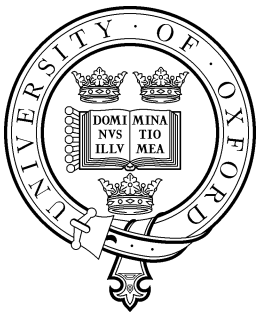
Fusion on Haskell Unicode Strings



Thomas Harper
St Anne's College
University of Oxford

Submitted for partial fulfilment of the degree of
Master of Science in Computer Science

Summer 2008

Acknowledgements

I would like to thank my supervisor, Dr Oege de Moor, for his support during this project. I would also like to thank my co-supervisor, Duncan Coutts, whose help was invaluable to me throughout this entire project. His constant availability on IRC went above and beyond the call of supervisorial duty, and his contagious enthusiasm for computer science made everything fun.

Thanks to everyone in Haskell community for being so helpful, and thanks to the organisers of AngloHaskell, who allowed me to get valuable feedback on this project.

Thanks to my new friends here at Oxford for helping me through my first year in a new country.

Thanks to everyone back home (especially Becca Tarsa and Salpy Kabaklian), who make an effort to keep me in their lives despite 6 time zones.

Thanks to Rhianedd Jewell and Julie Magar for being the worst (or rather, best) distractions a grad student could want.

To Jamie Anderson, thank you for being not only a good friend, but an inspiration. You're the reason I even dreamed of being here.

To Jim, thanks for making every day a good one.

And to my family, thank you for supporting me unconditionally.

Contents

List of Figures

Chapter 1

Introduction

One of the advantages of functional programming languages is their ability to express complex computations concisely. Programs written in functional programming languages are often shorter and easier to read than the same program written in an imperative or object-oriented language. One of the ways that this is achieved is through the composition of small functions to build up larger ones, forming a "pipeline" of functions. For example, this Haskell program

$$return \cdot words \cdot map\ toUpper \cdot filter\ isAlpha =\!\ll readFile\ f$$

filters out all non-alphabetic characters, converts them to upper case, and then tokenises them; this is a rather powerful program for one line of code using only standard functions. Although this method of programming is both logical and aesthetically pleasing, it is often at the cost of both speed and memory; string processing in Haskell is extremely inefficient in terms of both time and space, despite being a shining example of its beauty. This is due to Haskell's inefficient string representation, the *String* type.

The goal of this project is to address this problem by implementing a new string library in Haskell. This library will have a more compact representation and implement an API that mirrors Haskell's *List* library[10, ch. 17], but with better performance. Like Haskell's *String*, it will use a fusion framework (see Section 3.2.1) to improve speed and memory efficiency by removing intermediate structures. The representation and the library are both called *Text*. The need for a more efficient string representation has been the target of recent research in the form of the *ByteString* library. (see Section 2.1.2). However, support for the Unicode standard for representing text has not yet been addressed. *Text* fills this rather large gap in the current Haskell Hierarchical Libraries by providing a fast, packed string library that also supports Unicode. For example, Figure 1.1 shows the runtime of the above program in both *String* and *Text*, showing a speedup factor of 2.

In addition to the practical creation of a high-performance library, this project also has a theoretical aspect. It uses stream fusion not only as a way to eliminate intermediate data structures, but also as a way to convert between a diverse set of sequence types. In doing so, it allows functions over these sequences without regard for the underlying representation. This design method enables the creation of programs that can compute input from one representation and output it into another without needing to know about the specifics of either structure and without an explicit conversion step. Stream fusion's typical implementation is also adapted in *Text* to increase performance by propagating information about the length of the stream to aid in memory allocation.

The rest of this chapter more formally defines the objectives of this project, and then provides some necessary background information about Haskell, the Unicode standard, and fusion, which will be used throughout this dissertation. Chapter 2 discusses the previous work

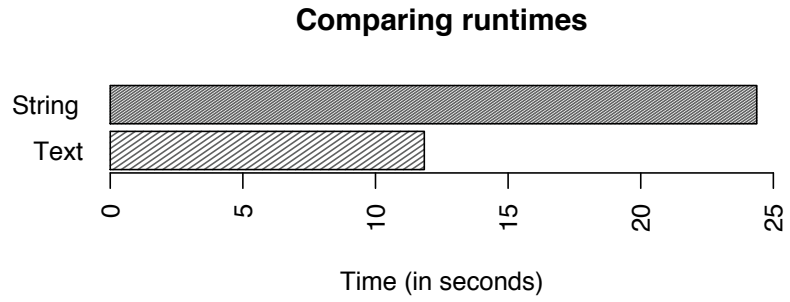


Figure 1.1: Comparison of runtimes between *String* and *Text* for a sample program

that is the basis for *Text*. It covers the string representations and fusion systems that relate to those used in *Text*. Chapter 3 presents the *Text* library itself, covering *Text*s memory layout in detail and the rationale behind its implementation. It also gives an overview of the *Text* API, with some specific examples. Chapter 4 presents the performance results of *Text* versus some other Haskell string representations and also details the methods used for benchmarking. That chapter also presents the testing library used to help establish confidence in the *Text* implementation and ensuring its behaviour is consistent with Haskell's current string representation. Finally, Chapter 5 presents the conclusions drawn from the work in this dissertation.

1.1 Objectives

The objectives of this project are as follows:

- **Create a compact representation for Unicode strings that can take advantage of stream fusion.** This representation should be transparent to the user, but provide faster performance and use less memory than Haskell strings. It also must use stream fusion to remove any intermediate structures in function pipelines.
- **Provide a library of fusible combinators over this structure, similar to those found in Haskell's *List* library.** The use of list combinators is already a common way to manipulate strings in Haskell. By implementing the same API as *List*, programmers will have access to a set of functions with which they are already familiar. This also makes migrating any code from using Haskell's *String* easy.
- **Create a benchmark system to evaluate the performance of *Text* versus other string representations in Haskell.** The purpose of *Text* is to provide Haskell programmers with faster, smaller strings. Benchmarking is important in tracking improvements in code over time and testing how much and when performance is better than other string types.
- **Create a test framework for *Text*.** A well-designed test framework will be able to check over each sort of case encountered by *Text*. Although there is an infinite space of strings, a reasonably defined subset can aid in testing *Text*.

1.2 Background

This section contains information relevant to the discussions in the rest of the dissertation. It briefly reviews the platform for *Text*, the Unicode Standard, and fusion.

1.2.1 Haskell

Introduction to Haskell

Haskell is a non-strict, purely functional programming language. It has a strong, static type system that supports polymorphism. It has first class and higher order functions. It uses these features, along with a built-in list type, to create a library of list combinators that operate over lists of any element type. These combinators are powerful enough to allow a programmer to do nearly anything with lists without resorting to recursing over the list. Many are found in Haskell's *Prelude*, but they can all be found together in Haskell's *List* module. Haskell's *String* type is implemented as a list type, thus allowing programmers to use list functions to write string manipulation functions. While this is very convenient, it makes string processing in Haskell extremely inefficient (see Section 2.1.1).

There are many different implementations of the Haskell programming language. Although they all implement the latest definition of Haskell, known as "Haskell 98"[10], many features have been developed and added to some of the implementations over time that do not appear in the language definition. Such features were influential in choosing the appropriate one to use for this project. The implementation chosen is the Glasgow Haskell Compiler.

The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is an implementation of Haskell that compiles programs to native machine code[12]. It also supports compiling Haskell to bytecode for use in interactive sessions via GHCi. GHC allows for extensive optimisation and gives the programmer control of it through various pragmas[13]. The *INLINE* and *NOINLINE* pragmas allow the programmer control whether or not a given function is inlined and at what stage(s) of compilation this can or cannot occur. The *RULES* pragma allows the programmer to specify rewrite rules to aid in code optimisation[9]. This project makes extensive use of these pragmas in the stream fusion framework to control the optimisation of fusible functions.

GHC also has "bang patterns". These function similarly to strict datatype fields. By preceding a variable name by an exclamation point (!) in a function declaration, that variable will automatically be marked as strict by the strictness analyser. Explicitly marking fields as such can help the compiler determine which values should be unboxed and reduce the allocation of unnecessary thunks.

Finally, GHC offers access to primitive types through *GHC.Prim* and *GHC.Ext*. Such primitives are the underlying machine-level types that are boxed by more familiar types such as *Int* or *Char*. The naming convention is that primitives types, values, constants, and functions all end with a hash sign (#), for example $Int\#$ is a type, $x\# :: Int\#$ is a variable, $42\#$ is a value, and $(+\#)$ is a function. In this project, these primitives are used to provide faster encoding and decoding of Unicode text.

1.2.2 Unicode

Introduction to Unicode

Unicode is a standard for representing most of the world's writing systems, including many historic scripts. Its name derives from its three principle goals: being "universal", "uniform",

and "unique" (the last referring to the uniqueness of bit sequences and their interpretation into character codes). Unicode began as a project among engineers from Apple and Xerox to incorporate the world alphabets into a single system using two-byte text. The Unicode Consortium, who now define the standard and publish *The Unicode Standard* books, was founded in 1991[14]. Currently, the newest *The Unicode Standard* is version 5.1.0, which is a delta document[16] consisting of changes to the text of the *The Unicode Standard, Version 5.0*[15]. The two documents together fully specify the current version of the Unicode standard.

A document encoded in Unicode consists of *code points*, each consisting of a 21-bit number that abstractly represents a character, such as Latin Capital Letter A (A), or a combining character that can modify another character, such as Combining Grave Accent,(`). When found together in the proper order, a parsing text renderer could then render this as (À)[1]. While Unicode code points abstractly refer to certain characters, it does not specify the *glyphs* that should be used to represent them. This is the responsibility of the system or application that is using Unicode.[2] In this case, the renderings of the above characters are a PDF reader's rendering of Unicode code points U+0041, U+0060, and the sequence of those two code points in that order, respectively. Code points are denoted by "U+" and the hexadecimal representation of that point. The leading zeroes are obligatory to show to which plane the code point belongs (see Section 1.2.2).

Unicode Encoding Forms

A sequence of Unicode code points may be encoded in one of three encoding formats: UTF-8, UTF-16, or UTF-32. The first two are variable-width encodings, and the last is fixed-width. A UTF-8 stream represents each code point as a sequence of one to four bytes, depending on the value of the code point. This a very compact coding because small characters can be represented in one or two bytes. It is also backwards compatible with ASCII, because byte-sized code points are represented by a byte with that value and have the same mappings as ASCII. The downside is that validating, parsing, and encoding in UTF-8 are more more computationally intensive than with UTF-16 or UTF-32.

A UTF-16 stream consists of 16-bit words, and a code point may be represented by one or two of them. A two-word sequence representing a single code point is referred to as a *surrogate pair*. UTF-16 is often less compact than UTF-8 due to the fact that even the lowest value characters are represented by a 16-bit (2 byte) value. Some characters, however, only take one UTF-16 word but three UTF-8 bytes. Validation and parsing of UTF-16 is far easier than UTF-8, however, because of the simpler encoding scheme. This can yield greater performance at the cost of memory.

Unlike the other two encodings, UTF-32 is a fixed width encoding. A UTF-32 stream consists of 32-bit words. The code points themselves are at most 21 bits, and so each one fits into a single UTF-32 value. UTF-32 is thus quite simple to validate, parse, or encode, but is very costly in terms of space; many characters take up one byte, and the majority of Unicode code points can be represented with two bytes, so the rest of the space is wasted. It can be useful when dealing with languages with many characters that need more than two-bytes because the memory versus performance trade-off is less.

Note that because UTF-16 and UTF-32 are specified in terms of words and not bytes, the order of the bytes may change depending on the endianness of the machine. The Unicode standard specifies that either endianness is acceptable, and that only the order of the bytes

[1] Note that there is also a single Unicode code point for the character À. This is known as a *precomposed* character. Many of these exist for compatibility with different languages and their legacy encoding systems

[2] This creates issues in Chinese characters that are used for multiple languages; some code points have multiple renderings depending on the language (Chinese, Japanese, or Korean). This issue is known as *Han unification*, but addressing this problem is outside the scope of this project

Plane	Range	Name	Content
0	U+0000 – U+FFFF	Basic Multilingual Plane	Modern scripts
1	U+10000 – U+1FFFF	Supplementary Multilingual Plane	Historic, musical, math
2	U+20000 – U+2FFFF	Supplementary Ideographic Plane	Extra CJK characters
3	U+30000 – U+3FFFF	Unused	
4	U+40000 – U+4FFFF		
5	U+50000 – U+5FFFF		
6	U+60000 – U+6FFFF		
7	U+70000 – U+7FFFF		
8	U+80000 – U+8FFFF		
9	U+90000 – U+9FFFF		
10	U+A0000 – U+AFFFF		
11	U+B0000 – U+BFFFF		
12	U+C0000 – U+CFFFF		
13	U+D0000 – U+DFFFF		
14	U+E0000 – U+EFFFF	Supplementary Special-purpose Plane	Control characters
15	U+F0000 – U+FFFFF	Private Use Area	
16	U+100000 – U+10FFFF		

Figure 1.2: Allocation of the Unicode planes

within each word is changed. To specify the endianness of a stream, "LE" or "BE" may be appended to the encoding name to specify endianness as little endian or big endian, respectively. Unicode also specifies a sequence of bytes that maybe appear at the beginning of a Unicode stream known as the *byte order mark* (BOM). The BOM can signify which encoding is used in the stream and, if UTF-16 or UTF-32, the endianness.

Code Point Planes

The Unicode code points are divided into 17 "planes", most of which are unused. Figure 1.2 shows the code point ranges for each plane. The discussion of planes is important when considering appropriate encoding forms. As will be demonstrated, the plane upon which the majority of a Unicode stream rests can have a significant impact on memory efficiency and performance, which can change depending on the choice of encoding form used.

The vast majority of current world writing systems, including Indian and East Asian scripts, are found in Plane 0, the Basic Multilingual Plane (BMP). Although this range of characters may need anywhere from one to three bytes in UTF-8, they only need one UTF-16 word (UTF-16 only uses surrogate pairs for any code point $\geq$U+10000). This means that parsing most code points in most documents is very straightforward in UTF-16 (and, of course, UTF-32), but still complex in UTF-8.

Plane 1, the Supplementary Multilingual Plane (SMP), consists mostly of historic scripts. It also contains musical notation and mathematical systems. For most documents, this plane will rarely be used, if ever. Plane 2, the Supplementary Ideographic Plane (SMP), consists of many CJK (Chinese/Japanese/Korean) characters that are less commonly used and would not fit in the BMP. Many of them are for historic or rare use, though a small number of common use CJK characters are found there, as well. Plane 14, the Supplementary Special-purpose Plane (SSP), contains format control characters that do not fit into the BMP. When considering encodings,

it is important to note that all of these planes require 4 bytes in any of the encoding forms, but that the parsing of the byte sequences representing these code points is more complicated (and thus less efficient) for a more compact encoding form.

The remainder of the planes are currently unused. The majority of them are likely to remain that way for some time, given the number and size of scripts left to be allocated. Two planes (15 and 16) are reserved for private use. The use of these unused planes was not considered in this project.

When developing a library for Unicode, considering plane allocation is very important when selecting the appropriate encoding form to maximise speed and memory efficiency. It also means that one encoding form is not necessarily the most appropriate for all document types. The importance of encoding form choice is further evaluated in the benchmarks in Chapter 4.

1.2.3 Fusion

Fusion, also known as *deforestation*, refers to program transformation techniques to remove intermediate structures from programs. Such intermediate structures arise commonly in functional programming, such as in this program:

$$(take\ 50 \cdot map\ toUpper \cdot filter\ (not \cdot isDigit))\ str$$

This program forms a pipeline of functions that process the string *str* by filtering out numeric characters, converting the entire string to upper case, and then returning the first 50 characters. Intermediate structures are created between each portion of the program that traverses the string to communicate its results to the next enclosing function. In this program, an intermediate string is produced by *filter*, but is then immediately consumed by *map*. The same is true between *map* and *take*. This program could be rewritten so that it creates no intermediate structures:

```
g str 0
  where
    g []     _          = []
    g (x:xs) n
        | n >= 5        = []
        | not (isDigit x) = toUpper x : g xs (n + 1)
        | otherwise     = g xs n
```

This program, while being more efficient, is far less desirable than the former one. It is harder to read, more complicated to write, and longer. Fusion techniques, however, automate the conversion from programs that creates intermediate structures to ones that do not. These techniques are used by the compiler and should be transparent to the programmer. Because of GHC's rewrite rules, these techniques can be implemented without any modifications to the actual compiler. The Haskell string representations discussed in Chapter 2 utilise various forms of fusion to help optimise programs and the *Text* library uses an implementation of stream fusion. Section 2.2 explains and compare various fusion frameworks including those used in *String*, *ByteString*, and *Text*.

Chapter 2

Previous Work

This chapter discusses some previous work related to strings in Haskell and stream fusion. First, it presents an overview of some current string representations in Haskell, their features, and their advantages and disadvantages. Then, it presents a summary of fusion with a focus on those fusion systems are currently used in Haskell. Finally, it gives an introduction to stream fusion and a Haskell implementation of it.

2.1 String Representation

String representation in Haskell has already been the target of recent research in the form of the *ByteString* library, which utilises stream fusion in its most recent versions. The contrast between Haskell's *String* and the structure and performance of *ByteString* is the main source of the inspiration for the *Text* library. Studying the structure of *String* is necessary to understand what makes it inefficient and how its shortcomings may be corrected. It is also important to identify its advantages in order to preserve those in a new representation. *ByteString* is largely successful and doing so, but introduces its own complications. It also does not support Unicode. *ByteString* does, however, serve as the main source of inspiration for the design of *Text*.

2.1.1 The Haskell *String* Type

The Haskell string representation is quite simple:

type *String* = [*Char*]

This representation, in addition to being short, is rather elegant and convenient. It allows the programmer to use the entire polymorphic list library for string manipulation. Pattern matching can be used to create inductively defined functions over strings. Also, new cells and characters may be dynamically allocated at run time to easily accommodate strings of arbitrary length. As with all lists, GHC uses *foldr/build*[8] fusion to remove some intermediate *String*s. An overview of *foldr/build* fusion is presented in Section 2.2.1.

This design amounts to an extremely convenient but painfully inefficient way to store and manipulated strings. Figure 2.1 shows the structure of the string `"Str"` represented as a *String* when allocated by GHC[12]. A *String*, being a specific type of Haskell's built-in list, is basically a linked-list of heap allocated *Char*s. The head of the list is a pointer to the first element. That element is composed of three word-sized cells. The first is a "header word", identifying what the object is. The second is a pointer to the first character of the string. The third is a pointer to the next cell. Each character also has a cell containing a header word, and then a word-sized

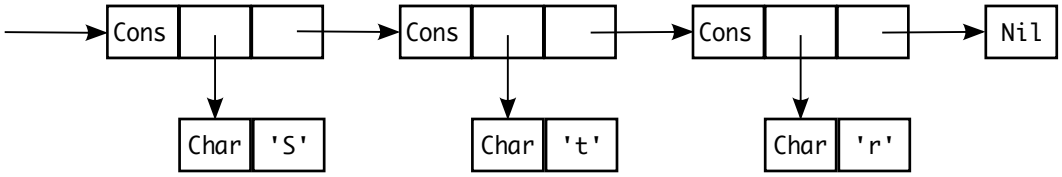


Figure 2.1: Structure of *String*

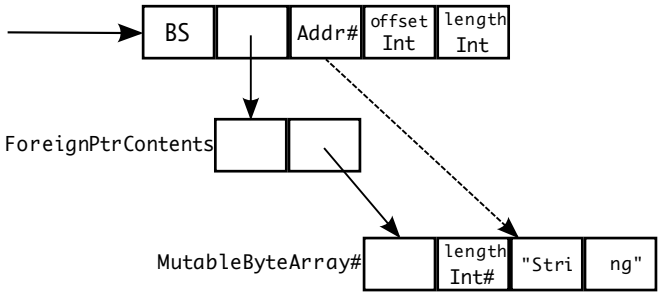


Figure 2.2: Structure of *ByteString*

cell containing the actual character. GHC uses either 32- or 64-bit words, depending on the platform. This means that each character is fully capable of holding a Unicode code point, and this is how Haskell natively supports Unicode. It also means, however, that each character plus its corresponding list cell comprise five words, which is either 20 or 40 bytes per character.[1]

This translates to poor performance. Access to a given character involves a huge amount of pointer chasing. Furthermore, this makes for poor cache locality with only three to five characters fitting into a 64 byte cache line on a 32-bit machine. In comparison, ASCII only requires one byte per character. At the worst case, any of the Unicode encoding forms only require four bytes per character. In the case of UTF-8 and UTF-16, the strings are usually far more compact. The limitations of *foldr/build* fusion also keep Haskell lists, and thus *String*s, from performing well in functions that are crucial from list processing; notably, folds with accumulating parameters (e.g. *foldl*) and zips cannot be fused.

Much of this inefficiency stems from the structure of Haskell lists. They use a traditional linked list model that requires allocating "cons" cells to form the spine of list. Of the five words needed for *Char* in a list, three of them are necessary because of the list structure. Such a sequence type is inappropriate for dealing with strings. Furthermore, *foldr/build* fusion cannot adequately deal with certain vital functions for list processing.

2.1.2 The *ByteString* Library

An alternative to *String* that attempts to overcome its inefficiencies is the *ByteString* library[7]. Figure 2.2 depicts the structure of a *ByteString*. Instead of a linked list, a *ByteString* consists of a *ForeignPtr*, an *Addr#*, an offset, and a length. The *ForeignPtr* provides a pointer for foreign code to access the string. The *ForeignPtrContents* are a *MutableByteArray#*, which is a raw memory space that is accessible at the byte level. This is where the string is stored is an array-based representation. The offset and length fields allow constant time creation of substrings without copying. Because the *ForeignPtr* creates an additional pointer indirection,

[1] Although this is true, characters whose values are below 255 are shared in the GHC runtime system, so if multiple instances of such a character exist, subsequent instances only need any additional 12 or 24 bytes.

the *Addr#* field is used to allow Haskell code to directly access the string without traversing the extra pointers.

Each character itself is represented by one byte. In terms of compactness, this is a major advantage. A *ByteString* efficiently represents ASCII characters with no extra memory. Furthermore, it offers constant time access to any element of the string. Because of the offset and length fields, forming substrings takes constant time and requires no copying (all substrings share the same copy of the array). The *ByteString* API implements most of the functions found in *List*, along with functions to convert to/from *ByteString*s and to perform I/O. Most of the functions in the *ByteString* library can be optimised using stream fusion in order to eliminate any intermediate structures that appear between two composed calls. Because this representation is fixed-width, functions can read and write the string top to bottom or bottom to top, depending upon which is most efficient (e.g. top to bottom for a *foldl* or bottom to top for a *foldr*). This is the case even when stream fusion is invoked. The result is an extremely fast string library. It beats out *String*s and, in some cases, naïve C code. It also only uses one byte per character, as opposed to the 20 or 40 bytes of *String*.

While *ByteString* accomplishes the desired speed and memory efficiency over *String*, in doing so it creates several disadvantages. First, a one-byte character can only represent the the ASCII character set, or Unicode code points U+0000 to U+00FF. Obviously, in dealing with Unicode it is necessary to have a representation capable of dealing with values U+0000 to U+10FFFF. Also, due to the use of a *ForeignPtr* in storing the string, it is inappropriate for small strings. This is because smaller objects are usually unpinned (i.e. movable by the garbage collector) in Haskell. In the case of *ForeignPtr*s, however, even the smallest objects must be pinned so that foreign code can always access them.

With very few very small strings, this issue may not be a problem. Issues could arise, however, in the case of using a large number of very small strings. In this case, pinned objects can lead to poorer performance, because memory allocation takes longer. In large blocks of text, this inefficiency is negligible in comparison with the performance achieved in manipulating strings. In smaller strings, however, this can be a more noticeable problem. The fact that small objects are pinned can also lead to memory fragmentation, which makes allocation of further objects harder and wastes space by forcing them to use other, contiguous blocks of memory.

2.2 Fusion Systems

The creation of intermediate structures is a key factor in determining the performance of *Text* and libraries like it. Functional programming lends itself to writing programs that need intermediate structures to communicate results between adjacent functions. Common list combinators, such as *map*, *filter*, and *foldr*, encourage this style in programs over lists. Because of the fact that strings themselves are lists in Haskell, these functions are also a useful way to manipulate strings. While intermediate structures are necessary in communicating results in such a style of programming, transforming these programs into forms with no intermediate structures yields a non-trivial increase in performance and memory efficiency.

Fusion was first introduced in Philip Wadler's deforestation algorithm[17]. It was called such because it was designed to remove intermediate trees in functional programs. This type of fusion is capable of fusing recursive functions to eliminate arbitrary intermediate data structures and was designed to be implemented in an optimising compiler. This algorithm, however, has a rather severe complication. Because it transforms arbitrary recursive functions, it cannot guarantee the termination of its optimisation. Overcoming this requires adding additional complexity to the algorithm, or placing additional restrictions of the forms of function definitions[8]. In this case, it is not realistic to place restrictions on the programming language, nor on the types of programs written in it.

