A Timed IO monad

David Janin
Univ. Bordeaux, CNRS, Bordeaux INP,
LaBRI, UMR 5800
France
janin@labri.fr

Abstract

We propose an application programming interface for reactive, concurrent and timed system programming in pure functional programing languages such as Haskell.

Simply said, we embed the IO monad into a timed IO monad by extending the implicit IO monad states with explicit timing information. This yields the notion of timed IO actions, that is, IO actions extended with specified durations, that can be composed in sequence in the resulting timed monad. Every complex timed actions induces an automatically defined scheduling specification of its sub-actions. Various tools are then proposed to measure and control the time drift of a running timed action, that is, the difference between the actual and the specified scheduling of these sub-actions.

An extension of the asynchronous concurrent library also offers the possibility to fork any timed action. The fork operator instantaneously returns a reference to the forked action that can be used to retrieve, transform and combined, in a safe and robust way, all the data dynamically (and timely) produced by that action. The full expressiveness of the proposed interface eventually appears when applying all of the above to (some notion of) higher order streams, that, instantiated over timed IO actions, provides a handy encoding of (locally finite) timed IO signals.

1 Introduction

Context, motivation and objectives. An example of timed programing is the programing of an interactive music system. In the simplest approach, the behavior of such a system alternates between:

(1) the computation of certain notes to be played, contributing to the *specification* of the music to be played[1],
(2) the rendering of these notes in time, contributing to the *actual performance* of that music.

One could think that, in order to realize such a behavior, it suffices to play each note for its specified duration. However, such an approach immediately yields a time leak. Indeed, the actual rendering of the music is getting delayed, note after note, by the actual computation time of each note specification. In other words, playing music in a even and regular way necessitates to reduce the *specified* duration of each note in order to compensate the *actual* duration of the computation of its specification: a computation that is implicitly (but wrongly) assumed to be of neglectable duration. Though simple, the explicit programing of such reductions in order to achieve a correct scheduling in time is repetitive, tedious and error prone.

There appears the need for a timed application programing interface that freely allows the programmer to declare certain computations as instantaneous while some others have a specified positive duration, in such a way that when running a timed program:

(1) *the* specified temporal scheduling *of that program shall automatically derive from the (possibly dynamically) specified* durations *of the actions it contains*,
(2) *the* actual temporal scheduling *observed when running that program shall match, within reasonable bound, the specified temporal scheduling*.

Additionally, and somehow conversely, given a running timed program, often full of side effects such as a sub-program actually playing music, the programmer may also wish to extract from running sub-program the dynamic information related to their executions in order to reuse it, as input, into some other sub-programs. There also appears the need for a timed application programing interface such that:

(3) *the* temporal and returned data *associated to any (effect-full) timed execution of a program shall be possibly read and freely reused by other programs in a safe (essentially effect-free) and robust (deadlock free) way.*

Our contribution. In this paper, we propose such a timed programing interface within a pure functional programing language such as Haskell, by simply lifting the IO monad interface to a timed IO monad interface.

More precisely, based on the observation that *passing time is a side effect*, we design a *timed IO monad* where timed action are essentially defined as IO action extended with timing information. Up to technical details, this is achieved by extending the (implicit) IO monad state by an (explicit) time stamp that refers to the *expected* or *specified* timestamp in that state. As a result, at runtime, the difference between the *actual* or *real* timestamp, as provided by the underlying OS, and such an expected timestamp, as deriving from the

[1]The specification of a note to be played is typically defined by its pitch, its expected duration, its velocity, etc…, possibly dynamically depending on the environment inputs

Haskell Symposium'19, , Submission
2019.

duration specifications, called the *time drift*, is provably positive in any state. Moreover, various tools are available in order to ensure and to check that, in practice, such a time drift remains bounded and, depending on the underlying timer, small.

Much like in the existing asynchronous concurrent library, but extended to a timed setting, a notion of timed IO action *references*, uniquely associated to *running* timed IO actions is also defined. It allows for collecting and freely reusing, essentially with no undesired side-effects, the dynamic information created by and associated with these running timed IO actions. Last, a notion of higher order streams, parameterized by a type constructor, allows the definition of timed IO streams of actions and references that lift all the above tools to timed data-flow programing.

Most concepts are presented via type classes and illustrated throughout by specific instances and uniformly derived utility functions.

Organization of the paper. The timed monad class, that refines the monad class, is first presented in Section 2. The expected properties of the resulting set of timed primitives combined with classical (untimed) primitives is stated via a series of invariant laws.

In section 3, a timer type class is defined in order to bind together time scale and duration type, timing and scheduling functions, and timed monad states. Though keeping the possibility to rely on other (possibly external) scheduling mechanisms, a default timer instance is described, based on *ghc* runtime and its concurrent library.

A concrete proposal for a timed IO monad instance is presented in Section 4 together with a simple parallel extension. Its correctness is examined with a special attention brought to the measure and control of time drift.

A more powerfull parallel extension is proposed in Section 5 via a generic notion of references to running monad actions. Simply said, monad references allow for accessing, in a safe and robust way, to the information dynamically produced by forked monad actions. Various invariant laws are proposed in order to capture such an expected semantics. Timed IO references are then defined as an instance of this notion, again based on Haskell's concurrent library.

A generic notion of monad streams is defined in Section 6 as a lifting of the classical lists data type with monad action accessors. Combined with the notion of monad references, this allows for developing numerous class instances and functions over (asynchronous) monad streams such as, for instance, the on-the-fly merge of two monad streams by arrival time of data. Monad streams built with timed IO actions are the timed IO streams that essentially behave like (locally finite) timed IO signals.

Monad references are also extended to timed streams in Section 7 so that monad streams can be forked returning a one-way broadcast communication channel, called a monad stream reference, that can be used to replay, share, duplicate the resulting streams of data without replaying the associated side effects.

As one last illustration of the combined simplicity and robustness of our approach, we eventually propose and instanciate, in the timed IO monad, a generalized notion of parallel fork of (traversable) finite structures of (timed) monad actions that returns a (timed) stream of monad references ordered by termination time. An linear time on-the-fly fold of these values then easily derives from such a parallel fork.

Related works are discussed in Section 8 before concluding in Section 9 with some performance tests and ongoing work in progress. The resulting library shall be made available on the internet under open source licence.

2 The timed monad type class

Before getting into technicalities, let us briefly review what we mean by a time scale and by a timed monad.

