ReactiveML, partie II

Louis Mandel & Guillaume Baudart

Ordre supérieur et polymorphisme

ReactiveML

Killable

oscillo.rml

```
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

oscillo.rml

```
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

oscillo.rml

```
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

oscillo.rml

```
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)
  || while true do
       await add(ids) in
       List.iter (fun (x,p) → if x = id then emit add_to_me p) ids
     done
val extensible : (state → 'a process) → state → unit process
```

Sémantiques

ReactiveML

Le noyau de ReactiveML

e := x | v | (e, e) | $\lambda x.e$ | e e | rec x = e | process e
| let x = e and x = e in e | pause | run e
| signal x default e gather e in e
| present e then e else e | emit e e | pre e | pre ?e
| do e until $e(x)$ → e done | do e when e done

c := true | false | () | 0 | … | + | - | …

Les autres opérateurs peuvent être obtenus par compilation vers ce noyau :

$e_1 \,||\, e_2$ ≜ let x_1 = e_1 and x_2 = e_2 in () …

Sémantiques statiques

Analyse d'instantanéité

KO

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

OK

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

Typage : extension conservative du typage de ML

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

Sémantique dynamique

Sémantique comportementale (“grands pas”)

- qu’est-ce qu’une réaction valide ?
- on abstrait 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

$$e/S_0 \rightarrow e_1/S_1 \rightarrow \ldots \rightarrow e_n/S_n \rightarrow_{eoi} e'$$

Sémantique comportementale

Inspirée de celle d’Esterel :

- 1 réduction = 1 instant
- l’environnement des signaux est connu au début de la réaction

Mais :

- ajout des signaux valués
- adaptation du modèle réactif (pas de réaction à l’absence)

Sémantique comportementale

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

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

Comme pour Esterel, on a l'invariant $E \sqsubseteq S$.

Définitions

$E := [m_1/n_1, \ldots, m_k/n_k]$
$S ::= [(d_1, g_1, p_1, m_1)/n_1, \ldots, (d_k, g_k, p_k, m_k)/n_k]$

Si le signal n_i est de type (τ_1, τ_2) alors :

$d_i : \tau_2$ $\qquad$ $p_i : \texttt{bool} \times \tau_2$
$g_i : \tau_1 \to \tau_2 \to \tau_2$ $\qquad$ $m_i : \tau_2$ `multiset`

Notations: si $S(n_i) = (d_i, g_i, p_i, m_i)$ alors $S^d(n_i) = d_i$, $S^g(n_i) = g_i$, $S^p(n_i) = p_i$, $S^m(n_i) = m_i$

- n est présent $n \in S$ $(S^v(n) \neq \emptyset)$
- n est absent $n \notin S$ $(S^v(n) = \emptyset)$

Expressions instantanées

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

