The Core of `streaming`

Jack Kelly

November 12^{th}, 2024

Why Stream?

- Same reasons as for other languages:
  - Interleave effectful work and the production of results
  - Process lots of data in bounded space

How to Stream?

- `iteratee`
- `io-streams`
- `pipes`
- `conduit`
- `machines`
- `streamly`
- `streaming`
- `...`

Why I like `streaming`

- (Reasonably) simple core type
- Not too many type variables
- No custom operators
- Not too many warts
- **Let's build its core type!**

Attempt 1: Lists

```
data Stream a =
  = Step a (Stream a)
  | Done
```

- **Problem:** When do effects happen?
- **Non-solution:** Lazy I/O.
- **Solution:** Make them explicit.

Aside: Problems with lazy I/O

What does this program print?

```
main :: IO ()
main = do
  lineCount <- withFile "temp.txt" ReadMode $ \h ->
    length . lines <$> hGetContents h
  putStrLn $ show (lineCount :: Int) ++ " lines counted."
```

Aside: Problems with lazy I/O

What does this program print?

```
main :: IO ()
main = do
  lineCount <- withFile "temp.txt" ReadMode $ \h ->
    length . lines <$> hGetContents h
  putStrLn $ show (lineCount :: Int) ++ " lines counted."
```

- `hGetContents: illegal operation (delayed read on closed handle)`

Attempt 2: Adding Effects

- Add type variable `m` and constructor `Effect`.
- `m` is almost always a `Monad`.
- Now we know when we need to do effectful work:

```
data Stream a m =
  = Step a (Stream a m)
  | Effect (m (Stream a m))
  | Done
```

- Example: `untilJust :: m (Maybe a) -> Stream a m`
- **Next Problem:** There could be an unbounded amount of work behind each `a`.
- **Solution:** Make the stream strict in `a`.

Attempt 3: Adding Strictness

- Force each "a" (to at least WHNF) before putting it in the `Stream`:

```
data Stream a m =
  = Step !a (Stream a m)
  | Effect (m (Stream a m))
  | Done
```

- **Next Problems:**
  - How to split a stream without needlessly buffering?
  - How to return an error result?
- **Solution:** Let `Done` carry a result.

Attempt 4: Adding Results

- Add a result type `r` to the `Done` constructor:

```
data Stream a m r =
  = Step !a (Stream a m r)
  | Effect (m (Stream a m r))
  | Done r
```

- **Example:**
  `splitAt :: Int -> Stream a m r -> Stream a m (Stream a m r)`
- **Example:** `untilLeft :: m (Either r a) -> Stream a m r`

One weird trick

- Un-inline the item and rest-of-stream from `Step`
- `Stream (Of a)` is isomorphic to the previous slide's `Stream a`
  - `Of` is partially applied!
- We have now reached the "real" type from `streaming`

```
data Of a b = !a :> b deriving Functor

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

What does this enable? Lots!

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

- A `Step` constructor now contains an inner stream, which returns the remainder of the outer stream when it's done

What else does this enable?

```
copy ::
  (Monad m) =>
  Stream (Of a) m r ->
  Stream (Of a) (Stream (Of a) m) r
```

- An `Effect` constructor now wraps an inner stream, which yields a second copy of every element.

But wait, there's more!

- Parsing an archive format:
  - First, a `Header` including the name and length of all blobs
  - Then, a concatenated sequence of compressed blobs

```
data Header = Header { records :: [Record] }
data Record = Record { name :: Text, compressedLength :: Int }
decodeHeader ::
  Stream (Of ByteString) m r ->
  m (Either String (Header, Stream (Of BytesString) m r))
```

But wait, there's more!

- Parsing an archive format:
  - First, a `Header` including the name and length of all blobs
  - Then, a concatenated sequence of compressed blobs

```
data Header = Header { records :: [Record] }
data Record = Record { name :: Text, compressedLength :: Int }
decodeHeader ::
  Stream (Of ByteString) m r ->
  m (Either String (Header, Stream (Of BytesString) m r))

data Blob m r =
  Blob { name :: Text, data_ :: Stream (Of ByteString) m r }
  deriving Functor
decodeBlobs ::
  MonadIO m =>
  Header -> Stream (Of ByteString) m r -> Stream (Blob m) m r
```

It's not perfect

- `ByteString` is idiomatically handled by `streaming-bytestring`
  - `ByteStream m r` is like `Stream (Of ByteString) m r`, unpacked and inlined for efficiency
- No early finalisation
  - Difficult to say "I'm done now, close the `Handle`" without extra effort
- Still possible to buffer more than planned, if you really try

The future of streaming

- A port of `streaming` to `linear-base`
- Enables some cool stuff:

```
data Header = Header { records :: [Record] }
data Record = Record { name :: Text, compressedLength :: Int }
data Blob m r =
  Blob {
    name :: Text,
    data_ :: Stream (Of ByteString) m r
  } deriving Functor

decodeBlob ::
  Handle %1 ->
  Record ->
  Blob m Handle
```