An introduction to Object-Oriented Calculus

Simon Fortier-Garceau

June 5-8, 2014

Why an object calculus?

Why an object calculus?

- Procedural languages are generally well understood and have a supporting theory (λ-calculus)

Why an object calculus?

- Procedural languages are generally well understood and have a supporting theory (λ-calculus)
- By contrast, no such foundations exist for object-oriented languages

Why an object calculus?

- Procedural languages are generally well understood and have a supporting theory (λ-calculus)
- By contrast, no such foundations exist for object-oriented languages
- Interpretations of untyped objects in untyped λ-calculus is possible...

Why an object calculus?

- Procedural languages are generally well understood and have a supporting theory (λ-calculus)
- By contrast, no such foundations exist for object-oriented languages
- Interpretations of untyped objects in untyped λ-calculus is possible...
- But typed object-oriented languages (OOL) are not easily emulated in simply typed λ-calculus

Why an object calculus?

- Procedural languages are generally well understood and have a supporting theory (λ-calculus)
- By contrast, no such foundations exist for object-oriented languages
- Interpretations of untyped objects in untyped λ-calculus is possible...
- But typed object-oriented languages (OOL) are not easily emulated in simply typed λ-calculus
- Yet, the basics of typed OOL are simple enough...

$\rightarrow$ This suggests investigation of a calculus where objects are used as primitives

Principles of object-oriented

The four major principles of object-oriented programming (OOP) are :

- Encapsulation : Restricting the manipulation of internal data to the host object methods
- Abstraction : The ability to abstract attributes and implementation details of an object
- Inheritance : The ability of certain classes to call upon method implementations of another class
- Polymorphism : The ability of an object to take many shapes

Principles of process calculi

- Describe concurrent systems and their behavior

Principles of process calculi

- Describe concurrent systems and their behavior
- Specify how processes communicate or interact (use of channels and messages)

Principles of process calculi

- Describe concurrent systems and their behavior
- Specify how processes communicate or interact (use of channels and messages)
- Sequentialization of actions

Principles of process calculi

- Describe concurrent systems and their behavior
- Specify how processes communicate or interact (use of channels and messages)
- Sequentialization of actions
- Non-deterministic choice in processing

Principles of process calculi

- Describe concurrent systems and their behavior
- Specify how processes communicate or interact (use of channels and messages)
- Sequentialization of actions
- Non-deterministic choice in processing
- Recursion and/or replication of processes

Principles of process calculi

- Describe concurrent systems and their behavior
- Specify how processes communicate or interact (use of channels and messages)
- Sequentialization of actions
- Non-deterministic choice in processing
- Recursion and/or replication of processes
- Hiding or restriction of channels

Principles of process calculi

- Describe concurrent systems and their behavior
- Specify how processes communicate or interact (use of channels and messages)
- Sequentialization of actions
- Non-deterministic choice in processing
- Recursion and/or replication of processes
- Hiding or restriction of channels

... can we combine these with OO principles?

Merging objects and processes

- OOL programs are composed of well-behaved independent parts that are easy to type and assemble (through contracts) to form more complicated and consistent systems...

Merging objects and processes

- OOL programs are composed of well-behaved independent parts that are easy to type and assemble (through contracts) to form more complicated and consistent systems...
- but they do not come equipped with evaluation strategies that provide insight in concurrent interactions

Merging objects and processes

- OOL programs are composed of well-behaved independent parts that are easy to type and assemble (through contracts) to form more complicated and consistent systems...
- but they do not come equipped with evaluation strategies that provide insight in concurrent interactions
- Process algebras are much better at rendering concurrent systems...

Merging objects and processes

- OOL programs are composed of well-behaved independent parts that are easy to type and assemble (through contracts) to form more complicated and consistent systems...
- but they do not come equipped with evaluation strategies that provide insight in concurrent interactions
- Process algebras are much better at rendering concurrent systems...
- but processes are volatile creatures that are much harder to type and, as such, are not the subject of “good” engineering principles

Merging objects and processes

- OOL programs are composed of well-behaved independent parts that are easy to type and assemble (through contracts) to form more complicated and consistent systems...
- but they do not come equipped with evaluation strategies that provide insight in concurrent interactions
- Process algebras are much better at rendering concurrent systems...
- but processes are volatile creatures that are much harder to type and, as such, are not the subject of "good" engineering principles

Ideally, we could interpret objects as processes, or processes as objects, or perhaps, find a different formal system that captures OO style typing and concurrency.

The basics of ς-calculus

Primitives of the syntax

Objects and methods

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)
$o.l$	invocation of method l of object o

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)
$o.l$	invocation of method l of object o
$o.l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)
$o.l$	invocation of method l of object o
$o.l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)
$o.l$	invocation of method l of object o
$o.l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

- Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$, a component $l_i = \varsigma(x_i)b_i$ of o consists of a label l_i and a method $\varsigma(x_i)b_i$.

