Efficient and Compositional Higher-Order Streams

Gergely Patai

Budapest University of Technology and Economics, Budapest, Hungary
patai@iit.bme.hu

Abstract. Stream-based programming has been around for a long time, but it is typically restricted to static data-flow networks. By introducing first-class streams that implement the monad interface, we can describe arbitrary dynamic networks in an elegant and consistent way using only two extra primitives besides the monadic operations. This paper presents an efficient stream implementation and demonstrates the compositionality of the constructs by mapping them to functions over natural numbers.

1 Introduction

One of the major advantages offered by pure functional programming is the possibility of equational reasoning in software development. This is achieved by enforcing referential transparency, which precludes the use of observable mutable state. However, while pure functions can easily describe the transformation of one data structure into another, interactive and embedded applications have to deal with input and output living in a time domain and describe temporal as well as functional dependencies. In practice, this means that the state of the computation has to be maintained and regularly projected on the output, be it a user interface, an actuator or a consumer process.

Stream-based programming is an approach that avoids the introduction of a monolithic world state. The basic idea is that every variable in the program represents *the whole lifetime* of a time-varying value. For instance, an expression like $x + y$ might describe a combinational network that takes two input streams and outputs a stream defined by their point-wise sum. On the implementation level, x and y can be represented with mutable variables, but the stream concept allows us to compose them as pure values.

Traditional stream-based languages like Lustre [8] allow us to describe static data-flow networks and compile them into efficient loops. In many cases this is not enough. As soon as we want to describe dynamically reconfigurable systems or a collection of entities that changes over time – and such a need can come up even in a relatively simple system like a sensor network node –, we need more expressive power.

Functional reactive programming is essentially an extension of the stream-based approach that adds higher-order constructs in some form. Its first incarnation, Fran [6], introduced time-varying values as first-class entities. Probably

the most important lesson of Fran was the realisation that the start times of streams must be treated with care. We can either decide to work with global time (fix the start time of every stream to the beginning of execution) or local time (have every stream count time from its own creation). Global-time streams are naturally composable, but it is easy to create space and time leaks with them. For instance, if we are allowed to synthesise the integral of an input in the middle of the execution, we have to keep all the past input in the memory (space leak), and the freshly created integrator has to catch up immediately by summing over all that past (time leak). Local-time semantics is arguably more intuitive and does not suffer from this problem, but it can easily break referential transparency if introduced naively: the meaning of $x + y$ can change if x or y is translated in time.

This paper presents an approach that unites the advantages of the two worlds: referentially transparent ('compositional') higher-order streams without space or time leaks ('efficient'). First we will discuss the original problem in more detail (Section 2), then we will derive a minimal set of combinators to describe a wide range of dynamic networks that can be potentially realised without a major performance hit (Section 3). The power of the resulting interface is illustrated through a longer example (Section 4). Afterwards, we can move to the implementation side; we will see the difficulties of a pure approach (Section 5), and outline an impure runnable implementation of the idea, making some practical adjustments to the interface on the way (Section 6). We conclude the discussion with a short evaluation (Section 7) and comparison to related work (Section 8).

2 Problems with Higher-Order Streams

For our purposes, a stream is an object that can be observed through two functions, *head* and *tail*, which extract its first element and its immediate continuation, respectively. If the type of the elements is a, the type of the stream is referred to as *Stream* a.

Streams are applicative functors [11], i.e. we can define two constructors, *pure* and $\circledast$, to describe the stateless combination of streams. In order to construct arbitrary causal static networks, we need only one more building block: a unit delay with initialisation. We will refer to it as *cons*, since cons streams are the trivial implementation of this interface. Feedback can be directly expressed through value recursion, so we do not need a specific fixed-point operator.

The exact meaning of the constructors can be defined coinductively through their interaction with the destructors, which is shown on Figure 1; we denote a stream s as $\langle s_0\ s_1\ s_2\ s_3 \dots \rangle$. If s is of type *Stream* a, then s_i is of type a.

Things get more interesting if we want to add dynamism to the network structure. If we can define an operation to flatten higher-order streams, we can model the changing topology by a stream that delivers a potentially changing part of the network at every point of time. The flattening operation turns the higher-order stream into the actual dynamic network. Not surprisingly, in order to end up with intuitive behaviour, the combinator we are looking for

$$\begin{array}{ccc}
\langle x\ s_0\ s_1\ s_2\ s_3 \ldots \rangle & \langle x\ x\ x\ x\ x \ldots \rangle & \langle (f_0\ x_0)\ (f_1\ x_1)\ (f_2\ x_2)\ (f_3\ x_3) \ldots \rangle \\
\\
cons\ x\ s & pure\ x & f \circledast x \\
\\
\begin{array}{r} head\ (cons\ x\ s) \equiv x \\ tail\ (cons\ x\ s) \equiv s \end{array} & \begin{array}{r} head\ (pure\ x) \equiv x \\ tail\ (pure\ x) \equiv pure\ x \end{array} & \begin{array}{r} head\ (f \circledast x) \equiv (head\ f)\ (head\ x) \\ tail\ (f \circledast x) \equiv tail\ f \circledast tail\ x \end{array}
\end{array}$$