Timestamp, duration and time scale. Simply said, a *timestamp* is defined here as the *duration* elapsed from some fixed but unknown initial time. We expect timestamps, therefore durations as well, to be totally ordered in a *time scale*. However, while the sum of two durations makes perfect sense, the sum of two timestamps does not.

This suggests defining a duration type as any type instance of the *Num* and *Ord* class of Haskell, and the associated timestamp type as some restricted encapsulation of durations. In Haskell, this is done by putting:

```
newtype Time d = Time d deriving (Eq, Ord)
```

where d is the duration type and $Time\ d$ is the timestamp type, with the associated functions:

```
duration :: Num d ⇒ Time d → Time d → d
duration (Time d1) (Time d2) = (d1 − d2)
shift :: Num d ⇒ Time d → d → Time d
shift (Time d1) d2 = Time (d1 + d2)
```

that measure the (relative) duration between two timestamps, and that shift a timestamp by some duration.

Expected vs real time. As already mentioned in the introduction, a key point of our proposal lays in the distinction between:

(1) *expected* timestamps used for scheduling specification,
(2) *actual* timestamps observed along scheduling execution,

This distinction induces a timing performance measure: the *time drift* defined as the difference between the actual timestamp and the expected timestamp.

It is a desirable property that, in a running timed program, such a time drift is kept *positive* so that no action is actually scheduled before its specified time, and *bounded* so that

any specified duration above that bound can accurately be handled by the underlying scheduler.

Timed monad. Simply said, a timed monad is a monad where every action take some specified duration. The interface of a timed monad is detailed by the following type class type:

```
class (Monad m, MonadIO m, Num d, Ord d)
  ⇒ TimedMonad m d | m → d where
  now :: m (Time d)
  drift :: m d
  runTIO :: m a → IO a
  delay :: d → m ()
```

where *now* instantaneously returns the current specified timestamp, *drift* instantaneously returns the current time drift, that is, the difference between the actual timestamp (as provided by the underlying runtime scheduler) and the specified timestamp (as stored in the underlying monad state), *runTIO* turned timed monad actions into IO actions that can be run, and, *delay d* is an action that waits until the current specified time stamp shifted by the given positive duration is eventually passed *for real*.

Defining such a type class, as any other classes later in the text, allows for distinguishing an adequate set of *primitive functions* that shall be defined in any instance, from which other *utility functions* may derive in a *uniform* way.

Timed action duration. A first exemple of a derived function is the duration of an action that can be defined by:

```
dur :: TimedMonad m d ⇒ m a → m d
dur m = do
  t0 ← now
  _ ← m
  t1 ← now
  return (duration t1 t0)
```

Observe that computing a duration implies running the action together with its side-effects. This first means that a duration is a dynamic value that can depend on when it is measured. This also means that function *dur* shall essentially be used for stating the various laws that shall be satisfied by timed monad instances.

Invariant laws for timed monad. We review below the main super classes of the class *TimedMonad*, their invariant laws , as well as the additional laws [2] that shall be satisfied when used in a timed monad instance. This includes the class *Functor* given by:

```
class Functor m where
  fmap :: (a → b) → m a → m b
```

that allows to lift a function into a monad. Any instance shall satisfy the two functor laws:

$$m \equiv fmap\ id\ m \tag{1}$$

$$fmap\ f\ (fmap\ g\ m) \equiv fmap\ (f \circ g)\ m \tag{2}$$

together with an additional duration law:

$$dur\ m \equiv dur\ (fmap\ f\ m) \tag{3}$$

that is, timed IO action durations shall be preserved under *fmap f*. This also includes the class *Monad* given by:

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

that allows to lift a value into a monad action with *return*, and to combine sequentially two actions with the bind operator ($\gg=$), the second being parametrized by the value returned by the first. Any instance shall satisfy the three monad laws:

$$return\ a \gg= f \equiv f\ a \tag{4}$$

$$m \gg= return \equiv m \tag{5}$$

$$(m \gg= f) \gg= g \equiv m \gg= (\lambda x \to f\ x \gg= g) \tag{6}$$

plus the duration laws:

$$dur\ (return\ a) \equiv return\ 0 \tag{7}$$

$$dur\ (m_1 \gg m_2) \equiv dur\ (m_1) \gg= \lambda d \to fmap\ (d+)\ (dur\ m_2) \tag{8}$$

that is, return actions take no time, and the duration of two actions composed by the bind operator is the sum of the durations of these actions. Last, this includes the class *MonadIO* given by:

```
class MonadIO m where
  liftIO :: IO a → m a
```

that allows for lifting any IO action to a timed monad action. It shall satisfy the two lifting laws:

$$liftIO \circ return \equiv return \tag{9}$$

$$liftIO\ (m \gg= f) \equiv liftIO\ m \gg= (liftIO \circ f) \tag{10}$$

with the duration law:

$$dur\ (liftIO\ m) \equiv liftIO\ m \gg return\ 0 \tag{11}$$

that states that every lifted action has zero duration and the additional retraction law:

$$m \equiv runTIO\ (liftTIO\ m) \tag{12}$$

that states that, up to untimed monad action equivalence, we shall have $runTIO \circ liftTIO = id$. Observe that over timed actions, the reverse direction does not hold since we have

$$m \not\equiv liftIO\ (runTIO\ m) \tag{13}$$

[2] Two timed actions m_1 and m_2 are stated equivalent, a property denoted by $m_1 \equiv m_2$, when they have the same type and, in any context of use, return equal values and produce the same *observable* side effects by, let's say, going through the same series of monad states.

as soon as the timed action m has a *non zero duration.*

3 The timer type class

In a given timed monad, the same time scale shall be used both for expected timestamps, as stored in the underlying timed monad states, and for actual timestamps, as handled by some underlying timing and scheduling functions. This is achieved by defining the type class *Timer* which every instance uniquely bind these features together:

```
class (Ord d, Num d) ⇒ Timer s d | s → d where
  getStateTime :: s → Time d
  shiftState :: s → d → s
  initialState :: IO s
  getRealTime :: s → IO (Time d)
  waitUntil :: s → Time d → IO s
```

There, type d refers to some duration type, and type s refers to some associated timed monad state type. Every timed state shall embed an *expected* (or specified) timestamp that can either be accessed via the function *getStateTime* or be shifted by some positive duration via the function *shiftState*. Additionally, every timed state embeds a (hidden) handle towards a (possibly external) timer[3] that can be accessed via the IO actions:

(1) *initialState*, that initializes such a timer and returns the corresponding initial state where the initial expected timestamp equals the initial actual timestamp,
(2) *getRealTime s*, that returns the *actual* (or real) timestamp in a state s as provided by that timer,
(3) *waitUntil s t*, that waits for a wake up call from that timer sent right *after* the expected timestamp t is actually passed *for real*, and returns the timed state updated with t as the new expected timestamp.

