A Timed IO monad

David Janin

A Timed IO monad

David Janin

LaBRI, Bordeaux INP
University of Bordeaux
33 405 Talence France
janin@labri.fr

Abstract. Programming with explicit timing information is often tedious and error prone. This is especially visible in music programming where, when played, the specified durations of notes and rests must be shortened in order to compensate the actual duration of all surrounding processing. In this paper, we develop the notion of timed extension of a monad that aims at relieving programmers from such a burden. We show how, under simple conditions, such extensions can be built, and how useful features of monad programming such as asynchronous concurrency with promises or data-flow programming with monadic streams can be uniformly lifted to the resulting timed programming framework. Even though presented and developed in the abstract, the notion of timed extension of a monad is nevertheless illustrated by two concrete instances: a default timed IO monad where programmers specify durations in microseconds, and a musically timed IO monad, where programmers specify durations in number of beats, the underlying tempo, that is, the speed of the music in beats per minute, possibly changed whenever needed.

1 Introduction

Timed programming. The simplest example of timed programming is probably a program that plays some music. Assume a function $f :: Int \to Note$ that tells which note is to be played at any instant n from start. Assume that we want to play each of these notes for one second, with one second of silence (or rest) between each note. With duration arguments expressed in seconds, in Haskell's IO monad, one could expect that the program:

$playMusic :: (Int \to Notes) \to Int \to IO\ ()$
$playMusic\ f\ n = \mathbf{do}\ \{playNote\ (f\ n)\ 1; delay\ 1; playMusic\ f\ (n+1)\}$

realizes such an expected behavior when launched by $playMusic\ f\ 0$. While this program *should* be correct, for it is defined with the correct timing specification, it is *actually not*, for it relies on the false assumption that every other computations but those that are specified with a given duration are instantaneous.

More precisely, one can define the *time drift* of this program as the difference between the actual instant a note is played and the specified instant it should have been played. Then one can observe that, when ran, the time drift of the program above is unbounded. Indeed, it increases, note after note, at least by

the actual duration of the computation of each note. In other words, such a program, when run in the IO Monad yields a *time leak*. Playing correctly such a music requires at least reducing the *specified* duration of each note or rest by the *actual* duration of the surrounding computations, implicitly but wrongly assumed to be of neglectable duration, not mentioning the time inaccuracy of actions such as delays.

Though simple, the explicit programming of such reductions in order to achieve a correct scheduling in time is repetitive, tedious and therefore error prone. We aim at relieving programmers from such a burden. One way to achieve this is to treat the program above as correct, for it specifies correct *specified durations*, and, instead, to change the way it is executed at runtime, that is, to change the underlying monad.

A timed IO monad. It is a well established fact that, thanks to monad modeling, pure functional programming can be safely extended to programming with side effects [19, 15]. Observing that *passing time* is a *side effect*, we thus aim at designing some timed monad that freely allows the programmer to assume that many actions are instantaneous, even if they certainly are not, and assume that all other actions have a specified positive duration, even if their actual durations will be shortened for achieving a correct scheduling.

This can be done by extending monads to timed monads in such a way that there is a clean distinction between:

(1) the *specified temporal scheduling*, automatically derived from the *specified durations* of the timed actions contained in a program,
(2) the *actual temporal scheduling* observed when running that program in the underlying monad, induced by the *actual durations* of these timed actions,

in such a way that, if possible and within reasonable bound, the actual temporal scheduling matches the specified temporal scheduling.

Observe that such a distinction between specified and actual temporal scheduling is fairly standard in music. The specified scheduling is described in the music score, as written by the composer, the actual scheduling is observed during the music performance, as defined by the musicians.

Organization of the paper. The notion of timed extension of a monad is defined in Section 2 via a type class and some equational properties every instance of that class shall satisfy. We also discuss the validity of these equations, fairly sensitive in presence of measured time.

In Section 3, we show how a monad can be uniformly extended into a timed monad as soon as it is equipped with basic timing primitives. This is achieved by extending the (implicit) monad state by an (explicit) timestamp that refers to the *expected* or *specified* timestamp in that state. Under simple conditions of the existing basic timing primitives, the *time drift* is provably positive in any state. Applied to the IO monad, this yields a timed IO monad with duration measured in microseconds.

Considering multi-scales approach, we provide in Section 4 another uniform timed extension of a monad, with basic timing primitives defined over one du-

ration type, that yields a timed monad extension defined over another duration type. Applied to the IO monad, this yields a musically timed IO monad with duration measured in number of beats, a dynamically changeable tempo defining the beat rate w.r.t. to the underlying physical duration type.

Last, we show how various monadic programming features, when available in the underlying monads, can be lifted to their timed extensions. This includes asynchronous concurrent programming with promises (Section 5), or data-flow programming with monadic streams (Section 6). Related works are then discussed in Section 7 before examining some potential followups in Section 8.

2 Timed monad class

We describe below the notion of timed extension of a monad. The monads we shall consider are *assumed to be strict*, that is, when executing a bind $m \gg\!= f$, the action m is always evaluated before evaluating f on the returned value, i.e. the bind is strict in its first argument.

