(An improved) Reactivity analysis in ReactiveML

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

16 nov. 2012

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

A reactive extension of ML

- Higher-order functional core
- Synchronous language constructs
- Dynamic creation
- No real-time constraints

Application

- Orchestration
- Discrete simulation (eg. sensor network)

A first example

Print a message every «second»

```
let process pclock timer s =
  let time = ref (Unix.gettimeofday ()) in
  loop
    let time' = Unix.gettimeofday () in
    if time' -. !time >= timer
    then (emit s (); time := time')
end
```

A first example

Print a message every «second»

```
let process print_clock s =
  loop
    await s;
    print_endline "top"
  end

let process main =
  signal s in
  run (pclock 1.0 s) || run (print_clock s)
```

A first example

Nothing happens !

- The loop is instantaneous
- Non-reactive program
- Scheduling is coperative

```
let process pclock 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
```

Goal

Statically detect (potentially) non-reactive programs

- Instantaneous loops

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

- Instantaneous recursion

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

Only warnings

- False positives

But ?

Such analysis already exists

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

Louis Mandel

Université Paris-Sud 11, LRI

Reactivity of ReactiveML programs

A new static analysis for ReactiveML

SYNCHRON 2007 – 28/11/2007

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

Overview

The idea

Abstract processes into «behaviors» [Amtoft, 99]

- Express the reactive behavior of a process
- Check reactivity on the behaviors
- Abstract away values, signal presence, etc.
- But keep some structure

Limitations

- No value analysis
- We don't prove that functions terminate
- No special case for blocking functions (IOs)

Behaviors

Atoms

- (Surely) Non-instantaneous action: •
  - `pause`, `await` `s(v)` `in` `e`, etc.
- (Maybe) Instantaneous action: 0
  - ML functions, `await immediate`
- Variables for function arguments: ϕ

Behaviors structure

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

- `q1` **or** `q2` must be non-instantaneous

```
let process if_comb c q1 q2 =
  loop
    if c then run q1 else run q2
  end
```

- `q1` **and** `q2` must be non-instantaneous
- Add parallel composition (`||`) and non-deterministic choice (+)

Behaviors structure

- Not reactive:

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

- Reactive:

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

- Add sequence (;) in behaviors

Behaviors structure

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

Dealing with recursive processes

- Behavior of `good_rec` : κ
- Behavior of the body : $\bullet; \kappa$
- Use explicit recursion operator: $\kappa = \mu\phi.\, \bullet; \phi$
- Loop: $\kappa^{\infty} = \mu\phi.\, \kappa; \phi$

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

Behavior structures

- Another example

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

$\kappa \equiv \kappa$

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

- Force the behaviors of a recursive process to be a recursive behavior

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

Behaviors

Atoms

- Instantaneous 0
- Non-instantaneous $\bullet$
- Variable ϕ

Structure

- Parallel composition $||$
- Sequential composition $;$
- Non-deterministic choice $+$
- Recursion operator $\mu\phi.$
- Running a process $\texttt{run}$

Checking reactivity

Non-instantaneous recursion

- The recursion variable does not appear in the first instant of the body
- Also works for loops: $\kappa^{\infty} = \mu\phi.\, \kappa; \phi$
- Examples

OK	Not OK
$\mu\phi.\, \bullet; \phi$	$\mu\phi.\, \phi$
$\bullet^{\infty}$	0^{∞}
$\mu\phi.\, (0 + \bullet; \phi)$	$\mu\phi.\, (0 + \bullet); \phi$

Abstracting processes

Type-and-effect system

- Add a behavior to the type of processes

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

- Associate to each expression a type and a behavior

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

Algorithm

- Compute the behavior associated to each process
- Check its reactivity

Some typing rules

Basic cases

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

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

Some typing rules

Type-and-effect system

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

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

Example

First example

```
let process pclock 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 pclock : unit process[0; rec 'r0. (0; (0;0 + 0); 'r0)]
Warning: This expression may produce an instantaneous recursion.
```

Example

First example fixed

```
let process pclock 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
```

```
val pclock : unit process[0; rec 'r0. (0; (0;0 + 0); *; 'r0)]
```

Example

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

```
val par_comb : unit process['r0] -> unit process['r1]
               -> unit process[rec 'r2. run 'r0 || run 'r1 ; 'r2]
```

```
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.
```

Example

Fix-point operator

```
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.
```

