Rewriting Haskell Strings

Duncan Coutts[1], Don Stewart[2], and Roman Leshchinskiy[2]

[1] Programming Tools Group
Oxford University Computing Laboratory
[2] Computer Science & Engineering
University of New South Wales
duncan.coutts@comlab.ox.ac.uk, {dons,rl}@cse.unsw.edu.au

Abstract. The Haskell *String* type is notoriously inefficient. We introduce a new data type, *ByteString*, based on lazy lists of byte arrays, combining the speed benefits of strict arrays with lazy evaluation. Equational transformations based on term rewriting are used to deforest intermediate ByteStrings automatically. We describe novel fusion combinators with improved expressiveness and performance over previous functional array fusion strategies. A library for ByteStrings is implemented, providing a purely functional interface, which approaches the speed of low-level mutable arrays in C.

Keywords: Program fusion, Deforestation, Functional programming.

1 Introduction

Haskell can be beautiful. Here we have a small Haskell program to compute the hash of the alphabetic characters in a file:

$$\begin{array}{l} \mathit{return} \cdot \mathit{foldl'}\ \mathit{hash}\ 5381 \cdot \mathit{map}\ \mathit{toLower} \cdot \mathit{filter}\ \mathit{isAlpha} =\!\ll \mathit{readFile}\ f \\ \quad \textbf{where}\ \mathit{hash}\ h\ c = h * 33 + \mathit{ord}\ c \end{array}$$

and an equivalent naive C implementation:

```
int c;
long h = 5381;
FILE *fp = fopen(f, "r");
while ((c = fgetc(fp)) != EOF)
  if (isalpha(c))
     h = h * 33 + tolower(c);
fclose(fp);
return h;
```

Although elegant, the naive Haskell program is many times slower than the naive C version! Sadly it is all too common an experience that idiomatic Haskell programs dealing with strings and I/O can have poor performance.

With some care, it is possible to produce a reasonable Haskell implementation a few times slower than the C version, but at the expense of simplicity and elegance. This is unsatisfying, as the benefits of higher abstraction are abandoned.

Ideally, we would have our cake and eat it too. That is, we would like to program in a high-level declarative style and also produce fast code that is competitive with C:

```
import Data.ByteString.Lazy.Char8 as B
return · B.foldl′ hash 5381 · B.map toLower · B.filter isAlpha =<< B.readFile f
   where hash h c = h * 33 + ord c
```

By replacing the string type with our *ByteString* representation, Haskell is able to approach the speed of C, while still retaining the elegance of the idiomatic implementation. With *stream fusion* enabled, it actually beats the original C program (Figure 1). Only by sacrificing clarity and explicitly manipulating mutable blocks is the C program able to outperform Haskell.

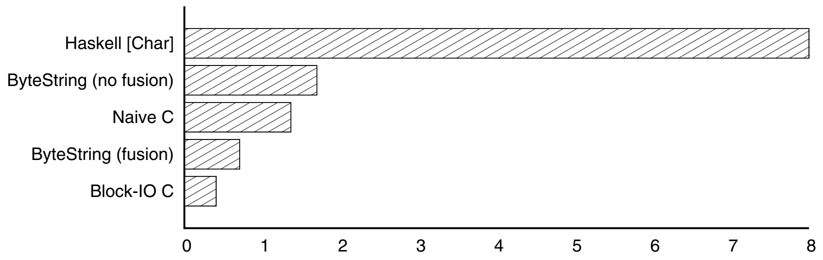

Fig. 1. Relative running times (seconds)

The main contribution of this paper is to introduce a new system for fusion, based on *streams*, offering greater expressiveness and generality than has been possible with previous work on functional array fusion [3,4]. Secondly, we describe a full scale, successful implementation of stream fusion for byte arrays, providing a fast *ByteString* type for Haskell. The implementation utilises existential types [10], the Haskell foreign function interface [2] and compiler rewrite rules [11], while presenting the user with a familiar, purely functional interface. The fusion techniques presented are not restricted to arrays or to Haskell, and should be generally applicable to sequence-like data structures, including lists.

The use of fusible array combinators dramatically improves both the time and space performance of I/O and string-based Haskell programs. Indeed, we are finally able to realise the performance promise of declarative programming in Haskell. The ByteString library is shipped with the latest Haskell implementations. The performance results therefore have practical impact, as the library is already used in performance-critical applications [1].

The remainder of the paper is organised as follows: in Section 2 we describe briefly the *ByteString* data types, both strict and lazy versions. Section 3 gives an overview of related fusion systems before presenting fusion based on streams and its application to *ByteStrings*. Section 4 explains the concrete implementation of the *ByteString* types. Section 5 presents benchmarks and finally in Section 6 we suggest further work before concluding.

2 Representing Strings

When the designers of Haskell chose a representation for strings they chose simplicity and elegance over performance:

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

The representation is certainly convenient. A wide range of polymorphic list functions are available, and the recursive structure of the list type makes it easy to write inductively defined functions. The use of a concrete, rather than abstract, data type, allows for a very expressive programming style using pattern matching.