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)
$o.l$	invocation of method l of object o
$o.l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

- Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$, a component $l_i = \varsigma(x_i)b_i$ of o consists of a label l_i and a method $\varsigma(x_i)b_i$.
- The letter ς in the method acts as a binder for the self parameter of the object. The self parameter is a reference to "self", that is, the methods' host object.

Primitives of the syntax

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	object with n methods labelled $l_1, \ldots, l_n$ (distinct)
$o.l$	invocation of method l of object o
$o.l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

- Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$, a component $l_i = \varsigma(x_i)b_i$ of o consists of a label l_i and a method $\varsigma(x_i)b_i$.
- The letter ς in the method acts as a binder for the self parameter of the object. The self parameter is a reference to "self", that is, the methods' host object.
- When the body b of a method $\varsigma(y)b$ does not use its self parameter y, we refer to the method as a field.

Primitives of the semantics

Two operations : method invocation (1) and method update (2)

Primitives of the semantics

Two operations : method invocation (1) and method update (2)

(Notation: $a \rightsquigarrow b$ means that the term a reduces to b in one step.)

Primitives of the semantics

Two operations : method invocation (1) and method update (2)

(Notation: $a \rightsquigarrow b$ means that the term a reduces to b in one step.)

Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ and $j \in \{1 \ldots n\}$:

Primitives of the semantics

Two operations : method invocation (1) and method update (2)

(Notation: $a \rightsquigarrow b$ means that the term a reduces to b in one step.)

Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ and $j \in \{1 \ldots n\}$:

(1) $o.l_j \rightsquigarrow b_j[o/x_j]$

Primitives of the semantics

Two operations : method invocation (1) and method update (2)

(Notation: $a \leadsto b$ means that the term a reduces to b in one step.)

Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ and $j \in \{1 \ldots n\}$:

(1) $o.l_j \leadsto b_j[o/x_j]$

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

Primitives of the semantics

Two operations : method invocation (1) and method update (2)

(Notation: $a \rightsquigarrow b$ means that the term a reduces to b in one step.)

Given an object $o = [l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ and $j \in \{1 \ldots n\}$:

(1) $o.l_j \rightsquigarrow b_j[o/x_j]$

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

In the case of fields, method invocation and method update are referred as field selection and field update respectively.

Example of a storage cell

$$\begin{aligned} myCell \;\triangleq\; & [\; contents = 0, \\ & \; get = \varsigma(s)s.contents, \\ & \; set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n)) \;] \end{aligned}$$

Example of a storage cell

$$
\begin{aligned}
myCell \triangleq \quad & [\, contents = 0, \\
& get = \varsigma(s)s.contents, \\
& set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))\,]
\end{aligned}
$$

Then $myCell.get \rightsquigarrow myCell.contents \rightsquigarrow 0$,

Example of a storage cell

$$
\begin{aligned}
myCell \triangleq \quad & [\ contents = 0, \\
& \ get = \varsigma(s)s.contents, \\
& \ set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))\ ]
\end{aligned}
$$

Then $myCell.get \rightsquigarrow myCell.contents \rightsquigarrow 0$,

$myCell.set(5).get = ((myCell.set)(5)).get$

Example of a storage cell

$$\begin{aligned} myCell \triangleq\ & [\ contents = 0, \\ & get = \varsigma(s)s.contents, \\ & set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

Then $myCell.get \rightsquigarrow myCell.contents \rightsquigarrow 0$,

$myCell.set(5).get = ((myCell.set)(5)).get$
$\rightsquigarrow ((\lambda(n)(myCell.contents \Leftarrow n))(5)).get$

Example of a storage cell

$$
\begin{aligned}
myCell \triangleq \ & [\ contents = 0, \\
& \ get = \varsigma(s)s.contents, \\
& \ set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))\ ]
\end{aligned}
$$

Then $myCell.get \rightsquigarrow myCell.contents \rightsquigarrow 0$,

$myCell.set(5).get = ((myCell.set)(5)).get$
$\rightsquigarrow ((\lambda(n)(myCell.contents \Leftarrow n))(5)).get$
$\rightsquigarrow (myCell.contents \Leftarrow 5).get$

Example of a storage cell

$$
\begin{aligned}
myCell \triangleq\ & [\ contents = 0, \\
& get = \varsigma(s)s.contents, \\
& set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))\ ]
\end{aligned}
$$

Then $myCell.get \rightsquigarrow myCell.contents \rightsquigarrow 0$,

$myCell.set(5).get = ((myCell.set)(5)).get$
$\rightsquigarrow ((\lambda(n)(myCell.contents \Leftarrow n))(5)).get$
$\rightsquigarrow (myCell.contents \Leftarrow 5).get$
$\rightsquigarrow [contents = 5,$
$\quad get = \varsigma(s)s.contents,\ set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))].get$

Example of a storage cell

$$\begin{aligned} \mathit{myCell} \;\triangleq\; & [\; \mathit{contents} = 0, \\ & \;\mathit{get} = \varsigma(s)s.\mathit{contents}, \\ & \;\mathit{set} = \varsigma(s)(\lambda(n)(s.\mathit{contents} \Leftarrow n))\;] \end{aligned}$$

