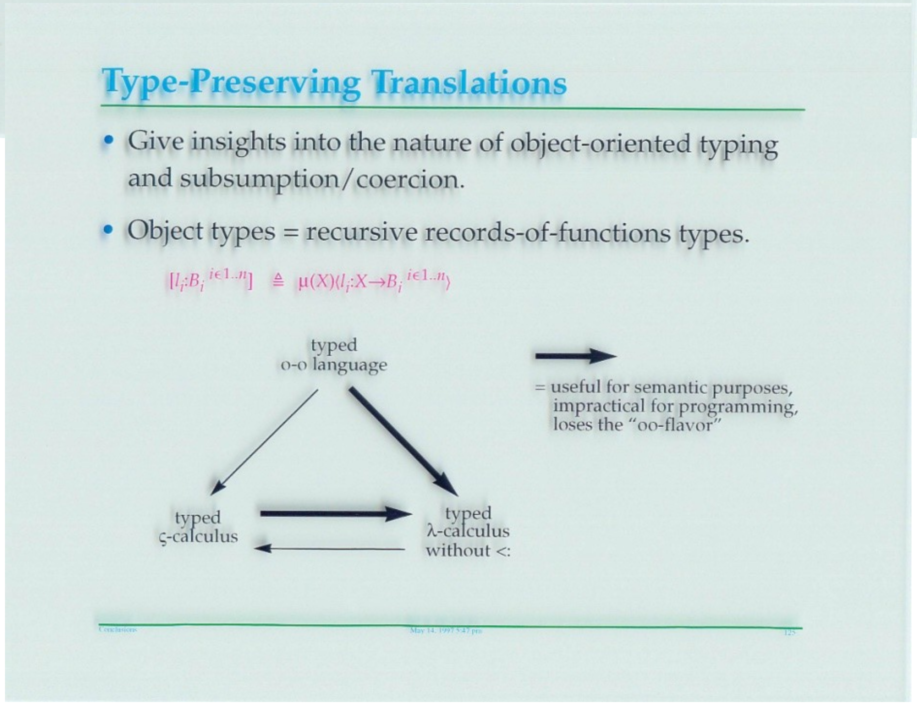


Untyped Translations

- Give insights into the nature of object-oriented computation.
- Objects = records of functions.

$$\begin{tikzcd}
 & \text{o-o language} \arrow[dl] \arrow[dr] & \\
\varsigma\text{-calculus} \arrow[rr, shift left] & & \lambda\text{-calculus} \arrow[ll, shift left]
\end{tikzcd}$$

