A Pretty But Not Greedy Printer (Functional Pearl)

JEAN-PHILIPPE BERNARDY, University of Gothenburg, Department of Philosophy, Linguistics and Theory of Science

This paper proposes a 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 sub-documents at will. We argue that our 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.

CCS Concepts: •**Software and its engineering → Functional languages;** •**Mathematics of computing** → Combinatorial optimization;

Additional Key Words and Phrases: Pretty Printing

ACM Reference format:
Jean-Philippe Bernardy. 2017. A Pretty But Not Greedy Printer (Functional Pearl). *PACM Progr. Lang.* 1, 1, Article 6 (September 2017), 22 pages.
DOI: 10.1145/3110250

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 members of the functional programming community to showcase good style. Proeminent examples include the pretty printer of Hughes [1995], which remains an influential example of functional programming design, and that of Wadler [2003] which was published as a chapter in a book dedicated to the "fun of programming".

In addition to their aesthetic and pedagogical value, the pretty printers of Hughes and Wadler are practical implementations. Indeed, they 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]. An ocaml implementation[3] of Wadler's design also exists.

While this paper draws much inspiration from the aforementioned landmark pieces of work in the functional programming landscape, my goal is slightly different to that of Hughes and Wadler. Indeed, they aim first and foremost to demonstrate general principles of functional programming development, with an emphasis on the efficiency of the algorithm. Their methodological approach is to derive a *greedy* algorithm from a functional specification. In the process, they give themselves some leeway as to what they accept as pretty outputs (see Sec. 3.1). In contrast, my primary goal is

[1]https://hackage.haskell.org/package/pretty
[2]https://hackage.haskell.org/package/wl-pprint
[3]https://gallium.inria.fr/~fpottier/pprint/doc

to produce *the prettiest output*, at the cost of efficiency. Yet, the final result is reasonably efficient (Sec. 7).

Let us specify the desired behavior 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 appropriate use of layout, to make it easy for a human to recognize the hierarchical organization 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 optimization problem, and derive a program which solves it efficiently enough for practical purposes.

Before diving into the details, let us pose a couple of methodological points. First, Haskell is used throughout this paper in its quality of *lingua franca* of functional programming pearls — yet, we make no essential use of laziness. Second, the source code for the paper and benchmarks, as well as a fully fledged pretty printing library based on its principles is available online: https://github.com/jyp/prettiest. A Haskell library based on the algorithm developed here is available as well [4].

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, which can either be atom or a list of S-Expressions. They can be represented in Haskell 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, Legibility and Frugality. 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, the elements should be either concatenated horizontally, or aligned vertically. (Even though there are other possible choices, ours is sufficient for illustrative purposes.) For example, the legible layouts of the abcd S-Expression defined above would be either

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

or

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

[4] https://hackage.haskell.org/package/pretty-compact

And thus, LEGIBILITY will interact in non-trivial ways with FRUGALITY and LEGIBILITY.

In general, a pretty printing library must provide the means to express the set of legible layouts: it is up to the user to instantiate 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': we can contatenate documents either vertically ($$) or horizontally (<>), as well as embed raw text and chose between layouts (</>) — but we lack a dedicated flexible space insertion operator (<+>). We give a formal definition of those operators in Sec. 4, but at this stage we keep the implementation of documents abstract. We do so 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; horizontal concatenation with a fixed 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 = foldDoc ($$)
hsep = foldDoc (<+>)
foldDoc :: Doc d ⇒ (d → d → d) → [d] → d
foldDoc _ []     = empty
foldDoc _ [x]    = x
foldDoc f (x : xs) = f x (foldDoc f xs)
```

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 straightforward:

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

3 SEMANTICS (INFORMALLY)

The above API provides a syntax to describe layouts. The next 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?

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

((abcde ((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))))
```

Fig. 1. The expression testData pretty-printed on 80 columns.

