Screaming in the IO Monad

A Realtime Audio Processing and Control Experiment in Haskell

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

Abstract

We investigate in this paper the applicability of the notion monad streams to media stream programming, and, more specifically, audio processing and control. Simply said, a monad stream is sort of a list guarded by a monad action that returns either nothing when the stream is over, or, otherwise, just the current value of the stream and the guarding action of its continuation.

Applied to the IO monad, it appears that monad streams can be used for modeling both input streams and output streams, with full control of the possibly synchronism between input and output streams in stream functions. This allows for defining both synchronous or asynchronous functions, or any combination of both notions.

In the abstract, this opens quite intriguing and generic solutions towards programming systems that are globally asynchronous and locally synchronous (GALS). In the concrete, applied to real-time audio, this allows for combining, in a fairly simple and unified way, both (synchronous) audio processing and (asynchronous) audio control.

As far as performance are concerned, our proposal allows non-trivial transformation of audio streams at 44100 Hz with a 10 ms latency, a performance comparable to functional programing languages dedicated to real-time audio processing such as Faust.

Keywords Realtime Programming, Monad Streams, Synchronous Audio Processing, Asynchronous Audio Control

1 Introduction

Motivation and Problems. Music programing with audio processing is a matter of streaming signals from inputs through various programmed transformation and control devices, possibly with feedback loops, towards outputs.

While audio transformation devices are generally synchronous, say governed by some fixed sampling rate, control devices may well be asynchronous, control parameters being produced by, say, human interactions. The resulting programing style, sometimes called *globally asynchronous and locally synchronous* (GALS), was identified a while ago [22].

For this kind of programing, we consider the possibility of using a lazy functional programing language such as Haskell [7] — a possibility often regarded as irrelevant, the garbage collector of Haskell preventing, *a priori*, high-frequency IO processing.

It occurs that, *a posteriori*, this is not only feasible, but with performances comparable to dedicated real-time audio processing and control languages such as Faust [16], still preserving many features of Haskell language: its laziness and its polymorphic types.

Main Result. More precisely, we consider some simple notion of monad streams that allow for defining both input and output stream in a single and simple formalism, in such a way that both synchronous and asynchronous stream transformation functions, and their combinations, can easily be implemented.

As a result, we obtain a fairly flexible library for audio processing and control, fully implemented in Haskell, that only uses the external *Jack audio* connection kit for defining connection with the underlying system. Experiments show that non trivial audio processing and control with 44100 Hz audio signals can be achieved with a 10ms latency — a latency considered to be unnoticeable by a general audience.

Overview of the Main Ideas. A *monad stream* is essentially defined as a monad action that, when executed, returns either nothing, when the stream is over, or the current value of the stream and a monad action that guards the access to the remainder of that same stream.

Thanks to the capacity offered by monad programing to execute and/or produce monad actions (see below), a monad stream can thus be viewed, depending on the way monad actions are handled in a function from stream to stream, as one of the following:

(1) an *input stream* when the *execution* of each monad stream action allows for accessing to the stream of input values,
(2) an *output stream*, when the *production* of each monad stream action tells how the associated output value shall be accessed later on.

Saying so, we follow the somewhat naive but fruitful point of view that a bind expression $m \gg\!= f$ is both a monad action *produced* by binding m with f, as well as the action describing the *execution* of action m with its returned value passed to f as argument. Of course, such a point of view is just false with an arbitrary monad, but it does make sense in monads such as the IO monad where the bind is strict.

As we shall see later in the text, applied to the IO monad, programming a synchronous IO function between streams is then simply a matter of (inductively) describing how each input monad stream action is transformed into a corresponding output monad stream action, much like in a synchronous Mealy machine [15] (see also [4] p 299). The fact that the monad stream type constructor is a functor provides examples of one state Mealy machines. Multi-state Mealy machines are also easily definable thanks to a loop operators.

Quite strikingly, such a principle is applicable both to lazy or eager functional programing languages. Indeed, a monad stream is (essentially) an action to be executed, not an executed action. In other words, monad stream can also be understood as a way to encode lazy stream, as implicitly defined in Haskell, into an eager programming language such as, say, OCaml [20].

Moreover, as such a laziness is explicitly encoded in monad stream, the programmer keeps full control on the synchrony between input and output streams when defining a (recursive) function over streams. There is no forced correspondence between the rank (or rather the nesting depth) of input monad stream actions performed in some recurrence step and the rank (or rather the nesting depth) of the output monad stream actions produced in that same recurrence step.

As a consequence, monad streams also allow for defining a wide range of partially asynchronous functions over streams in the sense that:

(1) the input streams are not necessarily red at the same rate,

(2) the output stream is produced at a rate that may depend on all, some or none of the rate at which the input streams are received.

As a typical example of a fully synchronous stream function, one can zip two input streams into a single one. On the opposite side, as a typical example of a fully asynchronous stream function, one can merge two monad streams into a single one by sorting their values according to their availability, thanks to some simple concurrency features extending our proposal.

For convenience, most utility functions we shall define in this paper are illustrated by block diagrams of the form depicted in Figure 1. More precisely, we shall define functions

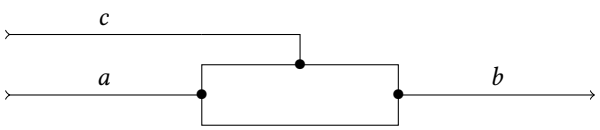

Figure 1. The block diagram of a typical stream function with horizontal (left) synchronous input of *a*s, vertical (top) asynchronous input of *c*s and horizontal (right) output of *b*s.

over streams whose output stream is (essentially) produced at the same rate as the rate of its synchronous inputs, the produced values possibly depending on some parameter values received from its asynchronous inputs.

