Contents

Inheritance in the Join-Calculus

Cédric Fournet
Microsoft Research, Cambridge

Cosimo Laneve
University of Bologna

Luc Maranget
INRIA-Rocquencourt

Didier Rémy
INRIA-Rocquencourt

May 2001

General goal

Design a language

- for concurrent and **distributed** programming
- that is both **expressive** and **safe**

  (*i.e.* it should allows to reason about programs, including types, program analyses, proofs)

Specific goals in this work

A concurrent, object-oriented calculus with:

- **object internal concurrency**

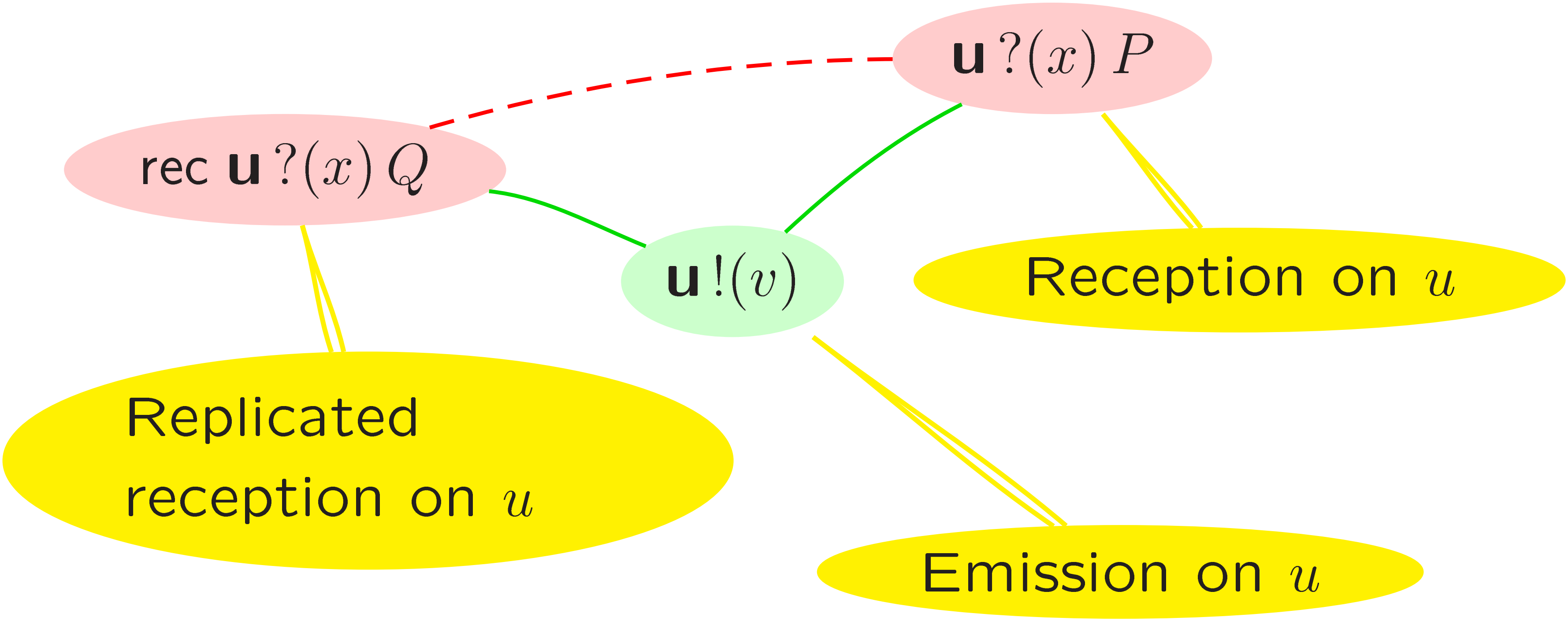
  — not just sequential objects running concurrently!

