

Stream fusion on Haskell Unicode strings

Tom Harper

<tom.harper@comlab.ox.ac.uk>

Oxford University Computing Laboratory
University of Oxford

AngloHaskell 2008
Friday, 8 August 2008

The String type

Built in Haskell String type:

```
> type String = [Char]
```

The String type

Built in Haskell String type:

```
> type String = [Char]
```

Why we love it:

- ▶ Elegant, logical
- ▶ Allows us to use polymorphic list functions
- ▶ Supports Unicode by using word size (30+ bits) values

The String type

Built in Haskell String type:

```
> type String = [Char]
```

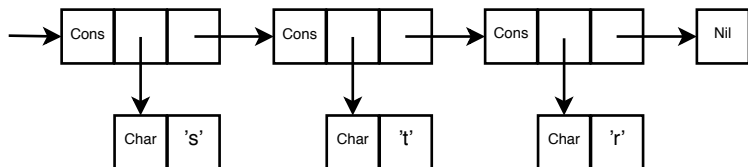
Why we love it:

- ▶ Elegant, logical
- ▶ Allows us to use polymorphic list functions
- ▶ Supports Unicode by using word size (30+ bits) values

BUT:

- ▶ Too slow (compared to C strings)
- ▶ Too big (one Char can take up 20 bytes)

The String type (cont'd)



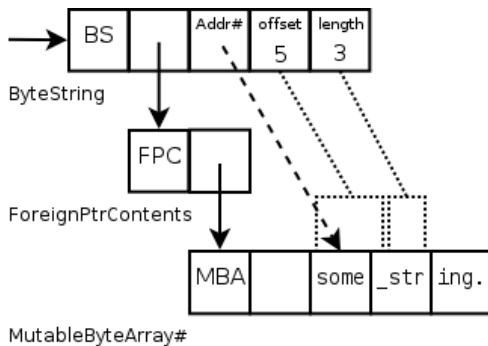
- ▶ Strings are represent as a linked list.
- ▶ Char has a header word and word containing the character itself. (8 bytes)
- ▶ Cons has a header word, a pointer to a Char, and a pointer to the next cell. (12 bytes)
- ▶ 20 bytes for a single character that could be a byte (for ASCII) or at most 21-bits (for a maximally large Unicode character)

What We Want

- ▶ A faster representation.
- ▶ A smaller representation.
- ▶ Keep the functionality.
- ▶ Keep Unicode support!

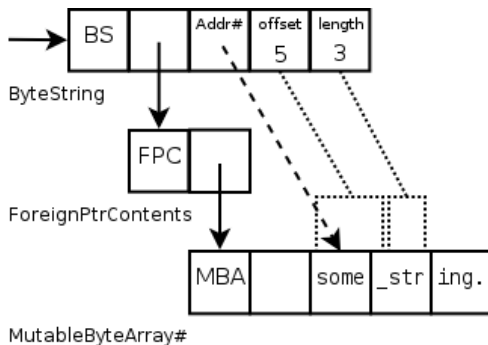
What We've Got

- ▶ An alternative to String exists: ByteString



What We've Got

- ▶ An alternative to String exists: ByteString



- ▶ More compact, each character is one byte.
- ▶ Faster, array based storage.
- ▶ Uses fusion to create an efficient way of creating pipelines for string processing.

What We've Got – Disadvantages

ByteString is great! But:

- ▶ Uses a `ForeignPtr`, which is pinned in memory.

What We've Got – Disadvantages

ByteString is great! But:

- ▶ Uses a `ForeignPtr`, which is pinned in memory.
- ▶ Its storage is fixed-width at one byte per character, which is too small for Unicode support.

The Solution

So, to address the issues that appear in ByteString we need a new representation that

The Solution

So, to address the issues that appear in ByteString we need a new representation that

- ▶ is Unicode aware,

The Solution

So, to address the issues that appear in `ByteString` we need a new representation that

- ▶ is Unicode aware,
- ▶ presents a character (i.e. `Char`)-level interface for manipulating strings,

The Solution

So, to address the issues that appear in `ByteString` we need a new representation that

- ▶ is Unicode aware,
- ▶ presents a character (i.e. `Char`)-level interface for manipulating strings,
- ▶ is more compact than `String` and

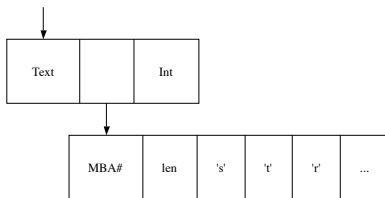
The Solution

So, to address the issues that appear in `ByteString` we need a new representation that

- ▶ is Unicode aware,
- ▶ presents a character (i.e. `Char`)-level interface for manipulating strings,
- ▶ is more compact than `String` and
- ▶ is fast.

Introducing...

```
data Text = Text !(UArray Word16) !Int
```



- ▶ An unboxed array that stores a Unicode stream.
- ▶ The encoding is UTF-16 (for now).
- ▶ The interface comprises list combinators over `Chars`.
- ▶ Nothing magical here...

