Pretty Printing with Lazy Dequeues

OLAF CHITIL
University of Kent, UK

There are several purely functional libraries for converting tree structured data into indented text, but they all make use of some backtracking. Over twenty years ago Oppen published a more efficient imperative implementation of a pretty printer. This paper shows that the same efficiency is also obtainable without destructive updates by developing a similar but purely functional Haskell implementation with the same complexity bounds. At its heart lie two lazy double ended queues.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; E.1 [**Data Structures**]: Lists, Stacks, and Queues

General Terms: Algorithms, Languages

1. PRETTY PRINTING

Pretty printing is the task of converting tree structured data into text, such that the indentation of lines reflects the tree structure. Furthermore, to minimise the number of lines of the text, substructures are put on a single line as far as possible within a given line-width limit. Here is the result of pretty printing an expression within a width of 35 characters:

```
|if True                            |
|   then if True then True else True|
|   else                            |
|      if False                     |
|         then False                |
|         else False                |
```

John Hughes [1995], Simon Peyton Jones [1997], Phil Wadler [2003], and Pablo Azero and Doaitse Swierstra [1998] have all developed pretty printing libraries for the functional language Haskell [Peyton Jones 2003]. Such a library implements the functionality common to a large class of pretty printers. For example, Wadler's library provides the following functions:

```
text   :: String -> Doc
line   :: Doc
(<>)   :: Doc -> Doc -> Doc
nest   :: Int -> Doc -> Doc
group  :: Doc -> Doc
pretty :: Int -> Doc -> String
```

The function `text` converts a string to an atomic document, the document `line` denotes a (potential) line break, and `<>` concatenates two documents. The function `nest` increases the indentation for all line breaks within its document argument. The function `group` marks the document as a unit to be formatted either *horizontally*, that is on a single line by converting each line break (and corresponding indentation) into a single space, or *vertically*, with all line breaks unchanged. Finally, the function `pretty` yields a string with a pretty layout. The function aims at minimising the number of lines by formatting groups horizontally while not exceeding the given line-width limit. The layout of a subdocument depends not only on its form but also on its context, the remaining document. The library functions enable easy *compositional* construction of a document

To pretty print a concrete data structure, we only have to define a function that transforms the data structure into a document. With the following function we obtain the pretty printed expression shown on the previous page:

```
data Exp = ETrue | EFalse | If Exp Exp Exp

toDoc :: Exp -> Doc
toDoc ETrue = text "True"
toDoc EFalse = text "False"
toDoc (If e1 e2 e3) =
  group (nest 3 (
    group (nest 3 (text "if" <> line <> toDoc e1)) <> line <>
    group (nest 3 (text "then" <> line <> toDoc e2)) <> line <>
    group (nest 3 (text "else" <> line <> toDoc e3))))
```

All previous implementations of Haskell pretty printing libraries use backtracking to determine the optimal layout. They limit backtracking to achieve reasonable efficiency, but their time complexity is worse than linear in the size of the input. However, more than 20 years ago Dereck Oppen [1980] published an imperative implementation of a pretty printer with linear time complexity. At the heart of his implementation lies an array that is updated in a complex pattern. Wadler tried to translate this implementation into a functional language but did not succeed [Wadler 2003]. Are destructive updates necessary to achieve efficiency? No, but the proof is not straightforward. We develop here, step by step, guided by Oppen's implementation, a similar but purely functional implementation in Haskell.

We implement Wadler's pretty printing interface. The interfaces of Hughes' and Peyton Jones' libraries are different, but changing the implementation to support them seems possible. Because Azero's and Swierstra's library is more expressive than the others, the implementation cannot support its interface.

2. THE PROBLEM

We follow Wadler in considering a document of type `Doc` as equivalent to a set of strings with the same content but different layout. These are the strings that we obtain for all the possible choices of formating the groups occurring in the construction of the document. Every group can be formatted independently either horizontally or vertically, except that all groups within a horizontal group have to be horizontal as well. The function `pretty` chooses one member of the set of strings.

To specify the output of `pretty`, we define a document as an algebraic data type with a constructor for each function that yields a document:

```
data Doc = Text String | Line | Doc :<> Doc | Group Doc

text  = Text
line  = Line
(<>)  = (:<>)
group = Group
```

For simplicity we ignore the function `nest` for the moment. We will see in Section 8 how it can easily be added to the final implementation.

We use a "document interpreter" `inter` to define the function `pretty`. While recursively traversing the document in-order, the function `inter` keeps track of two state variables: the boolean `h` states if the interpreter is *within a horizontal group*; the integer `r` contains the size of the *remaining space* on the current line. Besides the formatted output, the interpreter also has to return the remaining space on the last line.

For formatting a group the interpreter takes Oppen's approach: a group is formatted horizontally if and only if it fits on the remaining space of the line. In Section 7 this choice is discussed in more detail. A function `fits` checks if the remaining space is sufficient.[1]

```
pretty w d = fst (inter d False w)
  where
  inter :: Doc -> Bool -> Int -> (String,Int)
  inter (Text s)    h r = (s,r - length s)
  inter Line        h r = if h then (" ",r-1) else ("\n",w)
  inter (d1 :<> d2) h r = (o1 ++ o2,r2)
    where
    (o1,r1) = inter d1 h r
    (o2,r2) = inter d2 h r1
  inter (Group d)   h r = inter d (fits d r) r
```

A naïve implementation of `fits` evaluates the width of the document `d` and compares the result with the remaining space `r`.

```
fits :: Doc -> Int -> Bool
fits d r = widthDoc d <= r

widthDoc :: Doc -> Int
widthDoc (Text s)    = length s
widthDoc Line        = 1
widthDoc (d1 :<> d2) = widthDoc d1 + widthDoc d2
widthDoc (Group d)   = widthDoc d
```

[1]Because a group within a horizontal group is horizontal, we could replace the boolean expression `fits d r` by `h || fits d r` in the last equation of `inter`. This optimisation, however, does not improve the time complexity of this implementation nor any other one presented later.

The naïve implementation has the disadvantage that the additional traversals of the groups to determine their widths causes the function `pretty` to require exponential time for formatting some documents with nested groups.