- **incremental assembling of object definitions**

  — including the refinement of both

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

Definition of behaviors in the π-calculus



$\textbf{u}\,?(x)\,P$

rec $\textbf{u}\,?(x)\,Q$

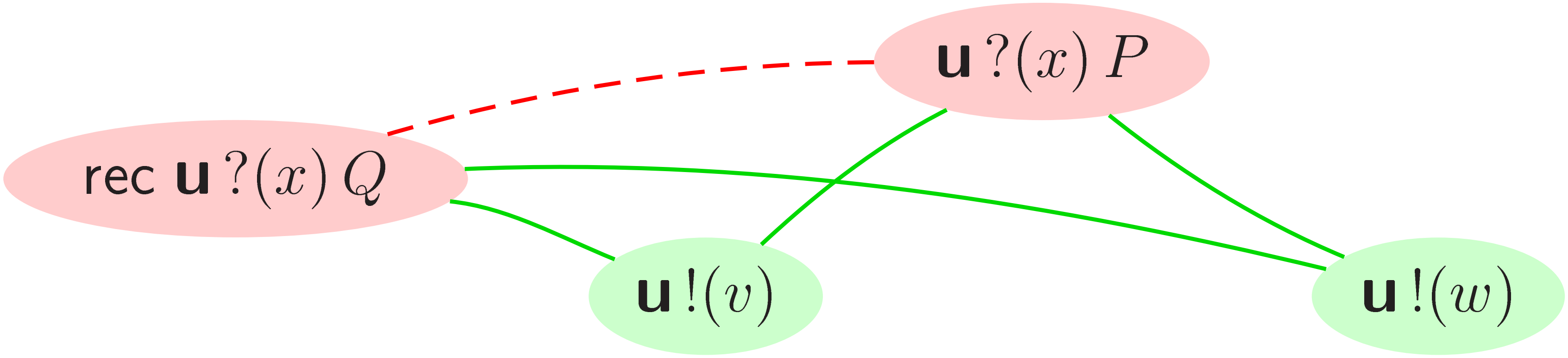
$\textbf{u}\,!(v)$

Reception on u

Replicated reception on u

Emission on u

Definition of behaviors in the π-calculus



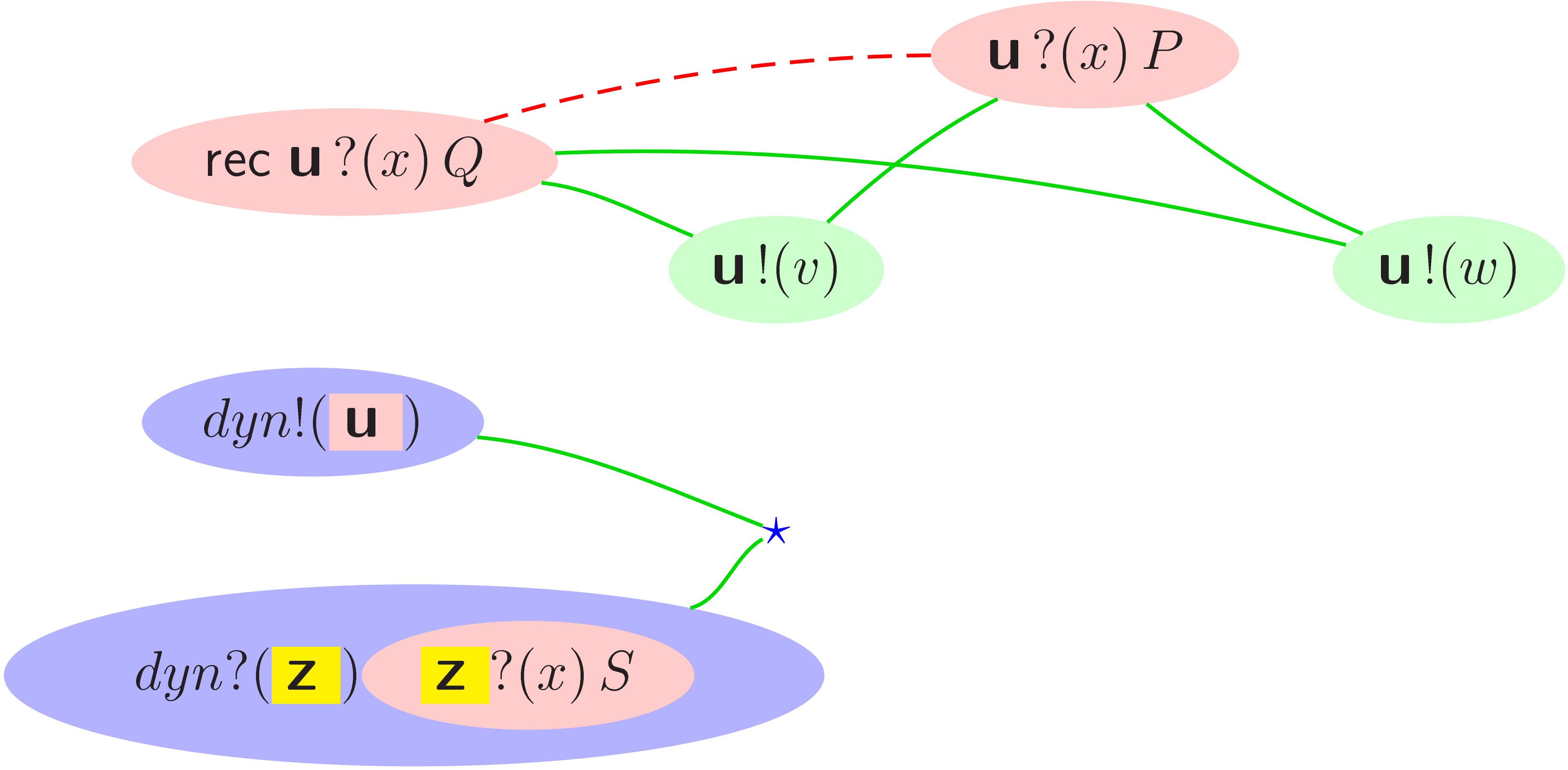
$\mathsf{rec}\ \mathbf{u}\,?(x)\,Q$

$\mathbf{u}\,?(x)\,P$

$\mathbf{u}\,!(v)$

$\mathbf{u}\,!(w)$

Definition of behaviors in the π-calculus



rec $\mathbf{u}\,?(x)\,Q$

$\mathbf{u}\,?(x)\,P$

$\mathbf{u}\,!(v)$

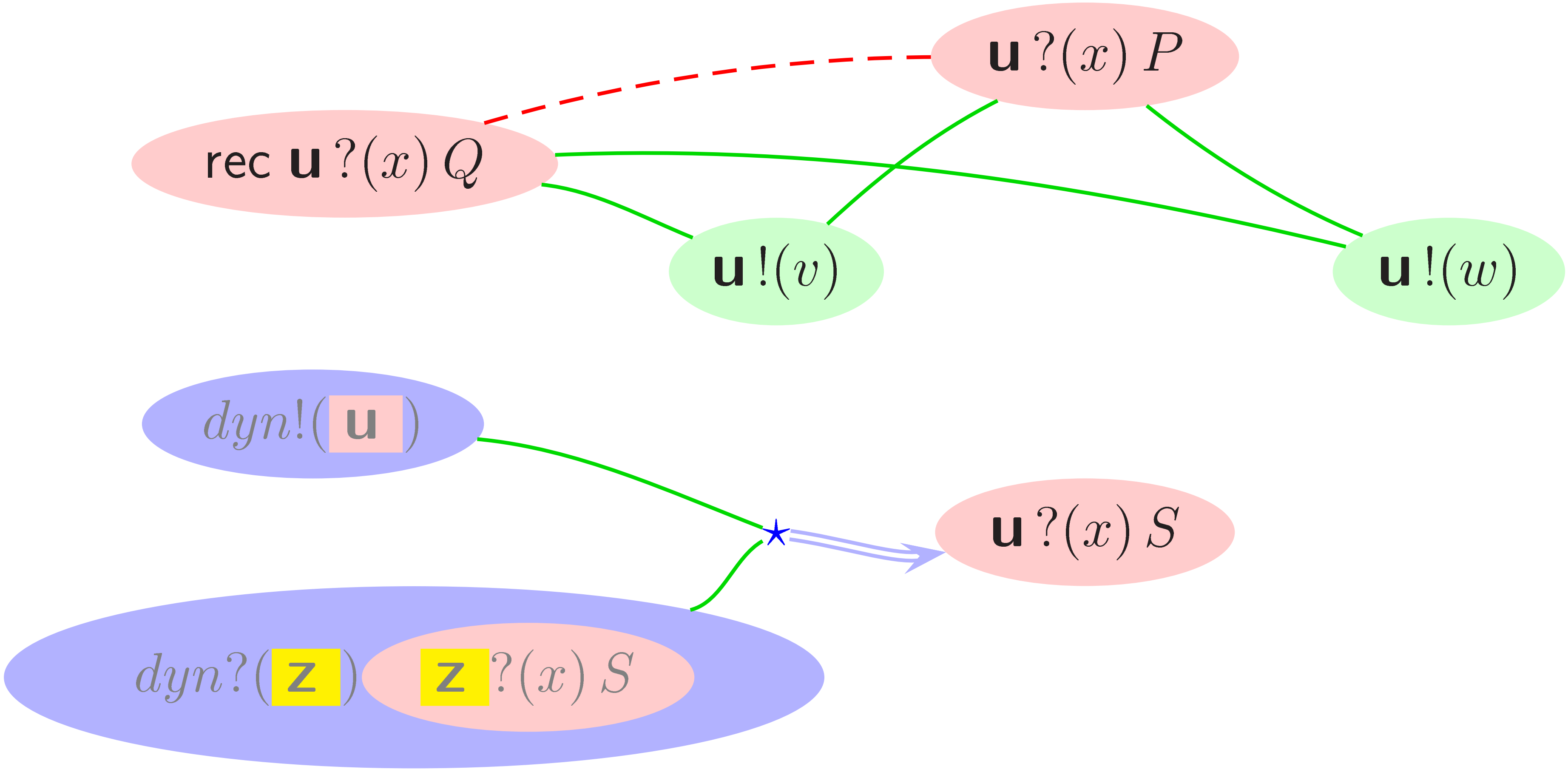
$\mathbf{u}\,!(w)$

$dyn!(\,\mathbf{u}\,)$

$\star$

$dyn?(\,\mathbf{z}\,)$ $\mathbf{z}\,?(x)\,S$

Definition of behaviors in the π-calculus



rec $\mathbf{u}\,?(x)\,Q$

$\mathbf{u}\,?(x)\,P$

$\mathbf{u}\,!(v)$

