A prettier printer

Philip Wadler

Joyce Kilmer and most computer scientists agree: there is no poem as lovely as a tree. In our love affair with the tree it is parsed, pattern matched, pruned — and printed. A pretty printer is a tool, often a library of routines, that aids in converting a tree into text. The text should occupy a minimal number of lines while retaining indentation that reflects the underlying tree. A good pretty printer must strike a balance between ease of use, flexibility of format, and optimality of output.

Over the years, Richard Bird and others have developed the algebra of programming to a fine art. John Hughes (1995) describes the evolution of a pretty printer library, where both the design and implementation have been honed by an appealing application of algebra. This library has become a standard package, widely used in the field. A variant of it was implemented for use in the Glasgow Haskell Compiler by Simon Peyton Jones (1997).

This chapter presents a new pretty printer library, which I believe is an improvement on the one designed by Hughes. The new library is based on a single way to concatenate documents, which is associative and has a left and right unit. This may seem an obvious design, but perhaps it is obvious only in retrospect. Hughes's library has two distinct ways to concatenate documents, horizontal and vertical, with horizontal composition possessing a right unit but no left unit, and vertical composition possessing neither unit. The new library is 30% shorter and runs 30% faster than Hughes's.

A widely used imperative pretty-printer is described by Derek Oppen (1980). Oppen's work appears to be the basis of the pretty-printing facilities in Caml. The pretty printer presented here uses an algorithm equivalent to Oppen's, but presented in a functional rather than an imperative style. Further comparison with Hughes's and Oppen's work appears in the conclusions.

A complete listing of the pretty printer library appears at the end of this chapter.

1 A simple pretty printer

To begin, we consider the simple case where each document has only one possible layout — that is, no attempt is made to compress structure onto a single line. There are six operators for this purpose.

```
(<>)        ::  Doc -> Doc -> Doc
nil         ::  Doc
text        ::  String -> Doc
line        ::  Doc
nest        ::  Int -> Doc -> Doc
layout      ::  Doc -> String
```

Here `<>` is the associative operation that concatenates two documents, which has the empty document `nil` as its left and right unit. The function `text` converts a string to the corresponding document, and the document `line` denotes a line break; we adopt the convention that the string passed to `text` does not contain newline characters, so that `line` is always used for this purpose. The function `nest` adds indentation to a document. Finally, the function `layout` converts a document to a string. (In practice, one might choose to make `(text "\n")` behave like `line`, where `"\n"` is the string consisting of a single newline.)

One simple implementation of simple documents is as strings, with `<>` as string concatenation, `nil` as the empty string, `text` as the identity function, `line` as the string consisting of a single newline, `nest i` as the function that adds `i` spaces after each newline (to increase indentation), and `layout` as the identity function. This simple implmentation is not especially efficient, since nesting examines every character of the nested document, nor does it generalize easily. We will consider a more algebraic implementation shortly.

Note that indentation is added only at an explicit newline, not at the beginning of a string. Readers familiar with Hughes's pretty printer may find this choice surprising, but it is precisely this choice that enables the use of one concatenation operator rather than two.

As an example, here is a simple tree data type, and functions to convert a tree to a document.

```
data  Tree               =  Node String [Tree]

showTree (Node s ts)     =  text s <> nest (length s) (showBracket ts)

showBracket []           =  nil
showBracket ts           =  text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees [t]            =  showTree t
showTrees (t:ts)         =  showTree t <> text "," <> line <> showTrees ts
```

This produces output in the following style.

```
aaa[bbbbb[ccc,
            dd]],
      eee,
      ffff[gg,
           hhh,
           ii]]
```

Alternatively, here is a variant of the above function.

```
showTree’ (Node s ts)    =  text s <> showBracket’ ts

showBracket’ []          =  nil
showBracket’ ts          =  text "[" <>
                            nest 2 (line <> showTrees’ ts) <>
                            line <> text "]")

showTrees’ [t]           =  showTree t
showTrees’ (t:ts)        =  showTree t <> text "," <> line <> showTrees ts
```

This now produces output in the following style.

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

It is easy to formulate variants to generate yet other styles.

Every document can be reduced to a normal form of text alternating with line breaks nested to a given indentation,

`text` s_0 `<> nest` i_1 `line <> text` s_1 `<>` $\cdots$ `<> nest` i_k `line <> text` s_k

where each s_j is a (possibly empty) string, and each i_j is a (possibly zero) natural number. For example, here is a document.

```
text "bbbbb" <> text "[" <>
nest 2 (
  line <> text "ccc" <> text "," <>
  line <> text "dd"
) <>
line <> text "]"
```

And here is its normal form.

