FUNCTIONAL PEARL

Pretty Printing with Lazy Dequeues

Olaf Chitil

University of York, UK

Abstract

There are several Haskell 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 without backtracking. We show 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.

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 [2], Simon Peyton Jones [3], Phil Wadler [7], and Pablo Azero and Doaitse Swierstra [1] have all developed pretty printing libraries for Haskell. A pretty printing library implements the functionality common to a large class of pretty printers. The layout of a subtree does not only depend on its form but also on its context, the remaining tree. A pretty printing library provides functions for *compositionally* defining a transformation of a tree data structure to an abstract document. Finally, such a library has one function to transform a document into the desired text. For example, Wadler's library [7] 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 printed on a single line by converting all its line breaks into single spaces, if this is possible without exceeding the line-width limit. Hence a document describes a set of texts with the same content but different layouts. Finally, the function `pretty` yields the text with the minimal number of lines that does not exceed the given line-width limit.

To obtain the pretty printed expression shown at the beginning of the paper, we only have to compositionally define a function that transforms an abstract expression into a document:

```
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 not just linear in the size of the input. However, back in 1980 Dereck Oppen published an imperative algorithm for a pretty printer with linear time complexity [6]. At the heart of his algorithm lies an array that is updated in a complex pattern. Are destructive updates necessary to achieve efficiency? No, but the proof is not straightforward. We develop here, starting with a simple but inefficient implementation, step by step, guided by Oppen's algorithm, a similar but purely functional implementation in Haskell.

We implement Wadler's pretty printing interface. The interfaces of Hughes' and Peyton Jones' libraries are more complex, but extending our implementation to support them seems straightforward. [1] Supporting the interface of

[1] Hughes' and Peyton Jones' implementations do not always yield the optimal layout with respect to Hughes' semantics. The layouts that the implementations can produce seem

```
pretty width d = fst (inter False width d)
  where
  inter :: Bool -- in fitting group
        -> Int  -- remaining space on current line
        -> Doc  -- input document
        -> (String   -- formatted output
           ,Int)     -- remaining space on last line
  inter f r (Group d)   = inter (f || fits d r) r d
  inter f r (Text z)    = (z,r - length s)
  inter f r (d1 :<> d2) = (s1 ++ s2,r2)
    where
    (s1,r1) = inter f r  d1
    (s2,r2) = inter f r1 d2
  inter True  r Line    = (" ",r-1)
  inter False r Line    = ("\n",width)
```

Fig. 1. Recursive Implementation using `fits`

Azero's and Swierstra's library is impossible, because it is considerably more expressive than the others.

2 Recursive Implementations

We can define a document simply 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 6 how it can easily be added to the final implementation [2].

We define the function `pretty` as an interpreter of documents that implements the whole functionality. The only slightly difficult case is the formatting of a group. We take Oppen's approach: a group is formatted in a single line if and only if it fits on the remaining space of the line. Previous Haskell libraries

to coincide with those of Wadler's library, except for a difference in nesting discussed in Section 6.

[2] Therefore it is fortunate that separate functions `nest` and `group` exist, although for most applications a single function combining the two is useful.

take a slightly different approach which we discuss in Section 5.

The implementation is given in Figure 1. The function `inter` has to pass some state information: first, if the interpreter is within a group that fits in the remaining space on the current line; second, the size of the remaining space on the current line.

For a `Group` document the interpreter has to determine if the group fits. It obviously fits if the group is within another group that fits. Otherwise, a function `fits` is used to determine if the document `d` fits within the remaining space `r`.

A naïve implementation of `fits` evaluates the width of the document `d` (with a `Line` equal to a single space) and compares the result with the remaining space `r`.

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

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

Unfortunately, the additional traversals of the sub-documents to determine their widths cause the function `pretty` to require exponential time for formatting some documents with nested groups.

The implementation is more efficient if `fits` traverses the document `d` at most up to the 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 (Group d) r = remaining d r
  remaining Line      r = r `natMinus` 1
  remaining (Text z)  r = r `natMinus` length z
  remaining (d1 :<> d2) r = do
    r1 <- remaining d1 r
    remaining d2 r1

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

This 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. This pruning method has, however, a major drawback. We want to obtain $O(n)$