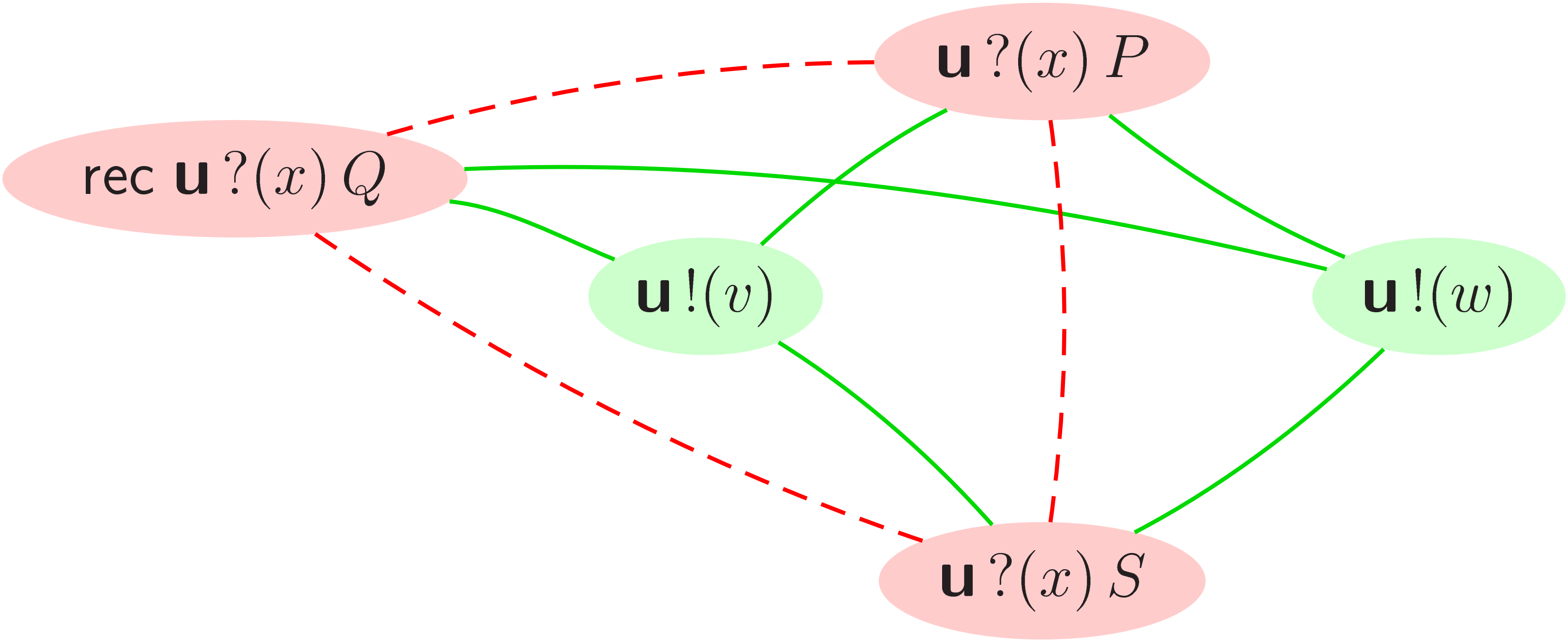
$\mathbf{u}\,!(w)$

$dyn!(\mathbf{u})$

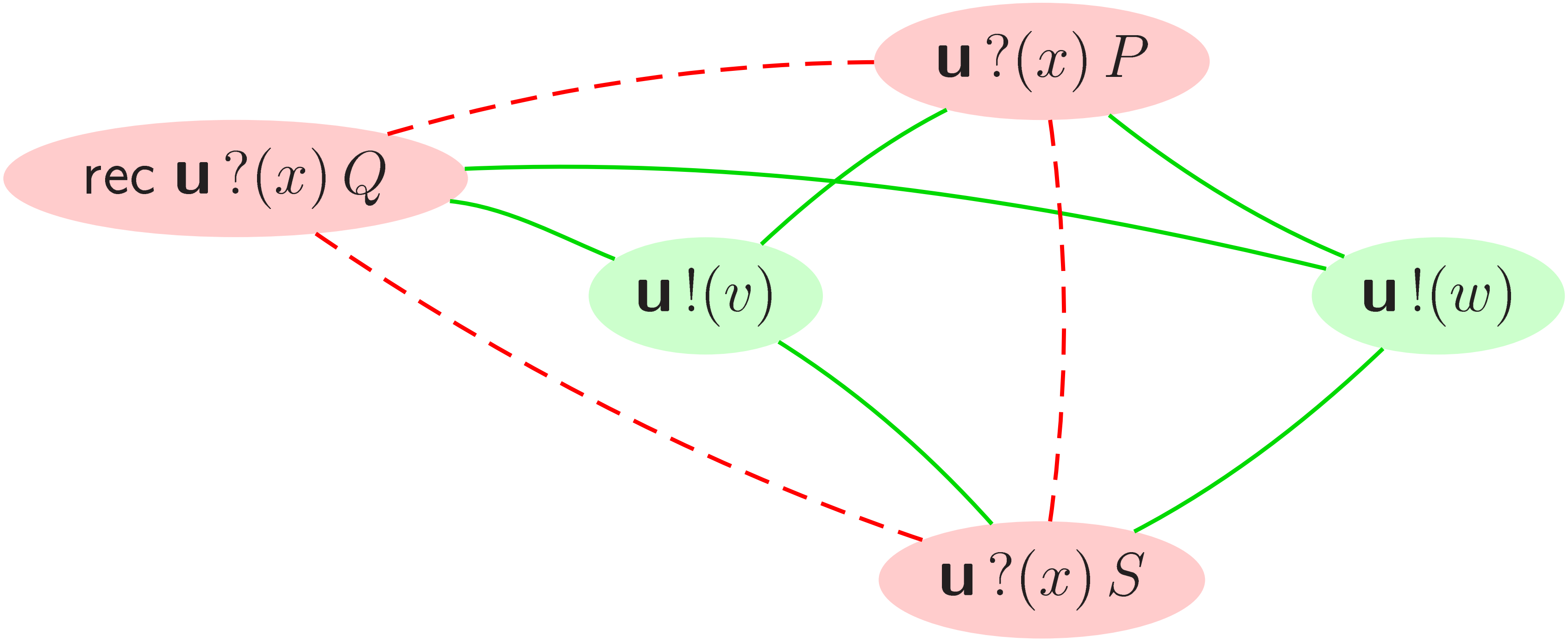
$\mathbf{u}\,?(x)\,S$

$dyn?(\mathbf{z})$ $\mathbf{z}\,?(x)\,S$

Definition of behaviors in the π-calculus

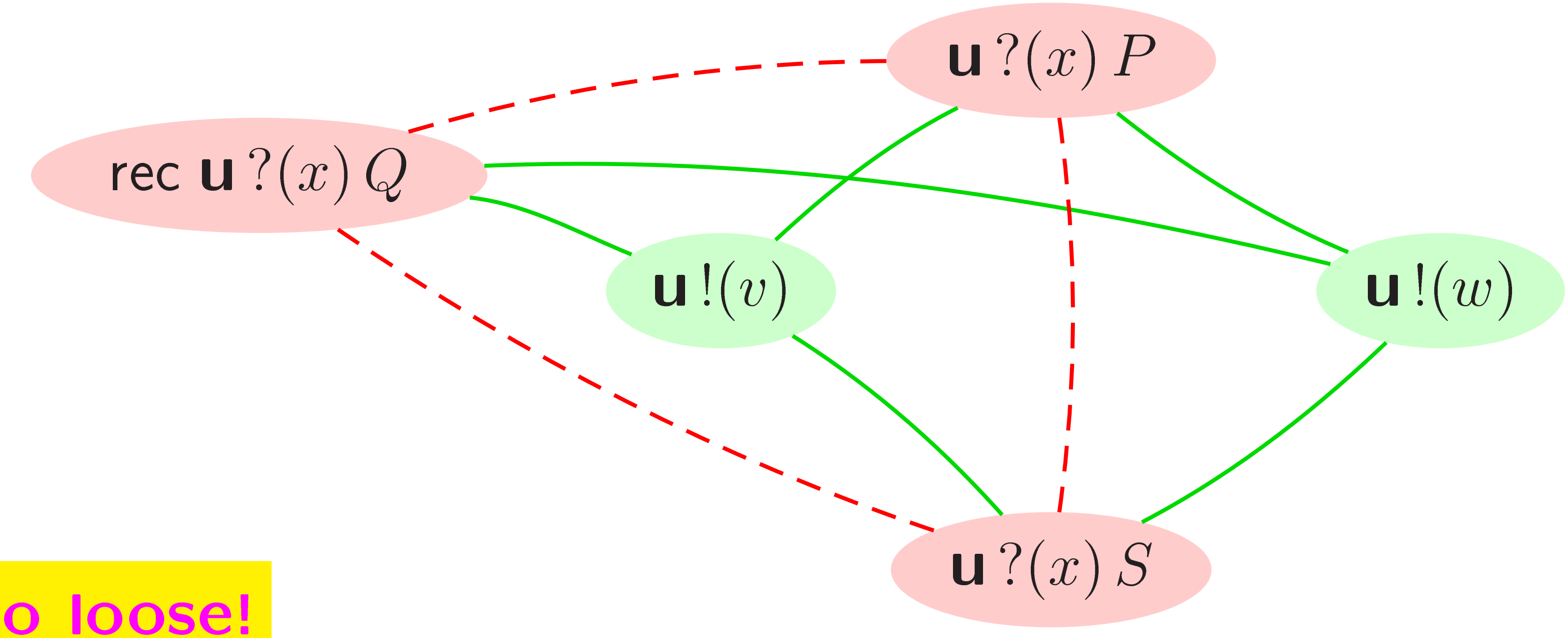


Definition of behaviors in the π-calculus



- Behaviors attached to a name are not syntactically known; they are reconfigurable, dynamically.
- This makes reasoning *and* distributed computing difficult.

Definition of behaviors in the π-calculus



Too loose!

- Behaviors attached to a name are not syntactically known; they are reconfigurable, dynamically.
- This makes reasoning *and* distributed computing difficult.

Definitions of behavior in the join-calculus

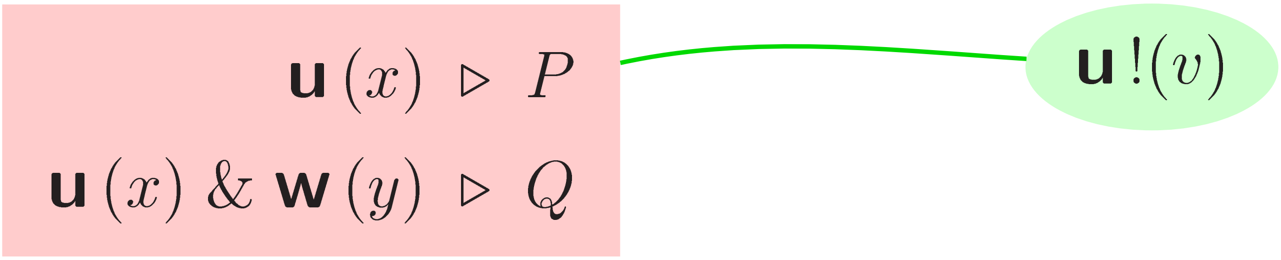
$$\begin{aligned} \mathbf{u}\,(x) \;&\triangleright\; P \\ \mathbf{u}\,(x) \,\&\, \mathbf{w}\,(y) \;&\triangleright\; Q \end{aligned}$$

