A case study in Haskell string performance

David Ruane
B.A. (Mod.) Computer Science
Final Year Project April 2013
Supervisor: Glenn Strong

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

______________________________________________        ________________________
Name            Date
Acknowledgements

I’d like to thank my supervisor, Glenn Strong, for his support in this project. I’d also like to thank Andrew Anderson, who told me to fix WASH in the first place; my classmates Gareth Foster and Kieran Manning; Don Browne for advice and sympathy; and Conor Cleary for his help in proof-reading.
Abstract

Strings in Haskell are represented as singly linked lists of characters. This is a convenient but inefficient notation. Alternative array-based implementations have been developed. As Haskell is evaluated lazily, it is more informative to test the performance of a data structure in the context of a larger system. To do so, WASH, a web framework written in Haskell, will be refactored to use the Text module. Several versions will be compared, using the strict and lazy implementations of Text. The effects of stream fusion will also be considered. A series of benchmarks will then determine if the resulting performance gain is in line with the expected improvement. The results indicate that simply using the Text type will result in faster, more efficient programs. The lazy implementation of Text was observed to be more useful when streaming IO was used. In other cases it performed no better, or worse, than the strict version. Extra effort was made to utilize stream fusion but this did not show any performance gains.
Contents

1 Introduction ........................................... 4
  1.1 Objective ........................................... 4
  1.2 An example: counting lines of text .................. 5
  1.3 Fusion ............................................. 6
  1.4 Evaluating performance ............................ 7

2 Background ............................................ 8
  2.1 Text in Haskell ...................................... 8
  2.2 The problem with Strings ........................... 8
  2.3 Alternative text representations .................... 9
    2.3.1 ByteString and Lazy ByteString ............... 9
    2.3.2 Text ........................................... 9
  2.4 Stream fusion ...................................... 10
  2.5 Previous benchmarks .............................. 10

3 Refactoring and benchmarking ......................... 11
  3.1 WASH ............................................... 11
    3.1.1 Profiling ..................................... 11
  3.2 Refactoring aims ................................... 12
  3.3 Refactoring hurdles .................................. 12
    3.3.1 List traversal ................................ 13
    3.3.2 Pattern matching ............................... 13
    3.3.3 Read and Show ................................. 14
    3.3.4 Other considerations ........................... 15
  3.4 Stream fusion ...................................... 16
  3.5 Benchmarks ....................................... 17

4 Results ............................................... 19
  4.1 Test 1 ............................................ 19
  4.2 Test 2 ............................................ 20
  4.3 Test 3 ............................................ 22
    4.3.1 Small input .................................. 22
    4.3.2 Large input ................................... 23
5 Conclusions
5.1 Difficulty
5.2 Performance
  5.2.1 Laziness
  5.2.2 Fusion
5.3 Future work

A Test programs
A.1 Test1.hs
A.2 Test2.hs
A.3 Test3.hs
A.4 Test4.hs
A.5 Test5.hs

B CD — WASH Source
List of Figures

4.1 Test 1 running time ............................................ 20
4.2 Test 1 memory usage ............................................ 20
4.3 Test 2 running time ............................................ 21
4.4 Test 2 memory usage ............................................ 21
4.5 Test 3 running time for a small input ...................... 22
4.6 Test 3 memory usage for a small input ..................... 23
4.7 Test 3 running time for a large (1gb) input .............. 24
4.8 Test 3 memory usage for a large (1gb) input ............. 24
4.9 Test 3 running time for a number of different inputs .. 25
4.10 Test 3 memory usage for a number of different inputs .. 25
4.11 Test 4 running time ............................................ 26
4.12 Test 4 memory usage ............................................ 26
4.13 Test 5 running time ............................................ 27
4.14 Test 5 memory usage ............................................ 27

5.1 Results from the original Text benchmarks ................. 30

List of Tables

3.1 Results of profiling a WASH program ........................ 12
4.1 Summary of results — runtime ............................... 28
5.1 concat vs append runtime and memory use ............... 31
Chapter 1

Introduction

Haskell is a pure, functional, lazy language. In functional languages, singly-linked lists are a very common data structure. Lists allow the programmer to write functions that recursively traverse lists. Functions that utilise pattern matching. Functions can be mapped or folded over these lists. This all allows for some very elegant, concise code.

As a result, in Haskell, strings are defined as a list of characters. This is very convenient, but not very efficient. Alternative string implementations have been developed. These are based on arrays of characters allocated in memory, similar to strings in C. Among the alternative implementations are ByteString and Text. ByteString are defined as an array of bytes. Text values are arrays of UTF-16 code points. Text is therefore more convenient for strings that may contain characters outside of simple ASCII. Such structures are inherently strict, so we cannot evaluate them lazily. They are also arrays, not lists, so we cannot use functions that are polymorphic over lists.

Haskell is lazy. This has significant implications when reasoning about the performance of Haskell programs. It is not very useful to discuss the performance of a function run in isolation. Depending on how the output of a function is used, it may or may not be fully evaluated.

Strict evaluation can be introduced into a Haskell program, but this will also affect performance. Strictness can be useful when it is known in advance that an expression must be evaluated.

