Stream Fusion on Haskell Unicode Strings

Thomas Harper[1]

Oxford University Computing Laboratory
Oxford, United Kingdom

Abstract. Prior papers have presented a fusion framework called stream fusion for removing intermediate data structures from both lists and arrays in Haskell. Stream fusion is unique in using an explicit datatype to accomplish fusion. We demonstrate how this can be exploited in the creation of a new Haskell string representation *Text*, which achieves better performance and data density than *String*. *Text* uses streams not only to accomplish fusion, but also as a way to abstract away from various underlying representations. This allows the same set of combinators to manipulate Unicode text that is stored in a variety of ways.

1 Introduction

Lists are the primary workhorse data structure of functional programming. In the programming language Haskell[1], strings are represented using the built-in list type. This allows programmers to use standard polymorphic list combinators to build complex string manipulation functions in the same way that they would manipulate other lists. Haskell programmers often take advantage of this by composing list functions to form "pipelines" for transforming lists (and strings). For example:

$$\mathit{return} \cdot \mathit{words} \cdot \mathit{map}\ \mathit{toUpper} \cdot \mathit{filter}\ \mathit{isAlpha} \lll \mathit{readFile}\ f$$

This program reads in a file, filters out the non-alphabetic characters, converts the remaining characters to uppercase, and then tokenises them. It exemplifies Haskell's ability to create concise yet powerful programs through the composition of modular functions. It is also, however, extremely inefficient. Haskell's *String*s are much larger than, for example, their C counterparts. To address these inefficiencies, there is an alternative to *String* in the Haskell core libraries called *ByteString*. *ByteString* addresses *String*'s inefficiencies and achieves greater performance, but it does so at the cost of support for non-ASCII characters. We are introducing a new data type, *Text*, to fill in the gap left between these two approaches. *Text* addresses the performance issues associated with *String* while maintaining Unicode support.

The *Text* type is an array-based string representation that is faster and more compact than *String*. Its API is based on Haskell's list library, which means that it can be used as a drop-in replacement for *String*. Figure 1 shows the speed-up achieved by using *Text*s to run the example program from above. The main contribution of this paper is a faster, more compact string representation

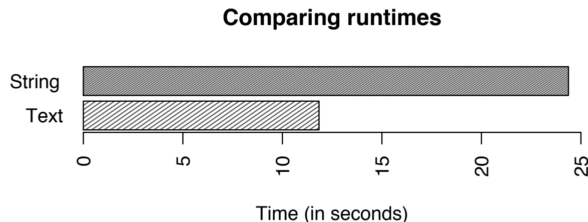

Fig. 1. Sample comparison of *Text* and *String* runtimes

for Haskell that incorporates Unicode support. We implement Haskell's list API over *Text*s. Like *ByteString*, we use *stream fusion* to remove intermediate data structures. Our use of stream fusion demonstrates how to exploit it in a novel way. Our API implementation uses the *Stream* type as an abstraction over more complex underlying representations. As we will show, this is an important aspect of how we implemented the API for *Text*.

The rest of the paper is organised as follows: Section 2 provides some important background information. It discusses Haskell's *String* type and its advantages and disadvantages. It presents some information on the Unicode standard and Unicode encoding standards. It also provides a short introduction to stream fusion. Section 3 discusses the internal structure of the *Text* datatype and how it addresses the inefficiencies of *String*. Section 4 describes the API for *Text* and how it uses stream fusion in its implementation. Section 5 presents and discusses some benchmarks of *Text* in comparison with both *String* and *ByteString*. Section 6 discusses some of the related fusion and string alternative efforts. Section 7 presents our conclusions and proposals for further work.

2 Background

2.1 The *String* and *ByteString* types

Haskell defines *String* using the built-in list and *Char* types:

type *String* = [*Char*]

There are many benefits to this design. It is consistent with the notion that a string is a list of individual characters and allows us to manipulate them as we would other lists. Haskell's polymorphic list library contains functions that encapsulate a variety of common recursion patterns. Programmers can compose these functions to create modular programs where each of the individual components can be reused. *String*s are also Unicode-compliant. This elegance, however, comes with a price. Figure 2 shows the structure of *String*. Using the Haskell list type as the basis for *String* means that it uses a series of *Cons* cells to store values. In the case of *String*, each of these *Cons* cells points to both a heap-allocated *Char* and then the next *Cons* cell in the list. Each heap-allocated object has a