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

Hence, the document prints as follows.

```
bbbbb[
  ccc,
  dd
]
```

The following laws are adequate to reduce a document to normal form, taken together with the fact that `<>` is associative with unit `nil`.

```
text (s ++ t)        =  text s <> text t
text ""              =  nil

nest (i+j) x         =  nest i (nest j x)
nest 0 x             =  x

nest i (x <> y)      =  nest i x <> nest i y
nest i nil           =  nil

nest i (text s)      =  text s
```

All but the last law come in pairs: each law on a binary operator is paired with a corresponding law for its unit. The first pair of laws state that `text` is a homomorphism from string concatenation to document concatenation. The next pair of laws state that `nest` is a homomorphism from addition to composition. The pair after that state that `nest` distributes through concatenation. The last law states that nesting is absorbed by text. In reducing a term to normal form, the first four laws are applied left to right, while the last three are applied right to left.

We can also give laws that relate a document to its layout.

```
layout (x <> y)      =  layout x ++ layout y
layout nil           =  ""
layout (text s)      =  s
layout (nest i line) =  '\n' : copy i ' '
```

The first two laws state that layout is a homomorphism from document concatenation to string concatenation. In this sense, `layout` is the inverse of `text`, which is precisely what the next law states. The final law states that the layout of a nested line is a newline followed by one space for each level of indentation.

A simple, but adequate, implementation can be derived directly from the algebra of documents. We represent a document as a concatenation of items, where each item is either a text or a line break indented a given amount.

```
data  Doc              =  Nil
                       |  String `Text` Doc
                       |  Int `Line` Doc
```

These constructors relate to the document operators as follows.

```
Nil                    =  nil
s `Text` x             =  text s <> x
i `Line` x             =  nest i line <> x
```

For example, the normal form above is represented as follows.

```
"bbbbb[" `Text` (
2 `Line` ("ccc," `Text` (
2 `Line` ("dd" `Text` (
0 `Line` ("]" `Text` Nil)))))
```

Note that the document need not be in normal form, because it is not required that text and line breaks alternate.

It is easy to derive representations for each function from the above equations.

```
nil                    =  Nil
text s                 =  s `Text` Nil
line                   =  0 `Line` Nil

(s `Text` x) <> y      =  s `Text` (x <> y)
(i `Line` x) <> y      =  i `Line` (x <> y)
Nil <> y               =  y

nest i (s `Text` x)    =  s `Text` nest i x
nest i (j `Line` x)    =  (i+j) `Line` nest i x
nest i Nil             =  Nil

layout (s `Text` x)    =  s ++ layout x
layout (i `Line` x)    =  '\n' : copy i ' ' ++ layout x
layout Nil             =  ""
```

For instance, here is the derivation of the first line of concatenation.

```
    (s `Text` x) <> y
=       { definition Text }
    (text s <> x) <> y
=       { associative <> }
    text s <> (x <> y)
=       { definition Text }
    s `Text` (x <> y)
```

The remaining derivations are equally trivial.

2 A pretty printer with alternative layouts

We now consider documents with multiple possible layouts. Whereas before we might view a document as equivalent to a string, now we will view it as equivalent to a set of strings, each corresponding to a different layout of the same document.

This extension is achieved by adding a single function.

```
group          ::  Doc -> Doc
```

Given a document, representing a set of layouts, `group` returns the set with one new element added, representing the layout in which everything is compressed on one line. This is achieved by replacing each newline (and the corresponding indentation) with text consisting of a single space. (Variants might be considered where each newline carries with it the alternate text it should be replaced by. For instance, some newlines might be replaced by the empty text, others with a single space.)

The function `layout` is replaced by one that chooses the prettiest among a set of layouts. It takes as an additional parameter the preferred maximum line width of the chosen layout.

```
pretty         ::  Int -> Doc -> String
```

(Variants might be considered with additional parameters, for instance a 'ribbon width' indicating the maximum number of non-indentation characters that should appear on a line.)

As an example, here is a revision of the first form of the function to convert a tree to a document, which differs by the addition of a call to `group`.

```
showTree (Node s ts)  =  group (text s <> nest (length s) (showBracket ts))
```

If the previous document is printed with `pretty 30`, this definition produces the following output.

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

This fits trees onto one line where possible, but introduces sufficient line breaks to keep the total width less than 30 characters.

