Implement your own reactive language: the ReactiveML experiment

Guillaume Baudart — IBM Research

Louis Mandel — IBM Research

Cédric Pasteur — Ansys

Marc Pouzet — ENS

ICFP 18 tutorial

ReactiveML

Examples of ReactiveML Programs

- interaction with external environment: `chesseye`, `ReactiveAsco`
- network simulation: `elip`, `glonemo`
- chatbot interfaces: `rulebot`

Characteristics of systems that we want to program:

- interactions with external environment
- no hard real-time constraints
- a lot of communications and synchronizations
- a lot of concurrency

ReactiveML

Based on an original idea of Frédéric Boussinot [ReactiveC 91]

- extension of a general purpose programming language (OCaml*)
  - data structures, control structures, higher order, ...
- based on the synchronous concurrency model
  - global logical time, parallel composition, broadcast communication
  - instants of unbounded but finite execution time

Numerous others implementations and formalizations:

- ReactiveC (1991), SL (1995), Reactive Objects (1995), Reactive Scripts (1996), Icobjs (1996), SugarCubes (1997), Junior (1999), Java Fair Theards (2001), Loft (2003), Scheme Fair Theards (2004), S-π calculus (2006), FunLoft (2010), SugarCubes JS (2017), ...

*without objects, foncters, labels, polymorphic variants, . . .

ReactiveML

The language

How to program in ReactiveML

Example of ReactiveML program:

- reproduction of “Carrés Noir et Blanc” of the artist Roger Vilder
- `http://www.rogervilder.com/projets/carre_16.html`

Pump

```
let process sum state delta =
  loop
    emit state (last ?state +. delta);
    pause
  end
```

val sum: (float, float) event -> float -> unit process

An integrator

- global logical time: `pause`
- communication by valued signals: `emit`, `last ?`, ...

Pump

```
let process incr_decr state delta =
  let rec process incr =
    do
      run sum state delta
    until state(x) when x >= 1. -> run decr done
  and process decr =
    do
      run sum state (-. delta)
    until state(x) when x <= 0. -> run incr done
  in
  run incr
val incr_decr: (float, float) event -> float -> unit process
```

Two state automaton

- preemption: `do/until`
- mutually recursive definitions: `let rec/and`

Pump

```
type dir = Up | Down | Left | Right

let process draw dir x y size state =
  loop
    begin match dir with
    | Up -> Graphics.fill_rect x y size (size *: last ?state)
    ...
    end;
    pause
  end
val draw: dir -> int -> int -> int -> ('a, float) event -> unit process
```

Display of the pump

- definition of types and pattern matching as in OCaml

Pump

```
let process pump dir x y size state delta =
  run incr_decr state delta
  ||
  run draw dir x y size state
val pump:
  dir -> int -> int -> int -> (float, float) event -> float ->
    unit process
```

A pump

- the process `incr_decr` computes the value of the `state`
- the process `draw` displays it
- the parallel composition `||` guaranties that both processes are executed at each instant

```
let rec process splittable split dir x y size init =
  signal state default 0. gather (+.) in
  emit state init;
  do
    run pump dir x y size state (random_speed ())
  until split ->
    run cell split x y size (last ?state)
  done
```

```
let rec process splittable split dir x y size init =
  signal state default 0. gather (+.) in
  emit state init;
  do
    run pump dir x y size state (random_speed ())
  until split ->
    run cell split x y size (last ?state)
  done
and process cell split x y size init =
  let size_2 = size / 2 in
  run splittable split Left x y size_2 init ||
  run splittable split Down (x + size_2) y size_2 init ||
  run splittable split Up x (y + size_2) size_2 init ||
  run splittable split Right (x + size_2) (y + size_2) size_2 init
```

Parallel composition and recursion allows the dynamic creation of processes.