There is another problem: Only after the full traversal of a group it is known whether the group fits on the remaining line. Hence the interpreter produces most of the output string for a group only after it has traversed the whole group, as the following computation of the Haskell interpreter Hugs[2] demonstrates:

```
Main> pretty 4 (group (text "Hi" <> line <> text "you" <> undefined))
"Hi
Program error: undefined
```

So the document for a whole group has to be kept in memory. A group is often large; it may encompass the whole document. Furthermore, the document itself can often be constructed lazily and hence should never exist as a whole in memory. In interactive applications the time delay at the beginning of a group may be disturbing. A pretty printer should only require a look-ahead of at most w characters, were w is the line-width limit. Wadler [2003] calls an implementation with this property *bounded*.

2.1 Bounded but not Linear

We can define `fits` so that it traverses the document `d` at most up to the remaining width `r`. When that point is reached, it is clear that the document does not fit.

```
fits :: Doc -> Int -> Bool
fits d r = isJust (remaining d r)
  where
  remaining :: Doc -> Int -> Maybe Int
  remaining (Text s)    r = r `natMinus` length s
  remaining Line        r = r `natMinus` 1
  remaining (d1 :<> d2) r = case remaining d1 r of
                              Just r1 -> remaining d2 r1
                              Nothing -> Nothing
  remaining (Group d)   r = remaining d r

natMinus :: Int -> Int -> Maybe Int
natMinus n1 n2 = if n1 >=n2 then Just (n1-n2) else Nothing
```

This implementation *with pruning* is bounded:

```
Main> pretty 4 (group (text "Hi" <> line <> text "you" <> undefined))
"Hi\nyou
Program error: undefined
```

The pruning method is similar to Wadler's method of pruning backtracking and hence we obtain the same time complexity: In the worst case it is $O(n \cdot w)$, where n is the size of the input and w the line-width limit. An example for the worst case is the following right-nested document:

[2]http://www.haskell.org/hugs

```
group (text "*" <> line <> group (text "*" <> line <> group ( ... )))
```

Pruning substantially improves the time complexity, but we want to obtain $O(n)$ time complexity, independent of w. However, no further optimisation is in sight. The optimisation leads into a cul-de-sac.

2.2 Linear but not Bounded

On the other hand, we can obtain a linear implementation from the naïve definition by tupling the document traversals and creating a circular program [Bird 1984]: instead of a separate function that traverses a document to determine its width, the interpreter `inter` can determine the width in addition to its other tasks. The new version of `inter` returns the formatted document, the size of the space remaining on the last line, and the width of the document:

```
pretty w d = (\(x,_,_)->x) (inter d False w)
  where
  inter :: Doc -> Bool -> Int -> (String,Int,Int)
  inter (Text s)    h r = (s,r-l,l)
    where
    l = length s
  inter Line        h r = (o,r',1)
    where
    (o,r') = if h then (" ",r-1) else ("\n",w)
  inter (d1 :<> d2) h r = (o1 ++ o2,r2,w1 + w2)
    where
    (o1,r1,w1) = inter d1 h r
    (o2,r2,w2) = inter d2 h r1
  inter (Group d)   h r = (o,r',w)
    where
    (o,r',w) = inter d (w <= r) r
```

This implementation of `inter` takes advantage of lazy evaluation: in the last equation the result width `w` is passed as part of the second argument. For this circular definition to work, the function `inter` has to be able to yield the width of a document without using the value of `h`. Therefore it would be wrong to “simplify” the equation for `Line` to

```
  inter Line h r = if h then (" ",r-1,1) else ("\n",w,1)
```

The implementation has linear time complexity[3], because a computation spends only constant time on each document constructor. However, the implementation is not bounded.

2.3 Recapitulation

This section has set the scene. We specified the meaning of the pretty printing functions and stated the two desired properties of boundedness and linear time

[3] The use of `(++)` for formatting a document `d1 :<> d2` actually leads to quadratic time complexity. To achieve linear time we can represent a document as a function and then function composition performs list concatenation [Hughes 1986]. We do not apply this optimisation here to not to distract from the main issues.

complexity. There are implementations that are bounded and others that are linear. The challenge is to marry the two properties in a single implementation.

3. A LINEAR IMPLEMENTATION WITH STACKS

The tree structured recursion of `inter` and pruning at a certain width to achieve boundedness do not fit together. Hence we follow Oppen and represent a document not as a tree structure but as a token sequence.

```
data Tokens = Empty
            | Text String Tokens
            | Line Tokens
            | Open Tokens
            | Close Tokens
```

A group is represented as an `Open` token, the sequence of the grouped document and a final `Close` token. To construct the token sequence in linear time we represent a document as a function and function composition performs concatenation [Hughes 1986].

```
newtype Doc = Doc (Tokens -> Tokens)

text s = Doc (Text s)
line = Doc (Line)
Doc l1 <> Doc l2 = Doc (l1 . l2)
group (Doc l) = Doc (Open . l . Close)

doc2Tokens :: Doc -> Tokens
doc2Tokens (Doc f) = f Empty
```

Similar to our previous implementations we define `pretty` through an interpreter `inter` of the token sequence. Because this interpreter iterates along the token list instead of recursively following the nesting structure of groups, it has to store information about surrounding groups explicitly in stacks. We will see that these explicit data structures are the key to obtaining the desired implementation.

