An implementation-oriented semantics of Wadler's pretty-printing combinators

Sam Kamin
Oregon Graduate Institute
DRAFT

June 10, 1998

Abstract

We present a new implementation of Wadler's pretty-printing combinators. In this implementation, the combinators explicitly manipulate document values, rather than being abstract syntax operators. The purpose of this paper is to advance a point of view about domain modelling and language embedding: that the first and most essential step in domain modelling is finding appropriate values to represent objects in the domain. In this case, we have found what appears to be the appropriate notion of "document value" (to the extent that Wadler's combinators represent an appropriate notion of "document"). We argue that proceeding from this point of view provides greater insight into the domain, as well as a more efficient implementation of the combinators.

1 Introduction

In [1], John Hughes presented a case study in the embedding of a domain in a functional language. The domain was that of pretty-printing — displaying a data type such as an abstract syntax tree neatly with appropriate indentations and line breaks — and the case study presented a set of operations, or *combinators*, for the domain, and their implementation. Simon Peyton Jones [2] added a modified version of Hughes's combinators to the libraries for the major Haskell implementations (GHC and Hugs). Phil Wadler [3, 4] presented a different set of pretty-printing combinators, with somewhat more satisfactory algebraic properties.

The presentations by Hughes and Wadler are distinguished by the method used to develop the implementations. In each case, algebraic properties were deduced from an abstract (not efficiently implementable) version of the combinators. The combinators themselves were simply constructors of an abstract syntax tree, but the deduced algebraic properties were then used to develop transformations of the abstract syntax tree. These transformations turned the abstract syntax trees into a form from which the "answer" could be easily read off. Thus, the development process gave good assurance of the correctness of the implementation, and the implementation was at the same time very efficient.

In this paper, I give new definitions of Wadler's combinators, starting from first principles. I view this implementation as being more conventional than Wadler's or Hughes's, in that it consists of a set of values that carry the information needed to pretty-print documents using these combinators, with the combinators themselves being functions over those values. The algebraic properties do not appear in the implementation in any explicit form (although in a broader sense they strongly influenced the overall development), but they are derivable as theorems.

We will present the implementation, make some comments about its efficiency, and then discuss its algebra. We begin, however, with a discussion of why we consider this to be a worthwhile topic of study.

2 Why pretty-printing combinators?

Why bother to consider such a simple domain in so much detail? Actually, the domain is not as simple as it looks, as Hughes and Wadler, among others, would likely agree. But, on the other hand, it is not of particularly great interest in itself. Furthermore, we have not contributed anything to the development of pretty-printing *per se*, since we have simply re-implemented a set of combinators that Wadler has already implemented efficiently. So, why bother?

One answer is that Wadler's combinators provide an efficient implementation in a lazy language, but not in an eager one. Thomas Nordin has translated Wadler's implementation to Standard ML, and the results seem to be satisfactory, but there are some examples for which the combinators are still inefficient.[1] The combinators presented here seem to be the most efficient set currently existing for Wadler's combinators in Standard ML.

However, the real motivation is to make a point about domain modelling. There seems to be some controversy in the functional programming community over how to embed domains in functional languages, with two opposing approaches that might be termed *operational* and *denotational*. I am myself a strong partisan of the denotational approach. The point of the current exercise for me is to explain the differences and show some advantages of the approach I favor.

In the operational approach, combinators are just abstract syntax operators, and the desired effect of the combinators is obtained by an operation — say, `eval` — that walks over the tree recursively in the usual way. In the denotational approach, a set of values is chosen to represent objects in the domain, and the combinators manipulate these values; the `eval` function does little or nothing.

In my opinion, the denotational approach is the one that produces the best results in the long run, because it focusses on the right issues and leads to a better understanding of the domain. Furthermore, it can give insight not only into the abstract behavior of the operations of the domain, but into their operational behavior as well. Indeed, I would assert as a motto the following:

Domain modelling	=	finding appropriate values to represent objects in the domain

To make the discussion more concrete, here are two definitions of a simple language. The one on the left is operational, the one on the right denotational:

[1]It seems to be characteristic of pretty-printing combinators that they are either very fast or very slow; Peyton Jones reported that his first version of Hughes's combinators was extremely slow for certain examples. Thomas and I have repeatedly observed this phenomenon as well.

```
datatype Exp =
    Int of int
  | Var of string
  | Plus of Exp*Exp
  | Let of string*Exp*Exp

fun eval (E:Exp) =
  let fun ev rho (Int i) = i
    | ev rho (Var s) = rho s
    | ev rho (Plus(e,e')) =
        (ev rho e)+(ev rho e')
    | ev rho (Let(s,e,e')) =
        ev (modify rho s (ev rho e)) e'
  in ev emptyenv E
  end;
```

```
type ExpValue = Env -> Int

fun Int i = fn rho => i;
fun Var s = fn rho => rho s;
fun Plus (e,e') =
    fn rho => (e rho) + (e' rho);
fun Let (s,e,e') =
    fn rho => e' (modify rho s (e rho))

fun eval e = e emptyenv;
```

In the end, these approaches will *merely* produce different implementations of the same combinators. Yet there is a philosophical difference in how the domain embedding is approached, and important practical consequences that flow from it. The philosophy of the denotational approach is that the type of value used to model the domains is the key point to be resolved; that this value somehow embodies the essential elements of the domain, insofar as the domain is reflected in the choice of combinators, and that it is therefore the central focus in any attempt to develop a theory of this, and related, domains. This has been the history of the study of programming languages; although the denotational approach to the study of general purpose languages is in some ways losing ground nowadays to more operational approaches, its emphasis on viewing programs as expressions denoting mathematical values underlies much of the modern understanding of languages.

The denotational approach has practical consequences as well:

Efficiency. Though it seems paradoxical, the denotational approach can provide greater insight into the implementation of the domain than the operational approach. The set of values becomes the focus of operational thinking: what is the most efficient way to represent the value space, or, more precisely, what is a good representation of the *definable* elements of the value space? The question can hardly be formulated so clearly in an operational setting.

Our definition of Wadler's pretty-printing combinators is a good example of this. The value space shows some key operational concepts that are only implicit in Wadler's implementation.

Modularity. Once the type of values has been chosen — to be perfectly clear, there may be invariants that need to be respected, representing the definable values of that type — combinator definitions can be given in isolation. With the operational definition, all "meaning" is embodied in one function ("`eval`"), which is where all changes and additions must be made.

Reasoning. When considering the truth of an equivalence $E = E'$, the first question is: *in what context?*

If the combinators are defined denotationally, the context may be small or even non-existent. For example, `x+0` might equal `x`, in some language, because $\lambda\sigma.\sigma(\texttt{x}) + 0$ equals $\lambda\sigma.\sigma(\texttt{x})$ as a matter of simple arithmetic.

In the operational model, on the other hand, expressions are never equal, but only equivalent, under a definition such as: $E \equiv E'$ iff for any "context" $C[\cdot]$, $\texttt{eval}(C[E]) = \texttt{eval}(C[E'])$. Such proofs can be difficult because they require induction on the structure of C.

In practice, the situation is worse, because equivalences are valid only only under a broad assumption about their uses: we can assert $E \equiv E'$ only to the extent that we can be certain that the only uses of E and E' will be as arguments to `eval`, or as arguments to other constructors that will in turn only be used as arguments to `eval`. In other words, the *possibility* of non-abstract uses of expressions makes equivalences difficult to *use*. For example, it makes transformations of separately-compiled code virtually impossible to perform with confidence.

