Concurrent Systems with JoCaml

- Programming with *join patterns* as coordination abstractions can be accomplished in the JoCaml programming language (Fessant, 1998; Conchon and Fessant, 1999; Mandel and Maranget, 2008).

- Join patterns can also be used as abstractions in other high-level programming languages, for example, Cω is an extension of C# with join definitions.

- JoCaml is an extension of the Objective Caml language (Leroy et al., 2008), itself an object-oriented dialect of the ML functional programming language (Milner et al., 1990).

- JoCaml was developed to model concurrency and distribution using join calculus abstractions.
Join Calculus and JoCaml

The most significant semantic differences between JoCaml and the join calculus are:

- **Synchronous and asynchronous channels**
  - While the join calculus exclusively models asynchronous communication, JoCaml supports the notion of *synchronous* channels, which are modeled as asynchronous communication channels with an implicit *continuation*, in the *continuation-passing style* of functional programming.

- A special *reply* primitive in join definitions and semi-colon (;) in otherwise asynchronous message sending processes are used to create an implicit continuation and to send a value over it.
Join Calculus and JoCaml

- Pattern matching
  - The join calculus admits only a limited form of patterns, where *formal arguments* on join definitions get matched by *actual arguments* in molecule solutions using simple substitutions of names to names.
  - JoCaml, being an extension of Objective Caml, inherits its extensive pattern matching capabilities.
  - Join patterns therefore can use arbitrarily complex Objective Caml patterns in its join definitions. However, join patterns remain non-deterministic, while Objective Caml patterns are deterministic.
Join Calculus and JoCaml

- **Distribution and mobility**
  
  - JoCaml follows the distribution and mobility model of the distributed join calculus (Fournet et al., 1996).
  
  - This includes the notion of a tree of locations with creation, migration, termination, and failure detection capabilities.
Join Calculus and JoCaml

A summary of the key join calculus primitives and the equivalent JoCaml programming language syntax follows:

<table>
<thead>
<tr>
<th>Join Calculus</th>
<th>JoCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \langle y \rangle$</td>
<td>spawn $x(y)$</td>
</tr>
<tr>
<td>$P \mid Q$</td>
<td>$p &amp; q$</td>
</tr>
<tr>
<td>def $x \langle u \rangle \mid y \langle v \rangle \triangleright P \text{ in } Q$</td>
<td>def $x(u) &amp; y(v) = p \text{ in } q$</td>
</tr>
<tr>
<td>$D_1 \land D_2$</td>
<td>$d1 \text{ or } d2$</td>
</tr>
</tbody>
</table>
Join Calculus and JoCaml

- A join calculus atom is a JoCaml expression resulting from spawning a process, using the `spawn` keyword.

- A molecule, or parallel composition of processes $P$ and $Q$ in the join calculus, is written as $p \& q$ in JoCaml.

- Join calculus definitions use join patterns that look like atom and molecule patterns in JoCaml, such as $x(u) \& y(v)$.

- These definitions can be composed in JoCaml using the `or` operator (equivalent to using $\wedge$ in the join calculus.) JoCaml also supports an `and` operator for definitions of pairwise disjoint channel names.
Example

For example, consider the join calculus expression:

\[
\text{def } \text{ready} \langle \text{printer} \rangle \ | \ \text{job} \langle \text{file} \rangle \triangleright \text{printer} \langle \text{file} \rangle \ \text{in} \ \text{ready} \langle \text{laser} \rangle \ | \ \text{job} \langle f1 \rangle
\]

An equivalent JoCaml program is written as follows:

\[
\begin{array}{l}
\text{def ready(printer) & job(file) = printer(file) in} \\
\quad \text{spawn ready(laser) & job(f1)}
\end{array}
\]
**JoCaml Programming Language Syntax**

\[ \text{Process} ::= \]

\[ \begin{align*}
0 \\
\text{Process} & \quad \& \quad \text{Process} \\
\text{Expr}_1 & \quad \text{Expr}_2 \\
\text{Expr} & \quad ; \quad \text{Process} \\
\text{reply} \quad [\text{Expr}] \quad \text{to} \quad \text{Id} \\
\text{if} \quad \text{Expr} \quad \text{then} \quad \text{Process} \\
\quad \text{else} \quad \text{Process} \\
\text{match} \quad \text{Expr} \quad \text{with} \\
\quad (\mid \text{OPattern} \quad \text{[when} \quad \text{Expr}] \\
\quad \quad \rightarrow \quad \text{Process})^+ \\
\text{let} \quad [\text{rec}] \quad \text{OBinding} \\
\quad (\text{and} \quad \text{OBinding})^* \quad \text{in} \quad \text{Process} \\
\text{for} \quad \text{Id} \quad = \quad \text{Expr} \\
\quad (\text{to} \mid \text{downto}) \quad \text{Expr} \\
\quad \text{do} \quad \text{Process} \quad \text{done} \\
\text{JoinDef} \quad \text{in} \quad \text{Process}
\end{align*} \]