We use the following abstract data type of sequences with stack operations:

```
newtype Seq a = S [a]
empty = S []
isEmpty (S qs) = null qs
cons x (S xs) = S (x:xs)
head (S xs) = List.head xs
tail (S xs) = S (List.tail xs)
```

Using an abstract data type avoids premature commitment to the concrete representation. If we used lists directly we would also be tempted to use pattern matching in place of the functions `isEmpty`, `head` and `tail`, which would make later changes of representation even harder. Okasaki [2000] gives a compelling example that demonstrates how easily premature commitment to a representation can make functional programmers blind to a natural implementation solution.

```
pretty :: Int -> Doc -> String
pretty w doc = fst (inter (doc2Tokens doc) 0 w empty)
  where
  inter :: Tokens -> Int -> Int -> Seq Int -> (String,Seq Bool)
  inter Empty       _ _ _  = ("",empty)
  inter (Text s ts) p r es = (s ++ o,hs)
    where
    (o,hs) = inter ts (p+l) (r-l) es
    l = length s
  inter (Line ts)   p r es = (if h then ' ':o else '\n':o,hs)
    where
    (o,hs) = inter ts (p+1) (if h then r-1 else w) es
    h = not (isEmpty hs) && head hs
  inter (Open ts)   p r es = (o,tail hs)
    where
    (o,hs) = inter ts p r (cons (p+r) es)
  inter (Close ts)  p r es = (o,cons (p <= head es) hs)
    where
    (o,hs) = inter ts p r (tail es)
```

Fig. 1. Implementation using Stacks

For our first implementation of the token interpreter we do not yet care about boundedness. So the interpreter decides if a group is formatted horizontally or vertically only after it has been traversed, that is, the decision is made at the `Close` token of the group. An implementation that makes the decision by determining the width of a group proves to be rather complicated. Instead it is easier to follow Oppen again and introduce an absolute measure of a token's position. The *absolute position* `p` gives the column in which a token would start, if the *whole* document that is passed to `pretty` was formatted in a single line. The interpreter keeps track of the absolute position of the current token and the remaining space, as used already in the last section. At an `Open` token the interpreter adds absolute position and remaining space to determine the *maximal end position* of the group. If the maximal end position is larger or equal than the absolute position of the `Close` token of the group, then the group fits on the line and hence is formatted horizontally. Otherwise it is formatted vertically.

To get the maximal end position of a group from its `Open` token past possibly many inner groups to its `Close` token, the interpreter passes a stack of maximal end positions along the token sequence. Inversely, the interpreter has to get the information whether a group is horizontal from its `Close` token back to all its `Line` tokens, because these need to be formatted accordingly. For this purpose the interpreter passes a stack of boolean values, called the *horizontal stack*, back along the token sequence.

Figure 1 shows the implementation. As the type of `inter` indicates, it passes the arguments `p` (absolute position), `r` (remaining space) and `es` (stack of maximal end positions) along the token sequence, whereas it returns `o` (output) and `hs` (horizontal stack) in the opposite direction. At an `Open` token an element is pushed on the stack of maximal end positions and the top element from the horizontal stack is popped. At a `Close` token the top element of the stack of maximal end positions is popped and the decision on formatting is pushed on the horizontal stack. Each element of a stack corresponds to a surrounding group.

The subsequent table shows the values of the main interpreter variables for an example token list. We assume that the strings of the `Text` tokens have length 1. The line-width limit is 3. Sequences are enclosed in $\langle\rangle$ and boolean values are abbreviated as T and F.

		`Open`		`Text`		`Line`		`Open`		`Text`		`Line`		`Text`		`Close`		`Close`	
`p`	0	$\rightarrow$	0	$\rightarrow$	1	$\rightarrow$	2	$\rightarrow$	2	$\rightarrow$	3	$\rightarrow$	4	$\rightarrow$	5	$\rightarrow$	5	$\rightarrow$	5
`r`	3	$\rightarrow$	3	$\rightarrow$	2	$\rightarrow$	3	$\rightarrow$	3	$\rightarrow$	2	$\rightarrow$	1	$\rightarrow$	0	$\rightarrow$	0	$\rightarrow$	0
`es`	$\langle\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle 5,3\rangle$	$\rightarrow$	$\langle 5,3\rangle$	$\rightarrow$	$\langle 5,3\rangle$	$\rightarrow$	$\langle 5,3\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle\rangle$
`hs`	$\langle\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle T,F\rangle$	$\leftarrow$	$\langle T,F\rangle$	$\leftarrow$	$\langle T,F\rangle$	$\leftarrow$	$\langle T,F\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle\rangle$

The arrows indicate the direction in which values are passed along the token sequence. The table does not show the complex data dependencies between values. Similar to the last interpreter in the previous section this one is circular: in the interpreter equation for the token `Line`, information from the returned horizontal stack is passed as part of the third argument, the remaining space. For this to work the decision if a group is horizontal may not depend on the remaining space at its inner tokens. Indeed, the decision only depends on the remaining space at its `Open` token.

Laziness leads to a kind of co-routine computation. The computation of each passed value, that is, absolute position, remaining space, maximal end position stack, horizontal stack and output, can be identified with a process. At one point in time each process may be at a different position in the token sequence. The computation of the horizontal stack is furthest along the sequence, "looking" for `Close` tokens. The computation of the output is the most backward process, directly or indirectly using the results of all other processes.

For interpreting a token the implementation only requires time linear in the size of a token (the size of `Text` s is the length of s). Hence with n as the size of the input, the implementation has $O(n)$ time complexity.

4. EARLIER DECISIONS THROUGH DEQUEUES

The linear implementation with stacks is not bounded, because the interpreter decides whether a group is formatted horizontally only at the `Close` token of the group. However, in the example table of the previous section the interpreter could have noticed already at the second `Line` token that the outer group does not fit, because the absolute position 4 after the token is larger than the maximal end position of the group, the 3 stored at the rear of the sequence `es` before the token. The interpreter could have removed the maximal end position from the *rear* of the sequence and could have added a `False` to the *rear* of the corresponding horizontal sequence `hs`.

The following table shows the variable values for a modified implementation. Underlined values differ from those of the previous table.