Fig. 1. First-order stream constructors

should obey the laws of the monadic *join* operation. The laws basically state that in case we have several layers, it does not matter which order we break them down in, and *join* commutes with point-wise function application (*fmap*, where $fmap\ f\ s \equiv pure\ f \circledast s$), so it does not matter either whether we apply a stateless transformation before or after collapsing the layers.

Finding the appropriate *join* operation is straightforward if we build on the isomorphism between the types $Stream\ a$ and $\mathbb{N} \to a$. All we need to do is adapt the monad instance of functions with a fixed input type (aka the reader monad) to streams, where *head* corresponds to applying the function to zero, while *tail* means composing it with *succ*. Just as the other constructors, *join* can be defined through its observed behaviour:

$$\begin{aligned} head\ (join\ s) &\equiv head\ (head\ s) \\ tail\ (join\ s) &\equiv join\ (fmap\ tail\ (tail\ s)) \end{aligned}$$

In other words, $join\ s = \langle s_{00}\ s_{11}\ s_{22}\ s_{33} \ldots \rangle$, the main diagonal of the stream of streams s.

This is all fine, except for the sad fact that the above definition, while technically correct, is completely impractical due to efficiency reasons: the n^{th} sample takes n^2 steps to evaluate, since every time we advance in the stream, we prepend a call to *tail* for every future sample.

The problem is that each sample can potentially depend on all earlier samples (cf. recursive functions over $\mathbb{N}$), so there are no shortcuts to take advantage of in general. The only optimisation that can help is special-casing for constant streams. However, it would only make *join* efficient in trivial cases that do not use the expressive power of the monadic interface, hence it is no real solution. Instead, we have to find a different set of constructors, which can only generate structures where the aforementioned shortcut is always possible.

3 Doing without Cons

First of all, we will assume that streams are accessed sequentially. The key to finding a leak-free constructor base is realising that we do not need to sample the past of the newly created streams. We want to ensure that whenever a new stream is synthesised, its origin is defined to be the point of creation. However, in order to maintain referential transparency, we will have to work with global

time. The basic idea is to map local-time components on the global timeline, effectively keeping track of their start times.

The monadic operations are all safe, because they are stateless, hence time invariant. The cause of our problems is *cons*, which has no slot to encode its start time, which is therefore implicitly understood to be zero. But this means that there is nothing to stop us from defining a stateful stream that starts in the past, so we will have to disallow *cons* altogether and look for a suitable alternative.

Let us first think about how starting time affects streams. For simplicity, we will step back and represent streams as functions of time, i.e. $\mathbb{N} \to a$, and try to derive a sensible interface using *denotational design* [4].

By choosing a representation we also inherit its class instances, so the monadic operations are readily available. Let us introduce a local-time memory element called *delay*. In order to express its dependence on the start time, we can simply pass that time as an extra argument:

$$
\begin{aligned}
&\mathit{delay}\ x\ s\ t_{start}\ t_{sample} \\
&\quad \mid t_{start} \equiv t_{sample} &&= x \\
&\quad \mid t_{start} < t_{sample} &&= s\ (t_{sample} - 1) \\
&\quad \mid \mathit{otherwise} &&= \mathit{error}\ \texttt{"Premature sample!"}
\end{aligned}
$$

There is a fundamental change here: *delay* x s – unlike *cons* x s – is not a stream, but a *stream generator*. In this model, stream generators are of type $\mathbb{N} \to a$, functions that take a starting time and return a data structure that might hold freshly created streams as well as old ones. Why not limit them to $\mathbb{N} \to \mathit{Stream}\ a$? Because that would limit us to generate one stream at each step. A stream that carries a list of values is not equivalent to a list that contains independent streams, since the former solution does not allow us to manage the lifetimes of streams separately. There is no way to tell if no-one ever wants to read the first value of the list, therefore we would not be able to get rid of it, while independent streams could be garbage collected as soon as all references to them are lost.