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

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

→ statut de n à l'instant précédent (présent, absent)

Pause et séquence

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

→ termine (“débloque”) à la réaction suivante

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

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

Émission et test de présence

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

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

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

→ délai pour la réaction à l'absence

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

Composition parallèle

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

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

→ l'environnement S est global ($E_1 \sqsubseteq S$, $E_2 \sqsubseteq S$)

Propriétés : déterminisme

Pour 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 $E_1 = E_2$, $b_1 = b_2$, et $e'_1 = e'_2$

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 S l'ensemble des environnement de signaux tels 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 $\exists N, E, b \,.\, N \vdash e \xrightarrow[\sqcap\mathcal{S}]{E,b} e'$

déterministe + unicité $\Longrightarrow$ tous les programmes réactifs sont causaux

Sémantique opérationnelle

Inspirée de la sémantique à petits pas de ML

- pas d'hypothèse sur l'environnement des signaux
- un instant : succession de réactions suivit d'une réaction 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

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

$\texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2/S \to e_1/S$ si $n \in S$ …

Contextes

Γ ::= [] | Γ; e | present Γ then e else e
| let x = Γ and x = e in e | let x = e and x = Γ in e | …

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

Sémantique opérationnelle

Fin d'instant

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

→ délai pour la réaction à l'absence

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

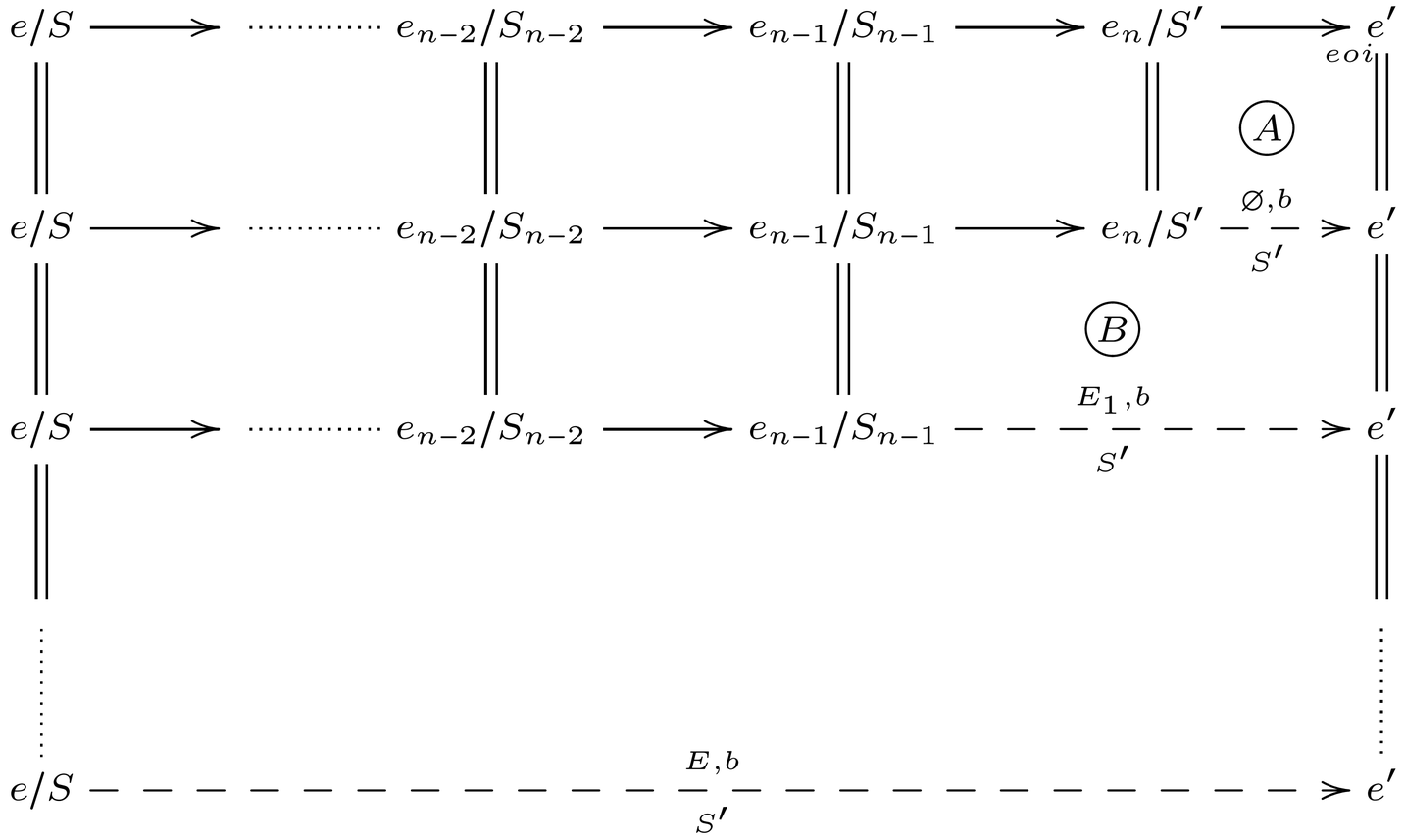
→ termine ("débloque") à la réaction suivante

$$\ldots$$

Avec $O = next(S)$

Propriété : opérationnelle $\Longrightarrow$ comportementale

- $\forall S_{init} \forall e$ si $e/S_{init} \rightarrow e_1/S_1 \rightarrow \ldots e_n/S_n \rightarrow_{eoi} e'$
- alors $\exists N, S, b$ tels que $N \vdash e \xrightarrow[S]{E,b} e'$ avec $E = S^v \backslash S^v_{init}$



Analyse de réactivité

ReactiveML

Motivation

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

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

Motivation

first.rml