The representation is also undeniably inefficient; for both processing and input/output. A linked list of boxed characters gives poor data density and often poor locality of reference. With the heap representation used by the Glasgow Haskell Compiler (GHC) [14] on a 32 bit machine the [*Char*] type uses 12 bytes per character[1]. This means only 5 characters fit into a 64 byte cache line.

The obvious solution to the performance problems is to use arrays of unboxed bytes. The first step is to implement an abstract type, *ByteString*, internally represented by unboxed byte arrays, along with a suite of operations over this type similar to those available for the standard *String* type. Full details of the representation are deferred to Section 4.1.

The lazy [*Char*] representation means that it is not necessary to keep the whole string resident in memory if it can be generated and consumed incrementally. Haskell supports this programming style by providing "lazy I/O": functions that transparently interleave processing of data with I/O, enabling programs to run in constant space.

A *ByteString* representation based on unboxed byte arrays, however, forces the entire string to be resident at all times – lazy I/O is impossible. When working with files larger than available memory, a strict *ByteString* representation can be simply unusable. Forcing users to explicitly manage data in blocks, as C programmers typically must do, would be a great shame in a language built on lazy evaluation. The solution to restore laziness is to define a lazy list structure containing strict elements:

```
import qualified Data.ByteString as Strict
newtype ByteString = LBS [Strict.ByteString]
```

This representation provides the best of both worlds, enabling both the performance benefits of strict *ByteStrings* and lazy processing of streams. The representation is described in more detail in Section 4.2.

3 Fusion

The program presented in the introduction is essentially a pipeline of simple computations. This is a typical example of high-level Haskell code: the ability to

[1] *Char* boxes are preallocated by GHC as an optimisation, reducing the space from 20 to 12 bytes per character.

formulate complex algorithms as compositions of primitive combinators is one of the main strengths of the functional paradigm. However, extensive optimisation is required to compile programs written in this style to efficient code. In particular, a naive implementation would create a large number of intermediate data structures, resulting in suboptimal performance with respect to both space and time.

Eliminating intermediate results is particularly important in array-based programs. Consider, for instance, the computation *sum* (*enumFromTo* 0 *n*). With lists, Haskell's non-strictness ensures that *enumFromTo* produces one element at a time which is then immediately consumed by *sum*. Thus, although an intermediate list is created the computation can still run in constant space. In the case of arrays, however, the entire intermediate array must be allocated and filled before *sum* can be applied to it. In addition to requiring $\mathcal{O}(n)$ space, this evaluation strategy is also ill-suited to modern hardware, especially with respect to cache behaviour.

If we want to generate efficient code for such computations we have to ensure that the intermediate data structure is eliminated automatically. In the context of inductive data structures, in particular lists, this problem is known as *deforestation* [15] and has been studied extensively [9]. Array fusion, on the other hand, has received comparatively little attention. In the following, we discuss a number of approaches to fusion for both arrays and lists, before describing the system used in the *ByteString* library.

3.1 Fusion Strategies

The Glasgow Haskell Compiler makes implementing fusion particularly easy due to its support for programmer-defined rewrite rules [11] which are applied by the compiler during optimisation. This allows us to specify custom equational transformations as part of the library without changing the compiler, in a manner similar to the list fusion system currently used by GHC. This flexibility has let us experiment with a number of fusion systems in the *ByteString* library. We review the most important ones below.

foldr/build. The most popular approach to list deforestation, and indeed the one used by GHC, is *foldr*/*build* fusion [7,6,5,8,13]. It requires basic list operations to be written in terms of two combinators:

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

Here, *foldr* is the list catamorphism, and *build* is an abstract list constructor. The fusion rule:

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

eliminates intermediate lists by passing the elements constructed by g directly to the consumer k. Though only a limited range of functions are fusible, this system works well and, despite initial appearances, is even applicable to non-inductive sequences such as arrays [7]. However *array* fusion based on *foldr*/*build* is currently not efficient enough to be practical. Fused array code requires a particular form of higher-order function that cannot be compiled to efficient code by current versions of GHC. For the same reason, GHC cannot produce efficient code for a fused *foldl* under this approach, greatly limiting the application of *foldr*/*build* to arrays, where many key functions make use of *foldl* traversals (for example, *sum*).

***destroy*/*unfoldr*.** Just as with *foldr*/*build* fusion, *destroy*/*unfoldr* fusion [12] defines two combinators, one for production and one for consumption:

$$\begin{aligned} &\mathit{destroy} :: (\forall b.\, (b \to \mathit{Maybe}\,(a, b)) \to b \to c) \to [a] \to c \\ &\mathit{unfoldr} :: (b \to \mathit{Maybe}\,(a, b)) \to b \to [a] \end{aligned}$$

The production of lists is captured by the list anamorphism *unfoldr*. It produces a list from the seed b and a stepper function which, given the current seed, either generates the next element and the new seed, or returns *Nothing* ending the list. List consumption is encapsulated by *destroy*. As before, intermediate lists are eliminated by a fusion rule which ensures that produced elements are immediately passed on to the consumer:

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

A major advantage of *destroy*/*unfoldr* is its support for *foldl* and *zip*-like algorithms, which cannot be implemented easily in the *foldr*/*build* framework.