Also, it must be mentioned that most discussed types and transformations are detailed throughout with (essentially) complete Haskell code. This first illustrates the simplicity of our proposal. Moreover, as the proposed code is detailed in *non-compact* form, this code shall still be understood by readers not familiar with Haskell and its numerous libraries — libraries that we essentially do not use, aside from Haskell prelude, so that we keep full control on performance issues.

Organization of the Paper. Monad streams are briefly presented in Section 2. Basic examples of standard input and output turned into streams are detailed. The notion of synchronous functions we consider is defined in Section 3 together with various examples.

The notions of monad and monad stream references are presented in Section 4. This first allows for freely and safely duplicating monad streams. This also yield some examples of function over streams with asynchronous features in Section 5.

Examples of typical audio processing and control functions are presented and discussed in Section 6, and connection to the audio connection kit (Jack) is then described in Section 7. This essentially amounts to transforming the synchronous stream functions we wish to play into the transition function of its underlying Mealy machine operational model — a transition function used as a callback by the audio driver. Performance issues and tuning are eventually discussed.

Connections with related works are then discussed in Section 8 before concluding in Section 9.

2 Basics on Monad Streams

In this paper we shall use a parameterized notion of streams that essentially yields two kind of streams:

(1) the *monad streams* parameterized by a monad type constructor m (in this section),
(2) the *monad reference streams* parameterized by a monad reference type constructor $Ref\ m$ (see Section 4).

More precisely the parametrized stream type is defined by:

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

with a type function $f :: * \to *$ and a type $a : *$ as parameters.

Defining the length of a parameterized stream as the number of nested *next* action that defines it, we shall soon observe, with *stdinStream* example below, that such a length is not only a dynamic value but it may be infinite as well.

Although the evaluation of *next s* for some stream s depends on the choice of the type function f, it shall return, in all cases, the immediate future of the stream, defined by:

(1) either *Nothing* when stream s *terminates*,
(2) or *Just* (a, sc) when stream s *produces* value a and *continues* as stream sc.

A *monad stream* is then defined as any stream parameterized by a monad functor.

Standard Input Output Example. As a simple stream programming example, reading a stream from the standard input (*stdin*) can be written as follows:

```
stdinStream :: Stream IO Char
stdinStream = Stream $ do
  c ← hIsEOF stdin
  if c then return Nothing
  else do
    a ← getChar
    return $ Just (a, stdinStream)
```

In this example, the stream terminates when *EOF* is received from the standard input.

Conversely, printing a stream to the standard output (*stdout*) can be written as follows:

```
streamStdout :: Stream IO Char → IO ()
streamStdout (Stream m) = do
  c ← m
  case c of
    Nothing → return ()
    Just (a, s) → do
      putChar a
      streamStdout s
```

Observe that its the execution of the IO monad action *streamStdout s* that will make the monad action in stream *s* to be performed. In other words, following Elliot's terminology [6], monad streams must be *pulled* for they are lazy data structures. This constrasts with synchronous language families with *push* semantics of functions over signals.

Continuing our IO examples, echoing the standard input on the standard output can then be defined by:

```
echoStdIO :: IO ()
echoStdIO = streamStdout stdinStream
```

It is worth noting that the above IO action runs in constant space when compiled with *ghc*. In other words, although a monad stream computation may be non terminating as above, on-the-fly processing can be defined over streams without any *memory leaks*.

Functor Instance. As another simple programing example over a monad stream, the following functor instance allows application of a function to every element of a stream given as input:

```
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)
```

Observe that every input action is mapped to an output action. Such a kind of function will later be called synchronous. The behavior of *fmap* is conveniently described by the block diagram depicted in Figure 2.

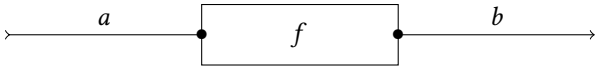

Figure 2. The *fmap f* block diagram

Observe that[1], the weaker constraint $Functor\ m$ suffices for turning $Stream\ m$ into a functor. Indeed, one can put

```
fmap f (Stream m) = Stream $ fmap (fmap (λ(a, sc) →
                      (f a, fmap f sc))) m
```

The proposed version, within the monad m, also aims at illustrating the notion of synchronous stream function we shall see in the next section.

Monad Streams vs Monad Actions. An interesting feature of monad streams is that they can be retrieved from monad actions. More precisely, there is the function:

```
toStream :: Monad m ⇒ m (Stream m a) → Stream m a
toStream m = Stream $ m ≫= next
```

[1] As noticed by one anonymous referee

that shows monad streams are closed under monad actions. This implies, for instance, that any function:

$$f :: a \to m\ (Stream\ m\ b)$$

that produces a monad stream via some monadic computation can be turned into the function

$$toStream \circ f :: a \to Stream\ m\ b$$

that stays within monad stream types. As a consequence, although monad streams are (deeply) built with monad actions, most functions acting on monad streams can be defined within monad streams as illustrated by the various (pure) stream combinators defined throughout.

