More on streaming

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-02-14

INPUT | OUTPUT

Goals

- Safe resource management
- More streaming combinators
- Folding streams and lists

Resource management

Reading files lazily

```
readFile :: FilePath -> IO String
```

In contrast to what was previously said, this function provides the resulting `String` incrementally.

How?

`unsafeInterleaveIO`

```
unsafeInterleaveIO :: IO a -> IO a
```

Rather than performing the given IO action, the function returns immediately. The IO action is performed at some point between the call to `unsafeInterleaveIO` and the moment the result is demanded (and potentially never, if the result is not demanded at all).

Copying a file

Using `readFile` and

```
writeFile :: FilePath -> String -> IO ()
```

we can provide a space-efficient way to copy a file:

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile source target = do
  contents <- readFile source
  writeFile target contents
```

Only `writeFile` actually triggers the reading, and even large files can be copied without fully loading them into memory.

When is the handle cleaned up?

- As long as the contents of the file have not been completely evaluated, the file stays open.
- This can potentially be a long time.
- It’s quite fragile if the use of a potentially scarce resource (such as a file handle, or a socket) is tied to the *evaluation* of a pure value.

Problematic examples

Clearly wrong:

```
catFiles :: [FilePath] -> IO String
catFiles =
  fmap concat . mapM readFile
```

(This is the standard `mapM`.)

Problematic examples

Clearly wrong:

```
catFiles :: [FilePath] -> IO String
catFiles =
  fmap concat . mapM readFile
```

(This is the standard `mapM`.)

```
GHCi> catFiles (replicate 2000 "Makefile")
*** Exception: Makefile: openFile:
resource exhausted (Too many open files)
```

Would streaming help?

```haskell
catFiles :: [FilePath] -> Stream String IO ()
catFiles =
  mapM readFile . each
```

Would streaming help?

```
catFiles :: [FilePath] -> Stream String IO ()
catFiles =
  mapM readFile . each
```

Whether this works depends on the way it's used:

```
GHCi> test = catFiles (replicate 2000 "Makefile")
GHCi> stdoutLn test
...(ok)
GHCi> concat <$> toList test
*** Exception: Makefile: openFile:
resource exhausted (Too many open files)
```

Recap: our own streams

Streams

```haskell
data StreamF b m r =
     Lift (m r)
   | Yield b r
  deriving (Functor)
```

```haskell
data Free f a =
     Return a
   | Wrap (f (Free f a))

type Stream b m = Free (StreamF b m)
```

Streams interface

```haskell
yield :: b -> Stream b m ()
lift :: Functor m => m a -> Stream b m a
each :: Monad m => [b] -> Stream b m ()
map :: Monad m =>
  (b -> c) -> Stream b m a -> Stream c m a
for :: Monad m =>
  Stream b m a -> (b -> Stream c m r)
     -> Stream c m a
take :: Monad m =>
  Int -> Stream b m a -> Stream b m ()
toList :: Monad m => Stream b m () -> m [b]
effects :: Monad m => Stream b m a -> m a
stdoutLn :: Stream String IO a -> IO a
```

Streaming in context

Our `Stream` type was / is extremely similar to the one offered by the `streaming` package, and can also be seen as a simplified form of what the `pipes` and `conduit` packages offer.

Back to resource handling

Would streaming help? (again)

```
catFiles :: [FilePath] -> Stream String IO ()
catFiles =
  mapM readFile . each
```

Whether this works depends on the way it's used:

```
GHCi> test = catFiles (replicate 2000 "Makefile")
GHCi> stdoutLn test
...(ok)
GHCi> concat <$> toList test
*** Exception: Makefile: openFile:
resource exhausted (Too many open files)
```

Delimiting the scope of a handle

Recall:

```haskell
withFile ::
  FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

Has nice properties:

- the handle is automatically closed when the continuation ends;
- the handle is also automatically closed if an exception occurs.

Does `withFile` help us?

Where do we apply it?

