Text.PrettyPrinting Documentation

Release 1.0.1

Quildreen Motta

December 08, 2015

Contents

Stability stable

Portability Portable

This module is an implementation of Wadler's Pretty Printer. As described in the paper, the pretty printer is an efficient algebra that supports multiple adaptable layouts according to the available space.

CHAPTER 1

Dependencies

```
var { Base }  = require('adt-simple');
var { curry } = require('core.lambda');
var { Done, trampoline, done, ternary, binary, nary } = require('./tramp')
```

CHAPTER 2

Data structures

The pretty printer uses two algebras:

The simple pretty printer is an algebra for documents that represents them as a concatenation of items. Each item may be one of the following:

```
union Doc {
  Nil,                  // Nothing, used as identity;
  Text(String, Doc),    // Some text;
  Line(Number, Doc)     // A line break, indented by a given amount;
} deriving (Base)
```

To allow (efficient) alternative layouts, a different algebra for documents is used, where we see things as a set of possible layouts for the same document.

In this version we have more tags, since we add explicit representations for concatenation and nesting, and make some internal assumptions for the sake efficiency. These assumptions must be preserved whenever documents are built, and to enforce that, users are not given these structures to work with directly.

```
union DOC {
  NIL,                  // Nothing, used as identity;
  CONCAT(DOC, DOC),     // Joins two documents horizontally;
  NEST(Number, DOC),    // Indents a document by a given amount;
  TEXT(String),         // Represents plain text;
  LINE,                 // Represents explicit line breaks;
  UNION(DOC, DOC)       // A set of possible layouts for a document;
} deriving (Base)

DOC::concat = function(aDOC) {
  return CONCAT(this, aDOC)
}
DOC::nest = function(indent) {
  return NEST(indent, this)
}
```

CHAPTER 3

Helpers

3.1 Repeat()

Text.PrettyPrinting.**Repeat**()

```
Int, String → String
```

Returns a String with *s* repeated *n* times.

```
function repeat(n, s) {
  return Array(n + 1).join(s)
}
```

3.2 flatten()

Text.PrettyPrinting.**flatten**()

```
DOC → DOC
```

Flatten replaces line breaks in a set of layouts by a single whitespace. It's defined privately so the invariants may hold.

```
function flatten {
  NIL            => NIL,
  CONCAT(a, b)   => CONCAT(flatten(a), flatten(b)),
  NEST(depth, a) => NEST(depth, flatten(a)),
  TEXT(s)        => TEXT(s),
  LINE           => TEXT(" "),
  UNION(a, b)    => flatten(a),
  a              => (function(){ throw new Error("No match: " + a) })();
}
```

3.3 best()

Text.PrettyPrinting.**best**()

```
Int, Int, DOC → DOC
```

Best chooses the best-looking alternative from a set of possible layouts a document may have. It takes into account the available layout for the rest of the document when doing so.

```
function best(width, indentation, doc) {
  return trampoline(go(width, indentation, [[0, doc]]));
```

go()

Text.PrettyPrinting.**go**()

```
Int, Int, (Int, DOC) → Doc
```

```
function go(w, k, x) {
  return match x {
    []                          => done(Nil),
    [[i, NIL], ...xs]           => ternary(go, w, k, xs),
    [[i, CONCAT(x, y)], ...xs] => ternary(go, w, k, [[i, x], [i, y]] +++ xs),
    [[i, NEST(j, x)], ...xs]    => ternary(go, w, k, [[i + j, x]] +++ xs),
    [[i, TEXT(s)], ...xs]       => binary(_text, s, go(w, k + s.length, xs)),
    [[i, LINE], ...xs]          => binary(_line, i, go(w, i, xs)),
    [[i, UNION(x, y)], ...xs]   => better(w, k,
                                          ternary(go, w, k, [[i, x]] +++ xs),
                                          λ[ternary(go, w, k, [[i, y]] +++ xs)]
                                         )
  }
}
```

_text()

