

# Reactivity of Cooperative Systems

## Application to ReactiveML

Louis Mandel<sup>1,3</sup> and Cédric Pasteur<sup>2,3</sup>

<sup>1</sup> Collège de France

<sup>2</sup> DI École normale supérieure (now at ANSYS-Esterel Technologies)

<sup>3</sup> INRIA Paris-Rocquencourt

**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 of the language: a well-typed program with reactive effects is reactive. The analysis is easy to implement and generic enough to be applicable to other models of concurrency such as coroutines.

## 1 Introduction

Most programming languages offer lightweight thread facilities, either integrated in the language like the asynchronous computations [30] of F#, or available as a library like GNU Pth [12] for C, Concurrent Haskell [15] or Lwt [34] 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 means to safely call blocking functions. See Marlow et al. [22] 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 [11], 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 of absence of cooperation points. The analysis checks that the programmer does not forget to cooperate with the scheduler. Our work is applied to the ReactiveML language [19], which is an extension of ML with a synchronous model of concurrency [6] (Section 2). However, we believe that our approach is generic enough to be applied to other models of concurrency (Section 6). The contributions of this paper are the following:

- A reactivity analysis based on a type-and-effect system [18] in Section 4. The computed effects are called *behaviors* [3] and are introduced in Section 3. They represent the temporal behaviors of processes by abstracting away values but keeping part of the structure of the program and are used to check if processes cooperate or not.
- A novel approach to *subeffecting* [23], that is, subtyping on effects, based on row polymorphism [26] in Section 4.4. It allows to build a conservative extension of the existing type system with little overhead.
- A proof of the soundness of the analysis (Section 4.5): *a well-typed program with reactive effects is reactive*.

The paper ends with some examples (Section 5), discussion (Section 6) and related work (Section 7). The work presented here is implemented in the ReactiveML compiler<sup>4</sup> and it has already helped detecting many reactivity bugs. An extended version of the paper, the implementation, the source code of the examples and an online toplevel are available at <http://reactiveml.org/sas14>.

## 2 Overview of the approach

ReactiveML extends ML with programming constructs inspired from synchronous languages [6]. It introduces a built-in notion 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 [9]. Synchrony gives us a simple definition for reactivity: a reactive ReactiveML program is one where each logical instant terminates.

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

---

<sup>4</sup> <http://www.reactiveml.org>

## 2.1 A first example

We start by creating a *process* that emits a signal every `timer` seconds:<sup>5</sup>

```
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 then (emit s (); time := time')
6   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 6). At each iteration, it reads the new current time and emits the unit value on the signal `s` if enough time has elapsed (line 5). The compiler prints the following warning when compiling this process:

*W: Line 3, characters 2-115, 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 6):

```
5   if time' -. !time >= timer then (emit s (); time := time');
6   pause
7 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
```

*W: Line 11, characters 2-78, 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,

---

<sup>5</sup> The vocabulary is the one of synchronous languages, not the one of FRP.

the body of the **do/when** executes one more time, and so on. This process can also be fixed by adding a **pause** statement in the loop.

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 x y -> ()) 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:

```
let rec process instantaneous s =  
  emit s (); run (instantaneous s)
```

*W: This expression may produce an instantaneous recursion*

The execution of **emit** is instantaneous, therefore the recursive call creates a loop that never cooperates. 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, matching is instantaneous):

```
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 cooperates:

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

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

ML functions are always considered 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 us to assume that ML functions always terminate and to issue warnings only for processes.

Furthermore, we do not deal with blocking functions, such as 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 [22].

