École Jeunes Chercheurs en Programmation

Cours de ReactiveML

Louis Mandel
LRI, Université Paris-Sud 11
INRIA Paris-Rocquencourt

16 juin 2012

Les systèmes réactifs

Caractéristiques des systèmes que nous voulons programmer :

- pas de contraintes temps réel
- beaucoup de communications et de synchronisations
- beaucoup de concurrence
- création dynamique de processus

ReactiveML

Extension d'un langage généraliste (OCaml*)

- structures de données
- structures de contrôle

Modèle de concurrence simple et déterministe

- composition parallèle
- communications entre processus

Compilé vers du code OCaml

- générateur de bytecode et de code natif
- exécutif efficace, glaneur de cellule (GC)

* sans objets, foncteurs, labels, variants polymorphes, ...

Plan

1. Programmer en ReactiveML
2. Programmer ReactiveML

Synchrone/Asynchrone

```
let plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
  done
```

```
let plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
  done


let main =
  Thread.create (plateforme c1 r a1) vitesse;
  Thread.create (plateforme c2 r a2) vitesse
```

```
let plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    Thread.yield()
  done

let main =
  Thread.create (plateforme c1 r a1) vitesse;
  Thread.create (plateforme c2 r a2) vitesse
```

```
let plateforme centre rayon alpha_init vitesse m1 m2 =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    Mutex.unlock m2; Mutex.lock m1
  done

let main =
  let m1, m2 = Mutex.create (), Mutex.create () in
  Mutex.lock m1; Mutex.lock m2;
  Thread.create (plateforme c1 r a1) vitesse m1 m2;
  Thread.create (plateforme c2 r a2) vitesse m2 m1
```

Synchrone/Asynchrone

```
let barriere n =
  let mutex, attente = Mutex.create (), Mutex.create () in
  Mutex.lock attente;
  let nb_att = ref 0 in
  fun () ->
    Mutex.lock mutex;
    incr nb_att;
    if !nb_att = n then begin
      for i = 1 to n-1 do Mutex.unlock attente done;
      nb_att := 0; Mutex.unlock mutex
    end else begin
      Mutex.unlock mutex; Mutex.lock attente
    end
```

```
let stop = barriere 3

let plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    stop ()
  done

let main =
  Thread.create (plateforme c1 r a1) vitesse;
  Thread.create (plateforme c2 r a2) vitesse;
  Thread.create (plateforme c3 r a3) vitesse
```

```
let process plateforme centre rayon alpha_init vitesse =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw centre rayon !alpha;
    pause
  done

let process main =
     run (plateforme c1 r a1 vitesse)
  || run (plateforme c2 r a2 vitesse)
  || run (plateforme c3 r a3 vitesse)
```

Le modèle réactif synchrone

Caractéristiques

- Instants logiques
- Composition parallèle synchrone
- Diffusion instantanée d'événements
- Création dynamique de processus

Origines

- Esterel [G. Berry & *al.* 1983]
- ReactiveC [F. Boussinot 1991]
- SL [F. Boussinot & R. de Simone 1996]

Autres langages :

- SugarCubes, Simple, Fair Threads, Loft, FunLoft, Lurc, S-pi, ...

ReactiveML

Le langage

Exécuter un programme ReactiveML

Compilation d'un fichier `a.rml` avec le processus principal `p` :

- le compilateur `rmlc` :

```
--> rmlc -s p a.rml
--> ocamlc -I `rmlc -where` unix.cma rmllib.cma a.ml
```

- l'outil de compilation `rmlbuild` :

```
--> cat a.rmlsim
sim: p
--> rmlbuild a.rml.byte
```

Mode interactif :

- dans un terminal :

```
--> rmltop
```

- dans un navigateur web : `http://rml.lri.fr/tryrml`

ReactiveML : les processus

Déclaration de processus :

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

Expressions de base :

- coopération : `pause`
- exécution : `run <expr>`

Composition :

- séquentielle : `<expr> ; <expr>`
- parallèle : `<expr> || <expr>`

ReactiveML : les communications

demo2.rml, demo_present.rml

Déclaration d'un signal :

- `signal` *<id>*

Émission d'un signal :

- `emit` *<signal>*

Statut d'un signal :

- attente : `await` *[* `immediate` *]* *<signal>*
- test de présence : `present` *<signal>* `then` *<expr>* `else` *<expr>*

Causalité à la Boussinot

Problème de causalité :

- incohérence logique sur le statut d'un signal :

  au cours d'un instant, un signal doit être : soit présent, soit absent !

