Strictly Pretty

Christian Lindig
Gärtner Datensysteme GbR
Hamburger Str. 273a
D-38 114 Braunschweig, Germany
`lindig@gaertner.de`

March 6, 2000

Abstract

Pretty printers are tools for formatting structured text. A recently taken algebraic approach has lead to a systematic design of pretty printers. Wadler has proposed such an algebraic pretty printer together with an implementation for the lazy functional language Haskell. The original design causes exponential complexity when literally used in a strict language. This note recalls some properties of Wadler's pretty printer on an operational level and presents an efficient implementation for the strict functional language Objective Caml.

1 Introduction

Pretty printing is the problem of finding a good layout for structured text under some constraints. John Huges has advanced the design of pretty printers considerably by taking an algebraic approach [1]: Pretty printers are a set of operators like horizontal or vertical concatenation which obey algebraic rules. This has lead to a consistent pretty printing library; a variant of his library is part of some Haskell [1] implementations. Based on Hughes' work Philip Wadler has proposed another algebraic pretty printer [4, 5]. It uses only six operators and a uniform document model that is well suited for pretty printing tree structures like source code. Like Hughes' he has also suggested an implementation of his approach in the functional language Haskell. It relies heavily on the lazy evaluation of Haskell and can not be easily ported to a strict language without loss of efficiency. This note recalls the properties of Wadler's pretty printer on an operational level and proposes an implementation for strict languages. The implementation uses the strict functional language *Objective Caml* [2], and can easily be ported to Standard ML [3].

2 Philip Wadler's Pretty Printer

From a user's perspective documents are made up from the six constructors shown in figure 1. Simple documents consist of string literals `"hello"` which are separated by optional line breaks (␣) and glued together with the cons (·) operator: `"hello"`·␣·`"world"`. For the moment the line operator ␣ should be thought of as a space character which may be replaced by a line break when necessary. Documents can be structured by groups: [`"begin"` · ␣ · [`"stmt;"` · ␣ · `"stmt;"` · ␣ · `"stmt;"`] · ␣ · `"end"`]. The nest- and nil-operator will be explained shortly.

$$
\begin{array}{lcll}
doc & ::= & \emptyset & \text{(nil)} \\
 & | & \texttt{"}\textit{string}\texttt{"} & \text{(text)} \\
 & | & \_ & \text{(line)} \\
 & | & doc \cdot doc & \text{(cons, right associative)} \\
 & | & {}^{int}\langle\, doc \,\rangle & \text{(nest)} \\
 & | & [\, doc \,] & \text{(group)}
\end{array}
$$

Figure 1: Six constructors for the *doc* data type

Groups in conjunction with the optional line breaks (_) are the key to different layouts according to the available space. For pretty printing all optional line breaks inside a group are either turned into a space or into a newline. The decision for each group affects all the line breaks of a group at a whole but is made for subgroups individually. Literal text (`"hello"`) is simply printed as is and the nil-constructor is not printed at all. The following layout policy describes how to process a group:

1. Print every optional line break of the current group and all its subgroups as spaces. If the current group then fits completely into the remaining space of current line this is the layout of the group.
2. If the former fails every optional line break of the current group is printed as a newline. Subgroups and their line breaks, however, are considered individually as they are reached by the pretty printing process.

Applying this policy to the istructured document from above at different line lengths yields the following results. For clarity groups in the output are still marked with square brackets. With a line width of 60 characters the document fits on a single line. All optional breaks in all groups can be printed as spaces:

```
[begin [stmt; stmt; stmt;] end]
```

On a line 30 characters wide the outermost group must print its breaks as newlines but the inner group can use spaces:

```
[begin
[stmt; stmt; stmt;]
end]
```

And on a 10 characters wide line even the breaks of the inner group must be turned into newlines:

```
[begin
[stmt;
stmt;
stmt;]
end]
```

The layout policy tries to print a document flat by printing optional line breaks inside groups as spaces. If this is not possible the outermost group is printed with breaks as newline to make room for inner groups. Inside a flat group (breaks printed as spaces) all subgroups are always flat, too.

The document layout from above lacks nice indentation to indicate its structure. It can be provided through the nest-operator $^{i}\langle\, doc \,\rangle$: when an optional line break gets printed as a newline it is followed by a number of spaces to indent the next line. If the line break comes out as a space no additional spaces are added. The number of spaces is controlled by the nest-operator $^{i}\langle\, d \,\rangle$: all breaks inside document d that are printed as newlines are followed by i spaces. Of course, the nest operator nests properly (cf. [4, 5]) and is independent from the grouping operator unlike some other pretty printers. To achieve a typical layout for the example we have to add a nest operator:

$$[\,\texttt{"begin"} \cdot {}^{3}\langle\, \_ \cdot [\,\texttt{"stmt;"} \cdot \_ \cdot \texttt{"stmt;"} \cdot \_ \cdot \texttt{"stmt;"}\,]\,\rangle \cdot \_ \cdot \texttt{"end"}\,]$$

When the inner group must be broken (line breaks as newlines) it gets indented by three spaces. Below is the output of this document for a line width of 50, 30, and 10 characters.

```
[begin [stmt; stmt; stmt;] end]  [begin                   [begin
                                    [stmt; stmt; stmt;]      [stmt;
                                 end]                        stmt;
                                                             stmt;]
                                                          end]
```

The nil-operator $\emptyset$ is hardly necessary when documents are constructed manually like in the example above. But it is essential to implement optional output: *if* ... *then* `"output"` *else* $\emptyset$; $\emptyset$ is mapped to the empty string by the pretty printer.