1.1 Objective

The goal of this project is to investigate the performance of the Text type in a real world situation. To this end an existing Haskell program that uses the String type, will be refactored to use the Text type. After this is completed, a series of benchmarks will be performed, to determine the impact on performance. Additionally, the difficulty of rewriting our code, in moving from a lazy, list based structure, to a strict, array based structure
will be discussed. The results of these benchmarks will also be compared to the results of the benchmarks presented for the Text type. [3]

1.2 An example: counting lines of text

To illustrate the performance of Haskell’s String, the following example program counts the number of lines of text in its input. Its runtime will be compared with the Unix program wc. The program will first be written using String, and then Text and ByteString.

```haskell
main :: IO ()
main = print . length . lines <$> getContents
```

It’s short, and easy to read—it reads lazily from stdin with getContents, creates a list where each element is a line from that file with lines, and finally prints the length of that list—but not very fast. Compared to wc, the following times are observed.

```
./wc < /usr/share/dict/* 2.50s user 0.06s system 99% cpu 2.564 total
wc -l /usr/share/dict/* 0.02s user 0.01s system 86% cpu 0.042 total
```

Our version may look nice, but it is significantly slower. Let’s try using Text instead. Note that because the Text API aims to mirror the set of functions for processing a String, we do not need to change our program, other than to specify that we wish to use the Text type. However this changes the behaviour of our code considerably. One important thing to note is that getContents is now strict, which means it reads the whole file into memory before passing it onto lines.

```haskell
import qualified Data.Text as T
import qualified Data.Text.IO as I

main :: IO ()
main = print . length . T.lines <$> I.getContents
```

```
./wc < /usr/share/dict/* 1.40s user 0.12s system 99% cpu 1.529 total
```

So without changing our logic or sacrificing readability, our code is already much faster. We can make this a bit faster if we count newlines to avoid creating and processing a list, and by using the Lazy implementation. Using the Lazy implementation will also allow us to stream the file, rather than reading the whole file into memory. This also means we are better able to handle larger input. The count function used here counts the number of occurrences of its first argument in its second argument, which in this case is the value returned by getContents. It is not available for lists, and therefore for the String type, but it is part of the standard Text API. The singleton function converts the ‘\n’ Char into a Text value.
import qualified Data.Text.Lazy.IO as I
import qualified Data.Text.Lazy as T

main :: IO ()
main = I.getContents >>= print . T.count (T.singleton '\n')

./wc < /usr/share/dict/* 1.11s user 0.05s system 99% cpu 1.170 total

Let's try using ByteString. We will need to change our code a little, as ByteString has no lines function, but we can count newlines as in the previous example. The important difference between Text and ByteString is that Text is better at handling Unicode values. However, in this example, Unicode isn't an issue, so we can safely use ByteString. Additionally we will use the Char8 version of ByteString, which is an array of Char8 types rather than an array of bytes. This means we don't have to convert \n as in the previous example. We could also use regular ByteString and count occurrences of 10 in the array of bytes, but it isn't immediately apparent that 10 is the value of the newline character.

import qualified Data.ByteString.Char8 as B

main :: IO ()
main = B.getContents >>= print . B.count '\n'

./wc < /usr/share/dict/* 0.03s user 0.04s system 91% cpu 0.074 total

This is a huge improvement over our original program, and getting close to performance of the C implementation. A lazy version of the ByteString type is also available that would allow for streaming the input.

From this small example, it looks like ByteString are a clear winner, performance wise. However Text offers much more utility for processing text. In particular, Text values are utf-16 encoded, which is convenient for handling text that may contain Unicode values.

A larger case study will be done to explore this. It should be noted that aside from the print function, all other functions are part of the Text or ByteString libraries in these examples. This means that we're not seeing how these types will interact within a larger context. This is important as these types will introduce strictness, and this will affect performance, and it will affect how functions are evaluated.

1.3 Fusion

Fusion, also known as deforestation, refers to a code optimization which aims to remove intermediate data structures from a program. Such structures are common in functional programs, where a number of functions are composed
to form a pipeline. For example, using stream fusion the function \( \text{map } f \circ \text{map } g \) is equivalent to \( \text{map} (f \circ g) \) \( [6] \) Such transformations can be specified using GHC’s rewrite rules. \( [3] \) Fusion allows for faster code, without the need for more verbose code.

An implementation of Haskell’s List library which uses stream fusion has been developed. \( [1] \) Additionally a number of the functions in the \textit{Text} and \textit{ByteString} modules are subject to fusion. By using these where possible GHC should be better able to detect instances of fusion, and produce faster programs.

1.4 Evaluating performance

The literature introducing the \textit{ByteString} and \textit{Text} modules includes benchmarking data. This shows that these modules are in fact faster and more compact than \textit{String}. However the benchmarks for the \textit{ByteString} and \textit{Text} modules simply look at some small test cases. The running time of single functions are compared. A number of stream fusion patterns are also compared.

It is noted that \textit{Text} usually outperforms \textit{String}, but that \textit{ByteString} are faster still. However it is also noted that while \textit{Text} is much better for manipulating strings, it’s performance is less desirable when constructing strings. \( [4] \)

Haskell is lazy. This can make reasoning about performance much more difficult. It also increases the importance of benchmarking as part of a larger system. By doing this we can make better judgements about the impact of laziness in our system. Additionally, we are looking specifically at a data structure. It is more informative to see this structure used in a real world system.
Chapter 2

Background

This chapter will outline previous work done on strings in Haskell. It will also outline the problem that this report aims to address, and how it will be addressed.