Horizontal Monoid Structure. Monad streams can be combined one after the other in the following manner, yielding a monoid instance.

```
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 the second monad stream is necessarily delayed until the first monad stream terminates. In other words, the second stream is implicitly buffered. This says that, in most cases, such a horizontal product shall only be used with a finite and bounded length first argument.

With a left stream c of constant length d, the mapping $\lambda s \to c \diamond s$ is still synchronous in some sense made precise in the next section. The resulting block diagram is depicted in Figure 3.

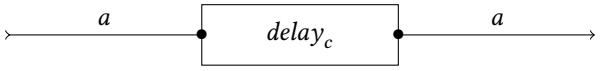

Figure 3. The delay block diagram

Monad Stream vs Lazy Lists. The monoid instance above illustrates the fact that monad streams can be seen as an alternative encoding of lazy lists. In particular, monad streams can be infinite.

Observe however that such a encoding of laziness *does not* rely on the underlying programing language semantics. Indeed, similar monad stream types can be defined in a programing language with eager evaluation such as OCaml, the monad action type $m\ a$ simply replaced by, say, the function type $s \to (s, a)$ for some (monad) state type s.

3 Synchronous Monad Stream Processing

Here, we review some more function examples over streams that we call synchronous.

Synchronous Stream Functions. There are various ways to define synchronous functions, as for instance done the Synchronous language family[1, 21] or as detailed by the author when developing the notion of timed domains [9].

In this paper, we shall rely on the rather intuitive (and special case) definition that a synchronous function over streams shall essentially behave like a Mealy machine, that is, a function that reads in a synchronized way all its (stream) inputs and produces its (stream) output at the same rate.

As streams have to be "pulled" for their actions to be executed and their values to be read, the above intuition can be rephrased as follows: a function over streams is said to be synchronous when, for every input and for every possible rank n, the execution of the nth monad action in the output stream of that function implies the execution of the nth actions of each of its input streams (therefore all preceding actions as well) and no more.

Additionally, a stream function obtained from a synchronous function by delaying its output by a stream of constant length d as above is called d-synchronous.

Simple Synchronous Function Examples. Both stream functions id and $fmap\ f$ are synchronous in the sense above. Moreover, in both these cases, the output stream terminates when the input stream terminates. The function $\lambda s \to c \diamond s$ with a monad stream c of known length d is d-synchronous.

Synchronous Zip. Another typical example of a synchronous function is the zip function defined over streams by:

```
zipStream :: Monad m ⇒
  Stream m a → Stream m b → Stream m (a, b)
zipStream = zipStreamWith (λa → λb → (a, b))
```

with

```
zipStreamWith :: Monad m ⇒ (a → b → c) →
      Stream m a → Stream m b → Stream m c
zipStreamWith f (Stream m1) (Stream m2)
 = Stream $ do
   c1 ← m1
   c2 ← m2
   case (c1, c2) of
     (Just (a1, sc1), Just (a2, sc2)) →
       return $ Just (f a1 a2, zipStreamWith f sc1 sc2)
     _ → return Nothing
```

The block diagram of the function $zipStreamWith\ f$ is depicted in Figure 4. Observe that, in such a zip, the output

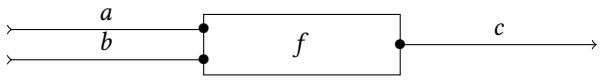

Figure 4. The $zipStreamWith\ f$ block diagram.

terminates, whenever on the input stream terminates. This is indeed allowed by our definition of synchronicity.

Constant Streaming. As a simple application example of the synchronous zip, the function *constSync* takes two streams as input and returns the first one.

```
constSync :: Monad m ⇒
  Stream m a → Stream m b → Stream m a
constSync = zipStreamWith const
```

Given a monad stream *s*, one may think that *constSync s* essentially behave like *const s*. However, the later makes no synchronization between the input and the output stream, and, even worse, does not execute the monad actions of the second stream at all, therefore its side-effects. These two functions truly have distinct behaviors.

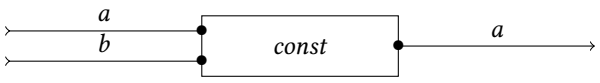

Figure 5. The *zipStreamWith const* block diagram.

More on (implicit) Monad Stream Clocks. Every monad stream induces a clock, the clock ticks being defined by the execution of the nested monad actions embedded in that stream.

In the *zipStreamWith* code given above, one can observe that streams are zipped by waiting each time on the slowest action to terminate. This waiting is even biased by waiting first for the action m_1 to terminate before running the second action m_2. It follows that the implicit clock of the resulting stream is the (element wise) upper bound of the implicit clocks of the input streams.

In other words, when defining synchronous processing, we somehow assume that all inputs streams are *essentially* synchronized on the same clock. However, strictly speaking, this assumption is false as illustrated by the following stream functions:

```
repeatN :: Monad m ⇒ Int → a → Stream m a
repeatN n a = Stream ∘ return $
  if (n ⩽ 0) then Nothing
  else Just (a, repeatN (n âĹŠ 1) a)
takeN :: Monad m ⇒ Int → Stream m a → Stream m a
takeN n s = zipStreamWith cons s (repeatN n ())
```

that takes , somehow in a synchronous way, the first *n* values of an input stream with, say, any input governed clock, while the clock of the stream produced by *repeatN n* () only depends on how fast it is read.

A a result, with monad streams, there is no explicit system clock synchronizing all streams. This apparently quite differs from the notion of synchronicity proposed and implemented in the synchronous language family [21], even though the resulting global behaviors are quite similar.

Stream Loops. Much like in the state monad, there is a loop function that allows for defining arbitrary synchronous function defined as a Mealy machine.

