FUNCTIONAL PEARLS

Inverting the Burrows-Wheeler Transform

RICHARD BIRD and SHIN-CHENG MU∗

Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract

Our aim in this pearl is to exploit simple equational reasoning to derive the inverse of the Burrows-Wheeler transform from its specification. As a bonus, we will also sketch the outlines of deriving the inverse of two more general versions of the transform, one proposed by Schindler and another by Chapin and Tate.

1 Introduction

The Burrows-Wheeler Transform (Burrows & Wheeler, 1994) is a method for permuting a string with the aim of bringing repeated characters together. As a consequence, the permuted string can be compressed effectively using simple techniques such as move-to-front or run-length encoding. In (Nelson, 1996), the article that brought the BWT to the world's attention, it was shown that the resulting compression algorithm could outperform many commercial programs available at the time. The BWT has now been integrated into a high-performance utility `bzip2`, available from (Seward, 2000).

Clearly the best way of bringing repeated characters together is just to sort the string. But this idea has a fatal flaw as a preliminary to compression: there is no way to recover the original string unless the complete sorting permutation is produced as part of the output. Instead, the BWT achieves a more modest permutation, one that aims to bring some but not all repeated characters into adjacent positions. Moreover, the transform can be inverted using a single additional piece of information, namely an integer b in the range $0 \leq b < n$, where n is the length of the output (or input) string.

It often puzzles people, at least on a first encounter, as to exactly why the BWT is invertible and how the inversion is actually carried out. Our objective in this pearl is to prove a fundamental property which made the inversion possible and, based on that, derive the inverse transform from its specification, all by equational reasoning. As a reward, we can further derive the inverse of two variations of the

∗ Thanks to Ian Bayley, Jeremy Gibbons, Geraint Jones, and Barney Stratford for constructive criticism of earlier drafts of this paper.

BWT transform, one proposed in (Schindler, 1997), another in (Chapin & Tate, 1998).

2 Defining the BWT

The BWT is specified by two functions: $bwp :: String \to String$, which permutes the string and $bwn :: String \to Int$, which computes the supplementary integer. The restriction to strings is not essential to the transform, and we can take *bwp* to have the Haskell type $Ord\ a \Rightarrow [a] \to [a]$, so lists of any type will do provided there is a total ordering relation on the elements. The function *bwp* is defined by

$$bwp \quad = \quad map\ last \cdot lexsort \cdot rots \tag{1}$$

The function $lexsort :: Ord\ a \Rightarrow [[a]] \to [[a]]$ sorts a list of lists into lexicographic order and is considered in greater detail in the following section. The function *rots* returns the rotations of a list and is defined by

$$\begin{array}{lcl} rots & :: & [a] \to [[a]] \\ rots\ xs & = & take\ (length\ xs)\ (iterate\ lrot\ xs) \end{array}$$

where $lrot\ xs = tail\ xs +\!\!+ [head\ xs]$, so *lrot* performs a single left rotation. The definition of *bwp* is constructive, but we won't go into details – at least, not in this pearl – as to how the program can be made more efficient.

The function *bwn* is specified by

$$posn\ (bwn\ xs)\ (lexsort\ (rots\ xs)) \quad = \quad xs \tag{2}$$

where $pos\ k$ applied to a list returns the element in position k. It is slightly more convenient in what follows to number positions from 1 rather than 0, so $posn\ (k + 1) = (!!k)$ in Haskell-speak. In words, $bwn\ xs$ returns some position at which xs occurs in the sorted list of rotations of xs. If xs is a repeated string, then $rots\ xs$ will contain duplicates, so $bwn\ xs$ is not defined uniquely by (2).

As an illustration, consider the string `yokohama`. The rotations and the lexicographically sorted rotations are as follows:

```
1  y o k o h a m a          6  a m a y o k o h
2  o k o h a m a y          8  a y o k o h a m
3  k o h a m a y o          5  h a m a y o k o
4  o h a m a y o k          3  k o h a m a y o
5  h a m a y o k o          7  m a y o k o h a
6  a m a y o k o h          4  o h a m a y o k
7  m a y o k o h a          2  o k o h a m a y
8  a y o k o h a m          1  y o k o h a m a
```

The output of *bwp* is the string `hmooakya`, the last column of the second matrix, and *bwn* `"yokohama"` = 8 because row number 8 in the sorted rotation is the input string.