For convenience, in the sequel, we shall also use the following derived functions:

```
compareStateTime s1 s2
  = compare (getStateTime s1) (getStateTime s2)
latestState s1 s2
  = if (getStateTime s1 < getStateTime s2)
    then s2 else s1
```

definable thanks to the fact that timestamps inherit from the duration total order.

A default timer instance. By default, using *ghc* runtime as a timer, durations are measured in nanoseconds and timed states are just defined as timestamps on these durations.

[3]It is common feature in a (distributed) timed system that a timer is located in one device and shared by the other devices. This also allows us to define an active timer that turns out to be more accurate than the default one as discussed in the conclusion.

More precisely, thanks to the *clock* and *concurrent* libraries of Haskell, we put:

```
newtype Nano = Time Integer
  deriving (Eq, Ord, Num, Integral)
getSystemTime = do
  rt ← getTime Monotonic
  return $ Time $ (fromInteger ∘ toNanoSecs) rt

instance Timer (Time Nano) Nano where
  getStateTime = id
  shiftState = shift
  initialState = getSystemTime
  getRealTime _ = getSystemTime
  waitUntil _ (Time d) = do
    Time rt ← getSystemTime
    threadDelay $ fromIntegral $ div (d − rt) 1000
    return (Time d)
```

with *threadDelay* expecting a microseconds argument.

4 The timed IO monad instances

Simply said, a *timed IO action* is defined as an IO action that not only acts on the (implicit) IO state, but also acts on an explicit timed state. In other words, we put:

```
newtype TIO s d a = TIO (s → IO (s, a))
```

parameterized by timed state type s and duration type d that form an instance of the type class *Timer*.

Functor, monad and monadIO instances. With no surprise, the mapping *TIO s d* that maps any type a to the type *TIO s d a* is monad functor with:

```
instance Functor (TIO s d) where
  fmap f (TIO m) = TIO $ λs0 → do
    (s1, a) ← m s0
    return (s1, f a)

instance Monad (TIO s d) where
  return a = TIO (λs → return (s, a))
  (>>=) (TIO a1) f = TIO $ λs → do
    (s1, v) ← a1 s
    let TIO a2 = (f v) in a2 s1
```

Since the IO monad is a strict monad, the timed IO monad is also *strict*, i.e. in a bind, the first timed IO actions is necessarily executed in order to produce the output state that is given to the next action. As a consequence, the bind operator is also *truly sequential.*

Every IO action can also be lifted into an instantaneous timed IO action thanks to the following instance:

```
instance MonadIO (TIO s d) where
  liftIO m = TIO $ λs → m >>= λa → return (s, a)
```

Since the underlying timed state is left unchanged this lift operator indeed creates timed IO actions that are (specified as) *instantaneous*.

With an aim at keeping a small time drift, such a lift operator should either be applied to an IO action which execution time is neglectable compared to the expected time accuracy, or, as detailled below, followed by a delay action with a duration large enough to compensate the real duration of the lifted IO action. So far in our proposal, it is programmer duty to ensure that such a lifting is properly used.

The timed IO monad instance. Thanks to the class *Timer*, we eventually define the timed IO monad by:

```
instance Timer s d ⇒ TimedMonad (TIO s d) d where
  now = TIO $ λs → do
    return (s, getStateTime s)
  drift = TIO $ λs → do
    rt ← getRealTime s
    let st = getStateTime s
    return (s, duration rt st)
  runTIO (TIO f) = do
    s ← initialState
    (_, a) ← f s
    return a
  delay d = TIO $ λs → do
    case d > 0 of
      True → do
        let t1 = shift (getStateTime s) d
        s1 ← waitUntil s t1
        return (s1, ())
      False → return (s, ())
```

with any timer instance as a parameter.

Observe that, as specified above, from the current specified timestamp t = *getStateTime* s, the action *delay* d does *not* delay the action by d unit of time. Instead, it waits until the new expected time stamp t_1 = *shift* t d is actually (just) passed *for real*. As a consequence, provided the time drift before a delay is smaller than its duration parameter, the timedrift after the execution of that delay is minimal and only depends on the time accuracy of the associated timer.

Correctness of the timed IO monad. One can easily check that properties (1)-(12) are satisfied. The standard laws follows directly from the fact that *TIO s d* is a simple variation on a classical state monad. The new laws follow from the way all the above defined functions act on timestamps in timed states.

Thanks to the specification of the *waitUntil* function, the delay action yields a positive time drift. Since every other definable timed action has an actual duration greater than its specified duration, we have:

(I) *in every timed state, the time drift is positive,*

However, keeping a bounded time drift *is not* ensured by default. Instead:

(II) *it is programmer duty to keep the time drift bounded*

which can be done by, say, inserting long enough delays between each series of actions specified as instantaneous in order to compensate the time drift created by the actual execution of these actions.

Limited parallelism. As an example of an instantaneous lifting of an IO action, one can define simple parallelism by

```
simpleForkTIO :: TIO s d a → TIO s d ()
simpleForkTIO (TIO m) = $ λ s → do
  _ ← forkIO (do {_ ← m s; return ()})
  return (s, ())
simpleParTIO :: TIO s d a → TIO s d b → TIO s d b
simpleParTIO a b = (simpleForkTIO a) >> b
```

that launches in parallel two actions, returning the value of the second action as soon as it terminates.

With parallelism, there appears the fact that the bind operator in the timed IO monad shall rather be understood as a synchronization operator between two timed actions. In other words, the bind product allows overlaps between synchronized timed behavior and thus appears as a fairly generic version of the *tiled product* proposed in [1, 9] for music.

Another IO action lifting. Alternatively, there is also a timed lift of IO actions. This can be useful when wishing to wait on an input IO action. With such a timed lifting, the timedrift is set, upon exit, to a minimal one, with a specified timestamp that shall match the actual timestamp corresponding to the termination of the lifted action. Such a timed lifting is defined by:

```
liftTimedIO :: TimedMonad m d ⇒ IO a → m a
liftTimedIO m = do
  a ← liftIO m
  drift >>= delay
  return a
```

Observe that IO action lifting is defined uniformly on any timed monad instance, musch like we have defined *liftIO* itself. Could it be used instead as the default lifting of IO actions ? The answer is no as illustrated by the timed IO monad. Indeed, with such a timed lifting, the law (9) cannot be satisfied since the function *return* is specified to produce zero duration timed actions while *liftTimedIO* ∘ *return* hardly ever does with any realistic timer since *any* computation takes some time.