```
loopStream :: Monad m ⇒ s →
  ((a, s) → m (b, s)) → Stream m a → Stream m b
loopStream s f (Stream m) = Stream $ do
  c ← m
  case c of
    Nothing → return Nothing
    Just (a, sc) → do
      (b, s') ← f (a, s)
      return $ Just (b, loopStream s' f sc)
```

More precisely, given a Mealy machine defined via state type *s*, input type *a*, output type *b*, initial state $s_0 :: s$, and (deterministic) transition function *delta* :: $(a, s) \to (b, s)$, the stream function *loopStream* s_0 (*return* ∘ *delta*) just encodes such a Mealy machine depicted in Figure 6. with *delayBy*

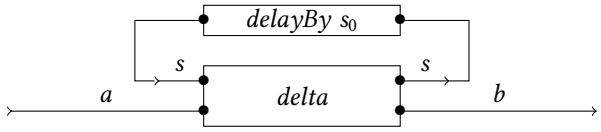

Figure 6. The *loopStream* s_0 (*return* ∘ *delta*) block diagram.

delaying its input stream by the one value stream defined by its argument.

Observe that in *loopStream* type, we more generally define transition function with type $(a, s) \to m\ (b, s)$. In the IO (concurrent) monad, this allows for defining more efficient implementation of transition functions. A non-trivial application of such a possibility is illustrated in Section 7 when transforming any synchronous IO stream function into its (implicit) transition function.

4 Monad Stream References

When handling IO with monad streams, as with *stdinStream* defined above, an important issue is that, a priori, such a stream *should not* be shared by two (threaded) independant IO monad action. Indeed, two running actions sharing a copy of *stdinStream will not* share the same sequence of inputs but, instead, will receive one of the subsequence resulting from the *distribution* of the original sequence between these two actions.

The notions of monad references and monad stream references define in this section offer a fairly generic way to cope with such an issue.

Monad Reference. Simply said, a monad reference is a reference to a location, uniquely associated to a *running* monad action, that, upon termination of that action, shall contain the value returned by that action and shall be freely read by any action possessing that reference.

Such a notion is conveniently described by the following class type:

```
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) *Ref m* is the type of references to running action,
(2) *forkToRef* forks an action and (immediately) returns a reference to that action,
(3) *readRef* returns (and possibly waits for) the value returned by a referenced action,
(4) *tryReadRef* immediately returns nothing if the referenced action is not terminated or just its returned value otherwise,
(5) and *parReadRef* returns the value of one of the first terminated referenced actions.

In the case that the two actions are already terminated or are terminating at the same time, the output of *parReadRef* may be non deterministic.

Conditional Cut. As a first application example of monad references, the following function runs a monad action and a monad stream, the termination of the action yielding a cut of the input stream.

```
takeStreamUntil :: MonadRef m ⇒
      m c → Stream m a → Stream m a
takeStreamUntil m s = Stream $ do
  r ← forkToRef m
  takeStreamUntilRef r s
    where
  takeStreamUntilRef r (Stream ms) = do
    rs ← forkToRef ms
    c ← parReadRef r rs
    case c of
      Left _ → return Nothing
      Right Nothing → return Nothing
      Right (Just (b, sc)) → return $
        Just (b, Stream $ takeStreamUntilRef r sc)
```

Such an action can be used for conditionally stopping a stream processing function upon some external termination condition. Observe that, when running *takeStreamUntil*, the output stream rate equals the input stream rate. Such a transformation is *synchronous*.

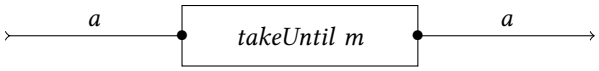

Figure 7. The block diagram of *takeStreamUntil m*.

Monad Stream References. The notion of monad references immediately lifts to monad stream by writing:

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

with the auxiliary functions

```
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

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

Clearly, while a stream *s* shall not be shared by independent actions, for it may contain monad actions with non sharable side effects, the stream *readStreamRef r* obtained after forking the stream *s* can be freely shared. Indeed, reading a stream reference has no side effect on the referenced stream value.

Standard Input Output Example Continued. There are various ways of defining a *MonadRef* instance for the IO monad. Haskell's *async* library defined by Simon Marlow [14] gives one possibility.

The following instance, simply obtained by defining references to IO actions as mutable variables containing their returned value, is simple enough to be described in full detail.

