A timed IO monad

David Janin
Bordeaux INP, UMR CNRS LaBRI,
University of Bordeaux
May 2019

The need for a timed monad

Music playing example

Plays every second *n* the note *s n* for half a second.

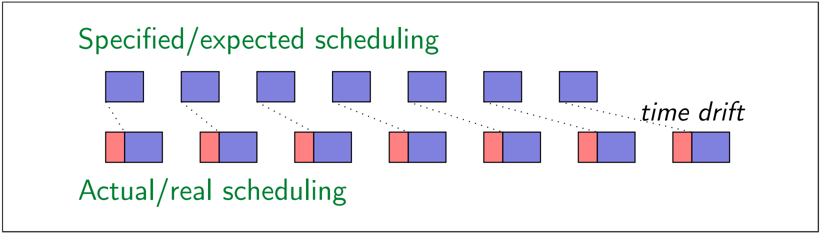
What is wrong with that code ?

```
perform :: Int → IO ()
perform n = do {play (s n) (5 ∗ 10ˆ5); perform (n + 1)}
   where
   play n d = noteOn n ≫ threadDelay d ≫
                    noteOff n ≫ threadDelay d
```

The need for a timed monad

Every computation takes some time,
every delay resumes too late

Specified vs actual scheduling



Time leak

Caused by a positive (good) but unbounded (bad) time drift defined, at any instant, by:

time drift = actual timestamp - specified timestamp

The need for a timed monad

General specifications

A timed monad shall provide:

1. specified positive or zero duration for each action,
2. induced expected scheduling of actions,
3. performed actual scheduling of actions,
4. tools to control/observe (positive) time drift, that is, the (miss)match between expected and actual scheduling,
5. an other useful features such as safe and robust IO, concurrency, . . .

Side remarks

1. passing time *is* a side effect,
2. every programmer *knows* monad programing style.

The need for a timed monad

Back to our exemple

Within the default timed IO monad, we would essentially write the same code:

```
perform :: Int → TIO ()
perform n = do {play (s n) (5 * 10^5); perform (n + 1)}
  where
  play n d = liftIO (noteOn n)        -- specified duration 0
               ≫ delay d               -- specified duration d
               ≫ liftIO (noteOff n)    -- specified duration 0
               ≫ delay d               -- specified duration d
```

but, in order to compensate the time drifts created by all other computations, the actual duration of *delay d* shorter than its specified duration of *d* microseconds.

Structure of the talk

Timed monad

Monad with references

Monad streams

Monad stream references

Conclusion

1. Timed monad

Timed programming with class

Timed Monad type class

```
newtype Time d = Time d deriving (Eq, Ord)
shift (Time t) d = Time (t + d)
duration (Time t1) (Time t2) = t1 − t2
```

with timestamp type (*Time d*) and duration type (*d*),

```
class (Monad m, MonadIO m, Num d, Ord d)
   ⇒ TimedMonad m d | m → d where
  runTIO :: m a → IO a    --
  delay :: d → m ()       -- negative d treated as zero
  now :: m (Time d)       -- current specified timestamp
  drift :: m d            -- current actual time drift
```

with all timed actions specified as instantaneous but *delay d* specified with duration *d*, . . . , and with *delay d* that resumes as soon as the new specified timestamp is actually passed for real.

Timed Monad type class

The specified duration of an action is computed by:

$$\begin{aligned} dur\ m = \textbf{do}\ \{ & t_0 \leftarrow now; \_ \leftarrow m; \\ & t_1 \leftarrow now; return\ (duration\ t_1\ t_0)\} \end{aligned}$$

In addition to superclasses laws, we shall have:

$$dur\ m \equiv dur\ (fmap\ f\ m) \tag{1}$$

$$dur\ (return\ a) \equiv return\ 0 \tag{2}$$

$$dur\ (m_1 \gg m_2) \equiv dur\ (m_1) \ggg= \lambda d \rightarrow fmap\ (d+)\ (dur\ m_2) \tag{3}$$

for every timed action m, m_1 and m_2, and

$$dur\ (liftIO\ io) \equiv liftIO\ io \gg return\ 0 \tag{4}$$

$$io \equiv (runTIO \circ liftTIO)\ io \tag{5}$$

for every IO action io, therefore $m \not\equiv (liftIO \circ runTIO)\ m$ whenever the timed action m has non zero duration.

