Reactivity of Cooperative Systems

Application to ReactiveML

Abstract

Cooperative scheduling enables efficient sequential implementations of concurrency. It is widely used to provide lightweight threads facilities as libraries or programming constructs in many programming languages. However, it is up to programmers to actually cooperate to ensure the reactivity of their programs.

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 objective is to find a good compromise between the complexity of the analysis and its precision for typical reactive programs. The simplicity of the analysis is mandatory for the programmer to be able to understand error messages and how to fix reactivity problems.

Our work is applied and implemented in the functional synchronous language ReactiveML. It handles recursion, higher-order processes and first-class signals. We prove the soundness of our analysis with respect to the big-step semantics: a well-typed program is reactive. The analysis 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 programming languages offer lightweight thread facilities, either integrated in the language like the asynchronous computations [27] of F#, or available as a library like Concurrent Haskell [16] or Lwt [31] 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 synchronization primitives like locks.

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

- Control must regularly be returned to the scheduler. This is particularly true for infinite loops, which are very often present in *reactive* and *interactive* systems.
- Blocking functions, like operating system primitives for I/O, cannot be called.

The solution to the latter is simple: never use blocking functions inside cooperative threads. All the facilities mentioned earlier provide either I/O libraries compatible with cooperative scheduling or a mean to safely call blocking functions. See [21] for an overview on how to implement such libraries.

Dealing with the first issue is usually the responsibility of the programmer. For instance, in the Lwt manual [13], one can find:

> [...] do not write function that may take time to complete without using Lwt [...]

The goal of this paper is to design a static analysis, called *reactivity analysis*, to statically remedy this problem. The analysis checks that the programmer does not forget to cooperate with the scheduler. Our work is applied to the ReactiveML language [20], which is an extension of ML with a synchronous model of concurrency [6]. Section 2 informally introduces the language and its semantics. The contributions of this paper are the following:

- A reactivity analysis presented as a type-and-effect system [19] in Section 4. The computed effects are called *behaviors* [3] and are introduced in Section 3. They represent the reactive behaviors of processes by abstracting away values but keeping 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. The use of a type system makes it possible to handle recursion, higher-order processes and first-class signals. We believe this approach is generic enough to be applied to other models of concurrency (Section 6.6).
- A novel approach to *subeffecting* [22], that is, subtyping on effects, based on row polymorphism [24] 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] and it has already helped detecting many reactivity bugs.

2. Problem statement

ReactiveML extends ML with programming constructs inspired from synchronous languages [6]. It introduces a built-in notion

[1] `http://www.reactiveml.org`

of parallelism where time is defined as a succession of logical instants. Each parallel process must cooperate to let time elapse. It is a deterministic model of concurrency that is compatible with the dynamic creation of processes [11]. Synchrony gives us a simple definition for reactivity: a reactive ReactiveML program is one where logical instants progress.

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. Processes 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 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 adding a **pause** statement.

We can then declare a local signal `s` and put these two processes in parallel. 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 when there are multiple emissions in a single instant. Here, the default value is `()` and the signal keeps this value in case of multi-emission. The `||` operator represents 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: 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 always satisfied by interesting reactive programs. 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 has instantaneous recursive calls, but it is reactive because the recursion is finite if the list `l` is finite. As the language allows to create mutable and recursive data structures, it is hard to prove the termination of such processes. For instance, the following expression never terminates:

```
let rec l = 0 :: l in run (par_map p l)
```

Consequently, our analysis only prints warnings and does not reject programs.

ML functions are always instantaneous so they are reactive if and only if they terminate. Since we do not want to prove their termination, our analysis has to distinguish recursions through functions and processes. This allows to assume that ML functions always terminate and to issue warnings only for processes.

Furthermore, we do not deal with blocking functions, like I/O primitives, that can also make programs non-reactive. Indeed, such functions should *never* be used in the context of cooperative scheduling. There are standard solutions for this problem. For example, a cooperative I/O library following the ideas in [21] could be implemented.

