# Partial Pretty Printing

Justin Pombrio

August 15, 2021

[FILL] introduction; relationship to Wadler's Prettier Printer and others

[FILL] only consider length of first line, trading off expressiveness for efficiency. Linear time.

[FILL] peephole efficiency

## 1 Reference Implementation

[FILL]

### 1.1 Documents

Documents will be made by eight constructors:

```
data Doc = Nil          -- empty document
         | Text String  -- text (without newlines)
         | Newline      -- newline
         | Flat x       -- disallow newlines in x
         | i :>> x      -- indent x by i spaces
         | x :+ y       -- concatenation of x and y
         | x :| y       -- choice between x and y
         | Err          -- error
```

`Nil` prints nothing at all, `Text t` prints a string verbatim, and `Newline` prints a newline. `x :+ y` is the *concatenation* of $x$ and $y$: it first prints $x$, then $y$, with no separation.

Parts of a document can be indented. `i :>> x` indents $x$ by $i$ spaces (specifically, it inserts $i$ spaces after every newline in $x$.) `Flat x`, on the other hand, *disallows* newlines: it produces an error `Err` if it encounters a newline in $x$. This is only useful in conjunction with choices, described next.

A choice `x :| y` will print either $x$ or $y$. Choices are resolved *greedily*, and only based on the current line. Specifically, $x$ is chosen iff it does not cause the current line to exceed the pretty printing width or choosing $y$ would result in an error.

[TODO: move para?] This rule for resolving choices does limit how pretty documents can be. Certain ways of formatting can be either great or terrible, and you cannot tell which just by looking at their first line. These must be avoided, lest someone runs into the terrible case. [FILL: examples, e.g. alignment]. However, this way of resolving choices also has a very strong advantage: it allows for partial pretty printing. [REF: more details].

For example, this document:

```
2 :>> (Text "Hello" :+ (Text "␣" :| Newline) :+ "world")
```

will either print on one line if the printing width is at least 11:

```
Hello world
```

on on two lines otherwise:

```
Hello
  world
```

## 1.2 Pretty Printing without Choices

Let's turn this informal definition into code. For the moment, we'll keep the implementation naive at the expense of efficiency (even to the extent of exponential inefficiency). We will later use it to derive a more efficient implementation that provably behaves the same.

First we need a data structure for the output of pretty printing, and some basic operations on it. The output—call it a *layout*—is either a list of lines, or an error in case of encountering `Err`:

```
data Layout = LErr | Layout [String]
```

Layouts can be displayed, flattened, indented, and appended:

```
display :: Layout -> String
display LErr = "Error!" -- convenient for testing
display (Layout lines) = intercalate "\n" lines

flatten :: Layout -> Layout
flatten LErr = LErr
flatten (Layout lines) =
  if length lines == 1
  then Layout lines
  else LErr

indent :: Int -> Layout -> Layout
indent _ LErr = LErr
indent i (Layout (line : lines)) =
  Layout (line : map addSpaces lines)
  where addSpaces line = replicate i ' ' ++ line

append :: Layout -> Layout -> Layout
append (Layout lines1) (Layout lines2) =
  Layout (init lines1 ++ [middleLine] ++ tail lines2)
  where middleLine = (last lines1 ++ head lines2)
append _ _ = LErr
```

Pretty printing a document that contains no choices is completely straightforward:

```
-- Preliminary!
pp :: Doc -> Layout
pp Err = LErr
pp Empty = Layout [""]
pp (Text t) = Layout [t]
pp Newline = Layout ["", ""]
pp (Flat x) = flatten (pp x)
pp (i :>> x) = indent (pp x)
pp (x :+ y) = append (pp x) (pp y)
```

In fact, since we aren't making any choices, notice that we aren't even using the pretty printing width yet.

