Reactivity of Cooperative Systems

Application to ReactiveML

Louis Mandel
LRI, Université Paris-Sud 11, Orsay, France
louis.mandel@lri.fr

Cédric Pasteur
DI, École normale supérieure, Paris, France
cedric.pasteur@ens.fr

Abstract

Cooperative scheduling enables efficient sequential implementation of concurrency. It is widely used to provide light-threads facilities in functional languages. However, it is up to the programmer to actually cooperate to ensure the reactivity of the program.

We present a static analysis that checks the reactivity of programs by abstracting them into so-called *behaviors* using a type-and-effect system. Our work is applied and implemented in the functional synchronous language ReactiveML. We prove the soundness of our analysis with respect to the big-step semantics: a well-typed program is reactive. Our analysis does not show too much false positives and has proven very useful for avoiding reactivity bugs. Furthermore, it is easy to implement and generic enough to be applicable to other models of concurrency.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 [*Processors*]: Compilers

General Terms Languages, Theory

Keywords Synchronous languages; Functional languages; Semantics; Type systems; Cooperative scheduling

1. Introduction

Most functional languages offer light-thread facilities, either integrated in the language like the asynchronous computations [23] in F#, or available as a library like Concurrent Haskell [13] or LWT [26] for OCaml. These libraries are based on cooperative scheduling: each thread of execution cooperates with the scheduler to let other threads execute. This enables an efficient and sequential implementation of concurrency, allowing to create up to millions of separate threads, which is impossible with operating system threads. Synchronization also comes almost for free, without requiring any synchronization primitive like locks. The functional paradigm makes the implementation even easier, by using *continuations* [9].

The downside of cooperative scheduling is that it is necessary to make sure that threads actually cooperate:

- The programmer has to return control to the scheduler regularly. This is particularly true for infinite loops, that are very often present in *reactive* and *interactive* systems [11].
- She cannot call blocking functions like operating system primitives for I/O.

The solution to the latter is pretty simple: never use blocking functions inside cooperative threads. All the facilities mentioned earlier provide either I/O libraries compatible with cooperative scheduling or means to safely call blocking functions. See [18] for an overview on how to implement blocking operations safely in this context.

The goal of this paper is to design a static analysis, called *reactivity analysis*, to remedy the first problem. The analysis checks that the programmer does not forget to cooperate with the scheduler. Our work is applied on the ReactiveML language [17], which is an extension of ML with a synchronous model of concurrency [3]. Section 2 informally introduces the language and its semantics. The idea of synchronous languages is to divide the execution of a program into discrete logical instants, where computations and communications are considered instantaneous. This results in a deterministic model of concurrency that is compatible with the dynamic creation of processes [8]. Synchrony gives us a simple definition for reactivity: a reactive ReactiveML program is one where logical instants progress. It also gives us a simple condition for reactivity: a program is reactive if all processes cooperate at each instant.

The contributions of this paper are the following:

- A reactivity analysis presented as a type-and-effect system [16] in Section 4. The computed effects are called *behaviors* [2] and are introduced in Section 3. They represent the reactive behaviors of processes, by abstracting away values but keeping some part of the structure of the program. Exposing concurrency in the language makes it possible to express the analysis easily, which would not have been the case if concurrency had been implemented as a library. We believe this approach is generic enough to be applied to other models of concurrency (Section 6.6).
- A novel approach to *subeffecting* [19], that is, subtyping on effects, based on row polymorphism [21] in Section 4.4. It allows a simple integration of the analysis into any existing ML type inference implementation.
- A proof of the soundness of the analysis (Section 5) with respect to the big-step semantics of ReactiveML (Section 5.2): *a well-typed program is reactive*.

The paper ends with some discussion and examples (Section 6) and related work (Section 7). The work presented here is implemented in the ReactiveML compiler.[1]

[1] It is available at `http://reactiveml.org/icfp13`

2. Problem statement

Let us first introduce ReactiveML syntax and informal semantics using a simple program that highlights the problem of non-reactivity.[2] Then we will discuss the design choices and limitations of our reactivity analysis using a few other examples.

2.1 A first example

We start by creating a *process* that emits a signal every `timer` seconds:

```
1  let process clock timer s =
2    let time = ref (Unix.gettimeofday ()) in
3    loop
4      let time' = Unix.gettimeofday () in
5      if time' -. !time >= timer
6      then (emit s (); time := time')
7    end
```

In ReactiveML, there is a distinction between regular ML functions and *processes*, that is, functions whose execution can span several logical instants. They are defined using the **process** keyword. The `clock` process is parametrized by a float `timer` and a signal `s`. Signals are communication channels between processes, with instantaneous broadcast. The process starts by initializing a local reference `time` with the current time (line 2), read using the `gettimeofday` function of the `Unix` module from the standard library. Then it enters an infinite loop (line 3 to 7). At each iteration, it reads the new current time and emits the unit value on the signal `s` if enough time has elapsed (line 6). The compiler prints the following warning when compiling this process:

```
Line 3, characters 2-120:
W: This expression may be an instantaneous loop
```

The problem is that the body of the loop is instantaneous. It means that this process never cooperates, so that logical instants do not progress. In ReactiveML, cooperating is done by waiting for the next instant using the **pause** operator. We solve our problem by calling it at the end of the loop (line 7):

```
5  [...]
6      then (emit s (); time := time');
7      pause
8    end
```

The second part of the program is a process that prints `'top'` every time a signal `s` is emitted. The **do/when** construct executes its body only when the signal `s` is present (i.e. it is emitted). It terminates by returning the value of its body instantaneously after the termination of the body. Processes have a consistent view of a signal's status during an instant. It is either present or absent and its status cannot change during the instant.

```
10 let process print_clock s =
11   loop
12     do
13       print_string "top"; print_newline ()
14     when s done
15   end
```

```
Line 11, characters 2-78:
W: This expression may be an instantaneous loop
```

Once again, this loop can be instantaneous, but this time it depends on the presence of the signal. While the signal `s` is absent, the process cooperates. When it is present, the body of the **do/when** executes and terminates instantaneously. So the body of the loop also terminates instantaneously, and a new iteration of the loop is started in the same logical instant. Since the signal is still present, the body of the **do/when** executes one more time, and so on. This process can also be fixed by a call to **pause**.

We can then put these two processes in parallel, after declaring a local signal `s`. The result is a program that prints `'top'` every second:

```
17 let process main =
18   signal s default () gather (fun () () -> ()) in
19   run (print_clock s) || run (clock 1. s)
```

The declaration of a signal takes as arguments the default value of the signal and a combination function that is used to compute the value of the signal in case of multiple emissions during the same instant. Here, the default value is `()` and the signal keeps this value in case of multi-emission. The `||` operator represents the synchronous parallel composition. Both branches are executed at each instant and communicate through the local signal `s`.

2.2 Intuitions and limitations

In the previous example, we have seen the first cause of non-reactivity, that is, instantaneous loops. The second one is instantaneous recursive processes, as in this example:

```
let rec process instantaneous s =
  emit s ();
  run (instantaneous s)
W: This expression may produce an instantaneous recursion
```

A sufficient condition to ensure that a recursive process is reactive is to have *at least one instant between the instantiation of the process and any recursive call*. The idea of our analysis is to statically check this condition.

This condition is very strong and is not verified by interesting programs that are reactive. For instance, it does not hold for a parallel `map` (the **let/and** construct executes its two branches in parallel):

```
let rec process par_map p l =
  match l with
  | [] -> []
  | x :: l -> let x' = run (p x)
              and l' = run (par_map p l) in
              x' :: l'
W: This expression may produce an instantaneous recursion
```

This process does instantaneous recursive calls, but it is reactive because the recursion is finite (if the list `l` is finite). As we don't want to prove the termination of such processes, our analysis only prints warnings and does not reject programs. As ML functions are always instantaneous, they are reactive if and only if they terminate. We thus restrict our analysis to processes and suppose that functions always terminate, in order to avoid showing a warning for each recursive function.

Furthermore, we do not deal with blocking functions, like I/O primitives, that can also make the program non-reactive. Indeed, such functions should *never* be used in the context of cooperative scheduling. A solution to this problem could be to implement a cooperative IO library following the ideas in [18].

The analysis does not either take into account the presence of signals. It over-approximates the possible behaviors, as in the following example:

```
let rec process imprecise =
  signal s default () gather (fun () () -> ()) in
  present s then () else (* implicit pause *) ();
  run imprecise
W: This expression may produce an instantaneous recursion
```

[2] This example is taken from an email sent by a ReactiveML programmer asking for help. It motivated the reflexion that led to this work.

The `present/then/else` construct executes instantaneously its first branch if the signal is present or executes the second branch with a delay of one instant if the signal is absent. This delayed reaction to absence, first introduced in [8], avoids inconsistencies in the status of signals. In the example, the signal is absent so the `else` branch is executed. It means that the recursion is not instantaneous and the process is reactive. Our analysis still prints a warning, because if the signal `s` could be present, the recursion would be instantaneous.