2.1 Text in Haskell

In Haskell, there are numerous types one may use to represent text. The simplest is the *String* type, defined as a singly linked list of *Char* values. This is quite convenient; lists are easy to reason with, and one can reuse functions that act over lists, such as *map*, to manipulate text. Many features of idiomatic Haskell, such as pattern-matching and tail recursion are facilitated by processing lists. However for serious text handling *String* is quite unsatisfactory, performing far worse than equivalent C code. There are a number of alternative text representations however, with varying degrees of performance and ease of use.

2.2 The problem with Strings

For each character a list of *Char* values, a number of additional values are allocated: A *Cons* cell, a pointer to the *Char*, and a pointer to the next value in the list, or *Nil*. The *Char* then contains a cell identifying it as a *Char*, and the value of the character. In total, GHC will allocate 5 machine words per character in a *String*. This adds up to 20 or 40 bytes per character, depending on the machine architecture. It also results in poor cache locality, as, for example, only 3 20 byte characters will fit on one 64 byte cache line. With preallocation this can be reduced to 12 bytes per character, but this is still considerable overhead. [2]
2.3 Alternative text representations

Due to the poor performance of Haskell's \textit{String}, several alternative libraries have been written. \textit{ByteString} are arrays of unboxed bytes allocated on the heap. It is suitable for binary data, or Strings of 8 bit characters. \textit{Text} is based on \textit{ByteString}, and is defined as an array of UTF-16 encoded byte pairs, making it much more useful for text which might contain Unicode values. The \textit{Text} and \textit{ByteString} libraries were also designed to take advantage of stream fusion, a Haskell optimization which attempts to remove intermediate data structures passed through a series of functions. \textit{ByteString} and \textit{Text} are strict data structures, since they are based on arrays, and this will affect the way we reason about Haskell which uses these types.

It would also be possible to define one's own data structure for strings, perhaps by using the Foreign Function Interface to marshal C strings. However this approach would likely be unsafe, unstable, and a great deal of effort.

2.3.1 ByteString and Lazy ByteString

\textit{ByteString} is implemented as an array of bytes allocated on the heap, an offset and a length. Through the use of the length and offset, non copying sub-strings may be used. This is also necessary to support the \texttt{head} and \texttt{tail} functions. While \textit{ByteString} is internally strict, a Lazy implementation is implemented as a list of \textit{ByteString} chunks, optimized to fit in the L2 cache. Lazy \textit{ByteString} is suitable for IO streaming, where the entire string being manipulated need not be in memory at one time, such as when reading from a large file. Lazy \textit{ByteString} also changes the complexity of functions such as \texttt{append}, \texttt{concat}, \texttt{cons} and \texttt{snoc} as they can be implemented without copying arrays, but instead manipulating the list of chunks.

2.3.2 Text

The \textit{Text} module is built on top of \textit{ByteString}, intended to store UTF-16 encoded text, rather than raw bytes. A lazy implementation also exists. \textit{Text} was designed to support the same functions that handle \textit{String} values. Internally, \textit{Text} is represented as an array of unboxed Word16 values, with offset and length indexes, similar to \textit{ByteString}. The significance of the unboxed values is that two 16-bit unboxed words can fit into one 32-bit machine word. A 16-bit boxed word will just use the lower 16 bits of a machine word. This results in a much more compact structure. As with \textit{ByteString} a lazy implementation is defined as a list of chunks of \textit{Text} values.
2.4 Stream fusion

Stream fusion is a series of optimization strategies which aim to remove intermediate data structures from compiled Haskell. It is well suited to fusing arrays, and this was an important consideration in the development of the Text and ByteString libraries.

One of the motivations in the development of the Text type is that it should take advantage of fusion as much as possible. It was especially important that the combinator functions were fusible. The reason for doing so is that when functions are combined only one Text value should be allocated. Combining functions is extremely common in Haskell, for example foldl \( f \circ map g \circ filter h \) to filter a list of values based some function \( h \), apply \( g \) to the resulting list of values and then reduce this list with the function \( f \). Without fusion intermediate values would be passed from filter to map, and then again from map to foldl. With fusion the compiler can optimize the code it generates to remove these. This is important for Text since it is based on arrays, when intermediate data structures are removed, the number of allocations required is reduced, and performance should improve.

2.5 Previous benchmarks

The literature introducing the Text and ByteString modules details benchmarks conducted on these modules. The benchmarks look at the performance of individual functions, and the performance of various fusion strategies. These benchmarks show that, in isolation, most of the functions in the Text and ByteString API outperform the String equivalents. This doesn't really tell us much. For example, the \textit{cons} function, which prepends an element is slow for Text and fast for String. Which is to be expected as Text must create a new array, whereas String need only change some pointers.

What is not shown is how this will affect performance as part of a larger system. How will introducing strictness affect the rest of the program. How will the strict and lazy flavours of Text compare in a larger system? What is the impact of stream fusion on a larger system?
Chapter 3

Refactoring and benchmarking

This chapter introduces WASH. The first part of this chapter will discuss the process of refactoring WASH to use the text module. The difficulties in migrating from String to Text are discussed. The remainder will outline the benchmarking framework used and the information that will be gathered by the benchmarks.