Wadler's and Hughes's implementations of their pretty-printing combinators use an approach in which some algebraic properties become self-evident because they are built into the implementation. In particular, the abstract syntax tree of pretty-printing operators is transformed according to certain rules, such as associativity; then these rules, interpreted as identities, are immediately true. However, this approach has some limitations. For one, this does not answer the objection raised in the previous paragraph; the transformations are not applied until the `eval` function is applied, and until then, all bets are off. Also, algebraic properties that do not appear as rewriting rules are as difficult to prove as in the operational model.

Thirdly, there is a fundamental flaw in the reasoning just given: a transformation rule can be invalidated by another rule that overlaps it. That is, if the abstract syntax tree is transformed by a rule $E \rightarrow E'$, we would normally assume that $E = E'$; but if there is another rule $C[E] \rightarrow E''$, the truth of the first rule might be violated. Thus, in principle, only by an analysis of *all* the transformation rules can the truth of a given algebraic identity be confirmed; by the same token, introduction of a new rule entails the re-proving of all previously-proven identities.

By contrast, in the denotational approach, introduction of a new combinator cannot invalidate known identities, and a change in the definition of a combinator can invalidate only identities involving that combinator (assuming, in both cases, that the new definitions do not violate invariants of the value space).

These are for the most part ancient arguments in favor of denotational semantics as opposed to operational semantics. Our point in reiterating them is to say that they are still valid for embeddings of new domains — domains which are generally much simpler than the domain of general-purpose computing — into functional languages.

3 Wadler's pretty-printing combinators

The pretty-printing combinators given by Wadler in [3] and [4] are slightly different. We implement those given in the more recent [4]. They are:

`text` s. Print s. s is assumed not to contain any newline characters, i.e. to be "flat."

`empty`. Print nothing.

`line`. Print a newline. The first character on the new line should be indented according to the current indent. (Note that the current indent is initially zero; using Wadler's combinators, there is no way to indent the first line of a document, except by printing spaces.)

`nest` k D. Add k to the current indent and then pretty-print D. Again, note that the indent does not take effect until the next time a new line is begun.

D `<>` D'. D followed by D'. The two documents are simply placed one after the other, with the first line of D' joined to the last line of D. The documents are otherwise independent; in particular, any indentation applied to D does not affect D'.

D `<|>` D'. D or D', whichever fits. The question of how to decide between D and D' — that is, what constitutes "fitting" — is one of the key elements of these combinators. Unlike many other pretty-printing combinators — including Hughes's — in Wadler's combinators the decision is based solely on the properties of the *first line* of D.

4 A denotational semantics of Wadler's combinators

An implementation-oriented denotational semantics for a set of combinators can lead to an efficient implementation and at the same time provide insight into the domain. The semantics we give for Wadler's combinators is based on three concepts:

1. Every document has a minimum number of characters that it can add to the current line. We use ω to denote this minimum width.
2. In addition, every document has a minimum number of characters that it can add to the current line, while at the same time *ending* that line. We use η to denote this "minimum width with a newline." (If the document cannot add a newline, we define η to equal ∞.) Since ω is defined in the same way as η but without the additional constraint, we always have $\omega \leq \eta$.
3. The current line is, in a sense, filled from both ends. That is, if a document containing a choice is surrounded by simple text on either side:

   `text` s `<>` $(D$ `<|>` $D')$ `<>` `text` t

   then D will be chosen only if it can fit in the amount of space left over after allowing for s on the left and t on the right. For example,

   ```
   text "a" <> (text "bc" <|> text "d") <> text "e",
   ```

   if pretty-printed on a page of width 3, will print `ade`. Thus, a document is pretty-printed in a "pretty-printing state" that includes the current column as well as the *effective* page width (the width after the full page width has been reduced by whatever will appear on the right). Thus, in the last example, when the choice is made between `text "bc"` and `text "d"`, the current column is 1 (counting from zero), due to the `a` having been printed, and the effective page width is 2, because space must be allowed for the `e`. With only one space left, the `bc` choice must be rejected because it does not fit.

These concepts, which are key to understanding how these combinators work, do not appear explicitly in Wadler's implementation. An important advantage of using model-oriented combinator definitions is that such concepts become explicit.

We use a kind of "Oxford style" of presentation of the combinators. This style was developed for the presentation of denotational definitions, and it seems to work well here.

In the following, $\mathcal{N}$ is the natural numbers, $\mathcal{Z}$ the integers, and $\mathcal{N}_\infty$ the natural numbers together with $+\infty$ (which we write as ∞) and $-\infty$. We define addition on $\mathcal{N} \times \mathcal{N}_\infty$ such that both infinities are preserved, and we define less-than on $\mathcal{N}_\infty \times \mathcal{N}_\infty$ such that, for all $k \in \mathcal{N}$, $-\infty < k < \infty$. (Note that when subtraction is used in the definition of `<>`, it is integer subtraction and not natural number subtraction.) In the way of notation, we use juxtaposition for string concatenation and $|\cdot|$ for string length, and we use McCarthy's conditional notation ($x \to y, z$ for "if x then y else z").

4.1 Domains

A pretty-printing state is a quadruple of integers, representing, respectively, the current indent, the current column, the page width, and the effective page width. A document is a triple containing a minimum width (ω), a minimum line-ending width (η), and a function which, given a pretty-printing state, returns a new ending column and a string.

$\texttt{text}\ s = (|s|, \infty, \lambda(\_, \kappa, \_, \_).(\kappa + |s|, s)$

$\texttt{empty} = \texttt{text ""}$

$\texttt{line} = (0, 0, \lambda(\iota, \_, \_, \_).(\iota, "\backslash n \underbrace{\qquad\quad}_{\iota \text{ times}}"))$

$\texttt{nest}\ k\ (\omega, \eta, F) = (\omega, \eta, \lambda(\iota, \kappa, \pi, \eta).F(\iota + \kappa, \kappa, \pi, \epsilon))$

$$\begin{aligned}
(\omega, \eta, F)\ \texttt{<>}\ (\omega', \eta', F') = (&min(\eta, \omega + \omega'),\ min(\eta, \omega + \eta'),\\
&\lambda(\iota, \kappa, \pi, \eta).\text{let } \epsilon' = max(\pi - \eta', \epsilon - \omega')\\
&\qquad\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon')\\
&\qquad\qquad\qquad (\kappa'', s') = F'(\iota, \kappa', \pi, \epsilon)\\
&\qquad\qquad\quad \text{in } (\kappa'', ss'))
\end{aligned}$$

$$\begin{aligned}
D \text{ as } (\omega, \eta, F)\ \texttt{<|>}\ (\omega', \eta', F') &= (min(\omega, \omega'), min(\eta, \eta'), \lambda\sigma.\text{fits } D\ \sigma \to F\sigma, F'\sigma)\\
&\text{where fits } (\omega, \eta, F)\ (\iota, \kappa, \pi, \epsilon) = (\kappa + \omega \leq \epsilon) \vee (\kappa + \eta \leq \pi)
\end{aligned}$$

Figure 1: Combinator definitions

$$\begin{array}{rclll}
(\omega, \eta, F) \in Doc & = & & \mathcal{N} & \text{– minimum width}\\
& & \times & \mathcal{N}_\infty & \text{– minimum width w/ newline}\\
& & \times & (PPState \to \mathcal{N} \times string) & \text{– print functions}\\
\\
(\iota, \kappa, \pi, \epsilon) \in PPState & = & & \mathcal{N} & \text{– current indent}\\
& & \times & \mathcal{N} & \text{– current column}\\
& & \times & \mathcal{N} & \text{– page width}\\
& & \times & \mathcal{Z} & \text{– effective page width}
\end{array}$$

We further stipulate that $PPState$ includes only tuples $(\iota, \kappa, \pi, \epsilon)$ in which $\epsilon \leq \pi$. It is easy to confirm that this restriction is respected in the combinator definitions in the next section, and that our reasoning about these values in subsequent sections is valid under this definition of $PPState$.[2]

4.2 Combinator definitions

The definitions are presented in their entirety in Figure 1. Here we explain the subtle points in each definition.

