Fun With Haskell: Fast Haskell

Nathaniel Wesley Filardo

January 18, 2012

Metadata
Overview of today

Today I want to give an overview of some of the plumbing used in at-scale Haskell programs.

- `bytestring` for chunked handling of strings.
- Builders (e.g. `blaze-builder`) for efficient construction of output.
- `conduit` package for managing streaming data.
  - (Replacement for `enumerator`.)

(In particular, all of these are used by Yesod.)

ByteString

- Suppose my goal is to fling bytes around as fast as possible.

ByteString

- Suppose my goal is to fling bytes around as fast as possible.
- Not a crazy goal: lots of bytes out there.

ByteString
What's wrong with String?

- Recall: `type String = [Char]`.
- What is `Char`, anyway?

```
> import Data.Char
> minBound :: Char
'\NUL'
> ord maxBound
1114111
```

ByteString
What's wrong with String?

- Recall: `type String = [Char]`.
- What is `Char`, anyway?

```
> import Data.Char
> minBound :: Char
'\NUL'
> ord maxBound
1114111
```

- What is *that*?

ByteString
What's wrong with String?

- Recall: `type String = [Char]`.
- What is `Char`, anyway?

```
> import Data.Char
> minBound :: Char
'\NUL'
> ord maxBound
1114111
```

- What is *that*?
- `0x10FFFF`.

ByteString
What's wrong with String?

- Recall: `type String = [Char]`.
- What is `Char`, anyway?
- It's an abstract *unicode code point*.

ByteString
What's wrong with String?

- Recall: `type String = [Char]`.
- What is `Char`, anyway?
- It's an abstract *unicode code point*.
- Unicode is sort of a figment of everybody's imagination.
    - It's great for what it is, but:
    - No canonical mapping to/from reality.

ByteString
What's wrong with String?

- Recall: `type String = [Char]`.
- What is `Char`, anyway?
- It's an abstract *unicode code point*.
- Unicode is sort of a figment of everybody's imagination.
    - It's great for what it is, but:
    - No canonical mapping to/from reality.
- Importantly: it's not a byte.

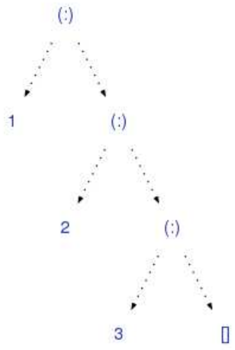
ByteString
What's wrong with `[Word8]`, then?

- Haskell has `Data.Word`.
- So: use `[Word8]`?
- What does a list look like, anyway?

ByteString

What's wrong with ***[Word8]****, then?*

Consider [1,2,3]:



- Each node here is a separate thing on the heap!
- Each arrow is a pointer, maybe with bad locality.
- *5 machine words* of data *for each byte*!

ByteString
What's wrong with `[Word8]`, then?

OW!

ByteString
What's wrong with `[Word8]`, then?

Probably fine for toy programs. What do we want for real?

- A structure with *good cache performance*:
- Amortize pointer overhead and chases by giving us lots of data each time.

ByteString
Strict ByteStrings

- Anybody know any data structures like that?
- Maybe even a structure that doesn't use pointers internally?

ByteString
Strict ByteStrings

- Anybody know any data structures like that?
- Maybe even a structure that doesn't use pointers internally?
- (Strict) Arrays!

ByteString
Strict ByteStrings

- Anybody know any data structures like that?
- Maybe even a structure that doesn't use pointers internally?
- (Strict) Arrays!
- Since they're strings we also want offset and length information.

ByteString
Strict ByteStrings

- Anybody know any data structures like that?
- Maybe even a structure that doesn't use pointers internally?
- (Strict) Arrays!
- Since they're strings we also want offset and length information.
- Behold! A `Data.ByteString`:

```
data ByteString = PS
 {-# UNPACK #-} !(ForeignPtr Word8) -- payload
 {-# UNPACK #-} !Int                -- offset
 {-# UNPACK #-} !Int                -- length
```

ByteString
Strict ByteStrings

```
data ByteString = PS
 {-# UNPACK #-} !(ForeignPtr Word8) -- payload
 {-# UNPACK #-} !Int                -- offset
 {-# UNPACK #-} !Int                -- length
```

- Lots of fanciness we haven't covered (sorry!)
- But: this is essentially just what you'd do in C.
    - Pointer to memory
    - Current position
    - Total length
- Implemented using all kinds of neat IO tricks.
- Hidden behind a much nicer interface.

