Functional Pearl: a pretty but not greedy printer

Jean-Philippe Bernardy
Tweag IO
jean-philippe.bernardy@tweag.io

Abstract

This paper proposes an new specification of pretty printing which is stronger than the state of the art: we require the output to be the shortest possible, and we also offer the ability to align subdocuments at will. We argue that specification precludes a greedy implementation. Yet, we provide an implementation which behaves linearly in the size of the output. The derivation of the implementation demonstrates functional programming methodology.

Keywords Pearl, Pretty Printing

1. Introduction

A pretty printer is a program that prints data structures in a way which makes them pleasant to read. (The data structures in question often represent programs, but not always.) Pretty printing has historically been used by the functional programming community to showcase proper style. The pretty printer of **?**] remains an influential example of functional programming design, while that of **?**] was published as a chapter in a book dedicated to the "fun of programming".

In addition to their esthetical and pedagogical value, the pretty printers of Hughes and Wadler are practical implementations which form the basis of industrial-strength pretty-printing packages, which remain popular today. Hughes' design has been refined by Peyton Jones, and is available as the Hackage package pretty[1], while Wadler's design has been extended by Leijen and made available as the wl-print package[2].

This paper is a bold attempt to improve some aspects of the aforementioned landmark pieces of work in the functional programming landscape. Yet, our goal is slightly different to that of Hughes and Wadler. Indeed, they aim first and formost to demonstrate general principles of functional programming development, with an emphasis on the efficency of the algorithm. The methodological idea is to derive a greedy algorithm from a functional specification. In the process, they give themselves some leeway in what they accept as pretty outputs (see #sec_notSoPretty). In contrast, my primary goal is to produce *the prettiest output*, at the cost of efficiency. Yet, we the final result is reasonably efficient (Sec. 6).

[1] `https://hackage.haskell.org/package/pretty`

[2] `https://hackage.haskell.org/package/wl-pprint`

Let specify the desired behaviour of a pretty printer, first informally, as the following principles:

Principle 1 (VISIBILITY)**.** *A pretty printer shall layout all its output within the width of the page.*

Principle 2 (LEGIBILITY)**.** *A pretty printer shall make clever use of layout, to make it easy for a human to recognise the hierarchical organisation of data.*

Principle 3 (FRUGALITY)**.** *A pretty printer shall minimize the number of lines used to display the data.*

Furthermore, the first principle takes precedence over the second one, which itself takes precedence over the third one. In the rest of the paper, we interpret the above three principles as an optimisation problem, and derive a program which solves it efficiently enough.

2. Interface (Syntax)

Let us use an example to guide the development of our pretty-printing interface. Assume that we want to pretty print S-Expressions, and that they are represented as follows:

```
data SExpr = SExpr [SExpr] | Atom String
  deriving Show
```

Using the above representation, the S-Expr `(a b c d)` has the following encoding:

```
abcd :: SExpr
abcd = SExpr [Atom "a", Atom "b", Atom "c", Atom "d"]
```

The goal of the pretty printer is to render a given S-Expr according to the three principles of pretty printing: VISIBILITY, FRUGALITY and LEGIBILITY. While it is clear how the first two principles constrain the result, it is less clear how the third principle plays out: we must specify more precisely which layouts are admissible. To this end, we assert that in a pretty display of an S-Expr, we would like the elements to be either concatenated horizontally, or aligned vertically. Thus the legible layouts of our abcd example would be either

```
(a b c d)
```

or

```
(a
 b
 c
 d)
```

In general, a pretty printing library must provide the means to express the set of legible layouts: it is up to the user to reify LEGIBILITY on the data structure of interest. The printer will then

automatically pick the smallest (FRUGALITY) legible layout which fits the page (VISIBILITY).

Our layout-description API is similar to Hughes's: we can express both vertical ($$) and horizontal (<>) composition of documents, as well as embedding raw text and provide choice between layouts (<|>). At this stage, we keep the representation of documents abstract, by using a typeclass (Doc) which provides the above combinators, as well as means of rendering a document:

```
text   :: Doc d ⇒ String → d
(<>)   :: Doc d ⇒ d → d → d
($$)   :: Doc d ⇒ d → d → d
(<|>)  :: Doc d ⇒ d → d → d
render :: Doc d ⇒ d → String
```

We can then define a few useful combinators on top of the above: the empty document; concatenation with an intermediate space (<+>); vertical and horizontal concatenation of multiple documents.

```
empty :: Layout d ⇒ d
empty = text ""
(<+>) :: Layout d ⇒ d → d → d
x <+> y = x <> text " " <> y
hsep, vcat :: Doc d ⇒ [d] → d
vcat = foldr1 ($$)
hsep = foldr1 (<+>)
```

We can furthermore define the choice between horizontal and vertical concatenation:

```
sep :: Doc d ⇒ [d] → d
sep [] = empty
sep xs = hsep xs <|> vcat xs
```

Turning S-expressions into a Doc is then child's play:

```
pretty :: Doc d ⇒ SExpr → d
pretty (Atom s)   = text s
pretty (SExpr xs) = text "(" <>
                    (sep $ map pretty xs) <>
                    text ")"
```

3. Semantics: an example

The above API provides a syntax to describe layouts. The natural question is then: what should its semantics be? In other words, how do we turn the three principles into a formal specification? In particular, how do we turn the above pretty function into a pretty printer of S-Expressions?