- en Esterel :

```
signal s in
  present s then nothing else emit s end;
end
```

- en ReactiveML :

```
signal s in
present s then () else emit s
```

le retard de la réaction à l'absence supprime les problèmes de causalité

Signaux valués

demo_signal.rml

Émission de valeurs sur les signaux :

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

Déclaration de signaux :

- `signal` *<id>* `default` *<value>* `gather` *<function>*
- type des signaux : `('a, 'b) event`
- type de la valeur par défaut : `'b`
- type de la fonction de combinaison : `'a -> 'b -> 'b`

Réception de valeurs sur les signaux :

- `await` *<signal>* (*patt*) `in` *<expr>*
- utilisation à l'instant suivant : absence de problèmes de causalité

Causalité à la Boussinot

Délai avant la récupération de la valeur d'un signal

- En Esterel :

```
signal s := 0 : combine integer with + in
  emit s(1);
  var x := ?s: integer in
    emit s(x)
  end
end
```

Fonctions de combinaison

```
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
```

Remarque :

- déterminisme si la fonction de combinaison est associative et commutative

Cas particulier

Attendre une seule valeur :

- exemple : `await s (x :: _) in print_int x`
- `await [ immediate ] one <signal> (<patt>) in <expr>`

Garantir l'émission unique :

- dynamiquement :

```
signal s5 default None gather
    (fun x y ->
      match y with
      | None -> Some x
      | Some _ -> assert false);;
val s5 : ('_a, '_a option) event
```

- statiquement : [Amadio et Dogguy 08]

Démo

`planets.rml`

Parcours d'arbres

arbre.rml

```
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre

let rec process iter_largeur f a =
  pause;
  match a with
  | Vide -> ()
  | Noeud (x, g, d) ->
      f x;
      (run (iter_largeur f g) || run (iter_largeur f d))
val iter_largeur : ('a -> 'b) -> 'a arbre -> unit process
```

Parcours d'arbres

```
let rec process mem x a =
  pause;
  match a with
  | Vide -> false
  | Noeud (y, g, d) ->
      if x = y then true
      else
        let b1 = run (mem x g)
        and b2 = run (mem x d) in
        b1 or b2
val mem : 'a -> 'a arbre -> bool process
```

Structures de contrôle

Préemption

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

Causalité à la Boussinot

Uniquement de la préemption faible

- Esterel :

```
signal k, s in
  abort
    pause;
    emit k;
    emit s
  when k end abort;
end
```

Causalité à la Boussinot

Délai avant l'exécution de la continuation d'une préemption faible

- Esterel :

```
signal s1, s2, k in
  weak abort
    pause;
    await immediate s1;
    emit s2
  when k do emit s1; end weak abort;
end
```

Parcours d'arbres

```
let rec process mem x a =
  pause;
  match a with
  | Vide -> false
  | Noeud (y, g, d) ->
      if x = y then true
      else
        let b1 = run (mem x g)
        and b2 = run (mem x d) in
        b1 or b2
val mem : 'a -> 'a arbre -> bool process
```

Parcours d'arbres

```
let rec process mem_aux x a ok =
  pause;
  match a with
  | Vide -> ()
  | Noeud (y, g, d) ->
      if x = y then emit ok
      else
        let b1 = run (mem_aux x g ok)
        and b2 = run (mem_aux x d ok) in
        ()
```

```
val mem_aux : 'a -> 'a arbre -> (unit, 'b) event -> unit process
```

Parcours d'arbres

```
let process mem x a =
  signal ok in
  do
    run (mem_aux x a ok);
    pause; false
  until ok -> true done
val mem : 'a -> 'a arbre -> bool process
```

Remarque :

```
let mem_aux x a ok =
  iter_largeur (fun y -> if x = y then emit ok) a
val mem_aux : 'a -> 'a arbre -> (unit , 'b) event -> unit process
```

Parcours d'arbres

```
let assoc_aux x a ok =
  iter_largeur (fun (y,v) -> if x = y then emit ok v) a
val assoc_aux :
   'a -> ('a * 'b) arbre -> ('b, 'c) event -> unit process

let process assoc x a =
  signal ok in
  do
    run (assoc_aux x a ok);
    pause; []
  until ok (x) -> x done
val assoc : 'a -> ('a * 'b) arbre -> 'b list process
```

Structures de contrôle

Suspension

- condition d'activation : `do` *<expr>* `when` *<signal>* `done`
- interrupteur : `control` *<expr>* `with` *<signal>* `done`

Création dynamique de plates-formes

```
let process read_click click =
  loop
    if Graphics.button_down() then emit click (Graphics.mouse_pos());
    pause
  end
val read_click : ((int * int) , 'a) event -> unit process
```

Création dynamique de plates-formes

```
let rec process add click =
  await click (x,y) in
  run (plateforme (float x, float y) 150. 0. vitesse)
  ||
  run (add click)
val add : ('a, (int * int)) event -> unit process
```

