Modèle réactif : ReactiveML

Louis Mandel
IBM Research

11 décembre 2023

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, ...

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
```

Synchrone/Asynchrone

```
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` :

- le compilateur `rmlc` :

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

Mode interactif :

- dans un terminal :

```
--> rmltop
```

- dans un navigateur web : `http://reactiveml.org/tryrml`

Synchrone

```
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)
```

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

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 =
  match a with
  | Vide -> ()
  | Noeud (x, g, d) ->
      f x;
      pause;
      (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 =
  match a with
  | Vide -> false
  | Noeud (y, g, d) ->
      if x = y then true
      else
        (pause;
         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 =
  match a with
  | Vide -> false
  | Noeud (y, g, d) ->
      if x = y then true
      else
        (pause;
         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 =
  match a with
  | Vide -> ()
  | Noeud (y, g, d) ->
      if x = y then emit ok
      else
        (pause;
         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
```

Suspension

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

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_click = (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

Causalité

La causalité

En Esterel, l'analyse de causalité permet de détecter les programmes :

- incohérents,
- non déterministes,
- ou avec des dépendances non causales.

Causalité dans le modèle réactif

- dans le cadre d'un langage généraliste (avec références, ordre supérieur, ...), il est difficile de réaliser une analyse de causalité
- $\Rightarrow$ nous voulons des programmes causaux par construction.

Causalité en Esterel : incoherence

Le programme Esterel suivant est incohérent :

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

En Esterel, le signal s est simultanément absent et présent !

Causalité en Esterel : incoherence

Le programme Esterel suivant est incohérent :

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

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

En Esterel, le signal s est simultanément absent et présent !

En ReactiveML, la réaction à l'absence est retardée.

- la symétrie entre présence et absence est perdue !

Causalité en Esterel : non-déterminisme

Le programme Esterel suivant est non-déterministe :

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

En Esterel, s peut être présent ou absent !

Causalité en Esterel : non-déterminisme

Le programme Esterel suivant est non-déterministe :

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

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

En Esterel, s peut être présent ou absent !

En ReactiveML, l'hypothèse d'absence est privilégiée.

Remarque : ReactiveML n'est pas un sous-ensemble strict d'Esterel. Le programme Esterel suivant n'est pas valide non plus.

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

Causalité en Esterel : dépendance non causale

Le programme Esterel suivant a une dépendance non causale :

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

En Esterel, l'émission de `t` dépend de `s` qui est émis après son test de présence durant la même réaction !

Causalité en Esterel : dépendance non causale

Le programme Esterel suivant a une dépendance non causale :

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

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

En Esterel, l'émission de `t` dépend de `s` qui est émis après son test de présence durant la même réaction !

En ReactiveML, ce programme est causal grâce à la combinaison de l'hypothèse d'absence privilégiée et du délai de la réaction à l'absence.

Causalité en Esterel : préemption forte

La préemption forte peut introduire des incohérences :

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

En Esterel, au deuxième instant, l'émission de s doit préempter son exécution !

En ReactiveML, il n'y pas de préemption forte.

Causalité en Esterel : préemption faible

La préemption faible peut également introduire des incohérences :

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

En Esterel, l'émission de s2 dépend de s1 qui est émis après la préemption !

Causalité en Esterel : préemption faible

La préemption faible peut également introduire des incohérences :

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

```
signal s1, s2, k in
do
  await s1;
  emit s2
until k ->
  emit s1
done
```

En Esterel, l'émission de s2 dépend de s1 qui est émis après la préemption !

En ReactiveML, l'exécution du handler est retardée.

Causalité en Esterel : signaux valués

Les signaux valués peuvent introduire des problèmes de causalité :

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

En Esterel, la valeur associée à s dépend instantanément d'elle même !

En ReactiveML, la lecture d'un signal prend un instant.

Causalité en Esterel : composition

Enfin, en Esterel, la composition parallèle de programmes causaux n'est pas nécessairement causale :