Let us use an example to try and answer the question, and outline why neither Wadler's or Hughes' answer is satisfactory. Suppose we want to pretty-print the following S-Expr (which is specially crafted to demonstrate issues with both Hughes and Wadler libraries):

```
testData :: SExpr
testData = SExpr [SExpr [Atom "abcde", abcd4],
                  SExpr [Atom "abcdefgh", abcd4]]
  where abcd4 = SExpr [abcd, abcd, abcd, abcd]
```

Remember that we would like elements inside an S-Expr to be either aligned vertically or concatenated horizontally (for LEGIBILITY), The second option will be preferred over the first (FRUGALITY), as long as the text fits within the page width (VISIBILITY). Interpreting the above (still informal) specification yields the following results. 1. On a 80-column-wide page, we would get the result displayed in Fig. 1. 2. On a 20-column-wide page, we would like to get the following output (the first line is a helper showing the column of a given character):

```
12345678901234567890

((abcde ((a b c d)
         (a b c d)
         (a b c d)
         (a b c d)
         (a b c d)))
 (abcdefgh
  ((a b c d)
   (a b c d)
   (a b c d)
   (a b c d)
   (a b c d))))
```

3.1 The liminations of Hughes and Wadler

Let us take a moment to survey the state of the art. On a 20-column page and using Hughes' library, we would get the following output instead:

```
12345678901234567890
((abcde ((a b c d)
         (a b c d)
         (a b c d)
         (a b c d)
         (a b c d)))
 (abcdefgh ((a
             b
             c
             d)
            (a
             b
             c
             d)
            (a
             b
             c
             d)
            (a
             b
             c
             d)
            (a
             b
             c
             d))))
```

The above output uses way more space than necessary, violating FRUGALITY. Why is that? Hughes states that "it would be unreasonably inefficient for a pretty-printer do decide whether or not to split the first line of a document on the basis of the content of the last." (sec. 7.4 of his paper). Therefore, he chooses a greedy algorithm, which processes the input line by line, trying to fit as much text as possible on the current line, without regard for what comes next. In our example, the algorithm can fit `(abcdefgh ((a` on the sixth line, but then it has committed to a very deep indentation level, which forces to display the remainder of the document in a narrow area, wasting vertical space.

How does Wadler's library fare on the example? Unfortunately, we cannot answer the question in a strict sense. Indeed, Wadler's API is too restrictive to even *express* the layout that we are after. That is, one can only specify a *constant* amount of indentation, not one that depends on the contents of a document. This means that Wadler's library lacks the capability to express that a multi-line

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

((abcde ((a b c d) (a b c d) (a b c d) (a b c d) (a b c d)))
 (abcdefgh ((a b c d) (a b c d) (a b c d) (a b c d) (a b c d))))
```

Figure 1. Example expression printed on 80 columns. The first line is a helper showing the column of a given character.

sub-document b should be laid out to the right of a document a (even if a is single-line). Instead, b must be put below a. Because of this restriction, with any reasonable specification, the best output that Wadler's library can produce is the following:

```
12345678901234567890
((abcde
  ((a b c d)
   (a b c d)
   (a b c d)
   (a b c d)
   (a b c d)))
 (abcdefgh
  ((a b c d)
   (a b c d)
   (a b c d)
   (a b c d)
   (a b c d))))
```

It does not look too bad — but there is a spurious line break after the atom `abcdefgh`. While Wadler's restriction may be acceptable to some, I find it unsatisfying for two reasons. First, spurious line breaks may appear in many places, so the rendering may be much longer than necessary, thereby violating FRUGALITY. Second, and more importantly, a document which is laid out after another cannot be properly indented in general. Let us say we would like to pretty print a ml-style equation composed of a `Pattern` and the following right-hand-side:

```
expression [listElement x,
            listElement y,
            listElement z,
            listElement w]
```

We reasonably hope to obtain the following result, which puts the list to the right of the `expression`, best respecting LEGIBILITY by clearly showing that the list is an argument of `expression`:

```
Pattern = expression [listElement x,
                      listElement y,
                      listElement z,
                      listElement w]
```

However, using Wadler's library, the indentation of the list can only be constant, so even with the best layout specification we would obtain instead the following output:

```
Pattern = expression
  [listElement x,
   listElement y,
   listElement z,
   listElement w]
```

Aligning the argument of the expression below to the left of the equal sign is bad, because it needlessly obscures the structure of the program; LEGIBILITY is not respected. In sum, the lack of a combinator for relative indentation is a serious drawback. In fact, Daan Leijen's implemenation of Wadler's design (wl-print), *does* feature an alignment combinator. However, the implemenation also uses a greedy algorithm, and thus suffers from the same issue as Hughes' library.

In sum, we have to make a choice between respecting all the principles of pretty printing or provide a greedy algorithm. Hughes does not fully respect FRUGALITY. Wadler does not fully respect LEGIBILITY. Here, I decide to respect both, but I give up on greediness. Yet, the final algorithm that I arrive at is fast enough for common pretty-printing tasks.

But; let us not get carried away: before attacking the problem of making an implementation, we need to finish the formalisation of the semantics. And before that, it is best if we spend a moment to further refine the API for defining pretty layouts.

4. Semantics, continued

4.1 Layouts

We ignore for a moment the choice between possible layouts ($\lhd\rhd$). As Hughes, we call a document without choice a *layout*.

Recall that we have inherited from Hughes a draft API for layouts:

$$\begin{aligned}
&\mathsf{text} &&:: \mathsf{Layout}\ \mathsf{l} \Rightarrow \mathsf{String} \to \mathsf{l}\\
&(\lhd\rhd) &&:: \mathsf{Layout}\ \mathsf{l} \Rightarrow \mathsf{l} \to \mathsf{l} \to \mathsf{l}\\
&(\$\$) &&:: \mathsf{Layout}\ \mathsf{l} \Rightarrow \mathsf{l} \to \mathsf{l} \to \mathsf{l}
\end{aligned}$$

At this stage, classic functional pearls would state a number of laws that the above API has to satisfy, then infer a semantics from them. Fortunately, in our case, Hughes and Wadler have already laid out this ground work, so we can take a shortcut and immediately state a compositional semantics. We will later check that the expected laws hold.

Let us interpret a layout as a *non-empty* list of lines to print. As Hughes, I'll simply use the type of lists, trusting the reader to remember the invariant.

$$\mathbf{type}\ \mathsf{L} = [\mathsf{String}]$$

Preparing a layout for printing is as easy as inserting a newline character between each string:

$$\begin{aligned}
&\mathsf{render} :: \mathsf{L} \to \mathsf{String}\\
&\mathsf{render} = \mathsf{intercalate}\ \texttt{"\textbackslash n"}
\end{aligned}$$

Embedding a string is thus immediate:

$$\begin{aligned}
&\mathsf{text} :: \mathsf{String} \to \mathsf{L}\\
&\mathsf{text}\ \mathsf{s} = [\mathsf{s}]
\end{aligned}$$

The interpretation of vertical concatenation ($$) requires barely more thought:

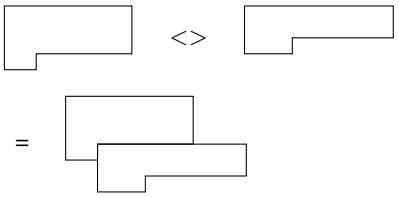
$$\begin{aligned}
&(\$\$) :: \mathsf{L} \to \mathsf{L} \to \mathsf{L}\\
&\mathsf{xs}\ \$\$\ \mathsf{ys} = \mathsf{xs} +\!\!+ \mathsf{ys}
\end{aligned}$$

The only potential difficulty is to figure out the interpretation of horizontal concatenation ($\lhd\rhd$). We will stick to Hughes' advice: "translate the second operand [to the right], so that is tabs against the last character of the first operand". For example:

```
xxxxxxxxxxxx        yyyyyyyyyyyyyyyyyyyy
xxxxxxxxx       <>  yyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxx        yyyy
xxxxxx
```

```
=   xxxxxxxxxxxxx
    xxxxxxxxx
    xxxxxxxxxxx
    xxxxxxyyyyyyyyyyyyyyyyyyyyy
          yyyyyyyyyyyyyyyyyyyyyyyyyyyyy
          yyyy
```

Or, diagramatically:



Algorithmically, one must handle the last line of the first layout and the first line of the second layout specially, as follows:

```
(<>) :: L → L → L
xs <> (y : ys) = xs₀ ++ [x ++ y] ++ map (indent ++) ys
  where xs₀ = init xs
        x = last xs
        n = length x
        indent = replicate n ' '
```

We take a quick detour to refine our API a bit. Indeed, as it becomes clear with the above definition, vertical concatenation is (nearly) a special case of horizontal composition. That is, instead of composing vertically, one can add an empty line to the left-hand-side layout and then compose horizontally. The combinator which adds an empty line is called flush, and has the following definition:

```
flush :: L → L
flush xs = xs ++ [""]
```

Horizontal concatenation is then:

```
($$) :: L → L → L
a $$ b = flush a <> b
```

One might argue that replacing ($$) by flush does not make the API shorter nor simpler. Yet, I stick this choice, for two reasons:

1. The new API clearly separates the concerns of concatenation and left-flushing documents.
2. The horizontal composition (<>) has a nicer algebraic structure than ($$). Indeed, the vertical composition ($$) has no unit, while (<>) has the empty layout as unit. (In Hughes' pretty-printer, not even (<>) has a unit, due to more involved semantics.)

To sum up, our API for layouts is the following:

```
class Layout l where
  (<>)   :: l → l → l
  text   :: String → l
  flush  :: l → l
  render :: l → String
```

Additionally, as mentioned above, layouts follow a number of algebraic laws, (written here as QuickCheck properties):

1. Layouts form a monoid, with operator (<>) and unit empty[3]:

```
prop_leftUnit :: (Doc a, Eq a) ⇒ a → Bool
prop_leftUnit a = empty <> a ≡ a
prop_rightUnit :: (Doc a, Eq a) ⇒ a → Bool
prop_rightUnit a = a <> empty ≡ a
prop_assoc :: (Doc a, Eq a) ⇒ a → a → a → Bool
prop_assoc a b c = (a <> b) <> c ≡ a <> (b <> c)
```

2. text is a monoid homomorphism:

```
prop_text_append s t = text s <> text t ≡ text (s ++ t)
prop_text_empty      = empty ≡ text ""
```

3. flush can be pulled out of concatenation, in this way:

```
prop_flush :: (Doc a, Eq a) ⇒ a → a → Bool
prop_flush a b = flush a <> flush b ≡ flush (flush a <> b)
```

   One might expect this law to hold instead:

```
a <> flush b ≡ flush (a <> b)
```

   . However, the inner flush on b goes back to the local indentation level, while the outer flush goes back to the outer indentation level, which are equal only if a ends with an empty line. In turn this condition is guaranteed only when a is itself flushed.

4.2 Choice

We proceed to extend the API with choice between layouts, yielding the final API to specify document. The extended API is accessible via a new type class:

```
class Layout d ⇒ Doc d where
  (<|>) :: d → d → d
  fail :: d
```

Again, we give the compositional semantics right away. Documents are interpreted as a set of layouts. We implement sets as lists, where order and number of occurences won't matter.

The interpretation of disjunction merely appends the list of possible layouts:

```
instance Doc [L] where
  xs <|> ys = (xs ++ ys)
  fail = []
```

Consequently, disjunction is associative.