- We cannot apply `withFile` on the outside. We'd still have to open *all the files*, causing the same problem. – In other situations, we might even only discover what files we have to open during the pipeline.
- Trying to apply `withFile` on the inside yields a different problem …

Does `withFile` help us? (contd.)

```
brokenReadFile :: FilePath -> IO String
brokenReadFile f =
  withFile f ReadMode hGetContents
```

where

```
hGetContents :: Handle -> IO String
```

is a version of `readFile` on handles (also using `unsafeInterleaveIO`).

```
catFiles :: [FilePath] -> Stream String IO ()
catFiles =
  mapM brokenReadFile . each
```

Testing this version

```
GHCi> test = catFiles (replicate 2000 "Makefile")
GHCi> stdoutLn test
*** Exception: hGetContents: illegal operation
(delayed read on closed handle)
GHCi> concat <$> toList test
*** Exception: hGetContents: illegal operation
(delayed read on closed handle)
```

We demand the `String` outside the scope of `withFile`, so the handle is already closed.

What else can we do?

- We could force the whole contents of the file within the scope of `withFile`, but that would *necessarily* prevent true streaming for large files, regardless of how we use the results.
- We could try to avoid `unsafeInterleaveIO` completely, and to combine reading the file step by step with the streaming mechanism we now already have.

Reading a file line by line

```
fromHandle :: Handle -> Stream String IO ()
fromHandle h = do
  eof <- lift (hIsEOF h)
  if eof
    then return ()
    else do
      l <- lift (hGetLine h)
      yield l
      fromHandle h
```

The function `hGetLine` eagerly reads an entire line, so we never tie evaluation of a pure value to an effect.

Reading a file line by line

```
fromHandle :: Handle -> Stream String IO ()
fromHandle h = do
  eof <- lift (hIsEOF h)
  if eof
    then return ()
    else do
      l <- lift (hGetLine h)
      yield l
      fromHandle h
```

The function `hGetLine` eagerly reads an entire line, so we never tie evaluation of a pure value to an effect.

However, once again, the question arises when we can close the handle.

A naive attempt

```haskell
readFile :: FilePath -> Stream String IO ()
readFile f = do
  h <- lift (openFile f ReadMode)
  fromHandle h
  lift (hClose h)
```

A naive attempt

```
readFile :: FilePath -> Stream String IO ()
readFile f = do
  h <- lift (openFile f ReadMode)
  fromHandle h
  lift (hClose h)
```

```
catFiles :: [FilePath] -> Stream String IO ()
catFiles fs =
  for (each fs) readFile
```

A naive attempt

```
readFile :: FilePath -> Stream String IO ()
readFile f = do
  h <- lift (openFile f ReadMode)
  fromHandle h
  lift (hClose h)
```

```
catFiles :: [FilePath] -> Stream String IO ()
catFiles fs =
  for (each fs) readFile
```

This actually works in our example, but is the handle guaranteed to be closed?

A naive attempt (contd.)

```
GHCi> stdoutLn (take 1 (readFile "Makefile"))
```

This seems to work, but it never executes the call to `hClose`:

- Via `take`, we only inspect a prefix of the stream, so we never reach the part where the cleanup code would be executed.
- A similar situation arises if an exception occurs. The handle would never be (explicitly) closed.
- In order to have safe and predictable resource use, we need a different approach.

Resource managemenet

A proper solution is to combine the statically scoped and dynamic approaches:

- Have a static scope that guarantees that after that part of the program is done, resources are properly cleaned up (also in the case of an exception).
- Within that scope, allow to dynamically allocate new resources.
- Within that scope, allow to explicitly free resources we know we are done with.

The `resourcet` package

One such solution is offered by the `resourcet` package:

```haskell
data ResourceT m a  -- abstract
instance Monad m => Monad (ResourceT m)
instance MonadTrans ResourceT
instance (MonadIO m, ...) =>
  MonadResource (ResourceT m)

allocate :: MonadResource m =>
  IO a -> (a -> IO ()) -> m (ReleaseKey, a)
register :: MonadResource m =>
  IO () -> m ReleaseKey
release :: MonadIO m => ReleaseKey -> m ()

runResourceT :: ... => ResourceT m a -> m a
```

The `resourcet` interface explained

```
allocate :: MonadResource m =>
  IO a -> (a -> IO ()) -> m (ReleaseKey, a)
```

Can be used to safely allocate a resource and register its cleanup code:

- First argument allocates.
- Second argument is the cleanup.
- Cleanup is guaranteed to be run exactly once, either by `runResourceT` (when the block ends, or an exception occurs), or if called explicitly via `release`.