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

Neither does the analysis consider 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
```

The `present/then/else` construct executes its first branch instantaneously 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 [11], 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 the duration of each instant is finite, not that the program is real-time, that is, that there exists a bound on this duration for all instants, as shown in 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 listens on a signal `add` to receive both a process `p` and a signal `ack` on which to send back results. 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 [3]. 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 reasonable precision.

3.1 The behaviors

The algebra of behaviors is given by:[3]

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

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, since ReactiveML has higher-order processes.

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. To represent such processes, behaviors include the parallel composition, simply denoted $\parallel$. 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.

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
W: This expression may produce an instantaneous recursion
```

The order between the recursive call and the `pause` statement 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 $\kappa = \bullet; \mathtt{run}\ \kappa$. The `run` operator is associated with running a process. This equation can be solved by introducing an explicit recursion operator μ so that $\kappa = \mu\phi.\,\bullet; \mathtt{run}\ \phi$. Recursive behaviors are defined as usual:

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

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:

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

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 which behaviors are reactive, we first have to define the notion of a *non-instantaneous* behavior, i.e. processes that take at least one instant to execute:

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

$$\frac{}{\bullet\downarrow} \qquad \frac{}{\phi\downarrow} \qquad \frac{\kappa_1\downarrow}{\kappa_1;\kappa_2\downarrow} \qquad \frac{\kappa_2\downarrow}{\kappa_1;\kappa_2\downarrow} \qquad \frac{\kappa_1\downarrow}{\kappa_1 \parallel \kappa_2\downarrow}$$

$$\frac{\kappa_2\downarrow}{\kappa_1 \parallel \kappa_2\downarrow} \qquad \frac{\kappa_1\downarrow \quad \kappa_2\downarrow}{\kappa_1 + \kappa_2\downarrow} \qquad \frac{\kappa\downarrow}{\mu\phi.\,\kappa\downarrow} \qquad \frac{\kappa\downarrow}{\mathtt{run}\ \kappa\downarrow}$$

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. This is formalized in the following definition.

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

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

Figure 1. Derived language constructs

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

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

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

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

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. We must still, however, check that κ_2 is reactive, that is, that $\varnothing \vdash \kappa_2$.

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. +). We can define the $\equiv$ relation as the smallest relation that also satisfies the following properties:

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

It is easy to show, for example, 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 [19], following the work of Amtoft et al. [3]. The behavior of a process is its effect computed using the type system. A type system is a simple way to specify a higher-order static analysis. It can be implemented efficiently using classic destructive unification techniques [17].

4.1 Abstract syntax

We consider here a kernel of ReactiveML:

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

Values are constants c (integers, booleans, unit value `()`, 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:

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

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 (τ_1, τ_2) `event` of a signal is parametrized by the type τ_1 of emitted values and the type τ_2 of the received value (since a gathering function of type $\tau_1 \to \tau_2 \to \tau_2$ is applied).

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

$$\mathit{fv}(\tau) = \mathit{ftv}(\tau), \mathit{fbv}(\tau)$$

Instantiation and generalization are defined in a classic way:

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

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \_}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \mid 0} \qquad (\text{APP}) \; \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \mid \_ \qquad \Gamma \vdash e_2 : \tau_2 \mid \_}{\Gamma \vdash e_1 \; e_2 : \tau_1 \mid 0} \qquad \frac{\Gamma, x : \tau \vdash e : \tau \mid \_}{\Gamma \vdash \texttt{rec}\; x = e : \tau \mid 0}$$

$$(\text{PROCESS}) \; \frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\; e : \tau\, \texttt{process}[\kappa + \kappa'] \mid 0} \qquad \frac{\Gamma \vdash e : \tau\, \texttt{process}[\kappa] \mid \_}{\Gamma \vdash \texttt{run}\; e : \tau \mid \texttt{run}\; \kappa}$$

$$\Gamma \vdash \texttt{pause} : \texttt{unit} \mid \bullet \qquad \frac{\Gamma \vdash e : \texttt{bool} \mid \_ \qquad \Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \texttt{if}\; e \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 : \tau \mid \kappa_1 + \kappa_2}$$

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

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

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

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

Figure 2. Type-and-effect rules

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

Analogously to the treatment of references in ML, we must be careful not to generalize expressions that allocate signals. We use the syntactic criterion of expansive and non-expansive expressions [30]. An expression is expansive if it can allocate a signal or a reference, in which case its type should not be generalized.