This analysis does not either consider the presence status of signals. It over-approximates the possible behaviors, as in the following example:

```
let rec process imprecise =
  signal s default () gather (fun x y -> ()) 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 by Boussinot [9], 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 Amtoft et al. [3] and to check reactivity on these behaviors. The main design choice is to completely abstract values and the presence of signals. It is however necessary to keep an abstraction of the structure of the program in order to have a reasonable precision.

#### 3.1 The behaviors

The algebra of behaviors is given by:<sup>6</sup>

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

<sup>6</sup> Precedence of operators is the following (from highest to lowest): **run**, **;**, **+**, **||** and finally  $\mu$ . For instance:  $\mu\phi. \kappa_1 \parallel \mathbf{run} \kappa_2 + \bullet ; \kappa_3$  means  $\mu\phi. (\kappa_1 \parallel ((\mathbf{run} \kappa_2) + (\bullet ; \kappa_3)))$ .

Actions that take at least one instant to execute, i.e. surely non-instantaneous actions, 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  $\phi$  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 `||`. 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. As we want to abstract values, we represent the different alternatives using a non-deterministic choice operator `+` 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  $\kappa$  associated with the `good_rec` process must verify that  $\kappa = \bullet; \text{run } \kappa$ . The `run` operator is associated with running a process. This equation can be solved by introducing an explicit recursion operator  $\mu$  so that  $\kappa = \mu\phi. \bullet; \text{run } \phi$ . Recursive behaviors are defined as usual:

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

We denote  $\text{fbv}(\kappa)$  the set of free behavior variables in  $\kappa$ . 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  $\kappa^\infty$  (where  $\kappa$  is the behavior of the body of the loop), is thus defined as a recursive behavior by  $\kappa^\infty \triangleq \mu\phi. (\kappa; \text{run } \phi)$ .

### 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. 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:

$$\begin{array}{c} \frac{}{\bullet \downarrow} \quad \frac{}{\phi \downarrow} \quad \frac{\kappa_1 \downarrow}{\kappa_1; \kappa_2 \downarrow} \quad \frac{\kappa_2 \downarrow}{\kappa_1; \kappa_2 \downarrow} \quad \frac{\kappa_1 \downarrow}{\kappa_1 \parallel \kappa_2 \downarrow} \quad \frac{\kappa_2 \downarrow}{\kappa_1 \parallel \kappa_2 \downarrow} \quad \frac{\kappa_1 \downarrow \quad \kappa_2 \downarrow}{\kappa_1 + \kappa_2 \downarrow} \\ \\ \frac{\kappa \downarrow}{\mu\phi. \kappa \downarrow} \quad \frac{\kappa \downarrow}{\mathbf{run} \kappa \downarrow} \end{array}$$

Note that function calls are not non-instantaneous. The fact that variables are considered non-instantaneous means that any process taken as argument is supposed to be non-instantaneous. The reactivity is then checked 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  $\phi$  does not appear in the first instant of the body  $\kappa$ . This enforces that there must be at least one instant between the instantiation of a process and each recursive call and is formalized in the following definition.

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

$$\begin{array}{c} \frac{}{R \vdash 0} \quad \frac{}{R \vdash \bullet} \quad \frac{\phi \notin R}{R \vdash \phi} \quad \frac{R \vdash \kappa_1 \quad \text{not}(\kappa_1 \downarrow) \quad R \vdash \kappa_2}{R \vdash \kappa_1; \kappa_2} \\ \\ \frac{R \vdash \kappa_1 \quad \kappa_1 \downarrow \quad \emptyset \vdash \kappa_2}{R \vdash \kappa_1; \kappa_2} \quad \frac{R \vdash \kappa_1 \quad R \vdash \kappa_2}{R \vdash \kappa_1 \parallel \kappa_2} \quad \frac{R \vdash \kappa_1 \quad R \vdash \kappa_2}{R \vdash \kappa_1 + \kappa_2} \\ \\ \frac{R \cup \{\phi\} \vdash \kappa}{R \vdash \mu\phi. \kappa} \quad \frac{R \vdash \kappa}{R \vdash \mathbf{run} \kappa} \end{array}$$

The predicate  $R \vdash \kappa$  means that the behavior  $\kappa$  is reactive with respect to the set of variables  $R$ , that is, these variables do not appear in the first instant of  $\kappa$  and all the recursions inside  $\kappa$  are not instantaneous. The rule for a variable  $\phi$  checks that  $\phi$  is not a recursion variable introduced in the current instant. The recursion variables are added to the set  $R$  when checking the reactivity of a recursive behavior  $\mu\phi. \kappa$ . In the case of the sequence  $\kappa_1; \kappa_2$ , we can remove variables from  $R$  if  $\kappa_1$  is non-instantaneous. One can also check from the definition of  $\kappa^\infty$  as a recursive behavior that this definition also implies that the body of a loop is non-instantaneous.