5 The monad reference class

We aim now at providing a more general notion of parallelism where all values and timing information produced by the execution of a forked action can be accessed in a safe and sound way in other actions. Inspired by Haskell asynchronous concurrent library, we eventually define a fairly general notion of *monad action reference* that, simply said, can be understood as broadcast channel from a uniquely referenced running monad action towards any monad action possessing a copy of that reference.

Monad reference class. As the concept of monad reference is fairly orthogonal to the concept of timed monad, this notion is conveniently described over arbitrary monads by:

```
class Monad m ⇒ MonadRef m where
  type Ref m :: ∗ → ∗
  forkToRef :: m a → m (Ref m a)
  readRef :: Ref m a → m a
  tryReadRef :: Ref m a → m (Maybe a)
  parReadRef :: Ref m a → Ref m b → m (Either a b)
```

where:

(1) type (*Ref m*) is the type of monad action references,
(2) function *forkToRef* launches a monad action and returns (in no time) a reference to it,
(3) function *readRef* possibly waits for and eventually returns the value returned by the referenced action, with function *tryReadRef* a non blocking of the same function,
(4) function *parReadRef* returns the value of the earliest terminated referenced action when that fastest action is terminated.

A key intention behind such a definition is that: *while a running monad action may perform some side effects, running a reading action via a reference to such a monad action shall essentially produce no or harmless side effects*[4] .

Monad reference laws. A number of laws are expected to be satisfied by any instance of that class. Though not stabilized yet, they aim at capturing monad reference semantics as well as the above intention. Indeed, the first monad reference law essentially describes the *semantics* of monad references stating that:

$$m \equiv forkToRef\ m \gg\!= readRef \tag{14}$$

that is, reading the data produced by a just forked action just replay that action, all side effects included ! The second monad reference law, or *idempotence* law:

$$readRef\ r \equiv readRef\ r \gg readRef\ r \tag{15}$$

shall capture the fact that *readRef* actions have no or harmless side effects as they can be repeated in a non noticeable way. The third monad reference law, or *commutativity* law:

$$\begin{aligned} & readRef\ r_1 \gg\!= \lambda x_1 \to readRef\ r_2 \\ & \quad \gg\!= \lambda\, x_2 \to return\ (x_1, x_2) \\ & \qquad \equiv readRef\ r_2 \gg\!= \lambda x_2 \to readRef\ r_1 \\ & \qquad\quad \gg\!= \lambda x_1 \to return\ (x_1, x_2) \end{aligned} \tag{16}$$

shall capture the fact that *readRef* actions are bound to the execution of a unique running action therefore commute one with the other.

Timed racing. As a usage example, one can run two actions in parallel and waits for the first to terminate thanks to the following derived function:

```
parRun :: MonadRef m ⇒ m a → m b → m (Either a b)
parRun m1 m2 = do {r1 ← forkToRef m1;
  r2 ← forkToRef m2; parReadRef r1 r2 }
```

Observe that in *parRun* m_1 m_2, both actions m_1 and m_2 are executed till they terminate. This differs from the *race* function behavior in the asynchronous concurrent library. Clearly, both functions *parReadRef* and *parRun* may have non deterministic outcome as shown by:

parRun (*return* "foo") (*return* "foo")

that returns either *Left* "foo" or *Right* "foo".

Timed IO references. For the timed IO monad we put:

```
newtype TIORef s d a = TIORef (s, MVar (s, a))
```

where, in a timed IO reference value of the form *TIORef* (s, v) the state s shall be the start state of the referenced action and the mutable variable v shall contains, upon termination of the referenced action, the stop state and the returned value of that action. These yields the following instance of monad references:

```
instance Timer s d ⇒ MonadRef (TIO s d) where
  type Ref (TIO s d) = TIORef s d
  readRef (TIORef (_, v)) = TIO $ λs → do
    (s1, a) ← readMVar v
    return (latestState s s1, a)
  tryReadRef (TIORef (_, v)) = TIO $ λs → do
    c ← tryReadMVar v
    case c of
      Nothing → return (s, Nothing)
      Just (s1, a) → return (latestState s s1, Just a)
  forkToRef (TIO m) = TIO $ λs → do
    v ← newEmptyMVar
    _ ← forkIO (m s >>= putMVar v)
    return (s, TIORef (s, v))
```

[4] A possible definition for an action m to produce no or harmless side effects could be that $m \equiv forkToRef\ m \gg m$. Examples in the IO monad are *return a* that has harmless side effect and *print* "foo" that has not.

for the simplest functions, where, we return the lastest state since read actions can be launched before or after the referenced action is terminated. The code for the parallel reading of references is a bit more involved:

```
parReadRef (TIORef (_, v1)) (TIORef (_, v2))
  = TIO $ λs → do
  v ← newEmptyMVar
  _ ← forkIO (toMVar Left v1 v)
  _ ← forkIO (toMVar Right v2 v)
  (s1, x1) ← takeMVar v
  case x1 of
    Left _ → do
      c ← tryReadMVar v2
      case c of
        Nothing → return (latestState s s1, x1)
        Just (s2, v2) → return $
          earliest s (s1, x1) (s2, Right v2)
    Right _ → do
      c ← tryReadMVar v1
      case c of
        Nothing → return (latestState s s1, x1)
        Just (s2, v2) → return $
          earliest s (s1, x1) (s2, Left v2)
    where
      toMVar dir vr v = do
        (s, a) ← readMVar vr
        _ ← tryPutMVar v (s, dir a)
        return ()
      earliest s (s1, x1) (s2, x2) =
        case compareStateTime s1 s2 of
          GT → (latestState s s2, x2)
          _ → (latestState s s1, x1)
```

since, in the case the two referenced actions are already terminated, we have to select the earliest terminated one, regardless of what read action has won the race for accessing the mutable variable v. We also make sure that every forked thread terminates.