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.

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 primitives:

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

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 [20], which is itself an extension of the ML type system. It is a conservative extension of this type system, that is, we are able to assign a behavior to any correct ReactiveML program. It is an important feature as we only want to show warnings and not reject programs. We discuss here the novelties of these rules related to behaviors:

- The PROCESS rule stores the behavior of the body in the type of the process, as usual in type-and-effect systems. The presence of the κ' behavior and the MASK rule 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 performed before typing checks this well-formation of expressions, denoted $k \vdash e$ in [20] and recalled 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 assume that function calls always terminate instantaneously. That is why there is no behavior associated to functions, that is, there are no behaviors on arrows unlike traditional type-and-effect systems.
- Note that there is no special treatment for recursive processes. Recursive behaviors appear during unification. An example of typing derivation justifying the presence of the `run` operator will be given in Section 6.2.
- 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 in 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.

- The encoding of primitives given in Figure 1 yields the expected behaviors. For example, consider the case of $e_1; e_2$:

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

It is easy to show that $(\kappa_1 \parallel 0); \kappa_2 \equiv \kappa_1; \kappa_2$. We can also show that $e_1 \parallel e_2$ has equivalent behavior to $\kappa_1 \parallel \kappa_2$ or that $\mathtt{await}\ e_1(x)\ \mathtt{in}\ e_2$ has behavior $\bullet^\infty + (\bullet; \kappa_2) \equiv \bullet; \kappa_2$.

- In [20], the `pause` operator is encoded by:

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

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, $0; (0 + \bullet; 0) \equiv 0$, is the opposite of the expected behavior of `pause`.

- The `loop` construct can be encoded as a recursive process by:

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

Unlike `pause`, we could have removed `loop` from our kernel and used this encoding without affecting the results given by the reactivity analysis. Indeed, by applying the rules here, we have that:

$$loop : \forall \phi.\ \alpha\ \mathtt{process}[\phi] \rightarrow \alpha'\ \mathtt{process}[\mu\phi'.\ \mathtt{run}\ \phi; \mathtt{run}\ \phi']$$

If we assume that $\Gamma \vdash e : \tau \mid \kappa$, then the behavior of this encoding is: $\mathtt{run}\ (\mu\phi'.\ \mathtt{run}\ \kappa; \mathtt{run}\ \phi')$. 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).

- The reason why the behavior associated with `loop` is equal to $(0; \kappa)^\infty$ and not simply κ^∞ will be explained in Section 5. Intuitively, the soundness proof requires the behavior of a subexpression to always be smaller. This also applies for `signal` and `do/when`.

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. The idea is to add a free behavior variable to represent other potential behaviors of the process. This subtyping restricted to effects is often referred to as *subeffecting* [22]: we can always replace an effect with a bigger, i.e. less precise, one. It allows to assign a behavior to any correct ReactiveML program. For instance, we can give a type to the following expression, that defines a list of two processes with different behaviors (the $\bullet$ behavior is printed `'*'` and behavior variables ϕ are denoted `'r`):

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

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

Usually, subeffecting is not expressed as a syntax directed rule. For instance, in [22, 28], it is defined as (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 [28] (of regions for example). In our case, It is defined by:

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

In [3], effects must 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:

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

These three formulations appear to be equivalent. Indeed, our system and the one of [3] are just syntax-directed versions of the classic approach, where the subsumption rule is mixed with lambda abstractions (or process abstractions in our case).