Timed lifting of IO actions

As an example of a derived function:

A timed lifting of an IO action:

```
liftTimedIO :: TimedMonad m d ⇒ IO a → m a
liftTimedIO m = do
  a ← liftIO m      -- specified duration 0
  drift ≫= delay    -- specified duration : the actual time drift
  return a          -- here time drift is nearly zero
```

where, upon exit, the expected timestamp essentially matches the specified time stamp, since the actual duration of *drift ≫= delay* shall be nearly zero.

A timed IO monad instance

Define timed IO monad states as (implicit) IO monad states extended with (explicit) timestamp, therefore, timed IO actions by:

```
newtype TIO a = TIO (Time Integer → IO (Time Integer, a))
```

with expected functor, monad and monadIO instances:

```
instance Functor TIO where
  fmap f (TIO g) = TIO $ λs → do
    (t, a) ← g s
    return (t, f a)
instance Monad TIO where
  return a = TIO $ λs → return (s, a)
  (≫=) (TIO f) g = TIO $ λs → do
    (t, a) ← f s
    let TIO h = g a in h t
instance MonadIO TIO where
  liftIO m = TIO $ λs → m ≫= λa → return (s, a)
```

The timed IO monad instance

Then we put:

```
instance TimedMonad TIO Integer where
  now = TIO (λs → return (s, s))
  drift = TIO (λs → systemTime ≫=
                 λ r → return (s, duration r s))
  delay d = TIO $ λs → if (d ⩽ 0) then (return (s, ()))
    else $ do
      let t = shift s d               -- expected timestamp target
      r ← systemTime                  -- current actual timestamp
      threadDelay (duration t r)      -- t just passed for real
      return (t, ())
  runTIO (TIO f) = systemTime ≫= f ≫= return ∘ snd
```

with *systemTime* that returns system timestamp in micro seconds.

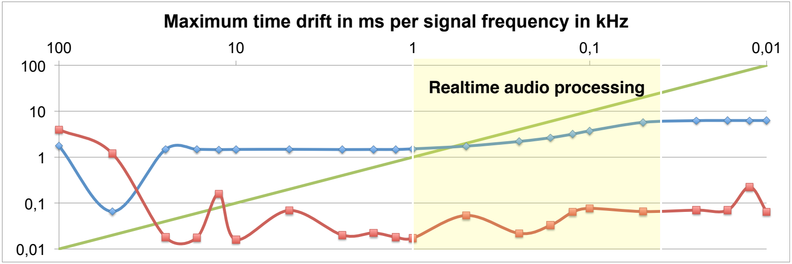
What we actually need for generic timed IO monads

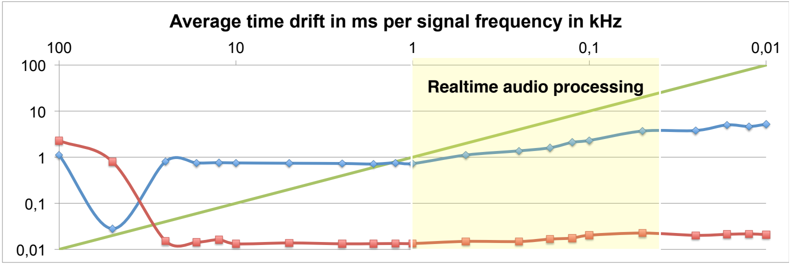
The class type: $Timer\ s\ d \mid s \rightarrow d$

that defines and binds together:

- a duration type d, with derived timestamps and time scale,
- a possible external scheduler,
- a timed state type s with embedded expected timestamp and scheduler handle,
- the associated runtime calls *systemTime*, *systemDelay* with (current) timed state argument,
- an initialization action *initialState*.

