Foundations of Object-Oriented Programming

Luca Cardelli

joint work with Martín Abadi

Digital Equipment Corporation
Systems Research Center

Horizon Day 10/13/95

Abstract

Object-oriented languages were invented to provide an intuitive view of data and computation, by drawing an analogy between software and the physical world of objects. The detailed explanation of this intuition, however, turned out to be quite complex; there are still no standard definitions of such fundamental notions as objects, classes, and inheritance.

Much progress was made by investigating the notion of subtyping within procedural languages and their theoretical models (lambda calculi). These studies clarified the role of subtyping in object-oriented languages, but still relied on complex encodings to model object-oriented features. Recently, in joint work with Martin Abadi, I have studied more direct models of object-oriented features: object calculi.

Object calculi embody, in a minimal setting, the object-oriented model of computation, as opposed to the imperative, functional, and process models. Object calculi are based exclusively on objects and methods, not on functions or data structures. They help in classifying and explaining the features of object-oriented languages, and in designing new, more regular languages. They directly inspired my design of Obliq, an object-oriented language for network programming.

Outline

- Background
  - ~ Types in programming languages.
  - ~ Object-oriented features.
- Foundations
  - ~ λ-calculi for procedural languages.
  - ~ Object calculi for object-oriented languages.
- Issues
  - ~ Expressiveness.
  - ~ Soundness.

A Brief History of Type

The early days

- Integers and floats (occasionally, also booleans and voids).
- Monomorphic arrays (Fortran).
- Monomorphic trees (Lisp).

The days of structured programming

- Product types (records in Pascal, structs in C).
- Union types (variant records in Pascal, unions in C).
- Function/procedure types (often with various restrictions).
- Recursive types (typically via pointers).

End of the easy part

- This phase culminated with user-definable monomorphic types obtained by combining the constructions above (Pascal, Algol68).

Four major innovations

Polymorphism (ML, etc.).	*(Impredicative universal types.)*
Abstract types (CLU, etc.).	*(Impredicative existentials types.)*
Modules (Modula 2, etc.).	*(Predicative dependent types.)*
Objects and subtyping (Simula 67, etc.).	*(Subtyping + ???)*

- The first three innovations are now largely understood, in isolation, both theoretically and practically. Some of their combinations are also well understood.
- There has been little agreement on the theoretical and practical properties of objects.
- Despite much progress, nobody really knows yet how to combine all four ingredients into coherent language designs.

Confusion

These four innovations are partially overlapping and certainly interact in interesting ways. It is not clear which ones should be taken as more prominent. E.g.:

- Object-oriented languages have tried to incorporate type abstraction, polymorphism, and modularization all at once. As a result, o-o languages are (generally) a mess. Much effort has been dedicated to separating these notions back again.
- Claims have been made (at least initially) that objects can be subsumed by either higher-order functions and polymorphism (ML camp), by data abstraction (CLU camp), or by modularization (ADA camp).
- One hard fact is that full-blown polymorphism can subsume data abstraction. But this kind of polymorphism is more general than, e.g., ML's, and it is not yet clear how to handle it in practice.
- Modules can be used to obtain some form of polymorphism and data abstraction (ADA generics, C++ templates) (Modula 2 opaque types), but not in full generality.

O-O PROGRAMMING

- Goals
  - ~ Data abstraction.
  - ~ Polymorphism.
  - ~ Code reuse.

- Mechanisms
  - ~ Objects with *self* (packages of data and code).
  - ~ Subtyping and subsumption.
  - ~ Classes and inheritance.

Object-oriented constructs

Objects and object types

Objects are packages of data (*instance variables*) and code (*methods*).

Object types describe the shape of objects.

```
ObjectType CellType;
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

object cell: CellType;
    var contents: Integer := 0;
    method get(): Integer; return self.contents end;
    method set(n: Integer); self.contents := n end;
end;
```

where $a : A$ means that the program a has type A. So, *cell* : *CellType*.

Classes

Classes are ways of describing and generating collections of objects.

```
class cellClass for CellType;
    var contents: Integer := 0;
    method get(): Integer; return self.contents end;
    method set(n: Integer); self.contents := n end;
end;
var cell : CellType := new cellClass;
procedure double(aCell: CellType);
    aCell.set(2 * aCell.get());
end;
```

Subclasses

Subclasses are ways of describing classes incrementally, reusing code.

```
ObjectType ReCellType;
    var contents: Integer;
    var backup: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;

subclass reCellClass of cellClass for ReCellType;            (Inherited:
    var backup: Integer := 0;                                   var contents
    override set(n: Integer);                                   method get)
        self.backup := self.contents;
        super.set(n);
    end;
    method restore(); self.contents := self.backup end;
end;
```

Subtyping and subsumption

- Subtyping relation, $A <: B$

  An object type is a subtype of any object type with fewer components.