The `resourcet` interface explained

```haskell
allocate :: MonadResource m =>
  IO a -> (a -> IO ()) -> m (ReleaseKey, a)
```

Can be used to safely allocate a resource and register its cleanup code:

- First argument allocates.
- Second argument is the cleanup.
- Cleanup is guaranteed to be run exactly once, either by `runResourceT` (when the block ends, or an exception occurs), or if called explicitly via `release`.

The `allocate` function is similar to `bracket`:

```haskell
bracket ::
  IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

The `resourcet` interface explained (contd.)

```haskell
register :: MonadResource m =>
  IO () -> m ReleaseKey
```

Registers a cleanup action without an explicit allocation. Otherwise like `allocate`.

The `resourcet` interface explained (contd.)

```haskell
register :: MonadResource m =>
  IO () -> m ReleaseKey
```

Registers a cleanup action without an explicit allocation. Otherwise like `allocate`.

```haskell
release :: MonadIO m => ReleaseKey -> m ()
```

Triggers the cleanup action for the given key immediately, if we know the resource will not be needed anymore. Cleanup actions are guaranteed to be run only once.

The `resourcet` interface explained (contd.)

```
runResourceT :: ... => ResourceT m a -> m a
```

The given computation is run in the underlying monad (typically `IO`), and at the end, all cleanup actions that have been registered and not yet been run are executed.

Note that `ResourceT` is not fully thread-safe. Special care has to be taken if the computations to be run inside of `ResourceT` use concurrency.

A brief look into `ResourceT`

```
newtype ResourceT m a =
  ResourceT
    {unResourceT :: IORef ReleaseMap -> m a}
```

Essentially a `ReaderT` passing around a mutable map that associates release keys with cleanup actions.

Using `ResourceT`

```
bracketStream :: (MonadResource m) =>
  IO c -> (c -> IO ()) -> (c -> Stream b m a)
     -> Stream b m a
bracketStream alloc free body = do
  (key, resource) <- lift (allocate alloc free)
  result <- body resource
  lift (release key)
  return result
```

- Cleanup code will definitely be run by `runResourceT`.
- Cleanup code will also be run if we reach the end of the stream.

Reading a file line by line, as a stream

This almost works:

```
readFile :: MonadResource m =>
  FilePath -> Stream String m ()
readFile f =
  bracketStream
    (openFile f ReadMode)
    hClose
    fromHandle
```

The only problem is that `fromHandle` produces a `Stream String IO ()`, but we are now operating in `ResourceT IO`.

Generalising `fromHandle`

```
fromHandle :: MonadIO m =>
  Handle -> Stream String m ()
fromHandle h = do
  eof <- lift (liftIO (hIsEOF h))
  if eof
    then return ()
    else do
      l <- lift (liftIO (hGetLine h))
      yield l
      fromHandle h
```

Now `readFile` typechecks.

(Other IO-specific streams such as `stdoutLn` can be similarly generalised.)

Concatenating files, once again

```haskell
catFiles :: MonadResource m =>
  [FilePath] -> Stream String m ()
catFiles fs = for (each fs) readFile
```

Concatenating files, once again

```
catFiles :: MonadResource m =>
  [FilePath] -> Stream String m ()
catFiles fs = for (each fs) readFile
```

```
GHCi> test = catFiles (replicate 2000 "Makefile")
GHCi> runResourceT (stdoutLn test)
...(ok)
GHCi> concat <$> runResourceT (toList test)
...(ok)
```

Use of `resourcet`

- The way we use the `resourcet` exactly follows the way it is also used in the `streaming` package.
- Similarly, the `conduit` ecosystem makes use of `resourcet` extensively for resource management.
- In the `pipes` world, a very similar package called `pipes-safe` exists.

Some problems remain

Note that not all problems are caught by the use of `resourcet`:

```
problem :: MonadResource m =>
  [FilePath] -> Stream String m ()
problem fs = for (each fs) (take 1 . readFile)
```

Some problems remain

Note that not all problems are caught by the use of `resourcet`:

```
problem :: MonadResource m =>
  [FilePath] -> Stream String m ()
problem fs = for (each fs) (take 1 . readFile)
```

```
GHCi> test = problem (replicate 2000 "Makefile")
GHCi> runResourceT (stdoutLn test)
...(output starts)
*** Exception: Makefile: openFile:
resource exhausted (Too many open files)
```

Some problems remain

Note that not all problems are caught by the use of `resourcet`:

```
problem :: MonadResource m =>
  [FilePath] -> Stream String m ()
problem fs = for (each fs) (take 1 . readFile)
```

- We are never "done" with any of the files, so the handles are not closed while performing the computation.
- When `runResourceT` ultimately does it, it is too late.
- There is no simple fix for this with current libraries (`pipes` and `conduit` suffer from this as well). One should be careful when opening an unbounded number of files whether handles can be closed soon enough.

Excursion: Changing the stream monad

Generalising `fromHandle` (again)

```
fromHandleM ::
  Handle -> Stream String IO ()
fromHandleM h = do
  eof <- lift (hIsEOF h)
  if eof
    then return ()
    else do
      l <- lift (hGetLine h)
      yield l
      fromHandle h
```

```
fromHandleP :: MonadIO m =>
  Handle -> Stream String m ()
fromHandleP h = do
  eof <- lift (liftIO (hIsEOF h))
  if eof
    then return ()
    else do
      l <- lift (liftIO (hGetLine h))
      yield l
      fromHandle h
```

Generalising `fromHandle` (again)

```
fromHandleM ::
  Handle -> Stream String IO ()
fromHandleM h = do
  eof <- lift (hIsEOF h)
  if eof
    then return ()
    else do
      l <- lift (hGetLine h)
      yield l
      fromHandle h
```

```
fromHandleP :: MonadIO m =>
  Handle -> Stream String m ()
fromHandleP h = do
  eof <- lift (liftIO (hIsEOF h))
  if eof
    then return ()
    else do
      l <- lift (liftIO (hGetLine h))
      yield l
      fromHandle h
```

We just apply `liftIO` to all lifted effects – could we do this from the outside?

Generalising `fromHandle` (again)

```
fromHandleM ::
  Handle -> Stream String IO (
fromHandleM h = do
  eof <- lift (hIsEOF h)
  if eof
    then return ()
    else do
      l <- lift (hGetLine h)
      yield l
      fromHandle h
```

```
fromHandleP :: MonadIO m =>
  Handle -> Stream String m ()
fromHandleP h = do
  eof <- lift (liftIO (hIsEOF h))
  if eof
    then return ()
    else do
      l <- lift (liftIO (hGetLine h))
      yield l
      fromHandle h
```

We just apply `liftIO` to all lifted effects – could we do this from the outside?

I.e., we aim to define `hoist` such that

```
hoist liftIO . fromHandleM = fromHandleP
```

The type of `hoist`

```
hoist ::
      ...
  -> Stream b m1 a -> Stream b m2 a
```

We want to change the monad.

The type of `hoist`

```
hoist ::
     (m1 ... -> m2 ...)
  -> Stream b m1 a -> Stream b m2 a
```

So we need a function taking one to the other.

But we need to apply an argument to `m1` and `m2`.

The type of `hoist`

```
hoist ::
      (m1 c -> m2 c)
   -> Stream b m1 a -> Stream b m2 a
```

Simply picking a type variable is not good enough!

Looking at `fromHandle` again

Consider:

```
fromHandleP :: MonadIO m =>
  Handle -> Stream String m ()
fromHandleP h = do
  eof <- lift (liftIO (hIsEOF h))
  if eof
    then return ()
    else do
      l <- lift (liftIO (hGetLine h))
      yield l
      fromHandle h
```

The function `liftIO` is used at two different types:

```
liftIO :: MonadIO m => IO Bool    -> m Bool
liftIO :: MonadIO m => IO String -> m String
```

Requiring an argument to be polymorphic

Fortunately, `liftIO` is polymorphic:

```
liftIO :: MonadIO m => IO a -> m a
```

Can we require a function *argument* to be polymorphic?

Requiring an argument to be polymorphic

Fortunately, `liftIO` is polymorphic:

```
liftIO :: MonadIO m => IO a -> m a
```

Can we require a function *argument* to be polymorphic?