A more realistic alternative is the set of fusion frameworks known as *shortcut fusion*. Shortcut fusion does not place any restriction on the input program. Instead, it doesn't guarantee that all intermediate structures will be removed in a program. It requires that all fusible functions be written in terms of recursive combinators that control how these structures (in this case, lists or strings) are produced, consumed, or transformed. It is then possible to establish rewrite rules that remove intermediate structures. This sort of fusion is well-suited to situations where access to a structure is implemented through an API. The API functions can be implemented in terms of fusion combinators, and then any program written with them will not create any intermediate structures.

The term "shortcut fusion" was first coined in "A Short Cut to Deforestation"[8], which describes the *foldr/build* system used by GHC. Since then, other short cut fusion systems have been created that address some of the limitations that appear in the *foldr/build* system. The following sections describe the major shortcut fusion systems that have been used in Haskell. The latter of them, stream fusion, is used in the *Text* library. Its use is described in more detail in Chapter 3.

2.2.1 *foldr/build* Fusion

The *foldr/build* approach to fusion[8] uses two combinators, *build* and *foldr*, and a single rewrite rule. This system was designed for and implemented in GHC to fuse functions over lists. The *foldr* function is the same function that is found in the *List* module:

$$\begin{aligned} &\mathit{foldr} && :: (a \to b \to b) \to b \to [a] \to b \\ &\mathit{foldr}\ f\ z\ [\,] && = z \\ &\mathit{foldr}\ f\ z\ (x : xs) && = f\ x\ (\mathit{foldr}\ f\ z\ xs) \end{aligned}$$

The calculation of *foldr* over a list can be envisioned as replacing each *Cons* of a list with a binary operator and *Nil* with a specified value (usually an identity value). For example:

$$\mathit{foldr}\ (+)\ 0\ [1,2,3,4,5] \equiv (1 + (2 + (3 + (4 + (5 + 0)))))$$

This abstracts list consumption over the *Cons*/*Nil* structure of the list. It is also possible to abstract a list construction function over this structure, without explicitly using *Cons* and *Nil*:

$$\begin{aligned} &\mathit{build} && :: (\forall b.\ (a \to b \to b) \to b \to b) \to [a] \\ &\mathit{build}\ g && = g\ (:)\ [\,] \end{aligned}$$

In *build*, the importance rests with the function g. It must specify the the building of a list in terms of two arguments: a function playing the role of *Cons*, and a value playing the role of *Nil*. When g is applied to (:) and [ ], it will build a list using its arguments as the *Cons* and *Nil*, respectively.

Now, because the consumption of a list substitutes a different function for *Cons* and a different value for *Nil*, and the *build* function forces its input function to abstract away form *Cons* and *Nil*, the following equivalence applies:

$$\langle \textbf{foldr/build fusion} \rangle \quad \forall g\ k\ z.\ \mathit{foldr}\ k\ z\ (\mathit{build}\ g)\ \mapsto g\ k\ z$$

The removal of the *foldr* and *build* eliminates the the intermediate structure that would have occurred between them. Instead of building a list and then folding it, each element is folded as is it is produced. Some examples[8] of list combinators defined in terms of *foldr* and *build* are show in Figure 2.3 The examples show the general structure of list combinators in the *foldr/build* system. With the exception of *foldl*, all of these functions are both list consumers and list producers. The consumption portion is defined in terms of *foldr*, and the production

```
map f xs      = build (λc n → foldr (λa b → c (f a) b) n xs)
filter f xs   = build (λc n → foldr (λa b → if f a then c a b else b) n xs)
zip xs ys     = build (λc n → let zip' (x : xs) (y : ys) = c (x, y) (zip' xs ys)
                                  zip' _ _ = n
                              in zip' xs ys)
foldl f z xs = foldr (λb g a → g (f a b)) id xs z
```

Figure 2.3: Examples of *foldr/build* list combinators

of the list is defined by build. As a fold does not produce a list, *foldl* is only defined in terms of *foldr*.

The limitation that list consumption can only be in terms of *foldr* places limits on which types of functions can be fused. As can be seen in the definition of *zip*, the output from *zip* is fusible, but it is not fusible on its inputs. This is because *foldr* cannot traverse two lists in parallel. While it would be possible to write the zip in terms of folding one list, it would still have to explore the other conventionally. It is also impossible to write functions that use an accumulating parameter; although *foldl* can be written in terms of *foldr* (as in Figure 2.3), such a definition builds up a series of nested function calls that are only evaluated at the end of the list.

This approach to fusion has been one of the most successful so far. It has been included in GHC to optimise programs on lists (and thus strings). This implementation is a concrete example of the ability to get real performance gains using this approach, and its simplicity of two combinators and only one rewrite make it an attractive possibility to accomplish fusion for *Text*s. Restricting function definitions to right folds, however, means that there are still un-optimised functions that would benefit from a less restrictive fusion system.

2.2.2 *destroy/unfoldr* Fusion

The *destroy/unfoldr* fusion system[11] is a more recent approach that addresses some of the issues in *foldr/build*, namely its inability to fuse *zip*s and true left folds. Like *foldr/build*, it defines list operations in terms of two functions, *unfoldr* and *destroy*. The *unfoldr* function is the dual to *foldr*, creating a list from a seed value:

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
unfoldr f b = case f b of
  Just (a, b') → a : unfoldr f b'
  Nothing      → []
```

This is used to define functions that produces lists. Functions that consume lists are defined in terms of *destroy*:

```
destroy :: (∀a.(a → Maybe (b, a)) → a → c) → [b] → c
destroy g xs = g listpsi xs
  where listpsi :: [a] → Maybe (a, [a])
    listpsi []       = Nothing
    listpsi (x : xs) = Just (x, xs)
```

When composed together as (*destroy* · *unfoldr*), the two from the identity function on lists. This is because *unfoldr* gets substituted for g, and *unfoldr listpsi* applied to a list unfolds the list back to itself. The principle of this fusion, however, is that as each element is unfolded, it

```
map f xs = destroy (λpsi a → unfoldr (mapDU psi) a) xs
  where
    mapDU psi xs = case psi xs of
      Nothing      → Nothing
      Just (x, ys) → Just (f x, ys)
filter p xs = destroy (λpsi a → unfoldr (filterDU psi) a) xs
  where
    filterDU psi xs = case psi xs of
      Nothing      → Nothing
      Just (b, ys) → if p b
                       then Just (b, ys)
                       else filterDU psi ys
foldl f b xs = destroy (foldlDU b) xs
  where
    foldlDU acc psi xs = foldlDU' acc xs
      where
        foldlDU' acc xs = case psi xs of
          Nothing      → acc
          Just (a, ys) → let acc' = f acc a in (foldlDU' acc' ys)
zip xs ys = destroy (λpsi1 e1 →
            destroy (λpsi2 e2 →
              unfoldr (zipDU psi1 psi2) (e1, e2)
                ) ys
              ) xs
  where
    zipDU psi1 psi2 (e1, e2) = case psi1 e1 of
      Nothing      → Nothing
      Just (x, xs) → case psi2 e2 of
                       Nothing      → Nothing
                       Just (y, ys) → Just ((x, y), (xs, ys))
```

Figure 2.4: Examples of *destroy/unfoldr* list combinators

is possible to apply a series of transformations to that element before moving on to the next element. This yields the following rewrite rule:

$$\langle \textbf{destroy/unfoldr fusion} \rangle \quad \{\forall g\ f\ e.\ destroy\ g\ (unfoldr\ f\ e)\ \mapsto g\ f\ e$$

This lets the transformation defined in g be performed on the list as it is created, instead of unfolding a list completely and then applying a transformation to the newly created intermediate structure. Figure 2.4 shows some typical list functions defined in terms *destroy/unfoldr*. Both *foldl* and *zip* can be defined in terms of *unfoldr* and *destroy*, and are thus fusible. In the case of *zip*, this reveals that the use of *unfoldr* allows each input list to be unfolded one at a time, and then combined, and then start the process all over again until one or both lists is empty.

The *filter* definition shows one of the limitations of this system; filter must be defined in terms of a recursive loop. While still fusible, adding these extra recursive loops introduces inefficiency into *filter* and similarly defined functions. Also, an additional rewrite rule is necessary in certain cases. If two *map* calls, for example, occur next to each other, both of them have an inner *unfoldr* and an outer *destroy*, meaning that the outer destroy cannot fuse with the inner

unfoldr to eliminate the intermediate structure. This requires the introduction of an additional rewrite rule:

$$\langle \textbf{destroy/destroy fusion} \rangle \quad \forall g\ g\ ls.\ destroy\ g\ (destroy\ g'\ ls) \mapsto destroy\ (\lambda psi\ a \rightarrow destroy\ g\ (g'\ psi\ a))\ ls$$

This rule can move the outer *destroy* inside the inner one so that it can encounter an *unfoldr*.

As can be seen, defining some functions, especially those that both consume and produce a list, can be quite complex. These functions are much longer than and more difficult to read than their *foldr/build* counterparts. The *destroy/unfoldr* version of *filter* even introduces efficiencies that didn't exist in the *foldr/build* version. While this fusion system addresses the problems in the *foldr/build* system, the additional complexity it incurs for doing so makes it difficult to work with.

2.2.3 Stream Fusion

Each of the previously described fusion systems have certain limitations that prevent them from being a complete fusion system for list combinators. A more recent fusion system known as stream fusion[6, 7] overcomes the limitations of both of these systems. The description here applies to lists for comparison with the preceding fusion systems. The implementation of stream fusion for *Text*s is described in 3.2.1.

Like *destroy/unfoldr*, stream fusion uses functions that work in terms of the unfold of a list. Unlike *destroy/unfoldr*, however, stream fusion converts the list to an explicit unfold type called a *Stream*:

$$\textbf{data}\ Stream\ a = \exists s.\ Stream\ (s \rightarrow Step\ s\ a)\ s$$

A *Stream a* consists of stepper function and a seed. An existential wrapper binding the type variable *s* of the seed means that the seeds can be different than the values they yield. This is important for streams that need to carry a state. The stepper function produces a stream by being applied to a seed and yielding the next step:

$$\begin{aligned}\textbf{data}\ Step\ s\ a = {} & Done \\ \mid {} & Skip\ s \\ \mid {} & Yield\ a\ s\end{aligned}$$

A *Step* can either be *Done*, signifying the end of a list, a *Skip*, saying an element should be passed over and giving the next seed, or a *Yield*, which produces both a value and the next seed. Converting to a list to a *Stream* is accomplished using the *stream* function:

$$\begin{aligned}&stream :: [a] \rightarrow Stream\ a \\ &stream\ xs_0 \quad = Stream\ next\ xs_0 \\ &\quad \textbf{where} \\ &\qquad next\ [\,] \quad = Done \\ &\qquad next\ (x : xs) = Yield\ x\ xs\end{aligned}$$

This creates a *Stream* by using the list itself as the seed, and then specifying a function that unfolds the list one value at a time using *Yield*. The empty list corresponds to a stream being *Done*. Converting back to a list is accomplished using *unstream*:

$$\begin{aligned}&unstream :: Stream\ a \rightarrow [a] \\ &unstream\ (Stream\ next_0\ s_0) = unfold\ s_0 \\ &\quad \textbf{where}\end{aligned}$$

$$
\begin{array}{l}
unfold\ s = \textbf{case}\ \ next_0\ s\ \textbf{of} \\
\quad Done \quad\ \ \rightarrow \quad [\,] \\
\quad Skip\ s' \quad \rightarrow \quad unfold\ s' \\
\quad Yield\ x\ s' \rightarrow x : unfold\ s'
\end{array}
$$

Here, *unfold* applies the function specified in *stream* to each new seed, *cons*ing any values it encounters, until it reaches *Done*.

Manipulating a list is accomplished by defining a function over a stream, so that when the stream is unfolded it is different from the initial list. Consider the function map_s over streams:

$$
\begin{array}{l}
map_s :: (a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b \\
map_s\ f\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
\quad \textbf{where} \\
\quad\quad next\ s = \textbf{case}\ \ next_0\ s\ \textbf{of} \\
\quad\quad\quad Done \quad\ \ \rightarrow Done \\
\quad\quad\quad Skip\ s' \quad \rightarrow Skip \quad\ \ s' \\
\quad\quad\quad Yield\ x\ s' \rightarrow Yield\ (f\ x)\ s'
\end{array}
$$

This function creates a new stepper by applying the old one, return the same values for *Done* and *Skip*, but modifies any values from a *Yield* by applying f to them, thus modifying all the values yielded by the original stream. When the stream is unfolded by *unstream*, it will apply the new stepper function *next* value, one a time, which in turn applies $next_0$, unfolding and then transforming each value of the stream and *cons*ing it to the list.

In comparison, a filter over a stream has a similar structure, but its stepper function is differentiates it from map_s:

$$
\begin{array}{l}
filter_s :: (a \rightarrow Bool) \rightarrow Stream\ a \rightarrow Stream\ a \\
filter_s\ p\ (Stream\ next_0\ s_0) = Stream\ nexts_0 \\
\quad \textbf{where} \\
\quad\quad next\ s = \textbf{case}\ \ next_0\ s\ \textbf{of} \\
\quad\quad\quad Done \quad\quad\quad\quad\ \ \rightarrow Done \\
\quad\quad\quad Skip\ s' \quad\quad\quad\quad\ \ \rightarrow Skip \quad\ \ s' \\
\quad\quad\quad Yield\ x\ s' \mid p\ x \quad\quad \rightarrow Yield\ x\ s' \\
\quad\quad\quad\quad\quad\quad\ \mid otherwise \rightarrow Skip \quad\ \ s'
\end{array}
$$

In this function, the need for *Skip* is revealed. Like map_s, $filter_s$ creates a new stepper function that applies the original, but then instead of transforming any values, it either yields the same value again or replaces it with a *Skip*, which keeps the seed but discards a value. This allows this function to pass over elements of the stream without resorting to recursion to keep track of the last value that satisfied the predicate.

Using *stream*, *unstream*, and functions over streams, it is possible to define the same functions over lists:

$$
\begin{array}{l}
map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
map\ f\ xs = unstream \cdot map_s\ f \cdot stream \\
filter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a] \\
filter\ p\ xs = unstream \cdot filter_s\ p \cdot stream
\end{array}
$$

Now the opportunity for fusion occurs when these functions are composed over a list. Consider the program:

$$(map\ f \cdot filter\ p)$$

$$
\begin{aligned}
&\mathit{foldl}_s :: (b \to a \to b) \to b \to \mathit{Stream}\ a \to b \\
&\mathit{foldl}_s\ f\ z_0\ (\mathit{Stream}\ \mathit{next}\ s_0) = \mathit{loop}\ z_0\ s_0 \\
&\quad \textbf{where} \\
&\qquad \mathit{loop}\ z\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\quad \mathit{Done} \to z \\
&\qquad\quad \mathit{Skip}\ s' \to \mathit{loop}\ z\ s' \\
&\qquad\quad \mathit{Yield}\ x\ s' \to \mathit{loop}\ (f\ z\ x)\ s' \\
&\mathit{zip}_s :: \mathit{Stream}\ a \to \mathit{Stream}\ b \to \mathit{Stream}\ (a, b) \\
&\mathit{zip}_s\ (\mathit{Stream}\ \mathit{next}_{a0}\ s_{a0})\ (\mathit{Stream}\ \mathit{next}_{b0}\ s_{b0}) = \mathit{Stream}\ \mathit{next}\ (s_{a0}, s_{b0}, \mathit{Nothing}) \\
&\quad \textbf{where} \\
&\qquad \mathit{next}\ (s_a, s_b, \mathit{Nothing}) = \textbf{case}\ \mathit{next}\ s_a\ \textbf{of} \\
&\qquad\quad \mathit{Done} \to \mathit{Done} \\
&\qquad\quad \mathit{Skip}\ s'_a \to \mathit{Skip}\ (s'_a, s_b, \mathit{Nothing}) \\
&\qquad\quad \mathit{Yield}\ a s'_a \to \mathit{Skip}\ (s'_a, s_b, \mathit{Just}\ a) \\
&\qquad \mathit{next}\ (s'_a, s_b, \mathit{Just}\ a) = \textbf{case}\ \mathit{next}\ s_b\ \textbf{of} \\
&\qquad\quad \mathit{Done} \to \mathit{Done} \\
&\qquad\quad \mathit{Skip}\ s'_b \to \mathit{Skip}\ (s'_a, s'_b, \mathit{Just}\ a) \\
&\qquad\quad \mathit{Yield}\ b\ s'_b \to \mathit{Yield}\ (a, b)\ (s'_a, s'_b, \mathit{Nothing})
\end{aligned}
$$

Figure 2.5: Examples of stream fusion list combinators

When these functions are inlined, the program becomes:

$$(\mathit{unstream} \cdot \mathit{map}_s\ f \cdot \mathit{stream} \cdot \mathit{unstream} \cdot \mathit{filter}_s\ p \cdot \mathit{stream})$$

Notice in the middle of this program that *filter* needs *unstream* to convert the input back to a list, but *map* needs to stream it again. This would create an unnecessary conversion and an unnecessary intermediate list. Removing these functions yields a more reasonable program:

$$(\mathit{unstream} \cdot \mathit{map}_s\ f \cdot \mathit{filter}_s\ p \cdot \mathit{stream})$$

In this program, no intermediate structure is created between map_s and filter_s, each of them merely modifies the stepper function that *unstream* uses to unfold the new list. Each original list element is tested by the predicate p, and is then transformed by map if it passes, and only then is added to the resulting list. Because every function is defined in terms of *stream* and *unstream*, the only rewrite rule that is necessary is

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

As previously mentioned, it is possible to define *filter* without any recursive loops using the *Skip* step, which is much more desirable than the *destroy/unfoldr* approach.

Figure 2.5 shows stream versions of *fold* and *zip*, two troublesome functions for fusion *foldr/build*. The *foldl* implementation can use an accumulating parameter. This is because functions that consume lists, but not produce them, can have recursive loops. The loop for *foldl* can unfold the stream and fold the values in the same way that it could were it using a list. This makes it much easier to define list consumers than in the *foldr/build* framework.

It is also possible to define a *zip* that can fuse both input lists. The stepper function for doing so is slightly more complicated than the previously discussed functions. It encapsulates a notion of state by modifying the next seed to reflect a result of the prior computation. The seed for the result stream always carries the current seed from each stream, plus a value of type

Maybe, which is uses for pattern matching. When that value is *Nothing*, the stepper function tries to unfold an element from the first stream. If that stream yields a *Skip*, it tries to get another value from that stream. When it counters a *Yield*, it stores the yielded value in the *Maybe* value. When the pattern matches to a non-empty *Maybe* value (i.e. of the form *Just* x for some x), it goes through the same process for the second stream. When it has a value from both streams (the first being stored in the *Maybe* value), it can form a pair, discard the old *Maybe* value and restart the process by replacing it with *Nothing*. If either stream ever ends, the result stream also ends, thus fully replicating the behaviour of *zip*.

In addition to overcoming the limitations of previous fusion systems, stream fusion's approach makes it easily portable. The explicit representation of streams allows for functions that deal with all sequences in terms of the same stream type. All of the functions in Figure 2.5 work for any data structure with the same *Stream* type. For example, the example functions are nearly identical to those used for the *Text* library, the only difference being that the *Stream* type has additional fields for performance reasons (see Chapter 3). The only differences between different data structures are the implements of their respective *stream* and *unstream* functions. This ability to fuse arbitrary sequences is an important feature for the *Text* library.

Chapter 3

The *Text* Library

Chapter 2 established the need for a faster, more memory efficient string representation in Haskell. Although *ByteString* is both of these things, it sacrifices the ability to represent Unicode characters (or any characters whose values are greater than 255) to achieve its goals. The *Text* library is so called because, unlike *ByteString*, its API abstracts away from the underlying representation, allowing characters to have any Unicode value. Whereas storing multi-byte encodings in *ByteString* would force the programmer to deal with the ensuing complications, in *Text* it is completely transparent To achieve this, *Text* has the additional complexity of encoding and decoding Unicode byte streams quickly. Furthermore, it must impose a character-level API on a more complex underlying representation.

This chapter discusses the *Text* library in detail. First, it covers the internal structure of *Text* and the reasons for its design. Then, it discusses the design of the *Text* API and the implementation of some functions. These functions are meant to be representative examples of the entire API, whose source code is in the appendices.

3.1 String Representation

Storage

Part of the limitation of Haskell's string representation is its choice of a linked list as a data structure. Arrays are a much more more efficient choice. They allow constant time access to any element, which allows faster access to an arbitrary character. Because an array is just an contiguous span of memory, no extra space is used for pointers and nodes. The downside, of course, is that an array's size must be decided at creation. It cannot grow or shrink like a list can. Deleting an arbitrary element requires copying part of the array to "close the gap".

To obtain the performance desired, *Text* uses an array-based representation. An array of unboxed types allows for the most compact representation, because unboxed types have no pointer indirection. While the array itself is heap-allocated, it's underlying memory space contains the values themselves instead of pointers to them. A boxed representation would allocate these values on the heap and then store their pointers in an array.

Encoding

The choice of storage format for *Text* does not answer the question of how the stored information is encoded. The Unicode Standard specifies three different encoding forms: UTF-8, UTF-16, and UTF-32. The most compact is UTF-8, using a byte for the small values and up to four bytes when necessary. The fastest is UTF-32, which represents all Unicode values by their code

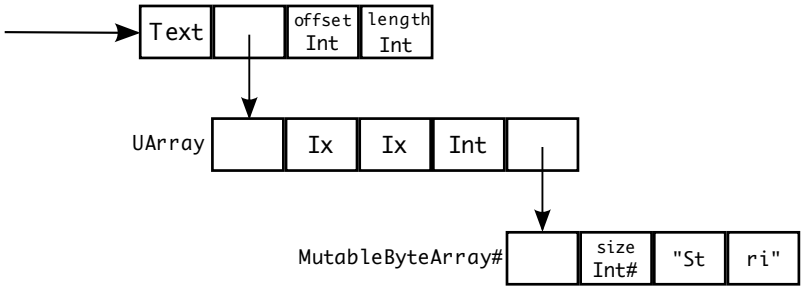


Figure 3.1: Structure of *Text*

points in 32-bit words. The encoding/decoding is trivial, but it uses 32-bits (4 bytes) for each character, making it extremely memory inefficient. Nearly all current world scripts are in the BMP, where each character only needs one to three bytes in UTF-8.

Although neither the simplest nor the smallest, UTF-16 represents a reasonable middle ground. UTF-16 streams consist of sequence of 16-bit words. Each character consists of one or two words. Although variable-width, this encoding is far simpler to work with, and thus much faster. It is important to note that all of the characters in the BMP fit into one UTF-16 word, so the majority of characters in common use are easy to encode. In the case of characters U+0800 to U+FFFF, UTF-16 is actually more compact than UTF-8, which requires three bytes. The fact that UTF-16 is so much simpler than UTF-8 but still takes up half the space of UTF-32 is why it was chosen for *Text* (see Section 4.1.4).

3.1.1 The *Text* Type

Having established a storage method and an encoding choice, the *Text* type takes the following form:

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

The *UArray*, or unboxed array, uses *Int*s for its indices and *Word16*s for its elements. The choice of *Word16* allows an easy implementation of the UTF-16 encoding standard, which is defined in terms of 16-bit numbers. Furthermore, their behaviour is the same regardless of a machine's endianness; although the byte order of a *Word16* is different between big endian and little endian machines, reading a *Word16* from this array will always yield the same value. The two *Int* fields are the offset and the length in 16-bit words of the string. The uses of these fields are explained in Section 3.2.3. These fields are denoted as strict by the exclamation points preceding them. The reason for this is to prevent any unwanted laziness. Because the strings are in stored in unboxed arrays, *Text* is an inherently strict representation anyway. Not shown are also the *UNPACK* pragmas in front of each field. This ensures that these fields are unboxed so that in addition to being strict they have no pointer indirection.

The structure of *Text* is shown in Figure 3.1. The *UArray* of the *Text* stores the string in a *ByteArray#*. The *ByteArray#* is a raw area of memory, and is read in 16-bit chunks when indexed. The instance of *Word16* for unboxed arrays stores them compactly, so that two *Word16*s only occupy one 32-bit word. This is in contrast to a boxed *Word16*, which merely reads the lower 16-bits of a hardware word (either 32 or 64 bits). A *ByteArray#* is immutable, and thus *Text*s are an immutable data structure. This allows sharing of a *ByteArray#* among substrings derived from a common string. Each of them has a different offset and length, but can point to the same *UArray*.

The overhead of storing a text takes 11 words, plus the actual buffer. While this may seem like a large amount of wasted space, it is close in size to a *String* consisting of three pre-allocated

*Char*s. Thus, even in relatively small strings *Text* is more compact than a *String* of the same length. Achieving a compact representation fulfils one of the objectives of the *Text* library. An array-based structure, which inherently has fast read and write capability, also lends itself to designing a high performance API.

3.2 The *Text* Interface

Text is meant to serve as a replacement for *String*. This means that programming using *Text*s should be similar to programming using *String*s. It isn't possible to replicate all mechanisms for accessing *String*s, though, because the internals of *String* and *Text* are fundamentally different. Specifically, it is possible to write an inductively defined function over a list using pattern matching, but not over Text. The internals are of Text are also far more complicated than *String*, and a programmer should not have to be intimately familiar with these differences in order to use *Text*. This rules out allowing the user to write inductively defined functions over lists, which would require the programmer to worry about array indices, offsets, and lengths. Furthermore, in order to benefit from stream fusion, *Text* must be accessible through a set of combinators that are already defined in terms of *stream* and *unstream*.

There is a suitable abstraction away from the *String* and *Text* internals that allows them both to be used in the same manner. Haskell's *List* module[10, ch. 17] implements an API over *String*s that models nearly the all the recursive patterns possible over *String*. The use of these is already the most common and desired way to manipulate *String*s. Implementing the same API over *Text*s allows programmers to manipulate strings in the same manner regardless of the underlying type and yet take advantage of the benefits that *Text* offers. Because these API functions are implemented using stream fusion, it is first necessary to implement a stream fusion framework that operates over *Text*s. Because of the explicit form of streams, the stream functions abstract away from the internal structure of *Text* are thus very straightforward to implement.

3.2.1 Stream Fusion over *Text*

Stream fusion over *Text* requires dealing with the additional complexities an array based representation and a variable-width encoding. With lists, conversion from a stream to a list was easy. It only required unfolding the stream and *cons*ing each yielded value. When converting to a *Text*, the array must allocated before the stream is unfolded, and there is no way to know the size of the stream beforehand. If an array is too small, there won't be enough space to accommodate the entire stream. It is possible to allocate a new, larger array and then copy the current array into the new, larger array, but such copying takes up additional time. Also, any guess about the size of the stream would be arbitrary; it could be far too small or far too big. To aid in making a reasonable guess about the necessary size of a stream, the *Stream* type gains a field:

$$\textbf{data}\ \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \to \mathit{Step}\ s\ a)\ !\ s\ !\ \mathit{Int}$$

This field carries the size of the Text from which the stream was created. When modifying the stream, functions can alter it if they know precisely how many additional fields with will be necessary (in *cons*, for example, exactly one character is added). While this guess may not be perfect, as some functions cannot predict what changes there will be in the stream (*filter*, for example, cannot know beforehand how many elements will satisfy its predicate), the guess has a very real impact on performance (see Section 4.1.3).

With the *Stream* type established, it is now possible to define the functions *stream* and *unstream* for *Text*. Here again, important differences arise between list streaming and *Text*

streaming. First, the type of the desired stream is always known; a *Text* is a sequence of characters, and therefore *stream* should always yield a stream of *Char*s. Second, although *Text* is a sequence of characters, they are encoded. This byte-level internal representation should not be known to the programmer. Their programming should be with respect to characters, not how they are represented. Therefore, to get a stream of characters, *stream* must decode the contents of *Text*. These requires lead to the following definition of *stream*:

```
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 (U16.chr2 n n2) (i + 2)
      | otherwise                        = Yield (unsafeChr n)    (i + 1)
      where
        n  = arr `unsafeAt` i
        n2 = arr `unsafeAt` (i + 1)
        chr2 (W16#a#) (W16#b#) = C#(chr#
          (upper# +# lower# +# 0x10000#))
          where
            x# = word2Int#a#
            y# = word2Int#b#
            upper# = uncheckedIShiftL#(x# −# 0xD800#) 10#
            lower# = y# −# 0xDC00#
```

This function uses array indices as seeds that say where the next character can be accessed. It uses the offset as the place for the first characters. The stepper traverses the array for the entire length of the string, and then returns Done when it has exceeded the length. There are two different cases where *stream* yields a character, and these correspond to the two possible widths a character can have, one or two *Word16*s. In the case where it only takes up one, it converts that *Word16* to a character and then sets the seed to the next index. If a *Word16* has is between the two test values, it is the first half of surrogate pair. It and the *Word16* following it are combined according to the UTF-16 standard and converted to a character. Then the index after the next is passed on as the seed, as the adjacent *Word16* was used for the current character.

This function makes use of a few other named functions: *unsafeAt*, *unsafeChr*, and *chr2*. The first is a function found in GHC's *Data.Array.Base* module, which provides access to immutable arrays. The reason it is "unsafe" is that, unlike the safe version (which uses the (!) operator), it does not provide bounds checking and does not allow arbitrary indices (they must be 0-based). The use of this, and other "unsafe" operations, provides a significant increase in performance (see Section 4.1.3 for benchmarks).

The reason *unsafeAt* can be used here is that *Text* already promises that there is data contained from the *off* to *off* + *len* indices, so bounds checking for every read is unnecessary. The *unsafeChr* and *chr2* functions are user-defined functions that carry out the conversions from a raw word to a character. Unlike the built-in function *chr*, neither of these functions checks to make sure value being converted is a valid Unicode code point. This is because strings are validated before being converted to a *Text*, so no invalid Unicode values will appear in them (see Sections 3.2.2 and 3.2.4 for operations pertaining to creating *Text*s). This is also why, in the case of a first surrogate, the next value is read without bounds checking on the array and that value is assumed to be a valid second surrogate value. Any isolated surrogates that would

cause these assumptions to fail are removed during validation, and doing unnecessary bounds checking is a performance bottleneck.

Once a stream needs to be written out, *unstream* must deal with the opposite task of encoding a stream of *Char*s into a UTF-16 stream that can be written to an array:

```
unstream :: Stream Char → Text
unstream (Stream next 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)
                         copy arr arr'
                         loop arr' i (max * 2) s
      | otherwise = case next 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
            m, n :: Int
            n = ord x
            m = n - 0x10000
            l, r :: Word16
            l = fromIntegral ((shiftR m 10) + (0xD800 :: Int))
            r = fromIntegral ((m .&. (0x3FF :: Int)) + (0xDC00 :: Int))
```

Obviously, array allocation adds additional complexity when unstreaming to a *Text* versus a list. This version of *unstream* uses the *len* field from the *Stream* to make a guess about how big the array should be. The guess is actually two *Word16*s bigger than the *len* specified. This is because, before another value is yielded, *loop* checks to see that the array is big enough to accommodate the worst case, which is a character that must be encoded as a surrogate pair. Although this is a rather out of order way of performing such a check, performing the check *after* the next value has been calculated is much slower. It is suspected that is due to branch prediction, as the Core language output for GHC revealed no major differences in the code other than checking the guard before unfolding the next value. The benchmarks demonstrating this significant performance difference can be found in Section 4.1.3. If the array is big enough to accommodate the stream without copying, each value yielded is turned into one or two *Word16*s, as necessary. If it is necessary to copy, the loop will allocate a new array twice as large as the original and copy any already-written values to the new array, and then use the new array in the next iteration.

The array itself is allocated as an *STUArray*, or a mutable unboxed array using the *ST* monad. When the loop has finished writing the array, it calls *unsafeFreezeSTUArray* on the array, which converts it to a *UArray*. This function is "unsafe" because it returns a reference to the *UArray* without first copying it, which would be a waste of memory and time. This means that any older references to the array would still allow a function to write to it, violating its immutability. Because this array is allocated, written, and then frozen in the same function

before returning *any* references, it is not possible for any code to write to the array after it is frozen and thus it is a safe and yet faster way to freeze the array. The loop also returns a length value for the actual number of bytes occupied by string that was just written out. This value is used for the length field in the new *Text* to give an accurate boundary of where in the array the string is stored.

In addition to *unsafeFreezeSTUArray*, *unstream* also contains calls to *unsafeWrite* and *unsafeNewArray_*. Like *unsafeAt*, *unsafeWrite* allow access to an array without performing bounds checking on each access. This is already performed by checking that the array is large enough to accommodate the next characters, and this operations is less costly than its "safe" counterpart. The function *unsafeNewArray_* allocates any array with the specified range of indices, but does not initialise array like its safer version. This makes allocating arrays, especially large ones, a much faster operation. This function is also used safely because no values are read from in the new array, and the offset and length fields in *Text* ensure that no other functions will attempt to read from any unwritten areas of the array, either.

3.2.2 Converting between *Text* and *String*

With *stream* and *unstream* defined over *Text*, a fully fusible API can written. The most crucial set of functions in such an API are those that allow the creation of *Text*s, otherwise the other combinators are rather useless. One way to create *Text*s is to convert them from *String*s. This is accomplished using the *pack* and *unpack* functions:

```
pack :: String → Text
pack str = (unstream (stream_list str))
  where
    stream_list s0 = Stream next s0 (length s0)
      where
        next []       = Done
        next (x : xs) = Yield x xs
unpack :: Text → String
unpack t = (unstream_list (stream t))
  where
    unstream_list (Stream next s0 len) = unfold s0
      where
        unfold s = case next s of
          Done       → []
          Skip s'    → unfold s'
          Yield x s' → x : unfold s'
```

Despite the fact that these strings involve both *String* and *Text*, they are implemented using a stream fusion. In the case of *pack*, a specialised stream function converts the list into a *Stream Char*, and then unstream will write it to a *Text*. Similarly, *stream* will convert a *Text* to a *Stream Char*, and a specialised unstream function writes it out to a list.

It would be possible to directly read/write these strings from one format to another, but there are two reasons why this implementation is better. First, *stream* and *unstream* are already highly optimised in order to provide fast encoding and decoding of UTF-16 streams. Second, by implementing these functions using streams, they are fusible. If a *String* is packed only to be immediately transformed, only one *Text* needs to be created: the one containing the final result. Likewise, if unpack is being used at the end of a series of transformations on a *Text*, it is wasteful to write that result out to a *Text* only to then copy to a *String*.

This use of streams reveals their dual purpose. They represent a way to eliminate intermediate structures using the *stream/unstream* fusion rule, but they also serve as a way of

```
append :: Stream Char → Stream Char → Stream Char
append (Stream next0 s01 len1) (Stream next1 s02 len2) =
  Stream next (Left s01) (len1 + len2)
  where
    next (Left s1) = case next0 s1 of
      Done     → Skip (Right s02)
      Skip s1' → Skip (Left s1')
      Yield x s1' → Yield x (Left s1')
    next (Right s2) = case next1 s2 of
      Done     → Done
      Skip s2' → Skip (Right s2')
      Yield x s2' → Yield x (Right s2')
tail :: Stream Char → Stream Char
tail (Stream next0 s0 len) = Stream next (False :! : s0) (len − 1)
  where
    next (False :! : s) = case next0 s of
      Done → errorEmptyList "tail"
      Skip s' → Skip (False :! : s')
      Yield _ s' → Skip (True :! : s')
    next (True :! : s) = case next0 s of
      Done → Done
      Skip s' → Skip (True :! : s')
      Yield x s' → Yield x (True :! : s')
```

Figure 3.2: Examples of stream combinators in *Text*

abstracting away from underlying representations. This enables conversion between different string representations using functions that are already highly specialised to be fast and efficient.

3.2.3 *Text* Combinators

As discussed previously, the *List* API represents the best choice for *Text* to give programmers access to a wide range of functions that they are already accustomed to using with *String*. These functions also represent most of the inductively defined operations possible over a string, so there are very few limitations on what can be accomplished with respect to string processing.

Stream Combinators

One of the benefits of stream fusion is the use of an explicit *Stream* type. A function defined in terms of streams does not depend on the mechanics of the source of the stream. The only functions that interact with the underlying data structure are *stream* and *unstream*. Because of this, the functions in Figure 2.5 work just as well on streams of *Text* as they do on lists, although with minor modifications to account for the length field and the less polymorphic types of *Text* combinators. Figure 3.2 shows some examples of the stream combinators used in *Text* (the entire library can be found in Appendix A). Note the modification of the length guesses based upon the nature of the function in question.

The stream combinators for *Text* are largely taken from two existing sources, the *ByteString* library[7] and the *Data.Stream* library[6], which is a stream fusion implementation of *List*. While the authors of these libraries wrote stream functions that appear in the *Text* library,

```
append :: Text → Text → Text
append t1 t2 = unstream (S.append (stream t1) (stream t2))
tail :: Text → Text
tail t = unstream (S.tail (stream t))
```

Figure 3.3: Examples of stream-defined *Text* combinators