```
pretty width d = fst3 (inter False width 0 d)
  where
  inter :: Bool -- in fitting group
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> Doc
        -> (String
           ,Int     -- remaining space on last line
           ,Int)    -- next absolute start position
  inter f r p (Group d) = (s,r',p')
    where
    (s,r',p') = inter (f || p'-p <= r) r p d
  inter f r p (Text z) = (z,r-l,p+l)
    where
    l = length z
  inter f r p (d1 :<> d2) = (s1 ++ s2,r2,p2)
    where
    (s1,r1,p1) = inter f r  p  d1
    (s2,r2,p2) = inter f r1 p1 d2
  inter f r p Line = (o,r',p+1)
    where
    (o,r') = if f then (" ",r-1) else ("\n",width)

fst3 :: (a,b,c) -> a
fst3 (x,_,_) = x
```

Fig. 2. Recursive Implementation that Returns the Next Start Position

time complexity, independent of w, but a further optimisation is not in sight. The optimisation leads into a cul-de-sac.

On the other hand, we can obtain a linear implementation[3] from the naïve definition by applying the tupling transformation: 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.

Because groups can be nested, it is not obvious how `inter` should be defined to return the width of its input document. The solution is to introduce an absolute measure of a document's position. The absolute position gives the column in which the document would start, if the *whole* document that is passed to `pretty` was formatted on a single line. The function `inter` receives the start position as argument and returns the next start position which is

[3] The use of `(++)` for formatting a document `d1 :<> d2` actually leads to quadratic time complexity. However, we can use the same optimisation as is used in the Haskell class `Show` to assemble the result string in linear time: `inter` has to return a value of type `String -> String` instead of just a `String` and list concatenation becomes function composition. We do not apply this optimisation here to not to distract from the main issues.

free after its input document. The difference between the two positions is the width of the input document.

Figure 3 shows the new implementation. It takes advantage of the lazy evaluation of the recursive call of `inter`: the result position `p'` is passed as part of the first argument. The implementation has linear time complexity, because a computation spends only constant time on each document constructor.

3 Iterative Implementations

Unfortunately our current implementation has a major drawback: only after the full traversal of a group it is known if the group fits on the remaining line. Hence a computation produces most of the output string for a group only after it has traversed the whole group. The time delay may not be a problem in practice, but the delayed computation of the output uses memory space linear in the size of the group. We would like our pretty printer to use only a small amount of space which is independent of the formatted document. Because the document will usually be constructed lazily or even be read sequentially from a file, pretty printing a document element should also only require a limited look-ahead into the remaining document.

The recursive implementation with pruning has the desired space behaviour, but it is not obvious how it can be married with the time efficient tuppled implementation. The problem is that pruning at a certain width and the tree structured recursion of `inter` do not fit together. Hence we move from tree structured recursion to sequential iteration. Following Oppen we represent the document as a list of tokens:

```
type Doc = [Token]

data Token = Text String | Line | Open | Close
```

A group is represented as the sequence of an `Open` token, the sequence of the grouped document and a final `Close` token. Translation from the old document data type to the new one is straightforward. Alternatively, an efficient direct construction can be defined in continuation-passing style. We assume in the following that documents are well-formed, that is, the `Open` and `Close` tokens are well-bracketed.

We redefine the tuppled implementation for the token sequence. Because we no longer use recursion that follows the structure of the document, we have to make the nesting structure of the groups explicit by using stacks. For every group the interpreter has to determine the next absolute start position. Hence it has to return a stack of positions — represented by a list. At the end of a group the interpreter needs to know if there is a surrounding fitting group. For this purpose we could pass a stack of booleans, but a natural number which states how deep the interpreter is in fitting groups is simpler. The interpreter no longer needs to return the size of the space that remains on the last line,