Placement de plates-formes


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

Programmation événementielle

```
class generate_new_plateforme = 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_plateforme (last_click, pos);
           state <- 0

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

ReactiveML

Ordre supérieur et polymorphisme

```
signal kill
```
val kill : (int, int list) event

```
let process killable p =
  let id = gen_id () in print_endline ("["^(string_of_int id)^"]");
  do run p
  until kill(ids) when List.mem id ids done
```
val killable : unit process -> unit process

Création dynamique : rappel

```
let rec process extend to_add =
  await to_add(p) in
  run p || run (extend to_add)
val extend : ('a, 'b process) event -> unit process

signal to_add
  default process ()
  gather (fun p q -> process (run p || run q))
val add_to_me : (unit process, unit process) event
```

Création dynamique avec état

```
let rec process extend to_add state =
  await to_add(p) in
  run (p state) || run (extend to_add state)
val extend : ('a , ('b -> 'c process)) event -> 'b -> unit process

signal to_add
  default (fun s -> process ())
  gather (fun p q s -> process (run (p s) || run (q s)))
val to_add : (('_state -> unit process) , ('_state -> unit process)) event
```

extensible

```
signal add
val add : ((int * (state -> unit process)),
           (int * (state -> unit process)) list) event

let process extensible p_init state =
  let id = gen_id () in print_endline ("{"^(string_of_int id)^"}");
  signal add_to_me
    default (fun s -> process ())
    gather (fun p q s -> process (run (p s) || run (q s))) in
  run (p_init state) || run (extend add_to_me state)
  || loop
       await add(ids) in
       List.iter (fun (x,p) -> if x = id then emit add_to_me p) ids
     end
val extensible : (state -> 'a process) -> state -> unit process
```

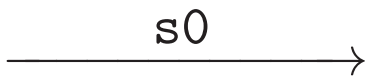
Implantation de ReactiveML

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```




```
await immediate s1 || await immediate s0; emit s1
```

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

$$\xrightarrow{\texttt{s0}}$$

```
await immediate s1 || await immediate s0; emit s1
```

$$\xrightarrow{\texttt{s0, s1}}$$

```
await immediate s1
```

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

$$\xrightarrow{\texttt{s0}}$$

```
await immediate s1 || await immediate s0; emit s1
```

$$\xrightarrow{\texttt{s0, s1}}$$

```
await immediate s1
```

$$\xrightarrow{\texttt{s0, s1}} \quad ()$$

$\Rightarrow$ Il faut réactiver une instruction uniquement lorsque le signal dont elle dépend est émis : utilisation de files d'attente

Les clés d'un interprète efficace

D'autres points clés :

- Exécution du code OCaml sans surcoût
- Gestion efficace des signaux
  - accès en temps constant
  - allocation/désallocation automatique
- ...

Sémantique et implantation sans suspension ni préemption

L_k : un langage à base de continuations

- traduction de ReactiveML vers L_k : $C_k[e_1\,;e_2] = C_{(C_k[e_2])}[e_1]$ ...
- exemple :

```
let nat k =
  fun _ ->
    (let cpt = ref 0 in
     Lk_record.rml_loop
      (fun k' ->
        Lk_record.rml_compute (fun () -> print_int !cpt; ...)
          (Lk_record.rml_pause k'))
      ())
```

Sémantique de $\mathbf{L}_k$

Sémantique gloutonne

- toujours aller de l'avant
- représentation du programme
  - ▷ $\mathcal{C}$ ensemble des expressions à exécuter instantanément
  - ▷ $\mathcal{W}$ ensemble des expressions en attente d'un signal
  - ▷ J ensemble des points de synchronisation

Exécution d'une étape de réaction

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

- e expression à exécuter
- v valeur précédente

Implantation en OCaml

Les règles de la sémantique L_k peuvent se traduire quasiment directement en des fonctions de transition de type :

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

En implantant l'environnement directement dans le tas, les fonctions de transitions ont le type OCaml suivant :

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

Implantation en OCaml : `compute`

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

Implantation en OCaml : `compute`

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

La fonction de transition `compute` est définie par :

```
let compute e k =
  fun v ->
    let v' = e() in
    k v'
val compute : (unit -> 'a) -> 'a step -> 'b step
```

Implantation en 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 en OCaml : await/immediate

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

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

```
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 en 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
```

Sémantique de $\mathbf{L}_k$

Les suspensions et préemptions ?

- on a perdu la structure du programme !
- utilisation d'un arbre de contrôle

Bibliothèque pour la programmation réactive

