Réactivité des systèmes coopératifs : le cas de ReactiveML

Cédric Pasteur, Louis Mandel
Equipe PARKAS, Département d'Informatique, Ecole normale supérieure,
Paris, France
cedric.pasteur@ens.fr

5 févr. 2013

ReactiveML (http://rml.lri.fr)

Une extension réactive de ML

- Noyau fonctionnel d'ordre supérieur
- Modèle de concurrence synchrone
  - Notion d'instant logique global
  - Communication par diffusion instantanée (broadcast)

Application

- Simulation (par ex. réseau de capteurs)

Un premier exemple

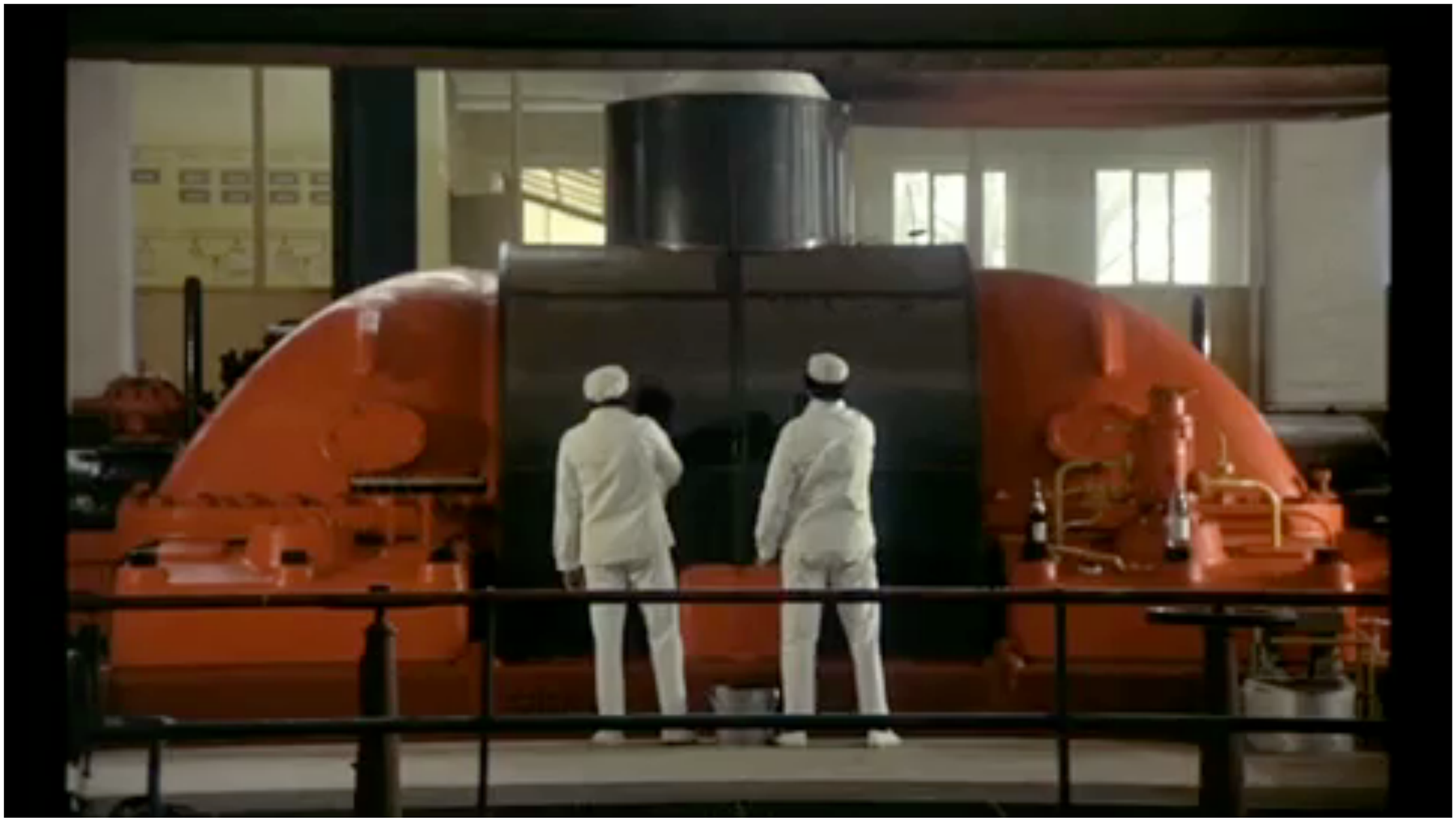
Je ne sais rien mais je dirai tout (Pierre Richard, 1973)