```
tail :: Text → Text
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
init :: Text → Text
init (Text arr off len) | len ⩽ 0 = errorEmptyList "init"
    | n ⩾ 0 xDC00 ∧ n ⩽ 0 xDFFF = (Text arr off (len − 2))
    | otherwise                  = (Text arr off (len − 1))
  where
    n = unsafeAt arr (off + len − 1)
```

Figure 3.4: Examples of non-stream *Text* combinators

their focus was different and there are many modifications to the original function definitions. *ByteString*, being a fixed width encoding, can read from either end of string. This allowed functions such as *tail* and *init*, which return all but the first element and all but the last element of a list/string, respectively, to be defined using the same stream combinator but using different streaming functions. In *Data.Stream*, functions were designed to read from the start of the list only, but were also designed to handle the polymorphism (including nested lists) and laziness of the list type, neither of which are features of *Text*.

The *Text* combinators implemented in terms of *stream* and *unstream* have a simple structure; stream in the input *Text*(s) and unstream the output stream, if it exists. The *Text* combinators for the stream combinators in Figure 3.2 are shown in Figure 3.3.

Non-stream Combinators

Stream-based combinators must use *stream* and *unstream* to achieve the necessary conversions. This involves reading, decoding, and then copying and re-encoding the entire string. Some operations do not require this, and can save time and space by being implemented without using streams. Examples of such functions are show in Figure 3.4.

The *tail* function only needs to remove the first character from the string. To do this, it can decode the first character and return a new *Text* with the same *UArray* but a different offset and length. This lets the two *Text*s share the same array, but represent two different strings. It also changes the time and space complexities of *tail* from $O(n)$ to $O(1)$. Similarly, *init* only needs to decode the last character of the string and modify the length field accordingly (depending of whether or not the last character is a single *Word16* or a surrogate pair).

While the non-stream functions are more efficient when called in isolation, they lose their

fusibility. It may be more efficient to implement *tail*, etc. on the structure *Text*, but those gains are less valuable if they force the creation of an intermediate *Text*. Also, if a *Text* has already been streamed for another function, there is no additional copying or complexity to apply the stream version of these non-stream versions.

The perfect solution is to have the compiler choose the stream version of the function when that is more efficient, and otherwise choose the non-stream version. This can be achieved through careful use the GHC's *RULES* pragma. The following example is the set of a rewrite rules for *tail*:

`"TEXT tail -> fused"` $[\sim 1]$ *forall* t.
  $tail\ t = unstream\ (S.tail\ (stream\ t))$
`"TEXT tail -> unfused"` $[1]$ *forall* t.
  $unstream\ (S.tail\ (stream\ t)) = tail\ t$

The first rule substitutes *tail* for the streamed version of it, streaming the input and unstreaming the output. The $[\sim 1]$ restricts this rule so that it only applies before Phase 1 of compilation. The second rule, which is applied only during Phase 1 and after, checks to see if the function has been fused. If it has been call to *stream* will have been removed by the *steam/unstream* rewrite rule, that function call should be left alone. If it remains unfused, then it substitutes the non-stream version back in for the stream version, because it will be more efficient.

3.2.4 File I/O

The issue of file input/output was complicated by the fact that the way that various text encodings were handled was due to change in the upcoming release of GHC 6.10. These have now been clarified[1], and GHC will provide handle based I/O of Unicode text for all encoding form. To avoid conflict with these features, which will undoubtedly be utilised in *Text*, the I/O implemented for *Text* deviates from Haskell's *String* model.

The functions *encode* and *decode* provide ways to convert a *Text* into a *ByteString* and vice versa. This gives then gives the programmer access to *ByteString*s extremely fast I/O. The Unicode values are not lost, however, because *ByteString* are not used in their typical one-character-byte matter. Instead, the characters are written out encoded, so that a given byte of a *ByteString* may or may not represent a whole character on its own. The *Encoding* allows the programmer to choose encoding form and endianness. From there it can be written to out a file using *ByteString*'s file I/O. library. The source for these functions is available in Appendix A.

These functions also utilise stream fusion as the mechanism for their conversion. The function *stream_bs* acts a stream function from *ByteString* to *Stream Char*, decoding and performing full Unicode validation. When converting back, *restream* converts a *Stream Char* to a *Stream Word8* using a specified encoding. Then, *unstream_bs* acts as the unstream function. This allows the conversions to take place without any needless intermediate *Text*s. In fact, a program that uses ByteString for reading from a file, performing a series of fusible functions, and uses ByteString to write back out to a file, never needs to write any *Text*s at all. This may lead to the conclusion that *Text* is an unnecessary representation, and that a layer over *ByteString* is sufficient for all these operations. Indeed, *Text* is very similar to *ByteString*, but diverges on the rather significant choice of storage. The memory allocation issues associated with *ByteString* can make it a less desirable choice in some circumstances (see Sections 2.1.2).

Chapter 4

Benchmarking and Testing

The goal of *Text* is to be an alternative to *String* that achieves better performance in time and space. Therefore, evaluating the success of *Text* depends on knowing precisely how much faster it is. This chapter presents the benchmarking system used to the compare the performance of *Text*, *String*, and *ByteString* (where applicable). The design of such a system in Haskell has the additional complication of dealing with laziness, particularly in benchmarks of *String*, which is a lazy data structure.

This chapter also presents the results of these benchmarks. It explores the reason for the performance of some of the representative combinators. It then discusses the performance impact of certain implementation decisions that were made in the process of developing *Text*. It also compares *Text*, which uses UTF-16 as its underlying representation, to alternative implementations that use UTF-8 and UTF-32.

The second half of the chapter covers the testing methods used for *Text*. It gives an overview of the testing library used, known as QuickCheck. It then discusses the validity of QuickCheck's methods for testing, and how these methods are implemented specifically for *Text*. It also shows examples of some the tests used to help verify *Text*.

4.1 Benchmarking

The goal of the benchmarking system for *Text* is to be able to compare its performance with *String* and *ByteString*. That performance is quantified by measuring the amount of time required to perform the same task on the same piece of text using each representation and their respective functions. A representation that takes less time than the others is said to have better performance.

The "time" specified here is not *real* time. The amount of time that elapses as a task is performed is highly non-deterministic; it depends on the resources allocated to that program for a given period of time. This depends on the configuration of the operating system's scheduler and what other tasks are competing for resources. Even in the instance where these two factors are the same, the choice of the scheduler may still be non-deterministic.

The amount of *processor* time a task takes, however, is far more regular. A deterministic function will always require the same resources to perform the same task. This means that it will require the same number and type of machine instructions in exactly the same order every time. The amount of time each of these instructions takes can be measured, and the sum of these times is processor time, or "CPU time". This time more readily reflect the notion of how long a function is taking to perform its task, and this is the quantifier that is used to compare the performance of each library in these benchmarks.

Haskell's laziness adds an additional complication in measuring performance. First, Haskell uses call by need parameter passing for functions, meaning that the arguments to a function may not have been evaluated yet. In measuring performance of a given function, the overhead of evaluating its arguments should be included. The arguments could be arbitrarily complex expressions, and their performance is not at all dependent upon the function itself. Thus, all arguments must be *fully* evaluated before being passed the calling function. Furthermore, Haskell's *String* type itself is lazy. Applying a function to a *String* does not always yield full evaluation of the resulting expression, and forcing evaluation fairly is not straightforward on lists.

Forcing evaluation "fairly" refers to forcing evaluation without incurring extra overhead. As an example, *map* and *cons* are both functions that return new *String*s, but lazily. The *map* function is supposed to traverse the entire list and compute a new value for each existing value. Forcing this list means traversing the entire list *and* inspecting each value. The *cons* function, however, does not traverse the list at all. It merely creates another node and adds it onto the front of the list without traversing it. To traverse and inspect every value of the new list incurs and overhead that the *cons* function does not cause. For this function, it suffices to put the list into Weak Head Normal Form (WHNF). WHNF does not guarantee that the entire list is evaluated, but that evaluation occurs until the outermost expression is not a reducible expression. This differs from Head Normal Form, which would also ensure that the inner expressions of functions are also not reducible. In this case, *cons* x xs becomes the list $x : xs$, but it is possible that neither x nor xs is evaluated. In benchmarking, this ensures that the *cons* operation is performed, but keeps the complexity or the size of x or xs from interfering with time, which would be inconsistent with the constant time complexity of *String*'s *cons*. Forcing evaluation on *Text*s and *ByteStrings* is much easier. These data structures are internally strict, so forcing them into WHNF also forces any other necessary evaluation.

4.1.1 Benchmark Implementation

Because of the similarities between *Text* and *ByteString*, the benchmark system for *Text* is based upon a similar system developed for testing *ByteString*[7]. This system included tests over ASCII text and strategies for forcing the evaluation of arguments. It was only used for *ByteString*, though, and did not need to address the issue of evaluating lazy results such as *String*s.

The code for this benchmarking system is available in Appendix A. It reads in a text file in each of the necessary formats. It then forces the evaluation of resulting string in each format. This forces all three files fully into memory, so that no I/O occurs during the actual timing process. Then, each version of a given function is applied to its respective representation. The result is forced based upon its representation or result. To force a value into WHNF, the function *seq* is used. For lists whose elements each need to be evaluated (e.g. map, filter), the function *foldl'* (*flip seq*) (*return* ()) is used. The latter forces each element of the list into normal form. In the case of *String*, this forces each element to be evaluated until the resulting *Char* is computed.

Corpus selection

The selection of test files are an important consideration. Documents using different Unicode code points can have different impacts on performance. For UTF-16, there are 2 possible cases for each character; it may take up one word, or it may form a surrogate pair. To evaluate the differences that each of these cases can have on performance, the tests are performed both on documents that have characters solely in the BMP and on a document that consists characters

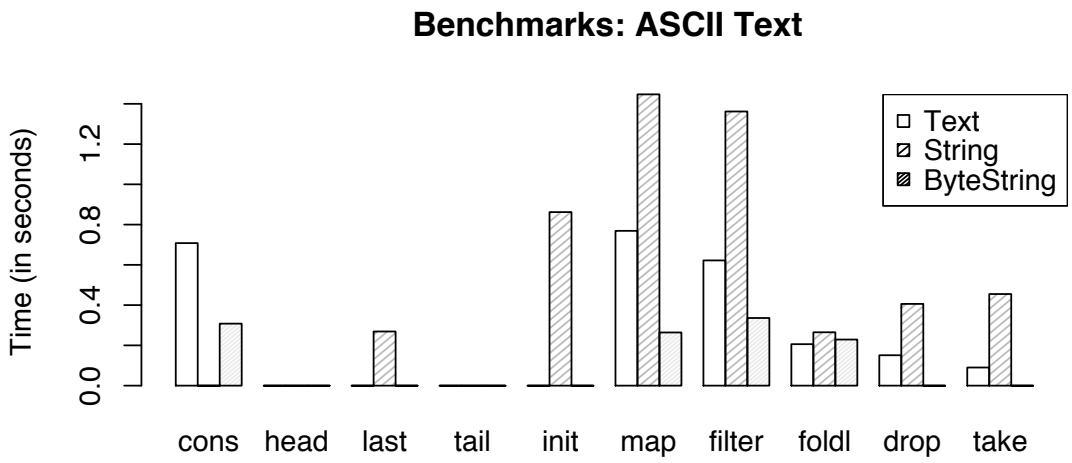


Figure 4.1: Benchmarks of *Text*, *String*, and *ByteString* on a 57.7 MB ASCII file

outside of it. Because ASCII characters are pre-allocated in *String*, a document consisting of only ASCII characters is also tested. This also allows for comparisons with *ByteString*.

Each of the documents tested is close to 50 megabytes. The ASCII text is the English version of the *The Universal Declaration of Human Rights*[2] concatenated to itself many times to reach the desired size. The BMP version is the same document, except in Russian[3], also concatenated to itself many times. The reason for this is that the Cyrillic alphabet used by the Russian language is completely outside the ASCII range, but still inside the BMP. It tests the performance of *Text* versus a non-pre-allocated version of *String*. The outer planes document is an automatically generated document of all possible values in the first and second planes, also concatenated to itself to make a suitably sized document. This was used because the availability of actual corpora containing many of these characters is limited. The occurrence of CJK characters in these planes is extremely rare, so that in a realistic document they would have a negligible impact on performance. The only other realistic type of corpus would be one in an ancient script such as Linear-B, Gothic, or Old Italic. Finding a suitable document in this form was not a realistic solution.[1]

4.1.2 Results

Single function results

The results presented in this section are a selection of functions taken from benchmarking. A run of all benchmarked functions is available in Appendix B. Benchmarks were taken for three different cases: ASCII text, BMP text, and SMP/SIP text. No distinction was made between the SMP and SIP planes because the conversion process in UTF-16 is the same. Figures 4.1, 4.2, and 4.3 show the performance over a sample of each of these texts, respectively.

Generally, *Text* beats *String* regardless of encoding. Some of these, such as *last* and *init*, take advantage of the ability to create substrings that share the original array. That, combined with constant time access to any member of the array, reduces the complexity of these operations from $O(n)$ to $O(1)$. Others, such as *map*, *filter*, and other $O(n)$ operations over lists, are

[1] There are very few free corpora. Of the few that are free, most are only accessible through a "concordancer", which allows the user to search for words and see the context in which they appear.

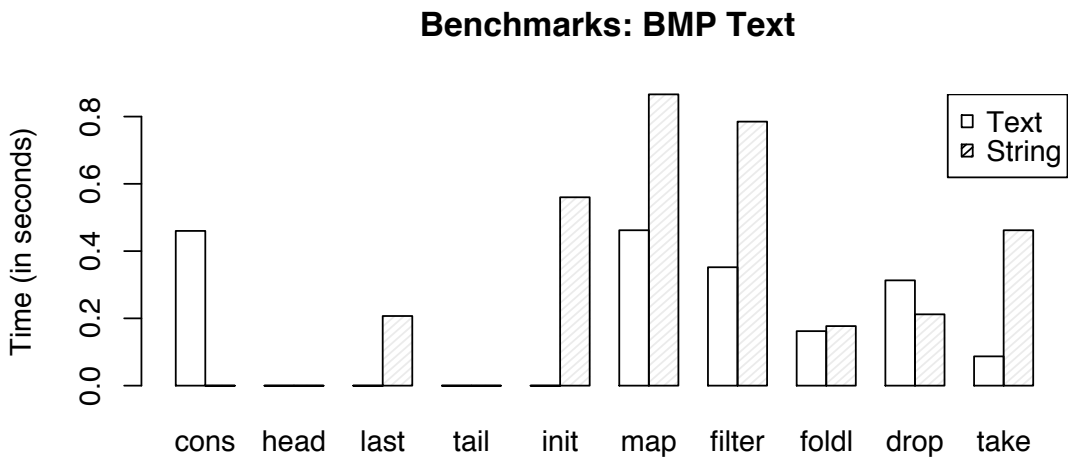


Figure 4.2: Benchmarks of *Text* and *String* on a 61.2 MB Unicode BMP file

implemented in terms of streams regardless of whether or not they fuse with other functions. The notable exception to *Text*'s performance is *cons*. The reason for this is that *Text*'s array based representation requires creating a new array, inserting the new element at the index, and then copying all of the old array. In *String*, its linked-list structure allows this operation to be completed in constant time.

In the ASCII test, *Text* and *String* are also compared to *ByteString*. *ByteString*'s performance represents an "ideal" for *Text*, but is difficult to reach because of the added Unicode overhead. *ByteString*'s complexity is the same as *Text*'s for most operations, hence the poor *cons* performance. While *ByteString* is still much faster than *Text*, it is worth noting that *Text*'s performance tends to be closer to *ByteString*'s than to *String*'s. *ByteString* has a huge advantage in *drop* and *take*, however, because these operations are constant time; *Text* has to check for surrogate pairs when forming substrings, whereas *ByteString* does not.

The results for tests on BMP text are roughly the same. This is expected, as no additional decoding overhead is incurred for any BMP text using UTF-16. In this test there is no *ByteString* comparison because *ByteString* is incapable of storing these Unicode values.

Performance is drastically different in SMP/SIP text, however. While *Text* still beats *String*, the margin is far smaller. The constant time functions maintain their same performance, but those functions that must traverse the string are much slower. The reason for this is that there is increased decoding overhead for this text; each character comprises two *Word16*s that must be assembled using a series of shifts and adds. Although *Text* suffers quite a bit in this example, it represents a worst case scenario rather than a likely case. In documents using any modern script, even those with CJK characters, there would be very, very few characters from the outer planes. Rather, they would be interspersed in the document, and their impact on performance would be much smaller, if not negligible. The only realistic case that this represents is a large document composed entirely of an ancient script.

Fusion results

The single function results demonstrate that *Text* has good performance; it beats *String* nearly all the time. A big part of developing *Text*, though, was is making sure that its combinators are fusible. Furthermore, programs using compositional functions are so common that their impact

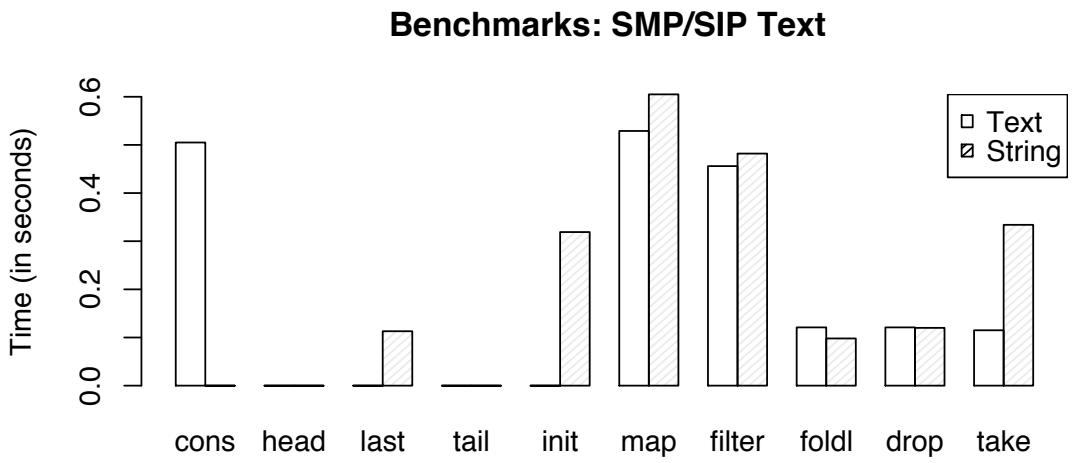


Figure 4.3: Benchmarks of *Text* and *String* on an 82.5 MB Unicode SMP/SIP file

on performance is an important measure of a library's speed. Figures 4.4 shows the timings for benchmarks of some common fusion patterns.

The fusion benchmarks reveal that fusion does not yield as dramatic a performance increase as single function calls. One reason for this is that *String* also implements fusion, so the gain on *String* is not as significant as it might otherwise be. Nevertheless, *Text* is consistently better that *String*, but there may be room for improvement to get closer to *ByteString*'s fusion performance. Note the sharper difference in performance when using *foldl*. This is due to *String*'s inability to use left folds using *foldr/build* fusion.

4.1.3 Impact of optimisations

Throughout the development of *Text*, many changes were made from the original design for performance purposes. The impacts of these changes on performance are quantified in this section to prove the efficacy of changes to the system. They can broadly be classified into three sets of changes: length guessing, "unsafe" function use, and branching changes.

Length guessing

One problem with an array-based representation is that the size of the array must be chosen before it is filled. If a *Text* is the result of a function, its array will be allocated and its sized fixed before the resulting string is computed. Obviously, if the array is well sized, this is not a problem. If the array is too big, the string will fit, but memory is wasted. Although this representation is more compact than *String*, wasted memory is still something to avoid. If the array is too small, the string will not fit. To deal with situations where this happens, functions that write new arrays (e.g. *unstream*) detect when it reaches the end of any array, allocates one twice as big, copies the already written contents, and continue. Doubling the size keeps the amortised cost of writing the whole array down to $O(n)$, but the point of arrays is that reads and writes are supposed to be $O(1)$.

The solution to this problem was to implement length fields in the necessary data types. In *Text*, this field is also used to make substrings. The more important addition was in the addition of a length field to the *Stream* type. Without this field, it is impossible to know anything about

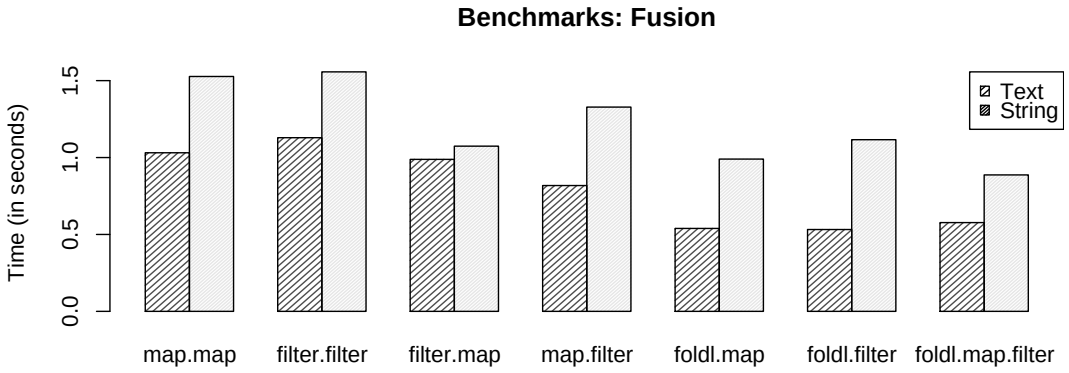


Figure 4.4: Benchmarks of fused functions for *Text*, *String* and *ByteString* on a 57.7 MB ASCII file

the length of a stream before it is unfolded. This additional field communicates the size of the original *Text* through the streaming process. It also lets stream-based combinators modify the length field if is known how the length will change (e.g. *cons* always adds exactly one character). The exact quantification of a how much length guessing affects performance depends on the size of the arrays without length guessing. Using a suitably large number, it could yield the same performance if the array is always bigger than any string that tries to fit into it. The amortised cost of the size-doubling method is still $O(n)$ even if the guess is very small, so the comparison here is against an implementation that always starts with an array of size 1.

Figure 4.1.3 shows a set of timings with length guessing versus a set without. The effect of so much copying is massive, causing slowdown by as much as a factor of 20. This impact would be lessened with a larger base guess, but it is clear that length guessing is pivotal is taking an array-based approach from unusably slow to a high-performance representation.

Using "unsafe" functions

One of the hallmark characteristics of Haskell is its emphasis on correctness. Haskell's ability to abstract away from machine level coding is one of the ways that it achieves this. Memory management, mutability, and pointers are all handled by the compiler and run-time system so that the programmer does not have to worry about them. These areas are extremely prone to error by programmers and distract away from focussing on the actual computation. In addition, this abstraction provides consistency across platforms.

This abstraction, however, does incur a performance cost. Array reads and writes are always bounds checked, which makes sure that no other data are overwritten. Furthermore, arrays in Haskell can use arbitrary indices, a convenient way to avoid, for example, off by one errors from 0-based arrays. Conversion between characters and numbers, far from the simple casting procedure they are in C, require using boxed (i.e. heap-allocated) numbers and characters, and numbers are always checked against Unicode bounds during conversion.

For a high-performance library, being able to control what a program does at a lower level can yield much better performance. Such control is available in the form of "unsafe" functions. These functions are considered "unsafe" not because using them inherently breaks a program, but it removes the ability for Haskell to protect the programmer from making errors it could

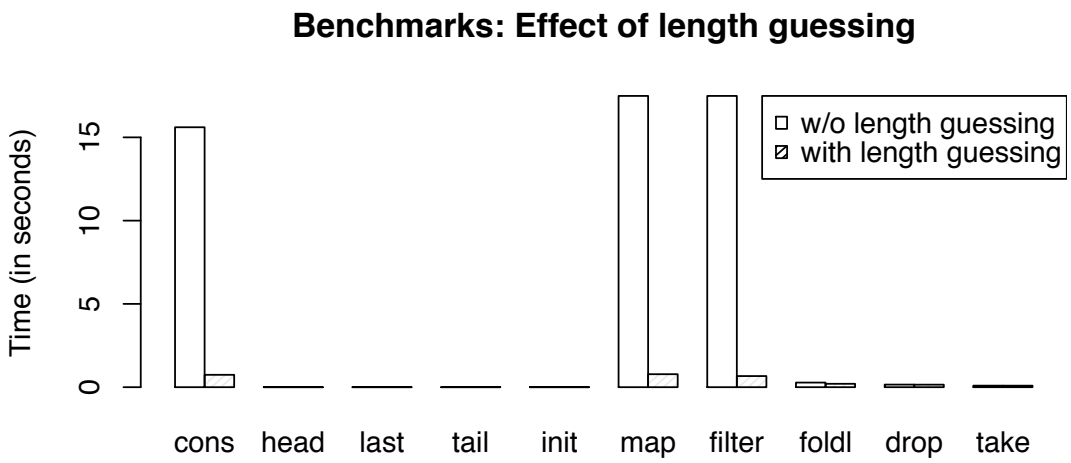


Figure 4.5: Benchmarks of *Text* without any length guessing

otherwise detect. Because most "unsafe" functions have a safe counterpart, it is a simple matter to make a safe prototype to test the correctness of a program and migrate to "unsafe" functions one at a time when the prototypes correctness is assured. This is the process that was used in refining *Text*.

The "unsafe" functions used in *Text* were for array allocation, reading, and writing and for conversion from numbers to text. The latter is via a user-defined function rather than built-in one. Figure 4.6 compares the performance of a "safe" version of *Text*. The impact on performance is not as large as that of length guessing, but it is nevertheless significant. Using safe array and character functions slows downs performance by factors of 2 to 3.

Reducing comparisons and branches

The number of comparisons and branches in the *Text* code was reduced drastically from the prototype. This was due to the fact that all characters are validated at the the time that they are first read into a *Text*, so any functions on *Text* only deal with valid characters. Thus, comparisons are only *necessary* to check for surrogate pairs, but not to check boundaries of characters as they are are encoded/decoded.

In addition to reducing the number of comparisons, reducing the number of branches yields a performance increase. This was accomplished by removing nested branches for checking array boundaries when reading to a new *Text*. Instead, this check was floated out to take place before the next character was read from the stream. While this leads to sometimes allocating 1-2 extra 16 bit words, the performance gain combined with the otherwise compact representation of *Text* means this trade-off has no negative consequences.

Figure 4.7 shows the comparison between a version of *Text* using full value checking and checking for array bounds after the next value has been yielded. The difference is far less dramatic than for the prior optimisations. The slowdown is between 1 and 2 times for most functions. In most cases, *Text* still beats out *String*, but by a smaller margin.

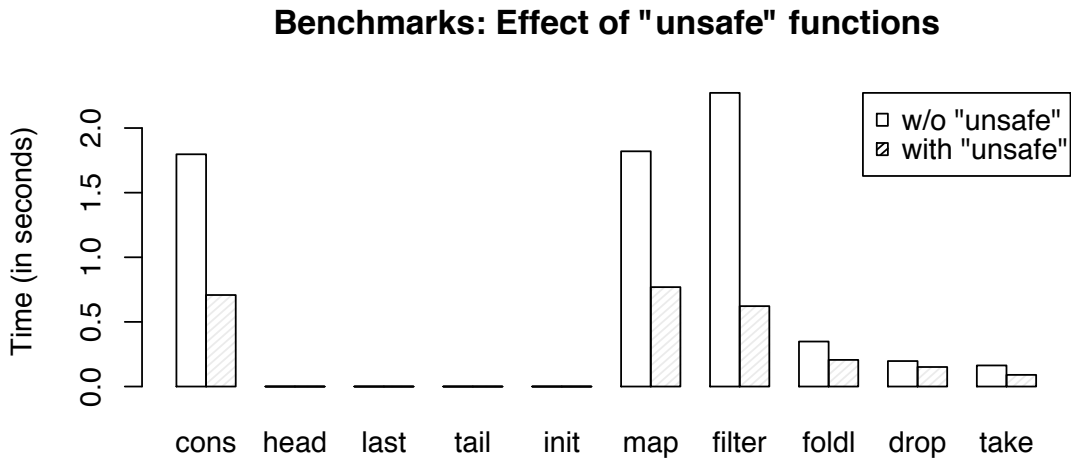


Figure 4.6: Benchmarks of *Text* using the safe form of its "unsafe" functions

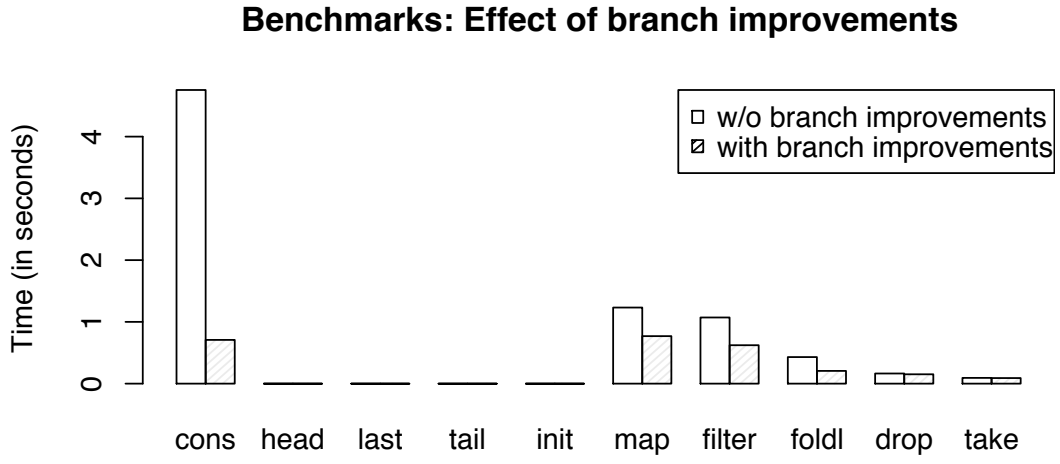


Figure 4.7: Benchmarks of comparing *Text* with and without branching improvements

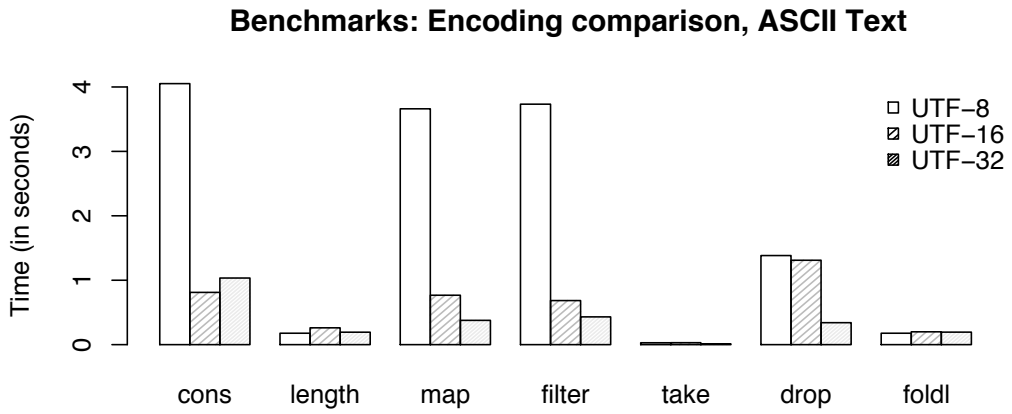


Figure 4.8: Comparison of different encoding implementations of *Text* on ASCII text

4.1.4 Encoding shootout

The advantages of UTF-16 as a relatively compact (compared to UTF-32) but also relatively simple (compared to UTF-8) encoding were the reasons it was chosen for *Text*. This is true from a theoretical point of view, but evaluating the actual impact this has on performance is important. Figures 4.8, 4.9, 4.10 show the timings of some common list combinators for *Text* implemented with various encodings.

The timings reveal that in the case of ASCII Text, UTF-8 is a very poor performer. This is probably to due to issues with branching in the UTF-8 implementation. Similar problems were encountered in *Text* using UTF-16 (see Section 4.1.3), and UTF-8 has more complicated branching. It is unlikely, though, that it has anything to do with decoding overhead because no ASCII text needs to be reassembled in any Unicode encoding. It may be possible to increase UTF-8 performance in this case, but it would be significantly more effort than with UTF-16. UTF-32, representing the simplest possible way to handle text, was on par or slightly faster than UTF-16. Again, these differences have more to do with branching and comparisons than any character reassembly.

In BMP text, UTF-16 is significantly faster than UTF-8. UTF-32 is predictably faster, though. These results are not too different from the difference among the encodings in the ASCII tests, and this is to be expected; UTF-16 and UTF-32 handle ASCII and BMP text the same. UTF-8 still performs poorly, but it is difficult to say whether this is due to branching issues or the overhead of reassembling characters.

In SMP/SIP text, the gap between UTF-16 and UTF-32 is far larger. This is due to the overhead in UTF-16 of reassembling characters, whereas UTF-32 does no reassembling whatsoever. UTF-8 still has very poor performance.

This comparison shows that UTF-8, while more compact in many cases, pays for its complexity. In this case, the implementation used the same optimising strategies that were used in writing the UTF-16 code, but this was clearly insufficient. Even though UTF-32 is faster than UTF-16, sometimes much faster, it's large footprint significantly reduces its usability. In many cases, the distance between UTF-32 and UTF-16 is still far smaller than the gap between UTF-16 and UTF-8, making it harder to justify doubling the space needed for a string.

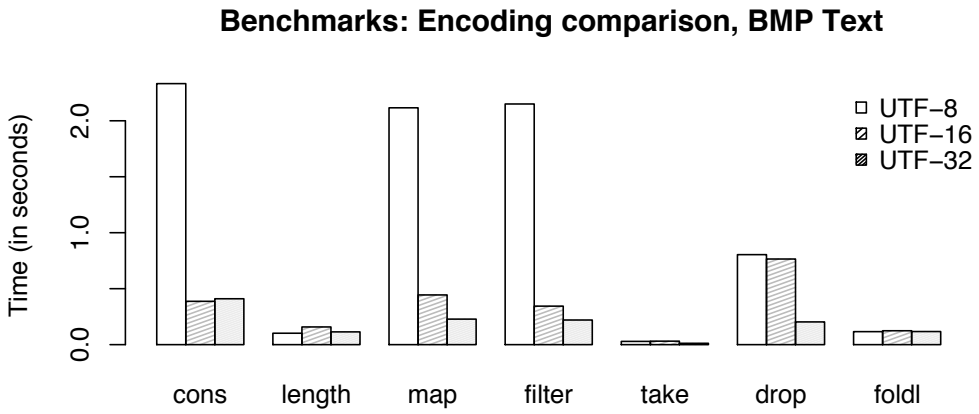


Figure 4.9: Comparison of different encoding implementations of *Text* on ASCII text

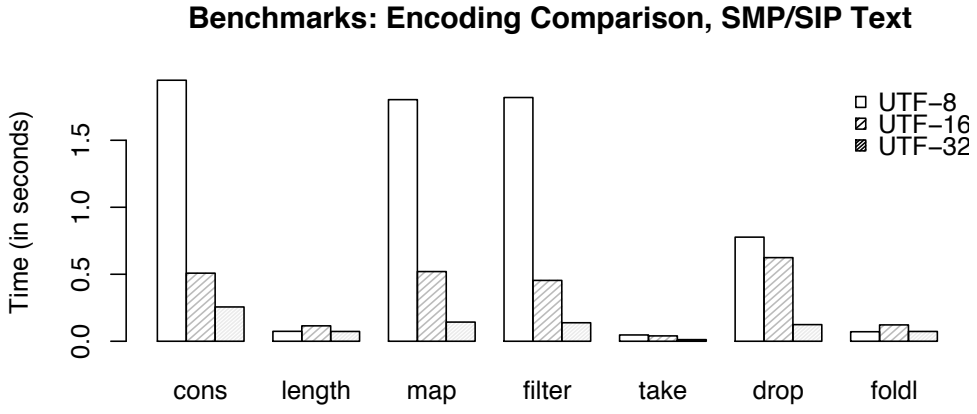


Figure 4.10: Comparison of different encodings representing strings in *Text*

4.2 Testing

In addition to measuring the performance of *Text*, it is important to ensure that it behaves as expected. It cannot be used for a string library if its representation is not trustworthy. Although no testing system can completely validate a program, it is negligent to forgo testing for this reason. Testing can find many errors that would be otherwise missed by the compiler and by examining the source code.

The testing system used for *Text* is designed to test that certain desired properties hold. Because *Text*'s library is designed to mirror *String*s library, there is already a "correct" version of each function that can be considered trustworthy. In testing the behaviour of library functions, comparing the *Text* output to the *String* output is a convenient way to see if that tests are successful. A testing system known as *QuickCheck*[5] has been developed that allows programmers to specify properties and design a testing framework that can provide good coverage of the test space and generate many more test cases than a programmer could do manually in the same amount of time.

4.2.1 Testing using QuickCheck

The *QuickCheck* library allows the specification of properties, which it can then check by generating a series of test inputs to try disprove the property. If *QuickCheck* does not find a disproving case, it considers the property true. A test case takes the form of a Haskell expression that evaluates to some boolean. Consider this property for *String*-to-*Text*-to-*String* identity:

$$prop\_pack\_unpack\ s = (unpack \cdot pack)\ s \equiv s$$

This property can then be checked by *QuickCheck*:

$$> quickCheck\ prop\_pack\_unpack$$
$$OK, passed\ 100\ tests\cdot$$

Although this code passed, *QuickCheck* also reveals information when failure occurs. In addition to signalling the failure, it provides the test case the cause the failure to aid in debugging.

These tests are performed by generating random strings of varying length (always including the empty list) and testing the equality. The control of how these strings are generated is what ensures good test coverage. *QuickCheck* already knows how to randomly generate lists of anything, it needs to know how to generate random *Char*s for the list. This can be specified using the *Arbitrary* type class:

```
instance Arbitrary Char where
  arbitrary = oneof [choose ('\0', '\55295'), choose ('\57334', '\1114111')]
```

This code defines the function arbitrary, which *QuickCheck* uses to generate random values. This definition tells *QuickCheck* to pick any valid Unicode code point. It has an equal chance of picking one from above or below the value of surrogate pairs, and then an equal probability of picking any character in that range. While many of these code points have not been assigned by the Unicode standard, this does not discount them as valid test cases, because all code points should be treated the same. Providing test coverage across the entire Unicode spectrum is important to explore both cases of UTF-16 (both single characters surrogate pairs).

Now that *pack* and *unpack* are considered safe conversion tools, it's possible to use them in creating arbitrary instances of *Text*[2]:

[2]The instances of *Arbitrary* for *Text* are based off of those found in the *ByteString* testing library