3.1 WASH

WASH, the Web Authoring System Haskell is a framework for writing CGI scripts in Haskell. It is a family of embedded domain specific languages for programming web applications. [7] WASH has a number of components for CGI scripting, HTML generation, email processing and database transactions.

Development on WASH predates the ByteString and Text libraries. Additionally, web applications are inherently text heavy. Not only because content on the web is sent as text, but WASH also uses string representations of some data structures internally, this is discussed in greater detail later in this chapter. Text will be used rather than ByteString due to its Unicode handling.

3.1.1 Profiling

Before refactoring could begin, I had to first get WASH to build on my platform. There have been some changes to Haskell which WASH needed to be updated to reflect. Additionally a number of modules in WASH were replaced with equivalent modules from Hackage. These were modules such as WASH’s own hand rolled Base64 encoding module, a hand rolled SHA-1 hash, and a module for handling timestamps. Standard implementations of these are all available. These implementations are also very likely to be more stable and efficient.
Once WASH was stable profiling was done to determine where WASH programs spent most of their time. A typical example is below.

<table>
<thead>
<tr>
<th>Cost centre</th>
<th>Module</th>
<th>%time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>hPutElement</td>
<td>WASH.CGI.CGIOutput</td>
<td>24.2</td>
<td>32.2</td>
</tr>
<tr>
<td>hPutElement.old</td>
<td>WASH.CGI.CGIOutput</td>
<td>21.6</td>
<td>32.2</td>
</tr>
<tr>
<td>shows_elements</td>
<td>WASH.HTML.HTMLBase</td>
<td>15.6</td>
<td>15.2</td>
</tr>
<tr>
<td>htmlAttr</td>
<td>WASH.HTML.HTMLBase</td>
<td>14.3</td>
<td>6.2</td>
</tr>
<tr>
<td>shows_attribute</td>
<td>WASH.HTML.HTMLBase</td>
<td>3.0</td>
<td>2.5</td>
</tr>
<tr>
<td>ptable.pLine</td>
<td>Main</td>
<td>2.6</td>
<td>2.7</td>
</tr>
<tr>
<td>shows_element</td>
<td>WASH.HTML.HTMLBase</td>
<td>2.2</td>
<td>0.9</td>
</tr>
<tr>
<td>shows_attributes</td>
<td>WASH.HTML.HTMLBase</td>
<td>1.7</td>
<td>0.5</td>
</tr>
<tr>
<td>showText</td>
<td>WASH.HTML.HTMLMonadBase</td>
<td>1.6</td>
<td>1.4</td>
</tr>
<tr>
<td>add</td>
<td>WASH.HTML.HTMLBase</td>
<td>1.4</td>
<td>0.9</td>
</tr>
<tr>
<td>addNode</td>
<td>WASH.HTML.HTMLMonadBase</td>
<td>1.2</td>
<td>0.7</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>WASH.CGI.CGIMonad</td>
<td>1.2</td>
<td>0.0</td>
</tr>
<tr>
<td>mkElement</td>
<td>WASH.HTML.HTMLMonadBase</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 3.1: Results of profiling a WASH program

Profiling made it clear that most of the time was spend doing IO (The function hPutElement prints a HTML element to a handle) or preparing elements for IO (shows_elements is a function mapping shows_elements over a list of elements, and combining the result. shows_element returns a string representation of a HTML element). The goal of this profiling was to determine how to improve the performance of WASH.

From profiling it became clear that IO was the largest cost and that String was the next largest. Additionally the IO function hPutElement is also doing some String processing. Therefore it was decided that by using a more efficient String implementation such as Text, we should see improvements to performance.

3.2 Refactoring aims

One very important consideration was that changes to WASH should not change the behaviour of WASH, unless this is unavoidable. This is so that changes in performance can be attributed to the move from String to Text.

3.3 Refactoring hurdles

In the simplest case, it is often sufficient to import Data.Text and replace String with Text in all type signatures. One of the aims of the Text API is to be transparent to the user. This does mean that a qualified import is necessary to avoid ambiguity between these and standard library functions.
However it is very rarely this simple. The only case where such substitution will be sufficient is where Strings are only processed using functions from the Prelude library. This was the case in the examples in Chapter 1, but it will rarely be the case in the real world, especially when considering a large framework such as WASH.

String literals will need to be handled. One option for doing this is to enable the OverloadedStrings extension in GHC. If this is not available, string literals will need to be converted to Text values with the pack function.

There are many cases where migrating String based Haskell to used Text will not be this simple.

- Any functions that traverse a String character-by-character will need to be rewritten. This is a very common pattern in idiomatic Haskell.
- Pattern matching on Strings cannot be used, but it is possible to emulate it.
- Code that uses Read and Show typeclasses.

### 3.3.1 List traversal

A very common and idiomatic pattern in Haskell is a function that recursively traverses a list one element at a time. Such code is quite simple and elegant. Written properly, it takes advantage of tail-recursion and laziness. So naturally functions that process strings can be written like this. However Text is based on arrays not lists. So when migrating to Text, any such functions will need to be rewritten.

### 3.3.2 Pattern matching

Pattern matching is very useful feature of Haskell that allows for some very elegant and expressive code. It is also very convenient when using lists, or in this case Strings. However since Text is not based on lists any code which uses pattern matching must be rewritten when migrating. If available, the ViewPatterns language extension may be used to emulate pattern matching.

