PPrint, a prettier printer

DAAN LEIJEN
University of Utrecht
Dept. of Computer Science
PO.Box 80.089, 3508 TB Utrecht
The Netherlands
daan@cs.uu.nl, http://www.cs.uu.nl/~daan

5 Oct 2001

1 Introduction

PPrint is an implementation of the pretty printing combinators described by Philip Wadler (1997). In their bare essence, the combinators of Wadler are not expressive enough to describe some commonly occurring layouts. The PPrint library adds new primitives to describe these layouts and works well in practice.

The library is based on a single way to concatenate documents, which is associative and has both a left and right unit. This simple design leads to an efficient and short implementation. The simplicity is reflected in the predictable behaviour of the combinators which make them easy to use in practice.

1.1 Compatibility

The library is written in Haskell98. It is successfully compiled with GHC 4.x, GHC 5.x, Hugs98 and nhc98, but should work with any Haskell98 compliant compiler or interpreter.

1.2 License

These days, one can't distribute free software without saying in incomprehensible words that it is free. Well, here it is ...

2 Users guide

To be written. A thorough description of the primitive combinators and their implementation can be found in Philip Wadlers paper (1997). Additions and the main differences with his original paper are:

- The `nil` document is called `empty`.
- The *above* combinator is called `<$>`. The operator `</>` is used for soft line breaks.
- There are three new primitives: `align`, `fill` and `fillBreak`. These are very useful in practice.
- Lots of other useful combinators, like `fillSep` and `list`.
- There are two renderers, `renderPretty` for pretty printing and `renderCompact` for compact output. The pretty printing algorithm also uses a ribbon-width now for even prettier output.
- There are two displayers, `displayS` for strings and `displayIO` for file based output.
- There is a `Pretty` class.
- The implementation uses optimised representations and strictness annotations.

3 The document algebra

The combinators in the library satisfy many algebraic laws. Here is a list of laws for the primitive combinators.

The concatenation operator `<>` is associative and has `empty` as a left and right unit.

```
x <> (y <> z)       = (x <> y) <> z
x <> empty          = x
empty <> x          = x
```

The `text` combinator is a homomorphism from string concatenation to document concatenation. The combinator `char` behaves like one-element text.

```
text (s ++ t)       = text s <> text t
text ""             = empty
char c              = text [c]
```

The `nest` combinator is a homomorphism from addition to document composition. `nest` also distributes through document concatenation and is absorbed by `text` and `align`.

```
nest (i+j) x        = nest i (nest j x)
nest 0 x            = x
nest i (x <> y)     = nest i x <> nest j y
nest i empty        = empty
nest i (text s)     = text s
nest i (align x)    = align x
```

The `group` combinator is absorbed by `empty`. It is commutative with `nest` and `align`.

```
group (empty)       = empty
group (text s <> x) = text s <> group x
group (nest i x)    = nest i (group x)
group (align x)     = align (group x)
```

The `align` combinator is absorbed by `empty` and `text`.

```
align (empty)       = empty
align (text s)      = text s
align (align x)     = align x
```

From the laws of the primitive combinators, we can derive many other laws for the derived combinators. For example, the *above* operator `<$>` is defined as:

```
x <$> y             = x <> line <> y
```

It follows that `<$>` is associative and that `<$>` and `<>` associate with each other.

```
x <$> (y <$> z)     = (x <$> y) <$> z
x <> (y <$> z)      = (x <> y) <$> z
x <$> (y <> z)      = (x <$> y) <> z
```

The same laws also hold for the other line break operators `</>`, `<$$>` and `<//>`.

