Join Calculus and Algebraic Pattern Matching

Qin[tʃin] MA 马勤

Laboratory of Advanced Software SYstems
Faculty of Science Technology and Communication
University of Luxembourg

5 March, 2008

A Short Bio of Myself

- Sep. 2001:DEA, Université Paris 7 - INRIA Rocquencourt, France
- Jun. 2002:Master in Computer Engineering, Nanjing University, China
- Sep. 2005:Ph.D. in Computer Science, Université Paris 7 - INRIA Rocquencourt, France
- 2005 – 2007: research assistant in OFFIS (Institute for Information Technology), Germany
- since 2008: BFR post-doc in University of Luxembourg under the supervision of Prof. Pierre Kelsen

Research Interests: Process Calculus, Functional Programming, Object-Oriented Programming, Component Models, Model-Based System Engineering.

Outline

1. The Join Calculus
2. Algebraic Patterns and ML Matching
3. Algebraic Pattern Matching in Join
4. Compiling Pattern Matching in Join Patterns

Outline

1. The Join Calculus
2. Algebraic Patterns and ML Matching
3. Algebraic Pattern Matching in Join
4. Compiling Pattern Matching in Join Patterns

The Join Calculus

Join is a name-passing calculus developed at INRIA

- Parallel processes
- Communication channels
- Message synchronization

Join has advantages w.r.t. language implementation

- Locality: at most one receptor for each channel.
- More powerful and higher level message synchronization in terms of pattern matching.
- A strong functional flavor, and can be adapted to OO settings in a straightforward manner.

The Join Calculus

But after all, why from process calculus to programming language?

- formally defined syntax and semantics
- sound type systems
- and an appropriate equivalence relation which allows reasoning about systems in terms of mathematical proofs.

Impacts of Join on Concurrent/Distributed Programming

- As language extensions:
    - JoCaml = Join + OCaml
    - JoinJava = Join + Java
    - Polyphonic $C^\sharp$ ($C\omega$) = Join + $C^\sharp$
- As concurrency libraries:
    - Scala
    - .Net
    - C++
- Taught in courses for concurrent computing, distributed computing, mobile computing, wide area computing, global computing ...

A One-Place Buffer in Join

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 in Empty() ...
```

join definitions	guarded processes
reaction rules	defined channels
join patterns	asynchronous message sending

Note: & is overloaded to denote both channel synchronization and process parallel composition.

A One-Place Buffer in Join

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 in Empty() ...
```

join definitions	guarded processes
reaction rules	defined channels
join patterns	asynchronous message sending

Note: & is overloaded to denote both channel synchronization and process parallel composition.

A One-Place Buffer in Join

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 in Empty() ...
```

join definitions	guarded processes
reaction rules	defined channels
join patterns	asynchronous message sending

Note: & is overloaded to denote both channel synchronization and process parallel composition.

A One-Place Buffer in Join

$$\begin{array}{ll} \textbf{def} & put(n)\ \&\ Empty() \ \triangleright\ Some(n) \\ \textbf{or} & get(r)\ \&\ Some(n) \ \triangleright\ r(n)\ \&\ Empty() \\ \textbf{in} & Empty()\ \ldots \end{array}$$

join definitions	guarded processes
reaction rules	defined channels
join patterns	asynchronous message sending

Note: & is overloaded to denote both channel synchronization and process parallel composition.

A One-Place Buffer in Join

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 in Empty() ...
```

join definitions	guarded processes
reaction rules	defined channels
join patterns	asynchronous message sending

Note: & is overloaded to denote both channel synchronization and process parallel composition.

A One-Place Buffer in Join

$$
\begin{array}{rl}
\textbf{def} & put(n)\ \&\ Empty() \ \triangleright\ Some(n) \\
\textbf{or} & get(r)\ \&\ Some(n) \ \triangleright\ r(n)\ \&\ Empty() \\
\textbf{in} & Empty()\ \ldots
\end{array}
$$

join definitions	guarded processes
reaction rules	defined channels
join patterns	asynchronous message sending

Note: & is overloaded to denote both channel synchronization and process parallel composition.