Let us use an example to pose the question in concrete terms, and outline why neither Wadler's nor Hughes' answer is satisfactory for our purposes. Suppose that we want to pretty-print the following S-Expr (which is specially crafted to demonstrate general shortcomings of 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 by assumption we would like elements inside an S-Expr to be either aligned vertically or concatenated horizontally (for Legibility), and that the second option should be preferred over the first (for Frugality), as long as the text fits within the page width (for Visibility). More precisely, the three principles demand the output with the smallest number of lines which still fits on the page among all the legible outputs described above. Thus on a 80-column-wide page, they demand the output displayed in Fig. 1 and on a 20-column-wide page, they demand the following output (the first line is not part of the output, but it helps by showing column numbers):

```
12345678901234567890

((abcde ((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))))
```

Yet, neither Hughes' nor Wadler's library can deliver those results.

3.1 The limitations of Hughes and Wadler

Let us take a moment to see why. On a 20-column page and using Hughes' library, we would get the output shown in Fig. 2 instead. That output uses much more space than necessary, violating Frugality. Why is that? Hughes states that "it would be unreasonably inefficient for a pretty-printer to 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. Such a waste occurs in many real examples: any optimistic fitting on an early line may waste tremendous amount of space later on.

```
12345678901234567890

((abcde ((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))))
```

Fig. 2. The expression testData pretty-printed using Hughes' library.

```
12345678901234567890

((abcde
  ((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))))
```

Fig. 3. The expression testData pretty-printed using Wadler's library.

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. In other words, Wadler's library lacks the capability to express that a multi-line 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, even the best pretty printer written using Wadler's library can only produce the output shown in Fig. 3. The result does not look too bad — but there is

a spurious line break after the atom `abcde`. 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. Suppose that 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]
```

Quite reasonably, we hope to obtain the following result, which puts the list to the right of the `expression`, clearly showing that the list is an argument of `expression`, and thus properly respecting Legibility:

```
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 and to the left of the equal sign is bad, because it needlessly obscures the structure of the program; Legibility is not respected. The the lack of a combinator for relative indentation is a serious drawback[5]. In fact, Leijen's implementation of Wadler's design (wl-print), *does* feature an alignment combinator. However, as Hughes' does, Leijen's uses a greedy algorithm, and thus suffers from the same issue as Hughes' library.

In summary, we have to make a choice between either respecting the three principles of pretty printing, or providing 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 formalization 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 (FORMALLY)

4.1 Layouts

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

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

[5]The work of Swierstra and Chitil [2009] suffers from the same drawback.

```
text :: Layout l ⇒ String → l
(<>) :: Layout l ⇒ l → l → l
($$) :: Layout l ⇒ l → l → l
```

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 shall simply use the type of lists, trusting the reader to remember the invariant of non-emptiness.

```
type L = [String]
```

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

```
render :: L → String
render = intercalate "\n"
```

where intercalate can be defined as follows:

```
intercalate :: String → [String] → String
intercalate x []       = []
intercalate x (y : ys) = y ++ x ++ intercalate ys
```

Embedding a string is thus immediate:

```
text :: String → L
text s = [s]
```

The interpretation of vertical concatenation ($$) requires barely more thought: it suffices to concatenate the input lists.

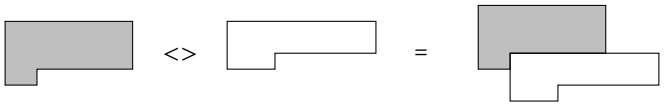
```
($$) :: L → L → L
xs $$ ys = xs ++ ys
```

The only potential difficulty is to figure out the interpretation of horizontal concatenation (<>). We follow the advice provided by Hughes [1995]: "translate the second operand [to the right], so that its first character abuts against the last character of the first operand". For example:

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

=   xxxxxxxxxxxxx
    xxxxxxxxx
    xxxxxxxxxxxx
    xxxxxxyyyyyyyyyyyyyyyyyyyy
          yyyyyyyyyyyyyyyyyyyyyyyyyyyy
          yyyy
```

Or, diagrammatically:



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 :: String
        x = last xs
        n = length x
        indent = replicate n ' '
```

We take a quick detour to refine our API a bit. Indeed, as 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 ++ [""]
```

Vertical 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, we 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[6]):

(1) Layouts form a monoid, with operator (<>) and unit empty[7]:

```
propLeftUnit :: (Doc a, Eq a) ⇒ a → Bool
propLeftUnit a = empty <> a ≡ a

propRightUnit :: (Doc a, Eq a) ⇒ a → Bool
propRightUnit a = a <> empty ≡ a

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

(2) text is a monoid homomorphism:

[6]These properties can be (and were) checked when properly monomorphized using either of the concrete implementations provided later. The same checks were performed for all properties stated in the paper.
[7]recall empty = text ""

```
propTextAppend s t = text s <> text t ≡ text (s ++ t)
propTextEmpty      = empty ≡ text ""
```

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

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

One might expect this law to hold instead:

```
flush 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 on the right-hand side.

4.2 Choice

We proceed to extend the API with choice between layouts, yielding the final API to specify legible documents. 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 straight away. Documents are interpreted as a set of layouts. We implement sets as lists, and we will take care not to depend on the order and number of occurrences.

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

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

Consequently, disjunction is associative.

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

We simply lift the layout operators idiomatically [McBride and Paterson, 2007] over sets: elements in sets are treated combinatorially.

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

Consequently, concatenation and flush distribute over disjunction:

```
propDistrL :: (Doc a, Eq a) ⇒ a → Bool
propDistrL a = (a <|> b) <> c ≡ (a <> c) <|> (b <> c)
propDistrR :: (Doc a, Eq a) ⇒ a → Bool
propDistrR a = c <> (a <|> b) ≡ (c <> a) <|> (c <> b)
propDistrFlush :: (Doc a, Eq a) ⇒ a → a → Bool
propDistrFlush a b = flush (a <|> b) ≡ flush a <|> flush b
```

4.3 Semantics

We can finally define formally what it means to render a document. We wrote above that the prettiest layout is the solution of the optimization problem given by combining all three principles. Namely, to pick a most frugal layout among the visible ones:

```
render = render .
         mostFrugal .
         filter visible
```

Note that the call to render in the above snippet invokes the implementation of the L instance. The rest of the above definition breaks down as follows. Visibility is formalized by the visible function, which states that all lines must fit on the page:

```
visible :: L → Bool
visible xs = maximum (map length xs) ≤ pageWidth
pageWidth = 80
```

Frugality is formalized by the mostFrugal function, which picks a layout with the least number of lines:

```
mostFrugal :: [L] → L
mostFrugal = minimumBy size
  where size = compare ‘ on ‘ length
```

Legibility is realized by the applications-specific set of layouts, specified by the API of Sec. 2, which comes as an input to render.

One may expect that disjunction should also be commutative. However, the implementation of mostFrugal only picks *one* of the most frugal layouts. That is fine, because 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:

```
propDisjCommut :: Doc a ⇒ a → a → Bool
propDisjCommut a b = 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, and thus we can implement the pretty printing of an S-Expr as follows:

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

Running showSExpr on our example (testData) may eventually yield the output that we demanded in Sec. 3. But one should not expect to see it any time soon. Indeed, while the above semantics provides an executable implementation, it is impracticably 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 $O(2^n)$.

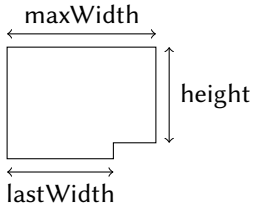
5 A MORE EFFICIENT IMPLEMENTATION

The next chunk of work is to transform the above, clearly correct but inefficient implementation to a functionally equivalent, but efficient one. To do so we need two insights.

5.1 Measures

The first insight is that it is not necessary to fully construct layouts to calculate their size: 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.

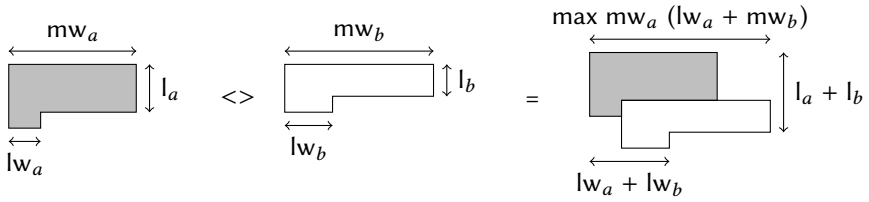
The only parameters that matter are the maximum width of the layout, the width of its last line and its height (and, because layouts cannot be empty and it is convenient to start counting from zero, we do not count the last line):



In code:

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

Astute readers may have guessed the above parameters 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:

```
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, thereby completing the class instance:

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

The correctness of the Layout M instance relies on intuition, and a proper reading of the concatenation diagram. This process being informal, we must 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). This homomorphism property can be spelled out as the following three laws:

Lemma 5.1. *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. □

Using the measure, we can check that a layout is fully visible simply by checking that maxWidth is small enough:

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

Having properly refined the problem, and continuing to ignore the detail of actually rendering the text, we may proceed to give a fast implementation of the pretty printer.

5.2 Early filtering out invalid results

The first optimization 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 de-construction preserves validity: the validity of a document implies the validity of its part.

Lemma 5.2. *de-construction preserves validity. The following two implications hold:*

```
valid (a <> b) ⇒ valid a ∧ valid b
valid (flush a) ⇒ valid a
```

Proof. We prove the two parts separately:

(1) valid (a <> b)
⇒ maxWidth (a <> b) < pageWidth
⇒ max (maxWidth a) (lastWidth a + maxWidth b) ≤ pageWidth
⇒ maxWidth a < pageWidth ∧ lastWidth a + maxWidth b ≤ pageWidth
⇒ maxWidth a < pageWidth ∧ maxWidth b ≤ pageWidth
⇒ valid a ∧ valid b

(2) valid (flush a) ⇒ maxWidth (flush a) ≤ pageWidth
⇒ maxWidth a ≤ pageWidth
⇒ valid a

□

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

Theorem 5.3. *Invalid layouts cannot be fixed*
not (valid a) ⇒ not (valid (a <> b))
not (valid b) ⇒ not (valid (a <> b))
not (valid a) ⇒ not (valid (flush a))

Proof. By contrapositive of Lem. 5.2 □

5.3 Pruning out dominated results

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

We write a ≺ b when a dominates b. We will arrange our domination relation such that

(1) Layout operators are monotonic with respect to domination. Consequently, for any document context ctx :: Doc d ⇒ d → d,
if a ≺ b then ctx a ≺ ctx b
(2) If a ≺ 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 5.4. *(Domination) For any context* ctx*, we have*
a ≺ b ⇒ height (ctx a) ≤ height (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 we can 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.

```
instance Poset M where
  m1 ≺ m2 = height     m1 ≤ height     m2 &&
            maxWidth m1 ≤ maxWidth m2 &&
            lastWidth  m1 ≤ lastWidth  m2
```

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

Lemma 5.5. flush *is monotonic.*
if

$m_1 \prec m_2$

then

flush $m_1 \prec$ flush m_2

Proof. By definition, the assumption expands to

height $m_1 \leq$ height m_2
maxWidth $m_1 \leq$ maxWidth m_2
lastWidth $m_1 \leq$ lastWidth m_2

similarly, the conclusion that we aim to prove expands to the following three conditions

height (flush m_1) $\leq$ height (flush m_2)
maxWidth (flush m_1) $\leq$ maxWidth (flush m_2)
lastWidth (flush m_1) $\leq$ lastWidth (flush m_2)

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

height $m_1 + 1 \leq$ height $m_2 + 1$
maxWidth $m_1 \leq$ maxWidth m_2
$0 \leq 0$

□

Lemma 5.6. *concatenation is monotonic*

if $m_1 \prec m_2$ and $m'_1 \prec m'_2 \Rightarrow (m_1 \diamond m'_1) \prec (m_2 \diamond m'_2)$

Proof. Each of the assumptions expand to three conditions. Thus we have:

height $m_1 \leq$ height m_2
maxWidth $m_1 \leq$ maxWidth m_2
lastWidth $m_1 \leq$ lastWidth m_2
height $m'_1 \leq$ height m'_2
maxWidth $m'_1 \leq$ maxWidth m'_2
lastWidth $m'_1 \leq$ lastWidth m'_2

and similarly we need to prove the following three conditions to obtain the conclusion:

height $(m_1 \diamond m'_1) \leq$ height $(m_2 \diamond m'_2)$
maxWidth $(m_1 \diamond m'_1) \leq$ maxWidth $(m_2 \diamond m'_2)$
lastWidth $(m_1 \diamond m'_1) \leq$ lastWidth $(m_2 \diamond m'_2)$

These are respectively equivalent to the following ones, by definition:

height m_1 + height $m'_1 \leq$ height m_2 + height m'_2
max (maxWidth m_1) (lastWidth m_1 + maxWidth m'_1)
  $\leq$ max (maxWidth m_2) (lastWidth m_2 + maxWidth m'_2)
lastWidth m_1 + lastWidth $m'_1 \leq$ lastWidth m_2 + lastWidth m'_2

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:

$$a \leq b \wedge c \leq d \Rightarrow \max\ a\ c \leq \max\ b\ d$$

□

5.4 Pareto frontier

We know by now that in any set of possible layouts, it is sufficent to consider the subset of non-dominated layouts. This subset is known as the Pareto frontier [Deb et al., 2016] and has the following definition.

Definition 5.7. Pareto frontier $Pareto(X) = \{x \in X \mid \neg\exists y \in X. x \neq y \wedge y \prec x\}$

When sets are represented as lists without duplicates, the Pareto frontier can be computed 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 loop 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 valid [x <> y | y ← ys] | x ← xs])
  flush xs = pareto (map flush xs)
  text s = filter valid [text s]
  render = render . minimum
instance Doc DM where
  fail = []
  xs <|> ys = pareto (xs ++ ys)
```