```
prop_disj_assoc :: (Doc a, Eq a) ⇒ a → a → a → Bool
prop_disj_assoc a b c = (a <|> b) <|> c ≡ a <|> (b <|> c)
```

We simply lift the layout operators idiomatically [**?**] over sets:

```
instance Layout [L] where
  text = pure . text
  flush = fmap flush
  xs <> ys = (<>) <$> xs <*> ys
```

Consequently, concatenation and flush distribute over disjunction:

```
prop_distrl :: (Doc a, Eq a) ⇒ a → Bool
prop_distrl a = (a <|> b) <> c ≡ (a <> c) <|> (b <> c)
prop_distrr :: (Doc a, Eq a) ⇒ a → Bool
prop_distrr a = c <> (a <|> b) ≡ (c <> a) <|> (c <> b)
prop_distrflush :: (Doc a, Eq a) ⇒ a → a → Bool
prop_distrflush a b = flush (a <|> b) ≡ flush a <|> flush b
```

4.3 Semantics

We can finally define formally what it means to render a document. To do so, we pick a frugal layout among the visiible ones, according to VISIBILITY:

```
render = render .
         mostFrugal .
         filter valid
```

[3] recall empty = text ""

A layout is valid if all its lines are fully valid on the page:

```
  where
    valid :: L → Bool
    valid xs = maximum (map length xs) ≤ pageWidth
    mostFrugal :: [L] → L
    mostFrugal = minimumBy (compare ‘ on ‘ length)
pageWidth = 80
```

One may expect that disjuction should also be commutative. However, the implementation of mostFrugal only picks *one* of the most frugal layouts. That is fine, as all most frugal layouts are equally good. However it also means that re-ordering the arguments of a disjunction may affect the layout being picked. Therefore, commutativity of disjunction holds only up to the length of the layout being rendered:

```
prop_disj_commut :: (Doc a, Eq a) ⇒ a → a → a → Bool
prop_disj_commut a b c = a <|> b ≅ b <|> a
infix 3 ≅
(≅) :: Layout a ⇒ a → a → Bool
(≅) = (≡) ‘ on ‘ (length . lines . render)
```

We have now defined semantics compositionally; furthermore this semantics is executable. Consequently, we can implement the pretty printing an S-Expr as follows:

```
showSExpr x = render (pretty x :: [L])
```

Running showSExpr on our example (testData) yields the expected output.

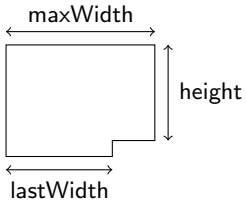
While the above semantics provide an executable implementation, it is insanely slow. Indeed, every possible combination of choices is first constructed, and only then a shortest output is picked. Thus, for an input with n choices, the running time is $O(2^n)$.

5. A More Efficient Implementation

5.1 Measures

The first insight to arrive at an efficient implementation is that it is not necessary to construct layouts fully: only some of their parameters are relevant. Let us remember that we want to sift through layouts based on the space that they take. Hence, from an algorithmic point of view, all that matters is a measure of that space. Let us define an abstract semantics for layouts, which ignores the text, and captures only the amount of space used.

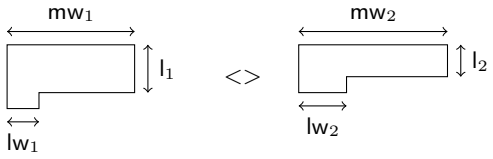
The only parameters that matter are the maximum width of the layout, the width of its last line and its height (and, because layouts cannot be empty and it's convenient to start at zero, we do not count the last line):

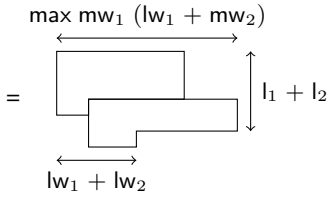


In code:

```
data M = M {height    :: Int,
            lastWidth :: Int,
            maxWidth  :: Int}
  deriving (Show, Eq, Ord)
```

Astute readers may have guessed the above semantics by looking at the diagram for composition of layouts shown earlier. Indeed, it is the above abstract semantics (M) which justifies the abstract representation of a layout that the diagram uses (a box with an odd last line). Here is the concatenation diagram annotated with those lengths:





The above diagram can be read out as Haskell code, as follows:

```
instance Layout M where
  a <> b =
    M {maxWidth  = max (maxWidth a)
                       (lastWidth a + maxWidth b),
       height    = height    a + height    b,
       lastWidth = lastWidth a + lastWidth b}
```

The other layout combinators are easy to implement:

```
text s  = M {height    = 0,
             maxWidth  = length s,
             lastWidth = length s}
flush a = M {maxWidth  = maxWidth a,
             height    = height a + 1,
             lastWidth = 0}
```

We can even give a rendering for these abstract layouts, by printing an `x` at each occupied position, completing the class instance:

```
render m = intercalate "\n"
  (replicate (height m) (replicate (maxWidth m) 'x') ++
  [replicate (lastWidth m) 'x'])
```

The correctness of the above instance relies on intution, and a proper reading of the concatenation diagram. This process being informal, we should cross-check the final result formally. To do so, we define a function which computes the measure of a full layout:

```
measure :: L → M
measure xs = M {maxWidth  = maximum $ map length $ xs,
                height    = length xs − 1,
                lastWidth = length $ last $ xs}
```

Then, to check the correctness of the Layout M instance, we verify that measure is a layout homomorphism (ignoring of course the renderer). The homomorphism property can be spelled out as the following three laws:

Lemma 1 (Measure is a Layout-homomorphism)**.**

```
measure (a <> b) ≡ measure a <> measure b
measure (flush a) ≡ flush (measure a)
measure (text s)  ≡ text s
```

(Note: on the left-hand-side of the above equations, the combinators (<>, flush*,* text*) come from the* L *instance of* Layout*, while on the right-hand-side they come from the* M *instance.)*

Proof. Checking the laws is a simple, if somewhat tedious exercise of program calculation, and thus it is deferred to the appendix. □

```
validMeasure :: M → Bool
validMeasure x = maxWidth x ≤ pageWidth
```

Having properly refined the problem, and ignoring puny details such as the actual text being rendered, we may proceed to give a fast implementation of the pretty printer.

5.2 Early filtering out invalid results

The first optimisation is to filter out invalid results early; like so:

```
text x = filter valid [text x]
xs <> ys = filter valid [x <> y | x ← xs, y ← ys]
```

We can do so because validity is monotonous:

Lemma 2 (valid is monotonous)**.** *The following two implications hold:*

$$\mathsf{valid}\ (a \mathbin{<\!\!>} b) \Rightarrow \mathsf{valid}\ a \wedge \mathsf{valid}\ b$$
$$\mathsf{valid}\ (\mathsf{flush}\ a) \Rightarrow \mathsf{valid}\ a$$

Proof. We prove the two parts separately:

1. $$\begin{aligned} & \mathsf{valid}\ (a \mathbin{<\!\!>} b) \\ \Rightarrow\ & \mathsf{maxWidth}\ (a \mathbin{<\!\!>} b) < \mathsf{pageWidth} \\ \Rightarrow\ & \mathsf{max}\ (\mathsf{maxWidth}\ a)\ (\mathsf{lastWidth}\ a + \mathsf{maxWidth}\ b) < \\ & \quad \mathsf{pageWidth} \\ \Rightarrow\ & \mathsf{maxWidth}\ a < \mathsf{pageWidth} \wedge \\ & \quad \mathsf{lastWidth}\ a + \mathsf{maxWidth}\ b < \mathsf{pageWidth} \\ \Rightarrow\ & \mathsf{maxWidth}\ a < \mathsf{pageWidth} \wedge \\ & \quad \mathsf{maxWidth}\ b < \mathsf{pageWidth} \\ \Rightarrow\ & \mathsf{valid}\ a \wedge \mathsf{valid}\ b \end{aligned}$$
2. $$\begin{aligned} \mathsf{valid}\ (\mathsf{flush}\ a) &\Rightarrow \mathsf{maxWidth}\ a < \mathsf{pageWidth} \\ &\Rightarrow \mathsf{maxWidth}\ a < \mathsf{pageWidth} \\ &\Rightarrow \mathsf{valid}\ a \end{aligned}$$

□

Consequently, keeping invalid layouts is useless: they can never be combined with another layout to produce something valid.

Lemma 3 (Invalid layouts cannot be fixed)**.**

$$\mathsf{not}\ (\mathsf{valid}\ a) \Rightarrow \mathsf{not}\ (\mathsf{valid}\ (a \mathbin{<\!\!>} b))$$
$$\mathsf{not}\ (\mathsf{valid}\ b) \Rightarrow \mathsf{not}\ (\mathsf{valid}\ (a \mathbin{<\!\!>} b))$$
$$\mathsf{not}\ (\mathsf{valid}\ a) \Rightarrow \mathsf{not}\ (\mathsf{valid}\ (\mathsf{flush}\ a))$$

Proof. By contraposition of Lem. 2 □

5.3 Pruning out dominated results

The second optimisation relies on the insight that even certain valid results are dominated by others. That is, they can be discarded early.

We write $a \prec b$ when a dominates b. We will arrange our domination relation such that

1. Layout operators are monotonous with respect to domination. Consequently, for any document context

   $\mathsf{ctx} :: \mathsf{Doc}\ d \Rightarrow d \to d$, if $a \prec b$ then $\mathsf{ctx}\ a \prec \mathsf{ctx}\ b$
2. If $a \prec b$, then a is at least as frugal as b.

Together, these properties mean that we can always discard dominated layouts from a set, as we could discard invalid ones. Indeed, we have:

Theorem 1 (domination)**.** *For any context* ctx*, we have*

$$a \prec b \Rightarrow \mathsf{height}\ (\mathsf{ctx}\ a) \leq \mathsf{height}\ (\mathsf{ctx}\ b)$$

Proof. By composition of the properties 1. and 2. □

We can concretize the above abstract result by defining our domination relation and proving its properties 1. and 2. Our domination relation is a partial order (a reflexive, transitive and antisymmetric relation), and thus make it an instance of the following class:

```
class Poset a where
  (≺) :: a → a → Bool
```

The order that we use is the intersection of ordering in all dimensions: if layout a is shorter, narrower, and has a narrower last line than layout b, then a dominates b.

$$\begin{aligned} \mathbf{instance}\ \mathsf{Poset}\ M\ \mathbf{where} \\ m_1 \prec m_2 = \mathsf{height}\ m_1 &\leq \mathsf{height}\ m_2\ \&\& \\ \mathsf{maxWidth}\ m_1 &\leq \mathsf{maxWidth}\ m_2\ \&\& \\ \mathsf{lastWidth}\ m_1 &\leq \mathsf{lastWidth}\ m_2 \end{aligned}$$

The second desired property is a direct consequence of the definition. The first one is broken down into the two following lemmas:

Lemma 4 (flush is monotonic)**.** *if*

$$m_1 \prec m_2$$

then

$$\mathsf{flush}\ m_1 \prec \mathsf{flush}\ m_2$$

Proof. We have:

$$\begin{aligned} \mathsf{height}\ m_1 &\leq \mathsf{height}\ m_2 \\ \mathsf{maxWidth}\ m_1 &\leq \mathsf{maxWidth}\ m_2 \\ \mathsf{lastWidth}\ m_1 &\leq \mathsf{lastWidth}\ m_2 \end{aligned}$$

and we need to prove the following three conditions

$$\begin{aligned} \mathsf{height}\ (\mathsf{flush}\ m_1) &\leq \mathsf{height}\ (\mathsf{flush}\ m_2) \\ \mathsf{maxWidth}\ (\mathsf{flush}\ m_1) &\leq \mathsf{maxWidth}\ (\mathsf{flush}\ m_2) \\ \mathsf{lastWidth}\ (\mathsf{flush}\ m_1) &\leq \mathsf{lastWidth}\ (\mathsf{flush}\ m_2) \end{aligned}$$

by definition, they reduce to the following inequalities, which are easy consequences of the assumptions.

$$\begin{aligned} \mathsf{height}\ m_1 + 1 &\leq \mathsf{height}\ m_2 + 1 \\ \mathsf{maxWidth}\ m_1 &\leq \mathsf{maxWidth}\ m_2 \\ 0 &\leq 0 \end{aligned}$$

□

Lemma 5 (concatenation is monotonic)**.**

$$\mathbf{if}\ m_1 \prec m_2 \text{ and } m'_1 \prec m'_2 \Rightarrow (m_1 \mathbin{<\!\!>} m'_1) \prec (m_2 \mathbin{<\!\!>} m'_2)$$

Proof. We have:

$$\begin{aligned} \mathsf{height}\ m_1 &\leq \mathsf{height}\ m_2 \\ \mathsf{maxWidth}\ m_1 &\leq \mathsf{maxWidth}\ m_2 \\ \mathsf{lastWidth}\ m_1 &\leq \mathsf{lastWidth}\ m_2 \\ \mathsf{height}\ m'_1 &\leq \mathsf{height}\ m'_2 \\ \mathsf{maxWidth}\ m'_1 &\leq \mathsf{maxWidth}\ m'_2 \\ \mathsf{lastWidth}\ m'_1 &\leq \mathsf{lastWidth}\ m'_2 \end{aligned}$$

and we need to prove the following three conditions:

$$\begin{aligned}
&\mathsf{height}\ \ (\mathsf{m}_1 \mathbin{<\!\!>} \mathsf{m'}_1) \le \mathsf{height}\ \ (\mathsf{m}_2 \mathbin{<\!\!>} \mathsf{m'}_2)\\
&\mathsf{maxWidth}\ (\mathsf{m}_1 \mathbin{<\!\!>} \mathsf{m'}_1) \le \mathsf{maxWidth}\ (\mathsf{m}_2 \mathbin{<\!\!>} \mathsf{m'}_2)\\
&\mathsf{lastWidth}\ \ (\mathsf{m}_1 \mathbin{<\!\!>} \mathsf{m'}_1) \le \mathsf{lastWidth}\ \ (\mathsf{m}_2 \mathbin{<\!\!>} \mathsf{m'}_2)
\end{aligned}$$

Those are by definition equivalent to the following ones:

$$\mathsf{height}\ \mathsf{m}_1 + \mathsf{height}\ \mathsf{m'}_1 \le \mathsf{height}\ \mathsf{m}_2 + \mathsf{height}\ \mathsf{m'}_2 \quad (1)$$

$$\begin{aligned}
&\mathsf{max}\ (\mathsf{maxWidth}\ \mathsf{m}_1)\ (\mathsf{lastWidth}\ \mathsf{m}_1 + \mathsf{maxWidth}\ \mathsf{m'}_1)\\
&\quad \le \mathsf{max}\ (\mathsf{maxWidth}\ \mathsf{m}_2)\ (\mathsf{lastWidth}\ \mathsf{m}_2 + \mathsf{maxWidth}\ \mathsf{m'}_2) \quad (2)
\end{aligned}$$

$$\mathsf{lastWidth}\ \mathsf{m}_1 + \mathsf{lastWidth}\ \mathsf{m'}_1 \le \mathsf{lastWidth}\ \mathsf{m}_2 + \mathsf{lastWidth}\ \mathsf{m'}_2 \quad (3)$$

The first and third inequalities are consequences of the assumptions combined with the monotonicity of $+$. The second inequation can be obtained likewise, with additionally using the monotonicity of max:

$$\mathsf{a} \le \mathsf{b} \wedge \mathsf{c} \le \mathsf{d} \Rightarrow\ \mathsf{max\ a\ c} \le \mathsf{max\ b\ d}$$

□

5.4 Pareto frontier

Filtering out the dominated elements is an operation known as the computation of the Pareto frontier, which can be implemented as follows.

