Efficient Monadic Streams

Josef Svenningsson[1], Emil Axelsson[1], Anders Persson[1], and Peter A. Jonsson[2]

[1] Chalmers University of Technology
[2] SICS Swedish ICT AB

Abstract. Functional stream representations allow for a high-level, compositional way of programming digital signal processing algorithms. However, some algorithms, such as filters, cannot be efficiently implemented using purely functional techniques, due to excessive copying of data.
We present a monadic representation of stream which introduces the ability to use mutation for efficiency when implementing algorithms. Still, our representation enjoys many of the benefits of purely functional streams, such as a functional API and fusion. Our representation enables further optimizations: we show how to remove duplicate loop variables, and how to keep buffers entirely in references.
The representation has been evaluated in the context of the Feldspar embedded DSL, and our measurements show that our new monadic representation consistently outperforms the functional representation by at least a factor of four.

Introduction

A popular functional stream representation is a transition function from and old state to an element and a new state, together with a starting state:

```
data Stream a = forall s . Stream (s -> (a,s)) s
```

The representation is expressive and can be compiled into efficient code by means of fusion.

Consider computing a simple moving average over a stream using the above representation. The implementation would keep track of the most recent values from the stream with a sliding window. Computing a new result involves inserting a new value at the front of the window and removing an element from the back. The typical functional implementation of a sliding window requires the whole history except the last element to be copied to avoid aliasing problems. Copying the history is safe and conceptually simple but performance suffers. Even if performance is sufficient, copying data consumes more power and generates more heat.

This paper presents a new representation of streams which allows for using mutation while retaining the advantages of the functional stream representation. Our contributions are:

– A new stream representation, `M (M a)`, for some monad `M` which supports mutation. Our representation combines the functional interface of the previous representation with the efficiency of imperative updates.
– We show how to optimize the representation to eliminate duplicate loop variables.
– We show how our new stream representation can be used in EDSLs. It is currently used in the Feldspar language.
– We demonstrate a performance advantage of a factor of four compared to the functional representation when using our monadic representation in Feldspar.

The new monadic formulation of streams is applicable to any language with a monad for mutable effects. We show an implementation in Haskell based on `IO` in section Efficient Monadic Streams as well as an implementation in the Feldspar EDSL (Axelsson et al. 2011) in the section Streams for EDSLs. The advantages of the new stream representation are easier to demonstrate in Feldspar because the generated C code makes excessive copying both easy to see and measure. We compromise by using the Haskell implementation for presentation purposes and the Feldspar implementation for the performance evaluation.

Most of the examples in the paper are available as Feldspar code in the Feldspar repository[3]. The Feldspar code is conceptually similar to the Haskell code shown in this paper.

The Problem

Consider the functional stream representation from the introduction again.

```
data Stream a = forall s. Stream (s -> (a,s)) s
```

Implementing an algorithm such as the moving average using the above representation can look as follows:

```
movingAvg :: Fractional a => Int -> Stream a -> Stream a
movingAvg n (Stream step init) = Stream step' init'
  where
    init' = (init, listArray (0,n-1) (replicate n 0))
    step' (s,window) =
      let (a,s')  = step s
          window' = ixmap (0,n) (\i -> i+1 `mod` n) window
                  // [(n-1,a)]
      in (avg window, (s',window'))
    avg w = sum (elems w) / fromIntegral n
```

[3] https://github.com/Feldspar/feldspar-language/blob/master/src/Feldspar/Stream.hs

This implementation is inefficient because the window needs to be copied each iteration, even if the operations `ixmap` and `\\` are fused. Copying can be avoided to some extent by using smarter window representations but smart window representations tend to have a high constant overhead making them unsuitable for the common case of small window sizes.

We see the problem clearly in the generated C code from the corresponding Feldspar implementation. The window is stored in `v46.member2.member2` and the for-loop on lines 4 to 7 performs the copying.[4]

```
1  for (uint32_t v47 = 0; v47 < 32; v47 += 1)
2  {
3    //Code for computing the average of the window elided
4    for (uint32_t v89 = 0; v89 < v184; v89 += 1)
5    {
6      ((v46).member2).member2[(v63 + 1)] = ((v48).member2).member2[v63];
7    }
8    //Code updating the struct which holds the window elided
9  }
```

Similar problems appear for many applications of streams, such as digital FIR and IIR filters. What we would like is a representation of streams where we can use mutation to efficiently implement such functions.

