Streaming

Haskell Performance

Andres Löh

Well-Typed

The Haskell Consultants

- Streaming in the presence of effects
- Safe resource management
- Some tools for measuring (space and time) performance

Example: Listing all files

Recursively exploring a file system

Let's try to write a Haskell program that – given an initial directory – lists all files underneath that directory (including files in subdirectories).

Well-Typed

A first attempt

```haskell
allFilesRecursively :: FilePath -> IO [FilePath]
allFilesRecursively dir = do
  xs <- getDirectoryContents dir
  ys <- forM xs $ \ x -> do
    if "." `isPrefixOf` x
      then return []  -- hidden file
      else do
        let f = dir </> x
        b <- doesDirectoryExist f
        if b
          then allFilesRecursively f
          else return [f]
  return (concat ys)
```

Well-Typed

```haskell
main :: IO ()
main = do
  [dir] <- getArgs  -- partial pattern match
  files <- allFilesRecursively dir
  mapM_ putStrLn files
```

- ▸ The program seems to work correctly on small directories.
- ▸ The program is slow and consumes a lot of memory on medium-sized directories.
- ▸ The program seems to consume lots of memory and hang for a long time on large directories.

Well-Typed

RTS info

We can obtain various run-time info on a Haskell program by passing the +RTS −s run-time system flag:

```
$ allFiles . +RTS -s
...
        165,800 bytes allocated in the heap
          3,408 bytes copied during GC
         44,504 bytes maximum residency (1 sample(s))
         25,128 bytes maximum slop
              2 MB total memory in use (0 MB lost due to fragmentation)

                                 Tot time (elapsed)  Avg pause  Max pause
  Gen  0         0 colls,     0 par    0.000s   0.000s     0.0000s    0.0000s
  Gen  1         1 colls,     0 par    0.000s   0.000s     0.0000s    0.0000s

  INIT    time    0.000s  (  0.000s elapsed)
  MUT     time    0.001s  (  0.001s elapsed)
  GC      time    0.000s  (  0.000s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time    0.001s  (  0.001s elapsed)

  %GC     time       0.0%  (0.0% elapsed)

  Alloc rate    265,577,872 bytes per MUT second

  Productivity  81.5% of total user, 82.1% of total elapsed
```

Well-Typed

RTS info (contd.)

Or in more compact form with +RTS -t:

```
$ allFiles . +RTS -t
...
<<ghc: 165800 bytes, 1 GCs, 44504/44504 avg/max bytes
residency (1 samples), 2M in use, 0.000 INIT (0.000
elapsed), 0.001 MUT (0.001 elapsed), 0.000 GC (0.000
elapsed) :ghc>>
```

Well-Typed

RTS info (contd.)

Or in more compact form with +RTS -t:

```
$ allFiles . +RTS -t
...
<<ghc: 165800 bytes, 1 GCs, 44504/44504 avg/max bytes
residency (1 samples), 2M in use, 0.000 INIT (0.000
elapsed), 0.001 MUT (0.001 elapsed), 0.000 GC (0.000
elapsed) :ghc>>
```

Important information:

- ▶ 44504 bytes max residency indicates the maximum amount of heap space used
- ▶ 0.001 MUT indicates the time in seconds spent in the **mutator** (i.e., doing useful work).
- ▶ 0.000 GC indicates time spent in **garbage collection**.
- ▶ Both CPU time and actual (elapsed) time are given.

On a larger directory:

```
$ allFiles ~/repos +RTS -s
...
<<ghc: 4354373912 bytes, 4198 GCs, 141265981/824306536
avg/max bytes residency (12 samples), 1624M in use,
0.000 INIT (0.000 elapsed), 4.860 MUT (8.020 elapsed),
2.476 GC (2.480 elapsed) :ghc>>
```

Observations:

- ▶ No output is printed for the first few seconds.
- ▶ 824 megabytes maximum residency!
- ▶ More than a third of total time spent in garbage collection.