Timed monad with references. In the case a timed monad has monad references, we also want to read via references the timing information produced by referenced actions, that is, its duration. So we define the class type:

```
class (TimedMonad m d, MonadRef m)
  ⇒ TimedMonadRef m d where
  durRef :: Ref m a → m d
```

As application examples, the following derived function creates a delay that last exactly the same time as the referenced running action:

```
delayRef :: TimedMonadRef m d ⇒ Ref m a → m ()
delayRef r = do
  t0 ← now
  d ← durRef r
  t1 ← now
  delay (d − duration t1 t0)
```

Replaying a referenced timed IO action without its associated side effects but the *same* duration can then be done thanks to the following derived function:

```
replayRef :: TimedMonadRef m d ⇒ Ref m a → m a
replayRef r = delayRef r ≫ readRef r
```

Observe that both *delayRef* and *replayRef* can be used as soon as their parameter is available, therefore right after a fork, even in the case the referenced action is not terminated yet.

Timed IO durations via references. In the timed IO monad, retreiving the duration of a referenced timed IO action can be done as follows:

```
instance Timer s d ⇒
  TimedMonadRef (TIO s d) d where
    durRef (TIORef (s0, v)) = TIO $ λs → do
      (s1, _) ← readMVar v
      return (latestState s s1,
        duration (getStateTime s1) (getStateTime s0))
```

6 Monad streams

We aim now at developing a type for modeling (finite or infinite) timed signals as they may appear in timed systems. Since passing time is a side effect, this eventually leads us to the following definition of higher order streams that are eventually applied to monad functors.

Higher-order streams. The higher-order streams we shall use are defined by:

```
newtype Stream f a
  = Stream {next :: f (Maybe (a, Stream f a))}
```

where $f : * \to *$ is a function over types.

Though the way to access values may may depend on the chosen type function f, every accessed element of type *Stream f a* eventually provides:

(1) either *Nothing*, that tells the stream terminates
(2) or *Just* (a, sc), that tells the stream contains the value a and continues via sc.

Applied to the *Identity* functor, we recover a type functor essentially equivalent to the classical list type functor.

Monad streams. Streams of monad actions, simply called *monad streams*, are defined as elements of type *Stream m a* for some monad functor *m*. As a particular case, we put:

```
type STIO s d a = Stream (TIO s d) a
```

for the type of *timed IO streams*.

To some extent, timed IO streams correspond to timed signals, but without the memory leaks inherent to explicit timestamp handling. Indeed, a monad stream is essentially a single monad action that, when no longer evaluated, is simply garbage collected.

Functor instance. The type constructor *Stream m* built over a monad *m* is a functor as shown by:

```
instance Monad m ⇒ Functor (Stream m) where
  fmap f (Stream m) = Stream $ do
    c ← m
    case c of
      Nothing → return Nothing
      Just (a, mc) → return $ Just (f a, fmap f mc)
```

which amounts to traversing the monad stream by recursively executing the embedded monad actions.

Horizontal monoid. Streams can be composed one after the other, yielding the following monoid structure called the horizontal monoid.

```
instance Monad m ⇒ Monoid (Stream m a) where
  mempty = Stream (return Nothing)
  (◇) (Stream m) s = Stream $ do
    c ← m
    case c of
      Nothing → next s
      Just (a, sc) → return $ Just (a, sc ◇ s)
```

Observe that, in the case the first monad stream is infinite, the second monad stream will be delayed endlessly which, if repeated, may create a memory leak. Still, the following monoid laws are satisfied:

$$s \equiv mempty \diamond s \tag{17}$$

$$s \equiv s \diamond mempty \tag{18}$$

$$s_1 \diamond (s_2 \diamond s_3) \equiv (s_1 \diamond s_2) \diamond s_3 \tag{19}$$

Iterator examples. Thanks to the following function that binds a monad action to a monad stream:

```
bindToStream :: Monad m ⇒
  m a → (a → Stream m a) → Stream m a
bindToStream m f = Stream $
  m ≫= λa → return $ Just (a, f a)
```

one can define various iterators for stream creations such as *unconditional iteration*:

```
iterateStream :: Monad m ⇒
  (a → m a) → a → Stream m a
iterateStream f a = bindToStream (f a) (iterateStream f)
```

or *conditional iteration* that depends on the fact that the iterated function produces a value or not,

```
iterateStreamMaybe :: Monad m ⇒
  (a → m (Maybe a)) → a → Stream m a
iterateStreamMaybe f a = Stream $ do
  c ← f a
  return $ fmap (λb → (a, iterateStreamMaybe f b)) c
```

or *iteration until* some referenced action terminates:

```
iterateStreamUntil :: MonadRef m ⇒
  Ref m b → (a → m a) → a → Stream m a
iterateStreamUntil r f a = Stream $ do
  tc ← tryReadRef r
  case tc of
    Just _ → return Nothing
    Nothing → next $ bindToStream (f a)
      (iterateStreamUntil r f)
```

Merge and vertical monoid. With monad references, monad streams can also be merged by *termination time* of the embedded actions by:

```
merge :: MonadRef m ⇒
  Stream m a → Stream m a → Stream m a
merge (Stream m1) (Stream m2) = Stream $ do
  r1 ← forkToRef m1
  r2 ← forkToRef m2
  c ← parReadRef r1 r2
  case c of
    Left Nothing → readRef r2
    Right Nothing → readRef r1
    Left (Just (a, sc)) → return $
      Just (a, merge sc (Stream $ readRef r2))
    Right (Just (a, sc)) → return $
      Just (a, merge (Stream $ readRef r1) sc)
```

This illustrates the intrinsic asynchronism of monad streams where values are produced…when they are produced ! One can also observe that the merge operator is associative and commutative, even with the non determinism induced by *parReadRef*, with the empty stream *mempty* (again) as neutral element. This commutative monoid is called the *vertical monoid*.

Monad instance. With *merge*, the functor *Stream m* turns out to be a monad functor as shown by:

```
instance MonadRef m ⇒ Monad (Stream m) where
  return a = (Stream ∘ return ∘ Just) (a, empty)
  (≫=) (Stream m) f = Stream $ do
    c ← m
    case c of
      Nothing → return Nothing
      Just (a, mc) → next $ merge (f a) (mc ≫= f)
```

where the bind operation is defined as the concurrent merge of all the parameterized monad streams generated from the values produced (in time) by the left monad stream.

An important aspect of such an operation is that all these merges are schedule *when* the corresponding production actions are terminated. This therefore perfectly fits application to timed system programing, with explicit time with timed IO actions, or even implicit time with regular IO actions.

7 Monad stream references

Much like we have defined action references bound to running action instances, we define below a notion of monad stream references, that are produced by forked monad streams and that can read in a safe and robust way. In other words, a monad stream reference acts as an unbounded broadcast fifo channel[5] from a uniquely defined running stream monad.