Then $\mathit{myCell}.\mathit{get} \rightsquigarrow \mathit{myCell}.\mathit{contents} \rightsquigarrow 0$,

$\mathit{myCell}.\mathit{set}(5).\mathit{get} = ((\mathit{myCell}.\mathit{set})(5)).\mathit{get}$
$\rightsquigarrow ((\lambda(n)(\mathit{myCell}.\mathit{contents} \Leftarrow n))(5)).\mathit{get}$
$\rightsquigarrow (\mathit{myCell}.\mathit{contents} \Leftarrow 5).\mathit{get}$
$\rightsquigarrow [\mathit{contents} = 5,$
$\quad \mathit{get} = \varsigma(s)s.\mathit{contents},\; \mathit{set} = \varsigma(s)(\lambda(n)(s.\mathit{contents} \Leftarrow n))].\mathit{get}$
$\rightsquigarrow [\mathit{contents} = 5,$
$\quad \mathit{get} = \varsigma(s)s.\mathit{contents},\; \mathit{set} = \varsigma(s)(\lambda(n)(s.\mathit{contents} \Leftarrow n))].\mathit{contents}$

Example of a storage cell

$$myCell \triangleq [\ contents = 0,\ get = \varsigma(s)s.contents,\ set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))\ ]$$

Then $myCell.get \rightsquigarrow myCell.contents \rightsquigarrow 0$,

$$\begin{aligned}
&myCell.set(5).get = ((myCell.set)(5)).get \\
&\rightsquigarrow ((\lambda(n)(myCell.contents \Leftarrow n))(5)).get \\
&\rightsquigarrow (myCell.contents \Leftarrow 5).get \\
&\rightsquigarrow [contents = 5, \\
&\quad get = \varsigma(s)s.contents,\ set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))].get \\
&\rightsquigarrow [contents = 5, \\
&\quad get = \varsigma(s)s.contents,\ set = \varsigma(s)(\lambda(n)(s.contents \Leftarrow n))].contents \\
&\rightsquigarrow 5
\end{aligned}$$

Example of Movable points

$$origin_1 \triangleq [\ x = 0,\ \ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)\ ]$$

$$\begin{aligned} origin_2 \triangleq [\ x = 0,\ \ y = 0,\ \ & mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx), \\ & mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ] \end{aligned}$$

Example of Movable points

$$origin_1 \triangleq [\ x = 0,\ \ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)\ ]$$

$$\begin{aligned} origin_2 \triangleq [\ x = 0,\ \ y = 0,\ \ & mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx), \\ & mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ] \end{aligned}$$

In this case, all operations possible on $origin_1$ are also possible on $origin_2$.

Example of Movable points

$$origin_1 \triangleq [\ x = 0,\ \ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)\ ]$$

$$\begin{aligned} origin_2 \triangleq [\ x = 0,\ \ y = 0,\ \ & mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx), \\ & mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ] \end{aligned}$$

In this case, all operations possible on $origin_1$ are also possible on $origin_2$.

That is, we need a typed system in which any context expecting an $origin_1$ object can also accept an $origin_2$ object.

Example of Movable points

$$origin_1 \triangleq [\ x = 0,\ \ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)\ ]$$

$$\begin{aligned} origin_2 \triangleq [\ x = 0,\ \ y = 0,\ \ & mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx), \\ & mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ] \end{aligned}$$

In this case, all operations possible on $origin_1$ are also possible on $origin_2$.

That is, we need a typed system in which any context expecting an $origin_1$ object can also accept an $origin_2$ object.

$\rightarrow$ Subtyping

The first-order ς-calculus

Formal syntax for first-order theory

A, B, C, D ::=	Types
Top	the biggest type
$[\, l_i{:}B_i \quad ^{i \in 1 \ldots n}]$	object (l_i distinct)
$A \rightarrow B$	function type

Formal syntax for first-order theory

$A, B, C, D \ ::=$	Types
Top	the biggest type
$[\ l_i{:}B_i \quad ^{i \in 1 \ldots n}]$	object (l_i distinct)
$A \to B$	function type
$a, b, c, d \ ::=$	Terms
x	variable
$[l_i = \varsigma(x_i{:}A_i)b_i \ ^{i \in 1 \ldots n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \varsigma(x{:}A)b$	method update
$\lambda(x{:}A)b$	function
$b(a)$	application

Scoping for the first-order calculus

$$
\begin{aligned}
FV(x) &\triangleq \{x\} \\
FV(\varsigma(x{:}A)b) &\triangleq FV(b) - \{x\} \\
FV([l_i = \varsigma(x_i{:}A_i)b_i \ ^{i \in 1 \dots n}]) &\triangleq \bigcup_{i=1}^{n} FV(\varsigma(x_i{:}A_i)b_i) \\
FV(a.l) &\triangleq FV(a) \\
FV(a.l \Leftarrow \varsigma(x{:}A)b) &\triangleq FV(a) \cup FV(\varsigma(x{:}A)b) \\
FV(\lambda(x{:}A)b) &\triangleq FV(b) - \{x\} \\
FV(b(a)) &\triangleq FV(b) \cup FV(a)
\end{aligned}
$$

Substitution : $a[b/x]$ is the term a in which all free occurrences of x are substituted for b.