2.1 Timestamp, duration, time scale and time drift

We briefly review here (our encoding of) basic timed concepts. 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*. In Haskell, this is done by putting:

$$\textbf{newtype}\ \mathit{Time}\ d = \mathit{Time}\ d\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})$$

where d is the duration type and $\mathit{Time}\ d$ is the timestamp type. While the sum of two durations makes perfect sense, the sum of two timestamps does not, so we (only) equip timescales with the following primitives:

$$\begin{aligned}
&\mathit{duration} :: \mathit{Num}\ d \Rightarrow \mathit{Time}\ d \to \mathit{Time}\ d \to d\\
&\mathit{duration}\ (\mathit{Time}\ d_1)\ (\mathit{Time}\ d_2) = (d_1 - d_2)\\
&\mathit{shift} :: \mathit{Num}\ d \Rightarrow \mathit{Time}\ d \to d \to \mathit{Time}\ d\\
&\mathit{shift}\ (\mathit{Time}\ d_1)\ d_2 = \mathit{Time}\ (d_1 + d_2)
\end{aligned}$$

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

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 realization.

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, the 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.

2.2 Timed monad

Simply said, a timed monad is a monad where every action has some specified (possibly dynamic) duration. The interface of a timed monad is detailed by the following type class:

$$\begin{aligned}&\textbf{class}\ (\mathit{Ord}\ d, \mathit{Num}\ d, \mathit{Monad}\ m, \mathit{Monad}\ t)\\&\quad \Rightarrow \mathit{TimedMonad}\ m\ d\ t \mid t \to m, t \to d\ \textbf{where}\\&\quad \mathit{now} :: t\ (\mathit{Time}\ d)\\&\quad \mathit{drift} :: t\ d\\&\quad \mathit{delay} :: d \to t\ ()\\&\quad \mathit{lift} :: m\ a \to t\ a\\&\quad \mathit{run} :: t\ a \to m\ a\end{aligned}$$

that describes a timed monad $t :: * \to *$ that extends a monad $m :: * \to *$ with duration measured over some type $d :: *$. There, the functional dependencies $t \to m$ and $t \to d$ ensure that both the initial monad m and the duration space d are uniquely determined by the timed monad t.

The meaning of these primitives is detailed below. Let us mention however that *now* shall return the current *specified* timestamp, *drift* shall return the current time drift, the actual or real timestamp being defined by the following derived action:

$$\begin{aligned}&\mathit{realNow} :: \mathit{TimedMonad}\ m\ d\ t \Rightarrow t\ (\mathit{Time}\ d)\\&\mathit{realNow} = \textbf{do}\ \{\, t \leftarrow \mathit{now}; d \leftarrow \mathit{drift}; \mathit{return}\ (\mathit{shift}\ t\ d)\,\}\end{aligned}$$

insisting again, if ever needed, on such a crucial distinction we are making between specified and actual timing information.

We provide below some equational laws that every timed monad instance shall satisfy. For such laws to be stated smoothly enough, we define the following timed monad action that, parameterized by a timed action m, returns its *specified* duration, that shall always be *positive*.

$$\begin{aligned}&\mathit{dur} :: \mathit{TimedMonad}\ m\ d\ t \Rightarrow t\ a \to t\ d\\&\mathit{dur}\ m = \textbf{do}\ \{\, t_0 \leftarrow \mathit{now}; \_ \leftarrow m; t_1 \leftarrow \mathit{now}; \mathit{return}\ (\mathit{duration}\ t_1\ t_0)\,\}\end{aligned}$$

Observe that computing such a specified duration implies running the action together with its side-effects but dropping its returned value. This means that, in practice, it shall be of little use. We will later see, in Section 5, another way to retrieve the specified duration of a running timed action for using it elsewhere in a program.

2.3 Timed monad laws

The semantics of timed primitives is detailed more formally by the following invariant laws that shall be satisfied by any monad m extended into a timed monad t over a duration type d. The reader shall keep in mind that *dur* measures *specified* durations, not real ones, most of the laws being obviously false when precise enough real durations are considered.

Monad primitives. First, since the timed monad t is first a monad, the usual monad laws shall be satisfied:

$$return\ a \gg\!= f \equiv f\ a \tag{1}$$
$$m \gg\!= return \equiv m \tag{2}$$
$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g) \tag{3}$$

with the following duration laws for monad primitives:

$$dur\ (return\ a) \equiv return\ 0 \tag{4}$$
$$dur\ (m \gg m') \equiv dur\ m \gg\!= \lambda d \to dur\ m' \gg\!= \lambda d' \to return\ (d + d') \tag{5}$$

for every value $a :: a$, action $m :: t\ a$, action $m' :: t\ b$ and function $f :: a \to t\ b$. In other words, 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. As a derived law, since $fmap\ f\ m = m \gg\!= (return \circ f)$, we also have:

$$dur\ m \equiv dur\ (fmap\ f\ m) \tag{6}$$

for every action $m :: t\ a$ and function $f :: a \to b$. In other words, in a timed setting, functors preserve specified durations, time measurement acting over types as a fibration [10].