ByteString
Strict ByteStrings

- OK, scary stuff off the screen.
- `ByteStrings` are `Eq`, `Ord`, `Show`, so on.
- Constructors and destructors; `empty` and `singleton` and

```
pack :: [Word8] -> ByteString
unpack :: ByteString -> [Word8]
```

- API has everything we might want: maps, folds, search by `Word8`, search by substring, IO, etc.

ByteString
Lazy ByteStrings

- Good so far, right?

ByteString
Lazy ByteStrings

- Good so far, right?
- Anything not so great about arrays?

ByteString
Lazy ByteStrings

- Good so far, right?
- Anything not so great about arrays?
- Do a lot of copying to maintain purity.

ByteString
Lazy ByteStrings

- Good so far, right?
- Anything not so great about arrays?
- Do a lot of copying to maintain purity.
- Suggestions for what we might do instead?

ByteString
Lazy ByteStrings

- Good so far, right?
- Anything not so great about arrays?
- Do a lot of copying to maintain purity.
- Suggestions for what we might do instead?
- List of array chunks!

ByteString
Lazy ByteStrings

- Good so far, right?
- Anything not so great about arrays?
- Do a lot of copying to maintain purity.
- Suggestions for what we might do instead?
- List of array chunks!
- Behold: `Data.ByteString.Lazy`!

```
import qualified Data.ByteString.Internal as S
data ByteString =
   Empty
 | Chunk {-# UNPACK #-} S.ByteString ByteString
```

ByteString
Lazy ByteStrings

```
import qualified Data.ByteString.Internal as S
data ByteString =
   Empty
 | Chunk {-# UNPACK #-} S.ByteString ByteString
```

- Ah ha, a *lazy* list of *strict* ByteString `Chunks`.

ByteString
Lazy ByteStrings

- Same API as last time, but different complexities for calls.
- Most $O(n)$ now down to $O(n/c + c)$.
- `length` a little more expensive, but that's OK.

ByteString
Do these solve all our problems?

- (Answer: no.)

ByteString
Do these solve all our problems?

- (Answer: no.)
- If I try to produce output, what operation am I likely to do over and over?

ByteString
Do these solve all our problems?

- (Answer: no.)
- If I try to produce output, what operation am I likely to do over and over?
- Append! Suppose I do n appends on
    - A strict `ByteString`: $O(n)$ copies of at least $O(n)$ data.

ByteString
Do these solve all our problems?

- (Answer: no.)
- If I try to produce output, what operation am I likely to do over and over?
- Append! Suppose I do n appends on
  - A strict `ByteString`: $O(n)$ copies of at least $O(n)$ data.
  - A list? $O(n)$ copies of $O(n)$ data.

ByteString
Do these solve all our problems?

- (Answer: no.)
- If I try to produce output, what operation am I likely to do over and over?
- Append! Suppose I do n appends on
  - A strict `ByteString`: $O(n)$ copies of at least $O(n)$ data.
  - A list? $O(n)$ copies of $O(n)$ data.
  - A lazy `ByteString`? Also quadratic.

ByteString
Do these solve all our problems?

- Strict byte strings are *great* if you can get them.
  - They might stink to build, however.
  - Managing lots of long-lived ones might lead to leaks.
- Lazy byte strings a little better:
  - Stink a little less to build.
  - Can produce them chunk-at-a-time, lazily.
  - Can collect them in pieces.

ByteString
Do these solve all our problems?