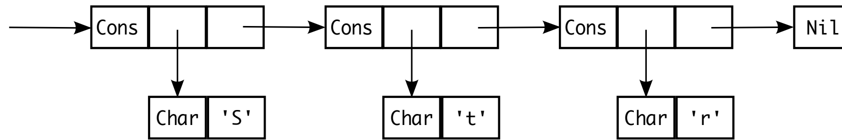

Fig. 2. The low-level structure of a *String*

word-sized (either 32-bit or 64-bit) header. All pointers are also word-sized. Each *Char* is also word-sized. This amounts to a string representation that requires 20 bytes per character on a 32-bit system where only 8 to 32 bits are needed [2].[1]

Coutts et al. [3] created a new string type, *ByteString*, which offers better density and performance than *String*. In order to overcome *String*'s memory and performance drawbacks, *ByteString* uses a strict, array-based representation. The underlying structure of this representation is shown in Figure 3. This

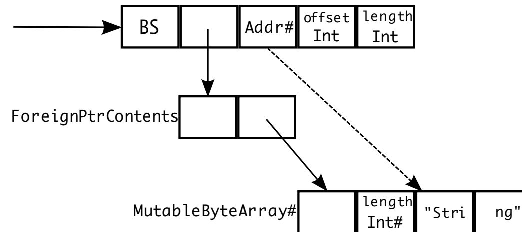

Fig. 3. The low-level structure of a *ByteString*

array structure eliminates the need for memory-consuming pointers and headers for each character. It only uses a 8 bits for each character instead of *String*'s 32 bits. This compactness, while desirable, also sacrifices support for Unicode characters, leaving a rather large gap between the performance of ASCII text and Unicode text in Haskell. As a strict data structure, the creation of intermediate *ByteString*s would have a severe impact on performance. To address this problem, the authors of *ByteString* introduced a new fusion technique which they called *stream fusion* [4]. This technique, which we also employ, is explained briefly in Section 2.3.

2.2 Unicode

Unicode is a world standard for representing text in nearly all of the world's modern and historical writing systems [5], including, for example, Cyrillic, Arabic, and East Asian scripts. The Unicode Standard also specifies three encoding systems for Unicode code points: UTF-8, UTF-16, and UTF-32. Of these, UTF-8 and UTF-16 are *variable-width*, and UTF-32 is *fixed-width*. For variable-width

[1] For ASCII characters, each repeated character only requires an additional 12 bytes because GHC pre-allocates and shares ASCII characters.

encodings, decoding a Unicode stream into Unicode code points involves some binary arithmetic.

UTF-8 is a byte-oriented encoding in which each code point requires between one and four bytes depending on its value (the larger the code point value, the more bytes required). This encoding is the most compact, using only one byte for the smallest code points. It is also backward compatible with ASCII (that is, a UTF-8 document consisting only of those characters in ASCII will be exactly the same). The price for UTF-8's compactness is increased overhead in decoding. Outside of traditional ASCII characters (i.e. any code point above U+7F[2]), it is necessary to reconstruct code points from two or more bytes for many commonly used characters (e.g. every character in the Cyrillic alphabet or Arabic alphabets).

UTF-16 is a 16-bit-word-oriented encoding. Characters are encoded using one or two words. Two adjacent words that are used to represent a character together are a *surrogate pair*. In comparison with UTF-8, UTF-16 is less compact; the minimum space required for a code point is 16 bits rather than 8. What this costs in space, however, is gained in efficiency. The range of code points that fit into one UTF-16 word is U+0000 to U+FFFF. This range is known as the Basic Multilingual Plane, and contains all writing systems currently in use around the world, as well as some historic ones.[3]. This means that surrogate pairs occur rarely in modern language documents. Those code points that are stored in only one 16-bit word are stored as raw values; no arithmetic is required to decode them.

Of the three encodings, UTF-32 is the simplest. It represents each code point as a 32-bit number. A code point requires at most 21-bits to represent its value, so UTF-32 can represent any of them without any splitting or arithmetic. While this is the most straightforward implementation of Unicode, it is also the most inefficient in terms of space. All common (and even many obscure) code points take up two to four times as much space as necessary.

Although UTF-8 is the most compact encoding, *Text* uses UTF-16 due to its much lower overhead. In the case of ASCII characters, this does represent a greater use of space that is strictly necessary. However, given the fact that UTF-8 also requires 16 bits for any non-Basic Latin character, UTF-16 represents a similar solution with far less overhead, especially for non-Latin-based scripts.

2.3 Stream fusion

Stream fusion is a technique for removing intermediate data structures that appear from the composition of recursive functions. These data structures are created when one function passes data on to the next. They are then discarded, waiting to be garbage collected. Depending on the situation, this can have a

[2] Unicode code points are usually represented as "U+" prefixed to the hexadecimal form of the number

[3] The exception to this is about 40,000 of the "Han Unification" characters used for East Asian languages. These are rarely used or obscure.

significant impact on performance. Although other techniques have been implemented in Haskell[6, 7], the use of stream fusion for *ByteString* has demonstrated that it is well-suited to fusing arrays. In contrast, other common fusion systems are generally adapted for fusing lists. Here, we give an overview of the stream fusion material published by Coutts et al., whose papers provide a more in-depth explanation of the technique.