Timed IO monad instances (simple performance tests)





2. Monad with references

Safe communicating processes with class,

musing around ideas of Simon Marlow in Control.Concurrent.Async.

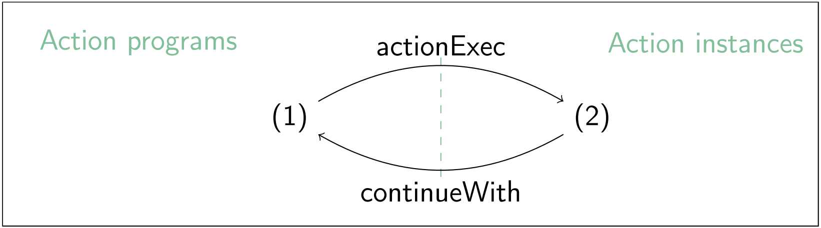
On monad programming

When programming within a monad, there are:

(1) action programs that can be freely reused, duplicated, in a pure functional programming style,

(2) action running instances that cannot be reused, duplicated, etc..., for they are uniquely located in the underlying monad state space.

In every strict monad, function $bind :: m\ a \to (a \to m\ b) \to m\ b$ (implicitly) encodes two “functions” *actionExec* and *continueWith* performed in this order:



Action programs

actionExec

Action instances

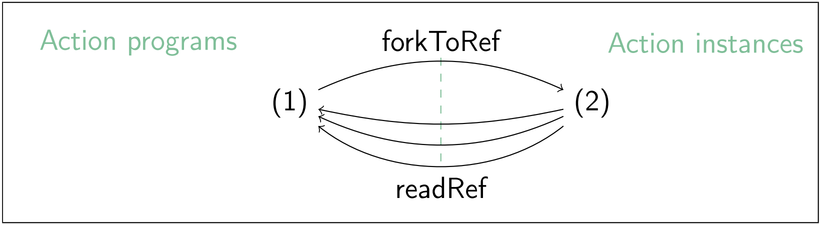
(1)

(2)

continueWith

Monad references

A monad reference shall be a broadcast channel from a uniquely associated running monad action, which can be freely read, in a safe and robust way.



with, generalizing *async* library, the following class type:

$$\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadRef}\ m\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Ref}\ m :: * \to * \\
\quad \mathit{forkToRef} :: m\ a \to m\ (\mathit{Ref}\ m\ a) \\
\quad \mathit{readRef} :: \mathit{Ref}\ m\ a \to m\ a \\
\quad \mathit{tryReadRef} :: \mathit{Ref}\ m\ a \to m\ (\mathit{Maybe}\ a) \\
\quad \mathit{parReadRef} :: \mathit{Ref}\ m\ a \to \mathit{Ref}\ m\ b \to m\ (\mathit{Either}\ a\ b)
\end{array}$$

Safely ? Robustly ?

Safely

Deadlock free communication schema for a process can only access the data produced by another if it “sees” its fork.

Robustly

No or harmless side effects when reading monad references for reading a process reference does not imply re-executing that process and its side effects.

Example in music (with actions replaced by streams of actions)

Given a playing music running process, the reference to that process could be the score of the played music. Reading the score does not imply replaying the music.

Monad reference laws

Monad with references instance shall satisfy the following laws:

- Basic semantics

$$m \equiv \mathit{forkToRef}\ m \ggg\!= \mathit{readRef} \tag{6}$$

- Idempotence

$$\mathit{readRef}\ r \equiv \mathit{readRef}\ r \gg \mathit{readRef}\ r \tag{7}$$