Implementing subeffecting with row polymorphism

The consequence of the typing rule for processes is that the principal type of an expression $\mathtt{process}\ e$ is always of the form $\kappa + \phi$. The idea to use a free type variable to represent other possible types is reminiscent of Remy's row types [24]. It makes it possible to implement subeffecting using only unification, without manipulating constraint sets as in traditional approaches [3, 28]. It thus becomes 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 introducing two different kinds of behaviors: behaviors and rows of behaviors. We chose instead to retain our simpler syntax. We can reuse any existing inference algorithm, like algorithm $\mathcal{W}$ or $\mathcal{M}$ [17] and add only 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 $[\phi_1 \mapsto \kappa_1; \phi_2 \mapsto \kappa_2; \ldots]$. It is defined as follows:

$$\mathcal{U}_\kappa(\kappa, \kappa) = []$$
$$\mathcal{U}_\kappa(\phi, \kappa) = \mathcal{U}_\kappa(\kappa, \phi) = \begin{cases} [\phi \mapsto \mu\phi.\ \kappa] & \text{if } \mathtt{occur\_check}(\phi, \kappa) \\ [\phi \mapsto \kappa] & \text{otherwise} \end{cases}$$
$$\mathcal{U}_\kappa(\kappa_1 + \phi_1, \kappa_2 + \phi_2) = [\phi_1 \mapsto \kappa_2 + \phi; \phi_1 \mapsto \kappa_1 + \phi],\ \phi \text{ fresh}$$
$$\mathcal{U}_\kappa(\mu\phi'_1.\ (\kappa_1 + \phi_1), \kappa_2) = \mathcal{U}_\kappa(\kappa_2, \mu\phi'_1.\ (\kappa_1 + \phi_1))$$
$$= \mathtt{let}\ K_1 = \mu\phi'_1.\ (\kappa_1 + \phi_1)\ \mathtt{in}$$
$$\mathcal{U}_\kappa(\kappa_1[\phi'_1 \leftarrow K_1] + \phi_1, \kappa_2)$$

It should be noted that unification never fails, so that we obtain a conservative extension of ReactiveML type system. This unification algorithm reuses traditional techniques for handling recursive types [14]. 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 become very big and unreadable. The purpose of the MASK rule is to remedy this, by using *effect masking* [19]. 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.

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

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

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

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

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

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

Figure 3. Big-step semantics

5. Proof of soundness

We will now prove the soundness of our analysis, that is, that a well-typed program is reactive. It means that, at each instant, the program admits a finite derivation in the big-step semantics of the language and rewrites to a well-typed program. 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 the finiteness of the semantics derivation by induction on the size of behaviors.

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:

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

In the case of a recursive behavior, the first-instant behavior is well-defined only if the behavior is reactive, that is, if 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:

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

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

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

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

Proof. By induction on the structure of behaviors. □

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 semantics of Esterel [7] from which it is inspired. The interested reader can refer to Appendix B or [20] 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. An execution of a program comprises a 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. when it has been emitted in the current instant) and $n \notin S$ otherwise.

Figure 3 shows some 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, if 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 that body stops to wait for the next instant, that is, until 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 we focus completely on processes. We assume 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. It must be expressed formally with respect to the big-step semantics before we can show soundness. To do so, we introduce the predicate $0 \vdash e$ which is defined in Appendix A and means that the expression e is surely instantaneous, that is, it 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.*

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

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

We can now express the soundness of our analysis, that is, that a well-typed program is reactive. Being reactive means 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 has two parts. To prove that the result is well-typed, we use classic syntactic techniques for type soundness [23] on the small-step semantics described in [20]. The proof of equivalence of the two semantics is also given in the same paper. The proof that the semantics derivation is finite is by induction on the size of the first-instant behavior of well-typed expressions. We only consider the first-instant behavior because it is finite for a reactive behavior, whereas the complete behavior can be infinite.