```
pareto :: Poset a ⇒ [a] → [a]
pareto = loop []
  where loop acc []       = acc
        loop acc (x : xs) =
      if any (≺ x) acc
        then loop acc xs
        else loop (x : filter (not . (x ≺)) acc) xs
```

The above function examines elements sequentially, and keeps a pareto frontier of the elements seen so far in the acc parameter. For each examined element x, if it is dominated, then we merely skip it. Otherwise, x is added to the current frontier, and all the elements dominated by x are then removed.

The implementation of the pretty-printing combinators then becomes:

```
type DM = [M]
instance Layout DM where
  xs <> ys = pareto $ concat
             [filter validMeasure [x <> y | y ← ys] | x ← xs]
  flush xs = pareto $ (map flush xs)
  text s = filter validMeasure [text s]
  render = render . minimum
instance Doc DM where
  fail = []
  xs <|> ys = pareto (xs ++ ys)
```

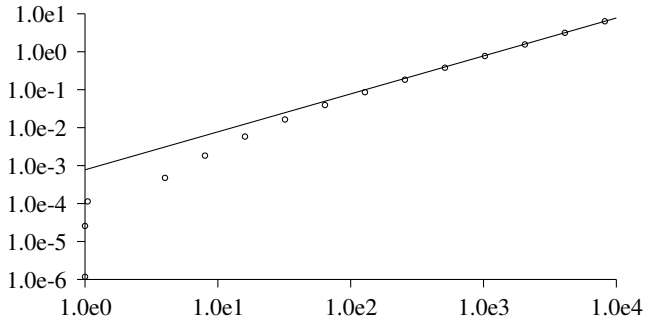
6. Timings

In order to benchmark our pretty printer on large but representative outputs, we have used it to lay out S-expressions representing full binary trees of increasing depth, as generated by the following function:

```
testExpr 0 = Atom "a"
testExpr n = SExpr [testExpr (n − 1), testExpr (n − 1)]
```

The set of layouts were given by using the pretty printer for S-Expressions shown above. The most efficient version of the pretty-printer was used. We then measured the time to compute the length of the layout. (Computing the length is enough to force the computation of the best layout.) This benchmark heavily exercises the disjuction construct. Let us compute the number of choices in printing testExpr n. For each SExpr with two sub-expressions, the printer introduces a choice, therefore the number of choices equal the number of nodes in a binary tree of depth n. Thus, for testExpr n the pretty printer is offered $2^n - 1$ choices, for a total of 2^{2^n-1} possible layouts to consider.

We have run the layout algorithm for n ranging from 1 to 15, and measured the time to perform pretty-printing. The following plot shows the time taken (in seconds) against the *number of lines of output*. (By using the number of lines rather than n, we have a more reasonable measure of the amount of work to perform for each layout task.) The following plot shows the data on a double logarithmic scale (note that several inputs can be printed on a single line):



Precise timings were obtained by using O'Sullivan's *criterion* benchmarking library. While *criterion* provides confidence intervals, they are so thin that they are not visible at this scale.

The plot shows a behaviour that tends to become linear when the output is large enough. For such large inputs approximately 1290.55 lines are laid out per second. We interpret this result as follows. Our pretty-printer essentially considers non-dominated layouts. If the input is sufficiently complex, this approximately means to consider one layout per possible width (80 in our tests) — when the width is given then the length and the width of last line are fixed. Therefore, the amount of work becomes independent of the number of disjuctions present in the input, and depends only on the amount of text to render.

7. Discussion

7.1 Re-pairing with text

Eventually, one might be interested in getting a complete pretty printed output, not just the amout of space that it takes. To do so we can pair measures with full-text layouts, while keeping the

```
instance Poset (M, L) where
  (a, _) ≺ (b, _) = a ≺ b
instance Layout (M, L) where
  (x, x′) <> (y, y′) = (x <> y, x′ <> y′)
  flush (x, x′) = (flush x, flush x′)
  text s = (text s, text s)
  render = render . snd
instance Layout [(M, L)] where
  xs <> ys = pareto $ concat
             [filter (validMeasure . fst) [x <> y | y ← ys] | x ← xs]
  flush xs = pareto $ (map flush xs)
  text s = filter (validMeasure . fst) [text s]
  render = render . minimumBy (compare ‘ on ‘ fst)
```

7.2 Hughes-Style nesting

Hughes proposes a nest conbinator, which indents its argument *unless* it appears on the right-hand-side of a horizontal concatenation. The above semantics are rather involved, and appear difficult to support by a local modification of the framework developed in this paper.

Fortunately, in practice nest is used only to implement the hang combinator, which offers the choice between horizontal concatenation and vertical concatenation with an indentation:

```
hang :: Doc d ⇒ Int → d → d → d
hang n x y = (x <> y) <|> (x $$ nest n y)
```

In this context, nesting occurs on the right-hand-side of horizontal concatenation, and thus its semantics is simple; in fact it can be implemented in terms of the combinators seen so far:

```
nest :: Layout d ⇒ Int → d → d
nest n y = spaces n <> y
  where spaces n = text (replicate n ' ')
```

7.3 Ribbon length

Another subtle feature of Hughes' library is the ability to limit the amount of text on a single line, ignoring the current indentation. The goal is to avoid long lines mixed with short lines. While such a feature is easily added to Hughes or Wadler's greedy pretty printer, it is harder to support as such on top of the basis we have so far.

What we would need to do is to record the length of the 1st line and length of the last line without indentation. When concatenating, we add those numbers and check that they do not surpass the ribbon length. Unfortunately this adds two dimensions to the search space, and renders the final algorithm impossibly slow.

An alternative is to interpret the ribbon length as the maximum size of a self-contained sublayout of one line. Then we just filter out the intermediate results that do not fit the ribbon, as follows:

```
fitRibbon m = height m > 0 || maxWidth m < ribbonLength
  where ribbonLength = 60
valid m = validMeasure m && fitRibbon m
```

This re-interpretation appears to fulfil the original goal as well.

8. Conclusion

Using three informal principles, we have defined what a pretty printer is. We have carefully refined this informal definition to a formal semantics (arguably simpler than that of the state of the art), and derived a reasonably efficient implementation. Along the way, we have demonstrated how to use the standard functional programming methodology.

Acknowledgments

Using the QuickSpec tool, Nicholas Smallbone helped finding a bug in the final implementation: the concatenation operator did not preserve the invariant that lists were sorted.

Appendix

8.1 Raw benchmark runtimes

1	1	(4.445150269261367e-6,4.5280822907269675e-6,4.5916913753665655e-6)
2	1	(2.543191374515173e-5,2.572839925029547e-5,2.5979652200549894e-5)
3	2	(1.1328907189454379e-4,1.1391140432376026e-4,1.1435003581556299e-4)
4	4	(4.7176172121903416e-4,4.744455825885828e-4,4.766890705863499e-4)
5	8	(1.8237314185457591e-3,1.8312912334179166e-3,1.838446045489322e-3)
6	16	(5.783464702298192e-3,5.818189430630023e-3,5.844266885297808e-3)
7	32	(1.62980927266792e-2,1.6435360409749928e-2,1.6529715277458553e-2)
8	64	(3.859506473263461e-2,3.945707022130122e-2,4.1163074183814666e-2)
9	128	(8.533378323034407e-2,8.59952411696479e-2,8.667526753622293e-2)
10	256	(0.1822876224924617 2,0.1834937564396142 4,0.1841060316645407)
11	512	(0.3789056722875015 5,0.3792040734850503,0.3794767752701219 7)
12	1024	(0.7690777575252713,0.7710383657360316,0.7724677260256857)
13	2048	(1.5500714478941475,1.557008440068839,1.5612347437000835)
14	4096	(3.1497701555238202,3.157368636818575,3.1614841082403706)
15	8192	(6.2976283727861215,6.347703961040026,6.374937758508082)

8.2 Proof details

Proof of measure being a Layout-homomorphism

1. measure (a ++ [""])
   ≡ M {maxWidth = maximum ((map length) (a ++ [""]))
   , height = length (a ++ [""]) − 1
   , lastWidth = length $ last $ (a ++ [""])
   }
   ≡ M {maxWidth = maximum ((map length a) ++ [0])
   , height = length a + 1 − 1
   , lastWidth = length ""
   }
   ≡ M {maxWidth = maximum (map length a)
   , height = length a − 1 + 1
   , lastWidth = 0
   }
   ≡ flush M {maxWidth = maximum (map length a)
   , height = length a − 1
   , lastWidth = length $ last $ a
   }
   ≡ flush (measure a)
2. measure (xs <> (y : ys))
   ≡ M {maxWidth = maximum ((map length) (init xs ++ [last xs ++ y] ++ map (indent ++) ys))
   , height = length (init xs ++ [last xs ++ y] ++ map (indent ++) ys) − 1
   , lastWidth = length $ last $ (init xs ++ [last xs ++ y] ++ map (indent ++) ys)
   }
   ≡ M {maxWidth = maximum ((init (map length xs) ++ [length (last xs) + length y] ++ map (\ y → length (last xs) + length y) ys))
   , height = length (init xs) + 1 + length ys − 1
   , lastWidth = last $ ((init (map length xs) ++ [length (last xs) + length y] ++ map (\ y → length (last xs) + length y) ys))
   }
   ≡ M {maxWidth = maximum (init (map length xs) ++ map (\ y → length (last xs) + length y) (y : ys))
   , height = (length xs − 1) + (length (y : ys) − 1)
   , lastWidth = last $ (init (map length xs) ++ map (\ y → length (last xs) + length y) (y : ys))
   }
   ≡ M {maxWidth = maximum [maximum (init (map length xs)), length (last xs) + maximum (map length (y : ys))]
   , height = (length xs − 1) + (length (y : ys) − 1)
   , lastWidth = last $ (map (\ y → length (last xs) + length y) (y : ys))
   }
   ≡ M {maxWidth = maximum [maximum (map length xs), length (last xs) + maximum (map length (y : ys))]
   , height = (length xs − 1) + (length (y : ys) − 1)
   , lastWidth = length (last xs) + last $ (map length (y : ys))
   }
   ≡ M {maxWidth = maximum (map length xs)
   , height = length xs − 1
   , lastWidth = length (last xs)

```
        } <>
      M {maxWidth = maximum (map length (y : ys))
        , height = length (y : ys) − 1
        , lastWidth = length (last (y : ys))
        }
  ≡ measure xs <> measure (y : ys)
```

3.
```
measure (text s)
  ≡ M {maxWidth = maximum (map length [s])
     , height = length [s] − 1
       , lastWidth = length $ last $ [s]}
  ≡ M {maxWidth = length s
       , height = 0
       , lastWidth = length s}
  ≡ text s
```