  (e.g.: *ReCellType* <: *CellType*)

- Subsumption rule

  if $a : A$ and $A <: B$ then $a : B$

  (e.g.: *reCell* : *CellType*)

- Subclass rule

  cClass can be a subclass of *dClass* only if *cType* <: *dType*

  (e.g.: *reCellClass* can indeed be declared as a subclass of *cellClass*)

Healthy skepticism

- Object-oriented languages have been plagued, more than any other kind of languages, but confusion and unsoundness.
- How do we keep track of the interactions of the numerous object-oriented features?
- How can we be sure that it all makes sense?

The λ-calculus

The simplest procedural language.

$b ::=$	terms	
x	identifiers	
$\lambda(x)b$	functions	(i.e. **procedure** (x) **return** b **end**)
$b_1(b_2)$	applications	

$x := b$	assignments

- Functional Semantics:

  $(\lambda(x)b)(b_1) \rightsquigarrow b\{x \leftarrow b_1\}$ (β-reduction)

- Imperative Semantics:

  more complicated, store-based.

... also the hardest procedural language.

~ Scoping (cf. Lisp's botch, Algol's blocks).

~ Data structures (numbers, trees, etc.).

~ Controls structures (parameters, declarations, state encapsulation, conditionals, loops, recursion, continuations).

~ module structures (interfaces, genericity, visibility).

~ Typing (soundness, polymorphism, data abstraction).

~ Semantics (formal language definitions).

The Functional Point of View

- *Functions* (or procedures) *are the most interesting aspect of computation.*
- Various λ-calculi are seen both as paradigms and foundations for procedural languages. (E.g.: Landin/Reynolds for Algol, Milner for ML.)

According to the functional approach, objects, like anything else, ought to be explained by some combination of functions.

But people working on and with object-oriented language do not think that functions are so interesting ...

However...

- The Simula lament:

  "Unlike procedural languages, object-oriented languages have no formal foundation."
  (I.e.: We made it up.)

- The Smalltalk axiom:

  "Everything is an object. I mean, EVERYTHING."
  (I.e.: If you have objects, you don't need functions.)

- The C++ / Eiffel / etc. trade press:

  "A revolutionary software life-cycle paradigm."
  (I.e.: Don't call procedures, invoke methods!)

They all say: *These are no ordinary languages*.

They reject the reductionist approach of mapping everything to the λ-calculus.

If there is something really unique to O-O, then ...

There ought to be a formalism comparable to the λ-calculus, such that:

- It is computationally complete.
- It is based entirely on objects, not functions.
- It can be used as a paradigm and a foundation for object-oriented language.
- It can explain object-oriented concepts more directly and fruitfully than functional encodings.

Some evidence to the contrary:

- Objects have methods, methods have parameters, parameters are λ's, therefore any object formalism is an extension of the λ-calculus, not a replacement for it.

And yet...

The ς-calculus

The simplest object-oriented language.

$b ::=$	terms
x	identifiers
$[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$	objects (i.e. **object**[l = **method**()...**self**...**end**, ...])
$b.l$	method invocation (with no parameters)
$b_1.l \Leftarrow \varsigma(x)b_2$	method update

clone(b)	cloning
let $x = b_1$ *in* b_2	local declaration (yields *fields*)

- Fields can be encoded:

  $[..., l = b, ...]$

  $b_1.l := b_2$

Reduction rules of the ς-calculus

- The notation $b \leadsto c$ means that b reduces to c.

  Let $o \equiv [l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$ (l_i distinct)

  $$o.l_j \leadsto b_j\{x_j{\leftarrow}o\} \qquad (j\in 1..n)$$

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

Theorem: Church-Rosser

If $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, then there exists d such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

(Where $\twoheadrightarrow$ is the reflexive, transitive, and contextual closure of $\leadsto$.)

~ We are dealing with a calculus of objects (not of functions).

~ The semantics is deterministic. It is neither imperative nor concurrent.

~ We have investigated imperative versions of the calculus.

~ We have not yet investigated a concurrent version.

