Time Refinement in a Functional Synchronous Language

Louis Mandel[a], Cédric Pasteur[b,c], Marc Pouzet[e,b,d]

[a] *IBM Research, Yorktown Heights, NY, USA*
[b] *DI, École normale supérieure, Paris, France*
[c] *now at ANSYS-Esterel Technologies, Toulouse, France*
[d] *INRIA Paris-Rocquencourt, France*
[e] *Université Pierre et Marie Curie, Paris, France*

Abstract

Concurrent and reactive systems often exhibit multiple time scales. This situation occurs, for instance, in the discrete simulation of a sensor network where the time scale at which agents communicate is very different from the time scale used to model the internals of an agent.

The paper presents *reactive domains* to simplify the programming of such systems. Reactive domains allow for several time scales to be defined and they enable *time refinement*, that is, the replacement of a system with a more detailed version, without changing its observed behavior.

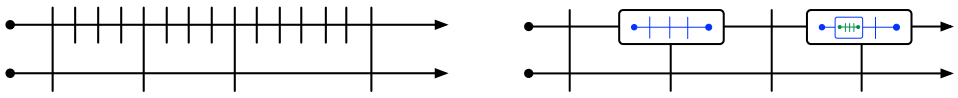
Our work applies to the ReactiveML language, which extends an ML language with synchronous programming constructs *à la Esterel*. We present an operational semantics for the extended language, a type system that ensures the soundness of programs, and a sequential implementation. We discuss how reactive domains can be used in a parallel implementation.

Keywords: Synchronous languages; Functional languages; Semantics; Type systems

1. Introduction

The concept of logical time greatly simplifies the programming of *concurrent* and *reactive* systems. It is the basis of synchronous languages [1] like Esterel [2]. Its principle is to see the execution of a reactive system as a sequence of reactions, called *instants*, where all communications and computations are considered to be instantaneous during one reaction. This

Email addresses: lmandel@us.ibm.com (Louis Mandel), cedric.pasteur@ansys.com (Cédric Pasteur), marc.pouzet@ens.fr (Marc Pouzet)



(a) Sampling (b) Reactive domains

Figure 1: Sampling vs. Reactive domains (each vertical line or box represent one instant of the corresponding clock, horizontal lines represent processes running in parallel)

interpretation of time is logical because it does not account for exact computation time and the precise way all the computations are done during a reaction. It has been originaly introduced for programming real-time embedded controllers, but it is applicable for a wider range of applications, in particular large scale simulations.

Consider, for example, the simulation of the power consumption in a sensor network [3]. In order to precisely estimate the power consumption, we need to simulate the hardware of certain nodes, in particular the radio. There are now multiple time scales: for example, the time scale of the software (i.e., MAC protocol) is in milliseconds, while the time step of the hardware would be in microseconds. The communication between these time scales must be restricted. E.g., a slow process, whose time scale is in millisecond, cannot observe all the changes of a faster process, whose scale is in microseconds. Said differently, a signal that is produced by a fast process cannot be used to communicate a value with a slower process. Furthermore, depending on the level of precision required for the simulation, it makes sense to be able to replace a precise but costly version of a process that may last several steps by an approximated version, possibly instantaneous. Moreover, this replacement should not impact the way the process interacts with other processes. Such a feature has been called *time* or *temporal refinement* [4].

Synchronous data-flow languages provide a solution to this problem that is based on *sampling*. A slower time scale is obtained by choosing a subset of instants according to a boolean condition. In this paper, we propose *reactive domains*, that allow doing the opposite. Instead of creating a new time scale which is slower than an other one, a reactive domain creates a faster time scale by subdividing an instant of the parent domain. The sequence of instants of a reactive domain stay local to it, that is, they are un-observable from outside, as shown in Figure 1. Reactive domains make time refinement easy as they allow local computation steps to be hidden (Section 3).

Our work is applied to the ReactiveML language [5], which augments ML with synchronous programming constructs *à la Esterel* (Section 2).[1] We show how to extend the operational semantics of the language to incorporate reactive domains (Section 4). The soundness of programs in the extended setting can be checked using a type-and-effect system, called a *clock calculus*, since it is reminiscent of the one in data-flow synchronous languages [1] (Section 5). Yet, the clock calculus of ReactiveML applies to a language where synchronous constructs are those of Esterel and with ML features. Then, we prove the soundness of the type system with respect to the semantics (Section 6). We also give an overview of the implementation of the extended language and some ideas for parallel execution (Section 7). The article ends with a discussion of several extensions (Section 8) and related work (Section 9).

2. The ReactiveML Language

ReactiveML[2] [5, 6] is based on the *reactive model* of Boussinot which first appeared in the ReactiveC language [7]. The *reactive model* applies to general purpose languages the principles of the *synchronous model* found in synchronous languages [1].

2.1. Examples

ReactiveML is a reactive extension of ML, so any ML program is also a valid ReactiveML program. The concrete syntax of the language is the one of OCaml,[3] upon which ReactiveML is built. For example, we can define a tree data type and the preorder iteration of a function on a tree by:

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree

let rec preorder f t = match t with
| Empty -> ()
| Node(l, v, r) -> f v; preorder f l; preorder f r
```

The type of trees, `'a tree`, is parametrized by the type `'a` of its labels. A tree is either empty, or made of a left child, a label and a right child. The

[1] The compiler and the examples mentioned in the paper are available at http://reactiveml.org/scp15

[2] http://www.reactiveml.org

[3] http://ocaml.org

preorder traversal of the tree is implemented with a simple recursive function that applies a given function to the label and recurses first on the left child and then on the right one. We can almost as easily define the level-order traversal of the tree in ReactiveML:

```
let rec process levelorder f t = match t with
| Empty -> ()
| Node (l, v, r) ->
    f v; pause;
    (run levelorder f l || run levelorder f r)
```

This example defines a recursive *process* named `levelorder`. Unlike regular ML expressions, such as a call to `preorder f t`, which is said to be *instantaneous*, the execution of a process can last several logical instants. Here, `levelorder` awaits the next instant by using the `pause` operator and then recursively calls itself on the left and right children in parallel. The `||` operator denotes logical parallel composition. The `run` operator is used to launch a process. As all processes share the same time scale, the two instances of the `levelorder` process synchronize on the next `pause`. As a consequence, `f` is applied to all the labels at the same depth during a given instant. It should be noted that the order in which parallel processes are executed is unspecified.

Processes running in parallel can communicate using *broadcast signals*: when a signal is emitted, it is seen by all the processes that observe it. Moreover, all the processes have a consistent view of a signal at every instant. It is either present or absent, henceforth unable to change for the rest of the instant. Running the following process prints `"Hello world"` at the first instant. The first branch of the parallel awaits for the presence of a signal `go` and the second branch emits `go`. As the is signal is emitted, it is thus present and the first branch of the parallel immediately executes the expression following the waiting of the presence of the signal.

```
let process hello_world =
 signal go in
 await immediate go; print_string "Hello world"
 ||
 emit go
```

A signal may also carry a value. Several processes can emit different values on a signal during the same instant. This is termed *multi-emission*. These values are combined using a function given in the definition of the signal. The value of a signal at a given instant is obtained by folding this function across emitted values, starting from a default value. In the following example, the

value of the signal `s` is the sum of the emitted values:

```
let process sig_gather =
  signal s default 0 gather (+) in
  emit s 2 || emit s 4
  || await s(v) in print_int v
```

This process prints `6` (i.e. `0 + 2 + 4`) on the second instant. Indeed, the values `2` and `4` are emitted on `s` during the first instant. In order to be sure that all the values of `s` has been emitted (and that no other value will be emitted later during the same instant), trying to access the value of a signal in delayed by one instant. `await s(v) in ...` intuitively means "await that all the values of `s` have been emitted, then execute the continuation". One can react immediately to the presence of a signal, as illustrated in the previous example, but it takes one instant to read its value.

Finally, a signal also stores its last value, that is the value that was carried at the previous reaction. If a signal `s` is declared with a default value, `last s` is this value until the first instant where `s` is emitted. `last s` can be used, for example, to maintain a value across several instants:

```
let process hold s =
  loop emit s (last s); pause end
```

2.2. Programming Agents in ReactiveML

Figure 2a shows an example of a node in a simulation of a sensor network made of small low-cost sensors that collect and communicate environmental data. A node receives messages on the signal `me` (line 10), decrements them and then forwards them to all of its neighbors (line 6) (`iter` iterates a process on all the elements of a list). The second part of the node (lines 13 to 18) models the energy consumption: the energy of the node is decremented by `max_power` at each time step, where each step corresponds to one millisecond of simulation time. The node terminates when its energy crosses the `e_min` threshold. This is achieved by using preemption through the `do/until` control structure. Indeed, `do e until dead done` executes the body `e` until the emission of the signal `dead`, then terminates in the instant following the emission.

Another simple example of simulation is the n-body problem, solved using a fixed-step numerical integration in Figure 2b. The idea is to use a global signal `env`, whose value is a force field, that is, a function mapping a position to a force. Each body, characterized by its current position, velocity and weight, is a process that, at each instant, sends its attraction by emitting on `env`, receives the sum of all the forces emitted by other bodies, and uses this

```
let process node me neighbors =
  signal dead in
  signal energy default e_0 gather (fun x _ -> x) in
  let process send msg n = emit n msg in
  let process forward_msg msg =
    if msg>1 then run iter (send (msg-1)) neighbors
  in
  do
    loop (* protocol *)
      await me(msgs) in run iter forward_msg msgs
    end
    ||
    loop (* power *)
      if last energy < e_min
      then emit dead
      else emit energy (last energy -. max_power);
      pause
    end
  until dead done
```

(a) A simple node in a sensor network

```
let dt = 0.01
signal env default (fun _ -> zero_vector) gather add_force

let rec process body (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f) in
  (* euler semi-implicit method *)
  let v_tp = v_t ++. (dt **. (f x_t)) in
  let x_tp = x_t ++. (dt **. v_tp) in
  run body (x_tp, v_tp, w)

let process main =
  for i = 1 to 100 dopar
    run body (random_planet ())
  done
```

(b) The n-body problem (++. and **. are operations on vectors)

Figure 2: Two examples

force to compute its position `dt` later. The main process is made of several bodies running in parallel using a parallel for loop.

The basic constructs of ReactiveML have been introduced. We present now the proposed extension.

3. Reactive Domains

A reactive domain introduces a notion of local instants, that is, instants that are unobservable from the outside. This notion can be seen as a reification of the execution engine attached to any ReactiveML program. A reactive domain behaves as if its body was executed by a separate execution engine, with its own notion of step.

3.1. Reactive Domains and Clocks

A reactive domain is declared by the keyword **`domain`**:

```
domain ck do e done
```

The name `ck` is the identifier of the domain, that we call a *clock*. It is bound to the expression `e`, which is the body of the domain. The clock defines a sequence of instants. That is why the **`pause`** operator now takes as argument a clock: **`pause`** `ck` waits for the next instant of the domain of clock `ck`. For instance, the following process prints `"Hello "` during the first instant of the clock `ck` and `"world"` during the second instant of `ck`.