To formalize the semantics of the new operations, we add two auxiliary operators.

```
(<|>)          ::  Doc -> Doc -> Doc
flatten        ::  Doc -> Doc
```

The `<|>` operator forms the union of the two sets of layouts. The `flatten` operator replaces each line break (and its associated indentation) by a single space. A document always represents a non-empty set of layouts, where all layouts in the set flatten to the same layout. As an invariant, we require in `(x <|> y)` that all layouts in `x` and `y` flatten to the same layout. We do not expose `<|>` or `flatten` directly to the user; instead, they are exposed only via `group` (defined below) or `fillwords` and `fill` (defined in Section 4), which all preserve the invariant required by `<|>`.

Laws extend each operator on simple documents pointwise through union.

```
(x <|> y) <> z       =  (x <> z) <|> (y <> z)
x <> (y <|> z)       =  (x <> y) <|> (x <> z)
nest i (x <|> y)     =  nest i x <|> nest i y
```

Since flattening gives the same result for each element of a set, the distribution law for `flatten` is a bit simpler.

```
flatten (x <|> y)    =  flatten x
```

Further laws explain how `flatten` interacts with other document constructors, the most interesting case being what happens with `line`.

```
flatten (x <> y)     =  flatten x <> flatten y
flatten nil          =  nil
flatten (text s)     =  text s
flatten line         =  text " "
flatten (nest i x)   =  flatten x
```

Now we can define `group` in terms of `flatten` and `<|>`.

```
group x              =  flatten x <|> x
```

These laws are adequate to reduce any document to a normal form

$$x_1 \texttt{ <|> } \cdots \texttt{ <|> } x_n \,,$$

where each x_j is in the normal form for a simple document.

Next, we need to specify how to choose the best layout among all those in a set. Following Hughes, we do so by specifying an ordering relation between lines, and extending this lexically to an ordering between documents.

The ordering relation depends on the available width. If both lines are shorter than the available width, the longer one is better. If one line fits in the available width and the other does not, the one that fits is better. If both lines are longer than the available width, the shorter one is better. Note that this ordering relation means that we may sometimes pick a layout where some line exceeds the given width, but we will do so only if this is unavoidable. (This is a key difference from Hughes, as we discuss later.)

One possible implementation is to consider sets of layouts, where sets are represented by lists, and layouts are represented by strings or by the algebraic representation of the preceding section. This implementation is hopelessly inefficient: a hundred choices will produce 2^{100} possible documents.

Fortunately, the algebraic specification above leads straightforwardly to a more tractable implementation. The new representation is similar to the old, except we add a construct representing the union of two documents.

```
data  Doc             =  Nil
                      |  String ‘Text‘ Doc
                      |  Int ‘Line‘ Doc
                      |  Doc ‘Union‘ Doc
```

These constructors relate to the document operators as follows.

```
Nil                   =  nil
s ‘Text‘ x            =  text s <> x
i ‘Line‘ x            =  nest i line <> x
x ‘Union‘ y           =  x <|> y
```

We now have two invariants on `(x ‘Union‘ y)`. As before, we require that `x` and `y` flatten to the same layout. Additionally, we require that no first line of a document in `x` is shorter than some first line of a document in `y`; or, equivalently, that every first line in `x` is at least as long as every first line in `y`. We must insure these invariants wherever we creat a `Union`.

To achieve acceptable performance, we will exploit the distributive law, and use the representation `(s ‘Text‘ (x ‘Union‘ y))` in preference to the equivalent `((s ‘Text‘ x) ‘Union‘ (s ‘Text‘ y))`. For instance, consider the document

```
group (
  group (
    group (
      group (text "hello" <> line <> text "a")
    <> line <> text "b")
  <> line <> text "c")
<> line <> text "d")
```

This has the following possible layouts:

```
      hello a b c     hello a b       hello a         hello
                      c               b               a
                                      c               b
                                                      c
```

To lay this out within a maximum width of 5, we must pick the last of these – and we would like to eliminate the others in one fell swoop. To achieve this, we pick a representation that brings to the front any common string. For instance, we represent the above document in the form

```
      "hello" ‘Text‘ ((" " ‘Text‘ x) ‘Union‘ (0 ‘Line‘ y))
```

for suitable documents x and y. Here `"hello"` has been factored out of all the layouts in `x` and `y`, and `" "` has been factored out of all the layouts in `x`. Since `"hello"` followed by `" "` occupies 6 characters and the line width is 5, we may immediately choose the right operand of `‘Union‘` without further examination of `x`, as desired.

The definitions of `nil`, `text`, `line`, `<>`, and `nest` remain exactly as before, save that `<>` and `nest` must be extended to specify how they interact with `Union`.

```
(x `Union` y) <> z    =  (x <> z) `Union` (y <> z)
nest k (x `Union` y)  =  nest k x `Union` nest k y
```

These lines follow immediately from the distributive laws. In the first equation, note that if no first line of `x` is shorter than any first line of `y`, then no first line of `x <> z` is shorter than any first line of `y <> z`, so we preserve the invariant required by `Union`, and similarly for the second equation.