```
instance Arbitrary Text where
  arbitrary = pack `fmap` arbitrary
```

This creates an arbitrary *Text* by creating an arbitrary string and packing it. Now it is possible to check the correctness of other functions in the *Text* library. For example, the proper to test map is:

$$prop\_map\ f\ s = (P.map\ f\ s) \equiv (unpack \cdot T.map\ f \cdot pack)\ s$$

This creates a string, and a random function of type $(Char \rightarrow Char)$ over it, and computes it using *String* and *Text*.

All of the other *Text* API functions are also tested. In the case where there are both stream and non-stream version, both are tested against *String* functions. These tests are all available in Appendix A and the test output is available in Appendix C.

Chapter 5

Conclusions

The goal of this project was to create a fast, compact string representation in Haskell that supports Unicode while also utilising stream fusion. It used some prior work both in string representation and in fusion to create an additional library, *Text*, that fills a large gap in the current Haskell libraries. A benchmarking system was devised that to measure *Text*'s performance versus the competition. It reached the desired level of performance, being significantly faster than *String* in performing string manipulation using *List*-like combinators. Although *ByteString* remains faster, its uses are divergent from *Text* and the complement each other nicely, with *ByteString* providing fast byte-level access and *Text* providing an abstraction away from it while still maintaining good performance.

In addition to creating a practically useful product in *Text*, this project also revealed some of the useful aspects of stream fusion that go beyond its original intention. Stream fusion's explicit representation created an opportunity to use multiple sequence types while transparently converting among them. *ByteString*, *String*, and *Text* types are all fusible when working with this library, and the underlying mechanics are completely transparent to the programmer. The combinators used, all defined in terms of streams, are also indifferent to the underlying source of the text they are computing.

This was a great benefit to the design of *Text*, and also made optimisation easier. The majority of functions depend only on a stream for their computation and not the design of *Text*. This meant that function definitions over streams were straightforward, and most of the overhead was concentrated into the *stream* and *unstream* functions. Targeting these functions for optimisation was extremely effective and much more efficient than spreading more low-level code around to each function. Thus, the use of streams for converting between sequences proved not only to be useful from a fusion point of view, but also turned out to be an elegant design decision by concentrating the bottlenecks into fewer, more easily identifiable spots.

5.1 Further Work

The *Text* library described in this dissertation represents a first design iteration that joins ideas from previous with new ones. While the result can be deemed successful insofar as it has accomplished it's goal of being a fast, compact string representation, this does not mean it has reached its full potential. The subset of *List* functions that were implemented for this project were chosen for their ubiquity in string processing and their ability to represent a set of similar functions. Nevertheless, a more developed library would have all of the necessary functions. Furthermore, many of the more complex functions, such as *mapAccumR* and *mapAccumL*, pose significant challenges in stream fusion. Overcoming the difficulties in implementing these

functions with good performance would be beneficial to other uses of stream fusion as well as to the development of the *Text* library.

There are also variations on *Text*'s design that were not explored in this project, but be useful. For example, a next obvious refinement is the replacement of the *UArray* in *Text* with a *MutableByteArray#*. This would remove a level of pointer indirection and decrease the size of a *Text*.

5.1.1 Lazy *Text*

The design of *Text* is such that it is inherently strict. This strictness is useful in terms of compactness and performance. Haskell's laziness adds to its usefulness as a very power functional programming language, and *Text* cannot completely replace some of the features of lazy strings. In particular, lazy I/O can be useful for deal with text that is larger than available memory, and *Text* is not capable of doing this, yet. A lazy version of *ByteString* uses a lazy list of strict *ByteString* chunks to accomplish laziness; it may be possible to adapt this to *Text* as well.

5.1.2 *Text* Ropes

Currently, *Text* uses an array-based representation for strings. While this representation was used because it had the potential to achieve the best possible performance, much of the design effort in this project was dedicated to overcoming the disadvantages of arrays, such as copying and fixed size. A Rope[4] is a heavy weight representation of strings that uses trees of arrays to represent strings. It proves a dynamically resizable structure and some easy string operations, such as constant time appending. The work in array-based representations using *Text* could be applied to such a data structure and yield a more flexible representation.

Appendix A

Source code

This appendix contains all of the source code for the implementation and testing of *Text*. This includes all of the modules of *Text* and the UTF-32 and UTF-8 internal implementations for the encoding shootout. It also includes the source code for all of the benchmarks run on *Test*, including the fusion tests for the encoding shootout. It also includes the *QuickCheck* properties and *Arbitrary* instances used in testing *Text*.

A.1 *Text* source code

A.1.1 *Text.hs*

```
{-# OPTIONS_GHC -fglasgow-exts -fbang-patterns #-}

module Text where

import Prelude (Char,Bool,Int,Maybe,String,
                Eq,(==),
                Show,showsPrec,show,not,
                Read,readsPrec,read,
                (&&),(||),(+),(-),($),(<),(>),(<=),(>=),(.),(>>=),
                return,otherwise,seq,fromIntegral)

import Char
import Data.Word
import Data.Bits
import qualified Data.List as L
import Data.Monoid

import Data.Array.Base
import Data.Array.ST
import Control.Monad.ST
import Data.Word

import qualified Data.ByteString as B
import Data.ByteString(ByteString)
import System.IO hiding (readFile)

import qualified Text.Fusion as S
import Text.Fusion (errorEmptyList)
import Text.Fusion (Stream(..),Step(..),Encoding(..),
                    stream,unstream,stream_bs,unstream_bs,restream)
import Text.Internal
import qualified Prelude as P
import Text.UnsafeChar
import Text.Utf16 as U16

instance Eq Text where
    t1 == t2 = (stream t1) `S.eq` (stream t2)

instance Show Text where
```

```
    showsPrec p ps r = showsPrec p (unpack ps) r

instance Read Text where
    readsPrec p str = [(pack x,y) | (x,y) <- readsPrec p str]

instance Monoid Text where
    mempty  = empty
    mappend = append
    mconcat = concat



-- ----------------------------------------------------------------------------
-- * Conversion to/from 'Text'

-- | /O(n)/ Convert a String into a Text.
--
-- This function is subject to array fusion, so calling other fusible
-- function(s) on a packed string will only cause one 'Text' to be written
-- out at the end of the pipeline, instead of one before and one after.
pack :: String -> Text
pack str = (unstream (stream_list str))
    where
      stream_list s0 = S.Stream next s0 (P.length s0) -- total guess
          where
            next []     = S.Done
            next (x:xs) = S.Yield x xs
{-# INLINE [1] pack #-}
-- TODO: Has to do validation! -- No, it doesn't, the

-- | /O(n)/ Convert a Text into a String.
-- Subject to array fusion.
unpack :: Text -> String
unpack txt = (unstream_list (stream txt))
    where
      unstream_list (S.Stream next s0 len) = unfold s0
          where
            unfold !s = case next s of
                          S.Done       -> []
                          S.Skip s'    -> seq s' $ unfold s'
                          S.Yield x s' -> seq s' $ x : unfold s'
{-# INLINE [1] unpack #-}

-- | Convert a character into a Text.
-- Subject to array fusion.
singleton :: Char -> Text
singleton c = unstream (Stream next (c:[]) 1)
    where
      {-# INLINE next #-}
      next (c:cs) = Yield c cs
      next []     = Done
{-# INLINE [1] singleton #-}

decode        :: Encoding -> ByteString -> Text
decode enc bs = unstream (stream_bs enc bs)
{-# INLINE decode #-}

encode         :: Encoding -> Text -> ByteString
encode enc txt = unstream_bs (restream enc (stream txt))
{-# INLINE encode #-}

-- ----------------------------------------------------------------------------
-- * Basic functions

-- | /O(n)/ Adds a character to the front of a 'Text'.  This function is more
-- costly than its 'List' counterpart because it requires copying a new array.
-- Subject to array fusion.
cons :: Char -> Text -> Text
cons c t = unstream (S.cons c (stream t))
{-# INLINE cons #-}

-- | /O(n)/ Adds a character to the end of a 'Text'.  This copies the entire
-- array in the process.
-- Subject to array fusion.
snoc :: Text -> Char -> Text
snoc t c = unstream (S.snoc (stream t) c)
```

```
{-# INLINE snoc #-}

-- | /O(n)/ Appends one Text to the other by copying both of them into a new
-- Text.
-- Subject to array fusion
append :: Text -> Text -> Text
append (Text arr1 off1 len1) (Text arr2 off2 len2) = Text (runSTUArray x) 0 len
    where
      len = len1+len2
      x = do
        arr <- unsafeNewArray_ (0,len-1) :: ST s (STUArray s Int Word16)
        copy arr1 off1 (len1+off1) arr 0
        copy arr2 off2 (len2+off2) arr len1
        return arr
            where
              copy arr i max arr' j
                  | i >= max  = return ()
                  | otherwise = do unsafeWrite arr' j (arr `unsafeAt` i)
                                   copy arr (i+1) max arr' (j+1)
{-# INLINE append #-}

{-# RULES
"TEXT append -> fused" [~1] forall t1 t2.
    append t1 t2 = unstream (S.append (stream t1) (stream t2))
"TEXT append -> unfused" [1] forall t1 t2.
    unstream (S.append (stream t1) (stream t2)) = append t1 t2
 #-}

-- | /O(1)/ Returns the first character of a Text, which must be non-empty.
-- Subject to array fusion.
head :: Text -> Char
head t = S.head (stream t)
{-# INLINE head #-}

-- | /O(n)/ Returns the last character of a Text, which must be non-empty.
-- Subject to array fusion.
last :: Text -> Char
last (Text arr off len)
    | len <= 0                  = errorEmptyList "last"
    | n < 0xDC00 || n > 0xDFFF = unsafeChr n
    | otherwise                 = U16.chr2 n0 n
    where
      n  = unsafeAt arr (off+len-1)
      n0 = unsafeAt arr (off+len-2)
{-# INLINE [1] last #-}

{-# RULES
"TEXT last -> fused" [~1] forall t.
    last t = S.last (stream t)
"TEXT last -> unfused" [1] forall t.
    S.last (stream t) = last t
  #-}


-- | /O(1)/ Returns all characters after the head of a Text, which must
-- be non-empty.
-- Subject to array fusion.
tail :: Text -> Text
tail (Text arr off len)
    | len <= 0                  = errorEmptyList "tail"
    | n >= 0xD800 && n <= 0xDBFF = Text arr (off+2) (len-2)
    | otherwise                 = Text arr (off+1) (len-1)
    where
      n = unsafeAt arr off
{-# INLINE [1] tail #-}



-- | /O(1)/ Returns all but the last character of a Text, which
-- must be non-empty.
-- Subject to array fusion.
init :: Text -> Text
init (Text arr off len) | len <= 0                  = errorEmptyList "init"
                        | n >= 0xDC00 && n <= 0xDFFF = Text arr off (len-2)
                        | otherwise                 = Text arr off (len-1)
    where
```

```
      n = unsafeAt arr (off+len-1)
{-# INLINE [1] init #-}

{-# RULES
"TEXT init -> fused" [~1] forall t.
    init t = unstream (S.init (stream t))
"TEXT init -> unfused" [1] forall t.
    unstream (S.init (stream t)) = init t
 #-}

-- | /O(1)/ Tests whether a Text is empty or not.
-- Subject to array fusion.
null :: Text -> Bool
null t = S.null (stream t)
{-# INLINE null #-}

-- | /O(n)/ Returns the number of characters in a text.
-- Subject to array fusion.
length :: Text -> Int
length t = S.length (stream t)
{-# INLINE length #-}

-- -----------------------------------------------------------------------------
-- * Transformations
-- | /O(n)/ 'map' @f @xs is the Text obtained by applying @f@ to each
-- element of @xs@.
-- Subject to array fusion.
map :: (Char -> Char) -> Text -> Text
map f t = unstream (S.map f (stream t))
{-# INLINE [1] map #-}

-- | /O(n)/ The 'intersperse' function takes a character and places it between
-- the characters of a Text.
-- Subject to array fusion.
intersperse     :: Char -> Text -> Text
intersperse c t = unstream (S.intersperse c (stream t))
{-# INLINE intersperse #-}

-- | /O(n)/ The 'transpose' function transposes the rows and columns of its
-- Text argument.  Note that this function uses pack, unpack, and the 'List'
-- version of transpose and is thus not very efficient.
transpose :: [Text] -> [Text]
transpose ts = P.map pack (L.transpose (P.map unpack ts))

-- -----------------------------------------------------------------------------
-- * Reducing 'Text's (folds)

-- | 'foldl', applied to a binary operator, a starting value (typically the
-- left-identity of the operator), and a Text, reduces the Text using the
-- binary operator, from left to right.
-- Subject to array fusion.
foldl :: (b -> Char -> b) -> b -> Text -> b
foldl f z t = S.foldl f z (stream t)
{-# INLINE foldl #-}

-- | A strict version of 'foldl'.
-- Subject to array fusion.
foldl' :: (b -> Char -> b) -> b -> Text -> b
foldl' f z t = S.foldl' f z (stream t)
{-# INLINE foldl' #-}

-- | 'foldl1' is a variant of 'foldl' that has no starting value argument,
-- and thus must be applied to non-empty 'Text's.
-- Subject to array fusion.
foldl1 :: (Char -> Char -> Char) -> Text -> Char
foldl1 f t = S.foldl1 f (stream t)
{-# INLINE foldl1 #-}

-- | A strict version of 'foldl1'.
-- Subject to array fusion.
foldl1' :: (Char -> Char -> Char) -> Text -> Char
foldl1' f t = S.foldl1' f (stream t)
{-# INLINE foldl1' #-}

-- | 'foldr', applied to a binary operator, a starting value (typically the
-- right-identity of the operator), and a Text, reduces the Text using the
```

```
-- binary operator, from right to left.
-- Subject to array fusion.
foldr :: (Char -> b -> b) -> b -> Text -> b
foldr f z t = S.foldr f z (stream t)
{-# INLINE foldr #-}

-- | 'foldr1' is a variant of 'foldr' that has no starting value argument,
-- and thust must be applied to non-empty 'Text's.
-- Subject to array fusion.
foldr1 :: (Char -> Char -> Char) -> Text -> Char
foldr1 f t = S.foldr1 f (stream t)
{-# INLINE foldr1 #-}

-- ----------------------------------------------------------------------------
-- ** Special folds

-- | /O(n)/ Concatenate a list of 'Text's. Subject to array fusion.
concat :: [Text] -> Text
concat ts = unstream (S.concat (L.map stream ts))
{-# INLINE concat #-}

-- | Map a function over a Text that results in a Text and concatenate the
-- results.  This function is subject to array fusion, and note that if in
-- 'concatMap' @f @xs, @f@ is defined in terms of fusible functions it will
-- also be fusible.
concatMap :: (Char -> Text) -> Text -> Text
concatMap f t = unstream (S.concatMap (stream . f) (stream t))
{-# INLINE concatMap #-}

-- | 'any' @p @xs determines if any character in the 'Text' @xs@ satisifes the
-- predicate @p@. Subject to array fusion.
any :: (Char -> Bool) -> Text -> Bool
any p t = S.any p (stream t)
{-# INLINE any #-}

-- | 'all' @p @xs determines if all characters in the 'Text' @xs@ satisify the
-- predicate @p@. Subject to array fusion.
all :: (Char -> Bool) -> Text -> Bool
all p t = S.all p (stream t)
{-# INLINE all #-}

-- | /O(n)/ 'maximum' returns the maximum value from a 'Text', which must be
-- non-empty. Subject to array fusion.
maximum :: Text -> Char
maximum t = S.maximum (stream t)
{-# INLINE maximum #-}

-- | /O(n)/ 'minimum' returns the minimum value from a 'Text', which must be
-- non-empty. Subject to array fusion.
minimum :: Text -> Char
minimum t = S.minimum (stream t)
{-# INLINE minimum #-}

-- ----------------------------------------------------------------------------
-- * Building 'Text's

-- ----------------------------------------------------------------------------
-- ** Generating and unfolding 'Text's

-- /O(n)/, where @n@ is the length of the result. The unfoldr function
-- is analogous to the List 'unfoldr'. unfoldr builds a Text
-- from a seed value. The function takes the element and returns
-- Nothing if it is done producing the Text or returns Just
-- (a,b), in which case, a is the next Char in the string, and b is
-- the seed value for further production.
unfoldr     :: (a -> Maybe (Char,a)) -> a -> Text
unfoldr f s = unstream (S.unfoldr f s)
{-# INLINE unfoldr #-}

-- O(n) Like unfoldr, unfoldrN builds a Text from a seed
-- value. However, the length of the result should be limited by the
-- first argument to unfoldrN. This function is more efficient than
-- unfoldr when the maximum length of the result and correct,
-- otherwise its complexity performance is similar to 'unfoldr'
unfoldrN     :: Int -> (a -> Maybe (Char,a)) -> a -> Text
unfoldrN n f s = unstream (S.unfoldrN n f s)
```

```
{-# INLINE unfoldrN #-}

-- -----------------------------------------------------------------------------
-- * Substrings

-- O(n) 'take' @n, applied to a Text, returns the prefix of the
-- Text of length n, or the Text itself if n is greater than the
-- length of the Text.
take :: Int -> Text -> Text
take n (Text arr off len) = Text arr off (loop off 0)
    where
      end = off+len
      loop !i !count
           | i >= end || count >= n   = i - off
           | c < 0xD800 || c > 0xDBFF = loop (i+1) (count+1)
           | otherwise                = loop (i+2) (count+1)
           where
             c = arr `unsafeAt` i
{-# INLINE [1] take #-}

{-# RULES
"TEXT take -> fused" [~1] forall n t.
    take n t = unstream (S.take n (stream t))
"TEXT take -> unfused" [1] forall n t.
    unstream (S.take n (stream t)) = take n t
  #-}

-- /O(n)/ 'drop' @n, applied to a Text, returns the suffix of the
-- Text of length @n, or the empty Text if @n is greater than the
-- length of the Text.
drop :: Int -> Text -> Text
drop n (Text arr off len) = (Text arr newOff newLen)
    where
      (newOff, newLen) = loop off 0 len
      end = off + len
      loop !i !count !l
          | i >= end || count >= n   = (i,l)
          | c < 0xD800 || c > 0xDBFF = loop (i+1) (count+1) (l-1)
          | otherwise                = loop (i+2) (count+1) (l-2)
          where
            c = arr `unsafeAt` i
{-# INLINE [1] drop #-}

{-# RULES
"TEXT drop -> fused" [~1] forall n t.
    drop n t = unstream (S.drop n (stream t))
"TEXT drop -> unfused" [1] forall n t.
    unstream (S.drop n (stream t)) = drop n t
  #-}

-- | 'takeWhile', applied to a predicate @p@ and a stream, returns the
-- longest prefix (possibly empty) of elements that satisfy p.
takeWhile :: (Char -> Bool) -> Text -> Text
takeWhile p t = unstream (S.takeWhile p (stream t))

-- | 'dropWhile' @p @xs returns the suffix remaining after 'takeWhile' @p @xs.
dropWhile :: (Char -> Bool) -> Text -> Text
dropWhile p t = unstream (S.dropWhile p (stream t))

-- -----------------------------------------------------------------------------
-- * Searching

------------------------------------------------------------------------------
-- ** Searching by equality

-- | /O(n)/ 'elem' is the 'Text' membership predicate.
elem :: Char -> Text -> Bool
elem c t = S.elem c (stream t)
{-# INLINE elem #-}

------------------------------------------------------------------------------
-- ** Searching with a predicate

-- | /O(n)/ The 'find' function takes a predicate and a 'Text',
-- and returns the first element in matching the predicate, or 'Nothing'
-- if there is no such element.
```

```
find :: (Char -> Bool) -> Text -> Maybe Char
find p t = S.find p (stream t)
{-# INLINE find #-}

-- | /O(n)/ 'filter', applied to a predicate and a 'Text',
-- returns a 'Text' containing those characters that satisfy the
-- predicate.
filter :: (Char -> Bool) -> Text -> Text
filter p t = unstream (S.filter p (stream t))
{-# INLINE filter #-}


-------------------------------------------------------------------------------
-- ** Indexing 'Text's

-- | /O(1)/ 'Text' index (subscript) operator, starting from 0.
index :: Text -> Int -> Char
index t n = S.index (stream t) n
{-# INLINE index #-}

-- | The 'findIndex' function takes a predicate and a 'Text' and
-- returns the index of the first element in the 'Text'
-- satisfying the predicate.
findIndex :: (Char -> Bool) -> Text -> Maybe Int
findIndex p t = S.findIndex p (stream t)
{-# INLINE findIndex #-}

-- | /O(n)/ The 'elemIndex' function returns the index of the first
-- element in the given 'Text' which is equal to the query element, or
-- 'Nothing' if there is no such element.
elemIndex :: Char -> Text -> Maybe Int
elemIndex c t = S.elemIndex c (stream t)

-------------------------------------------------------------------------------
-- * Zipping

-- | /O(n)/ 'zipWith' generalises 'zip' by zipping with the function
-- given as the first argument, instead of a tupling function.
zipWith :: (Char -> Char -> Char) -> Text -> Text -> Text
zipWith f t1 t2 = unstream (S.zipWith f (stream t1) (stream t2))

-- File I/O

readFile :: Encoding -> FilePath -> IO Text
readFile enc f = B.readFile f >>= return . unstream . stream_bs enc
{-# INLINE [1] readFile #-}

words :: Text -> [Text]
words (Text arr off len) = loop0 off off
    where
      loop0 start n
            | isSpace (unsafeChr c) = if start == n
                                      then loop0 (start+1) (start+1)
                                      else (Text arr start (n-start)) :
                                               loop0 (n+1) (n+1)
            | n < (off+len) = loop0 start (n+1)
            | otherwise = if start == n
                          then []
                          else [(Text arr start (n-start))]
            where
              c = arr `unsafeAt` n
{-# INLINE words #-}
```

A.1.2 *Text/Fusion.hs*

```
{-# OPTIONS_GHC -fglasgow-exts -fbang-patterns #-}

module Text.Fusion where

import Prelude hiding (map, tail, head, foldr, filter,concat)
import qualified Prelude as P
import Char
import Data.Bits
import Data.Word
```

```
import Control.Monad.ST
import Control.Monad(liftM2)
import Data.Array.Base
import System.IO.Unsafe(unsafePerformIO)

import GHC.Prim
import GHC.Exts

import qualified Text.Utf8 as U8
import qualified Text.Utf16 as U16
import qualified Text.Utf32 as U32
import Text.Internal(Text(..),empty)

import qualified Data.ByteString as B
import Data.ByteString.Internal(ByteString(..))
import Text.UnsafeChar

import Data.ByteString.Internal(mallocByteString,memcpy)
import Foreign.Storable(pokeByteOff)
import Foreign.ForeignPtr(withForeignPtr,ForeignPtr(..))
import Control.Exception(assert)

default(Int)

infixl 2 :!:
data PairS a b = !a :!: !b

data Switch = S1 | S2

data EitherS a b = LeftS !a | RightS !b

data Stream a = forall s. Stream (s -> Step s a) !s {-# UNPACK #-}!Int

data Step s a = Done
              | Skip !s
              | Yield !a !s

data Encoding = ASCII | Utf8 | Utf16BE | Utf16LE | Utf32BE | Utf32LE

-- | /O(n)/ Convert a Text into a Stream Char.
stream :: Text -> Stream Char
stream (Text arr off len) = Stream next off len
    where
      end = off+len
      {-# INLINE next #-}
      next !i
          | i >= end = 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 #-}

-- | /O(n)/ Convert a Stream Char into a Text.
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))
                 >>= (\arr -> 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 :: Int
                      n = ord x
                      m :: Int
                      m = n - 0x10000
                      l :: Word16
                      l = fromIntegral $ (shiftR m 10) + (0xD800 :: Int)
                      r :: Word16
                      r = fromIntegral $ (m .&. (0x3FF :: Int)) + (0xDC00 :: Int)
{-# INLINE [0] unstream #-}


copy src dest = ({-# SCC "TEXT copy" #-} do
                  (_,max) <- getBounds src
                  copy_loop 0 max)
    where
      copy_loop !i !max
          | i > max    = return ()
          | otherwise = do v <- unsafeRead src i
                           unsafeWrite dest i v
                           copy_loop (i+1) max

-- | /O(n)/ Determines if two streams are equal.
eq :: Ord a => Stream a -> Stream a -> Bool
eq (Stream next1 s1 _) (Stream next2 s2 _) = compare (next1 s1) (next2 s2)
    where
      compare Done Done = True
      compare Done _    = False
      compare _    Done = False
      compare (Skip s1')     (Skip s2')     = compare (next1 s1') (next2 s2')
      compare (Skip s1')     x2             = compare (next1 s1') x2
      compare x1             (Skip s2')     = compare x1          (next2 s2')
      compare (Yield x1 s1') (Yield x2 s2') = x1 == x2 &&
                                              compare (next1 s1') (next2 s2')
{-# SPECIALISE eq :: Stream Char -> Stream Char -> Bool #-}


-- | /O(n) Convert a ByteString into a Stream Char, using the specified encoding standard.
stream_bs :: Encoding -> ByteString -> Stream Char
stream_bs ASCII bs = Stream next 0 (B.length bs)
    where
      {-# INLINE next #-}
      next i
          | i >= l    = Done
          | otherwise = Yield (unsafeChr8 x1) (i+1)
          where
            l  = B.length bs
            x1 = B.index bs i
stream_bs Utf8 bs = Stream next 0 (B.length bs)
    where
      {-# INLINE next #-}
      next i
          | i >= l = Done
          | U8.validate1 x1 = Yield (unsafeChr8 x1) (i+1)
          | i+1 < l && U8.validate2 x1 x2 = Yield (U8.chr2 x1 x2) (i+2)
          | i+2 < l && U8.validate3 x1 x2 x3 = Yield (U8.chr3 x1 x2 x3) (i+3)
          | i+3 < l && U8.validate4 x1 x2 x3 x4 = Yield (U8.chr4 x1 x2 x3 x4) (i+4)
          | otherwise = error "bsStream: bad UTF-8 stream"
          where
            l  = B.length bs
            x1 = index i
            x2 = index (i + 1)
            x3 = index (i + 2)
            x4 = index (i + 3)
            index = B.index bs
stream_bs Utf16LE bs = Stream next 0 (B.length bs)
    where
      {-# INLINE next #-}
      next i
          | i >= l                        = Done
          | i+1 < l && U16.validate1 x1    = Yield (unsafeChr x1) (i+2)
          | i+3 < l && U16.validate2 x1 x2 = Yield (U16.chr2 x1 x2) (i+4)
          | otherwise = error $ "bsStream: bad UTF-16LE stream"
          where
            x1    :: Word16
```