Basic Examples

Let	$o_1 \triangleq [l{=}\varsigma(x)[\,]]$	A convergent method.
then	$o_1.l \leadsto [\,]$	
Let	$o_2 \triangleq [l{=}\varsigma(x)x.l]$	A divergent method.
then	$o_2.l \leadsto x.l\{x{\leftarrow}o_2\} \equiv o_2.l \leadsto \dots$	
Let	$o_3 \triangleq [l = \varsigma(x)x]$	A self-returning method.
then	$o_3.l \leadsto x\{x{\leftarrow}o_3\} \equiv o_3$	
Let	$o_4 \triangleq [l = \varsigma(y)\ (y.l{\Leftarrow}\varsigma(x)x)]$	A self-modifying method.
then	$o_4.l \leadsto (o_4.l{\Leftarrow}\varsigma(x)x) \leadsto o_3$	

... also the hardest object-oriented language.

~ role of self (hidden recursion)

~ data structures (numbers, trees, etc.)

~ controls structures (functions, classes, state encapsulation, conditionals, loops, recursion)

~ typing (soundness, subtyping, Self types)

~ semantics (formal o-o language definitions)

A.k.a. Obliq

b ::=	terms
x	identifiers
$\{l_i$ **=> meth**$(x_i)b_i$ **end** $^{i \in 1..n}\}$	objects
$b.l$	method invocation
$b_1.l$ **:= meth**$(x)b_2$ **end**	method update
clone(b)	cloning
let $x = b_1$ **in** b_2 **end**	local declaration (yields *fields*)

Functions from Objects

$$\langle\!\langle x \rangle\!\rangle \triangleq x$$

$$\langle\!\langle b(a) \rangle\!\rangle \triangleq (\langle\!\langle b \rangle\!\rangle.arg \Leftarrow \varsigma(x)\langle\!\langle a \rangle\!\rangle).val \qquad x \notin FV(a)$$

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

Example:

$$\langle\!\langle (\lambda(x)x)(a) \rangle\!\rangle \equiv ([arg = \varsigma(x)\, x.arg,\ val = \varsigma(x)\langle\!\langle x \rangle\!\rangle\{x \leftarrow x.arg\}].arg \Leftarrow \varsigma(z)\langle\!\langle a \rangle\!\rangle).val$$
$$\rightsquigarrow \langle\!\langle a \rangle\!\rangle$$

Preview: this encoding extends to typed calculi:

$$\langle\!\langle A \rightarrow B \rangle\!\rangle \triangleq [arg: \langle\!\langle A \rangle\!\rangle, val: \langle\!\langle B \rangle\!\rangle] \qquad \text{1st-order } \lambda \text{ into 1st-order } \varsigma$$

- β-reduction is validated:

$$let\ o \equiv [arg = \varsigma(z)\, \langle\!\langle a \rangle\!\rangle, val = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle\{x \leftarrow x.arg\}]$$

$$\begin{aligned}
&\langle\!\langle (\lambda(x)b\{x\})(a) \rangle\!\rangle \\
&\quad \equiv ([arg = \varsigma(x)\, x.arg,\ val = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle\{x \leftarrow x.arg\}].arg \Leftarrow \varsigma(z)\langle\!\langle a \rangle\!\rangle).val \\
&\quad = o.val = (\langle\!\langle b\{x\} \rangle\!\rangle\{x \leftarrow x.arg\})\{x \leftarrow o\} \\
&\quad = \langle\!\langle b\{x\} \rangle\!\rangle\{x \leftarrow o.arg\} = \langle\!\langle b\{x\} \rangle\!\rangle\{x \leftarrow \langle\!\langle a \rangle\!\rangle\} \\
&\quad = \langle\!\langle b\{a\} \rangle\!\rangle
\end{aligned}$$

- Roughly the same technique extends to imperative calculi, and to various typed calculi.
- Generalizes to default parameters and call-by-keyword.
- Thus, procedural languages are reduced to object-oriented languages.

Objects from Functions

$$\langle\!\langle x \rangle\!\rangle \triangleq x$$
$$\langle\!\langle [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}] \rangle\!\rangle \triangleq \langle l_i = \lambda(x_i)\langle\!\langle b_i \rangle\!\rangle \ ^{i\in 1..n}\rangle$$
$$\langle\!\langle b.l \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle.l(\langle\!\langle b \rangle\!\rangle)$$
$$\langle\!\langle b_1.l \Leftarrow \varsigma(x)b_2 \rangle\!\rangle \triangleq \langle\!\langle b_1 \rangle\!\rangle.l{:=}\lambda(x)\langle\!\langle b_2 \rangle\!\rangle$$

Preview: this translation does *not* extend to typed calculi.