- Often, really need to do lots of appends.
- Especially of small objects!
    - Implies a lot of copies of arrays and/or list spines.
- What are we to do?
    - Those imperative people are laughing at us. :(

Builder
Difference Lists

- Wait, we do have a trick up our sleeve!
- Function composition! :)

Builder
Difference Lists

- Why is (++) quadratic in the first place?
- Consider `x = ([1,2] ++ [3,4]) ++ [5]`.
- Recall (++):

```
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

Builder
Difference Lists

```
(++) (x:xs) ys = x : xs ++ ys


(([1,2] ++ [3,4]) ++ [5])
(1:([2] ++ [3,4]) ++ [5])    -- visit 1
(1:2:([] ++ [3,4]) ++ [5])  -- visit 2
(1:2:[3,4] ++ [5])
1:(2:[3,4] ++ [5])           -- visit 1 again
1:2:([3,4] ++ [5])           -- visit 2 again
1:2:3:([4] ++ [5])           -- visit 3
1:2:3:4:([] ++ [5])          -- visit 4
1:2:3:4:[5]
```

Builder
Difference Lists

- So: want to avoid visiting things over and over.
- Try this:
  - Build a *function* which takes the "rest of the list" and returns the "whole list"
  - A *prefix-concatenation* function.
  - `:: [a] -> [a]`
- `[1,2] == (\t -> 1:2:t) []`

Builder
Difference Lists

- So: want to avoid visiting things over and over.
- Try this:
  - Build a *function* which takes the "rest of the list" and returns the "whole list"
  - A *prefix-concatenation* function.
  - `:: [a] -> [a]`
- `[1,2] == (\t -> 1:2:t) []`
- Easy to append:

```
append da db t = da (db t)
```

Builder
Difference Lists

- So what happens when we evaluate this one?

```
append (\t -> 1:2:t)
       (append (\t -> 3:4:t) (\t -> 5:t))
       x
```

- It turns out to be pretty quick!

```
append (\t -> 1:2:t)
       (append (\t -> 3:4:t) (\t -> 5:t))
       x
1:2:(append (\t -> 3:4:t) (\t -> 5:t) x)
1:2:3:4:((\t -> 5:t) x)
1:2:3:4:5:x
```

Builder

- Moral of the story: sometimes you can get what you want by *building* a *recipe* and then invoking it.
- So, what did we want, again?

Builder

- Moral of the story: sometimes you can get what you want by *building* a *recipe* and then invoking it.
- So, what did we want, again?
- Fast concatenation of small objects.

Builder

- Moral of the story: sometimes you can get what you want by *building* a *recipe* and then invoking it.
- So, what did we want, again?
- Fast concatenation of small objects.
- But big buffers for amortization (e.g. syscalls).

Builder

- Moral of the story: sometimes you can get what you want by *building* a *recipe* and then invoking it.
- So, what did we want, again?
- Fast concatenation of small objects.
- But big buffers for amortization (e.g. syscalls).
- Builders capture this for us.

Builder

- There are a few lurking around Hackage.
- `blaze-builder` currently popular.
    - Provides `Blaze.ByteString.Builder` modules.
    - Uses UTF-8 encodings to get bytes.

- Once you have built your recipe, you run it with

```
toLazyByteString :: Builder -> ByteString
toByteString :: Builder -> ByteString
-- other, more fancy forms
```

Builder

- There are a few lurking around Hackage.
- `blaze-builder` currently popular.
  - Provides `Blaze.ByteString.Builder` modules.
  - Uses UTF-8 encodings to get bytes.
  - (Likely replaced by a new builder in the `bytestring` library in the next release.)
- Once you have built your recipe, you run it with

```
toLazyByteString :: Builder -> ByteString
toByteString :: Builder -> ByteString
-- other, more fancy forms
```

Builder
Building Builders

- OK, so...

Builder
Building Builders

- OK, so...
- Oh right, appending them.
- `Builder` is a `Monoid`, so use

```
mappend :: Monoid a => a -> a -> a
```

- or a `Writer` monad (transformer).

Builder
Building Builders

- Ah, getting one in the first place.
- `Blaze.ByteString.Builder.Char.Utf8`:

```
fromChar :: Char -> Builder
fromString :: String -> Builder
fromShow :: Show a => a -> Builder
```

- `Blaze.ByteString.Builder.Int`:

```
fromInt8 :: Int8 -> Builder
{- ... -}
fromInt64sle :: [Int64] -> Builder
```

- Other builtins, and mechanisms for adding your own.

Builder
Building Builders

BuilderTest.hs