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
```

```
in Empty() & put(1) & get(reply)
```

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
in Empty() & put(1) & get(reply)
```

≡ put(1) & Empty() & get(reply)

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
in Empty() & put(1) & get(reply)
```

$\equiv$ put(1) & Empty() & get(reply)

$\longrightarrow$ Some(1) & get(reply)

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
in Empty() & put(1) & get(reply)
```

$\equiv$ put(1) & Empty() & get(reply)

$\longrightarrow$ Some(1) & get(reply)

$\equiv$ get(reply) & Some(1)

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
in Empty() & put(1) & get(reply)
```

$\equiv$ put(1) & Empty() & get(reply)

$\longrightarrow$ Some(1) & get(reply)

$\equiv$ get(reply) & Some(1)

$\longrightarrow$ reply(1) & Empty()

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
in Empty() & put(1) & get(reply)
```

$\equiv$ put(1) & Empty() & get(reply)

$\longrightarrow$ Some(1) & get(reply)

$\equiv$ get(reply) & Some(1)

$\longrightarrow$ reply(1) & Empty()

$\longrightarrow$ print_int(1) & Empty()

A Computation: Intuition

```
def put(n) & Empty() ▷ Some(n)
 or get(r) & Some(n) ▷ r(n) & Empty()
 ......
def reply(n) ▷ print_int(n)
in Empty() & put(1) & get(reply) & put(2)
```

$\equiv$ put(1) & Empty() & get(reply)

$\longrightarrow$ Some(1) & get(reply)

$\equiv$ get(reply) & Some(1)

$\longrightarrow$ reply(1) & Empty()

$\longrightarrow$ print_int(1) & Empty()

Join-Pattern Matching Is Nondeterministic

```
def a() & b() ▷ P
 or a() & c() ▷ Q
in b() & c()
```

Join-Pattern Matching Is Nondeterministic

```
def a() & b() ▷ P
 or a() & c() ▷ Q
in  b() & c() & a()
```

Join-Pattern Matching Is Nondeterministic

```
def a() & b() ▷ P
 or a() & c() ▷ Q
in b() & c() & a()
```

Nondeterministic choice
between launching P and launching Q ! May even vary from run to run!

Join-Pattern Matching Is Nondeterministic

```
def a() & b() ▷ P
 or a() & c() ▷ Q
in  b() & c() & a()
```

Nondeterministic choice between launching P and launching Q ! May even vary from run to run!

- A join pattern is matched $\Longrightarrow$ there are messages sent to all its channels.
- All such non-determinism complies to the concurrency nature of the calculus, and should be respected when programming.

Join-Pattern Matching Is Nondeterministic

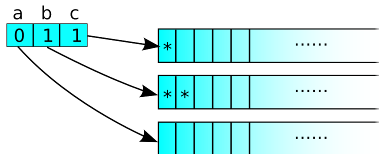
```
def a() & b() ▷ P
 or a() & c() ▷ Q
in  b() & c() & a()
```

Nondeterministic choice between launching P and launching Q ! May even vary from run to run!

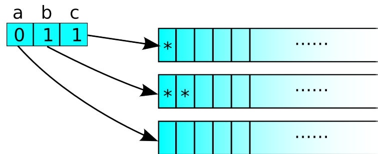
- A join pattern is matched $\Longrightarrow$ there are messages sent to all its channels.
- All such non-determinism complies to the concurrency nature of the calculus, and should be respected when programming.