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

But NOT, *e.g.*: $\mu(X)\langle l{:}X\rightarrow A,\ l'{:}X\rightarrow B\rangle <: \mu(Y)\langle l{:}Y\rightarrow A\rangle$

Example: A Storage Cell

Let $cell \triangleq [contents = 0,\ set = \varsigma(x)\ \lambda(n)\ x.contents := n]$

then $cell.set(3)$

$\rightsquigarrow (\lambda(n)[contents = 0,\ set = \varsigma(x)\ \lambda(n)\ x.contents := n]$
$\quad .contents{:=}n)(3)$

$\rightsquigarrow [contents = 0,\ set = \varsigma(x)\lambda(n)\ x.contents := n].contents{:=}3$

$\rightsquigarrow [contents = 3,\ set = \varsigma(x)\ \lambda(n)\ x.contents := n]$

and $cell.set(3).contents$

$\rightsquigarrow$ ...

$\rightsquigarrow 3$

Basic types (such as booleans and integers) can be added as primitive, or encoded.

Example: Object-Oriented Naturals

- Each numeral has a *case* field that contains either $\lambda(z)\lambda(s)z$ for zero, or $\lambda(z)\lambda(s)s(x)$ for non-zero, where x is the predecessor (self).
- Each numeral has a *succ* method that can modify the *case* field to the non-zero version.

Informally: $n.case(z)(s) = $ if n is *zero* then z else $s(n\text{-}1)$

$zero \triangleq$
$\quad [case = \lambda(z)\ \lambda(s)\ z,$
$\quad\ succ = \varsigma(x)\ x.case := \lambda(z)\ \lambda(s)\ s(x)\ ]$

So:

$$\begin{array}{llll}
zero & & & \equiv\ [case = \lambda(z)\ \lambda(s)\ z,\ \ succ = ...\ ] \\
one & \triangleq & zero.succ & \equiv\ [case = \lambda(z)\ \lambda(s)\ s(zero),\ \ succ = ...\ ] \\
two & \triangleq & one.succ & \equiv\ [case = \lambda(z)\ \lambda(s)\ s(one),\ \ succ = ...\ ]
\end{array}$$

$iszero \triangleq \lambda(n)\ n.case(true)(\lambda(p)false)$

$pred \triangleq \lambda(n)\ n.case(zero)(\lambda(p)p)$

Classes from Objects

- Inheritance is method reuse. But one cannot reuse methods of existing objects: method extraction is not type-sound in typed languages. This is why we need classes, on top of objects, to achieve inheritance.
- A *pre-method* is a function that is later used as a method.
- A class is a collection of pre-methods plus a way of generating new objects.

- If $o \equiv [l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$ is an object,

  $c \equiv [l_i = \lambda(x_i)b_i\ ^{i\in 1..n},$

  $\quad new = \varsigma(z)[l_i = \varsigma(s)\ z.l_i(s)\ ^{i\in 1..n}]\ ]$

  then c is a class for generating objects like o.

- A (sub)class c' may inherit pre-methods from c:

  $c' \equiv [\ldots,\ l_k = c.l_k,\ \ldots$

  $\quad new = \ldots\ ]$
- Roughly the same technique extends to imperative calculi, and to various typed calculi.
- Thus, class-based languages are reduced to object-based languages.

Object Types

An **object type**

$[l_i{:}B_i\ ^{i\in 1..n}]$

is the type of those objects with methods l_i, with a self parameter of type $A <: [l_i{:}B_i\ ^{i\in 1..n}]$ and a result of type B_i.

An object type with more methods is a **subtype** of one with fewer methods:

$[l_i{:}B_i\ ^{i\in 1..n+m}] \quad <: \quad [l_i{:}B_i\ ^{i\in 1..n}]$

Object types are **invariant** (not covariant, not contravariant) in their components.