Finally, we only guarantee that a program will react, not that is is real-time, that is, that it reacts in bounded time, as shown by this example:

```
let rec process server add =
  await add(p, ack) in
  run (server add) || let v = run p in emit ack v
```

The `server` process receives on a signal `add` both a process `p` and a signal `ack` on which to send back the result. As it creates one new process each time the `add` signal is emitted, this program can execute an arbitrary number of processes at the same time. It is thus not real-time, but it is indeed reactive, as waiting for the value of a signal takes one instant (one has to collect and combine all the values emitted during the instant).

3. The algebra of behaviors

The idea of our analysis is to abstract processes into a simpler language called *behaviors*, following the work of [2]. Behaviors abstract the reactive behavior of processes. The main design choice is to completely abstract values and the presence of signals. It is however necessary to keep part of the structure of the program (or an abstraction of it) in order to have a precise analysis.

3.1 The behaviors

The algebra of behaviors is given by: [3]

$$\kappa ::= \bullet \mid 0 \mid \phi \mid \kappa \,||\, \kappa \mid \kappa + \kappa \mid \kappa;\kappa \mid \mu\phi.\,\kappa \mid \texttt{run}\ \kappa$$

Surely non-instantaneous actions that take at least one instant to execute, such as `pause`, are denoted $\bullet$. Potentially instantaneous ones, like calling a pure ML function or emitting a signal, are denoted 0. The language also includes behavior variables ϕ to represent the behaviors of processes taken as arguments.

Behaviors must reflect the structure of the program, starting with parallel composition. This is illustrated by the following example, which defines a combinator `par_comb` that takes as inputs two processes `q1` and `q2` and runs them in parallel in a loop:

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

The synchronous parallel composition terminates when both branches have terminated. It means that the loop is non-instantaneous if either `q1` or `q2` is non-instantaneous. That is why behaviors include the parallel composition, simply denoted $||$. Similarly, we can define another combinator that runs one of its two inputs depending on a condition `c`:

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

In the case of `if_comb`, both processes must be non-instantaneous. To represent this process, we use a non-deterministic choice operator denoted $+$. This shows how we abstract values: we only keep the different alternatives and forget about the conditions.

[3] The order of precedence of operators is the following (from highest to lowest): `run`, $;$, $+$, $||$ and finally μ. For instance:
$\mu\phi.\,\kappa_1 \,||\, \texttt{run}\ \kappa_2 + \bullet;\kappa_3$ means $\mu\phi.\,(\kappa_1 \,||\, (\texttt{run}\ (\kappa_2) + (\bullet;\kappa_3)))$

It is also necessary to have a notion of sequence, denoted $;$ in the language of behaviors, as illustrated by the two following processes:

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

The order between the recursive call and the call to `pause` is crucial as the `good_rec` process is reactive while `bad_rec` loops instantaneously. As it is defined recursively, the behavior κ associated to the `good_rec` process must verify that $\kappa = \bullet;\texttt{run}\ \kappa$. The `run` operator is associated to running a process and is necessary to solve technical problems in the type system, that will be discussed in Section 6.2. This equation can be solved by introducing an explicit recursion operator μ so that $\kappa = \mu\phi.\,\bullet;\texttt{run}\ \phi$. Recursive behaviors verify the usual properties:

$$\mu\phi.\,\kappa = \kappa[\phi \leftarrow \mu\phi.\,\kappa] \qquad \mu\phi.\,\kappa = \kappa \text{ if } \phi \notin \mathit{fbv}(\kappa)$$

We denote $\mathit{fbv}(\kappa)$ the set of free behavior variables in κ. It should be noted that there is no operator for representing the behavior of a loop. Indeed, a loop is just a special case of recursion. The behavior of a loop, denoted κ^∞ (where κ is the behavior of the body of the loop), is thus defined as a recursive behavior by:

$$\kappa^\infty \triangleq \mu\phi.\,\kappa;\texttt{run}\ \phi$$

3.2 Reactive behaviors

Using the language of behaviors, we can now characterize the behaviors that we want to reject, that is instantaneous loops and recursions. We actually enforce a stronger sufficient condition, that can be checked easily and efficiently: there must be at least one instant before each recursive call. This condition is not necessary, as the `par_map` example of Section 2.2 showed. One can also check from the definition of κ^∞ as a recursive behavior that this condition also implies that the body of a loop is non-instantaneous.

To formally define what is a reactive behavior, we first have to define the notion of *non-instantaneous* behavior, that is associated to a processes that takes at least one instant to execute:

Definition 1 (Non-instantaneous behavior)**.** A behavior is *non-instantaneous*, denoted $\kappa \downarrow^\bullet$, if:

$$\frac{}{\bullet \downarrow^\bullet} \qquad \frac{}{\phi \downarrow^\bullet} \qquad \frac{\kappa_1 \downarrow^\bullet}{\kappa_1;\kappa_2 \downarrow^\bullet} \qquad \frac{\kappa_2 \downarrow^\bullet}{\kappa_1;\kappa_2 \downarrow^\bullet} \qquad \frac{\kappa_1 \downarrow^\bullet}{\kappa_1 \,||\, \kappa_2 \downarrow^\bullet}$$

$$\frac{\kappa_2 \downarrow^\bullet}{\kappa_1 \,||\, \kappa_2 \downarrow^\bullet} \qquad \frac{\kappa_1 \downarrow^\bullet \quad \kappa_2 \downarrow^\bullet}{\kappa_1 + \kappa_2 \downarrow^\bullet} \qquad \frac{\kappa \downarrow^\bullet}{\mu\phi.\,\kappa \downarrow^\bullet} \qquad \frac{\kappa \downarrow^\bullet}{\texttt{run}\ \kappa \downarrow^\bullet}$$

The fact that variables are considered non-instantaneous means that any process taken as argument is supposed to be non-instantaneous. If this is not the case, then the verification of reactivity is done when this variable is instantiated with the actual behavior of the process.

A behavior is said to be *reactive* if for each recursive behavior $\mu\phi.\,\kappa$, the recursion variable ϕ does not appear in the first instant of the body κ.

Definition 2 (Reactive behavior)**.** A behavior κ is *reactive* if $\emptyset \vdash \kappa$, where the relation $R \vdash \kappa$ is defined by:

$$\frac{}{R \vdash 0} \qquad \frac{}{R \vdash \bullet} \qquad \frac{\phi \notin R}{R \vdash \phi} \qquad \frac{R \vdash \kappa_1 \quad \kappa_1 \downarrow^\bullet \quad \emptyset \vdash \kappa_2}{R \vdash \kappa_1;\kappa_2}$$

$$\frac{R \vdash \kappa_1 \quad \mathit{not}(\kappa_1 \downarrow^\bullet) \quad R \vdash \kappa_2}{R \vdash \kappa_1;\kappa_2} \qquad \frac{R \vdash \kappa_1 \quad R \vdash \kappa_2}{R \vdash \kappa_1 \,||\, \kappa_2}$$

$$\frac{R \vdash \kappa_1 \quad R \vdash \kappa_2}{R \vdash \kappa_1 + \kappa_2} \qquad \frac{R \cup \{\phi\} \vdash \kappa}{R \vdash \mu\phi.\,\kappa} \qquad \frac{R \vdash \kappa}{R \vdash \texttt{run}\ \kappa}$$

$$e_1 \,||\, e_2 \triangleq \texttt{let \_} = e_1 \texttt{ and \_} = e_2 \texttt{ in } ()$$
$$\texttt{let } x = e_1 \texttt{ in } e_2 \triangleq \texttt{let } x = e_1 \texttt{ and \_} = () \texttt{ in } e_2$$
$$\texttt{let } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = \lambda x_1 \ldots . \lambda x_p . e_1 \texttt{ in } e_2$$
$$\texttt{let rec } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = (\texttt{rec } f = \lambda x_1 \ldots . \lambda x_p . e_1) \texttt{ in } e_2$$
$$\texttt{let process } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = \lambda x_1 \ldots . \lambda x_p . \texttt{process } e_1 \texttt{ in } e_2$$
$$\texttt{let rec process } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = (\texttt{rec } f = \lambda x_1 \ldots . \lambda x_p . \texttt{process } e_1) \texttt{ in } e_2$$
$$e_1 ; e_2 \triangleq \texttt{let \_} = e_1 \texttt{ in } e_2$$
$$\texttt{await } e_1(x) \texttt{ in } e_2 \triangleq \texttt{do } (\texttt{loop pause}) \texttt{ until } e_1(x) \to e_2$$

Figure 1. Derived language constructs

The predicate $R \vdash \kappa$ means that the behavior κ is reactive with respect to the set of variables R, that is, these variables do not appear in the first instant of κ and all the recursions inside κ are not instantaneous. In the case of the sequence $\kappa_1; \kappa_2$, if the behavior κ_1 is non-instantaneous, then it is not necessary to check if the variables in R appear free in κ_2. However, we still have to check that κ_2 is reactive, that is, that $\emptyset \vdash \kappa_2$.