Since the types of streams and stream generators coincide while the arguments of the functions have different meanings (sampling time and starting time, respectively), let us introduce type synonyms to prevent confusion:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Stream}\ a = \mathbb{N} \to a \\
&\textbf{type}\ \mathit{StreamGen}\ a = \mathbb{N} \to a
\end{aligned}
$$

Using the new names, the type of *delay* can be specified the following way:

$$\mathit{delay} :: a \to \mathit{Stream}\ a \to \mathit{StreamGen}\ (\mathit{Stream}\ a)$$

It is clear from the above definition that a *delay* is not safe to use if its start time is in the future, therefore we should not allow generators to be used in arbitrary ways. The only safe starting time is the smallest possible value, which

will lead us to the sole way of directly executing a stream generator: passing it zero as the start time.

$$start :: StreamGen\ (Stream\ a) \to Stream\ a$$
$$start\ g = g\ 0$$

While there is no immediate need to restrict its output type, we basically want a function that is used only once at the beginning to extract the top-level stream. This makes our interface as tight as possible.

Of course, *start* is not sufficient by itself, since it does not give us any means to create streams at any point later than the very first sampling point. We need another combinator that can extract generators in a disciplined way, ensuring that no unsafe starting times can be specified. The most basic choice is simply extracting a stream of generators by passing each the current sampling time:

$$generator :: Stream\ (StreamGen\ a) \to Stream\ a$$
$$generator\ g\ t_{sample} = g\ t_{sample}\ t_{sample}$$

As it turns out, in our model *generator* happens to coincide with *join* for functions. However, this does not hold if *Stream* and *StreamGen* are different structures, so *generator* remains an essential combinator.

Of course, if we look at it from a different direction, we can say that stream generators can also be considered streams. In this case, *start* is simply equivalent to *head* (which also hints at the fact that *start* is not a constructor), *generator* is the same as *join*, and *delay* constructs a stream of streams. We also inherit the fixed-point combinator *mfix* [7] from the reader monad, which will be necessary to define feedback loops. In the end, we can reduce the combinator base required to work with higher-order dataflow networks to the one on Figure 2; we use *return* and $\gg\!=$ in the minimal base, since they can express the applicative combinators as well as *join*.

Even though the four-combinator base is quite elegant, the distinction between streams and stream generators is still useful. The main difference between *join* (on streams) and *generator* is that the latter is used to create new streams, while the former can only sample existing streams. Also, we will see that *mfix* for stream generators allows us to define streams in terms of each other (e.g. express the circular dependency between position, velocity and acceleration in the context of spring motion), while *mfix* for streams has no obvious practical application.

4 A Motivating Example

Let us now forget about the representations we chose and keep only the interfaces. These are quite small: both streams and stream generators are *MonadFix* instances, and we have *delay*, *generator* and *start* to work with.

We saw that *delay* is the only operation that lives in the *StreamGen* monad, and it allows us to create a new stateful stream every time the monad is extracted. As a simple example, we can start by defining a generic stateful stream (`$` stands for infix *fmap*).

$$\begin{array}{l}\langle\langle\ x\ s_0\ s_1\ s_2\ s_3\ \ldots\rangle \\ \ \langle\ \bot\ x\ s_1\ s_2\ s_3\ \ldots\rangle \\ \ \langle\ \bot\ \bot\ x\ s_2\ s_3\ \ldots\rangle \\ \ \langle\ \bot\ \bot\ \bot\ x\ s_3\ \ldots\rangle \\ \ \ldots\qquad\qquad\qquad\rangle\end{array} \qquad \langle x\ x\ x\ x\ x\ldots\rangle \qquad \begin{array}{c}\langle y_{00}\ y_{11}\ y_{22}\ y_{33}\ y_{44}\ldots\rangle \\ \textbf{where}\ y_i = f\ s_i\end{array}$$

$$\mathit{delay}\ x\ s \qquad\qquad \mathit{return}\ x \qquad\qquad s \gg\!= f$$

$$\begin{array}{l}\langle(\mathit{fix}\ (\mathit{nth}\ 0 \circ f))\ (\mathit{fix}\ (\mathit{nth}\ 1 \circ f))\ (\mathit{fix}\ (\mathit{nth}\ 2 \circ f))\ (\mathit{fix}\ (\mathit{nth}\ 3 \circ f))\ldots\rangle \\ \quad \textbf{where}\ \mathit{nth}\ 0\ s = \mathit{head}\ s \\ \qquad\quad \mathit{nth}\ n\ s = \mathit{nth}\ (n-1)\ (\mathit{tail}\ s)\end{array}$$

$$\mathit{mfix}\ f$$

$$\begin{array}{rl}\mathit{head}\ (\mathit{delay}\ x\ s) \equiv \mathit{cons}\ x\ s & \qquad \mathit{head}\ (\mathit{return}\ x) \equiv x \\ \mathit{tail}\ (\mathit{delay}\ x\ s) \equiv \mathit{fmap}\ (\mathit{cons}\ \bot)\ (\mathit{delay}\ x\ (\mathit{tail}\ s)) & \qquad \mathit{tail}\ (\mathit{return}\ x) \equiv \mathit{return}\ x\end{array}$$

$$\begin{array}{rl}\mathit{head}\ (s \gg\!= f) \equiv \mathit{head}\ (f\ (\mathit{head}\ s)) & \qquad \mathit{head}\ (\mathit{mfix}\ f) \equiv \mathit{fix}\ (\mathit{head} \circ f) \\ \mathit{tail}\ (s \gg\!= f) \equiv \mathit{tail}\ s \gg\!= (\mathit{tail} \circ f) & \qquad \mathit{tail}\ (\mathit{mfix}\ f) \equiv \mathit{mfix}\ (\mathit{tail} \circ f)\end{array}$$