$\mathbf{u}\,!(v)$

Asynchronous emission on **u**

Defines synchronizing names **u** and **w** and their behaviors altogether

Definitions of behavior in the join-calculus

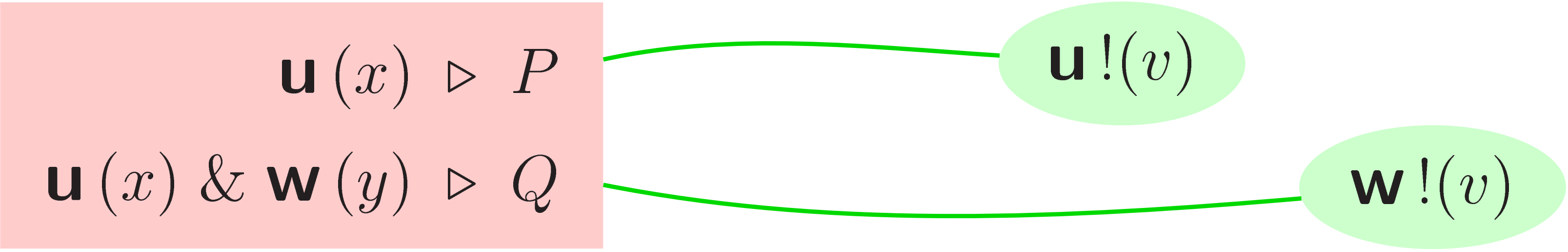


$$\mathbf{u}(x) \triangleright P$$

$$\mathbf{u}(x) \,\&\, \mathbf{w}(y) \triangleright Q$$

$$\mathbf{u}\,!(v)$$

Definitions of behavior in the join-calculus



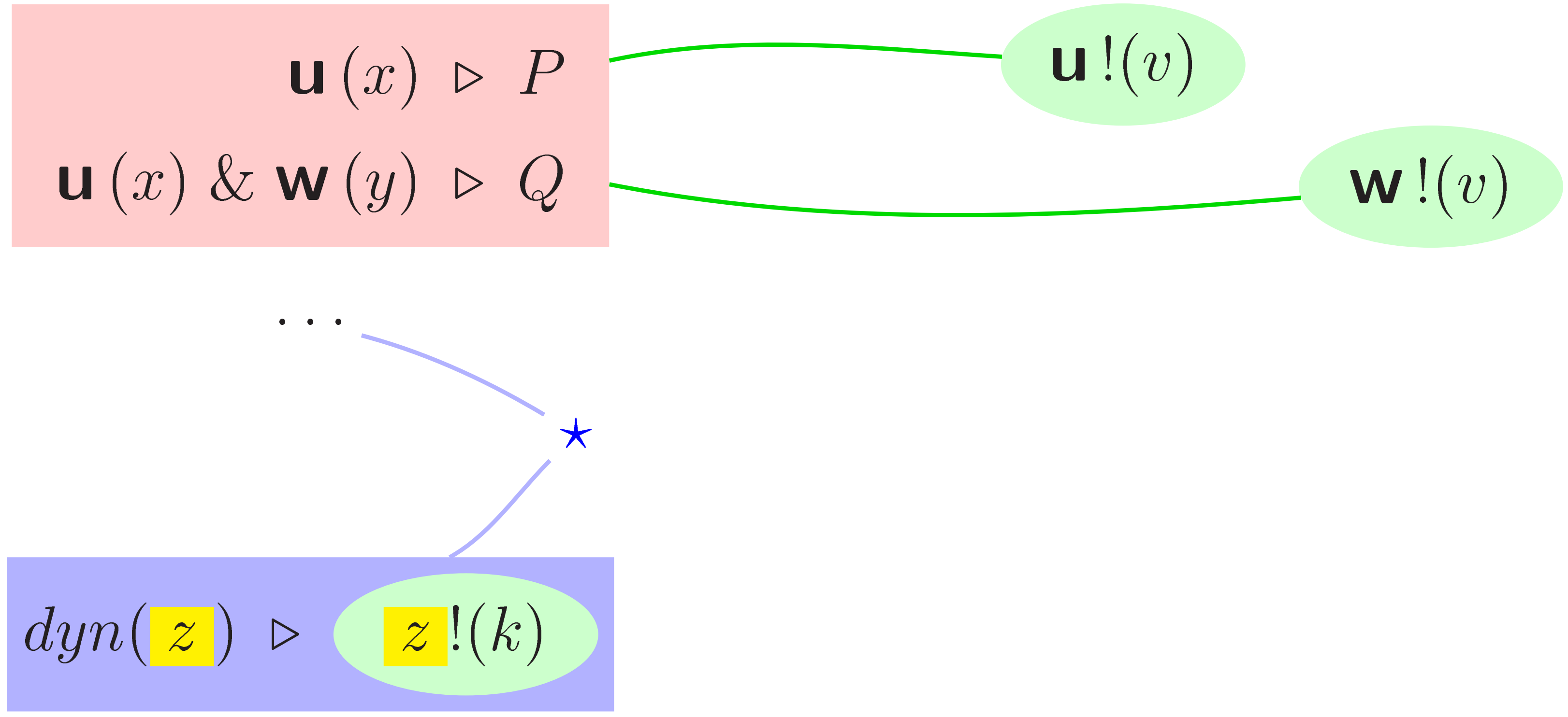
$\mathbf{u}\,(x) \;\triangleright\; P$

$\mathbf{u}\,(x) \;\&\; \mathbf{w}\,(y) \;\triangleright\; Q$

$\mathbf{u}\,!(v)$

$\mathbf{w}\,!(v)$

Definitions of behavior in the join-calculus



$$\mathbf{u}(x) \triangleright P$$

$$\mathbf{u}(x) \,\&\, \mathbf{w}(y) \triangleright Q$$

$$\mathbf{u}!(v)$$

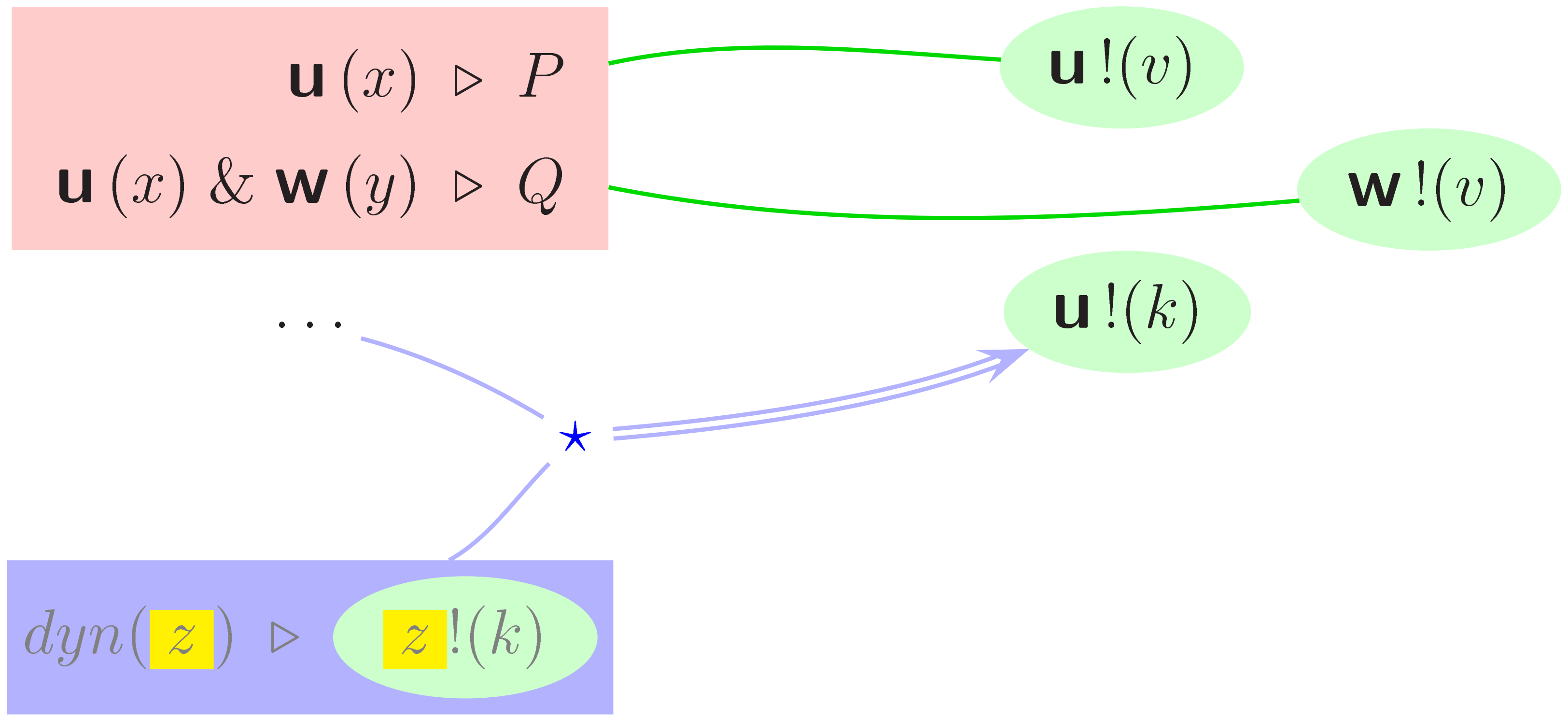
$$\mathbf{w}!(v)$$

$$\ldots$$

$$\star$$

$$dyn(z) \triangleright z!(k)$$

Definitions of behavior in the join-calculus



$\mathbf{u}(x) \triangleright P$

$\mathbf{u}(x) \,\&\, \mathbf{w}(y) \triangleright Q$

$\mathbf{u}!(v)$

$\mathbf{w}!(v)$

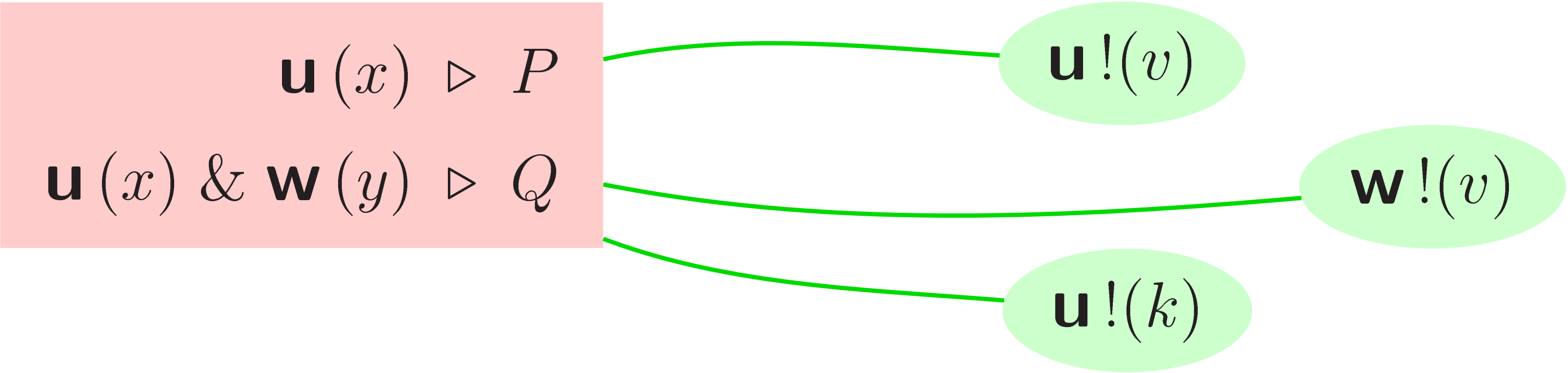
$\mathbf{u}!(k)$

$\ldots$

$\star$

$dyn(z) \triangleright z!(k)$

Definitions of behavior in the join-calculus



$$\mathbf{u}\,(x) \;\triangleright\; P$$

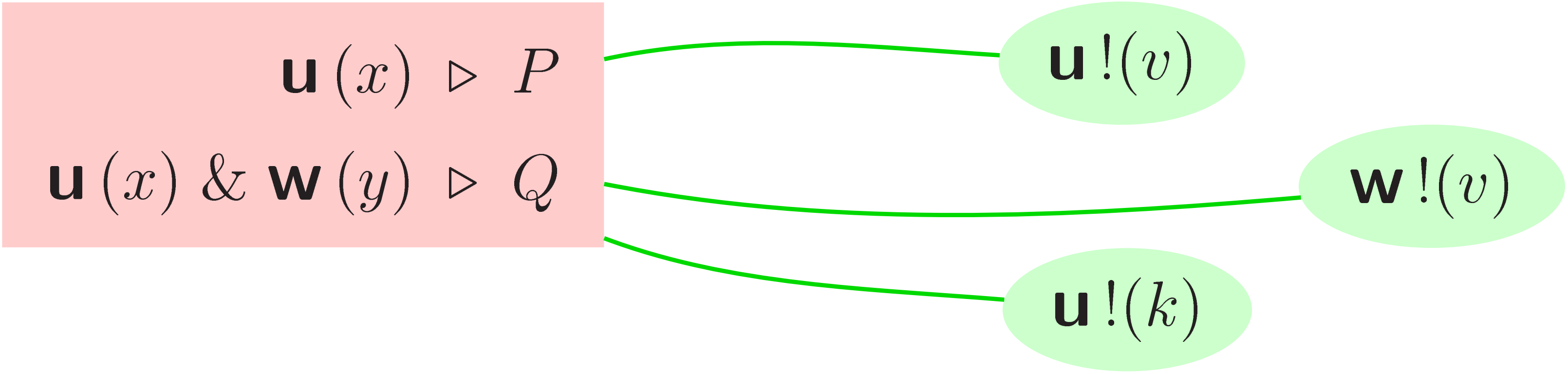
$$\mathbf{u}\,(x) \,\&\, \mathbf{w}\,(y) \;\triangleright\; Q$$