That the BWT helps compression depends on the probability of repetitions in the input. To give a brief illustration, an English text may contain many occurrences

of words such as "this", "the", "that" and some occurrences of "where", "when", "she", " he" (with a space), etc. Consequently, many of the rotations beginning with "h" will end with a "t", some with a "w", an "s" or a space. The chance is smaller that it would end in a "x", a "q", or an "u", etc. Thus the BWT brings together a smaller subset of alphabets, say, those "t"s, "w"s and "s"s. A move-to-front encoding phase is then able to convert the characters into a series of small-numbered indexes, which improves the effectiveness of entropy-based compression techniques such as Huffman-encoding. For a fuller understanding of the role of the BWT in data compression, consult (Burrows & Wheeler, 1994; Nelson, 1996).

The inverse transform $unbwt :: Ord\ a \Rightarrow Int \to [a] \to [a]$ is specified by

$$unbwt\,(bwn\,xs)\,(bwp\,xs) \quad = \quad xs \tag{3}$$

To compute *unbwt* we have to show how the lexicographically sorted rotations of a list, or at least its tth row where $t = bwn\,xs$, can be recreated solely from the knowledge of its last column. To do so we need to examine lexicographic sorting in more detail.

3 Lexicographic sorting

Let $(\leq) :: a \to a \to Bool$ be a linear ordering on a. Define $(\leq_k) :: [a] \to [a] \to Bool$ inductively by

$$\begin{array}{lcl} xs \leq_0 ys & = & True \\ (x : xs) \leq_{k+1} (y : ys) & = & x < y \ \vee \ (x = y \ \wedge \ xs \leq_k ys) \end{array}$$

The value $xs \leq_k ys$ is defined whenever the lengths of xs and ys are equal and no smaller than k.

Now, let $sort\,(\leq_k) :: [[a]] \to [[a]]$ be a stable sorting algorithm that sorts an $n \times n$ matrix, given as a list of lists, according to the ordering $\leq_k$. Thus $sort\,(\leq_k)$, which we henceforth abbreviate to $sort\,k$, sorts a matrix on its first k columns. Stability means that rows with the same first k elements appear in their original order in the output matrix. By definition, $lexsort = sort\,n$.

Define $cols\,j = map\,(take\,j)$, so $cols\,j$ returns the first j columns of a matrix. Our aim in this section is to establish the following fundamental relationship, which is the key property establishing the existence of an algorithm for inverse BWT. Provided $1 \leq j \leq k$ we have

$$cols\,j \cdot sort\,k \cdot rots \quad = \quad sort\,1 \cdot cols\,j \cdot map\,rrot \cdot sort\,k \cdot rots \tag{4}$$

This looks daunting, but take $j = n$ (so $cols\,j$ is the identity) and $k = n$ (so $sort\,k$ is a complete lexicographic sorting). Then (4) states that the following transformation on the sorted rotations is the identity: move the last column to the front and resort the rows on the new first column. As we will see, this implies that the (stable) permutation that produces the first column from the last column is the same as that which produces the second from the first, and so on.

To prove (4) we will need some basic properties of rotations and sorting. For

rotations, one identity suffices:

$$map\ rrot \cdot rots \quad = \quad rrot \cdot rots \tag{5}$$

where *rrot* denotes a single right rotation. More generally, applying a rotation to the columns of a matrix of rotations has the same effect as applying the same rotation to the rows.

For sorting we will need

$$sort\ k \cdot map\ rrot^k \quad = \quad (sort\ 1 \cdot map\ rrot)^k \tag{6}$$

where f^k is the composition of f with itself k times. This identity formalises the fact that one can sort a matrix on its first k columns by first rotating the matrix to bring these columns into the last k positions, and then repeating k times the process of rotating the last column into first position and *stable* sorting according to the first column only. Since $map\ rrot^n = id$, the initial processing is omitted in the case $k = n$, and we have the standard definition of *radix sort*. In this context see (Gibbons, 1999) which deals with the derivation of radix sorting in a more general setting.

It follows quite easily from (6) that

$$sort\ (k+1) \cdot map\ rrot \quad = \quad sort\ 1 \cdot map\ rrot \cdot sort\ k \tag{7}$$

Finally, we will need the following two properties of columns. Firstly, for arbitrary j and k:

$$cols\ j \cdot sort\ k \quad = \quad cols\ j \cdot sort\ (j\ \mathbf{min}\ k) \ = \ sort\ (j\ \mathbf{min}\ k) \cdot cols\ j \tag{8}$$

In particular, $cols\ j \cdot sort\ k = cols\ j \cdot sort\ j$ whenever $j \leq k$. Since $sort\ j \cdot cols\ j$ is a complete sorting algorithm that does not depend on the order of the rows, we have our second property, namely,

$$sort\ j \cdot cols\ k \cdot perm \quad = \quad sort\ j \cdot cols\ k \tag{9}$$

whenever $j \leq k$ and *perm* is any function that permutes its argument.