One aspect that feels somewhat suboptimal is that defining functions that both produce and consume lists (such as *map*) is not totally straightforward and the full fusion transformation for them requires many steps, including an additional *destroy*/*destroy* rule.

Functional Array Fusion. Chakravarty and Keller [3,4] introduce a fusion system designed specifically for array code. It is based on a single combinator which captures left-to-right array traversals:

$$\mathit{loop}\ ::\ (s \to a \to (s,\ \mathit{Maybe}\ a)) \to s \to \mathit{Array}\ a \to (\mathit{Array}\ a,\ s)$$

The semantics of a traversal is given by a stepper function which, given a state and an array element, produces a new state and, optionally, a new element. The main fusion rule combines adjacent loops by suitably composing the stepper functions:

$$\begin{aligned} &\langle\textbf{loop/loop fusion}\rangle\ \forall\ f\ g\ s\ t\ . \\ &\quad \mathit{loop}\ f\ s \cdot \mathit{fst}\ \cdot \mathit{loop}\ g\ t\ \mapsto\ \mathit{loop}\ (\mathit{fuse}\ f\ g)\ (s, t) \end{aligned}$$

While this system has been shown to work well for standard array algorithms such as *map*, *filter* and *scan*, it does not readily support more complex computations, in particular those which process arrays from right to left or consume multiple arrays. In particular, zips can only be fused in this framework if the array type is polymorphic in the type of the elements which *ByteString* is not. Furthermore, array transformers that produce more elements than they consume cannot be implemented at all; this rules out a fusible *concatMap*.

3.2 Stream Fusion

Of the three fusion systems, our contribution is most closely related to the *destroy/unfoldr* system and indeed inherits many of its benefits.

Both *foldr/build* and *destroy/unfoldr* reflect the inductive structure of lists, effectively requiring fusible algorithms to process elements from head to tail. An array fusion framework, however, should support other access patterns if we are to effectively make use of $\mathcal{O}(1)$ array indexing. Thus, we would like to decouple the order in which array elements are read or written from the computation performed for each element. In general, we are interested in a range of single-pass algorithms which access each element exactly once. Such algorithms can be split into three phases:

– read the array producing a stream of elements,
– process the elements transforming the stream, and
– write the resulting stream into a new array.

With such a separation, access patterns can be fully captured by the read and write phases, without affecting the processing phase. Furthermore, in a pipeline composed of such computations, adjacent write/read phases can be eliminated provided they access elements in the same order.

Obviously, a crucial question is how streams of elements are represented. Since they will always be used sequentially, lists seem to be an obvious choice. However, this would leave us with the problem of eliminating intermediate lists in addition to fusing the write/read phases. We can do better than that by encapsulating a list anamorphism:

```
data Step s  = Done
             | Yield Word8 s
             | Skip s
data Stream = ∃s. Stream (s → Step s) s Int
```

Here, a *Stream* is defined by an existentially wrapped seed and a stepper function which, in each step, can indicate one of three possible results: no more elements will be produced (*Done*); a new element is produced together with a new seed (*Yield*); or a new seed is returned without producing an element (*Skip*). The last alternative, while not strictly necessary, leads to more efficient code. Streams also store a hint on the number of elements. This helps to reduce the number of costly array reallocations in the write phase. For the *ByteString* library we restrict ourselves to streams of *Word*8. The above definition, however, can be easily made polymorphic in the type of elements. For efficiency reasons, we make extensive use of strictness annotations, omitted here for clarity.

We can now easily convert an array to a stream by reading the elements from left to right (we defer the discussion of other access patterns to Section 3.6):

```
readUp   :: ByteString → Stream
readUp s =  Stream next 0 n
  where
    n                    = length s
    next i | i < n       = Yield (index s i) (i + 1)
           | otherwise   = Done
```

The implementation of $\mathit{writeUp} :: \mathit{Stream} \to \mathit{ByteString}$, which constructs an array from a stream, is omitted for space reasons but is equally straightforward.

Crucially, converting a stream to an array and then back is just the identity operation on streams. Hence, the two conversions can be eliminated, avoiding the creation of the intermediate array. This insight is captured by the following rewrite rule, which is central to our fusion framework:

$$\langle \textbf{readUp/writeUp fusion} \rangle \quad \mathit{readUp} \cdot \mathit{writeUp} \;\mapsto\; \mathit{id}$$

3.3 Stream Transformers

The reading and writing phases of array algorithms are captured by *readUp* and *writeUp*, respectively, but how do we implement the processing phase? In general, an array transformer of type $\mathit{ByteString} \to \mathit{ByteString}$ will have the form $\mathit{writeUp} \cdot h \cdot \mathit{readUp}$ where h is a stream transformer of type $\mathit{Stream} \to \mathit{Stream}$. For instance, we can implement *map* as follows:

```
map :: (Word8 → Word8) → ByteString → ByteString
map f = writeUp · mapS f · readUp
```