Well-Typed

```
allFilesRecursively :: FilePath -> IO [FilePath]
```

When does the list become available?

```
allFilesRecursively :: FilePath -> IO [FilePath]
```

When does the list become available?

- ► After all the effects have been performed.
- ► In particular, producing the list and performing the effects to produce the list is not interleaved.
- ► As a consequence, the whole list is built up in memory and printing only starts once the list is complete.

Well-Typed

Code smells

In many monads (in particular `IO`), functions such as the following are problematic:

```
mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
filterM   :: Monad m => (a -> m Bool) -> [a] -> m [a]
replicateM :: Monad m => Int -> m a -> m [a]
sequence  :: Monad m => [m a] -> m [a]
...
```

In many monads (in particular `IO`), functions such as the following are problematic:

```
mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
filterM   :: Monad m => (a -> m Bool) -> [a] -> m [a]
replicateM :: Monad m => Int -> m a -> m [a]
sequence  :: Monad m => [m a] -> m [a]
...
```

All of these produce a list wrapped in an effect type, and bear the risk of allocating a large structure in memory.

Well-Typed

According to the functor laws, we have

```
(map f . map g) xs = map (f . g) xs
```

According to the functor laws, we have

```
(map f . map g) xs = map (f . g) xs
```

In Haskell, due to lazy evaluation, even without optimisations, both versions are of comparable efficiency.

```
   (map f . map g) (x : xs)
=
   map f (map g (x : xs))
=
   map f (g x : map g xs)
=
   f (g x) : map f (map g xs)
```

Well-Typed

Evaluating the rhs on a non-empty list

```
  map (f . g) (x : xs)
=
  (f . g) x : map (f . g) xs
=
  f (g x) : map (f . g) xs
```

Well-Typed

There is a similar law for `mapM` :

```
(mapM f >=> mapM g) xs = mapM (f >=> g) xs
```

However, here the right hand side is in many cases dramatically more efficient than the left hand side.

Well-Typed

There is a similar law for `mapM` :

```
(mapM f >=> mapM g) xs = mapM (f >=> g) xs
```

However, here the right hand side is in many cases dramatically more efficient than the left hand side.

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(>=>) g f a = g a >>= f
```

Well-Typed

If `f :: a -> Maybe b` and `xs :: [a]`, then

`mapM f xs :: Maybe [b]`

- ▶ The result is `Nothing` if `f` applied to any element of `xs` yields `Nothing`.
- ▶ Therefore, we cannot even determine the top-level constructor of the result without inspecting the entire original list.
- ▶ Thus, in the successful case, we have to build the entire result list in memory before we can return.

Well-Typed

If `f :: a -> IO b` and `xs :: [a]`, then

`mapM f xs :: IO [b]`

- ▶ We expect all effects of `f` applied to any element of `xs` to be performed before we look at the result.
- ▶ In particular, if any of the `f` calls yields an exception, we would excpect it to be triggered before we go on.
- ▶ Therefore, we once again have to build the entire result list in memory before we can return.

Well-Typed

If `f :: a -> Identity b` and `xs :: [a]`, then

`mapM f xs :: Identity [b]`

- ▸ The type `Identity a` is isomorphic to `a`.
- ▸ The function `mapM` on the `Identity` monad behaves exactly as the normal `map`.
- ▸ As a consequence, `mapM f xs` in this case still allows to incrementally consume the result.

Well-Typed

```
(mapM f >=> mapM g) xs = mapM (f >=> g) xs
```

In most cases, the left hand side will build a full intermediate structure, whereas the right hand side will not.