Synchronous/Asynchronous


```
let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha speed;
    draw center radius !alpha;
  done


let main =
  swing c1 r a1 speed
```

```
let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha speed;
    draw center radius !alpha;
  done


let main =
  Thread.create (swing c1 r a1) speed;
  Thread.create (swing c2 r a2) speed
```

```
let process swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha speed;
    draw center radius !alpha;
    pause
  done

let process main =
     run swing c1 r a1 speed
  || run swing c2 r a2 speed
```

ReactiveML: processes

Declaration of processes:

- `let process <id> { <pattern> } = <expr>`

Basic expressions:

- cooperation: `pause`
- execution: `run <expr>`

Composition:

- sequential: `<expr> ; <expr>`
- parallel: `<expr> || <expr>`
- parallel and sequential:

  `let <patt> = <expr> and <patt> = <expr> in <expr>`

ReactiveML: communications

Declaration of a signal:

- `signal` *<id>*

Emission of a signal:

- `emit` *<signal>*

Status of a signal:

- waiting: `await` *[* `immediate` *]* *<signal>*
- presence test: `present` *<signal>* `then` *<expr>* `else` *<expr>*
  - reaction to absence is delayed

Valued signals

Emit values on signals:

- `emit` *<signal>* *<value>*

Declaration signals:

- `signal` *<id>* `default` *<value>* `gather` *<function>*
- type of signals: `('a, 'b) event`
  - type of the emitted values: `'a`
  - type of the received values: `'b`
- type of the default value: `'b`
- type of the combination function: `'a -> 'b -> 'b`

Receiving signals with values:

- `await` *<signal>* (*patt*) `in` *<expr>*

Combination functions

```
signal s1 default [] gather (fun x y -> x :: y);;
val s1 : ('_a, '_a list) event

signal s2 default 0 gather (+);;
val s2 : (int , int) event

signal s3 default 0 gather (fun x y -> x);;
val s3 : (int , int) event
```

Remark:

- determinism if the combination function is associative and commutative
- linear use [Dogguy 08]

Preemption

- `do` $<expr>$ `until` $<signal>$ `done`
- `do` $<expr>$ `until` $<signal>$ `->` $<expr>$ `done`
- `do` $<expr>$ `until` $<signal>(<patt>)$ `->` $<expr>$ `done`

Suspension

- activation condition: `do` $<expr>$ `when` $<signal>$ `done`
- suspend/resume switch: `control` $<expr>$ `with` $<signal>$ `done`

Tree traversal


```
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a tree

let rec process iter_breadth f a =
  match a with
  | Empty -> ()
  | Node (x, g, d) ->
      f x;
      pause;
      (run (iter_breadth f g) || run (iter_breadth f d))
val iter_breadth : ('a -> 'b) -> 'a tree -> unit process
```

Reactivity analysis

Motivating Example

```
From: Julien Blond
To: Louis Mandel
Subject: Problem with ReactiveML

Hello,
[...]
I wrote my first ReactiveML program, but when I run it,
nothing happens.
[...]
```

```
let process print_top s =
  loop
    await s;
    print_endline "top"
  end
```

Motivating Example

```
let process clock timer s =
  let time = ref (Unix.gettimeofday ()) in
  loop
    let time' = Unix.gettimeofday () in
    if time' -. !time >= timer
    then (emit s (); time := time')
  end


let process main =
  signal s in
  run (print_top s) || run (clock 1. s)
```

Motivating Example

```
let process clock timer s =
  let time = ref (Unix.gettimeofday ()) in
  loop
    let time' = Unix.gettimeofday () in
    if time' -. !time >= timer
    then (emit s (); time := time');
    pause
  end

let process main =
  signal s in
  run (print_top s) || run (clock 1. s)
```

Goal

Detect at compile time programs that are (potentially) non-reactive

- instantaneous loops

```
let process instantaneous_loop =
  loop () end
```

Warning: This expression may be an instantaneous loop.

- instantaneous recursions

```
let rec process instantaneous_rec =
  run instantaneous_rec
```

Warning: This expression may produce an instantaneous recursion.