Efficient Monadic Streams

We present a new representation for streams which uses monads to enable mutation.

```
data Stream a = Stream (IO (IO a))
```

It is straightforward to parameterize this representation on the particular choice of monad. We use the `IO` monad here for the sake of concreteness.

Why does the representation have two levels of monads? The key to understanding this representation is that the outer monadic computation performs initialization and is only meant to be called once. The outer monadic computation returns a new monadic computation of type `IO a`. Repeatedly calling this inner computation produces the elements of the stream.

Our new monadic representation of streams can still be given an API which is functional in flavour and similar to what a programmer would expect from a functional representation. As an initial example consider the `map` function:

[4] We have elided most of the generated code for presentation purposes.

```
map :: (a -> b) -> Stream a -> Stream b
map f (Stream init) = Stream $ do
  next <- init
  loop $ do
    a <- next
    return (f a)
```

The new stream is initialized by running the initialization computation from the input stream, yielding the step function `next`. Then, in the new step function, the function `next` is run to produce an element `a` which is transformed by the function `f` and then returned. The combinator `loop` is defined as `return`. We use the name `loop` to convey that the code returned by `loop` is executed an indefinite number of times.

There are various ways of creating streams, below we show the function `cycle` which cycles through the elements of an array. A particular aspect of this function is that it has to create a reference which is used to keep track of what element in the array to read from.

```
cycle :: Array Int a -> Stream a
cycle arr = Stream $ do
  r <- newIORef 0
  loop $ do
    i <- readIORef r
    writeIORef r ((i+1) `mod` length arr)
    return (arr!i)
```

An infinite stream cannot be stored in memory. Saving a prefix of an infinite stream for later processing is often convenient. Below is the code for saving:

```
remember :: Int -> Stream a -> IO (Array Int a)
remember len (Stream init) = do
  arr  <- newArray_ (0,len-1)
  next <- init
  forM [0..len-1] $ \i -> do
    a <- next
    writeArray arr i a
  freeze arr
```

The code starts by allocating a mutable array of the appropriate size, followed by an initialization of the array. The initialization produces the `next` function which is used in the loop body to produce new elements in the stream which are successively stored in the array. When the loop is done, the mutable array is frozen, returning an immutable array as the final result.

We are now in a position to write an efficient moving average using the imperative features of the new monadic stream representation.

```
movingAvg :: Int -> Stream Double -> Stream Double
movingAvg n s = recurrence (listArray (0,n-1) (replicate n 0.0)) s
                   (\input -> sum (elems input) / fromIntegral n)

recurrence :: Array Int a -> Stream a ->
              (Array Int a -> b) ->
              Stream b
recurrence ii (Stream init) mkExpr = Stream $ do
    next <- init
    ibuf <- initBuffer ii
    loop $ do
      a <- next
      putBuf ibuf a
      b <- withBuf ibuf $ \ib ->
             mkExpr ib
      return b
```

The core functionality is exposed by the `recurrence` function which uses a mutable cyclic buffer. The type signatures for the operations we use are:

```
initBuffer :: Array Int a -> IO (Buffer a)
putBuf     :: Buffer a -> a -> IO ()
withBuf    :: Buffer a -> (Array Int a -> b) -> IO b
```

The function `initBuffer` creates a new buffer, `putBuf` adds a new element while discarding the oldest element. The programmer can get an immutable view of the current contents of the buffer in a local scope by using `withBuf`. The function `withBuf` can be implemented without copying but program correctness relies on the programmer to ensure that the provided function does not return the whole array.

Returning to the function `recurrence`; the input stream stream is initialized as is the cyclic buffer. For each element in the output stream an element from the input stream is computed and stored in the cyclic buffer. The content of the cyclic buffer is processed by a function provided by the caller of `recurrence` and the result is returned as the next element in the output stream. The resulting stream will contain values computed from a sliding window of the input stream.