```
import Data.Char
import Data.Monoid
import Blaze.ByteString.Builder
import Blaze.ByteString.Builder.Char.Utf8

main = print $ toByteString $
  fromString "I would like to show you: "
    `mappend`
  fromShow (6*5)
    `mappend`
  fromInt8 (fromIntegral $ ord '!')
```

More on This

If this has piqued your interest, and you want more detail:

- Real World Haskell [2, ch. 8,13].
- Simon Meier's Guided Tour Through The ByteString Library [1].
- Hackage documentation for `ByteString` and `blaze-builder`.
- The upcoming `ByteString-builder` in ByteString `0.10.0.0`.

Conduit
What are they?

"[a] solution to the streaming data problem" [3]. What's the problem?

- Imagine writing the `zgrep` tool.
- Need to read a chunk of compressed data, expand it, and grep through it.
- And: don't forget the boundary cases.
- Who thinks manual buffer management is fun? (more than once?)

Conduit
What are they?

"[a] solution to the streaming data problem" [3]. What's the problem?

- Imagine writing the `zgrep` tool.
- Need to read a chunk of compressed data, expand it, and grep through it.
- And: don't forget the boundary cases.
- Who thinks manual buffer management is fun? (more than once?)
- Want, instead, a logical abstraction of streams of data:
  - grep asks the decompressor for data
  - the decompressor asks the file for data
  - Eventually, maybe, grep is done and closes the stream or the file ends.

Conduit
What are they?

- "[a] solution to the streaming data problem" [3].

Conduit
What are they?

- "[a] solution to the streaming data problem" [3].
  - Generic `Source` and `Sink` abstractions.

Conduit
What are they?

- "[a] solution to the streaming data problem" [3].
  - Generic `Source` and `Sink` abstractions.
  - Transformations in the middle: gzip, UTF-8 codec, ...

Conduit
What are they?

- "[a] solution to the streaming data problem" [3].
  - Generic `Source` and `Sink` abstractions.
  - Transformations in the middle: gzip, UTF-8 codec, ...
  - *Deterministic* resource management (e.g. file descriptors)

Conduit
What are they?

- “[a] solution to the streaming data problem” [3].
  - Generic `Source` and `Sink` abstractions.
  - Transformations in the middle: gzip, UTF-8 codec, ...
  - *Deterministic* resource management (e.g. file descriptors)
  - All in a relatively simple API.

Conduit
What are they?

- "[a] solution to the streaming data problem" [3].
  - Generic `Source` and `Sink` abstractions.
  - Transformations in the middle: gzip, UTF-8 codec, ...
  - *Deterministic* resource management (e.g. file descriptors)
  - All in a relatively simple API.
- Disclaimer: I am "borrowing" from Michael Snoyman's (the package author) blog. [4] and prior in the series.

Conduit
What are they?

Glossary:

- `Source`: whence data comes.
- `Sink`: whither data goes.
- `Conduit`: A (stateful) data manipulation function.
    - Looks like a sink on one side, and a source on the other.
- `Fuse`: The act of combining a conduit to ...
    - a source; results in a source. `($=)`
    - a sink; results in a sink. `(=$)`
    - a conduit; results in a conduit. `(=$=)`
- `Combine`: Joining a source and a sink `($$)`; causes data to flow until either (or both) are done.

Conduit
What are they?

Why do we need “resource management” in this story at all?

- Sources and sinks might want to open files.

Conduit
What are they?

Why do we need "resource management" in this story at all?

- Sources and sinks might want to open files.
- Conduits might also!

Conduit
What are they?

Why do we need "resource management" in this story at all?

- Sources and sinks might want to open files.
- Conduits might also!
- Want to close files as soon as we're done with them (don't want to wait for all references to go away and the GC to run "finalizers").

Conduit
What are they?

Why do we need "resource management" in this story at all?

- Sources and sinks might want to open files.
- Conduits might also!
- Want to close files as soon as we're done with them (don't want to wait for all references to go away and the GC to run "finalizers").
- Want to free resources on exceptions.
    - Including asynchronous exceptions, so we can kill off long-running threads by timeouts, etc.

Conduit
Sources

- Sources are relatively simple:

```
data SourceResult a = Open a | Closed
data PreparedSource m a = PreparedSource
  { sourcePull :: ResourceT m (SourceResult a)
  , sourceClose :: ResourceT m ()
  }
```

- Only two things you can do to `PreparedSources`:
  - Close the source
  - Ask for more data ("pull")

Conduit
Sources

- Sources are relatively simple:

```
data SourceResult a = Open a | Closed
data PreparedSource m a = PreparedSource
  { sourcePull :: ResourceT m (SourceResult a)
  , sourceClose :: ResourceT m ()
  }
```

- Only two things you can do to `PreparedSources`:
    - Close the source
    - Ask for more data ("pull")
- The source will respond to a pull either with data (`Open a`) or by signaling the end.

Conduit
Sources

Consider a source which forever returns the same piece of data:

```
repeat :: Monad m => a -> PreparedSource m a
repeat a = PreparedSource
  { sourcePull = return $ Open a
  , sourceClose = return ()
  }
```

What would the source `eof` which never returned any data look like?

Conduit
Sources

Invariants of the system for `Sources`

- Won't ask for another pull after one returns `Closed`.
- Don't close a source after it has said it was closed.
- Don't close a source multiple times.

(These invariants are not enforced but may be assumed by all `Sources` and should be maintained by all other bits of code.)

Conduit
Sources with State

- The source "methods" are monadic actions.
- Suppose we want a source which streams all Nats:
- Make a `PreparedSource` that closes over an `IORef`:

```
mkNatSource = do
  r <- newRef 0 -- provided by ResourceT
  return $ PreparedSource
         { sourceClose = return ()
         , sourcePull = do
             next <- readRef r
             writeRef r (next+1)
             return next
         }
```

Conduit
Sources with State

- Very frequently have a monadic action to make a `PreparedSource`.
- Conduit calls these actions Sources:

```
newtype Source m a = Source
  { prepareSource :: ResourceT m
                       (PreparedSource m a) }
```

- So should have

```
mkNatSource = Source $ do
    {- ... -}
```

Conduit
Sources with State

- Utility function for stateful sources:

```
sourceState :: Resource m
  => state
  -> (state -> ResourceT m (state
                           ,SourceResult output))
  -> Source m output
```

- And one for IO state:

```
sourceIO :: ResourceIO m
  => IO state              -- open
  -> (state -> IO ())      -- close
  -> (state -> m (SourceResult output))
  -> Source m output
```

Conduit
Sources with State

These curious `ResourceT` and such are how `conduit` manages to track and free resources.

- Register some cleanup action:

```
register :: IO () -> ResourceT IO ReleaseKey
```

- Explicitly call some cleanup (guaranteed to happen at most once):

```
release :: ReleaseKey -> ResourceT IO ()
```

- Run a computation ensuring that everything gets released:

```
runResourceT :: ResourceT IO a -> IO a
```

Conduit
Sources with State

These curious `ResourceT` and such are how `conduit` manages to track and free resources.

- Register some cleanup action:

```
register :: IO () -> ResourceT IO ReleaseKey
```

- Explicitly call some cleanup (guaranteed to happen at most once):

```
release :: ReleaseKey -> ResourceT IO ()
```

- Run a computation ensuring that everything gets released:

```
runResourceT :: ResourceT IO a -> IO a
```

- (Actually more polymorphic than slideware allows)

Conduit
Sinks

- OK, now we can produce data.
- Could actually use sources directly in monadic code (call `sourcePull` and `sourceClose` ourselves).
- Sinks take a *stream of input* and produce *exactly one output*.
- As with sources, sinks can be in two states:

```
data SinkResult in out =
   Processing
 | Done (Maybe in) out
```

- (When it's `Done` it may have some input left over.)

Conduit
Sinks

- Sinks come in one of two flavors:

```
data PreparedSink in m out =
    SinkNoData out
  | SinkData
    { sinkPush :: in
               -> ResourceT m (SinkResult in out)
    , sinkClose :: ResourceT m out
    }
```

- Some sinks are trivial and need no data.
- The rest need to be fed some input.
    - May return a result before the end of the stream.
    - Obligated to return a result when the stream ends.

Conduit
Sinks

- And, as before, a Sink is really a monadic computation returning a prepared Sink:

```
newtype Sink in m out = Sink
  { prepareSink :: ResourceT m
                   (PreparedSink in m out) }
```

Conduit
Sinks

A simple sink which counts the number of inputs:

```
count = Sink $ do
  r <- newRef 0
  return $ PreparedSink
         { sinkClose = readRef r
         , sinkPush _ = do
             n <- readRef r
             writeRef r (n+1)
             return Processing
         }
```

Conduit
Sinks

- Helpers for state state and IO. e.g.:

```
sinkState :: Resource m
  => s
  -> (s -> in -> ResourceT m
                (s, SinkResult in out))
  -> (s -> ResourceT m out)
  -> Sink in m out
```

- So:

```
count' = sinkState 0
  (\s _ -> return (s+1,Processing))
  (\s -> return s)
```

Conduit
Sinks

- The `Data.Conduit.List` module provides a useful source:

```
sourceList :: Resource m => [a]
                         -> Source m a
```

- and many useful sinks, such as

```
fold :: Resource m => (b -> a -> b)
                   -> b -> Sink a m b
take :: Resource m => Int -> Sink a m [a]
drop :: Resource m => Int -> Sink a m ()
```

Conduit
Sinks

And (ta-da): `Sinks` are `Monads`!

- What does that even mean?

Conduit
Sinks

And (ta-da): `Sinks` are `Monads`!

- What does that even mean?
- It means we can compose sinks together!

SinkMonadEx.hs

```
import Data.Conduit.List as CL
foo b = do
  xs <- CL.take 5
  CL.drop 5
  CL.fold (+) (foldr (*) b xs)
```

Conduit
Combining Sources and Sinks

A first example: sum up the entries in a list. (`ConduitSumList.hs`)

- Source: `sourceList`.
- Sink:

```
sinkSum = CL.fold (+) 0
```

- Glue them together with `($$)`:

```
pipe l = sourceList l $$ sinkSum
```

- Then run it:

```
main = runResourceT $ (pipe [1,2,3])
        >>= print
```

Conduit
Conduits

- We'd like to do something more interesting,

Conduit
Conduits

- We'd like to do something more interesting,
- Maybe
    - Read from a file.
    - Decode UTF-8.
    - Chunk file into lines.
    - Accumulate each Int into the total
    - Write the stream of totals to file.

Conduit
Conduits

- We'd like to do something more interesting,
- Maybe
    - Read from a file.
    - Decode UTF-8.
    - Chunk file into lines.
    - Accumulate each Int into the total
    - Write the stream of totals to file.
- Middle three stages are *data transformers*, or `Conduits`.

Conduit
Conduits

Let's look at the types:

```
data ConduitResult i o =
   Producing [o] | Finished (Maybe i) [o]

data PreparedConduit i m o = PreparedConduit
  { conduitPush :: i
                -> ResourceT m (ConduitResult i o)
  , conduitClose :: ResourceT m [o] }

newtype Conduit i m o = Conduit
  { prepareConduit :: ResourceT m
                        (PreparedConduit i m o) }
```

Conduit
Conduits

- A pass-through conduit is straightforward:

```
pt = Conduit $ return $ PreparedConduit
  { conduitPush = \i -> return (Producing [i])
  , conduitClose = return [] }
```

- So is a one-to-one mapper:

```
mapM f = Conduit $ return $ PreparedConduit
  { conduitPush = \i -> do
        fi <- lift $ f i
        return (Producing [fi])
  , conduitClose = return [] }
```

Conduit
Conduits

- A pass-through conduit is straightforward:

```
pt = Conduit $ return $ PreparedConduit
  { conduitPush = \i -> return (Producing [i])
  , conduitClose = return [] }
```

- So is a one-to-one mapper:

```
mapM f = Conduit $ return $ PreparedConduit
  { conduitPush = \i -> do
        fi <- lift $ f i
        return (Producing [fi])
  , conduitClose = return [] }
```

- Available as `map` in `Data.Conduit.List`.

Conduit
Conduits

- But conduits may produce many outputs for a given input:
    - Consider taking a stream of strings and producing a stream of characters.
- Or may require many inputs for a given output.
    - Such as a filter

Conduit
Conduits

- But conduits may produce many outputs for a given input:
    - Consider taking a stream of strings and producing a stream of characters.
- Or may require many inputs for a given output.
    - Such as a filter
- That's why they produce *lists* of output.

Conduit
Conduits

- Even better: conduits can maintain *state*.
- And we have the usual helper functions `conduitState` and `conduitIO`.
- And, of course, the libraries often save us.
- Longer, out-of-slide example: `ConduitSumFile.hs`

Conduit
Buffered Sources

Summary of the world thus far:

- Traditional I/O monad:
  - Open a file.
  - Read a line from the file.
  - Do something to that line and adjust state.
  - Do those for a while.
  - Close the file.
- Conduits:
  - Write a custom sink, source, or conduit.
  - Glue it into a pipeline.
  - Run the pipeline.

Conduit
Buffered Sources

But there's a problem:

- Go back to our easier example:

```
pipe l = sourceList l $$ sinkSum
```

- What if `sinkSum` stops after, say, the sum is ≥ 5?

Conduit
Buffered Sources

But there's a problem:

- Go back to our easier example:

```
pipe l = sourceList l $$ sinkSum
```

- What if `sinkSum` stops after, say, the sum is ≥ 5?
- There might be stuff left in the list!

Conduit
Buffered Sources

But there's a problem:

- Go back to our easier example:

```
pipe l = sourceList l $$ sinkSum
```

- What if `sinkSum` stops after, say, the sum is ≥ 5?
- There might be stuff left in the list!
- Maybe even stuff we care about!

Conduit
Buffered Sources

But there's a problem:

- Go back to our easier example:

```
pipe l = sourceList l $$ sinkSum
```

- What if `sinkSum` stops after, say, the sum is ≥ 5?
- There might be stuff left in the list!
- Maybe even stuff we care about!
- How do we get at it?

Conduit
Buffered Sources

Enter `BufferedSources`.

```
data BufferedSource m a = BufferedSource
  { bsourcePull :: ResourceT m (SourceResult a)
  , bsourceUnpull :: a -> ResourceT m ()
  , bsourceClose :: ResourceT m ()
    }
```

- Just like sources, but now with *unpull*.

Conduit
Buffered Sources

Enter `BufferedSources`.

```
data BufferedSource m a = BufferedSource
  { bsourcePull :: ResourceT m (SourceResult a)
  , bsourceUnpull :: a -> ResourceT m ()
  , bsourceClose :: ResourceT m ()
    }
```

- Just like sources, but now with *unpull*.
- Puts things back so that they will be read next.

Conduit
Buffered Sources

- If we run two pipelines on a buffered source,
- And the first stops early,
- We'll get the left-over data on the second.

```
runResourceT $ do
    bsrc <- bufferSource $ sourceList [1,2,3]
    bsrc $$ drop 2
    x <- bsrc $$ take 1
    print x
```

Next time

- Web development using Yesod.
  - I will send out instructions tonight for bringing the stack up to “hello world” stage.
  - (I will also bring enough power strips to class for everybody to plug in.)
- Intended to be a mixture of lecture and workshop.
  - Rough plan: walk through mechanics of Yesod.
  - Then you guys group up or work alone and I float around answering questions.
- Seem reasonable?

Bib

Simon Meier.
A guided tour through the bytestring library, January 2012.
Available from: http://meiersi.github.com/HaskellerZ/meetups/2012%2001%2019%20-%20The%20bytestring%20library/slides.html.

Bryan O'Sullivan, John Goerzen, and Don Stewart.
Real World Haskell.
O'Reilly Media, Inc., 1st edition, 2008.
Available from: http://book.realworldhaskell.org/.

Michael Snoyman.
Conduits, December 2011.

Available from: http://www.yesodweb.com/blog/2011/12/conduits.

Michael Snoyman.
Conduits, part 5: Buffering, January 2012.
Available from: http://www.yesodweb.com/blog/2012/01/conduits-buffering.