```
signal s1, s2 in
  present s1 then emit s2 else nothing end
  ||
  present s2 then nothing else emit s1 end
end
```

Suppression des problèmes de causalité

Les 5 caractéristiques du modèle réactif de Boussinot :

1. ajouter un retard pour la réaction à l'absence d'un signal
2. toujours privilégier l'hypothèse d'absence d'un signal
3. la préemption forte est interdite
4. le traitement d'échappement d'une préemption faible est reporté à l'instant suivant la préemption
5. ajouter un retard pour la récupération de la valeur d'un signal

Conséquences : perte de compositionnalité

L'ajout de retard dans la sémantique introduit des difficultés

- L'expression

```
await s_in(x) in emit s_out (x + 2)
```

  n'est pas équivalente à

```
signal tmp default 0 gather (+) in
await s_in(x) in emit tmp (x + 1)
||
await tmp(y) in emit s_out (y + 1)
```

- l'ajout d'une communication interne a ajouté un retard

Conséquences : pas besoin d'analyse de causalité

Tous les programmes sont causaux par construction.

```
let process causalite =
  signal s1, s2 in
  present s1 then emit s2 else nothing
  ||
  present s2 then nothing else emit s1
val causalite: unit process
```

Conséquences : pas besoin d'analyse de causalité

Tous les programmes sont causaux par construction.

```
let process causalite p =
  signal s1, s2 in
  present s1 then emit s2 else nothing
  ||
  run p s1 s2
val causalite:
  (('a, 'a list) event -> (unit , unit list) event -> 'b process) ->
  unit process
```

Ordre supérieur

- les processus sont des valeurs de première classe

Conséquences : pas besoin d'analyse de causalité

Tous les programmes sont causaux par construction.

```
let process causalite s =
  signal s1, s2 in
  present s1 then emit s2 else nothing
  ||
  await s (p) in
  run p s1 s2
val causalite:
  ('a, (('b, 'b list) event -> (unit, unit list) event -> 'c process)) eve
  unit process
```

Ordre supérieur

- les processus sont des valeurs de première classe
- les processus peuvent être émis sur des signaux

Conséquences : signaux valeurs de première classe

Comme dans le π-calcul, les signaux peuvent échapper de leur portée.

```
let process client new_proc p =
  signal ack in
  emit new_proc (process (run p; emit ack));
  await immediate ack
val client:
  (unit process, 'a) event -> 'b process -> unit process
```

Conséquences : signaux sur l'horloge de base

L'horloge d'un signal n'est pas l'horloge de son point de déclaration.

