Assembling Objects with Internal Concurrency

Didier Rémy

INRIA-Rocquencourt

Joint work with $\begin{cases} \textit{Cédric Fournet,} \\ \textit{Cosimo Laneve}, \text{ and} \\ \textit{Luc Maranget} \end{cases}$

September 1999

Design a concurrent, object-oriented calculus with:

- **object internal concurrency**

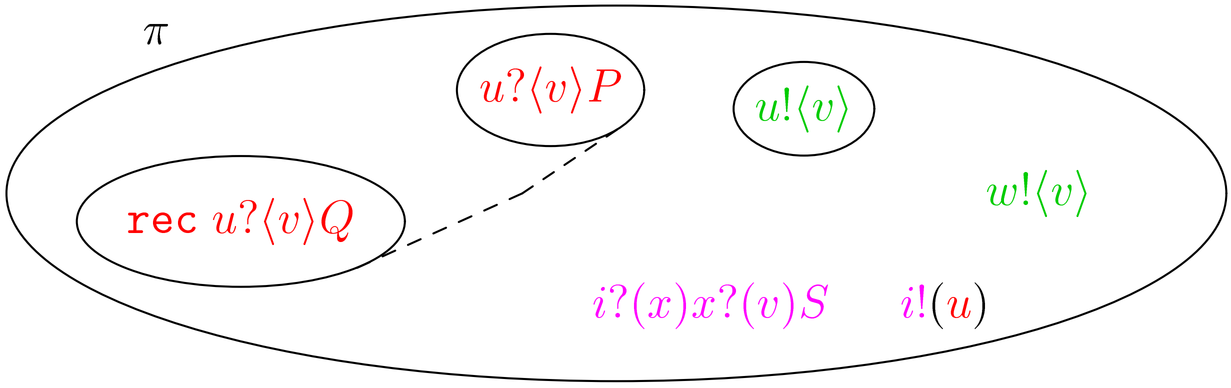
  —*not just sequential objects running concurrently!*

- **incremental assembling of object definitions**

  including the refinement of both

  - behavior — *traditional in object-orientation*
  - synchronization — *expected in a concurrent setting*

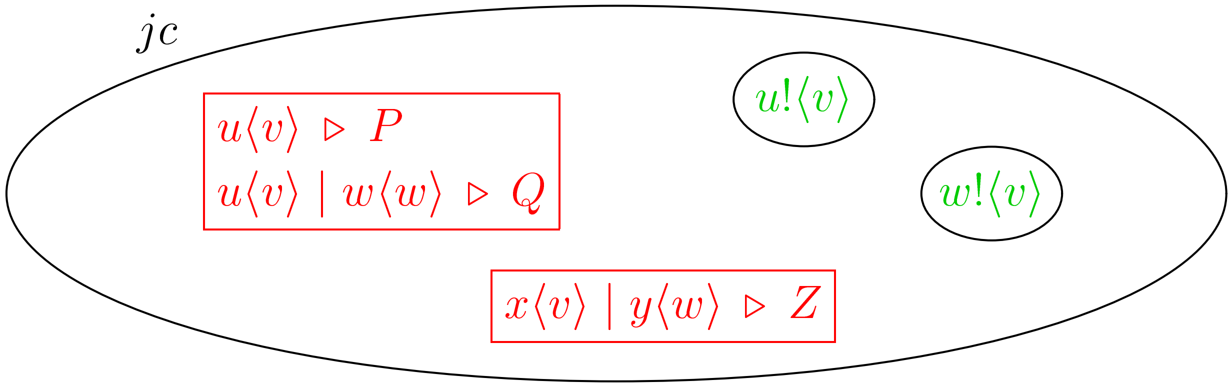
The π-calculus



Communication is difficult in a distributed setting, since it involves 3 sites: the sender, the receiver and the channel's location.

Reasoning is difficult, since the behavior attached to a name is not statically known.

The join-calculus



Derived from the pi-calculus for easier distribution

Name scoping is more constrained than in pi-calculus

- Behavior of a name is statically defined and lexically scoped.
- Synchronizing names are jointly defined.