```
(mapM f >=> mapM g) xs = mapM (f >=> g) xs
```

In most cases, the left hand side will build a full intermediate structure, whereas the right hand side will not.

This is unfortunate, because we like to be able to write programs in a **compositional** style.

Towards effectful streams

A non-solution to the original problem

```haskell
allFilesRecursively ::
  FilePath -> IO ()
allFilesRecursively dir = do
  xs <- getDirectoryContents dir
  forM_ xs $ \ x -> do
    if "." `isPrefixOf` x
      then return ()
      else do
        let f = dir </> x
        b <- doesDirectoryExist f
        if b
          then allFilesRecursively f
          else putStrLn f
```

We integrate the printing into the code.

This is better yet non-compositional

```haskell
main :: IO ()
main = do
  [dir] <- getArgs
  allFilesRecursively dir
```

```
$ allFiles ~/repos +RTS -s
...
<<ghc: 3893436296 bytes, 3754 GCs, 1961138/17698152
avg/max bytes residency (65 samples), 36M in use, 0.000
INIT (0.000 elapsed), 3.145 MUT (5.698 elapsed), 0.295 GC
(0.295 elapsed) :ghc>>
```

Much improved maximum residency and GC time.

Well-Typed

Abstracting from the continuation

```haskell
allFilesRecursively ::
  FilePath -> IO ()
allFilesRecursively dir = do
  xs <- getDirectoryContents dir
  forM_ xs $ \ x -> do
    if "." `isPrefixOf` x
      then return ()
      else do
        let f = dir </> x
        b <- doesDirectoryExist f
        if b
          then allFilesRecursively f
          else putStrLn f
```

Well-Typed

Abstracting from the continuation

```haskell
allFilesRecursively ::
  FilePath -> (FilePath -> IO ()) -> IO ()
allFilesRecursively dir yield = do
  xs <- getDirectoryContents dir
  forM_ xs $ \ x -> do
    if "." `isPrefixOf` x
      then return ()
      else do
        let f = dir </> x
        b <- doesDirectoryExist f
        if b
          then allFilesRecursively f yield
          else yield f
```

Well-Typed

```haskell
main :: IO ()
main = do
  [dir] <- getArgs
  allFilesRecursively dir putStrLn
```

- ▶ Restores most of the compositionality.
- ▶ Manually abstracting from the continuation is tedious, error-prone and easy to forget.
- ▶ Can we capture this idea more generally?

Well-Typed

We want a way to define an incremental computation in a monadic way such that

- we can **lift** operations from an underlying monad (e.g. `IO`) and perform them at any point in time,
- we can **yield** individual result elements at any point in time.

A functor for streams

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

A functor for streams

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

Recall `Free` :

```
data Free f a =
    Return a
  | Wrap (f (Free f a))
type Stream b m = Free (StreamF b m)
```

Well-Typed

A functor for streams

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

Recall `Free` :

```
data Free f a =
    Return a
  | Wrap (f (Free f a))
type Stream b m = Free (StreamF b m)
```

We thus have a monad instance for `Stream` .

Wrappers

```
yield :: b -> Stream b m ()
yield b = Wrap (Yield b (Return ()))
```

Well-Typed

Wrappers

```haskell
yield :: b -> Stream b m ()
yield b = Wrap (Yield b (Return ()))

lift :: Functor m => m a -> Stream b m a
lift m = Wrap (Lift (fmap Return m))
```

Wrappers

```haskell
yield :: b -> Stream b m ()
yield b = Wrap (Yield b (Return ()))

lift :: Functor m => m a -> Stream b m a
lift m = Wrap (Lift (fmap Return m))
```

(We cannot make `Stream b` an instance of `MonadTrans` in this form because partial application of type synonyms is not possible in Haskell. Even if it was, `Stream b` would not strictly follow the `MonadTrans` laws – although this would not be such a big issue here.)

Building a stream from a list

```
each :: Monad m => [b] -> Stream b m ()
each []       = return ()
each (x : xs) = yield x >> each xs
```

Mapping over a stream

```
map :: Monad m =>
  (b -> c) -> Stream b m a -> Stream c m a
map _ (Return x)        = return x
map f (Wrap (Lift m))    =
  Wrap (Lift (fmap (map f) m))
map f (Wrap (Yield b k)) =
  Wrap (Yield (f b) (map f k))
```

Well-Typed

Monadically mapping over a stream