$\mathbf{u}\,!(v)$

$\mathbf{w}\,!(v)$

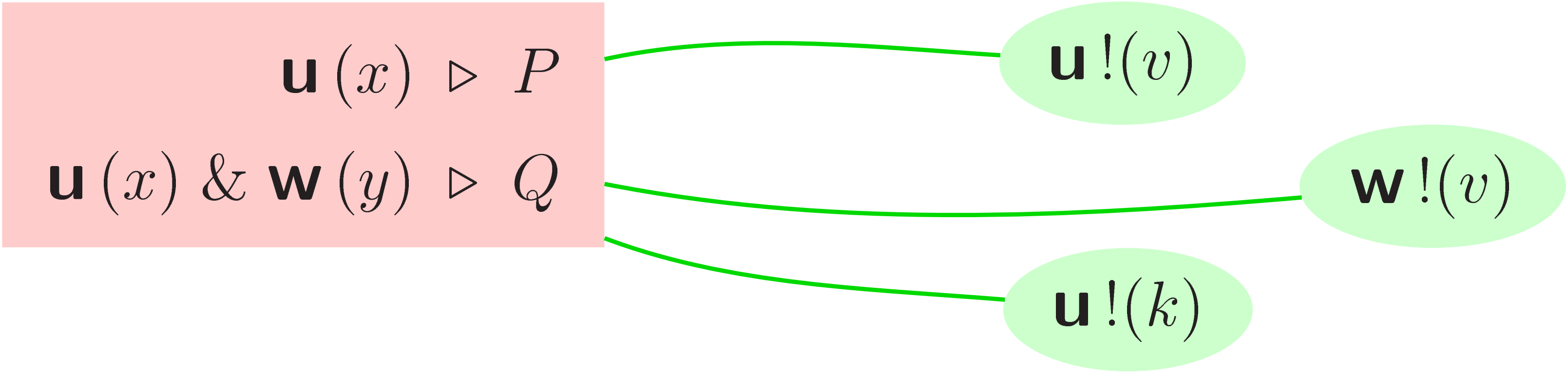
$\mathbf{u}\,!(k)$

Definitions of behavior in the join-calculus



- Name scoping is more constrained than in π-calculus
  - *Behavior of a name is statically defined and lexically scoped.*
  - *Synchronizing names are jointly defined.*

Definitions of behavior in the join-calculus



$$\mathbf{u}\,(x) \ \triangleright\ P$$

$$\mathbf{u}\,(x)\ \&\ \mathbf{w}\,(y) \ \triangleright\ Q$$

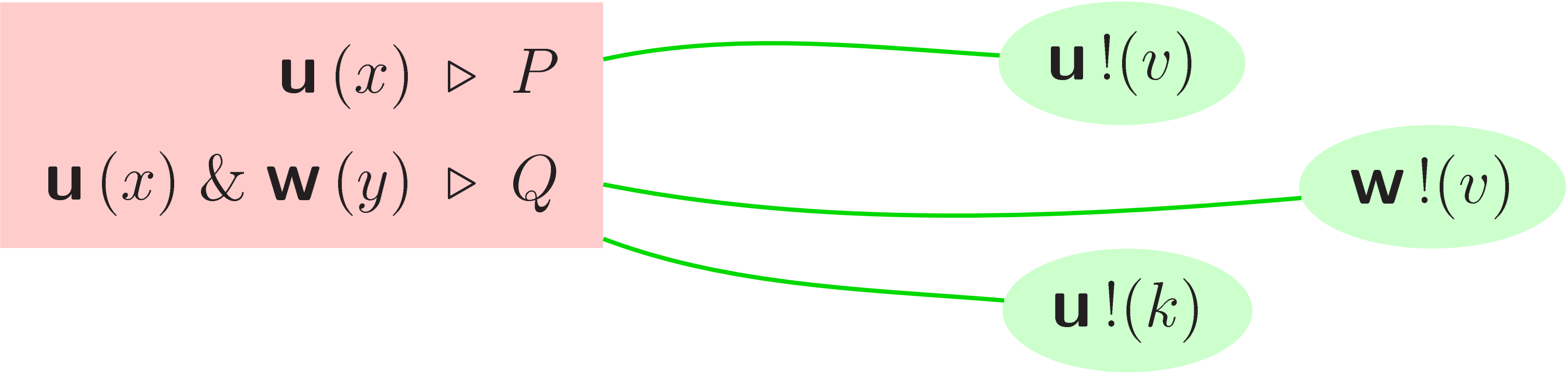
$$\mathbf{u}\,!(v)$$

$$\mathbf{w}\,!(v)$$

$$\mathbf{u}\,!(k)$$

- Name scoping is more constrained than in π-calculus
- Easier for reasoning and distributed computation

Definitions of behavior in the join-calculus



$\mathbf{u}\,(x) \;\triangleright\; P$

$\mathbf{u}\,(x) \;\&\; \mathbf{w}\,(y) \;\triangleright\; Q$

$\mathbf{u}\,!(v)$

$\mathbf{w}\,!(v)$

$\mathbf{u}\,!(k)$

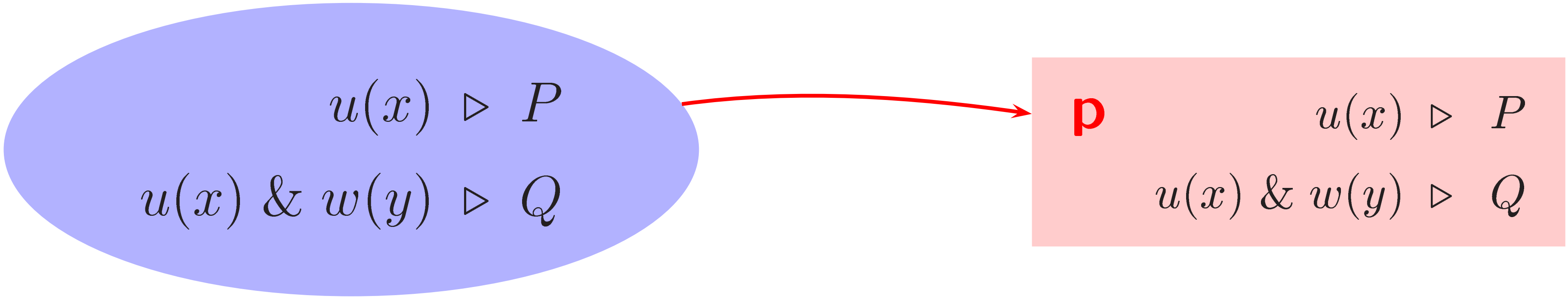
- Name scoping is more constrained than in π-calculus
- Easier for reasoning and distributed computation

But it's too rigid!