- only warnings
  - false positives

Idea of the analysis

Abstract processes as *behaviors* [Amtoft, 99]

- abstract values, signals status, etc.
- keep only the structure of the processes
- check reactivity on behaviors

Limitations

- no value analysis
- no termination proof
- no special case for blocking functions

Behaviors

Atoms

- instantaneous: 0
  - examples: ML functions, `await immediate s`, etc.
- non-instantaneous: $\bullet$
  - examples : `pause`, `await s(x) in e`, etc.
- variable: ϕ
  - process names

Structures

- parallel composition: $||$
- sequential composition: ;
- non-deterministic choice: $+$
- recursion operator: $\mu\phi.$
- process execution: `run`

Check reactivity

non-instantaneous recursion

- recursion variable does not appears in the first instant of the body
- examples

reactive	non-reactive
$\mu\phi.\, \bullet ; \phi$	$\mu\phi.\, \phi$
$\mu\phi.\, (0 + (\bullet ; \phi))$	$\mu\phi.\, ((0 + \bullet) ; \phi)$

Abstract processes

Type system with effects

- add a behaviors to the type of processes

$$\tau\,\texttt{process}[\kappa]$$

- add a behavior to each expression

$$\Gamma \vdash e : \tau \mid \kappa$$

Some typing rules

Pause

$$\Gamma \vdash \texttt{pause} : \texttt{unit} \mid \bullet$$

If/then/else

$$\frac{\Gamma \vdash e : \texttt{bool} \mid 0 \qquad \Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa_1 + \kappa_2}$$

Some typing rules: effects

Process definition

$$\frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\ e : \tau\ \texttt{process}[\kappa] \mid 0}$$

Process execution

$$\frac{\Gamma \vdash e : \tau\ \texttt{process}[\kappa] \mid 0}{\Gamma \vdash \texttt{run}\ e : \tau \mid \texttt{run}\ \kappa}$$

Examples

```
let process clock timer s =
  let time = ref (Unix.gettimeofday ()) in
  loop
    let time' = Unix.gettimeofday () in
    if time' -. !time >= timer
    then (emit s (); time := time')
  end
val clock:
  float -> (unit , 'a) event ->
  unit process[((0; (rec 'r1. ((0; ((0; 0) + 0)); run 'r1))))]
Warning: This expression may be an instantaneous loop.
```

Examples

```
let process par_comb p q =
  loop
    run p || run q
  end
val par_comb: 'a process['r1] -> 'b process['r2] ->
  unit process[rec 'r3. ((run 'r1 || run 'r2); run 'r3)]
```

Examples

```
let process par_comb p q =
  loop
    run p || run q
  end
val par_comb: 'a process['r1] -> 'b process['r2] ->
  unit process[rec 'r3. ((run 'r1 || run 'r2); run 'r3)]

let process good =
  run (par_comb (process ()) (process (pause)))
val good: unit process[run (rec 'r1. ((run 0 || run *); run 'r1))]
```

Examples

```
let process par_comb p q =
  loop
    run p || run q
  end
val par_comb: 'a process['r1] -> 'b process['r2] ->
  unit process[rec 'r3. ((run 'r1 || run 'r2); run 'r3)]

let process good =
  run (par_comb (process ()) (process (pause)))
val good: unit process[run (rec 'r1. ((run 0 || run *); run 'r1))]

let process bad =
  run (par_comb (process ()) (process ()))
val bad: unit process[run (rec 'r1. ((run 0 || run 0); run 'r1))]
Warning: This expression may produce an instantaneous recursion.
```

Examples


```
let rec fix f x = f (fix f) x
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

let process main =
  let process p k v =
    print_int v; print_newline ();
    run (k (v+1))
  in
  run (fix p 0)
val main: 'a process[0; run (rec 'r1. (0; 0; run 'r1))]
Warning: This expression may produce an instantaneous recursion.
```

Limitations: value abstraction


```
let rec process imprecise =
  if true then pause else ();
  run imprecise
```

val imprecise: 'a process[rec 'r1. ((+ 0); run 'r1)]*

Warning: This expression may produce an instantaneous recursion.