The above is the final, optimized version of the layout-computation algorithm.

6 ADDITIONAL FEATURES

To obtain a complete library from the above design, one should pay attention to a few more points that we discuss in this section.

6.1 Re-pairing with text

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

```
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 (valid . fst) [x <> y | y ← ys] | x ← xs]
  flush xs = pareto $ (map flush xs)
  text s = filter (valid . fst) [text s]
  render = render . minimumBy (compare `on` fst)
instance Doc [(M, L)] where
  fail = []
  xs <|> ys = pareto (xs ++ ys)
```

6.2 Hughes-Style nesting

Hughes proposes a nest combinator, 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 vertical concatenation, and thus its semantics can be simplified. In fact, in the context of hang, it can be implemented easily in terms of the combinators provided so far:

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

6.3 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 first 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 method adds two dimensions to the search space, and slows the final algorithm to impractical speeds.

An alternative approach to avoid too long lines is to interpret the ribbon length as the maximum size of a self-contained sublayout fitting on a single line. This interpretation can be implemented efficiently, by filtering out intermediate results that do not fit the ribbon. This can be done be re-defining valid as follows:

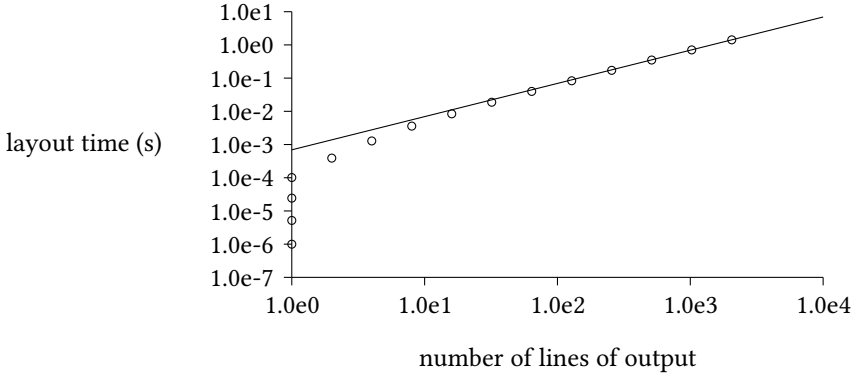


Fig. 4. Layout times for full trees.

```
fitRibbon m = height m > 0 || maxWidth m < ribbonLength
valid′ m = valid m && fitRibbon m
```

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

7 PERFORMANCE TESTS

Having optimized our algorithm as best we could, we turn to empirical test to evaluate its performance. Our benchmarking tool is O'Sullivan's *criterion* benchmarking library, which provides precise timings even for operations lasting less than a microsecond. All benchmarks ran on a single core of an Intel Xeon E5-2640 v4, using GHC 8.0.

7.1 Behaviour at scale

In order to benchmark our pretty printer on large outputs, we have used it to lay out full binary trees and random trees, represented as S-Expressions. The set of layouts were computed using the pretty printer for S-Expressions shown above. The most efficient version of the pretty-printer (shown at the end of Sec. 5) was used. Then we then measured the time to compute the length of the best layout. Indeed, computing the length is enough to force the computation of the best layout. The results are displayed in plots which uses a double logarithmic scale and show the time taken against the *number of lines of output*. By using the number of lines (rather than, say, the depth of the tree), we have a more reasonable measure of the amount of work to perform for each layout task.