### 3.3 Equivalence on behaviors

We can define an equivalence relation  $\equiv$  on behaviors that will be used to simplify the behaviors. The relation is reflexive, symmetric and transitive closure of the following rules. The operators  $;$  and  $\parallel$  and  $+$  are idempotent and associative.  $\parallel$  and  $+$  are commutative (but not  $;$ ). The  $0$  behavior (resp.  $\bullet$ ) is the neutral

element of  $;$  and  $\parallel$  (resp.  $+$ ). The relation also satisfies the following rules (where  $op$  is  $;$  or  $\parallel$  or  $+$ ):

$$\frac{\kappa_1 \equiv \kappa_2}{\mu\phi. \kappa_1 \equiv \mu\phi. \kappa_2} \quad \frac{\kappa_1 \equiv \kappa_2}{\mathbf{run} \kappa_1 \equiv \mathbf{run} \kappa_2} \quad \bullet^\infty \equiv \bullet \quad \frac{\kappa_1 \equiv \kappa'_1 \quad \kappa_2 \equiv \kappa'_2}{\kappa_1 \mathit{op} \kappa_2 \equiv \kappa'_1 \mathit{op} \kappa'_2}$$

It is easy to show, for example, that:  $\mu\phi. ((\bullet \parallel 0); (\mathbf{run} \phi + \mathbf{run} \phi)) \equiv \mu\phi. (\bullet; \mathbf{run} \phi)$ . An important property of this relation is that it preserves reactivity. It is expressed as follows:

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

*Proof.* By induction on the proof of  $\kappa_1 \equiv \kappa_2$ . □

## 4 The type-and-effect system

The link between processes and behaviors is done by a type-and-effect system [18], following the work of Amtoft et al. [3]. The behavior of a process is its effect computed using the type system. After type-checking, the compiler checks the inferred behaviors and prints a warning if one of them is not reactive.

The type system is a conservative extension of the original one of ReactiveML, that is, it is able to assign a behavior to any ReactiveML program that was accepted previously. It is an important feature as we only want to show warnings and not reject programs.

### 4.1 Abstract syntax

We consider here a kernel of ReactiveML:

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

Values are constants  $c$  (integers, 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  $\mathbf{do} \ e_1 \ \mathbf{until} \ s(x) \rightarrow e_2$  executes its body  $e_1$  and interrupts it if  $s$  is present. In case of preemption, the continuation  $e_2$  is executed 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:

$$\begin{aligned}
e_1 \parallel e_2 &\triangleq \text{let } \_ = e_1 \text{ and } \_ = e_2 \text{ in } () \\
e_1; e_2 &\triangleq \text{let } \_ = () \text{ and } \_ = e_1 \text{ in } e_2 \\
\text{await } e_1(x) \text{ in } e_2 &\triangleq \text{do (loop pause) until } e_1(x) \rightarrow e_2 \\
\text{let rec process } f x_1 \dots x_p = e_1 \text{ in } e_2 & \\
&\triangleq \text{let } f = (\text{rec } f = \lambda x_1 \dots \lambda x_p. \text{process } e_1) \text{ in } e_2
\end{aligned}$$

## 4.2 Types

Types are defined by:

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

A type is either a type variable  $\alpha$ , 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  $(\tau_1, \tau_2)$  `event` of a signal is parametrized by the type  $\tau_1$  of emitted values and the type  $\tau_2$  of the received value (since a gathering function of type  $\tau_1 \rightarrow \tau_2 \rightarrow \tau_2$  is applied).

Types schemes quantify universally over type variables  $\alpha$  and behavior variables  $\phi$ . We denote  $ftv(\tau)$  (resp.  $fbv(\tau)$ ) the set of type (resp. behavior) variables free in  $\tau$  and  $fv(\tau) = ftv(\tau), fbv(\tau)$ . Instantiation and generalization are defined in a classic way:

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

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 [33]. 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  $\Gamma$ , the expression  $e$  has type  $\tau$  and behavior  $\kappa$ . We write  $\Gamma \vdash e : \tau \mid \_ \equiv 0$  when the behavior of the expression  $e$  is equivalent to 0. The initial typing environment  $\Gamma_0$  gives the types of primitives:

$$\Gamma_0 \triangleq [\text{true} : \text{bool}; \text{fst} : \forall \alpha_1, \alpha_2. \alpha_1 \times \alpha_2 \rightarrow \alpha_1; \dots]$$

The rules defining the type system are given in Figure 1. If all the behaviors are erased, it is exactly the same type system as the one presented in [19], which is itself an extension of the ML type system. We discuss here the novelties of the rules related to behaviors:

$$\begin{array}{c}