Fig. 2. Higher-order stream constructors

$$\begin{array}{l}\mathit{stateful} :: a \to (a \to a) \to \mathit{StreamGen}\ (\mathit{Stream}\ a) \\ \mathit{stateful}\ x_0\ f = \mathit{mfix}\ \$\ \lambda \mathit{str} \to \mathit{delay}\ x_0\ (f \circledS\ \mathit{str})\end{array}$$

Simply put, a stream generated by $\mathit{stateful}\ x_0\ f$ starts out as x_0, and each of its subsequent outputs equals f applied to the previous one. Note that we could not define *stateful* using direct recursion, since *delay* adds an extra monadic layer, so we had to rely on *StreamGen* being *MonadFix*.

Let us run a little test. This is the only place where we rely on functions being our representation, since we pass the generated stream to *map*.

$$\begin{array}{l}\mathit{strtest} :: \mathit{StreamGen}\ (\mathit{Stream}\ a) \to [a] \\ \mathit{strtest}\ g = \mathit{map}\ (\mathit{start}\ g)\ [0\,..\,15] \\ > \mathit{strtest}\ \$\ \mathit{stateful}\ 2\ (+3) \\ [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47]\end{array}$$

This looks promising, but a more complex example involving higher-order streams would be more motivating. Let us create a dynamic collection of countdown timers, where each expired timer is removed from the collection. First we will define named timers:

$$\begin{array}{l}\mathit{countdown} :: \mathit{String} \to \mathit{Int} \to \mathit{StreamGen}\ (\mathit{Stream}\ (\mathit{String}, \mathit{Maybe}\ \mathit{Int})) \\ \mathit{countdown}\ \mathit{name}\ t = \textbf{do} \\ \quad \textbf{let}\ \mathit{tick}\ \mathit{prev} = \textbf{do}\ \{t \leftarrow \mathit{prev}; \mathit{guard}\ (t > 0); \mathit{return}\ (t-1)\} \\ \quad \mathit{timer} \leftarrow \mathit{stateful}\ (\mathit{Just}\ t)\ \mathit{tick} \\ \quad \mathit{return}\ ((,)\ \mathit{name} \circledS\ \mathit{timer}) \\ > \mathit{strtest}\ \$\ \mathit{countdown}\ \texttt{"foo"}\ 4\end{array}$$

$[(\texttt{"foo"}, \mathit{Just}\ 4), (\texttt{"foo"}, \mathit{Just}\ 3), (\texttt{"foo"}, \mathit{Just}\ 2), (\texttt{"foo"}, \mathit{Just}\ 1),$
$(\texttt{"foo"}, \mathit{Just}\ 0), (\texttt{"foo"}, \mathit{Nothing}), (\texttt{"foo"}, \mathit{Nothing}), \ldots]$

Next, we will define a timer source that takes a list of timer names, starting values and start times and creates a stream that delivers the list of new timers at every point. Naturally, it would be more efficient to consume the original list as we advance in the stream, but mapping over a simple counter will do for now.

```
timerSource :: [(String, Int, Int)] →
               StreamGen (Stream [Stream (String, Maybe Int)])
timerSource ts = do
  let gen t = mapM (uncurry countdown) newTimers
        where newTimers = [(n, v) | (n, v, st) ← ts, st ≡ t]
  cnt ← stateful 0 (+1)
  return $ generator (gen ⊛ cnt)
```

Now we need to encapsulate the timer source stream in another stream expression that takes care of maintaining the list of live timers. Since working with dynamic collections is a recurring task, let us define a generic combinator that maintains a dynamic list of streams given a source and a test that tells from the output of each stream whether it should be kept. We can use **μdo** expressions (a variant of **do** expressions allowing forward references) as syntactic sugar for *mfix* to make life easier.

```
collection :: Stream [Stream a] → (a → Bool) → StreamGen (Stream [a])
collection source isAlive = μdo
  coll ← liftA2 (++) source ⊛ delay [] coll'
  let collWithVals = zip ⊛ (sequence =≪ coll) ⊛ coll
      collWithVals' = filter (isAlive ∘ fst) ⊛ collWithVals
      coll' = map snd ⊛ collWithVals'
  return $ map fst ⊛ collWithVals'
```

We need recursion to define the *coll* stream as a delayed version of *coll′*, which represents its continuation. At every point of time its output is concatenated with that of the source (we need to lift the (++) operator twice in order to get behind both the *StreamGen* and the *Stream* abstraction). Then we define *collWithVals*, which simply pairs up every stream with its current output. The output is obtained by extracting the current value of the stream container and sampling each element with *sequence*. We can then derive *collWithVals′*, which contains only the streams that must be kept for the next round along with their output. By throwing out the respective parts, we can get both the final output and the collection for the next step (*coll′*).