Current (specified) time and drift. The action *now* shall instantaneously return the current specified timestamp as evaluated by accumulating the specified durations of the action performed before that one. The action *drift* shall instantaneously return the current time drift, that is, the difference between the actual timestamp (as measured by the underlying runtime) and the specified timestamp (as stored in the underlying timed monad state). By instantaneous, we mean that the following equations shall be satisfied:

$$dur\ (now) \equiv return\ 0 \tag{7}$$
$$dur\ (drift) \equiv return\ 0 \tag{8}$$

These equations also imply that neither *now* nor *drift* have any side effect.

Delays. The action $delay\ d$ shall wait until the current specified timestamp (as returned by *now*) shifted by the given positive duration d is eventually passed *for real* therefore, in optimal cases, reducing the time drift to a minimal value. The following laws shall be satisfied:

$$dur\ (delay\ d) \equiv delay\ d \gg return\ d \tag{9}$$
$$delay\ (d_1 + d_2) \equiv delay\ d_1 \gg delay\ d_2 \tag{10}$$
$$delay\ (-d) \equiv return\ () \tag{11}$$

for every *positive* duration $d\ d_1\ d_2 :: d$. The first law states that the specified duration of $delay\ d$ is the parameter d, the second one states that *delay* restricted

to positive durations is additive with respect to bind, the third one states that delays with negative durations have no effects at all. As we shall observe in 5.3, as safer assumption could be that a delay of negative duration creates a *temporal causality error*.

Instantaneous lift. The function *lift* shall turn an action of the underlying monad m to an action in the timed monad t with the following laws that shall be satisfied:

$$lift \circ return \equiv return \tag{12}$$

$$lift\ (m \gg\!= f) \equiv lift\ m \gg\!= (lift \circ f) \tag{13}$$

$$dur\ (lift\ (m)) \equiv lift\ m \gg return\ 0 \tag{14}$$

for any $m :: m\ a$ and $f :: a \to m\ b$. The first two laws are the usual laws for monad transformers. Any instance $(TimedMonad\ m\ d\ t)$ *is* a transformation of the monad m into the monad t. However, we have not specified here a monad transformer since only specific monads, equipped with some timing primitives, can be transformed this way. The third law states that, by *definition*, the specified duration of the timed action $lift\ m :: t\ a$ with $m :: m\ a$ is zero, *regardless* of the actual duration of the action m.

In practice, this assumption means that *lift* shall only be used on actions that are reasonably instantaneous, e.g. *printChar c* in the IO monad, but should not be used on actions that are visibly not instantaneous, e.g. *getChar* in the IO monad as this would immediately yield an unbounded time drift. As we shall see below a timed lift function is available for that latter case.

Runs. The function *run* allows for moving a timed action back into the underlying untimed monad with:

$$run \circ lift \equiv id \tag{15}$$

i.e. *lift* preserves the essence of the actions it lifts. Observe that over timed actions, the reverse direction does not hold since we have

$$lift\ (run\ m) \not\equiv m \tag{16}$$

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

Timed lift. Deriving from these primitives, we can lift any monad action from the underlying monad to the timed monad taking into account its actual duration[1] by:

$timedLift :: TimedMonad\ m\ d\ t \Rightarrow m\ a \to t\ a$
$timedLift\ m = \mathbf{do}\ \{\, a \leftarrow lift\ m; d \leftarrow drift; delay\ d; return\ a \,\}$

Such a timed lifting is then applicable to visibly non-instantaneous such as blocking actions, e.g. *getChar* in the IO monad.

[1] A careful reading of this code shows that the resulting specified duration of a timed lifted action is, more precisely, the actual duration of its execution minus the existing time drift right before its execution.

2.4 On the validity of timed monad extensions

One may wonder if there exists any timed extension of a monad at all that fulfills the properties stated above. Strictly speaking, with unrestricted usage of *drift* combined with an accurate measurement of physical time, the answer is no ! Indeed, given two distinct but equivalent timed actions m_1 and m_2, we have:

$$m_1 \gg drift \not\equiv m_2 \gg drift \tag{17}$$

unless m_1 and m_2 have the same *actual* duration, which is very unlikely.

This suggests that, unless measuring time drift for testing purposes, the function *drift* shall not be used. However, such a suggestion is not applicable for, as seen above, the function *timedLift*, necessarily defined with *drift*, is needed for lifting monad action with unbounded real duration. In other words, when extending an untimed monad into a timed one, there necessarily are functions such as *timedLift* that do not preserve (untimed) monad action equivalence. This implies that the validity of all timed equations but (15) shall only be observed in a timed setting. In some sense, timed and untimed worlds shall be kept distinct and incomparable, each of them being equipped with its own induced action equivalence.

3 Default timed monad instances

We provide below a default instance of a timed extension of a monad that can be defined as soon as that monad admits enough primitives for time handling.