Class-based inheritance of object definitions

$$\begin{aligned} u(x) &\rhd P \\ u(x) \mathbin{\&} w(y) &\rhd Q \end{aligned}$$

Class-based inheritance of object definitions



$$u(x) \triangleright P$$

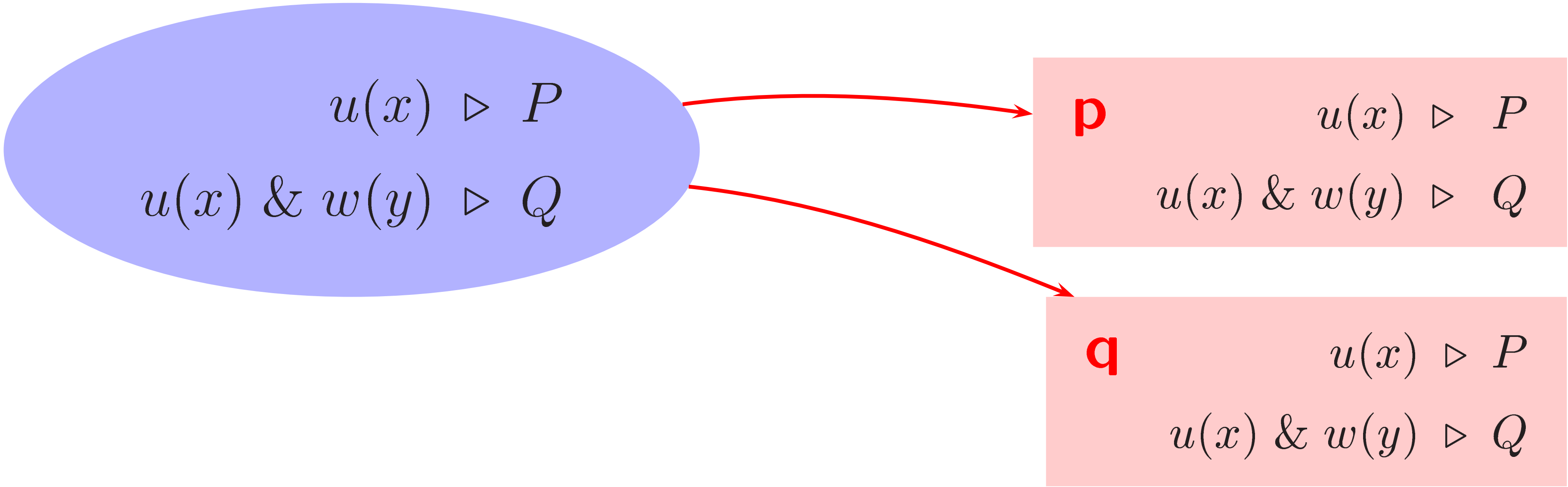
$$u(x) \,\&\, w(y) \triangleright Q$$

p

$$u(x) \triangleright P$$

$$u(x) \,\&\, w(y) \triangleright Q$$

Class-based inheritance of object definitions



$$u(x) \triangleright P$$

$$u(x) \,\&\, w(y) \triangleright Q$$

p

$$u(x) \triangleright P$$

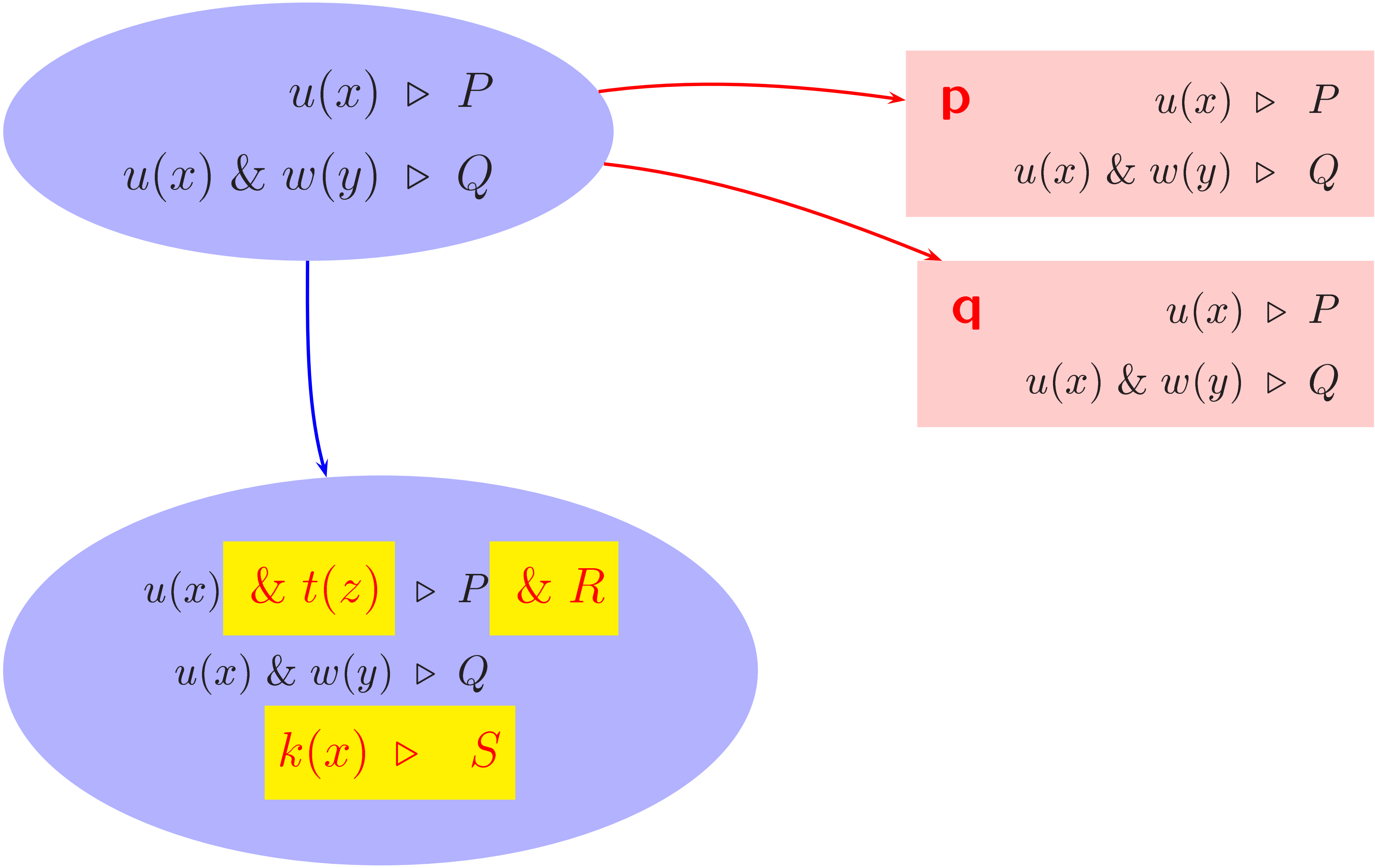
$$u(x) \,\&\, w(y) \triangleright Q$$

q

$$u(x) \triangleright P$$

$$u(x) \,\&\, w(y) \triangleright Q$$

Class-based inheritance of object definitions



$u(x) \triangleright P$

$u(x) \,\&\, w(y) \triangleright Q$

p

$u(x) \triangleright P$

$u(x) \,\&\, w(y) \triangleright Q$

q

$u(x) \triangleright P$

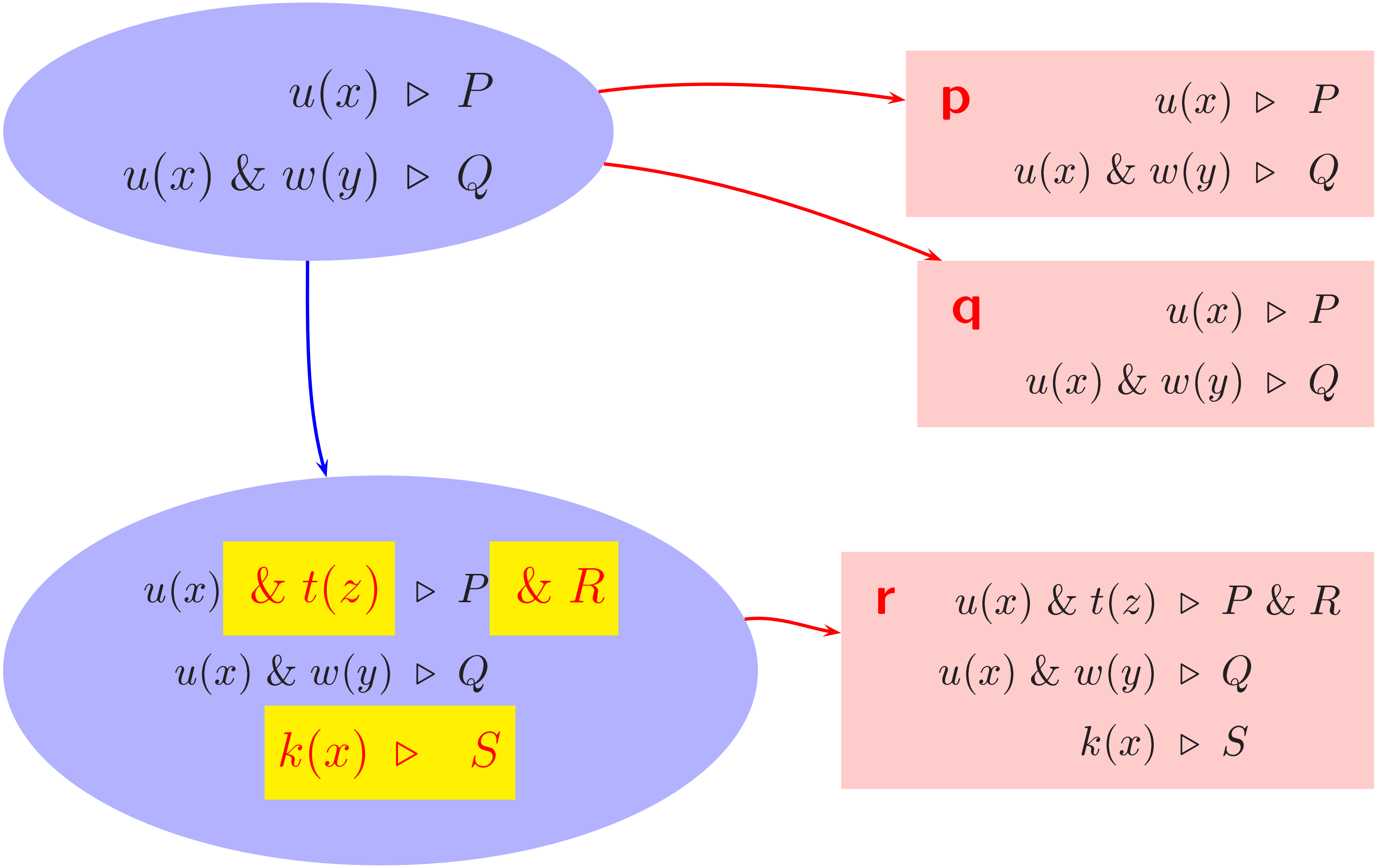
$u(x) \,\&\, w(y) \triangleright Q$

$u(x) \,\&\, t(z) \triangleright P \,\&\, R$

$u(x) \,\&\, w(y) \triangleright Q$

$k(x) \triangleright S$

Class-based inheritance of object definitions



$$u(x) \triangleright P$$
$$u(x) \,\&\, w(y) \triangleright Q$$

p

$$u(x) \triangleright P$$
$$u(x) \,\&\, w(y) \triangleright Q$$

q

$$u(x) \triangleright P$$
$$u(x) \,\&\, w(y) \triangleright Q$$

$$u(x) \,\&\, t(z) \triangleright P \,\&\, R$$
$$u(x) \,\&\, w(y) \triangleright Q$$
$$k(x) \triangleright S$$

r

$$u(x) \,\&\, t(z) \triangleright P \,\&\, R$$
$$u(x) \,\&\, w(y) \triangleright Q$$
$$k(x) \triangleright S$$