The definitions of `text`, `empty`, `line`, and `nest` seem self-evident. Note that since, by assumption, the string argument of `text` contains no newlines, its η component is ∞.

For the composition operator, `<>`, it is important to bear in mind the meaning of the ω and η components of a document. The ω component represents the shortest string that this document can possibly add to the current line. The combined document D `<>` D' can add text to the current line in two ways: It can add text from D *without* a line break, and then add the shortest possible string from D'; the shortest string that can be added in this way has length $\omega + \omega'$. Alternatively, it can add text from D *with* a line break, in which case obviously the text from D' will not go on the current line; the shortest string that can be added in this way has length η.[3] Thus, the minimum amount of text the combined document can add is the

[2]This is cheesy, but other ways of handling this problem seem more complicated than it's worth.

[3]The string from D with length ω may include a line break, but in that case we will have $\eta = \omega$, so that $\eta \leq \omega + \omega'$, and η is the minimum.

lesser of these two quantities.

A similar argument explains the η component of D `<>` D': either D can take the line break, or it can print a string and let D' take the line break.

The third component of D `<>` D' is perhaps the most difficult to understand. The idea here is that when printing D followed by D', D' is going to take up some space before its first newline character (if any); this amount of space will reduce the effective page width available to D. The question is: what is the maximum possible effective page width of D? D' can reduce the page width in two ways: it can print a certain number of non-newline characters — the minimal number being ω' — thus reducing the effective page width for D from ϵ (D''s own effective page width) to $\epsilon - \omega'$; or it can print a certain number of characters followed by a newline — the minimal number being η' — thus giving D an effective page of $\pi - \eta'$ columns; in this case, D''s effective page width is irrelevant, since it applies only to its *last* line. Thus, the effective page width for D is computed as the maximum of these two quantities. D is pretty-printed with this effective page width and D' is pretty-printed with the effective page width of the combined document.

The alternation operator, `<|>`, is fairly easy to understand. The minimum widths of the alternation, with and without newlines, are the minimums of the respective widths of the components. The printing function is asymmetric: if D fits it is printed; otherwise D' is printed. Keep in mind that "fitting" refers only to the first line. D's first line fits if either it can be printed flat without exceeding the effective page width or it can be printed with a line break without exceeding the total page width.

5 Performance

The translation of an abstract syntax tree into pretty-printing combinators can produce an enormous "tree" of combinators. In an eager language, there is no choice but to evaluate each node in this tree. However, these combinators evaluate each combinator application exactly once; each evaluation takes constant time. Furthermore, the third component of any document is applied at most once; each evaluation runs in constant time, except that composition performs string concatenation, which takes time proportional to the sizes of the strings. It is possible to use a "continuation-based" representation of strings in which concatenation takes constant time and the translation to ordinary strings takes time proportional to the entire length of the resulting string; in any case, this string concatenation is unavoidable, no matter how these combinators are implemented.

6 Full abstraction

The importance of full abstraction is that it allows us to treat equalities and inequalities identically: two expressions denoting documents are equivalent if they denote the same value in the domain *Doc*, and they are different — that is, really different, in the sense that in some circumstances they can produce different output — if they denote different values in *Doc*.

Definition A **definable document** is a value of the set $DDoc \subseteq Doc$ defined inductively by:

- `text` $s \in DDoc$ for any string s
- `line` $\in DDoc$
- `nest` $k\ D \in DDoc$, if $D \in DDoc$
- D `<>` D', D `<|>` $D' \in DDoc$, if $D, D' \in DDoc$

- Nothing else is in *DDoc*.

Definition A **context** is a value in $Context \subseteq Doc \rightarrow Doc$. *Context* is the smallest set of functions that includes the functions `nest` k for all k, and the functions $\lambda d.d$`<>`D, $\lambda d.D$`<>`d, $\lambda d.d$`<|>`D, $\lambda d.D$`<|>`d, for all $D \in DDoc$, and is closed under composition.

In other words, contexts are just the functions on documents that can be defined by an expression over the document operations that contains a single "hole."

Definition Documents D and D' are **observationally equivalent**, written $D =_{obs} D'$, if for any context C and any page width $\pi > 0$, $(CD)_3(0, 0, \pi, \pi) = (CD')_3(0, 0, \pi, \pi)$.

We must add one caveat: we assume that spaces added for indentation are different form spaces added in text. For example, we assume `line <> text " a"` is distinguishable from `nest 1 (line <> text "a")`.[4]

Theorem If $D = D'$, then $D =_{obs} D'$.

Definition A set of definitions of the pretty-printing combinators is **fully abstract** if, for any definable documents $D, D' \in DDoc$, $D = D'$ if and only if $D =_{obs} D'$.

Theorem The combinator definitions given in Figure 1 are fully abstract.

Proof Assume $D = (\omega, \eta, F) \neq D' = (\omega', \eta', F')$.

If $\omega < \omega'$, then they are distinguished by the context

$$C \texttt{ = } \lambda d.d \texttt{ <> (("a" <> line) <|> line)}$$

because, given page width $\pi = \omega + 1$, D will have room on its first line for the `a`, while D' will not. (If D' happens to print exactly the same thing on its first line as D does, but with an added `a`, then just change the `a` to `b` in the context.)[5]

If $\eta < \eta'$, then D and D' are distinguished by the same context.

If $F \neq F'$, then there exists at least one state $\sigma = (\iota, \kappa, \pi, \epsilon)$ such that $F\sigma \neq F'\sigma$. Then D and D' are distinguished by the context

$$C \texttt{ = } \lambda\ d.\ \ \texttt{nest}\ \iota\ \texttt{(text}\ s\ \texttt{<>}\ d\ \texttt{<> text}\ t\texttt{)}$$

where $|s| = \kappa$ and $|t| = \pi - \epsilon$. In this case, $(Cd)_3(0, 0, \pi, \pi)$ will apply d_3 — that is, F or F' — to σ, which will produce pairs (κ', s) and (κ'', s'), respectively. Since we must have $s \neq s'$, these two documents will produce observably different output.

We should explain the last sentence of the proof. In the pairs (κ', s) produced by the third components of definable documents, κ' is actually redundant. It is easy to show that κ' is a simple function of s: if s has no newlines, then $\kappa' = \kappa + |s|$; otherwise, κ is the length of the last line of s.

[4] I'm not sure this caveat is necessary. In this case, if not for the caveat, the two documents would simply be equal. I need an example where we would have two distinct elements of *Doc* which always produce the same results if the difference between the two kinds of spaces is ignored, but I can't think of one. Needs more thought.