$\longrightarrow$ = easy translation



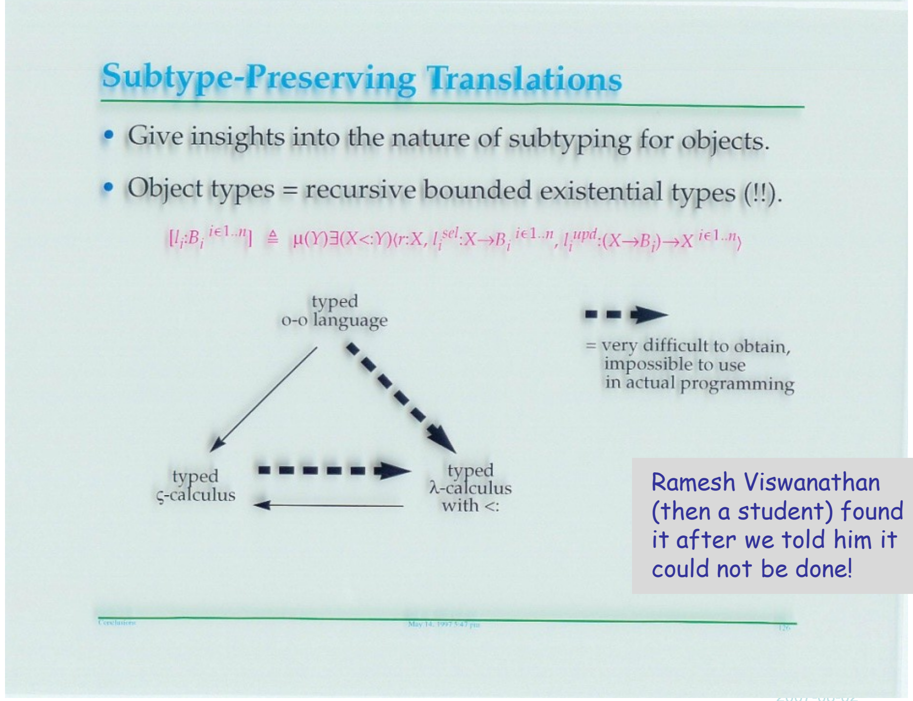
Type-Preserving Translations

- Give insights into the nature of object-oriented typing and subsumption/coercion.
- Object types = recursive records-of-functions types.

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(X)\langle l_i{:}X\to B_i\ ^{i\in 1..n}\rangle$$

$$\begin{tikzcd}
 & \text{typed o-o language} \arrow[dl] \arrow[dr, thick] & \\
\text{typed } \varsigma\text{-calculus} \arrow[rr, shift left, thick] & & \text{typed } \lambda\text{-calculus without } <: \arrow[ll, shift left]
\end{tikzcd}$$

$\pmb{\longrightarrow}$ = useful for semantic purposes, impractical for programming, loses the "oo-flavor"



Subtype-Preserving Translations

- Give insights into the nature of subtyping for objects.
- Object types = recursive bounded existential types (!!).

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(Y)\exists(X<:Y)\langle r{:}X, l_i^{sel}{:}X\to B_i\ ^{i\in 1..n}, l_i^{upd}{:}(X\to B_i)\to X\ ^{i\in 1..n}\rangle$$

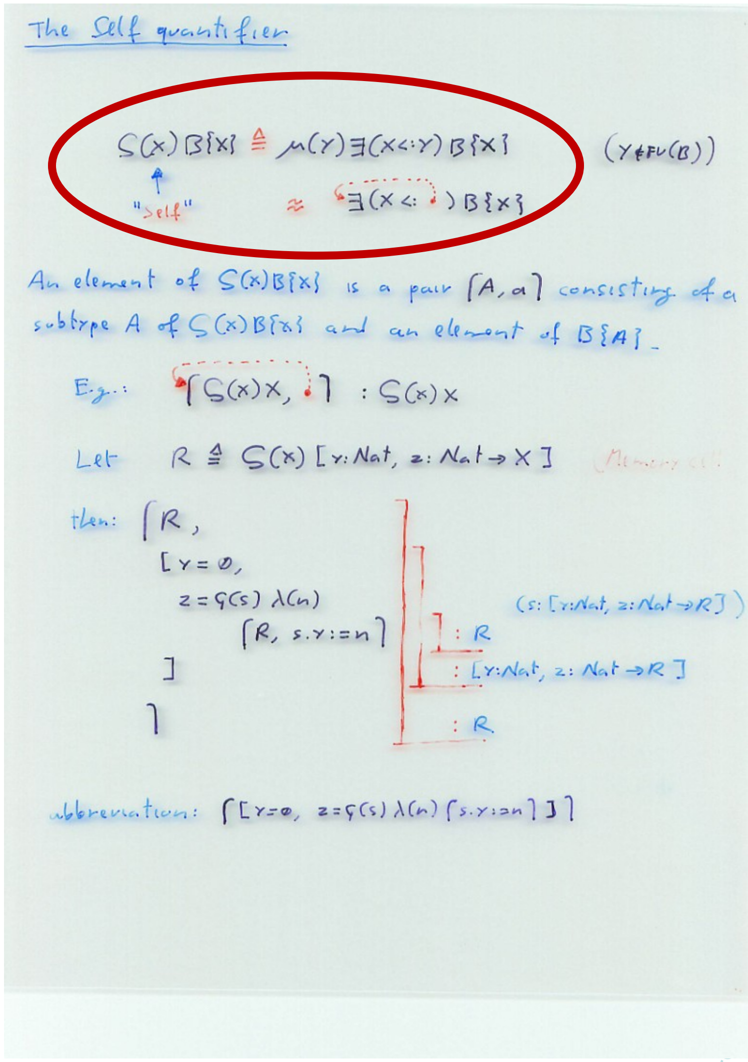
$$\begin{tikzcd}
 & \text{typed o-o language} \arrow[dl] \arrow[dr, dashed, thick] & \\
\text{typed } \varsigma\text{-calculus} \arrow[rr, shift left, dashed, thick] & & \text{typed } \lambda\text{-calculus with } <: \arrow[ll, shift left]
\end{tikzcd}$$

$\dashrightarrow$ = very difficult to obtain, impossible to use in actual programming

Ramesh Viswanathan (then a student) found it after we told him it could not be done!