```
pretty width d = fst (inter 0 width 0 d)
  where
  inter :: Int  -- depth of fitting groups
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> [Token]
        -> (String
           ,[Int])  -- next absolute start positions
  inter f r p (Open:ts) = (s,es')
    where
    e':es' = es
    (s,es) = inter (if f>0 then succ f
                           else (if (e'-p)<=r then 1 else 0))
                r p ts
  inter f r p (Close:ts) = (s,p:es)
    where
    (s,es) = inter (pred f) r p ts
  inter f r p (Text z : ts) = (z ++ s,es)
    where
    (s,es) = inter f (r-l) (p+l) ts
    l = length z
  inter f r p (Line:ts) = (o:s,es)
    where
    (o,r') = if f>0 then (' ',r-1) else ('\n',width)
    (s,es) = inter f r' (p+1) ts
  inter f r p [] = ("",[])
```

Fig. 3. Iterative Implementation that Returns Next Start Positions

because the document constructor `(:<>)` has vanished. Figure 3 shows the implementation.

To see how the implementation works, consider a simple example. The following table shows the values of most variables for each iteration step. The document is given in the top row. We assume that the strings of the `Text` tokens have length 1. The line-width limit is 3. The arrows indicate in which directions values are passed. The inner group fits on a single line whereas the outer one does not.

	`Open`	`Text`	`Line`	`Open`	`Text`	`Line`	`Text`	`Close`	`Close`
`f`	$0 \rightarrow$	$0 \rightarrow$	$0 \rightarrow$	$0 \rightarrow$	$1 \rightarrow$	$1 \rightarrow$	$1 \rightarrow$	$1 \rightarrow$	$0 \rightarrow$ 0
`r`	$3 \rightarrow$	$3 \rightarrow$	$2 \rightarrow$	$3 \rightarrow$	$3 \rightarrow$	$2 \rightarrow$	$1 \rightarrow$	$0 \rightarrow$	$0 \rightarrow$ 0
`p`	$0 \rightarrow$	$0 \rightarrow$	$1 \rightarrow$	$2 \rightarrow$	$2 \rightarrow$	$3 \rightarrow$	$4 \rightarrow$	$5 \rightarrow$	$5 \rightarrow$ 5
`es`	$[] \leftarrow$	$[5] \leftarrow$	$[5] \leftarrow$	$[5] \leftarrow$	$[5,5] \leftarrow$	$[5,5] \leftarrow$	$[5,5] \leftarrow$	$[5] \leftarrow$	$[] \leftarrow$

Laziness leads to a kind of co-routine computation. The applications of `inter` to `Close` tokens can be identified with a *front* process and the other applications of `inter`, especially to `Open` tokens, as a *back* process. The front process determines the position of each `Close` token and passes this information in the variable `es` backwards to the back process. The back process determines the position of each `Open` token and the remaining space on the line. Together with the end position obtained from the front process it can determine if the group still fits. Despite this image, however, there is no simple implementation of the interpreter through two real processes, because the communication channel between them is a stack.

This iterative implementation has the same time and space behaviour as our last recursive implementation. [4] However, in the example we can already see that the outer group does not fit, when we reach the absolute position 4 (`p` = 4). The group does not fit, because it starts at position 0 and the line space remaining for it is 3 (the values of `p` and `r` at the first `Open` token). Unfortunately, the interpreter does not know these values 0 and 3 when it reaches position 4.

Therefore we introduce an additional argument that is passed from left to right: a stack which holds for each `Open` token of a group the sum of the absolute position and the space remaining on the line; we call this sum the group's maximal end position. We also simplify the resulting stack of end positions to a stack of booleans instead. At a `Close` token we just have to take the top position from the maximal end positions stack, compare it with the current position, and push the result on the stack of booleans which we return. Figure 4 shows the modified implementation and the following table shows the values of the new variables for our example document.

	`Open`	`Text`	`Line`	`Open`	`Text`	`Line`	`Text`	`Close`	`Close`
`f`	0 → 0	→ 0	→ 0	→ 0	→ 1	→ 1	→ 1	→ 1	→ 0 → 0
`r`	3 → 3	→ 2	→ 3	→ 3	→ 2	→ 1	→ 0	→ 0	→ 0
`p`	0 → 0	→ 1	→ 2	→ 2	→ 3	→ 4	→ 5	→ 5	→ 5
`ps`	[] → [3]	→ [3]	→ [3]	→ [5,3]	→ [5,3]	→ [5,3]	→ [5,3]	→ [3]	→ []
`fs`	[] ← [F]	← [F]	← [F]	← [T,F]	← [T,F]	← [T,F]	← [T,F]	← [F]	← []

Now for the optimisation. At position 4 the information for determining that the outer group does not fit is now available. The bottom of the stack `ps` contains the maximal end position of the outermost group. Because it is 3, smaller than the current position, the outermost group cannot fit. Hence

[4] In contrast to the recursive implementation the use of `(++)` does not lead to quadratic time complexity here, because the first argument of `(++)` is not constructed by a recursive call of `inter`. No optimisation of list concatenation is necessary. Compare for Footnote 3.