Stream fusion differs from other fusion strategies in using an explicit data type:

$$\textbf{data}\ \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \rightarrow \mathit{Step}\ s\ a)\ s$$
$$\textbf{data}\ \mathit{Step}\ s\ a = \mathit{Done} \mid \mathit{Yield}\ a\ s \mid \mathit{Skip}\ s$$

A *Stream* is a co-recursive form of a list, where each element of the list may be yielded one at a time. It contains a stepper function, which is used to unfold the *Stream*, and an initial seed to pass to the stepper function. The results of the stepper function cover three possibilities. The *Yield* constructor produces an element and a new seed. The *Done* constructor signals the end of the list. The case of *Skip* allows a new seed to be produced without yielding a value. This is important in allowing us to define functions that have potentially non-productive steps (e.g. *filter*). The central idea of stream fusion is that instead of recursive functions over data structures, we can define non-recursive ones over streams. We can then convert our data structures to and from streams to achieve the desired transformation.

Data structures are converted to and from streams by the functions *stream* and *unstream*. The *stream* function converts a data structure into a *Stream*. It creates a new *Stream* with a stepper function that yields successive values of the data structure until its end, finally yielding *Done*. For example, the *stream* function over lists is

```
stream :: [a] → Stream a
stream s0 = Stream next s0
  where
    next []       = Done
    next (x : xs) = Yield x xs ·
```

The way to convert back to our original structure is with *unstream*. This function actually applies a *Stream*'s stepper function recursively to each successive seed until it encounters *Done*. For lists, the *unstream* function is

```
unstream :: Stream a → [a]
unstream (Stream next s0) = unfold s0
  where unfold s = case next s of
    Done       → []
    Skip s′    → unfold s′
    Yield x xs → x : unfold xs ·
```

Functions over streams transform it by modifying the definition of its stepper function. This is done by defining a new function that calls the original stepper

```
mapS :: (a → b) → Stream a → Stream b
mapS (Stream next s0) = Stream next′ s0
  where
    next′ s = case next s of
      Done       → Done
      Skip s′    → Skip s′
      Yield a s′ → Yield (f a) s′

filterS :: (a → Bool) → Stream a → Stream a
filterS p (Stream next s0) = Stream next′ s0
  where
    next′ s = case next s of
      Done                     → Done
      Skip s′                  → Skip s′
      Yield a s′ | p x         → Yield a s′
                 | otherwise   → Skip s′
```

Fig. 4. Some examples of stream functions

function and pattern matching on each of the three *Step* constructors. Figure 4 shows the stream version of some common list functions. The function *filterS* is particularly notable in demonstrating the use of the *Skip* constructor. In the case where the predicate p is not satisfied the *Yield* is replaced with a *Skip* which discards the value and only keeps the seed. This allows the stepper function to be productive without yielding an element we wish to discard.

It is important to note that stepper functions are non-recursive. When a stream is unfolded, *all* the transformations are applied to a yielded element before producing a new one. This merges what would be several recursive traversals over a structure into a single recursive unfold of a stream. In general, fusible functions on streamable data structures are defined in terms of their analogous stream transformers, *stream*, and *unstream*. A fusible function defined in terms of streams tends to have the following structure:

$$f\ x = unstream \cdot fS \cdot stream$$

where fS is the stream transformation analogue of f. Functions that only consume streams have no *unstream*, and functions that only produce streams will have no *stream*. When two fusible functions f and g are composed as $f \cdot g$, they can be inlined to form

$$unstream \cdot fS \cdot stream \cdot unstream \cdot gS \cdot stream$$

In the middle of this function, there is an instance of unfolding a stream, only to create a new one. Eliminating occurrences of $stream \cdot unstream$ yields the program:

$$unstream \cdot fS \cdot gS \cdot stream$$

Removing this portion of the program produces a new program that is equivalent. The occurrence of $stream \cdot unstream$ would have created an intermediate

data structure, only to convert it to new stream. Instead, the original stream is transformed twice and then unfolded. To accomplish this fusion automatically, we specify the following rewrite rule:

$\langle$**stream/unstream fusion**$\rangle$ $\forall s.\ \mathit{stream}\ (\mathit{unstream}\ s) \mapsto s$

This rule can be specified in GHC using compiler directives [8], which allows us to apply it automatically during compilation.

