Building a Program from Streams

Tim Williams | October 2018

Sponsored by

Digital Asset®

What is a stream?

A potentially infinite sequence of data elements, processed incrementally rather than as a whole.

- An abstraction that offers an alternative to mutable state.
- An abstraction that captures a programs interactions with the outside world.
- Program with pure functions and (immutable) values.
- Components can be reasoned about in isolation and composed safely.

A simple backup program

```haskell
backup :: FilePath -> FilePath -> IO ()
backup src dest = do
   files <- Dir.listDirectory src
   forM_ files $ \file -> do
       let src'  = src  </> file
           dest' = dest </> file
       isDir <- Dir.doesDirectoryExist src'
       if isDir
         then backup src' dest'
         else do
           Dir.createDirectoryIfMissing True dest
           Dir.copyFile src' dest'
```

Can we make it modular?

- We want to break-up the previous monolithic program into composable (and reusable) pieces.
- The resultant code should be easier to reason about, modify and extend.

Enumerating directories for files

- What is the biggest issue with the following function?

```haskell
enumDir :: FilePath -> IO [FilePath]
enumDir root = do
    files <- Dir.listDirectory root
    flip foldMap files $ \file -> do
        let path = root </> file
        isDir <- Dir.doesDirectoryExist path
        if isDir
            then enumDir path
            else return [path]
```

- It has unbounded memory use! Monadic IO is strict, thus sequencing an action of, e.g. `IO [a]`, will fully evaluate the entire output list in memory.
- We want to be able to write efficient bounded-space effectful programs from a composition of smaller programs.

Parameterise with a callback?

- A simple and common solution seen in mainstream imperative languages;
- but programs soon become difficult to reason about at scale.

```
enumDir :: FilePath -> (FilePath -> IO ()) -> IO ()

backup :: FilePath -> FilePath -> IO ()
backup src dest =
    enumDir src $ \src' -> do
        let dest' = dest </> relativise src src'
        copyFile src' dest'
```

Lazy evaluation?

- Lazy evaluation does allow many (pure) pipeline compositions to run efficiently one-element at-a-time, e.g. `map f . map g` has similar efficiency to `map (f . g)`.

BUT

- It has unpredictable space use,
  if `f :: [a] -> [b]` and `g :: [b] -> [c]` is `g . f` space efficient?
- It does not work well when made to mix with effects.

The Lazy IO abomination

- The following has historically been used as a way to add lazy evaluation to computations involving IO:

```haskell
-- | unsafeInterleaveIO allows an IO computation to be deferred lazily.
-- When passed a value of type IO a, the IO will only be performed when
-- the value of the a is demanded.
unsafeInterleaveIO :: IO a -> IO a
```

However, such Lazy IO is highly problematic.

- Evaluating pure functions shouldn't trigger IO!
- It is no longer clear where exceptions will be thrown or when file handles will be released!

```
main = do
    handle   <- openFile "foo.txt" ReadMode
    contents <- hGetContents handle
    hClose handle
    putStr contents -- PRINTS NOTHING!
```

Effectful Streaming

ListT done right

- Can we add streaming to Monadic IO in a safer and more principled fashion?
- Let’s start by generalising a linked-list to perform arbitrary monadic actions:

```
-- List elements interleaved with effect m.
newtype ListT m a = ListT { runListT :: m (Step m a) }
  deriving Functor

data Step m a
  = Cons (a, ListT m a)
  | Nil
  deriving Functor
```

- We can define append and concat in a analogous fashion to vanilla Lists:

```haskell
instance Monad m => Monoid (ListT m a) where
    mempty = ListT $ return Nil
    mappend (ListT m) s' = ListT $ m >>= \case
        Cons (a, s) -> return $ Cons (a, s `mappend` s')
        Nil         -> runListT s'

concat :: Monad m => ListT m (ListT m a) -> ListT m a
concat (ListT m) =
    ListT $ m >>= \case
        Cons (s, ss) -> runListT $ s `mappend` concat ss
        Nil          -> return Nil
```

- A monad instance lets us sequence actions using `do` notation:

```
instance Monad m => Monad (ListT m) where
    return x = ListT $ return $ Cons (x, mempty)
    -- (>>=) :: ListT m a -> (a -> ListT m b) -> ListT m b
    s >>= f = concat $ fmap f s
```

- `MonadTrans` and `MonadIO` instances let us lift underlying and IO monads respectively:

```
instance MonadTrans ListT where
   lift m = ListT $ m >>= \x -> return (Cons (x, mempty))

instance MonadIO m => MonadIO (ListT m) where
   liftIO m = lift (liftIO m)
```

- `return` is used to yield control and deliver a result.
- `mapM_` can be used to evaluate the stream computation.

```
mapM_ :: Monad m => (a -> m ()) -> ListT m a -> m ()
mapM_ f (ListT m) = m >>= \case
    Cons (a, s) -> f a >> mapM_ f s
    Nil         -> return ()
```

- Define `Stream' a` as an incremental on-demand computation built upon `IO`:

```
type Stream' a = ListT IO a
```

- `Stream' a` is similar in expressiveness to the `Iterable<A>` in Java or `IEnumerable<A>` in C#/F#.