The function `movingAvg` uses `recurrence` to provide sliding windows of the input stream and passes a function to compute the average of a window. The initial window only contains zeros. In the generated code from the corresponding Feldspar implementation, line 7 shows the window update performed through mutation (the window is stored in `v7`) :[5]

[5] We have removed some variable-to-variable assignments in the code to make it more readable.

```
  v14 = 0;
  copy(v7, zeros);
  for (uint32_t v24 = 0; v24 < 32; v24 += 1) {
    v48 = v0[v25];
    v27 = v14;
    v14 = ((v27 + 1) % 8);
    v7[v27] = v48;
    // Code computing the average elided
  }
```

More advanced digital filters, like FIR filters, can be implemented in a similar fashion to the moving average:

```
fir :: Num a => Array Int a -> Stream a -> Stream a
fir b inp = recurrence (listArray (0,l-1) (replicate l 0)) inp
                       (scalarProd b)
  where l = rangeSize (bounds b)
```

Implementing IIR filters requires a version of `recurrence` which also has a cyclic buffer for the elements of the output stream.

Fusion

The functional stream representation supports fusion, meaning that intermediate streams are removed. Stream fusion (Coutts, Leshchinskiy, and Stewart 2007) is a good demonstration of this although it employs a slightly more sophisticated stream representation.

Our new monadic representation also supports fusion in a similar fashion to the functional representation. A key difference is that two of the monad laws are essential to achieve good code generation: the left identity law and associativity of bind. Here is the left identity law:

```
do a <- return x
   f a
==
do f x
```

And the following demonstrates the associativity of bind:

```
do b <- do a <- m
           f a
   g b
```

```
==
do a <- m
   b <- f a
   g b
```

We demonstrate fusion using a concrete example: `map f . map g`. To refresh our memories we repeat the definition of `map` below:

```
map :: (a -> b) -> Stream a -> Stream b
map f (Stream init) = Stream $ do
  next <- init
  loop $ do
    a <- next
    return (f a)
```

What follows is a derivation of an efficient implementation of `map f . map g`. Each step is annotated with the law used in the transformation. In order to get fusion going we will apply `map f . map g` to a concrete but arbitrary stream `Stream init`.

```
map f (map g (Stream init))

=> { inlining map }

map f (Stream $ do
  next <- init
  loop $ do
    a <- next
    return (f a)

=> { inlining map }

Stream $ do
  next' <- do
    next <- init
    loop $ do
    a <- next
    return (g a)
  loop $ do
    b <- next'
    return (f b)

=> { bind associativity }

Stream $ do
```

```
  next <- init
  next' <- loop $ do
    a <- next
    return (g a)
  loop $ do
    b <- next'
    return (f b)

=> { loop = return, left identity }

Stream $ do
  next <- init
  loop $ do
    b <- do
      a <- next
      return (g a)
    return (f b)

=> { bind associativity }

Stream $ do
  next <- init
  loop $ do
    a <- next
    b <- return (g a)
    return (f b)

=> { left identity }

Stream $ do
  next <- init
  loop $ do
    a <- next
    return (f (g a))
```

The final result is as efficient as one can possibly hope for.

Fusing combinators other than `map` follows a similar pattern.

Avoiding Multiple Loop Variables

The stream representation already presented allows for mutation which improves efficiency of the generated code considerably. The generated code still suffers from a problem where fused functions will cause multiple loop indices to appear

in the same loop. An extra loop counter might be tolerable but the issue runs deeper than that and Lippmeier et al. report having seen eight loop counters appear in the wild (Lippmeier et al. 2013). Consider the following pattern:

```
foo arr = remember n $ .. $ cycle arr
```

The Feldspar-generated code for this pattern contains one loop index called `v4` originating from `remember` and one called `v5` originating from `cycle`:

```
for (uint32_t v4 = 0; v4 < v0; v4 += 1) {
    v5 = v3;
    v3 = ((v5 + 1) % v9);
    *out[v4] = v1[v5];
}
```

Good C compilers might remove multiple loop indices but relying on the C compiler to perform that optimization on signal processing applications is a risk. The mere presence of multiple loop indices might prevent earlier optimizations at the functional level from kicking in.

Parameterising the stream representation with the loop counter solves the problem:

```
data Stream a = Stream (IO (Int -> IO a))
```

The function performing allocation is responsible for providing the loop index. Here is the new version of `remember` for the new representation:

```
remember :: Int -> Stream a -> IO (Array Int a)
remember len (Stream init) = do
  arr  <- newArray_ (0,len-1)
  next <- init
  forM [0..len-1] $ \i -> do
    a <- next i
    writeArray arr i a
  freeze arr
```

The key difference from the previous version is that the loop variable `i` is fed to the step function `next` Functions like `cycle` can now take advantage of the provided loop index, and don't need to create their own loop variables:

```
cycle :: Array Int a -> Stream a
cycle arr = Stream $ do
  let l = length arr
  loop $ \i -> do
    return (arr!(i `mod` l))
```