Full trees. S-expressions representing full binary trees of increasing depth were generated by the following function:

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

Pretty-printing the generated S-expression heavily exercises the disjunction construct. Indeed, for each S-Expressions with two sub-expressions, the printer introduces a choice, therefore the number of choices is equal to 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 16, and obtained the results shown in Fig. 4.

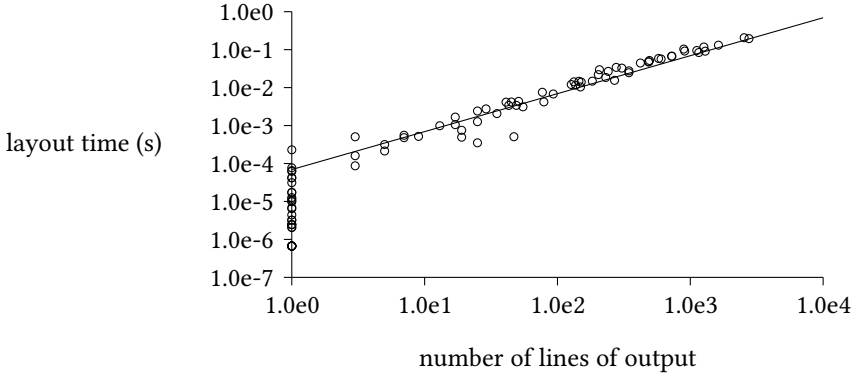


Fig. 5. Layout times for random trees.

While *criterion* provides confidence intervals, they are so thin that they are not visible at the scale of the plots, thus we have not attempted to render them. We observe that inputs for $n \in [0, 1, 2, 3]$ can all be printed on a single line, and thus these data points should be dismissed.

Otherwise, the plot shows a behavior that tends to become linear when the output is large enough.

For such large inputs approximately 1444.27 lines are laid out per second. (Data points exhibiting this speed lay on the straight line which we overlaid to the plot.)

We interpret this result as follows. Our pretty-printer essentially considers non-dominated layouts. If the input is sufficiently complex, this means to consider approximately 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, for sufficiently large outputs the amount of work becomes independent of the number of disjunctions present in the input, and depends only on the amount of text to render.

Random trees. One may wonder if the effect that we observe is not specific to full trees. To control this hypothesis we ran the same experiment on 100 random S-expressions of exponentially increasing length. These S-expressions were generated by picking random Dyck words of a certain length (using the randomDyck function shown below) and then interleaving the obtained parentheses with atoms.

```
randomDyck maxLen = go 0 0 where
  go opened closed
    | closed ≥ maxLen = return []
    | opened ≥ maxLen = close
    | closed ≥ opened = open
    | otherwise = do
      b ← randomIO
      if b then open else close
    where open = (Open :) <$> go (1 + opened) closed
          close = (Close :) <$> go opened (1 + closed)
```

We obtained the results shown in Fig. 5. The results corroborate those obtained for full trees: the observed running time is proportional to the length of the output. Furthermore the layout speed

Table 1. Pretty-printing times (in milliseconds) for typical data, using various libraries

Input	Ours	Wadler-Leijen	Hughes-PJ
JSON 1k	9.7	1.5	3.0
JSON 10k	145.5	14.8	30.0
XML 1k	20.0	3.2	11.9
XML 10k	245.0	36.1	192.0

for random trees is roughly 10 times that of full trees; the straight line corresponding to this speed is shown for reference on the plot.

One may wonder how the elimination of dominated outputs impacts the performance. In fact, in this configuration the algorithm has an exponential behavior, and thus our test machine ran out of memory even for relatively simple outputs. Thus we could not produce more than four useful data points, and thus omitted the corresponding plot.