```
            x1    = (shiftL (index (i + 1)) 8) + (index i)
            x2    :: Word16
            x2    = (shiftL (index (i + 3)) 8) + (index (i + 2))
            l     = B.length bs
            index = fromIntegral . B.index bs :: Int -> Word16
stream_bs Utf16BE bs = Stream next 0 (B.length bs)
    where
      {-# INLINE next #-}
      next i
          | i >= l                        = Done
          | i+1 < l && U16.validate1 x1    = Yield (unsafeChr x1) (i+2)
          | i+3 < l && U16.validate2 x1 x2 = Yield (U16.chr2 x1 x2) (i+4)
          | otherwise = error $ "bsStream: bad UTF16-BE stream "
          where
            x1    :: Word16
            x1    = (shiftL (index i) 8) + (index (i + 1))
            x2    :: Word16
            x2    = (shiftL (index (i + 2)) 8) + (index (i + 3))
            l     = B.length bs
            index = fromIntegral . B.index bs
stream_bs Utf32BE bs = Stream next 0 (B.length bs)
    where
      {-# INLINE next #-}
      next i
          | i >= l                    = Done
          | i+3 < l && U32.validate x = Yield (unsafeChr32 x) (i+4)
          | otherwise                 = error "bsStream: bad UTF-32BE stream"
          where
            l     = B.length bs
            x     = shiftL x1 24 + shiftL x2 16 + shiftL x3 8 + x4
            x1    = index i
            x2    = index (i+1)
            x3    = index (i+2)
            x4    = index (i+3)
            index = fromIntegral . B.index bs :: Int -> Word32
stream_bs Utf32LE bs = Stream next 0 (B.length bs)
    where
      {-# INLINE next #-}
      next i
          | i >= l                    = Done
          | i+3 < l && U32.validate x = Yield (unsafeChr32 x) (i+4)
          | otherwise                 = error "bsStream: bad UTF-32LE stream"
          where
            l     = B.length bs
            x     = shiftL x4 24 + shiftL x3 16 + shiftL x2 8 + x1
            x1    = index i
            x2    = index $ i+1
            x3    = index $ i+2
            x4    = index $ i+3
            index = fromIntegral . B.index bs :: Int -> Word32
{-# INLINE [0] stream_bs #-}

-- | /O(n)/ Convert a Stream Char into a Stream Word8 using the specified encoding standard.
restream :: Encoding -> Stream Char -> Stream Word8
restream ASCII (Stream next0 s0 len) =  Stream next s0 (len*2)
    where
      next !s = case next0 s of
                  Done -> Done
                  Skip s' -> Skip s'
                  Yield x xs -> Yield x' xs
                      where x' = fromIntegral (ord x) :: Word8
restream Utf8 (Stream next0 s0 len) =
    Stream next ((Just s0) :!: Nothing :!: Nothing :!: Nothing) (len*2)
    where
      {-# INLINE next #-}
      next ((Just s) :!: Nothing :!: Nothing :!: Nothing) = case next0 s of
          Done              -> Done
          Skip s'           ->
              Skip ((Just s') :!: Nothing :!: Nothing :!: Nothing)
          Yield x xs
              | n <= 0x7F   ->
                  Yield c ((Just xs) :!: Nothing :!: Nothing :!: Nothing)
              | n <= 0x07FF ->
                   Yield (fst c2)  ((Just xs) :!: (Just $ snd c2) :!: Nothing   :!: Nothing)
              | n <= 0xFFFF ->
                   Yield (fst3 c3) ((Just xs) :!: (Just $ snd3 c3) :!: (Just $ trd3 c3) :!: Nothing)
```

```
              | otherwise   ->
                  Yield (fst4 c4) ((Just xs) :!: (Just $ snd4 c4) :!: (Just $ trd4 c4) :!: (Just $ fth4 c4))
              where
                n  = ord x
                c  = fromIntegral n
                c2 = U8.ord2 x
                c3 = U8.ord3 x
                c4 = U8.ord4 x
      next ((Just s) :!: (Just x2) :!: Nothing :!: Nothing) = Yield x2 ((Just s) :!: Nothing :!: Nothing :!: Nothing)
      next ((Just s) :!: (Just x2) :!: x3 :!: Nothing)      = Yield x2 ((Just s) :!: x3 :!: Nothing :!: Nothing)
      next ((Just s) :!: (Just x2) :!: x3 :!: x4)           = Yield x2 ((Just s) :!: x3 :!: x4 :!: Nothing)
restream Utf16BE (Stream next0 s0 len) =
    Stream next (Just s0 :!: Nothing :!: Nothing :!: Nothing) (len*2)
    where
      {-# INLINE next #-}
      next (Just s :!: Nothing :!: Nothing :!: Nothing) = case next0 s of
          Done -> Done
          Skip s' -> Skip (Just s' :!: Nothing :!: Nothing :!: Nothing)
          Yield x xs
              | n < 0x10000 -> Yield (fromIntegral $ shiftR n 8) (Just xs :!: Just (fromIntegral n) :!: Nothing :!: Noth!
!ing)
              | otherwise   -> Yield c1                          (Just xs :!: Just c2 :!: Just c3 :!: Just c4)
              where
                n  = ord x
                n1 = n - 0x10000
                c1 = fromIntegral (shiftR n1 18 + 0xD8)
                c2 = fromIntegral (shiftR n1 10)
                n2 = n1 .&. 0x3FF
                c3 = fromIntegral (shiftR n2 8 + 0xDC)
                c4 = fromIntegral n2
      next ((Just s) :!: (Just x2) :!: Nothing :!: Nothing) = Yield x2 ((Just s) :!: Nothing :!: Nothing :!: Nothing)
      next ((Just s) :!: (Just x2) :!: x3 :!: Nothing)      = Yield x2 ((Just s) :!: x3 :!: Nothing :!: Nothing)
      next ((Just s) :!: (Just x2) :!: x3 :!: x4)           = Yield x2 ((Just s) :!: x3 :!: x4 :!: Nothing)
restream Utf16LE (Stream next0 s0 len) =
    Stream next (Just s0 :!: Nothing :!: Nothing :!: Nothing) (len*2)
    where
      {-# INLINE next #-}
      next (Just s :!: Nothing :!: Nothing :!: Nothing) = case next0 s of
          Done -> Done
          Skip s' -> Skip (Just s' :!: Nothing :!: Nothing :!: Nothing)
          Yield x xs
              | n < 0x10000 -> Yield (fromIntegral n) (Just xs :!: Just (fromIntegral $ shiftR n 8) :!: Nothing :!: Noth!
!ing)
              | otherwise   -> Yield c1                          (Just xs :!: Just c2 :!: Just c3 :!: Just c4)
              where
                n  = ord x
                n1 = n - 0x10000
                c2 = fromIntegral (shiftR n1 18 + 0xD8)
                c1 = fromIntegral (shiftR n1 10)
                n2 = n1 .&. 0x3FF
                c4 = fromIntegral (shiftR n2 8 + 0xDC)
                c3 = fromIntegral n2
      next ((Just s) :!: (Just x2) :!: Nothing :!: Nothing) = Yield x2 ((Just s) :!: Nothing :!: Nothing :!: Nothing)
      next ((Just s) :!: (Just x2) :!: x3 :!: Nothing)      = Yield x2 ((Just s) :!: x3 :!: Nothing :!: Nothing)
      next ((Just s) :!: (Just x2) :!: x3 :!: x4)           = Yield x2 ((Just s) :!: x3 :!: x4 :!: Nothing)
restream Utf32BE (Stream next0 s0 len) =
    Stream next (Just s0 :!: Nothing :!: Nothing :!: Nothing) (len*2)
    where
    {-# INLINE next #-}
    next (Just s :!: Nothing :!: Nothing :!: Nothing) = case next0 s of
        Done       -> Done
        Skip s'    -> Skip (Just s' :!: Nothing :!: Nothing :!: Nothing)
        Yield x xs -> Yield c1 (Just xs :!: Just c2 :!: Just c3 :!: Just c4)
          where
            n  = ord x
            c1 = fromIntegral $ shiftR n 24
            c2 = fromIntegral $ shiftR n 16
            c3 = fromIntegral $ shiftR n 8
            c4 = fromIntegral n
    next ((Just s) :!: (Just x2) :!: Nothing :!: Nothing) = Yield x2 ((Just s) :!: Nothing :!: Nothing :!: Nothing)
    next ((Just s) :!: (Just x2) :!: x3 :!: Nothing)      = Yield x2 ((Just s) :!: x3 :!: Nothing :!: Nothing)
    next ((Just s) :!: (Just x2) :!: x3 :!: x4)           = Yield x2 ((Just s) :!: x3 :!: x4 :!: Nothing)
restream Utf32LE (Stream next0 s0 len) =
    Stream next (Just s0 :!: Nothing :!: Nothing :!: Nothing) (len*2)
    where
    {-# INLINE next #-}
```

```
    next (Just s :!: Nothing :!: Nothing :!: Nothing) = case next0 s of
        Done       -> Done
        Skip s'    -> Skip (Just s' :!: Nothing :!: Nothing :!: Nothing)
        Yield x xs -> Yield c1 (Just xs :!: Just c2 :!: Just c3 :!: Just c4)
          where
            n  = ord x
            c4 = fromIntegral $ shiftR n 24
            c3 = fromIntegral $ shiftR n 16
            c2 = fromIntegral $ shiftR n 8
            c1 = fromIntegral n
    next ((Just s) :!: (Just x2) :!: Nothing :!: Nothing) = Yield x2 ((Just s) :!: Nothing :!: Nothing :!: Nothing)
    next ((Just s) :!: (Just x2) :!: x3 :!: Nothing)      = Yield x2 ((Just s) :!: x3 :!: Nothing :!: Nothing)
    next ((Just s) :!: (Just x2) :!: x3 :!: x4)           = Yield x2 ((Just s) :!: x3 :!: x4 :!: Nothing)
{-# INLINE restream #-}


fst3 (x1,_,_)   = x1
snd3 (_,x2,_)   = x2
trd3 (_,_,x3)   = x3
fst4 (x1,_,_,_) = x1
snd4 (_,x2,_,_) = x2
trd4 (_,_,x3,_) = x3
fth4 (_,_,_,x4) = x4

-- | /O(n)/ Convert a Stream Word8 to a ByteString
unstream_bs :: Stream Word8 -> ByteString
unstream_bs (Stream next s0 len) = unsafePerformIO $ do
    fp0 <- mallocByteString len
    loop fp0 len 0 s0
    where
      loop !fp !n !off !s = case next s of
          Done -> trimUp fp n off
          Skip s' -> loop fp n off s'
          Yield x s'
              | n == off -> realloc fp n off s' x
              | otherwise -> do
            withForeignPtr fp $ \p -> pokeByteOff p off x
            loop fp n (off+1) s'
      {-# NOINLINE realloc #-}
      realloc fp n off s x = do
        let n' = n+n
        fp' <- copy0 fp n n'
        withForeignPtr fp' $ \p -> pokeByteOff p off x
        loop fp' n' (off+1) s
      {-# NOINLINE trimUp #-}
      trimUp fp _ off = return $! PS fp 0 off
      copy0 :: ForeignPtr Word8 -> Int -> Int -> IO (ForeignPtr Word8)
      copy0 !src !srcLen !destLen = assert (srcLen <= destLen) $ do
          dest <- mallocByteString destLen
          withForeignPtr src  $ \src'  ->
              withForeignPtr dest $ \dest' ->
                  memcpy dest' src' (fromIntegral destLen)
          return dest
{-# RULES "STREAM stream/unstream fusion" forall s. stream (unstream s) = s #-}

-- ----------------------------------------------------------------------------
-- * Basic stream functions

-- | /O(n)/ Adds a character to the front of a Stream Char.
cons :: Char -> Stream Char -> Stream Char
cons w (Stream next0 s0 len) = Stream next (S2 :!: s0) (len+2)
    where
      {-# INLINE next #-}
      next (S2 :!: s) = Yield w (S1 :!: s)
      next (S1 :!: s) = case next0 s of
                          Done -> Done
                          Skip s' -> Skip (S1 :!: s')
                          Yield x s' -> Yield x (S1 :!: s')
{-# INLINE [0] cons #-}

-- | /O(n)/ Adds a character to the end of a stream.
snoc :: Stream Char -> Char -> Stream Char
snoc (Stream next0 xs0 len) w = Stream next (Just xs0) (len+2)
  where
    {-# INLINE next #-}
    next (Just xs) = case next0 xs of
```

```
      Done        -> Yield w Nothing
      Skip xs'    -> Skip    (Just xs')
      Yield x xs' -> Yield x (Just xs')
    next Nothing = Done
{-# INLINE [0] snoc #-}

-- | /O(n)/ Appends one Stream to the other.
append :: Stream Char -> Stream Char -> Stream Char
append (Stream next0 s01 len1) (Stream next1 s02 len2) =
    Stream next (Left s01) (len1 + len2)
    where
      {-# INLINE next #-}
      next (Left s1) = case next0 s1 of
                         Done        -> Skip    (Right s02)
                         Skip s1'    -> Skip    (Left s1')
                         Yield x s1' -> Yield x (Left s1')
      next (Right s2) = case next1 s2 of
                          Done        -> Done
                          Skip s2'    -> Skip    (Right s2')
                          Yield x s2' -> Yield x (Right s2')
{-# INLINE [0] append #-}

-- | /O(1)/ Returns the first character of a Text, which must be non-empty.
-- Subject to array fusion.
head :: Stream Char -> Char
head (Stream next s0 len) = loop_head s0
    where
      loop_head !s = case next s of
                       Yield x _ -> x
                       Skip s' -> loop_head s'
                       Done -> error "head: Empty list"
{-# INLINE [0] head #-}

-- | /O(n)/ Returns the last character of a Stream Char, which must be non-empty.
last :: Stream Char -> Char
last (Stream next s0 len) = loop0_last s0
    where
      loop0_last !s = case next s of
                        Done       -> error "last: Empty list"
                        Skip s'    -> seq s' $ loop0_last  s'
                        Yield x s' -> seq s' $ loop_last x s'
      loop_last !x !s = case next s of
                          Done        -> x
                          Skip s'     -> seq s' $ loop_last x  s'
                          Yield x' s' -> seq s' $ loop_last x' s'
{-# INLINE[0] last #-}

-- | /O(1)/ Returns all characters after the head of a Stream Char, which must
-- be non-empty.
tail :: Stream Char -> Stream Char
tail (Stream next0 s0 len) = Stream next (False :!: s0) (len-1)
    where
      {-# INLINE next #-}
      next (False :!: s) = case next0 s of
                           Done -> error "tail"
                           Skip s' -> Skip (False :!: s')
                           Yield _ s' -> Skip (True :!: s')
      next (True :!: s) = case next0 s of
                           Done -> Done
                           Skip s' -> Skip (True :!: s')
                           Yield x s' -> Yield x (True :!: s')
{-# INLINE [0] tail #-}


-- | /O(1)/ Returns all but the last character of a Stream Char, which
-- must be non-empty.
init :: Stream Char -> Stream Char
init (Stream next0 s0 len) = Stream next (Nothing :!: s0) (len-1)
    where
      {-# INLINE next #-}
      next (Nothing :!: s) = case next0 s of
                               Done       -> errorEmptyList "init"
                               Skip s'    -> seq s' $ Skip (Nothing :!: s')
                               Yield x s' -> seq s' $ Skip (Just x  :!: s')
      next (Just x :!: s)  = case next0 s of
                               Done        -> Done
```

```
                              Skip s'     -> seq s' $ Skip    (Just x  :!: s')
                              Yield x' s' -> seq s' $ Yield x (Just x' :!: s')
{-# INLINE [0] init #-}

-- | /O(1)/ Tests whether a Stream Char is empty or not.
null :: Stream Char -> Bool
null (Stream next s0 len) = loop_null s0
    where
      loop_null !s = case next s of
                       Done      -> True
                       Yield _ _ -> False
                       Skip s'   -> loop_null s'
{-# INLINE[0] null #-}

-- | /O(n)/ Returns the number of characters in a text.
length :: Stream Char -> Int
length (Stream next s0 len) = loop_length 0# s0
    where

      loop_length z# !s  = case next s of
                             Done       -> (I# z#)
                             Skip    s' -> loop_length z# s'
                             Yield _ s' -> loop_length (z# +# 1#) s'
{-# INLINE[0] length #-}

-- ----------------------------------------------------------------------------
-- * Stream transformations

-- | /O(n)/ 'map' @f @xs is the Stream Char obtained by applying @f@ to each element of
-- @xs@.
map :: (Char -> Char) -> Stream Char -> Stream Char
map f (Stream next0 s0 len) = Stream next s0 len
    where
      {-# INLINE next #-}
      next !s = case next0 s of
                  Done       -> Done
                  Skip s'    -> Skip s'
                  Yield x s' -> Yield (f x) s'
{-# INLINE [0] map #-}

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

-- | /O(n)/ The 'intersperse' function takes a character and places it between each of
-- the characters of a Stream.
intersperse :: Char -> Stream Char -> Stream Char
intersperse c (Stream next0 s0 len) = Stream next (s0 :!: Nothing :!: S1) len
    where
      {-# INLINE next #-}
      next (s :!: Nothing :!: S1) = case next0 s of
        Done       -> Done
        Skip s'    -> Skip (s' :!: Nothing :!: S1)
        Yield x s' -> Skip (s' :!: Just x :!: S1)
      next (s :!: Just x :!: S1)  = Yield x (s :!: Nothing :!: S2)
      next (s :!: Nothing :!: S2) = case next0 s of
        Done       -> Done
        Skip s'    -> Skip    (s' :!: Nothing :!: S2)
        Yield x s' -> Yield c (s' :!: Just x :!: S1)

-- ----------------------------------------------------------------------------
-- * Reducing Streams (folds)

-- | foldl, applied to a binary operator, a starting value (typically the
-- left-identity of the operator), and a Stream, reduces the Stream using the
-- binary operator, from left to right.
foldl :: (b -> Char -> b) -> b -> Stream Char -> b
foldl f z0 (Stream next s0 len) = loop_foldl z0 s0
    where
      loop_foldl z !s = case next s of
                          Done -> z
                          Skip s' -> loop_foldl z s'
                          Yield x s' -> loop_foldl (f z x) s'
{-# INLINE [0] foldl #-}
```

```
-- | A strict version of foldl.
foldl' :: (b -> Char -> b) -> b -> Stream Char -> b
foldl' f z0 (Stream next s0 len) = loop_foldl' z0 s0
    where
      loop_foldl' !z !s = case next s of
                            Done -> z
                            Skip s' -> loop_foldl' z s'
                            Yield x s' -> loop_foldl' (f z x) s'
{-# INLINE [0] foldl' #-}

-- | foldl1 is a variant of foldl that has no starting value argument,
-- and thus must be applied to non-empty Streams.
foldl1 :: (Char -> Char -> Char) -> Stream Char -> Char
foldl1 f (Stream next s0 len) = loop0_foldl1 s0
    where
      loop0_foldl1 !s = case next s of
                          Skip s' -> loop0_foldl1 s'
                          Yield x s' -> loop_foldl1 x s'
                          Done -> errorEmptyList "foldl1"
      loop_foldl1 z !s = case next s of
                           Done -> z
                           Skip s' -> loop_foldl1 z s'
                           Yield x s' -> loop_foldl1 (f z x) s'
{-# INLINE [0] foldl1 #-}

-- | A strict version of foldl1.
foldl1' :: (Char -> Char -> Char) -> Stream Char -> Char
foldl1' f (Stream next s0 len) = loop0_foldl1' s0
    where
      loop0_foldl1' !s = case next s of
                           Skip s' -> loop0_foldl1' s'
                           Yield x s' -> loop_foldl1' x s'
                           Done -> errorEmptyList "foldl1"
      loop_foldl1' !z !s = case next s of
                             Done -> z
                             Skip s' -> loop_foldl1' z s'
                             Yield x s' -> loop_foldl1' (f z x) s'
{-# INLINE [0] foldl1' #-}

-- | 'foldr', applied to a binary operator, a starting value (typically the
-- right-identity of the operator), and a stream, reduces the stream using the
-- binary operator, from right to left.
foldr :: (Char -> b -> b) -> b -> Stream Char -> b
foldr f z (Stream next s0 len) = loop_foldr s0
    where
      loop_foldr !s = case next s of
                        Done -> z
                        Skip s' -> loop_foldr s'
                        Yield x s' -> f x (loop_foldr s')
{-# INLINE [0] foldr #-}

-- | foldr1 is a variant of 'foldr' that has no starting value argument,
-- and thust must be applied to non-empty streams.
-- Subject to array fusion.
foldr1 :: (Char -> Char -> Char) -> Stream Char -> Char
foldr1 f (Stream next s0 len) = loop0_foldr1 s0
  where
    loop0_foldr1 !s = case next s of
      Done       -> error "foldr1"
      Skip    s' -> loop0_foldr1  s'
      Yield x s' -> loop_foldr1 x s'

    loop_foldr1 x !s = case next s of
      Done        -> x
      Skip     s' -> loop_foldr1 x s'
      Yield x' s' -> f x (loop_foldr1 x' s')
{-# INLINE [0] foldr1 #-}

-- ---------------------------------------------------------------------------
-- ** Special folds

-- | /O(n)/ Concatenate a list of streams. Subject to array fusion.
concat :: [Stream Char] -> Stream Char
concat = P.foldr append (Stream next Done 0)
    where
      next Done = Done
```

```
-- | Map a function over a stream that results in a steram and concatenate the
-- results.
concatMap :: (Char -> Stream Char) -> Stream Char -> Stream Char
concatMap f = foldr (append . f) (stream empty)

-- | /O(n)/ any @p @xs determines if any character in the stream
-- @xs@ satisifes the predicate @p@.
any :: (Char -> Bool) -> Stream Char -> Bool
any p (Stream next0 s0 len) = loop_any s0
    where
      loop_any !s = case next0 s of
                      Done                  -> False
                      Skip s'               -> seq s' $ loop_any s'
                      Yield x s' | p x      -> True
                                 | otherwise -> seq s' $ loop_any s'

-- | /O(n)/ all @p @xs determines if all characters in the 'Text'
-- @xs@ satisify the predicate @p@.
all :: (Char -> Bool) -> Stream Char -> Bool
all p (Stream next0 s0 len) = loop_all s0
    where
      loop_all !s = case next0 s of
                      Done                  -> True
                      Skip s'               -> seq s' $ loop_all s'
                      Yield x s' | p x      -> seq s' $ loop_all s'
                                 | otherwise -> False

-- | /O(n)/ maximum returns the maximum value from a stream, which must be
-- non-empty.
maximum :: Stream Char -> Char
maximum (Stream next0 s0 len) = loop0_maximum s0
    where
      loop0_maximum !s   = case next0 s of
                             Done       -> errorEmptyList "maximum"
                             Skip s'    -> seq s' $ loop0_maximum s'
                             Yield x s' -> seq s' $ loop_maximum x s'
      loop_maximum !z !s = case next0 s of
                             Done            -> z
                             Skip s'         -> seq s' $ loop_maximum z s'
                             Yield x s'
                                 | x > z     -> seq s' $ loop_maximum x s'
                                 | otherwise -> seq s' $ loop_maximum z s'

-- | /O(n)/ minimum returns the minimum value from a 'Text', which must be
-- non-empty.
minimum :: Stream Char -> Char
minimum (Stream next0 s0 len) = loop0_minimum s0
    where
      loop0_minimum !s   = case next0 s of
                             Done       -> errorEmptyList "minimum"
                             Skip s'    -> seq s' $ loop0_minimum s'
                             Yield x s' -> seq s' $ loop_minimum x s'
      loop_minimum !z !s = case next0 s of
                             Done            -> z
                             Skip s'         -> seq s' $ loop_minimum z s'
                             Yield x s'
                                 | x < z     -> seq s' $ loop_minimum x s'
                                 | otherwise -> seq s' $ loop_minimum z s'



-- ---------------------------------------------------------------------------
-- * Building streams

-- ---------------------------------------------------------------------------
-- ** Generating and unfolding streams

-- | /O(n)/, where @n@ is the length of the result. The unfoldr function
-- is analogous to the List 'unfoldr'. unfoldr builds a stream
-- from a seed value. The function takes the element and returns
-- Nothing if it is done producing the stream or returns Just
-- (a,b), in which case, a is the next Char in the string, and b is
-- the seed value for further production.
unfoldr :: (a -> Maybe (Char,a)) -> a -> Stream Char
```

```
unfoldr f s0 = Stream next s0 1
    where
      {-# INLINE next #-}
      next !s = case f s of
                  Nothing      -> Done
                  Just (w, s') -> Yield w s'
{-# INLINE [0] unfoldr #-}