Join Definitions Implemented as Automata in JoCaml

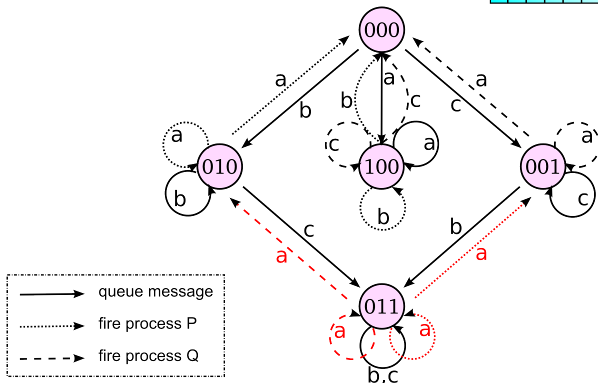
$$\begin{array}{ll}\textbf{def} & a() \;\&\; b() \triangleright P \\ \textbf{or} & a() \;\&\; c() \triangleright Q\end{array}$$



Join Definitions Implemented as Automata in JoCaml

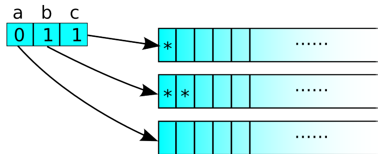


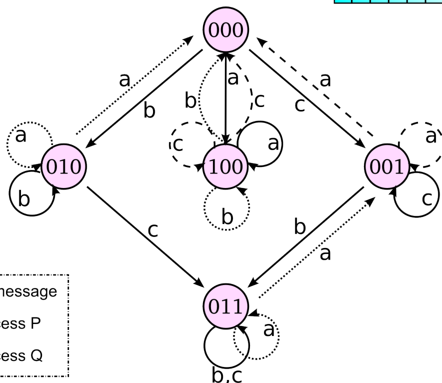
def a ( ) & b ( ) ▷ P
or a ( ) & c ( ) ▷ Q



Join Definitions Implemented as Automata in JoCaml

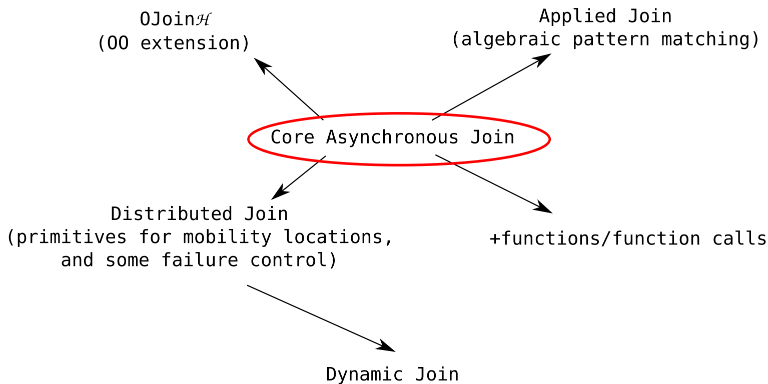
```
def  a() & b() ▷ P
 or  a() & c() ▷ Q
```



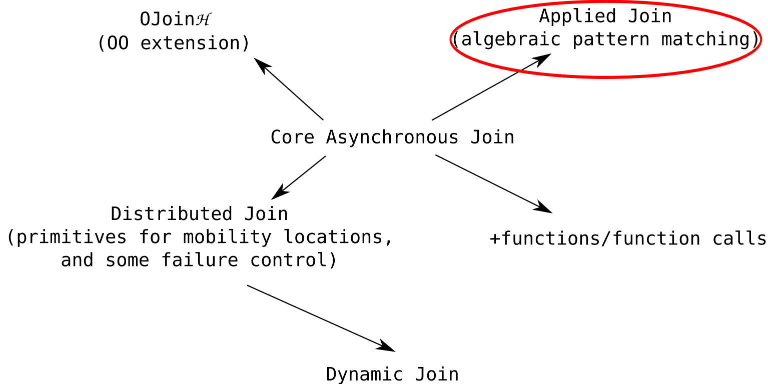


——→	queue message
·······→	fire process P
– – –→	fire process Q

Join Evolution Diagram



Join Evolution Diagram



Outline

1. The Join Calculus
2. Algebraic Patterns and ML Matching
3. Algebraic Pattern Matching in Join
4. Compiling Pattern Matching in Join Patterns