Definitions of `group` and `flatten` are easily derived.

```
group Nil              =  Nil
group (i `Line` x)     =  (" " `Text` flatten x) `Union` (i `Line` x)
group (s `Text` x)     =  s `Text` group x
group (x `Union` y)    =  group x `Union` y

flatten Nil            =  Nil
flatten (i `Line` x)   =  " " `Text` flatten x
flatten (s `Text` x)   =  s `Text` flatten x
flatten (x `Union` y)  =  flatten x
```

For instance, here is the derivation of the second line of `group`.

```
  group (i `Line` x)
=     { definition Line }
  group (nest i line <> x)
=     { definition group }
  flatten (nest i line <> x) <|> (nest i line <> x)
=     { definition flatten }
  (text " " <> flatten x) <|> (nest i line <> x)
=     { definition Text, Union, Line }
  (" " `Text` flatten x) `Union` (i `Line` x)
```

In the last line, each document on the left begins with a space while each document on the right begins with a newline, so we preserve the invariant required by `Union`.

The derivation of the third line of `group` reveals a key point.

```
  group (s `Text` x)
=     { definition Text }
  group (text s <> x)
=     { definition group }
  flatten (text s <> x) <|> (text s <> x)
=     { definition flatten }
  (text s <> flatten x) <|> (text s <> x)
=     { <> distributes through <|> }
  text s <> (flatten x <|> x)
=     { definition group }
```

```
    text s <> group x
=       { definition Text }
    s ‘Text‘ group x
```

Distribution is used to bring together the two instances of `text` generated by the definition of `group`. As we saw above, this factoring is crucial in efficiently choosing a representation.

The other lines of `group` and `flatten` are also easily derived. The last line of each follows from the invariant that the two operands of a union both flatten to the same document.

Next, it is necessary to choose the best among the set of possible layouts. This is done with a function `best`, which takes a document that may contain unions, and returns a document containing no unions. A moment's thought reveals that this operation requires two additional parameters: one specifies the available width `w`, and the second specifies the number of characters `k` already placed on the current line (including indentation). The code is fairly straightforward.

```
best w k Nil            =  Nil
best w k (i ‘Line‘ x)   =  i ‘Line‘ best w i x
best w k (s ‘Text‘ x)   =  s ‘Text‘ best w (k + length s) x
best w k (x ‘Union‘ y)  =  better w k (best w k x) (best w k y)

better w k x y          =  if fits (w-k) x then x else y
```

The two middle cases adjust the current position: for a newline it is set to the indentation, and for text it is incremented by the string length. For a union, the better of the best of the two options is selected. It is essential for efficiency that the inner computation of `best` is performed lazily. By the invariant for unions, no first line of the left operand may be shorter than any first line of the right operand. Hence, by the criterion given previously, the first operand is preferred if it fits, and the second operand otherwise.

It is left to determine whether a document's first line fits into `w` spaces. This is also straightforward.

```
fits w x | w < 0        =  False
fits w Nil              =  True
fits w (s ‘Text‘ x)     =  fits (w - length s) x
fits w (i ‘Line‘ x)     =  True
```

If the available width is less than zero, then the document cannot fit. Otherwise, if the document is empty or begins with a newline then it fits trivially, while if the document begins with text then it fits if the remaining document fits in the remaining space. Handling negative widths is not merely esoteric, as the case for text may yield a negative width. No case is required for unions, since the function is only applied to the best layout of a set.

Finally, to pretty print a document one selects the best layout and converts it to a string.

```
pretty w x              =  layout (best w 0 x)
```

The code for `layout` is unchanged from before.

3 Improving efficiency

The above implementation is tolerably efficient, but we can do better. It is reasonable to expect that pretty printing a document should be achievable in time $O(s)$, where s is the size of the document (a count of the number of `<>`, `nil`, `text`, `nest`, and `group` operations plus the length of all string arguments to `text`). Further, the space should be proportional to $O(w \max d)$ where w is the width available for printing, and d is the depth of the document (the depth of calls to `nest` or `group`).