```
let process print_top s =
  while true do
    await s; print_endline "top"
  done

let process clock timer s =
  let time = ref (Unix.gettimeofday ()) in
  while true do
    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)
```

Motivation

first.rml

```
let process print_top s =
  while true do
    await s; print_endline "top"
  done

let process clock timer s =
  let time = ref (Unix.gettimeofday ()) in
  while true do
    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)
```

Objectifs

Détecter statiquement les programmes non-réactifs

- boucle instantanée

```
let process instantaneous_loop =
  while true do () done
```

- récursion instantanée

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

On n’émet que des avertissement : faux positifs possibles

Idée

Abstraire les processus par des “comportemens” [Amtoft 99]

- abstraire les valeurs, la présence des signaux, etc.
- garder seulement la structure du programme
- vérifier la réactivité sur les comportements

Limites

- pas d’analyse de valeur
- pas de traitement des fonctions bloquantes (IO)
- on ne prouve pas que les fonctions terminent

Comportements : atomes

actions toujours non-instantanées : $\bullet$

- `pause`,
- `await s(x) in e` …

actions peut-être instantanées : 0

- fonctions ML,
- `await immediate s` …

variables pour abstraire les processus : ϕ

Comportements : structure

Composition parallèle : $\kappa \,||\, \kappa$

```
let process par_comp p q =
  while true do
    run p || run q
  done
```

Choix non-déterministe : $\kappa + \kappa$

```
let process if_comb c p q =
  while true do
    if c then run p else run q
  done
```

← p and q doivent être non-instantanés

Séquence : $\kappa ; \kappa$

- non réactive : `let rec process bad_rec = run bad_rec; pause`
- réactive : `let rec process good_rec = pause; run good_rec`

Comportements : structure

Processus récursif : $\mu\phi\,.\,\kappa$

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

- comportement de good_rec : κ
- comportement du corps : $\bullet\,;\kappa$
- on résout l'équation : $\kappa = \bullet\,;\kappa \implies \kappa \leftarrow \mu\phi\,.\,\bullet\,;\phi$

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

`loop` $e \triangleq$ `run ((rec` $loop$ `=` λx `. process (run` x`; run` $loop\ x$`)) (process` e`))`

Comportements : structure

Autre exemple

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

- comportement de p : κ
- comportement du corps : κ
- on résout l'équation : $\kappa = \kappa$!

Comportements : structure

Autre exemple

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

- comportement de p : κ
- comportement du corps : κ
- on résout l'équation : $\kappa = \kappa$!

Exécuter un processus : $\texttt{run}\ \kappa$

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

- comportement de p : κ
- comportement du corps : $\texttt{run}\ \kappa$
- on résout l'équation : $\kappa = \texttt{run}\ \kappa \implies \kappa \leftarrow \mu\phi\,.\,\texttt{run}\ \phi$

Comportements : résumé

Atomes

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

Structure

- composition parallèle : $||$
- séquence : $;$
- choix non-déterministe : $+$
- opérateur de récursion : $\mu\phi\,.$
- exécuter un processus : $\texttt{run}$

Vérifier la réactivité

Récursion non-instantanée

- la variable de recursion n'apparait pas dans le premier instant du corps

OK

$\mu\phi\,.\,\bullet\,;\phi$

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

KO

$\mu\phi\,.\,\phi$

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

Abstraire les processus

Système de types et effets

- Ajouter un comportement au type des processus
  τ `process`$[\kappa]$
- Associer à chaque expression un type et un comportement
  $\Gamma \vdash e : \tau \,|\, \kappa$

Quelques règles de typage

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

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

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

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

Exemples

first.rml

```
let process clock timer s =
  let time = ref (Unix.gettimeofday ()) in
  while true do
    let time' = Unix.gettimeofday () in
    if time' -. !time ⩾ timer
    then (emit s (); time := time')
  done
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.
```

Exemples

par_comb.rml