3.1 Monads with timer

Monads with timing informations are defined by the following class type:

$$\begin{aligned}
&\textbf{class}\ (Ord\ d, Num\ d, Monad\ m) \Rightarrow HasTimer\ m\ d\ \textbf{where}\\
&\quad getRealTime :: m\ (Time\ d)\\
&\quad waitUntil :: Time\ d \to m\ ()\\
&\quad getDrift :: (Time\ d) \to m\ d\\
&\quad getDrift\ t = \textbf{do}\ \{r \leftarrow getRealTime; return\ (duration\ r\ t)\}
\end{aligned}$$

where *getRealTime* shall return the real timestamp measured over the duration type d, *waitUntil* shall wait until the specified time stamps is passed (for real), and the derived action *getDrift* shall therefore compute the difference between the real current timestamp and the one passed in parameter. Any monad with timing information shall satisfy the following properties:

(1) Time monotonicity: for every action $m :: m\ a$, the action

$$getRealTime \gg\!= \lambda t_1 \to m \gg getRealTime \gg\!= \lambda t_2 \to return\ (t_0, t_1)$$

shall return (t_1, t_2) with $t_1 \leqslant t_2$,

(2) Coherent waits: for every timestamp t_1 the action

$$waitUntil\ t_1 \gg getRealTime$$

shall return t_2 with $t_1 \leqslant t_2$.

The first property states that time shall flow from the past to the future. The second one states that a *waitUntil* action shall never resume before the expected timestamp is actually passed *for real*.

The IO example. As an instance example, thanks to the *System.Clock* and *Control.Concurrent* libraries in Haskell, one can put:

```
newtype Micro = Micro Int deriving (Show, Eq, Ord, Num)

getSystemTime :: IO (Time Micro)
getSystemTime = do {t ← getTime Monotonic;
  (return ∘ Time ∘ fromInteger) (div (toNanoSecs t) 1000)}

instance HasTimer IO Micro where
  getRealTime = getSystemTime
  waitUntil (Time d) = do {Time r ← getSystemTime;
    (threadDelay ∘ fromInteger ∘ toInteger) (d − r)}
```

where *Micro* is a type of durations measured in microseconds.

3.2 Derived timed monad instance

Deriving a timed monad instance from a monad with timing information can then be achieved by extending the (implicit) monad state by an explicit timestamp. More precisely, we define the timed action data type:

```
data TA m d a = TA (Time d → m (Time d, a))
```

over a monad $m :: * \to *$ and a duration type $d :: *$, from which we derive:

```
instance (Monad m, HasTimer m d) ⇒ Monad (TA m d) where
  return a = TA (λs → return (s, a))
  TA m >>= f = TA (λs → m s >>= λ(s1, a) → let (TA m1) = f a in m1 s1)
```

and

```
instance (Monad m, HasTimer m d)
      ⇒ TimedMonad m d (TA m d) where
  now = TA (λs → return (s, s))
  drift = TA $ λs → getDrift s >>= λd → return (s, d)
  delay d | d ⩽ 0 = return ()
  delay d | d > 0 = TA $ λs → do
    {dr ← getDrift s; waitUntil (shift s (d − dr)); return (shift s d, ())}
  lift m = TA $ λs → m >>= λa → return (s, a)
  run (TA m) = getRealTime >>= m >>= λ(_, a) → return a
```

This eventually provides the expected default timed monad extension of a monad m with timing information.

The correctness of such a construction, that is, the fact that laws (1)– (15) are satisfied by *TA m d* under the restriction described in 2.4, can easily be proved from the above code and the hypothesis made on m, d and t. More precisely, the standard monad laws follow directly from the fact that *TA m d* is a simple variation on a classical state monad transformer. Thanks to property (1) assumed for *getRealTime*, durations are always positive. Then, timed laws follow from the way all the above defined functions act on timestamps.

3.3 More on temporal correctness issues

One can observe that *run* initializes the time drift with a positive (if not zero) value since the initial specified timestamp is set to the actual timestamp. Thanks to property (2) on *waitUntil*, one can also observe that delays always resume after the specified timestamp is actually passed for real. It follows that the time drift after a delay is always positive. Since every other primitive timed action has an actual duration greater than its specified duration, it follows that:

(1) *the time drift is always positive,*

as easily proved by induction on the syntactic complexity of timed monad actions built from timed monad primitives. In other words, the action scheduling in the default instance is made in such a way that no action is actually scheduled before its specified scheduling time.

Temporal correctness also requires that such a time drift is bounded. Here, we can only observe that, obviously, in the general case:

(2) *nothing ensures the time drift is bounded.*

Indeed, as already mentioned, lifting a blocking IO action as an instantaneous one immediately induces an unbounded time drift. We shall discuss such an issue in the conclusion.

The IO example. As a particular case, the default timed extension of the IO monad, we call TIO, is simply defined by:

type *TIO* = *TA IO Micro*

with the instance *TimedMonad IO Micro TIO* deriving from the above instance *HasTimer IO Micro*.

4 Symbolic timed extension of a monad

We consider now the case of a time scale for the programmer distinct from the timescale of the underlying monad. More precisely, given an inner time scale, e.g. the physical time, measured by some inner duration type i, we aim at offering a symbolic timescale measured by some outer duration type o. This requires having some type s for measuring the possible speed (or tempo) of outer durations w.r.t. to inner durations.