Text.PrettyPrinting.**_text**()

```
String, Continuation
```

Wraps the Text() constructor for trampolining.

```
function _text(s, g) {
  if (g instanceof Done) {
    return done(Text(s, g.value))
  } else {
    return binary(_text, s, g.apply())
  }
}
```

_line()

Text.PrettyPrinting.**_line**()

```
Int, Continuation
```

Wraps the Line() constructor for trampolining.

```
function _line(i, g) {
  if (g instanceof Done) {
    return done(Line(i, g.value))
  } else {
    return binary(_line, i, g.apply())
  }
}
```

go()

Text.PrettyPrinting.**go**()

```
Int, Int, Doc, (Unit → Doc) → Doc
```

Chooses the best-looking of two styles. `y` is thunked to avoid costly computations.

```
function better(w, k, x, y) {
  if (x instanceof Done) {
    return fits(w - k, x.value)? done(x.value) : y()
  } else {
    return nary(better, [w, k, x.apply(), y])
  }
}
```

fits()

Text.PrettyPrinting.**fits**()

```
Int, Doc → Boolean
```

Checks if some document fits in the rest of the line.

```
  function fits {
    (w, x) if w < 0 => false,
    (w, Nil)        => true,
    (w, Text(s, x)) => fits(w - s.length, x),
    (w, Line(i, x)) => true,
    (w, x) => (function(){ throw new Error("No match: " + show(w) + ", " + show(x)) })()
  }
}
```

3.4 layout()

Text.PrettyPrinting.**layout**()

```
Doc → String
```

Converts a simple document to a string.

```
function layout {
  Nil        => "",
  Text(s, a) => s + layout(a),
  Line(i, a) => '\n' + repeat(i, ' ') + layout(a)
}
```

3.5 horizontalConcat()

Text.PrettyPrinting.**horizontalConcat**()

```
DOC, DOC → DOC
```

Concatenates two documents horizontally, separated by a single space.

```
function horizontalConcat(x, y) {
  return x +++ text(" ") +++ y
}
```

3.6 verticalConcat()

Text.PrettyPrinting.**verticalConcat**()

```
DOC, DOC → DOC
```

Concatenates two documents vertically, separated by a new line.

```
function verticalConcat(x, y) {
  return x +++ line() +++ y
}
```

3.7 words()

Text.PrettyPrinting.**words**()

```
String → Array(String)
```

Converts a string into a list of words.

```
function words(s) {
  return s.split(/\s+/)
}
```

CHAPTER 4

Primitives

Wadler's pretty printer define several primitive functions for working with the two aforementioned algebras. Combinators can be easily derived from these basic building blocks (and a few area also provided).

4.1 nil()

Text.PrettyPrinting.**nil**()

```
Unit → DOC
```

Constructs an empty document.

Example:

```
pretty(80, nil()) // => ""
```

```
function nil() {
  return NIL
}
```

4.2 concat()

Text.PrettyPrinting.**concat**()

```
DOC → DOC → DOC
```

Joins two documents horizontally, without any separation between them.

Example:

```
pretty(80, concat(text('a'), text('b'))) // => "ab"
pretty(80, concat(text('a'), nil()))     // => "a"
```

```
function concat(a, b) {
  return CONCAT(a, b)
}
```

4.3 nest()

`Text.PrettyPrinting.nest()`

```
Int → DOC → DOC
```

Indents a document a given amount of spaces.

Example:

```
pretty(80, stack([
  text('a'),
  text('b'),
  text('c')
]))
// => "a\n    b\n    c"
```

```
function nest(depth, a) {
  return NEST(depth, a)
}
```

4.4 text()

`Text.PrettyPrinting.text()`

```
String → DOC
```

Represents plain text in a document.

Example:

```
pretty(80, text("a")) // => "a"
```

```
function text(s) {
  return TEXT(s)
}
```

4.5 line()

`Text.PrettyPrinting.line()`

```
Unit → DOC
```

Represents a line break in a document.