7.2 Tests for full outputs and typical inputs

Even though the asymptotic behavior of the optimized algorithm is linear, one may wonder if its absolute performance is satisfactory for typical pretty-printing tasks. Thus we evaluated a complete pretty-printing task, including not only the selection of the layout but its actual printing. We did so using our complete library[8]. For reference, we performed the same tests using the Wadler-Leijen library and the Hughes-Peyton Jones library. The inputs were typical JSON and XML files 1k and 10k lines. The JSON data was generated by the tool found at http://www.json-generator.com/, which aims to generate typical JSON files. The XML files are typical database-like files with a nesting depth of 5. The results are displayed in Table 1. We observe that our library is capable of outputting roughly 70.000 lines of pretty-printed JSON per second. Its speed is roughly 40.000 lines per second for XML outputs. This performance is acceptable for many applications, and makes our library about ten times as slow as that of Wadler-Leijen. The Hughes-Peyton Jones library stands in between.

8 CONCLUSION

As Bird and de Moor [1997], Wadler [1987], Hughes [1995] and many others have argued, program calculation is a useful tool, and a strength of functional programming languages, with a large body of work showcasing it. Nevertheless, I had often wondered if the problem of pretty-printing had not been contrived to fit the mold of program calculation, before becoming one of its paradigmatic applications. In general, one could wonder if program calculation was only well-suited to derive greedy algorithms.

Thus I have taken the necessary steps to put my doubts to rest. I have taken a critical look at the literature to re-define what pretty-printing means, as three informal principles. I have carefully refined this informal definition to a formal semantics (arguably simpler than that of the state of the art). I avoided cutting any corner and went for the absolute prettiest layout. Doing so I could not obtain a greedy algorithm, but still have derived a reasonably efficient implementation. In the end, the standard methodology worked well: I could use it from start to finish.

acknowledgements. Some of the work described in this paper was carried out while the author was employed by Chalmers University of Technology. Facundo Domingez, Atze van der Ploeg, Arnaud Spiwack and anonymous ICFP reviewers provided useful feedback on drafts of this paper.

[8] https://hackage.haskell.org/package/pretty-compact

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.

9 ADDENDUM

After this paper settled to a final version, Anton Podkopaev pointed to us that Azero and Swierstra [1998] proposed pretty printing combinators with the same semantics as that presented here (no compromise between greediness and Frugality). However their implementation had exponential behavior. Podkopaev and Boulytchev [2014] took that semantics and proposed a more efficient implementation, which computes for every document its minimal height for every pair of maxWidth and lastWidth. Their strategy is similar to mine, with the following tradeoff. In this paper I do not keep track of every witdth and lastWidth, but only of those which lie on the pareto frontier. In return I have to pay a larger constant cost to sieve through intermediate results. Yet I conjecture that for non-pathological inputs the asymptotic complexities for both algorithms are the same.

REFERENCES

Pablo R. Azero and S. Doaitse Swierstra. 1998. Optimal Pretty Printing Combinators. (1998). Submitted to ICFP 1998.

Richard Bird and Oege de Moor. 1997. *Algebra of programming*. Prentice-Hall, Inc. http://portal.acm.org/citation.cfm?id=248932

Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. 2016. Multi-objective optimization. In *Decision Sciences: Theory and Practice*. CRC Press, 145–184.

John Hughes. 1995. The Design of a Pretty-printing Library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, 53–96. http://portal.acm.org/citation.cfm?id=734154

Conor McBride and Ross Paterson. 2007. Applicative programming with effects. *Journal of Functional Programming* 18, 01 (2007), 1–13. DOI: http://dx.doi.org/10.1017/S0956796807006326

Anton Podkopaev and Dmitri Boulytchev. 2014. Polynomial-Time Optimal Pretty-Printing Combinators with Choice. In *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. 257–265. DOI: http://dx.doi.org/10.1007/978-3-662-46823-4_21

S. Doaitse Swierstra and Olaf Chitil. 2009. Linear, bounded, functional pretty-printing. *Journal of Functional Programming* 19, 01 (2009), 1–16. DOI: http://dx.doi.org/10.1017/S0956796808006990

Philip Wadler. 1987. A critique of Abelson and Sussman or why calculating is better than scheming. *ACM SIGPLAN Notices* 22, 3 (1987), 83–94.

Philip Wadler. 2003. *A prettier printer*. Palgrave MacMillan, 223–243.

APPENDIX

9.1 Proof details

Proof of measure being a Layout-homomorphism.

```
(1)  measure (flush a)
       ≡ 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
```