```
pretty :: Int -> Doc -> String
pretty width d = fst (inter 0 width 0 [] d)
  where
  inter :: Int  -- depth of fitting groups
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> [Int]  -- maximal end positions
        -> [Token]
        -> (String
           ,[Bool])  -- fitting infos
  inter f r p ps (Open:ts) = (s,fs')
    where
    f':fs' = fs
    (s,fs) = inter (if f>0 then succ f
                           else (if f' then 1 else 0))
                r p (r+p:ps) ts
  inter f r p ps (Close:ts) = (s,(p<=p'):fs)
    where
    p':ps' = ps
    (s,fs) = inter (pred f) r p ps' ts
  inter f r p ps (Text z : ts) = (z ++ s,fs)
    where
    (s,fs) = inter f (r-l) (p+l) ps ts
    l = length z
  inter f r p ps (Line:ts) = (o:s,fs)
    where
    (o,r') = if f>0 then (' ',r-1) else ('\n',width)
    (s,fs) = inter f r' (p+1) ps ts
  inter f r p [] [] = ("",[])
```

Fig. 4. Iterative Implementation that Returns Fitting Information in a Stack

at this point we can already remove the end position from the bottom of the "stack" `ps` and add `False` (F) to the bottom of the boolean "stack" `fs`:

	Open	Text	Line	Open	Text	Line	Text	Close	Close
f	0 →	0 →	0 →	0 →	1 →	1 →	1 →	0 →	0
r	3 →	3 →	2 →	3 →	3 →	2 →	1 →	0 →	0 → 0
p	0 →	0 →	1 →	2 →	2 →	3 →	4 →	5 →	5 → 5
ps	[] →	[3] →	[3] →	[3] →	[5,3] →	[5,3] →	$\underline{[5]}$ →	$\underline{[5]}$ →	$\underline{[]}$ → []
fs	[] ←	[F] ←	[F] ←	[F] ←	[T,F] ←	[T,F] ←	$\underline{[T]}$ ←	$\underline{[T]}$ ←	$\underline{[]}$ ← []

4 Lazy Dequeues

Obviously we no longer simply use `ps` and `fs` as stacks but as double ended queues. Fortunately we find in Okasaki's book [4] the Haskell implementation of the banker's dequeue. If used in a single threaded manner as here, each operation runs in $O(1)$ amortized time. Hence our optimised iterative implementation still has linear time complexity.

There is a problem left: the intention of our optimisation is to enable the interpreter to determine with a limited look-ahead if a group fits. By looking at the bottom of the end positions dequeue, the decision can be made with a look-ahead of at most the line-width limit. However, the interpreter adds this information to the *bottom* of the dequeue `fs` and removes it from the *top* of the dequeue `fs` at the `Open` token. To avoid further look-ahead these operations must work without fully evaluating the dequeue `fs`. That means in the example that the interpreter must be able to add and remove `False` (F) without ever "touching" the `True` (T).

We can add and remove elements from a *list* without "touching" the remaining list, but can we do the same for *dequeues*? Yes, within the special context of our pretty printer we can.

The two dequeues `ps` and `fs` are accessed in perfect synchrony: Each time an operation is performed on one dequeue, exactly the inverse operation is performed on the other dequeue. Hence we combine the operations on the two dequeues. So