The actual computation is performed by *mapS*, which applies f to each element of a stream:

```
mapS :: (Word8 → Word8) → Stream → Stream
mapS f (Stream next s n) = Stream next′ s n
  where
    next′ s = case next s of
      Done       → Done
      Yield x s′ → Yield (f x) s′
      Skip s′    → Skip s′
```

With these definitions we can already fuse simple *map* pipelines:

```
  map f · map g
= writeUp · mapS f · readUp ·            {inline map ×2}
  writeUp · mapS g · readUp
= writeUp · mapS f · mapS g · readUp     {readUp/writeUp fusion}
```

Here, eliminating the $\mathit{readUp} \cdot \mathit{writeUp}$ has brought the two stream transformers together. One might expect that a separate rewrite rule is required for the two applications of *mapS* to be fused, however, as the definition of *mapS* is non-recursive, the standard optimisations performed by GHC are sufficient[2].

Indeed, it is precisely the desire to avoid recursion in stream transformers which has led us to allow stepper functions to return a new seed without producing a new element. Consider the following definition of *filter*:

```
filter :: (Word8 → Bool) → ByteString → ByteString
filter p = writeUp · filterS p · readUp
```

and the corresponding stream transformer:

[2] This is quite similar to *destroy*/*unfoldr* fusion where the compiler is expected to automatically eliminate temporary *Maybe* values.

```
filterS :: (Word8 → Bool) → Stream → Stream
filterS p (Stream next s n) = Stream next' s n
  where
    next' s = case next s of
      Done                    → Done
      Yield x s' | p x        → Yield x s'
                 | otherwise  → Skip s'
      Skip s'                 → Skip s'
```

Note how *next′* yields *Skip s′* for each deleted element. The alternative — recursively skipping to the next element satisfying the predicate — would prevent pipelines involving *filter* from being optimised satisfactorily.

3.4 Folding

Pure consumers, such as folds, are similarly easy to implement in the stream fusion framework. These algorithms only have a reading and a processing phase, so, for instance, *foldl′* is implemented as:

```
foldl' :: (a → Word8 → a) → a → ByteString → a
foldl' f z = foldlS' f z · readUp
```

where *foldlS′* folds a stream from left to right:

```
foldlS' :: (a → Word8 → a) → a → Stream → a
foldlS' f z (Stream next s n) = loop z s
  where
    loop z s = case next s of
      Done       → z
      Yield x s' → loop (f z x) s'
      Skip s'    → loop z s'
```

Some fold-like algorithms can produce a result without necessarily traversing the entire array. A prime example is *find* which searches for the first element satisfying a given predicate. We would like such computations to terminate as soon as possible while still being fusible. With *foldr*/*build* fusion this can only be done by employing laziness while with streams (and *destroy*/*unfoldr*) it can be done directly and efficiently. As before, we split the algorithm into two phases:

```
find   :: (Word8 → Bool) → ByteString → Maybe Word8
find p = findS p · readUp
```

In contrast to the algorithms presented so far, *findS* does not consume the entire stream. Instead, it returns as soon as it encounters an element which satisfies the predicate:

```
findS :: (Word8 → Bool) → Stream → Maybe Word8
findS p (Stream next s n) = loop s
  where
    loop s = case next s of
      Done                    → Nothing
      Yield x s' | p x        → Just x
                 | otherwise  → loop s'
      Skip s'                 → loop s'
```

3.5 Fusing Pipelines

We are now in the position to demonstrate how the program presented in Section 1 is transformed by GHC using our fusion framework. For simplicity, let us just consider the inner pipeline, omitting I/O-related functions:

$$\begin{aligned}
& \mathit{foldl}'\ f\ z \cdot \mathit{map}\ g \cdot \mathit{filter}\ h \\
={} & \mathit{foldlS}'\ f\ z \cdot \mathit{readUp} \cdot \mathit{writeUp} \cdot \mathit{mapS}\ g && \{\text{inline } \mathit{foldl}',\ \mathit{map} \\
& \quad \cdot \mathit{readUp} \cdot \mathit{writeUp} \cdot \mathit{filterS}\ h \cdot \mathit{readUp} && \text{ and } \mathit{filter}\} \\
={} & \mathit{foldlS}'\ f\ z \cdot \mathit{mapS}\ g \cdot \mathit{filterS}\ h \cdot \mathit{readUp} && \{\mathit{readUp}/\mathit{writeUp}\text{ fusion}\}
\end{aligned}$$

Note how the original code, which used three loops and two intermediate arrays, has been *automatically* transformed into a single array traversal. Moreover, GHC is able to further optimise the code by inlining and combining the stream transformers and, thus, eliminating intermediate *Step* values. Overall, stream fusion improves the performance of this example by a factor of around 2.4.