Class-based inheritance of object definitions

Definitions are built incrementally:

- Objects are closed join definitions,

  Core Ojoin $\approx$ *Join* $+$ *Record-like structure (names* $\longleftrightarrow$ *methods)*

- Classes are open definitions:

  They can be extended, refined, and combined

Syntax of the core object calculus

$P ::=$	**Processes**
$\mathsf{0}$	null process
$x.M$	message sending
$P_1 \& P_2$	parallel composition
$\mathsf{obj}\ x = D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
$D ::=$	**Definitions**
$M \ \triangleright\ P$	reaction rule
$D_1\ \mathsf{or}\ D_2$	disjunction
$M ::=$	**Patterns**
$\ell(\widetilde{u})$	message
$M_1 \& M_2$	synchronization

Object definitions

An object definition

$$\text{obj } b = J \triangleright P \text{ or } J' \triangleright P' \text{ or } \ldots \text{ in } Q$$

defines:

- a set of reaction rules of the form:

$$\underbrace{\ell_1(\widetilde{u}_1) \;\&\; \ldots \ell_k(\widetilde{u}_k)}_{\text{Pattern } J} \quad \triangleright \quad \underbrace{P}_{\text{Process}}$$

- the set of methods ℓ's appearing in patterns J's.
- the name b of the object (visible in P's and Q)

Reaction rules

Describe how to process messages sent to the object.

Intuitively, if the object b is active, messages send to b may react according to any reaction rule:

$$b.\ell_1(\widetilde{t_1}) \mathbin{\&} \ldots b.\ell_k(\widetilde{t_n}) \;\longrightarrow\; P\{\widetilde{u}_1 \leftarrow \widetilde{t}_1, \ldots \widetilde{u}_k \leftarrow \widetilde{t}_k\}$$

- messages matching a pattern are jointly consumed
- the process of the matched rule is activated, after replacing formal parameters by actual names carried by messages.

(All reactions rules compete.)

Behavior, state, and synchronization

Methods are used to describe altogether Behavior, STATE, and SYNCHRONIZATION.

Example of a buffer

```
obj b =
   self (z)
      Get(r)    & SOME(a)  ▷  r.Return(a) & z.EMPTY()
   or Put(a,r) & EMPTY()  ▷  r.Return()  & z.SOME(a)
in
   init b.EMPTY() & b.Put(2,k)  & do_something(b)
```

Behavior, state, and synchronization

Methods are used to describe altogether Behavior, STATE, and SYNCHRONIZATION.

Example of a buffer

```
obj b =
   self (z)
      Get(r)    & SOME(a)  ▷  r.Return(a) & z.EMPTY()
   or Put(a,r) & EMPTY()  ▷  r.Return()  & z.SOME(a)
in
   b.Put(2,k) & b.EMPTY() & do_something_else(b)
```

Behavior, state, and synchronization

Methods are used to describe altogether Behavior, STATE, and SYNCHRONIZATION.

Example of a buffer

```
obj b =
   self (z)
      Get(r)    & SOME(a)  ▷  r.Return(a) & z.EMPTY()
   or Put(a,r) & EMPTY()  ▷  r.Return()  & z.SOME(a)
in
   k.Return() & b.SOME(2)  & do_something_else(b)
```

Synchronization is primitive

Synchronization is represented as messages:

- The state may be composed of exclusive messages
  e.g. as SOME(a) and EMPTY()
  or concurrent messages
  e.g. LEFT(a) and RIGHT(b) contents of a pair.
- Methods working on independent pieces of the state may run concurrently (object internal concurrency).
- Methods may compete on the same piece of state or between themselves
  e.g. a one-place synchronous buffer may just be:
  self(z) Put(a) & Get(r) ▷ r.Return(a)

Synchronization is primitive

- Primivite synchronization is safer, but limited.
- Complex synchronization can always be programmed.

Classes are object templates

A class **buffer**

```
class buffer =
   self (z)
       Get(r)    & SOME(a)  ▷ r.Return(a) & z.EMPTY()
   or Put(a,r) & EMPTY()  ▷ r.Return()  & z.SOME(a)
in  ...
```

An instance of **buffer**

```
obj b = buffer  init EMPTY() in
do_something(b) & b.Put(2,k)
```

(Each object gets created with a different state.)

Inheritance

```
class logged_buffer =
  self (z)
    buffer
  or Log() & SOME(a) ▷ out.Int(a) & z.SOME(a)
  or Log() & EMPTY() ▷
                  out.String("Empty") & z.EMPTY()
```

- New rules may define new methods (such as Log)
- They may refer to old ones (such as SOME).
- Here, they new methods/definitions do not replace older ones but compete with them!

  (This has no counterpart in a sequential setting.)

Classes with multiple inheritance

So far, the proposal is straightforward

$$
\begin{aligned}
P &::= \ \ldots \mid \mathsf{class}\ c = C\ \mathsf{in}\ P \mid \mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q \\
C &::= \ c \mid \mathsf{self}(z)\, C \mid J \triangleright P \mid C\ \mathsf{or}\ C' \mid \ldots
\end{aligned}
$$

But not sufficient

- To refine a parent method (as in sequential languages), the old behavior should be removed.
- To refine synchronization, new locks should be added to the reaction rules of the parent class.

The solution is to introduce a new powerful operation for doing precise surgery on the parent class.

Patching (of reaction rules)

patch C with $J_1 \Rightarrow J'_1 \rhd P_1 \mid \ldots J_n \Rightarrow J'_n \rhd P_n$ end

The effect is, for each reaction rule $J \rhd P$ of C,

- find the smallest k such that J matches J_k, *i.e.* is of the form $J_k \mathbin{\&} J''$; and

  replace the rule $J \rhd P$ by $J'_k \mathbin{\&} J'' \rhd P \mathbin{\&} P_k$.
- leave the rule $J \rhd P$ unchanged, if no such k exists.
- Modularity is preserved

Adding behavior to a reaction

Logging Put successes:

```
class logged_buffer2 =
   patch buffer with
     Put(a,r) ⇒ Put(a,r) ▷ out.Int(a)
   end
```

The body of logged_buffer2 is equivalent to:

```
 self(z)
    Get(r) & SOME(a) ▷ r.Return(a) & z.EMPTY()
or  Put(a,r) & EMPTY()  ▷
        r.Return()  & z.SOME(a) & out.Int(a)
```

(Standard) Overriding with call to super

Logging Put attempts

```
class logged_buffer3 =
 patch buffer with Put(a,r) ⇒ Parent_Put(a,r) end
 or Put(a,r) ▷ Parent_Put(a,r) & out.Int(a)
```

The body of logged_buffer3 is equivalent to:

```
self(z)
   Get(r)  & SOME(a) ▷ r.Return(a) & z.EMPTY()
or Parent_Put(a,r) & EMPTY() ▷
                          r.Return()  & z.SOME(a)
or Put(a,r) ⇒ Parent_Put(a,r) & out.Int(a)
```

Refinement of synchronization

As refinement of behavior, but applies to STATE or SYNCHRONIZATION methods instead of Behavioral methods.

Modification of acceptable states

```
class  lockable_buffer =
   self(z)
      patch buffer with
         Put(a,r) ⇒ Put(a,r) & FREE() ▷ z.FREE()
      end
   or Suspend(r) & FREE()    ▷  r.Reply() & z.LOCKED()
   or Resume(r)  & LOCKED()  ▷  r.Reply() & z.FREE()
```

Calls to method Put can then be enabled or disabled by calling methods Resume and Suspend, respectively,

Other refinements

Some difficult examples of class refinement during inheritance are known as *inheritance anomalies* [Matsuoka and Yonezawa].

Their example suite... can all be solved

- Modification of acceptable states. (*c.f.* lock)
- Partitioning of acceptable states
- History-dependent acceptable states.

And more...

Locking external calls during internal recursive calls.

Of course, any limit of expressiveness can be turned into some cooked-up example of inheritance anomaly...

Formal semantics

Dynamics semantics

- Chemical semantics for the core object calculus (no classes) that is parameterized by some process reductions.
- Reduction semantics for classes.

Static semantics

- including a notion of privacy

Soundness

- by subject reduction

Lots of related works but...

Foundations for concurrent objects

– Use encodings or add concurrent primitives to existing sequential object-oriented languages.

– Ignore the composition of object definitions.

Inheritance of concurrent objects

Conversely, those works do usually not consider the foundations:

– Do not consider typechecking.

– The constructions are higher-level and less atomic.

– Synchronization and behavior are two different mechanisms.

– Different examples use different, somewhat ad-hoc, solutions.

Odersky's Functional nets is the closest work (also based on the join-calculus), but has no counterpart to selective refinement.

Conclusions

The language

- A simple well-understood, expressive calculus of objects,
- One powerful operation for doing surgery on parent classes.

Expressiveness

- We can solve a wide variety of examples of inheritance in a uniform manner.