4.1 Inner and outer durations with tempi

The relationship between duration types i, o and tempo s, when time speed is assumed to be piecewise constant, is conveniently modeled by the following type class:

$$\begin{aligned}&\textbf{class}\ (\mathit{Num}\ i, \mathit{Num}\ o, \mathit{Num}\ s) \Rightarrow \mathit{ScaleChange}\ i\ o\ s \mid s \to i, s \to o\ \textbf{where}\\ &\quad \mathit{initialSpeed} :: s\\ &\quad \mathit{step} :: s \to i \to o\\ &\quad \mathit{backStep} :: s \to o \to i\end{aligned}$$

where *initialSpeed* is some fixed initial speed value, $\mathit{step}\ s\ i$ that essentially computes the outer duration obtained by "multiplying" the speed s by the inner duration i, and $\mathit{backStep}\ s\ o$ that essentially computes the inner duration obtained by "dividing" the outer duration o by (some non zero) speed s.

The way these "multiplication" and "division" are actually performed shall depend on the chosen types. In the abstract, the following equations shall be satisfied by any instances:

$$\mathit{backStep}\ s\ (\mathit{step}\ s\ i) \equiv i \tag{18}$$

$$\mathit{step}\ d\ (\mathit{backStep}\ d\ o) \equiv o \tag{19}$$

for any inner duration i, outer duration o and non zero speed s, up to the possible rounding errors due to changes of numerical types.

As a consequence, the function mapping the inner timescale $\mathit{Time}\ i$ to the outer timescale $\mathit{Time}\ o$ shall be bijective (up to rounding) and, in case *step* and *backStep* functions are truly implemented as some multiplication and division, piecewise linear.

The IO example. As an instance example, one can define:

$$\begin{aligned}&\textbf{newtype}\ \mathit{Beat} = \mathit{Beat}\ \mathit{Double}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show}, \mathit{Num})\\ &\textbf{newtype}\ \mathit{BPM} = \mathit{BPM}\ \mathit{Double}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show}, \mathit{Num})\\ &\textbf{instance}\ \mathit{ScaleChange}\ \mathit{Micro}\ \mathit{Beat}\ \mathit{BPM}\ \textbf{where}\\ &\quad \mathit{initialSpeed} = \mathit{BPM}\ 60\\ &\quad \mathit{step}\ (\mathit{BPM}\ t)\ (\mathit{Micro}\ d) = \mathit{Beat}\ \$\ t * ((\mathit{fromInteger} \circ \mathit{toInteger})\ d) / \mathit{ratio}\\ &\quad \mathit{backStep}\ (\mathit{BPM}\ t)\ (\mathit{Beat}\ d) = \mathit{Micro}\ \$\ \mathit{fromInteger}\ (\mathit{floor}\ (d * \mathit{ratio} / t))\end{aligned}$$

with $\mathit{ratio} = 60 * 10^6$, the time speed being expressed in beats per minutes (bpm).

4.2 Derived symbolic timed monad instance

A symbolic timed extension of a monad can then be built quite like the default timed extension described above. Indeed, we define symbolic timed states by:

$$\textbf{data}\ \mathit{ST}\ i\ o\ s = \mathit{ST}\ \{\mathit{innerTime} :: \mathit{Time}\ i, \mathit{outerTime} :: \mathit{Time}\ o, \mathit{speed} :: s\}$$

with symbolic timed actions defined by:

$$\textbf{data}\ \mathit{STA}\ m\ i\ o\ s\ a = \mathit{STA}\ (\mathit{ST}\ i\ o\ s \to m\ (\mathit{ST}\ i\ o\ s, a))$$

This eventually yields the instance $TimedMonad\ m\ o\ (STA\ m\ i\ o\ s)$, defined essentially like the instance of $TA\ m\ d$, taking care however to maintain coherent inner and outer timestamps in every symbolic timed state. This can be done without any real difficulty, laws (18)–(19) ensuring, for a given time speed s, coherent back and forth translation between duration types i and o.

The IO example. As a particular case, the promised musically timed extension of the IO monad is defined by:

$$\textbf{type}\ MusicIO = STA\ IO\ Micro\ Beat\ BPM$$

The underlying tempo can be changed at any time by the following parameterized timed action:

$$\begin{aligned}
&setTempo :: BPM \to MusicIO\ () \\
&setTempo\ t \mid t \leqslant 0 = error\ \$\ \texttt{"setTempo : forbidden negative tempo"} \\
&setTempo\ t \\
&\quad = STA\ \$\ \lambda s \to \textbf{let}\ ST\ ti\ to\ _ = s\ \textbf{in}\ return\ (ST\ ti\ to\ t, ())
\end{aligned}$$

Given function $f :: Int \to Note$, our initial example can then simply and correctly be encoded by:

$$\begin{aligned}
&playInIO = run\ playMusic \\
&playMusic :: (Int \to Notes) \to Int \to MusicIO\ () \\
&playMusic\ f\ n \\
&\quad = \textbf{do}\ \{lift\ (playNote\ (f\ n))\ 1; delay\ 1; playMusic\ f\ (n+1)\} \\
&playNote :: Note \to Beat \to MusicIO\ () \\
&playNote\ n\ d = startNote\ n \gg delay\ d \gg stopNote\ n
\end{aligned}$$

By construction, the tempo has been initialized to 60 bpm, that is, one beat per second.