```
let process par_comb p q =
  while true do
    run p || run q
  done
val par_comb: 'a process['r1] → 'b process['r2] →
  unit process[rec 'r3. ((run 'r1 || run 'r2); run 'r3)]
```

Exemples

par_comb.rml

```
let process par_comb p q =
  while true do
    run p || run q
  done
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))]
```

Exemples

par_comb.rml

```
let process par_comb p q =
  while true do
    run p || run q
  done
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.
```

Exemples

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.
```

Limites : abstraction des valeurs

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.
```

Limites : fonctions bloquantes (IO)

io.rml

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

Limites : fonctions bloquantes (IO)

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"
```

Limites : fonctions bloquantes (IO)

Alternative

- bibliothèques coopératives
- promesses satisfaites de manière asynchrone (programmation web, javascript)
- exemple OCaml : Lwt

```
let rec process io_lwt () =
  let p = Lwt.readline () in
  match p with
  | Sleep    → pause; run io_lwt ()
  | Return x → x
  | Fail exn → assert false
```

Mais

- pas de structure de contrôle réactif : `do/until`, `do/when`
- pas d'ordre supérieur / polymorphisme : `killable`, `extensible`
- pas de déterminisme synchrone

Limites : pas de preuve de terminaison

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.
```

Limites : pas de preuve de terminaison

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.
```

Liste infinies !

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

Limites : pas de lien avec les ressources

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)))]
```

Un problème de typage

```
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
```

Sous-effet

Idée

- sous-effet : sous-typage des effets
- inspiré des types rangés [Remy 93]
- un processus à au moins le comportement de son corps
- nouvelle règle de typage

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

Tous les programmes ReactiveML corrects ont un comportement

Exemple

```
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
```

Propriété : correction

Les programmes bien typés sont réactifs

Propriété (correction)

- si $\Gamma \vdash e : \tau \,|\, \kappa$ et τ et κ sont réactifs
- si touts les appels de fonction terminent
- alors $\exists e'$ tel que $N \vdash e \xrightarrow[S]{E,b} e'$ avec $\Gamma \vdash e' : \tau \,|\, \kappa'$ et κ' est réactif

Schéma de preuve

- définir le premier instant d'un comportement
- pour un programme bien typé, le premier instant d'un comportement est fini (nombre fini de dérivations)
- induction : le typage est préservé au cours du temps

Attention: la réciproque est fausse (faux positifs possibles)

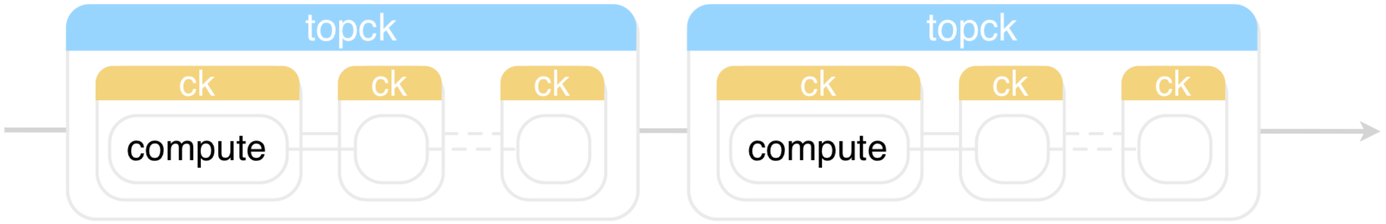
Domaines d’horloges

ReactiveML

Domaines d’horloges (C. Pasteur)

Idée

- Notion d’instant local
- Masquer les instants locaux



Exemple : simulation des n-corps

- méthodes d’intégration numérique 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
    while true do
      await s_in(v) in
      let res = run compute ck v in
      emit s_out res
    done
  done
```

Une nouvelle analyse statique

Limitation de 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
- pas de dépendance immédiate sur les signaux plus lent

Un système de type pour prévenir l'échappement de portée

- système de types et effets
- extension du typage de ML

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

Exemple

```
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
```

Exemple

```
let process p3 =
  signal s in n
  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.
```

http://reactiveml.org