3 The *Text* Data Type

The first step in creating *Text* is to design its underlying datatype. The purpose of the *Text* datatype is to store Unicode text more efficiently than *String*. Having seen the effectiveness of array-based storage, we use this approach in *Text*. Using an array removes the numerous pointers and *Cons* cells that account for so much of the space consumed by a *String*. We can also remove the pointers to elements, and their associated headers, by using an array of unboxed elements. Finally, we can decrease the minimum size of a character by utilising a Unicode encoding instead of representing them as raw code points. In this case, we chose UTF-16. The reason for this decision is not arbitrary; UTF-16 achieved better performance in benchmarks that dealt with large amounts of non-ASCII test, most likely because of its simpler arithmetic [9]. We consider such a case to be a major use of this library and therefore an important factor.

The result is the following definition of *Text*:

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

Text is an unboxed array of 16-bit words indexed by integers. The other integers are offset and length fields. These fields allow "free" creation of substrings merely by modifying these fields and pointing to the original string's array. The exclamation points are strictness annotations which prevent these fields from being calculated lazily; since the underlying array is strict anyway, any laziness will only introduce inefficiencies. Figure 5 shows the underlying structure of *Text*. The use of *Word16* reflects our use of UTF-16, which is based on converting 1

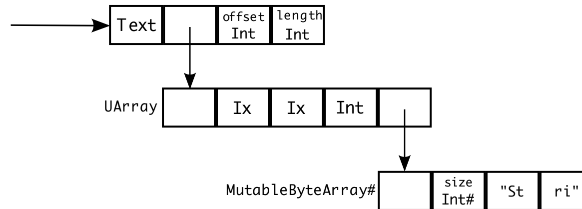

Fig. 5. The low-level structure of a *Text*

or 2 16-bit words into a Unicode code point. Even in the worst case, *Text* still only requires 32 bits instead of *Strings* 20 bytes.

The switch from a list-based representation to an array-based one alters the complexity of some fundamental string operations. Some operations are now

faster and consume less memory. Indexing is now a constant time operation, allowing for easy reads in the middle of the string. Because of the length and offset fields, operations involving substring creation (e.g. *take* and *drop*) do not require any additional space. Functions that construct strings, however, require more resources than before. Both *cons* and *concat* require all of its inputs to be copied into a new array. This shifts *cons* from a constant time operation to linear one, and *concat* has gone from being linear in the length of its first argument to linear in the length of *both* arguments combined. The impact of these design decisions is also measured in Section 5.

4 Fusion and the *Text* API

In Section 2, we introduced the *Stream* datatype, along with the associated functions *stream* and *unstream*. These functions allowed us to use transformers over *Stream* as transformers over a target data structure. The use of *stream* and *unstream* allows us to syntactically identify and remove intermediate data structures from a program automatically.

In order to take advantage of stream fusion, all we need to do is implement *Text* versions of *stream* and *unstream*. The *stream* function must define a stepper function and seed that will traverse an array. Slightly more complicated is *unstream*, which needs to unfold the stream and place the yielded elements in an array. This entails allocating an array of the appropriate size. The initial solution to this problem was to start with a small array, allocating a new array that was double the size of the original when needed. The cost of copying, however, quickly overtook any performance gains. Instead, we added a strict *Int* field to *Stream*. This field holds the length of the array from which the stream was created, allowing the *unstream* to make a good guess of what the string size will be. Copying can still be performed if necessary but is avoided in most cases. Functions that modify the length of a string (e.g. *cons*,*concat*,*take*,*drop*) can modify this length field.

The crucial question in streaming *Text*s is how to manipulate them. In prior implementations of stream fusion, the elements of the underlying data structure are simply turned into a *Stream* of the same elements. This logic would lead us to converting *Text*s to *Stream Word16*s. This would mean that programmers would have to deal with encoding and decoding UTF-16 values themselves, which is highly undesirable. The idea is to abstract away from the underlying representation in our API, thus letting programmers deal with *Char*s. Therefore, we need to implement *stream* and *unstream* so that they not only create a *Stream*, but decode and re-encode UTF-16 values.