```
cons :: a -> Q1 a -> Q2 b -> (Q1 a, b, Q2 b)
```

adds an element to the front of the first dequeue and splits the second dequeue into its front element and the tail dequeue;

```
rview :: Q1 a -> Q2 b -> b -> (Q1 a, a, Q2 b)
```

splits the first dequeue into an initial dequeue and a rear element and adds an element to the rear of the second dequeue;

```
lview :: Q1 a -> b -> Q2 b -> (a, Q1 a, Q2 b)
```

splits the first dequeue into its front element and its tail dequeue and adds an element to the front of the second dequeue.

The dequeue `ps` is passed from left to right and the dequeue `fs` is passed from right to left. Furthermore, both dequeues are empty at the beginning and at the end of interpreting the token sequence. Hence the internal structures of the two dequeues are the same at each interpretation step. So we can use the knowledge about the internal structure of `ps`, which may be fully evaluated, to apply an operation to `fs` without evaluating any part of `fs`.

Because we use the structure of `fs` to manipulate `ps`, the operations `cons` and `lview` are not identical up to swapping of arguments and result elements but have different strictness properties. We define separate types for the two dequeues to stress the asymmetry and enable a minor optimisation.

```
pretty width d = fst (inter 0 width 0 empty1 d)
  where
  inter :: Int  -- depth of fitting groups
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> Q1 Int  -- maximal end positions
        -> [Token]
        -> (String
           ,Q2 Bool)  -- fitting infos
  inter f r p ps (Open:ts) = (s,fs')
    where
    (ps',f',fs') = cons (r+p) ps fs
    (s,fs) = inter (if f>0 then succ f
                           else (if f' then 1 else 0))
                r p ps' ts
  inter f r p ps (Close:ts)
    | isEmpty1 ps = inter (pred f) r p ps ts
    | otherwise   = (s,fs')
    where
    (_,ps',fs') = lview ps True fs
    (s,fs) = inter (pred f) r p ps' ts
  inter f r p ps (Text z : ts) = (z ++ s,fs)
    where
    (s,fs) = prune f (r-l) (p+l) ps ts
    l = length z
  inter f r p ps (Line : ts) = (o : s,fs)
    where
    (o,r') = if f>0 then (' ',r-1) else ('\n',width)
    (s,fs) = prune f r' (p+1) ps ts
  inter _ _ _ _ [] = ("",empty2)

  prune :: Int -> Int -> Int -> Q1 Int -> [Token]
        -> (String,Q2 Bool)
  prune f r p ps ts
    | isEmpty1 ps || p <= p' = inter f r p ps ts
    | otherwise              = (s,fs')
    where
    (ps',p',fs') = rview ps fs False
    (s,fs) = prune f r p ps' ts
```

Fig. 5. Iterative Implementation with Lazy Dequeues

Without going into details of the implementation we note that a banker's dequeue is represented by two lists. One holds the top elements and the other the bottom elements of the dequeue. An invariant requires that the lengths of the lists are not too far apart. When addition or removal of an element threatens to invalidate the invariant, list elements are moved from one list to the other. The only operations applied to the two lists are `reverse`, `(++)` and `splitAt`. We can easily define lazy variants of these standard list functions which — given the length of a list argument or a result — construct the list structure of the result without demanding evaluation of any of its list arguments. Only demanding some list element of the result will lead to more demand of the arguments. Here, for example, is the lazy variant of `(++)`:

```
lappend :: Int -> [a] -> [a] -> [a]

lappend 0 _ zs = zs
lappend n xs zs = y : lappend (n-1) ys zs
  where
  y:ys = xs
```

Using the lazy variants to implement the dequeue operations `cons`, `rview` and `lview`, we obtain the required lazy dequeues. The full implementation is given in Appendix B.

With these dequeues we can finally define our time and space efficient pretty printer. Figure 5 shows the implementation. For each `Text` and `Line` token the function `prune` tests if some surrounding groups do not fit. Hence, if when reaching a `Close` token the maximal positions dequeue is non-empty, then the group certainly fits.

5 Overfull Lines

We took Oppen's approach to formatting a group: a group is formatted in a single line if and only if it fits on the remaining space of the line. Unfortunately this approach may yield layouts with lines wider than the width limit, although 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 does not fit. Breaking the group may have avoided the problem.

Our solution is to normalise the token list with respect to the following two rewriting rules before applying `pretty`:

```
Close, Text s  ⇒  Text s, Close
Open, Text s   ⇒  Text s, Open
```

The normalised token list has the property that between a `Close` token and the next `Text` token there is always a `Line` token. Hence the aforementioned problem can no longer occur. Like Wadler's pretty printer ours always produces a fitting layout if it exists. Note that rewriting only moves `Text` to-

kens in and out of groups. Hence the set of `line`s "belonging" to each group, which are either all formatted as new lines or all as spaces, is unchanged. So rewriting does not change the set of texts described by a document.

Normalisation can be implemented by a linear, straightforward traversal of the token list, keeping track of the number of currently opened and closed groups. Note that analogous normalisation of the tree structured documents, which we used in Section 2, is hard to implement efficiently.

6 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. We can easily implement either of these variants by introducing two new tokens

```
data Token =  ... | NestOpen Int | NestClose
```

and interpreting them appropriately in the function `inter` which also acquires a stack of current left margins as additional argument. Alternatively, we can implement Wadler's variant just as he does by a preceding transformation which moves the indentation information to every `Line` token.

7 Conclusions

We have developed a purely functional pretty printer that only requires time linear in the size of the input/output and space linear in the line-width limit. It demonstrates that we do not need updateable data structures to achieve the same efficiency as Oppen's imperative algorithm and also throws some light on this rather monolithic algorithm. Oppen's algorithm consists of two parts which also work together in a co-routine like fashion. For communication between the two processes an array is used as dequeue. The difference is that dequeue elements are updated where our implementation uses a second, synchronous, lazy dequeue.

We have obtained a useful library. An extended version is part of the distribution of the Haskell compiler nhc98 [5]. The compiler itself uses the library to provide pretty printing of the abstract syntax tree after any compiler phase.

On a general level the derivation of our pretty printing implementation demonstrates two points in algorithm design: First, defining a function recursively along the structure of the main data type may not lead to the best solution. We sometimes have to leave the limits of an implicit recursive control

[5] http://www.cs.york.ac.uk/fp/nhc98

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). [6] Second, there are useful lazy variants of non-inductively defined abstract data structures such as dequeues.

Acknowledgements

I thank Colin Runciman and the anonymous referees for many suggestions for improving this paper.

The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant number GR/M81953.

References

[1] Pablo Azero and Doaitse Swierstra. Optimal pretty-printing combinators. http://www.cs.uu.nl/groups/ST/Software/PP/, 1998.

[2] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925. Springer Verlag, 1995.

[3] Simon Peyton Jones. A pretty printer library in Haskell. Available from http://research.microsoft.com/Users/simonpj/downloads/pretty-printer/pretty.html, 1997.

[4] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

[5] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming*, pages 131–136, 2000.

[6] Dereck C Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[7] Philip Wadler. A prettier printer. Available from http://cm.bell-labs.com/cm/cs/who/wadler/topics/language-design.html, 1998.

[6] 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 [5].

A The Complete Pretty Printing Library

```
module Pretty (Doc,text,line,(<>),group,nestInc,nestSet,pretty)
  where

import LazyDequeue

-- Public interface:

-- precondition: string contains no formatting characters
-- \n \t etc.
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 indentation
-- the delta (i) may be any integer,
-- but runtime error if indentation becomes negative
nestInc :: Int -> Doc -> Doc
nestInc i (Doc l) =
  Doc ((NestIncOpen i :) . l . (NestIncClose :))

-- set indentation to current column plus given increment
nestSet :: Int -> Doc -> Doc
nestSet i (Doc l) =
  Doc ((NestSetOpen i :) . l . (NestSetClose :))

pretty :: Int -> Doc -> String
pretty width (Doc l) =
  fst (inter width [0] 0 width 0 empty1
        (normalise 0 0 (l [])))
```

```
-- Internal parts:

newtype Doc = Doc ([Token] -> [Token])
data Token = Text String
           | Line
           | Open
           | Close
           | NestIncOpen Int
           | NestIncClose
           | NestSetOpen Int
           | NestSetClose

-- normalise the stream of tokens with respect to the rules
--  Open, Close  ==>
--  Open, t      ==>  t, Open
--  Close, t     ==>  t, Close
-- for all tokens t except Line, Open and Close
normalise :: Int  -- number of deferred opening brackets
          -> Int  -- number of deferred closing brackets
          -> [Token]
          -> [Token]
normalise o c [] = replicate c Close
  -- there should be no deferred opening brackets
normalise o c (Open : ts) = normalise (o+1) c ts
normalise o c (Close : ts)
  | o == 0    = normalise o (c+1) ts
  | otherwise = normalise (o-1) c ts
normalise o c (t@(NestIncOpen _) : ts) = t : normalise o c ts
normalise o c (t@NestIncClose : ts) = t : normalise o c ts
normalise o c (t@(NestSetOpen _) : ts) = t : normalise o c ts
normalise o c (t@NestSetClose : ts) = t : normalise o c ts
normalise o c (t@(Text _) : ts) = t : normalise o c ts
normalise o c (t@Line : ts) =
  rep c Close . rep o Open . (t :) . normalise 0 0 $ ts

inter :: Int  -- width
      -> [Int] -- left margins (current on top)
      -> Int   -- depth of fitting groups
      -> Int   -- remaining space on current line
      -> Int   -- absolute start position
      -> Q1 Int  -- maximal end positions
      -> [Token]
      -> (String
         ,Q2 Bool)  -- fitting infos
```

```
inter _ _ _ _ _ _ [] = ("",empty2)
inter width ms f r p ps (Open:ts) = (s,fs')
  where
  (ps',f',fs') = cons (r+p) ps fs
  (s,fs) = inter width ms (if f>0 then succ f
                                  else (if f' then 1 else 0))
             r p ps' ts
inter width ms f r p ps (Close:ts)
  | isEmpty1 ps = inter width ms (pred f) r p ps ts
  | otherwise   = (s,fs')
  where
  (_,ps',fs') = lview ps True fs
  (s,fs) = inter width ms (pred f) r p ps' ts
inter width ms f r p ps (Text t : ts) = (t ++ s,fs)
  where
  (s,fs) = prune width ms f (r-l) (p+l) ps ts
  l = length t
inter width ms@(m:_) f r p ps (Line : ts) = (o s,fs)
  where
  (o,r') = if f>0 then ((' ':),r-1)
                  else (('\n':) . rep m ' ', width-m)
  (s,fs) = prune width ms f r' (p+1) ps ts
inter width ms@(m:_) f r p ps (NestIncOpen i : ts) =
  inter width (m+i : ms) f r p ps ts
inter width (_:ms) f r p ps (NestIncClose : ts) =
  inter width ms f r p ps ts
inter width ms@(m:_) f r p ps (NestSetOpen i : ts) =
  inter width (width-r+i : ms) f r p ps ts
inter width (_:ms) f r p ps (NestSetClose : ts) =
  inter width ms f r p ps ts

prune :: Int -> [Int] -> Int -> Int -> Int -> Q1 Int -> [Token]
      -> (String,Q2 Bool)
prune width ms f r p ps ts
  | isEmpty1 ps || p <= p' = inter width ms f r p ps ts
    -- note: to evaluate p' fs does not need to be evaluated
  | otherwise              = (s,fs')
  where
  (ps',p',fs') = rview ps fs False
  (s,fs) = prune width ms f r p ps' ts