```
hoist ::
     (forall c . m1 c -> m2 c)
  -> Stream b m1 a -> Stream b m2 a
```

This is called a rank-2 polymorphic type and requires the `RankNTypes` language extension.

Implementing `hoist`

```
hoist :: Functor m1 =>
     (forall c . m1 c -> m2 c)
  -> Stream b m1 a -> Stream b m2 a
hoist f (Return x)          = Return x
hoist f (Wrap (Lift m))     =
  Wrap (Lift (f (fmap (hoist f) m)))
hoist f (Wrap (Yield b k)) =
  Wrap (Yield b (hoist f k))
```

Implementing `hoist`

```haskell
hoist :: Functor m1 =>
     (forall c . m1 c -> m2 c)
  -> Stream b m1 a -> Stream b m2 a
hoist f (Return x)          = Return x
hoist f (Wrap (Lift m))     =
  Wrap (Lift (f (fmap (hoist f) m)))
hoist f (Wrap (Yield b k)) =
  Wrap (Yield b (hoist f k))
```

```haskell
GHCi> :t hoist liftIO . fromHandleM
hoist liftIO . fromHandleM
  :: MonadIO m2 => Handle -> Stream String m2 ()
```

Rank-1 vs. rank-2 polymorphism

Normal (rank-1) polymorphism is quantifying a type on the outside – with the `RankNTypes` language extension (and others), we can make that quantification visible, e.g.:

```
fmap ::
  forall f a b . Functor f => (a -> b) -> f a -> f b
```

- The caller of the function is flexible and can choose at what instantiations of the quantified variables to call the function.
- The callee (i.e., the function itself) is constrained and cannot make any assumptions about the quantified type variables in the implementation.

Rank-1 vs. rank-2 polymorphism (contd.)

Rank-2 polymorphism is quantifying a function argument:

```
hoist :: Functor m1 =>
     (forall c . m1 c -> m2 c)
   -> Stream b m1 a -> Stream b m2 a
```

- The caller of the function is constrained and has to provide an argument that is sufficiently polymorphic.
- The callee (i.e., the function itself) is flexible and can use the function argument at several different types in the implementation.

Rank-1 vs. rank-2 polymorphism (contd.)

Rank-2 polymorphism is quantifying a function argument:

```
hoist :: Functor m1 =>
      (forall c . m1 c -> m2 c)
   -> Stream b m1 a -> Stream b m2 a
```

- The caller of the function is constrained and has to provide an argument that is sufficiently polymorphic.
- The callee (i.e., the function itself) is flexible and can use the function argument at several different types in the implementation.
- GHC cannot infer rank-2 (or higher) polymorphic types (similar to polymorphic recursion), and a type signature is required in such a case.

Higher-rank polymorphism

- A rank-3 type would be one where a function argument of a function argument is quantified – and so on.
- In principle, GHC allows arbitrary rank polymorphic types with the language extension, but everything higher than rank 2 is extremely rare.

Connection to category theory

```
hoist :: Functor m1 =>
     (forall c . m1 c -> m2 c)
   -> Stream b m1 a -> Stream b m2 a
```

can be rewritten to

```
hoist :: Functor m1 =>
     (forall a .          m1 a ->          m2 a)
   -> (forall a . Stream b m1 a -> Stream b m2 a)
```

and is like a `map`, but operating on *natural transformations* rather than standard Haskell functions.

Connection to category theory

```
hoist :: Functor m1 =>
     (forall c . m1 c -> m2 c)
   -> Stream b m1 a -> Stream b m2 a
```

can be rewritten to

```
hoist :: Functor m1 =>
     (forall a .          m1 a ->          m2 a)
   -> (forall a . Stream b m1 a -> Stream b m2 a)
```

and is like a `map`, but operating on *natural transformations* rather than standard Haskell functions.

The function `hoist` on streams also fulfills the *functor laws* in the category of natural transformations.

Explicit folds

A classic problem

Let's define a function to compute an average over a list of numbers:

```
average :: Fractional a => [a] -> a
average xs = sum xs / fromIntegral (length xs)
```

What's the space complexity of this function?

A classic problem

Let's define a function to compute an average over a list of numbers:

```
average :: Fractional a => [a] -> a
average xs = sum xs / fromIntegral (length xs)
```

What's the space complexity of this function?

It consumes linear space in the length of the list, because `xs` is needed for both the computation of the sum and the computation of the length.

Testing this

```haskell
main :: IO ()
main = do
  [size] <- getArgs
  print $ average [1 .. read size]
```

Testing this

```haskell
main :: IO ()
main = do
  [size] <- getArgs
  print $ average [1 .. read size]
```

```
$ ./Average1 1000000 +RTS -t
500000.5
<<ghc: 168109984 bytes, 161 GCs, 9806353/34409616
avg/max bytes residency (8 samples), 79M in use, 0.000
INIT (0.000 elapsed), 0.047 MUT (0.047 elapsed), 0.078
GC (0.078 elapsed) :ghc>>
```

34 megabytes maximum residency, and more than half the time spent doing GC!

An observation

As we know, both `sum` and `length` are best defined as (strict) left folds:

```
sum    :: (Num a) => [a] -> a
length :: [a] -> Int

sum    = foldl' (+)                0
length = foldl' (const . (+ 1)) 0
```

An observation

As we know, both `sum` and `length` are best defined as (strict) left folds:

```
sum    :: (Num a) => [a] -> a
length :: [a] -> Int

sum    = foldl' (+)              0
length = foldl' (const . (+ 1)) 0
```

Both left folds traverse the list with an accumulator – can we combine the two?

Combining `sum` and `length`

```haskell
sumAndLength :: (Num a) => [a] -> (a, Int)
sumAndLength = foldl' op (0, 0)
  where
    op (!rs, !rl) x =
      ((+) rs x, (const . (+ 1)) rl x)
```

Combining `sum` and `length`

```haskell
sumAndLength :: (Num a) => [a] -> (a, Int)
sumAndLength = foldl' op (0, 0)
  where
    op (!rs, !rl) x =
      ((+) rs x, (const . (+ 1)) rl x)
```

```
GHCi> sumAndLength [1 .. 10]
(55, 10)
```

An improved `average`

```haskell
average :: Fractional a => [a] -> a
average xs = case sumAndLength xs of
  (rs, rl) -> rs / fromIntegral rl
```

Testing again

```haskell
main :: IO ()
main = do
  [size] <- getArgs
  print $ average [1 .. read size]
```

Testing again

```haskell
main :: IO ()
main = do
  [size] <- getArgs
  print $ average [1 .. read size]
```

```
$ ./Average2 1000000 +RTS -t
500000.5
<<ghc: 168109968 bytes, 161 GCs, 36536/44504 avg/max
bytes residency (2 samples), 2M in use, 0.000 INIT
(0.000 elapsed), 0.068 MUT (0.068 elapsed), 0.001
GC (0.001 elapsed) :ghc>>
```

Only 44 kilobytes maximum residency, and hardly any time spent doing GC!

Combining `sum` and `length` (again)

```haskell
sum    :: (Num a) => [a] -> a
length :: [a] -> Int

sum    = foldl' (+)               0
length = foldl' (const . (+ 1)) 0
```

```haskell
sumAndLength :: (Num a) => [a] -> (a, Int)
sumAndLength = foldl' op (0, 0)
  where
    op (!rs, !rl) x =
      ((+) rs x, (const . (+ 1)) rl x)
```

This construction only depends on both functions being left folds, not on the specific folds being used.

Combining left folds

If we have

```
f1 :: Foldable t => t A -> B1
f2 :: Foldable t => t A -> B2

f1 = foldl' op1 acc1
f2 = foldl' op2 acc2
```

then we also have

```
f1And2 :: Foldable t => t A -> (B1, B2)
f1And2 = foldl' op (acc1, acc2)
  where
    op (!r1, !r2) x =
      (op1 r1 x, op2 r2 x)
```

Can we do this programmatically?

The problem is that if we are given two functions