```haskell
mapM :: Monad m =>
  (b -> m c) -> Stream b m a -> Stream c m a
mapM _ (Return x)        = return x
mapM f (Wrap (Lift m))   =
  Wrap (Lift (fmap (mapM f) m))
mapM f (Wrap (Yield b k)) = do
  c <- lift (f b)
  yield c
  mapM f k
```

Well-Typed

Producing a stream for every element of a stream

```
for :: Monad m =>
  Stream b m a -> (b -> Stream c m r)
     -> Stream c m a
for (Return a)        _ = return a
for (Wrap (Lift m))    f =
  Wrap (Lift (fmap (flip for f) m))
for (Wrap (Yield b k)) f = do
  f b
  for k f
```

Well-Typed

Taking the first few elements from a stream

```
take :: Monad m =>
  Int -> Stream b m a -> Stream b m ()
take n s
  | n <= 0     = return ()
  | otherwise =
   case s of
     Return _           -> return ()
     Wrap (Lift m)    ->
       Wrap (Lift (fmap (take n) m))
     Wrap (Yield b k) -> do
       yield b
       take (n - 1) k
```

Well-Typed

Back from a stream to a list

```haskell
toList :: Monad m => Stream b m () -> m [b]
toList (Return ())      = return []
toList (Wrap (Lift m))   = m >>= toList
toList (Wrap (Yield b k)) = do
  bs <- toList k
  return (b : bs)
```

This function suffers from the same problem as the original monadic
list functions and will usually not provide the result list incrementally.

Well-Typed

```
effects :: Monad m => Stream b m a -> m a
effects (Return x)         = return x
effects (Wrap (Lift m))    = m >>= effects
effects (Wrap (Yield _ k)) = effects k
```

Well-Typed

```
stdoutLn :: Stream String IO a -> IO a
stdoutLn = effects . mapM putStrLn
```

Note that this is using the stream version of `mapM` .

Original example using streams

Directory contents, as a stream:

```
directoryContents ::
  FilePath -> Stream FilePath IO ()
directoryContents dir =
  lift (getDirectoryContents dir) >>= each
```

Well-Typed

Original example using streams

Directory contents, as a stream:

```
directoryContents ::
  FilePath -> Stream FilePath IO ()
directoryContents dir =
  lift (getDirectoryContents dir) >>= each
```

Note that the files from an individual directory are still not produced incrementally, because

```
getDirectoryContents :: FilePath -> IO [FilePath]
```

does not deliver them that way.

Well-Typed

Original example using streams (contd.)

```
allFilesRecursively ::
  FilePath -> Stream FilePath IO ()
allFilesRecursively dir =
  for (directoryContents dir) $ \ x -> do
    if "." `isPrefixOf` x
      then return ()
      else do
        let f = dir </> x
        b <- lift (doesDirectoryExist f)
        if b
          then allFilesRecursively f
          else yield f
```

Well-Typed

Original example using streams (contd.)

```haskell
main :: IO ()
main = do
  [dir] <- getArgs
  stdoutLn (allFilesRecursively dir)
```

```
$ allFiles ~/repos +RTS -s
...
<<ghc: 5221176184 bytes, 5031 GCs, 1309257/15058184
avg/max bytes residency (117 samples), 39M in use, 0.000
INIT (0.000 elapsed), 3.900 MUT (6.226 elapsed), 0.395 GC
(0.394 elapsed) :ghc>>
```

Comparable to non-compositional or hand-written continuation
versions.

Stream functions can be composed easily:

```
main :: IO ()
main = do
  [dir, n] <- getArgs
  stdoutLn
    (take (read n) (allFilesRecursively dir))
```

This will stop early and not traverse the parts of the directory structure that are not needed to produce the first `n` results.

Well-Typed

Using streams with other monads

```
halve :: Int -> Maybe Int
halve n =
  if odd n then Nothing else Just (n `div` 2)

GHCi> toList (take 3
        (mapM halve (each [2, 4 ..])))
Just [1, 2, 3]
GHCi> toList (take 3
        (mapM halve (each [1 ..])))
Nothing
```

Well-Typed

As a library

The functionality we just described is offered in very similar form by the `streaming` package.

Our type:

```haskell
data StreamF b m r =
    Lift (m r)
  | Yield b r
  deriving (Functor)
data Free f a =
    Return a
  | Wrap (f (Free f a))
type Stream b m = Free (StreamF b m)
```

Well-Typed

Their type:

```
data Stream f m r =
    Step !(f (Stream f m r))
  | Effect (m (Stream f m r))
  | Return r
data Of a b = !a :> b  -- a left-strict pair
```

Apart from the strictness annotations, their `Stream (Of a)` is isomorphic to our `Stream a` .

The `streaming` package comes with its own prelude module,
providing replacements for many common list functions and
generalised versions of some of our own stream functions, e.g.:

```
each       :: (Monad m, Foldable f) => f a -> Stream (Of a) m ()
fromHandle :: MonadIO m => Handle -> Stream (Of String) m ()
toHandle   :: MonadIO m => Handle -> Stream (Of String) m r -> m r
stdinLn    :: MonadIO m => Stream (Of String) m ()
stdoutLn   :: MonadIO m => Stream (Of String) m () -> m ()
iterateM   :: Monad m => (a -> m a) -> m a -> Stream (Of a) m r
repeatM    :: Monad m => m a -> Stream (Of a) m r
mapM       :: Monad m => (a -> m b) -> Stream (Of a) m r -> Stream (Of b) m r
filterM    :: Monad m =>
  (a -> m Bool) -> Stream (Of a) m r -> Stream (Of a) m r
for        :: (Monad m, Functor f) =>
  Stream (Of a) m r -> (a -> Stream f m x) -> Stream f m r
```

Well-Typed

More advanced libraries

Some applications require yet more control:

- creating a buffer of a particular size,
- applying "back pressure", i.e., detecting that a consumer has difficulty keeping up and slowing down,
- …

To a certain extent, the `streaming` package allows this by replacing `Of` with a different functor – but there are also packages such as `pipes` and `conduit`.

In the `streaming` approach, next to lifting an effect, we have but one option, to `yield` a value "downstream". Yielding a value has no response.

In the `streaming` approach, next to lifting an effect, we have but one option, to `yield` a value "downstream". Yielding a value has no response.

In both `pipes` and `conduit`, each component can communicate both upstream and downstream:

- it can "request" a piece of information upstream, by sending a message;
- it can "respond" a piece of information downstream, receiving a confirmation.

The `Proxy` type

The core type of the `pipes` package is a `Proxy`:

```haskell
data Proxy a' a b' b m r =
    Request a' (a  -> Proxy a' a b' b m r )
  | Respond b  (b' -> Proxy a' a b' b m r )
  | M        (m       (Proxy a' a b' b m r))
  | Pure     r
```

`Request` is for upstream communication.

`Respond` is for downstream communication.

`M` corresponds to `Lift`.

`Pure` corresponds to `Return`.

Well-Typed

The core type of the `pipes` package is a `Proxy` :

```
data Proxy a' a b' b m r =
    Request a' (a  ->  Proxy a' a b' b m r )
  | Respond b  (b' ->  Proxy a' a b' b m r )
  | M        (m         (Proxy a' a b' b m r))
  | Pure     r
```

The `Stream (Of a)` type corresponds to

```
type Producer a = Proxy Void () () a
```

Indeed, we also have

```
yield :: Monad m => a -> Producer a m ()
```

Well-Typed

Producers

```haskell
type Producer a = Proxy Void () () a
```

Producers cannot send requests upstream – indicated by `Void`.

Producers can send `a` values downstream and receive nothing – indicated by `()` – in return.

Well-Typed

Another special case:

```
type Consumer a = Proxy () a () Void
```

Consumers can request values of type `a` from upstream by sending `()`.

Consumers cannot send anything downstream – indicated by `Void`.

Well-Typed

Consumers

Another special case:

```
type Consumer a = Proxy () a () Void
```

Consumers can request values of type `a` from upstream by sending `()` .

Consumers cannot send anything downstream – indicated by `Void` .

Where producers `yield` , consumers