3.3 Equivalence on behaviors

We can define an equivalence relation $\equiv$ on behaviors. An important property of this relation is that it preserves reactivity, which is expressed by the following property:

Property 1. *if* $\kappa_1 \equiv \kappa_2$ *and* $R \vdash \kappa_1$ *then* $R \vdash \kappa_2$

The $\equiv$ relation is an equivalence relation, i.e. it is reflexive, symmetric and transitive. The operators ; and $||$ and $+$ are compatible with this relation, idempotent and associative. $||$ and $+$ are commutative (but not ;). The 0 behavior (resp. $\bullet$) is the neutral element of ; and $||$ (resp. $+$). The equivalence relation also verifies the following properties:

$$\frac{\kappa_1 \equiv \kappa_2}{\mu\phi.\, \kappa_1 \equiv \mu\phi.\, \kappa_2} \qquad \frac{\kappa_1 \equiv \kappa_2}{\texttt{run } \kappa_1 \equiv \texttt{run } \kappa_2} \qquad \bullet^\infty \equiv \bullet$$

We can define the $\equiv$ relation as the smallest relation that verifies all these properties. For instance, it is easy to show that:

$$\mu\phi.\, ((\bullet \,||\, 0); (\texttt{run } \phi + \texttt{run } \phi)) \equiv \mu\phi.\, \bullet; \texttt{run } \phi$$

4. The type-and-effect system

The link between processes and behaviors is done by a type-and-effect system [16], following the work of Amtoft et al. [2]. The behavior of a process is its effect computed using the type system. A type system is a simple and efficient way to implement a higher-order static analysis.

4.1 Abstract syntax

We consider here a kernel of ReactiveML:

$$\begin{aligned}
v &::= c \mid (v, v) \mid n \mid \lambda x.e \mid \texttt{process } e \\
e &::= x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec } x = e \mid \texttt{process } e \mid \texttt{run } e \\
&\quad \mid \texttt{pause} \mid \texttt{let } x = e \texttt{ and } x = e \texttt{ in } e \\
&\quad \mid \texttt{signal } x \texttt{ default } e \texttt{ gather } e \texttt{ in } e \\
&\quad \mid \texttt{emit } e\ e \mid \texttt{present } e \texttt{ then } e \texttt{ else } e \\
&\quad \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \\
&\quad \mid \texttt{loop } e \mid \texttt{do } e \texttt{ until } e(x) \to e \mid \texttt{do } e \texttt{ when } e
\end{aligned}$$

Values are constants (integers, booleans, etc.), pairs of values, signal names n, functions and processes. The language is a call-by-value lambda-calculus, extended with constructs for creating (`process`) and running (`run`) processes, waiting for the next instant (`pause`), parallel definitions (`let`/`and`), declaring signals (`signal`), emitting a signal (`emit`) and several control structures: the test of presence of a signal (`present`), the unconditional loop (`loop`), weak preemption (`do`/`until`) and suspension (`do`/`when`). The expression `do` e_1 `until` $s(x) \to e_2$ executes its body e_1 and, when the signal s is present, stops the execution of e_1 and then executes the continuation e_2 on the next instant, binding x to the value of s. We denote _ variables that do not appear free in the body of a `let` and $()$ the unique value of type `unit`. From this kernel, we can encode most constructs of the language, as shown in Figure 1.

4.2 Types

Types and behaviors are defined by:

$$\begin{aligned}
\tau &::= \quad \alpha \mid T \mid \tau \times \tau \mid \tau \to \tau \\
&\qquad \mid \tau\, \texttt{process}[\kappa] \mid (\tau, \tau)\, \texttt{event} && \text{(types)} \\
\sigma &::= \tau \mid \forall \phi.\, \sigma \mid \forall \alpha.\, \sigma && \text{(type schemes)} \\
\Gamma &::= \emptyset \mid \Gamma, x : \sigma && \text{(environments)}
\end{aligned}$$

A type is either a type variable α, a base type T (like `bool` or `unit`), a product, a function, a process or a signal. The type of a process is parametrized by its return type and its behavior. The type $(\tau_1, \tau_2)\, \texttt{event}$ of a signal is parametrized by the type τ_1 of emitted values and the type τ_2 of the read value.

Types schemes quantify universally over type variables α and behavior variables ϕ. We denote $ftv(\tau)$ (resp. $fbv(\tau)$) the set of type (resp. behavior) variables free in τ and:

$$fv(\tau) = ftv(\tau), fbv(\tau)$$

Instantiation and generalization are defined in a classic way:

$$\sigma[\alpha \leftarrow \tau] \leq \forall \alpha.\, \sigma \qquad \sigma[\phi \leftarrow \kappa] \leq \forall \phi.\, \sigma$$

$$\begin{aligned}
&gen(\tau, e, \Gamma) = \tau && \text{if } e \text{ is expansive} \\
&gen(\tau, e, \Gamma) = \forall \bar{\alpha}. \forall \bar{\phi}.\, \tau && \text{otherwise} \\
&\qquad \text{where } \bar{\alpha}, \bar{\phi} = fv(\tau) \setminus fv(\Gamma)
\end{aligned}$$

As for references in ML, we have to be careful to not generalize expressions that allocate signals. We use the syntactic criterion of expansive and non-expansive expressions [25].

The notions of reactivity and equivalence are lifted from behaviors to types. A type is reactive if it contains only reactive behaviors. Two types are equivalent, also denoted $\tau_1 \equiv \tau_2$, if they have the same structure and their behaviors are equivalent.

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau \mid 0} \qquad \frac{\tau \leq \Gamma_0(c)}{\Gamma \vdash c : \tau \mid 0} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \mid \_ \qquad \Gamma \vdash e_2 : \tau_2 \mid \_}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mid 0} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \_}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \mid 0}$$

$$\text{(App)}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \mid \_ \qquad \Gamma \vdash e_2 : \tau_2 \mid \_}{\Gamma \vdash e_1\ e_2 : \tau_1 \mid 0} \qquad \frac{\Gamma, x : \tau \vdash e : \tau \mid \_}{\Gamma \vdash \texttt{rec}\ x = e : \tau \mid 0} \qquad \text{(Process)}\ \frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\ e : \tau\, \texttt{process}[\kappa + \kappa'] \mid 0}$$

$$\frac{\Gamma \vdash e : \tau\, \texttt{process}[\kappa] \mid \_}{\Gamma \vdash \texttt{run}\ e : \tau \mid \texttt{run}\ \kappa} \qquad \Gamma \vdash \texttt{pause} : \texttt{unit} \mid \bullet \qquad \frac{\Gamma \vdash e : \texttt{bool} \mid \_ \qquad \Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa_1 + \kappa_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid \kappa_2 \qquad \Gamma, x_1 : gen(\tau_1, e_1, \Gamma), x_2 : gen(\tau_2, e_2, \Gamma) \vdash e_3 : \tau \mid \kappa_3}{\Gamma \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 : \tau \mid (\kappa_1 \mid\mid \kappa_2); \kappa_3}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \mid \_ \qquad \Gamma \vdash e_2 : \tau_1 \to \tau_2 \to \tau_2 \mid \_ \qquad \Gamma, x : (\tau_1, \tau_2)\, \texttt{event} \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \tau \mid 0; \kappa} \qquad \frac{\Gamma \vdash e : (\tau_1, \tau_2)\, \texttt{event} \mid \_ \qquad \Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa_1 + (\bullet; \kappa_2)}$$

$$\frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{loop}\ e : \texttt{unit} \mid (0; \kappa)^\infty} \qquad \frac{\Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e : (\tau_1, \tau_2)\, \texttt{event} \mid \_ \qquad \Gamma, x : \tau_2 \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x) \to e_2 : \tau \mid \kappa_1 + (\bullet; \kappa_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau \mid \kappa \qquad \Gamma \vdash e : (\tau_1, \tau_2)\, \texttt{event} \mid \_}{\Gamma \vdash \texttt{do}\ e_1\ \texttt{when}\ e : \tau \mid \kappa + \bullet^\infty} \qquad \text{(Mask)}\ \frac{\Gamma \vdash e : \tau \mid \kappa \qquad \phi \notin fbv(\Gamma, \tau)}{\Gamma \vdash e : \tau \mid \kappa[\phi \leftarrow \bullet]}$$

Figure 2. Type-and-effect rules

4.3 Typing rules

Typing judgments are given by

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

meaning that, in the type environment Γ, the expression e has type τ and behavior κ. We write $\Gamma \vdash e : \tau \mid \_$ when the behavior of the expression e is not constrained and does not appear in the conclusion of the inference rule. The initial typing environment Γ_0 gives the types of the different primitives:

$$\Gamma_0 \triangleq [\texttt{emit} \ : \forall \alpha_1, \alpha_2.\, (\alpha_1, \alpha_2)\, \texttt{event} \to \alpha_1 \to \texttt{unit};$$
$$\texttt{true} : \texttt{bool}; \texttt{fst} : \forall \alpha_1, \alpha_2.\, \alpha_1 \times \alpha_2 \to \alpha_1; \ldots]$$

The rules defining the type system are given in Figure 2. If all the behaviors are erased, it is exactly the same type system as the one presented in [17], which is itself an extension of ML type system. We discuss here the novelties of these rules related to behaviors. The rules Process and Mask are related to subeffecting and will be discussed in Section 4.4.