The new code for `cycle` is considerably shorter and will also generate better code:

```
for (uint32_t v3 = 0; v3 < v0; v3 += 1) {
    *out[v3] = v1[v3 % v6];
}
```

Streams for EDSLs

Our new monadic representation of streams is a natural fit for embedded domain specific languages and works particularly well with the technique of combining shallow and deep embeddings (Svenningsson and Axelsson 2013). Monads can be embedded in an EDSL using the technique by Persson, Axelsson, and Svenningsson (2012). Embedding monads in this way is particularly attractive: the embedding of monads applies the two monad laws by evaluation in the host language. This means that the kind of rewriting explained in the Fusion section happens automatically, no extra code needs to be written in order to achieve the optimization.

Feldspar (Axelsson et al. 2011) has a stream library that uses a monadic embedding. Instead of the `IO` monad, it uses Feldspar's `M` monad (Persson, Axelsson, and Svenningsson 2012) for mutable effects:

```
data Stream a = Stream (M (M a))
```

The Feldspar implementation of a simple function like `map` is identical to the Haskell definition in this paper. For more complicated functions, the difference is mainly in the use of different types and different names for similar functions. For example, `cycle` is defined as follows in Feldspar:

```
cycle :: Syntax a => Pull DIM1 a -> Stream a
cycle vec = Stream $ do
  c <- newRef (0 :: Data Index)
  loop $ do
    i <- getRef c
    setRef c ((i + 1) `rem` length vec)
    return (vec ! (Z :. i))
```

A notable difference is the `Syntax` constraint on the type of the elements in the Feldspar implementation. This constraint is to restrict the function to elements that can be stored in memory when generating C code.

Using the monadic stream representation with EDSLs enables another trick not available with the functional representation: the buffer can be stored entirely in

references. In the versions of the moving average function we've presented so far the purely functional representation uses an immutable array as a buffer while the monadic representation uses a mutable cyclic buffer. However, in an EDSL the buffer can be represented as a Haskell list of mutable references, provided that the EDSL has support for references. Since the buffer is implemented as a Haskell list, the list will be traversed at EDSL compile time and not be present in the generated code. Below is a Feldspar version of the `recurrence` function which stores the buffer in references.

```
recurrenceS :: (Type a, Type b) =>
               [Data a] -> Stream (Data a) ->
               ([Data a] -> Data b) ->
               Stream (Data b)
recurrenceS ii (Stream init) mkExpr = Stream $ do
    next <- init
    ris  <- mapM newRef ii
    loop $ do
      a <- next
      if (not $ null ii) then pBuf ris a else return ()
      b <- wBuf ris $ \ib ->
             return $ mkExpr ib
      return b
  where
    pBuf rs a = zipWithM (\r1 r2 -> getRef r1 >>= setRef r2)
                         (tail $ reverse rs) (reverse rs)
             >> setRef (head rs) a
    wBuf rs f = mapM getRef rs >>= f
```

Without going into all details about how Feldspar is embedded, we focus on the use of references. The variable `ris` is bound to a list of references which are initialized by `mapM newRef ii`. The function `wBuf` is used to read from all references and pass the resulting list of values to a continuation. It is used similarly to `withBuf` in the cyclic buffer implementation. The workhorse in this implementation is `pBuf` which conceptually rotates the buffer one step and adds the latest element. It is achieved by shifting the values between references. This version is fast as we will see in the Evaluation section below.

Evaluation

We evaluate the performance of our monadic representation using Feldspar, a staged embedded domain specific language targeting digital signal processing algorithms. Feldspar is embedded in Haskell, and its stream library is conceptually similar to the code shown in this paper. The primary differences from the perspective of the performance evaluation is that Feldspar is a strict language