An object can be used in place of another object with fewer methods, by **subsumption**:

$a : A \quad \wedge \quad A <: B \quad \Rightarrow \quad a : B$

This is the basis for a kind of polymorphism, and useful for inheritance:

$f : B \to C \quad \wedge \quad a : A \quad \wedge \quad A <: B \quad \Rightarrow \quad f(a) : C$

f implements l in B $\quad \wedge \quad A <: B \quad \Rightarrow \quad$ f can implement l in A

A First-Order Calculus

Judgments

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash a : A$	a has type A in E

Environments

$E \equiv x_i{:}A_i\ ^{i\in 1..n}$	environments, with x_i distinct

Types

$A,B \ ::=$	Top	the biggest type
	$[l_i{:}B_i\ ^{i\in 1..n}]$	object types, with l_i distinct

Terms

As for the untyped calculus, but with types for bound variables.

Typing Rules

The object fragment:

(Type Object) $\quad$ (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i\in 1..n}{E \vdash [l_i{:}B_i\ ^{i\in 1..n}]}$$

(Sub Object) $\quad$ (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i\in 1..n{+}m}{E \vdash [l_i{:}B_i\ ^{i\in 1..n+m}] <: [l_i{:}B_i\ ^{i\in 1..n}]}$$

(Val Object) $\quad$ (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

$$\frac{E,\ x_i{:}A \vdash b_i : B_i \qquad \forall i\in 1..n}{E \vdash [l_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}] : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i{:}B_i\ ^{i\in 1..n}] \qquad j\in 1..n}{E \vdash a.l_j : B_j}$$

(Val Update) $\quad$ (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E,\ x{:}A \vdash b : B_j \qquad j\in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A}$$

With some additional, standard rules we obtain a complete calculus:

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E, x{:}A \vdash \diamond}$$

(Val x)
$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x{:}A}$$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Val Subsumption)
$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

Theorem (Minimum types)
If $E \vdash a : A$ then there exists B such that $E \vdash a : B$ and, for any A', if $E \vdash a : A'$ then $E \vdash B <: A'$.

Theorem (Subject reduction)
If $\emptyset \vdash a : C$ and $a \rightsquigarrow v$ then $\emptyset \vdash v : C$.

Function Types

Translation of function types:

$$\langle\!\langle A \to B \rangle\!\rangle \triangleq [arg{:}\langle\!\langle A \rangle\!\rangle, val{:}\langle\!\langle B \rangle\!\rangle]$$

$$\langle\!\langle x \rangle\!\rangle_\rho \triangleq \rho(x)$$

$$\langle\!\langle b(a) \rangle\!\rangle_\rho \triangleq$$
$$(\langle\!\langle b \rangle\!\rangle_\rho.arg \Leftarrow \varsigma(x)\, \langle\!\langle a \rangle\!\rangle_\rho).val \qquad \text{for } x \notin FV(\langle\!\langle a \rangle\!\rangle_\rho)$$

$$\langle\!\langle \lambda(x{:}A)b \rangle\!\rangle_\rho \triangleq$$
$$[arg = \varsigma(x)\, x.arg,$$
$$val = \varsigma(x)\, \langle\!\langle b \rangle\!\rangle_{\rho\{x \leftarrow x.arg\}}]$$

According to this translation, $A \to B$ is invariant!

(There are several ways to obtain variant function types in richer object calculi.)

Classes

If $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$ is an object type, then:

$$Class(A) \triangleq [new{:}A,\ l_i{:}A \to B_i\ ^{i \in 1..n}]$$

where

$new{:}A$	is a **generator** for objects of type A
$l_i{:}A \to B_i$	is a **pre-method** for objects of type A

$$c : Class(A) \triangleq$$
$$[new = \varsigma(c{:}Class(A))\ [l_i = \varsigma(x{:}A)\ c.l_i(x)\ ^{i \in 1..n}],$$
$$l_i = \lambda(x_i{:}A)\ b_i\{x_i\}\ ^{i \in 1..n}]$$

We can produce new objects as follows:

$$c.new \equiv [l_i = \varsigma(x{:}A)\ b_i\{x\}\ ^{i \in 1..n}] : A$$

Inheritance

Define inheritance as a new relation between class types:

$$Class(A') \text{ may inherit from } Class(A) \quad \text{iff} \quad A'<:A$$

Let $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ and $A' \equiv [l_i{:}B_i\ ^{i\in 1..n}, l_j{:}B_j\ ^{j\in n+1..m}]$, with $A' <: A$.
Note that $Class(A)$ and $Class(A')$ are not related by subtyping.