```
let process escape global =
  signal s in
  do
    signal local in
    emit global local;
    emit local 1;
    pause;
    if pre local then print_endline "PRES" else print_endline "ABS"
  when s done
  ||
  emit s; pause; pause; emit s
val escape: ((int, int list) event, 'a) event -> unit process
```

Ici, l'exécution du processus peut afficher ABS !

ReactiveML

Toplevel

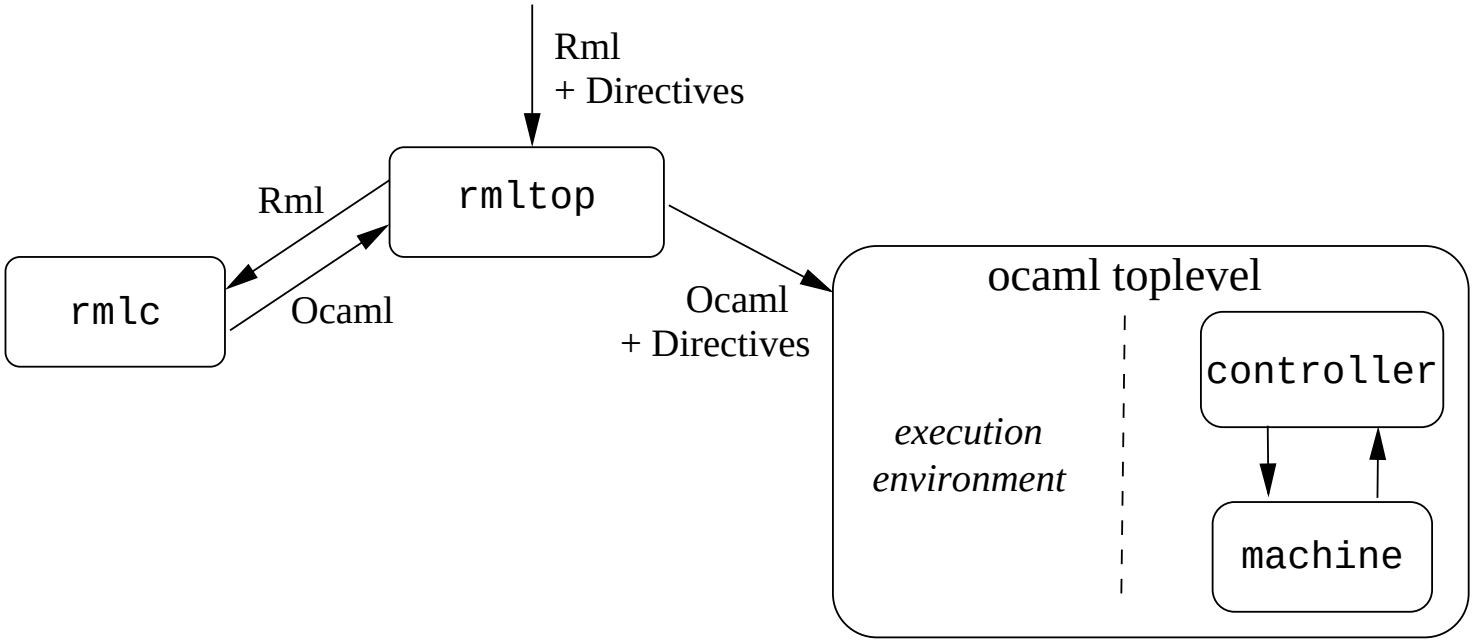
`rmltop` : le toplevel ReactiveML

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



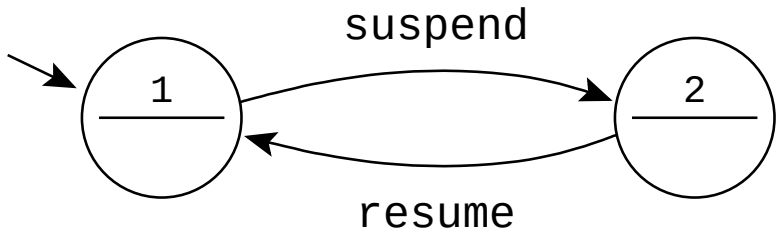
contrôleur implémenté en ReactiveML

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
```

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
```

ReactiveML

Sémantique

Le noyau ReactiveML

$$
\begin{array}{rcl}
e & ::= & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x\ \texttt{=}\ e \mid \texttt{process}\ e \\
& & \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{and}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{pause} \mid \texttt{run}\ e \\
& & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \\
& & \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{emit}\ e\ e \mid \texttt{pre}\ e \mid \texttt{pre}\ \texttt{?}e \\
& & \mid \texttt{do}\ e\ \texttt{until}\ e(x)\ \texttt{->}\ e\ \texttt{done} \mid \texttt{do}\ e\ \texttt{when}\ e \\
c & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{0} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots
\end{array}
$$

Opérateurs dérivés

$$
\begin{array}{rcl}
e_1 \texttt{||} e_2 & \stackrel{def}{=} & \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ \texttt{()} \\
& & \ldots
\end{array}
$$

Sémantiques statiques

Analyse d'instantanéité

- exemple :