5 Timed promises

One may ask how robust our constructions of timed monads are, or, more precisely, to which extent additional features of a given monad can be lifted to its timed extension. We shall describe here the case of asynchronous concurrent promises that can uniformly be lifted from any monad where they are defined to its (default) timed extension when there is one.

5.1 Monad references

Since the 70s, there is the concept of promises that successfully extends functional programming to asynchronous concurrent features. Simply said, a promise is a place holder returned by a forked program that is eventually fulfilled by the value returned by that program [6].

In Haskell, the notion of promise is conveniently replaced by the notion of monad references [11] specified as follows:

```
class Monad m ⇒ MonadRef m where
  type Ref m :: ∗ → ∗
  fork :: m a → m (Ref m a)
  read :: Ref m a → m a
  tryRead :: Ref m a → m (Maybe a)
  parRead :: Ref m a → Ref m b → m (Either a b)
```

where the action *fork m* shall fork the monad action m and *immediately* returns a reference to that action, the action *read r* shall return the value produced by the running action referenced by r *as soon as* it is available, the action *tryRead r* shall be a non blocking version of *read r* and the action *parRead* r_1 r_2 shall take two monad references $r_1$and r_2 as parameters and return the value of the *first referenced action that terminates*, or either of the values if both are already terminated or are terminating at the same (or indistinguishable) time.

The basic (non-concurrent) semantics of monad reference basic primitives is governed by the following laws:

$$(fork\ m) \gg\!= read \equiv m \tag{20}$$

$$fork \circ read \equiv return \tag{21}$$

$$fork\ (m \gg\!= f) \equiv (fork\ m) \gg\!= \lambda r \to fork\ (read\ r \gg\!= f) \tag{22}$$

for every $m :: m\ a$, $f :: a \to m\ b$. Other laws, specifying the expected concurrent semantics of monad references are detailed in the companion article [11].

5.2 Timed monad references

Equipping a timed extension of monad by monad references, as soon as the underlying monad itself has references, is (almost) easy as shown by the following instance:

```
data TRef m d a = TRef (Time d) (Ref m (Time d, a))
instance (MonadRef m, HasTimer m d) ⇒ MonadRef (TA m d) where
  type Ref (TA m d) = TRef m d
  fork (TA m) = TA $ λs → do {r ← fork (m s); return (s, TRef s r)}
  read (TRef _ r) = TA $ λs → do {(t, a) ← read r; return (max s t, a)}
  tryRead (TRef _ r) = TA $ λs → do {c ← tryRead r; case c of
    Nothing → return (s, Nothing)
    Just (t, a) → return (max s t, Just a)}
  parRead (TRef _ r1) (TRef _ r2) = TA $ λs → do {c ← parRead r1 r2;
    case c of {Left (t, a) → return (max s t, Left a);
               Right (t, b) → return (max s t, Right b)}}
```

One can observe that in all read actions above, variable t refers to the (specified) timestamp at which the referenced action is eventually completed while variable s refers to the (specified) timestamp at which the read action is called. As these two timestamps refer to independant events, we need to compute the (specified)

timestamp $max\ s\ t$ right after which these two events have occured. Then, one can check that equations (20)–(22) are satisfied, even though this requires equivalent actions to also have the same (specified) duration.

Remark. With the above proposed instance, nothing ensures the function *parRead* returns the value of the soonest terminated action *as specified* by termination timestamps. Indeed, the function *parRead* is non deterministic in two cases, the first one being when two referenced actions are already terminated, the second one being when there are terminating *almost* at the same time. Clearly, the first case can easily be solved by performing a *tryRead* on the second reference right after the first one is received and, if terminated, sorting the returned values according to their timestamp. However, the second case is more tricky to solve. One solution is to ensure that function *tryRead* returns *Nothing* only in the case the referenced action *provably* terminates later than the specified time at which *tryRead* is launched. Such a possibility is however yet not implemented.

5.3 Time specific action on timed monad references

The reader may have observed that, in the code above, we do not use the timestamp recorded in a timed reference when forking an action. Its relevance appears in the function *durRef* given below, from which various timed specific actions are derived.

$$
\begin{aligned}
&durRef :: TRef\ m\ d\ a \to TA\ m\ d\ d\\
&durRef\ (TRef\ t_0\ r)\\
&\quad = TA\ (\lambda s \to \mathbf{do}\ \{(t, \_) \leftarrow read\ r; return\ (max\ s\ t, duration\ t\ t_0)\})\\
&replayRef :: TRef\ m\ d\ a \to TA\ m\ d\ a\\
&replayRef\ r = \mathbf{do}\ \{t_1 \leftarrow now; d \leftarrow durRef\ r; a \leftarrow read\ r; t_2 \leftarrow now;\\
&\qquad\qquad delay\ (d - duration\ t_2\ t_1); return\ a\}\\
&expandRef :: (d \to d) \to TRef\ m\ d\ a \to TA\ m\ d\ a\\
&expandRef\ f\ r = \mathbf{do}\ \{t_1 \leftarrow now; d \leftarrow durRef\ r; a \leftarrow read\ r; t_2 \leftarrow now;\\
&\qquad\qquad \mathbf{let}\ d_1 = f\ d - duration\ t_2\ t_1\ \mathbf{in}\ delay\ d_1; return\ a\}
\end{aligned}
$$

where *durRef* returns, when finished, the specified duration of a referenced action, *replayRef* replays the referenced action from start, with the same duration but no side effect, and *expandRef* replays a referenced action but expanding (or shrinking) its duration by appliying some function parameter f.