- A design choice made in ReactiveML is to separate pure ML expressions, that are surely instantaneous, from processes. For instance, it is impossible to call `pause` within the body of a function, that must be instantaneous. A static analysis done before typing checks this well-formation of expressions, denoted $k \vdash e$ in [17] and reminded in Appendix A. That is why our type system ignores the behavior of expressions that must be ML expressions, like the body of a function. We could prove that their behavior is always equivalent to the instantaneous behavior 0.
- We do not try to prove the termination of pure ML functions without any reactive behavior. The App rule shows that we suppose that function calls always terminate instantaneously. That is why there is no behavior associated to functions, that is, there is no behaviors on arrows unlike traditional type-and-effect systems.
- As explained earlier, in the case of `present` e `then` e_1 `else` e_2, the first branch e_1 is executed immediately if the signal e is present and the second branch e_2 is executed at the next instant if it is absent. This is reflected by the behavior associated to the expression. Similarly, for `do` e_1 `until` $e(x) \to e_2$, e_2 is executed at the instant following the presence of e.
- We can also check that the encoding of primitives given in Figure 1 yields the expected behaviors. This is for instance the case of $e_1; e_2$:

$$\frac{\dfrac{\Gamma \vdash e_1 : \tau_1 \mid \kappa_1 \qquad \Gamma \vdash () : \texttt{unit} \mid 0 \qquad \Gamma \vdash e_2 : \tau_2 \mid \kappa_2}{\Gamma \vdash \texttt{let}\ \_ = e_1\ \texttt{and}\ \_ = ()\ \texttt{in}\ e_2 : \tau_2 \mid (\kappa_1 \mid\mid 0); \kappa_2}}{\Gamma \vdash e_1; e_2 : \tau_2 \mid (\kappa_1 \mid\mid 0); \kappa_2}$$

  It is easy to check that $(\kappa_1 \mid\mid 0); \kappa_2 \equiv \kappa_1; \kappa_2$. We can also check that $e_1 \,||\, e_2$ has a behavior equivalent to $\kappa_1 \mid\mid \kappa_2$ or that `await` $e_1(x)$ `in` e_2 has a behavior $\bullet^\infty + (\bullet; \kappa_2) \equiv \bullet; \kappa_2$.
- In [17], the `loop` construct is encoded as a recursive process by:

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

  By applying the rules here, we have that:

  $$\mathit{loop} : \forall \phi.\, \alpha\, \texttt{process}[\phi] \to \alpha'\, \texttt{process}[\mu\phi'.\, \texttt{run}\ \phi; \texttt{run}\ \phi']$$

  If we suppose that $\Gamma \vdash e : \tau \mid \kappa$, then the behavior of this encoding is: $\texttt{run}\ (\mu\phi'.\, \texttt{run}\ \kappa; \texttt{run}\ \phi')$. It is not equivalent to κ^∞ in the sense of Section 3.3, but it is reactive iff κ^∞ is reactive, as the `run` operator does not influence reactivity (see Definition 2). It means that we could have removed `loop` from our kernel without any influence on the result of the reactivity analysis. The reason why the behavior associated with a loop is not equal to κ^∞ as expected will be explained during the proof of soundness in Section 5.
- As for `loop`, the `pause` operator can also be encoded by:

  $\texttt{pause} \triangleq \texttt{signal}\ s\ \texttt{default}\ ()\ \texttt{gather}\ (\lambda x.\lambda y.())\ \texttt{in}$
  $\texttt{present}\ s\ \texttt{then}\ ()\ \texttt{else}\ ()$

We have chosen to completely abstract values. As in the `imprecise` example of Section 2.2, we do not consider the fact the signal s is always absent, so that only the second branch of the `present` is executed. The consequence is that the behavior computed by the type system, that is, $0 + \bullet; 0 \equiv 0$, is the opposite of the expected behavior of `pause`.

4.4 Subeffecting with row polymorphism

The typing rule for the creation of processes intuitively mean that a process has *at least* the behavior of its body. This subtyping restricted to effects is often referred to as *subeffecting* [19]: we can always replace an effect with a bigger, i.e. less precise, one. It allows the type system to be a conservative extension of ReactiveML type system, that is, we are able to give a behavior to any correct ReactiveML program. For instance, we can give a type to the following expression:

```
let l = [process (); process (pause)]
val l : unit process[0 + * + 'r]
```

If the behavior of a process had been equal to the behavior of its body, this expression would have been rejected by the type system.

Subeffecting

Subeffecting [19, 24] is usually expressed as a non-syntax directed rule (we reuse the notations of our type system for comparison):

$$\frac{\Gamma \vdash e : \tau \mid \kappa \qquad \kappa \sqsubseteq \kappa'}{\Gamma \vdash e : \tau \mid \kappa'}$$

The order $\sqsubseteq$ on effects is given by set inclusion when effects are sets [24] (of regions for example). In our case, It is defined by:

$$\kappa_1 \sqsubseteq \kappa_1 + \kappa_2 \qquad \kappa_2 \sqsubseteq \kappa_1 + \kappa_2 \qquad \frac{\kappa_1 \equiv \kappa_2}{\kappa_1 \sqsubseteq \kappa_2}$$

A similar approach is used in [2]. It enforces effects to be *simple*, that is, effects on arrows are syntactically forced to be variables. A constraint set C is added to the type system to keep track of the relations between variables and effects. Subeffecting is then expressed as:

$$\frac{\Gamma, C \vdash e : \tau \mid \kappa}{\Gamma, C \cup \{\kappa \sqsubseteq \phi\} \vdash \texttt{process}\ e : \tau\ \texttt{process}[\phi] \mid 0}$$

These two formulations and our version appear to be equivalent. Indeed, our system and the one of [2] are just syntax-directed versions of the first one, where the subtyping relation is applied only for lambda abstractions (or processes in our case).

Implementing subeffecting with row polymorphism

The consequence of the typing rule for processes is that the principal type of an expression `process` e is always of the shape $\kappa + \phi$. The idea to use a free type variable to represent other possible types is reminiscent of Remy's row types [21]. It makes it possible to implement subeffecting using only unification, without manipulating constraint sets as in traditional approaches [2, 24]. It is thus easier to integrate it into any existing ML type inference implementation. For instance, OCaml type inference is also based on row polymorphism, so it would be easy to implement our analysis on top of the full language.

During unification, the behavior of a process is always either a behavior variable ϕ, a row $\kappa + \phi$ or a recursive row $\mu\phi.\,(\kappa + \phi')$. We could have made this fact more visible by having two different kinds of behaviors: behaviors and rows of behaviors. We chose here to stick with a more simple syntax. We can reuse any existing inference algorithm, like algorithm $\mathcal{W}$ or $\mathcal{M}$ [14] and only use the following algorithm $\mathcal{U}_\kappa$ for unification of behaviors. It takes as input two behaviors and returns a substitution that maps behavior variables to behaviors, that we denote $[\phi_1 \mapsto \kappa_1; \phi_2 \mapsto \kappa_2; \dots]$. It is defined as follows:

$$\begin{aligned}
\mathcal{U}_\kappa(\kappa, \kappa) &= [] \\
\mathcal{U}_\kappa(\phi, \kappa) = \mathcal{U}_\kappa(\kappa, \phi) &= [\phi \mapsto \mu\phi.\,\kappa] \text{ if } \texttt{occur_check}(\phi, \kappa) \\
&= [\phi \mapsto \kappa] \\
\mathcal{U}_\kappa(\kappa_1 + \phi_1, \kappa_2 + \phi_2) &= [\phi_1 \mapsto \kappa_2 + \phi; \phi_1 \mapsto \kappa_1 + \phi],\ \phi \text{ fresh} \\
\mathcal{U}_\kappa(\mu\phi_1'.\,\kappa_1 + \phi_1, \kappa_2) &= \mathcal{U}_\kappa(\kappa_2, \mu\phi_1'.\,\kappa_1 + \phi_1) \\
&= \texttt{let}\ K_1 = \mu\phi_1'.\,\kappa_1 + \phi_1 \\
&\quad \texttt{in}\ \mathcal{U}_\kappa(\kappa_1[\phi_1' \leftarrow K_1] + \phi_1, \kappa_2)
\end{aligned}$$

It should be noted that unification never fails, so that we obtain a conservative extension of ReactiveML type system. This unification algorithm also reuse traditional techniques for handling recursive types [12]. The last case unfolds a recursive row to reveal the row variable, so that it can be unified with other rows.