		`Open`		`Text`		`Line`		`Open`		`Text`		`Line`		`Text`		`Close`		`Close`	
`p`	0	$\rightarrow$	0	$\rightarrow$	1	$\rightarrow$	2	$\rightarrow$	2	$\rightarrow$	3	$\rightarrow$	$\mathbf{4}$	$\rightarrow$	5	$\rightarrow$	5	$\rightarrow$	5
`r`	3	$\rightarrow$	3	$\rightarrow$	2	$\rightarrow$	3	$\rightarrow$	3	$\rightarrow$	2	$\rightarrow$	1	$\rightarrow$	0	$\rightarrow$	0	$\rightarrow$	0
`es`	$\langle\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle 3\rangle$	$\rightarrow$	$\langle 5,3\rangle$	$\rightarrow$	$\langle 5,\mathbf{3}\rangle$	$\rightarrow$	$\underline{\langle 5\rangle}$	$\rightarrow$	$\underline{\langle 5\rangle}$	$\rightarrow$	$\underline{\langle\rangle}$	$\rightarrow$	$\langle\rangle$
`hs`	$\langle\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle F\rangle$	$\leftarrow$	$\langle T,F\rangle$	$\leftarrow$	$\langle T,F\rangle$	$\leftarrow$	$\underline{\langle T\rangle}$	$\leftarrow$	$\underline{\langle T\rangle}$	$\leftarrow$	$\underline{\langle\rangle}$	$\leftarrow$	$\langle\rangle$

The modified interpreter removes elements from the rear of `es` and adds elements to the rear of `hs` as well as still adding and removing elements from the front. So the sequences `es` and `hs` are no longer used as stacks but as double ended queues (dequeues). For the moment we extend our sequence type accordingly in an inefficient but straightforward way:

```
snoc x (S xs) = S (xs++[x])
last (S xs) = List.last xs
init (S xs) = S (List.init xs)
```

The general idea of earlier formatting decisions is as follows: every time the interpreter increases the absolute position, it checks if the maximal end position for the outermost surrounding group is smaller than the absolute position. If it is smaller, than that group has to be formatted vertically. So the interpreter removes the maximal end position from the rear of `es` and adds the boolean `False` to the rear of `hs`. The maximal end position is removed from the rear of `es` for two reasons. First, the new maximal end position at the rear of the sequence has to be compared with the current absolute position: the interpreter should be able to decide for several nested groups that they are vertical. Second, removal of the maximal end position ensures that the sequence is empty when the `Close` token of the group is reached and thus the interpreter notices at the `Close` token that the decision on the formatting of the group has been taken already.

As in the previous implementation, every element of a sequence `es` or `hs` contains information about a surrounding group. The front element corresponds to the directly surrounding group, the next element to the next surrounding group, etc. Only, now there may no longer be an element for every surrounding group. If there is no maximal end position for a group in `es`, then the maximal end position is smaller than the absolute position of the current token. If there is no “horizontal” boolean for a group in `hs`, then that group is formatted vertically.

Figure 2 shows the new implementation with `es` and `hs` as dequeues. The function `prune` implements the new early check. The definition is recursive, because at a given absolute position several surrounding groups may be found to require vertical formatting. The definition of `inter` differs from the previous one in that the cases for `Text` and Line tokens call the function `prune`, because these are the tokens that cause an increase of the absolute position. Because of earlier decisions, the maximal end positions sequence can already be empty at a `Close` token. In that case both sequences are unchanged. Otherwise a boolean is put in front of the horizontal sequence as before.[4]

This implementation is not linear, because of the inefficient new dequeue functions. The recursive definition of the function `prune` does not endanger linear runtime, because there is at most one recursive call of `prune` per `Open` token in the token list.

[4]We could replace the test `p <= head es` by `True`. The correctness of this minor optimisation relies, however, on `es` being ordered, with the largest element at the rear. When in Section 8 the function `nest` is added to the implementation, this ordering property will no longer hold.

```
pretty :: Int -> Doc -> String
pretty w doc = fst (inter (doc2Tokens doc) 0 w empty)
  where
  inter :: Tokens -> Int -> Int -> Seq Int -> (String,Seq Bool)
  inter Empty       _ _ _  = ("",empty)
  inter (Text s ts) p r es = (s ++ o,hs)
    where
    (es',hs) = prune p' es hs'
    (o,hs') = inter ts p' (r-l) es'
    l = length s
    p' = p+l
  inter (Line ts)   p r es = (if h then ' ':o else '\n':o,hs)
    where
    (es',hs) = prune p' es hs'
    (o,hs') = inter ts p' (if h then r-1 else w) es'
    h = not (isEmpty hs') && head hs'
    p' = p+1
  inter (Open ts)   p r es = (o,tail hs')
    where
    (o,hs') = inter ts p r (cons (p+r) es)
  inter (Close ts)  p r es = (o,hs)
    where
    (es',hs) = if isEmpty es then (es,hs')
                             else (tail es,cons (p <= head es) hs')
    (o,hs') = inter ts p r es'

prune :: Int -> Seq Int -> Seq Bool -> (Seq Int,Seq Bool)
prune p es hs' = if isEmpty es || p <= last es then (es,hs')
                                                else (es',snoc False hs)
  where
  (es',hs) = prune p (init es) hs'
```

Fig. 2. Implementation using Dequeues

From a given Open token the new interpreter requires only a look-ahead of at most the width limit[5] along the token sequence to decide if the group is formatted horizontally or vertically. However, despite the earlier decisions the implementation is still unbounded.

5. LIMITING LOOK-AHEAD THROUGH LAZY DEQUEUES

Why is the implementation with earlier decisions still unbounded? It is the fault of the functions applied to the horizontal sequence hs. On the one hand, when the need for vertical formatting is detected early, a False value is added to the *rear* of hs. On the other hand, at every Line token of the group, the *front* value of hs is used to decide the formatting. So the False "travels" from the rear to the front of the sequence. Evaluation of the front element of hs forces nearly full evaluation of hs.

[5]The look-ahead is also determined by the length of Text strings. Calling prune after every character of a Text string would remove this dependency, but increase the runtime considerably. In practice, Text strings should be shorter than the width limit; otherwise no "pretty" formatting is possible anyway.

```
pretty :: Int -> Doc -> String
pretty w doc = fst (inter (doc2Tokens doc) 0 w empty)
  where
  inter :: Tokens -> Int -> Int -> Seq Int -> (String,Seq Bool)
  inter Empty       _ _ _  = ("",empty)
  inter (Text s ts) p r es = (s ++ o,hs)
    where
    (es',hs) = prune p' es hs'
    (o,hs') = inter ts p' (r-l) es'
    l = length s
    p' = p+l
  inter (Line ts)   p r es = (if h then ' ':o else '\n':o,hs)
    where
    (es',hs) = prune p' es hs'
    (o,hs') = inter ts p' (if h then r-1 else w) es'
    h = not (isEmpty es') && head hs'
    p' = p+1
  inter (Open ts)   p r es = (o,hs)
    where
    (es',hs) = consTail (p+r) es hs'
    (o,hs') = inter ts p r es'
  inter (Close ts)  p r es = (o,hs)
    where
    (es',hs) = if isEmpty es then (es,hs') else tailCons es (p <= head es) hs'
    (o,hs') = inter ts p r es'

prune :: Int -> Seq Int -> Seq Bool -> (Seq Int,Seq Bool)
prune p es1 hs3 = if isEmpty es1 || p <= last es1 then (es1,hs3) else (es3,hs1)
  where
  (es2,hs1) = initSnoc es1 False hs2
  (es3,hs2) = prune p es2 hs3
```