With $1 \leq j \leq k$ we can now calculate:

$$\begin{array}{ll} & sort\ 1 \cdot cols\ j \cdot map\ rrot \cdot sort\ k \cdot rots \\ = & \quad \{\text{identity (8)}\} \\ & cols\ j \cdot sort\ 1 \cdot map\ rrot \cdot sort\ k \cdot rots \\ = & \quad \{\text{identity (7)}\} \\ & cols\ j \cdot sort\ (k+1) \cdot map\ rrot \cdot rots \\ = & \quad \{\text{identity (8)}\} \\ & cols\ j \cdot sort\ k \cdot map\ rrot \cdot rots \\ = & \quad \{\text{identity (5)}\} \\ & cols\ j \cdot sort\ k \cdot rrot \cdot rots \\ = & \quad \{\text{identity (9)}\} \\ & cols\ j \cdot sort\ k \cdot rots \end{array}$$

```
recreate :: Ord a => Int -> [a] -> [[a]]
recreate 0 ys     = map (take 0) ys
recreate (k+1) ys = sortby leq (join ys (recreate k ys))
    where leq us vs = head us <= head vs
          join xs xss = [y:ys | (y,ys) <- zip xs xss]
```

Fig. 1. Computation of *recreate*

Thus, (4) is established.

4 The derivation

Our aim is to develop a program which reconstructs the sorted matrix from its last column. In other words, we wish to construct $sort\, n \cdot rots \cdot unbwt\, t$. In fact, we will try to construct a more general expression $cols\, j \cdot sort\, k \cdot rots \cdot unbwt\, t$ such that the same derivation still applies for Section 5.

First observer that for $1 \leq j$

$$cols\, j \cdot map\, rrot \quad = \quad join \cdot fork\, (map\, last, cols\, (j-1))$$

where $join\, (xs, xss)$ is the matrix xss with xs adjoined as a new column, and $fork\, (f, g)\, x = (f\, x, g\, x)$. Hence

$$\begin{array}{ll} & cols\, j \cdot sort\, k \cdot rots \cdot unbwt\, t \\ = & \quad \{\text{property (4)}\} \\ & sort\, 1 \cdot cols\, j \cdot map\, rrot \cdot sort\, k \cdot rots \cdot unbwt\, t \\ = & \quad \{\text{above}\} \\ & sort\, 1 \cdot join \cdot fork\, (map\, last, cols\, (j-1)) \cdot sort\, k \cdot rots \cdot unbwt\, t \\ = & \quad \{\text{since } fork\, (f, g) \cdot h = fork\, (f \cdot h, g \cdot h)\} \\ & sort\, 1 \cdot join \cdot fork\, (map\, last \cdot sort\, k \cdot rots \cdot unbwt\, n, \\ & \qquad\qquad cols\, j \cdot sort\, k \cdot rots \cdot unbwt\, t) \\ = & \quad \{\text{definition of } bwp\} \\ & sort\, 1 \cdot join \cdot fork\, (bwp \cdot unbwt\, t, cols\, j \cdot sort\, k \cdot rots \cdot unbwt\, t) \end{array}$$

If t is $bwn\, xs$ where xs is the input, we have $bwp \cdot unbwt\, t = id$. Setting $recreate\, j = cols\, j \cdot sort\, k \cdot rots \cdot unbwt\, t$, we therefore have

$$\begin{array}{lcl} recreate\, 0 & = & map\, (take\, 0) \\ recreate\, (j+1) & = & sort\, 1 \cdot join \cdot fork\, (id, recreate\, j) \end{array}$$

The last equation is valid only for $j + 1 \leq k$. The Haskell code for *recreate* is given in Figure 1. The function $sortby :: Ord\, a \Rightarrow (a \to a \to Bool) \to [a] \to [a]$ is a stable variant of the standard function *sortBy*.