4 Reference guide

4.1 Documents

Doc

The abstract data type `Doc` represents pretty documents.

show `:: Doc -> String`

`Doc` is an instance of the `Show` class. `(show doc)` pretty prints document `doc` with a page width of 100 characters and a ribbon width of 40 characters.

```
show (text "hello" <$> text "world")
```

Which would return the string `"hello\nworld"`, i.e.

```
hello
world
```

putDoc `:: Doc -> IO ()`

The action `(putDoc doc)` pretty prints document `doc` to the standard output. with a page width of 100 characters and a ribbon width of 40 characters.

```
main :: IO ()
main = do{ putDoc (text "hello" <+> text "world") }
```

Which would output

```
hello world
```

hPutDoc `:: Handle -> Doc -> IO ()`

`(hPutDoc handle doc)` pretty prints document `doc` to the file handle `handle` with a page width of 100 characters and a ribbon width of 40 characters.

```
main    = do{ handle <- openFile "MyFile" WriteMode
            ; hPutDoc handle (vcat (map text
                                    ["vertical","text"]))
            ; hClose handle
            }
```

4.2 Basic combinators

empty `:: Doc`

The empty document is, indeed, empty. Allthough `empty` has no content, it does have a 'height' of 1 and behaves exactly like `(text "")` (and is therefore not a unit of `<$>`).

char `:: Char -> Doc`

The document `(char c)` contains the literal character `c`. The character shouldn't be a newline (`'\n'`), the function `line` should be used for line breaks.

text `:: String -> Doc`

The document `(text s)` contains the literal string `s`. The string shouldn't contain any newline (`'\n'`) characters. If the string contains newline characters, the function `string` should be used.

```
(<>) :: Doc -> Doc -> Doc
infixr 6
```

The document `(x <> y)` concatenates document `x` and document `y`. It is an associative operation having `empty` as a left and right unit.

nest `:: Int -> Doc -> Doc`

The document `(nest i x)` renders document `x` with the current indentation level increased by `i` (See also `hang`, `align` and `indent`).

```
nest 2 (text "hello" <$> text "world") <$> text "!"
```

outputs as:

```
hello
  world
!
```

line `:: Doc`

The `line` document advances to the next line and indents to the current nesting level. Document `line` behaves like `(text " ")` if the line break is undone by `group`.

linebreak `:: Doc`

The `linebreak` document advances to the next line and indents to the current nesting level. Document `linebreak` behaves like `empty` if the line break is undone by `group`.

group `:: Doc -> Doc`

The `group` combinator is used to specify alternative layouts. The document `(group x)` undoes all line breaks in document `x`. The resulting line is added to the current line if that fits the page. Otherwise, the document `x` is rendered without any changes.

softline `:: Doc`

The document `softline` behaves like `space` if the resulting output fits the page, otherwise it behaves like `line`.

```
softline  = group line
```

softbreak `:: Doc`

The document `softbreak` behaves like `empty` if the resulting output fits the page, otherwise it behaves like `line`.

```
softbreak  = group linebreak
```

4.3 Alignment

The combinators in this section can not be descibed by Wadler's original combinators. They align their output relative to the *current* output position – in contrast to `nest` which always aligns to the current nesting level. This deprives these combinators from being 'optimal'. In practice however they prove to be very useful. The combinators in this section should be used with care, since they are more expensive than the other combinators. For example, `align` shouldn't be used to pretty print all top-level declarations of a language, but using `hang` for `let` expressions is fine.

align `:: Doc -> Doc`

The document `(align x)` renders document `x` with the nesting level set to the current column. It is used for example to implement `hang`.

As an example, we will put a document right above another one, regardless of the current nesting level:

```
x $$ y  = align (x <$> y)

test    = text "hi" <+> (text "nice" $$ text "world")
```

which will be layed out as:

```
hi nice
   world
```

hang `:: Int -> Doc -> Doc`

The `hang` combinator implements hanging indentation. The document `(hang i x)` renders document `x` with a nesting level set to the current column plus `i`. The following example uses hanging indentation for some text:

```
test  = hang 4 (fillSep (map text
        (words "the hang combinator indents these words !")))
```

Which lays out on a page with a width of 20 characters as:

```
the hang combinator
    indents these
    words !
```

The `hang` combinator is implemented as:

```
hang i x  = align (nest i x)
```

indent `:: Int -> Doc -> Doc`

The document `(indent i x)` indents document `x` with `i` spaces.

```
test  = indent 4 (fillSep (map text
        (words "the indent combinator indents these words !")))
```