[5] Needs work. Need to prove that there is actually some connection between the three components of a document, e.g. the document is really able to produce a first line containing ω characters.

7 Algebraic properties

In [3, 4], Wadler lists various algebraic properties of these combinators, upon which his implementation is based. In [3], he gives the following identities:

text "" = empty

text (st) = text s <> text t

nest i $(D$ <> $D')$ = nest i D <> nest i D'

nest i empty = empty

nest $(i+j)$ D = nest i (nest j D)

nest 0 D = D

nest i (text s) = text s

D <> $(D'$ <> $D'')$ = $(D$ <> $D')$ <> D''

D <> empty = D

empty <> D = D

D <|> $(D'$ <|> $D'')$ = $(D$ <|> $D')$ <|> D''

nest i $(D$ <|> $D')$ = nest i D <|> nest i D'

$(D$ <|> $D')$ <> D'' = $(D$ <> $D'')$ <|> $(D'$ <> $D'')$

D <> $(D'$ <|> $D'')$ = $(D$ <> $D')$ <|> $(D$ <> $D'')$

The last three of these are not mentioned in [4]; the last one is, in fact, untrue (we explain why in section 8). Wadler adds two rules in [4] involving the new constant `none`, stating that it is a right and left identity for `<|>`. `none` does not appear to be a "user-level" combinator, as it is not used in any of the examples given in [4]. In any case, we do not have `none`, and it would be hard for us to define it; in particular, it is hard to add a right-identity for `<|>`, because the definition of `<|>` makes it clear that D <|> D' does not equal D if D doesn't fit.

In this section, we show that the properties listed above — all but the last, which is false, and the first, which is true by definition — hold in our model. We then discuss some other algebraic properties, true and false.

The algebraic properties listed above are not sufficient to determine exactly the string produced by each document, since they do not describe how choices are made. In particular, they say nothing about "fitting." (Furthermore, lacking the last identity above, they say nothing about documents that start with a choice operator.) The proofs we give here show that our combinators produce identical output to Wadler's combinators for choice-free documents (assuming Wadler's combinators satisfy all the identities concerning choice-free documents). However, we have not proven the two implementations of these combinators identical. In our experiments, using Thomas Nordin's Standard ML version of Wadler's combinators, we have been unable to find any documents on which they differ.

We first prove an invariant of documents stated earlier:

Theorem All values $(\omega, \eta, F) \in DDoc$ satisfy the invariant $\omega \leq \eta$.

Proof By induction on the construction of the value:

- text s: $|s| \leq \infty$
- line: $0 \leq 0$
- nest k $(\omega, \eta, F) = (\omega, \eta, \ldots)$, and $\omega \leq \eta$ by induction.

- (ω, η, F) `<>` $(\omega', \eta', F') = (min(\eta, \omega{+}\omega'), min(\eta, \omega{+}\eta'), \ldots)$. We need to prove $min(\eta, \omega + \omega') \leq min(\eta, \omega + \eta')$, but this is obvious from the induction hypothesis, $\omega' \leq \eta'$.
- (ω, η, F) `<|>` $(\omega', \eta', F') = (min(\omega, \omega'), min(\eta, , \eta'), \ldots)$. If $\omega \leq \omega'$, we have both $\omega \leq \eta$ (by induction) and $\omega \leq \omega' \leq \eta'$ (by induction). We obtain the result symmetrically if $\omega' \leq \omega$.

In all proofs, we follow the convention that $D = (\omega, \eta, F)$, $D' = (\omega', \eta', F')$, and so on.

Theorem `text` st = `text` s `<>` `text` t.

Proof We proceed component by component:

First component:
$$\begin{aligned}(\texttt{text } s \texttt{ <> } \texttt{text } t)_1 &= min((\texttt{text } s)_2, (\texttt{text } s)_1 + (\texttt{text } t)_1)\\ &= min(\infty, |s| + |t|)\\ &= |s| + |t|\\ &= |st|\\ &= (\texttt{text } st)_1.\end{aligned}$$

Second component:
$$\begin{aligned}(\texttt{text } s \texttt{ <> } \texttt{text } t)_2 &= min((\texttt{text } s)_2, (\texttt{text } s)_1 + (\texttt{text } t)_2)\\ &= min(\infty, |s| + \infty)\\ &= \infty\\ &= (\texttt{text } st)_2.\end{aligned}$$

Third component: We refer to the third components of `text` s and `text` t as F_s and F_t, respectively.[6]

$$(\texttt{text } st)_3 = \lambda(\_, \kappa, \_, \_).(\kappa + |st|, st).$$

$$\begin{aligned}(\texttt{text } s \texttt{ <> } \texttt{text } t)_3 &= \lambda(\_, \kappa, \_, \_).\ \text{let } \epsilon' = \ldots\\ &\qquad\qquad (\kappa', s) = F_s(\_, \kappa, \_, \epsilon')\\ &\qquad\qquad (\kappa'', s') = F_t(\_, \kappa', \_, \epsilon)\\ &\qquad\quad \text{in } (\kappa'', ss')\\ &= \lambda(\_, \kappa, \_, \_).\text{let } (\kappa', s) = (\kappa + |s|, s)\\ &\qquad\qquad (\kappa'', s') = (\kappa + |s| + |t|, t)\\ &\qquad\quad \text{in } (\kappa'', ss')\\ &= \lambda(\_, \kappa, \_, \_).(\kappa + |s| + |t|, st)\\ &= \lambda(\_, \kappa, \_, \_).(\kappa + |st|, st).\end{aligned}$$

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

Proof Trivial. The first two components of the two sides are identical, and the third component ignores the current indent (ι), so is clearly unaffected by the `nest` operation.

For the next two proofs, keep in mind that `empty` $= (0, \infty, \lambda(\_, \kappa, \_, \_).(\kappa, \texttt{""}))$. We will refer to the third component of `empty` as F_ϕ.

Theorem `empty` `<>` $D = D$.

[6] In this and subsequent proofs, we often use underscores (_) for some components of *PPState*'s, when those components are not explicitly used in the function's body. All underscores should be replaced uniformly by appropriate names throughout the function. For example, $\lambda(\_, \kappa, \_, \_).(\_, \kappa', \_, \_)$ should be read as $\lambda(\iota, \kappa, \pi, \epsilon).(\iota, \kappa', \pi, \epsilon)$.

Proof Again, we proceed component-wise. Recall that, by convention, $D = (\omega, \eta, F)$.
First component: $(\texttt{empty <> } D)_1 = min(\infty, 0 + \omega) = \omega$.
Second component: $(\texttt{empty <> } D)_2 = min(\infty, 0 + \eta) = \eta$.

Third component: $(\texttt{empty <> } D)_3 =$

$$\begin{aligned}
&\lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = \ldots \\
&\qquad\qquad (\kappa', s) = F_\phi(\iota, \kappa, \pi, \epsilon') \\
&\qquad\qquad (\kappa'', s') = F(\iota, \kappa', \pi, \epsilon) \\
&\qquad \text{in } (\kappa'', ss') \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = \ldots \\
&\qquad\qquad (\kappa', s) = (\kappa, \texttt{""}) \\
&\qquad\qquad (\kappa'', s') = F(\iota, \kappa', \pi, \epsilon) \\
&\qquad \text{in } (\kappa'', ss') \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } (\kappa'', s') = F(\iota, \kappa, \pi, \epsilon) \\
&\qquad \text{in } (\kappa'', s') \\
&= F
\end{aligned}$$

Theorem $D \texttt{ <> empty} = D$.