The first function, *stream*, decodes a UTF-16 array and creates a *Stream* from the result. This function is show in Figure 6. This function creates a *Stream Char* whose stepper function *next* both decodes and streams elements of the array. The seeds for this function are indices of the array. This implementation makes some important assumptions about the input *Text*. First, it assumes that all elements are *valid* UTF-16 values. This eliminates the need to perform certain

```
stream :: Text → Stream Char
stream (Text arr off len) = Stream next off len
  where
    end = off + len
    next !i
      | i ⩾ end                        = Done
      | n ⩾ 0xD800 ∧ n ⩽ 0xDBFF = Yield (chr2 n n2) (i + 2)
      | otherwise                      = Yield (unsafeChr n) (i + 1)
      where
        n  = unsafeAt arr i
        n2 = unsafeAt arr (i + 1)
```

Fig. 6. The *stream* function for *Text*

bounds checks. The only condition we check about the elements is whether or not they are the beginning of a surrogate pair. If they are, it is assumed the following element is a valid second member of a surrogate pair. Finally, we use *unsafeAt* and *unsafeChr*, which assume that the values they are given are valid indices and character values, respectively. Making these assumptions cuts down significantly on the number of bounds checks we need to perform. For large strings, doing these for ever character has a significant impact on performance. We allow ourselves to make these assumptions by assuming that all *Text*s are always valid UTF-16 streams. We can do this because we do full Unicode checks when *Text*s are created from other sources, and then control the manipulation of *Text*s through our API.

The *unstream* function needs to convert a *Char* back into its UTF-16 equivalent. This is shown in Figure 7. This function allocates an array based upon length information given in the *Stream*. It then converts each character into one or two *Word16*s according to the UTF-16 standard. We again make the assumption of safety of our characters and do not perform full Unicode bounds checking. We can assume this rather safely because, unless the programmer is doing something tricky, *Char* will not contain an invalid Unicode code point. Furthermore, we manually track the bounds of the allocated array so that bounds checking does not need to be done with every write by the array API functions (hence *unsafeWrite*).

Together, these two functions perform all of the Unicode encoding and decoding necessary for string manipulation. This decision makes it easy to find and eliminate bottlenecks in our code. All of the encoding overhead is concentrated in only these two functions. Because stream fusion requires writing transformations over an explicit datatype, the usual stream transformers will work with a streamed *Text*. The only modifications we make are to restrict the type of certain functions (elements of a *Text* are always characters, and there are no nested *Text*s) and modify our length field where appropriate.

Placing our decoding and encoding functionality in the stream conversion functions has another benefit: the stream fusion rewrite rule removes interme-

```
unstream :: Stream Char → Text
unstream (Stream next0 s0 len) = x `seq` Text (fst x) 0 (snd x)
  where
    x :: ((UArray Int Word16), Int)
    x = runST ((unsafeNewArray_ (0, len + 1) :: ST s (STUArray s Int Word16))
      >>= (λarr → loop arr 0 (len + 1) s0))
    loop arr !i !max !s
      | i + 1 > max = do
        arr' ← unsafeNewArray_ (0, max ∗ 2)
        case next0 s of
          Done → liftM2 (,) (unsafeFreezeSTUArray arr) (return i)
          _       → copy arr arr' >> loop arr' i (max ∗ 2) s
      | otherwise = case next0 s of
        Done   → liftM2 (,) (unsafeFreezeSTUArray arr) (return i)
        Skip s' → loop arr i max s'
        Yield x s'
          | n < 0x10000 → do
            unsafeWrite arr i (fromIntegral n :: Word16)
            loop arr (i + 1) max s'
          | otherwise → do
            unsafeWrite arr i l
            unsafeWrite arr (i + 1) r
            loop arr (i + 2) max s'
          where
            n = ord x; m = n − 0x10000
            l = fromIntegral ((shiftR m 10) + (0xD800))
            r = fromIntegral ((m .&. (0x3FF)) + (0xDC00))
```

Fig. 7. The *unstream* function for *Text*

diate data structures *and* reduces the number of decodings/encodings that take place within a program automatically. This concept is key to the abstraction that we wish to achieve from the underlying data structure. Once we have a definition for *stream* and *unstream*, we no longer care about the data representation that is being used in *Text* when writing transformations. Regardless of the encoding of the characters, or the structure of the underlying sequence, the string manipulation functions are identical.

This abstraction has useful implications. Suppose that we want to use a function from some other library that only gives us *String*s. For our *Text* library to work, we have to convert it first. For this, we use the *pack* function.

```
pack :: String → Text
pack str = (unstream (stream_list str))
  where
    stream_list s0 = Stream next s0 (length xs)
      where
        next []       = Done
        next (x : xs) = Yield x xs
```

The *pack* function allows us to convert a *String* into a *Text* using streams. It uses its own *stream_list* function, which is actually just the list version of *stream*. We then use *unstream*, which has already been heavily optimised, to write out the stream to a *Text*. Using streams here doesn't just make the function concise, though. If we transform a *Text* created using *pack*, we can fuse any intermediate *Text*s that are created. This means that in such a pipeline, only the final output *Text* is created, even though the input to the pipeline is a *String*.