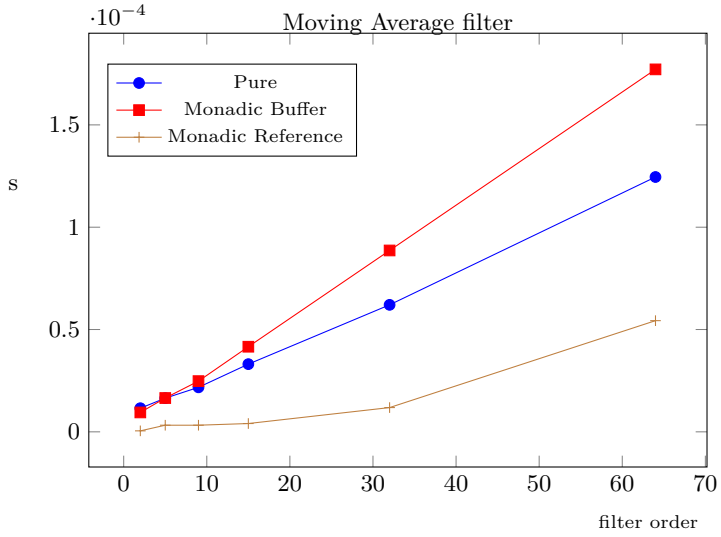

Fig. 1. Running time of filters compared to reference C implementations.

and that much of the overhead due to high-level data types like `Stream` is reduced at compile time.

We have measured the difference between functional and monadic streams on two different benchmarks: moving average and FIR filter. The measurements have been performed on a Linux desktop, equipped with a 3.5 GHz Intel Core i7-3770K and 16 GB 1600 MHz DDR3. One core is used throughout all benchmarks.

The results for the moving average is shown in Figure 1. The points labeled "Pure" show the results for the purely functional stream representation, while the points labeled "Monadic Buffer" show the results for the monadic streams using a cyclic buffer. Filter orders up to five are typical in digital signal processing applications but we tested the different implementations with a wide variety of buffer sizes to see the scalability of our technique. The monadic buffer version is slightly better for small buffer sizes but worse for large window sizes. The reason is that the cyclic buffer implementation uses the modulus operation frequently to ensure that the buffer is presented to the programmer with elements in the right order and not shifted. The number of modulus operations grows with the window size. The third set of points shows the result of an implementation where the buffer is kept entirely in references (see Streams for EDSLs). That version readily outperforms the two other versions, and is consistently at least 4× faster than the functional representation.

The FIR filter benchmark is presented in Figure 2. The "Pure" points again show the performance of purely functional stream. "Monadic" shows monadic streams where the buffer is stored in references. The monadic stream representation is superior up to filter orders around 50. Apart from the Feldspar version, we also

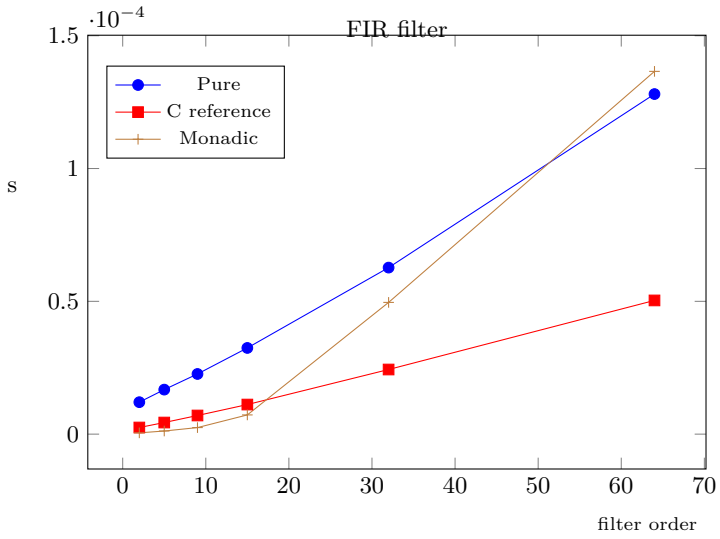

Fig. 2. Running time of filters compared to reference C implementations.

have a handwritten C benchmark to get a baseline for our measurements. However, it is not entirely an apples-to-apples comparison since the C implementation still stores the buffer in memory and uses loops to traverse it. Yet, the measurements give some indication of how performant our implementation is. For filter orders below 20 it pays off to unroll the loops and store the buffer in references.

Relation to Functional Streams

The functional representation of streams can be recovered from the monadic representation to shed new light on the monadic representation. Consider again the type `M (M a)`. The outer and the inner monads are the same but they could be different as long as the necessary functionality is provided by the respective monads. We could imagine a representation `M (N a)` where the outer monad `M` is responsible for initializing memory and the inner monad `N` is responsible for reading and writing that memory. We can let `M a` be `(a,s)` and `N a` be `(s -> (a,s))` if we forego mutation. We recognize them as the writer monad and the state monad. Combining these two monads results in the functional stream representation.