## 1.3 Pretty Printing with Choices

Ideally, pretty printing would recursively walk the document and resolve any choices as it encountered them. However, this is not easy because at that point we have not printed what comes after the choice, which might effect how long the current line is. Instead, let's delay resolving any choices until we have processed the whole document. Thus `pp` will produce a decision tree of `Layouts`, with a branch for each `choice` and storing the line number at which the choice is to be made:

```
data Layouts = Branch Int Layouts Layouts
             | Leaf Layout
```

We need to extend our previous functions to handle `Layouts` as well:

```
flatten' :: Layouts -> Layouts
flatten' (Branch n x y) = Branch n (flatten' x) (flatten' y)
flatten' (Leaf layout) = Leaf (flatten layout)

indent' :: Int -> Layouts -> Layouts
indent' i (Branch n x y) = Branch n (indent' i x) (indent' i y)
indent' i (Leaf layout) = Leaf (indent i layout)

append' :: Layouts -> Layouts -> Layouts
append' (Branch n x y) z = Branch n (append' x z) (append' y z)
append' (Leaf layout) (Branch n x y) =
  Branch (n + numNewlines layout)
         (append' (Leaf layout) x)
         (append' (Leaf layout) y)
append' (Leaf layout1) (Leaf layout2) = Leaf (append layout1 layout2)
```

This is mostly a pointwise extension (i.e., boilerplate), except that if `append'` is given two branches, see that it nests the right one into the left one rather than vice-versa. This represents a decision to resolve choices from left to right.

The `pp` function is now trivial: it just calls the appropriate function:

```
pp :: Doc -> Layouts
pp Err = Leaf LErr
pp Empty = Leaf (Layout [""])
pp (Text t) = Leaf (Layout [t])
pp Newline = Leaf (Layout ["", ""])
pp (Flat x) = flatten' (pp x)
pp (i :>> x) = indent' i (pp x)
pp (x :+ y) = append' (pp x) (pp y)
pp (x :| y) = Branch 0 (pp x) (pp y)
```

This produces a decision tree. We can resolve the decisions in the tree to get a single `Layout` by simple recursion:

```
resolve :: Int -> Layouts -> Layout
resolve _ (Leaf layout) = layout
resolve w (Branch n x y) =
  pick w n (resolve w x) (resolve w y)
```

where `pick` picks the "best" of two layouts. It avoids errors if possible, and otherwise picks whichever layout does not exceed the maximum width `w` on the current line `n`:

```
pick :: Int -> Int -> Layout -> Layout -> Layout
pick w _ LErr LErr = LErr
pick w _ (Layout lines) LErr = Layout lines
pick w _ LErr (Layout lines) = Layout lines
pick w n (Layout lines1) (Layout lines2) =
  if length (lines1 !! n) <= w
  then Layout lines1
  else Layout lines2
```

A user-facing interface should obtain a decision tree, resolve the decisions, and display the resulting layout:

```
pretty :: Int -> Doc -> String
pretty w x = display (resolve w (pp x))
```

This is a complete and correct implementation. However, it is exponential time, as it tries resolving each choice both ways.

This makes it sound useless, but it is not! We can use it to prove algebraic laws about equivalence between `Doc`s, and then use these laws to derive a more efficient implementation that (provably) behaves the same.

## 2 Laws of Pretty Printing

This section lists equivalences between `Doc`s. An equivalence `x = y` between docs typically means that `pp x = pp y`. However, for a couple of the laws [FILL] it instead means that `x` and `y` are contextually equivalent in the expression `resolve w (pp C[x])`, for all widths `w` and contexts `C`.

All of the equivalences in this section can be proven from the implementation of `naivePP` using basic equational reasoning. The proofs are given in [TODO: the appendix, but for now proofs_tree.md].

**Concatenation.** The concatenation of two documents just prints them one after another (with no space or newline in between). Thus concatenation with an empty document has no effect, and concatenation is associative.

$$\texttt{Nil :+ x} = \texttt{x :+ Nil} = x \qquad \text{(concat-unit)}$$

$$\texttt{(x :+ y) :+ z} = \texttt{x :+ (y :+ z)} \qquad \text{(concat-assoc)}$$

**Text.** `Text t` is rendered exactly as is. Thus the empty string is the same as the empty document, and the concatenation of two texts just concatenates their strings.

$$\texttt{""} = \texttt{Nil} \qquad \text{(text-empty)}$$

$$\texttt{Text t1 :+ Text t2} = \texttt{Text (t1 ++ t2)} \qquad \text{(text-concat)}$$

**Indentation and Flattening.** Indentation and flattening can be lowered to the leaves of the document. They both leave text (and thus also empty documents) unchanged, but behave differently on newlines:

- A newline indented by $i$ is a newline followed by $i$ spaces.
- The flattening of a newline is an error, since it's impossible to fit a newline on one line.

