A prettier printer

Philip Wadler
Bell Labs, Lucent Technologies
wadler@research.bell-labs.com

April 1997, revised March 1998

Abstract

John Hughes has made pretty printers one of the prime demonstrations of using combinators to develop a library, and algebra to implement it. This note presents a new design for pretty printers which improves on Hughes's classic design. The new design is based on a single concatenation operator which is associative and has a left and right unit. Hughes's design requires two separate operators for concatenation, where horizontal concatenation has a right unit but no left unit, and vertical concatenation has neither unit.

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 a 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 many years, John Hughes has refined the design of pretty printers to a fine art. 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 note 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 the horizontal composition possessing a right unit but no left unit, and the vertical composition possessing neither unit. The new library, being more uniform, is slightly easier to implement than Hughes's; our final version is more efficient than Hughes's, and about 60% as long.

However, I come not to bury Hughes but to praise him. The new library was inspired by Hughes's previous work and designed using his algebraic approach. Readers familiar with Hughes's

paper will spot many similarities to this one. Nonetheless, this note assumes no previous familiarity with pretty printers. A more detailed comparison with Hughes's work is reserved for the end.

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 primitives 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 implementation is simple, but it is not especially efficient (nesting examines every character of the nested document) nor does it generalise easily. We will consider a more algebraic implementation shortly.

We will occasionally use the following utility operators.

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

The first is concatenation with an added space, the second is concatenation with an added newline.

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) (showList ts)
```

```
showList []           =  nil
showList 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 <> showList ts

showList []           =  nil
showList 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.

$$\texttt{text}\ s_0\ \texttt{<>}\ \texttt{nest}\ i_1\ \texttt{line}\ \texttt{<>}\ \texttt{text}\ s_1\ \texttt{<>}\ \cdots\ \texttt{<>}\ \texttt{nest}\ i_k\ \texttt{line}\ \texttt{<>}\ \texttt{text}\ s_k$$

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 concatentation. 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 pair of laws state that layout is a homomorphism from document concatention to string concatentation. 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                  -- nil
                      |  String `Text` Doc    -- text s <> x
                      |  Int `Line` Doc       -- nest i line <> x
```

Here `Nil` represents `nil`, and `(s ‘Text‘ x)` represents `(text s <> x)`, and `(i ‘Line‘ x)` represents `(nest i line <> x)`.

It is then 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 could naively view a document as equivalent to a string, now we should 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 same set with one layout added 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) (showList ts))
```

If the previous document is printed with `layout 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 give a formal 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 documents. The `flatten` operator replaces each line break (and its associated indentation) by a single space; note that sets should be created in such a way that all documents in the set flatten to the same document.

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.

An alternative formulation is possible. As it turns out, union is only introduced by grouping, so its left argument is always already flattened. Hence, we might replace the law above by the following.

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

However, this precludes any more flexible use of union to describe alternative layouts, and measurements show it improves efficiency by less than 1%, so we will forego this trick.

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                  -- nil
                      |  String `Text` Doc    -- text s <> x
                      |  Int `Line` Doc       -- nest i line <> x
                      |  Doc `Union` Doc      -- x <|> y
```

As an invariant, we require that in `(x `Union` y)` that all the first lines in `x` are longer than all the first lines in `y`.

To achieve acceptable performance, we will aim to 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
```

If we are to lay this out with a field width of 5, then we must pick the last of these – and we would like to eliminate the others in one fell swoop. Our best bet for achieving this is to pick a representation that brings to the front any common string. For instance, we will aim to 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, one 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.

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 union, each document on the left begins with a space while each document on the right begins with a newline, so we have maintained the invariant that first lines are longer in the left operand of `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 n (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, the first line of the left operand must be longer than the 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. The case for negative widths is not merely esoteric, as the code 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 generalise 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 generalising each operation to work on a list of indentation-document pairs.

Here is how the fix acts on the simple documents (with a single possible layout) from the first section. 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
```

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
```

Operators will act on a list of indentation-document pairs. The representation function maps these into the corresponding document.

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

A generalised layout operation, `lay`, is defined in terms of the old layout operation and the representation function.

```
lay  =  layout . rep                              (hypothesis)
```

Now it is possible to compute the new layout function from the old. The result is as follows.

```
layout x              =  lay [(0,x)]

lay []                =  ""
lay ((i,NIL):z)       =  lay z
lay ((i,x:<>y):z)     =  lay ((i,x):(i,y):z)
lay ((i,NEST j x):z)  =  lay ((i+j,x):z)
lay ((i,TEXT s):z)    =  s ++ lay z
lay ((i,LINE):z)      =  '\n' : copy i ' ' ++ lay z
```

Each line follows by a straightforward computation. Here is the derivation of the first line.

```
    lay [(0,x)]
=     { hypothesis, definition rep }
    layout (nest 0 x <> nil)
=     { nil is unit for <> }
    layout (nest 0 x)
=     { nest homomorphism from addition to composition }
    layout x
```

Here is the case for `:<>`.

```
  lay ((i,x:<>y):z)
=     { hypothesis, definition rep, definition :<> }
  layout (nest i (x <> y) <> rep z)
=     { nest distributes over <> }
  layout ((nest i x <> nest i y) <> rep z)
=     { <> is associative }
  layout (nest i x <> (nest i y <> rep z))
=     { definition rep, hypothesis }
  lay ((i,x):(i,y):z)
```

Here is the case for `NEST`.