Another example of where this is exploited is in file I/O. One of the benefits of *ByteString* is its extremely fast file I/O. To take advantage of this, we implement stream fusion over *ByteString*s as well. Unlike the original implementation of *ByteString* fusion, though, we don't treat *ByteString* as an array of characters but rather as an array of bytes. We then implement the functions *encode* and *decode*, which read to and write from *ByteString*s. Again, we do so using streams so that conversions do not result in unnecessary *Text*s being written. In addition, we implement the arithmetic to encode/decode *all* possible Unicode encoding standards. This is an example of where Unicode validation takes place when creating *Text*s . Because it involves reading from an external source, we must also checks to make sure all the Unicode characters are valid and insert fallback characters if necessary.

The use of stream fusion also allows for easy expansion of the API, for example by an end-user of the library. As described in Section 2.3, transformations are composed of *stream*, *unstream*, and a stream transformer. Programmers can easily define their own stream transformers and compose them with *stream* and/or *unstream* as necessary to define their own fusible functions. The fusion can be applied to them as it is to any pre-defined function.

While stream fusion is an extremely useful abstraction, there are some instances where exploiting the low-level data structure can achieve better performance. For example, the length and offset fields allow us to create a version of *tail* that does not require copying:

```
unfused_tail :: Text → Text
unfused_tail (Text arr off len)
   | len ⩽ 0           = errorEmptyList "tail"
   | n ⩾ 0 xD800 ∧
     n ⩽ 0 xDBFF = Text arr (off + 2) (len − 2)
   | otherwise       = Text arr (off + 1) (len − 1)
  where
     n = unsafeAt arr off
```

By checking whether the first character is a surrogate pair or not, we can decide whether to move the offset by one or two characters and have a *Text* with the desired contents without the copying of the *Stream* version. It isn't fusible, though. We would prefer a way to automatically choose the best version of *tail* for a given situation. This can be accomplished with the following rule:

$$\langle \textbf{tail/unfused} \rangle \quad \forall t.\ \mathit{unstream}\ (\mathit{tailS}\ (\mathit{stream}\ t)) \mapsto \mathit{unfused_tail}\ t$$

This rule states that any occurrence of *tailS* that is not fused with another stream transformer should be converted to our low-level definition of *unfused_tail*.

If a call to *tail* were fusible, it would not directly follow a call to *stream* because stream fusion would have removed this call. Now, we have GHC choosing the most appropriate function for us. This technique is useful for a variety of functions where totally decoding or copying the string is not strictly necessary, such as *init*, *last*, and *append*.

5 Performance

As a more compact and more efficient version of *String*, it is expected that *Text* should be much faster than *String*. This is generally the case. Compared to *String*, *Text* usually achieves much better performance. In comparison with *ByteString*, the extra overhead of Unicode encoding and decoding means that, for ASCII text, *ByteString* is still faster (and, at 8-bits per character, more compact). Figure 8 shows the runtimes for each of the three string representations for some common functions. Although both *Text* and *ByteString* generally

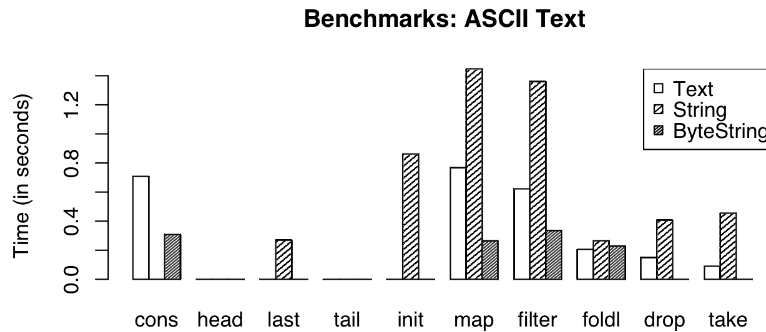

Fig. 8. Benchmarks: ASCII Text

outperform *String*, the runtimes of the *cons* function exhibit some of inherent disadvantages of using an array based string representation. Unlike lists, implementations of *cons* over *Text*s require copying the source array completely, resulting in a linear time operation instead of a list's constant time one. Similar functions, such as *append*, suffer from similar problems.

Figures 9 and 10 show the performance of *String* versus *Text* for different sets of Unicode text. Figure 9 shows benchmarks for Unicode text in the Basic Multilingual Plane. The performance figures are very similar to the ASCII ones. This is to be expected, as ASCII text is treated the same as other low-numbered Unicode code points in UTF-16.