Fig. 3. Implementation using Combined Dequeue Functions

The following equations demonstrate the problem:

```
head (tail (snoc False (undefined))) = undefined
head (tail (snoc False (S [undefined]))) = False
head (tail (snoc False (S [True]))) = False
```

To determine the front element, the values of other sequence elements (e.g. `True`) are not needed, but the structure of the sequence, in particular its length, is.

The solution of the problem lies in the close relationship between the maximal end position sequence and the horizontal sequence. In the implementation of Figure 2 the two sequences are modified in perfect synchrony. At every program point where a function yielding a new sequence is applied to one sequence, the inverse function is applied to the other sequence. Hence we can combine the operations on two sequences. Let

```
consTail :: a -> Seq a -> Seq b -> (Seq a,Seq b)
```

add an element to the front of the first sequence and take the tail of the second one;

```
tailCons :: Seq a -> b -> Seq b -> (Seq a,Seq b)
```

take the tail of the first sequence and add an element to the front of the second sequence;

```
initSnoc :: Seq a -> b -> Seq b -> (Seq a,Seq b)
```

take the initial part of the first sequence and add an element to the rear of the second sequence. The implementation in Figure 3 differs from that of Figure 2 only insofar that it uses the new combined functions on sequences and uses `isEmpty es'` instead of `isEmpty hs'`.

The sequence `es` is passed forwards along the token sequence and the sequence `hs` is passed backwards along the token sequence. Both sequences are empty at the beginning and at the end of the token sequence. Together with the perfect synchrony of modifying the two sequences follows that for any token of the token sequence both `es` and `hs` have the same length. Hence the interpreter can use the internal structure of `es`, which may be fully evaluated, to apply a function to `hs` without evaluating any part of `hs`.

We define an auxiliary function:

```
copyListStructure :: [a] -> [b] -> [b]
copyListStructure [] _ = []
copyListStructure (_:xs) zs = y: copyListStructure xs ys
  where
  (y:ys) = zs
```

This function takes two lists that should be of the same length. The function makes a copy of the second list, using the structure of the first list. Thus the list structure of the result can be fully evaluated, without evaluating the second list at all. For example:

```
copyListStructure [1,2,3] undefined = [undefined,undefined,undefined]
```

Using `copyListStructure` we define new *lazy* combined dequeue functions. Each combined function copies the structure of the first sequence (`es`) to the second sequence (`hs`) and only then applies the actual function to the copy.

```
withFirstStructure :: Seq a -> (Seq b -> Seq b) -> Seq b
                   -> (Seq a,Seq b)
withFirstStructure sq1@(S xs1) f (S xs2) =
  (sq1,f (S (copyListStructure xs1 xs2)))

consTail x sq1 sq2 = withFirstStructure (cons x sq1) tail sq2
tailCons sq1 x sq2 = withFirstStructure (tail sq1) (cons x) sq2
initSnoc sq1 x sq2 = withFirstStructure (init sq1) (snoc x) sq2
```

With these lazy dequeue functions the implementation of Figure 3 is bounded.

6. EFFICIENT LAZY DEQUEUES

In making the pretty printer bounded we have lost again the time efficiency of the first token list implementation. The functions `snoc` and `last`, and in particular `copyListStructure` take time linear in the length of the sequence argument. To

```
data Seq a = S !Int [a] !Int [a]

empty :: Seq a
empty = S 0 [] 0 []

isEmpty :: Seq a -> Bool
isEmpty (S lenf _ lenr _) = (lenf + lenr == 0)

head :: Seq a -> a
head (S lenf f lenr r) = List.head (if lenf==0 then r else f)

last :: Seq a -> a
last (S lenf f lenr r) = List.head (if lenr==0 then f else r)

cons :: a -> Seq a -> Seq a
cons x (S lenf f lenr r) = check (lenf+1) (x:f) lenr r

tail :: Seq a -> Seq a
tail (S _ [] _ _) = empty
tail (S lenf f lenr r) = reverse (check lenr r (lenf-1) (List.tail f))

snoc :: a -> Seq a -> Seq a
snoc x (S lenf f lenr r) = reverse (check (lenr+1) (x:r) lenf f)

init :: Seq a -> Seq a
init (S _ _ _ []) = empty
init (S lenf f lenr r) = check lenf f (lenr-1) (List.tail r)

reverse :: Seq a -> Seq a
reverse (S lenf f lenr r) = S lenr r lenf f

-- Keep lists in balance: rebalance if front list too long
check :: Int -> [a] -> Int -> [a] -> Seq a
check lenf f lenr r = if lenf <= 3 * lenr + 1
                        then S lenf f lenr r
                        else S lenf' f' lenr' r'
  where
  len = lenf + lenr
  lenf' = len `div` 2
  lenr' = len - lenf'
  (f',rf') = splitAt lenf' f
  r' = r ++ List.reverse rf'
```

Fig. 4. The Banker's Implementation of Lazy Dequeues

regain for `pretty` a runtime that is only linear in the size of the input, we need dequeue functions that take only constant amortised time.

Figure 6 gives the banker's dequeue implementation of the sequence data type. A banker's dequeue is represented by two lists and their respective lengths. The first list holds the front elements and the other list holds the rear elements of the sequence in reverse order. An invariant requires that the lengths of the lists are not too far apart. A function `check` moves some elements from one list to the other, when addition or removal of an element threatens to invalidate the invariant. This rebalancing takes time linear in the length of the sequence, but rebalancing happens so seldomly that the amortised time of every function is constant. Okasaki gives more detailed explanations in his book [Okasaki 1998][6].