View patterns are a convenient way of pattern-matching against abstract types. The most basic usage of view patterns allow for (expression → pattern) matching. In this case the expression is evaluated against the pattern. An example from WASH which checks if a string starts with the ‘=’ character:

```haskell
dropSpecialParameters = filter (f . fieldName)
  where f ('=':_ _) = False
        f _ = True
```

And the same code using view patterns:
\[
\text{dropSpecialParameters} = \text{filter} \ (f \circ \text{fieldName})
\]
\[
\text{where } f \ (\text{head} \rightarrow \text{'}='\text{')} = \text{False}
\]
\[
f _{-} = \text{True}
\]

If ViewPatterns are unavailable—when using a compiler other than GHC, or if they are removed from a future version of GHC—a different approach will be necessary. One option would be using an \textit{if...then...else} structure, but this is awkward, will grow very large if the patterns to match are complex, and it is not a very idiomatic Haskell convention.

### 3.3.3 \textit{Read and Show}

The \textit{Read} and \textit{Show} typeclasses define functions for parsing and producing String representations of a member of the class. As they deal with the \textit{String} type, and not \textit{Text}, they cannot be used with functions that process \textit{Text}. In order to use the \textit{Read} and \textit{Show} typeclasses, it will be necessary to either convert to and from Strings. Alternatively, new typeclasses can be created to handle \textit{Text} values. This could be quite difficult, depending on the types to be handled. However, this option removes a \textit{String} implementation and replaces it with a \textit{Text} one, which is the goal of this refactoring.

\textit{WASH} was making use of \textit{Read} and \textit{Show} typeclasses. The reason for this was not obvious at first. In Haskell, lists are homogeneous, which means a list of type \textit{a} may only contain \textit{a} values. However by using a list of \textit{Strings}, it is possible to use a list of string representations of any type which is a member of the \textit{Show} typeclass. This introduces the potential for runtime errors when we try to parse this string representation: it may not be the type we were expecting. This is also considered an inelegant solution to this problem, it would be more correct to use GADTs for example.

In \textit{WASH} there were a number of new types defined to encapsulate various concepts necessary for CGI and HTML. Some of these were passed around as a list to be processed elsewhere. In order to circumvent homogeneous lists, \textit{WASH} marshalls these into \textit{String} representations, and then unmarshalls them when they are to be used later.

While it would have been possible to implement this functionality differently, this would have changed the behaviour of \textit{WASH}. Any impact this might have on performance would be separate from the impact of migrating from \textit{String} to \textit{Text}. And so for every \textit{Read} and \textit{Show} instance, an equivalent \textit{ReadText} and \textit{ShowText} instance had to be implemented.

As an example the following code outlines the instances of the \textit{Read}, \textit{Show}, \textit{ReadText} and \textit{ShowText} typeclasses for the \textit{AT} data type. An \textit{AT} value is a list of lists of \textit{Strings} and two \textit{Ints}, representing the size of the structure. Oddly enough the original \textit{Read} and \textit{Show} instances discard the lists of \textit{Strings}.

```hs
\textbf{data AT =}
\textbf{ \hspace{0.5cm} AT} \ { \textbf{as_rav} :: [[\textbf{String}]\]}
```
3.3.4 Other considerations

Quite a few functions in WASH had poor documentation or obtuse names. As a result of this a considerable amount of time was spent inspecting these functions to determine their behaviour. It is important when changing the types that the function still produce the same output for a given input. An example from WASH was the `advanceIC` function.

```
advanceIC :: String -> String -> Maybe String
advanceIC [] ys = Just ys
advanceIC xs [] = Nothing
advanceIC (' ':xs) (y:ys)
    | isSpace y = advanceIC xs (dropWhile isSpace ys)
advanceIC ('\n':xs) ('\13':'\10':ys) = advanceIC xs ys
advanceIC (x:xs) (y:ys)
    | toUpper x == toUpper y = advanceIC xs ys
    | otherwise = Nothing
```

At a glance all we can discern is that this function takes two `Strings` and returns a `Maybe String`. The name suggests it advances an IC, but it’s not clear what that means. By reading through it more carefully we see that it processes its inputs recursively, one character at a time. It skips over leading whitespace ("\13","\10" is equivalent to "\r\n", or a Windows style line ending)
and then compares the two inputs. Finally if the first argument is a prefix
of the second argument, it returns \texttt{Just} the remainder of the second input.
Otherwise it returns \texttt{Nothing}.

This took some time to understand, and then some more time to determine
how to write the same function using the \textit{Text} type, as processing a
\textit{String} (or any list) one character at a time recursively like this is not the
best way to handle arrays such as \textit{Text}. Rather conveniently in this case,
there are functions in the \textit{Text} library that can be used. The resulting \textit{Text}
function is as follows.

\begin{verbatim}
advanceIC :: Text -> Text -> Maybe Text
advanceIC x y = stripPrefix (trim x) (trim y)
    where trim = stripStart . toUpper
\end{verbatim}

\subsection{Stream fusion}

In order to observe the effects of stream fusion on WASH, separate versions
were developed for both strict and lazy implementations of \textit{Text}. One version
simply used \textit{Text}. The second version used the functions in the \textit{Text} API
that are listed as subject to fusion as much as possible. Functions that are
not listed as fusible were avoided. For example \texttt{append} was used rather than
\texttt{concat}.