A downside of our approach is that it introduces one behavior variable for each process, so that the computed behaviors may get very big and unreadable. The purpose of the MASK rule is to remedy this, by using *effect masking* [16]. The idea is that if a behavior variable appearing in the behavior is free in the environment, it is not constrained so we can give it any value. In particular, we choose to replace it with $\bullet$, which is the neutral element of $+$, so that it can be simplified away.

5. Proof of soundness

We will now prove the soundness of our analysis, that is, that a well-typed program is reactive. The intuition of the proof is that the first instant of a reactive behavior (as defined in Section 3.2) is a finite behavior, without any recursion. We then prove by induction on the size of behaviors that a well-typed program admits a finite derivation in the big-step semantics.

5.1 First instant of a behavior

Definition 3. The *first-instant* of a behavior, denoted $\texttt{fst}(\kappa)$ is the part of the behavior that corresponds to the execution of the first instant of the corresponding process. It is formally defined by:

$$\begin{aligned}
\texttt{fst}(0) &= \texttt{fst}(\bullet) = 0 \\
\texttt{fst}(\phi) &= \phi \\
\texttt{fst}(\texttt{run}\ \kappa) &= \texttt{run}\ (\texttt{fst}(\kappa)) \\
\texttt{fst}(\kappa_1 \mid\mid \kappa_2) &= \texttt{fst}(\kappa_1) \mid\mid \texttt{fst}(\kappa_2) \\
\texttt{fst}(\kappa_1 + \kappa_2) &= \texttt{fst}(\kappa_1) + \texttt{fst}(\kappa_2) \\
\texttt{fst}(\kappa_1; \kappa_2) &= \begin{cases} \texttt{fst}(\kappa_1) & \text{if } \kappa_1 \downarrow^\bullet \\ \texttt{fst}(\kappa_1); \texttt{fst}(\kappa_2) & \text{otherwise} \end{cases} \\
\texttt{fst}(\mu\phi.\,\kappa) &= \texttt{fst}(\kappa[\phi \leftarrow \mu\phi.\,\kappa])
\end{aligned}$$

In the case of a recursive behavior, the first-instant behavior is well-defined only if the behavior is reactive, that is, the recursion is not instantaneous.

Definition 4. A behavior is *finite*, denoted $\kappa \in \kappa^\star$, if it does not contain any true recursive behavior. The finite behaviors $\kappa^\star$ are defined by:

$$\kappa^\star ::= \bullet \mid 0 \mid \phi \mid \kappa^\star \mid\mid \kappa^\star \mid \kappa^\star + \kappa^\star \mid \kappa^\star; \kappa^\star \mid \texttt{run}\ \kappa^\star$$

We can now express the important property of reactive behaviors, that is, the first instant of a reactive behavior is finite.

Property 2. *If κ is reactive, then $\texttt{fst}(\kappa)$ is a finite behavior, i.e.:*

$$\emptyset \vdash \kappa \ \Rightarrow\ \texttt{fst}(\kappa) \in \kappa^\star$$

Proof. By induction on the structure of behaviors. □

$$\frac{e_1 \xrightarrow[S]{E_1,tt} \lambda x.\ e \qquad e_2 \xrightarrow[S]{E_2,tt} v_2 \qquad e[x \leftarrow v_2] \xrightarrow[S]{E_3,tt} v}{e_1\ e_2 \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E_3,tt} v} \qquad \frac{e[x \leftarrow \texttt{rec}\ x = e] \xrightarrow[S]{E,tt} v}{\texttt{rec}\ x = e \xrightarrow[S]{E,tt} v} \qquad \frac{e \xrightarrow[S]{E,tt} \texttt{process}\ e_1 \qquad e_1 \xrightarrow[S]{E_1,b} e_1'}{\texttt{run}\ e \xrightarrow[S]{E \sqcup E_1,b} e_1'}$$

$$\text{(LET-PAR)}\ \frac{e_1 \xrightarrow[S]{E_1,b_1} e_1' \qquad e_2 \xrightarrow[S]{E_2,b_2} e_2' \qquad b_1 \wedge b_2 = \mathit{ff}}{\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 \xrightarrow[S]{E_1 \sqcup E_2,\mathit{ff}} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e_3}$$

$$\text{(LET-DONE)}\ \frac{e_1 \xrightarrow[S]{E_1,tt} v_1 \qquad e_2 \xrightarrow[S]{E_2,tt} v_2 \qquad e_3[x_1 \leftarrow v_1; x_2 \leftarrow v_2] \xrightarrow[S]{E_3,b_3} e_3'}{\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3 \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E_3,b_3} e_3'}$$

$$\texttt{pause} \xrightarrow[S]{\emptyset,\mathit{ff}} () \qquad \frac{e \xrightarrow[S]{E,tt} n \qquad n \in S \qquad e_1 \xrightarrow[S]{E_1,b} e_1'}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_1,b} e_1'} \qquad \frac{e \xrightarrow[S]{E,tt} n \qquad n \notin S}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E,\mathit{ff}} e_2}$$

$$\frac{e \xrightarrow[S]{E,tt} n \qquad n \notin S}{\texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E,\mathit{ff}} \texttt{do}\ e_1\ \texttt{when}\ n} \qquad \frac{e \xrightarrow[S]{E,tt} n \qquad n \in S \qquad e_1 \xrightarrow[S]{E_1,\mathit{ff}} e_1'}{\texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E \sqcup E_1,\mathit{ff}} \texttt{do}\ e_1'\ \texttt{when}\ n} \qquad \frac{e \xrightarrow[S]{E,tt} n \qquad n \in S \qquad e_1 \xrightarrow[S]{E_1,tt} v}{\texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E \sqcup E_1,tt} v}$$

$$\text{(LOOP-STUCK)}\ \frac{e \xrightarrow[S]{E,\mathit{ff}} e'}{\texttt{loop}\ e \xrightarrow[S]{E,\mathit{ff}} e'; \texttt{loop}\ e} \qquad \text{(LOOP-UNROLL)}\ \frac{e \xrightarrow[S]{E_1,tt} v \qquad \texttt{loop}\ e \xrightarrow[S]{E_2,b} e'}{\texttt{loop}\ e \xrightarrow[S]{E_1 \sqcup E_2,b} e'}$$

Figure 3. Big-step semantics

5.2 Big-step Semantics

In this section, we give an overview of the big-step semantics of ReactiveML, also called the behavioral semantics in reference to the one of Esterel [4] from which it is inspired. The interested reader can refer to Appendix B or [17] for a more detailed and formal presentation of this semantics.

The reaction of an expression is defined by the smallest signal environment S_i such that:

$$e_i \xrightarrow[S_i]{E_i,b_i} e_{i+1}$$

which means that during the instant, in the signal environment S_i, the expression e_i rewrites to e_{i+1} and emits the signals in E_i. b_i is a boolean that indicates if e_{i+1} has terminated. Additional conditions express for instance the fact that the emitted values in E_i must agree with the signal environment S_i. The execution of a program is made of the succession of a (potentially infinite) number of reactions and terminates when the status b_i is equal to true. We write $n \in S$ when the signal n is present in the signal environment S (i.e. it has been emitted in the current instant) and $n \notin S$ otherwise.

Figure 3 shows part of the rules defining the relation. The remaining rules can be found in Appendix B. Let us briefly discuss some important points of the semantics:

- The rule for `pause` shows the meaning of the boolean b: if it is false, it means that the expression is stuck waiting for the next instant.
- The `let`/`and` construct executes its two branches until both are terminated.
- The `present` construct executes the `then` branch immediately if the signal is present, but it executes the `else` branch on the next instant if it is absent.
- The `do`/`when` construct executes its body only if the signal n is present. If the body terminates, that is, it rewrites to a value v, then the construct also terminates instantaneously and rewrites to the same value.
- The unconditional loop keeps executing its body until it awaits the next instant, that is, its termination status b becomes false. In particular, an expression like `loop ()`, where the body always terminates instantaneously, does not have a semantics as it would require an infinite derivation tree.

5.3 Soundness

As we said earlier, we do not try to prove that functions terminate and only care about processes. We suppose that all functions terminate, which is reflected in the APP rule of Figure 2 by the fact that the behavior of the application is always the instantaneous behavior 0. This hypothesis is made possible by the syntactic distinction between functions and processes. Before going further with the proof of soundness, we have to express this hypothesis more formally with respect to the big-step semantics. For that, we use the predicate $0 \vdash e$ defined in Appendix A and meaning that the expression e is surely instantaneous, that is, an ML expression

Hypothesis 3 (Function calls always terminate)**.** *For any expression e such that $0 \vdash e$, there exists a finite derivation Π and a value v such that:*

$$\frac{\Pi}{e \xrightarrow[S]{E,tt} v}$$

We first need to prove the soundness of our definition of non-instantaneous behavior, as expressed in the following lemma:

Lemma 4. *An expression whose behavior is not instantaneous never reduces instantaneously.*

$$\left(\Gamma \vdash e : \tau \mid \kappa \ \wedge \ \kappa \downarrow^{\bullet} \ \wedge \ e \xrightarrow[S]{E,b} e'\right) \Rightarrow b = \mathit{ff}$$

Proof. By induction on the derivation of the big-step semantics. □

We can now express the soundness of our analysis, that is, a well-typed program is reactive. Being reactive means here that there exists a derivation, necessarily finite, to rewrite the program into a well-typed program.