```
f1 = foldl' op1 acc1
f2 = foldl' op2 acc2
```

then we have no way to reconstruct `op1`, `op2`, `acc1` and `acc2` from `f1` and `f2`.

Can we do this programmatically?

The problem is that if we are given two functions

```
f1 = foldl' op1 acc1
f2 = foldl' op2 acc2
```

then we have no way to reconstruct `op1`, `op2`, `acc1` and `acc2` from `f1` and `f2`.

Similarly for

```
sum    = foldl' (+)             0
length = foldl' (const . (+ 1)) 0
```

Can we do this programmatically?

The problem is that if we are given two functions

```
f1 = foldl' op1 acc1
f2 = foldl' op2 acc2
```

then we have no way to reconstruct `op1`, `op2`, `acc1` and `acc2` from `f1` and `f2`.

Similarly for

```
sum    = foldl' (+)               0
length = foldl' (const . (+ 1)) 0
```

But we can delay the use of `foldl'` and store the components as data!

A fold as data

```haskell
data Fold a b =
  Fold (b -> a -> b) b
```

A value of type `Fold a b` processes elements of type `a` and produces a result of type `b`.

A fold as data

```haskell
data Fold a b =
  Fold (b -> a -> b) b
```

A value of type `Fold a b` processes elements of type `a` and produces a result of type `b`.

We can still *run* a fold:

```haskell
fold :: Foldable f => Fold a b -> f a -> b
fold (Fold op acc) = foldl' op acc
```

Examples

```
sum_    :: Num b => Fold b b
length_ :: Fold a Int
```

```
sum_    = Fold (+)               0
length_ = Fold (const . (+ 1)) 0
```

Combining `Fold`s

Recall:

```haskell
f1And2 :: Foldable t => t A -> (B1, B2)
f1And2 = foldl' op (acc1, acc2)
  where
    op (!r1, !r2) x =
      (op1 r1 x, op2 r2 x)
```

Combining `Fold`s

Recall:

```haskell
f1And2 :: Foldable t => t A -> (B1, B2)
f1And2 = foldl' op (acc1, acc2)
  where
    op (!r1, !r2) x =
      (op1 r1 x, op2 r2 x)
```

We can now write this as a proper program:

```haskell
combine :: Fold a b -> Fold a c -> Fold a (b, c)
combine (Fold op1 acc1) (Fold op2 acc2) =
  Fold op (acc1, acc2)
  where
    op (!r1, !r2) x =
      (op1 r1 x, op2 r2 x)
```

What about average?

We can define

```
average :: (Foldable f, Fractional a) => f a -> a
average xs =
  case fold (combine sum_ length_) xs of
    (rs, rl) -> rs / fromIntegral rl
```

but that does not feel compositional enough.

We have to leave the `Fold` abstraction too early – what if we wanted to reuse `average` in yet another fold?

A slight generalisation

```haskell
data Fold a b =
  Fold (b -> a -> b) b
```

The old `Fold` type.

A slight generalisation

```haskell
data Fold :: * -> * -> * where
  Fold :: (b -> a -> b) -> b -> Fold a b
```

Rewrite in GADT syntax.

A slight generalisation

```haskell
data Fold :: * -> * -> * where
  Fold :: (x -> a -> x) -> x -> (x -> b) -> Fold a b
```

Add an extra *extractor* function and hide (existentially quantify) the type of the accumulator.

A slight generalisation

```
data Fold :: * -> * -> * where
  Fold :: (x -> a -> x) -> x -> (x -> b) -> Fold a b
```

Add an extra *extractor* function and hide (existentially quantify) the type of the accumulator.

Why do we do all this? Because this trick (which is actually a more general technique also available in the libraries as `Coyoneda`) turns `Fold` into a `Functor`!

A `Functor` instance for folds

```haskell
instance Functor (Fold a) where
  fmap f (Fold op acc ex) = Fold op acc (f . ex)
```

We even have `Applicative`!