Additionally functions from the \textit{Data.List.Stream} library was used in
place of the \textit{Data.List} library. These libraries are identical to the programmer,
but \textit{Data.List.Stream} was implemented in order to take advantage of
stream fusion.

When first investigating the effects of stream fusion, profiling was done,
to determine which functions were used the most. These were then targeted
for changes to make greater use of fusion. One such function was
\texttt{show\_element}.

\begin{verbatim}
show\_element :: ELEMENT_ -> Text
show\_element (EMPTY_ bt tag atts) = concat [ "<", tag, show\_attributes atts , "\n/>"
show\_element (ELEMENT_ bt tag atts elts) =
    concat ["<", tag, show\_attributes atts, ">
    , show\_elements elts, "\n>
show\_element (DOCTYPE_ bt strs elems) =
    concat ["<!DOCTYPE", intercalate " " strs, "\n>
    , "!--- generated by WASH/HTML 0.11\n-->
    , show\_elements elems, "\n"
show\_element (CDATA_ bt str) = str
show\_element (COMMENT_ bt str) =
    concat ["!--", comment\_Encode str, "\n--"]
\end{verbatim}

The code above is the revised version of \texttt{show\_element} which aims to
be fusible.
Calls to `concat` in the above example were changed to calls to `append`, which is fusible. The aim in doing so was to make `show_element` more subject to fusion, as it is later used with a map function. This code does not read as well as the non-fusing code however, using

```haskell
append "abc" $ append "def" $ append "ghi" "jkl"
```
rather than

```haskell
concat ["abc", "def", "ghi", "jkl"]
```
is harder to understand, and it is also less flexible. While `concat` can operate over a list of any size, successive calls to `append` will need to be extended if the list grows. Otherwise `append` will need to be folded over the list to be joined. In Chapter 5 the consequences of this are discussed, as using folds will affect performance in ways that can be difficult to predict.

```haskell
show_element :: ELEMENT_ → T.Text
show_element (EMPTY_ bt tag atts) = T.cons '<' $ T.append tag $ T.append (show_attributes atts) "\n/>

show_element (ELEMENT_ bt tag atts elts) = T.cons '<' $ T.append tag $ T.append (show_attributes atts) $ T.append '>' $ T.append (show_elements elts) $ T.append "\n")
show_element (DOCTYPE_ bt strs elems) = T.append "<!DOCTYPE" $ T.append (T.intercalate " " strs) $ T.append "!DOCTYPE" $ T.append (show_elements elems) '\n'
show_element (CDATA_ bt str) = str 
show_element (COMMENT_ bt str) = T.append "<!--" $ T.append (commentEncode str) "$ -->"
```

## 3.5 Benchmarks

In order to show changes in performance, a benchmarking framework was created. The purpose of this was to make it easy to run a series of tests and to gather information about the test run. The framework was very simple: a script to run a program multiple times, saving runtime statistics such as execution time and memory allocation. Outliers are removed and average values are computed. This is in order to smooth out any impact that OS scheduling or other programs running in the background might have on the tests.

A variety of WASH scripts were tested. The goal was to test as many uses of `Text` as possible, including generating large sets of data, processing large files, doing so lazily and eagerly. Another consideration was that as many components of WASH should be tested as possible, as the aim was not to show that `Text` is faster, but that `Text` will still be faster when used in a larger program. For programs that read a file for input, a large test file was generated. This file contains several randomly selected words from the
english dictionary per line, and millions of lines. This should be generally representative of normal text for the purposes of these benchmarks.
Chapter 4

Results

The results of the benchmarks to WASH are presented in this chapter. For each set of results 5 version of WASH were compared:

- The original version of WASH, using Strings
- WASH using Data.Text
- WASH using Data.Text where attempts were made to take advantage of fusion
- WASH using Data.Text.Lazy
- WASH using Data.Text.Lazy where attempts were made to take advantage of fusion

A number of WASH programs were written, attempting to test WASH in different usages. The source for these programs is listed in Appendix A. Chapter 5 will contain an analysis of these results. This chapter will present the results and describe how they were obtained.

4.1 Test 1

The following program is a fairly typical WASH program. It generates a HTML page containing a large multiplication table. The goal of this is to present a simple WASH program representing a typical use case.

Running times in seconds and memory usage are graphed below. Memory usage refers to the maximum heap allocation as a percentage of the original, String version of WASH.
4.2 Test 2

This program reads in a file, and puts each line into a HTML list. In doing so WASH takes each line and encloses it in HTML $<li>$ and $</li>$ tags. This will show the performance of IO where lazy evaluation is not occurring,
as in this case the entire file is processed.

Figure 4.3: Test 2 running time

Figure 4.4: Test 2 memory usage
4.3 Test 3

Similar to the previous, this program reads in a file and puts the lines beginning with a vowel into a HTML list. The results given are for 2 input files: a large dictionary, and a 1GB file of random words. This has greater scope for lazy evaluation, a line of input can be discarded if the first character is not a vowel.

In the results here, a significant difference in memory usage between the strict and lazy implementations of Text can be seen. It is also important to note that in the second set of results, the non-lazy versions spent a lot of time thrashing as the OS struggled to fit everything in available memory.