Finite Streams

Finite streams can also be represented by adding an extra length parameter to our monadic representation:

```
data Stream a = Stream (IO (IO a)) Int
```

Most function definitions for finite stream are similar to those for infinite streams, with the addition of passing around the length parameter. Additionally, functions like appending two streams now make sense, and it is possible to allocate the whole stream to memory.

A Pure Interface

We have used `IO` as the monad for mutability so far. We can get away with using the `ST` monad f we only care about mutability and not about general effects that affect the external world:

```
data Stream s a = Stream (ST s (ST s a))
```

We can reimplement all stream functions for this new representation by using `STRef` instead of `IORef` and `STArray` instead of `IOArray`. The advantage of using `ST` becomes visible in functions that consume streams, such as `remember`:[6]

```
remember :: Int -> (forall s . Stream s a) -> Array Int a
remember len str = runSTArray $ remember' len str

remember' :: Int -> Stream s a -> ST s (STArray s Int a)
remember' len (Stream init) = do
  arr  <- newArray_ (0,len-1)
  next <- init
  forM [0..len-1] $ \i -> do
    a <- next
    writeArray arr i a
  return arr
```

The result is now a pure `Array` value rather than a monadic one. This means that we can hide all uses of monads from the user and provide a pure interface to streams. However, in order to make use of mutability, one has still to write monadic code (or use canned solutions, such as `recurrence`).

Related Work

Lazy streams Streams can be represented succinctly in lazy languages like Haskell with the following definition:

[6] The helper function `remember'` is needed to get the types right.

```
data Stream a = Cons a (Stream a)
```

Lazy streams suffer from the same problem as the functional stream representation presented earlier in the paper: some algorithms are not possible to implement efficiently. The monadic variant of this type is:

```
type Stream a = M (Stream' a)

data Stream' a = Cons a (Stream a)
```

The above definition enables the use of mutation which allows for more efficient implementations of filters. However, recursive definitions are problematic in the context of code generating EDSLs. Our monadic representation of streams has the advantage of being usable even in an EDSL context.

Coiterative streams A coiterative representation of streams makes it possible to avoid recursion in the definition of streams and in functions defined for streams (Caspi and Pouzet 1998). Our initial representation in The Problem section was based on coiteration.

The stream fusion framework (Coutts, Leshchinskiy, and Stewart 2007) builds on the following coiterative representation:

```
data Stream a = forall s . Stream (s -> Step a s) s
```

The difference to our initial representation is the `Step` type that is returned by the step function. The `Step` type has three cases: (1) a pair of an element `a` and a new state `s`, or (2) just a new state, or (3) a value signaling that the stream has ended. This stream representation makes it possible to give efficient definitions of many stream operations and make sure that streams are fused when operations are composed. The stream fusion framework uses GHC rewrite rules to convert list-based code to stream-based code where possible.

In contrast to our work, stream fusion does not support streams with mutable state.

Effectful stream programming There are many Haskell libraries for dealing with streaming data, such as Fudgets (Carlsson and Hallgren 1993), Conduit (Snoyman 2014a), Pipes (Gonzalez) and Iteratees (Kiselyov 2012). Most of these libraries define streams over an underlying monad. Choosing `IO` as the underlying monad allows for the streaming programs to perform external communication. However, there is nothing stopping from using the `IO` monad also for "internal" effects, such as mutable state.

Stream representations such as the one in Conduits can describe more general networks than our `Stream` type (e.g. nodes with different input and output

rates). However, being based on recursive definitions, those stream programs are generally not guaranteed to fuse. Though, when certain requirements are met, conduits are subject to fusion (Snoyman 2014b).

The fusion framework in Conduits relies on GHC rules to rewrite recursive stream programs to corresponding programs based on a non-recursive stream type (an extension of the stream fusion representation above):

```
data Stream m o r = forall s . Stream (s -> m (Step s o r)) (m s)
```

This type is quite close to our `Stream` representation: the initialization action of type `m s` can be used to initialize mutable state, and the step function can be used to mutate this state. The main difference is that there is still immutable state of type `s` passed around, which is unnecessary if we put all the state in the monad.