Algebraic Data Types and Patterns

Integer Lists and Patterns in OCaml:

```
type list = [] | int :: list
```

Algebraic Data Types and Patterns

Integer Lists and Patterns in OCaml:

```
type list = [] | int :: list
```

values:

```
[]
1::[]
1::2::[]
...
```

Algebraic Data Types and Patterns

Integer Lists and Patterns in OCaml:

```
type list = [] | int :: list
```

values:

```
[]
1::[]
1::2::[]
...
```

patterns:

```
[]
x::[]
1::xs
x
...
```

Algebraic Data Types and Patterns

Integer Lists and Patterns in OCaml:

```
type list = [] | int :: list
```

values:

```
[]
1::[]
1::2::[]
...
```

patterns:

```
[]
x::[]
1::xs
x
...
```

- Value 1::[] is an instance of both pattern x::[] and 1::xs.
- Any value is an instance of pattern x.

Algebraic Pattern Matching à la ML

```
let isEmpty ls =
  match ls with
      | [] -> True
      | hd::tl -> False
```

Algebraic Pattern Matching à la ML

```
let isEmpty ls =
  match ls with
     | [] -> True
     | hd::tl -> False
```

Deterministic first match policy:

Algebraic Pattern Matching à la ML

```
let isEmpty ls =
  match ls with
     | [] -> True
     | hd::tl -> False
         x
```

Deterministic first match policy:
Although [] is an instance of both [] and x, the preceding pattern [] always has privilege to match, so only non-empty lists will be matched by the second pattern.

Outline

1. The Join Calculus
2. Algebraic Patterns and ML Matching
3. Algebraic Pattern Matching in Join
4. Compiling Pattern Matching in Join Patterns

ML Pattern Matching in Processes

A stack in applied join:

```
def  push(n) & Empty() ▷Some([n])
 or  push(n) & Some(ls) ▷ Some(n::ls)
 or  pop(r) & Some(ls) ▷ match ls with
                         | [x] -> r(x) & Empty()
                         | x::xs -> r(x) & Some(xs)
in Empty() ...
```

ML Pattern Matching in Processes

A stack in applied join:

```
                 Some([])?
def  push(n) & Empty() ▷Some([n])
 or  push(n) & Some(ls) ▷ Some(n::ls)
 or  pop(r) & Some(ls) ▷ match ls with
                       | [x] -> r(x) & Empty()
                       | x::xs -> r(x) & Some(xs)
in  Empty() ...
```

ML Pattern Matching in Processes

A stack in applied join:

```
                Some([])? Not yet supported!
def  push(n) & ~~Empty()~~ ▷Some([n])
 or  push(n) & Some(ls) ▷ Some(n::ls)
 or  pop(r) & Some(ls) ▷ match ls with
                       | [x] -> r(x) & Empty()
                       | x::xs -> r(x) & Some(xs)
in  Empty() ...
```

Join Patterns with Pattern Arguments

Another stack in applied join:

```
def  push(n) & State(ls) ▷State(n::ls)
 or  pop(r) & State(x::xs) ▷r(x) & State(xs)
in  State([]) ...
```

Observations:

- Two versions of stack express the same behavior.
- The second version requires less programming effort.