\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau \mid 0} \quad \frac{\tau \leq \Gamma_0(c)}{\Gamma \vdash c : \tau \mid 0} \quad \frac{\Gamma \vdash e_1 : \tau_1 \mid \_ \equiv 0 \quad \Gamma \vdash e_2 : \tau_2 \mid \_ \equiv 0}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mid 0} \\
\\
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \_ \equiv 0}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \mid 0} \quad (\text{APP}) \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \mid \_ \equiv 0 \quad \Gamma \vdash e_2 : \tau_2 \mid \_ \equiv 0}{\Gamma \vdash e_1 e_2 : \tau_1 \mid 0} \\
\\
\frac{\Gamma, x : \tau \vdash v : \tau \mid \_ \equiv 0}{\Gamma \vdash \mathbf{rec} x = v : \tau \mid 0} \quad (\text{PROCESS}) \frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \mathbf{process} e : \tau \mathbf{process}[\kappa + \kappa'] \mid 0} \\
\\
\frac{\Gamma \vdash e : \tau \mathbf{process}[\kappa] \mid \_ \equiv 0}{\Gamma \vdash \mathbf{run} e : \tau \mid \mathbf{run} \kappa} \quad \Gamma \vdash \mathbf{pause} : \mathbf{unit} \mid \bullet \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \mid \kappa_1 \quad \Gamma \vdash e_2 : \tau_2 \mid \kappa_2 \quad \Gamma, x_1 : \mathbf{gen}(\tau_1, e_1, \Gamma), x_2 : \mathbf{gen}(\tau_2, e_2, \Gamma) \vdash e_3 : \tau \mid \kappa_3}{\Gamma \vdash \mathbf{let} x_1 = e_1 \mathbf{and} x_2 = e_2 \mathbf{in} e_3 : \tau \mid (\kappa_1 \parallel \kappa_2); \kappa_3} \\
\\
\frac{\Gamma \vdash e_1 : \tau_2 \mid \_ \equiv 0 \quad \Gamma \vdash e_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \mid \_ \equiv 0 \quad \Gamma, x : (\tau_1, \tau_2) \mathbf{event} \vdash e : \tau \mid \kappa}{\Gamma \vdash \mathbf{signal} x \mathbf{default} e_1 \mathbf{gather} e_2 \mathbf{in} e : \tau \mid 0; \kappa} \\
\\
\frac{\Gamma \vdash e : (\tau_1, \tau_2) \mathbf{event} \mid \_ \equiv 0 \quad \Gamma \vdash e_1 : \tau \mid \kappa_1 \quad \Gamma \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \mathbf{present} e \mathbf{then} e_1 \mathbf{else} e_2 : \tau \mid \kappa_1 + (\bullet; \kappa_2)} \\
\\
\frac{\Gamma \vdash e_1 : (\tau_1, \tau_2) \mathbf{event} \mid \_ \equiv 0 \quad \Gamma \vdash e_2 : \tau_1 \mid \_ \equiv 0}{\Gamma \vdash \mathbf{emit} e_1 e_2 : \mathbf{unit} \mid 0} \quad \frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \mathbf{loop} e : \mathbf{unit} \mid (0; \kappa)^\infty} \\
\\
\frac{\Gamma \vdash e_1 : \tau \mid \kappa_1 \quad \Gamma \vdash e : (\tau_1, \tau_2) \mathbf{event} \mid \_ \equiv 0 \quad \Gamma, x : \tau_2 \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \mathbf{do} e_1 \mathbf{until} e(x) \rightarrow e_2 : \tau \mid \kappa_1 + (\bullet; \kappa_2)} \\
\\
\frac{\Gamma \vdash e_1 : \tau \mid \kappa \quad \Gamma \vdash e : (\tau_1, \tau_2) \mathbf{event} \mid \_ \equiv 0}{\Gamma \vdash \mathbf{do} e_1 \mathbf{when} e : \tau \mid \kappa + \bullet^\infty} \quad (\text{MASK}) \frac{\Gamma \vdash e : \tau \mid \kappa \quad \phi \notin \mathbf{fbv}(\Gamma, \tau)}{\Gamma \vdash e : \tau \mid \kappa[\phi \leftarrow \bullet]}
\end{array}$$

**Fig. 1.** Type-and-effect rules

- 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  $\kappa'$  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 (used for efficient code generation) performed before typing checks this well-formation of expressions, denoted  $k \vdash e$  in [19] and recalled in Appendix A [20]. Requiring the behavior of some expressions, like the arguments of an application or the body of a function, to be equivalent to 0 does not add any new constraint with respect to this existing analysis.

- 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.
- In the case of `present e then e1 else e2`, 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 with the expression. Similarly, for `do e1 until e(x) → e2`, the expression  $e_2$  is executed at the instant following the presence of  $e$ .
- The reason why the behavior associated with `loop` is equal to  $(0; \kappa)^\infty$  and not simply  $\kappa^\infty$  will be explained in Section 4.5. Intuitively, the soundness proof will use an induction on the size of the behaviours and thus requires the behavior of a sub-expression to always be smaller. This also applies for `signal` and `do/when`.
- Finally, note that there is no special treatment for recursive processes. We will see in the next section that recursive behaviors are introduced during unification.

#### 4.4 Subeffecting with row polymorphism

The typing rule (PROCESS) 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* [23]: 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, it is possible to build a list of two processes with different behaviors (the  $\bullet$  behavior is printed '`*`' and behavior variables  $\phi$  are denoted '`r`'):