Theorem 5 (Soundness)**.** *If $\Gamma \vdash e : \tau \mid \kappa$ and τ and κ are reactive and we suppose that function calls terminate, then there exists e' such that $e \xrightarrow[S]{E,b} e'$ and $\Gamma \vdash e' : \tau \mid \kappa'$ with κ' reactive.*

Proof. The proof is done by induction on the size of the first-instant behavior of well-typed expressions. The main point is that the behaviors of sub-expressions is always smaller, but we have to be careful that we must consider only the first-instant behavior. To prove that the result is well-typed, we can use classic syntactic techniques for type soundness [20] on the small-step semantics described in [17]. The proof of equivalence of the two semantics is also given in the same paper.

- Case $e_1\ e_2$ and $\texttt{rec}\ x = e$: By Hypothesis 3.
- Case $\texttt{run}\ e$: We know that $0 \vdash e$ so there exists Π such that

$$\frac{\Pi}{e \xrightarrow[S]{E,tt} \texttt{process}\ e_1}$$

Then, as $\texttt{run}\ e$ is well-typed, we have that:

$$\frac{\Gamma \vdash e_1 : \tau \mid \kappa}{\dfrac{\Gamma \vdash \texttt{process}\ e_1 : \tau\, \texttt{process}[\kappa + \kappa'] \mid 0}{\Gamma \vdash \texttt{run}\ (\texttt{process}\ e_1) : \tau \mid \texttt{run}\ (\kappa + \kappa')}}$$

We can apply the induction hypothesis as the first-instant behavior of e_1, i.e. $\texttt{fst}(\kappa)$, is smaller than the first-instant behavior of $\texttt{run}\ e$. Indeed, we have:

$$\texttt{fst}(\texttt{run}\ (\kappa + \kappa')) = \texttt{run}\ (\texttt{fst}(\kappa)) + \texttt{run}\ (\texttt{fst}(\kappa'))$$

The induction hypothesis allows to conclude that:

$$\frac{\Pi_1}{e_1 \xrightarrow[S]{E_1,b} e_1'}$$

which enables us to build the complete derivation of $\texttt{run}\ e$:

$$\frac{\dfrac{\Pi}{e \xrightarrow[S]{E,tt} \texttt{process}\ e_1} \qquad \dfrac{\Pi_1}{e_1 \xrightarrow[S]{E_1,b} e_1'}}{\texttt{run}\ e \xrightarrow[S]{E \sqcup E_1,b} e_1'}$$

- Case $\texttt{pause}$: The derivation is already finite without any hypothesis.
- Case $\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Like in the first case, we have:

$$\frac{\Pi}{e \xrightarrow[S]{E,tt} n}$$

The typing rule is as follows:

$$\frac{\begin{array}{c}\Gamma \vdash e : (\tau_1, \tau_2)\, \texttt{event} \mid \_ \\ \Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2\end{array}}{\Gamma \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa_1 + (\bullet; \kappa_2)}$$

The are then two cases depending on the status of n:

- If $n \in S$: We can notice that

$$\texttt{fst}(\kappa_1 + (\bullet; \kappa_2)) = \texttt{fst}(\kappa_1) + 0$$

so we can apply the induction hypothesis on the first-instant behavior of e_1 and conclude.

- If $n \notin S$: The derivation is finite.

- Case $\texttt{do}\ e_1\ \texttt{when}\ e_2$: We can use the same reasoning. It is interesting to note that the behavior associated to the expression, i.e. $\kappa + \bullet^{\infty}$, is not equal to the behavior of the body, as one could expect, so that we can apply the induction hypothesis.
- Case $\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e$: When we compute the first instant of a sequence, there are two possible cases:
  - If $(\kappa_1 \mid\mid \kappa_2) \downarrow^{\bullet}$, then we have that

$$\texttt{fst}(\kappa) = \texttt{fst}(\kappa_1) \mid\mid \texttt{fst}(\kappa_2)$$

From $(\kappa_1 \mid\mid \kappa_2) \downarrow^{\bullet}$, we get that either $\kappa_1 \downarrow^{\bullet}$ which implies that $b_1 = \mathit{ff}$, or $\kappa_2 \downarrow^{\bullet}$ which implies that $b_2 = \mathit{ff}$ using Lemma 4. So we are sure that $b_1 \wedge b_2 = \mathit{ff}$. We can then apply the induction hypothesis on e_1 and e_2 using the rule LET-PAR.

  - Otherwise, we have that

$$\texttt{fst}(\kappa) = (\texttt{fst}(\kappa_1) \mid\mid \texttt{fst}(\kappa_2)); \texttt{fst}(\kappa_3)$$

We can apply the induction hypothesis on e_1, e_2 and e_3 using either LET-PAR or LET-DONE depending on the values of b_1 and b_2.

- Case $\texttt{loop}\ e$: We have that $\texttt{fst}((0; \kappa)) = 0; \texttt{fst}(\kappa)$, so we can apply the induction hypothesis on e. As the behavior $(0; \kappa)^{\infty}$ is reactive, we know that $\kappa \downarrow^{\bullet}$. By applying Lemma 4, we get that $b = \mathit{ff}$, so we reconstruct the complete derivation for e using the LOOP-STUCK rule.

□

6. Discussion

6.1 Simplifying behaviors

The behaviors computed by the type system of Section 4 are very big. For instance, the behavior associated to the `timer` example is

$$((0 \mid\mid 0); (0 + (0; 0)))^{\infty}$$

This behavior is unnecessarily detailed and almost as big (if not bigger) than the source program. However, we can notice that this behavior is equivalent to 0^{∞}.

We would like to use the equivalence relation on behaviors (defined in Section 3.3) in our type system to reduce the size of the computed behaviors. We must make sure that this does not break the proof of soundness of Section 5. This is ensured by the fact that the equivalence relation preserves reactivity (Property 1), which is the only condition requested by the proof.

We can thus define a variant of the type system, which is the one implemented in the compiler, with a new typing judgment $\Gamma \vdash_S e : \tau \mid \kappa$. It is defined by the rules given in Figure 2 with one additional rule, that allows to simplify behaviors at any time using the equivalence relation:

$$(\text{EQUIV})\ \frac{\Gamma \vdash_S e : \tau \mid \kappa_1 \qquad \kappa_1 \equiv \kappa_2}{\Gamma \vdash_S e : \tau \mid \kappa_2}$$

We cannot add this rule to the original system as the proof of soundness relies on the fact that the behavior of sub-expressions is always smaller, which is no longer the case once we allow simplifications. We can also simplify some rules, by combining the

original rule with the EQUIV rule:

$$\frac{\Gamma \vdash_S e_1 : \tau_2 \mid \_ \qquad \Gamma \vdash_S e_2 : \tau_1 \to \tau_2 \to \tau_2 \mid \_ \qquad \Gamma, x : (\tau_1, \tau_2)\,\texttt{event} \vdash_S e : \tau \mid \kappa}{\Gamma \vdash_S \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \tau \mid \kappa}$$

$$\frac{\Gamma \vdash_S e_1 : \tau \mid \kappa \quad \Gamma \vdash_S e : (\tau_1, \tau_2)\,\texttt{event} \mid \_}{\Gamma \vdash_S \texttt{do}\ e_1\ \texttt{when}\ e : \tau \mid \kappa} \qquad \frac{\Gamma \vdash_S e : \tau \mid \kappa}{\Gamma \vdash_S \texttt{loop}\ e : \texttt{unit} \mid \kappa^\infty}$$

The EQUIV rule does not change the fact that a program is accepted or rejected by the type system. It only allows to reduce the size of computed behaviors:

Property 6. *If $\Gamma \vdash_S e : \tau \mid \kappa$, then $\Gamma \vdash e : \tau' \mid \kappa'$ with $\kappa \equiv \kappa'$ and $\tau \equiv \tau'$.*

Proof. Straightforward by induction on the typing rules □

We can then express the soundness theorem in terms of the variant of the type system:

Theorem 7 (Soundness (variant))**.** *If $\Gamma \vdash_S e : \tau \mid \kappa$ and τ and κ are reactive and we suppose that function calls terminate, then there exists e' such that $e \xrightarrow[S]{E,b} e'$ and $\Gamma \vdash_S e' : \tau \mid \kappa'$ with κ' reactive.*

Proof. By applying Property 6, Property 1 and Theorem 5. □

6.2 The run operator

So far, we have not justified the presence of the `run` operator in the language of behaviors. It is here to ensure that, even when adding simplification to the type system, the behavior associated to a recursive process is always a recursive behavior, that is, $\mu\phi.\,\kappa$ with $\phi \in fbv(\kappa)$.