- **Empty process**
- **Parallel composition**
- **Asynchronous send**
- **Sequencing**
- **Reply to synchronous send**
- **Conditional**
- **Pattern matching**
- **Local definition**
- **Concurrent loop**
- **Local channel definition**
JoCaml Programming Language Syntax

\[
\begin{align*}
Expr & ::= \quad \text{Expression} \\
& \quad \text{Objective Caml expression} \\
& \quad \text{Asynchronous execution} \\
& \quad \text{Local channel definition} \\
\text{OExpression} & ::= \quad \text{OExpression} \\
\text{spawn Process} & ::= \quad \text{spawn Process} \\
\text{JoinDef in Expr} & ::= \quad \text{JoinDef in Expr}
\end{align*}
\]

\[
\begin{align*}
\text{JoinDef} & ::= \quad \text{Join Definition} \\
& \quad \text{def ReactionSet} \\
& \quad (\text{and ReactionSet})^* \\
\text{ReactionSet} & ::= \quad \text{Reaction (or Reaction)}^* \\
\text{Reaction} & ::= \quad \text{JoinPattern } = \text{Process} \\
\text{JoinPattern} & ::= \quad \text{Channel (\& Channel)}^* \\
\text{Channel} & ::= \quad \text{Id (OPattern)}
\end{align*}
\]
JoCaml Programming Language Syntax

• JoCaml inherits Objective Caml’s operations on values of primitive types.

• JoCaml extends the primitive types of Objective Caml with an asynchronous channel type: Join.chan.

• The main non-terminal syntax extensions to OCaml are expressions, processes, and join definitions.
**JoCaml Programming Language Syntax**

*Expressions* are extended with process spawning.

- Process spawning as an expression evaluates to () and executes the process given as an argument asynchronously.
- *Processes* are executed producing no result.
- The empty process and parallel composition are denoted 0 and p & q respectively.
- The expression e1 e2 evaluates e1 to an asynchronous channel and e2 to a value and sends the value to the channel.
- The process e; p evaluates e first, then it executes p.
- The expression e must be of unit type.
- The expression reply e to c sends the value of e as a reply over synchronous channel c.
JoCaml Programming Language Syntax

• Conditional expressions enable guarding a process by a boolean expression.

• Pattern matching and local definitions have the same name-value binding behavior as the Objective Caml counterparts but on processes rather than on expressions.

• Loop expressions execute iterations concurrently.

• Local channels can be defined inside both expressions and processes.
Join definitions specify the creation of channels, their binding to names, and the process to execute when reactions are enabled.

- A join definition may specify one or more disjoint sets of reactions (separated by the and operator.) That is, the sets of channel names defined by each set of reactions must be pairwise disjoint.

- Each reaction set consists of a group of reactions (separated by the or keyword) that may share defined names, thereby creating a need for synchronization. However, a channel name may not be declared more than once in a single reaction’s join pattern.

- A reaction consists of a join pattern and a process. When the join pattern is matched by incoming messages on defined channels, the guarded process may be executed.
JoCaml Programming Language Syntax

- If multiple join patterns are matched, only one of the guarded processes is chosen nondeterministically and executed.
- If the guarded process includes a `reply e to c` construct, then the defined channel `c` is synchronous.
- *Join patterns* are equivalent to the molecule patterns in the join calculus, except that arbitrary patterns from Objective Caml may be used in the matching process.

The *OPattern* and *OBinding* non-terminals are inherited from Objective Caml, and refer to patterns and let bindings respectively.
JoCaml Programming Language Syntax

JoCaml also inherits syntactic sugar from Objective Caml as follows:

\[
\text{begin } Process \text{ end} \triangleq Process
\]
\[
( Process ) \triangleq Process
\]
\[
Expr_1 (Expr_2) \triangleq Expr_1 Expr_2
\]

- Processes may be surrounded by parentheses or begin and end delimiters for clarity.
- Furthermore, asynchronous communication can use the more familiar function invocation syntax from functional programming.
Objective Caml Pattern Matching in Join Patterns

A key feature of JoCaml is its ability to use Objective Caml patterns inside join definition patterns. For example, to merge two sorted lists, you may use the following definition:

```ocaml
def merge([],[]) = reply [] to merge
or merge(xs,[]) = reply xs to merge
or merge([],xs) = reply xs to merge
or merge(x::xs,y::ys) =
    if x>y then reply x::merge(xs,y::ys) to merge
    else reply y::merge(x::xs,ys) to merge

in
spawn merge([1;3;4],[2;3]) ;;
```
Objective Caml Pattern Matching in Join Patterns

The non-determinism semantics inherent in join calculus patterns is respected in JoCaml. Consider for example:

```ocaml
def c([]) = echo_string "Nil"

or  c(_) = echo_string "Anything"

in

  spawn c([]) ;;
```

The execution is free to print Nil or Anything. Note there is no requirement of *fairness* in selecting among multiple matching patterns over time.
Objective Caml Pattern Matching in Join Patterns

Objective Caml patterns on the other hand are resolved deterministically: the first textual pattern to match will always be chosen. Consider for example:

```ocaml
def c(x) =
    match x with
    | [] -> echo_string "Nil"
    | _  -> echo_string "Anything"
    in
    spawn c([]);;
```