```

let process p1 = ()
val p1: unit process[0 + 'r1]
let process p2 = pause
val p2: unit process[* + 'r2]
let l = [p1; p2]
val l: unit process[0 + * + 'r] list

```

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

The consequence of the typing rule for processes is that the principal type of an expression `process e` is always of the form  $\kappa + \phi$  where  $\kappa$  is the behavior of  $e$  and  $\phi$  a free variable. The idea to use a free type variable to represent other possible types is reminiscent of Remy's row types [26]. It makes it possible to implement subeffecting using only unification, without manipulating constraint sets as in traditional approaches [31,3]. 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 [27], so it would be easy to implement our type system on top of the full language.

We can reuse any existing inference algorithm, like algorithm  $\mathcal{W}$  or  $\mathcal{M}$  [16] 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  $[\phi_1 \mapsto \kappa_1; \phi_2 \mapsto \kappa_2; \dots]$ . During unification, the behavior of a process is always either a behavior variable  $\phi$ , a row  $\kappa + \phi$  or a recursive row  $\mu\phi.(\kappa + \phi')$ . Therefore, the unification algorithm only has to consider these cases:

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

It should be noted that unification never fails, so that we obtain a conservative extension of ReactiveML type system. This unification algorithm reuses traditional techniques for handling recursive types [13]. 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* [18]. 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.

#### 4.5 Proof of soundness

We now present the proof sketch of the soundness of our analysis, that is, 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 with reactive effects.

The big-step semantics of ReactiveML, also called the behavioral semantics in reference to the semantics of Esterel [25], describes the reaction of an expression during an instant  $i$  by the smallest signal environment  $S_i$  (the set of present signals) such that:

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

which means that during the instant  $i$ , in the signal environment  $S_i$ , the expression  $e_i$  rewrites to  $e_{i+1}$  and emits the signals in  $E_i$ . The boolean  $b_i$  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. The definition of the semantics was introduced in [19] and is recalled in Appendix B [20].

A program is reactive if at each instant, the semantics derivation of  $e_i$  is finite. To prove that, we first isolate the first instant of the behavior of  $e_i$ , noted  $\text{fst}(\kappa_i)$ . This function is formally defined by:

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

The important property of this function is that if a behavior  $\kappa_i$  of  $e_i$  is reactive (as defined in Section 3.2), then  $\mathbf{fst}(\kappa_i)$  is finite. Hence we can prove by induction on the size of  $\mathbf{fst}(\kappa_i)$  that the semantics derivation is finite. The soundness theorem is stated as follows:

**Theorem 1 (Soundness).** *If  $\Gamma \vdash e : \tau \mid \kappa$  and  $\tau$  and  $\kappa$  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  $\kappa'$  reactive.*

*Proof.* The proof has two parts. The first part is the proof that the result is well-typed. We use classic syntactic techniques for type soundness [24] on the small-step semantics described in [19]. The proof of equivalence of the two semantics is also given in the same paper. The second part is the proof that the semantics derivation of one instant is finite by induction on the size of the first-instant behavior of well-typed expressions. We only consider one instant because thanks to type preservation, if the theorem is true for one instant, it is true for the following ones. The details of the proof are given in Appendix C [20].  $\square$

## 5 Examples

We present here the result of the analysis on some examples. These examples can be downloaded and tried at <http://reactiveml.org/sas14>.

Using a type-based analysis makes it easy to deal with cases of aliasing, as in the following example:<sup>7</sup>