http://www.youtube.com/watch?v=C1Y9hpovDeg

Un premier exemple

Je ne sais rien mais je dirai tout (Pierre Richard, 1973)

http://www.youtube.com/watch?v=C1Y9hpovDeg



Un premier exemple

Concurrence avec des threads

```
let brush draw p () =
  let st = new_state p in
  while true do
    move draw st
  done

let main () =
  let draw = init () in
  let t1 = Thread.create (brush draw p1) ()
  and t2 = Thread.create (brush draw p2) () in
  Thread.join t1; Thread.join t2
```

Un premier exemple

Concurrence synchrone

```
let process brush draw p =
  let st = new_state p in
  loop
    move draw st
  end

let process main =
  let draw = init () in
  run (brush draw p0) || run (brush draw p1)
```

Un premier exemple

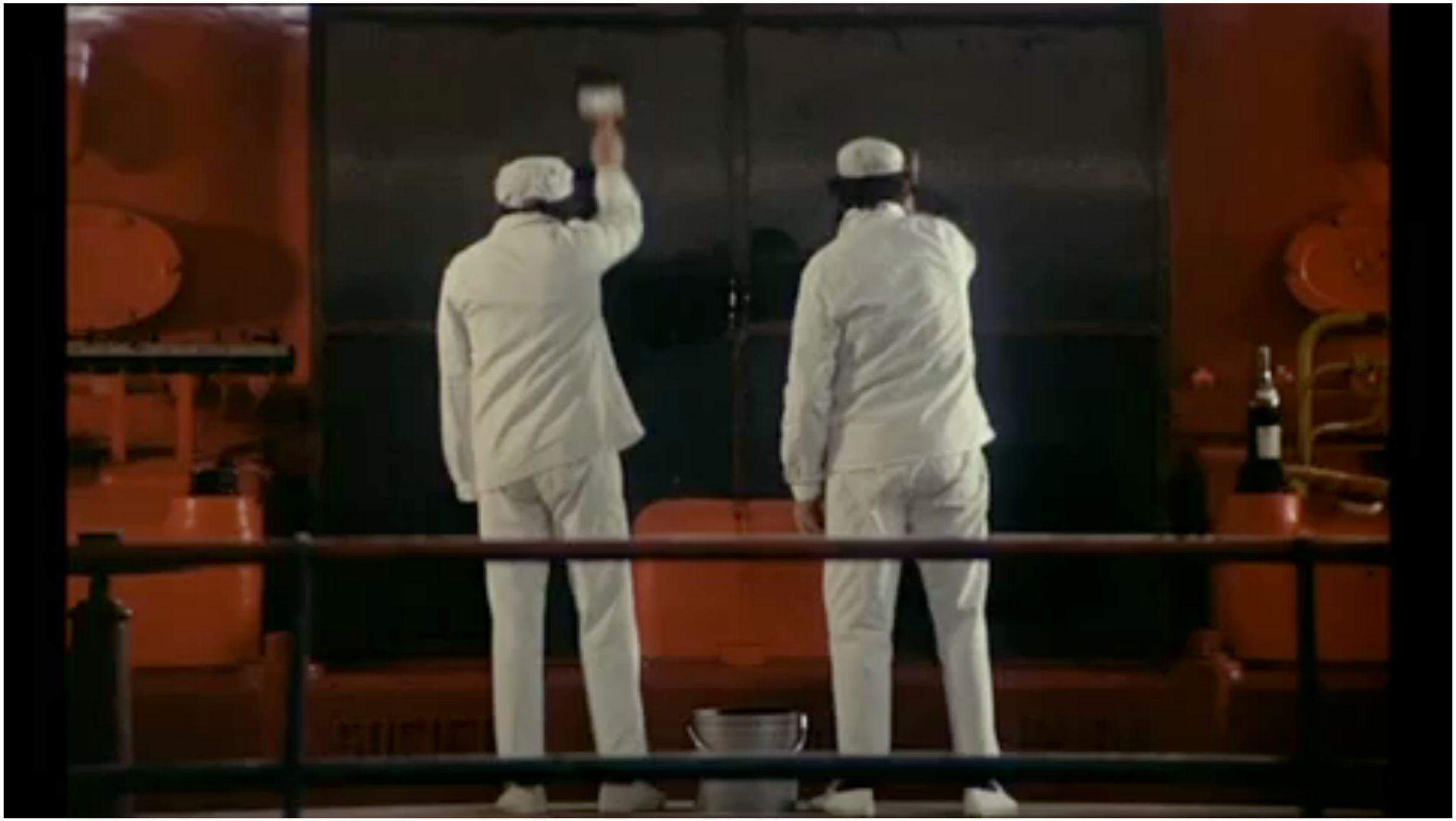
Concurrence synchrone (corrigé)

```
let process brush draw p =
  let st = new_state p in
  loop
    move draw st;
    pause
  end

let process main =
  let draw = init () in
  run (brush draw p0) || run (brush draw p1)
```