Monad stream reference. The type of monad stream reference is simply defined from streams by the following type synonym:

```
type StreamRef m = Stream (Ref m)
```

Forking a (effect-full) monad stream into a stream of references is done as follows:

```
forkStreamToRef :: MonadRef m ⇒
  Stream m a → m (StreamRef m a)
forkStreamToRef s = do
  r ← forkToRef (evalAndFork s)
  return $ Stream r
      where
  evalAndFork (Stream m) = do
    c ← m
    case c of
      Nothing → return Nothing
      Just (a, sc) → do
        rc ← forkToRef (evalAndFork sc)
        return $ Just (a, Stream rc)
```

[5]As such, monad stream references may lead to memory and time leak if used without care.

where, each monad reference, when later read, is either producing *Nothing*, when the stream terminates, or the value *Just* (*a*, *vc*) with the read value *a* and a sort of a continuation reference *vc*.

Conversely, reading a stream of monad references by converting it into a (essentially effect-free) monad streams can be achieved by the following function:

```
readStreamRef :: MonadRef m ⇒
  StreamRef m a → Stream m a
readStreamRef (Stream v) = Stream $ do
  c ← readRef v
  case c of
    Nothing → return Nothing
    Just (a, rc) → return $ Just (a, readStreamRef rc)
```

Though clearly useful, just as *forkToRef* and *readRef*, these functions do not induce an instance of *MonadRef* for *Stream m* with *StreamRef m* as associated reference type. Indeed,,such an instance would yield a notion of streams of streams, that is, elements of type *Stream* (*Stream m*) *a*, which, to state it simply, sounds a bit hard to manipulate.

Heterogeneous merge and split. Quite in the style of arrows programming[10] though in an asynchronous setting (see remark below), one can define heterogeneous merge and split by

```
mergeH :: MonadRef m ⇒
  Stream m a → Stream m b → Stream m (Either a b)
mergeH s1 s2 = merge (fmap Left s1) (fmap Right s2)
```

together with the associated split function defined by

```
splitH :: MonadStreamRef m ⇒
  Stream m (Either a b) → m (Stream m a, Stream m b)
splitH s = do
  r ← forkStreamToRef s
  return (fstStream (readStreamRef r),
    sndStream (readStreamRef r))
```

defined thanks to the projections defined via:

```
filterMaybe :: MonadRef m ⇒
  (a → Maybe b) → Stream m a → Stream m b
filterMaybe f (Stream m) = Stream $ do
  c ← m
  case c of
    Nothing → return Nothing
    Just (a, mc) → case (f a) of
      Nothing →
        let Stream mr = filterMaybe f mc in mr
      Just b → return $ Just (b, filterMaybe f mc)
```

by

```
fstStream :: MonadRef m ⇒
    Stream m (Either a b) → Stream m a
fstStream s = filterMaybe fromLeft s
    where fromLeft (Left a) = Just a
        fromLeft _ = Nothing
sndStream :: MonadRef m ⇒
    Stream m (Either a b) → Stream m b
sndStream s = filterMaybe fromRight s
    where fromRight (Right b) = Just b
        fromLeft _ = Nothing
```

It occurs that, up to side effects, that is, assuming actions have essentially no side effects but explicit changes of state, the type *Stream m (Either a b)* with *fstStream* and *sndStream* stream projection functions, is the categorical product of *Stream m a* and *Stream m b*.

Monad stream reference class. Last, combining the notion of monad references with the above definition of streams of references, we eventually define an extension of the *MonadRef* class defined by:

```
class MonadRef m ⇒ MonadStreamRef m where
    forkAllToRefs :: Traversable t ⇒
        t (m a) → m (StreamRef m a)
```

where function *forkAllToRefs* shall fork all monad actions stored in a (finite) traversable structure and returns a stream of references towards these actions, ordered by termination time. With timed IO streams we put:

```
instance Timer s d ⇒
    MonadStreamRef (TIO s d) where
        forkAllToRefs l = TIO $ λs → do
            v ← newEmptyMVar
            mapM_ (λ(TIO m) →
                forkIO (m s ≫= putMVar v)) l
            rv ← newEmptyMVar
            forkIO (toSTIORef (length l) v s rv)
            return (s, Stream (TIORef (s, rv)))
                where
            toSTIORef 0 _ s rv = do
                putMVar rv (s, Nothing)
            toSTIORef n v s rv = do
                (s1, a) ← takeMVar v
                rvc ← newEmptyMVar
                let s2 = latestState s s1
                forkIO (toSTIORef (n − 1) v s2 rvc)
                putMVar rv (s2, Just (a,
                    Stream (TIORef (s, rvc))))
```

where we are using a single mutable variable to receive all information produced by the forked timed IO actions.

Combined with sort of a folding function defined over streams[6]:

```
foldStream :: Monad m ⇒ (b → a → m b) →
    b → Stream m a → m b
foldStream f b (Stream m) = m ≫= (maybe (return b)
    (λ(a, s) → (f b a) ≫= λb → foldStream f b s))
```

one can finally define:

```
foldOnTime :: (MonadStreamRef m, Traversable t) ⇒
(b → a → m b) → b → t (m a) → m b
foldOnTime f b c =
    forkAllToRefs c ≫= foldStream f b ∘ readStreamRef
```

that forks all timed monad actions stored in a structure and performs a linear time on-the-fly folding of the returned values ordered by termination time.

No arrow instance. As a final remark, it is known that the *Arrow* programming interface[10] is quite appealing for data flow (therefore stream) programing. However, it seems bound to synchronous flow programming. Indeed, the categorical product of streams of type a and b is, in arrow programing, the type of streams of type (a, b). As detailed above, this is not the case with monad streams where the categorical product is rather, by asynchronism, the type of streams of type *Either a b*. This means that monad streams do not provide any instance of the *Arrow* type class.

8 Related Works

In functional programing languages, there already are many proposals for programing timed reactive concurrent systems. These proposals range from the synchronous language family [25] possibly extended with modern polymorphisms a with Reactive ML [18], to the many variants of functional reactive program (FRP) series initiated with FRAN [6] and continued with Yampa [22], to name but a few.

Still, the amount of recent publications concerning new type systems [12, 13, 15] or refined interfaces [7, 23, 24, 27] for FRP as well as the fairly recent extension proposals to higher-order timed features for synchronous languages [4, 5], suggest that the topic is still very active and yet not stabilized.