4.3.1 Small input

![Figure 4.5: Test 3 running time for a small input]
4.3.2 Large input

A comparison of memory usage shows the advantages of the Lazy implementation of text. In total, the lazy versions used 100KB to execute this program. The strict versions on the other hand used about 4GB, as the entire file had to be read into memory. Performance statistics for the original, String based program are omitted, as my computer was unable to run this—all available memory and swap was filled, and the running time was severely affected by thrashing. The memory usage for the lazy implementations is also so low that it is not visible on the graph.

Figure 4.6: Test 3 memory usage for a small input
Figure 4.7: Test 3 running time for a large (1gb) input

Figure 4.8: Test 3 memory usage for a large (1gb) input

The next graph shows the increase in running time and memory use for larger input files for the lazy and strict version of WASH. The lazy type’s running time increases linearly while the memory used stays low. The strict Text type runs in the same time as the lazy type (The two lines are difficult to distinguish on the graph for this reason.). The memory usage of the
strict \textit{Text} type, and the \textit{String} type increases sharply with larger file sizes however.

![Graph showing test 3 running time for a number of different inputs.](image)

\textbf{Figure 4.9:} Test 3 running time for a number of different inputs

![Graph showing test 3 memory usage for a number of different inputs.](image)

\textbf{Figure 4.10:} Test 3 memory usage for a number of different inputs

\section*{4.4 Test 4}

This program simply prints a single, very long string. The overhead in creating this using \textit{String} is quite clear.
4.5 Test 5

The same as Test 4, but the input is taken from a file.
4.6 Summary

The table below presents a comparative summary of the results above. String is used as the baseline for the comparison, and so is listed as 100%.
<table>
<thead>
<tr>
<th></th>
<th>String</th>
<th>Text</th>
<th>Lazy Text</th>
<th>Fused Text</th>
<th>Fused Lazy Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>100</td>
<td>47</td>
<td>124</td>
<td>114</td>
<td>114</td>
</tr>
<tr>
<td>Test 2</td>
<td>100</td>
<td>36</td>
<td>76</td>
<td>64</td>
<td>75</td>
</tr>
<tr>
<td>Test 3</td>
<td>100</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>Test 4</td>
<td>100</td>
<td>5</td>
<td>16</td>
<td>42</td>
<td>53</td>
</tr>
<tr>
<td>Test 5</td>
<td>100</td>
<td>11</td>
<td>10</td>
<td>50</td>
<td>57</td>
</tr>
<tr>
<td>Average</td>
<td>100</td>
<td>25.6</td>
<td>49</td>
<td>58</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of results — runtime
Chapter 5

Conclusions

This chapter presents conclusions drawn from the results in chapter 4 and the work described in Chapter 3.

5.1 Difficulty

WASH is a large framework, and it uses strings in some surprising ways. One significant factor in the difficulty of migrating from String to Text was the complexity of WASH. While one module may have been trivial to refactor, changing the type, and therefore the behaviour of that module cascaded through all modules which imported it. This made incremental changes especially difficult, as changes in one module necessitated changes elsewhere and so on. The whole framework would not build until all these changes had fully propagated, and so it often wasn’t clear if a change to one module had introduced errors until all modules which used it also compiled successfully.

Additionally when changing the type of functions from String to Text the behaviour will change as these are very different types implemented in very different ways. While Haskell’s type checking is very useful in this regard, it is still important to test that the new function produces the same output as the old one. For a large framework such as WASH, a good deal of the effort involved will be figuring out what a function should produce, and ensuring that this does not change. The difficulty of this will vary depending on available documentation.

5.2 Performance

The benchmarks that were released when Text was introduced show that it is faster and more compact than String. The lazy Text type is often slower than the strict type due to the overhead of its chunked implementation, but this is not always the case. The complexity of several functions is different in the strict and lazy types, and this may have a significant effect on performance.
The results of the benchmarks in this report indicate that simply using the `Text` type instead of `String` will result in faster programs. Additionally, these will use much less memory. In most cases the strict `Text` type is faster and more compact than the lazy type. This supports claims made in the original benchmarks performed for the `Text` type. [4]

When streaming IO is desirable, the lazy `Text` type is generally preferable. The strict `Text` type requires the entire `Text` to be loaded into memory at once, which may not be practical if it is very large. However, the lazy version of `hGetContents` is unsafe, using the “back door” to the IO monad.

### 5.2.1 Laziness

The impact of lazy `Text` was only seen when attempting to read in a large file. The lazy implementation was able to use a small, constant amount of memory. The strict version on the other hand attempted to read the entire file and process it. When this exceeded available memory, most of the running time of the program was caused by the OS thrashing.

In cases where the code did not handle large `Text` values the benefits of lazy `Text` became overshadowed by the overhead of its list structure. Test 1 shows this. However, the lazy implementations of functions such as `concat` and `append` are quite different. This is because lazy `Text` values, being list based, can be combined by manipulating the spines of the lists, with no need for additional allocations. When combining strict `Text` values, a new array must be allocated to hold the result.

The code in Test 4 spends a good portion of its execution time building a large string by combining smaller strings. The running time of the strict and lazy versions are also much closer than in other tests. It appears that as the lazy version of `concat` is more efficient, it is desirable to use the lazy `Text` implementation when building text in this manner.
5.2.2 Fusion

Stream fusion is presented as a significant potential gain in performance, but attempts to make use of it where possible did not seem worthwhile. It is possible that by spending more time on fusion in WASH, using profiling to create a less naive implementation that better performance would be observed. However due to time constraints this was impractical.