```
instance MonadRef IO where
  type Ref IO = MVar
  forkToRef m = do
    v ← newEmptyMVar
    _ ← forkIO (m ≫= putMVar v)
    return v
  readRef = readMVar
  tryReadRef = tryReadMVar
  parReadRef r1 r2 = do
    v ← newEmptyMVar
    _ ← forkIO (readRef r1 ≫= return ∘ Left
          ≫= tryPutMVar v ≫ return ())
    _ ← forkIO (readRef r2
          ≫= return ∘ Right ≫= tryPutMVar v
          ≫ return ())
    takeMVar v
```

Then, continuing our standard IO example, one can check that the action:

```
echoStdIOviaRef :: IO ()
echoStdIOviaRef = do
  r ← forkStreamToRef stdinStream
  streamStdout (readStreamRef r)
```

that echoes the standard input to the standard output much like *echoStdIO* defined above still runs with constant memory space even though it is now defined via the forking of the input stream.

Such an example illustrate the power of monad streams and monad references that, when properly used, fully preserve the garbage collector capacity to avoid any memory leaks.

5 Asynchronous Monad Stream Control

Aside synchronous stream processing, there is also the need for asynchronous stream control: series of control values (or events) that are received at a much lower and irregular frequency than the synchronous stream to be processed.

Thanks to the notion of monad references defined above, various asynchronous functions over stream are definable.

Asynchronous Control. A first, quite involved example is a function that transforms an input stream depending on a parameter received asynchronously.

```
streamMap :: MonadRef m ⇒ (a → m b) →
  Stream m (a → m b) → Stream m a → Stream m b
streamMap f (Stream mf) (Stream m) = Stream $ do
  rf ← forkToRef mf
  r ← forkToRef m
  c ← parReadRef rf r
  case c of
    Left (Just (g, sf)) → next $
      streamMap g sf (Stream $ readRef r)
    Right (Just (a, s)) → do
      b ← f a
      return $
        Just (b, streamMap f (Stream $ readRef rf) s)
    _ → return Nothing
```

Observe that the output stream ends whenever one of the two input stream ends. The corresponding block diagram is depicted in Figure 8. Observe that, as already done for

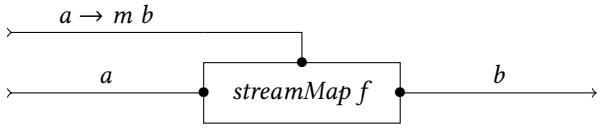

Figure 8. The block diagram of *streamMap f sf*.

loopStream function, we allow ourselves transformation functions of type $a \to m\ b$. Again, this allows for using more efficient implementations of stream transformation functions.

Vertical Monoid Structure. Perhaps the most striking example of an asynchronous function is the following function that merges two streams according to the termination time of the underlying actions.

```
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)
```

Clearly, the outcome of such a function is possibly nondeterministic since each execution of *parReadRef* $r_1\ r_2$ may induce a race with nondeterministic outcome.

Of course, such a function is of no use for synchronous programing as required with audio processing. What could be the meaning of the merge of two audio signal ? However, such a merge function can be used below for merging two independent streams of control values.

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

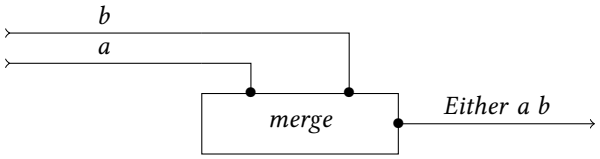

Figure 9. The block diagram of *mergeStream*.

6 Audio Processing and Control

We are now ready to define functions more specifically dedicated to audio processing and control. Below, we review several archetypal examples of this.

Oscillators. As a first example, one can define an sinusoidal signal at a given period described in number of samples by

```
osc :: Int → Stream IO Sample
osc n = oscN 0 n
  where
  oscN :: Int → Int → Stream IO Sample
  oscN n k = Stream $ do
    let v = sin (2 * pi * fromIntegral (mod n k)
        / fromIntegral n)
    return $ Just (v, oscN (n + 1) k)
```

where *Sample* is in Jack interface just a type synonim for *CFloat*. Observe that the actual frequency in Hz of such an sinusoidal shall only be defined by the reading speed of the resulting stream.

Forward Action. A delay line followed by some summing of samples is a typical device for defining (simple) low cut or high cut filter. They can be defined as follows:

```
feedForward :: Int → a → (a → a → b) →
  Stream IO a → Stream IO b
feedForward n a0 f s = toStream $ do
  ch ← newChan
  writeList2Chan ch (take n (repeat a0))
  let delta (u, ch) = do
        v ← readChan ch
        writeChan ch u
        let w = f u v
        seq w $ return (f u v, ch)
  return $ loopStream ch delta s
```

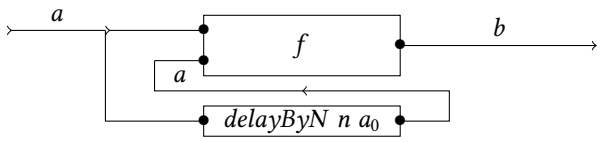

Figure 10. The block diagram of *feedForward n a*.

Retro Action. A retro action (or retro feedback) is another typical device for defining a pass band filter.