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

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

The expression e rewrites to a value by Hypothesis 3, that is necessarily of the form $\texttt{process}\ e_1$ because it is the only kind of value that can have type $\tau\,\texttt{process}[\kappa]$. Then, we can construct the following type derivation:

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

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 $\texttt{run}\ (\texttt{process}\ e_1)$, which is also the first-instant behavior of $\texttt{run}\ e$ because e and $\texttt{process}\ e_1$ have the same type (by subject reduction). Indeed, we have:

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

The induction hypothesis allows the conclusion:

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

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

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

- Case `pause`: The derivation is already finite.
- Case $\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: As in the first case:

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

The typing rule is:

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

There are then two cases depending on the status of n:
  - If $n \in S$: We 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 might 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 $(\kappa_1 \parallel \kappa_2)\downarrow$, then we have that

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

From $(\kappa_1 \parallel \kappa_2) \downarrow$, we get that either $\kappa_1 \downarrow$ which implies that $b_1 = \mathit{ff}$, or $\kappa_2 \downarrow$ 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

$$\mathtt{fst}(\kappa) = (\mathtt{fst}(\kappa_1) \parallel \mathtt{fst}(\kappa_2)); \mathtt{fst}(\kappa_3)$$

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 `loop` e: We have that $\mathtt{fst}((0; \kappa)) = 0; \mathtt{fst}(\kappa)$, so we can apply the induction hypothesis on e. As the behavior $(0; \kappa)^\infty$ is reactive, we know that $\kappa \downarrow$. 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

$$((0 \parallel 0); (0 + (0; 0)))^\infty$$

This behavior is unnecessarily detailed and almost as big as the source program. It is, however, 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. This can be expressed as a new typing rule, that allows to simplify behaviors at any time using the equivalence relation. We denote $\vdash_S$ the type system augmented with the new rule:

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

In order to use this rule, we have to show that it is *admissible*, that is, that it does not change the set of accepted programs. This is ensured by the fact that the equivalence relation preserves reactivity (Property 1), which is one of the conditions requested by the soundness proof.

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 type derivation □

The immediate consequence of this property is that the augmented type system is also sound. We can also simplify some rules by combining the original rule with the EQUIV rule:

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

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

6.2 The `run` operator

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

Suppose that we remove the `run` operator from the language of behaviors. Now, consider the process $\mathtt{rec}\ p = \mathtt{process}\ (\mathtt{run}\ p)$. In the type system without `run`, we could give it the instantaneous behavior 0 and miss the instantaneous recursion:

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

where $\Gamma' = \Gamma, p : \beta\,\mathtt{process}[0 + \bullet]$. Thanks to the addition of `run`, $\mathtt{run}\ p$ now has a behavior equal to $\mathtt{run}\ \phi$. Then, the only way to type the process is to give it the behavior $\mu\phi.\,(\mathtt{run}\ \phi + \kappa')$ (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 minimal 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 that is 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, as in the following example:[4]

```
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
```