Limitations: blocking IOs


```
let process io =
  (let s = read_line () in print_endline s)
  ||
  pause; print_endline "bye"
val io : unit process[(0; 0) || (*; 0)]
```

Limitations: blocking IOs


```
let process io =
  (let s = read_line () in print_endline s)
  ||
  pause; print_endline "bye"
val io : unit process[(0; 0) || (*; 0)]


let process io_async =
  (let s = run (Async.proc_of_fun read_line) () in print_endline s)
  ||
  pause; print_endline "bye"
```

Limitations: no termination proofs

```
let rec process par_iter p l =
  match l with
  | [] -> ()
  | x :: l' ->
      run (p x) || run (par_iter p l')
```

```
val par_iter: ('a -> 'b process['r1]) -> 'a list ->
  unit process[rec 'r2. (0 + (run 'r1 || run 'r2))]
Warning: This expression may produce an instantaneous recursion.
```

Limitations: no termination proofs

```
let rec process par_iter p l =
  match l with
  | [] -> ()
  | x :: l' ->
      run (p x) || run (par_iter p l')
val par_iter: ('a -> 'b process['r1]) -> 'a list ->
  unit process[rec 'r2. (0 + (run 'r1 || run 'r2))]
Warning: This expression may produce an instantaneous recursion.
```

Infinite list:

```
let rec l = 0 :: l
```

Limitations: no bound on resources


```
let rec process server add =
  await add(p, ack) in
  run (server add) || let v = run p in emit ack v
val server:
  ('a, ('b process['r1] * ('b, 'c) event)) event ->
  unit process[rec 'r2. (*; (run 'r2 || (run 'r1; 0)))]
```

Typing issue


```
let process p = pause
val p : unit process[*]

let process q = ()
val q : unit process[0]

let l = [p; q]
val l : unit process[???] list
```

Reactivity analysis with rows

Idea

- subeffecting = sub-typing on effects
- inspired by row types [Remy, 93]
- a process has **at least** the behavior of its body
- new typing rule

$$\frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\, e : \tau\, \texttt{process}[\kappa + \phi] \mid 0}$$

Example

```
let process p = pause
val p : unit process[* + 'r0]

let process q = ()
val q : unit process[0 + 'r1]

let l = [p; q]
val l : unit process[* + 0 + 'r] list
```

Causality

Causality

What is the behavior of the following program?

```
signal s1, s2 in
  present s1 then emit s2 else ()
  ||
  present s2 then () else emit s1
```

Semantics

Behavioral semantics (based on Esterel)

Shape of the reductions

$$N \vdash e \xrightarrow[S]{E, b} e'$$

- N set of signal names n created during the reaction of e
- E signals emitted by the reaction of e e
- S signal environment in which e must react
- b termination status

We have the invariant $E \sqsubseteq S$.

Behavioral semantics

$$\frac{N_1 \vdash e \xrightarrow[S]{E,\,true} n \qquad n \in S \qquad N_2 \vdash e_1 \xrightarrow[S]{E_1,\,b} e_1'}{N_1 \cdot N_2 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_1,\,b} e_1'}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \qquad n \notin S}{N \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E,\,false} e_2}$$

$\Rightarrow$ Delay to react to absence

Property: determinism

In a given signal environment, a program can react in only one way.

- Property (Determinism)

  $\forall e, \forall S, \forall N.$

  if $\forall n \in Dom(S).\ S^g(n) = f$ and $f(x, f(y,z)) = f(y, f(x,z))$

  and $N \vdash e \xrightarrow[S]{E_1\,,\,b_1} e'_1$ and $N \vdash e \xrightarrow[S]{E_2\,,\,b_2} e'_2$

  then $(E_1 = E_2 \wedge b_1 = b_2 \wedge e'_1 = e'_2)$

Property: unicity

If a program is reactive, then there exist a unique smallest signal environment in which it can react.