```
let process hello_world_ck =
  domain ck do
    print_string "Hello "; pause ck;
    print_string "world"
  done
```

`ck` are included in the first instant of the global clock, called `global_ck`.[4] The two local instants of `ck` can only be observed by processes inside the reactive domain. Thus `hello_world_ck` is equivalent to:

```
let process hello_world_seq =
  print_string "Hello ";
  print_string "world"
```

as if the synchronization on the local clock `ck` was erased.

[4] `global_ck` is a global variable which is the global clock of the program.

Reactive domains form a tree, called the *clock tree*, where one reactive domain is a child of another if it is defined in the latter's scope. A clock `ck'` is said to be faster than `ck` if `ck'` is a descendant of `ck` in the clock tree. The global clock `global_ck` is the root of the clock tree, and is thus slower than any other clock.

While the process `hello_world_ck` terminates instantaneously, it is possible for the execution of a reactive domain to span several instants of its parent domain. It is then necessary to relate the instants of the reactive domain to those of its parent, that is, to know how many steps of the reactive domain should be taken in each step of the parent reactive domain. The simplest form of relation is given by periodic reactive domain. A periodic reactive domain performs n local instants per instant of its parent reactive domain, using the keyword `by`:

```
let process stutter msg =
  domain ck by 6 do
    loop print_string msg; pause ck end
  done
```

The expression `run stutter "a"` prints six `a`'s at each instant of the global clock. Instants of sibling reactive domains are unrelated. For instance, `run stutter "a" || run stutter "b"` prints six `a`'s and six `b`'s at each instant in an unspecified order.

Reactive domains can be created dynamically and nested arbitrarily. For instance, the `stutter` process can be rewritten as follows:

```
let process stutter_nested msg =
  domain ck1 by 3 do
    domain ck2 by 2 do
      loop print_string msg; pause ck2 end
    done
  done
```

3.2. Reactive Domains and Signals

The consequence of introducing reactive domains is that every signal is now attached to a reactive domain, that is, it has one value for every instant instant of the domain. This explains why we use the term clock for a domain's identifier. Indeed, in synchronous data-flow languages, a clock for a signal defines the instant where the signal is present [1]. The semantics of a signal defined inside a reactive domain is unchanged. For instance, if we run one of the examples of Section 2.1 inside a reactive domain, the result is the same:

`sig_gather_ck` prints `6` during the second instant of `ck`, but during the first instant of the global clock:

```
let process sig_gather_ck =
  domain ck do run sig_gather done
```

Emitting a value on a signal with a slower clock (that we will call a slow signal) is not an issue thanks to multi-emission: all the values emitted during the instant of the signal's clock, including the multiple instants of child reactive domains, are gathered to compute the value for the instant. It is also possible to await the emission of a slow signal. The continuation of the `await` will occur in the next instant of the emitted signal's clock, as in the following process:

```
let process slow_signal =
  signal s default 0 gather (+) in
  domain ck by 3 do
    await s(v) in print_int v
  done
  ||
  emit s 4
```

A `4` is printed during the second instant of the global clock, as if there were no reactive domain, but during the fourth instant of clock `ck` because there where three instants of `ck` during the first instant of the global clock. The result would have been the same if the `emit` statement had been inside the reactive domain. Sibling reactive domains can thus communicate using signals attached to a common ancestor in the clock tree.

While in ReactiveML, any process can use any signal, reactive domains impose restrictions. First, as the instants of a reactive domain are unobservable from the outside, it does not make sense to access a signal attached to a reactive domain from outside of that domain where the different values of the signal cannot be distinguished. The second restriction is that it is forbidden to react immediately to the presence of a slow signal.

We illustrate the problem with the following process which is rejected by the compiler:

```
let process immediate_dep_wrong =
  signal s in
  domain ck do
    await immediate s; print_string "Ok"
    ||
    pause ck; emit s
  done
```

In this example, `s` has clock `global_ck`. During the first instant of `ck`, we suppose that `s` is not present, so the first branch of the parallel is blocked. But in the second instant of `ck` —yet still in the first instant of `global_ck`— `s` is emitted, which should trigger the printing of `Ok` because of the immediate dependency. We reject this process because it makes two different assumptions about the presence of `s` during the same instant of the clock of `s`, which goes against the principle that all processes have the same view of a signal's status and value at an instant. The type system defined in Section 5 ensures that these two kinds of errors never occur.

3.3. Automatic waiting of reactive domains

Consider the following process:

```
let process delayed_hello_world =
  signal s default "" gather (^) in
  domain ck by 10 do
    pause global_ck; emit s "Hello world"
    ||
    await s(v) in print_string v
  done
```

At the end of the first instant of `ck`, the first branch of the parallel is waiting for the next instant of `global_ck`. The second branch is waiting for the signal `s` of clock `global_ck`. If the reactive domain were executing another local instant, its body would not evolve. It is not necessary to execute ten local instants: the reactive domain can directly wait for the next instant of `global_ck` before doing its next local instant. We can thus interpret the number given after **`by`** as a bound on the number of instants that a reactive domain can do.

We could just treat this property as a run-time optimization, but we believe it can be usefully incorporated in the semantics of the language so as to accept more programs. Indeed, in most cases, as in the previous example, it is clear that the body of the reactive domain will be blocked waiting for a slower clock at some point. It is thus permitted to omit the bound (as was done in the first examples). A reactive domain then not only decides when its local instants are finished, but also when to wait for the next instant of its parent clock. It does so automatically if all the processes it contains are waiting for the next instant of a slower clock, either via an explicit **`pause`** or by waiting for a signal with a slower clock.

3.4. Clocks and Reactivity

Once we allow reactive domains to perform an unbounded number of instants per instant of the parent clock, it becomes possible for a reactive domain to be non-reactive, that is, to never wait for the next instant of the parent clock, as in the following example:

```
let process nonreactive_domain =
  domain ck do
    loop pause ck end
  done
```

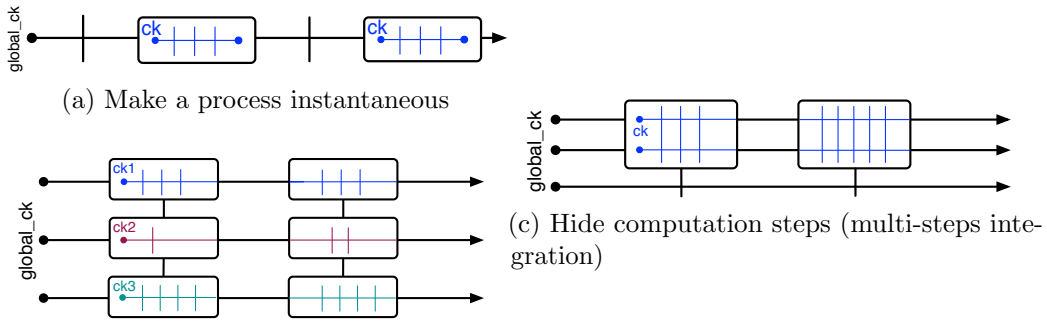
This reactive domain never waits for the next instant of its parent clock and behaves like an instantaneous infinite loop. A static analysis, based on the extension of [8], warns the programmer that this program is problematic. But, it means that it is no longer possible to use the `hold` process from Section 2 to hold the value of a signal inside an unbounded reactive domain. To solve this problem, we introduce a variant of the **`pause`** operator denoted **`quiet pause`**. A call to **`quiet pause`** `ck` terminates in the next instant of `ck`, like for **`pause`** `ck`, but if all the other parallel processes in the reactive domain are waiting for a slower clock, the reactive domain synchronizes with the parent clock (another local instant is not performed in the same instant). Using this operator, it is possible to sustain a signal inside an unbounded reactive domain:[5]

```
let process quiet_sustain_v s ck =
  loop emit s (last s); quiet pause ck end

let process hold_domain =
  domain ck do
    signal s default 0 gather (+) in
    run quiet_sustain_v s ck
    ||
    pause ck; emit s 3; pause global_ck; print_int (last s)
  done
```

At the end of the first instant of `ck`, the second branch of the parallel asks for another instant of `ck` by calling **`pause`** `ck`. During the next instant, it is stuck waiting for the next instant of the global clock. As the first branch of the parallel uses the **`quiet pause`** operator, it does not influence the choice of the reactive domain of clock `ck`, which then awaits the next instant of the global clock before executing its next local instant and printing the last value of `s`.

[5] You can notice in this example that clocks are first class citizens and thus can be function arguments.



(a) Make a process instantaneous

(b) Agents: one reactive domain per agent

(c) Hide computation steps (multi-steps integration)

Figure 3: Several patterns of programming with reactive domains

3.5. Using Reactive Domains

Reactive domains are useful for several typical patterns. The first is to make a process instantaneous. For instance, one can hide the internal steps used in the `levelorder` example of Section 2.1 (**`pause`** without any argument waits for the next instant of the local clock, that can also be obtained using the `local_ck` operator):

```
let process levelorder_inst f t =
  domain ck do
    run levelorder f t
  done
```

Figure 3a illustrates the behavior of the reactive domain: it hides all internal steps and behaves as an instantaneous process on the global clock. This process could not have been written without the automatic waiting of reactive domains (Section 3.3) as it executes an unbounded number of local instants, equal to the depth of the tree.

The second pattern is for programming agent-based simulations. Reactive domains allow each agent to perform an arbitrary number of internal steps during each step of the simulation, that corresponds to one instant of the global clock. One simply has to declare one reactive domain per agent, as in Figure 3b. Agents only synchronize at the end of the instant of the global clock. Signals for communication between agents remain attached to the global reactive domain, and are thus buffered automatically.

We can use this idea to better simulate the power consumption of the node from Figure 2a, by modeling the fact that power consumption is related to the number of messages sent. An abbreviated version of the resulting program

```
let process node_with_energy me neighbors =
 domain us by 1000 do
  signal dead in
  signal energy default e_0 gather (fun x _ -> x) in
  signal power default 0.0 gather (+.) in
  signal r_in default (0,me) gather (fun x _ -> x) in
  signal r_ack in
  let process send msg n =
    emit r_in (msg, n);
    await immediate r_ack
  in
  ...
  do
    ... (* protocol *) ||
    loop (* radio *)
      await r_in (msg, n) in
      for i=1 to packet_send_time do
        emit power send_power; pause us
      done;
      emit n msg; emit r_ack
    end
    ||
    loop  (* power *)
      emit power on_power;
      if last energy < e_min
      then emit dead
      else emit energy (last energy -. (last power /. 1000.0));
      pause us
    end
  until dead done
 done
```

Figure 4: A node with refined power consumption

```
let rec process body_heun env (x_t, v_t, w) =
  emit env (force (x_t, w));
  await env(f_t) in
  (* step 1 *)
  let f_t = f_t x_t in
  let v_int = v_t ++. (dt **. f_t) in
  let x_int = x_t ++. (dt **. v_t) in
  (* step 2 *)
  emit env (force (x_int, w));
  await env(f_int) in
  let f_int = f_int x_int in
  let v_tp = v_t ++. ((dt /. 2.0) **. (f_t ++. f_int)) in
  let x_tp = x_t ++. ((dt /. 2.0) **. (v_t ++. v_int)) in
  (* next step*)
  pause global_ck;
  run body_heun env (x_tp, v_tp, w)

let process main =
  domain computation_ck do
    signal env default (fun _ -> zero_vector) gather add_force in
    for i = 1 to 100 dopar
      run body_heun env (random_planet ())
    done
  done
```

Figure 5: Two-step integration method (Heun's method)

is shown in Figure 4. The idea is to use a reactive domain to introduce a new local time scale, corresponding to microseconds of simulation time. The radio is represented by a process receiving a message to be sent and a destination on the `r_in` signal. The sending of the message is modeled by waiting `packet_send_time` microseconds, during which the power consumption is raised by `send_power`. After that, the radio actually sends the message to the destination and acknowledges the sending on the `r_ack` signal.

A similar use is to hide computation steps shared by many agents. The fast clock is then shared by several processes as in Figure 3c, whereas in Figure 3b each process has its own local clock. An example of this pattern is an extension of the n-body simulation of Figure 2b to use multi-steps integration methods, here Heun's method. The resulting code is shown in Figure 5. Each step of the computation corresponds to one instant of a reactive domain, shared by all bodies. As these instants are unobservable from the outside, it is easy to add processes such as the GUI on the global clock (last line in Figure 3c) or to dynamically switch methods (e.g. from a two-steps to a four-steps method) without any influence on the rest of the program.

3.6. A Modularity Issue

We have seen that some communications take time because of multi-emission. This can lead to modularity problems, as we will see on a few examples, and makes it even more necessary to be able to hide local instants. Let's first define a higher-order process `lift` that turns a function on values into a function on streams (like the `arr` combinator in FRP [9]). It awaits a new value on a signal `s_in`, applies `f` to it and emits the result on another signal `s_out`:

```
let process lift f s_in s_out =
  loop
    await s_in(v) in
    emit s_out (f v)
  end
```

We can now define a process `fg1` that applies the composition of two functions `g` and `f`:

```
let process fg1 s_in s_out =
  run lift (fun v -> f (g v)) s_in s_out
```

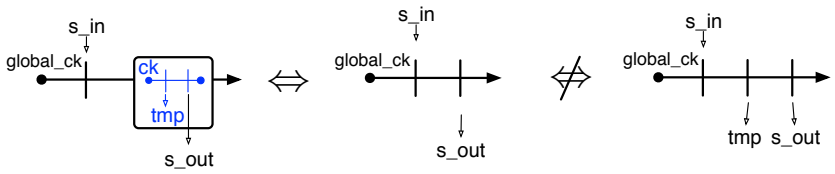


Figure 6: Fixing a modularity problem with reactive domains (left: `fg2_good`, center: `fg1`, right: `fg2`)

Suppose that, for modularity reasons, we want to separate the computations of `f` and `g`. We use a local signal `tmp` to communicate between the two processes:

```
let process fg2 s_in s_out =
  signal tmp default 0 gather (+) in
  run lift f s_in tmp || run lift g tmp s_out
```

The problem is that, while `fg1` emits the result one instant after the emission of a value on `s_in`, it takes two instants for `fg2` to do the same. We can fix this problem by running the process inside a reactive domain:

```
let process fg2_good s_in s_out =
  domain ck do
    run fg2 s_in s_out
  done
```

The `fg2_good` process has the same behavior as the `fg1` process: it takes two instants of the local clock `ck` to compute the result, but only one on the global clock. Figure 6 illustrates the behavior of these three processes.

4. Operational Semantics

In this section, we extend the ReactiveML operational semantics [5] to support reactive domains. It is itself an extension of the small-step reduction semantics of ML.

4.1. Language Abstract Syntax

We present the semantics on a core language, based on a call-by-value functional kernel extended with synchronous primitives: defining and running a process, waiting for the next instant of a clock (`pause`), quietly waiting for the next instant (`quiet pause`), a parallel `let`, declaring a signal, emitting a value, getting its last value, preemption (`do`/`until`) and suspension (`do`/`when`) control structures, declaring a reactive domain and accessing the local clock (`local_ck`):

$$
\begin{aligned}
e ::= {} & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \\
& \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause}\, e \mid \texttt{quiet pause}\ e \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \\
& \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \mid \texttt{emit}\ e\ e \mid \texttt{last}\ e \\
& \mid \texttt{do}\ e\ \texttt{until}\ e(x) \to e \mid \texttt{do}\ e\ \texttt{when}\ e \\
& \mid \texttt{domain}\ x\ \texttt{by}\ e\ \texttt{do}\ e \mid \texttt{local\_ck}
\end{aligned}
$$

The expression $\texttt{do}\ e\ \texttt{when}\ s$ executes its body only when s is present. We denote by _ variables that do not appear free in the body of a `let` and by () the unique value of type `unit`. Among others, it is possible to derive the following constructs from this kernel:

$$
\begin{aligned}
e_1 \,||\, e_2 &\triangleq \texttt{let}\ \_ = e_1\ \texttt{and}\ \_ = e_2\ \texttt{in}\ () \\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 &\triangleq \texttt{let}\ x = e_1\ \texttt{and}\ \_ = ()\ \texttt{in}\ e_2 \\
e_1; e_2 &\triangleq \texttt{let}\ \_ = e_1\ \texttt{in}\ e_2 \\
\texttt{domain}\ x\ \texttt{do}\ e &\triangleq \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e \\
\texttt{loop}\ e &\triangleq\ \texttt{run}\ ((\texttt{rec}\ \mathit{loop} = \\
&\qquad \lambda x.\texttt{process}\ (\texttt{run}\ x; \texttt{run}\ (\mathit{loop}\ x)))\ (\texttt{process}\ e)) \\
\texttt{signal}\ s\ \texttt{in}\ e &\triangleq \texttt{signal}\ s\ \texttt{default}\ []\ \texttt{gather}\ (\lambda x.\lambda y.x :: y)\ \texttt{in}\ e \\
\texttt{emit}\ e\ &\triangleq \texttt{emit}\ e\ () \\
\texttt{pause}\ &\triangleq \texttt{pause}\,\texttt{local\_ck} \\
\texttt{await immediate}\ e &\triangleq \texttt{do}\ ()\ \texttt{when}\ e \\
\texttt{await}\ e_1(x)\ \texttt{in}\ e_2 &\triangleq \texttt{do}\ (\texttt{loop}\ (\texttt{quiet pause local\_ck}))\ \texttt{until}\ e_1(x) \to e_2
\end{aligned}
$$

It should be noted that $\texttt{await}\ e_1(x)\ \texttt{in}\ e_2$ is encoded as the preemption of an infinite loop, but it is important to use the `quiet pause` operator so that it does not make the local reactive domain make an infinite number of instants.

Values are the regular ML values (constants, pairs and functions), plus processes, signal names indexed by their clock and clock names:

$$
v ::= c \mid (v, v) \mid \lambda x.e \mid \texttt{process}\ e \mid n^{ck} \mid ck \qquad \text{(values)}
$$

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

$$\frac{0 \vdash e}{k \vdash \texttt{rec}\ x = e} \qquad \frac{1 \vdash e}{k \vdash \texttt{process}\ e} \qquad \frac{0 \vdash e}{1 \vdash \texttt{run}\ e} \qquad \frac{0 \vdash e}{1 \vdash \texttt{pause}\ e}$$

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

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

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

$$\frac{0 \vdash e_1 \qquad 1 \vdash e}{1 \vdash \texttt{domain}\ x\ \texttt{by}\ e\ \texttt{do}\ e_1} \qquad \frac{1 \vdash e}{1 \vdash e\ \texttt{in}\ ck'} \qquad k \vdash \texttt{local\_ck}$$

Figure 7: Well-formation rules

Finally, we add e `in` ck to the expressions of the language. It represents the result of instantiating the expression `domain` x `by` e `do` e and cannot itself be used directly in a program.

$$\begin{array}{lll} e ::= & \ldots & \\ & \mid e\ \texttt{in}\ ck & \text{(executing reactive domain)} \end{array}$$

4.2. Well-formation of expressions

A simple syntactic classification, called well-formation predicate and denoted $k \vdash e$ with $k \in \{0, 1\}$, is used to distinguish instantaneous and non-instantaneous expressions. It was introduced in [5]. It allows for example to separate pure ML expressions from reactive expressions, which is useful for code generation. It is also used in Section 6 for the proof of soundness. The rules defining this predicate are given in Figure 7.

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 predicate $k \vdash e$ 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. The design choices of this analysis, like the fact that pairs must be instantaneous, are discussed in [5].

4.3. Notations

$\mathcal{C}$ is a denumerable set of clock names, denoted ck. The global clock is denoted $\top_{ck} \in \mathcal{C}$. $\mathcal{N}$ is a denumerable set of signal names, denoted n. A *local signal environment* is a partial mapping from signal names to tuples (d, g, l, m) where d and g are the default value and gather function, l the last value and m the multiset of values emitted at an instant. A *signal environment* $\mathcal{S}$ is a partial mapping from clock names to local signal environments. If $\mathcal{S}(n^{ck}) = \mathcal{S}(ck)(n) = (d, g, l, m)$, we write $\mathcal{S}^d(n^{ck}) = d$ (similarly for the g, l and m) and $\mathcal{S}^v(n^{ck}) = \text{fold } g\ d\ m$ if $m \neq \emptyset$ ($\mathcal{S}^v(n^{ck})$ is not defined otherwise). We write $n^{ck} \in \mathcal{S}$ when n is present, that is, $\mathcal{S}^m(n^{ck}) \neq \emptyset$, and $n^{ck} \notin \mathcal{S}$ otherwise. We denote by $S + [v/n^{ck}]$ the environment where v is added to the multiset $\mathcal{S}^m(n^{ck})$ and by $\texttt{next}(\mathcal{S}, ck)$ the environment where the last value of any signal with clock ck is set to its current value $\mathcal{S}^v(n^{ck})$ (if defined) and $\mathcal{S}^m(n^{ck})$ is set to $\emptyset$.

Similarly, a *clock environment* $\mathcal{H}$ maps clock names ck to tuples (pck, r, m), where pck is the parent clock of ck and r (resp. m) tracks the number of steps remaining (resp. the maximum number of steps) in the current instant of the parent clock ($r, m \in \mathbb{N} \cup \{\infty\}$). The same notation is used to refer to the individual fields (for instance $\mathcal{H}^r(ck)$). We denote by $\mathcal{H}[ck \leftarrow i]$ the environment where $\mathcal{H}^r(ck)$ is set to i and by $Dom(\mathcal{H})$ the domain of $\mathcal{H}$, that is the set of clock names that are mapped to a tuple in $\mathcal{H}$.

A clock environment induces a partial order $\preceq_{\mathcal{H}}$, which is the smallest reflexive, transitive and antisymmetric relation such that $ck \preceq_{\mathcal{H}} \mathcal{H}^{pck}(ck)$. Intuitively, $ck_F \preceq_{\mathcal{H}} ck_S$ means that ck_S is slower than ck_F. We write $ck \preceq_{\mathcal{H}} C$ iff $\forall ck' \in C.\, ck \preceq_{\mathcal{H}} ck'$. $C^{\uparrow\mathcal{H}}$ denotes the upward closure of C, that is:

$$C^{\uparrow\mathcal{H}} = \{ck' \mid \exists ck \in C.\, ck \preceq_{\mathcal{H}} ck'\}$$

4.4. Semantics

We define two reductions: the *step reduction*, denoted $\xrightarrow{ck}$, and the *end-of-instant reduction* $\xrightarrow[\text{eoi}]{C}$. The step reduction is parametrized by the local clock ck. The execution of a reactive domain consists in applying the step

reduction with the local clock as many times as possible, to get a so-called *end-of-instant expression*. Then, the end-of-instant reduction prepares the execution of the next instant of the domain (where C is a set containing the clock of the domain).

A program is executed inside the global reactive domain of clock $\top_{ck}$. The variable `global_ck` is bound to this clock. This means that the semantics of a program p is given by the reduction of the expression $\tilde{p}$ defined by:

$$\tilde{p} \triangleq \texttt{let}\ \mathit{global\_ck} = \texttt{local\_ck}\ \texttt{in}\ p$$

A program step, denoted $\Rightarrow$, is made of as many step reductions as possible followed by one end-of-instant reduction in the local clock $\top_{ck}$:

$$\frac{e/\mathcal{H},\mathcal{S} \xrightarrow{\top_{ck}}{}^{*} e'/\mathcal{H}',\mathcal{S}' \qquad e'/\mathcal{H}',\mathcal{S}' \not\xrightarrow{\top_{ck}} \qquad e'/\mathcal{H}',\mathcal{S}' \xrightarrow{\{\top_{ck}\}}_{\text{eoi}} e''/\mathcal{H}'',\mathcal{S}''}{e/\mathcal{H},\mathcal{S} \Rightarrow e''/\mathcal{H}'',\mathcal{S}''}$$

The reduction starts from $e_0 = \tilde{p}$ and the initial clock and signal environments are both empty: $\mathcal{H}_0 \triangleq []$ and $\mathcal{S}_0 \triangleq []$.

Automatic waiting of reactive domains. Before we can define the reductions, we first have to define an auxiliary predicate that will implement the automatic waiting of reactive domains. It will be used to know if the body of a reactive domain wants to execute another local instant or is stuck waiting for a slower clock. It is defined by:

$$\mathcal{S}, C \vdash_{\text{next}} e$$

which means that during the end-of-instant of the clocks in the set C and in the signal environment $\mathcal{S}$, the expression e wants to execute another step of the local clock.

The predicate is defined in Figure 8:

- There are two main cases where an expression wants to execute another local instant: if it waits for the next instant by calling `pause` ck' (Next-Pause rule) or if the body of a `do`/`until` has been preempted (Next-Until rule). In both cases, we have to check that it is the end-of-instant of the clock ck', that is, that it belongs to the set C.

- We see here the difference between the `pause` and `quiet pause` operators. Calling `pause` ck' asks for another instant of the clock ck'. On the other hand, a call to `quiet pause` is ignored by the reactive domain for the decision to make another local instant, as no rule mentions it.

$$\textsc{NextPause}\ \frac{ck' \in C}{\mathcal{S}, C \vdash_{\text{next}} \texttt{pause}\ ck'} \qquad \textsc{NextUntil}\ \frac{n^{ck'} \in \mathcal{S} \qquad ck' \in C}{\mathcal{S}, C \vdash_{\text{next}} \texttt{do}\ e_1\ \texttt{until}\ n^{ck'}(x) \to e_2}$$

$$\textsc{NextIn}\ \frac{\mathcal{S}, C \cup \{ck'\} \vdash_{\text{next}} e}{\mathcal{S}, C \vdash_{\text{next}} e\ \texttt{in}\ ck'} \qquad \frac{\mathcal{S}, C \vdash_{\text{next}} e_1}{\mathcal{S}, C \vdash_{\text{next}} \texttt{do}\ e_1\ \texttt{until}\ n^{ck'}(x) \to e_2}$$

$$\frac{n^{ck} \in \mathcal{S} \qquad \mathcal{S}, C \vdash_{\text{next}} e}{\mathcal{S}, C \vdash_{\text{next}} \texttt{do}\ e\ \texttt{when}\ n^{ck}} \qquad \frac{\mathcal{S}, C \vdash_{\text{next}} e_1}{\mathcal{S}, C \vdash_{\text{next}} \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e}$$

$$\frac{\mathcal{S}, C \vdash_{\text{next}} e_2}{\mathcal{S}, C \vdash_{\text{next}} \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e}$$

Figure 8: Automatic waiting predicate

- The other rules are structural. In the case of a reactive domain (NextIn rule), we add the local clock of the domain in the set C of clocks at the end of their instant. It means that the end of instant of a clock implies the end of instant of its sub-clocks. There is no rule for the `do` e `when` n^{ck} when $n^{ck} \in \mathcal{S}$ because waiting for a signal is like a `quiet pause`. Finally, only one of the branches of a `let`/`and` has to require a next instant for the complete expression.

Step reduction. The step reduction is expressed as:

$$e/\mathcal{H}, \mathcal{S} \xrightarrow{ck} e'/\mathcal{H}', \mathcal{S}'$$

meaning that under the local clock ck, the expression e reduces to e' and transforms the clock and signal environments $\mathcal{H}$ and $\mathcal{S}$ into $\mathcal{H}'$ and $\mathcal{S}'$. The rules are given in Figure 9, where the basic rules are adapted directly from ReactiveML [5] and new rules are introduced for executing reactive domains:

- The expression `local_ck` returns the clock of its evaluation context.
- A signal can only be accessed if its clock ck' is accessible, that is, if ck' is slower than or equal to the local clock ck, denoted $ck \preceq_{\mathcal{H}} ck'$. Therefore, all expressions using signals have to add this condition.

$$\lambda x.e\ v/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow v]/\mathcal{H},\mathcal{S} \qquad \texttt{rec}\ x = e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow \texttt{rec}\ x = e]/\mathcal{H},\mathcal{S}$$

$$\texttt{local\_ck}/\mathcal{H},\mathcal{S} \xrightarrow{ck} ck/\mathcal{H},\mathcal{S} \qquad \texttt{run}\ (\texttt{process}\ e)/\mathcal{H},\mathcal{S} \xrightarrow{ck} e/\mathcal{H},\mathcal{S}$$

$$\texttt{let}\ x_1 = v_1\ \texttt{and}\ x_2 = v_2\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x_1 \leftarrow v_1; x_2 \leftarrow v_2]/\mathcal{H},\mathcal{S}$$

NewSig
$$\frac{n \notin Dom(\mathcal{S}(ck)) \qquad \mathcal{S}' = \mathcal{S}(ck)[n \mapsto (v_d, v_g, v_d, \emptyset)]}{\texttt{signal}\ x\ \texttt{default}\ v_d\ \texttt{gather}\ v_g\ \texttt{in}\ e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow n^{ck}]/\mathcal{H},\mathcal{S}'}$$

$$\frac{ck \preceq_{\mathcal{H}} ck'}{\texttt{emit}\ n^{ck'}\ v/\mathcal{H},\mathcal{S} \xrightarrow{ck} ()/\mathcal{H},\mathcal{S} + [v/n^{ck'}]} \qquad \frac{ck \preceq_{\mathcal{H}} ck'}{\texttt{last}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} \mathcal{S}^l(n^{ck'})/\mathcal{H},\mathcal{S}}$$

$$\frac{ck \preceq_{\mathcal{H}} ck'}{\texttt{do}\ v\ \texttt{until}\ n^{ck'}(x) \rightarrow e_2/\mathcal{H},\mathcal{S} \xrightarrow{ck} v/\mathcal{H},\mathcal{S}} \qquad \frac{ck = ck' \qquad n^{ck'} \in \mathcal{S}}{\texttt{do}\ v\ \texttt{when}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} v/\mathcal{H},\mathcal{S}}$$

Context
$$\frac{e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'/\mathcal{H}',\mathcal{S}'}{\Gamma(e)/\mathcal{H},\mathcal{S} \xrightarrow{ck} \Gamma(e')/\mathcal{H}',\mathcal{S}'}$$

When
$$\frac{ck = ck' \qquad n^{ck'} \in \mathcal{S} \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'/\mathcal{H}',\mathcal{S}'}{\texttt{do}\ e\ \texttt{when}\ n^{ck'}/\mathcal{H},\mathcal{S} \xrightarrow{ck} \texttt{do}\ e'\ \texttt{when}\ n^{ck'}/\mathcal{H}',\mathcal{S}'}$$

Inst
$$\frac{i > 0 \qquad ck' \notin Dom(\mathcal{H}) \qquad \mathcal{H}' = \mathcal{H}[ck' \mapsto (ck, i-1, i-1)] \qquad \mathcal{S}' = \mathcal{S}[ck' \mapsto []]}{\texttt{domain}\ x\ \texttt{by}\ i\ \texttt{do}\ e/\mathcal{H},\mathcal{S} \xrightarrow{ck} e[x \leftarrow ck']\ \texttt{in}\ ck'/\mathcal{H}',\mathcal{S}'}$$

Step
$$\frac{e/\mathcal{H},\mathcal{S} \xrightarrow{ck'} e'/\mathcal{H}',\mathcal{S}'}{e\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'\ \texttt{in}\ ck'/\mathcal{H}',\mathcal{S}'}$$

Term
$$v\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} v/\mathcal{H},\mathcal{S}$$

LocalEoi
$$\frac{\mathcal{H}^r(ck') > 0 \qquad e/\mathcal{H},\mathcal{S} \not\xrightarrow{ck'} \qquad \mathcal{S},\{ck'\} \vdash_{\text{next}} e \qquad e/\mathcal{H},\mathcal{S} \xrightarrow{\{ck'\}}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}' \qquad \mathcal{H}'' = \mathcal{H}'[ck' \leftarrow \mathcal{H}^r(ck') - 1] \qquad \mathcal{S}'' = \texttt{next}(\mathcal{S}', ck')}{e\ \texttt{in}\ ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} e'\ \texttt{in}\ ck'/\mathcal{H}'',\mathcal{S}''}$$

Figure 9: The step reduction

- The CONTEXT rule applies a step reduction in any valid evaluation context Γ, defined by:

$$\begin{aligned}\Gamma ::= {} & [] \mid \Gamma\ e \mid e\ \Gamma \mid (\Gamma, e) \mid (e, \Gamma) \mid \texttt{run}\ \Gamma \mid \texttt{pause}\ \Gamma \\ & \mid \texttt{let}\ x = \Gamma\ \texttt{and}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ x = e\ \texttt{and}\ x = \Gamma\ \texttt{in}\ e \\ & \mid \texttt{signal}\ x\ \texttt{default}\ \Gamma\ \texttt{gather}\ e\ \texttt{in}\ e \\ & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ \Gamma\ \texttt{in}\ e \\ & \mid \texttt{emit}\ \Gamma\ e \mid \texttt{emit}\ e\ \Gamma \mid \texttt{last}\ \Gamma \\ & \mid \texttt{do}\ \Gamma\ \texttt{until}\ v(x) \to e \mid \texttt{do}\ e\ \texttt{until}\ \Gamma(x) \to e \\ & \mid \texttt{do}\ e\ \texttt{when}\ \Gamma \mid \texttt{domain}\ x\ \texttt{by}\ \Gamma\ \texttt{do}\ e\end{aligned}$$

- We need to add a special rule for `do` e `when` n, as its body is an evaluation context only if the signal n is present. The clock of the signal must also be equal to the local clock as suspension represents an immediate dependency.

- A reactive domain is initialized by first evaluating the bound on the number of steps, initializing the clock environment and instantiating the clock variable with a fresh clock (INST rule).

- Then, local reduction steps (STEP rule) are applied while possible. If the body is reduced to a value, the reactive domain terminates (TERM rule), returning that value. Otherwise, a new local instant is started if the steps remaining counter has not reached zero and work remains to be done in the next local step (LOCALEOI rule). Indeed, if the end-of-instant relation leaves the body unchanged (here $e = e'$), doing more local steps would not change anything, as the body is already stuck with respect to the step reduction. The reactive domain is then stuck waiting for the end-of-instant of its parent reactive domain, as explained in Section 3.3.

End-of-instant reduction. The end-of-instant reduction is expressed as:

$$e/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e'/\mathcal{H}', \mathcal{S}'$$

meaning that during the end-of-instant of the clocks in the set C, e reduces to e' and transforms the clock and signal environments $\mathcal{H}$ and $\mathcal{S}$ into $\mathcal{H}'$ and $\mathcal{S}'$. We also write:

$$e/\mathcal{H}, \mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}} \Leftrightarrow\ e/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e/\mathcal{H}, \mathcal{S}$$

$$\frac{ck \in C}{\texttt{pause}\, ck/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} ()/\mathcal{H}, \mathcal{S}} \qquad \frac{ck \in C}{\texttt{quiet pause}\, ck/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} ()/\mathcal{H}, \mathcal{S}}$$

$$\frac{e_1/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e_1'/\mathcal{H}', \mathcal{S}' \qquad e_2/\mathcal{H}', \mathcal{S}' \xrightarrow{C}_{\text{eoi}} e_2'/\mathcal{H}'', \mathcal{S}''}{\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e/\mathcal{H}'', \mathcal{S}''}$$

EoiValue
$$v/\mathcal{H}, \mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad n^{ck} \in \mathcal{S} \qquad e/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e'/\mathcal{H}', \mathcal{S}'}{\texttt{do}\ e\ \texttt{when}\ n^{ck}/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} \texttt{do}\ e'\ \texttt{when}\ n^{ck}/\mathcal{H}', \mathcal{S}'}$$

$$\frac{ck \in C \qquad n^{ck} \in \mathcal{S}}{\texttt{do}\ e_1\ \texttt{until}\ n^{ck}(x) \to e_2/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e_2[x \leftarrow \mathcal{S}^v(n^{ck})]/\mathcal{H}, \mathcal{S}}$$

$$\frac{ck \in C^{\uparrow\mathcal{H}} \qquad ck \notin C \vee n^{ck} \notin \mathcal{S} \qquad e_1/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e_1'/\mathcal{H}', \mathcal{S}'}{\texttt{do}\ e_1\ \texttt{until}\ n^{ck}(x) \to e_2/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} \texttt{do}\ e_1'\ \texttt{until}\ n^{ck}(x) \to e_2/\mathcal{H}', \mathcal{S}'}$$

ParentEoi
$$\frac{e/\mathcal{H}, \mathcal{S} \overset{ck'}{\not\longrightarrow} \qquad not\ \left(\mathcal{H}^r(ck') > 0 \wedge \mathcal{S}, \{ck'\} \vdash_{\text{next}} e\right) \qquad e/\mathcal{H}, \mathcal{S} \overset{C \cup \{ck'\}}{\longrightarrow}_{\text{eoi}} e'/\mathcal{H}', \mathcal{S}' \qquad \mathcal{H}'' = \mathcal{H}'[ck' \leftarrow \mathcal{H}^m(ck')] \qquad \mathcal{S}'' = \texttt{next}(\mathcal{S}', ck')}{e\ \texttt{in}\ ck'/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{\text{eoi}} e'\ \texttt{in}\ ck'/\mathcal{H}'', \mathcal{S}''}$$

$$\frac{ck \in C^{\uparrow\mathcal{H}} \qquad ck \notin C}{\texttt{pause}\, ck/\mathcal{H}, \mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad ck \notin C}{\texttt{quiet pause}\, ck/\mathcal{H}, \mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}} \qquad \frac{ck \in C^{\uparrow\mathcal{H}} \qquad n^{ck} \notin \mathcal{S}}{\texttt{do}\ e\ \texttt{when}\ n^{ck}/\mathcal{H}, \mathcal{S} \overset{C}{\hookrightarrow}_{\text{eoi}}}$$

Figure 10: The end-of instant reduction

The rules are given in Figure 10. As for the step reduction, the basic rules are the same as in regular ReactiveML. The novelties are as follows:

- In several cases, we require the clock of signals to be in the upward closure of C, denoted $C^{\uparrow\mathcal{H}}$, which is the set of accessible clocks. The relation is not defined if we try to access a clock that is not in this set.
- Expressions that await a signal only reduce during the end-of-instant of the signal clock.
- There is only one rule for reactive domains (ParentEoi). It is applied when the body of the domain cannot do a step reduction on the local clock and is stuck waiting for a slower clock. We then do an end-of-instant reduction of the body e_1, adding the local clock ck' to the set C of clocks. Finally, we modify the clock and signal environments to prepare for the next instant.

Comparison with PPDP 2013. Compared to [10], the LocalEoi rule has been modified. The rule was:

$$\frac{\begin{array}{cc}\mathcal{H}^r(ck') > 0 & e/\mathcal{H},\mathcal{S} \xrightarrow{\{ck'\}}_{\text{eoi}} e'/\mathcal{H}',\mathcal{S}' \qquad e' \neq e \\ \mathcal{H}'' = \mathcal{H}'[ck' \leftarrow \mathcal{H}^r(ck') - 1] & \mathcal{S}'' = \texttt{next}(\mathcal{S}', ck')\end{array}}{e \texttt{ in } ck'/\mathcal{H},\mathcal{S} \xrightarrow{ck} e' \texttt{ in } ck'/\mathcal{H}'',\mathcal{S}''}$$

The modification is to take into account the new construct `quiet pause`. In this article, to detect if a domain has to react, we have introduced the predicate $\mathcal{S},\mathcal{C} \vdash_{\text{next}} e$. In [10], this predicate was not necessary: to detect if a domain had to react, we used the syntactic criterion $e \neq e'$. Here, this trick cannot be applied because `quiet pause` reduces to () at the end-of-instant. Hence, the body of a domain can change during the end-of-instant reaction, but it does not mean that the domain needs to react again.

5. Clock Calculus

Reactive domains induce restrictions on the use of signals. As the local instants of a reactive domain are unobservable from the outside, a signal attached to a reactive domain cannot be used outside of that domain. Immediate dependencies on slow signals are also forbidden.

We want to statically reject such programs which have an incorrect behavior. This is done by using a standard type-and-effect system that we

call a *clock calculus* in reference to synchronous languages [11]. This ability is one of the benefits of exposing concurrency in the language, as opposed to introducing it through a library. As usual, well-typed programs do not go wrong, which means here that they do not access a signal outside of its domain and do not depend immediately on slow signals.

5.1. Motivation

A first example of the sort of program that we want to reject is one where the result of a reactive domain contains a local signal:

```
let process result_escape =
  domain ck do
    signal s in s
  done
```

Such programs are rejected by including clocks in the type of signals and checking that the return types of reactive domains do not contain local clocks.

Signals are first-class values in the language, which means that a signal can be put inside any data structure or emitted on another signal. The consequence is that a signal can escape its lexical scope and be used anywhere in the program. We also have to make sure to reject programs where a signal escapes its reactive domain through a slow signal, like this one:

```
let process signal_escape =
  signal slow in
  domain ck do
    signal fast default 0 gather (+) in
    emit slow fast
  done
```

To avoid this case, we should also check that the local clock does not appear in the type of free variables when typing a reactive domain. To ensure this, clocks are seen as abstract data types as in [12]. However, signal accesses might not appear in the type of an expression, as in the following example:

```
let process effect_escape =
  domain ck do
    signal fast in
    let f () = emit fast in f
  done
```

The traditional solution to this problem is to associate an expression with both a type and an *effect* [13, 14]. In our case, the effect records the clocks of the signals accessed by the expression.

5.2. Notations

Types are defined by:

$$
\begin{array}{lll}
ct ::= T \mid \alpha \mid \{ce\} \mid ct \times ct \mid (ct, ct)\,\texttt{event}\{ce\} & & \text{(types)} \\
\quad \mid ct \xrightarrow{cf} ct \mid ct\,\texttt{process}\{ce|cf\} & & \\
ce ::= \gamma \mid ck & & \text{(clocks)} \\
cf ::= \phi \mid \emptyset \mid \{ce\} \mid cf \cup cf & & \text{(effects)} \\
cs ::= ct \mid \forall\alpha.\, cs \mid \forall\gamma.\, cs \mid \forall\phi.\, cs & & \text{(type schemes)} \\
\Gamma ::= [x_1 \mapsto cs_1; \ldots; x_p \mapsto cs_p] & & \text{(environment)}
\end{array}
$$

A type is either a basic type T, a type variable α, a singleton type $\{ce\}$ corresponding to the clock ce, a product, a signal, a function or a process. The type $(ct_1, ct_2)\,\texttt{event}\{ce\}$ of a signal is defined by the type ct_1 of values emitted, the type ct_2 of the received value (and default value) and its clock ce. A clock is either a clock variable or a clock name. An effect cf is attached to functions and processes, and it is a set of clocks or effect variables ϕ. Processes also have an activation clock, which can however be omitted if it does not appear in the return type or the effect of the process. Types schemes generalize over the three kinds of variables. Instantiation and generalization are defined classically by:

$$cs[\alpha \leftarrow ct] \leq \forall\alpha.\, cs \qquad cs[\gamma \leftarrow ce] \leq \forall\gamma.\, cs \qquad cs[\phi \leftarrow cf] \leq \forall\phi.\, cs$$

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

where $ftv(ct)$ returns the free type, clock and effect variables in the type ct. As signals are mutable structures, we need to distinguish *expansive* expressions [15] – for which types cannot be generalized.

5.3. Typing Rules

A typing judgment is given by:

$$\Gamma, ce \vdash e : ct \mid cf$$

meaning that under the environment Γ and local clock ce, the expression e has type ct and effect cf.

The initial typing environment Γ_0 contains the signatures of all primitives:

$$\begin{aligned}\Gamma_0 \triangleq [\, &\mathit{global\_ck} : \{\top_{ck}\};\ \texttt{pause}, \texttt{quiet pause} : \forall\gamma.\, \{\gamma\} \xrightarrow{\{\gamma\}} \texttt{unit};\\ &\texttt{last} : \forall\alpha_1, \alpha_2, \gamma.\, (\alpha_1, \alpha_2)\, \texttt{event}\{\gamma\} \xrightarrow{\{\gamma\}} \alpha_2;\\ &\texttt{emit} : \forall\alpha_1, \alpha_2, \gamma.\, (\alpha_1, \alpha_2)\, \texttt{event}\{\gamma\} \xrightarrow{\emptyset} \alpha_1 \xrightarrow{\{\gamma\}} \texttt{unit};\\ &\texttt{true} : \texttt{bool};\ \texttt{fst} : \forall\alpha_1, \alpha_2.\, \alpha_1 \times \alpha_2 \xrightarrow{\emptyset} \alpha_1;\ \ldots]\end{aligned}$$

The typing rules are given in Figure 11:

- The rules of the functional kernel are the usual rules in a type-and-effect system. The ProcAbs rule types the body of the process using its activation clock as the new local clock. Only a process on the local clock can be run (rule ProcApp). Often, this is done by instantiating a process whose activation clock is a clock variable.

- In order to forbid immediate dependencies on slow signals, the type system ensures that the clock of a signal is equal to the local clock (evident in the typing judgment as the *ce* next to the typing environment). See, for instance, the When rule.

- A design choice made in ReactiveML is to separate ML expressions from reactive expressions. For instance, tuples can only contain ML expressions. It is enforced in [5] by a separate syntactic analysis before typing. In the case of our extended type system, we enforce an even stronger separation, by forcing ML expressions to have no effect. This does not reduce expressivity since one can always use a `let` to isolate effectful expressions.

- The most important typing rule is Domain. It checks that the local clock does not escape from its reactive domain. This is done by using a fresh variable for the clock type. The side condition prevents scope extrusion of this fresh name by checking that it does not appear free in the return type *ct* of the domain nor in the typing environment Γ. It is similar to the typing of `let` in [12] and `let clock` in Lucid Synchrone [11].

$$\frac{ct \leq \Gamma(x)}{\Gamma, ce \vdash x : ct \mid \emptyset} \qquad \frac{ct \leq \Gamma_0(c)}{\Gamma, ce \vdash c : ct \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_1 : ct_1 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash (e_1, e_2) : ct_1 \times ct_2 \mid \emptyset}$$

$$\frac{\Gamma; x : ct, ce \vdash e : ct \mid cf}{\Gamma, ce \vdash \texttt{rec}\ x = e : ct \mid cf} \qquad \frac{\Gamma; x : ct_1, ce \vdash e : ct_2 \mid cf}{\Gamma, ce \vdash \lambda x.e : ct_1 \xrightarrow{cf} ct_2 \mid \emptyset} \qquad \frac{\Gamma, ce \vdash e_1 : ct_2 \xrightarrow{cf} ct_1 \mid \emptyset \quad \Gamma, ce \vdash e_2 : ct_2 \mid \emptyset}{\Gamma, ce \vdash e_1\ e_2 : ct_1 \mid cf}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : ct_1 \mid cf_1 \qquad \Gamma, ce \vdash e_2 : ct_2 \mid cf_2 \\ \Gamma; x_1 : gen(ct_1, e_1, \Gamma); x_2 : gen(ct_2, e_2, \Gamma), ce \vdash e : ct \mid cf\end{array}}{\Gamma, ce \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e : ct \mid cf_1 \cup cf_2 \cup cf}$$

ProcAbs
$$\frac{\Gamma, ce' \vdash e : ct \mid cf}{\Gamma, ce \vdash \texttt{process}\ e : ct\, \texttt{process}\{ce' | cf\} \mid \emptyset}$$

ProcApp
$$\frac{\Gamma, ce \vdash e_1 : ct\, \texttt{process}\{ce | cf\} \mid \emptyset}{\Gamma, ce \vdash \texttt{run}\ e_1 : ct \mid cf}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : ct_2 \mid \emptyset \qquad \Gamma, ce \vdash e_2 : ct_1 \longrightarrow ct_2 \longrightarrow ct_2 \mid \emptyset \\ \Gamma; x : (ct_1, ct_2)\, \texttt{event}\{ce\}, ce \vdash e : ct \mid cf\end{array}}{\Gamma, ce \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : ct \mid cf \cup \{ce\}}$$

$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : ct \mid cf_1 \qquad \Gamma, ce \vdash e : (ct_1, ct_2)\, \texttt{event}\{ce'\} \mid \emptyset \\ \Gamma; x : (ct_1, ct_2)\, \texttt{event}\{ce'\}, ce \vdash e_2 : ct \mid cf_2\end{array}}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x) \rightarrow e_2 : ct \mid cf_1 \cup cf_2 \cup \{ce'\}}$$

When
$$\frac{\begin{array}{c}\Gamma, ce \vdash e_1 : ct \mid cf_1 \\ \Gamma, ce \vdash e_2 : (ct_1, ct_2)\, \texttt{event}\{ce\} \mid \emptyset\end{array}}{\Gamma, ce \vdash \texttt{do}\ e_1\ \texttt{when}\ e_2 : ct \mid cf_1 \cup \{ce\}}$$

Domain
$$\frac{\begin{array}{c}\Gamma; x : \{\gamma\}, \gamma \vdash e : ct \mid cf \\ \Gamma, ce \vdash e_1 : \texttt{int} \mid \emptyset \qquad \gamma \notin ftv(\Gamma, ct)\end{array}}{\Gamma, ce \vdash \texttt{domain}\ x\ \texttt{by}\ e_1\ \texttt{do}\ e : ct \mid cf \setminus \{\gamma\}}$$

In
$$\frac{\Gamma, ck' \vdash e : ct \mid cf}{\Gamma, ce \vdash e\ \texttt{in}\ ck' : ct \mid cf \setminus \{ck'\}} \qquad \frac{}{\Gamma, ce \vdash \texttt{local\_ck} : \{ce\} \mid \emptyset}$$

Figure 11: Typing rules

5.4. Examples

The `result_escape` process is rejected by the type system because the fresh clock variable associated to `ck` appears in the result type of the reactive domain (denoting $e \triangleq$ `signal` s `in` s):

$$\frac{\Gamma_0; x : \{\gamma\}, \gamma \vdash e : (\alpha, \alpha\,\texttt{list})\,\texttt{event}\{\gamma\} \mid \{\gamma\}}{\Gamma_0, ce \vdash \texttt{domain}\ x\ \texttt{by}\ \infty\ \texttt{do}\ e : (\alpha, \alpha\,\texttt{list})\,\texttt{event}\{\gamma\} \mid \emptyset}$$

In the case of the `signal_escape` example, this variable appears in the type of the variable *slow* in the typing environment:

$$slow : ((\texttt{int}, \texttt{int})\,\texttt{event}\{\gamma\}, (\texttt{int}, \texttt{int})\,\texttt{event}\{\gamma\})\,\texttt{event}\{\top_{ck}\}$$

Finally, this variable also appears in the return type for the `effect_escape` process, in the effect of the function, as $f : \texttt{unit} \xrightarrow{\{\gamma\}} \texttt{unit}$.

6. Proof of Soundness

In this section, we prove the soundness of our type system using standard syntactic soundness techniques [16]. Clocks and effects are treated similarly to [17], where the proof is performed on a functional language with regions and references. As there are two reduction relations, we need to prove that an expression that cannot do any step reduction must be able to do an end-of-instant reduction. The novelty compared to the proof of soundness of ReactiveML is that we prove that signals are never accessed unless their clocks are accessible.

As in [17], we introduce two new typing environments in order to demonstrate the soundness of the type system: the *signal typing environment*, denoted Σ, and the *clock typing environment*, denoted H. Whereas the typing environment Γ maps variables x to types, Σ maps clock names to a partial map from signal names n to signal types $(ct_1, ct_2)\,\texttt{event}\{ce'\}$. It is thus used to give the type of signal names, which appear during the rewriting of expressions. H maps clock names to the type of their parent clocks and represents the clock tree. We also denote $Sig(\mathcal{S}) = \{n \in Dom(\mathcal{S}(ck)) \mid ck \in Dom(\mathcal{S})\}$ the set of all signal names defined in a signal environment.

Clocking judgments are now $\Gamma, \Sigma, H, ce \vdash e : ct \mid cf$. There are two new rules for typing signal names and clocks, which appear during reductions:

$$\Gamma, \Sigma, H, ce \vdash ck : \{ck\} \mid \emptyset \qquad \Gamma, \Sigma, H, ce \vdash n^{ck} : \Sigma(n^{ck}) \mid \emptyset$$

There is a new side-condition in the In rule: $H(ck') = ce$. It links the clock of the domain with the type of its parent clocks.

$$\frac{\Gamma, ck' \vdash e : ct \mid cf \qquad H(ck') = ce}{\Gamma, ce \vdash e \texttt{ in } ck' : ct \mid cf \setminus \{ck'\}} \text{ In}$$

Unlike the typing environment Γ, Σ and H are not populated by the typing rules. They are left untouched by all typing rules and are only used to type clock names n^{ck} for Σ and to access the clock tree for H. These environments will be built during the proof of soundness, in particular for type preservation. For instance, the signal typing environment Σ is enlarged each time a new signal is created, which corresponds to the NewSig rule of the step reduction (see Figure 9). The other crucial point is that in the Domain rule, the side-condition that γ does not escape only applies to the typing environment Γ, not the signal typing environment Σ.

We can first show two simple lemmas:

Lemma 1. *If $\Gamma, \Sigma, H, ce \vdash v : ct \mid cf$, then $cf = \emptyset$.*

Lemma 2. *If $\Gamma, \Sigma, H, ce \vdash v : ct \mid \emptyset$, then $\Gamma, \Sigma, H, ce' \vdash v : ct \mid \emptyset$ for any ce'.*

Proof. The activation clock that appears in the typing rules is only used to type processes. So the only difficult case is if $v = \texttt{process } e$, but then we replace the activation clock by a fresh clock variable to type e, so the activation clock does not matter. □

We now link signal environments with the signal typing environments that describe them, and similarly for clock environments:

Definition 1. *We say that a signal environment $\mathcal{S}$ is well-typed in the signal typing environment Σ, denoted $\Sigma \vdash \mathcal{S}$ if all the names in $\mathcal{S}$ appear in Σ, that is, if $n^{ck} \in Dom(\mathcal{S}) \Rightarrow n^{ck} \in Dom(\Sigma)$, and for all $n \in Sig(\mathcal{S})$, there exists ct_1, ct_2 and ce' such that:*

$$\begin{aligned}
&\Sigma(n^{ck}) = (ct_1, ct_2)\,\texttt{event}\{ck\}\\
&\Gamma_0, \Sigma, [], \top_{ck} \vdash \mathcal{S}^d(n^{ck}) : ct_2 \mid \emptyset\\
&\Gamma_0, \Sigma, [], \top_{ck} \vdash \mathcal{S}^g(n^{ck}) : ct_1 \xrightarrow{\emptyset} ct_2 \xrightarrow{\emptyset} ct_2 \mid \emptyset\\
&\Gamma_0, \Sigma, [], \top_{ck} \vdash \mathcal{S}^m(n^{ck}) : ct_1\ \texttt{multiset} \mid \emptyset
\end{aligned}$$

We do not put the clock in $\Sigma \vdash \mathcal{S}$ as the signal environment only contains values, for which Lemma 2 shows that the activation clock does not matter.