Example

```
λ> return 1 <> return 2 <> return 3 :: ListT Identity Int
ListT (Identity (Cons (1,ListT (Identity (Cons (2,
  ListT (Identity (Cons (3,ListT (Identity Nil)))))))))))
```

- We can now write the following pipeline composition:

```
backup :: FilePath -> FilePath -> IO ()
backup src dest
    = copyFiles src dest
    . fmap (relativise src)
    $ enumDir src

copyFiles  :: FilePath -> FilePath -> Stream' FilePath -> IO ()
enumDir    :: FilePath -> Stream' FilePath
```

```haskell
copyFiles :: FilePath -> FilePath -> Stream' FilePath -> IO ()
copyFiles src dest =
    Stream.mapM_ $ \file -> do
        Dir.createDirectoryIfMissing True dest
        Dir.copyFile (src </> file) (dest </> file)

enumDir :: FilePath -> Stream' FilePath
enumDir dir = do
    files <- liftIO $ Dir.listDirectory dir
    flip foldMap files $ \file -> do
        let absFile = dir </> file
        exists <- liftIO $ Dir.doesDirectoryExist absFile
        if exists
            then enumDir absFile
            else return absFile
```

Problems

- No final return value, which makes it impossible to implement streaming versions of many common list operations, e.g. `splitAt`.
- We may want to parameterise the hard-coded functor `(a,)` in order to correctly implement a Stream-of-Streams (e.g. for `chunksOf`) and other additional features.

A better Stream type

- `Stream f m r` is a succession of steps, each with a structure determined by `f`, arising from actions in the monad `m`, and returning a value of type `r`.

```
newtype Stream f m r = Stream { runStream :: m (Step f m r) }
  deriving Functor

data Step f m r
   = Wrap (f (Stream f m r))
   | Return r
  deriving Functor
```

- Note that `Stream f m r` is isomorphic to `FreeT f m r`, the *free monad transformer*. This abstraction is not adhoc!

- The "streamed functor" `Of a` is just the left-strict pair:

```
data Of a r = !a :> r
```

- A yield primitive is used to suspend control and deliver a result:

```
yield :: Monad m => a -> Stream (Of a) m ()
yield a = Stream . return $ Wrap (a :> return ())
```

- Note the bind `(>>=)` is concat, rather than concatMap. The stream `s >>= \r -> s'` is the stream of values produced by `s`, followed by the stream of values produced by `s'`.

```haskell
instance (Functor f, Monad m) => Monad (Stream f m) where
  return  = Stream . return . Return
  s >>= f = Stream $ runStream s >>= \case
      Wrap fs'  -> return . Wrap $ fmap (>>=f) fs'
      Return x  -> runStream $ f x

instance MonadTrans (Stream a) where
  lift = Stream . liftM Return
```

- `mapM_` is similar to previous implementations and can be used to evaluate the stream:

```
mapM_ :: Monad m => (a -> m ()) -> Stream (Of a) m r -> m r
mapM_ f s = runStream s >>= \case
    Wrap (a :> s') -> f a >> mapM_ f s'
    Return x       -> return x
```

Example

```
λ>  S.yield 1 >> S.yield 2 >> S.yield 3 :: Stream (Of Int) Identity ()
Stream (Identity (Wrap (1 :> Stream (Identity (Wrap (2 :>
  Stream (Identity (Wrap (3 :> Stream (Identity (Return ()))))))))))
```

Haskell `streaming` package

The *streaming* Hackage package implements essentially the same `Stream` type in a manner that is efficient for GHC. It includes a comprehensive Prelude of list-like operations.

```
import Streaming
import qualified Streaming.Prelude as S

data Stream f m r
    = Return r
    | Step !(f (Stream f m r))
    | Effect (m (Stream f m r))

yield :: Monad m => a -> Stream (Of a) m ()
yield a = Step (a :> Return ())
```

The return type and parameterised functor allow streaming variants of the common list functions `splitAt` and `chunksOf` respectively:

```
splitAt
    :: (Monad m, Functor f)
    => Int -> Stream (Of a) m r -> Stream (Of a) m (Stream (Of a) m r)

chunksOf
    :: (Monad m, Functor f)
    => Int -> Stream f m r -> Stream (Stream f m) m r
```

Atavachron

Atavachron is an example of a large and full-featured backup program developed using the *streaming* package[1].

https://github.com/willtim/Atavachron

The definitions for the main top-level pipelines can be found here.

[1]Note that Atavachron is still under development and not yet ready for widespread use.

Tips

- Streaming is a good fit for the large-scale architecture of an application, but not for fine-grained performance critical sections, i.e. `Stream Word8` is not good practice.
- Parallelism often means sacrificing ordering, either the ordering of the elements or ordering of the effects. Element ordering can be recovered at the expense of additional space and time.
- *Synchronous* streams may make more sense with some complex pipeline requirements. Synchronous streams allow for parallel composition `f *** g` and Arrow combinators for building "circuits".
- Automatic releasing of file handles and other finite resources can be achieved by layering the ResourceT and/or Managed monad transformers. Prompt finalisation remains an issue.

Advanced libraries

- The state-of-the-art in Haskell streaming is currently embodied by Iteratee and its variants, which offer:
  - two way communication
  - prompt finalisation
  - "backpressure"
  - buffering
  - concurrency
- Pipes and Conduits are popular variations of the idea, they provide abstract APIs which help ensure streams are used correctly (i.e. enforcing *linearity*, no discarding or duplicating), but are somewhat complex to use.
- In the future, Linear types may offer safe use with less complex and abstract interfaces.

Summary

- Streaming is a fundamental abstraction and key to building many real-world applications.
- There is no one-size fits all streaming library. They are all a trade-off between ease of use and features.
- Understanding ListT and Stream (a.k.a. FreeT) will help to understand all approaches.
- The `streaming` Hackage package strikes a good balance between simplicity and practicality.

The slides for this talk will be available at:
http://www.timphilipwilliams.com/slides/streaming.pdf