Proof First component: $(D \texttt{ <> empty})_1 = min(\eta, \omega + 0) = \omega$, because $\omega \leq \eta$.
Second component: $(D \texttt{ <> empty})_2 = min(\eta, \omega + \infty) = \eta$.

Third component: $(D \texttt{ <> empty})_3 = \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = max(\pi - \infty, \epsilon - 0)$

$$\begin{aligned}
&\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon') \\
&\qquad\qquad (\kappa'', s') = F_\phi(\iota, \kappa', \pi, \epsilon) \\
&\qquad \text{in } (\kappa'', ss') \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = \epsilon \\
&\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon') \\
&\qquad\qquad (\kappa'', s') = (\kappa, \text{""}) \\
&\qquad \text{in } (\kappa'', ss') \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } (\kappa', s) = F(\iota, \kappa, \pi, \epsilon) \\
&\qquad \text{in } (\kappa', s) \\
&= F
\end{aligned}$$

Theorem $\texttt{nest } i \ (D \texttt{ <> } D') = (\texttt{nest } i \ D) \texttt{ <> } (\texttt{nest } i \ D')$.

Proof For the first two components, the result is immediate. For the third, define the function $indent_i = \lambda(\iota, \kappa, \pi, \epsilon).(\iota + i, \kappa, \pi, \epsilon)$. It is easy to see that for any document D, $(\texttt{nest } i \ D)_3 = D_3 \circ indent_i$. Now consider

$$\begin{aligned}
&(\texttt{nest } i \ D) \texttt{ <> } (\texttt{nest } i \ D')_3 = \\
&\quad \lambda(\_, \kappa, \_, \epsilon).\text{let } \epsilon' = \ldots \\
&\qquad\qquad (\kappa', s) = (F \circ indent_i)(\_, \kappa, \_, \epsilon') \\
&\qquad\qquad (\kappa'', s') = (F' \circ indent_i)(\_, \kappa', \_, \epsilon) \\
&\qquad \text{in } (\kappa'', ss')
\end{aligned}$$

But since $indent_i$ doesn't affect any components of σ that are used in the calculation of ϵ', this expression is equal to

$$\begin{array}{l} (\lambda(\_, \kappa, \_, \epsilon).\text{let } \epsilon' = \ldots \\ \qquad\qquad\qquad (\kappa', s) = F(\_, \kappa, \_, \epsilon') \\ \qquad\qquad\qquad (\kappa'', s') = F'(\_, \kappa', \_, \epsilon) \\ \qquad\qquad \text{in } (\kappa'', ss') \\ ) \circ indent_i \end{array}$$

which equals $(D \texttt{<>} D')_3 \circ indent_i$.

Theorem `nest` i `empty` = `empty`.

Proof The equality of the first two components is immediate. For the third component, the result follows from the fact that the `nest` operation affects only the ι part of the state, and F_ϕ looks only at the κ part.

Theorem `nest` i (`nest` j D) = `nest` $(i + j)$ D.

Proof As usual, the equivalence of the first two components is obvious. For the third component, note that $(\texttt{nest}\ i\ (\texttt{nest}\ j\ D))_3 = F \circ indent_i \circ indent_j$, $(\texttt{nest}\ i+j\ D)_3 = F \circ indent_{i+j}$, and clearly $indent_{i+j} = indent_i \circ indent_j$.

Theorem `nest` 0 $D = D$.

Proof Immediate from definition of `nest`.

Theorem $D \texttt{<>} (D' \texttt{<>} D'') = (D \texttt{<>} D') \texttt{<>} D''$.

Proof We will show that both expressions have the following value:

$$\begin{array}{l} D <> D' <> D'' = \\ \quad (min(\eta, \omega + \eta', \omega + \omega' + \omega''), \\ \quad\ min(\eta, \omega + \eta', \omega + \omega' + \eta''), \\ \quad \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = max(\pi - \eta', \pi - \eta'' - \omega', \epsilon - \omega'' - \omega') \\ \qquad\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon') \\ \qquad\qquad\qquad \epsilon'' = max(\pi - \eta'', \epsilon - \omega'') \\ \qquad\qquad\qquad (\kappa'', s') = F'(\iota, \kappa', \pi, \epsilon'') \\ \qquad\qquad\qquad (\kappa''', s'') = F''(\iota, \kappa'', \pi, \epsilon) \\ \qquad\qquad \text{in } (\kappa''', ss's'') \end{array}$$

Define $E = D \texttt{<>} D'$ and $E' = D' \texttt{<>} D''$. Consider first $D \texttt{<>} E'$:

First component: $(D \texttt{<>} E')_1 = min(\eta, \omega + E'_1) = min(\eta, \omega + min(\eta', \omega' + \omega'')) = min(\eta, \omega + \eta', \omega + \omega' + \omega'')$.

Second component: $(D \texttt{<>} E')_2 = min(\eta, \omega + E'_2) = min(\eta, \omega + min(\eta', \omega' + \eta'')) = min(\eta, \omega + \eta', \omega + \omega' + \eta'')$.

Third component: $(D \texttt{<>} E')_3 = \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = max(\pi - E'_2, \epsilon - E'_1)$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon')$

$$
\begin{aligned}
&\qquad\qquad\quad (\kappa'', s') = E_3'(\iota, \kappa', \pi, \epsilon) \\
&\qquad\quad \text{in } (\kappa'', ss') \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = max(\pi - min(\eta', \omega' + \eta''), \epsilon - min(\eta', \omega' + \omega'')) \\
&\qquad\qquad\quad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon') \\
&\qquad\qquad\quad (\bar\kappa, \bar s) = E_3'(\iota, \kappa', \pi, \epsilon) \\
&\qquad\quad \text{in } (\bar\kappa, s\bar s)
\end{aligned}
$$

Here, $\epsilon' = max(\pi - \eta', \pi - \omega' - \eta'', \epsilon - \eta', \epsilon - \omega' - \omega'')$. We can ignore $\epsilon - \eta'$ because $\epsilon \leq \pi$ implies $\epsilon - \eta' \leq \pi - \eta'$. Thus, we find that ϵ', and therefore (κ', s), are as earlier stated.

As for $(\bar\kappa, \bar s)$:

$$
\begin{aligned}
E_3'(\iota, \kappa', \pi, \epsilon) = &\text{ let } \epsilon'' = max(\pi - \eta'', \epsilon - \omega') \\
&\quad (\kappa'', s') = F'(\iota, \kappa', \pi, \epsilon'') \\
&\quad (\kappa''', s'') = F''(\iota, \kappa'', \pi, \epsilon) \\
&\text{ in } (\kappa''', s's'')
\end{aligned}
$$

Thus, s' and s'' are also as above.

E <> D'' has the same value, by similar algebraic reasoning:

First component: $(E <> D'')_1 = min(E_2, E_1 + \omega'') = min(min(\eta, \omega + \eta'), min(\eta, \omega + \omega) + \omega'')$ $= min(\eta, \omega + \eta', \eta + \omega'', \omega + \omega' + \omega'')$. $= min(\eta, \omega + \eta', \omega + \omega' + \omega'')$.