One of the reason is that, aside all these concrete, efficient and useful development of programing languages, there is yet no agreements on what could be the underlying relevant mathematical model for timed value semantics, a model from which robust primitives should be derived.

Of course, timed signals, that is, functions from time to values, constitute quite an obvious model for timed value

[6]Strictly speaking, the type *Stream m a* itself is not foldable since most functions on streams yields values in the monad *m*.

semantics, with interesting mathematical properties [16, 17]. However, the unrestricted use of explicit timestamps is a well known source of memory leaks [15, 24][7]. Synchronous programing languages, somehow also based on the signal model [2, 3], provide safe and efficient timed programing interfaces, but only by enforcing severe syntactical or typing contraints on programs.

In other words, while timed signals provide models for describing what *are* timed values, they do not induce programing interfaces adequate enough that tell *how* timed values *should* be accessed, combined and produced in an efficient, safe and robust way. As a recent illustration of this fact, the rather exhaustive study of timed signals in a domain theoretical setting [11] still fails by itself to induce any programing interface that would be adequate in the sense depicted above.

In this paper, we follow a somehow orthogonal approach initiated by Hudak with the notion of Polymorphic Temporal Media [8]. There, it is suggested that timed values could be modeled via the algebraic properties of their combinators, that is, one sequential and one parallel operators, two operators that, as later shown, can be merged into a single one : the tiled product [9]. Although initially designed by modeling music systems, such an approach has some appeal for generic programing purpose. Indeed, these algebraic properties yield some notion of normal form for temporal media value [8] which, in turn, induces some canonical constructors and accessors [1], without any explicit timestamps, therefore not so much memory leaks.

Could such an approach be truly extended to more general applications with IO was far from being clear and not even suggested in the afore mentioned papers. Indeed, algebraic modeling approaches, often limited to some first order algebraic data types, often seem bound to yield only static structures, missing all the dynamic features necessary for programing reactive timed systems. The timed IO monad presented here is, in that respect, quite a success, especially through the notion of (higher-order) monad streams that eventually offers both the comfort and flexibility of monad programming while preserving the mathematical elegance of algebraic approaches.

Last, a key point of our proposal clearly lays in the definition of monad references. Such a concept is *not new*. Though under a different name, it is for instance defined and used in the asynchronous concurrent library of Haskell [14, 19]. However, our treatment of monad action references is somewhat different. For instance, with *parReadRef*, we do not seek, as with function *race*, at stoping the latest action as soon as the earliest is terminated. On the contrary, monad references are used here to replay in a safe way, till their ends, referenced monad actions. Also, to the best of our knowledge, no axiomatization of this concept had yet been attempted.

[7]Incidentally, the notion of monad stream defined in these pages offers yet another solution to the memory leak problem stated and solved by Ploeg and Claessen in FRPNow [24].

9 Conclusion

Monad action, instance and reference. It is a well established facts that, thanks to monad modeling, pure functional programing can be safely extended to programming with side effects [20, 26]. Simply said, the monad approach offers a clear distinction between :

(1) monad action *programs* that can be duplicated and combined at will in a pure functional programing style,
(2) monad action *instances*, that is, *running* programs, that are uniquely defined and positioned at runtime in the underlying monad state space,

the values returned by monad action instances being reinjected into monad action programs only via the bind operator.

In this paper, somehow extending this approach, timed monads and monad references provide together :

(1) timed action *programs*, simply referred to as timed *actions*, with (possibly dynamically) *specified durations*, that can be duplicated and combined at will (in parallel or in sequence) in a pure functional programing style,
(2) timed action *instances*, obtained by running timed actions, with *actual durations*, that cannot be duplicated nor combined one with the other since they are already and uniquely located in the spacetime of the underlying running system,
(3) timed action *references*, uniquely associated to forked timed action instances, that can be used to read, duplicate, share and combine freely, robustly and safely, within new timed actions, the runtime information dynamically created by action instances.

The notion of monad references thus induces a new and versatile interface through which running monad actions can be safely and robustly re-injected into monad action programs, more freely than with the bind operator only.

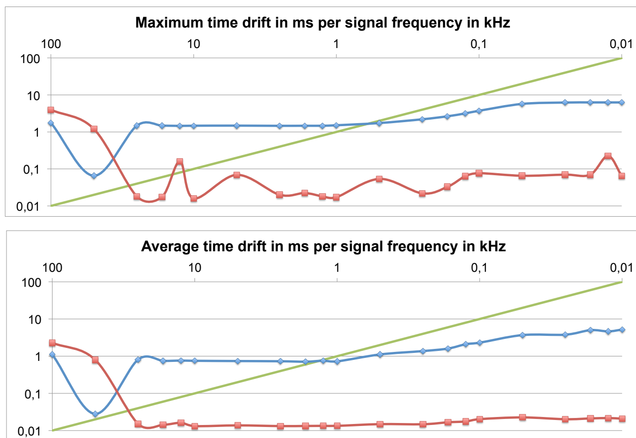
Time drift control. As detailled throughout, the timed IO monad is designed in such a way that, when running a complex timed action, the time drift remains *positive*, *documented* and somewhat *controlable* thanks to the design of delay actions. Of course, we have not yet offered any mechanism that can *ensure* that the time drift is bounded. Yet, we expect that the notion of timed monad and reference is formal enough to allow for developing some type system for such a purpose.

Garbage collected signals. Lifting timed IO actions to timed IO streams, we eventually recover a fairly robust notion of timed signal which can safely be used thanks to the embedding of signal continuations into timed IO actions. Most importantly, the garbage collector capacity to get rid of unused pieces of timed signals is preserved, just as any monad

action is collected when non longer referenced, in the GC sense, at runtime.

Performance tests. The schema below gives an overview of the time accuracy one can achieve under ghc using our library.

This performance test consisted in running timed streams at certain frequency (on the horizontal logarithmic scale) that accumulates all measured time drift before displaying the average or maximum timedrift (on the vertical logarithmic scale). It was run on an average laptop. The diagonal curve maps each frequency to its corresponding time period. The blue curve (starting below the other one) describes the



measured timed drifts when using the default *ghc* timer, implemented using *threadDelay*. The red curve (starting above the other) describes the measured timed drifts when using an active timer, implemented as an independant thread.

These performance tests show that with high frequency signal, from 100 kHz to 60 kHz, the processor is saturated and the resulting timedrift, just function of the duration of the performance test, is way higher than the corresponding time period. The default timer is the more efficient for it requires less computational power. From 60 kHz, the active scheduler becomes about a thousand times more efficient than the default one on the average, though, as observed with maximum time drift, it remains sensitive to the processor usage at any frequency.