-- continuation style variant of ‘replicate’
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 Lazy Dequeues

```
module LazyDequeue(Q1,Q2,empty1,empty2,isEmpty1
                  ,cons,rview,lview) where

-- q12List (Q1 _ f _ r) = f ++ reverse r
-- Q1 also contains the lengths of the two lists
-- Q2 does not contain lengths
data Q1 a = Q1 !Int [a] !Int [a]
data Q2 a = Q2 [a] [a]

reverse1 :: Q1 a -> Q1 a
reverse1 (Q1 lenf f lenr r) = Q1 lenr r lenf f

reverse2 :: Q2 a -> Q2 a
reverse2 (Q2 f r) = Q2 r f

empty1 = Q1 0 [] 0 []
empty2 = Q2 [] []

isEmpty1 (Q1 lenf _ lenr _) = lenf + lenr == 0

-- Keep lengths of the two lists in balance
check :: Int -> [a] -> Int -> [a] -> Q2 b -> (Q1 a, [b], [b])
check lenf f lenr r q2 =
  if lenf > balanceConstant * lenr + 1 then
    let
      len = lenf + lenr
      lenf' = len `div` 2
      lenr' = len - lenf'
      (f', rf') = splitAt lenf' f
      (r2, rf2) = lsplitAt lenr r2'
    in (Q1 lenf' f' lenr' (r ++ reverse rf')
       ,lappend lenf' f2' (lreverse (lenr'-lenr) rf2)
       ,r2)
   else
    (Q1 lenf f lenr r, f2', r2')
  where
  Q2 f2' r2' = q2
  balanceConstant = 3 :: Int
```

```
cons :: a -> Q1 a -> Q2 b -> (Q1 a, b, Q2 b)
cons x (Q1 lenf f lenr r) q2' = (q', head f2, Q2 (tail f2) r2)
  where
  (q', f2, r2) = check (lenf+1) (x:f) lenr r q2'

rview :: Q1 a -> Q2 b -> b -> (Q1 a, a, Q2 b)
rview (Q1 _ (x:_) _ []) q2' y = (empty1, x, Q2 [y] [])
rview (Q1 _ [] _ []) _ _ = error "empty dequeue"
rview (Q1 lenf f lenr (x:r)) q2' y = (q', x, Q2 f2 (y:r2))
  where
  (q', f2, r2) = check lenf f (lenr-1) r q2'

lview :: Q1 a -> b -> Q2 b -> (a, Q1 a, Q2 b)
lview q1 y q2 = (x, reverse1 q1', reverse2 q2')
  where
  (q1', x, q2') = rview (reverse1 q1) (reverse2 q2) y

-- The lazy variants of standard list functions:

-- The first argument gives the length
-- of the argument/result list.
lreverse :: Int -> [a] -> [a]
lreverse n xs = lreverseAcc n xs []
  where
  lreverseAcc 0 _ acc = acc
  lreverseAcc n xs acc = lreverseAcc (n-1) ys (y:acc)
    where
    y:ys = xs

-- The first argument gives the length of the second argument.
lappend :: Int -> [a] -> [a] -> [a]
lappend 0 _ zs = zs
lappend n xs zs = y : lappend (n-1) ys zs
  where
  y:ys = xs

-- The first argument gives the position at which the input list
-- shall be split. The list must be at least that long.
lsplitAt :: Int -> [a] -> ([a], [a])
lsplitAt 0 xs = ([],xs)
lsplitAt n ys = (x:xs',xs'')
  where
  x:xs = ys
  (xs',xs'') = lsplitAt (n-1) xs
```