Un premier exemple

Un premier exemple



Un problème général

Ordonnancement coopératif

- Lwt [Vouillon08], Async [1] ou muthreads [Deleuze13]
- Il faut rendre la main à l'ordonnanceur

Exemple

- Documentation de Lwt [2]

Rules: [...]
- *do not write function that may takes time to complete without using **Lwt***
- *do not do IOs that may block, otherwise the whole program will hang.*

You must instead use asynchronous IOs operations.

Objectif

Détecter statiquement les processus (potentiellement) non coopératifs/non réactifs

- Boucle instantanée

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

- Récursion instantanée

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

Objectif

Détecter statiquement les processus (potentiellement) non coopératifs/non réactifs

- Boucle instantanée

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

- Récursion instantanée

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

Condition suffisante

- Un instant pour le corps d'une boucle
- Un instant avant chaque appel récursif

Principe

Abstraire les processus en "comportements"

- Amtoft et al: *Type and Effect Systems: Behaviours for Concurrency.* 1999

```
let process p i q r =
  let y = i mod 4 in
  run q || (pause; if y = 1 then run r)
```

$$0; (\mathtt{run}\,\phi_q \,||\, (\bullet; (\mathtt{run}\,\phi_r + 0)))$$

Vérifier la réactivité sur les comportements

Processus récursifs

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

Processus récursifs

Processus récursifs

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

Processus récursifs

- Comportement de `good_rec` : κ

Processus récursifs

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

Processus récursifs

- Comportement de `good_rec` : κ
- Comportement du corps : $\bullet; \mathtt{run}\,\kappa$

Processus récursifs

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

Processus récursifs

- Comportement de `good_rec` : κ
- Comportement du corps : $\bullet; \mathtt{run}\,\kappa$
- Opérateur de récursion explicite : $\kappa = \mu\phi.\, \bullet; \mathtt{run}\,\phi$