Conclusions

Implementation

- This remains to be implemented in JoCaml.
- Hopefully, JoCaml objects could be merged with Ocaml's objects, but it remains to be investigated.

Click on...

`http://pauillac.inria.fr/jocaml/`

`http://cristal.inria.fr/~remy/work/ojoin/`

Do you want to hear
anything about types?

Type menu

- Why it's easy
- Polymorphism in Ojoin
- A trace of synchronization in types
- More care is needed for virtual labels
- Dynamic surgery errors
- Type expressions and selected typing rules

Why typing is easy

It is similar to typing ML/Ocaml.

- Object types are record-like types (describe public methods and their types)
- Polymorphic messages from polymorphic row-variables provide (subtyping is optional).
- Classes are toplevel, hence class types never need to be guessed.
- Abstraction relies on a distinction between private and public names (private names only visible within classes).

Virtual labels

Class types keep track of virtual labels (labels that are used but *may* not defined):

- introduced as virtual (for abstract classes)
- their definition might have been removed during some refinement;
- the type system will ensure that those will be eventually (re-)defined.

Refinement failure

- A selective refinement patch C with S end fails if it would otherwise reduce to a class C' where $dl(C') \not\supseteq dl(S)$.

  With A() $\triangleright$ 0 **or** B() $\triangleright$ 0 for C,

  patch C **with** A() & B() $\Rightarrow$ Lock() $\triangleright$ Lock() **end**

  reduces to C, which does not declare the method lock (declared in S).

Refinement failure

- A selective refinement patch C with S end fails if it would otherwise reduce to a class C' where $dl(C') \not\supseteq dl(S)$.

  With A() $\triangleright$ 0 **or** B() $\triangleright$ 0 for C,

  patch C **with** A() & B() $\Rightarrow$ Lock() $\triangleright$ Lock() **end**

  reduces to C, which does not declare the method lock (declared in S).
- This can only happen if a clause of the refinement has not been used, and in particular, if its left hand side joins several messages.

Refinement failure

- A selective refinement patch C with S end fails if it would otherwise reduce to a class C' where $dl(C') \not\supseteq dl(S)$.
- This can only happen if a clause of the refinement has not been used, and in particular, if its left hand side joins several messages.
- This is somehow similar to pattern matching failures in ML.

Refinement failure

- A selective refinement patch C with S end fails if it would otherwise reduce to a class C' where $dl(C') \not\supseteq dl(S)$.
- This can only happen if a clause of the refinement has not been used, and in particular, if its left hand side joins several messages.
- This is somehow similar to pattern matching failures in ML.
- Most refinement patterns have single messages as left hand-sides and cannot fail dynamically.

Messages allow side effects

Ojoin uses ML-like polymorphism, with type generalization at class and object definitions.

However, some generalizations are unsafe!

Example

obj sbuffer $=$ Get(r) & Put(n,s) $\triangleright$ r.Reply(n) & s.Reply()

is typed as:

$$\mathsf{Get} : [\mathsf{Reply} : \alpha; \varrho]\ ;\quad \mathsf{Put} : (\alpha, [\mathsf{Reply} : (); \varrho'])$$

- to preserve consistency between Get and Put, variable α cannot be generalized;
- conversely, variables ρ and ρ' can be generalized.

Objects vs. classes

- When objects are created from scratch, all synchronization information is available.
- When created from classes, synchronization information is only available from their types.
  - If no information is available, no variable can generalized (except for single method objects).
  - If all synchronization is available, better polymorphic types can be given.
- We keep only an approximation of the synchronization patterns in types.

Coupled labels

- Coupled labels *cl*(C) of a class C are labels appearing in a multi-message pattern.
- Statically, we keep track of an approximation —a superset W of coupled labels.
- Coupled typed variables of a set of typing assertions B is: $\bigcup_{\ell \neq \ell' \in dom(B)} B(\ell) \cap B(\ell')$
- In an instance of a class C declaring methods B with coupled labels W coupled type variables of $B \upharpoonright W$ cannot be generalized
- Classes are abstractions and can always be generalized.

Types expressions

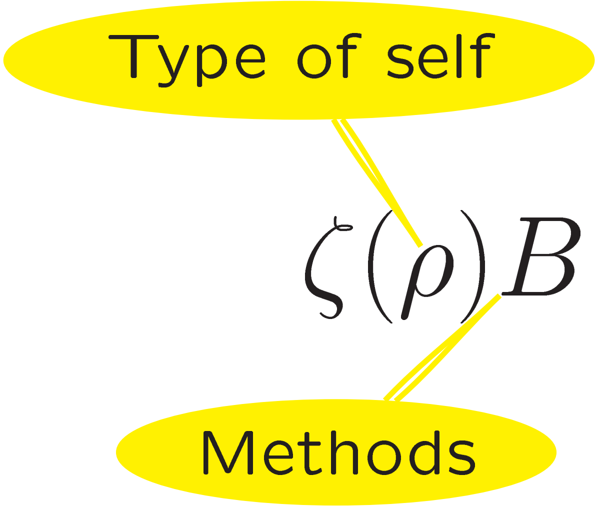
Labels ℓ are split into public m and private u.

$\tau ::= \alpha \mid [\rho]$	Object types
$\rho ::= \emptyset \mid \varrho \mid m : \widetilde{\tau}; \rho$	Row types
$B ::= \emptyset \mid \ell : \widetilde{\tau}; B$	Internal types
$\zeta(\rho)B$	Class types

Types expressions

Labels ℓ are split into public m and private u.

$\tau ::= \alpha \mid [\rho]$	Object types
$\rho ::= \emptyset \mid \varrho \mid m : \widetilde{\tau}; \rho$	Row types
$B ::= \emptyset \mid \ell : \widetilde{\tau}; B$	Internal types



$$\zeta(\rho)B$$

Class types

Types expressions

Labels ℓ are split into public m and private u.

$\tau ::= \alpha \mid [\rho]$	Object types
$\rho ::= \emptyset \mid \varrho \mid m : \widetilde{\tau}; \rho$	Row types
$B ::= \emptyset \mid \ell : \widetilde{\tau}; B$	Internal types
$\zeta(\rho) B^{W},$	Class types

Coupled labels

Types expressions

Labels ℓ are split into public m and private u.

$\tau ::= \alpha \mid [\rho]$	Object types
$\rho ::= \emptyset \mid \varrho \mid m : \tilde{\tau}; \rho$	Row types
$B ::= \emptyset \mid \ell : \tilde{\tau}; B$	Internal types

Virtual labels

$\zeta(\rho)B^{W,V}$	Class types

Types expressions

Labels ℓ are split into public m and private u.

$\tau ::= \alpha \mid [\rho]$	Object types
$\rho ::= \emptyset \mid \varrho \mid m : \widetilde{\tau}; \rho$	Row types
$B ::= \emptyset \mid \ell : \widetilde{\tau}; B$	Internal types
$\zeta(\rho) B^{W,V}$	Class types

Selected typing rules

Class

$$\frac{\begin{array}{c} A \vdash C :: \zeta(\rho)B^{W,V} \\ A + c : \forall\ \mathsf{Gen}(\rho, B, A).\,\zeta(\rho)B^{W,V} \vdash P \end{array}}{A \vdash \mathsf{class}\ c = C\ \mathsf{in}\ P}$$

Object

$$\frac{\begin{array}{c} A \vdash \mathsf{self}\,(x)\,C :: \zeta(\rho)B^{W,\emptyset} \\ A + x : \forall X.\,[\rho], x : \forall X.\,(B \upharpoonright \mathcal{F}) \vdash P \qquad A + x : \forall X.\,[\rho] \vdash Q \\ \rho = B \upharpoonright \mathcal{M} \qquad X = \mathsf{Gen}(\rho, B, A) \setminus ctv(B \upharpoonright W) \end{array}}{A \vdash \mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q}$$

The end

Further work

- More precise typechecking of synchronization could be explored
- It would be useful for selective refinement,
- but also, and more importantly, for the detection of synchronization errors.

Modularity of selective transformation

In the reaction $J \triangleright P$ of a class C:

- The process P is abstract (compiled with respect to the bound variables of J)
- The pattern J is concrete (it can be interpreted and modified).

Hence, the synchronization is entirely recompiled for each class.

Partitioning of acceptable states (principle)

An n-place buffer can be implemented with three states EMPTY(a), SOME(a), or FULL(a) (where e.g. a is itself an unsafe buffer structure)

In a subclass, one may wish to implement a method Get2 to retrieve two elements at once, atomically. For that purpose, the state SOME(a) must be partitioned into ONE(a) and MANY(a).

This can be done by

- Replacing messages SOME(a) by ONE(a) **or** MANY(a) in all patterns, and
- Transforming messages SOME(a) to either ONE(a) or MANY(a) (depending on the actual size of a) by adding a new reaction rule.

Partitioning of acceptable states (code)

```
class buffer2 =
  self(z)
    patch buffer with
      SOME(a) ⇒ ONE(a) or MANY(a) ▷ 0
    end
  or SOME(a) ▷
        if ( size (a) > 1) then z.MANY else z.ONE(a)
  or Get2(r) & (ONE(a) or MANY(a)) ▷
          let u = get(a) and v = get(a) in r.Return (u, v)
      &  if  size(a) = 0 then z.EMPTY(a) else z.SOME(a)
```

An operational account of privacy

We can instrument the semantics to observe privacy:

- An object x created from within the body of an object y is a descendant of x and has access to private methods of x.
- The parent relation defines a forest of objects. (The position of an object can be represented by the path from the object to the root.)
- Object keep tracks of their origins went their are send out after their creation.
- A message send from an object x to a private label of object y violates privacy if x is not a parent of y.

Well types programs cannot violate privacy.