The join-calculus

A join definition creates a set of names and a set of reaction rules that describe how to process messages sent to those names.

$$
\begin{array}{ll}
\texttt{def} & \boxed{\begin{array}{lclcl} & \texttt{read}\langle\texttt{r}\rangle & \mid \texttt{REF}\langle\texttt{v}\rangle & \triangleright & \texttt{r}\langle\texttt{v}\rangle \mid \texttt{REF}\langle\texttt{v}\rangle \\ \wedge & \texttt{write}\langle\texttt{x,r}\rangle & \mid \texttt{REF}\langle\texttt{v}\rangle & \triangleright & \texttt{r}\langle\rangle \mid \texttt{REF}\langle\texttt{x}\rangle \end{array}} \\
\texttt{in} & \\
& \texttt{REF}\langle 0\rangle \mid \ldots\ \texttt{do\_something}\langle\texttt{write, read}\rangle\ \ldots \mid \texttt{write}\langle 2,\texttt{k}\rangle
\end{array}
$$

A reaction, intuitively:

$$\texttt{write}\langle 2,\texttt{k}\rangle \mid \texttt{REF}\langle 0\rangle \longrightarrow \texttt{k}\langle\rangle \mid \texttt{REF}\langle 2\rangle$$

Objects

The join-calculus has an obvious object-oriented flavor:

$$\text{names} \Longleftrightarrow \text{methods}$$

However, all methods must be passed to clients, separately.

Use record-like structure to group all names of a definition as a single value.

Yet more rigidity!

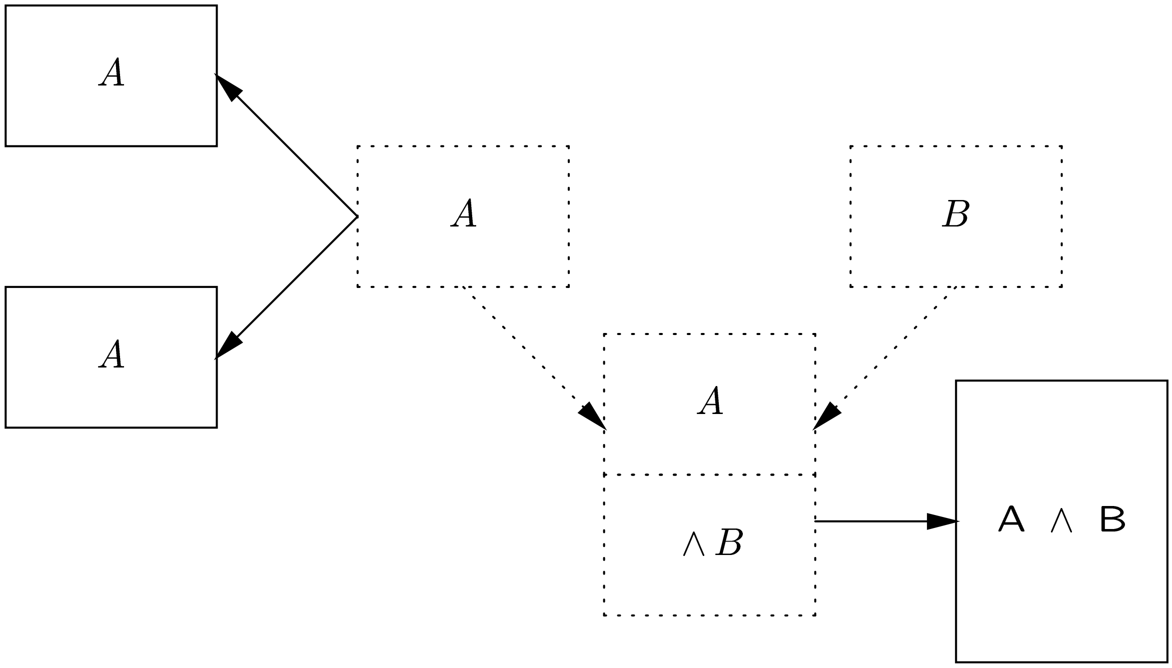
Classes

Join-definitions cannot be assembled incrementally, anymore.

Use classes as extendible join-definitions.

Bring back some flexibility!

Open classes and closed objects



Objects

```
obj cell =  self (z)
               read ⟨r⟩    | REF⟨v⟩  ▷  r.reply⟨v⟩ | z.REF⟨v⟩
            ∧  write⟨x,r⟩ | REF⟨v⟩  ▷  r.reply⟨⟩  | z.REF⟨x⟩
in
    cell.REF⟨0⟩ | ... do_something⟨cell⟩ ... | cell.write⟨2,k⟩
```

Translation

- Definitions are named
- Names became labels (of a definition)
- All methods are sent (and exposed) together under the object name.