```
feedBack :: Int → b → (a → b → b) →
  Stream IO a → Stream IO b
feedBack n b0 f s = toStream $ do
  ch ← newChan
  writeList2Chan ch (take n (repeat b0))
  let delta (u, ch) = do
        v ← readChan ch
        let w = f u v
        seq w $ writeChan ch w
        return (w, ch)
  return $ loopStream ch delta s
```

Observe that such a code is essentially the same as above up to the fact that, in this case, it is the output value *w* that is re-injected into the FIFO parameter *ch*.

In both case, the content of the channel is used as the state of the underlying Mealy machine defined by the transition function *delta*. Worth being noticed, we use the type *Chan* in this function instead of the type *List* for it appears it is way more efficient as far as performance are concerned. This fully justifies the choice made in function *loopStream* to have a transition function of type $(a, s) \to m\ (a, s)$.

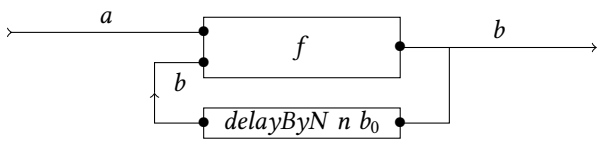

Figure 11. The block diagram of *feedBackward n a*.

Stereo vs Mono Streams. Turning two mono streams into a stereo streams is simply achieved by the *zipStream* function already described above.

7 Connecting with Driver Interfaces

Last, we need to connect our stream function to the underlying system. For such a purpose, we shall use the Haskell interface to the fairly flexible and cross-plateform Jack audio connection kit.

For audio processing programs, we expect to use stream function, that is, synchronous functions of the form:

$$g :: Stream\ m\ a \to Stream\ m\ b$$

with input sample type a and output sample type b. However, the Haskell interface to Jack expects instead *step functions* of the form:

$$f :: a \to m\ b$$

that shall maps every (new) input sample a to the action $f\ a$ that returns the corresponding output sample b.

The purpose of this section is to shows that, in the IO monad, one can actually converts any synchronous function into such a step function, hiding its underlying memory state into the IO monad.

From Step Functions to Synchronous Functions. Observe that the other direction, defining a synchronous stream function from a step function, is easy. Indeed, this can be achieved by:

```
mapToStream :: Monad m ⇒ (a → m b) →
  Stream m a → Stream m b
mapToStream f (Stream m) = Stream $ do
  c ← m
  case c of
    Nothing → return Nothing
    Just (a, sc) → do
      b ← f a
      return $ Just (b, mapToStream f sc)
```

Clearly, the resulting function is synchronous.

FIFO vs Monad Streams. In the IO monad, converting any synchronous stream function into its corresponding step function is achieved by means of creating two communication channels:

(1) one from the step function input to the stream given as argument to the stream function,
(2) one from the stream resulting from that application, to the output of the step function.

These communication channels are conveniently and efficiently encoded by means of FIFO channels (*Chan*). The connection diagram induced by such a translation of stream functions into step functions is depicted in Figure 12. The difficulty in drawing such a picture is that the IO monad is (almost) everywhere around, the snake-like arrow describing not functions, but, instead, IO communication between certain typed value.

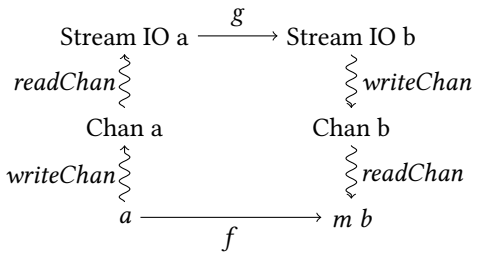

Figure 12. A stream function g and its step function f created via FIFO channels.

From FIFOs to Input Streams. Turning a FIFO into a stream can simply be done as follows.

```
makeStreamFromChan :: Chan a → Stream IO a
makeStreamFromChan ch = Stream $ do
  a ← readChan ch
  return $ Just (a, makeStreamFromChan ch)
```

Observe that the frequency of the produced infinite stream depends on the frequency at which values are put in the FIFO.

From Output Streams to FIFOs. Conversely, turning an output stream into a FIFO can be done by:

```
makeChanFromStream :: Int → a →
  Stream IO a → IO (Chan a)
makeChanFromStream n ia s = do
  ch ← newChan
  writeList2Chan ch (take n (repeat ia))
  _ ← forkIO $ dropStreamToChan ch s
  return ch
    where
  dropStreamToChan ch (Stream m) = do
    c ← m
    case c of
      Nothing → return ()
      Just (a, mc) → seq a $ (writeChan ch a)
        ≫ dropStreamToChan ch mc
```

In the above signature, we also take an integer n and an initial value a in order to initialize the FIFO with n times the initial value n. This allows for inserting some delay buffering to the expected function.

Observe that, after such an initial delay, the frequency at which the argument stream will be traversed depends on the frequency at which the returned FIFO will be read by a *readChan* action.

Observe also that the function *dropStreamToChan* is forked so that, when used, feeding and reading the FIFO channel are mostly desynchronized actions.

Extracting Step Function. Last, turning a stream synchronous function into its corresponding step function can be done as follows:

```
streamToMapN :: Int → b →
  (Stream IO a → Stream IO b) → IO (a → IO b)
streamToMapN n b f = do
  inChan ← newChan
  outChan ← makeChanFromStream n b $
    f (makeStreamFromChan inChan)
  return (λa → seq a $ writeChan inChan a
    ≫ readChan outChan)
```

The first integer argument is the length of the buffer (in number of samples).

It is quite a striking fact that *streamToMapN* manages to re-create and handle the implicit Mealy machine state of its stream function argument. Making an explicit extraction of the Mealy machine induced by a streaming function would be fairly involved and would necessitate both the code of that function and a precise operational model.

Connection with Jack. Last, connection with Jack audio connection kit, via the Jack Haskell interface implemented by Thielemann et al., is simply achieved by:

```
playWithJack :: Int →
  (AudioSStream → AudioSStream) → IO ()
playWithJack n f =
  streamToMapN n (0, 0) f ≫= mainStereo
```

where *AudioSStream* is just a type synonym of IO streams of stereo samples and *mainStereo* is one the main function provided by the interface to connect a Haskell defined audio transformation to Jack audio server.

Performance Issues and Parameter Tuning. The integer parameter in *playWithJack* defines the size of the delay buffer, internal to our Haskell code, between sample inputs and sample outputs. Such a buffering is essential for achieving good performance. Indeed, it allows for desynchronizing input-reading from Jack server and output-writing to Jack server. Experiments shows that, at 44100 Hz, a 128 samples buffer length suffices.

An additional buffer size is defined on Jack server. It defines the frequency at which Jack server is calling a Haskell function as a callback. Again, experiments show that, at 44100 Hz, a 128 samples buffer length suffices.

Together, this eventually yields to running audio processing and control programs written is Haskell, essentially with no loss of samples, with a cumulative latency of 11.6 ms, twice the 5.8 ms latencies induced by the two buffers of 128 samples size.

To the best of our knowledge, no existing library written in Haskell yet reaches such a performance level together while offering such a level of abstraction.

8 Related Works

In this paper, we use the presentation of monad stream defined by the author [10] that develop the notion of timed extension of a monad. The present paper can be seen as an experiment in using that notion in the untimed case.

One can observe that such a type *Stream m a* is isomorphic to the type *ListT m a* where *ListT* is the Haskell (non-deprecated) monad transformer defined in *List* package. Although isomorphic, our treatment of that type is fairly distinct. Indeed, the flavor of sequentiality induced by guarding the list tail by monad actions (especially in the IO monad) is essentially ignored in the above package that relies on the horizontal monoid structure induced by appending lists.

More precisely, in both synchronous or asynchronous settings, appending two input monad streams hardly make any sense. In the synchronous setting, delays, zips and maps covers most of our need and, in the asynchronous setting, it is rather the vertical monoid structure of streams, induced by the function merge, that is useful.

Compared to more generic proposals, as already observed by Perez et al. [18], stream monads provide a efficient way to instanciate FRP approaches [3, 5, 6]. Programing with stream monads is also quite related with FRPNow [19].

It shall be mentioned that Perez et al. use a version of monad streams distinct from ours, defined by