In particular, taking $k = n$ we have $unbwt\, t = posn\, t \cdot recreate\, n$. This implementation of *unbwt* involves computing $sort\, 1$ a total of n times. To avoid repeated sorting, observe that $recreate\, 1\, ys = sort\, (\leq)\, ys$, where $sort\, (\leq)$ sorts a list rather

```
unbwt :: Ord a => Int -> [a] -> [a]
unbwt t ys = take (length ys) (thread (spl t))
  where thread (x,j) = x:thread (spl j)
        spl = lookup (zip [1..] (sortby (<=) (zip ys [1..])))
```

Fig. 2. Computation of *unbwt*

than a matrix of one column. Furthermore, for some suitable permutation sp we have

$$sort\,(\leq)\,ys \quad = \quad apply\;sp\;ys$$

where $apply :: (Int \to Int) \to [a] \to [a]$ applies a permutation to a list:

$$apply\;p\,[x_1,\ldots,x_n] \quad = \quad [x_{p(1)},\ldots,x_{p(n)}]$$

It follows that

$$recreate\,(j+1)\,ys \quad = \quad join\,(apply\;sp\;ys, apply\;sp\,(recreate\;j\;ys))$$

Equivalently,

$$recreate\;n\;ys \quad = \quad transpose\,(take\;n\,(iterate1\,(apply\;sp)\;ys)) \tag{10}$$

where $transpose :: [[a]] \to [[a]]$ is the standard Haskell function for transposing a matrix and $iterate1 = tail \cdot iterate$. The tth row of a matrix is the tth column of the transposed matrix, ie. $posn\;t \cdot transpose = map\,(posn\;t)$, so we can use the naturality of $take\;n$ to obtain

$$unbwt\;t\;ys \quad = \quad take\;n\,(map\,(posn\;t)\,(iterate1\,(apply\;sp)\;ys))$$

Suppose we define $spl :: Ord\;a \Rightarrow [a] \to Int \to (a, Int)$ by

$$spl\;ys \quad = \quad lookup\,(zip\,[1..]\,(sort\,(\leq)\,(zip\;ys\,[1..])))$$

where $lookup :: Eq\;a \Rightarrow [(a,b)] \to (a \to b)$ is a standard function and $sort\,(\leq)$ sorts a list of pairs. Then

$$spl\;ys\;j \quad = \quad (posn\,(sp\;j)\;ys, sp\;j)$$

Hence

$$map\,(posn\;k)\,(iterate1\,(apply\;sp)\;ys) \quad = \quad thread\,(spl\;ys\;k)$$

where $thread\,(x,j) = x : thread\,(spl\;ys\;j)$.

The final algorithm, written as a Haskell program, is given in Figure 2. If we use arrays with constant-time lookup, the time to compute *unbwt* is dominated by the sorting in *spl*.

5 Schindler's variation

The main variation of BWT is to exploit the general form of (4) rather than the special case $k = n$. Suppose we define

$$bwpS\;k \quad = \quad map\;last \cdot sort\;k \cdot rots$$

```
unbwt :: Ord a => Int -> Int -> [a] -> [a]
unbwt k p ys = us ++ reverse (take (length ys - k) vs)
  where us  = posn p yss
        yss = recreate k ys
        vs  = u:search k (reverse (zip yss ys)) (take k (u:us))
        u   = posn p ys

search :: Eq a => Int -> [([a],a)] -> [a] -> [a]
search k table xs = x:search k table' (take k (x:xs))
                    where (x,table') = dlookup table xs

dlookup :: Eq a => [(a,b)] -> a -> (b,[(a,b)])
dlookup ((a,b):abs) x = if a==x then (b,abs)
                        else (c,(a,b):cds)
                        where (c,cds) = dlookup abs x
```

Fig. 3. Computation of Schindler's variation

This version, which sorts only on the first k columns of the rotations of a list, was considered in (Schindler, 1997). The derivation of the previous section shows how we can recreate the first k columns of the sorted rotations from $ys = bwp\,k\,xs$, namely by computing $recreate\,k\,ys$.

The remaining columns cannot be computed in the same way. However, we can reconstruct the tth row, where $t = bwn\,k\,xs$ and

$$posn\,(bwn\,k\,xs)\,(sort\,k\,(rots\,xs)) \quad = \quad xs$$

The first k elements of xs are given by $posn\,t\,(create\,k\,ys)$, and the last element of xs is $posn\,t\,ys$. So certainly we know

$$take\,k\,(rrot\,xs) \quad = \quad [x_n, x_1, \ldots, x_{k-1}]$$

This list begins the *last* row of the unsorted matrix, and consequently, since sorting is stable, will be the *last* occurrence of the list in $create\,k\,ys$. If this occurrence is at position p, then $posn\,p\,ys = x_{n-1}$. Having discovered x_{n-1}, we know $take\,k\,(rrot^2\,xs)$. This list begins the penultimate row of the unsorted matrix, and will be either the last occurrence of the list in the sorted matrix, or the penultimate one if it is equal to the previous list. We can continue this process to discover all of $[x_{k+1}, \ldots, x_n]$ in reverse order. Efficient implementation of this phase of the algorithm requires building an appropriate data structure for repeatedly looking up elements in reverse order in the list $zip\,(recreate\,k\,ys)\,ys$ and removing them when found. A simple implementation is given in Figure 3.