Processus récursifs

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

Processus récursifs

- Comportement de `good_rec` : κ
- Comportement du corps : $\bullet; \mathtt{run}\,\kappa$
- Opérateur de récursion explicite : $\kappa = \mu\phi.\, \bullet; \mathtt{run}\,\phi$
- Boucle : $\kappa^{\infty} = \mu\phi.\, \kappa; \mathtt{run}\,\phi$

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

Comportements

Base

- (Potentiellement) Instantané 0
- (Sûrement) Non-instantané $\bullet$
- Variable ϕ

Structure

- Composition parallèle $||$
- Séquence $;$
- Choix non-déterministe $+$
- Récursion $\mu\phi.$
- Lancement de processus $\texttt{run}$

Vérification de réactivité

Récursion non-instantanée

- La variable de récursion n'apparait pas dans le premier instant du corps
- Exemples

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

Résumé

Abstraire les processus en comportements

- Abstraction des valeurs, de la présence des signaux
- Mais on garde la structure du programme
- On suppose que les fonctions OCaml terminent

Limites

- Pas d'analyse de terminaison
- Pas de cas spécial pour les fonctions bloquantes (E/S)

Abstraction des processus

Système de types-et-effets

- Comportement dans le type des processus

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

- On associe à chaque expression un type et un comportement

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

Algorithme

- Calcul des comportements
- Vérification de réactivité

Règles de typage

Cas de base

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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \mid 0 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2 \qquad \Gamma \vdash e_3 : \tau \mid \kappa_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \kappa_2 + \kappa_3}$$

Règles de typage

Types-et-effets

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

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

Exemple

Premier exemple

```
let process brush draw p =
  let st = new_state p in
  loop
    move draw st
  end
```

```
val brush : draw_fun -> pos ->
              unit process[0; rec 'r0. (0; run 'r0)]
Warning: This expression may produce an instantaneous recursion.
```

Exemple

Premier exemple corrigé

```
let process brush draw p =
  let st = new_state p in
  loop
    move draw st;
    pause
  end
```

```
val brush : draw_fun -> pos ->
              unit process[0; rec 'r0. (0; *; run 'r0)]
```

Exemple

Premier exemple corrigé

```
let process brush draw p =
  let st = new_state p in
  loop
    move draw st;
    pause
  end

val brush : draw_fun -> pos ->
              unit process[rec 'r0. (*; run 'r0)]
```

Combinateurs

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

```
val par_comb : unit process['r1] -> unit process['r2]
                 -> unit process[rec 'r3. (run 'r1 || run 'r2); 'r3]
```

Combinateurs

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

```
val par_comb : unit process['r1] -> unit process['r2]
               -> unit process[rec 'r3. (run 'r1 || run 'r2); 'r3]
```

```
let process good =
    run (par_comb (process ()) (process (pause)))
```

```
val good : unit process[rec 'r. (run 0 || run *); 'r]
```

Combinateurs

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

```
val par_comb : unit process['r1] -> unit process['r2]
               -> unit process[rec 'r3. (run 'r1 || run 'r2); 'r3]
```

```
let process good =
  run (par_comb (process ()) (process (pause)))
```

```
val good : unit process[rec 'r. (run 0 || run *); 'r]
```

```
let process bad =
  run (par_comb (process ()) (process ()))
```

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

Combinateurs

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

```
val par_comb : unit process['r1] -> unit process['r2]
               -> unit process[rec 'r3. (run 'r1 || run 'r2); 'r3]
```

```
let process good =
  run (par_comb (process ()) (process (pause)))
```

```
val good : unit process[rec 'r. *; 'r]
```

```
let process bad =
  run (par_comb (process ()) (process ()))
```

```
val bad : unit process[rec 'r.'r]
Warning: This expression may produce an instantaneous recursion.
```

Limites

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

```
val par_iter: ('a -> unit process['r0]) -> 'a list ->
                    unit process[rec 'r1. (0 + (run 'r0 || run 'r1))]
Warning: This expression may produce an instantaneous recursion.
```