Applied Join Calculus = Join + Algebraic Value Passing + Algebraic Pattern Matching

Expressions and Patterns:

$$e ::= x \mid \kappa(e_1, e_2, \ldots, e_n) \qquad \pi ::= x \mid \kappa(\pi_1, \pi_2, \ldots, \pi_n)$$

Join Definitions and Join Patterns:

$$D ::= \top \mid J \triangleright P \mid D_1 \textbf{ or } D_2 \qquad J ::= x(\pi) \mid J_1 \,\&\, J_2$$

Process:

$$P ::= 0 \mid x(e) \mid P_1 \,\&\, P_2 \mid \textbf{def } D \textbf{ in } P \mid \textbf{match } e \textbf{ with } \mid \pi_1 \to P_1 \mid \ldots \mid \pi_m \to P_m$$

Semantics Is for Closed Terms

- Two kinds of values:
  - channel names;
  - algebraic values.
- Two kinds of variables:
  - variables of channel type are treated as values, i.e. channel names;
  - only variables of algebraic data types are real ones, which would be bound to algebraic values.
- We define closed terms as terms with only free channel names i.e. all free variables are of channel types.
- We define reduction semantics only for closed terms.

Reduction Rules for Algebraic Pattern Matching

$(\textbf{match } v \textbf{ with } \mid \pi_1 \rightarrow P_1 \mid \ldots \mid \pi_m \rightarrow P_m) \longrightarrow P_i\sigma$

- v is an instance of π_i.
- v is not an instance of any pattern before π_i.
- $P_i\sigma$ is a copy of P_i.

```
def  ...
 or  x1(π1) & ... & xn(πn)  ▷P        ⟶        Pσ1 ... σn
 or  ...
in  x1(v1) & ... & xn(vn)
```

- v_i is an instance of π_i, for all i such that $1 < i < n$.
- $P\sigma_1 \ldots \sigma_n$ is a copy of P.

Outline

1. The Join Calculus
2. Algebraic Patterns and ML Matching
3. Algebraic Pattern Matching in Join
4. Compiling Pattern Matching in Join Patterns

Refine and Dispatch

$$
\begin{array}{rl}
\textbf{def} & a() \;\&\; c(\pi_1) \;\triangleright \ldots \\
\textbf{or} & b() \;\&\; c(\pi_2) \;\triangleright \ldots
\end{array}
$$

Refine and Dispatch

$$\begin{array}{ll}\textbf{def} & a() \ \& \ c(\pi_1) \ \triangleright \ldots \\ \textbf{or} & b() \ \& \ c(\pi_2) \ \triangleright \ldots\end{array}$$

$$\begin{array}{ll}\textbf{def} & a() \ \& \ c_{\pi_1}(x_1) \ \triangleright \ldots \\ \textbf{or} & b() \ \& \ c_{\pi_2}(x_2) \ \triangleright \ldots\end{array}$$

Refine and Dispatch

```
def a() & c(π1) ▷...
 or b() & c(π2) ▷...
```

```
def a() & cπ1(x1) ▷...
 or b() & cπ2(x2) ▷...
 or c(x) ▷ match x with
              | π1 -> cπ1(x)
              | π2 -> cπ2(x)
```

Refine and Dispatch

```
def a() & c(π1) ▷...
 or b() & c(π2) ▷...
```

```
def a() & c_π1(x1) ▷...
 or b() & c_π2(x2) ▷...
 or c(x) ▷ match x with
            | π1 -> c_π1(x)
            | π2 -> c_π2(x)
```

This naive transformation does not work because of the nondeterminism-determinism gap between join pattern matching and ML pattern matching.

Compatible Algebraic Patterns

Definition

Two patterns are compatible if they share common instance.

Property

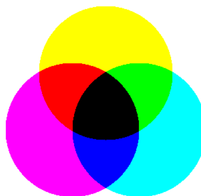
Two compatible patterns admit a least upper bound written $\pi_1 \uparrow \pi_2$, whose instance set is the intersection of the original two.

Eg. $(x :: [\,]) \uparrow (1 :: xs) = 1 :: [\,]$

Note: least upper bounds can be computed at the same time when compatibility is checked. And there already are efficient algorithms for these purposes implemented in the Jocaml compiler.