So we use the banker's dequeue for implementing the abstract sequence data type. However, we come across the same problem that we observed in the last section for the naïve sequence implementation: the standard functions on banker's dequeues are too strict in the horizontal sequence `hs`. We cannot directly apply our previous solution: if the functions `consTail`, `tailCons` and `initSnoc` always copied the structure of the first to the second banker's dequeue, then we would not achieve our desired time complexity. However, the banker's dequeue implementation of `tail` demands only the first constructor of the first list representing the dequeue, the implementations of `cons` and `snoc` do not demand any part of the two lists, *except* when the two lists have to be rebalanced. Therefore the idea is that the linear time function `copyListStructure` is used only for rebalancing. Rebalancing takes linear time anyway.

The lazy Banker's dequeue implementation is given in Appendix C. This implementation cannot reuse the functions that operate on a single dequeue as the naïve sequence implementation in Section 5 does: First, `copyListStructure` has to be used by the internal rebalancing function `check`. Second, the function `check` has to operate on both dequeues simultaneously. Whenever it rebalances the end position sequence, it also rebalances the horizontal sequence in exactly the opposite way using the structure of the end position sequence. Without this combined rebalancing corresponding dequeues `es` and `hs` would not have the same internal structure! There is more than one way to represent a sequence as a Banker's dequeue.[7]

The implementation of the function `pretty` of Figure 3 together with the sequence implementation of Appendix C give a bounded, linear time pretty printer.

7. OVERFULL LINES

We took Oppen's approach for formatting a group: a group is formatted horizontally if and only if it fits in the remaining space of the line. Unfortunately this approach may yield layouts with lines wider than the width limit, even when a fitting layout exists. A group that still fits on a line may be followed by further text without a separating `line`. Because there is no `line`, the text has to be added to the current line, even if it does not fit. Breaking the group might have avoided the problem.

The problem is solved by normalising the token list with respect to the following (confluent) rewriting rules before applying `pretty`:

```
Close (Text s ts) ⇒  Text s (Close ts)
Open (Text s ts)  ⇒  Text s (Open ts)
Open (Close ts)   ⇒  ts
```

[6]The Haskell implementation in the book contains several mistakes. Also the implementation of Figure 6 uses `reverse`, taking advantage of symmetry, to simplify the implementation of `check`.
[7]For example, for most dequeues d the dequeue `tail (cons 42` d`)` has a different representation from d. It is essential for the constant amortised runtime of all functions that there is more than one way to represent a sequence.

The normalised token list has the property that between a `Close` token and the next `Text` token there is always a `Line` token.[8] In other words, the end of every group can be chosen to be the end of the line. Hence the aforementioned problem can no longer occur. With normalisation the pretty printer always produces a fitting layout if it exists, the pretty printer is *optimal* in Wadler's sense.

Rewriting only moves `Text` tokens in and out of groups. Therefore the set of `line`s "belonging" to each group, which are either all formatted as new lines or all as spaces, is unchanged. So normalisation leaves the set of strings denoted by a document unchanged. Only the representation of the document is changed, so that Oppen's formating criterium (possibly) selects a different set element as output.

In Appendix B normalisation is implemented by a linear traversal of the token list, which collects `Open` and `Close` tokens until the next `Line` token is reached.

8. INDENTATION

To complete the library we still have to implement the function `nest`. There are different interpretations of the expression `nest` n. In Wadler's library it increases the current left margin by n columns whereas in Oppen's pretty printer (and other libraries) it sets the left margin to the current column position plus n. In Appendices A and B both variants are implemented. There are two new tokens:

```
data Tokens =  ...
             | OpenNest (Int -> Int -> Int) Tokens | CloseNest Tokens
```

The function of the first token takes the current margin and column to determine a new margin and the second token resets the margin to its previous value. An extended version of `inter` keeps a list of left margins and interprets the new tokens.

Alternatively, we could implement Wadler's variant as he does by a transformation which moves the indentation information into each `Line` token.

9. COMPARISON WITH WADLER'S IMPLEMENTATION

We followed Wadler in considering a document as equivalent to a set of strings. Like Hughes Wadler defines an ordering relation on lines: if two lines fit within the maximal width the longer one is better, otherwise the shorter one is better. The ordering on single lines is extended lexically to strings. Wadler's pretty printer outputs the string that is best with respect to the ordering.

Here we followed Oppen in formatting a group horizontal if and only if it fits on the current line. Because this specification refers directly to the construction of the document from combinators, it cannot be expressed in terms of the semantics of a document, the set of strings. However, we observed in Section 7 that token normalisation ensures that every group is followed by a potential end of line. Hence for a normalised token sequence Oppen's formatting criterium of filling a line with groups as far as possible without violating the width limit yields the output that is best with respect to Wadler's order. So the lazy dequeue pretty printer and Wadler's pretty printer yield the same output for any fully defined document. They

[8]Oppen [1980] states in his Section 2 that he assumes this property to hold for all input token sequences and hints at the end of Section 5 that other token sequences can be changed.

are, however, not semantically equivalent. Although Wadler's implementation is bounded, it is still more strict, that is, it often requires more look-ahead.[9]

For example, with Wadler's library the Haskell interpreter Hugs computes

```
Main> pretty 8 (group (text "Hi" <> line) <>
                group (text "you" <> line) <> undefined)
"
Program error: undefined
```

whereas with the lazy dequeue library it computes

```
Main> pretty 8 (group (text "Hi" <> line) <>
                group (text "you" <> line) <> undefined)
"Hi you
Program error: undefined
```

10. IN PRACTICE

Appendices A, B and C give the full implementation of the lazy dequeue pretty printing library including token normalisation and two sorts of indentation.

Table I compares this library with the bounded but not linear implementation of Section 2.1, Wadler's library [Wadler 2003] and the one by Simon Peyton Jones based on John Hughes' [Peyton Jones 1997]. Note that the bounded implementation provides only basic pretty printing. In particular, it does not avoid overfull lines as discussed in Section 7 like all other libraries do. The table gives for different line-widths the time in seconds that is needed to format a sequence of 500 right-nested documents. The nested document (cf. Section 2.1) is generated by

```
doc n = if n==0 then text "" else group (text "*" <> line <> doc (n-1))
```

The test program for HPJ is

```
main = print . length .
         renderStyle (style{lineLength=width,ribbonsPerLine=1.0}) .
         vcat . take 500 . repeat $ doc 200
```

and for the other libraries it is the equivalent

```
main = print . length . pretty width .
         foldr1 (\x y -> x <> line <> y) . take 500 . repeat $ doc 200
```

All programs were compiled with GHC[10] version 6.2 with `-O2` and run on a 900 MHz sparcv9 under Solaris.