Formal system and judgments

We present a formal system for deriving judgments of the form $E \vdash \mathcal{T}$ where E is an environment and $\mathcal{T}$ is an assertion whose shape depends on the judgment.

- An *environment* E is a list of assumptions for variables, of the form $x_1 : A_1, \ldots, x_n : A_n$
- $\emptyset$ stands for the empty environment
- The judgment $E \vdash \diamond$ means that the environment E is well-formed
- The judgment $E \vdash A$ for a type A means that A is a well-formed type in the environment E.
- The judgment $E \vdash b : B$ is value typing judgment, stating that the term b has type B in E

Formation rules fragments

Assertions describing how to form well-typed objects and functions :

- Δ_x : environments and term variables
- Δ_K : ground types
- Δ_{Ob} : objects
- $\Delta_{\rightarrow}$: functions
- $\Delta_{<:}$: subtyping and subsumption
- $\Delta_{<:Ob}$: objects subtyping
- $\Delta_{<:\rightarrow}$: functions subtyping

Standard First-Order Fragments

Δ_x : environments and term variables

(Env $\emptyset$)

$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)

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

(Val x)

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

Standard First-Order Fragments

Δ_x : environments and term variables

(Env ∅)

$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)

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

(Val x)

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

Δ_K : ground types

(Type Const)

$$\frac{E \vdash \diamond}{E \vdash K}$$

Object Fragment

Δ_{Ob} : building objects and object types

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1 \ldots n \qquad (l_i \text{ distinct})}{E \vdash [l_i{:}B_i \ ^{i \in 1 \ldots n}]}$$

Object Fragment

Δ_{Ob} : building objects and object types

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1 \ldots n \qquad (l_i \text{ distinct})}{E \vdash [l_i{:}B_i \ {}^{i \in 1 \ldots n}]}$$

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

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

Object Fragment

Δ_{Ob} : building objects and object types

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1 \ldots n \qquad (l_i \text{ distinct})}{E \vdash [l_i{:}B_i \ ^{i \in 1 \ldots n}]}$$

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

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

(Val Select)

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

Object Fragment

Δ_{Ob} : building objects and object types

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1 \ldots n \qquad (l_i \text{ distinct})}{E \vdash [l_i{:}B_i \ ^{i \in 1 \ldots n}]}$$

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

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

(Val Select)

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

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

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

Function Fragment

Abstraction and application mechanisms are used as primitives of the calculus.

$\Delta_{\rightarrow}$: function types

(Type Arrow)

$$\frac{E \vdash A \quad E \vdash B}{E \vdash A \rightarrow B}$$

(Val Fun)

$$\frac{E, x{:}A \vdash b{:}B}{E \vdash \lambda(x{:}A)b : A \rightarrow B}$$

(Val Appl)

$$\frac{E \vdash b{:}A \rightarrow B \quad E \vdash a{:}A}{E \vdash b(a){:}A}$$

Subtyping

Subtyping fragment

$\Delta_{<:}$: subtypes ("$A <: B$" means A is a subtype of B)

(Sub Refl)

$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)

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

(Val Subsumption)

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

(Type Top)

$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Sub Top)

$$\frac{E \vdash A}{E \vdash A <: Top}$$

Object and function subtypes

$\Delta_{<:Ob}$: Object subtypes

(Sub Object)

$$\frac{E \vdash B_i \quad \forall i \in 1 \ldots n+m \qquad (l_i \text{ distinct})}{E \vdash [l_i{:}B_i \ ^{i \in 1 \ldots n+m}] <: [l_i{:}B_i \ ^{i \in 1 \ldots n}]}$$

Object and function subtypes

$\Delta_{<:Ob}$: Object subtypes

(Sub Object)

$$\frac{E \vdash B_i \quad \forall i \in 1 \ldots n+m \qquad (l_i \text{ distinct})}{E \vdash [l_i{:}B_i \;^{i \in 1 \ldots n+m}] <: [l_i{:}B_i \;^{i \in 1 \ldots n}]}$$

$\Delta_{<:\rightarrow}$: Function subtypes

(Sub Arrow)

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

(contravariant in the domain, covariant in the codomain)

Example of typed Movable points

$$A \triangleq [\ x : Real,\ mv_x : Real \to A\ ]$$

$$B \triangleq [\ x : Real,\ y : Real,\ mv_x : Real \to B,\ mv_y : Real \to B\ ]$$

$$origin_1 \triangleq [x = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)] : A$$

$$origin_2 \triangleq [x = 0,\ y = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx),\ mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)] : B$$

Movable points

$$A \triangleq [\ x : Real,\ mv_x : Real \rightarrow A\ ]$$
$$B \triangleq [\ x : Real,\ y : Real,\ mv_x : Real \rightarrow B,\ mv_y : Real \rightarrow B\ ]$$

$$origin_1 \triangleq [x = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)] : A$$
$$origin_2 \triangleq [x = 0,\ y = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx),\ mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ]\ :\ B$$

Movable points

$$A \triangleq [\ x : Real,\ mv_x : Real \to A\ ]$$
$$B \triangleq [\ x : Real,\ y : Real,\ mv_x : Real \to B,\ mv_y : Real \to B\ ]$$

$$origin_1 \triangleq [x = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)] : A$$
$$origin_2 \triangleq [x = 0,\ y = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx),\ mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ]\ :\ B$$

- We have $B <: A$. Thus, by subsumption, $origin2 : A$.