Which lays out with a page width of 20 as:

```
the indent
combinator
indents these
words !
```

encloseSep `:: Doc -> Doc -> Doc -> [Doc] -> Doc`

The document `(encloseSep l r sep xs)` concatenates the documents `xs` seperated by `sep` and encloses the resulting document by `l` and `r`. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All seperators are put in front of the elements. For example, the combinator `list` can be defined with `encloseSep`:

```
list xs  = encloseSep lbracket rbracket comma xs
test     = text "list" <+> (list (map int [10,200,3000]))
```

Which is layed out with a page width of 20 as:

```
list [10,200,3000]
```

But when the page width is 15, it is layed out as:

```
list [10
     ,200
     ,3000]
```

list `:: [Doc] -> Doc`

The document `(list xs)` comma seperates the documents `xs` and encloses them in square brackets. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma seperators are put in front of the elements.

tupled `:: [Doc] -> Doc`

The document `(tupled xs)` comma seperates the documents `xs` and encloses them in parenthesis. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma seperators are put in front of the elements.

semiBraces `:: [Doc] -> Doc`

The document `(semiBraces xs)` seperates the documents `xs` with semi colons and encloses them in braces. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All semi colons are put in front of the elements.

4.4 Operators

```
(<+>) :: Doc -> Doc -> Doc
infixr 6
```

The document `(x <+> y)` concatenates document `x` and `y` with a `space` in between.

`(<$>) :: Doc -> Doc -> Doc`
`infixr 5`

The document `(x <$> y)` concatenates document `x` and `y` with a `line` in between.

`(</>) :: Doc -> Doc -> Doc`
`infixr 5`

The document `(x </> y)` concatenates document `x` and `y` with a `softline` in between. This effectively puts `x` and `y` either next to each other (with a `space` in between) or underneath each other.

`(<$$>) :: Doc -> Doc -> Doc`
`infixr 5`

The document `(x <$$> y)` concatenates document `x` and `y` with a `linebreak` in between.

`(<//>) :: Doc -> Doc -> Doc`
`infixr 5`

The document `(x <//> y)` concatenates document `x` and `y` with a `softbreak` in between. This effectively puts `x` and `y` either right next to each other or underneath each other.

4.5 List combinators

hsep `:: [Doc] -> Doc`

The document `(hsep xs)` concatenates all documents `xs` horizontally with `(<+>)`.

vsep `:: [Doc] -> Doc`

The document `(vsep xs)` concatenates all documents `xs` vertically with `(<$>)`. If a `group` undoes the line breaks inserted by `vsep`, all documents are seperated with a `space`.

```
someText = map text (words ("text to lay out"))

test     = text "some" <+> vsep someText
```

This is layed out as:

```
some text
to
lay
out
```

The `align` combinator can be used to align the documents under their first element

```
test     = text "some" <+> align (vsep someText)
```

Which is printed as:

```
some text
     to
     lay
     out
```

fillSep `:: [Doc] -> Doc`

The document `(fillSep xs)` concatenates documents `xs` horizontally with `(<+>)` as long as its fits the page, than inserts a `line` and continues doing that for all documents in `xs`.

```
fillSep xs  = foldr (</>) empty xs
```

sep `:: [Doc] -> Doc`

The document `(sep xs)` concatenates all documents `xs` either horizontally with `(<+>)`, if it fits the page, or vertically with `(<$>)`.

```
sep xs  = group (vsep xs)
```

hcat `:: [Doc] -> Doc`

The document `(hcat xs)` concatenates all documents `xs` horizontally with `(<>)`.

vcat `:: [Doc] -> Doc`

The document `(vcat xs)` concatenates all documents `xs` vertically with `(<$$>)`. If a `group` undoes the line breaks inserted by `vcat`, all documents are directly concatenated.

fillCat `:: [Doc] -> Doc`

The document `(fillCat xs)` concatenates documents `xs` horizontally with `(<>)` as long as its fits the page, than inserts a `linebreak` and continues doing that for all documents in `xs`.

```
fillCat xs  = foldr (<//>) empty xs
```

cat `:: [Doc] -> Doc`

The document `(cat xs)` concatenates all documents `xs` either horizontally with `(<>)`, if it fits the page, or vertically with `(<$$>)`.

```
cat xs  = group (vcat xs)
```

punctuate `:: Doc -> [Doc] -> [Doc]`

`(punctuate p xs)` concatenates all in documents `xs` with document `p` except for the last document.

```
someText = map text ["words","in","a","tuple"]
test     = parens (align (cat (punctuate comma someText)))
```

This is layed out on a page width of 20 as:

```
(words,in,a,tuple)
```

But when the page width is 15, it is layed out as:

```
(words,
 in,
 a,
 tuple)
```

(If you want put the commas in front of their elements instead of at the end, you should use `tupled` or, in general, `encloseSep`.)