```
let f x =
  let y = x + 1 in
  pause;
  print_int y
```

incorrect

```
let process f x =
  let y = x + 1 in
  pause;
  print_int y
```

correct

Typage

- Extension conservative du typage de ML

$$\frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_2 : \tau_1}{H \vdash \texttt{emit}\ e_1\ e_2 : \texttt{unit}} \quad \ldots$$

Sémantiques dynamiques

Sémantique comportementale ("grands pas")

- qu'est ce qu'une réaction valide ?
- abstraction de l'ordonnancement à l'intérieur d'un instant

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

Sémantique opérationnelle ("petits pas")

- comment obtenir une réaction valide ?
- description de tous les ordonnancements possibles

$$e/S_0 \to e_1/S_1 \to \cdots \to e_n/S \to_{eoi} e'$$

La sémantique comportementale

La sémantique comportementale s'inspire de celle d'Esterel :

- une réduction = un instant
- l'environnement des signaux est connu au début de la réaction

Différences par rapport à la sémantique comportementale d'Esterel

- ajout des valeurs
- adaptation au modèle réactif

La sémantique comportementale

Forme des réductions

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

- N ensemble des noms de signaux n créés par la réaction de e
- E signaux émit par la réaction de e
- S environnement de signaux dans lequel e doit réagir
- b statut de terminaison

Comme pour Esterel, nous avons l'invariant $E \sqsubseteq S$.

Définitions

$$
\begin{array}{lcl}
E & ::= & [m_1/n_1, ..., m_k/n_k] \\
S & ::= & [(d_1, g_1, p_1, m_1)/n_1, ..., (d_k, g_k, p_k, m_k)/n_k]
\end{array}
$$

Si le signal n_i est de type $(\tau_1, \tau_2)\,\texttt{event}$ alors :

$$
\begin{array}{ll}
d_i : \tau_2 & p_i : \texttt{bool} \times \tau_2 \\
g_i : \tau_1 \to \tau_2 \to \tau_2 & m_i : \tau_1\,\texttt{multiset}
\end{array}
$$

Notations :

Si $S(n_i) = (d_i, g_i, p_i, m_i)$ alors
$S^d(n_i) = d_i,\ S^g(n_i) = g_i,\ S^p(n_i) = p_i,\ S^v(n_i) = m_i$

n est présent $n \in S$ $(S^v(n) \neq \varnothing)$

n est absent $n \notin S$ $(S^v(n) = \varnothing)$

Sémantique Comportementale : expressions instantanées

$$\varnothing \vdash v \xrightarrow[S]{\varnothing, true} v$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, true} v_1 \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2, true} v_2}{N_1 \cdot N_2 \vdash (e_1, e_2) \xrightarrow[S]{E_1 \sqcup E_2, true} (v_1, v_2)}$$

$$\frac{N \vdash e \xrightarrow[S]{E, true} n \qquad (b, v) = S^p(n)}{N \vdash \texttt{pre}\ e \xrightarrow[S]{E, true} b}$$

Sémantique Comportementale : pause et mise en séquence

$$\varnothing \vdash \texttt{pause} \xrightarrow[S]{\varnothing, \mathit{false}} ()$$

$$\frac{N \vdash e_1 \xrightarrow[S]{E_1, \mathit{false}} e_1'}{N \vdash e_1 \,;e_2 \xrightarrow[S]{E_1, \mathit{false}} e_1' \,;e_2}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, \mathit{true}} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, b} e_2'}{N_1 \cdot N_2 \vdash e_1 \,;e_2 \xrightarrow[S]{E_1 \sqcup E_2, b} e_2'}$$