Movable points

$$A \triangleq [\ x : Real,\ mv_x : Real \to A\ ]$$
$$B \triangleq [\ x : Real,\ y : Real,\ mv_x : Real \to B,\ mv_y : Real \to B\ ]$$

$$origin_1 \triangleq [x = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)] : A$$
$$origin_2 \triangleq [x = 0,\ y = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx),\ mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ]\ :\ B$$

- We have $B <: A$. Thus, by subsumption, $origin2 : A$.
- $origin1 \leftrightarrow origin2 : A$, but not $origin1 \leftrightarrow origin2 : B$

Movable points

$$A \triangleq [\ x : Real,\ mv_x : Real \rightarrow A\ ]$$
$$B \triangleq [\ x : Real,\ y : Real,\ mv_x : Real \rightarrow B,\ mv_y : Real \rightarrow B\ ]$$

$$origin_1 \triangleq [x = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx)] : A$$
$$origin_2 \triangleq [x = 0,\ y = 0,\ mv_x = \varsigma(s)\lambda(dx)(s.x \Leftarrow s.x + dx),$$
$$mv_y = \varsigma(s)\lambda(dy)(s.y \Leftarrow s.y + dy)\ ]\ :\ B$$

- We have $B <: A$. Thus, by subsumption, $origin2 : A$.
- $origin1 \leftrightarrow origin2 : A$, but not $origin1 \leftrightarrow origin2 : B$
- Once $origin2$ is subsumed to the type A, we cannot invoke the methods mv_y or y on $origin_2$.

  Example : if $B \triangleq [x{:}Real,\ y{:}Real,\ mv_x{:}Real \rightarrow B,\ mv_y{:}Real \rightarrow A\ ]$, then $origin_2.mv_y(4)$ has type A, which means that I cannot derive $(origin_2.mv_y(4)).mv_y$ as a typed value in the formal system.

Cells and encapsulation

A *RomCell* is a storage cell that can only be read.

Cells and encapsulation

A *RomCell* is a storage cell that can only be read.
A *PromCell* can be written once, and then becomes a *RomCell*.

Cells and encapsulation

A *RomCell* is a storage cell that can only be read.
A *PromCell* can be written once, and then becomes a *RomCell*.
A *PrivateCell* is used for operating on a *contents* field, that is hidden in the previous cell types.

Cells and encapsulation

A *RomCell* is a storage cell that can only be read.
A *PromCell* can be written once, and then becomes a *RomCell*.
A *PrivateCell* is used for operating on a *contents* field, that is hidden in the previous cell types.

Let's consider storage for natural numbers (*Nat*) :

$$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \to RomCell\ ]$$
$$PromCell \triangleq [\ get : Nat,\ set : Nat \to RomCell\ ]$$
$$RomCell \triangleq [get : Nat]$$

Cells and encapsulation

A *RomCell* is a storage cell that can only be read.
A *PromCell* can be written once, and then becomes a *RomCell*.
A *PrivateCell* is used for operating on a *contents* field, that is hidden in the previous cell types.

Let's consider storage for natural numbers (Nat) :

$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \to RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ set : Nat \to RomCell\ ]$
$RomCell \triangleq [get : Nat]$

We get $PrivateCell <: PromCell <: RomCell$.

Cells and encapsulation

$$PrivateCell \triangleq [\ contents : Nat,\ \ get : Nat,\ \ set : Nat \to RomCell\ ]$$
$$PromCell \triangleq [\ get : Nat,\ \ set : Nat \to RomCell\ ]$$
$$RomCell \triangleq [get : Nat]$$

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ \ get : Nat,\ \ set : Nat \to RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ \ set : Nat \to RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$\begin{aligned} myCell : PrivateCell \ \triangleq \quad & [contents = 0, \\ & get = \varsigma(s : PrivateCell)s.contents, \\ & set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

We can use subsumption to get *myCell*:*PromCell* and hide *contents*.

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ \ get : Nat,\ \ set : Nat \rightarrow RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ \ set : Nat \rightarrow RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$\begin{aligned} myCell : PrivateCell \ \ \triangleq \quad & [contents = 0, \\ & get = \varsigma(s : PrivateCell)s.contents, \\ & set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

We can use subsumption to get *myCell*:*PromCell* and hide *contents*.

The subtyping *PrivateCell* <: *RomCell* is required to type the *set* method because its body has a *PrivateCell* method, whereas the expected return type is *RomCell*.