Internal concurrency

Names are used for operations, STATE, and SYNCHRONIZATION.
Internal concurrency by construction (join-calculus):

```
obj pair =
    read_hd ⟨r⟩     | HEAD⟨v⟩  ▷  r.reply⟨v⟩ | HEAD⟨v⟩
  ∧ write_hd⟨x,r⟩ | HEAD⟨v⟩  ▷  r.reply⟨⟩  | HEAD⟨x⟩
  -----------------------------------------------
  ∧ read_tl ⟨r⟩     | TAIL⟨v⟩  ▷  r.reply⟨v⟩ | TAIL⟨v⟩
  ∧ write_tl⟨x,r⟩ | TAIL⟨v⟩  ▷  r.reply⟨⟩  | TAIL⟨x⟩
in
    pair.HEAD⟨1⟩ | pair.TAIL⟨2⟩ ...  | pair.write_hd⟨0,k⟩
```

The two pieces of state HEAD and TAIL are kept independent.
Primitive synchronization is expressive and safer, *but limited*.
More complex synchronization can always be programmed.

The client-server pattern

```
obj server = self (z)
   do_incr⟨x,r⟩ ▷  let v = read⟨_⟩ in x.write⟨1 + v, r⟩
       (* obj k = reply⟨v⟩ ▷ x.write⟨1 + v, r⟩ in x.read⟨k⟩ *)
∧ ...
in obj counter = self (z)
   read⟨r⟩     | REF⟨a⟩ ▷ r.reply⟨a⟩ | z.REF⟨a⟩
∧ write⟨b,r⟩ | REF⟨a⟩ ▷ r.reply⟨⟩  | z.REF⟨b⟩
∧ incr⟨r⟩ ▷ server.do_incr⟨z,r⟩
in ...
```

While the server computes do_incr, external calls to the method write of the client could interfere.

Locking write of the client would deadlock.

Well-known problem

Internal (server) calls should be allowed while external (client) calls should be locked.

A solution

Give the server privileged access to the internal view, and

Lock external methods during (cooperative) calls to the server.

Views

Views map methods names to actual method slots.

Catalogs map view names to views.


```
obj counter =
 self (z)
   read⟨r⟩    | REF⟨a⟩ | INT⟨⟩ ▷ r.reply⟨a⟩ | z.REF⟨a⟩ | z.INT⟨⟩
 ∧ w_int⟨b,r⟩ | REF⟨a⟩ | INT⟨⟩ ▷ r.reply⟨⟩ | z.REF⟨b⟩ | z.INT⟨⟩
 ------------------------------------------------------------------
 ∧ read⟨r⟩    | REF⟨a⟩ | EXT⟨⟩ ▷ r.reply⟨a⟩ | z.REF⟨a⟩ | z.EXT⟨⟩
 ∧ w_ext⟨b,r⟩ | REF⟨a⟩ | EXT⟨⟩ ▷ r.reply⟨⟩ | z.REF⟨b⟩ | z.EXT⟨⟩
 ------------------------------------------------------------------
 ∧ ext⟨⟩ | INT⟨⟩ ▷ EXT⟨⟩
 ∧ incr⟨r⟩ | EXT⟨⟩ ▷ z.INT⟨⟩ |
      (obj finish = reply⟨v⟩ ▷ r.reply⟨v⟩ | ext⟨⟩ in
       server.do_incr⟨internal z, finish⟩)
 ^[ read; write → w_ext; incr; REF; INT; EXT; ext  ]
 _[ internal : [read; write → w_int ] ]
in ...
```

Syntax

$x,\ z \in$ names $\qquad \ell \in$ labels

Processes	Definitions	Join patterns
$P ::= 0$	$D ::= 0$	$J ::= \ell\langle\widetilde{x}\rangle$
$\mid\ v.\ell\langle\widetilde{v}\rangle$	$\mid\ J \triangleright P$	$\mid\ J \mid J$
$\mid\ P \mid P$	$\mid\ D \wedge D$	
$\mid\ \text{obj}\ x = \texttt{self(z)}\ W\ \text{in}\ P$		