Sous-typage des effets

```
let process p = pause
```

```
val p : unit process[*]
```

```
let process q = ()
```

```
val p : unit process[0]
```

Sous-typage des effets

```
let process p = pause
```

```
val p : unit process[*]
```

```
let process q = ()
```

```
val p : unit process[0]
```

```
let l = [p; q]
```

```
val p : unit process[??] list
```

Sous-typage des effets

```
let process p = pause
```

```
val p : unit process[* + 'r]
```

```
let process q = ()
```

```
val p : unit process[0 + 'r]
```

```
let l = [p; q]
```

```
val p : unit process[0 + * + 'r] list
```

- Subeffecting [Nielson99]
- Type rangées [Remy93]
- Extension conservative du système de types

Conclusion

Analyse de réactivité

- Abstraction des processus
- Système de types-et-effets
- Implémenté dans ReactiveML (http://rml.lri.fr/jfla13)
- Toplevel en ligne (http://rml.lri.fr/jfla13/tryrml/)
- Preuve de correction (pas dans le papier)

Perspectives

- Propriétés du système de types (principalité)
- Adaptation pour d'autres modèles de concurrence

Références

[Amtoft99] T. Amtoft, F. Nielson, and H. Nielson. ***Type and Effect Systems: Behaviours for Concurrency.*** Imperial College Press, 1999.

[Vouillon08] J. Vouillon. ***Lwt: a cooperative thread library.*** In Proceedings of the 2008 ACM SIGPLAN workshop on ML, pages 3–12. ACM, 2008.

[Deleuze13] Christophe Deleuze. ***Concurrence légère en OCaml : muthreads***. JFLA 2013

[1] https://bitbucket.org/yminsky/ocaml-core/overview
[2] http://ocsigen.org/lwt/manual

Références

[Nielson99] F. Nielson and H. Nielson. ***Type and effect systems.*** Correct System Design, pages 114–136, 1999.

[Remy93] D. Rémy. ***Type inference for records in a natural extension of ML.*** Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press, 1993.

Analyse de valeur

```
let rec process imprecise () =
  signal s in
  present s then () else (*pause implicite*) () in
  run imprecise
```

```
val imprecise : 'a process[(rec 'r1. run 'r1) + 'r2]
Warning: This expression may produce an instantaneous recursion.
```

Preuve de correction

Théorème

- Si on suppose que les fonctions terminent, alors un programme bien typé est réactif

Principe de la preuve

- Le premier instant d'un effet réactif est fini (c-à-d pas de récursion)
- Preuve par récurrence sur la taille du comportement
  - Sémantique à grands pas

Structure des comportements

Structure des comportements

- Un autre example

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

$$\kappa \equiv \kappa$$

Structure des comportements

- Un autre example

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

$$\kappa \equiv \kappa$$

- Lancement de processus : **run** κ

Structure des comportements

- Un autre example

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

$$\kappa \equiv \kappa$$

- Lancement de processus : $\mathbf{run}\,\kappa$
- Force le comportement d'un processus récursif à être un comportement récursif:

$$\kappa \equiv \mathbf{run}\,\kappa \qquad\qquad \kappa = \mu\phi.\,\mathbf{run}\,\phi$$

Opérateur de point fixe

```
let rec fix f x = f (fix f) x

let process main =
  let process p k v =
    print_int v; print_newline ();
    run (k (v+1))
  in
  run (fix p 0)
```

```
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
val main : 'a process[run rec 'r0. run 'r0]
Warning: This expression may produce an instantaneous recursion.
```

Références

```
let landin () =
  let f = ref (process ()) in
  f := process (run !f);
  !f
```

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

But ?

Such analysis already exists

- Implemented in ReactiveML v1.06 (dec. 2006)
- Presented in SYNCHRON 2007

What am I doing here ?

- A more precise analysis
- Can be extended to a more general setting
  - Reactive/clock domains
  - Multiple clocks/time scales