The example clearly demonstrates that the runtimes of all implementations except our own depends on the line-width.

In practice the runtimes for line-widths of around 80 characters are of most interest. Because there is no "typical" document and the interface of HPJ is different, systematic comparisons are hard. In our experience the libraries provide similar

[9]Early on Wadler's implementation applies a flattening operation to the content of a group and the document following it. Thus information about the structure of the remaining document is lost. In contrast, the lazy dequeue implementation uses this information.
[10]The Glasgow Haskell compiler. http://www.haskell.org/ghc

Line-width	20	40	60	80	100	120	140
HPJ	2.9	5.0	6.9	8.4	9.8	10.4	11.0
Wadler	1.1	1.9	2.5	3.2	3.8	4.1	4.4
Bounded	0.8	1.3	1.6	2.0	2.4	2.7	3.1
Dequeue	1.5	1.5	1.7	1.6	1.7	1.7	1.7

Table I. Runtimes in seconds for pretty printing a document with different line-width limits

runtimes, but for any two libraries there is a document where one library is up to 3-4 times faster than the other. Whereas the runtimes of HPJ vary most with changes of the document structure, those of Dequeue are the most stable.

11. FINAL REMARKS

We have developed a purely functional bounded pretty printer with the same time complexity as Oppen's imperative implementation. This proves that Oppen's algorithm can also be implemented purely functionally. We have seen that for a bounded pretty printer a dequeue is the natural choice for passing the maximal end position of a group forward along the token sequence to the point where it can be decided if the group is formatted horizontally or vertically. The key problem lies in passing the result of the decision backwards along the token sequence without jeopardising boundedness or the desired time complexity. For this purpose Oppen uses a mutable array; we use a second, synchronous, lazy dequeue.

Oppen's implementation consists of two parts which work together in a co-routine like fashion. So an explicitly concurrent version of the pretty printer seems natural. However, it would require additional explicit synchronisation to ensure that no process looked too far ahead, using unnecessary space for communication buffers. In the lazy dequeue implementation demand-driven evaluation ensures that the "process" computing the horizontal sequence never "goes" further ahead than the "output process" requires for formatting the next token.

The lazy dequeue implementation demonstrates two further points in algorithm design: First, defining a function recursively along the structure of the main data type (here a tree-structured document) may not lead to the best solution. We sometimes have to leave the limits of an implicit recursive control structure by making it explicit as data structure. A data structure can be replaced by a more flexible one (here a stack by a dequeue).[11] Second, there are useful lazy variants of non-inductively defined abstract data structures such as dequeues.

A version of the pretty printing library with an extended interface is part of the distribution of the Haskell compiler nhc98[12] and has been used in various programs.

The lazy dequeue pretty printer is more complex than the non-linear implementation by Wadler and those given in Section 2. Numerous alternative implementations have been explored by me and other people. The failure to find a more simple linear and bounded implementation still does not settle the question whether such an implementation exists.

[11]A related well-known example is breadth-first traversal of a tree: Depth-first traversal can be implemented easily by direct recursion. However, instead of recursion we can also use a stack. Replacing the stack by a queue we obtain breadth-first traversal of a tree [Okasaki 2000].
[12]http://www.haskell.org/nhc98

We developed the lazy dequeue pretty printer in a number of steps. Nonetheless we did not derive implementations from another by equational reasoning as Hughes and Wadler did. In fact, bounded and unbounded implementations have different strictness properties and hence are not semantically equivalent. We stepped several times from unbounded to bounded implementations and back. How can we prove the correctness of the development, so that it gives further insight into the problem and its solution?

Acknowledgements

I thank the referees and Colin Runciman for many suggestions for improving this paper. The work reported in this paper was partially supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant number GR/M81953.

REFERENCES

AZERO, P. AND SWIERSTRA, D. 1998. Optimal pretty-printing combinators. http://www.cs.uu.nl/groups/ST/Center/SoftwareDistributions.

BIRD, R. S. 1984. Using circular programs to eliminate multiple traversals of data. *Acta Informatica 21*, 239–250.

HUGHES, J. 1986. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters 22,* 3, 141—144.

HUGHES, J. 1995. The design of a pretty-printing library. In *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds. LNCS 925. Springer Verlag.

OKASAKI, C. 1998. *Purely Functional Data Structures.* Cambridge University Press.

OKASAKI, C. 2000. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming.* 131–136.

OPPEN, D. C. 1980. Prettyprinting. *ACM Transactions on Programming Languages and Systems 2,* 4, 465–483.

PEYTON JONES, S. L. 1997. A pretty printer library in Haskell. Part of the GHC distribution at http://www.haskell.org/ghc.

PEYTON JONES, S. L., Ed. 2003. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press.

WADLER, P. 2003. A prettier printer. In *The Fun of Programming.* Palgrave Macmillan, Chapter 11, 223–244.