Let c: $Class(A)$, then

$$c.l_i{:}\ A \rightarrow B_i <: A' \rightarrow B_i.$$

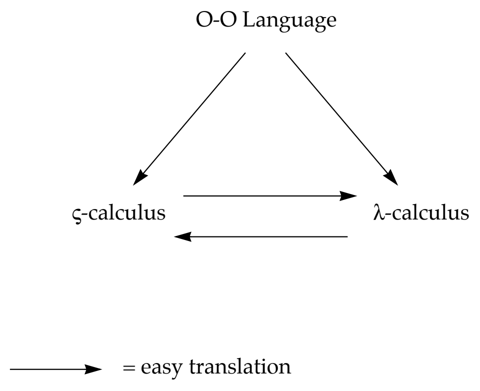
Hence $c.l_i$ is a good pre-method for $Class(A')$. For example, we may define:

$$c' \triangleq [new{=}..., l_i{=}c.l_i\ ^{i\in 1..n}, l_j{=}...\ ^{j\in n+1..m}] : Class(A')$$

where class c' **inherits** the methods l_i from class c.

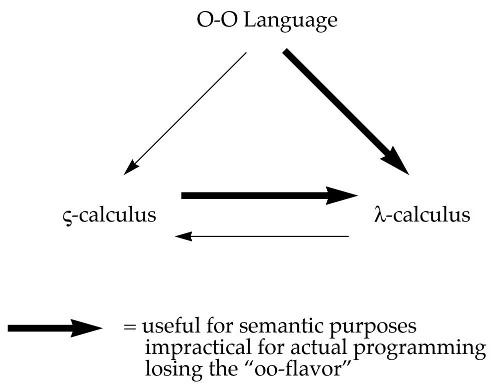
Untyped Translations

- Give insights into the nature of object-oriented computation.



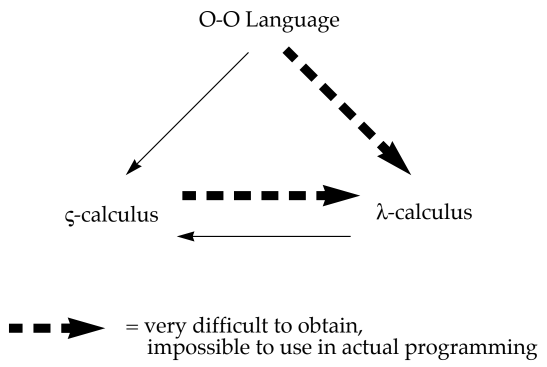
Type-preserving Translations

- Give insights into the nature of object-oriented typing and subsumption/coercion.


O-O Language
ς-calculus
λ-calculus
= useful for semantic purposes
impractical for actual programming
losing the "oo-flavor"

Subtype-preserving Translations

- Give insights into the nature of subtyping for object types.


O-O Language
ς-calculus
λ-calculus
= very difficult to obtain,
impossible to use in actual programming

CONCLUSIONS

- Expressiveness
  - ~ Pure object-based languages are as expressive as procedural languages. (Despite all the Smalltalk claims, to our knowledge nobody had previously shown formally that one can build functions out of objects.)
  - ~ Classes can be easily and faithfully encoded into object calculi. Thus, *object-based* languages are simpler and just as expressive as *class-based* ones. (To our knowledge, nobody had previously shown that one can build type-correct classes out of objects.)
- Language soundness
  - ~ The simple untyped ς-calculus is a good foundation for studying rich object-oriented type systems (including polymorphism, Self types, etc.) and to prove their soundness. We have done much work in this area.
  - ~ Practical object-oriented languages can be shown sound by fairly direct subtype-preserving translations into object calculi.
  - ~ We can make (some) sense of object-oriented languages.

- Foundations
  - ~ Subtype-preserving translations of object calculi, into lambda-calculi are extremely difficult to obtain.
  - ~ In contrast, subtype-preserving translations of lambda-calculi into object-calculi can be easily obtained.
  - ~ In this sense, object calculi are more fundamental than λ-calculi.
- Other developments
  - ~ Imperative calculi.
  - ~ Second-order object types for "Self types".
  - ~ Higher-order object types for "matching".
- Potential future areas
  - ~ Typed ς-calculi should be a good simple foundation for studying object-oriented specification and verification (a still largely underdeveloped area).
  - ~ They should also give us a formal platform for studying object-oriented concurrent languages (as opposed to "ordinary" concurrent languages).

References

http://www.research.digital.com/SRC/personal/Luca_Cardelli/TheoryOfObjects.html