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$myCell : PrivateCell \triangleq \begin{array}{l}[contents = 0,\\ get = \varsigma(s : PrivateCell)s.contents,\\ set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ]\end{array}$$

We can use subsumption to get $myCell{:}PromCell$ and hide *contents*.

The subtyping $PrivateCell <: RomCell$ is required to type the *set* method because its body has a *PrivateCell* method, whereas the expected return type is *RomCell*.

$s : PrivateCell \vdash \lambda(n : Nat)(s.contents \Leftarrow n) : Nat \rightarrow PrivateCell <:$
$Nat \rightarrow RomCell$ by covariance on codomain types

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$\begin{aligned} myCell : PrivateCell \ \triangleq \quad & [contents = 0, \\ & get = \varsigma(s : PrivateCell)s.contents, \\ & set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

Now, we can show that :

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$\begin{aligned} myCell : PrivateCell \ \triangleq \quad & [contents = 0, \\ & get = \varsigma(s : PrivateCell)s.contents, \\ & set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

Now, we can show that :

$myCell.set : Nat \rightarrow RomCell$

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \to RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ set : Nat \to RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$\begin{aligned} myCell : PrivateCell \triangleq \quad & [contents = 0, \\ & get = \varsigma(s : PrivateCell)s.contents, \\ & set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

Now, we can show that :

$myCell.set : Nat \to RomCell$
$myCell.set(3) : RomCell$
(can't write $myCell.set(3).contents$ or $myCell.set(3).set$)

Cells and encapsulation

$PrivateCell \triangleq [\ contents : Nat,\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$PromCell \triangleq [\ get : Nat,\ set : Nat \rightarrow RomCell\ ]$
$RomCell \triangleq [get : Nat]$

$$\begin{aligned} myCell : PrivateCell \ \triangleq \quad & [contents = 0, \\ & get = \varsigma(s : PrivateCell)s.contents, \\ & set = \varsigma(s : PrivateCell)(\lambda(n)(s.contents \Leftarrow n))\ ] \end{aligned}$$

Now, we can show that :

$myCell.set : Nat \rightarrow RomCell$
$myCell.set(3) : RomCell$
(can't write $myCell.set(3).contents$ or $myCell.set(3).set$)
$myCell.set(3).get : Nat \ \leadsto\ 3 : Nat$

Classes and inheritance

Classes

- Given an object type $A \equiv [\ l_i{:}B_i\ ^{i \in 1 \ldots n}]$:

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

  is the type of classes generating objects of type A.

Classes

- Given an object type $A \equiv [\ l_i{:}B_i\ ^{i \in 1\ldots n}]$:

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

is the type of classes generating objects of type A.

- These classes are objects of the form :

$$[\ new = \varsigma(z{:}Class(A))([l_i = \varsigma(s{:}A)z.l_i(s)^{i \in 1\ldots n}]), \\ l_i = \lambda(s{:}A)b_i\ ^{i \in 1\ldots n}\ ].$$

Classes

- Given an object type $A \equiv [\ l_i{:}B_i\ ^{i \in 1 \ldots n}]$:

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

  is the type of classes generating objects of type A.

- These classes are objects of the form :

  $$[\ new = \varsigma(z{:}Class(A))([l_i = \varsigma(s{:}A)z.l_i(s)^{i \in 1 \ldots n}]), \\ l_i = \lambda(s{:}A)b_i\ ^{i \in 1 \ldots n}\ ].$$

- The methods of a class are called pre-methods, whose λ binders are meant to be replaced with ς binders, that will link an instantiated object to its methods.

Inheritance

Inheritance is the re-use of pre-methods from a class within another class.

Inheritance

Inheritance is the re-use of pre-methods from a class within another class.

An ad-hoc criteria for re-use :

$Class(A')$ may inherit from $Class(A)$ iff $A' <: A$

Inheritance

For example, consider $A' \equiv [l_i{:}B_i \ ^{i \in 1 \ldots n+m}] <: A \equiv [l_i{:}B_i \ ^{i \in 1 \ldots n}]$ and the following classes :

Inheritance

For example, consider $A' \equiv [l_i{:}B_i \ ^{i \in 1 \ldots n+m}] <: A \equiv [l_i{:}B_i \ ^{i \in 1 \ldots n}]$ and the following classes :

$$\begin{aligned} c : Class(A) \triangleq [\ & new = \varsigma(z{:}Class(A))([l_i = \varsigma(s{:}A)z.l_i(s)^{i \in 1 \ldots n}]), \\ & l_i = \lambda(s{:}A)b_i \ ^{i \in 1 \ldots n}\ ] \end{aligned}$$

$$\begin{aligned} c' : Class(A') \triangleq [\ & new = \varsigma(z{:}Class(A'))([l_1 = \varsigma(s{:}A')z.l_i(s)^{i \in 1 \ldots n+m}]), \\ & l_1 = \lambda(s{:}A')b_1', \\ & l_j = c.l_j \ ^{j \ \in 2 \ldots n}, \\ & l_k = \lambda(s{:}A')b_k \ ^{k \ \in n+1 \ldots n+m}\ ] \end{aligned}$$