Values	Object values
$v ::= x$	$W ::= D$

Syntax

$x,\ z \in$ names $\qquad \ell \in$ labels

Processes	Definitions	Join patterns
$P ::= 0$	$D ::= 0$	$J ::= \ell\langle\widetilde{x}\rangle$
$\mid\ v.\ell\langle\widetilde{v}\rangle$	$\mid\ J \triangleright P$	$\mid\ J \mid J$
$\mid\ P \mid P$	$\mid\ D \wedge D$	
$\mid\ \text{obj}\ x = \texttt{self(z)}\ W\ \text{in}\ P$		

Values	Object values
$v ::= x$	$W ::= \nu(\widetilde{\varphi})\ D\,{}^{\Theta}_{\Gamma}$
$\mid\ \varphi v$	

$\varphi \in$ view names $\qquad \Theta \in$ views $\qquad \Gamma \in$ catalogs

$\nu(\widetilde{\varphi})$ $\genfrac{}{}{0pt}{}{\Theta}{\Gamma}$

$|$ φv

$\varphi \in$ view names $\Theta \in$ views $\Gamma \in$ catalogs

Objects values

- $\Theta \in$ views (or dictionaries) maps labels to labels
- $\Gamma \in$ catalogs maps view names to views

An object value $v \equiv \nu(\widetilde{\varphi})\, D\,{}^{\Theta}_{\Gamma}$ carries a default view Θ and a catalog Γ and is abstracted over private view names $\widetilde{\varphi}$.

Private views prevent from accidental access (and overriding).

Messages

- $v.\ell$ is forwarded to the method $\Theta\ (\ell)$.
- $\varphi(\varphi'(\varphi'' \ldots v)).\ell$ is forwarded to the method $(\Gamma(\varphi))\ (\ell)$.

Chemical Semantics

Uses a reflexive chemical abstract machine that operates on solutions $\mathcal{D} \Vdash \mathcal{P}$ where $\mathcal{D}$ and $\mathcal{P}$ are multi-sets of definitions and processes:

$$\begin{array}{lrcl}
 & \Vdash 0 & \rightleftharpoons & \Vdash \\
 & \Vdash P \mid P' & \rightleftharpoons & \Vdash P,\, P' \\
(1) & \Vdash \mathsf{obj}\ x = \mathtt{self}(\mathtt{y}) \qquad D \quad \mathsf{in}\ P & \rightleftharpoons & z.\,D \quad \{z/y \qquad\} \Vdash P\{z/x\} \\
(2) & x.[\prod_i \ell'_i\langle\widetilde{y_i}\rangle \triangleright P] \ \Vdash \ \prod_i v_i.\ell_i\langle\widetilde{w_i}\rangle & \longrightarrow & x.[\prod_i \ell'_i\langle\widetilde{y_i}\rangle \triangleright P] \ \Vdash P\{\widetilde{w_i}/\widetilde{x}_i\}
\end{array}$$

Side conditions:

(1) x' are fresh names;

(2) $\ell'_i = \quad \ell_i$, $\quad v_i = x$,

Chemical Semantics

Uses a reflexive chemical abstract machine that operates on solutions $\mathcal{D} \Vdash \mathcal{P}$ where $\mathcal{D}$ and $\mathcal{P}$ are multi-sets of definitions and processes:

$$\begin{array}{lrcl}
 & \Vdash 0 & \rightleftharpoons & \Vdash \\
 & \Vdash P \mid P' & \rightleftharpoons & \Vdash P,\, P' \\
(1) & \Vdash \mathsf{obj}\ x = \mathtt{self}(\mathtt{y})\ \nu(\widetilde{\varphi})\, D^{\Theta}_{\Gamma}\ \mathsf{in}\ P & \rightleftharpoons & z.\,D^{\Theta}_{\Gamma}\{z/y, \widetilde{\psi}/\widetilde{\varphi}\} \Vdash P\{z/x\} \\
(2) & x.[\prod_i \ell'_i\langle\widetilde{y_i}\rangle \triangleright P]^{\Theta}_{\Gamma} \Vdash \prod_i v_i.\ell_i\langle\widetilde{w_i}\rangle & \longrightarrow & x.[\prod_i \ell'_i\langle\widetilde{y_i}\rangle \triangleright P]^{\Theta}_{\Gamma} \Vdash P\{\widetilde{w_i}/\widetilde{x}_i\}
\end{array}$$

Side conditions:

(1) x' and ψ are fresh names;

(2) $\ell'_i = \Theta(\ell_i)$, if $v_i = x$, or $\ell'_i = \Gamma(\varphi^1_i)(\ell_i)$, if $v_i = \varphi^1_i \ldots \varphi^k_i x$.