Second component: $(E <> D'')_2 = min(E_2, E_1 + \eta'') = min(min(\eta, \omega + \eta'), min(\eta, \omega + \omega') + \eta'')$ $= min(\eta, \omega + \eta', \eta + \eta'', \omega + \omega' + \eta'')$. $= min(\eta, \omega + \eta', \omega + \omega' + \eta'')$.

Third component:
$$
\begin{aligned}
(E <> D'')_3 = &\ \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon'' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\qquad (\bar\kappa, \bar s) = E_3(\iota, \kappa, \pi, \epsilon'') \\
&\qquad\qquad (\kappa''', s'') = F''(\iota, \bar\kappa, \pi, \epsilon) \\
&\qquad \text{in } (\kappa''', \bar s s'') \\
= &\ \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon'' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\qquad (\bar\kappa, \bar s) = \text{let } \epsilon' = max(\pi - eta', \epsilon'' - \omega') \\
&\qquad\qquad\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon') \\
&\qquad\qquad\qquad\qquad (\kappa'', s') = F'(\iota, \kappa', \pi, \epsilon'') \\
&\qquad\qquad\qquad \text{in } (\kappa'', ss') \\
&\qquad\qquad (\kappa''', s'') = F''(\iota, \bar\kappa, \pi, \epsilon) \\
&\qquad \text{in } (\kappa''', \bar s s'')
\end{aligned}
$$

Thus, F, F', and F'' are each applied to the same arguments, and the result is the concatenation of the three strings.

Theorem `nest` k $(D$ `<|>` $D')$ = (`nest` k D) `<|>` (`nest` k D').

Proof The equality of the first two components is obvious, since `nest` does not affect these. As for the third component:

$$
\begin{aligned}
(\texttt{nest}\ k\ (D \texttt{<|>} D'))_3 &= \lambda(\iota, \kappa, \pi, \epsilon).(D \texttt{<|>} D')_3(\iota + k, \kappa, \pi, \epsilon) \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{fits } D\,(\iota + k, \kappa, \pi, \epsilon) \to D_3(\iota + k, \kappa, \pi, \epsilon), D_3'(\iota + k, \kappa, \pi, \epsilon) \\
(\ (\texttt{nest}\ k\ D) \texttt{<|>} (\texttt{nest}\ k\ D')_3 & \\
&= \lambda(\iota, \kappa, \pi, \epsilon).\text{fits } D\,(\iota, \kappa, \pi, \epsilon) \to D_3(\iota + k, \kappa, \pi, \epsilon), D_3'(\iota + k, \kappa, \pi, \epsilon)
\end{aligned}
$$

But fits does not look at the ι component at all, so fits $D\,(\iota, \kappa, \pi, \epsilon) \equiv$ fits $D\,(\iota + k, \kappa, \pi, \epsilon)$, and the result follows.

The proof that the choice operator is associative relies on the following lemma:

Lemma For all $\sigma \in PPState$, fits $(D \mathrel{<|>} D')\ \sigma$ iff fits $D\ \sigma \vee$ fits $D'\ \sigma$.

Proof Let $\hat{D} = D \mathrel{<|>} D'$. By definition of fits and $<|>$:

$$\begin{aligned}
\text{fits } \hat{D}\ \sigma \text{ iff } & \kappa + \hat{D}_1 \leq \epsilon \vee \kappa + \hat{D}_2 \leq \pi \\
\text{iff } & \kappa + min(\omega, \omega') \leq \epsilon \vee \kappa + min(\eta, \eta') \leq \pi \\
\text{iff } & \kappa + \omega \leq \epsilon \vee \kappa + \omega' \leq \epsilon \vee \kappa + \eta \leq \pi \vee \kappa + \eta' \leq \pi
\end{aligned}$$

while

$$\text{fits } D\ \sigma \vee \text{fits } D'\ \sigma \vee \text{ iff } (\kappa + \omega \leq \epsilon \vee \kappa + \eta \leq \pi) \vee (\kappa + \omega' \leq \epsilon \vee \kappa + \eta' \leq \pi)$$

Theorem $D \mathrel{<|>} (D' \mathrel{<|>} D'') = (D \mathrel{<|>} D') \mathrel{<|>} D''$.

Proof Clearly, the first two components of both documents are $min(\omega, \omega', \omega'')$ and $min(\eta, \eta', \eta'')$, respectively. For the third component, we have

$$\begin{aligned}
D \mathrel{<|>} (D' \mathrel{<|>} D'')_3 &= \lambda\sigma.\text{fits } D\sigma \to F\sigma, (D' \mathrel{<|>} D'')_3\sigma \\
&= \lambda\sigma.\text{fits } D\sigma \to F\sigma, (\text{fits } D'\sigma \to F'\sigma, F''\sigma) \\
&= \lambda\sigma.(\text{fits } D\sigma \vee \text{fits } D'\sigma) \to (\text{fits } D\sigma \to F\sigma, F'\sigma), F''\sigma \\
&= \lambda\sigma.(\text{fits } (D \mathrel{<|>} D')\sigma \to (D \mathrel{<|>} D')_3\sigma, F''\sigma \\
&= ((D \mathrel{<|>} D') \mathrel{<|>} D'')_3
\end{aligned}$$

The third line is a logical equivalence, while the fourth follows from the lemma.

Theorem $(D \mathrel{<|>} D') \mathrel{<>} D'' = (D \mathrel{<>} D'') \mathrel{<|>} (D' \mathrel{<>} D'')$

Proof Let $\hat{D} = D{<|>}D'$, $E = D{<>}D''$, $E' = D'{<>}D''$.

We proceed component-wise, as usual.

$$\begin{aligned}
\text{First component: } (\hat{D}{<>}D'')_1 &= min(\hat{D}_2, \hat{D}_1 + \omega'') \\
&= min(min(\eta, \eta'), min(\omega, \omega') + \omega'') \\
&= min(\eta, \eta', \omega + \omega'', \omega' + \omega'') \\
(E{<|>}E')_1 &= min(E_1, E'_1) \\
&= min(min(\eta, \omega + \omega''), min(\eta', \omega' + \omega'') \\
&= min(min(\eta, \omega + \omega'', \eta', \omega' + \omega'')
\end{aligned}$$

$$\begin{aligned}
\text{Second component: } (\hat{D}{<>}D'')_2 &= min(\hat{D}_2, \hat{D}_1 + \omega'') \\
&= min(min(\eta, \eta'), min(\omega, \omega') + \eta'') \\
&= min(\eta, \eta', \omega + \eta'', \omega' + \eta'') \\
(E{<|>}E')_2 &= min(E_2, E'_2) \\
&= min(min(\eta, \omega + \eta''), min(\eta', \omega' + \eta'') \\
&= min(min(\eta, \omega + \eta'', \eta', \omega' + \eta'')
\end{aligned}$$