- Property (Unicity)

  For all expression e, let $\mathcal{S}$ the set of signal environments such that

$$\mathcal{S} = \left\{ S \mid \exists N, E, b.\ N \vdash e \xrightarrow[S]{E, b} e' \right\}$$

  then there exists a unique smallest environment $(\sqcap \mathcal{S})$ such that

$$\exists N, E, b.\ N \vdash e \xrightarrow[\sqcap \mathcal{S}]{E, b} e'$$

Determinism + Unicity $\Rightarrow$ all reactive programs are causal

Principles of the implementation: continuations [LFP 80 by Wand]

Compilation

ReactiveML:

$$
\begin{aligned}
e ::= {} & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause} \\
& \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \mid e; e \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \\
& \mid \texttt{emit}\ e\ e \mid \texttt{await immediate}\ e \mid \texttt{await}\ e(x)\ \texttt{in}\ e \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e
\end{aligned}
$$

Compilation

ReactiveML:

$$e ::= x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause}$$
$$\mid \texttt{let}\ x = e\ \textsf{and}\ x = e\ \texttt{in}\ e \mid e; e \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e$$
$$\mid \texttt{emit}\ e\ e \mid \texttt{await immediate}\ e \mid \texttt{await}\ e(x)\ \texttt{in}\ e \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e$$

$\mathcal{L}_k$: a language with continuations

$$k ::= \textsf{end} \mid \kappa \mid e_i.k \mid \texttt{present}\ e_i\ \texttt{then}\ k\ \texttt{else}\ k \mid \texttt{run}\ e_i.k$$
$$\mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ k$$
$$\mid \texttt{await immediate}\ e_i.k \mid \texttt{await}\ e_i(x)\ \texttt{in}\ k$$
$$\mid \texttt{split}\ (\lambda x.(k, k)) \mid \texttt{join}\ x\ i.k \mid \texttt{def}\ x\ \textsf{and}\ x\ \texttt{in}\ k \mid \texttt{bind}\ \kappa = k\ \texttt{in}\ k$$

$$e_i ::= x \mid c \mid (e_i, e_i) \mid \lambda x.e_i \mid \texttt{rec}\ x = e_i \mid \texttt{process}\ \Lambda\kappa.k$$
$$\mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ e_i \mid \texttt{emit}\ e_i\ e_i$$

Partial CPS translation

Separation between instantaneous and reactive expressions:

$$\frac{}{\gamma \vdash c} \qquad \frac{}{1 \vdash \texttt{pause}} \qquad \frac{0 \vdash e_1}{\gamma \vdash \lambda x.e_1} \qquad \frac{1 \vdash e_1}{\gamma \vdash \texttt{process}\ e_1} \qquad \ldots$$

Translation from ReactiveML to $\mathcal{L}_k$:

$$C[\texttt{process}\ e] = \texttt{process}\ \Lambda\kappa.C_\kappa[e]$$

$$C_k[e_1; e_2] = C_{(C_k[e_2])}[e_1]$$

$$C_k[e] = C[e].k \quad \text{si} \quad 0 \vdash e$$

$$\ldots$$

Runtime

Runtime

Data structures:

- *current*: set of continuations to execute in the current instant
- *next*: set of continuations to execute in the next instant
- *wait*: set of waiting lists associated to signals

Execution:

- execute all the continuations that are in *current*
- prepare the reaction of the next instant (end of instant reaction):
  - react to the absence of a signal, get the value of a signal, . . .
  - transfer *next* to *current*

Compilation to OCaml

Source program:

```
let process sum state delta =
  loop
    emit state (last ?state +. delta);
    pause
  end
```

Generated code (after pretty-printing):

```
let sum state delta k =
  rml_loop
    (fun k ->
      rml_emit_v_e state (fun () -> rml_last state +. delta)
        (rml_pause k))
```

Semantics of $\mathcal{L}_k$

Greedy semantics

- data structures
  - $\mathcal{C}$ set of expressions to execute at the current instant
  - $\mathcal{W}$ set of expressions waiting a signal
  - J set of synchronisation points

Execute one reaction step

$$S,\, J,\, \mathcal{W} \vdash \texttt{<}e\,\texttt{,}\,v\texttt{>} \longrightarrow S',\, J',\, \mathcal{W}' \vdash \mathcal{C}$$

- e expression to execute
- v previous value

Implantation in OCaml

Rules of the $\mathcal{L}_k$ semantics can be translated in transition functions of type:

$$
\begin{array}{rcl}
step & = & env \times value \rightarrow env \\
env & = & signal\_env \times join \times waiting \times current
\end{array}
$$