3.6 Down Loops

Unlike lists, arrays provide $\mathcal{O}(1)$ indexing, making left-to-right and right-to-left traversals equally efficient. Several important functions, most prominently *foldr* and its strict version *foldr′*, are best implemented as down loops. Fortunately, we can easily extend our framework with functions for reading and writing arrays from right to left:

$$\begin{aligned}
& \mathit{readDn}\ :: \mathit{ByteString} \to \mathit{Stream} \\
& \mathit{writeDn} :: \mathit{Stream} \to \mathit{ByteString}
\end{aligned}$$

Adding a fusion rule for these is straightforward:

$$\langle\textbf{readDn/writeDn fusion}\rangle \quad \mathit{readDn} \cdot \mathit{writeDn} \;\mapsto\; \mathit{id}$$

We are thus able to fuse both up and down loops equally well.

3.7 Bidirectional Loops

Combinations of up and down loops are more problematic. It is clear that it is not generally possible to directly combine up *and* down traversals into a single traversal. However, there are several important special case functions for which it would be valid to do so. Consider:

$$\begin{aligned}
& \mathit{foldr}'\ f\ z \cdot \mathit{map}\ g \\
={} & \mathit{foldrS}'\ f\ z \cdot \mathit{readDn} \cdot \mathit{writeUp} \cdot \mathit{mapS}\ g \cdot \mathit{readUp} && \{\text{inline } \mathit{foldr}' \text{ and } \mathit{map}\}
\end{aligned}$$

and we can fuse no further. However, *map* is able to generate the same result traversing either up or down, so a valid optimisation would be instead to *map* the stream in reverse, enabling fusion:

$$\begin{aligned}
& \mathit{foldrS}'\ f\ z \cdot \mathit{readDn} \cdot \mathit{writeDn} \cdot \mathit{mapS}\ g \cdot \mathit{readDn} \\
={} & \mathit{foldrS}'\ f\ z \cdot \mathit{mapS}\ g \cdot \mathit{readDn} && \{\mathit{readDn}/\mathit{writeDn}\text{ fusion}\}
\end{aligned}$$

We need a way to specially tag functions whose semantics allow them to be safely applied to either up or down streams. There is a difficulty though, as any change in stream direction, to fuse one *readDn*/*writeUp* pair, will require flipping other *readUp*s into *readDn*s. To deal with this we define wrappers over functions categorised by their direction and result type. We define:

$$\begin{aligned}
&\mathit{producerDn} && :: \mathit{Stream} \rightarrow \mathit{ByteString}\\
&\mathit{consumerDn} && :: (\mathit{Stream} \rightarrow a) \rightarrow (\mathit{ByteString} \rightarrow a)\\
&\mathit{transformerDn} && :: (\mathit{Stream} \rightarrow \mathit{Stream}) \rightarrow (\mathit{ByteString} \rightarrow \mathit{ByteString})
\end{aligned}$$

$$\begin{aligned}
&\mathit{producerDn}\, f && = \mathit{writeDn}\, f\\
&\mathit{consumerDn}\, f && = f \cdot \mathit{readDn}\\
&\mathit{transformerDn}\, f && = \mathit{writeDn} \cdot f \cdot \mathit{readDn}
\end{aligned}$$

and matching *Up* versions. From these definitions, and the existing *read/write* fusion rules, we can derive:

$\langle$**consumerDn/producerDn fusion**$\rangle\ \forall\, f\ g\ .$
$\quad \mathit{consumerDn}\ f\ (\mathit{producerDn}\ g) \ \mapsto\ f\ g$

$\langle$**consumerDn/transformerDn fusion**$\rangle\ \forall\, f\ g\ .$
$\quad \mathit{consumerDn}\ f \cdot \mathit{transformerDn}\ g \ \mapsto\ \mathit{consumerDn}\ (f \cdot g)$

$\langle$**transformerDn/producerDn fusion**$\rangle\ \forall\, f\ g\ .$
$\quad \mathit{transformerDn}\ f\ (\mathit{producerDn}\ g) \ \mapsto\ \mathit{producerDn}\ (f\ g)$

$\langle$**transformerDn/transformerDn fusion**$\rangle\ \forall\, f\ g\ .$
$\quad \mathit{transformerDn}\ f \cdot \mathit{transformerDn}\ g \ \mapsto\ \mathit{transformerDn}\ (f \cdot g)$

The rules for up loops follow the same pattern. We can now tag our traversal-independent functions as *bidirectional*, with special loop primitives:

$$\begin{aligned}
&\mathit{producerBi} && :: \mathit{Stream} \rightarrow \mathit{ByteString}\\
&\mathit{consumerBi} && :: (\mathit{Stream} \rightarrow a) \rightarrow (\mathit{ByteString} \rightarrow a)\\
&\mathit{transformerBi} && :: (\mathit{Stream} \rightarrow \mathit{Stream}) \rightarrow (\mathit{ByteString} \rightarrow \mathit{ByteString})
\end{aligned}$$