Sémantique Comportementale : émission et test de présence

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} n \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, true} v}{N_1 \cdot N_2 \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S]{E_1 \sqcup E_2 \sqcup [\{v\}/n],\, true} ()}$$

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

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

$\Rightarrow$ Délai pour la réaction à l'absence.

Sémantique Comportementale : déclaration de signal

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} v_1 \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, true} v_2 \qquad S(n) = (v_1, v_2, (false, v_1), m) \qquad N_3 \vdash e[x \leftarrow n] \xrightarrow[S]{E,\, b} e'}{N \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E,\, b} e'}$$

avec $N = N_1 \cdot N_2 \cdot N_3 \cdot \{n\}$

Sémantique Comportementale : composition parallèle

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, b_1} e_1' \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2, b_2} e_2' \qquad b_1 \wedge b_2 = false}{N_1 \cdot N_2 \vdash e_1 \,||\, e_2 \xrightarrow[S]{E_1 \sqcup E_2, \, false} e_1' \,||\, e_2'}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, \, true} v_1 \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2, \, true} v_2}{N_1 \cdot N_2 \vdash e_1 \,||\, e_2 \xrightarrow[S]{E_1 \sqcup E_2, \, true} ()}$$

$\Rightarrow$ L'environnement S est global.

Propriétés : déterminisme

Dans un environnement de signaux donné, un programme ne peut réagir que d'une seule façon.

- Propriété (Déterminisme)

  $\forall e, \forall S, \forall N.$
  si $\forall n \in Dom(S).\ S^g(n) = f$ et $f(x, f(y, z)) = f(y, f(x, z))$
  et $N \vdash e \xrightarrow[S]{E_1, b_1} e'_1$ et $N \vdash e \xrightarrow[S]{E_2, b_2} e'_2$
  alors $(E_1 = E_2 \wedge b_1 = b_2 \wedge e'_1 = e'_2)$

Propriétés : unicité

si un programme est réactif, alors il existe un unique plus petit environnement de signaux dans lequel il peut réagir.

- Propriété (Unicité)

  Pour toute expression e, soit $\mathcal{S}$ l'ensemble des environnements de signaux tel que

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

  alors il existe un unique plus petit environnement $(\sqcap \mathcal{S})$ tel que

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

Déterminisme + Unicité $\Rightarrow$ tous les programmes réactifs sont causaux.

Sémantique opérationnelle

La sémantique s'inspire de la sémantique à petits pas de ML.

- Il n'y a pas d'hypothèses sur l'environnement des signaux
- La réaction d'un instant est construite comme une succession de réductions suivie d'une réduction de fin d'instant.