- Commutation

$$\begin{aligned}&\mathit{readRef}\ r_1 \ggg\!= \lambda x_1 \rightarrow \mathit{readRef}\ r_2\\ &\quad \ggg\!= \lambda\, x_2 \rightarrow \mathit{return}\ (x_1, x_2)\\ &\qquad \equiv \mathit{readRef}\ r_2 \ggg\!= \lambda x_2 \rightarrow \mathit{readRef}\ r_1\\ &\qquad\quad \ggg\!= \lambda x_1 \rightarrow \mathit{return}\ (x_1, x_2)\end{aligned} \tag{8}$$

Parallel reading

As a derived function:

$$\begin{aligned}&\mathit{parRun} :: \mathit{MonadRef}\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ (\mathit{Either}\ a\ b)\\ &\mathit{parRun}\ m_1\ m_2 = \mathbf{do}\ \{ r_1 \leftarrow \mathit{forkToRef}\ m_1; r_2 \leftarrow \mathit{forkToRef}\ m_2;\\ &\qquad\qquad\qquad\qquad \mathit{parReadRef}\ r_1\ r_2 \}\end{aligned}$$

with an induced race as illustrated by:

$$\mathit{parRun}\ (\mathit{return}\ \texttt{"foo"})\ (\mathit{return}\ \texttt{"foo"})$$

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

Timed IO references

Reference to timed IO actions are defined by:

```
newtype TIORef a =
   TIORef (Time Integer, MVar (Timed Integer, a))
```

where, in *TIORef* (s, v), there shall be the start timestamp s with the mutable variable v filled with stop timestamp and returned value upon termination of the referenced running timed action.

```
instance MonadRef TIO where
   type Ref TIO = TIORef
   readRef (TIORef (_, v)) = TIO $ λs → do
      (s₁, a) ← readMVar v
      return (max s s₁, a)
   forkToRef (TIO m) = TIO $ λs → do
      v ← newEmptyMVar
      _ ← forkIO (m s ≫= putMVar v)
      return (s, TIORef (s, v))
    ....
```

with some more lines of code for *parReadRef*. . .

Timed monad with references

Combining timed monad with monad references yields:

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

with:

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

Replaying a timed monad reference

As application example, one can replay a referenced timed monad action with same returned value and same duration by:

$$
\begin{array}{l}
\mathit{delayRef} :: \mathit{TimedMonadRef}\ m\ d \Rightarrow \mathit{Ref}\ m\ a \rightarrow m\ ()\\
\mathit{delayRef}\ r = \mathbf{do}\\
\quad t_0 \leftarrow \mathit{now}\\
\quad d \leftarrow \mathit{durRef}\ r\\
\quad t_1 \leftarrow \mathit{now}\\
\quad \mathit{delay}\ (d - \mathit{duration}\ t_1\ t_0)\\
\mathit{replayRef} :: \mathit{TimedMonadRef}\ m\ d \Rightarrow \mathit{Ref}\ m\ a \rightarrow m\ a\\
\mathit{replayRef}\ r = \mathit{delayRef}\ r \gg \mathit{readRef}\ r
\end{array}
$$

Reading, replaying or expanding a forked process reference

- Forked process: $r \leftarrow forkToRef\ (delay\ 5)$

- Read reference: $delay\ 2 \gg readRef\ r$

- Replay reference: $delay\ 2 \gg replayRef\ r$

- Expand reference: $delay\ 2 \gg expandRef\ (2*)\ r$

```
expandRef f r = do {
  t0 ← now; a ← readRef r; d ← durRef r; t1 ← now
  let d1 = f d − duration t1 t0 in case (d1 ⩾ 0) of
    True → delay d1
    False → liftIO (print "Non causal shrink")
  return a}
```

3. Monad streams

To achieve way more expressiveness

Higher-order streams

Lists extended with a type constructor for head/tail access.

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

with $\mathit{next}\ s :: f\ (\mathit{Maybe}\ (a, \mathit{Stream}\ f\ a))$ "evaluated" into:

(1) either *Nothing* for the terminated stream ,

(2) or $\mathit{Just}\ (a, sc)$ for a produced value a
and a stream continuation sc.