Finally Nailed Object Types

- Finally, covariance of Self Types
  - By combining bounded Existantials
    - Which are naturally covariant
  - With recursive subtypes
    - I *knew* they would be needed someday!
  - To give a covariant Self Type
    - With recursion going *through the bound!*



The Self quantifier

$\varsigma(X)B\{X\} \triangleq \mu(Y)\exists(X<:Y)B\{X\}$ $(Y \notin FV(B))$

"self" $\approx \exists(X <: \bullet)B\{X\}$

An element of $\varsigma(X)B\{X\}$ is a pair $[A, a]$ consisting of a subtype A of $\varsigma(X)B\{X\}$ and an element of $B\{A\}$.

E.g.: $[\varsigma(X)X, \bullet] : \varsigma(X)X$

Let $R \triangleq \varsigma(X)[y:Nat, z:Nat \to X]$ (Memory cell!)

then:

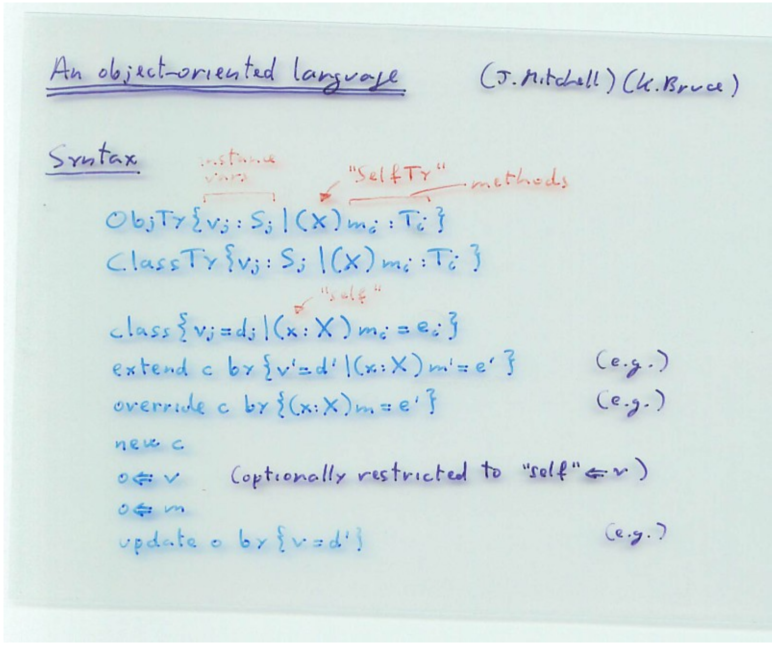
```
[R,
  [y = 0,
   z = ς(s) λ(n)
         [R, s.y:=n]          : R
  ]                           : [y:Nat, z: Nat → R]
]                             : R
```

$(s: [y:Nat, z:Nat \to R])$

abbreviation: $[[y=0,\ z=\varsigma(s)\lambda(n)[s.y:=n]]]$

What About Classes?

- By this time (after 20 years)
  - About ready to tackle Simula's classes.
- In the later chapters of our book
  - (which almost nobody read)
  - we encode Class Types with Self-type, into object calculus.
  - we encode Matching [Bruce] as *third-order* subtyping.
- Reductionism
  - It's still a "reductionist" encoding of class calculi (e.g. the ones due to Mitchell and Bruce) into object calculi (but at least not λ).
  - Is it a good idea to reduce/encode again? That's where we stopped.
  - But we can systematically derive interesting and general type rules for different flavors of class-based languages.



instance vars
"SelfTy"
methods
"self"

An object-oriented language (J. Mitchell) (K. Bruce)

Syntax

$\mathrm{ObjTy}\{v_j : S_j \mid (X)\, m_i : T_i\}$

$\mathrm{ClassTy}\{v_j : S_j \mid (X)\, m_i : T_i\}$

$\mathrm{class}\{v_j = d_j \mid (x : X)\, m_i = e_i\}$

$\mathrm{extend}\ c\ \mathrm{by}\ \{v' = d' \mid (x : X)\, m' = e'\}$ (e.g.)