3 Implementation

This section presents an implementation of the pretty printer in Objective Caml and explains the differences to the original design. However,many of its aspects follow the Haskell implementation suggested by Wadler [4, 5].

The six constructors that make up a document are captured by `doc`. To add a little flexibility breaks have a user defined representation.

⟨*pp.ml*⟩≡
  ⟨*preliminaries*⟩

```
  type doc =
      | DocNil
      | DocCons           of doc * doc
      | DocText           of string
      | DocNest           of int * doc
      | DocBreak          of string
      | DocGroup          of doc
```

Constructor functions provide help to build documents easier. The infix operator `^^` is right associative. In Objective Caml associativity and precedence are determined by the first character of an operator so no extra declaration is necessary.

⟨*pp.ml*⟩+≡
```
let (^^) x y            = DocCons(x,y)
let empty               = DocNil
let text s              = DocText(s)
let nest i x            = DocNest(i,x)
let break               = DocBreak(" ")
let breakWith s         = DocBreak(s)
let group d             = DocGroup(d)
```

The explicit implementation of groups is the main difference to the lazy implementation. It encodes groups implicitly by unfolding a group into two alternative documents: a *flat* one, where all breaks are rendered as spaces; and a *broken* one, where all breaks are rendered as newlines. Since this expansion is done lazily it does not lead to an exponential growth as it would do in a strict language. To avoid the exponential growth with the number of nested groups the strict implementation must encode groups explicitly.

Documents of type `doc` are not printed directly but transformed into simpler documents of type `sdoc`. During this transformation the layout for each group is decided, as `sdoc` does no longer provide groups.

⟨*pp.ml*⟩+≡
```
type sdoc =
    | SNil
    | SText             of string * sdoc
    | SLine             of int    * sdoc    (* newline + spaces *)
```

A simple document is either empty `SNil`, consists of a string which is followed by another simple document `SText`, or is a newline followed by a number of spaces and then another simple document `SLine`. Wadler has shown that every complex document can be transformed into an equivalent simple document which is straight forward to print[1].

⟨*pp.ml*⟩+≡
```
let rec sdocToString = function
    | SNil              -> ""
    | SText(s,d)        -> s ^ sdocToString d
    | SLine(i,d)        -> let prefix = String.make i ' '
                           in  nl ^ prefix ^ sdocToString d
```

[1]In Objective Caml `^` concatenates strings and `String.make` i c creates a string of i characters c.

The transformation of a complex document into a simple document must decide whether a group is broken (line breaks as newlines) or flattened (line breaks as spaces). An efficient predicate `fits` checks whether a flat document fits completely into w characters. It does this by expanding the document and consuming characters as it goes. All line breaks are regarded as spaces since the width must be checked of the flat document according to the layout policy; this also holds for line breaks in subgroups.

The `fits` predicate actually checks a list of triples because the cons-operator is unfolded into a list. Each triple `(i,m,d)` hold the current indentation `i`, the mode `m` of the current group and the document `d`. The function can stop after it has w characters are consumed or the document ended – whatever happens first. So at most w characters must be consumed. Since a group is checked when rendered flat it never can contain a break which indicates a newline.

⟨*pp.ml*⟩+≡

```
  ⟨mode of a group⟩

  let rec fits w = function
      | _ when w < 0                    -> false
      | []                              -> true
      | (i,m,DocNil)               :: z -> fits w z
      | (i,m,DocCons(x,y))         :: z -> fits w ((i,m,x)::(i,m,y)::z)
      | (i,m,DocNest(j,x))         :: z -> fits w ((i+j,m,x)::z)
      | (i,m,DocText(s))           :: z -> fits (w - strlen s) z
      | (i,Flat, DocBreak(s))      :: z -> fits (w - strlen s) z
      | (i,Break,DocBreak(_))      :: z -> true (* impossible *)
      | (i,m,DocGroup(x))          :: z -> fits w ((i,Flat,x)::z)
```

The `fits` function of the lazy implementation does not work on complex documents but on simple documents instead. The laziness of Haskell permits to transform the alternative group variants into simple documents and check them which is easier than checking complex documents. The strict implementation does not expand groups beforehand and thus must check the width of complex documents.

Formatting a complex document into a simple document requires to maintain indentation informations and whether line breaks inside a group are printed as spaces or newlines. Every element of a complex document can be either part of a flat or broken group; this is captured by a `mode`.

⟨*mode of a group*⟩≡

```
  type mode =
      | Flat
      | Break
```

When an optional line break is encountered and it is turned into a newline (i.e. the line break is part of a broken group) we must know how many spaces are to be printed after the newline for indentation of the next line. So the actual mode `m` and indentation level `i` are paired with every element by the `format` function. Its parameter `w` denotes the actual line length and the parameter `k` how much characters of the current line have already been consumed.