4.6 Fillers

fill `:: Int -> Doc -> Doc`

The document `(fill i x)` renders document `x`. It than appends spaces until the width is equal to `i`. If the width of `x` is already larger, nothing is appended. This combinator is quite useful in practice to output a list of bindings. The following example demonstrates this.

```
types  = [("empty","Doc")
         ,("nest","Int -> Doc -> Doc")
         ,("linebreak","Doc")]

ptype (name,tp)
       = fill 6 (text name) <+> text "::" <+> text tp

test   = text "let" <+> align (vcat (map ptype types))
```

Which is layed out as:

```
let empty  :: Doc
    nest   :: Int -> Doc -> Doc
    linebreak :: Doc
```

fillBreak `:: Int -> Doc -> Doc`

The document `(fillBreak i x)` first renders document `x`. It than appends spaces untill the width is equal to `i`. If the width of `x` is already larger than `i`, the nesting level is increased by `i` and a `line` is appended. When we redefine `ptype` in the previous example to use `fillBreak`, we get a useful variation of the previous output:

```
ptype (name,tp)
       = fillBreak 6 (text name) <+> text "::" <+> text tp
```

The output will now be:

```
let empty  :: Doc
    nest   :: Int -> Doc -> Doc
    linebreak
           :: Doc
```

4.7 Bracketing combinators

enclose `:: Doc ->Doc -> Doc -> Doc`

The document `(enclose l r x)` encloses document `x` between documents `l` and `r` using `(<>)`.

```
enclose l r x   = l <> x <> r
```

squotes `:: Doc -> Doc`

Document `(squotes x)` encloses document `x` with single quotes `"'"`.

dquotes `:: Doc -> Doc`

Document `(dquotes x)` encloses document `x` with double quotes `'"'`.

parens `:: Doc -> Doc`

Document `(parens x)` encloses document `x` in parenthesis, `"("` and `")"`.

angles `:: Doc -> Doc`

Document `(angles x)` encloses document `x` in angles, `"<"` and `">"`.

braces `:: Doc -> Doc`

Document `(braces x)` encloses document `x` in braces, `"{"` and `"}"`.

brackets `:: Doc -> Doc`

Document `(brackets x)` encloses document `x` in square brackets, `"["` and `"]"`.

4.8 Character documents

lparen `:: Doc`

The document `lparen` contains a left parenthesis, `"("`.

rparen `:: Doc`

The document `rparen` contains a right parenthesis, `")"`.

langle `:: Doc`

The document `langle` contains a left angle, `"<"`.

rangle `:: Doc`

The document `rangle` contains a right angle, `">"`.

lbrace `:: Doc`

The document `lbrace` contains a left brace, `"{"`.

rbrace `:: Doc`

The document `rbrace` contains a right brace, `"}"`.

lbracket `:: Doc`

The document `lbracket` contains a left square bracket, `"["`.

rbracket `:: Doc`

The document `rbracket` contains a right square bracket, `"]"`.

squote `:: Doc`

The document `squote` contains a single quote, `"'"`.

dquote `:: Doc`

The document `dquote` contains a double quote, `'"'`.

semi `:: Doc`

The document `semi` contains a semi colon, `";"`.

colon `:: Doc`

The document `colon` contains a colon, `":"`.

comma `:: Doc`

The document `comma` contains a comma, `","`.

space `:: Doc`

The document `space` contains a single space, `"" ""`.

```
x <+> y   = x <> space <> y
```

dot `:: Doc`

The document `dot` contains a single dot, `"."`.

backslash `:: Doc`

The document `backslash` contains a back slash, `"\"`.

equals `:: Doc`

The document `equals` contains an equal sign, `"="`.