$\mathrm{override}\ c\ \mathrm{by}\ \{(x : X)\, m = e'\}$ (e.g.)

$\mathrm{new}\ c$

$o \Leftarrow v$ (optionally restricted to "self" $\Leftarrow v$)

$o \Leftarrow m$

$\mathrm{update}\ o\ \mathrm{by}\ \{v = d'\}$ (e.g.)

V: Mainstream

Post-Book

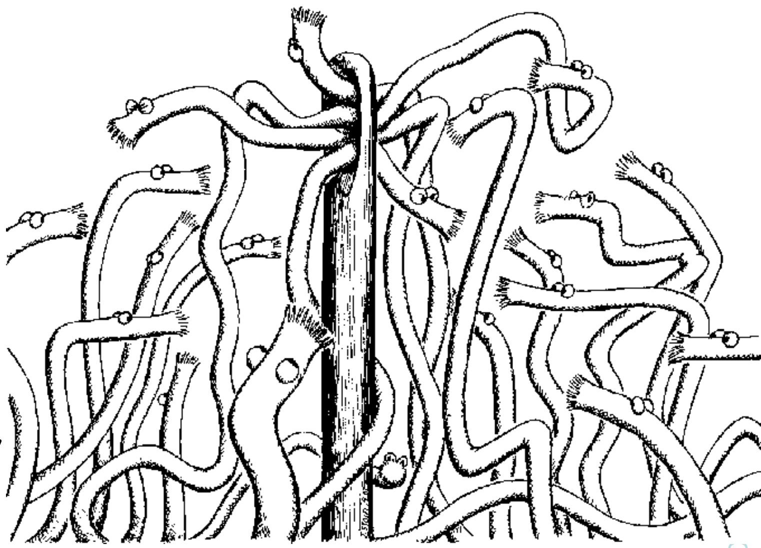
- Java eventually kills Modula-3
  - Without being obviously superior
    - Apart from bytecode verification
  - As I discovered when trying to translate my Modula-3 programs to Java.
    - Programs --> Programs
    - Abstract Types (with subtyping) --> Interfaces
    - Modules --> ?!? Packages ?!? Yucc! gross!

- But progress nonetheless
  - I hear that Gosling "carefully read" our Modula-3 tech reports.
  - Java allowed me to move from one type safe language (Modula-3) to another, narrowly avoiding the entire age of C++ !

  - The original dream comes true: proofs of type soundness for (subsets of) a mainstream o-o language [Drossopoulou, Eisenbach]

At Microsoft Cambridge

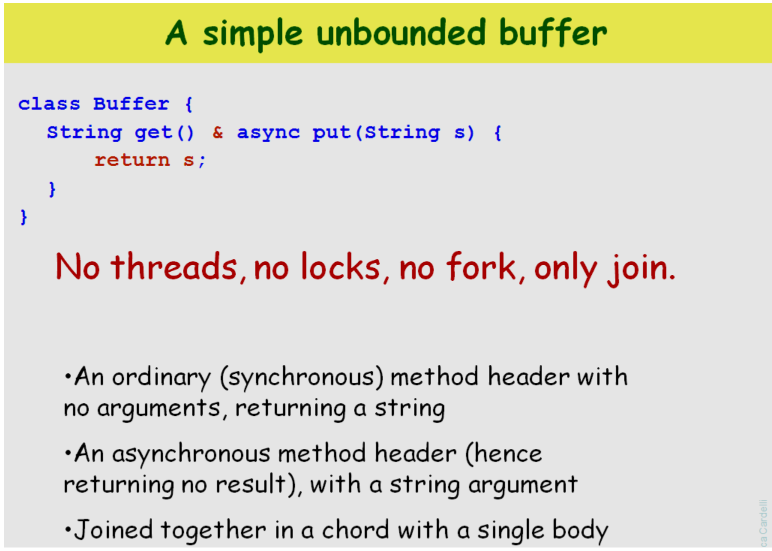
- Generics in .NET and C#
  - Another dream comes true: full parametric polymorphism into mainstream o-o.
  - Thanks to (former) ML programmers! [Don Syme and Andrew Kennedy].
- Still a bit pissed that Java killed Modula-3
  - And a bit burned out about type theory.
  - I wanted to work on some feature that Java didn't have.
- Turning to concurrency (didn't Simula ...)
  - Java copied threads from Modula-3, and I *knew* from extensive personal experience at DEC that threads were *a big yucc! gross! to work with.*
  - (Unfortunately, C# then copied them from Java!)
  - Working on Ambient Calculus (with Andy Gordon), originally inspired by Obliq object mobility, but the world was not ready for that kind of concurrency.