Third component:

$$
\begin{aligned}
(\hat{D} \mathbin{<>} D'')_3 &= \lambda(\iota,\kappa,\pi,\epsilon).\text{let } \epsilon' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\qquad (\kappa', s) = \hat{D}_3(\iota,\kappa,\pi,\epsilon') \\
&\qquad\qquad (\kappa'', s') = F''(\iota,\kappa',\pi,\epsilon) \\
&\qquad\quad \text{in } (\kappa'', ss') \\
&= \lambda(\iota,\kappa,\pi,\epsilon).\text{let } \epsilon' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\qquad (\kappa', s) = \text{fits } D(\iota,\kappa,\pi,\epsilon') \to F(\iota,\kappa,\pi,\epsilon'), F'(\iota,\kappa,\pi,\epsilon') \\
&\qquad\qquad (\kappa'', s') = F''(\iota,\kappa',\pi,\epsilon) \\
&\qquad\quad \text{in } (\kappa'', ss') \\
&= \lambda(\iota,\kappa,\pi,\epsilon).\text{let } \epsilon' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\quad \text{in fits } D(\iota,\kappa,\pi,\epsilon') \to \\
&\qquad\qquad \text{let } (\kappa', s) = F(\iota,\kappa,\pi,\epsilon') \\
&\qquad\qquad\quad (\kappa'', s') = F''(\iota,\kappa',\pi,\epsilon) \\
&\qquad\qquad \text{in } (\kappa'', ss'), \\
&\qquad\qquad \text{let } (\kappa', s) = F'(\iota,\kappa,\pi,\epsilon') \\
&\qquad\qquad\quad (\kappa'', s') = F''(\iota,\kappa',\pi,\epsilon) \\
&\qquad\qquad \text{in } (\kappa'', ss')
\end{aligned}
$$

$$
\begin{aligned}
(E \mathbin{<|>} E')_3) &= \lambda\sigma.\text{fits } E\sigma \to E_3\sigma, E'_3\sigma \\
&= \lambda(\iota,\kappa,\pi,\epsilon).\text{fits } E(\iota,\kappa,\pi,\epsilon) \to \\
&\qquad\qquad \text{let } \epsilon' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\qquad\quad (\kappa', s) = F(\iota,\kappa,\pi,\epsilon') \\
&\qquad\qquad\quad (\kappa'', s') = F''(\iota,\kappa',\pi,\epsilon) \\
&\qquad\qquad \text{in } (\kappa'', ss'), \\
&\qquad\qquad \text{let } \epsilon' = max(\pi - \eta'', \epsilon - \omega'') \\
&\qquad\qquad\quad (\kappa', s) = F'(\iota,\kappa,\pi,\epsilon') \\
&\qquad\qquad\quad (\kappa'', s') = F''(\iota,\kappa',\pi,\epsilon) \\
&\qquad\qquad \text{in } (\kappa'', ss')
\end{aligned}
$$

We now need only prove that fits $D(\iota,\kappa,\pi,\epsilon')$ if and only iff fits $E(\iota,\kappa,\pi,\epsilon)$. Expanding the definition of fits:

$$
\begin{aligned}
\text{fits } D(\iota,\kappa,\pi,\epsilon') &\equiv \kappa + \omega \le \epsilon' \lor \kappa + \eta \le \pi \\
&\equiv \kappa + \omega \le max(\pi - \eta'', \epsilon - \omega'') \lor \kappa + \eta \le \pi \\
\text{fits } E(\iota,\kappa,\pi,\epsilon) &\equiv \kappa + E_1 \le \epsilon \lor \kappa + E_2 \le \pi \\
&\equiv \kappa + min(\eta, \omega + \omega'') \le \epsilon \lor \kappa + min(\eta, \omega + \eta'') \le \pi
\end{aligned}
$$

We now show the implications in both directions:

$$
\begin{aligned}
(\Rightarrow)\quad & \kappa + \omega \le \pi - \eta'' \Rightarrow \kappa + \omega + \eta'' \le \pi \Rightarrow \kappa + min(\ldots, \omega + \eta'') \le \pi \\
& \kappa + \omega \le \epsilon - \omega'' \Rightarrow \kappa + \omega + \omega'' \le \epsilon \Rightarrow \kappa + min(\ldots, \omega + \omega'') \le \epsilon \\
& \kappa + \eta \le \pi \Rightarrow \kappa + min(\eta, \ldots) \le \pi \\
(\Leftarrow)\quad & \kappa + \eta \le \epsilon \Rightarrow \kappa + \eta \le \pi(\text{ since } \epsilon \le \pi) \\
& \kappa + \omega + \omega'' \le \epsilon \Rightarrow \kappa + \omega \le \epsilon - \omega'' \Rightarrow \kappa + \omega \le max(\ldots, \epsilon - \omega'') \\
& \kappa + \eta \le \pi \Rightarrow \kappa + \eta \le \pi \\
& \kappa + \omega + \eta'' \le \pi \Rightarrow \kappa + \omega \le \pi - \eta'' \Rightarrow \kappa + \omega \le max(\pi - \eta'', \ldots)
\end{aligned}
$$

8 Further algebraic equivalences and non-equivalences

Here are refutations of what seem like obvious identities. The examples have been tested using both the definitions given in this paper and using Thomas Nordin's SML version of Wadler's combinators, giving identical results in all cases.

The following proposed identity, given by Wadler in [3], comes from the intuition that these combinators work by lifting all the choice operators to the top.

Observation D `<>` $(D'$ `<|>` $D'') \neq (D$ `<>` $D')$ `<|>` $(D$ `<>` $D'')$

Proof Let $D =$ `text "ab" <|> text "a"`, $D' =$ `text "bcd"`, and $D'' =$ `text "bc"`. If $\pi = 4$, then D `<>` $(D'$ `<|>` $D'')$ prints `abbc`, while $(D$ `<>` $D')$ `<|>` $(D$ `<>` $D'')$ prints `abcd`.

A restricted form of the above identity holds:

Theorem D `<>` $(D'$ `<|>` $D'')$ = $(D$ `<>` $D')$ `<|>` $(D$ `<>` $D'')$, if $D =$ `text` t for some t.

Proof Let $G = D'$ `<|>` D'', $H = D$ `<>` D', and $I = D$ `<>` D''.

First component:
$$\begin{aligned}(D\text{<>}G)_1 &= min(\eta, \omega + G_1)\\ &= min(\eta, \omega + min(\omega', \omega''))\\ &= min(\eta, \omega + \omega', \omega + \omega'')\\ (H\text{<|>}I)_1 &= min(H_1, I_1)\\ &= min(min(\eta, \omega + \omega'), min(\eta, \omega + \omega''))\\ &= min(\eta, \omega + \omega', \omega + \omega'')\end{aligned}$$

Second component:
$$\begin{aligned}(D\text{<>}G)_1 &= min(\eta, \omega + G_2)\\ &= min(\eta, \omega + min(\eta', \eta''))\\ &= min(\eta, \omega + \eta', \omega + \eta'')\\ (H\text{<|>}I)_1 &= min(H_1, I_1)\\ &= min(min(\eta, \omega + \eta'), min(\eta, \omega + \eta'')\\ &= min(\eta, \omega + \eta', \omega + \eta'')\end{aligned}$$