Any valid execution must always print Nil.
**π Calculus Channels Example**

π calculus channels can be modeled in JoCaml as follows:

```ocaml
def let new_pi_channel () =
  def send(x) & receive() = reply x to receive in
  send, receive ;;
```

For example, the π calculus expression:

$$(\nu c)(\nu d)(\overline{c}_1 \mid \overline{c}_2 \mid c(x).\overline{d}(x + x) \mid d(y).\overline{py})$$

can then be written in JoCaml as follows:

```ocaml
spawn begin
  let sc, rc = new_pi_channel ()
  and sd, rd = new_pi_channel () in
  sc(1) & sc(2) & (let x = rc() in sd(x+x)) &
  (let y = rd() in print_int y; 0)
end ;;
```

This code can print 2 or 4 depending on which process communicates first: $\overline{c}_1$ or $\overline{c}_2$. 
**π Calculus Channels Example**

Synchronous π calculus channels can easily be modelled by making send synchronous:

```plaintext
let new_sync_pi_channel () =
  def send(x) & receive() = reply x to receive &
  reply to send
in
  send, receive ;;
```

The following π calculus expression:

\[
(\nu c)(\overline{ca}.P \mid c(x).Q)
\]

can be modeled as follows:

```plaintext
spawn begin
  let sc, rc = new_sync_pi_channel () in
  (sc(1); print_string "Send 1 succeeded"; 0) &
  (let x = rc() in print_int x; print_string " received"; 0)
end ;;
```
Reference Cell in JoCaml

A reference cell in JoCaml can be written as follows:

```plaintext
let cell c0 =
  def content(c) & get() = content(c) & reply c to get
  or content(_) & set(c) = content(c) & reply to set in
  spawn content(c0) ;
(get, set) ;;
```

The JoCaml cell client code can be written as follows:

```plaintext
let cg, cs = cell 0 ;;

print_int (cg()) ;;
cs(5) ;;
print_int (cg()) ;;
```

The style of putting together all the methods to access an encapsulated data structure, is reminiscent of *object-based programming*.
Join Continuations and Concurrent Iterators

Barrier synchronization enables concurrent activities to be synchronized on finishing execution so that another new activity only happens after they are all finished. For example, two concurrent activities can be barrier-synchronized in JoCaml as follows:

```caml
def join1() & join2() = reply to join1 & reply to join2
in
spawn begin
  (print_string "(" ; join1() ; print_string "a" ; join1() ;
   print_string ")" ; 0) &
  (join2() ; print_string "b" ; join2() ; 0)
end ;;
```

The code when executed prints (ab) or (ba).
Join Continuations and Concurrent Iterators

Waiting for $n$ concurrent events to complete can be coded as follows:

```python
def count(n) & tick() = count(n-1)
or  count(0) & wait() = reply to wait
;;
```

In this code, `tick` is used to notify completion of an event, and `wait` is used to be notified of completion of all events. The code assumes that at most $n$ ticks are ever sent, otherwise, the first join pattern (`count(n) & tick()`) could fire on the $n+1^{th}$ tick, preventing the second join pattern (`count(0) & wait()`) from ever being fired.
Join Continuations and Concurrent Iterators

An example of using these channels follows:

```ocaml
let n = 9 in
  def print_tick(i) = print_int i; tick()
  in
  print_string "(");
  spawn begin
    count(n) &
    for int i = 1 to n do print_tick(i) done
  end;
  wait();
  print_string ")" ;;
```

This example will print numbers 1 . . . 9 in arbitrary order, but properly enclosed by parentheses.
Join Continuations and Concurrent Iterators

A generalization of waiting for \( n \) events is collecting the results from \( n \) concurrent events. The following higher-order code takes \( f \) as a function to combine the \( n \) results and \( y_0 \) as the collection operation identity.

```plaintext
let create_collector f y0 n =
  def count(y,n) & collect(x) = count(f x y,n-1)
  or count(y,0) & wait() = reply y to wait in
  spawn count(y0,n) ;
  collect, wait ;;
```

Join Continuations and Concurrent Iterators

For example, summing the first $n$ numbers can be implemented as follows:

```ocaml
let n = 10
and add, total = create_collector (+) 0 n
in
spawn for i = 1 to n do add(i) done;
print_int (total());
```

The code prints 55.
Join Continuations and Concurrent Iterators

Producing a list of squares in arbitrary order can be done as follows:

```ocaml
let n = 10
and square x = x*x
and digits = [1;2;3;4;5;6;7;8;9;10]
and iter f xs = List.iter (fun x -> spawn f(x)) xs
and squares xs =
  let add, total = create_collector
      (fun x xs -> x::xs) [] (List.length xs)
  def add_square(x) = add(square(x)) in
  iter add_square xs;
  total ()
in
squares digits ;;
```

The code uses the asynchronous list iterator `List.iter` which takes an arbitrary function and a list, and calls the function on all the list elements, in this case, spawning \( n \) `add_square` processes which themselves produce \( n \) `add` processes that get collected by the `total` channel created in the collector.