Definition 2. *We say that a clock environment $\mathcal{H}$ is well-typed in the clock typing environment H, denoted $H \vdash \mathcal{H}$, if they have the same domain and:*

$$\forall ck \in Dom(\mathcal{H}).\, \Gamma_0, [], [], \gamma \vdash \mathcal{H}^{ck}(ck) : H(ck) \mid \emptyset$$

We can now define the typing judgment of a configuration $e/\mathcal{H}, \mathcal{S}$:

$$\frac{\Gamma_0, \Sigma, H, ce \vdash e : ct \mid cf \qquad \Sigma \vdash \mathcal{S} \qquad H \vdash \mathcal{H} \qquad Dom(H) = Dom(\Sigma) \qquad ce \preceq_{\mathcal{H}} cf}{\Sigma, H, ce \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf}$$

A configuration is well-typed if the expression is well-typed, the signal and clock environments are well-typed and the effect of the expression is included in the set of accessible clocks, which are the clocks slower than the local clock ce.

The following lemmas are common in soundness proofs and can be proved using usual techniques [16]:

Lemma 3. *If $\Gamma, \Sigma, H, ce \vdash e : ct \mid cf$ and $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$ and $H \subseteq H'$, then $\Gamma', \Sigma', H', ce \vdash e : ct \mid cf$.*

Lemma 4 (Value substitution)**.** *If $\Gamma; x : \forall \bar{\alpha}. \forall \bar{\gamma}. \forall \bar{\phi}.\, ct', \Sigma, H, ce \vdash e : ct \mid cf$ and $\Gamma, \Sigma, H, ce \vdash v : ct' \mid \emptyset$ then $\Gamma, \Sigma, H, ce \vdash e[x \leftarrow v] : ct \mid cf$.*

Lemma 5 (Clock substitution)**.** *Let θ be a substitution that maps a clock variable to a clock name (i.e. $\theta = \{\gamma \leftarrow ck'\}$). If $\Sigma, H, ce \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf$, then $\Sigma, H, \theta(ce) \vdash e/\mathcal{H}, \mathcal{S} : \theta(ct) \mid \theta(cf)$.*

Proof. By induction on the typing rules, we can prove that $\theta(\Sigma), \theta(H), \theta(ce) \vdash \theta(e)/\theta(\mathcal{H}), \theta(\mathcal{S}) : \theta(ct) \mid \theta(cf)$. e, $\mathcal{S}$ and $\mathcal{H}$ do not contain any clock variables by definition, so they are left untouched. As $\Sigma \vdash \mathcal{S}$, so does Σ (similarly for H). □

Before proving typing preservation for the step reduction, we first need to prove that the end-of-instant relation preserves typing, as it is used by the LocalEoi rule.

Property 1 (Typing preservation for $\xrightarrow{C}_{\text{eoi}}$)**.** *If* $\Sigma, H, ck \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf$ *and* $e/\mathcal{H}, \mathcal{S} \xrightarrow{C}_{eoi} e'/\mathcal{H}', \mathcal{S}'$ *with* $ck \in C$*, then there exists* Σ'*,* H' *and* cf' *such that:*

$$\Sigma', H', ck \vdash e'/\mathcal{H}', \mathcal{S}' : ct \mid cf'$$

Proof. By induction on the typing derivation. □

Property 2 (Typing preservation for $\xrightarrow{ck}$)**.** *If* $\Sigma, H, ck \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf$ *and* $e/\mathcal{H}, \mathcal{S} \xrightarrow{ck} e'/\mathcal{H}', \mathcal{S}'$*, then there exists* Σ'*,* H' *and* cf' *such that:*

$$\Sigma', H', ck \vdash e'/\mathcal{H}', \mathcal{S}' : ct \mid cf'$$

Proof. By induction on the typing derivation of e.

Case `domain` x `by` v `do` e_1**:** Then $e' = e[x \leftarrow ck']$ `in` ck' and $\mathcal{S}' = \mathcal{S}[ck' \mapsto []]$, where $ck' \notin Dom(\mathcal{H})$. From Domain, we deduce that $\Gamma_0, \Sigma, H, \gamma \vdash e_1 : ct \mid cf_1$ with $\gamma \notin fcv(ct)$ and $cf = cf_1 \setminus \{\gamma\}$ where $fcv(ct)$ is the set of free clock variables in ct. Let's denote $\theta = \{\gamma \leftarrow ck'\}$. By applying Lemma 5, we get $\Gamma_0, \Sigma, H, \theta(\gamma) \vdash e_1 : \theta(ct) \mid \theta(cf_1)$. Then, we have $\theta(ct) = ct$ as $\gamma \notin fcv(ct)$. If we denote $H' = H[ck' \mapsto ck]$ and $\Sigma' = \Sigma[ck' \mapsto []]$, then as $\mathcal{H}'(ck') = ck$ and $H' \vdash \mathcal{H}'$, we have $H'(ck') = \{ck\}$. We can replace Σ with Σ' and H with H' in the previous typing judgment using Lemma 3. By applying In, we then get that $\Gamma_0, \Sigma', H', ck \vdash e' : ct \mid cf'$, where $cf' = cf[\gamma \leftarrow ck'] = cf$.

Case v `in` ck'**:** Then $e' = v$, $\mathcal{S}' = \mathcal{S}$ and $\mathcal{H}' = \mathcal{H}$. From In, we get that $\Gamma_0, \Sigma, H, ck' \vdash v : ct \mid cf_1$ with $cf = cf_1 \setminus \{ck'\}$ and $ck' \notin fcv(ct)$. From Lemma 1, we know that $cf_1 = cf = \emptyset$. We can apply Lemma 2 to get $\Gamma_0, \Sigma, H, ck \vdash v : ct \mid \emptyset$ and the other conditions are immediate.

Case e_1 `in` ck' **(Step):** Then we have $e_1/\mathcal{H}, \mathcal{S} \xrightarrow{ck'} e_1'/\mathcal{H}', \mathcal{S}'$. From In, we deduce that $\Gamma_0, \Sigma, H, ck' \vdash e_1 : ct \mid cf_1$ with $cf = cf_1 \setminus \{ck'\}$. As $H(ck') = \{ck\}$ and $H \vdash \mathcal{H}$, we can deduce that $\Gamma_0, [], [], \gamma \vdash \mathcal{H}^{ck}(ck') : \{ck\} \mid \emptyset$, which means that $\mathcal{H}^{ck}(ck') = ck$, i.e. $ck' \preceq_{\mathcal{H}} ck$. As we also have $ck \preceq_{\mathcal{H}} cf$ from the typing rule, it follows that $ck' \preceq_{\mathcal{H}} cf_1$. By induction, we have that $\Gamma_0, \Sigma', H', ck' \vdash e_1' : ct \mid cf_1'$ with $ck' \preceq_{\mathcal{H}'} cf_1'$. We can then apply the In rule to get the desired result.

Case e_1 `in` ck' **(LocalEoi):** We just have to apply Property 1 that shows that the end-of-instant reduction preserves typing.

Case `signal` x `default` v_1 `gather` v_2 `in` e_1 `:` Then $\mathcal{S}' = \mathcal{S}(ck)[n \mapsto (v_1, v_2, \emptyset)]$ and $e' = e[x \leftarrow n^{ck}]$. We get $\Gamma_0; x : (ct_1, ct_2)\,\texttt{event}\{ck\}, \Sigma, H, ck \vdash e_1 : ct \mid cf_1$ from the typing rule, where $cf = cf_1 \cup \{ck\}$.
Let's denote $\Sigma' = \Sigma[n \leftarrow (ct_1, ct_2)\,\texttt{event}\{ck\}]$. By applying Lemma 3 and 4, we get $\Gamma_0, \Sigma', H, ck \vdash e_1[x \leftarrow n^{ck}] : ct \mid cf_1$ and we easily prove that $\Sigma' \vdash \mathcal{S}'$.

Case `emit` $n^{ck'}$ v `:` We have $e' = ()$ and $\mathcal{S}' = \mathcal{S} + [v/n]$. We can easily prove that $\Gamma_0, \Sigma, H, ck \vdash () : \texttt{unit} \mid \emptyset$ and that $\Sigma \vdash \mathcal{S}'$ as $\Gamma_0, \Sigma, H, \gamma \vdash v : ct_1 \mid \emptyset$ where $\Sigma(ck')(n) = (ct_1, ct_2)\,\texttt{event}\{ck'\}$.

□

Definition 3. *We say that $e/\mathcal{H}, \mathcal{S}$ is an end-of-instant configuration for ck if there exists e', $\mathcal{H}'$ and $\mathcal{S}'$ such that $e/\mathcal{H}, \mathcal{S} \xrightarrow{\{ck\}}_{eoi} e'/\mathcal{H}', \mathcal{S}'$.*

For the proof of the *progress* property, we use the well-formation rules defined in Section 4.2. Stuck instantaneous expressions are values, whereas stuck non-instantaneous expressions are end-of-instant expressions.

Property 3 (Progress for instantaneous expressions)**.** *If e is instantaneous (i.e. $0 \vdash e$) and $\Sigma, H, ck \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf$, then either:*

- *e is a value*
- *There exists e', $\mathcal{H}'$ and $\mathcal{S}'$ such that $e/\mathcal{H}, \mathcal{S} \xrightarrow{ck} e'/\mathcal{H}', \mathcal{S}'$*

Property 4 (Progress for $\xrightarrow{ck}$)**.** *If $\Sigma, H, ck \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf$, then either:*

- *e is a value*
- *$e/\mathcal{H}, \mathcal{S}$ is an end-of-instant configuration for ck*
- *There exists $e', \mathcal{S}'$ such that $e/\mathcal{H}, \mathcal{S} \xrightarrow{ck} e'/\mathcal{H}', \mathcal{S}'$*

Proof. By induction on the structure of e.

Case `domain` x `by` v `do` e_1`:` Then $e' = e_1[x \leftarrow ck']$ `in` ck'.

Case e_1 `in` ck' `:` We can show that $\Sigma, H, ck' \vdash e_1/\mathcal{H}, \mathcal{S} : ct \mid cf_1$ like in the proof of Property 2. By induction, either there exists e_1', $\mathcal{H}'$ and $\mathcal{S}'$ such that $e_1/\mathcal{H}, \mathcal{S} \xrightarrow{ck'} e_1'/\mathcal{H}', \mathcal{S}'$, in which case we can apply Step, or e_1 is an end-of-instant expression for ck' and then:

- Either we can apply LocalEoi to do a step reduction on ck
- Otherwise, e_1 `in` ck' is an end-of-instant expression for ck as the ParentEoi rule necessarily applies.

Case `emit` $e_1\ e_2$: From the well-formation rules, we know that e_1 and e_2 are instantaneous expressions, so we can apply Property 3. If e_1 or e_2 is not a value, then we can reduce by using the context rule. Otherwise, we have $e_1 = n^{ck'}$ and $e_2 = v$. From the typing rule, we get that $ck' \in cf$, so $ck \preceq_{\mathcal{H}} ck'$ as $ck \preceq_{\mathcal{H}} cf$. We can thus reduce e to $e' = ()$. The proof is similar for all actions on signals.

□

Property 5 (Soundness of the type system for $\xrightarrow{ck}$)**.** *If* $\Sigma, H, ck \vdash e/\mathcal{H}, \mathcal{S} : ct \mid cf$*, then either :*

- *There exists* e'*,* $\mathcal{H}'$ *and* $\mathcal{S}'$ *such that* $e/\mathcal{H}, \mathcal{S} \xrightarrow{ck}{}^{*} e'/\mathcal{H}', \mathcal{S}'$ *and* e' *is an end-of-instant expression for ck (i.e. there exists* e'' *such that* $e'/\mathcal{H}', \mathcal{S}' \xrightarrow{\{ck\}}_{eoi} e''/\mathcal{H}'', \mathcal{S}''$*)*
- *For each* $e'/\mathcal{H}, \mathcal{S}$ *such that* $e/\mathcal{H}, \mathcal{S} \xrightarrow{ck}{}^{*} e'/\mathcal{H}', \mathcal{S}'$*, there exists* $e''/\mathcal{H}'', \mathcal{S}''$ *such that* $e'/\mathcal{H}', \mathcal{S}' \xrightarrow{ck} e''/\mathcal{H}'', \mathcal{S}''$*.*

The initial clock typing environment is $H_0 = []$. We easily check that $H_0 \vdash \mathcal{H}_0$.

Theorem 6.1 (Soundness of the type system)**.**
If $[], H_0, \top_{ck} \vdash p/\mathcal{H}_0, \mathcal{S}_0 : ct \mid cf$*, then either :*

- *There exists* $v, \mathcal{H}, \mathcal{S}$ *such that* $\tilde{p}/\mathcal{H}_0, \mathcal{S}_0 \Rightarrow^{*} v/\mathcal{H}, \mathcal{S}$*.*
- *For each* $p', \mathcal{H}', \mathcal{S}'$ *such that* $\tilde{p}/\mathcal{H}_0, \mathcal{S}_0 \Rightarrow^{*} p'/\mathcal{H}', \mathcal{S}'$*, there exists* $p'', \mathcal{H}'', \mathcal{S}''$ *such that* $p'/\mathcal{H}', \mathcal{S}' \Rightarrow p''/\mathcal{H}'', \mathcal{S}''$*.*

7. Implementation

In this section, we give an overview of the implementation of ReactiveML [6] and its extension with reactive domains [18].

7.1. Overview of ReactiveML *implementation*

After typing, the ReactiveML compiler generates sequential OCaml code. The main transformation is a partial CPS (Continuation Passing Style) transformation: at each point where a process can block, for instance when calling **pause** or awaiting for the emission of a signal, the compiler creates a continuation with the rest of the process. Instantaneous functions are left untouched for better performance. The generated code is linked to an OCaml library providing the ReactiveML runtime, containing mainly the execution engine that schedules the processes.

This execution engine is a task scheduler that uses cooperative scheduling. It means that, at the end of each instant, a process must cooperate with the scheduler to let other processes execute. The list $\mathcal{C}$ contains continuations that have to be executed in the current instant, where the scheduler picks tasks to execute. Some combinators, like the parallel composition, add processes to the list. A second list denoted *next* contains processes that should be executed in the next instant. This is where the **pause** operator puts its continuation. The execution of one instant of a program follows the algorithm below:

1. We execute one instant of the program. For that, we execute all the processes in the list $\mathcal{C}$ until it is empty. This corresponds to the step reduction of Figure 9.
2. We execute the end of instant (rules of Figure 10). We wake up all the processes awaiting the value of a signal and then transfer the processes from the list *next* into $\mathcal{C}$ to prepare the execution of the next instant.

Apart from this simple algorithm, the execution engine of ReactiveML has two other important features:

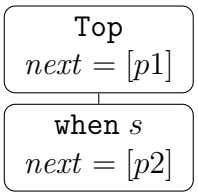
- For an efficient execution, it is important to not busy wait for signals. It means that a process awaiting a signal should only be awaken when the signal is present. In order to achieve that, we associate to each signal a list of the processes awaiting its emission. These continuations are only awaken when the signal is emitted.

- The most complex part of the execution engine is the handling of preemption (`do`/`until`) and suspension (`do`/`when`). By using a list of continuations, we lose the structure of the program. We need another control structure to deal with the fact that processes may not be active at all instants. This data structure is called *control tree*, as it is a n-ary

tree corresponding to the nesting of preemptions and suspensions in the program. Each `do/until` and `do/when` construct in the program is associated to a node in the tree and contains a list *next* of processes that should be executed at the next instant where this execution context is active. The root of the tree is associated with processes that are launched outside of any control structure and are active each instant. It contains the main *next* list that we have mentioned earlier. During the end of instant, we only transfer processes from the *next* list of nodes that will be active during the next step. In the case of a suspension, we will only do that when the suspension signal is emitted.

Here is an example of a program and the associated control tree:

```
let process control_tree s p1 p2 =
  emit s; pause; run p1
  ||
  do
    pause; run p2
  when s done
```



At the end of the first instant, the *next* list of the root of the tree contains `p1`, whereas the one associated with the suspension contains `p2`. We then transfer `p1` into $\mathcal{C}$ so that it is executed at the next instant, but we wait for the next emission of `s` before transferring `p2`.

7.2. Implementation of reactive domains

A reactive domain is an instance of an execution engine. In order to be able to create several reactive domains and nest them, it is necessary to create a data structure holding the state of the engine. It contains the list $\mathcal{C}$ of continuations to be executed in the current instant, the list of functions awaiting the end of instant, the root of the control tree and the counter of the number of instants executed during the current instant of the parent domain. The continuations implementing the different constructs of the language are then parametrized by the reactive domain they are running in.

The execution of a reactive domain follows the following algorithm, which is very similar to the operational semantics of Section 4:

- We execute one instant of the domain by launching all continuations in the list $\mathcal{C}$ until it is empty.
- We execute the end of the instant of the domain, by waking up processes that await the value of a signal on the local clock.

- The reactive domain then has to decide whether to execute another local instant or not, which corresponds to the $\vdash_{\text{next}}$ predicate defined in Figure 8. In the implementation, it amounts to checking whether there exists a process in the *next* list of an active control tree node. In that case, it means that there is a process that wants to be executed at the next local instant. If all the *next* list are empty or the instant counter of the domain has reached its maximum value, then the domain awaits the next instant of its parent domain.

- We then prepare the execution of the next instant by transferring processes from the *next* lists into the $\mathcal{C}$ list. If the reactive domain awaits its parent domain of clock `ck`, it is important to do this last step after the end of instant of the parent domain. Indeed, a process calling **`pause`** `ck` inside the local domain will be awaken during the end of the instant of the parent domain, but we have to make sure it is put in $\mathcal{C}$ list of the local domain before starting its next local instant.

The implementation of the `quiet pause` operator is also simple. Just like in the operational semantics, it differs from the `pause` operator only with respect to the automatic waiting of reactive domains. The idea is to associate with each node of the control tree another list called *next_control*. Processes that ask for another local instant, like `pause`, are put in the *next* list. On the other hand, processes that do not influence the automatic waiting, like `quiet pause` are put in the *next_control* list. The reactive domain only checks *next* lists to decide whether to do another local instant or not. But processes in *next* and *next_control* lists are transferred into the $\mathcal{C}$ list when preparing the next instant.

7.3. Reactive Domains and Parallelism

Apart from the gains in terms of expressivity and modularity, reactive domains are also useful for an efficient parallel execution of ReactiveML programs. This is not by chance as parallelism was one of the motivations for developing reactive domains. Indeed, by creating local time scales, we can separate local synchronizations from global synchronizations. Furthermore, we have decided to forbid immediate dependencies on slower signals, as this violates the assumption that a signal has only one value per instant. This restriction has another benefit: during each instant of its parent reactive domain, a reactive domain can run independently of other processes and

reactive domains at the same level. In particular, it can be run in parallel inside another thread of execution and will only synchronize at the end of the instant of its parent reactive domain, when all local instants have been executed. A possible execution scheme is to execute each reactive domain sequentially, as usual, and to use *work stealing* [19] to balance the load between the different threads. Furthermore, signals declared inside a reactive domain never escape, so they can remain local to the thread and do not require any mechanism, such as locks, to deal with concurrent accesses.

8. Discussion

8.1. Signals Clock

So far, all the signals we have used have been attached to the reactive domain at the point of definition. It is often desirable to be able to declare a signal attached to a slower clock than the local one. For example, consider this process that performs a blocking request:

```
let process send_query s =
  signal tmp in
  emit s tmp;
  await tmp(v) in v
```

The process sends the local signal `tmp` on the input signal `s`, and then awaits a reply on `tmp`. It is not possible to use this mechanism to communicate between two sibling reactive domains (e.g. two agents) as the signal `tmp` is attached to the reactive domain of the sender and cannot escape it. This can be solved by allowing a declaration of the clock of a signal:

```
signal tmp clock global_ck in ..
```

The semantics and type system are readily adapted to address this extension. The basic signal declaration can then be interpreted as a declaration on the local clock:

$$\texttt{signal}\ s\ \texttt{in}\ e \triangleq \texttt{signal}\ s\ \texttt{clock}\ \texttt{local\_ck}\ \texttt{in}\ e$$

8.2. Limitations of the Type System

The type system as it was presented imposes restrictions when writing combinators, as in this example:

```
let process run_domain q =
  domain ck do run q done
```

This process is rejected because the activation clock of the process `q` is equal to the local clock `ck` at the point where it is run, and by consequence `ck` escapes the scope of its domain.

The source of the problem is that, in ML, the type of a function argument is monomorphic: it is a type ct, not a type scheme cs. In particular, the activation clock of the process `q` cannot be universally quantified, as it is for most processes declared at toplevel. Many solutions to add *higher-rank polymorphism* to ML exist in the literature (see [20, 21] for instance). Most of them require some form of typing annotations from the programmer.

Another important limitation of the type system is that functions or processes stored together must have exactly the same effect. This restriction can be lifted using a simple form of subtyping restricted to effects, often called *subeffecting* [9].

As our type system is standard, these two extensions can be realized by adapting existing solutions [18].

9. Related Work

Our work is related to the *clock refinement* [22, 23] introduced by Gemünde et al. in the synchronous language Quartz [24]. There, the main idea is also to introduce the ability to synchronize on faster local clocks. However, their semantics is based on a transition system that defines the values of all the variables of the program and checks that it does not introduces causality issues. The solution adopted in Quartz cannot be applied to ReactiveML because the latter is dynamic: an arbitrary number of processes and signals can be created at runtime using recursion and they can be stored in data-structures and sent via signals (similarly to mobility in the π-calculus). In this context, it is impossible to determine the potential emitters of a signal and thus to decide signal absence, which is a requirement for the clock refinement described by Gemünde et al.

Furthermore, our work enables more possibilities for communication and synchronization between reactive domains. In particular, Quartz does not permit waiting for the emission of a signal within a domain: the `delayed_hello_world` example (page 10) could not be written in Quartz, and this is also true for the sensor node and n-body examples (Figures 4 and 5). This greatly reduces the expressivity of the language as this construct is used in almost all programs. In Quartz, the `await` expression can be encoded:

$$\texttt{await}\ e_1(x)\ \texttt{in}\ e_2 \triangleq \texttt{do}\ (\texttt{loop}\ (\texttt{pause}\ \texttt{local\_ck}))\ \texttt{until}\ e_1(x) \rightarrow e_2$$

whereas ReactiveML uses a `quiet pause` instead of `pause`. The consequence is that a reactive domain in Quartz that uses an `await` is not reactive (it executes an infinite number of local steps). We solve this problem by treating each reactive domain as a separate entity that decides to wait the next instant when its body is blocked or after a certain number of local instants as explained in Section 3.3 and formalized by the LocalEoi rule in Figure 9. Note that it is the automatic waiting of the reactive domain which is important and not the presence of the `quiet pause` expression: we could have defined directly the semantics of `await` as in [10].

Finally, reactive domains can be created dynamically and nested arbitrarily. This is not the case in Quartz: the clock tree must be known at compile time.

Other related works include SugarCubes [25], which shares the same concurrency model as ReactiveML but uses Java as the base language. SugarCubes allows the creation of *reactive machines*, the equivalent of our reactive domains, anywhere in the program, but does not offer any ways to communicate and synchronize between machines. It is the responsibility of the programmer to manually schedule the machines, by calling a `react` method as many times as necessary.

In other synchronous languages, time refinement is achieved using *oversampling*, as in Lustre [4] or Signal (see example 4 of [26]). However, oversampling is less modular than reactive domains: it makes a subprogram go faster by slowing down everybody else. The parallel composition of processes that oversample is problematic, especially if each has a different number of internal steps, as in Figure 3b. In Lustre, such programs are rejected by the clock calculus. In Signal, one can specify such behaviors but the compiler is not able to generate sequential code (see the FWS example in [27]). Furthermore, oversampling cannot turn a process taking n instants into an instantaneous one. It can only reduce the delay to one instant. The `levelorder_inst` example shows that this is possible with reactive domains.

The discussion on the correctness of the `immediate_dep_wrong` process in Section 3.2 is reminiscent of the causality problems of Esterel [2]. Indeed, this process is *logically correct*, as we can give one and only one status to the signal `s`. But it is not *constructively correct* as the effect, that is the reaction to the presence of `s`, happens before the cause, that is the emission of `s`. We have decided here to follow the same approach as ReactiveC and restrict the expressivity of the language so that causality problem are eliminated by construction. In the case of ReactiveC and ReactiveML, it means that

one can react to the absence of a signal only at the next instant. In our case, it means forbidding immediate dependencies on slow signals.

10. Conclusion

We have presented an extension to the synchronous model of concurrency, called reactive domains, and applied it to the ReactiveML language. It allows the creation of local notions of instant, thereby improving the modularity of the language and facilitating refinement. We have extended the semantics of the language to include this feature and formalized a type system that prevents the unsound use of signals.

The most important future work is to evaluate the usefulness of reactive domains on bigger programs, including the existing sensor network simulations [3]. We are also currently developing a parallel runtime for our extension using system processes communicating via message passing, based on the ideas presented in Section 7.3.

Acknowledgments

We would like to thank Abdoulaye Gamatié for fruitful discussions on the subject and Timothy Bourke and Guillaume Baudart for their detailed review. Thanks to the PPDP 2013 and SCP reviewers for their useful comments and suggestions.

References

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, R. D. Simone, The synchronous languages twelve years later, in: Proceedings of the IEEE, 2003, pp. 64–83.

[2] G. Berry, The constructive semantics of pure Esterel, 1996.

[3] L. Samper, F. Maraninchi, L. Mounier, L. Mandel, GLONEMO: global and accurate formal models for the analysis of ad-hoc sensor networks, in: InterSense '06, 2006.

[4] J. Mikac, P. Caspi, Temporal refinement for Lustre, in: International Workshop on Synchronous Languages, Applications and Programs, 2005.

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

[6] L. Mandel, C. Pasteur, M. Pouzet, ReactiveML, ten years later, in: Proceedings of the 17th ACM SIGPLAN international conference on Principles and practice of declarative programming, 2015.

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

[8] L. Mandel, C. Pasteur, Reactivity of cooperative systems, in: Proceedings of 21st International Static Analysis Symposium (SAS'14), 2014.

[9] H. Nilsson, A. Courtney, J. Peterson, Functional reactive programming, continued, in: Haskell '02, 2002, pp. 51–64.

[10] L. Mandel, C. Pasteur, M. Pouzet, Time refinement in a functional synchronous language, in: Proceedings of 15th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming, Madrid, Spain, 2013.

[11] J. Colaço, M. Pouzet, Clocks as First Class Abstract Types, in: R. Alur, I. Lee (Eds.), Embedded Software, volume 2855 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2003, pp. 134–155.

[12] K. Laufer, M. Odersky, An extension of ML with first-class abstract types, in: ACM SIGPLAN Workshop on ML and its Applications, 1992, pp. 78–91.

[13] J. M. Lucassen, D. K. Gifford, Polymorphic effect systems, in: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88, 1988, pp. 47–57.

[14] J.-P. Talpin, P. Jouvelot, The type and effect discipline, in: IEEE Symposium on Logic in Computer Science, 1992, pp. 162 –173.

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

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

[17] C. Calcagno, S. Helsen, P. Thiemann, Syntactic type soundness results for the region calculus, Information and Computation 173 (2002) 199–221.

[18] C. Pasteur, Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel, Ph.D. thesis, Université Paris 6, 2013.

[19] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.

[20] S. Jones, D. Vytiniotis, S. Weirich, M. Shields, Practical type inference for arbitrary-rank types, Journal of Functional Programming 17 (2007) 1–82.

[21] D. Rémy, Simple, partial type-inference for System F based on type-containment, in: ACM SIGPLAN Notices, volume 40, 2005, pp. 130–143.

[22] M. Gemünde, J. Brandt, K. Schneider, Clock refinement in imperative synchronous languages, EURASIP Journal on Embedded Systems 3 (2013) 1–21.

[23] M. Gemünde, Clock Refinement in Imperative Synchronous Programs, Ph.D. thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, 2013.

[24] K. Schneider, The synchronous programming language Quartz, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[25] F. Boussinot, J. Susini, The SugarCubes tool box: a reactive Java framework, Software: Practice and Experience 28 (1998) 1531–1550.

[26] P. Le Guernic, J. Talpin, J. Le Lann, Polychrony for system design, Journal of Circuits, Systems, and Computers 12 (2003) 261–303.

[27] A. Gamatié, T. Gautier, The signal synchronous multiclock approach to the design of distributed embedded systems, IEEE Transactions on Parallel and Distributed Systems 21 (2010) 641–657.