Now we can easily finish our task:

```
timers :: [(String, Int, Int)] → StreamGen (Stream [(String, Int)])
timers timerData = do
  src ← timerSource timerData
```

```
  getOutput Ⓢ collection src (isJust ∘ snd)
    where getOutput = fmap (map (λ(name, Just val) → (name, val)))
```

Let us start four timers as a test: 'a' at $t = 0$ with value 3, 'b' and 'c' at $t = 1$ with values 5 and 3, and 'd' at $t = 3$ with value 4:

```
> strtest $ timers [("a", 3, 0), ("b", 5, 1), ("c", 3, 1), ("d", 4, 3)]
[[("a", 3)], [("b", 5), ("c", 3), ("a", 2)], [("b", 4), ("c", 2), ("a", 1)],
 [("d", 4), ("b", 3), ("c", 1), ("a", 0)], [("d", 3), ("b", 2), ("c", 0)],
 [("d", 2), ("b", 1)], [("d", 1), ("b", 0)], [("d", 0)], [], [], [], [], [], [], [], []]
```

While it might look like a lot of trouble to create such a simple stream, it is worth noting that *timers* is defined in a modular way: the countdown mechanism that drives each timer is completely separated, and we do not even need to worry about explicitly updating the state of the timers. Similarly, the timer source is also independent of the dynamic list, and we even defined a generic combinator that turns a source of streams into a dynamic collection of streams.

A valid concern about the interface could be the ubiquitous use of lifting. There are basically two ways to deal with this problem. First of all, since streams are monads, we could simply use **do** notation to extract the current samples of streams we need and refer to those samples directly afterwards. However, this would no doubt make the code less succinct. The other option is using syntactic support for applicative functors, e.g. *idiom brackets* [11]. For instance, the Strathclyde Haskell Enhancement [10] allows us to rewrite *collection* with a somewhat lighter applicative notation:

```
collection :: Stream [Stream a] → (a → Bool) → StreamGen (Stream [a])
collection source isAlive = μdo
  dcoll′ ← delay [] coll′
  let coll = (| source ++ dcoll′ |)
      collWithVals = (| zip (sequence =≪ coll) coll |)
      collWithVals′ = (| filter ~(isAlive ∘ fst) collWithVals |)
      coll′ = (| map ~snd collWithVals′ |)
  return (| map ~fst collWithVals′ |)
```

5 Problems with Purity

Our goal is to come up with a way to execute any higher-order stream representable by our constructors in a way that the computational effort required for a sample is at worst proportional to the size of the network. In terms of equations, this means that applying *tail* to a stream several times results in an expression that is roughly the same size as the original. This clearly does not hold according to the rules given on Figure 2, since most streams keep growing without bounds.

Thanks to the way the *delay* combinator is designed, all higher-order streams have a main diagonal that does not depend on any element on its left – that

is what makes the shortcut possible in theory. This observation points into the direction of a solution where we skew every higher-order stream in a way that its main diagonal becomes its first column, as illustrated on Figure 3. However, if we want to implement this idea directly, we will quickly run into roadblocks.

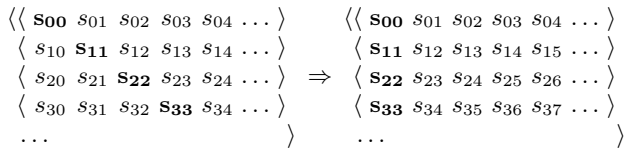

Fig. 3. Skewing higher-order streams to get an efficient *join*

The new definition of *delay* is straightforward: we simply do not add the triangular $\perp$ padding, which can be achieved by rewriting the second rule in its definition to $tail\ (delay\ x\ s) \equiv delay\ x\ (tail\ s)$. If we use *fmap*, *return* and *join* for our monadic combinator base, we will find that *join* is also easy to adapt to the skewed version: $tail\ (join\ s) \equiv join\ (tail\ s)$, since *join* now extracts the first column, shown in bold on Figure 3. This definition of *join* obeys the law of associativity, which is promising.

Somewhat surprisingly, the first bump comes when we want to describe the behaviour of *return*, which was trivial in the original model. In order to respect the monad laws, we have to ensure that if *return* is applied to a stream, then the resulting higher-order stream must contain the aged versions of the original, so the first column (which could be extracted by a subsequent *join*) is the same as the stream passed to *return*. In fact, the situation is even more complicated: the argument of *return* can be an arbitrary structure, and we have to make sure that all streams referenced in this structure are properly aged. Given a sufficiently flexible type system, the problem of *return* can be solved at a price of some extra administrative burden on the programmer. For instance, type families in Haskell can be used to define *head* and *tail* operations that traverse any structure and project every stream in the structure to its head or tail.