Observe that all these actions can be used as soon as their parameters are available therefore even before the referenced actions are terminated and there specified durations are known. This means that shrinking a duration may fail to be done correctly as illustrated by $fork\ m \gg\!= expandRef\ (\ /2)$ that unsucessfully tries to replay twice faster a just forked action. Such a resulting action is not temporal causal or, equivalently, duration d_1 in the code of *expandRef* is strictly negative hence no delay is applied. Executing a negative delay is here a clear sign of a causality error, an error that could well be raised as such.

As already observed in music experiments conducted along a previous modeling of interactive music by temporal tiles [1], programming interactive music

systems, these functions yield a completely new branch of realtime musical effects, such has repeating every note with its same duration, an effect that is essentially not available in the existing reactive music application software.

6 Data flow programming with timed monad streams

Defining data flows by means of nested monad actions is a technique that is getting more and more popular for data-flow-like programming within generic functional programming languages as illustrated by Snoyman's *Conduit* library. Following the (simpler) definition of monad streams recently (re)defined and developed for audio processing and control [12], there is the type constructor:

$$\textbf{newtype}\ \mathit{Stream}\ f\ a = \mathit{Stream}\ \{\mathit{next} :: f\ (\mathit{Maybe}\ (a, \mathit{Stream}\ f\ a))\}$$

either defining monad streams with $f = m$ for some monad m, or defining references to (running) monad streams with $f = \mathit{Ref}\ m$ for some monad m with references [11]. Then, as a derived function example, we have:

$$\begin{aligned}&\mathit{merge} :: \mathit{MonadRef}\ m\\ &\quad \Rightarrow \mathit{Stream}\ m\ a \rightarrow \mathit{Stream}\ m\ b \rightarrow \mathit{Stream}\ m\ (\mathit{Either}\ a\ b)\end{aligned}$$

that merges two monad streams by order of arrival of their elements. Applied to timed monad and timed monad references, these kind of functions have clear application in timed data flow programming, especially for handling asynchronous control flows as recently illustrated [12].

In other words, timed data flow programming automatically derives from timed monads and timed monad references. This somehow illustrates the fundational nature of these two concepts that both extend monads.

7 Related Works

In functional programming languages, there already are many proposals for programming timed reactive concurrent systems ranging from the synchronous language family [18], possibly extended with modern polymorphism as with Reactive ML [14], to the many variants of functional reactive program (FRP) series initiated with FRAN [3, 4, 20]. However, to the best of our knowledge, most of these approaches consider a qualitative timing, as defined by series of events, instead of a quantitative timing that the programmer may specify as proposed here.

More precisely, the synchronous language approach mostly aims at defining timed programs over symbolic time scales (series of clock ticks) for which the actual duration between any two ticks is provably bounded. This eventually led to the successful development of programming languages such as Lustre or Esterel [18] that allows programmers to implement (provably correct) synchronous realtime applications. Yet, programming timed applications with weaker but quantitative time specification seems to go out of the application scope of these languages, despite interesting extensions toward polymorphic and higher-order timing mechanisms [2].

The FRP approach, somewhat more theoretical, aims at defining an adequate API for programming with timed signal. Initially designed to cope with arbitrary timescale [3], concrete implementations of FRP are mostly limited to reactive programming, though with some exceptions [16]. In practice, timescales are defined by event arrivals, that act as qualitative clock ticks, function between signals being restricted to Mealy machines. There, the initially proposed API with signals defined as functions from timestamps to values yields (fairly easily) memory leaks which are avoided by such a restriction, either by syntactic means [17] or by modal type mechanisms [13]. Our timed extension of monads, with derived timed monad streams, brings back qualitative time measurement between clock ticks without the associated time leaks.

The multitime scale approach, presented in Section 4 is an example of a fairly simple (piece-wise linear and locally finite) hybrid (bijective) signal. Along the lines proposed in [16], it can probably be extended for defining more general hybrid signals, with their associated monitoring actions. Our approach is also influenced by Hudak's aim at defining temporal objects by means of the properties their combinators shall satisfy [7]. Though our proposal eventually diverges from Hudak's polymorphic temporal media, it nevertheless inherits from its underlying intention to bridge the gap between theory and practice, as already illustrated by Euterpea [9].

Last, the initial motivation for the present development of timed monads was to define an efficient and more generic programming interface for encoding temporal tiles [8, 1]. As a matter of fact, all interactive music experiments conducted with such a model are easily and more efficiently re-implemented within the proposed timed monad framework. In particular, experiments with music control conducted within the symbolic timed IO monad defined in these pages yield a time drift smaller than 5ms therefore a resulting latency unnoticeable by human hears.