There are two sources of inefficiency. First, concatenation of documents might pile up to the left.

$$(\cdots((\texttt{text}\ s_0\ \texttt{<>}\ \texttt{text}\ s_1)\ \texttt{<>}\ \cdots)\ \texttt{<>}\ \texttt{text}\ s_n$$

Assuming each string has length one, this may require time $O(n^2)$ to process, though we might hope it would take time $O(n)$. Second, even when concatenation associates to the right, nesting of documents adds a layer of processing to increment the indentation of the inner document.

$$\texttt{nest}\ i_0\ (\texttt{text}\ s_0\ \texttt{<>}\ \texttt{nest}\ i_1\ (\texttt{text}\ s_1\ \texttt{<>}\ \cdots\ \texttt{<>}\ \texttt{nest}\ i_n\ (\texttt{text}\ s_n)\cdots))$$

Again assuming each string has length one, this may require time $O(n^2)$ to process, though we might hope it would take time $O(n)$.

A possible fix for the first problem is to add an explicit representation for concatenation, and to generalize each operation to act on a list of concatenated documents. A possible fix for the second problem is to add an explicit representation for nesting, and maintain a current indentation which is incremented as nesting operators are processed. Combining these two fixes suggests generalizing each operation to work on a list of indentation-document pairs.

To implement this fix, we introduce a new representation for documents, with one constructor corresponding to each operator that builds a document. The representation is changed so that there is one constructor corresponding to each operator that builds a document. We use names in all caps to distinguish from the previous representation.

```
data  DOC      =  NIL
               |  DOC :<> DOC
               |  NEST Int DOC
               |  TEXT String
               |  LINE
               |  DOC :<|> DOC
```

The operators to build a document are defined trivially.

```
nil          =  NIL
x <> y       =  x :<> y
nest i x     =  NEST i x
text s       =  TEXT s
line         =  LINE
```

Again, as an invariant, we require in `(x :<|> y)` that all layouts in `x` and `y` flatten to the same layout, and that no first line in `x` is shorter than any first line in `y`.

Definitions of `group` and `flatten` are straightforward.

```
group x              =  flatten x :<|> x

flatten NIL          =  NIL
flatten (x :<> y)    =  flatten x :<> flatten y
flatten (NEST i x)   =  flatten x
flatten (TEXT s)     =  TEXT s
flatten LINE         =  TEXT " "
flatten (x :<|> y)   =  flatten x
```

These follow immediately from the equations given previously.

The representation function maps a list of indentation-document pairs into the corresponding document.

```
rep z  =  fold (<>) nil [ nest i x | (i,x) <- z ]
```

The operation to find the best layout of a document is generalized to act on a list of indentation-document pairs. The generalized operation is defined by composing the old operation with the representation function.

```
be w k z  =  best w k (rep z)                          (hypothesis)
```

The new definition is easily derived from the old.

```
best w k x              =  be w k [(0,x)]

be w k []               =  Nil
be w k ((i,NIL):z)      =  be w k z
be w k ((i,x :<> y):z)  =  be w k ((i,x):(i,y):z)
be w k ((i,NEST j x):z) =  be w k ((i+j,x):z)
be w k ((i,TEXT s):z)   =  s `Text` be w (k + length s) z
be w k ((i,LINE):z)     =  i `Line` be w i z
be w k ((i,x :<|> y):z) =  better w k (be w k ((i,x):z))
                                      (be w k ((i,y):z))
```

Here is the derivation of the first line.

```
   best w k x
=     { 0 is unit for nest }
   best w k (nest 0 x)
```

```
  =     { nil is unit for <> }
     best w k (nest 0 x <> nil)
  =     { definition rep, hypothesis }
     be w k [(0,x)]
```

Here is the case for `:<>`.

```
    be w k ((i,x :<> y):z)
  =     { hypothesis, definition rep, definition :<> }
    best w k (nest i (x <> y) <> rep z)
  =     { nest distributes over <> }
    best w k ((nest i x <> nest i y) <> rep z)
  =     { <> is associative }
    best w k (nest i x <> (nest i y <> rep z))
  =     { definition rep, hypothesis }
    be w k ((i,x):(i,y):z)
```

Here is the case for `NEST`.

```
    be w k ((i,NEST j x):z)
  =     { hypothesis, definition rep, definition NEST }
    best w k (nest i (nest j x) <> rep z)
  =     { nest homomorphism from addition to composition }
    best w k (nest (i+j) x <> rep z)
  =     { definition rep, hypothesis }
    be w k ((i+j,x):z)
```

Here is the case for `TEXT`.

```
    be w k ((i,TEXT s):z)
  =     { hypothesis, definition rep, definition TEXT }
    best w k (nest i (text s) <> rep z)
  =     { text absorbs nest }
    best w k (text s <> rep z)
  =     { definition best }
    s `Text` best w (k + length s) (rep z)
  =     { hypothesis }
    s `Text` be w (k + length s) z
```

The remaining cases are similar.