La sémantique opérationnelle se décompose en 3 étapes :

- réaction pendant l'instant

$$e/S \rightarrow^* e'/S'$$

- calcul des sorties

$$O = next(S)$$

- réaction de fin d'instant

$$O \vdash e' \rightarrow_{eoi} e''$$

Sémantique opérationnelle

Réduction en tête de terme

$$(\lambda x.e)\, v \,/\, S \;\to_\varepsilon\; e[x \leftarrow v] \,/\, S \qquad\qquad \texttt{emit}\ n\ v \,/\, S \;\to_\varepsilon\; () \,/\, S + [v/n]$$

$$\texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \,/\, S \;\to_\varepsilon\; e_1 \,/\, S \;\text{ si }\; n \in S \qquad \ldots$$

Contextes

$$\begin{aligned}\Gamma \;::=\;& [\,] \mid \Gamma\,; e \mid \texttt{present}\ \Gamma\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid \texttt{let}\ x = \Gamma\ \texttt{and}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ x = e\ \texttt{and}\ x = \Gamma\ \texttt{in}\ e \mid \ldots\end{aligned}$$

$$\frac{e \,/\, S \;\to_\varepsilon\; e' \,/\, S'}{\Gamma(e)/S \to \Gamma(e')/S'} \qquad \frac{n \in S \qquad e/S \to e'/S'}{\Gamma(\texttt{do}\ e\ \texttt{when}\ n)/S \to \Gamma(\texttt{do}\ e'\ \texttt{when}\ n)/S'}$$

Sémantique opérationnelle

Fin d'instant

$$\frac{n \notin O}{O \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2}$$

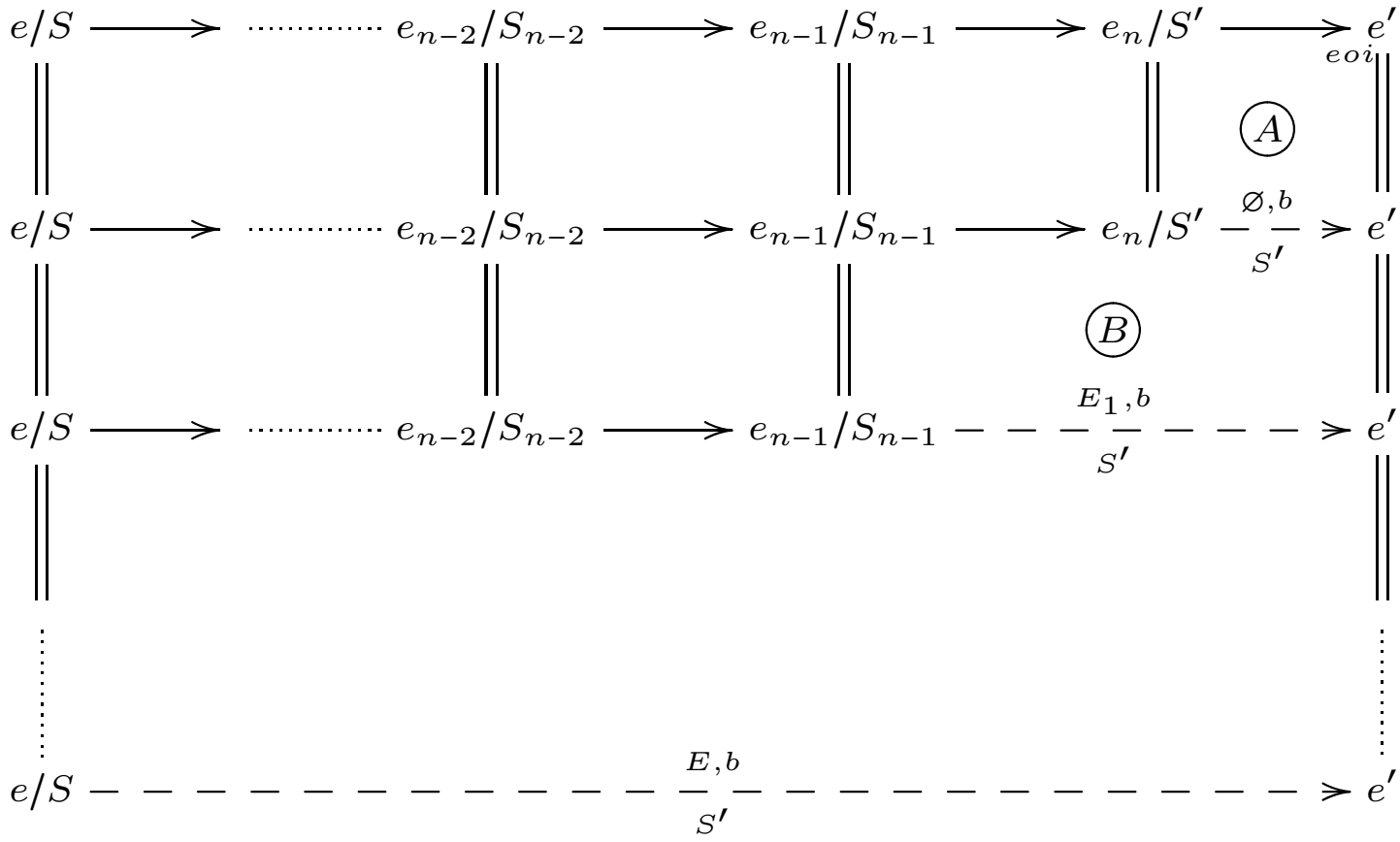
$$O \vdash \texttt{pause} \rightarrow_{eoi} ()$$

$$\ldots$$

avec $O = next(S)$

Propriété : Opérationnelle ⇒ Comportementale

Pour tout S_{init} et e tels que $e/S_{init} \rightarrow e_1/S_1 \rightarrow \cdots \rightarrow e_n/S \rightarrow_{eoi} e'$ alors il existe N, E et b tels que $N \vdash e \xrightarrow[S]{E,b} e'$ avec $E = S^v \backslash S^v_{init}$



Propriétés

Sémantique comportementale

- déterministe

Sémantique opérationnelle

- preuve de sûreté du typage

Sémantiques comportementale et opérationnelle

- équivalence entre les deux sémantiques
- absence de problème de causalité

Analyse de réactivité

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

Statically detect (potentially) non-reactive programs

- instantaneous loops

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

- instantaneous recursions

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

Only warnings

- false positives

The idea

Abstract processes into "behaviors" [Amtoft, 99]

- abstract away values, signal presence, etc.
- only keep the some structure of the process
- check reactivity on the behaviors

Limitations

- no value analysis
- we don't prove that functions terminate
- so special case for blocking functions (IOs)

Behaviors: atoms

- Surely non-instantaneous action: $\bullet$
  - examples: `pause`, `await s(x) in e`, etc.
- Maybe instantaneous action: 0
  - examples: ML functions, `await immediate s`, etc.
- Variables for process arguments: ϕ

Behaviors: structure

- Parallel composition: $\kappa \,||\, \kappa$