```

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 [27], row variables that appear only once are printed `'..'`.

The analysis can also handle combinators like the `par_comb` and `if_comb` examples of Section 3.1. 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:

<sup>7</sup> Some behaviors are simplified using the extension described in Appendix D [20].

```

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 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:

```

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; 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

```

In the example, the analysis detects the problem of reactivity although there is no explicit recursive process.

## 6 Discussion

**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.

In practice, the analysis has an impact on the type-checking time but it is negligible compared to the global compilation time. For example on a 1.7GHz Intel Core i5 with 4Go RAM, the compilation of the examples of the ReactiveML distribution (about 5000 lines of code) takes about 0.15s where 0.02s are spent in the type checker (0.005s without the reactivity analysis). Then it takes 3.5s to compile the generated OCaml code.

**Evaluation** The analysis is very useful to detect early small reactivity bugs such as the one presented Section 2.1. We have also used the analysis on bigger applications: a mobile ad-hoc network simulator (1800 Source Lines Of Code), a sensor network simulator (1700 SLOC), and a mixed music sequencer (3400 SLOC).

There is no warning for both simulators. For the mixed music sequencer, there are warnings on eleven processes. Eight warnings are due to structural recursions (similar to the example `par_map`). Most of them come from the fact that the program is a language interpreter defined as a set of mutually recursive processes on the abstract syntax tree. Another warning comes from a record containing a process that is not annotated with a non-instantaneous behavior. The last two warnings are due to loops around the execution of a musical score. In this case, the analysis does not use the fact that the score is a non-empty list.

In all these cases, it was easy for the programmer to check that these warnings were false positives. The last three warnings can be removed by adding a `pause` in parallel to the potentially instantaneous expressions.

***Other models of concurrency*** We have already extended our analysis to take into account time refinement [21]. 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`. We are considering an extension to the X10 language,<sup>8</sup> where cooperation points could be clocks.

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 [15]. The distinction between processes and functions is important to avoid showing a warning for all recursive functions.

## 7 Related work

The analysis of instantaneous loops is an old topic on which much has already been written, even recently [1,14,4]. It is related to the analysis of productivity and deadlocks in list programs [29] 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 [28] language, which extends ML with message passing primitives. The behavior of a process records emissions and receptions 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 precedes the emission on a second channel in a given program. The idea of using a type-and-effect

---

<sup>8</sup> <http://x10-lang.org/>

system for checking reactivity or termination is not new. For instance, Boudol [8] 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 [25], 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* [25]. Our concurrency model, inherited from the work of Boussinot [9], 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 [25]. It is easy to check that an Esterel program is loop-safe as the language is first order without recursion [32].

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 [7], 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 defined by Leroy et al. [17]. Both are a more direct application of row types [26], 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 with reactive effects 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. We believe that this work can be applied to other models of cooperative scheduling.