While the argument to `best` is represented using `DOC`, the result is represented using the older representation `Doc`. Thus, the function `pretty` can be defined as before.

```
  pretty w x  =  layout (best w 0 x)
```

The functions `layout`, `better`, and `fits` are unchanged. The final code is collected in Section 7.

4 Examples

A number of convenience functions can be defined.

```
x <+> y            =  x <> text " " <> y
x </> y            =  x <> line <> y

folddoc f []       =  nil
folddoc f [x]      =  x
folddoc f (x:xs)   =  f x (folddoc f xs)

spread             =  folddoc (<+>)
stack              =  folddoc (</>)
```

The reader may come up with many others.

Often a layout consists of an opening bracket, an indented portion, and a closing bracket.

```
bracket l x r  =  group (text l <>
                    nest 2 (line <> x) <>
                    line <> text r)
```

The following abbreviates the second tree layout function.

```
showBracket' ts  =  bracket "[" (showTrees' ts) "]"
```

Another use of `bracket` appears below.

The function `fillwords` takes a string, and returns a document that fills each line with as many words as will fit. It uses `words`, from the Haskell standard library, to break a string into a list of words.

```
x <+/> y   =  x <> (text " " :<|> line) <> y
fillwords  =  folddoc (<+/>) . map text . words
```

Recall that we do not expose `:<|>` to the user, but `x <+/> y` may be safely exposed, because it satisfies the invariant required by `:<|>`. Both `text " "` and `line` flatten to the same layout, and the former has a longer first line than the latter. Alternatively, one can rewrite the above by noting that `(text " " :<|> line)` is equivalent to `(group line)`.

A variant of `fillwords` is `fill`, which collapses a list of documents into a document. It puts a space between two documents when this leads to reasonable layout, and a newline otherwise. (I stole this function from Peyton Jones's expansion of Hughes's pretty printer library.)

```
fill []         =  nil
fill [x]        =  x
fill (x:y:zs)   =  (flatten x <+> fill (flatten y : zs))
                   :<|>
                   (x </> fill (y : zs))
```

Note the use of `flatten` to ensure that a space is only inserted between documents which occupy a single line. Note also that the invariant for `:<|>` is again satisfied. The next example demonstrates `fill` and the role of `flatten`.

Here are functions that pretty print a simplified subset of XML containing elements, attributes, and text.

```
data XML              =  Elt String [Att] [XML]
                      |  Txt String

data Att              =  Att String String

showXML x             =  folddoc (<>) (showXMLs x)

showXMLs (Elt n a []) =  [text "<" <> showTag n a <> text "/>"]
showXMLs (Elt n a c)  =  [text "<" <> showTag n a <> text ">" <>
                          showFill showXMLs c <>
                          text "</" <> text n <> text ">"]
showXMLs (Txt s)      =  map text (words s)

showAtts (Att n v)    =  [text n <> text "=" <> text (quoted v)]

quoted s              =  "\"" ++ s ++ "\""

showTag n a           =  text n <> showFill showAtts a

showFill f []         =  nil
showFill f xs         =  bracket "" (fill (concat (map f xs))) ""
```

Here is some XML printed for page width 30.

```
<p
  color="red" font="Times"
  size="10"
>
  Here is some
  <em> emphasized </em> text.
  Here is a
  <a
    href="http://www.eg.com/"
  > link </a>
  elsewhere.
</p>
```

Here is the same XML printed for page width 60.

```
<p color="red" font="Times" size="10" >
  Here is some <em> emphasized </em> text. Here is a
  <a href="http://www.eg.com/" > link </a> elsewhere.
</p>
```

Observe how embedded markup either is flattened or appears on a line by itself. If the two occurrences of `flatten` did not appear in `fill`, then one might have layouts such as the following.

```
<p color="red" font="Times" size="10" >
  Here is some <em>
    emphasized
  </em> text. Here is a <a
    href="http://www.eg.com/"
  > link </a> elsewhere.
</p>
```

This latter layout is not so pretty, because the start and close tags of the emphasis and anchor elements are crammed together with other text, rather than getting lines to themselves.

5 Related work and conclusion

Algebra Hughes has two fundamentally different concatenation operators. His horizontal concatenation operator (also written `<>`) is complex: any nesting on the first line of the second operand is cancelled, and all succeeding lines of the second operand must be indented as far as the text on the last line of the first operand. His vertical concatenation operator (written `$$`) is simple: it always adds a newline between documents. For a detailed description, see Hughes (1995).