In one case stream fusion caused massive slowdown. Consider the following code snippets:

```haskell
concat ["a", "list", "of", "texts"]

foldr append empty ["a", "list", "of", "texts"]
```

These are semantically identical, the output of both is “alistoftexts”. However in the Text module the `append` function is marked as “subject to fusion” while `concat` is not. Using either structure showed largely identical results in the lazy implementation of text. However when running the `foldr` version over a large input, the strict version of Text performs very, very poorly. To demonstrate this 4 programs were written. Each concatenates a large list of strings using either `concat` or `append` for both implementations of Text. Runtime statistics were as follows:

<table>
<thead>
<tr>
<th></th>
<th>Time (s)</th>
<th>Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict <code>concat</code></td>
<td>2.56</td>
<td>27634477</td>
</tr>
<tr>
<td>Strict <code>append</code></td>
<td>10885.33</td>
<td>31370672</td>
</tr>
<tr>
<td>Lazy <code>concat</code></td>
<td>3.21</td>
<td>36628</td>
</tr>
<tr>
<td>Lazy <code>append</code></td>
<td>3.29</td>
<td>36636</td>
</tr>
</tbody>
</table>

Table 5.1: `concat` vs `append` runtime and memory use

Clearly the strict implementation is doing something odd here. The lazy and strict versions of Text have very different implementations of `concat` and `append`, as the lazy implementation can simply manipulate the spines of the lists of chunks. The strict version must copy into a new array. This also helps to illustrate the difficulty in reasoning with lazy evaluation — strange behaviours such as this can be introduced from innocent looking code. Especially as we are mixing strict types and lazy evaluation in this case.

5.3 Future work

There are still some areas of WASH that, due to time constraints, have not yet been migrated to Text. With some more work this new version could be uploaded to Hackage. This includes the cookie module, the `wash2hs` parser, and some issues with Unicode corruption. Additionally WASH still makes
use of a number of hand rolled modules which could be replaced by better modules from Hackage.

It would be interesting to investigate a tool for automating the migration from String to Text. However in all but the simplest cases this would need to do a great deal of work, ensuring that types matched correctly and that the behaviour of the program was unchanged. However a very naïve tool that substituted functions from the Text library appropriately would still save the programmer a considerable amount of time. This could be achieved with simple substitution, or some type checking could be done to ensure that inappropriate substitutions are avoided. It would also be quite limited in some cases, such as the Read and Show typeclass behaviour which was discussed in Chapter 3. Such behaviours require a larger understanding of the underlying reasons behind the design choices that informed them.
Bibliography


Appendix A

Test programs

The following programs were used to test WASH. They will need to be modified slightly depending on which version of WASH is used. For the most part this will mean importing either Data.Text, Data.Text.Lazy, or not importing Data.Text at all.

A.1 Test1.hs

{-# LANGUAGE OverloadedStrings, FlexibleInstances #-}

module Main where
import WASH.CGI.CGI hiding (head, div, span, map)

main = run $ ptable 1234567890

ptable :: Integer → CGI ()
ptable mpy =
  standardQuery "Multiplication Table" $
  <table>
    mapM_ pLine [1..100000]
  </table>
  where
    pLine i =
      <tr>
        <td align="right">i</td>
        <td align="right">∗</td>
        <td align="right">mpy</td>
        <td align="right">(i * mpy)</td>
      </tr>

A.2 Test2.hs

{-# LANGUAGE OverloadedStrings #-}

module Main where
import WASH.CGI.CGI hiding (map)
import Data.Text.IO as I
import qualified Data.Text as T
import Data.List.Stream (foldr, foldl', map)
import Prelude hiding (foldr, map)

main = run ◦ listify ◦ T.lines =<< I.getContents

listify xs = standardQuery "Hello" $
    ul $ foldl' (++) (li $ text "done") $ map (li ◦ text) $ filter (T.length > 6) xs

A.3 Test3.hs

{-# LANGUAGE OverloadedStrings #-}

module Main where

import WASH.CGI.CGI hiding (map, head)
import qualified Data.Text as T
import qualified Data.Text.IO as I
import Data.List

main = run ◦ listify ◦ T.lines =<< I.getContents

listify xs = standardQuery "Hello" $
    ul $ foldl' (++) (li $ text "done") $ map (li ◦ text) $ filter myfilter xs

myfilter :: T.Text → Bool
myfilter empty = False
myfilter x = (T.head x) `elem` "aeiou"

A.4 Test4.hs

{-# LANGUAGE OverloadedStrings #-}

module Main where

import qualified Data.Text as T
import WASH.CGI.CGI

main =
    run mainCGI

mainCGI =
    (standardQuery (T.concat $ replicate 1000000 "a big ol' string") empty)

A.5 Test5.hs
module Main where

import WASH.CGI.CGI hiding (map)
import Data.Text.IO as I

main = run ◦ listify =<< I.getContents

listify xs = standardQuery xs empty
Appendix B

CD — WASH Source

The source for WASH is included in the attached CD. This contains the 5 versions used in the benchmarks. The code is also available at https://github.com/daruane/fyp as of April 2013, however this is subject to change.