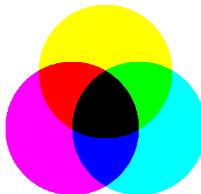
Pattern Lattice

$$
\begin{array}{lllll}
\textbf{def} & \mathsf{c}(\pi_1) & \& & \mathsf{a}() & \triangleright P_1 \\
\textbf{or} & \mathsf{c}(\pi_2) & \& & \mathsf{b}() & \triangleright P_2 \\
\textbf{or} & \mathsf{c}(\pi_3) & \& & \mathsf{d}() & \triangleright P_3
\end{array}
$$



Pattern Lattice

$$
\begin{array}{llllll}
\textbf{def} & c(\pi_1) & \& & a() & \triangleright P_1 \\
\textbf{or} & c(\pi_2) & \& & b() & \triangleright P_2 \\
\textbf{or} & c(\pi_3) & \& & d() & \triangleright P_3
\end{array}
$$



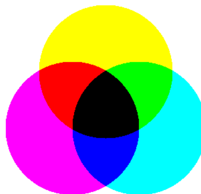
$\pi_1 \uparrow \pi_2 \uparrow \pi_3$

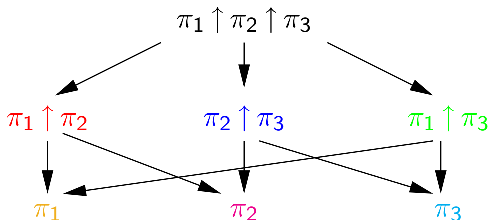
$\pi_1 \uparrow \pi_2$ $\pi_2 \uparrow \pi_3$ $\pi_1 \uparrow \pi_3$

π_1 π_2 π_3

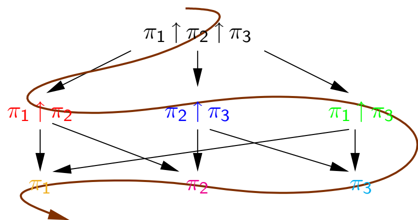
Pattern Lattice

$$\begin{array}{llll} \textbf{def} & c(\pi_1) & \& \ a() & \triangleright P_1 \\ \textbf{or} & c(\pi_2) & \& \ b() & \triangleright P_2 \\ \textbf{or} & c(\pi_3) & \& \ d() & \triangleright P_3 \end{array}$$





A Topological Order



or c(x) $\triangleright$ **match** x **with**

| $\pi_1 \uparrow \pi_2 \uparrow \pi_3 \rightarrow c_{\pi_1 \uparrow \pi_2 \uparrow \pi_3}(x)$

| $\pi_1 \uparrow \pi_2 \rightarrow c_{\pi_1 \uparrow \pi_2}(x)$

| $\pi_2 \uparrow \pi_3 \rightarrow c_{\pi_2 \uparrow \pi_3}(x)$

| $\pi_1 \uparrow \pi_3 \rightarrow c_{\pi_1 \uparrow \pi_3}(x)$

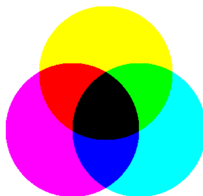
| $\pi_3 \rightarrow c_{\pi_3}(x)$

| $\pi_2 \rightarrow c_{\pi_2}(x)$

| $\pi_1 \rightarrow c_{\pi_1}(x)$

Disjunction of Refined Channels

$$
\begin{array}{llllll}
\textbf{def} & c(\pi_1) & \& & a() & \triangleright P_1 \\
\textbf{or} & c(\pi_2) & \& & b() & \triangleright P_2 \\
\textbf{or} & c(\pi_3) & \& & d() & \triangleright P_3
\end{array}
$$