Inheritance

For example, consider $A' \equiv [l_i{:}B_i\ ^{i \in 1 \ldots n+m}] <: A \equiv [l_i{:}B_i\ ^{i \in 1 \ldots n}]$ and the following classes :

$$\begin{aligned} c : Class(A) \triangleq [\ & new = \varsigma(z{:}Class(A))([l_i = \varsigma(s{:}A)z.l_i(s)^{i \in 1 \ldots n}]), \\ & l_i = \lambda(s{:}A)b_i\ ^{i \in 1 \ldots n}\ ] \end{aligned}$$

$$\begin{aligned} c' : Class(A') \triangleq [\ & new = \varsigma(z{:}Class(A'))([l_1 = \varsigma(s{:}A')z.l_i(s)^{i \in 1 \ldots n+m}]), \\ & l_1 = \lambda(s{:}A')b'_1, \\ & l_j = c.l_j\ ^{j\ \in 2 \ldots n}, \\ & l_k = \lambda(s{:}A')b_k\ ^{k\ \in n+1 \ldots n+m}\ ] \end{aligned}$$

- c' overrides the pre-method l_1

Inheritance

For example, consider $A' \equiv [l_i{:}B_i \ ^{i \in 1 \ldots n+m}] <: A \equiv [l_i{:}B_i \ ^{i \in 1 \ldots n}]$ and the following classes :

$$
\begin{aligned}
c : Class(A) \triangleq [\ & new = \varsigma(z{:}Class(A))([l_i = \varsigma(s{:}A)z.l_i(s)^{i \in 1 \ldots n}]),\\
& l_i = \lambda(s{:}A)b_i \ ^{i \in 1 \ldots n} \ ]
\end{aligned}
$$

$$
\begin{aligned}
c' : Class(A') \triangleq [\ & new = \varsigma(z{:}Class(A'))([l_1 = \varsigma(s{:}A')z.l_i(s)^{i \in 1 \ldots n+m}]),\\
& l_1 = \lambda(s{:}A')b_1',\\
& l_j = c.l_j \ ^{j \ \in 2 \ldots n},\\
& l_k = \lambda(s{:}A')b_k \ ^{k \ \in n+1 \ldots n+m} \ ]
\end{aligned}
$$

- c' overrides the pre-method l_1
- c' inherits the pre-methods l_j with $j \in 2 \ldots n$ from c ($c.l_j : A \rightarrow B <: A' \rightarrow B$ by contravariance on $A' <: A$).

Inheritance

For example, consider $A' \equiv [l_i{:}B_i\ ^{i \in 1 \ldots n+m}] <: A \equiv [l_i{:}B_i\ ^{i \in 1 \ldots n}]$ and the following classes :

$$\begin{aligned} c : Class(A) \triangleq [\ & new = \varsigma(z{:}Class(A))([l_i = \varsigma(s{:}A)z.l_i(s)^{i \in 1 \ldots n}]), \\ & l_i = \lambda(s{:}A)b_i\ ^{i \in 1 \ldots n}\ ] \end{aligned}$$

$$\begin{aligned} c' : Class(A') \triangleq [\ & new = \varsigma(z{:}Class(A'))([l_1 = \varsigma(s{:}A')z.l_i(s)^{i \in 1 \ldots n+m}]), \\ & l_1 = \lambda(s{:}A')b'_1, \\ & l_j = c.l_j\ ^{j\ \in 2 \ldots n}, \\ & l_k = \lambda(s{:}A')b_k\ ^{k\ \in n+1 \ldots n+m}\ ] \end{aligned}$$

- c' overrides the pre-method l_1
- c' inherits the pre-methods l_j with $j \in 2 \ldots n$ from c ($c.l_j : A \to B <: A' \to B$ by contravariance on $A' <: A$).
- c' has some new pre-methods of it's own $k \in n+1 \ldots n+m$

Polymorphism

A class that instantiates polymorphic cells

Private cell type with a dependency on a type variable X :

A class that instantiates polymorphic cells

Private cell type with a dependency on a type variable X :

$$PrivateCell\{X\} \triangleq [\ contents : X,\ get : X,\ set : X \rightarrow RomCell\ ]$$

A class that instantiates polymorphic cells

Private cell type with a dependency on a type variable X :

$$PrivateCell\{X\} \triangleq [\ contents : X,\ get : X,\ set : X \rightarrow RomCell\ ]$$

Abstraction on the type : $\forall(X)PrivateCell\{X\}$

A class that instantiates polymorphic cells

Private cell type with a dependency on a type variable X :

$$PrivateCell\{X\} \triangleq [\ contents : X,\ get : X,\ set : X \to RomCell\ ]$$

Abstraction on the type : $\forall(X)PrivateCell\{X\}$

Polymorphic class type for cells : $Class(\forall(X)PrivateCell\{X\}) \triangleq$

$$\begin{aligned}[\ & new : \forall(X)PrivateCell\{X\}, \\ & contents : \forall(X)(PrivateCell\{X\} \to X), \\ & get : \forall(X)(PrivateCell\{X\} \to X), \\ & set : \forall(X)(PrivateCell\{X\} \to X \to RomCell\{X\})\ ]\end{aligned}$$