FRP Functional Reactive Programming (FRP) was initially concieved as a way to program compositionally with time varying values where time is treated continuously (Elliott and Hudak 1997). In contrast, many implementations of FRP use a discrete notion of time (Nilsson, Courtney, and Peterson 2002; Czaplicki and Chong 2013; Patai 2011). Discrete time FRP and streams are similar in many respects, as explained by Wan and Hudak (2000). However, while the goal of FRP is often on expressivity, we use streams in the context of digital signal processing where we are happy to trade expressivity for efficiency. Perhaps some of the techniques presented in this paper can be applied to speed up FRP implementations; such investigations are future work.

EDSLs The stream representation in this paper is used by the stream library in the Feldspar EDSL (Axelsson et al. 2011). It is also used as an intermediate representation in recent work on adding data flow networks on top of Feldspar (Aronsson, Axelsson, and Sheeran 2015).

Conclusions

This paper presents a new monadic stream representation. It is motivated by algorithms in digital signal processing which require mutation to be implemented efficiently. Somewhat surprisingly, our measurements show that a straight-forward mutable implementation using a cyclic buffer is often slower than a purely functional copying implementation. However, the monadic representation enables a much more important optimization: keeping the buffer in references and unrolling the loop. For typical filter orders this implementation beats a handwritten C implementation, although performance degrades when parts of the buffer has to be stored in memory. Clearly, the implementation of a filter has to be chosen depending on its order.

Acknowledgements

This research was funded by the Swedish Foundation for Strategic Research (in the RAWFP project) and the Swedish Research Council.

References

Aronsson, Markus, Emil Axelsson, and Mary Sheeran. 2015. "Stream Processing for Embedded Domain Specific Languages." Presented at IFL 2014, to appear.

Axelsson, E., K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. 2011. "The Design and Implementation of Feldspar – an Embedded Language for Digital Signal Processing." In *IFL 2010*. Vol. 6647. LNCS.

Carlsson, Magnus, and Thomas Hallgren. 1993. "FUDGETS: A Graphical User Interface in a Lazy Functional Language." In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 321–30. FPCA '93. New York, NY, USA: ACM.

Caspi, Paul, and Marc Pouzet. 1998. "A Co-Iterative Characterization of Synchronous Stream Functions." *Electronic Notes in Theoretical Computer Science* 11 (0): 1–21.

Coutts, D., R. Leshchinskiy, and D. Stewart. 2007. "Stream fusion: From lists to streams to nothing at all." *Proc. 12th ACM SIGPLAN Int. Conf. on Functional Programming*. ACM.

Czaplicki, Evan, and Stephen Chong. 2013. "Asynchronous Functional Reactive Programming for GUIs." In *ACM SIGPLAN Notices*, 48:411–22. 6. ACM.

Elliott, Conal, and Paul Hudak. 1997. "Functional Reactive Animation." In *ACM SIGPLAN Notices*, 32:263–73. 8. ACM.

Gonzalez, Gabriel. "Pipes Library." http://hackage.haskell.org/package/pipes.

Kiselyov, Oleg. 2012. "Iteratees." In *Functional and Logic Programming*, edited by Tom Schrijvers and Peter Thiemann, 7294:166–81. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Lippmeier, Ben, Manuel M. T. Chakravarty, Gabriele Keller, and Amos Robinson. 2013. "Data Flow Fusion with Series Expressions in Haskell." In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, edited by Chung-chieh Shan, 93–104. ACM.

Nilsson, Henrik, Antony Courtney, and John Peterson. 2002. "Functional Reactive Programming, Continued." In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, 51–64. ACM.

Patai, Gergely. 2011. "Efficient and Compositional Higher-Order Streams." In *Functional and Constraint Logic Programming*, 137–54. Springer.

Persson, Anders, Emil Axelsson, and Josef Svenningsson. 2012. "Generic Monadic Constructs for Embedded Languages." In *Implementation and Application of Functional Languages*, edited by Andy Gill and Jurriaan Hage, 7257:85–99. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Snoyman, Michael. 2014a. "Conduit Overview." https://www.fpcomplete.com/user/snoyberg/library-documentation/conduit-overview.

———. 2014b. "Conduit Stream Fusion." https://www.fpcomplete.com/blog/2014/08/conduit-stream-fusion.

Svenningsson, Josef, and Emil Axelsson. 2013. "Combining Deep and Shallow Embedding for EDSL." In *Trends in Functional Programming*, 21–36. Springer Berlin Heidelberg.

Wan, Zhanyong, and Paul Hudak. 2000. "Functional Reactive Programming from First Principles." In *ACM SIGPLAN Notices*, 35:242–52. 5. ACM.