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

Here is the case for `TEXT`.

```
    lay ((i,TEXT s):z)
=       { hypothesis, definition rep, definition TEXT }
    layout (nest i (text s) <> rep z)
=       { text absorbs nest }
    layout (text s <> rep z)
=       { definition layout }
    s ++ layout (rep z)
=       { hypothesis }
    s ++ lay z
```

The remaining cases are similar.

The same fix acts for documents with alternative layouts. The code is collected in Figures 1 and 2. Only one function requires generalisation, `best`. Since `best` processes a document and chooses the best layout, its result (and the argument to `layout` and `fits`) can be represented exactly as in the first section.

4 Comparison with Hughes

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 simpler: it always adds a newline between documents. For a detailed description of the operators, see Hughes (1995).

Hughes's operators are both associative, but associate with each other only one way around. 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 inamicable to a right unit. Vertical concatenation always adds a newline, so it has neither unit.

```
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] ]
```

Figure 1: Pretty printer

```
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)
```

Figure 2: Pretty printer, continued

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, 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. In contrast, here there is a `group` operator that fits a document on one line if possible. For Hughes, some pretty-printers return documents, while others return lists (to be concatenated with other lists and processed by `sep`); for us, all pretty-printers return documents.

The two approaches encourage different styles. For Hughes, a layout is typically specified by a `sep` operator with an appropriate nesting added to each line. A typical entry might look like this.

```
showExp (Cond e0 e1 e2)  =  sep [ text "if " <> pp e0,
                                  nest 2 (text "then " <> pp e1),
```

```
                                    nest 2 (text "else " <> pp e2)]
```

For us, a layout is typically delimited by a single nested group.

```
showExp (Cond e0 e1 e2)  =  group (nest 2 (text "if " <> pp e0 <> line <>
                                           text "then " <> pp e1 <> line <>
                                           text "else " <> pp e2))
```

Although the styles are slightly different, it seems equally easy to use either format.

However, there are some layouts that are easy for Hughes but hard or impossible for us, and vice versa. For instance, the first form of `showTree` we gave in this paper is easier for Hughes, since horizontal concatenation will compute the amount by which to nest; while the second form is harder for Hughes, since it requires restructuring the program to combine `showTree` and `showList`, and `showTrees` must return a list of documents to be combined with the `sep` combinator.

Further, there are some layouts that Hughes can express but we cannot. It is not clear whether these layouts are actually useful in practice, but it is clear that they impose difficulties for Hughes in choosing where to place line breaks, 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 both optimal and bounded. Here is his example.

```
sep [text "a", text "b"] <> (text "c" $$
                             text "d" $$
                             text "e" $$
                             text "f" $$
                             text "g" $$
                             text "hijklmnop")
```

This has two possible layouts.

```
a bc              a
   d              bc
   e               d
   f               e
   ghijklmnop      f
                   ghijklmnop
```

To print in a line of width twelve, the second layout must be chosen. Since the vertical segment could be arbitrarily long, optimality requires unbounded lookahead. To avoid arbitrary lookahead,

Hughes uses a heuristic algorithm that occasionally overruns a line when this might have been avoided.

In contrast, the pretty-printing algorithm presented here is optimal and bounded. This is possible because the combinators are less expressive, as the indentation of a line does not depend on the layout chosen for the previous lines.

Derivation Hughes derives his pretty-printer by an application of algebra, and his work inspired the approach taken here.

However, Hughes derivations is more complex than ours. For Hughes, not every document set contains a flat alternative (one without a newline), and `sep` offers a flat alternative only if each component documents has a flat alternative. As a result, Hughes must treat empty document sets specially. Hughes requires code sequences that apply negative nesting, to unindent code when the scope of nesting introduced by horizontal concatenation is exited. No empty document sets or negative indentations are required here. The final code given here is more efficient than that of Hughes, and about 60% as long.

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 fairly tricky to implement. My first attempt at implementing 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.

So three cheers for Hughes! His algebraic approach works.

Acknowledgements I thank John Hughes, Simon Peyton Jones, Jeffrey Lewis, and Daniel J. P. Leijen for comments on earlier versions of this note.

References

[1] Hug95 Hughes, J. 1995. The design of a pretty-printer library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag LNCS 925.

[2] Opp80 Oppen, D. 1980. Pretty-printing. *ACM Transactions on Programming Languages and Systems*, 2(4).

[3] Pey97 Peyton Jones, S. L. 1997. Haskell pretty-printer library. Available at `http://www.cse.ogi.edu/~simonpj/`.