⟨*pp.ml*⟩+≡
```
let rec format w k = function
    | []                                 -> SNil
    | (i,m,DocNil)              :: z -> format w k z
    | (i,m,DocCons(x,y))        :: z -> format w k ((i,m,x)::(i,m,y)::z)
    | (i,m,DocNest(j,x))        :: z -> format w k ((i+j,m,x)::z)
    | (i,m,DocText(s))          :: z -> SText(s,format w (k + strlen s) z)
    | (i,Flat, DocBreak(s))     :: z -> SText(s,format w (k + strlen s) z)
    | (i,Break,DocBreak(s))     :: z -> SLine(i,format w i z)
    | (i,m,DocGroup(x))         :: z -> if fits (w-k) ((i,Flat,x)::z)
                                        then format w k ((i,Flat ,x)::z)
                                        else format w k ((i,Break,x)::z)
```

The indentation of two cons'ed elements never differs so the actual indentation is distributed over `DocCons` elements. The same applies for the mode `m`. Indentation only changes upon entering a nested document and the mode upon entering a group. The `fits` predicate determines the mode for a group of elements. Line breaks are turned into spaces or `SLine`'s accordingly.

4 Examples

As an example for a document a *if–then–else* expression is build. For using the pretty printing functions some additional functions are helpful. The infix operator `^|` connects two documents with an optional line break and `binop` builds a binary *left* ⊕ *right* expression.

⟨*pp.ml*⟩+≡
```
let (^|) x y            = match x,y with
                              | DocNil, _    -> y
                              | _, DocNil    -> x
                              | _, _         -> x ^^ break ^^ y

let binop left op right = group (nest 2
                                    ( group (text left ^| text op)
                                   ^| text right
                                    )
                                 )
```

The example `doc` contains quite a number of groups for demonstration purposes which result in many possible layouts. Whether such a flexible layout is adequate is debatable; in most cases space constraints will be less tight and layout consistency more important. This can be achieved by using fewer groups.

⟨*pp.ml*⟩+≡
```
let cond                = binop "a" "==" "b"
let expr1               = binop "a" "<<" "2"
let expr2               = binop "a" "+"  "b"

let ifthen c e1 e2      = group ( group (nest 2 (text "if"   ^| c ))
                                ^| group (nest 2 (text "then" ^| e1))
                                ^| group (nest 2 (text "else" ^| e2))
                                 )

let doc                 = ifthen cond expr1 expr2
```

Six different layouts are shown below together with the width the expression was formatted for. The minimal line width is 5 where every text element will be on a separate line. Formatting for even smaller line widths will lead to the same layout and thus over-long lines.

```
<------------- 32 ------------> <---- 15  ----> <-- 10 -->
if a == b then a << 2 else a + b if a == b       if a == b
                                 then a << 2     then
                                 else a + b        a << 2
                                                 else a + b

<-  8 ->                         <- 7 ->         <- 6->
if                               if              if
  a == b                           a ==            a ==
then                                 2               b
  a << 2                         then            then
else                               a <<            a <<
  a + b                              2               2
                                 else            else
                                   a + b           a +
                                                     b
```

5 Conclusion

An implementation of Wadler's pretty printer in a strict language must avoid a literal translation of the original implementation because of its exponential complexity. The implementation given here implements groups of documents directly at the price of a slightly more complicated predicate for checking the width of a document. Additionally each element of a document is associated with a mode to capture the two different behaviors of a group. The original implementation expands a group into two alternative variants beforehand and thus does not need to distinguish them explicitly. As already noted by Wadler the proposed pretty printer is small, predictable, and flexible.

Acknowledgements A previous version of the pretty printer grew out of a joint work with Andreas Rossberg and Franz-Josef Grosch.

References

[1] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[2] Xavier Leroy. Objective Caml. Implementation and documentation of the *Objective Caml* system. `http://pauillac.inria.fr/ocaml/`.

[3] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[4] Philip Wadler. A prettier printer. Technical report, Bell Labs, Lucent Technologies, 1998. Available from the author's home page: `http://cm.bell-labs.com/cm/cs/who/wadler/`.

[5] Philip Wadler. A prettier printer. *Journal of Functional Programming*, 1999. To appear.

A Missing Parts

The source code presented in this document makes up a complete Objective Caml source file. In the previous sections some definitions have been omitted for clarity; to complete the source code they are shown here.

⟨*preliminaries*⟩≡
```
let strlen  = String.length
let nl      = "\n"
```

The document `doc` to be printed is enclosed on the outermost level by a virtual group where all breaks are printed as spaces. To make the document independent of this decision it is enclosed into another group. Parameter `w` determines the available line length document `doc` is formatted for.

⟨*pp.ml*⟩+≡
```
let pretty w doc =
    let sdoc = format w 0 [0,Flat,DocGroup(doc)]  in
    let str  = sdocToString sdoc        in
        print_endline str
```