4.9 Primitive type documents

string `:: String -> Doc`

The document `(string s)` concatenates all characters in `s` using `line` for newline characters and `char` for all other characters. It is used instead of `text` whenever the text contains newline characters.

int `:: Int -> Doc`

The document `(int i)` shows the literal integer `i` using `text`.

integer `:: Integer -> Doc`

The document `(integer i)` shows the literal integer `i` using `text`.

float `:: Float -> Doc`

The document `(float f)` shows the literal float `f` using `text`.

double `:: Double -> Doc`

The document `(double d)` shows the literal double `d` using `text`.

rational `:: Rational -> Doc`

The document `(rational r)` shows the literal rational `r` using `text`.

4.10 Pretty class

Pretty

The class `Pretty` has two members:

```
class Pretty a where
  pretty     :: a -> Doc
  prettyList :: [a] -> Doc
```

The member `prettyList` is only used to define the instance `Pretty a => Pretty [a]`. In normal circumstances only the `pretty` function is used. Library defined instances of `Pretty` are `()`, `Char`, `Int`, `Integer`, `Float`, `Double`, `Rational`, `Pretty a => Maybe a`, `Pretty a => [a]`, `Pretty a,Pretty b => (a,b)` and `Pretty a,Pretty b,Pretty c => (a,b,c)`.

4.11 Rendering

SimpleDoc

The data type `SimpleDoc` represents rendered documents and is used by the display functions.

```
data SimpleDoc  = SEmpty
                | SChar Char SimpleDoc
                | SText !Int String SimpleDoc
                | SLine !Int SimpleDoc
```

The `Int` in `SText` contains the length of the string. The `Int` in `SLine` contains the indentation for that line. The library provides two default display functions `displayS` and `displayIO`. You can provide your own display function by writing a function from a `SimpleDoc` to your own output format.

renderPretty `:: Float -> Int -> Doc -> SimpleDoc`

This is the default pretty printer which is used by `show`, `putDoc` and `hPutDoc`. `(renderPretty ribbonfrac width x)` renders document `x` with a page width of `width` and a ribbon width of `(ribbonfrac * width)` characters. The ribbon width is the maximal amount of non-indentation characters on a line.

The parameter `ribbonfrac` should be between 0.0 and 1.0. If it is lower or higher, the ribbon width will be `0` or `width` respectively.

renderCompact `:: Doc -> SimpleDoc`

`(renderCompact x)` renders document `x` without adding any indentation. Since no 'pretty' printing is involved, this renderer is very fast. The resulting output contains fewer characters as a pretty printed version and can be used for output that is read by other programs.

displayS `:: SimpleDoc -> ShowS`

`(displayS simpleDoc)` takes the output `simpleDoc` from a rendering function and transforms it to a `ShowS` type (for use in the `Show` class).

```
showWidth :: Int -> Doc -> String
showWidth w x   = displayS (renderPretty 0.4 w x) ""
```

displayIO `:: Handle -> SimpleDoc -> IO ()`

`(displayIO handle simpleDoc)` writes `simpleDoc` to the file handle `handle`. This function is used for example by `hPutDoc`:

```
hPutDoc handle doc  = displayIO handle (renderPretty 0.4 100 doc)
```

References

John Hughes. (1995) *The design of a pretty-printer library.* In J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag LNCS **925**.
http://www.cs.chalmers.se/~rjmh/Papers/pretty.ps.

Simon Peyton Jones. (1997) *A Haskell pretty-printer library.*
http://www.research.microsoft.com/~simonpj/Haskell/pretty.html.

Doaitse Swierstra, Pablo Azero Alcocer and Joao Saraiva. (1998) *Designing and Implementing Combinator Languages.* Advanced Functional Programming, Springer-Verlag, LNCS **1608**:150-206.
http://www.cs.uu.nl/groups/ST/Software.

Philip Wadler. (April 1997, revised March 1998) *A prettier printer.* Draft paper.
http://cm.bell-labs.com/cm/cs/who/wadler/topics/language-design.html.

Index