Unfortunately, we have two combinators, *fmap* and *mfix*, which require us to skew a higher-order stream that is produced by a function. This appears in the original rules as a composition with *tail*. The net effect of the composition is that every stream referenced in the closure is aged in each step, so in order to have a pure implementation, we would have to open up the closure, store all the streams in question alongside the function, age all these streams by applying *tail* to them in each step, and reconstruct the closure by applying the function to the aged streams.

While it is possible to go down the road outlined above, it would essentially amount to adding a layer of interpretation. However, what we are trying to do here is basically emulating mutable references, so we could as well pick the straightforward solution and use mutable variables to represent streams. The following section presents a solution in non-portable (GHC specific) Haskell.

6 Making It Run

We used the type $\mathbb{N} \to a$ to model streams. Since in the case of sequential sampling the index is implicit, we can drop the argument of the function and substitute it with side effects. This will give us the actual type:

$$\textbf{newtype}\ \mathit{Stream}\ a = S\ (\mathit{IO}\ a)\ \textbf{deriving}\ (\mathit{Functor}, \mathit{Applicative}, \mathit{Monad})$$

Every stream is represented by an action that returns its current sample. Just as it was the case in our model above, this operation has to be idempotent within each sample, and the *Stream* monad has to be commutative, i.e. it should not matter which order we sample different streams in. Both criteria are trivially fulfilled if the action has no other side effects than caching the sample after the first reading.

To be able to fill this action with meaning, we have to think about building and executing the stream network. Since *delay* is a stateful combinator, it has to be associated with a mutable variable. Delay elements are created in the *StreamGen* monad, which is therefore a natural source for these variables. But what should the internal structure of *StreamGen* look like?

When we execute the network, we have to update it in each sampling step in a way that preserves consistency. In effect, this means that no sampling action can change its output until the whole network is stepped, since streams can depend on each other in arbitrary ways. The solution is a two-phase superstep: first we go through all the variables and update them for the next round while preserving their current sample, and we discard these samples in a second sweep. We should sample the output of the whole network before the two sweeps. This means that each delay element requires the following components:

1. a mutable variable
2. a sampling action that produces its current value
3. an updating action that does not change the output of the sampling action, but creates all the data needed for the next step
4. a finalising action that concludes the update and advances the stream for the next superstep, discarding the old sample on the way

As we have seen, the sampling action is stored in the stream structure, and it obviously has to contain a reference to the variable, so the first two items are catered for. The updating and finalising actions have to be stored separately, because they have to be executed regardless of whether the stream was sampled in the current superstep or not – we certainly do not want the behaviour of any stream to be affected by how we observe it.

In short, we need to maintain a pool for these update actions. We have to make sure that whenever all references to a stream are lost, its update actions are also thrown away. This is easy to achieve using weak references to the actions, where the key is the corresponding stream. It should be noted that weak references constitute the non-portable bit, and they also force us into the IO monad even if our stream network has no external input. Summing up, our *update pool*

is a list of weak references that point us to the updating and finalising actions required for the superstep.

```
type UpdatePool = [Weak (IO (), IO ())]
```

The next step is to integrate the update pool with the sampling actions. One might first think that *StreamGen* should be a state monad stacked on top of IO, with the pool stored in the state. However, after creating the executable stream structure, it does not live in the *StreamGen* monad any more, therefore there is no way to thread this state through the monads extracted by *generator*. We have no other choice but to provide a mutable reference to the pool associated with the network the generator lives in.

```
newtype StreamGen a = SG {unSG :: IORef UpdatePool → IO a}
```

This is in fact a reader monad stacked on top of IO, but we will just use a 'raw' function for simplicity; we can easily define the equivalent *Monad* and *MonadFix* instances by emulating the behaviour of the reader monad transformer.

In order to be able to embed the stream framework in our applications, we need a function to turn a *StreamGen*-supplied stream into an IO computation that returns the next sample upon each invocation. All we need to do is create a variable for the update pool (initialised with an empty list), extract the top-level signal from its generator and assemble an IO action that performs the superstep as described above. The type of *start* is changed to reflect its new meaning.

```
start :: StreamGen (Stream a) → IO (IO a)
start (SG gen) = do
  pool ← newIORef []
  S sample ← gen pool
  return $ do
    let deref ptr = (fmap ∘ fmap) ((,) ptr) (deRefWeak ptr)
    ‡ Extracting the top-level output
    res ← sample
    ‡ Extracting the live references and throwing out the dead ones
    (ptrs, acts) ← unzip ∘ catMaybes $ (mapM deref =≪ readIORef pool)
    writeIORef pool ptrs
    ‡ Updating variables
    mapM_ fst acts
    ‡ Finalising variables
    mapM_ snd acts
    return res
```

The *deRefWeak* function returns *Nothing* when its key is unreachable, so we have to pair up every reference with the data it points to in order to be able to tell which ones we need to keep. The rest is straightforward: we extract the current output then perform the two sweeps. But how do the updating actions look like inside?