$\nu(\widetilde{\varphi})$ $\overset{\Theta}{\Gamma}$ $\overset{\Theta}{\Gamma}$ $,\widetilde{\psi}/\widetilde{\varphi}$

$\overset{\Theta}{\Gamma}$ $\overset{\Theta}{\Gamma}$

and ψ

$\Theta(\ \ )$ if or $\ell'_i = \Gamma(\varphi_i^1)(\ell_i)$, if $v_i = \varphi_i^1 \ldots \varphi_i^k x$.

Simple Objective Join = Join + Records

Join can be encoded in the typed Simple Ojoin (without views).

It uses recursive objects, but recursion can be encoded.

$$
\begin{array}{rcl}
[\![x\langle\tilde{u}\rangle]\!] & \triangleq & x.\ell_0\langle\tilde{u}\rangle \\
[\![\text{def } D \text{ in } P]\!] & \triangleq & \text{obj } x = [\![D]\!] \underset{y \in dv(D)}{\text{and}} y = \ell_0\langle u\rangle \triangleright x.\ell_y\langle u\rangle \text{ in } [\![P]\!] \\
[\![y\langle\tilde{u}\rangle]\!] & \triangleq & \ell_y\langle\tilde{u}\rangle
\end{array}
$$

Conversely, there is a typed encoding of the Simple Ojoin into the Join extended with records.

However, the typed encoding does not extend to the calculus with views.

To assemble definitions incrementally

As usual, to structure and share the code.

Even more useful, because of complex internal/external patterns.

To ease reasoning about programs (share proofs of invariants).

Overview

Traditional class-based approach, *i.e.* a small class language on top of the object calculus.

Well-known operators on classes can be transposed to the concurrent setting, sometimes with several variants (overriding).

Refinement of synchronization relies on a new operator — the **ajdunction**.

They are object templates

```
class cell = self (z)
     read ⟨r⟩    | REF⟨a⟩ ▷ r.return⟨a⟩ | z.REF⟨a⟩
  ∧ write⟨b,r⟩ | REF⟨a⟩ ▷ r.return⟨⟩ | z.REF⟨b⟩
  ∧ init⟨⟩ ▷ z.REF⟨0⟩ in
```

They can be instantiated into objects

```
obj a_cell = cell in

a_cell.init⟨⟩ | ... do_something⟨a_cell⟩ ...
```

Traditional inheritance

Adding new methods

```
class logged_cell = self (z)
   cell
∧ log⟨⟩ ▷ obj k = reply⟨a⟩ ▷ print.do⟨a⟩ in z.read⟨k⟩
```

Overriding the behavior (calls to the super class using views).

```
class auto_logged_cell = self (z)
   super ( logged_cell ) hide write
∧ write⟨b,r⟩ ▷
    obj k = reply⟨⟩ ▷ z.log⟨⟩ | r.reply⟨⟩ in super z.write⟨b,k⟩
```

The expression `super (logged_cell)` binds the view name `super` to the default dictionary of `logged_cell`.

A new form of inheritance

Adding competing behavior

```
class fast_cell = self (z)
    cell
 ∧ read⟨r⟩ | write⟨s,b⟩ ▷
      obj k = reply⟨⟩ ▷ r.reply⟨b⟩ | s.reply⟨⟩ in z.write⟨k,b⟩
```

Overriding is always competing: names are shared, unless they have been explicitly hidden.