```
let process par_comb p q =
  loop
    run p || run q
  end
```

p **or** q must be non-instantaneous

- Non-deterministic choice : $\kappa + \kappa$

```
let process if_comb c p q =
  loop
    if c then run p else run q
  end
```

p **and** q must be non-instantaneous

Behaviors: structure

- Sequence: $\kappa; \kappa$
  - not reactive

```
let rec process bad_rec =
  run bad_rec; pause
```

  - reactive

```
let rec process good_rec =
  pause; run good_rec
```

Behaviors: structure

- Recursive processes : $\mu\phi.\,\kappa$

```
let rec process good_rec =
  pause; run good_rec
```

  - behavior of `good_rec`: κ
  - behavior of the body: $\bullet;\kappa$
  - solve the equation: $\kappa = \bullet;\kappa$

    $\Rightarrow \quad \kappa \leftarrow \mu\phi.\,\bullet;\phi$

- Loop: $\kappa^{\infty} = \mu\phi.\,\kappa;\phi$

$$\texttt{loop}\ e \triangleq \texttt{run}\ ((\texttt{rec}\ loop = \lambda x.\ \texttt{process}\ (\texttt{run}\ x; \texttt{run}\ (loop\ x)))\ (\texttt{process}\ e))$$

Behaviors: structure

- Another example

```
let rec process p = run p
```

  - behavior of p: κ
  - behavior of the body: κ
  - solve the equation: $\kappa = \kappa$!

Behaviors: structure

- Another example

```
let rec process p = run p
```

  - behavior of p: κ
  - behavior of the body: κ
  - solve the equation: $\kappa = \kappa$!

- Running a process: $\mathtt{run}\ \kappa$

```
let rec process p = run p
```

  - behavior of p: κ
  - behavior of the body: $\mathtt{run}\ \kappa$
  - solve the equation: $\kappa = \mathtt{run}\ \kappa$

    $\Rightarrow \quad \kappa \leftarrow \mu\phi.\, \mathtt{run}\ \phi$

Behavior: summary

Atoms

- instantaneous: 0
- non-instantaneous: $\bullet$
- variable: ϕ

Structure

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

Checking reactivity

Non-instantaneous recursion

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

OK	Not OK
$\mu\phi.\, \bullet; \phi$	$\mu\phi.\, \phi$
$\mu\phi.\, (0 + (\bullet; \phi))$	$\mu\phi.\, ((0 + \bullet); \phi)$

Abstracting processes

Type-and-effect system

- add a behavior to the type of a processes

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

- associate to each expression a type and a behavior

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

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: type-and-effect system

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

par_comb.rml

```
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

par_comb.rml

```
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

par_comb.rml

```
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

fix.rml

```
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

imprecise.rml

```
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

io.rml

```
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

io.rml

```
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

iter.rml

```
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

server.rml

```
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)))]
```

A type system problem


```
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

Subeffecting

Idea

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

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

Any correct ReactiveML program has a behavior

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
```

Property: correctness

All well-typed programs are reactive.

Property (correctness)

- if $\Gamma \vdash e : \tau \mid \kappa$ and κ is reactive
- if all the function calls terminate
- then $\exists e'$ such that $N \vdash e \xrightarrow[S]{E, b} e'$ with $\Gamma \vdash e' : \tau \mid \kappa'$ and κ' is reactive

Proof sketch

- Define the first instant of a behavior.
- The first instant of a behavior is finite.
- Here reduction step of the semantics reduces the size of the behavior.

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
```

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

```
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
```