Hughes's operators are both associative, but associate with each other in only one of two possible ways. That is, of the two equations,

```
x $$ (y <> z)  =  (x $$ y) <> z
x <> (y $$ z)  =  (x <> y) $$ z
```

the first holds but the second does not. Horizontal concatenation has a left unit, but because horizontal composition cancels nesting of its second argument, it is inherently inimicable to a right unit. Vertical concatenation always adds a newline, so it has neither unit.

In comparison, here everything is based on a single concatenation operator that is associative and has both a left and right unit. We can define an analogue of vertical concatenation.

```
x </> y  =  x <> line <> y
```

It follows immediately that `</>` is associative, and that `<>` and `</>` associate with each other both ways, though `</>` has neither unit.

Expressiveness Hughes has a `sep` operator that takes a list of documents and concatenates them horizontally with spaces in between if the result fits on one line, and vertically otherwise. Here there is a `group` operator that fits a document on one line if possible.

Despite their differences, the two libraries let one express many typical layouts in roughly the same way, with some things being more convenient in Hughes's library and others being more convenient for the library given here.

However, there are some layouts that Hughes's library can express and the library given here cannot. It is not clear whether these layouts are actually useful in practice, but it is clear that they impose difficulties for Hughes's library, as discussed next.

Optimality Say that a pretty printing algorithm is *optimal* if it chooses line breaks so as to avoid overflow whenever possible; say that it is *bounded* if it can make this choice after looking at no more than the next w characters, where w is the line width. Hughes notes that there is no algorithm to choose line breaks for his combinators that is optimal and bounded, while the layout algorithm presented here has both properties.

Derivation Both Hughes's library and the one presented here demonstrate the use of algebra to derive an efficient implementation.

Hughes's derivation is more complex than the one here. For Hughes, not every document set contains a flat alternative (one without a newline), and `sep` offers a flat alternative only if each component document has a flat alternative. Hence Hughes's analogue of `flatten` may return an empty set of layouts, and such empty sets require special treatment. Here, every document has a non-empty set of layouts. Hughes also requires code sequences that apply negative nesting, to unindent code when the scope of nesting introduced by horizontal concatenation is exited. Here, all nesting is non-negative.

Oppen (1980) describes a pretty-printer with similar capabilities to the one described here. Like the algorithm given here, it is optimal and bounded: line breaks are chosen to avoid overflow whenever possible, and lookahead is limited to the width of one line. Oppen's algorithm is based on a buffer, and can be tricky to implement. My first attempt to implement the combinators described here used a buffer in a similar way to Oppen, and was quite complex. This paper presents my second attempt, which uses algebra as inspired by Hughes, and is much simpler.

(Chitil has since published an implementation of Oppen's algorithm that shows just how tricky it is to get it right in a purely functional style (Chitil 2001).)

Bird's influence Richard Bird put algebraic design on the map (so to speak). John Hughes made pretty printer libraries a prime example of the algebraic approach. The greatest homage is imitation, and here I have paid as much homage as I can to Hughes and Bird.

In this respect, it may be worth drawing special attention to the handling of indentation. In Hughes's development, indentation has a decidedly imperative cast, with a fiendishly clever use of negative indentations to cancel positive ones. Here, the same result is achieved by close adherence to Bird's algebraic approach, exploiting the fact that the indentation operator distributes through concatenation. One might say, the prettier printer hatched in Bird's nest.

6 Acknowledgements

I thank Richard Bird, Koen Claessen, John Hughes, Sam Kamin, Daniel J. P. Leijen, Jeffrey Lewis, Chris Okasaki, Simon Peyton Jones, and Andreas Rossberg for comments on earlier versions of this note.