Limitation

```
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.
```

Overview

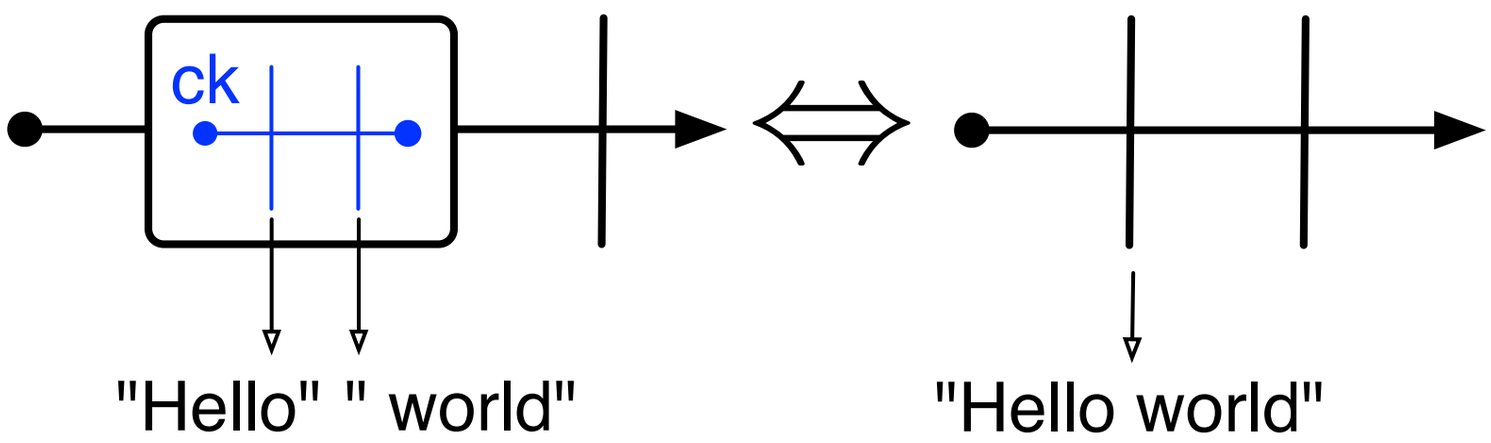
Reactive domains

Local time scales

- Instants are unobservable from the outside
- aka Clock domains (**`newclock`**)

```
signal s default "" gather (fun x y -> y^x)

let process hello_world_ck =
domain ck do
   emit s "Hello"; pause ck;
   emit s " world!"
done
```

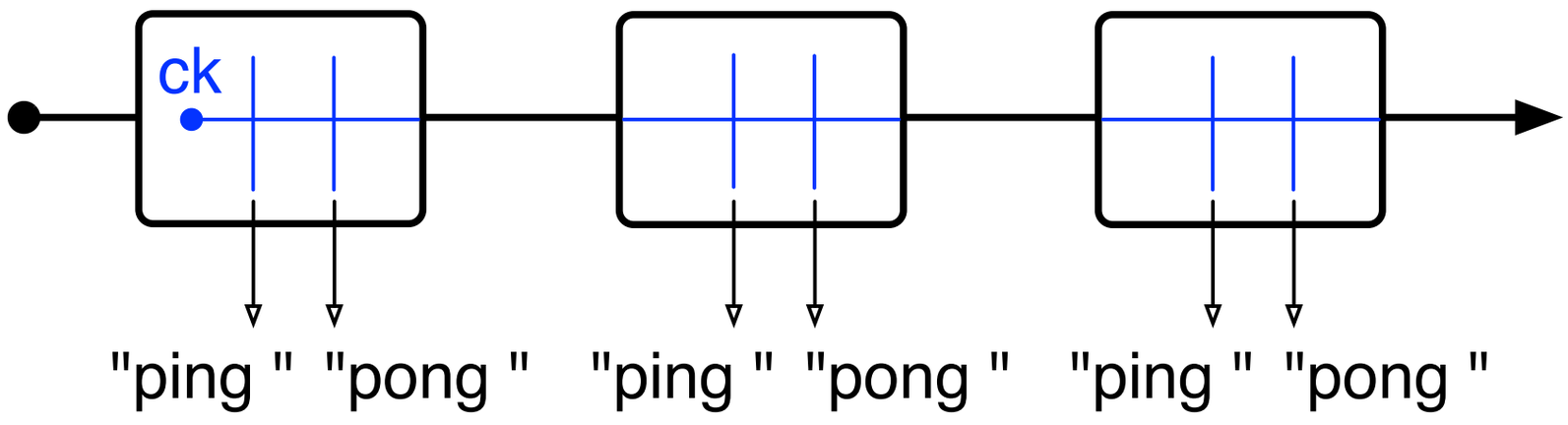


Reactive domains

A periodic reactive domain

```
signal s default "" gather (fun x y -> y^x)

let process ping_pong =
  domain ck do
    loop
      emit s "ping "; pause ck;
      emit s "pong "; pause topck
    end
done
```