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

⇒ 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 :

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

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`

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

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

Implantation en OCaml : `await/immediate`

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

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

```
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 :

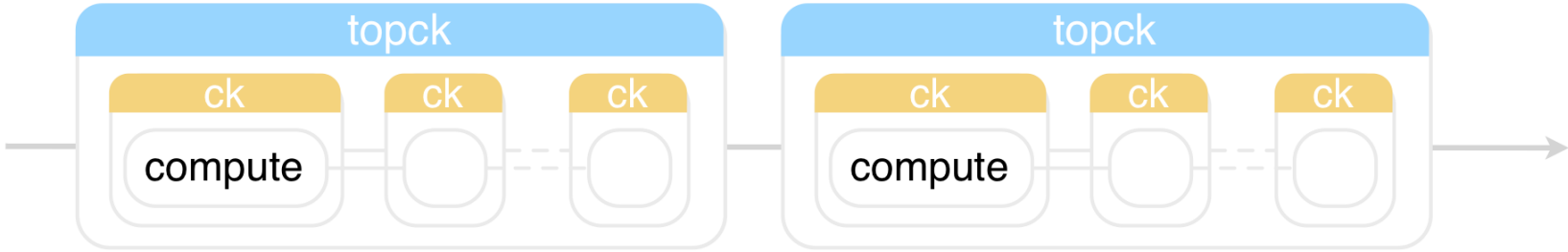
```
rml_seq
  (rml_par
    (rml_await (fun () -> s1))
    (rml_await (fun () -> s2)))
  (rml_emit (fun () -> s3)))
```

ReactiveML

Domaines d'horloges

Domaines d'horloges (Cédric Pasteur)

- Notion d'instant local
- Masque les instants locaux



Exemple: simulation des n-corps

Méthodes d'intégration numériques à pas multiples

- Instants de calcul masqués par un domaine d'horloge
- Changement de méthode à la volée

```
let process f compute s_in s_out =
  domain (ck) do
    loop
      await s_in(v) in
      let res = run compute ck v in
      emit s_out res
    end
  done
```

Une nouvelle analyse statique

Limitations sur l'utilisation des signaux

- Un signal est attaché à un domaine d'horloge
- On ne peut pas utiliser un signal en dehors de son domaine d'horloge
- Pas de dépendence immédiate sur des signaux lents

Un système de types contre l'échappement de portée

- Système de types et effets
- Extension du typage de ML

$$\frac{\Gamma; x : \{\gamma\} \vdash_\gamma e : ct \mid cf \quad \gamma \notin ftv(\Gamma, ct)}{\Gamma \vdash_{ce} \texttt{domain}(x)\, e : ct \mid cf \setminus \{\gamma\}}$$

Exemples (I)

```
let process p1 =
  domain(ck) do
    signal s in s
  done
```

```
The clock of this expression
‘(’_a list , ’_a) event{?ck0|} process’
depends on ck which escape its scope.
```

```
let process p2 =
  signal s in
  domain(ck) do
    signal s2 in
    emit s s2
  done
```

```
The emitted value has clock
(’_a list , ’_a) event{?ck0|},
and would thus escape its scope ck
```

Exemples (II)

```
let process p3 =
  signal s in
  domain (ck) do
    signal s2 in
    let f () =
      emit s2 0
    in
    emit s f
  end
```

```
The emitted value has clock
. =>{[?ckc0]} ., and would
thus escape its scope ck.
```

Conclusion

http://reactiveml.org

Bibliographie

Sélection d'article sur ReactiveML

- [PPDP 05] ReactiveML, a Reactive Extension to ML
  - définition du langage
- [SLAP 08] Interactive Programming of Reactive Systems
  - boucle d'interaction de ReactiveML
- [FARM 13] Programming Mixed Music in ReactiveML
  - exemple d'application
- [SAS 14] Reactivity of Cooperative Systems
  - analyse de réactivité
- [SCP 15] Time refinement in a functional synchronous language
  - domaines réactifs
- [PPDP 15] ReactiveML, Ten Years Later
  - rétrospective et implantation