Figure 10 shows the performance of *Text* and *String* with text solely from the Supplementary Multilingual Plane (SMP). The SMP consists of more rarely used characters such as musical and mathematical notations. In general, a document will very rarely contain more than a relatively small number of these characters. This benchmark represents a "worst case" of dealing with a document that comprises exclusively such characters. This has a critical impact on

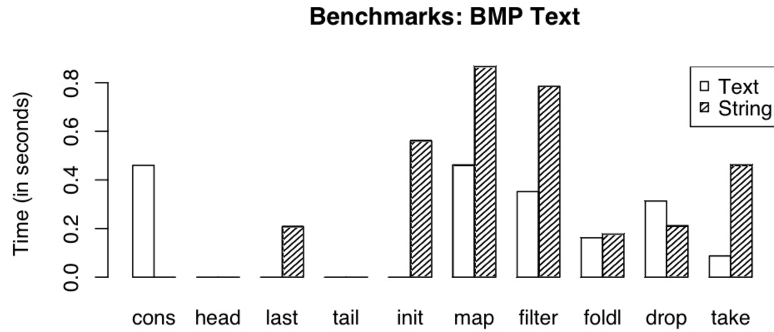

Fig. 9. Benchmarks: BMP Text

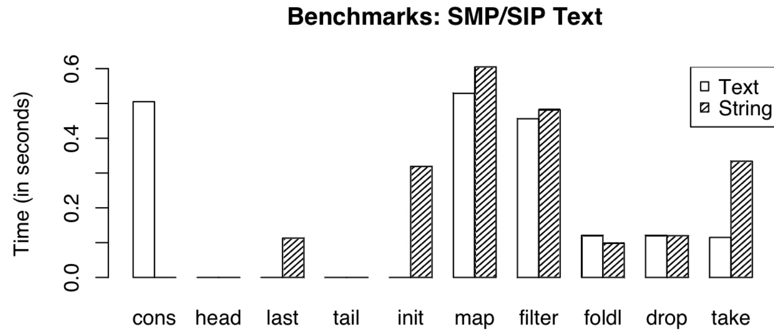

Fig. 10. Benchmarks: SMP Text

performance because all SMP characters require two UTF-16 code points and must be assembled and disassembled when being streamed and unstreamed. In this case, *Text* slows down nearly to the performance of *String*, although still outperforms it in most cases. This shows that, in a worst case scenario, *Text* still scales well and can outperform *String*.

The benchmarks above show that *Text* outperforms *String* in single transformations, but fusion is an important aspect of this library. Figure 11 shows the performance of *Text* versus *String* in a variety of common fusion patterns. These benchmarks compare *Text* using stream fusion with *String* using *foldr/build* fusion. The figures show that *Text* significantly outperforms *String* in these situations. This figure is perhaps the most crucial, as string manipulation functions are more likely to be pipelined than called singly.

These benchmarks reveal that *Text* usually outperforms *String*, but that the low-level differences between the two must be considered when using *Text*. The inherent differences between arrays and lists makes some operations differ with

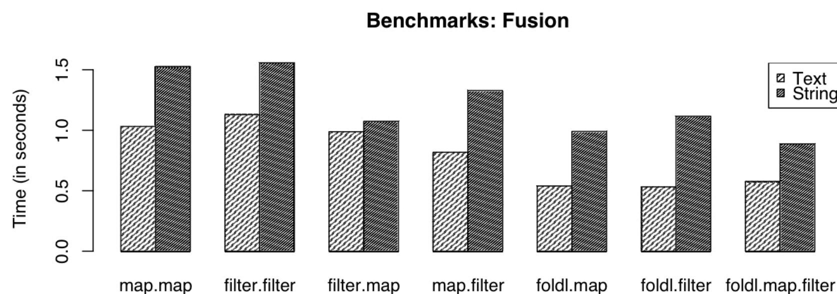

Fig. 11. Fusion Benchmarks

respect to complexity. This performance can sometimes be regained by fusion. The *Stream* version of these slower functions can easily fuse with other functions, in which case there is very little *extra* cost to performing them because the *Text* is already being copied anyway, for example, in $cons\ x \cdot map\ f\ xs$, *cons* would not have the same impact on performance that it does in isolation.

Text is therefore most useful for manipulating strings, for example from user input or a file. Its performance becomes less desirable when constructing strings through concatenation and similar operations.

Another consideration is the strictness of *Text*. In programs that use file I/O, a lazy data structure will only read data into memory as necessary. Consider the example program in Section 1. Although *Text*'s runtime was better, the maximal memory consumption was actually much smaller in *String*. This is because, in *String*, new data only be read in by *readFile* as it was consumed by *foldl*, but in *Text* the entire file is read in, and then consumed. For more discussion of lazy *Text*s, see Section 7.