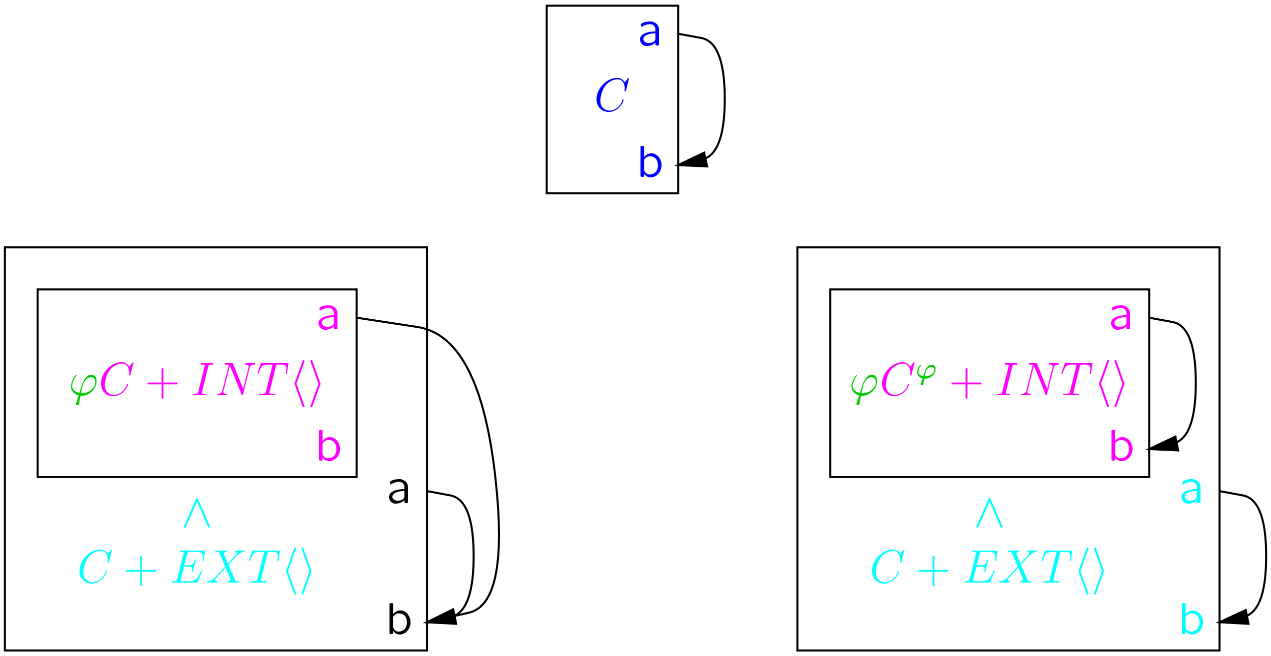
Note that methods can synchronize directly.

Ajdunction for refinement of synchronization

```
class counter =
  self (z)
    internal cell^internal + INT⟨⟩ hide write
  -------------------------------------------
  ∧ (cell + EXT⟨⟩)
  -------------------------------------------
  ∧ ext⟨⟩ | INT⟨⟩ ▷ z.EXT⟨⟩
  ∧ incr⟨r⟩ | EXT⟨⟩ ▷ z.INT⟨⟩ |
      (obj finish = reply⟨v⟩ ▷ r.reply⟨v⟩ | ext⟨⟩ in
       server.do_incr⟨internal z, finish⟩)
```

The expression `cell^internal` rewires all recursive calls of the class `cell` to the view `internal` (hence, it stops late-binding).

Rewiring recursive calls



likely to deadlock

deadlock free

Syntax

Processes			Class bodies		
P	$::=$	$\ldots$	C	$::=$	c
	$\mid$	class $c = \texttt{self}(\texttt{x})\ C$ in P		$\mid$	W
	$\mid$	obj $x = c$ in P		$\mid$	$C \wedge C$
				$\mid$	$C\ \texttt{hide}\ G$
G	$::=$	finite subset of labels			
				$\mid$	φC
				$\mid$	C^{φ}
				$\mid$	$C \oplus \ell\langle\widetilde{x}\rangle$

Small step reduction semantics

$$\frac{C \overset{x}{\rightsquigarrow} C'}{\text{class } c = \texttt{self}(\texttt{x})\ C \text{ in } P \rightsquigarrow \text{class } c = \texttt{self}(\texttt{x})\ C' \text{ in } P}$$

$$\text{class } c = \texttt{self}(\texttt{z})\ W \text{ in } P \rightsquigarrow P\{\texttt{self}(\texttt{z})\ W/_{c}\}$$

(in any context)