Their implementation are that of the *Up* or *Down* versions; here we will use the *Up* definition. Their use however must satisfy these side conditions:

$$\begin{aligned}
&\forall f\ .\ \mathit{producerBi}\, f && = \mathit{reverse} \cdot \mathit{producerBi}\, f\\
&\forall f\ .\ \mathit{consumerBi}\, f && = \mathit{consumerBi}\, f \cdot \mathit{reverse}\\
&\forall f\ .\ \mathit{transformerBi}\, f && = \mathit{reverse} \cdot \mathit{transformerBi}\, f \cdot \mathit{reverse}
\end{aligned}$$

Traversals that do satisfy these conditions include:

$$\begin{aligned}
&\mathit{replicate}\ x\ n && = \mathit{producerBi} && (\mathit{replicateS}\ x\ n)\\
&\mathit{sum} && = \mathit{consumerBi} && (\mathit{foldlS}'\ (+)\ 0)\\
&\mathit{map}\, f && = \mathit{transformerBi} && (\mathit{mapS}\, f)
\end{aligned}$$

Using the side conditions we can derive the fusion rules for bidirectional loops. For the derivations we will make use of a *reverse* lemma: that $\mathit{readUp} \cdot \mathit{reverse} = \mathit{readDn}$ and $\mathit{readDn} \cdot \mathit{reverse} = \mathit{readUp}$. There are many derived fusion rules; as an example, to fuse a *foldr′* with *map* we would have:

$$\begin{aligned}
&\ \ \mathit{consumerDn}\, f \cdot \mathit{transformerBi}\ g\\
&= \mathit{consumerDn}\, f \cdot \mathit{reverse} \cdot\\
&\qquad \mathit{transformerBi}\ g \cdot \mathit{reverse} && \{\text{bidirection side condition}\}\\
&= f \cdot \mathit{readDn} \cdot \mathit{reverse} \cdot && \{\text{definition of } \mathit{consumerDn} \text{ and}\\
&\qquad \mathit{writeUp} \cdot g \cdot \mathit{readUp} \cdot \mathit{reverse} && \ \text{definition of } \mathit{transformerBi}\}
\end{aligned}$$

$$\begin{aligned}
&= f \cdot \mathit{readUp} \cdot \mathit{writeUp} \cdot g \cdot \mathit{readDn} && \{\mathit{reverse}\text{ lemma } \times 2\}\\
&= f \cdot g \cdot \mathit{readDn} && \{\mathit{Up}/\mathit{Up}\text{ fusion}\}\\
&= \mathit{consumerDn}\ (f \cdot g) && \{\text{definition of } \mathit{consumerDn}\}
\end{aligned}$$

giving us the rule:

$$\langle\textbf{consumerDn/transformerBi fusion}\rangle\ \forall\, f\ g\ .$$
$$\mathit{consumerDn}\ f \cdot \mathit{transformerBi}\ g \ \mapsto\ \mathit{consumerDn}\ (f \cdot g)$$

and allowing us to fuse our example:

$$\begin{aligned}
&\mathit{foldr}'\ f\ z \cdot \mathit{map}\ g\\
&= \mathit{consumerDn}\ (\mathit{foldrS}'\ f\ z) \cdot \mathit{transformerBi}\ (\mathit{mapS}\ g) && \{\text{inline } \mathit{foldr}' \text{ and } \mathit{map}\ \}\\
&= \mathit{consumerDn}\ (\mathit{foldrS}'\ f\ z \cdot \mathit{mapS}\ g) && \{\text{fusion}\}
\end{aligned}$$

Being able to fuse bidirectional functions, such as *map*, *filter* and *length*, with such simplicity, is a great advantage: there is no penalty for using either up or down loops. The programmer can switch between *foldl′* and *foldr′* as their program requires. In contrast, *foldr/build*, and other fusion systems designed for inductive structures, have much greater difficulty with direction changes.

4 Implementation

4.1 ByteString

We implement a complete list-like interface to the *ByteString* type. To support an inductive view of strings we need a representation that supports *head* and *tail* efficiently. The simplest representation would be to use an array of unboxed bytes. However, such a structure cannot directly support *head* or *tail* without copying. The addition of offset and length fields is required. A zero-copy substring can then be constructed by simply modifying the length and offset fields.

For pragmatic reasons, instead of using Haskell's native unboxed arrays, we use a *ForeignPtr* to a contiguous block of bytes. The advantage is that this allows memory for the string to be allocated either on the Haskell GC-managed heap, or outside of Haskell (with a finaliser function to control deallocation). We can thus share *ByteStrings* with libraries written in foreign languages, such as C, without copying. For example, it is possible to memory-map a file directly to a *ByteString*, and to attach a finaliser to unmap the file when the garbage collector determines it is no longer in use. The concrete representation of *ByteStrings* is thus merely the pointer, offset and length:

```
data ByteString = BS !(ForeignPtr Word8) !Int !Int
```

GHC is able to optimise this representation by unboxing the *ForeignPtr* and the two integers into the *ByteString* constructor. There is therefore only a single indirection to access the string data.