$$\texttt{i :>> Nil} = \texttt{Nil} \qquad \text{(indent-absorb-empty)}$$

$$\texttt{i :>> Text t} = \texttt{Text t} \qquad \text{(indent-absorb-text)}$$

$$\texttt{i :>> Newline} = \texttt{Newline :+ (replicate i '')} \qquad \text{(indent-newline)}$$

$$\texttt{i :>> (x :+ y)} = \texttt{(i :>> x) :+ (i :>> y)} \qquad \text{(indent-distr-concat)}$$

$$\texttt{i :>> (x :| y)} = \texttt{(i :>> x) :| (i :>> y)} \qquad \text{(indent-distr-choice)}$$

$$\texttt{Flat Nil} = \texttt{Nil} \qquad \text{(flat-absorb-empty)}$$

$$\texttt{Flat (Text t)} = \texttt{Text t} \qquad \text{(flat-absorb-text)}$$

$$\texttt{Flat Newline} = \texttt{Err} \qquad \text{(flat-newline)}$$

$$\texttt{Flat (x :+ y)} = \texttt{Flat x :+ Flat y} \qquad \text{(flat-distr-concat)}$$

$$\texttt{Flat x :| y} = \texttt{Flat x :| Flat y} \qquad \text{(flat-distr-choice)}$$

Also, indentation respects addition: indenting by zero spaces is the same as not indenting at all, and indenting by $i$ spaces and then $j$ spaces is the same as indenting by $i + j$ spaces. Along the same lines, flattening twice has no effect.

$$\texttt{0 :>> x} = x \qquad \text{(indent-identity)}$$
$$\texttt{j :>> (i :>> x)} = \texttt{(i+j) :>> x} \qquad \text{(indent-compose)}$$
$$\texttt{Flat (Flat x)} = \texttt{Flat x} \qquad \text{(flat-compose)}$$

**Errors.** There is only one source of error: flattening a newline (shown above). Once created, errors are contagious and propagate up to the root of the document, except when inside a choice in which cae they eliminate that option of the choice.

$$\texttt{Err :+ x} = \texttt{x :+ Err} = \texttt{Err} \qquad \text{(error-concat)}$$
$$\texttt{i :>> Err} = \texttt{Err} \qquad \text{(error-indent)}$$
$$\texttt{Flat Err} = \texttt{Err} \qquad \text{(error-flat)}$$
$$\texttt{Err :| x} = \texttt{x :| Err} = x \qquad \text{(error-choice)}$$

**Choice.** Choice is associative, and if you concatenate a choice with some text, that's the same as concatenating the text inside the choice. [FILL]

$$\texttt{x :| (y :| z)} = \texttt{(x :| y) :| z} \qquad \text{(choice-assoc)}$$
$$\texttt{Text t :+ (x :| y)} = \texttt{(Text t :+ x) :| (Text t :+ y)} \qquad \text{(choice-distr-text-left)}$$
$$\texttt{(x :| y) :+ z} = \texttt{(x :+ z) :| (y :+ z)} \qquad \text{(choice-distr-right)}$$

**A law that doesn't hold.** Surprisingly, concatenation does *not* in always distribute over choice. That is, `x :+ (y :| z)` is not always equal to `(x :+ y) :| (x :+ z)`. This is because $x$ could contain a newline. In that case, we would have:

$$\texttt{Newline :+ (y :| z)} \stackrel{?}{=} \texttt{(Newline :+ y) :| (Newline :+ z)}$$

The document on the left makes a legitimate choice between $y$ and $z$, comparing their first lines to see which one fits. However, the document on the right makes a degenerate choice: it compares the first lines of `Newline :+ y` and `Newline :+ z`, which are both empty! Thus it always picks `Newline :+ y`, regardless of $y$ and $z$. As a result, the concatenation of a newline and a choice cannot be simplified.

# 3 Efficient Implementation

We can use these laws to derive an efficient implementation. The first step is to see that they are sufficient to convert any document into a normal form, as follows:

- Eliminate `Nil` by converting it into text, using the "text-empty" law.
- Lower each `Flat` and `(:>>)` (indent) to the leaves of the document. If it hits a newline, `Flat` will produce an `Err`, and `(:>>)` will insert spaces after the newline. This uses all of the "indent" and "flat" laws except for indent-identity.
- Raise all `Err`s up towards the root of the document. They will either eliminate an option from a choice via the "error-choice" law, or cause the entire document to become an `Err`. This uses the four "error" laws.

[FILL]