A Short Introduction to Fusion

```
print . take x . map toUpper . filter isAlphaNum
```

- ▶ This pipeline of functions performs a series of transformations on a string.
- ▶ Unlike a lazy list, each of these functions will allocate an array with its results.
- ▶ In this case, that's three copies of string that are immediately discarded.

A Short Introduction to Fusion (cont'd)

- ▶ What we want is one pass over the data.
- ▶ Techniques for removing intermediate data structures through program transformation is called *fusion* or *deforestation*.

A Short Introduction to Fusion (cont'd)

- ▶ What we want is one pass over the data.
- ▶ Techniques for removing intermediate data structures through program transformation is called *fusion* or *deforestation*.

A simple example is map fusion, where:

```
map f . map g
```

would become:

```
map (f . g)
```

A Short Introduction to Fusion (cont'd)

- ▶ What we want is one pass over the data.
- ▶ Techniques for removing intermediate data structures through program transformation is called *fusion* or *deforestation*.

A simple example is map fusion, where:

```
map f . map g
```

would become:

```
map (f . g)
```

This rule could be specified by using the RULES pragma in GHC as:

```
{-# RULES "map/map fusion" forall f g s.  
  map f (map g s) = map (\x -> f (g x)) s
```

A Short Introduction to Fusion (cont'd)

- ▶ Many general purpose fusion systems exist. For example:
 - ▶ forall k z g. foldr k z (build g) = g k z
 - ▶ forall g f e. destroy g (unfoldr f e) = g f e

Stream Fusion

- ▶ Uses one rewrite rule for all fusion.
- ▶ Can fuse arbitrary sequence types into the same stream type
- ▶ Can define a wide range of fusible functions
- ▶ Fast

The Stream type

Stream fusion works by converting sequences into a stream, processing it, and converting back to your sequence of choice.

```
data Stream = forall s. Stream (s -> Step s a) s !Int
data Step s a = Done
               | Skip !s
               | Yield !a !s
```

The Stream type

Stream fusion works by converting sequences into a stream, processing it, and converting back to your sequence of choice.

```
data Stream    = forall s. Stream (s -> Step s a) s !Int
data Step s a = Done
              | Skip !s
              | Yield !a !s
```

- ▶ A stream comprises a stepper function, a seed, and a length “hint” field for reallocation later.

The Stream type

Stream fusion works by converting sequences into a stream, processing it, and converting back to your sequence of choice.

```
data Stream = forall s. Stream (s -> Step s a) s !Int
data Step s a = Done
               | Skip !s
               | Yield !a !s
```

- ▶ A stream comprises a stepper function, a seed, and a length “hint” field for reallocation later.
- ▶ The stepper function uses the seed to generate one of three results:
 - ▶ Done – end of stream
 - ▶ Skip – skip this element, use the seed to get the next element
 - ▶ Yield – The seed produced an element of the sequence, and the seed for the next element.

Streaming lists

```
stream :: [a] -> Stream a
stream xs = Stream next xs 0
  where
    next (x:xs) = Yield x xs
    next []     = Done
```

Streaming lists

```
stream :: [a] -> Stream a
stream xs = Stream next xs 0
  where
    next (x:xs) = Yield x xs
    next []     = Done

unstream :: Stream a -> [a]
unstream (Stream next s len) = unfold s
  where
    unfold s = case next s of
      Done      -> []
      Skip s'   -> unfold s'
      Yield x s' -> x : unfold s'
```

Some streaming functions: map

```
mapS :: (a -> b) -> Stream a -> Stream b
mapS f (Stream next s len) = Stream next' s0 len
  where
    next' s = case next s of
      Done          -> Done
      Skip s'       -> Skip s'
      Yield x s'    -> Yield (f x) s'
```

Some streaming functions: map

```
mapS :: (a -> b) -> Stream a -> Stream b
mapS f (Stream next s len) = Stream next' s0 len
  where
    next' s = case next s of
      Done          -> Done
      Skip s'       -> Skip s'
      Yield x s'   -> Yield (f x) s'

map :: (a -> b) -> [a] -> [b]
map f = unstream . mapS f . stream
```

Some streaming functions: filter

```
filterS :: (a -> Bool) -> Stream a -> Stream a
filterS p (Stream next s len) = Stream next' s0 len
  where
    next' s = case next s of
      Done                -> Done
      Skip s'              -> Skip s'
      Yield x s' | p x     -> Yield x s'
                  | otherwise -> Skip s'
```

Some streaming functions: filter

```
filterS :: (a -> Bool) -> Stream a -> Stream a
filterS p (Stream next s len) = Stream next' s0 len
  where
    next' s = case next s of
      Done                -> Done
      Skip s'              -> Skip s'
      Yield x s' | p x     -> Yield x s'
                  | otherwise -> Skip s'

filter :: (a -> Bool) -> [a] -> [a]
filter p = unstream . filterS p . stream
```

The fusion rule

```
filter p . map f
```

is equivalent to

```
unstream . filterS p . stream . unstream . mapS f . stream
```

It's a waste to convert back to a list just to stream again, so we use the following fusion rule:

```
forall s. stream (unstream s) = s
```

Applying this gives us

```
unstream . filterS p . mapS f . stream
```

Streaming lists vs. Streaming Text

Lists are fine, but what happened to Text?