## Acknowledgments

This work would not have been possible without previous experiments made with Florence Plateau and Marc Pouzet. Timothy Bourke helped us a lot in the preparation of this article. We are grateful for the proofreading and discussions with Guillaume Baudart and Adrien Guatto. Finally, we also thank the reviewers for there numerous comments and suggestions.

## References

1. Abel, A., Pientka, B.: Well-founded recursion with copatterns. In: International Conference on Functional Programming. (2013)
2. Amadio, R., Dabrowski, F.: Feasible reactivity in a synchronous  $\pi$ -calculus. In: Principles and Practice of Declarative Programming. (2007) 221–230
3. Amtoft, T., Nielson, F., Nielson, H.: Type and Effect Systems: Behaviours for Concurrency. Imperial College Press (1999)
4. Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: International Conference on Functional Programming. (2013)
5. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. *Mathematical Structures in Computer Science* **14**(01) (2004) 97–141
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., De Simone, R.: The synchronous languages twelve years later. In: Proc. of the IEEE. (2003)
7. Boudol, G.: ULM: A core programming model for global computing. In: European Symposium on Programming. (2004)
8. Boudol, G.: Typing termination in a higher-order concurrent imperative language. *Information and Computation* **208**(6) (2010) 716–736
9. Boussinot, F.: Reactive C: an extension of C to program reactive systems. *Software: Practice and Experience* **21**(4) (1991) 401–428
10. Cuoq, P., Pouzet, M.: Modular Causality in a Synchronous Stream Language. In: European Symposium on Programming. (2001)
11. Dimino, J.: Lwt User Manual. (2014) <http://ocsigen.org/lwt/>.
12. Engelschall, R.: Portable multithreading: The signal stack trick for user-space thread creation. In: USENIX Annual Technical Conference. (2000)
13. Huet, G.: A unification algorithm for typed  $\lambda$ -calculus. *Theoretical Computer Science* **1**(1) (1975) 27–57
14. Jeffrey, A.: Functional reactive programming with liveness guarantees. In: International Conference on Functional Programming. (2013)
15. Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: Principles of Programming Languages. (1996) 295–308
16. Lee, O., Yi, K.: Proofs about a folklore let-polymorphic type inference algorithm. *Transactions on Programming Languages and Systems* **20**(4) (1998) 707–723
17. Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. *Transactions on Programming Languages and Systems* **22**(2) (2000) 340–377
18. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Principles of Programming Languages. (1988)
19. Mandel, L., Pouzet, M.: ReactiveML: a reactive extension to ML. In: Principles and Practice of Declarative Programming. (2005)
20. Mandel, L., Pasteur, C.: Reactivity of cooperative systems – extended version. Research Report 8549, INRIA (2014) Available at: <http://reactiveml.org/sas14>.
21. Mandel, L., Pasteur, C., Pouzet, M.: Time refinement in a functional synchronous language. In: Principles and Practice of Declarative Programming. (2013)
22. Marlow, S., Jones, S., Thaller, W.: Extending the Haskell foreign function interface with concurrency. In: Haskell’04, ACM (2004) 22–32
23. Nielson, F., Nielson, H.: Type and effect systems. *Correct System Design* (1999)
24. Pierce, B.: Types and programming languages. The MIT Press (2002)
25. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel. Springer (2007)

26. Rémy, D.: Type inference for records in a natural extension of ML. *Theoretical Aspects of Object-Oriented Programming*. MIT Press (1993)
27. Rémy, D.: Using, understanding, and unraveling the OCaml language from practice to theory and vice versa. In: *Applied Semantics*. Springer (2002) 413–536
28. Reppy, J.: *Concurrent programming in ML*. Cambridge University Press (2007)
29. Sijtsma, B.A.: On the productivity of recursive list definitions. *Transactions on Programming Languages and Systems* **11**(4) (1989) 633–649
30. Syme, D., Petricek, T., Lomov, D.: The F# asynchronous programming model. *Practical Aspects of Declarative Languages* (2011) 175–189
31. Talpin, J.P., Jouvelot, P.: The type and effect discipline. In: *Logic in Computer Science*. (1992)
32. Tardieu, O., de Simone, R.: Loops in Esterel. *Transaction on Embedded Computing* **4**(4) (2005) 708–750
33. Tofte, M.: Type inference for polymorphic references. *Information and computation* **89**(1) (1990) 1–34
34. Vouillon, J.: Lwt: a cooperative thread library. In: *ACM workshop on ML*. (2008)