A. THE LAZY DEQUEUE PRETTY PRINTING LIBRARY

```
module Pretty (Doc,pretty,text,line,(<>),group
              ,nestMargin,nestCol)
  where

import Prelude hiding (head,last,init,tail)
import qualified List (head,tail)
import Sequence
import Doc
```

```
pretty :: Int -> Doc -> String
pretty w doc = fst (inter (doc2Tokens doc) 0 w [0] empty)
  where
  inter :: Tokens -> Int -> Int -> [Int] -> Seq Int
        -> (String,Seq Bool)
  inter Empty          _ _ _  _ = ("",empty)
  inter (Text s ts)    p r ms es = (s ++ o,hs)
    where
    (es',hs) = prune p' es hs'
    (o,hs') = inter ts p' (r-l) ms es'
    l = length s
    p' = p+l
  inter (Line ts)      p r ms es =
    (if h then ' ' : o else '\n' : rep m ' ' o,hs)
    where
    (es',hs) = prune p' es hs'
    (o,hs') = inter ts p' (if h then r-1 else w-m) ms es'
    h = not (isEmpty es') && head hs'
    p' = p+1
    m = List.head ms
  inter (Open ts)       p r ms es = (o,hs)
    where
    (es',hs) = consTail (p+r) es hs'
    (o,hs') = inter ts p r ms es'
  inter (Close ts)      p r ms es = (o,hs)
    where
    (es',hs) =
      if isEmpty es then (es,hs') else tailCons es (p <= head es) hs'
    (o,hs') = inter ts p r ms es'
  inter (OpenNest f ts) p r ms es =
    inter ts p r ((f (List.head ms) (w-r)) : ms) es
  inter (CloseNest ts)  p r ms es =
    inter ts p r (List.tail ms) es

prune :: Int -> Seq Int -> Seq Bool -> (Seq Int,Seq Bool)
prune p es1 hs3 =
  if isEmpty es1 || p <= last es1 then (es1,hs3) else (es3,hs1)
  where
  (es2,hs1) = initSnoc es1 False hs2
  (es3,hs2) = prune p es2 hs3

-- variant of ‘replicate’: rep n x rs = replicate n x ++ rs
rep :: Int -> a -> [a] -> [a]
rep n x rs = if n <= 0 then rs else x : rep (n-1) x rs
```

B. IMPLEMENTATION OF THE DOCUMENT TYPE

```
module Doc (text,line,(<>),group,nestMargin,nestCol,doc2Tokens,Doc
           ,Tokens(Text,Line,Open,Close,OpenNest,CloseNest,Empty))
  where

text :: String -> Doc
text s = Doc (Text s)

line :: Doc
line = Doc (Line)

(<>) :: Doc -> Doc -> Doc
Doc l1 <> Doc l2 = Doc (l1 . l2)

group :: Doc -> Doc
group (Doc l) = Doc (Open . l . Close)

-- increment current left margin
nestMargin :: Int -> Doc -> Doc
nestMargin i (Doc l) =
  Doc (OpenNest (flip (const (+i))) . l . CloseNest)

-- set left margin to current column plus given increment
nestCol :: Int -> Doc -> Doc
nestCol i (Doc l) = Doc (OpenNest (const (+i)) . l . CloseNest)

doc2Tokens :: Doc -> Tokens
doc2Tokens (Doc f) = normalise (f Empty)

newtype Doc = Doc (Tokens -> Tokens)

data Tokens = Empty | Text String Tokens | Line Tokens
            | Open Tokens | Close Tokens
            | OpenNest (Int -> Int -> Int) Tokens
            | CloseNest Tokens

normalise :: Tokens -> Tokens
normalise = go id
  where
  go :: (Tokens -> Tokens) -> Tokens -> Tokens
  go co Empty = co Empty  -- no opening brackets
  go co (Open ts) = go (co . open) ts
  go co (Close ts) = go (co . Close) ts
  go co (Line ts) = co . Line . go id $ ts
  go co (Text s ts) = Text s (go co ts)
  go co (OpenNest f ts) = OpenNest f (go co ts)
  go co (CloseNest ts) = CloseNest (go co ts)

  open (Close ts) = ts
  open ts = Open ts
```

C. IMPLEMENTATION OF EFFICIENT LAZY DEQUEUES

```
module Sequence(Seq,empty,isEmpty,consTail,tailCons,initSnoc
               ,last,head) where

import Prelude hiding (head,last,init,tail,reverse)
import qualified List (head,tail,reverse)

copyListStructure :: [a] -> [b] -> [b]
copyListStructure [] _ = []
copyListStructure (_:xs) zs = y: copyListStructure xs ys
  where
  (y:ys) = zs

data Seq a = S !Int [a] !Int [a]

empty :: Seq a
empty = S 0 [] 0 []

isEmpty :: Seq a -> Bool
isEmpty (S lenf _ lenr _) = (lenf + lenr == 0)

head (S lenf f lenr r) = List.head (if lenf==0 then r else f)
last (S lenf f lenr r) = List.head (if lenr==0 then f else r)

consTail :: a -> Seq a -> Seq b -> (Seq a,Seq b)
consTail x (S lenf f lenr r) sq =
  (sq',S lenf (List.tail f') lenr r')
  where
  (sq',f',r') = check (lenf+1) (x:f) lenr r sq
  -- precondition: sq and sq' have same structure

tailCons :: Seq a -> b -> Seq b -> (Seq a,Seq b)
tailCons (S _ [] _ _) x _ = (empty,S 0 [] 1 [x])
tailCons (S lenf f lenr r) x sq =
  (reverse sq',S lenf (x:f') lenr r')
  where
  (sq',r',f') = check lenr r (lenf-1) (List.tail f) (reverse sq)
  -- precondition: sq and sq' have same structure

initSnoc :: Seq a -> b -> Seq b -> (Seq a,Seq b)
initSnoc (S _ _ _ []) x _ = (empty,S 1 [x] 0 [])
initSnoc (S lenf f lenr r) x sq =
  (sq',S lenf f' lenr (x:r'))
  where
  (sq',f',r') = check lenf f (lenr-1) (List.tail r) sq
  -- precondition: sq and sq' have same structure
```

```
reverse :: Seq a -> Seq a
reverse (S lenf f lenr r) = S lenr r lenf f

-- Keep lists in balance: rebalance if front list too long
check :: Int -> [a] -> Int -> [a] -> Seq b -> (Seq a,[b],[b])
check lenf f lenr r sq = if lenf <= 3 * lenr + 1
                           then (S lenf f lenr r,f2,r2)
                           else (S lenf' f' lenr' r',f2',r2')
  where
  S _ f2 _ r2 = sq
  len = lenf + lenr
  lenf' = len `div` 2
  lenr' = len - lenf'
  (f',rf') = splitAt lenf' f
  r' = r ++ List.reverse rf'
  lf2 = copyListStructure f' f2
  lr2 = copyListStructure r' r2
  (r2',rf2') = splitAt lenr lr2
  f2' = lf2 ++ List.reverse rf2'
```