Streaming lists vs. Streaming Text

Lists are fine, but what happened to Text?

- ▶ The combinator functions stay the same, but they are restricted to `Stream Char`, and most polymorphic type variables also become `Char`.

Streaming lists vs. Streaming Text

Lists are fine, but what happened to Text?

- ▶ The combinator functions stay the same, but they are restricted to `Stream Char`, and most polymorphic type variables also become `Char`.
- ▶ The rewrite rule is the same, we still use streams the same way.

Streaming Text

The differences:

- ▶ The combinators don't care about the underlying representation, they just want a `Stream Char`
- ▶ This is `stream`'s job, so `stream :: Text -> Stream Char`
- ▶ Similarly, `unstream :: Stream Char -> Text`
- ▶ This means `stream` and `unstream` do the Unicode encoding/decoding.

Streaming Text (cont'd)

```
stream :: Text -> Stream Char
stream (Text arr len) = Stream next 0 len
  where
    {-# INLINE next #-}
    next i
      | i >= len = Done
      | n >= 0xD800 && n <= 0xDBFF =
          Yield (U16.chr2 n n2) (i + 2)
      | otherwise = Yield (unsafeChr n) (i + 1)
    where
      n  = unsafeAt arr i
      n2 = unsafeAt arr (i + 1)
    {-# INLINE [0] stream #-}
```

Unstreaming Text

- ▶ Unstreaming is not as straightforward.
- ▶ Have to worry about allocating a properly sized array, use length hint to make a good guess.
- ▶ Characters are encoded in either one or two Word16s
- ▶ Uses a STUArray that is written and then frozen to yield a UArray

File I/O

- ▶ File I/O is not fully implemented yet, but a version of `readFile` is used to read in files for testing.

File I/O

- ▶ File I/O is not fully implemented yet, but a version of `readFile` is used to read in files for testing.
- ▶ `readFile` uses a `ByteString` to read in the file, and then a stream function performs full Unicode validation (for any of the standards).

```
readFile :: Encoding -> FilePath -> IO Text
readFile enc f = do
    bs <- BS.readFile f
    return (unstream (bsStream bs enc))
```

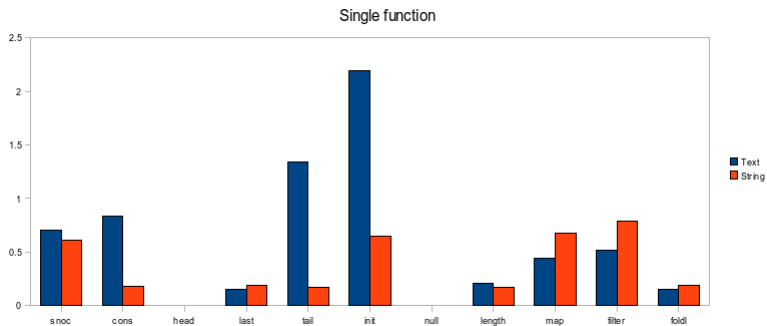
File I/O

- ▶ File I/O is not fully implemented yet, but a version of `readFile` is used to read in files for testing.
- ▶ `readFile` uses a `ByteString` to read in the file, and then a stream function performs full Unicode validation (for any of the standards).

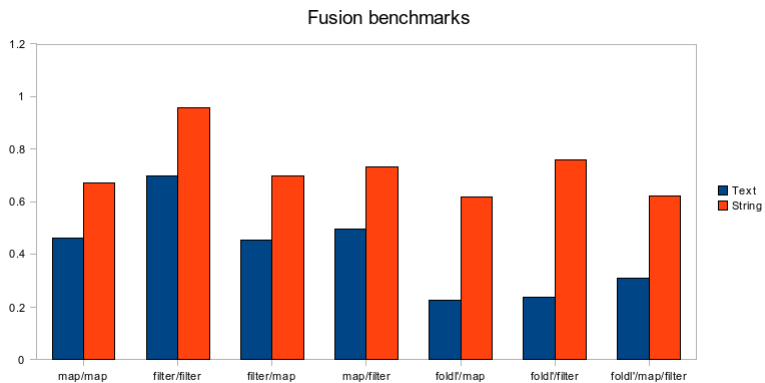
```
readFile :: Encoding -> FilePath -> IO Text
readFile enc f = do
    bs <- BS.readFile f
    return (unstream (bsStream bs enc))
```

- ▶ This also makes the function fully fusible. If a consumer consumes the string before it is unstreamed, it is never written as `Text`.

Results



Results



Further Work

- ▶ Add offset fields, use manipulation of length of offset fields to get “free” versions of `tail` and `init`.
- ▶ Testing
 - ▶ Comparing UTF-8/UTF-16 performance
 - ▶ Testing on small words as well as large (order of 100MB) strings
 - ▶ Comparing results for corpora with large Unicode code points (and also merely ASCII text)
 - ▶ Full file I/O

Questions?

Questions? Comments? Suggestions?