Third component:

$$\begin{aligned}(D \text{ <> } G)_3 &= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } \epsilon' = max(\pi - G_2, \epsilon - G_1)\\ &\qquad\qquad (\kappa', s) = F(\iota, \kappa, \pi, \epsilon')\\ &\qquad\qquad (\kappa'', s') = G_3(\iota, \kappa', \pi, \epsilon)\\ &\qquad\quad \text{in } (\kappa'', ss')\\ &= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } (\kappa', s) = (\kappa + |t|, t)\\ &\qquad\qquad (\kappa'', s') = G_3(\iota, \kappa', \pi, \epsilon)\\ &\qquad\quad \text{in } (\kappa'', ss')\\ &= \lambda(\iota, \kappa, \pi, \epsilon).\text{let } (\kappa', s) = (\kappa + |t|, t)\\ &\qquad\qquad (\kappa'', s') = \text{fits } D'(\iota, \kappa', \pi, \epsilon) \rightarrow F'(\iota, \kappa', \pi, \epsilon), F''(\iota, \kappa', \pi, \epsilon)\\ &\qquad\quad \text{in } (\kappa'', ss')\\ (H \text{ <|> } I)_3 &= \lambda\sigma.\text{fits } H\sigma \rightarrow H_3\sigma, I_3\sigma\\ &= \lambda(\iota, \kappa, \pi, \epsilon).\kappa + min(\eta, \omega + \omega') \leq \epsilon \vee \kappa + min(\eta, \omega + \eta') \leq \pi \rightarrow\\ &\qquad\quad \text{let } \epsilon' = \ldots\\ &\qquad\qquad (\kappa', s) = (\kappa + |t|, t)\\ &\qquad\qquad (\kappa'', s') = F'(\iota, \kappa', \pi, \epsilon)\\ &\qquad\quad \text{in } (\kappa'', ss'),\\ &\qquad\quad \text{let } \epsilon' = \ldots\\ &\qquad\qquad (\kappa', s) = (\kappa + |t|, t)\\ &\qquad\qquad (\kappa'', s') = F''(\iota, \kappa', \pi, \epsilon)\\ &\qquad\quad \text{in } (\kappa'', ss'),\end{aligned}$$

So we need to show that fits $D'(\iota, \kappa', \pi, \epsilon)$ if and only if $\kappa + min(\eta, \omega + \omega') \leq \epsilon \vee \kappa + min(\eta, \omega + \eta') \leq \pi$. By definition, fits $D'(\iota, \kappa', \pi, \epsilon) \equiv \kappa' + \omega' \leq \epsilon \vee \kappa' + \eta' \leq \pi$. The equivalence follows since $\eta = \infty$ and $\kappa' = \kappa + \omega$.

The following identity seems intuitively obvious, since a line always fits and therefore will always be chosen.

Observation `line <|>` $D \neq$ `line`.

Proof In context $C = \lambda d.$`text "abc" <>` d, with page width 2, C (`line <|> text "d"`) prints `abcd`, while C (`line`) prints `abc\n`.

The problem is that `line` does *not* always fit — in particular, it doesn't fit if the line on which it is to be placed is already over-filled.[7] The same intuition says that `empty` always fits, and the same counter-example works:

Observation `empty <|>` $D \neq$ `empty`.

Proof In context $C = \lambda d.$`text "abc" <>` d, with page width 2, C (`empty <|> text "d"`) prints `abcd`, while C (`empty`) prints `abc`.

On the other hand, `line` is a two-sided zero for the choice operator: `line` = `line <|>` D `<|> line`. The same identity holds for `empty`, on the condition that D is flat ($D_2 = \infty$).

9 Implementation

The code shown in Figure 2 is a direct transcription of Figure 1, with two small enhancements.

First, we have included a `tab` operation that sets the indent for subsequent lines to equal the current column; it is easy to verify — as must be done when adding new combinators — that the invariant $\omega \leq \eta$ is satisfied. On the other hand, we would not be able to define an operation that tabs over to a certain column on the *current* line, because it would be impossible to supply a minimum width: the width of the combinator depends upon the current column and is therefore not "static."

We have also included an emptiness test, `isEmpty`, which is useful for combinators such as:

```
fun x <+> y = if isEmpty x then y else
                    if isEmpty y then x else x <> space <> y;
```

Note that this is a *user-level* definition.

However, it should also be understood that this emptiness test is not perfectly accurate. For example, if D = `text "a" <|> empty`, then `isEmpty` D is true, because $D_1 = 0$ and $D_2 = \infty$, but clearly D iss not the same as empty. Perhaps applying D_3 to some small set of states would be a correct test, but it would not be efficient. Another idea is to add an emptiness bit to each document, but that is not simple either. It would be easy if we could say that the empty document can only be constructed from other empty documents, but as we saw in the last section, `(empty <|> text "a") <|> empty` = `empty`. (For this reason, it is also not clear how to to define `isEmpty` in Wadler's implementation.)

[7]Actually, it is not clear whether this is a bug or a feature in Wadler's definitions. It seems odd. I could fix it by changing the definition of "fits" to "$\omega = 0 \vee \ldots$". I don't know how easy it would be to fix in Wadler's implementation.

```
infixr 5 <|>
infixr 6 <>

fun copies 0 s = "" | copies n s = s ^ (copies (n-1) s);
val max = Int.max;
val min = Int.min;

type PPState = int  (* ci = current indent *)
             * int  (* cc = current column in output *)
             * int  (* pw = page width *)
             * int; (* epw = effective page width *)

type Doc = int   (* mwo = minimum width, w/o newline *)
         * int   (* mw = minimum width , w/ newline *)
         * (PPState -> int * string); (* last column and text *)

val infinity = 99999999;

fun text s = (size s, infinity, (fn (_,cc,_,_) => (cc+size s, s))) : Doc;

val empty = text "";

val line = (0, 0, fn (ci,cc,pw,epw) => (ci, "\n" ^ (copies ci " "))) : Doc;

fun nest k ((mwo, mw, F):Doc) =
       (mwo, mw, (fn (ci,cc,pw,epw) => F (ci+k,cc,pw,epw))) : Doc;

fun ((mwo,mw,F):Doc) <> ((mwo',mw',F'):Doc) =
      (min(mw, mwo+mwo'), min(mw, mwo+mw'),
       (fn st as (ci,cc,pw,epw) =>
          let val epw' = max(pw - mw', epw - mwo')
              val (cc',s) = F(ci,cc,pw,epw')
              val (cc'',s') = F'(ci,cc',pw,epw)
          in (cc'', s^s')
          end)) : Doc ;

fun ((mwo,mw,F):Doc) <|> ((mwo',mw',F'):Doc) =
   (min(mwo,mwo'), min(mw,mw'),
    fn st as (ci,cc,pw,epw) =>
      let val Dfits = cc+mwo <= epw orelse cc+mw <= pw
      in if Dfits then F st else F' st
      end) : Doc;

fun tab ((mwo,mw,F):Doc) =
  (mwo, mw, (fn (ci,cc,pw,epw) => F (cc,cc,pw,epw)));

fun isEmpty ((mwo,mw,_):Doc) = mwo = 0 andalso mw = infinity;

fun pretty pw ((_,_,F):Doc) = #2 (F (0,0,pw,pw)) ^ "\n";
```

Figure 2: Standard ML implementation of combinators

Acknowledgements

This work would never have been completed without the help of Thomas Nordin; aside from providing the ML version of Wadler's combinators, Thomas provided test cases that acted as counter-examples to numerous earlier versions of these combinators.

References

[1] J. Hughes, "The design of a pretty-printer library," in J. Jeuring and E. Meijer (eds.), **Advanced Functional Programming**, Springer-Verlag LNCS 925, 1995.

[2] S. L. Peyton Jones, Haskell pretty-printing library, available at `http://www.dcs.gla.ac.uk/~simonpj/`, 1997.

[3] P. Wadler, "A prettier printer," Bell Labs, March 1998.

[4] P. Wadler, Notes for presentation at IFIP Working Group 2.8 meeting, March 1998.