First of all, the mutable variable has to hold a data structure capable of encoding the phases. Since simple delays are the only stateful entities we have to worry about, the structure is straightforward.

$$\textbf{data}\ \mathit{Phase}\ a = \mathit{Ready}\ \{\mathit{state} :: a\} \mid \mathit{Updated}\ \{\mathit{state} :: a, \mathit{cache} :: a\}$$

A stream is either *Ready*, waiting to be sampled, or *Updated*, holding its next state and remembering its current sample. It is definitely *Updated* after executing the first sweep action, and definitely *Ready* after finalisation. In the case of *delay* we also have to make sure that the delayed signal is not sampled before the update phase, otherwise we would introduce an unnecessary dependency into the network that would cause even well-formed loops to be impossible to evaluate.

While the full implementation cannot fit in the paper, it is worth looking at the internals of at least one of the primitives. The simpler one is *delay*, where sampling is trivial and updating triggers a sampling on the delayed stream. It is advisable to force the evaluation of this sample, otherwise huge thunks might build up if a stateful stream is not read by anyone for a long time.

```
delay :: a → Stream a → StreamGen (Stream a)
delay x0 (S s) = SG $ λpool → do
  ref ← newIORef (Ready x0)
  let upd = readIORef ref >>= λv → case v of
        Ready x → s >>= λx' → x' `seq` writeIORef ref (Updated x' x)
        _       → return ()
      fin = readIORef ref >>= λ(Updated x _) → writeIORef ref $! Ready x
      str = S $ readIORef ref >>= λv → case v of
        Ready x     → return x
        Updated _ x → return x
  updateActions ← mkWeak str (upd, fin) Nothing
  modifyIORef pool (updateActions:)
  return str
```

As for *generator*, even though it is stateless, it has to know about the update pool, since its job is to populate it from time to time. Consequently, it has to live in *StreamGen*, just like *delay*. Also, it is a good idea to cache the result of its first sampling, otherwise it would create the same streams over and over, as many times as it is referenced. While this would not affect the output, it would certainly harm performance. In terms of the implementation this means that *generator* requires a variable just like *delay*. Its final type is the following:

$$\mathit{generator} :: \mathit{Stream}\ (\mathit{StreamGen}\ a) \to \mathit{StreamGen}\ (\mathit{Stream}\ a)$$

Upon the first sampling it executes the current snapshot of the monad in its associated stream by passing it the reference to the update pool, and stores the result by flipping the state to *Updated*, keeping the state $\bot$ all the time.

There are two problems left. First of all, since we have not considered efficiency issues during the design phase, we left a source of redundant computations in the system. If the result of an applicative operation is used in more than one place, it will be recalculated each time it is requested (e.g. the *collWithVals′* stream in the *collection* function, Section 4), because functions are not memoised by default. We can overcome this problem by introducing a third primitive called *memo*, which is observationally equivalent to *return* within *StreamGen*, but it adds a cache to any stream we pass to it.

$$memo :: Stream\ a \to StreamGen\ (Stream\ a)$$

The last issue is embedding streams in the real world. While we already have a way to read the output of a data-flow network, we cannot feed values into it. Fortunately, there is a simple standard solution in reactive programming frameworks: we can define a stream whose current reading can be explicitly set from outside using an associated sink action:

```
external :: a → IO (Stream a, a → IO ())
external x = do
  ref ← newIORef x
  return (S (readIORef ref), writeIORef ref)
```

An *external* stream is basically equivalent to a constant if we look at it from inside the network, therefore it does not have to be connected to the update mechanism in any way.

7 Closing Thoughts

A major advantage of using self-contained actions for sampling and updating the individual network nodes is that part of the evaluator is trivial to parallelise. We could introduce an update pool for each core, and execute actions in parallel, only synchronising at the end of each update sweep. The only change needed to make the code thread safe is to introduce an MVar (a variable protected with a mutex) to store actions generated during the update, and distribute these actions among the pools. As for the sampling phase, we would need to track the dependencies of the streams and possibly work in a bottom-up fashion as much as possible.

The biggest problem with the system is that it keeps updating streams even if they are not referenced any more until the garbage collector physically gets rid of them. The solution is not clear yet, and it might require more direct runtime support than general weak references.