Reduction Semantics for class bodies

$$
\begin{array}{rcl}
\psi(\nu(\widetilde{\varphi})\, D\,{}^{\Theta}_{\Gamma}) & \overset{x}{\rightsquigarrow} & \nu(\widetilde{\varphi} \cup \mathit{dom}(\Gamma))\, D\,{}^{\Theta}_{\Gamma,\psi \mapsto \Theta} \\
(\nu(\widetilde{\varphi})\, D\,{}^{\Theta}_{\Gamma})^{\psi} & \overset{x}{\rightsquigarrow} & \nu(\widetilde{\varphi})(\, D\,{}^{\Theta}_{\Gamma}\{\psi x/x\}) \\
\nu(\widetilde{\varphi})\, D\,{}^{\Theta}_{\Gamma}\ \mathtt{hide}\ G & \overset{x}{\rightsquigarrow} & \nu(\widetilde{\varphi})\, D\,{}^{\Theta\setminus G}_{\Gamma} \\
\nu(\widetilde{\varphi}_1)\, D_1\,{}^{\Theta_1}_{\Gamma_1} \wedge \nu(\widetilde{\varphi}_2)\, D_2\,{}^{\Theta_2}_{\Gamma_2} & \overset{x}{\rightsquigarrow} & \nu(\widetilde{\varphi}_1\widetilde{\varphi}_2)\, (D_1 \wedge D_2)\,{}^{\Theta_1 \oplus \Theta_2}_{\Gamma_1 \uplus \Gamma_2} \\
\nu(\widetilde{\varphi})\, D\,{}^{\Theta}_{\Gamma} \oplus \ell\langle\widetilde{y}\rangle & \overset{x}{\rightsquigarrow} & \nu(\widetilde{\varphi})\, (D \oplus (\Theta(\ell)\langle\widetilde{y}\rangle \triangleright x.\ell\langle\widetilde{y}\rangle))^{\Theta}_{\Gamma}
\end{array}
$$

Many obvious easy conditions...

Type system

A la ML: Join + Objective ML + Views

Type soundness

Views

Riecke-Stone, Vouillon, ...

We use a simpler but less expressive version of Vouillon's views.

Concurrent objects and inheritance

Many works in the area.

We have a small (?), expressive calculus.

We can do some —but not all— refinement of synchronization.

Do we have the right primitives?

Expressive operators, simple semantics, useful proof techniques.

Operators were chosen for their semantics properties in jc:

$$D \sim D \wedge D \qquad\qquad \mathsf{def}\ D\ \mathsf{in}\ P \sim \mathsf{def}\ D + \ell\langle\rangle\ \mathsf{in}\ \ell\langle\rangle \mid P$$

Thus, we should easily prove equations such as

$$\varphi(C \wedge \psi C) \sim \varphi C \qquad\qquad \varphi, \psi \notin fv(C)$$

to be used for program optimizations or as proof techniques.

Implementation

Still to be done to validate our choice of primitives.

Joe-caml + Ocaml (classed-based multiple inheritance) + views.

Views

Views play a crucial role for expressiveness. *Still useful for privacy.*

Views are not new; their application in a concurrent setting is.
Our motivations and requirements are different

Calculi with views should be studied in more details.
There is much room for improvement.

From sequential to concurrent language

When transfering sequential technology to the concurrent setting, we found new needs and news challenges.

Objects and classes are one approach to modular programming. Modules and especially *mixins* should also be explored.

Private fields

Using groups: private fields can be hidden altogether at object creation:

```
obj a_cell =
   public (cell hide private)
in
   a_cell.init⟨⟩ | ... do_something⟨a_cell⟩ ...
```

Initialization of classes

The init method is left accessible and can be called several time.

A primitive treatment is preferable.

Hiding and View Ajunction

```
class safe_counter = self (z)
  (  local (cell alloc EXT ∧ ext⟨⟩ | INT⟨⟩ ▷ z.EXT⟨⟩)
  ∧ internal cell^internal + INT⟨⟩ hide write
  ∧ (cell + local.EXT⟨⟩)
  ∧ incr⟨r⟩ | EXT⟨⟩ ▷ local z.INT⟨⟩ |
       (obj k = return⟨⟩ ▷ local z.ext⟨r⟩ in
        server.do_inc r⟨internal z,k⟩)
  ) hide ext INT EXT
```