Example:

```
pretty(80, concat(concat(text("a"), line()), text("b"))
// => "a\nb"
```

```
function line() {
  return LINE
}
```

4.6 group()

Text.PrettyPrinting.**group**()

```
DOC → DOC
```

Creates a set of alternative layouts for the document.

Example:

```
pretty(5, group(stack([text('foo'), text('bar')])))
// => "foo\nbar"

pretty(10, group(stack([text('foo'), text('bar')])))
// => "foo bar"
```

```
function group(a) {
  return UNION(flatten(a), a)
}
```

CHAPTER 5

Conversions

5.1 pretty()

`Text.PrettyPrinting.pretty()`

```
Int → DOC → String
```

Returns the best representation of a document for the given amount of horizontal space available, as a String.

Example:

```
pretty(80, spread([text('hello'), text('world')]))
// => "hello world"
```

```
function pretty(width, doc) {
  return layout(best(width, 0, doc))
}
```

CHAPTER 6

Combinators

6.1 foldDoc()

Text.PrettyPrinting.**foldDoc**()

```
(DOC, DOC → DOC) → Array(DOC) → DOC
```

Allows folding over pairs of documents (similar to a catamorphism).

```
function foldDoc {
  (f, [])          => nil(),
  (f, [x])         => x,
  (f, [x, ...xs]) => f(x, foldDoc(f, xs))
}
```

6.2 spread()

Text.PrettyPrinting.**spread**()

```
Array(DOC) → DOC
```

Lays out a series of documents horizontally, with each document separated by a single space.

Example:

```
pretty(80, spread([text('foo'), text('bar')]))
// => "foo bar"
```

```
function spread(xs) {
  return foldDoc(horizontalConcat, xs)
}
```

6.3 stack()

Text.PrettyPrinting.**stack**()

```
Array(DOC) → DOC
```

Lays out a series of documents vertically, with each document separated by a single new line.

Example:

```
pretty(80, stack([text('foo'), text('bar')]))
// => "foo\nbar"
```

```
function stack(xs) {
  return foldDoc(verticalConcat, xs)
}
```

6.4 bracket()

`Text.PrettyPrinting.`**`bracket`**`()`

```
Int → DOC → DOC → DOC → DOC
```

Example:

```
pretty(5, bracket(2, '[', stack([
  text('a'), text('b'), text('c')
]), ']'))
// => "[\n  a\n  b\n  c \n]"
```

```
function bracket(indent, left, x, right) {
  return group(text(left)
           +++ nest(indent, line() +++ x)
           +++ line()
           +++ text(right))
}
```

6.5 join()

`Text.PrettyPrinting.`**`join`**`()`

```
DOC → DOC → DOC
```

Joins two documents together, either by separating with a single horizontal space or a single new line.

```
function join(x, y) {
  return x +++ UNION(text(" "), line()) +++ y
}
```

6.6 fillWords()

`Text.PrettyPrinting.`**`fillWords`**`()`

```
String → DOC
```

Makes the best use of the available space for laying out words, either separated by a space or a new line.

```
function fillWords(s) {
  return foldDoc(join, words(s).map(text))
}
```

6.7 fill()

Text.PrettyPrinting.**fill**()

```
Array(DOC) → DOC
```

Makes the best use of the available space for layout out a series of documents, either separated by a space or a new line.

```
function fill {
  []             => nil(),
  [x]            => x,
  [x, y, ...zs] => UNION(horizontalConcat(flatten(x), fill([flatten(y)] +++ zs)),
                         verticalConcat(x, fill([y] +++ zs)))
}
```

CHAPTER 7

Exports

```
module.exports = {
  nil       : nil,
  concat    : curry(2, concat),
  nest      : curry(2, nest),
  text      : text,
  line      : line,
  group     : group,
  pretty    : curry(2, pretty),
  foldDoc   : curry(2, foldDoc),
  spread    : spread,
  stack     : stack,
  bracket   : curry(4, bracket),
  join      : curry(2, join),
  fillWords : fillWords,
  fill      : fill
}
```

t

Symbols

B

C

F

G

H

J

L

N

P

R

S

T

V

W