8 Conclusion

Along these pages, we have proposed and instantiated a fairly generic notion of timed extension of a monad. We have also shown how additional programming features of monads can simply be lifted into these extension. This time extension relies on distinguishing specified and actual duration: a distinction already put in practice in interactive music systems, the musician and the computer interacting one with the other [1]. Of course, the topic is far from being closed, how to add timing information into interactive program being a vast subject. On the positive side, our proposal allows for clearly identifying what are the yet unsolved timing problems and where to look for solutions.

As already observed in 3.3, nothing ensures that the time drift is bounded in a timed monad action. The reader might be disappointed by such a fact. However, ensuring that a given program has a bounded time drift is a well-known tricky issue, undecidable in the general case. What restrictions on programs may yield timed actions with provably bounded time drift is a long standing

research problem [18]. Clearly, recursively defined action shall only be allowed when each recursive call is both terminal, in order to avoid a (stack) memory leak, and guarded by some sufficient long delays, in order to avoid a time leak. One possibility could be to extend modal types [13] with quantified durations. Existing results in quantitative temporal logic may help. Also, extending the underlying monad actions type by some bound on their actual durations might help as well. This would allow checking that actions are properly lifted with adequate delays. Indeed, as already observed, lifting a blocking action as an instantaneous action is nonsense. Similar techniques should probably be used for detecting temporally non causal transformations as illustrated in 5.3 example.

As also observed in 2.4, two actions in a base monad m can be equivalent while their timed lifting is not. Indeed, with durations measured in microseconds, it is very unlikely that two distinct but equivalent actions have the same duration. Real time measurement is a property killer. One partial solution might be to measure time drift with less accuracy. In a music system, with symbolic time, such a problem is well-known. It arises when aiming at translating back realtime performances into music scores, as in score followers [5]. Rephrased in terms of timed monad, the implementation of *drift* is the key to handling such a problem. Is there any time drift between two different musicians performing the same score ? Measurements say yes, but listeners say no. The measurement of the time drift should follow listeners.

Of course, developing programming languages towards application in music may sound a bit pointless compared to applications with more economical impact such as, say, autonomous vehicles or connected objects. Is that so ? Clearly, experiments are easier to conduct in music than in most other application fields, with less dramatic consequences in case of errors. Moreover, time is known and handled by musicians for centuries. As illustrated throughout, various musical concepts can be generalized or abstracted into useful timed programming concepts. Our approach is hopefully abstract enough to be potentially applicable to other areas. Timed programming, with its need for automatic handling of time drift, is surely in the close neighborhood of spacetime programming, with its need for automatic handling of spacetime drifts.

References

1. S. Archipoff and D. Janin. Structured reactive programming with polymorphic temporal tiles. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*, pages 29–40. ACM Press, 2016.
2. J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *Int. Conf. On Embedded Software (EMSOFT)*, pages 230–239. ACM, 2004.
3. C. Elliott and P. Hudak. Functional reactive animation. In *Int. Conf. Func. Prog. (ICFP)*. ACM, 1997.
4. C. M. Elliott. Push-pull functional reactive programming. In *Symp. on Haskell*, pages 25–36. ACM, 2009.

5. J.-L. Giavitto, J. Echeveste, A. Cont, and P. Cuvillier. Time, timelines and temporal scopes in the Antescofo DLS v1.0. In *International Computer Music Conference (ICMC)*, 2017.
6. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
7. P. Hudak. An algebraic theory of polymorphic temporal media. In *International Workshop on Practical Aspects of Declarative Languages (PADL)*, pages 1–15. Springer Verlag LNCS 3057, 2004.
8. P. Hudak and D. Janin. From out-of-time design to in-time production of temporal media. Research report, LaBRI, Université de Bordeaux, 2015.
9. P. Hudak and D. Quick. *The Haskell School of Music : From signals to Symphonies*. Cambridge University Press, 2018.
10. D. Janin. Spatio-temporal domains: an overview. In *Int. Col. on Theor. Aspects of Comp. (ICTAC)*, volume 11187 of *LNCS*, pages 231–251. Springer-Verlag, 2018.
11. D. Janin. An equational modeling of asynchronous concurrent programming. Technical report, LaBRI, Université de Bordeaux, 2019.
12. D. Janin. Screaming in the IO monad. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM, 2019.
13. N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Int. Conf. Func. Prog. (ICFP)*, 2013.
14. L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2005.
15. E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science (CTCS)*, volume 530 of *LNCS*. Springer-Verlag, 1991.
16. H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling. In *Int. Symp. On Practical Aspects of Declarative Languages (PADL)*, pages 376–390, 2003.
17. A. van der Ploeg and K. Claessen. Practical principled FRP: forget the past, change the future, FRPNow! In *Int. Conf. Func. Prog. (ICFP)*, pages 302–314. ACM, 2015.
18. R. de Simone, J.-P. Talpin, and D. Potop-Butucaru. The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook*. CRC Press, 2005.
19. P. Wadler. Comprehending monads. In *Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, 1990. ACM.
20. D. Winograd-Cort and P. Hudak. Settable and non-interfering signal functions for FRP: how a first-order switch is more than enough. In *Int. Conf. Func. Prog. (ICFP)*, pages 213–225. ACM, 2014.