With implace modification of the state, the transition functions can have the following OCaml type:

```
type 'a step = 'a -> unit
```

Implantation in OCaml: `compute`

$$\frac{e/S \Downarrow v'/S'}{S,\, J,\, \mathcal{W} \vdash \texttt{<}\, e.k\, \texttt{,}\ v\, \texttt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \texttt{<}\, k\, \texttt{,}\ v'\, \texttt{>}}$$

Implantation in OCaml: `compute`

$$\frac{e/S \Downarrow v'/S'}{S,\, J,\, \mathcal{W} \vdash \texttt{<} e.k \,\texttt{,}\, v \texttt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \texttt{<} k \,\texttt{,}\, v' \texttt{>}}$$

The transition function `compute` is defined by:

```
let compute e k =
  fun v ->
    let v' = e() in
    k v'
```

val compute : (unit -> 'a) -> 'a step -> 'b step

Implantation in OCaml: `await/immediate`

$$\frac{e/S \Downarrow n/S' \qquad n \in S'}{S,\, J,\, \mathcal{W} \vdash \texttt{<}\, \texttt{await immediate}\ e.k\, \texttt{,}\ v\, \texttt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \texttt{<}\, k\, \texttt{,}\ \texttt{()}\, \texttt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad \mathit{self} = \texttt{await immediate}\ n.k}{S,\, J,\, \mathcal{W} \vdash \texttt{<}\, \texttt{await immediate}\ e.k\, \texttt{,}\ v\, \texttt{>} \longrightarrow S',\, J,\, \mathcal{W} + [\texttt{<}\, \mathit{self}\, \texttt{,}\ v\, \texttt{>}/n] \vdash \emptyset}$$

Implantation in OCaml: `await/immediate`

$$\frac{e/S \Downarrow n/S' \qquad n \in S'}{S,\, J,\, \mathcal{W} \vdash \texttt{<} \texttt{await immediate } e.k \texttt{,}\, v \texttt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \texttt{<} k \texttt{,}\, () \texttt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad \textit{self} = \texttt{await immediate } n.k}{S,\, J,\, \mathcal{W} \vdash \texttt{<} \texttt{await immediate } e.k \texttt{,}\, v \texttt{>} \longrightarrow S',\, J,\, \mathcal{W} + [\texttt{<} \textit{self} \texttt{,}\, v \texttt{>}/n] \vdash \emptyset}$$

```
let await_immediate e k =
  fun v ->
    let (n, w) = e() in
    let rec self () =
      if Event.status n then k ()
      else w := self :: !w
    in self ()
val await_immediate : (unit -> ('a, 'b) event) -> unit step -> 'c step
```

Implantation in OCaml: `emit`

```
let emit e1 e2 k =
  fun v ->
    let (n, w) = e1() in
    let v' = e2() in
    Event.emit n v';
    current := !w @ !current;
    !w := [];
    k ()
val emit :
  (unit -> ('a, 'b) event) -> (unit -> 'a) -> unit step
    -> 'c step
```

Library for reactive programming

```
val rml_compute: (unit -> 'a) -> 'a expr
val rml_seq: 'a expr -> 'b expr -> 'b expr
val rml_par: 'a expr -> 'b expr -> unit expr
...
```

The ReactiveML expression:

```
(await s1 || await s2); emit s3
```

is translated in OCaml by:

```
rml_seq
  (rml_par
    (rml_await (fun () -> s1))
    (rml_await (fun () -> s2)))
  (rml_emit (fun () -> s3)))
```

Suspension and preemption

Preemption


```
let process generate_new_swing click key new_swing =
  loop
    await click (p1) in
    do
      await click (p2) in
      emit new_swing (p1, p2)
    until key(Key_ESC) done
  end
```

Event drive programming

```
class generate_new_swing = object(self)
  val mutable state = 0
  val mutable last_clock = (0, 0)

  method on_click pos =
    match state with
    | 0 -> last_click <- pos;
           state <- 1
    | 1 -> emit new_swing (last_click, pos);
           state <- 0

  method on_key_down k =
    match k with
    | Key_ESC -> state <- 0
    | _ -> ()
end
```