Suppose that we remove the `run` operator from the language of behaviors. Now, consider the process $\texttt{rec}\ p = \texttt{process}\ (\texttt{run}\ p)$. In the variant of the type system, we could give it the instantaneous behavior 0 and miss the instantaneous recursion:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash_S p : \beta\,\texttt{process}[0 + \bullet] \mid 0}{\Gamma' \vdash_S \texttt{run}\ p : \beta \mid 0 + \bullet} \quad 0 + \bullet \equiv 0}{\Gamma' \vdash_S \texttt{run}\ p : \beta \mid 0}\ \text{EQUIV}}{\dfrac{\Gamma' \vdash_S \texttt{process}\ (\texttt{run}\ p) : \beta\,\texttt{process}[0 + \bullet] \mid 0}{\Gamma \vdash_S \texttt{rec}\ p = \texttt{process}\ (\texttt{run}\ p) : \beta\,\texttt{process}[0] \mid 0}}$$

where $\Gamma' = \Gamma, p : \beta\,\texttt{process}[0 + \bullet]$. Thanks to the addition of `run`, the only way to type this process is to give it the behavior $\mu\phi.\,(\texttt{run}\ \phi + \kappa')$ (where κ' is not constrained). This also explains why there is no equivalence rule to simplify a `run`. For instance, `run` 0 is not equivalent to 0.

6.3 Implementation

The type inference algorithm of ReactiveML has been extended to compute the behaviors of processes, with a small impact on its structure and complexity thanks to the use of row polymorphism for subeffecting (see Section 4.4). The rules given in Section 3.2 are easily translated into an algorithm for checking the reactivity of behaviors, polynomial in the size of behaviors. Inference simplifies behaviors during the computation, but does not necessarily compute the smallest behavior possible. For instance, simplifying $\kappa + \kappa$ can be costly in some cases, so it only checks simple cases (e.g. if $\kappa = 0$ or $\kappa = \bullet$). Overall, the analysis has a small impact on the compilation time of ReactiveML programs.

6.4 Examples

Using a type-based analysis makes it easy to deal with cases of aliasing, like in the following example:

```
let rec process p =
  let q = (fun x -> x) p in
  run q
val p : 'a process[rec 'r. (run 'r + ..)]
W: This expression may produce an instantaneous recursion
```

`q` has the same type as `p`, and thus the same behavior, so the instantaneous recursion is easily detected. As for objects in OCaml [26], row variables that appear only once are printed `'..'`.

The analysis can also handle combinators. For instance, the type system computes the following behavior for the `par_comb` example of Section 2.2:

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
val par_comb : 'a process['r1] -> 'b process ['r2] ->
 unit process[rec 'r3. ((run 'r1 || run 'r2); 'r3 + ..)]
```

There is no warning when defining the combinator because it is not possible to decide of its reactivity. Indeed, the synchronous parallel composition terminates when both branches have terminated. It means that the loop is non-instantaneous if either `q1` or `q2` is non-instantaneous. Formally, the computed behavior is reactive because free behavior variables are considered to be non-instantaneous (see Definition 2). The reactivity is then checked at the instantiation. If we instantiate the `par_comb` combinator with two anonymous processes, one instantaneous and the other non-instantaneous, then we obtain a process that is indeed reactive, so no warning is printed:

```
let process p1 =
  run (par_comb (process ()) (process (pause)))
val p1: unit process[run(rec 'r.(run 0 || run *);'r)+..]
```

However, if the two processes are instantaneous, then the loop becomes instantaneous. The behavior that results is obviously non-reactive, so a warning is shown:

```
let process p2 =
  run (par_comb (process ()) (process ()))
val p2: unit process[run(rec 'r.(run 0||run 0);'r) + ..)]
W: This expression may produce an instantaneous recursion
```

Here is another more complex example using higher-order functions and processes. We define a function `h_o` that takes as input a combinator `f`. It then creates a recursive process that applies `f` on itself and runs the result:

```
let h_o f =
 let rec process p =
   let q = f p in
   run q
 in p
val h_o : ('a process[run 'r1 + 'r2] -> 'a process['r1])
               -> 'a process[run 'r1 + 'r2]
```

If we instantiate this function with a process that waits an instant before calling its argument, we obtain a reactive process:

```
let process good =
  run (h_o (fun x -> process (pause; run x)))
val good :
 'a process[run (run (rec 'r1. *; run (run 'r1))) + ..]
```

This is no longer the case if the process calls its argument instantaneously. The instantaneous recursion is again detected by our static analysis:

```
let process pb =
  run (h_o (fun x -> process (run x)))
val pb :
    'a process[run (run (rec 'r1. run (run 'r1))) + ..]
W: This expression may produce an instantaneous recursion
```

Another interesting process that can be analyzed is a fix-point operator. It takes as input a function expecting a continuation, and applies it with itself as the continuation. This fix-point operator can be used to create a recursive process, which reactivity is checked by our analysis:

```
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[(run (rec 'r. run 'r)) + ..]
W: This expression may produce an instantaneous recursion
```

6.5 Adding references

References are not included in the kernel of our language. However, they are relevant to the matter as they can be used to encode recursivity, as in the following example that creates a process that loops instantaneously:

```
let landin () =
  let f = ref (process ()) in
  f := process (run !f);
  !f
val landin : unit ->
       unit process[0 + (rec 'r1. run (0 + 'r1)) + ..]
W: This expression may produce an instantaneous recursion
```

As our analysis does not have any special case for recursive processes and only relies on unification, it is able to detect the reactivity issue even though there is no explicit recursion.

6.6 Other models of concurrency

We believe this work could be applied to other models of concurrency. One just need to give the behavior $\bullet$ to operations that cooperate with the scheduler, like `yield`. The distinction between processes and functions is important to avoid showing a warning for all recursive function definitions.

In a synchronous world, the fact that each process cooperates at each instant implies the reactivity of the whole program, as processes are executed in lock-step. In another model, it may require assumptions on the fairness of the scheduler. This should not be a problem, as these hypotheses are already made in most systems, e.g. in Concurrent Haskell [13].

7. Related work

Our language of behaviors and type system is inspired by the work of [2]. Their analysis is done on the ConcurrentML [22] language, which extends ML with message passing primitives. The behavior of a process records every emission and reception on communication channels. The authors use the type system to prove properties on particular examples, not for a general analysis. For instance, they prove that the emission on a given channel always precede the emission on a second channel in a given program. The idea to use a type-and-effect system for checking reactivity or termination is not new. [7] uses a type-and-effect system to prove termination of functional programs using references, by stratifying memory to avoid recursion through references.

Reactivity analysis is a classic topic in synchronous languages, that can also be related to causality. In Esterel [5], the first imperative synchronous language, it is possible to react immediately to the presence *and* the absence of a signal. The consequence is that a program can be non-reactive because there is no consistent status for a given signal: the program supposes that a signal is both present and absent during the same instant. This problem is solved by checking that programs are *constructively correct* [4]. Our concurrency model, inherited from [8], avoids these problems by making sure that processes are causal by construction. We then only have to check that loops are not instantaneous, what is called *loop-safe* in [4]. It is easy to check that an Esterel program is loop-safe as the language is first order without recursion.

Closer to ReactiveML, the reactivity analysis of FunLoft [1] not only checks that instants terminate, but also give a bound on the duration of the instants trough a value analysis. The analysis is also restricted to the first-order setting. In ULM [6], each recursive call induces an implicit pause. Hence, it is impossible to have instantaneous recursions, at the expense of expressivity. For instance, in the `server` example of Section 2.2, a message could be lost between receiving a message on `add` and awaiting a new message.

The causality analysis of Lucid Synchrone [10] is a type-and-effect system using row types. It is based on the exception analysis done in [15]. Both are a more direct application of row types [21], whereas our system differs in the absence of labels in rows.

8. Conclusion

We have presented a reactivity analysis on the ReactiveML language. The idea of the analysis is to abstract processes into a simpler language called behaviors using a type-and-effect system. Checking reactivity of behaviors is then straightforward. We have proven the soundness of our analysis, that is, a well-typed program is reactive. Thanks in particular to the syntactic separation between functions and processes, the analysis does not detect too much false positives in practice. A previous version of this analysis was implemented in the ReactiveML compiler several years ago and it has been proven very useful for avoiding reactivity bugs.

An important direction for future work is to study the properties of our type system. In particular, it would be interesting to compare our approach to subeffecting using row polymorphism with traditional solutions and see if it could be applied in a different setting.

References

[1] R.M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous π-calculus. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 221–230. ACM, 2007.

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

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[4] G. Berry. The constructive semantics of pure esterel, 1996.

[5] G. Berry. The Esterel v5 language primer. *Ecole des Mines and INRIA*, 1997.

[6] G. Boudol. Ulm: A core programming model for global computing. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24725-8_17.

[7] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6):716–736, 2010.

[8] F. Boussinot. Reactive C: an extension of C to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[9] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(03):313–323, 1999.

[10] P. Cuoq and M. Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[11] D. Harel and A. Pnueli. On the Development of Reactive Systems. *Logics and models of concurrent systems*, page 477, 1985.

[12] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[13] S.P. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 21, pages 295–308. Citeseer, 1996.

[14] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):707–723, 1998.

[15] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):340–377, 2000.

[16] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[17] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[18] S. Marlow, S.P. Jones, and W. Thaller. Extending the Haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32. ACM, 2004.

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

[20] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[21] 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.

[22] J.H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 2007.

[23] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. *Practical Aspects of Declarative Languages*, pages 175–189, 2011.

[24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162 –173, jun 1992.

[25] M. Tofte. Type inference for polymorphic references. *Information and computation*, 89(1):1–34, 1990.

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

A. Well-formation of expressions

In order to separate pure ML expressions from reactive expressions, we define a well-formation predicate denoted $k \vdash e$ with $k \in \{0, 1\}$. An expression e is necessarily instantaneous (or combinatorial) if $0 \vdash e$. It is reactive (or sequential in classic circuit terminology) if $1 \vdash e$. The rules defining this predicate are given in Figure 4. The design choices of this analysis, like the fact that pairs must be instantaneous, are discussed in [17].

$k \vdash e$ means that $1 \vdash e$ and $0 \vdash e$. It means that e can be used in any context. This is true of any instantaneous expressions, as there is no rule with $0 \vdash e$ in the conclusion. The important point is that the body of functions must be instantaneous, while the body of a process may be reactive.

B. Big-step semantics (continued)

B.1 Notations

Signal environment

A *signal environment* S is a function

$$S \triangleq [(d_1, g_1, m_1)/n_1, \ldots, (d_k, g_k, m_k)/n_k]$$

that maps a signal name n_i to a tuple (d_i, g_i, m_i) where d_i is the default value of the signal, g_i its combination function and m_i is the multi-set of values emitted during the reaction. If the signal n_i has the type (τ_1, τ_2) `event`, then these fields have the following types:

$$d_i : \tau_2 \qquad g_i : \tau_1 \to \tau_2 \to \tau_2 \qquad m_i : \tau_2 \texttt{ multiset}$$

We denote $S^d(n_i) = d_i$, $S^g(n_i) = g_i$ and $S^m(n_i) = m_i$. We also define $S^v(n_i) = \mathit{fold}\ g_i\ m_i\ d_i$ where:

$$\mathit{fold}\ f\ (\{v_1\} \uplus m)\ v_2 = \mathit{fold}\ f\ m\ (f\ v_1\ v_2)$$
$$\mathit{fold}\ f\ \emptyset\ v_2 = v$$

We denote $n \in S$ when the signal n is present, that is, when $S^m(n) \neq \emptyset$, and $n \notin S$ otherwise.

Events

An *event* E is a function mapping a signal name to a multi-set of values:

$$E \triangleq [m_1/n_1, \ldots, m_k/n_k]$$

Events represent the values emitted during an instant. S^m is the event associated to the signal environment S.

Operations on signal environments and events

The union of events is the point-wise union, that is, if $E = E_1 \sqcup E_2$, then for all $n \in Dom(E_1) \cup Dom(E_2)$:

$$E(n) = E_1(n) \uplus E_2(n)$$

Similarly, the inclusion of events is the point-wise inclusion. We define the inclusion of signal environments by:

$$S_1 \sqsubseteq S_2 \text{ iff } S_1^m \sqsubseteq S_2^m$$

B.2 Big-step semantics

At each instant, the program reads inputs I_i and produces outputs O_i. The reaction of an expression is defined by the smallest signal environment S_i (for the relation $\sqsubseteq$) such that:

$$e_i \xrightarrow[S_i]{E_i, b_i} e_{i+1}$$

where

$$(I_i \sqcup E_i) \sqsubseteq S_i^m \tag{1}$$
$$O_i \sqsubseteq E_i \tag{2}$$

$$\frac{}{k \vdash x} \qquad \frac{}{k \vdash c} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad \frac{0 \vdash e}{k \vdash \lambda x.e} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash e_1\ e_2} \qquad \frac{0 \vdash e}{k \vdash \texttt{rec}\ x = e}$$

$$\frac{1 \vdash e}{k \vdash \texttt{process}\ e} \qquad \frac{0 \vdash e}{1 \vdash \texttt{run}\ e} \qquad \frac{}{1 \vdash \texttt{pause}} \qquad \frac{k \vdash e_1 \qquad k \vdash e_2 \qquad k \vdash e}{k \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e}$$

$$\frac{0 \vdash e_1 \qquad 0 \vdash e_2 \qquad k \vdash e}{k \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash \texttt{emit}\ e_1\ e_2} \qquad \frac{0 \vdash e \qquad 1 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{0 \vdash e \qquad k \vdash e_1 \qquad k \vdash e_2}{k \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{1 \vdash e}{1 \vdash \texttt{loop}\ e} \qquad \frac{1 \vdash e_1 \qquad 0 \vdash e \qquad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x) \to e_2} \qquad \frac{1 \vdash e_1 \qquad 0 \vdash e}{1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e}$$

Figure 4. Well-formation rules

$$v \xrightarrow[S]{\emptyset, tt} v \qquad \frac{e_1 \xrightarrow[S]{E_1, tt} v_1 \qquad e_2 \xrightarrow[S]{E_2, tt} v_2}{(e_1, e_2) \xrightarrow[S]{E_1 \sqcup E_2, tt} (v_1, v_2)} \qquad \frac{e_1 \xrightarrow[S]{E_1, tt} n \qquad e_2 \xrightarrow[S]{E_2, tt} v}{\texttt{emit}\ e_1\ e_2 \xrightarrow[S]{E_1 \sqcup E_2 \sqcup [\{v\}/n], tt} ()}$$

$$\frac{e_1 \xrightarrow[S]{E_1, tt} v_1 \qquad e_2 \xrightarrow[S]{E_2, tt} v_2 \qquad e[x \leftarrow n] \xrightarrow[S]{E, b} e' \qquad S(n) = (v_1, v_2, m)}{\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E, b} e'} \qquad \frac{e \xrightarrow[S]{E, tt} n \qquad e_1 \xrightarrow[S]{E_1, tt} v}{\texttt{do}\ e_1\ \texttt{until}\ e(x) \to e_2 \xrightarrow[S]{E \sqcup E_1, tt} v}$$

$$\frac{e \xrightarrow[S]{E, tt} n \qquad n \in S \qquad e_1 \xrightarrow[S]{E_1, ff} e_1'}{\texttt{do}\ e_1\ \texttt{until}\ e(x) \to e_2 \xrightarrow[S]{E \sqcup E_1, ff} e_2[x \leftarrow S^v(n)]} \qquad \frac{e \xrightarrow[S]{E, tt} n \qquad n \notin S \qquad e_1 \xrightarrow[S]{E_1, ff} e_1'}{\texttt{do}\ e_1\ \texttt{until}\ e(x) \to e_2 \xrightarrow[S]{E \sqcup E_1, ff} \texttt{do}\ e_1'\ \texttt{until}\ e(x) \to e_2}$$

$$\frac{e \xrightarrow[S]{E, tt} tt \qquad e_1 \xrightarrow[S]{E_1, b} e_1'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_1, b} e_1'} \qquad \frac{e \xrightarrow[S]{E, tt} ff \qquad e_2 \xrightarrow[S]{E_2, b} e_2'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_2, b} e_2'}$$

Figure 5. Remaining rules for the big-step semantics

$$S_i^d \subseteq S_{i+1}^d \text{ and } S_i^g \subseteq S_{i+1}^g \tag{3}$$

(1) The signal environment must contain the inputs and emitted signals.

(2) The outputs are included in the set of emitted signals.

(3) Default values and combination functions are kept from one instant to the next.

The rules defining the relation are given in Figure 3 and Figure 5:

- `emit` e_1 e_2 evaluates e_1 into a signal name n and adds the result of the evaluation of e_2 to the multi-set of values emitted on n.
- The declaration of a signal evaluates the default value and combination function, and then evaluates the body after substituting the variable x with a fresh signal name n. In this paper, we have kept implicit the sets of signal names that are used to ensure the freshness of this name (see [17] for the details).
- The preemption in ReactiveML is weak, that is, a process can only be preempted at the end of the instant. This is reflected by the fact that e_1 is always evaluated, regardless of the status of the signal n. It also means that, when the signal is present, e_2 is executed at the next instant.

The proof of soundness for these expressions follows the same patterns as in the proof of Theorem 5.