4.2 Lazy ByteString

Lazy *ByteStrings* are represented as a list of strict *ByteString* chunks. There is some redundancy in this representation as zero-sized chunks might appear in the list, yet have no semantic value. To avoid this redundancy, empty list elements are disallowed, simplifying the logic required to manipulate lazy *ByteStrings*.

Profiling was used to find an optimal chunk size: too small, and performance approaches that of a [*Char*] structure, too large (larger than the L2 cache) and performance also falls away. In practice, a chunk size that allows the working set to fit comfortably in the L2 cache has been found to be best.

There are some additional advantages to the chunked representation: some operations requiring copying in the strict *ByteString* case only need manipulation of the spine of the lazy *ByteString* structure. For example, *append* runs in $\mathcal{O}(n/c)$ time (for chunk size c), versus $\mathcal{O}(n)$ for the strict version, with similar results for *concat*, *cons* and *snoc*. For these gains, we willingly pay a small overhead: the extra indirection from the list spine and the extra cases to consider when processing the more-complex representation.

5 Results

Comparing Haskell Lists and ByteStrings. Figure 2 compares standard [*Char*] library functions to their equivalent lazy *ByteString* implementations, applied to a 5M input string. Care is taken to explicitly force the evaluation of lazy lists, ensuring the cost of their construction is measured. As expected the lazy *ByteString* type is dramatically faster than [*Char*]. Memory usage of the fused *ByteString* is also *95% less* than that of the [*Char*] version.

Comparative Fusion Strategies. In order to quantify the effect of stream fusion, we implemented the complete functional array fusion described by Chakravarty and Keller [3,4]. The original formulation, based on the *loop* combinator, only fuses functions that make "up" traversals of arrays. We extended this system to also support fusion of down and bidirectional array traversals. In Figure 3 we compare the running time of a range of fusible strict *ByteStrings* expressions, implemented either via streams or *loop*. Each column represents a fusible expression, and we test all array traversal combinations. Results are averaged over 10 runs, with the cache dirtied between runs. The stream-based implementation of *ByteStrings* runs *on average 41% faster* than the *loop*-based implementation, and *up to 88% faster* in the best case. We believe this is because the *loop* system needs more glue code to construct the fused versions. It appears that this glue code cannot always be fully eliminated and this may also interfere with additional optimisations.

Effect of Fusion. Figure 4 measures the effect fusion has on strict *ByteStrings*, by measuring running time with and without the stream fusion rule enabled. When stream fusion occurs it greatly improves the running time of array code. Over the micro benchmark suite the average speed increase due to fusion is 74%, and 89% in the best case. The memory usage decreases by around 85% when fusion is enabled, due to the deforestation of intermediate arrays.