The frequencies between 100 Hz and 20 Hz are the frequencies at which realtime signal audio processing can be performed, with a latency from 10ms to 50ms, hardly noticeable by human hears. This suggests that, with both timers, realtime audio processing can be performed in Haskell; a possibility we shall soon experiment as a real timed system programing test case.

The need for safe interruptions. As probably noticed by readers well aware of concurrent programming features, we have not considered interruption handling. The main reason for such an omission is that designing an interruption mechanism safely applicable to temporal media and, beyond, automation, is far from a mere lifting of the underlying runtime or OS interruption system. Indeed, with application as simple as MIDI music playing, a safe interruption of a music stream necessitates to stop by sending a *note off* event every note started with a *note on* event that is still running. Also, in a context of timed system programing, new interruption type such as a *change speed* signal would certainly be of interest though leading to hybrid system programing as discussed below.

Towards hybrid system programming. One can observe that the proposed notion of timer is, by itself, a simple hybrid signal, that maps real time to expected time, much like a metronome does in music. Saying so, one missing feature of our proposal appears : the capacity to change the underlying tempo; although, doing so would turn our notion of timer into fairly general hybrid signal, mapping the *real time scale*, as handled by the underlying OS, to some *symbolic time scale*, now handled by every timed program.

This suggests that our proposal could be extended towards hybrid system modeling, extending the proposed notion of timer into some notion of hybrid signal handler, equipped with all a menagerie of hybrid signal monitoring and controlling functions. Then, as a related open question, one may ask how the yet quite implicit algebraic framework induced by our application oriented approach could possibly be related with the recent theoretic proposal of hybrid system modeling by hybrid monads [21].

Acknowledgments

This work benefited from author's reading of various Haskell libraries code including *MVar* and *async* by Simon Marlow and *threads* by Bas and Roel van Dijk, all available on *Hackage*. My great thanks also goes to Simon Archipoff, for his help setting up the alternative for a timer code, and Bernard Serpette for various application examples suggestions and thorough discussions on the topic of the present paper.

References

[1] S. Archipoff and D. Janin. 2016. Structured Reactive Programming with Polymorphic Temporal Tiles. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM Press, 29–40.
[2] P. Caspi and M. Pouzet. 1996. Synchronous Kahn Networks. In *Int. Conf. Func. Prog. (ICFP)*. 226–238.
[3] P. Caspi and M. Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. *Electr. Notes Theor. Comput. Sci.* 11 (1998), 1–21.
[4] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. 2004. Towards a higher-order synchronous data-flow language. In *Int. Conf. On Embedded Software (EMSOFT)*. ACM, 230–239.
[5] J.-L. Colaço and M. Pouzet. 2003. Clocks as First Class Abstract Types. In *Int. Conf. On Embedded Software (EMSOFT)*. ACM, 134–155.
[6] C. Elliott and P. Hudak. 1997. Functional Reactive Animation. In *Int. Conf. Func. Prog. (ICFP)*. ACM.

[7] C. M. Elliott. 2009. Push-pull functional reactive programming. In *Symp. on Haskell*. ACM, 25–36.

[8] P. Hudak. 2008. *A Sound and Complete Axiomatization of Polymorphic Temporal Media*. Technical Report RR-1259. Department of Computer Science, Yale University.

[9] P. Hudak and D. Janin. 2014. Tiled Polymorphic Temporal Media. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM Press, 49–60.

[10] J. Hughes. 2005. Programming with Arrows. In *Advanced Functional Programming (AFP) (LNCS)*, Vol. 3622. Springer, 73–129.

[11] D. Janin. 2018. Spatio-temporal domains: an overview. In *Int. Col. on Theor. Aspects of Comp. (ICTAC) (LNCS)*, Vol. 11187. Springer-Verlag, 231–251.

[12] A. Jeffrey. 2014. Functional Reactive Types. In *IEEE Symp. on Logic in Computer Science (LICS)*. ACM.

[13] W. Jeltsch. 2014. An Abstract Categorical Semantics for Functional Reactive Programming with Processes. In *Workshop on Programming Languages Meets Program Verification*. ACM, 47–58.

[14] S. Peyton Jones, A. Gordon, and S. Finne. 1996. Concurrent Haskell. ACM, New York.

[15] N. R. Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. In *Int. Conf. Func. Prog. (ICFP)*.

[16] X. Liu and E. A. Lee. 2008. CPO semantics of timed interactive actor networks. *Theoretical Computer Science* 409, 1 (2008), 110 – 125.

[17] X. Liu, E. Matsikoudis, and E. A. Lee. 2006. Modeling timed concurrent systems using generalized ultrametrics. In *Conf. on Conc. Theory (CONCUR) (LNCS)*, Vol. 4137. Springer.

[18] L. Mandel and M. Pouzet. 2005. ReactiveML, a Reactive Extension to ML. In *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. ACM.

[19] S. Marlow. 2012. Parallel and Concurrent Programming in Haskell. In *4th Summer School Conference on Central European Functional Programming School (CEFP'11)*. Springer-Verlag, 339–401.

[20] E. Moggi. 1991. A modular approach to denotational semantics. In *Category Theory and Computer Science (CTCS) (LNCS)*, Vol. 530. Springer-Verlag.

[21] Renato Neves. 2018. *Hybrid programs*. Ph.D. Dissertation. Minho University.

[22] H. Nilsson, J. Peterson, and P. Hudak. 2003. Functional Hybrid Modeling. In *Int. Symp. On Practical Aspects of Declarative Languages (PADL)*. 376–390.

[23] I. Perez and H. Nilsson. 2015. Bridging the GUI Gap with Reactive Values and Relations. In *Symp. on Haskell*. ACM, 47–58.

[24] A. van der Ploeg and K. Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Int. Conf. Func. Prog. (ICFP)*. ACM, 302–314.

[25] R. de Simone, J.-P. Talpin, and D. Potop-Butucaru. 2005. The Synchronous Hypothesis and Synchronous Languages. In *Embedded Systems Handbook*. CRC Press.

[26] P. Wadler. 1990. Comprehending Monads. In *Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, 61–78.

[27] D. Winograd-Cort and P. Hudak. 2014. Settable and non-interfering signal functions for FRP: how a first-order switch is more than enough. In *Int. Conf. Func. Prog. (ICFP)*. ACM, 213–225.