The code is available in executable form in the experimental branch of the Elerea library[1] [12], which is available through *cabal-install*. For historical reasons, the library uses the name *Signal* instead of *Stream*, but the code is otherwise identical.

[1] see the `FRP.Elerea.Experimental.Simple` module

8 Related Work

While first-order stream-based languages are well understood, there has been comparatively little effort going into introducing first-class streams. Functional reactive programming, pioneered by Fran [6], brought a change in this regard, but the question of start times was ignored first, leading to great confusion about the interface. Also, FRP systems never had a focus on providing general higher-order constructs, and efficiency issues with Fran-like systems (besides the above mentioned confusion) led to Yampa [3], which is essentially a first-order stream library amended with switching combinators to describe dynamic networks. However, fitting the switching combinators into the arrow framework used by Yampa is not without problems, e.g. they have no place in the *causal commutative normal form* [9] of a signal function. Reactive [5] is a recent reformulation of Fran that fixes the starting time of all varying quantities to zero, therefore its general higher-order capabilities (i.e. monad instances of behaviours) are of little use in practice. FrTime [2] relies on a specialised evaluator to be able to use signals inside expressions, and it uses lexical scope to determine starting time. Unfortunately, this approach cannot be used in the presence of optimisations like let-floating. The previous version of the Elerea library supports higher-order streams by maintaining the original call graph of the program and adding a layer of interpretation. It differs from the system discussed in this paper in only updating streams that are sampled during a superstep, therefore its flattening combinator (*sampler*) breaks compositionality and does not obey the laws of *join*. In short, the present work inherited the concept of *stream generator* from the old Elerea, but the evaluation mechanism is completely reworked.

Looking at the well-known stream-based languages, Lucid Synchrone has recently been extended with some higher-order capabilities [1]. Its *every* combinator is very similar to *generator*, but it is guarded by a boolean stream that dictates when to instantiate the stateful stream function, and stores it in a memory element. In essence, it creates a piecewise function. The question of free variables in stateful definitions is solved by simply disallowing them due to the lack of clear semantics according to the authors. This restriction results in more tractable time and space behaviour, and it can be implemented without a global update pool. In particular, it does not suffer from the problem of updating lost streams, because references to generated stream functions cannot be passed around freely, so we can maintain an explicit update tree that tells us precisely which streams are still alive.

An alternative way to look at streams is modelling them as comonads [13]. However, comonads need extra machinery to be able to express applicative combinators, which come for free given the monadic interface. Also, the authors say nothing about higher-order streams, and they provide no guidance to arrive at an efficient implementation.

Finally, it is worth noting that there are several other Haskell libraries that focus on streams, but none of them support higher-order streams to the degree presented in this paper. The Stream library contains the 'leaky' *Monad* instance for cons streams that corresponds to the discussion in Section 2.

9 Future Work

Designing and structuring systems with higher-order streams is mostly uncharted land with vast areas to explore. The next step is to create a non-trivial interactive application using the library, which will give us a clearer picture of the strengths and weaknesses of this approach. Also, equational reasoning might be used to optimise stream networks at compile time, which is another interesting line of research to pursue.

References

1. Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-Order Synchronous Data-Flow Language. In *Proceedings of the 4th ACM International Conference on Embedded Software*, Pisa, Italy, 2004, pages 230–239, ACM, New York, NY, USA.
2. Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006, pages 294–308.
3. Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18. ACM Press, 2003, Uppsala, Sweden.
4. Conal Elliott. Denotational design with type class morphisms (extended version) 2009. `http://conal.net/papers/type-class-morphisms/`
5. Conal Elliott. Push-pull functional reactive programming In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Edinburgh, Scotland, 2009, pages 25–36, ACM, New York, NY, USA, 2009.
6. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, June 1997, pages 263–273.
7. Levent Erkök. Value Recursion in Monadic Computations. PhD Dissertation, Oregon Graduate Institute School of Science Engineering, OHSU, October 2002.
8. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud. The Synchronous Data-Flow Programming Language LUSTRE. In *Proceedings of the IEEE*, Vol. 79, No. 9, September 1991, pages 1305–1320.
9. Hai Liu, Eric Cheng, and Paul Hudak. Causal Commutative Arrows and Their Optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, 2009, Edinburgh, Scotland, pages 35–46, ACM, New York, NY, USA, 2009.
10. Conor Mcbride. The Strathclyde Haskell Enhancement. 2009. `http://personal.cis.strath.ac.uk/~conor/pub/she/`
11. Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), pages 1–13, 2008.
12. Gergely Patai. Eventless Reactivity from Scratch. In *Draft Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages*, 2009, South Orange, NJ, USA, pages 126–140, Marco T. Morazán (ed), Seton Hall University, 2009.
13. Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. In *Central European Functional Programming School*, 2006, pages 135–167, Lecture Notes in Computer Science, Springer, Berlin.