A class that instantiates polymorphic cells

Polymorphic class for cells : $c : \mathit{Class}(\forall(X)\mathit{PrivateCell}\{X\}) \triangleq$

A class that instantiates polymorphic cells

Polymorphic class for cells : $c : Class(\forall(X)PrivateCell\{X\}) \triangleq$

$$
\begin{aligned}
&[\ new = \varsigma(z : Class(\forall(X)PrivateCell\{X\}))\\
&\qquad\qquad \lambda(X)(\ [\ contents = \varsigma(s : PrivateCell\{X\})z.contents(X)(s),\\
&\qquad\qquad\qquad\quad get = \varsigma(s : PrivateCell\{X\})z.get(X)(s),\\
&\qquad\qquad\qquad\quad set = \varsigma(s : PrivateCell\{X\})z.set(X)(s)\ ]\ ),\\
&contents = \lambda(X)(\lambda(s{:}PrivateCell\{X\})s.contents),\\
&get = \lambda(X)(\lambda(s{:}PrivateCell\{X\})s.contents),\\
&set = \lambda(X)(\lambda(s{:}PrivateCell\{X\})(\lambda(x)(s.contents \Leftarrow x)))\ ]
\end{aligned}
$$

A class that instantiates polymorphic cells

Polymorphic class for cells : $c : Class(\forall(X)PrivateCell\{X\}) \triangleq$

$$
\begin{aligned}
[\ new = \varsigma(&z : Class(\forall(X)PrivateCell\{X\})) \\
&\lambda(X)(\ [\ contents = \varsigma(s : PrivateCell\{X\})z.contents(X)(s), \\
&\qquad get = \varsigma(s : PrivateCell\{X\})z.get(X)(s), \\
&\qquad set = \varsigma(s : PrivateCell\{X\})z.set(X)(s)\ ]\ ), \\
contents &= \lambda(X)(\lambda(s{:}PrivateCell\{X\})s.contents), \\
get &= \lambda(X)(\lambda(s{:}PrivateCell\{X\})s.contents), \\
set &= \lambda(X)(\lambda(s{:}PrivateCell\{X\})(\lambda(x)(s.contents \Leftarrow x)))\ ]
\end{aligned}
$$

Finally $c.new(Nat) : PrivateCell\{Nat\} = [\ contents : Nat,\ get : Nat,\ set : Nat \to RomCell\ ]$ is a private cell that stores natural numbers as desired.

Conclusion

Conclusion

- The theory of objects is not recognized as having a foundational basis as of now

Conclusion

- The theory of objects is not recognized as having a foundational basis as of now
- Yet, through the study of ς-calculus, we saw that such foundations can be provided (at least, some foundation exists)

Conclusion

- The theory of objects is not recognized as having a foundational basis as of now
- Yet, through the study of ς-calculus, we saw that such foundations can be provided (at least, some foundation exists)
- Through the notion of “self” of this calculus, objects have acquired an expressive and robust typing system.

Conclusion

- The theory of objects is not recognized as having a foundational basis as of now
- Yet, through the study of ς-calculus, we saw that such foundations can be provided (at least, some foundation exists)
- Through the notion of "self" of this calculus, objects have acquired an expressive and robust typing system.
- Through subtyping alone, the four OOL principles of abstraction, encapsulation, inheritance, and polymorphism have been properly enforced.

Relation to my research

Objects do not come equipped with elaborate forms of reduction strategies
$\Rightarrow$ no facility for the expression of concurrency

Relation to my research

Objects do not come equipped with elaborate forms of reduction strategies
$\Rightarrow$ no facility for the expression of concurrency

Should objects be rendered as processes, or processes rendered as objects?
Or is there an alternative solution?

Relation to my research

Objects do not come equipped with elaborate forms of reduction strategies
$\Rightarrow$ no facility for the expression of concurrency

Should objects be rendered as processes, or processes rendered as objects?
Or is there an alternative solution?

My intuition : investigation of the typing theory of processes, such as presented in π-calculus, may allow a representation of objects that is faithful with respect to subtyping and the notion of "self";
that would mean endowing objects with a natural form of concurrency.

Relation to my research

In parallel, I am investigating a model of my own that represents labelled transition systems through localized functions that fire recurrently.

The firing of these localized functions resembles the firing of transitions in a Petri net; but in my model, the state of the system is not represented by tokens in a region, but by values that fall under certain types associated to regions...

Relation to my research

In parallel, I am investigating a model of my own that represents labelled transition systems through localized functions that fire recurrently.

The firing of these localized functions resembles the firing of transitions in a Petri net; but in my model, the state of the system is not represented by tokens in a region, but by values that fall under certain types associated to regions...

A circuit made of logic gates operating on boolean types would be a good example. The state of the system would be represented by values that circulate in the wires. I would model logic gates as methods that fire to affect the state of the system locally on the wires.

My belief is that, perhaps, encapsulation, abstraction and subtyping can be rendered through typing and localization in my model...