$$
\begin{array}{lllllllll}
c(\pi_1): & c_{\pi_1}(x_1) & \textbf{or} & c_{\pi_1\uparrow\pi_2}(x_1) & \textbf{or} & c_{\pi_1\uparrow\pi_3}(x_1) & \textbf{or} & c_{\pi_1\uparrow\pi_2\uparrow\pi_3}(x_1) \\
c(\pi_2): & c_{\pi_2}(x_2) & \textbf{or} & c_{\pi_1\uparrow\pi_2}(x_2) & \textbf{or} & c_{\pi_2\uparrow\pi_3}(x_2) & \textbf{or} & c_{\pi_1\uparrow\pi_2\uparrow\pi_3}(x_2) \\
c(\pi_3): & c_{\pi_3}(x_3) & \textbf{or} & c_{\pi_1\uparrow\pi_3}(x_3) & \textbf{or} & c_{\pi_2\uparrow\pi_3}(x_3) & \textbf{or} & c_{\pi_1\uparrow\pi_2\uparrow\pi_3}(x_3)
\end{array}
$$

Note: $(J_1 \textbf{ or } J_2) \,\&\, J \triangleright P \stackrel{\text{def}}{=} J_1 \,\&\, J \triangleright P \textbf{ or } J_2 \,\&\, J \triangleright P$

Concurrent Stack after Transformation

Before

```
def push(n) & State(ls) ▷ State(n::ls)
 or pop(r) & State(x::xs) ▷
                         r(x) & State(xs)
```

x::xs

ls

After

```
def push(n) & (State_x::xs(x1) or State_ls(x1)) ▷
                                   ... State(n::ls)
 or pop(r) & State_x::xs(x2) ▷
                               ... r(x) & State(xs)
---------------------------------------------------
 or State(x) ▷ match x with
               | x::xs -> State_x::xs(x)
               | ls -> State_ls(x)
```

Concurrent Stack after Transformation

Before

```
def push(n) & State(ls) ▷ State(n::ls)
 or pop(r) & State(x::xs) ▷
                          r(x) & State(xs)
```

x::xs

ls

After

```
def push(n) & (State_x::xs(x1) or State_ls(x1)) ▷
                      match x1 with ls -> State(n::ls)
 or pop(r) & State_x::xs(x2) ▷
                 match x2 with x::xs -> r(x) & State(xs)
--------------------------------------------------------
 or State(x) ▷ match x with
               | x::xs -> State_x::xs(x)
               | ls -> State_ls(x)
```

Compilation Scheme $[\![\cdot]\!]$

- Y_c: to transform one join definition D w.r.t. channel c defined in D.
- $[\![\cdot]\!]$: to compile away pattern arguments in processes:

$$
\begin{array}{rcl}
[\![0]\!] & \stackrel{\text{def}}{=} & 0 \\
[\![x(e)]\!] & \stackrel{\text{def}}{=} & x(e) \\
[\![P_1 \,\&\, P_2]\!] & \stackrel{\text{def}}{=} & [\![P_1]\!] \,\&\, [\![P_2]\!] \\
[\![\mathbf{def}\ D\ \mathbf{in}\ P]\!] & \stackrel{\text{def}}{=} & \mathbf{def}\ Y_{c_n} \ldots Y_{c_1}([\![D]\!])\ \mathbf{in}\ [\![P]\!] \\
[\![\mathbf{match}\ e\ \mathbf{with}\ \ |^{i \in I}\ \pi_i \rightarrow P_i]\!] & \stackrel{\text{def}}{=} & \mathbf{match}\ e\ \mathbf{with}\ \ |^{i \in I}\ \pi_i \rightarrow [\![P_i]\!] \\
 & & \\
[\![\top]\!] & \stackrel{\text{def}}{=} & \top \\
[\![J \triangleright P]\!] & \stackrel{\text{def}}{=} & J \triangleright [\![P]\!] \\
[\![D_1\ \mathbf{or}\ D_2]\!] & \stackrel{\text{def}}{=} & [\![D_1]\!]\ \mathbf{or}\ [\![D_2]\!]
\end{array}
$$

where we suppose D defines channels $c_1, \ldots, c_n$.

Equivalence relations

Weak barbed congruence ($\approx$): between closed processes

$\approx$ is the largest equivalence relation that meets the following:

- $\approx$ is a weak reduction bisimulation;
- $\approx$ preserves weak barbs;
- $\approx$ is closed under evaluation contexts.