```haskell
instance Applicative (Fold a) where
  pure x =
    Fold (\ _ _ -> ()) () (const x)
  Fold opf accf exf <*> Fold opx accx exx =
    Fold op (accf, accx) ex
    where
      op (!r1, !r2) x =
        (opf r1 x, opx r2 x)
      ex (!r1, !r2) =
        (exf r1) (exx r2)
```

Redoing the example

```haskell
sum_    :: Num b => Fold b b
length_ :: Fold a Int
```

```haskell
sum_    = Fold (+)                0 id
length_ = Fold (const . (+ 1)) 0 id
```

Redoing the example

```haskell
sum_    :: Num b => Fold b b
length_ :: Fold a Int
```

```haskell
sum_    = Fold (+)              0 id
length_ = Fold (const . (+ 1)) 0 id
```

```haskell
average_ :: Fractional a => Fold a a
average_ =
  (/) <$> sum_ <*> (fromIntegral <$> length_)
```

Still efficient, and very concise.

We can define `Num` and `Fractional` instances for folds

```haskell
instance Num b => Num (Fold a b) where
  fromInteger = pure . fromInteger
  negate = fmap negate
  abs = fmap abs
  signum = fmap signum
  (+) = liftA2 (+)
  (*) = liftA2 (*)
  (-) = liftA2 (-)
```

We can define `Num` and `Fractional` instances for folds

```haskell
instance Num b => Num (Fold a b) where
  fromInteger = pure . fromInteger
  negate = fmap negate
  abs = fmap abs
  signum = fmap signum
  (+) = liftA2 (+)
  (*) = liftA2 (*)
  (-) = liftA2 (-)
```

```haskell
instance Fractional b => Fractional (Fold a b) where
  fromRational = pure . fromRational
  recip = fmap recip
  (/) = liftA2 (/)
```

Average yet again

```
average_ :: Fractional a => Fold a a
average_ = sum_ / (fromIntegral <$> length_)
```

It’s a matter of taste whether this is preferable over the previous version.

Doing the same for monadic folds

```
data FoldM :: (* -> *) -> * -> * -> * where
  FoldM ::
    (x -> a -> m x) -> m x -> (x -> m b)
      -> FoldM m a b
```

The same constructions work for this type – it can also be made an instance of `Functor`, `Applicative`, `Num` and `Fractional`.

The `foldl` package

All this is available in the `foldl` package by Gabriel Gonzalez (who is also the author of `pipes`).

The `Control.Foldl` module exports both the datatypes and many predefined folds.

Combining folds and streams

Folding a stream

```
fold :: Monad m =>
  (x -> b -> x) -> x -> (x -> c)
    -> Stream b m () -> m c
```

Note that the arguments correspond directly to the components of the `Fold` type.

Folding a stream

```haskell
fold :: Monad m =>
  (x -> b -> x) -> x -> (x -> c)
     -> Stream b m () -> m c
```

Note that the arguments correspond directly to the components of the `Fold` type.

```haskell
fold op acc ex = go acc
  where
    go !acc (Return ())          = return (ex acc)
    go !acc (Wrap (Lift m))      = m >>= go acc
    go !acc (Wrap (Yield b k)) = go (op acc b) k
```

Upgrading a fold

```haskell
purely ::
  (forall x . (x -> a -> x) -> x -> (x -> b) -> r)
    -> Fold a b -> r
purely f (Fold op acc ex) = f op acc ex
```

Upgrading a fold

```
purely ::
  (forall x . (x -> a -> x) -> x -> (x -> b) -> r)
     -> Fold a b -> r
purely f (Fold op acc ex) = f op acc ex
```

Another example of rank-2 polymorphism. Function `f` *must* be polymorphic in `x`, because we don't know the hidden type in the `Fold` application.

Putting the two together

```
GHCi> :t purely fold
purely fold ::
  Monad m => Fold a b -> Stream a m () -> m b
```

Putting the two together

```
GHCi> :t purely fold
purely fold ::
  Monad m => Fold a b -> Stream a m () -> m b
```

- We could also have written a version of `fold` that takes a `Fold` directly.
- Our functions mirror the library situation better: `streaming` defines `fold` (so does `pipes`), and `foldl` defines `purely`. No dependency on `foldl` is needed.
- Similarly, there exist `impurely` and `foldM` for monadic folds.

A final example

```
GHCi> purely fold average_
        (mapM (\ x -> putStr "." >> return x)
          (each [1 .. 10]))
..........5.5
```