7 Code

```
-- The pretty printer

infixr 5                 :<|>
infixr 6                 :<>
infixr 6                 <>

data  DOC                =  NIL
                         |  DOC :<> DOC
                         |  NEST Int DOC
                         |  TEXT String
                         |  LINE
                         |  DOC :<|> DOC

data  Doc                =  Nil
                         |  String `Text` Doc
                         |  Int `Line` Doc

nil                      =  NIL
x <> y                   =  x :<> y
nest i x                 =  NEST i x
text s                   =  TEXT s
line                     =  LINE

group x                  =  flatten x :<|> x

flatten NIL              =  NIL
flatten (x :<> y)        =  flatten x :<> flatten y
flatten (NEST i x)       =  NEST i (flatten x)
flatten (TEXT s)         =  TEXT s
flatten LINE             =  TEXT " "
flatten (x :<|> y)       =  flatten x

layout Nil               =  ""
layout (s `Text` x)       =  s ++ layout x
layout (i `Line` x)       =  '\n' : copy i ' ' ++ layout x

copy i x                  =  [ x | _ <- [1..i] ]

best w k x                =  be w k [(0,x)]

be w k []                 =  Nil
be w k ((i,NIL):z)        =  be w k z
be w k ((i,x :<> y):z)    =  be w k ((i,x):(i,y):z)
be w k ((i,NEST j x):z)   =  be w k ((i+j,x):z)
be w k ((i,TEXT s):z)     =  s `Text` be w (k+length s) z
be w k ((i,LINE):z)       =  i `Line` be w i z
be w k ((i,x :<|> y):z)   =  better w k (be w k ((i,x):z))
                                         (be w k ((i,y):z))

better w k x y            =  if fits (w-k) x then x else y

fits w x | w < 0          =  False
fits w Nil                =  True
fits w (s `Text` x)       =  fits (w - length s) x
fits w (i `Line` x)       =  True

pretty w x                =  layout (best w 0 x)

-- Utility functions

x <+> y                   =  x <> text " " <> y
x </> y                   =  x <> line <> y

folddoc f []              =  nil
folddoc f [x]             =  x
folddoc f (x:xs)          =  f x (folddoc f xs)

spread                    =  folddoc (<+>)
stack                     =  folddoc (</>)

bracket l x r             =  group (text l <>
                               nest 2 (line <> x) <>
                               line <> text r)

x <+/> y                  =  x <> (text " " :<|> line) <> y

fillwords                 =  folddoc (<+/>) . map text . words

fill []                   =  nil
fill [x]                 =  x
fill (x:y:zs)            =  (flatten x <+> fill (flatten y : zs))
                            :<|>
                            (x </> fill (y : zs))

-- Tree example

data  Tree               =  Node String [Tree]

showTree (Node s ts)     =  group (text s <> nest (length s) (showBracket ts))

showBracket []           =  nil
showBracket ts           =  text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees [t]            =  showTree t
showTrees (t:ts)         =  showTree t <> text "," <> line <> showTrees ts

showTree' (Node s ts)    =  text s <> showBracket' ts

showBracket' []          =  nil
showBracket' ts          =  bracket "[" (showTrees' ts) "]"

showTrees' [t]           =  showTree t
showTrees' (t:ts)        =  showTree t <> text "," <> line <> showTrees ts

tree                     =  Node "aaa" [
                              Node "bbbbb" [
                                Node "ccc" [],
                                Node "dd" []
                              ],
                              Node "eee" [],
                              Node "ffff" [
                                Node "gg" [],
                                Node "hhh" [],
                                Node "ii" []
                              ]
                            ]

testtree w               =  putStr (pretty w (showTree tree))
testtree' w              =  putStr (pretty w (showTree' tree))

-- XML example

data XML                 =  Elt String [Att] [XML]
                         |  Txt String
```

```
data Att                = Att String String

showXML x               = folddoc (<>) (showXMLs x)

showXMLs (Elt n a [])   = [text "<" <> showTag n a <> text "/>"]
showXMLs (Elt n a c)    = [text "<" <> showTag n a <> text ">" <>
                           showFill showXMLs c <>
                           text "</" <> text n <> text ">"]
showXMLs (Txt s)        = map text (words s)

showAtts (Att n v)      = [text n <> text "=" <> text (quoted v)]

quoted s                = "\"" ++ s ++ "\""

showTag n a             = text n <> showFill showAtts a

showFill f []           = nil
showFill f xs           = bracket "" (fill (concat (map f xs))) ""

xml                     = Elt "p" [
                            Att "color" "red",
                            Att "font" "Times",
                            Att "size" "10"
                          ] [
                            Txt "Here is some",
                            Elt "em" [] [
                              Txt "emphasized"
                            ],
                            Txt "text.",
                            Txt "Here is a",
                            Elt "a" [
                              Att "href" "http://www.eg.com/"
                            ] [
                              Txt "link"
                            ],
                            Txt "elsewhere."
                          ]

testXML w               = putStr (pretty w (showXML xml))
```

References

[Chitil 2001] Olaf Chitil. Pretty Printing with Lazy Dequeues. ACM SIGPLAN Haskell Workshop, Firenze, Italy, 2 September 2001, Universiteit Utrecht UU-CS-2001-23, p. 183–201.

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

[Oppen 1980] Derek Oppen. Pretty-printing. *ACM Transactions on Programming Languages and Systems*, 2(4): 1980.

[Peyton Jones 1997] Simon Peyton Jones. Haskell pretty-printer library, 1997. Available from `http://www.haskell.org/libraries/#prettyprinting`.