```
data Stream′ m a
  = Stream′ { next′ :: m (a, (Stream′ m a)) }
```

Their proposal is more general than ours since our notion of streams can be recovered from theirs by composing *Maybe* with *m*. However, with our specialization, we stress on the fact that streams may terminate: a feature that may lead to delicate issue when not properly addressed.

Comparing our proposal to the general FRP approaches, it must be mentioned that the API induced by our approach strongly differ from the one defined in FRP [5, 6]. Indeed, in our approach, we make no distinction between stream values and events. Instead, it is more a matter of distinguishing synchronous and asynchronous treatments in functions over streams, and distinguishing streams that shall be treated as synchronous or asynchronous ones.

One advantage of such our approach, though not yet secured by any typing constraints, is that one is not tempted to define asynchronous treatments on streams that are supposed to be run in a synchronous way. As an illustrative example, functions *zipStream* and *mergeStream* shall *never* be used on the same kind of streams.

A long standing approach for handling asynchronism amounts to encoding an asynchronous stream *Stream m a*

by a synchronous one *Stream m* (*Maybe a*) that yields *Just a* when there is a change of control value, of *Nothing* when there is not.

This is an approach classically followed in synchronous systems governed by one global clock [1, 21] as well as, to the best of our understanding, in recent FRP approaches [18]. While such an approach is unavoidable at circuit (or OS kernel) level, it may yield an unnecessary load of processing at system level, control signal mostly containing the value *Nothing*.

Worth being mentioned, as a generalization of the above, there is the *wormholes* approach [23, 24] where every stream *Stream m a* can be lifted to a stream of type *Stream m* [*a*]. While preserving inputs arrival order, this fairly flexible approach allows for contracting or expanding data depending on the frequency one aim at achieving. However, the streaming the empty list may also yield an unnecessary load of data at system level.

Our proposal extends the alternative for encoding asynchronous functions by relaying on the notion of monad references and concurrent wait. Asynchronism is then efficiently handled by *ghc* runtime.

A type quite similar to our notion monad stream references, that is, our type *Stream MVAR a*, is already defined and used pretty much in the same way by Marlow when defining unbounded FIFO communication channel [14].

The efficiency of the resulting compiled code seems to rely on the fact that *ghc* optimizes quite efficiently programs that uses (even implicit) continuations : a programing style that arises fairly often when handling monad streams or monad stream references.

Worth being mentioned, our desire to achieve efficient and fully understood audio processing prevents us from reusing existing Haskell libraries. As a matter of fact, aside the Jack/Haskell interface package, we essentially use basic features that are available in Haskell's prelude, and some of its concurrent features: multithreading capacity (*forkIO*), mutable variables (*MVar*) and FIFO channels (*Chan*).

As far as audio is concerned, the main and most efficient functional programming language for realtime audio processing available nowadays is probably Faust [16]. This language is thus the most relevant yardstick for examining the qualities of our proposal.

In terms of performance, Faust is, a priori, way more efficient than our proposal for it is currently compiled into optimized C code. Experiments show that audio processing we performed in Haskell is more expensive by some (small) factor compared to Faust.

Still, thanks to our multithreaded implementation, audio processing is performed on a dedicated core, the resulting cost of connecting audio streams with audio drivers being comparable with the one achieved with Faust. Moreover, thanks to Haskell language, our monad stream approach is polymorphic and can probably be applicable to many other media types. Observe that audio processing is probably the most demanding media type both in terms of frequency and admissible latency.

In terms of programing API, Faust eases the programmer task by offering a rather abstract view on signals, promoting a data flow programing style based of some number of primitive blocks to be assembled at will. Non programmer experts can use it quite easily.

On the contrary, yet in its infancy, our approach still necessitates explicit description of monad stream traversal. This clearly requires more expertise in programming. However, as illustrated by the simplicity of the numerous function code examples given throughout our paper, such an expertise may remain quite low.

9 Conclusion

We have thus proposed and implemented a version of monad streams that allows for combining both high-frequency synchronous transforms, as needed in audio processing, as well as asynchronous low-frequency parameterization of these synchronous transforms, as needed in audio system control.

The notion of monad streams induces a fairly versatile interface for both synchronous and asynchronous programing. It is even quite striking how efficient the resulting library is. This is due to both the fact that, despite our functions being pure and defined in safe Haskell, we make a rather heavy though quite transparent use of the IO monad, and the ghc compiler optimizes especially well continuation programing style.

The Haskell library resulting from the present paper shall eventually be available with open source license on some publicly accessible platform.

However, as far as software engineering is concerned, our proposal is truly at an early stage. For instance, as already mentioned, it would probably be worth distinguishing streams that are to be used in a synchronous way from streams that are to be used in an asynchronous way. For such a purpose, one could define a generic data type for stream functions that, as depicted in block diagrams throughout our presentation, have explicit synchronous and/or asynchronous inputs.

Such a distinction could open a way towards an extension of arrow programing [8]. Indeed, the basic operator $(***)$ in *Arrow* class suggests that, when applied to signal processing, arrows are acting synchronously on signals. On the contrary, the basic operator $(+++)$ in the *ArrowChoice* class suggests that, when applied to signal processing, arrows are acting asynchronously on signals. Adding explicit distinction between synchronous and asynchronous inputs might lead to the definition of a finer model of streams programing.

Anyhow, nothing yet ensures in our proposal that a given function input stream is treated in a synchronous or asynchronous way, and, in both cases, that the function does not encode a behavior with intrinsic memory leaks. There clearly lays the possibility of designing a type system for functions over streams that would enforce adequate synchronous or asynchronous usage and traversal of monad streams inputs.

Temporal logic typing as already proposed for FRP [11–13] might be a starting point for such a purpose. However, we also suspect that linear types could be of some interest [2, 17]. Indeed, as already discussed in Section 4, monad streams shall not be shared in general as input. On the contrary, monad stream references can be freely shared. A linear type system could accurately model such a distinction.

Acknowledgments

This work has benefited from many discussions with Simon Archipoff and Bernard Serpette, and some help from Donya Quick, who all deserve our great thanks. We also feel deeply in debts towards anonymous referees for their tolerance and open mind in reviewing a first quickly written draft that eventually got considerably improved thanks to their numerous, accurate and fruitful remarks.

References

[1] A. Benveniste, P. Caspi, S. A. Edwards, P. Le Guernic N. Halbwachs, and R. de Simone. 2002. The Synchronous Languages Twelve Years Later. *Proc. IEEE* (2002).

[2] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017), 5:1–5:29.

[3] A. Courtney, H. Nilsson, and J. Peterson. 2003. The Yampa arcade. In *Workshop on Haskell*. ACM, 7–18.

[4] S. Eilenberg. 1973. *Automata, Languages and Machines, Volume 1*. Academic Press.

[5] C. Elliott and P. Hudak. 1997. Functional Reactive Animation. In *Int. Conf. Func. Prog. (ICFP)*. ACM.

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

[7] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. 2007. A History of Haskell: Being Lazy With Class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*. ACM Press.

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

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

[10] D. Janin. 2019. *A Timed IO monad*. Technical Report. LaBRI, Université de Bordeaux.

[11] A. Jeffrey. 2012. LTL Types FRP: Linear-time Temporal Logic Propositions As Types, Proofs As Functional Reactive Programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*. ACM, 49–60.

[12] W. Jeltsch. 2014. Categorical Semantics for Functional Reactive Programming with Temporal Recursion and Corecursion. In *Mathematically Structured Functional Programming (MSFP) (EPTCS)*, Vol. 153. 127–142.

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

[14] 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.

[15] G. H. Mealy. 1955. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal* (1955), 1045 –1079.

[16] Y. Orlarey, D. Fober, and S. Letz. 2009. Faust: an Efficient Functional Approach to DSP Programming. In *New Computationals Paradigms for Computer Music*. Editions Delatour France.

[17] J. Paykin and S. Zdancewic. 2017. The Linearity Monad. In *Haskell Symposium*. ACM.

[18] I. Perez, M. Bärenz, and H. Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell*. ACM, 33–44.

[19] 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.

[20] D. Rémy. 2002. Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and Vice Versa. In *Applied Semantics*, G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva (Eds.). LNCS, Vol. 2395. Springer, 413–536.

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

[22] P. Teehan, M. R. Greenstreet, and G. G. Lemieux. 2007. A Survey and Taxonomy of GALS Design Styles. *IEEE Design & Test of Computers* 24, 5 (2007), 418–428.

[23] D. Winograd-Cort. 2015. *Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming*. Ph.D. Dissertation. Yale University.

[24] 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.