Static equivalence ($\risingdotseq$): open extension of $\approx$

$P \risingdotseq Q \iff \forall\sigma.$ s.t. $P\sigma$ and $Q\sigma$ closed, $P\sigma \approx Q\sigma$.

Equivalence relations, cont.

Theorem

The static equivalence $\doteq$ is a full congruence.

Reflexive, commutative, and transitive.

Evaluation contexts:

$$
\begin{array}{lll}
P_1 \doteq P_2 & \Longrightarrow & P \mathbin{\&} P_1 \doteq P \mathbin{\&} P_2 \\
P_1 \doteq P_2 & \Longrightarrow & P_1 \mathbin{\&} P \doteq P_2 \mathbin{\&} P \\
P_1 \doteq P_2 & \Longrightarrow & \mathbf{def}\ D\ \mathbf{in}\ P_1 \doteq \mathbf{def}\ D\ \mathbf{in}\ P_2 \\
P_1 \doteq P_2 & \Longrightarrow & \mathbf{match}\ e\ \mathbf{with}\ \ldots \mid \pi_k \rightarrow P_1 \mid \ldots \\
 & & \doteq \mathbf{match}\ e\ \mathbf{with}\ \ldots \mid \pi_k \rightarrow P_2 \mid \ldots
\end{array}
$$

Guarded contexts:

$$
\begin{array}{lll}
P_1 \doteq P_2 & \Longrightarrow & \mathbf{def}\ D\ \mathbf{or}\ J \triangleright P_1\ \mathbf{in}\ P \doteq \mathbf{def}\ D\ \mathbf{or}\ J \triangleright P_2\ \mathbf{in}\ P \\
P_1 \doteq P_2 & \Longrightarrow & \mathbf{def}\ J \triangleright P_1\ \mathbf{or}\ D\ \mathbf{in}\ P \doteq \mathbf{def}\ J \triangleright P_2\ \mathbf{or}\ D\ \mathbf{in}\ P
\end{array}
$$

Correctness in Terms of Equivalence

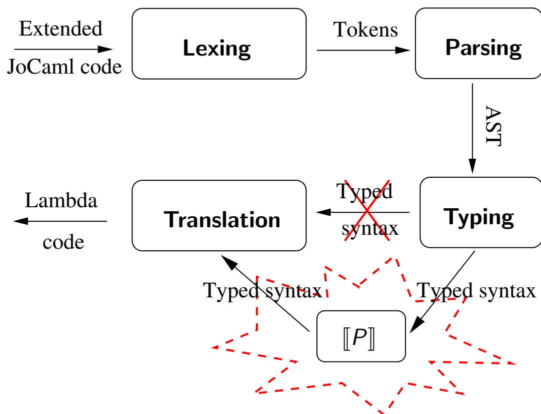
Lemma (Y_c is correct)

For any join definition D, channel name c defined in D, and process P,
$\textbf{def}\ Y_c(D)\ \textbf{in}\ P \doteqdot \textbf{def}\ D\ \textbf{in}\ P.$

Theorem (Correctness)

For any process P, $[\![P]\!] \doteqdot P.$

Extended JoCaml Compiler Front End



Summary of Contributions

- General Motivation:
  - theoretical foundation for languages in concurrent settings;
  - multi-paradigm programming, language feature interaction.
- Applied Join: Join with algebraic data types.
  - Algebraic pattern arguments.
  - An equivalence theory for open terms.
- A compilation scheme from Applied Join to Join.
  - The nondeterminism problem by pattern lattice.
  - Compilation correctness in terms of process equivalence.
  - Implementation in JoCaml.

Cf.

- *Algebraic Pattern Matching in Join Calculus*, Q. Ma and L. Maranget, to appear in Journal of Logical Methods in Computer Science, 2008.
- http://jocaml.inria.fr

Future Work