ICFP, Saint-Louis, USA
G. Baudart, L. Mandel,
C. Pasteur, & M. Pouzet

Implement your ReactiveML runtime

The goal of this exercise is to present the implementation of the ReactiveML runtime. The ReactiveML compiler with the skeleton of a runtime to complete is available here:

```
http://reactiveml.org/icfp18/rml-1.09.06-dev-2018-09-29.tar.gz
```

You can test your compiler with the examples provided in `examples/tutorial`. The following command will compile and execute the tests and compare them with the expected output.

```
--> ./test.sh
```

Question 1
Compile and install this version of the compiler:

```
--> ./configure [--prefix PATH]
--> make
--> make install
```

We are going complete the runtime which is defined in `interpreter/lk_tutorial.ml`. To use this runtime, a file `a.rml` must be compiled and linked with the compiler option `-runtime Lk_tutorial`.

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

This runtime must implement the `interpreter/lk_interpreter.mli` interface.

1 Parallel composition

First, let us look at the parallel composition. Consider the following ReactiveML code:

```
let process par p q =
  run p || run q
```

The generated OCaml code is to following:

```
let par p q k ctrl =
  Lk_tutorial_record.rml_split_par 2
    (fun j ->
       let kj _ = Lk_tutorial_record.rml_join_par j k ()
       in
         [ Lk_tutorial_record.rml_run_v p kj ctrl;
           Lk_tutorial_record.rml_run_v q kj ctrl ])
```

Question 2
Update the implementation of the functions `rml_split_par`, `sched`, *and* `rml_join_par` *to implement the parallel composition.*

```
val rml_split_par:
    int -> (join_point -> (unit step) list) -> 'a step
val sched: unit -> unit
val rml_join_par: join_point -> unit step -> 'a step
```

These functions will use the `current` *state variable.*

ReactiveML also support parallel definitions as in the following example:

```
let process letand p q =
  let a = run p
  and b = run q in
  a + b
```

The corresponding generated code is:

```
let letand p q k ctrl =
  let body v =
    let (a, b) = v in
    Lk_tutorial_record.rml_compute (fun () -> a + b) k ()
  in
  Lk_tutorial_record.rml_split_par 2
    (fun j ->
       let a_ref = Pervasives.ref 4012
       and b_ref = Pervasives.ref 4012 in
       let get_vrefs () = (!a_ref, !b_ref) in
       [ Lk_tutorial_record.rml_run_v p
           (Lk_tutorial_record.rml_join_def j a_ref
              get_vrefs body)
           ctrl;
         Lk_tutorial_record.rml_run_v q
           (Lk_tutorial_record.rml_join_def j b_ref
              get_vrefs body)
           ctrl ])
```

Question 3
Implement the function `rml_join_def` *that synchronize the termination of a parallel definition.*

```
val rml_join_def:
    join_point -> 'a ref -> (unit -> 'b) -> 'b step -> 'a step
```

The first argument is the synchronization point, the second one is where to store the result of the branch, the third argument is a function that allows to read the result of all the branches and the last one is the continuation.

2 Logical time

Currently, the `pause` expression is not implemented.

Question 4
Add a list `next` *of continuations to execute to the next instant in the global state.*

Question 5
Update the code of `rml_pause`*:*

```
val rml_pause: unit step -> control_tree -> 'a step
```

Question 6
Update the function `rml_make` *to prepare the execution of the next instant.*

3 Communication

Question 7
Implement the `emit` *and* `present` *constructs.*

```
val rml_present_v:
    control_tree -> ('a, 'b) event -> unit step -> unit step -> 'c step
val rml_emit_v_v: ('a, 'b) event -> 'a -> unit step -> 'c step
```

The function `Event.status n` *return a Boolean indicating if a value has been emitted on the signal* `n`*. The function* `Event.emit n v` *update the data structure* `n` *with the information of the emission of the value* `v`*.*

Hint 1: these constructs might require the creation of a global flag `eoi` *indicating the end of instant.*

Hint 2: do not forget to prepare the next state. The function `Event.next ()` *updates the signal environment for the next instant.*

In the current implementation, the `await immediate` construct is implemented like:

```
let rec process await_immediate s =
  preset s then () else run await_immediate s
```

This implementation requires to test the presence of `s` at each instant. Therefore, the execution of the following ReactiveML program is pretty slow:

```
let rec process slow acc i =
  if i > 0 then
    signal s in
    await s; print_int i; print_newline() ||
    run slow (s :: acc) (i - 1)
  else
    run Rml_list.iter (proc s -> pause; emit s) acc

let () =
  run slow [] 10000
```

Question 8
Modify the implementation of `rml_await_immediate_v` *to allows passive waiting.*

The expression `await s(x) in ...` can be decompose in two steps, await the presence of the signal and get its value: `await immediate s; let s<x> in ...`.

The construct `let s<x> in ...` bind to `x` the value of the signal `s` in its body. The body is executed at the next instant. If the signal is not emitted, `x` takes the default value of the signal.

Question 9
Update the implementation of `rml_get_v`.

```
val rml_get_v:
  ('a, 'b) event -> ('b -> unit step) -> control_tree -> 'c step
```

The function `Event.value n` *get the current value of the signal* `n` *and* `Event.default n` *gets its default value.*

4 Control structure

ReactiveML provides high level control structures that allows to interrupt or suspend the execution of a process. A process that can be killed can be programmed as follows:

```
let process killable s p =
  do run p
  until s done
```

The corresponding generated code is:

```
let killable s p k ctrl =
  Lk_tutorial_record.rml_start_until_v ctrl s
    (fun ctrl_until ->
       Lk_tutorial_record.rml_run_v p
         (Lk_tutorial_record.rml_end_until ctrl_until k) ctrl_until)
    (fun _ -> k)
```

Question 10
Update the implementation of `rml_start_until_v`.

```
val rml_start_until_v:
    control_tree -> ('a, 'b) event ->
      (control_tree -> unit step) -> ('b -> unit step) -> 'c step
```

This function will use the `control_tree` *data structure. The function* `new_ctrl kind cond` *creates a new control tree node. Here, the node should be of kind* `Kill`*. The function* `eval_control_and_next_to_current` *handle the treatment of the control tree at the end of instant. It must be called in the* `rml_make` *function.*

Question 11

Update the implementation of `rml_start_when_v` *to implement the suspension.*

```
val rml_start_when_v:
    control_tree -> ('a, 'b) event ->
      (control_tree -> unit step) -> 'c step
```