-- | O(n) Like unfoldr, unfoldrN builds a stream from a seed
-- value. However, the length of the result should be limited by the
-- first argument to unfoldrN. This function is more efficient than
-- unfoldr when the maximum length of the result and correct,
-- otherwise its complexity performance is similar to 'unfoldr'
unfoldrN :: Int -> (a -> Maybe (Char,a)) -> a -> Stream Char
unfoldrN n f s0 = Stream next (0 :!: s0) (n*2)
    where
      {-# INLINE next #-}
      next (z :!: s) = case f s of
          Nothing                  -> Done
          Just (w, s') | z >= n    -> Done
                       | otherwise -> Yield w ((z + 1) :!: s')
-------------------------------------------------------------------------------
--  * Substreams

-- | /O(n)/ take n, applied to a stream, returns the prefix of the
-- stream of length @n@, or the stream itself if @n@ is greater than the
-- length of the stream.
take :: Int -> Stream Char -> Stream Char
take n0 (Stream next0 s0 len) = Stream next (n0 :!: s0) len
    where
      {-# INLINE next #-}
      next (n :!: s) | n <= 0    = Done
                     | otherwise = case next0 s of
                                     Done -> Done
                                     Skip s' -> Skip (n :!: s')
                                     Yield x s' -> Yield x ((n-1) :!: s')
{-# INLINE [0] take #-}

-- | /O(n)/ drop n, applied to a stream, returns the suffix of the
-- stream of length @n@, or the empty stream if @n@ is greater than the
-- length of the stream.
drop :: Int -> Stream Char -> Stream Char
drop n0 (Stream next0 s0 len) = Stream next (Just ((max 0 n0)) :!: s0) (len - n0)
  where
    {-# INLINE next #-}
    next (Just !n :!: s)
      | n == 0    = Skip (Nothing :!: s)
      | otherwise = case next0 s of
          Done       -> Done
          Skip    s' -> Skip (Just n    :!: s')
          Yield _ s' -> Skip (Just (n-1) :!: s')
    next (Nothing :!: s) = case next0 s of
      Done       -> Done
      Skip    s' -> Skip    (Nothing :!: s')
      Yield x s' -> Yield x (Nothing :!: s')
{-# INLINE [0] drop #-}

-- | takeWhile, applied to a predicate @p@ and a stream, returns the
-- longest prefix (possibly empty) of elements that satisfy p.
takeWhile :: (Char -> Bool) -> Stream Char -> Stream Char
takeWhile p (Stream next0 s0 len) = Stream next s0 len
    where
      {-# INLINE next #-}
      next !s = case next0 s of
                  Done    -> Done
                  Skip s' -> Skip s'
                  Yield x s' | p x       -> Yield x s'
                             | otherwise -> Done
{-# INLINE [0] takeWhile #-}

-- | dropWhile @p @xs returns the suffix remaining after takeWhile @p @xs.
dropWhile :: (Char -> Bool) -> Stream Char -> Stream Char
dropWhile p (Stream next0 s0 len) = Stream next (S1 :!: s0) len
    where
    {-# INLINE next #-}
    next (S1 :!: s)  = case next0 s of
```

```
      Done                  -> Done
      Skip    s'            -> Skip    (S1 :!: s')
      Yield x s' | p x      -> Skip    (S1 :!: s')
                 | otherwise -> Yield x (S2 :!: s')
    next (S2 :!: s) = case next0 s of
      Done       -> Done
      Skip    s' -> Skip    (S2 :!: s')
      Yield x s' -> Yield x (S2 :!: s')
{-# INLINE [0] dropWhile #-}

-- ---------------------------------------------------------------------------
-- * Searching

-- -----------------------------------------------------------------------------
-- ** Searching by equality

-- | /O(n)/ elem is the stream membership predicate.
elem :: Char -> Stream Char -> Bool
elem w (Stream next s0 len) = loop_elem s0
    where
      loop_elem !s = case next s of
                       Done -> False
                       Skip s' -> loop_elem s'
                       Yield x s' | x == w -> True
                                  | otherwise -> loop_elem s'
{-# INLINE [0] elem #-}

-- -----------------------------------------------------------------------------
-- ** Searching with a predicate

-- | /O(n)/ The 'find' function takes a predicate and a stream,
-- and returns the first element in matching the predicate, or 'Nothing'
-- if there is no such element.

find :: (Char -> Bool) -> Stream Char -> Maybe Char
find p (Stream next s0 len) = loop_find s0
    where
      loop_find !s = case next s of
                       Done -> Nothing
                       Skip s' -> loop_find s'
                       Yield x s' | p x -> Just x
                                  | otherwise -> loop_find s'
{-# INLINE [0] find #-}

-- | /O(n)/ 'filter', applied to a predicate and a stream,
-- returns a stream containing those characters that satisfy the
-- predicate.
filter :: (Char -> Bool) -> Stream Char -> Stream Char
filter p (Stream next0 s0 len) = Stream next s0 len
  where
    {-# INLINE next #-}
    next !s = case next0 s of
                Done                  -> Done
                Skip    s'            -> Skip    s'
                Yield x s' | p x      -> Yield x s'
                           | otherwise -> Skip    s'
{-# INLINE [0] filter #-}

{-# RULES
  "Stream filter/filter fusion" forall p q s.
  filter p (filter q s) = filter (\x -> q x && p x) s
  #-}

-- -----------------------------------------------------------------------------
-- ** Indexing streams

-- | /O(1)/ stream index (subscript) operator, starting from 0.
index :: Stream Char -> Int -> Char
index (Stream next s0 len) n0
  | n0 < 0    = error "Stream.(!!): negative index"
  | otherwise = loop_index n0 s0
  where
    loop_index !n !s = case next s of
      Done                  -> error "Stream.(!!): index too large"
      Skip    s'            -> loop_index  n    s'
      Yield x s' | n == 0    -> x
```

```
                | otherwise -> loop_index (n-1) s'
{-# INLINE [0] index #-}

-- | The 'findIndex' function takes a predicate and a stream and
-- returns the index of the first element in the stream
-- satisfying the predicate.
findIndex :: (Char -> Bool) -> Stream Char -> Maybe Int
findIndex p (Stream next s0 len) = loop_findIndex 0 s0
  where
    loop_findIndex !i !s = case next s of
      Done                -> Nothing
      Skip    s'          -> loop_findIndex i     s' -- hmm. not caught by QC
      Yield x s' | p x    -> Just i
                 | otherwise -> loop_findIndex (i+1) s'
{-# INLINE [0] findIndex #-}

-- | /O(n)/ The 'elemIndex' function returns the index of the first
-- element in the given stream which is equal to the query
-- element, or 'Nothing' if there is no such element.
elemIndex :: Char -> Stream Char -> Maybe Int
elemIndex a (Stream next s0 len) = loop_elemIndex 0 s0
  where
    loop_elemIndex !i !s = case next s of
      Done                -> Nothing
      Skip    s'          -> loop_elemIndex i     s'
      Yield x s' | a == x    -> Just i
                 | otherwise -> loop_elemIndex (i+1) s'
{-# INLINE [0] elemIndex #-}

-------------------------------------------------------------------------------
-- * Zipping

-- | zipWith generalises 'zip' by zipping with the function given as
-- the first argument, instead of a tupling function.
zipWith :: (Char -> Char -> Char) -> Stream Char -> Stream Char -> Stream Char
zipWith f (Stream next0 sa0 len1) (Stream next1 sb0 len2) = Stream next (sa0 :!: sb0 :!: Nothing) (min len1 len2)
    where
      {-# INLINE next #-}
      next (sa :!: sb :!: Nothing) = case next0 sa of
                                       Done -> Done
                                       Skip sa' -> Skip (sa' :!: sb :!: Nothing)
                                       Yield a sa' -> Skip (sa' :!: sb :!: Just a)

      next (sa' :!: sb :!: Just a) = case next1 sb of
                                       Done -> Done
                                       Skip sb' -> Skip (sa' :!: sb' :!: Just a)
                                       Yield b sb' -> Yield (f a b) (sa' :!: sb' :!: Nothing)
{-# INLINE [0] zipWith #-}

errorEmptyList :: String -> a
errorEmptyList fun =
  error ("Prelude." ++ fun ++ ": empty list")
```

A.1.3 *Text/Internal.hs*

```
module Text.Internal where

import Data.Array.ST
import Data.Array.Unboxed
import Data.Word




data Text = Text !(UArray Int Word16) {-# UNPACK #-}!Int {-# UNPACK #-}!Int

empty :: Text
empty = Text (runSTUArray (newArray_ (0,0))) 0 0
{-# INLINE [1] empty #-}
```

A.1.4 *Text/UnsafeChar.hs*

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Text.UnsafeChar where

import GHC.Exts
import GHC.Prim
import GHC.Word

unsafeChr8 :: Word8 -> Char
unsafeChr8 (W8# w#) = C# (chr# (word2Int# w#))
{-# INLINE unsafeChr8 #-}

unsafeChr :: Word16 -> Char
unsafeChr (W16# w#) = C# (chr# (word2Int# w#))
{-# INLINE unsafeChr #-}

unsafeChr32 :: Word32 -> Char
unsafeChr32 (W32# w#) = C# (chr# (word2Int# w#))
{-# INLINE unsafeChr32 #-}
```

A.1.5 *Text/Utf8.hs*

```
{-# OPTIONS_GHC -fglasgow-exts #-}
module Text.Utf8 where

import Char
import Data.Bits
import Data.Word

import GHC.Exts
import GHC.Prim
import GHC.Word

between       :: Word8 -> Word8 -> Word8 -> Bool
between x y z = x >= y && x <= z
{-# INLINE between #-}

ord2   :: Char -> (Word8,Word8)
ord2 c = (x1,x2)
    where
      n  = ord c
      x1 = fromIntegral $ (shiftR n 6) + (0xC0 :: Int) :: Word8
      x2 = fromIntegral $ (n .&. 0x3F) + (0x80 :: Int) :: Word8

ord3   :: Char -> (Word8,Word8,Word8)
ord3 c = (x1,x2,x3)
    where
      n  = ord c
      x1 = fromIntegral $ (shiftR n 12) + (0xE0::Int) :: Word8
      x2 = fromIntegral $ ((shiftR n 6) .&. (0x3F::Int)) + (0x80::Int) :: Word8
      x3 = fromIntegral $ (n .&. (0x3F::Int)) + (0x80::Int) :: Word8

ord4   :: Char -> (Word8,Word8,Word8,Word8)
ord4 c = (x1,x2,x3,x4)
    where
      n  = ord c
      x1 = fromIntegral $ (shiftR n 18) + (0xF0::Int) :: Word8
      x2 = fromIntegral $ ((shiftR n 12) .&. (0x3F::Int)) + (0x80::Int) :: Word8
      x3 = fromIntegral $ ((shiftR n 6) .&. (0x3F::Int)) + (0x80::Int) :: Word8
      x4 = fromIntegral $ (n .&. (0x3F::Int)) + (0x80::Int) :: Word8

chr2       :: Word8 -> Word8 -> Char
chr2 (W8# x1#) (W8# x2#) = C# (chr# (z1# +# z2#))
    where
      y1# = word2Int# x1#
      y2# = word2Int# x2#
      z1# = uncheckedIShiftL# (y1# -# 0xC0#) 6#
      z2# = y2# -# 0x8F#
{-# INLINE chr2 #-}

chr3          :: Word8 -> Word8 -> Word8 -> Char
chr3 (W8# x1#) (W8# x2#) (W8# x3#) = C# (chr# (z1# +# z2# +# z3#))
    where
```

```
      y1# = word2Int# x1#
      y2# = word2Int# x2#
      y3# = word2Int# x3#
      z1# = uncheckedIShiftL# (y1# -# 0xE0#) 12#
      z2# = uncheckedIShiftL# (y2# -# 0x80#) 6#
      z3# = y3# -# 0x80#
{-# INLINE chr3 #-}

chr4             :: Word8 -> Word8 -> Word8 -> Word8 -> Char
chr4 (W8# x1#) (W8# x2#) (W8# x3#) (W8# x4#) =
    C# (chr# (z1# +# z2# +# z3# +# z4#))
    where
      y1# = word2Int# x1#
      y2# = word2Int# x2#
      y3# = word2Int# x3#
      y4# = word2Int# x4#
      z1# = uncheckedIShiftL# (y1# -# 0xF0#) 18#
      z2# = uncheckedIShiftL# (y2# -# 0x80#) 12#
      z3# = uncheckedIShiftL# (y3# -# 0x80#) 6#
      z4# = y4# -# 0x80#
{-# INLINE chr4 #-}

validate1    :: Word8 -> Bool
validate1 x1 = between x1 0x00 0x7F
{-# INLINE validate1 #-}

validate2       :: Word8 -> Word8 -> Bool
validate2 x1 x2 = between x1 0xC2 0xDF && between x2 0x80 0xBF
{-# INLINE validate2 #-}

validate3          :: Word8 -> Word8 -> Word8 -> Bool
validate3 x1 x2 x3 = validate3_1 x1 x2 x3 ||
                     validate3_2 x1 x2 x3 ||
                     validate3_3 x1 x2 x3 ||
                     validate3_4 x1 x2 x3
{-# INLINE validate3 #-}

validate4             :: Word8 -> Word8 -> Word8 -> Word8 -> Bool
validate4 x1 x2 x3 x4 = validate4_1 x1 x2 x3 x4 ||
                        validate4_2 x1 x2 x3 x4 ||
                        validate4_3 x1 x2 x3 x4
{-# INLINE validate4 #-}



validate3_1 x1 x2 x3 = (x1 == 0xE0) &&
                       between x2 0xA0 0xBF &&
                       between x3 0x80 0xBF
{-# INLINE validate3_1 #-}

validate3_2 x1 x2 x3 = between x1 0xE1 0xEC &&
                       between x2 0x80 0xBF &&
                       between x3 0x80 0xBF
{-# INLINE validate3_2 #-}

validate3_3 x1 x2 x3 = x1 == 0xED &&
                       between x2 0x80 0x9F &&
                       between x3 0x80 0xBF
{-# INLINE validate3_3 #-}

validate3_4 x1 x2 x3 = between x1 0xEE 0xEF &&
                       between x2 0x80 0xBF &&
                       between x2 0x80 0xBF
{-# INLINE validate3_4 #-}


validate4_1 x1 x2 x3 x4 = x1 == 0xF0 &&
                          between x2 0x90 0xBF &&
                          between x3 0x80 0xBF &&
                          between x4 0x80 0xBF
{-# INLINE validate4_1 #-}

validate4_2 x1 x2 x3 x4 = between x1 0xF1 0xF3 &&
                          between x2 0x80 0xBF &&
                          between x3 0x80 0xBF &&
                          between x4 0x80 0xBF
```

```
{-# INLINE validate4_2 #-}

validate4_3 x1 x2 x3 x4 = x1 == 0xF4 &&
                          between x2 0x80 0x8F &&
                          between x3 0x80 0xBF &&
                          between x4 0x80 0xBF
{-# INLINE validate4_3 #-}
```

A.1.6 *Text/Utf16.hs*

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Text.Utf16 where

import GHC.Exts
import GHC.Word

import Data.Word

chr2 :: Word16 -> Word16 -> Char
chr2 (W16# a#) (W16# b#) = C# (chr# (upper# +# lower# +# 0x10000#))
    where
      x# = word2Int# a#
      y# = word2Int# b#
      upper# = uncheckedIShiftL# (x# -# 0xD800#) 10#
      lower# = y# -# 0xDC00#
{-# INLINE chr2 #-}

validate1    :: Word16 -> Bool
validate1 x1 = (x1 >= 0 && x1 < 0xD800) || (x1 > 0xDFFF && x1 < 0x10000)

validate2       ::  Word16 -> Word16 -> Bool
validate2 x1 x2 = (x1 >= 0xD800 && x1 <= 0xDBFF) &&
                  (x2 >= 0xDC00 && x2 <= 0xDFFF)
```

A.2 UTF-8 Implementation Files

A.2.1 *Utf8/Internal.hs*

```
module Text.Utf8.Internal where

import Data.Array.Unboxed
import Data.Word

data Text = Text !(UArray Int Word8) !Int !Int
```

A.2.2 *Utf8/Fusion.hs*

```
{-# OPTIONS_GHC -fbang-patterns -fglasgow-exts #-}

module Text.Utf8.Fusion where

import Data.Array.Base
import Data.Word
import Control.Monad.ST
import Data.Text.UnsafeChar
import Control.Monad
import Char

import Text.Utf8
import Text.Utf8.Internal
import Text.Fusion hiding (stream,unstream)

stream :: Text -> Stream Char
stream (Text arr off len) = Stream next off len
    where
      end = off+len
      {-# INLINE next #-}
      next !i
          | i >= end = Done
```

```
          | n <= 0x7F = Yield (unsafeChr8 n)    (i + 1)
          | n <= 0xDF = Yield (chr2 n n2)       (i + 2)
          | n <= 0xEF = Yield (chr3 n n2 n3)    (i + 3)
          | otherwise = Yield (chr4 n n2 n3 n4) (i + 4)
          where
            n  = arr `unsafeAt`  i
            n2 = arr `unsafeAt` (i + 1)
            n3 = arr `unsafeAt` (i + 2)
            n4 = arr `unsafeAt` (i + 3)
{-# INLINE [0] stream #-}


unstream :: Stream Char -> Text
unstream (Stream next0 s0 len) = x `seq` (Text (fst x) 0 (snd x))
    where
      x :: ((UArray Int Word8), Int)
      x = runST ((unsafeNewArray_ (0,len+4) :: ST s (STUArray s Int Word8))
                 >>= (\arr -> loop arr 0 (len+4) s0))
      loop !arr !i !max !s
          | i + 4 > max = do arr' <- unsafeNewArray_ (0,max*2)
                             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 <= 0x7F -> do
                       unsafeWrite arr i n
                       loop arr (i+1) max s'
                  | n <= 0x07FF -> do
                       unsafeWrite arr i     (fst n2)
                       unsafeWrite arr (i+1) (snd n2)
                       loop arr (i+2) max s'
                  | n <= 0xFFFF -> do
                       unsafeWrite arr i     (fst3 n3)
                       unsafeWrite arr (i+1) (snd3 n3)
                       unsafeWrite arr (i+2) (trd3 n3)
                       loop arr (i+3) max s'
                  | otherwise  -> do
                      unsafeWrite arr i     (fst4 n4)
                      unsafeWrite arr (i+1) (snd4 n4)
                      unsafeWrite arr (i+2) (trd4 n4)
                      unsafeWrite arr (i+3) (fth4 n4)
                      loop arr (i+4) max s'
                  where
                    n  = (fromIntegral . ord) x :: Word8
                    n2 = ord2 x
                    n3 = ord3 x
                    n4 = ord4 x
                    fst3 !x = let (x1,_,_)   = x in x1
                    snd3 !x = let (_,x2,_)   = x in x2
                    trd3 !x = let (_,_,x3)   = x in x3
                    fst4 !x = let (x1,_,_,_) = x in x1
                    snd4 !x = let (_,x2,_,_) = x in x2
                    trd4 !x = let (_,_,x3,_) = x in x3
                    fth4 !x = let (_,_,_,x4) = x in x4
{-# INLINE [0] unstream #-}

{-# RULES
"STREAM stream/unstream fusion" forall s.
   stream (unstream s) = s
 #-}
```

A.3 UTF-32 Implementation Files

A.3.1 *Utf32/Internal.hs*

```
{-# OPTIONS_GHC -fbang-patterns #-}

module Text.Utf32.Fusion where

import Text.Fusion hiding (stream, unstream)
import Text.Utf32.Internal
```

```
import Text.UnsafeChar
import Data.Array.Base
import Data.Word
import Data.Array.ST
import Control.Monad.ST
import Char
import Control.Monad

stream :: Text -> Stream Char
stream (Text arr off len) = Stream next off len
    where
      end = off+len
      {-# INLINE next #-}
      next !i
          | i >= end  = Done
          | otherwise = Yield (unsafeChr32 (arr `unsafeAt` i)) (i+1)
{-# INLINE [0] stream #-}

unstream :: Stream Char -> Text
unstream (Stream next0 s0 len) = x `seq` Text (fst x) 0 (snd x)
    where
      x :: ((UArray Int Word32),Int)
      x = runST ((unsafeNewArray_ (0,len) :: ST s (STUArray s Int Word32))
                 >>= (\arr -> loop arr 0 (len) s0))
      loop arr !i !max !s
          | i > max = do arr' <-unsafeNewArray_ (0,max*2)
                         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' -> do
                  unsafeWrite arr i n
                  loop arr (i+1) max s'
                    where
                      n :: Word32
                      n = fromIntegral $ ord x
{-# INLINE [0] unstream #-}

{-# RULES
"STREAM stream/unstream fusion" forall s.
   stream (unstream s) = s
 #-}
```

A.3.2 *Utf32/Fusion.hs*

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Text.Utf32.Internal where

import Data.Array.Unboxed
import Data.Word

data Text = Text !(UArray Int Word32) !Int !Int
```

A.4 Benchmarking code

A.4.1 *BenchUtils.hs*

```
{-# OPTIONS_GHC -fglasgow-exts -fbang-patterns #-}

module BenchUtils where

import qualified Data.List as L
import Data.ByteString (ByteString(..))
import Data.Word
import Text.Printf
import System.IO
import Text.Internal (Text(..))
import System.Mem
import System.CPUTime
import Control.Exception
```

```
import Control.Concurrent

data Result = T | B

data F a = forall b. F (a -> b) | forall b. Flist (a -> [b])

class Forceable a where
    force :: a -> IO Result
    force v = v `seq` return T

instance Forceable Text

seqList = L.foldl' (flip seq) (return ())
instance Forceable [a] where
    force = L.foldl' (flip seq) (return T)

instance Forceable ByteString
instance Forceable Char
instance Forceable Bool
instance Forceable Int
instance Forceable Word8

instance (Forceable a, Forceable b) => Forceable (a,b) where
    force (a,b) = force a >> force b

instance (Forceable a, Forceable b, Forceable c) => Forceable (a,b,c) where
    force (a,b,c) = force a >> force b >> force c

run c x tests = sequence_ $ zipWith (runTest c x) [1..] tests

runTest :: Int -> a -> Int -> (String,[F a]) -> IO ()
runTest count x n (name,tests) = do
  printf "%2d " n
  fn tests
  printf "\t# %-16s\n" (show name)
  hFlush stdout
         where fn xs = case xs of
                         [f,g,h] -> runN count f x >> putStr "\t"
                                  >> runN count g x >> putStr "\t"
                                  >> runN count h x >> putStr "\t"
                         [f,g]   -> runN count f x >> putStr "\t"
                                 >> runN count g x >> putStr "\t\t"
                         [f]     -> runN count f x >> putStr "\t\t\t"
                         _       -> return ()
               run f x = performGC >> threadDelay 100 >> time f x
               runN 0 f x = return ()
               runN c f x = run f x >> runN (c-1) f x

time (Flist f) a = do
  start <- getCPUTime
  v     <- seqList (f a)
  end   <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12
  printf "%0.3f" (diff :: Double)
  hFlush stdout

time (F f) a = do
  start <- getCPUTime
  v <- evaluate (f a)
  end <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12
  printf "%0.3f" (diff :: Double)
  hFlush stdout

app1 f (x,y,z) = f x
app2 f (x,y,z) = f y
app3 f (x,y,z) = f z
```