J. Stolfi

Joins

- Looking again for inspiration to functional programming.
  - Concurrency (π-calculus) had been adopted in functional programming (at INRIA) as the much more convenient and very elegant Join Calculus [Gonthier & Fournet]
  - The only new idea in concurrent programming since Simula coroutines and Hoare monitors.
  - What could be the o-o version of Join Calculus?



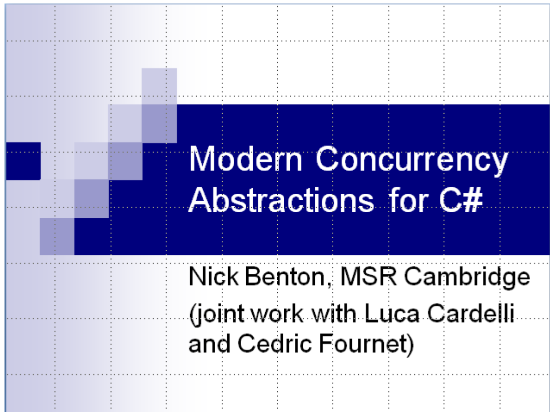
A simple unbounded buffer

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

No threads, no locks, no fork, only join.

- An ordinary (synchronous) method header with no arguments, returning a string
- An asynchronous method header (hence returning no result), with a string argument
- Joined together in a chord with a single body

- Polyphonic C#, and Cω
  - Amazingly, Join Calculus fits extremely naturally in the o-o framework. No threads required!
  - With Cedric Fournet and Nick Benton (and then Claudio Russo) we worked out an extension of C# for join concurrency.
  - I still think it is a widely underappreciated idea.



Modern Concurrency Abstractions for C#

Nick Benton, MSR Cambridge
(joint work with Luca Cardelli and Cedric Fournet)

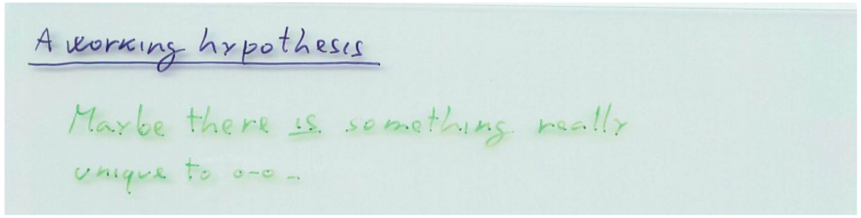
VI: Epilogue

Morale: Embrace Higher-Order

- A goal of Simula (and Beta)
  - has been to embed functions into data, and unify them
- A goal of functional programming
  - has been to embed data into functions, and unify them
- We still see this trend developing today
  - Functional languages (F#) talk to object frameworks.
  - O-o languages (C#) acquire parametric polymorphism, and closures. And soon type inference!
- The true power and the true difficulty
  - Is in higher-order structures meaning: passing functions around as data or inside data.
  - Is in data and computation mixed together.
  - This is the legacy of both Simula and λ-calculus.

Conclusions

- An accidental Simula user
  - How o-o snuck behind a functional programmer and hit him in the head.
- A (very incomplete) fragment of a fragment of history.
  - Too many things I have forgotten!
  - Too many people and ideas to acknowledge!
  - Apologies to EVERYBODY!
- And most of all...
  - Never throw away old slides! They know more than you do...


A working hypothesis

Maybe there is something really unique to o-o.