Difficulty

Keep the program structure

```
signal s, p in
do
  await s; print_endline "do not print"
when p done
||
emit p; pause; emit s
```

The message `"do not print"` must not be printed

Control tree

Data structure that:

- keep the structure of preemptions and suspensions
- associate a set *next* to each node of the tree

```
type control_tree =
   { kind: control_kind;
     mutable cond: (unit -> bool);
     mutable children: control_tree list;
     mutable next: next; ... }

and control_kind =
   Top
 | Kill of unit step
 | Susp
 ...
```

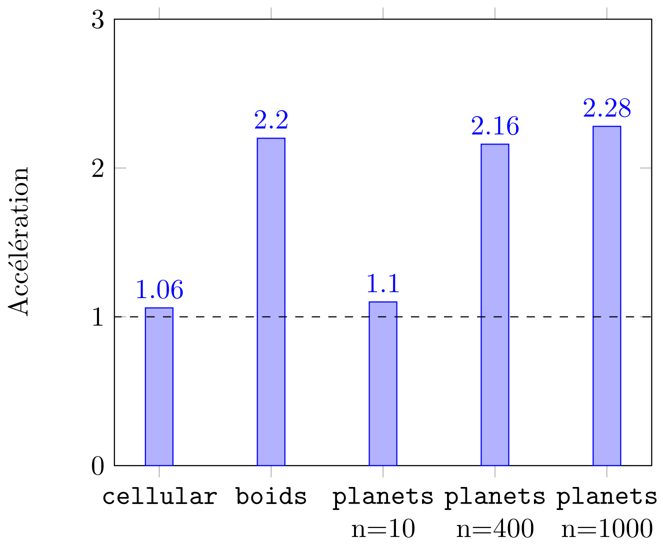
Conclusion on the implementation

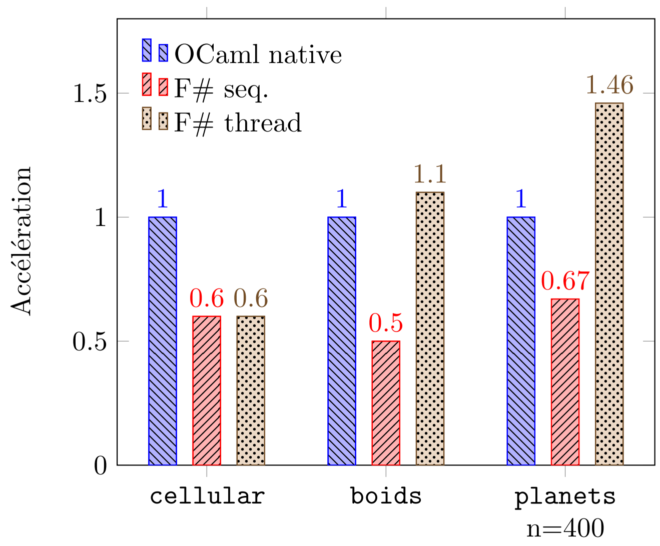
Generation of sequential code

- dynamic scheduling to express concurrency
- take advantage of the OCaml tools: `ocamlopt`, `js_of_ocaml`, ...

Generate parallel code:

- work steeling in the *current* set
- implantation in F#
- with 4 threads, 2 processors with 2 cores each





Reactive Probabilistic Programming

Conclusion

`http://reactiveml.org`

Bibliographie

Selection of articles on ReactiveML

- [PPDP 05] ReactiveML, a Reactive Extension to ML
  - language definition
- [SLAP 08] Interactive Programming of Reactive Systems
  - ReactiveML toplevel
- [FARM 13] Programming Mixed Music in ReactiveML
  - example of application
- [SAS 14] Reactivity of Cooperative Systems
  - reactivity analysis
- [SCP 15] Time refinement in a functional synchronous language
  - reactive domains
- [PPDP 15] ReactiveML, Ten Years Later
  - retrospective and implantation