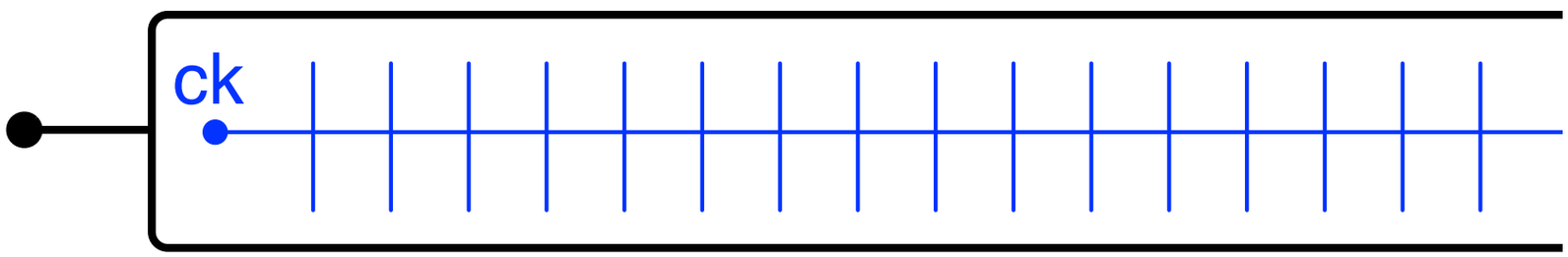
A new reactivity problem

Non-reactive domains

- Like an infinite loop

```
signal s default "" gather (fun x y -> y^x)

let process ping_pong =
  domain ck do
    loop
      emit s "ping "; pause ck;
      emit s "pong "
    end
done
```



Extending our analysis

Behaviors

- Replace • with `ck`

Typing rule for domains

$$\frac{\Gamma, x : \{\gamma\} \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{domain}(x)\, e : \tau \mid \kappa[\gamma \leftarrow 0]}$$

Example

A non-reactive domain

```
signal s default "" gather (fun x y -> y^x)

let process ping_pong =
  domain ck do
    loop
      emit s "ping "; pause ck;
      emit s "pong "
    end
done
```

$$(0;\gamma;0)^{\infty} \longrightarrow (0;0;0)^{\infty}$$

Warning: This reactive domain may not be reactive.

Overview

Introduction

Reactivity analysis

Extension to the multi-clock case

Towards a static analysis

Conclusion

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
```

An extension of the type system

Idea

- Subeffecting = subtyping of effects
- Inspired by row types [Remy, 93]
- New typing rule

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

- A process has **at least** the behavior of its body

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 + 'r0]
```

```
let l = [p; q]
val l : unit process[* + 0 + 'r0] list
```

Overview

Introduction

Reactivity analysis

Extension to the multi-clock case

Towards a static analysis

Conclusion

Try ReactiveML online

An interactive toplevel in a browser

- Work by Mehdi Dogguy
- Based on a compiler from OCaml bytecode to Javascript (http://ocsigen.org/js_of_ocaml/)
- Extends TryOCaml (http://try.ocamlpro.com/)
- http://rml.lri.fr/tryrml

Try ReactiveML online

How does it work ?

- ReactiveML compiler + OCaml interpreter compiled to Javascript
- Execution
  - ReactiveML phrase compiled to OCaml phrase
  - OCaml phrase executed by OCaml interpreter
- No connection required (only to download the whole page ~4 MB !!)

Conclusion

Improved reactivity analysis

- Abstract process into behaviors
- Type-and-effect system
- Implemented in ReactiveML (http://rml.lri.fr)
- Paper submitted to JFLA'13 (in French)

Future work

- Proof of soundness
- Proof of completeness and principality

References

- [Amtoft, 99]: T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
- [Remy, 93]: 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.

More examples

```
let higher_order f =
  let rec process p =
    let q = f p in
    run q
  in p

let process good =
  run (higher_order
         (fun p -> process (pause; run p)))

let process pb =
  run (higher_order
         (fun p -> process (run p)))
```

```
val higher_order : ('a process[run 'r1] -> 'a process['r1])
                        -> 'a process[run 'r1]
val good : 'a process[run (run (rec 'r1. * ; run (run 'r1)))]
val pb : 'a process[run (run (rec 'r2. run (run 'r2)))]
Warning: This expression may produce an instantaneous recursion.
```

More examples

```
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.
```