Exécution efficace de programmes ReactiveML

Louis Mandel Cédric Pasteur

Collège de France
École normale supérieure
INRIA Paris-Rocquencourt

JFLA 2014

ReactiveML

Un langage de programmation de systèmes interactifs :

- un langage généraliste
- modèle de concurrence coopératif

Exemples de programmes ReactiveML :

- une solution du concours ICFP 2013
- "Carrés Noir et Blanc" de Roger Vilder
  `http://www.rogervilder.com/projets/carre_16.html`

Pompe


```
let process sum state delta =
  loop
    emit state (last ?state +. delta);
    pause
  end
val sum: (float, float) event -> float -> unit process
```

Pompe

```
let process incr_decr state delta =

  let rec process incr =
    do run sum state delta
    until state(x) when x >= 1. -> run decr done

  and process decr =
    do run sum state (-. delta)
    until state(x) when x <= 0. -> run incr done

  in
  run incr
val incr_decr: (float, float) event -> float -> unit process
```

Pompe

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

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

Pompe

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

A la Roger Vilder


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

ReactiveML

Langage généraliste similaire à Caml

Concurrence synchrone similaire à Esterel

- instants logiques
- composition parallèle synchrone
- diffusion instantanée d'événements

Extension réactive de Boussinot

- création dynamique de processus
- ordonnancement dynamique

ReactiveML

Langage généraliste similaire à Caml

Concurrence synchrone similaire à Esterel

- instants logiques
- composition parallèle synchrone
- diffusion instantanée d'événements

Extension réactive de Boussinot

- création dynamique de processus
- ordonnancement dynamique

Concurrence avec des continuations [Wand, LISP 1980]

ReactiveML

Principes de l'implantation

Compilation

ReactiveML :

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

Compilation

ReactiveML :

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

$\mathcal{L}_k$: un langage à base de continuations

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

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

CPS partielle

Séparation des expressions instantanées et réactives :

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

Traduction de ReactiveML vers $\mathcal{L}_k$:

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

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

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

$$\ldots$$

Runtime

Structures de données :

- *current* : ensemble des continuations à exécuter à l'instant courant
- *next* : ensemble des continuations à exécuter à l'instant suivant
- *wait* : ensemble des files d'attente sur les signaux

Exécution :

- exécuter le contenu de l'ensemble *current*
- préparer la réaction de l'instant suivant (réaction de fin d'instant) :
  - réagir à l'absence, récupérer la valeur des signaux, ...
  - transférer *next* dans *current*

ReactiveML

Implantation

Plongement dans OCaml

Programme source :

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

Code généré (après pretty-print) :

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

ReactiveML comme une bibliothèque

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


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

val rml_pause : unit step -> 'a step

val rml_await_immediate :
  (unit -> ('a, 'b) event) -> unit step -> 'c step

val rml_await_all :
  (unit -> ('a, 'b) event) -> ('b -> unit step) -> 'c step

...
```

Implantation de la bibliothèque : `rml_compute`

L'expression ReactiveML :

```
print_string "hello"; print_newline ()
```

se traduit avec la continuation `k` en :

```
rml_compute (fun () -> print_string "hello"; print_newline ()) k
```

La fonction `rml_compute` est définie par :

```
let rml_compute e k = (fun _ -> k (e ()))
```

Implantation de la bibliothèque : `rml_await_immediate`

L'expression ReactiveML :

```
await immediate !s
```

se traduit avec la continuation `k` en :

```
rml_await_immediate (fun () -> !s) k
```

La fonction `rml_await_immediate` est définie par :

```
let rml_await_immediate expr_evt k =
  (fun _ -> Runtime.on_event (expr_evt()) k ())
```

Gestion des signaux

Un signal est représenté par :

- son état : `n`
- une file d'attente "inter-instant" : `wa`
- une file d'attente "intra-instant" : `wp`

`on_event ev k` exécute `k` à l'émission de `evt`.

```
let on_event (n, wa, wp) k =
  if Event.status n then k () else wa := k :: !wa
```

Gestion des signaux

`on_event_or_next ev f_ev f_n` exécute

- `f_ev` si le signal est émis,
- sinon exécute `f_n` à l'instant suivant.

```
let on_event_or_next (n, wa, wp) f_ev f_n =
  let act () = if is_eoi () then next := f_n :: !next else f_ev () in
  if Event.status n then f_ev ()
  else (wp := act :: !wp; weoi := wp :: !weoi)
```

ReactiveML

Suspension et préemption

Préemption:

```
let rec process resetable r p =
  do
    run p
  until r -> run resetable r p done
```

Structures de contrôle

Préemption:

```
let rec process resetable r p =
  do
    run p
  until r -> run resetable r p done
```

Suspension:

```
let rec process suspendable s p =
  control
    run p
  with s done
```

Difficulté

Conserver la structure du programme

```
signal s, k in
do
  pause
  ||
  await s
until k done
||
emit k
```

Arbre de contrôle

Structure de données qui :

- conserve la structure des préemptions et suspensions
- associe une liste *next* à chaque noeud de l'arbre

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

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

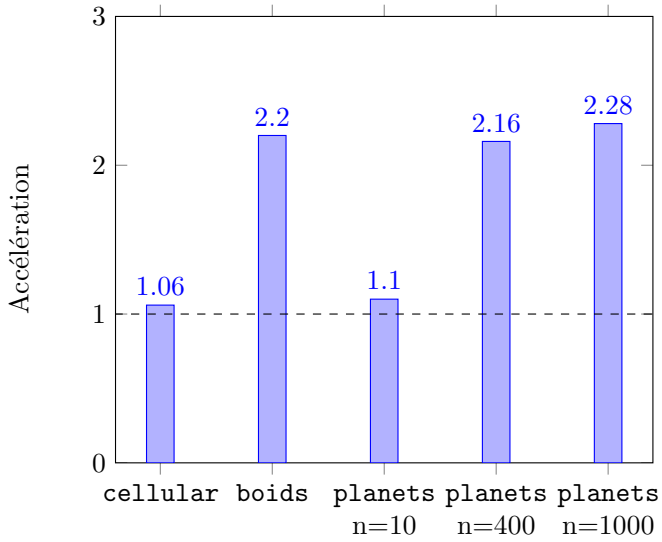
Conclusion

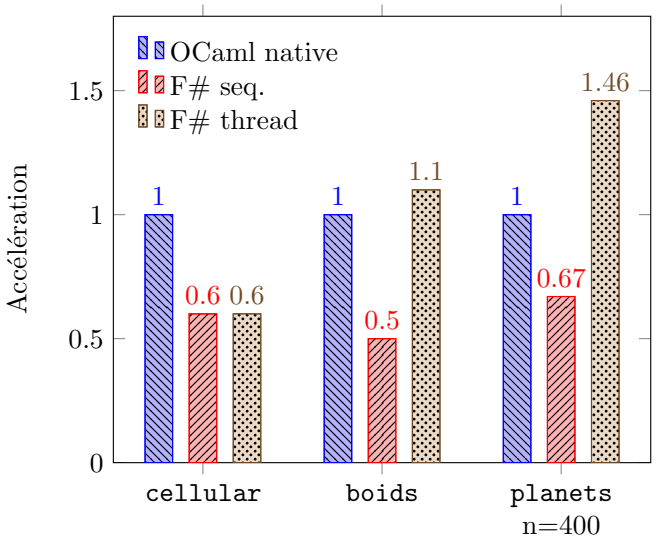
Génération de code séquentiel

- profite des outils : `ocamlopt`, `js_of_ocaml`, ...

Génération de code parallèle

- F# avec vol de tâches





`http://reactiveml.org`