The process `q` has the same type as `p`, and thus the same behavior, so the instantaneous recursion is easily detected. As for objects in OCaml [31], 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 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)+..]
```

[4] These examples can be downloaded and tried in the supplementary material submitted with the paper.

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` to 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, whose 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 Handling references

References are not included in the kernel of our language. However, they are relevant 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 such reactivity problems 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 needs to give the behavior • 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, assumptions on the fairness of the scheduler may be required. This should not be a major obstacle, as these hypotheses are already made in most systems, e.g. in Concurrent Haskell [16].

7. Related work

The analysis of instantaneous loops is an old topic on which much has already been written, even very recently [1, 4, 15]. It is related to the analysis of productivity and deadlocks in list programs [26] or guard conditions in proof assistants [5], etc. Our purpose was to define an analysis that can be used in the context of a general purpose language (mutable values, recursion, etc.). We tried to find a good compromise between the complexity of the analysis and its precision for typical reactive programs written in ReactiveML. The programmer must not be surprised by the analysis and the error messages. We focus here only on directly related work.

Our language of behaviors and type system are inspired by the work of Amtoft et al. [3]. Their analysis is defined on the ConcurrentML [25] 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 emission on a given channel always precede the emission on a second channel in a given program. The idea of using a type-and-effect system for checking reactivity or termination is not new. For instance, Boudol [10] 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 [8], 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* [7]. Our concurrency model, inherited from the work of Boussinot [11], avoids these problems by making sure that processes are causal by construction. We then only have to check that loops are not instantaneous, which is called *loop-safe* by Berry [7]. It is easy to check that an Esterel program is loop-safe as the language is first order without recursion [29].

Closer to ReactiveML, the reactivity analysis of FunLoft [2] not only checks that instants terminate, but also gives a bound on the duration of the instants through a value analysis. The analysis is also restricted to the first-order setting. In ULM [9], 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 [12] is a type-and-effect system using row types. It is based on the exception analysis defined by Leroy et al. [18]. Both are a more direct application of row types [24], whereas our system differs in the absence of labels in rows.

8. Conclusion

We have presented a reactivity analysis for 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, that a well-typed program is reactive. Thanks in particular to the syntactic separation between functions and processes, the analysis does not detect too many false positives in practice. It is implemented in the ReactiveML compiler and it has been proven very useful for avoiding reactivity bugs.

References

[1] A. Abel and B. Pientka. Well-founded recursion with copatterns. In *ICFP'13*, 2013.

[2] R. Amadio and F. Dabrowski. Feasible reactivity in a synchronous π-calculus. In *PPDP'07*, pages 221–230. ACM, 2007.

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

[4] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP'13*, 2013.

[5] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in computer science*, 14(01):97–141, 2004.

[6] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[7] G. Berry. The Constructive Semantics of Pure Esterel, 1996.

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

[9] G. Boudol. ULM: A core programming model for global computing. In D. Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *LNCS*, pages 234–248. Springer Berlin / Heidelberg, 2004.

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

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

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

[13] J. Dimino. *Lwt User Manual*, 2012. `http://ocsigen.org/lwt/manual/`.

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

[15] A. Jeffrey. Functional reactive programming with liveness guarantees. In *ICFP'13*, 2013.

[16] S. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL'96*, volume 21, pages 295–308. Citeseer, 1996.

[17] 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.

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

[19] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88*, pages 47–57, New York, NY, USA, 1988. ACM.

[20] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP'05*, pages 82–93. ACM, 2005.

[21] S. Marlow, S. 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.

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

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

[24] 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.

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

[26] B. A. Sijtsma. On the productivity of recursive list definitions. *TOPLAS'89*, 11(4):633–649, 1989.

[27] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. *PADL*, pages 175–189, 2011.

[28] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *LICS '92*, pages 162 –173, jun 1992.

[29] O. Tardieu and R. de Simone. Loops in Esterel. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):708–750, 2005.

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

[31] 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 [20].

$k \vdash e$ means that $1 \vdash e$ or $0 \vdash e$, that is, 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 $(\tau_1, \tau_2)\,\texttt{event}$, 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:

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

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

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

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

$$\frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash \texttt{emit}\ e_1\ e_2} \qquad \frac{0 \vdash e \qquad 1 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{0 \vdash e \qquad k \vdash e_1 \qquad k \vdash e_2}{k \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{1 \vdash e}{1 \vdash \texttt{loop}\ e}$$

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

Figure 4. Well-formation rules

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

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

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

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

Figure 5. Remaining rules for the big-step semantics

Events

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

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

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:

$$e_i \xrightarrow[S_i]{E_i, b_i} e_{i+1} \quad \text{where} \quad \begin{array}{ll} (I_i \sqcup E_i) \sqsubseteq S_i^m & (1) \\ O_i \sqsubseteq E_i & (2) \\ S_i^d \subseteq S_{i+1}^d \text{ and } S_i^g \subseteq S_{i+1}^g & (3) \end{array}$$

(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 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 [20] 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.