Timed IO streams

Timed IO streams

$$\textbf{type}\ \mathit{STIO}\ a = \mathit{Stream}\ \mathit{TIO}\ a$$

that essentially behaves like timed IO signals. . .

With good behavior

(1) GC with full capacity to prevent memory leaks,

(2) data flow programing with bounded memory usage,

solving some problems arising with FRP approach.

Example: timed standard IO with timed IO streams

Input as a timed IO stream:

```
streamInChar :: STIO Char
streamInChar = Stream $ do
  b ← liftIO $ hIsEOF stdin
  if b then return Nothing
  else do
    a ← liftTimedIO getChar
    return $ Just (a, streamInChar)
```

and streaming to output:

```
streamOutChar :: STIO Char → STIO ()
streamOutChar (Stream m) = Stream $ do
  c ← m
  case c of
    Nothing → return Nothing
    Just (a, s) → do
      liftIO $ putChar a
      return $ Just ((), streamOutChar s)
```

Horizontal monoid structure

Putting streams one after the other:

```
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)
```

with the second stream delayed for ever when first is infinite.

Vertical monoid structure

Merging streams by local termination time:

```
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, mc1)) → return $
         Just (a, merge mc1 (Stream $ readRef r2))
      Right (Just (a, mc2)) → return $
         Just (a, merge (Stream $ readRef r1) mc2)
```

A resulting associative and, moreover, commutative, operator still with empty stream as neutral element.

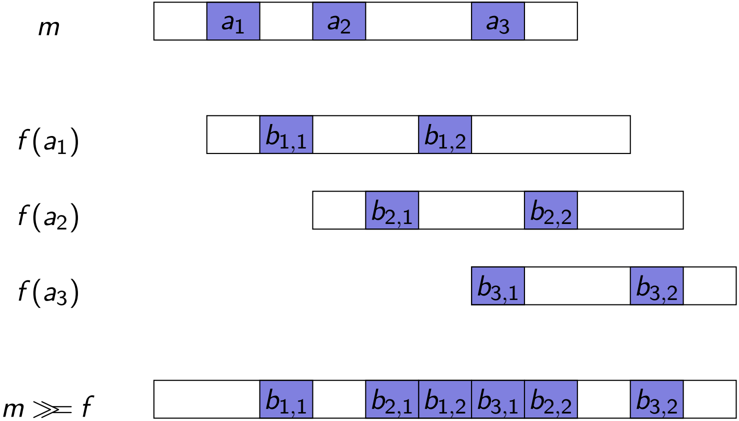
Induced asynchronous monad

The induced (non standard) monad instance:

```
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 by the merge of all parameterized monad streams from *when* there are produced !

The asynchronous bind of streams with monad references



4. Monad stream references

Or unbounded fifo chanel

Monad stream references

Monad streams can also be forked and referenced by:

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

with stream references reading given by:

```
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)
```

Monad stream references

and streams forking into references given by:

```
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)
```

And more

Forking monad actions in traversable structures, and sorting them by termination time:

```
forkAllToRefs :: Traversable t ⇒
  t (m a) → m (StreamRef m a)
```

with an associated (linear time) on-the-fly folding

```
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))
  --
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
```

5. Conclusion

That's more than enough for now. . .

Conclusion

- We thus have defined:

$$\begin{aligned} \textit{Timed IO monad} &= \textit{Timed monad} + \textit{monad references} \\ &\quad + \textit{higher order streams} + \textit{IO} \end{aligned}$$

  on top of Haskell IO monad + Haskell concurrent library.
- This extension provides timed streams programming while preserving GC capacity to keep memory bounded.

Future directions of research

- Theoretical study of monad references in CT ?
- Static analysis tools for detecting:
  - time contraction ⇒ non causal behavior,
  - time expansion ⇒ unbounded buffering,

  in timed IO stream functions ?
- Safe interruption mechanism for timed streams ?
- Extension to hybrid system : from timer to signal handler ?
- Synchronous vs asynchronous streaming ?
- and more experiments, e.g. audio processing and music. . .

Thanks for your attention

Any question ?