```
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
...
```

L'expression ReactiveML :

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

se traduit en OCaml par :

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

Toplevel

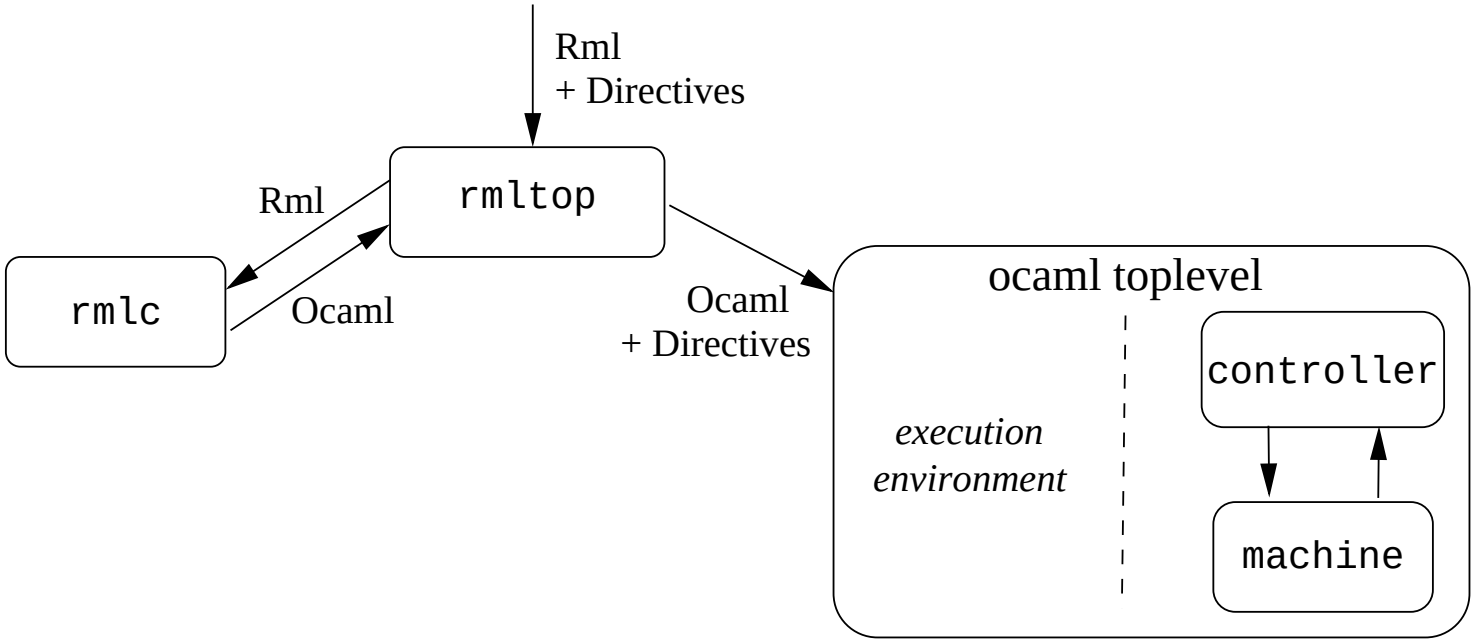
`rmltop` : le toplevel ReactiveML

Basé sur l'idée des Reactive Scripts [Boussinot & Hazard 96]

Utile pour :

- comprendre le modèle réactif
- faire des expériences de reconfiguration dynamique
- concevoir des systèmes réactifs

Implantation



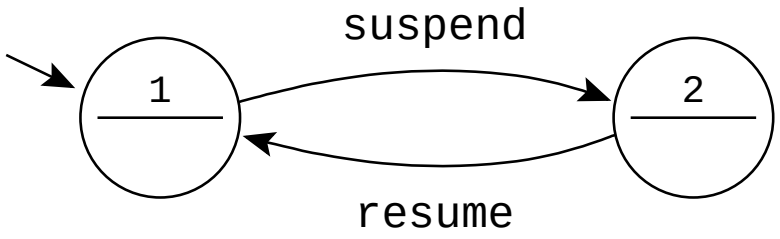
contrôleur implémenté en ReactiveML

Contrôleur

```
let process sampled =
  loop Rmltop_reactive_machine.rml_react(get_to_run()); pause end

let process step_by_step =
  loop
    await step(n) in
    do
      for i = 1 to n do
        Rmltop_reactive_machine.rml_react(get_to_run()); pause
      done
    until suspend done
  end
```

Contrôleur




```
let process machine_controller =
  loop
    do run sampled until suspend done;
    do run step_by_step until resume done
  end
```

Exemple d'application

Suivi de partition

Antescofo~ (`http://repmus.ircam.fr/antescofo`)

- détection d'événements musicaux
- synchronisation d'accompagnement

Ré-implémentation de la partie accompagnement en ReactiveML
(Guillaume Baudart)

Conclusion

http://rml.lri.fr