6 Related Work

Although stream fusion was chosen as the fusion framework for this particular library, there are other related fusion frameworks that have been implemented in Haskell:

foldr/build The *foldr/build* fusion framework is currently used in GHC for fusing lists [6]. It uses *foldr* with its traditional definition to consume lists and a function called *build* to produce them. This makes some functions, such as *filter*, much more straightforward to implement. It cannot, however, fuse zips.

destroy/unfoldr The *destroy/unfoldr* fusion system [7] is the most similar to stream fusion. Like stream fusion, *destroy/unfoldr* uses the notion of co-data to produce a list, but using the well-known list function *unfoldr*. Compared

with stream fusion, *destroy/unfoldr* has similar fusion capabilities, being able to express left folds and zips. However, unlike stream fusion, it requires recursion for *filter*-like functions, which affect the compiler's ability to perform certain optimisations and can drastically impact performance.

Neither of these frameworks uses an explicit data type. This unique characteristic of stream fusion is what makes it such a desirable candidate for our library, because we can clearly separate the conversion between *Char*s and *Word16*s using the *Stream* datatype.

Stream fusion also already appears in the Haskell library *ByteString* [3]. As previously mentioned, it has helped *ByteString* achieve (and sometimes beat) the performance of similar programs written in C. *ByteString* also uses a *ForeignPtr* for it's underlying data structure, making it accessible to code in other languages (notably C). There is also an implementation of stream fusion over Haskell lists [4].

7 Conclusions

Stream fusion is already a known and successful fusion framework, but we have exploited a useful aspect of it in the creation of *Text*. By treating stream fusion's *Stream* type as an abstraction from underlying representations, we have used the stream fusion framework as tool for designing a library over strings that is independent of various low-level representations. In doing so, stream fusion not only prevents the creation of intermediate *Text*s in a string transformation pipeline, it also prevents unnecessary conversion between different encodings and data structures.

There is still plenty of work being done for *Text*. Since the completion of this project, Bryan O'Sullivan has continued to maintain and refine *Text*. His major contributions to the project have been to replace *UArray* with a lower level (and thus faster and smaller) data type and to create a lazy version of *Text*. He has done this by using a list of array chunks. He has also achieved 93% QuickCheck [10] coverage of API functions, weeding out several subtle issues. He has also shifted certain list API functions so that integer-based indexing is less used (e.g. instead of returning the index of the first occurrence of `'c'`, split at the first occurrence of `'c'`), which is more efficient. The most recent version of *Text* is available in the Hackage database.

The performance of *Text* is generally better than that of *String*, and provides a fast way to transformation Unicode text in a functional style. It's current, array-based representation limits its flexibility in creating strings efficiently, and so while the API abstracts away from its low-level representation, this still must be considered when using *Text* to achieve the best performance. However, it may be possible to use more sophisticated persistent data structures to create a more versatile *Text* with even better performance.

Text's array-based structure also has some drawbacks with respect to persistence. Currently, substrings can be created by modifying the length and offset fields of an existing *Text*. While this has the benefit of allowing constant time

substring creation, it does not take into account the size of the substring relative to its parent string. A very small substring may keep a much larger string from garbage collection. We also have many operations with undesirable complexities for building strings. We wish to investigate the possibility of using another data structure for the underlying representation of *Text*. A possible candidate is to use finger trees [11] of arrays to create a fusible rope [12]. This would make the library better suited to manipulating *and* building strings.

References

1. Peyton-Jones, S.: The Haskell 98 Report (2002)
2. The GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.1
3. Coutts, D., Stewart, D., Leshchinskiy, R.: Rewriting Haskell Strings. In: PADL '07. LNCS 4354, Springer-Verlag (2007) 50–64
4. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream Fusion: From Lists to Streams to Nothing At All. In: ICFP '07, New York, ACM (October 2007)
5. The Unicode Consortium: The Unicode Standard, Version 5.2.0 (2010)
6. Gill, A., Launchbury, J., Peyton-Jones, S.: A Short Cut to Deforestation. In: ICFP '93, New York, ACM (1993) 223–232
7. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: ICFP '02, New York, ACM (2002) 124–132
8. Petyon-Jones, S., Tolmach, A., Hoare, T.: Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In: Haskell Workshop, ACM SIGPLAN (2001) 203–233
9. Harper, T.: Fusion on Haskell Unicode Strings, Master's Thesis, University of Oxford (2008)
10. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00, New York, ACM (2000)
11. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of Functional Programming **16**(2) (2006)
12. Boehm, H.J., Atkinson, R., Plass, M.: Ropes: an Alternative to Strings. Software: Practice and Experience **25**(12) (1995) 1315–1330