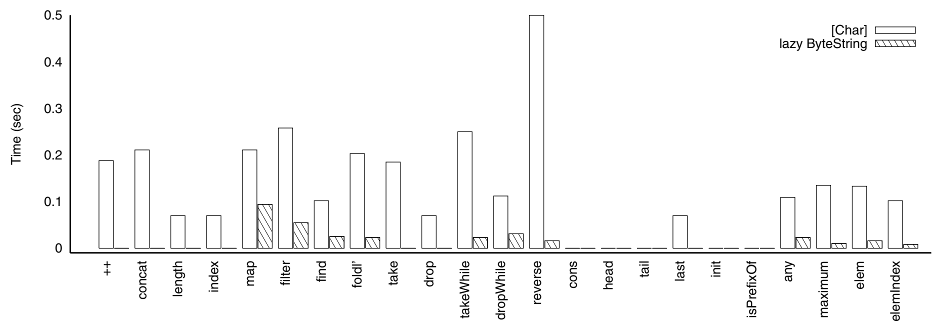

Fig. 2. [*Char*] and lazy *ByteString* : running time

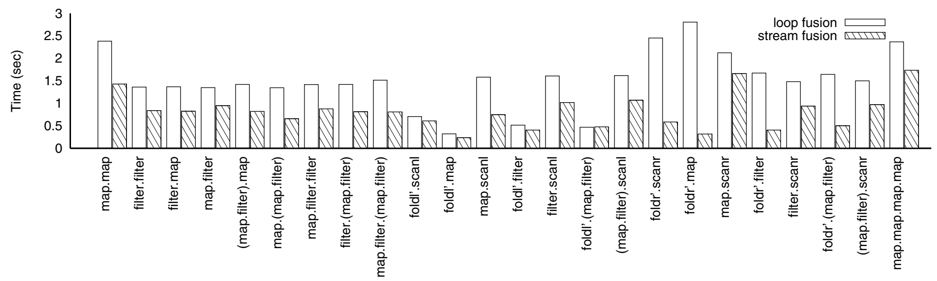

Fig. 3. Fusion strategies: loop versus stream fusion

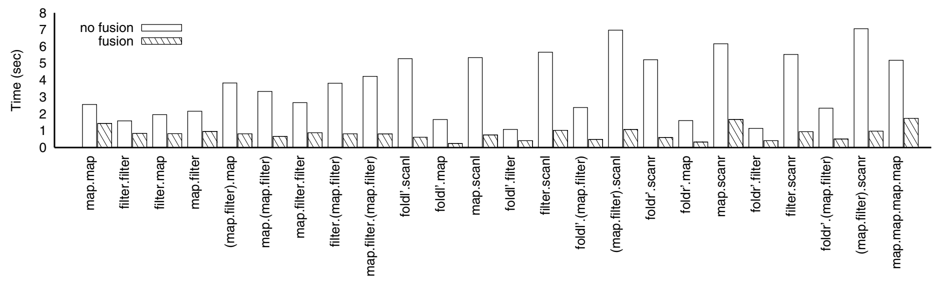

Fig. 4. Effect of fusion: streams with and without fusion

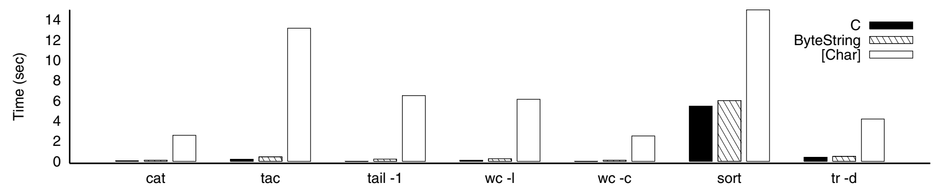

Fig. 5. Comparative results for C, ByteString and [Char] Unix tools

Comparing with C. Performance was measured against a range of standard Unix tools implemented in C in Figure 5. We measure both *ByteString* and [*Char*] implementations (one line Haskell programs) against their C equivalent. Although the C programs use a wide variety of optimisations (such as *seek*), the *ByteString* implementations are certainly competitive.

6 Further Work

More remains to be done, and this work has highlighted some promising directions for improving the performance of various aspects of Haskell.

Haskell lists. Adapting the polymorphic Haskell [*a*] type to use stream fusion, as a potential solution to the limitations of *foldr*/*build* fusion, seems a fruitful area to pursue.

Code generation. The object code GHC produces from stream combinators is fast enough that several low level issues become significant. For example, improving GHC's ability to arrange code blocks to make best use of the branch-prediction behaviour of modern CPUs is one area we wish to investigate.

Multiple traversals. A range of common functions traverse two or more streams simultaneously: for example, *append* or *zip*. Developing efficient stream fusion techniques for such functions is ongoing work.

7 Conclusion

By exploiting equational transformations via rewrite rules, it is possible to automatically fuse a wide range of array-based functions. This work goes beyond previous functional array fusion techniques by enabling fusion of bidirectional traversals and short-circuiting loops. Stream fusion is not limited to a single concrete type, but provides a general fusion mechanism for arbitrary data types expressible as streams. To demonstrate the application of stream fusion we have implemented a high-performance string processing library for Haskell, providing C-like speed, yet retaining idiomatic Haskell brevity and clarity. The source code for the *ByteString* library, all examples and a list of applications are available online [1].

Acknowledgements. We are indebted to Simon Peyton Jones for clarifying and extending GHC's rewrite rules and optimiser, which has been important to the success of this work, and to Simon Marlow for help with the GHC storage manager and *ForeignPtr* optimisations. The design of the stream fusion combinators benefited from discussions with Janis Voigtländer. We thank the anonymous referees for their helpful comments. We are also grateful to Manuel Chakravarty, Bertram Felgenhauer, Spencer Janssen, Neil Mitchell, Andres Löh, Elizabeth Baldwin, Suzie Allen and Audrey Tang for feedback on drafts.

References

1. The website accompanying this paper. `http://www.cse.unsw.edu.au/~dons/papers/CSL06.html`.
2. M. M. T. Chakravarty et al. *The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*, 2004. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.
3. M. M. T. Chakravarty and G. Keller. Functional array fusion. In X. Leroy, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM Press, 2001.
4. M. M. T. Chakravarty and G. Keller. An approach to fast arrays in Haskell. In J. Jeuring and S. P. Jones, editors, *Lecture Notes for The Summer School and Workshop on Advanced Functional Programming 2002*, pages 27–58. Springer-Verlag, 2003. LNCS 2638.
5. O. Chitil. Typer inference builds a short cut to deforestation. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 249–260, New York, NY, USA, 1999. ACM Press.
6. A. Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, University of Glasgow, January 1996.
7. A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.
8. P. Johann. Short cut fusion: Proved and improved. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 47–71, London, UK, 2001. Springer-Verlag.
9. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.
10. N. Perry. *The Implementation of Practical Functional Programming Languages.* PhD thesis, Imperial College, 1991.
11. S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop.* ACM SIGPLAN, September 2001.
12. J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 124–132, New York, NY, USA, 2002. ACM Press.
13. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 306–313. ACM Press, New York, 1995.
14. The GHC Team. The Glasgow Haskell Compiler (GHC). `http://haskell.org/ghc`, 2006.
15. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science, (Special issue of selected papers from 2nd European Symposium on Programming)*, 73(2):231–248, 1990.