6 Chapin and Tate's variation

Just for the purpose of showing that the pattern of derivation in this paper can be adapted to other cases, we will consider another variation. Define the following alternative of BWT:

$$bwpCT\,k \quad = \quad map\,last \cdot twists\,k \cdot lexsort \cdot rots$$

Here the function *twists k* rearranges the rows of the matrix. It can be defined as a sequence of steps:

$$\begin{array}{lcl} \mathit{twists}\,0 & = & \mathit{id} \\ \mathit{twists}\,(k+1) & = & \mathit{tstep}\,(k+1) \cdot \mathit{twists}\,k \end{array}$$

Each of the *tstep k* rearranges the matrix based on only the value of the first k columns. This idea was proposed by (Chapin & Tate, 1998), where they considered a choice of *twists*, based on gray code, which marginally improves the compression ratio of the transformed text.

To derive an algorithm performing the reverse transform, we assume an input *ys* to start with

$$(\mathit{col}\,j \cdot \mathit{twists}\,k \cdot \mathit{lexsort} \cdot \mathit{rots})\;\mathit{ys}$$

and follow a derivation similar to the reverse of that of Section 3, which will eventually show that the above expression is equivalent to

$$(\mathit{apply}\,q \cdot \mathit{sort}_1 \cdot \mathit{apply}\,q' \cdot \mathit{cols}\,j \cdot \mathit{map}\,\mathit{rrot} \cdot \mathit{twists}\,k \cdot \mathit{lexsort} \cdot \mathit{rots})\;\mathit{ys}$$

where q is a permutation capturing the effects of *twists k* on the input *ys*, while q' is the inverse permutation of q. In words, the equality means that if we start with the sorted and twisted matrix, move the last column to the first, untwist it, sort it by the first character, and twist it again, we get the same matrix we started with. The role of this equality is similar to that of (4) in the derivation of the standard inverse BWT.

Based on the relationship established above, one would try to derive an algorithm in a way similar to that in Section 4, starting with

$$\mathit{recreateCT}\;j\;k \quad = \quad \mathit{col}\,j \cdot \mathit{twists}\,k \cdot \mathit{lexsort} \cdot \mathit{rots} \cdot \mathit{unbwtCT}\;t$$

which eventually leads to (10), except for that the permutation *sp* must take the twisting q into account. Recall that *twists* is composed of many steps, with step i inspecting only the first i column. Every time a new column is computed, we can construct a new permutation q simulating what *twists* would do and integrate it into *sp*.

The resulting algorithm would thus be a *paramorphism* returning a pair whose first component is the reconstructed matrix and the second component is a permutation representating q. Further details will be omitted for this pearl.

7 Conclusions

We have shown how the inverse Burrows-Wheeler transform can be derived by equational reasoning. The derivation can be re-used to invert the more general versions proposed by Schindler and by Chapin and Tate.

Other aspects of the BWT also make interesting topics. The BWT can be modified to sort the tails of a list rather than its rotations, and in (Manber & Myers, 1993) it is shown how to do this in $O(n \log n)$ steps using suffix arrays. How fast it can be done in a functional setting remains unanswered, though we conjecture that $O(n(\log n)^2)$ step is the best possible.

References

Burrows, Mike, & Wheeler, David J. (1994). *A block-sorting lossless data compression algorithm*. Tech. rept. Digital Systems Research Center. Research Report 124.

Chapin, Brenton Kenneth, & Tate, Steve. (1998). Higher compression from the Burrows-Wheeler transform by modified sorting. *Page 532 of: Data compression conference 1998*. IEEE Computer Society Press. (Poster Session).

Gibbons, Jeremy. (1999). A pointless derivation of radixsort. *Journal of functional programming*, **9**(3), 339–346.

Manber, Udi, & Myers, Gene. (1993). Suffix arrays: A new method for on-line string searches. *Siam journal on computing*, **22**(5), 935–948.

Nelson, Mark. (1996). Data compression with the Burrows-Wheeler transform. *Dr. dobb's journal*, September.

Schindler, Michael. (1997). A fast block-sorting algorithm for loseless data compression. *Page 469 of: Data compression conference 1997*. IEEE Computer Society Press. (Poster Session).

Seward, Julian. (2000). `bzip2`. `http://sourceware.cygnus.com/bzip2/`.