A.4.2 Single function benchmarking (*Bench.hs*)

```
{-# OPTIONS_GHC -fglasgow-exts -fbang-patterns #-}

--module Bench where
```

```
import BenchUtils
import System.Mem
import Control.Concurrent
import Char
import Data.Array.IArray
import System.CPUTime
import System.IO
import System.IO.Unsafe
import Text.Printf
import Control.Exception

import qualified Text as T
import Text.Internal
import qualified Text.Fusion as S
import Text.Fusion (Encoding(..))

import qualified Data.List as L
import qualified Data.ByteString as B
import Data.ByteString (ByteString)
import Data.Word
import qualified System.IO.UTF8 as UTF8

main = do ascii_bs <- B.readFile "ascii.txt"
          let ascii_txt = T.decode ASCII ascii_bs
          let ascii_str = T.unpack ascii_txt
          force (ascii_txt,ascii_str,ascii_bs)
          printf " # Text\t\tString\tByteString\n"
          run 1 (ascii_txt,ascii_str,ascii_bs) ascii_tests
          performGC
          bmp_txt <- T.readFile Utf8 "bmp.txt"
          let bmp_str = T.unpack bmp_txt
          force (bmp_txt,bmp_str)
          printf " # Text\t\tString\t\n"
          run 1 (bmp_txt, bmp_str, B.empty)     bmp_tests
          performGC
          smp_sip_txt <- T.readFile Utf8 "smp_sip.txt"
          let smp_sip_str = T.unpack smp_sip_txt
          force (smp_sip_txt, smp_sip_str)
          printf " # Text\t\tString\t\n"
          run 1 (smp_sip_txt, smp_sip_str,B.empty) smp_sip_tests


ascii_tests = [
                ("cons",
                 [F (app1 (T.cons '\88')),
                  F (app2 ((:) '\88')    ),
                  F (app3 (B.cons 88) )]),
                ("head",
                 [F (app1 T.head),
                  F (app2 L.head),
                  F (app3 B.head)]),
                ("last",
                 [F (app1 T.last),
                  F (app2 L.last),
                  F (app3 B.last)]),
                ("tail",
                 [F (app1 T.tail),
                  F (app2 L.tail),
                  F (app3 B.tail)]),
                ("init",
                 [F (app1 T.init),
                  Flist (app2 L.init),
                  F (app3 B.init) ]),
                ("null",
                 [F (app1 T.null),
                  F (app2 L.null),
                  F (app3 B.null) ]),
                ("length",
                 [F (app1 T.length),
                  F (app2 L.length),
                  F (app3 B.length)]),
                 ("map",
                  [F (app1 $ T.map succ),
                   Flist (app2 (L.map succ)),
                   F (app3 $ B.map succ)]),
                 ("filter",
```

```
                  [F $ app1 $ T.filter (/= '\101'),
                   Flist $ app2 $ L.filter (/= '\101'),
                   F $ app3 $ B.filter (/= 101)]),
                 ("foldl'",
                  [F (app1 $ T.foldl' (\a w -> a+1::Int) 0),
                   F (app2 $ L.foldl' (\a w -> a+1::Int) 0),
                   F (app3 $ B.foldl' (\a w -> a+1::Int) 0)
                  ]),
                 ("drop",
                  [F (app1 $ T.drop 30000000),
                   Flist (app2 $ L.drop 30000000),
                   F (app3 $ B.drop 30000000)
                  ]),
                 ("take",
                  [F (app1 $ T.take 30000000),
                   Flist (app2 $ L.take 30000000),
                   F (app3 $ B.take 30000000)]),
                 ("words",
                  [F (app1 $ T.words),
                   Flist (app2 $ L.words)])
 ]

 bmp_tests = [
                ("cons",
                 [F (app1 (T.cons '\88')),
                  F (app2 ((:) '\88')    )]),
                ("head",
                 [F (app1 T.head),
                  F (app2 L.head)]),
                ("last",
                 [F (app1 T.last),
                  F (app2 L.last)]),
                ("tail",
                 [F (app1 T.tail),
                  F (app2 L.tail)]),
                ("init",
                 [F (app1 T.init),
                  Flist (app2 L.init)]),
                ("null",
                 [F (app1 T.null),
                  F (app2 L.null),
                  F (app3 B.null)]),
                  ("length",
                   [F (app1 T.length),
                    F (app2 L.length),
                    F (app3 B.length)]),
                 ("map",
                  [F (app1 $ T.map succ),
                   Flist (app2 (L.map succ))]),
                 ("filter",
                  [F $ app1 $ T.filter (/= '\101'),
                   Flist $ app2 $ L.filter (/= '\101')]),
                 ("foldl'",
                  [F (app1 $ T.foldl' (\a w -> a+1::Int) 0),
                   F (app2 $ L.foldl' (\a w -> a+1::Int) 0)]),
                 ("drop",
                  [F (app1 $ T.drop 30000000),
                   Flist (app2 $ L.drop 30000000)]),
                 ("take",
                  [F (app1 $ T.take 30000000),
                   Flist (app2 $ L.take 30000000)]),
                 ("words",
                  [F (app1 $ T.words),
                   Flist (app2 $ L.words)])
 ]

 smp_sip_tests = [
                ("cons",
                 [F (app1 (T.cons '\65624')),
                  F (app2 ((:) '\65624'))]),
                ("head",
                 [F (app1 T.head),
                  F (app2 L.head)]),
                ("last",
                 [F (app1 T.last),
                  F (app2 L.last)]),
```

```
                ("tail",
                 [F (app1 T.tail),
                  F (app2 L.tail)]),
                ("init",
                 [F (app1 T.init),
                  Flist (app2 L.init)]),
                ("null",
                 [F (app1 T.null),
                  F (app2 L.null),
                  F (app3 B.null) ]),
                ("length",
                 [F (app1 T.length ),
                  F (app2 L.length),
                  F (app3 B.length)]),
                 ("map",
                  [F (app1 $ T.map succ),
                   Flist (app2 (L.map succ))]),
                 ("filter",
                  [F $ app1 $ T.filter (/= '\65624'),
                   Flist $ app2 $ L.filter (/= '\65624')]),
                 ("foldl'",
                  [F (app1 $ T.foldl' (\a w -> a+1::Int) 0),
                   F (app2 $ L.foldl' (\a w -> a+1::Int) 0)]),
                 ("drop",
                  [F (app1 $ T.drop 30000000),
                   Flist (app2 $ L.drop 30000000)]),
                 ("take",
                  [F (app1 $ T.take 30000000),
                   Flist (app2 $ L.take 30000000)])
                ]
```

A.4.3 Fusion benchmarking (*FusionBench.hs*)

```
import Prelude hiding (zip,zip3,fst,snd)

import BenchUtils
import Char
import qualified Data.List as L
import qualified Data.ByteString as B
import qualified Text as T
import Text.Fusion (Encoding(..))
import qualified Text.Fusion as S
import Text.Printf
import System.IO
import System.Mem
import qualified System.IO.UTF8 as UTF8

main = do ascii_str <- readFile "ascii.txt"
          ascii_bs  <- B.readFile "ascii.txt"
          let ascii_txt = T.decode ASCII ascii_bs
          force (ascii_txt,ascii_str,ascii_bs)
          printf " # Text\t\tString\tByteString\n"
          run 1 (ascii_txt,ascii_str,ascii_bs) ascii_tests

ascii_tests =  [
 ("map/map",
  [F $ T.map pred . T.map succ . fst,
   Flist $ L.map pred . L.map succ . snd,
   F $ B.map pred . B.map succ . trd]),
 ("filter/filter",
  [F $ T.filter (/= '\101') . T.filter (/= '\102') . fst,
   Flist $ L.filter (/= '\101') . L.filter (/= '\102') . snd,
   F $ B.filter (/= 101) . B.filter (/= 102) . trd]),
 ("filter/map",
  [F $ T.filter (/= '\103') . T.map succ . fst,
   Flist $ L.filter (/= '\103') . L.map succ . snd,
   F $ B.filter (/= 103) . B.map succ . trd]),
 ("map/filter",
  [F $ T.map succ . T.filter (/= '\104') . fst,
   Flist $ L.map succ . L.filter (/= '\104') . snd,
   F $ B.map succ . B.filter (/= 104) . trd]),
 ("foldl'/map",
  [F $ T.foldl' (const . (+1)) (0 :: Int) . T.map succ . fst,
   F $ L.foldl' (const . (+1)) (0 :: Int) . L.map succ . snd,
```

```
   F $ B.foldl' (const . (+1)) (0 :: Int) . B.map succ . trd]),
 ("foldl'/filter",
  [F $ T.foldl' (const . (+2)) (0::Int) . T.filter (/= '\105') . fst,
   F $ L.foldl' (const . (+2)) (0::Int) . L.filter (/= '\105') . snd,
   F $ B.foldl' (const . (+2)) (0::Int) . B.filter (/= 105) . trd]),
 ("foldl'/map/filter",
  [F $ T.foldl' (const.(+3)) (0::Int) . T.map succ . T.filter (/='\110') . fst,
   F $ L.foldl' (const.(+3)) (0::Int) . L.map succ . L.filter (/='\110') . snd,
   F $ B.foldl' (const . (+3)) (0::Int) . B.map succ . B.filter (/= 110) . trd])
 ]
```

A.4.4 Encoding shootout (*EncodingBench.hs*)

```
{-# OPTIONS_GHC -fglasgow-exts #-}

import BenchUtils
import qualified Data.Text.Utf8.Fusion as U8
import qualified Data.Text.Utf8.Internal as U8I
import qualified Data.Text.Utf32.Fusion as U32
import qualified Data.Text.Utf32.Internal as U32I
import qualified Data.Text.Fusion as S
import Data.Text.Fusion (bsStream,Encoding(..))
import qualified Data.Text as T
import qualified Data.ByteString as B
import Text.Printf
import System.Mem

instance Forceable U32I.Text
instance Forceable U8I.Text

data E a = forall b. E (a -> b) | EText (a -> S.Stream Char)

main = do force (ascii_tests, smp_sip_tests)
          ascii       <- B.readFile "ascii.txt"
          let ascii8  = U8.unstream  (bsStream ascii ASCII)
          let ascii16 = S.unstream   (bsStream ascii ASCII)
          let ascii32 = U32.unstream (bsStream ascii ASCII)
          force (ascii8, ascii16, ascii32)
          printf " # Utf8\t\tUtf16\tUtf32\n"
          run 1 (ascii8, ascii16, ascii32) ascii_tests
          performGC
          bmp   <- B.readFile "bmp.txt"
          let bmp8  = U8.unstream  (bsStream bmp Utf8)
          let bmp16 = S.unstream   (bsStream bmp Utf8)
          let bmp32 = U32.unstream (bsStream bmp Utf8)
          force (bmp8, bmp16, bmp32)
          printf " # Utf8\t\tUtf16\tUtf32\n"
          run 1 (bmp8, bmp16, bmp32)     ascii_tests
          performGC
          smp_sip       <- B.readFile "smp_sip.txt"
          let smp_sip8  = U8.unstream (bsStream smp_sip Utf8)
          let smp_sip16 = S.unstream (bsStream smp_sip Utf8)
          let smp_sip32 = U32.unstream (bsStream smp_sip Utf8)
          force (smp_sip8, smp_sip16, smp_sip32)
          printf " # Utf8\t\tUtf16\tUtf32\n"
          run 1 (smp_sip8, smp_sip16, smp_sip32) smp_sip_tests

ascii_tests = [
    ("cons"  ,
     [F $ app1 $ U8.unstream  . S.cons '\88' . U8.stream,
      F $ app2 $ S.unstream   . S.cons '\88' . S.stream,
      F $ app3 $ U32.unstream . S.cons '\88' . U32.stream]),
    ("length",
     [F $ app1 $ S.length . U8.stream,
      F $ app2 $ S.length . S.stream,
      F $ app3 $ S.length . U32.stream]),
    ("map"   ,
     [F $ app1 $ U8.unstream  . S.map succ . U8.stream,
      F $ app2 $ S.unstream   . S.map succ . S.stream,
      F $ app3 $ U32.unstream . S.map succ . U32.stream]),
    ("filter",
     [F $ app1 $ U8.unstream  . S.filter (/= '\101') . U8.stream,
      F $ app2 $ S.unstream   . S.filter (/= '\101') . S.stream,
      F $ app3 $ U32.unstream . S.filter (/= '\101') . U32.stream]),
```

```
    ("take",
     [F $ app1 $ U8.unstream  . S.take 1000000 . U8.stream,
      F $ app2 $ S.unstream   . S.take 1000000 . S.stream,
      F $ app3 $ U32.unstream . S.take 1000000 . U32.stream]),
    ("drop"  ,
     [F $ app1 $ U8.unstream  . S.drop 1000000 . U8.stream,
      F $ app2 $ S.unstream   . S.drop 1000000 . S.stream,
      F $ app3 $ U32.unstream . S.drop 1000000 . U32.stream]),
    ("foldl'",
     [F $ app1 $ S.foldl' (\a w -> a+1::Int) 0 . U8.stream,
      F $ app2 $ S.foldl' (\a w -> a+1::Int) 0 . S.stream,
      F $ app3 $ S.foldl' (\a w -> a+1::Int) 0 . U32.stream
     ])
 ]

smp_sip_tests = [
    ("cons"  ,
     [F $ app1 $ U8.unstream  . S.cons '\88' . U8.stream,
      F $ app2 $ S.unstream   . S.cons '\88' . S.stream,
      F $ app3 $ U32.unstream . S.cons '\88' . U32.stream]),
    ("length",
     [F $ app1 $ S.length . U8.stream,
      F $ app2 $ S.length . S.stream,
      F $ app3 $ S.length . U32.stream]),
    ("map"   ,
     [F $ app1 $ U8.unstream  . S.map succ . U8.stream,
      F $ app2 $ S.unstream   . S.map succ . S.stream,
      F $ app3 $ U32.unstream . S.map succ . U32.stream]),
    ("filter",
     [F $ app1 $ U8.unstream  . S.filter (/= '\101') . U8.stream,
      F $ app2 $ S.unstream   . S.filter (/= '\101') . S.stream,
      F $ app3 $ U32.unstream . S.filter (/= '\101') . U32.stream]),
    ("take",
     [F $ app1 $ U8.unstream  . S.take 1000000 . U8.stream,
      F $ app2 $ S.unstream   . S.take 1000000 . S.stream,
      F $ app3 $ U32.unstream . S.take 1000000 . U32.stream]),
    ("drop"  ,
     [F $ app1 $ U8.unstream  . S.drop 1000000 . U8.stream,
      F $ app2 $ S.unstream   . S.drop 1000000 . S.stream,
      F $ app3 $ U32.unstream . S.drop 1000000 . U32.stream]),
    ("foldl'",
     [F $ app1 $ S.foldl' (\a w -> a+1::Int) 0 . U8.stream,
      F $ app2 $ S.foldl' (\a w -> a+1::Int) 0 . S.stream,
      F $ app3 $ S.foldl' (\a w -> a+1::Int) 0 . U32.stream])
 ]
```

A.5 Testing code

A.5.1 *Properties.hs*

```
{-# OPTIONS_GHC -fno-rewrite-rules #-}

import Test.QuickCheck
import Text.Show.Functions

import Prelude
import qualified Text as T
import Text (pack,unpack)
import qualified Text.Fusion as S
import Text.Fusion (unstream,stream)
import qualified Data.List as L


import QuickCheckUtils

prop_pack_unpack s   = (unpack . pack) s == s
prop_stream_unstream t = (unstream . stream) t == t
prop_singleton c     = [c] == (unpack . T.singleton) c

prop_cons x xs       = (x:xs) == (unpack . T.cons x . pack) xs
prop_snoc x xs       = (xs ++ [x]) == (unpack . (flip T.snoc) x . pack) xs
prop_append s1 s2    = (s1 ++ s2) == (unpack $ T.append (pack s1) (pack s2))
prop_appendS s1 s2   = (s1 ++ s2) == ((unpack . unstream) $ S.append ((stream . pack) s1) ((stream . pack) s2))
```

```
prop_head s          = not (null s) ==> head s == (T.head . pack) s
prop_last s          = not (null s) ==> last s == (T.last . pack) s
prop_lastS s         = not (null s) ==> last s == (S.last . stream . pack) s
prop_tail s          = not (null s) ==> tail s == (unpack . T.tail . pack) s
prop_tailS s         = not (null s) ==> tail s == (unpack . unstream . S.tail . stream . pack) s
prop_init s          = not (null s) ==> init s == (unpack . T.init . pack) s
prop_initS s         = not (null s) ==> init s == (unpack . unstream . S.init . stream . pack) s
prop_null s          = null s == (T.null . pack) s
prop_length s        = length s == (T.length . pack) s
prop_map f s         = (map f s) == (unpack . T.map f . pack) s
prop_intersperse c s = (L.intersperse c s) == (unpack . T.intersperse c . pack) s
prop_transpose ss    = (L.transpose ss) == (map unpack . T.transpose . map pack) ss

prop_foldl f z s     = L.foldl f z s == T.foldl f z (pack s)
prop_foldl' f z s    = L.foldl' f z s == T.foldl' f z (pack s)
prop_foldl1 f s      = not (null s) ==> L.foldl1 f s == T.foldl1 f (pack s)
prop_foldl1' f s     = not (null s) ==> L.foldl1' f s == T.foldl1' f (pack s)
prop_foldr f z s     = L.foldr f z s == T.foldr f z (pack s)
prop_foldr1 f s      = not (null s) ==> L.foldr1 f s == T.foldr1 f (pack s)

prop_concat ss       = (L.concat ss) == (unpack . T.concat . map pack) ss
prop_concatMap f s   = (L.concatMap f s) == (unpack (T.concatMap (pack . f) (pack s)))
prop_any p s         = L.any p s == T.any p (pack s)
prop_all p s         = L.all p s == T.all p (pack s)
prop_minimum s       = not (null s) ==> L.minimum s == T.minimum (pack s)
prop_maximum s       = not (null s) ==> L.maximum s == T.maximum (pack s)

prop_take n s        = L.take n s == (unpack . T.take n . pack) s
prop_drop n s        = L.drop n s == (unpack . T.drop n . pack) s
prop_takeWhile p s   = L.takeWhile p s == (unpack . T.takeWhile p . pack) s
prop_dropWhile p s   = L.dropWhile p s == (unpack . T.dropWhile p . pack) s
prop_elem c s        = L.elem c s == (T.elem c . pack) s
prop_find p s        = L.find p s == (T.find p . pack) s
prop_filter p s      = L.filter p s == (unpack . T.filter p . pack) s
prop_index x s       = x < L.length s && x >= 0 ==> (L.!!) s x == T.index (pack s) x
prop_findIndex p s   = L.findIndex p s == T.findIndex p (pack s)
prop_elemIndex c s   = L.elemIndex c s == T.elemIndex c (pack s)
prop_zipWith c s1 s2 = L.zipWith c s1 s2 == unpack (T.zipWith c (pack s1) (pack s2))
prop_words s         = L.words s == L.map unpack (T.words (pack s))
```

A.5.2 *QuickCheckUtils.hs*

```
module QuickCheckUtils where

import Test.QuickCheck
import Test.QuickCheck.Batch

import Char

import Text
import Text.Internal

instance Arbitrary Char where
    arbitrary    = oneof [choose ('\0','\55295'), choose ('\57334','\1114111')]
    coarbitrary c = variant (ord c `rem` 4)

instance Arbitrary Text where
    arbitrary     = pack `fmap` arbitrary
    coarbitrary s = coarbitrary (unpack s)
```

Appendix B

Benchmark results

These are final results from the benchmarking system used to measure the performance of *Text* versus other libraries. It is a subset of the functions implemented, but provides good coverage of the performance of *Text* functions. Nearly all the functions in *Text* have a similar complexity and structure to one of these functions.

```
[rtharper@eternity] ~/code.git/tests> ./Bench
 # Text         String  ByteString
 1 0.675        0.000   0.033           # "cons"
 2 0.000        0.000   0.000           # "head"
 3 0.000        0.263   0.000           # "last"
 4 0.000        0.000   0.000           # "tail"
 5 0.000        0.815   0.000           # "init"
 6 0.000        0.000   0.000           # "null"
 7 2.746        0.272   0.000           # "length"
 8 0.778        1.481   1.306           # "map"
 9 0.631        1.248   0.327           # "filter"
10 0.277        0.262   0.220           # "foldl'"
11 0.156        0.365   0.000           # "drop"
12 0.090        0.430   0.000           # "take"
13 0.000        5.450                   # "words"
 # Text         String
 1 0.426        0.000                   # "cons"
 2 0.000        0.000                   # "head"
 3 0.000        0.159                   # "last"
 4 0.000        0.000                   # "tail"
 5 0.000        0.467                   # "init"
 6 0.000        0.000                   # "null"
 7 1.596        0.153                   # "length"
 8 0.447        0.860                   # "map"
 9 0.343        0.779                   # "filter"
10 0.162        0.167                   # "foldl'"
11 0.156        0.211                   # "drop"
12 0.090        0.434                   # "take"
13 0.000        2.860                   # "words"
 # Text         String
 1 0.508        0.000                   # "cons"
 2 0.000        0.000                   # "head"
 3 0.000        0.098                   # "last"
 4 0.000        0.000                   # "tail"
 5 0.000        0.292                   # "init"
 6 0.000        0.000                   # "null"
 7 1.503        0.103                   # "length"
 8 0.516        0.541                   # "map"
 9 0.459        0.453                   # "filter"
10 0.121        0.098                   # "foldl'"
11 0.123        0.124                   # "drop"
12 0.113        0.312                   # "take"
```

Appendix C

Test output

```
[rtharper@eternity] ~/code.git> ghci tests/Properties.hs tests/QuickCheckUtils.hs
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Ok, modules loaded: QuickCheckUtils, Main, Text, Text.Fusion,
Text.UnsafeChar, Text.Internal, Text.Utf32, Text.Utf16, Text.Utf8.
*Main> quickCheck prop_words
Loading package array-0.1.0.0 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package random-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package unix-2.3.0.0 ... linking ... done.
Loading package process-1.0.0.0 ... linking ... done.
Loading package haskell98 ... linking ... done.
Loading package QuickCheck-1.1.0.0 ... linking ... done.
Loading package bytestring-0.9.1.0 ... linking ... done.
OK, passed 100 tests.
*Main> quickCheck prop_pack_unpack
OK, passed 100 tests.
*Main> quickCheck prop_stream_unstream
OK, passed 100 tests.
*Main> quickCheck prop_singleton
OK, passed 100 tests.
*Main> quickCheck prop_cons
OK, passed 100 tests.
*Main> quickCheck prop_snoc
OK, passed 100 tests.
*Main> quickCheck prop_append
OK, passed 100 tests.
*Main> quickCheck prop_appendS
OK, passed 100 tests.
*Main> quickCheck prop_head
OK, passed 100 tests.
*Main> quickCheck prop_lastS
OK, passed 100 tests.
*Main> quickCheck prop_last
OK, passed 100 tests.
*Main> quickCheck prop_tail
OK, passed 100 tests.
*Main> quickCheck prop_tailS
OK, passed 100 tests.
*Main> quickCheck prop_init
OK, passed 100 tests.
*Main> quickCheck prop_initS
OK, passed 100 tests.
*Main> quickCheck prop_null
OK, passed 100 tests.
*Main> quickCheck prop_length
OK, passed 100 tests.
*Main> quickCheck prop_map
OK, passed 100 tests.
*Main> quickCheck prop_intersperse
OK, passed 100 tests.
```

```
*Main> quickCheck prop_transpose
OK, passed 100 tests.
*Main> quickCheck prop_foldl
OK, passed 100 tests.
*Main> quickCheck prop_foldl'
OK, passed 100 tests.
*Main> quickCheck prop_foldl1
OK, passed 100 tests.
*Main> quickCheck prop_foldl1'
OK, passed 100 tests.
*Main> quickCheck prop_foldr
OK, passed 100 tests.
*Main> quickCheck prop_foldr1
OK, passed 100 tests.
*Main> quickCheck prop_concat
OK, passed 100 tests.
*Main> quickCheck prop_concatMap
OK, passed 100 tests.
*Main> quickCheck prop_any
OK, passed 100 tests.
*Main> quickCheck prop_all
OK, passed 100 tests.
*Main> quickCheck prop_minimum
OK, passed 100 tests.
*Main> quickCheck prop_maximum
OK, passed 100 tests.
*Main> quickCheck prop_take
OK, passed 100 tests.
*Main> quickCheck prop_drop
OK, passed 100 tests.
*Main> quickCheck prop_takeWhile
OK, passed 100 tests.
*Main> quickCheck prop_dropWhile
OK, passed 100 tests.
*Main> quickCheck prop_elem
OK, passed 100 tests.
*Main> quickCheck prop_find
OK, passed 100 tests.
*Main> quickCheck prop_filter
OK, passed 100 tests.
*Main> quickCheck prop_index
OK, passed 100 tests.
*Main> quickCheck prop_findIndex
OK, passed 100 tests.
*Main> quickCheck prop_elemIndex
OK, passed 100 tests.
*Main> quickCheck prop_zipWith
OK, passed 100 tests.
*Main> quickCheck prop_words
OK, passed 100 tests.
```

Bibliography

[1] Plans for GHC 6.10 [online]. Available from: `http://hackage.haskell.org/trac/ghc/wiki/Status/Releases`.

[2] Universal Declaration on Human Rights [online]. Available from: `http://www.unhchr.ch/udhr/lang/eng.htm`.

[3] Universal Declaration on Human Rights (Russian Translation) [online]. Available from: `http://www.unhchr.ch/udhr/lang/rus.htm`.

[4] Boehm, J. H., Atkinson, R., and Plass, M. F. Ropes: an alternative to strings. *Software Practice and Experience* (1995), 1315–1330.

[5] Claessen, K., and Hughes, J. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2000), ACM.

[6] Coutts, D., Leshchinskiy, R., and Stewart, D. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 2007 ACM SIGPLAN international conference on Functional Programming* (April 2007), SIGPLAN, ACM, pp. 315–326.

[7] Coutts, D., Stewart, D., and Leshchinskiy, R. Rewriting Haskell Strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007* (January 2007), Springer-Verlag, pp. 50–64.

[8] Gill, A., Launchbury, J., and Peyton-Jones, S. L. A short cut to deforestation. In *Proceedings of the conference on Functional Programming languages and computer architecture* (Copenhagen, Denmark, 1993), ACM, pp. 124–132.

[9] Jones, S. P., Tolmach, A., and Hoare, T. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop* (2001), ACM SIGPLAN, pp. 203–233.

[10] Peyton-Jones, S. L., Ed. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, April 2003.

[11] Svenningsson, J. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (Pittsburgh, PA, USA, 2002), ACM, pp. 124–132.

[12] The GHC Team. The Glasgow Haskell Compiler (GHC). Available from: `http://www.haskell.org/ghc`.

[13] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.8.3.* 2007.

[14] The Unicode Consortium. Summary Narrative, August 2006. Available from: `http://www.unicode.org/history/summary.html` [cited 19 August 2008].

[15] The Unicode Consortium. *The Unicode Standard, Version 5.0*, 5th ed. Addison-Wesley Professional, November 2006.

[16] The Unicode Consortium. *The Unicode Standard, Version 5.1*, 5th ed. Unicode, Inc., April 2008.

[17] Wadler, P. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73* (1990), 344–358.