```
await :: Monad m => Consumer a m a
```

Well-Typed

The generality to send multiple types of requests, or receive multiple kinds of confirmations, is rarely used:

```
type Pipe a b = Proxy () a () b
```

A **pipe** can receive `a` items from upstream, and send `b` items to downstream.

There is a choice between **push**- and **pull**-based composition:

- ▶ we can start running the downstream proxy, and once it requests a value from upstream, evaluate upstream as far as necessary to be able to pull;
- ▶ or we can start running the upstream proxy, and once it responds a value to downstream, evaluate downstream as far as necessary to be able to push.

The default is pull-based composition, but the `pipes` package offers both if full control is desired.

The standard composition operator is

```
(>->) :: Monad m
  => Proxy a' a () b m r
  -> Proxy () b c' c m r
  -> Proxy a' a c' c m r
```

The resulting proxy has:

- the upstream interface of the first argument,
- the downstream interface of the second argument,
- the intermediate interface must match.

```
type Effect = Proxy Void () () Void
```

An effect can neither `yield` nor `await`.

It can only produce effects in the underlying monad, and have a final result.

```haskell
type Effect = Proxy Void () () Void
```

An effect can neither `yield` nor `await`.

It can only produce effects in the underlying monad, and have a final result.

Only effects can be "run":

```haskell
runEffect :: Monad m => Effect m r -> m r
```

Well-Typed

Examples

```haskell
stdinLn  :: MonadIO m => Producer String m ()
stdoutLn :: MonadIO m => Consumer String m ()
```

```
stdinLn  :: MonadIO m => Producer String m ()
stdoutLn :: MonadIO m => Consumer String m ()

echo :: MonadIO m => m ()
echo = runEffect (stdinLn >-> stdoutLn)
```

performs an "echo" of each user input.

Well-Typed

```
map :: Monad m => (a -> b) -> Pipe a b m r
```

```
map :: Monad m => (a -> b) -> Pipe a b m r

shout :: MonadIO m => m ()
shout = runEffect $
  stdinLn >-> map (fmap toUpper) >-> stdoutLn
```

Well-Typed

```
take :: Monad m => Int -> Pipe a a m ()
```

```
take :: Monad m => Int -> Pipe a a m ()

shoutTwice :: MonadIO m => m ()
shoutTwice = runEffect $
      stdinLn
  >-> map (fmap toUpper)
  >-> take 2
  >-> stdoutLn
```

Well-Typed

```
readLn :: (Read a, MonadIO m) => Producer a m ()
takeWhile :: Monad m => (a -> Bool) -> Pipe a a m ()
sum :: (Num a, Monad m) => Producer a m () -> m a
```

```haskell
readLn :: (Read a, MonadIO m) => Producer a m ()
takeWhile :: Monad m => (a -> Bool) -> Pipe a a m ()
sum :: (Num a, Monad m) => Producer a m () -> m a

sumInputs :: MonadIO m => m Int
sumInputs = sum $ readLn >-> takeWhile (/= 0)
```

Well-Typed

Yet another package (ecosystem) based on the same ideas:

```
ConduitM i o m r    Pipe i o m r
Source m o          Producer o m ()
Sink i m r          Consumer i m r
(.|)                (>->)
```

There are some minor differences, e.g. the conduit type of `await` can detect whether the upstream component is finished:

```
await :: Monad m => Sink i m (Maybe i)
```

Summary and comparison

- Understanding the `Stream` type is key to understanding all the approaches.
- For unstanding the `Stream` type, the most important ingredient is understanding the it is just an instance of a free monad, and running streams makes use of the fact that we can inspect the streams we build this way.

Summary and comparison (contd.)

- The `streaming` package is the most recent of the discussed packages, and in a way, the simplest. For many cases, it is enough, and compellingly easy to use.

- The `pipes` package has a reputation as the theoretically most elegant. It is immensely powerful, but can also be a bit intimidating.

- The `conduit` package has gone through many iterations and is now very similar to `pipes`. It is currently the most widely used package in this area.