Linear, bounded, functional pretty-printing

S. DOAITSE SWIERSTRA and OLAF CHITIL

Journal of Functional Programming / Volume 19 / Issue 01 / January 2009, pp 1 - 16
DOI: 10.1017/S0956796808006990, Published online: 07 October 2008

JFP **19** (1): 1–16, 2009.
doi:10.1017/S0956796808006990 First published online 7 October 2008 Printed in the United Kingdom

FUNCTIONAL PEARL
Linear, bounded, functional pretty-printing

S. DOAITSE SWIERSTRA
Utrecht University, The Netherlands

OLAF CHITIL
University of Kent, UK

Abstract

We present two implementations of Oppen's pretty-printing algorithm in Haskell that meet the efficiency of Oppen's imperative solution but have a simpler and a clear structure. We start with an implementation that uses lazy evaluation to simulate two co-operating processes. Then we present an implementation that uses higher-order functions for delimited continuations to simulate co-routines with explicit scheduling.

1 Introduction

Over 25 years ago, Derek Oppen (1980) published an imperative pretty-printer that formats a document nicely within a given width. The algorithm is efficient: it takes time linear in the size of the input and is independent of the given width. Furthermore, the algorithm is *optimally bounded*, that is, for a partial input it already produces that part of the output for which no further inspection of the input is necessary. Oppen's work inspired numerous pretty-printing libraries, in particular several Haskell libraries (Hughes, 1995; Peyton Jones, 1997; Wadler, 2003); all of these, however, are less efficient than Oppen's libraries. Then Chitil (2001, 2005) presented a purely functional Haskell implementation that has all the nice properties of Oppen's original algorithm. That implementation, however, uses an intricate lazy coupling of two double-ended queues; it is quite complex and requires a special, modified implementation of double-ended queues.

The key problem is that information about *what* is to be printed and information about *how* it is to be printed does not become available at the same time. In this pearl we present more straightforward implementations. Section 3.2 describes a solution that makes sure that the information about *how* to format groups of text is passed to the place where we know *what* to format; Section 3.3 describes a solution that builds functions that know *what* to print and calls these functions once it is known *how* to format. Our first solution (Swierstra, 2004) relies heavily on lazy evaluation, whereas in the last one (Chitil, 2006) the scheduling of the necessary computation has been made completely explicit.

2 Problem description

2.1 The basic combinators

We will present the different versions—each of which can be seen as a deforested interpreter of a data type describing the structure to be printed—of our algorithm as instances of the following class, which closely follows the interface introduced by Wadler (2003):

```
type Indent = Int    -- zero or positive
type Width  = Int    -- positive
type Layout = String

class Doc d where
  text   :: String          → d
  line   ::                   d
  group  ::   d             → d
  (<>)   ::   d    → d      → d
  nest   :: Indent → d      → d
  pretty :: Width  → d      → Layout
  nil    ::                   d
  nil           =  text ""

prettyIO        :: Doc d ⇒ Width → d → IO ()
prettyIO w d = putStrLn (pretty w d)
```

Each instance of the class *Doc* describes a way to format documents within a given line width (to be referred to as w). The function *text* produces a primitive document containing just the *String* argument, *line* indicates a potential line break, and the operator <> concatenates two documents. The function *nest* is used to control indentation; it increments the indentation of the document in its second argument by its first argument. Finally, the function *pretty* renders a document of type d given a width of type *Width*, and *prettyIO* finally prints it.

How a document is to be formatted is governed by the *group* and *line* combinators. All *line* markers directly contained in a group are to be either formatted as a space or as a newline with indentation. In the first case, we say that the group is formatted *horizontally*, otherwise *vertically*. All groups contained in a horizontally formatted group are to be formatted horizontally too. All "top level" line markers are to be printed as newlines, i.e., the implicit top group is to be formatted vertically.

The problem to be solved is to find the "best" layout from the set of layouts described by a document. We define what is "best" in Section 2.2. Some might consider the best layout to be the one that uses the least number of lines. However, such an optimality criterion does not admit any bounded implementation; the end of a document can influence a layout decision at the very beginning (Hughes, 1995). An efficient algorithm computing such a shortest layout has been given by Swierstra *et al.* (1999). Here, we will consider greedy algorithms.

Example. We can define a simple layout for lists of integers

```
toDoc :: Doc d ⇒ [Int] → d
toDoc =  (text "["<>)
         ∘ foldr (<>) (text "]")
         ∘ intersperse (group (text "," <> line))
         ∘ map (text ∘ show)
```

which gives the following result:

```
> prettyIO 60 (toDoc [1..40])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33, 34, 35, 36, 37, 38, 39, 40]
```

Because each *line* marker is contained in a separate *group*, it is formatted horizontally if and only if the text up to the next *line* marker still fits on the current line, and vertically otherwise. (The combination of *group* and *line* corresponds to the "inconsistent blank" of Section 5 in Oppen, 1980.) Further examples can be found in Wadler (2003).

2.2 Straightforward implementations

First, we present a specification of our algorithm in a number of steps. We start with a basic specification. This specification is then refined to make it comply with Oppen's original specification. Next, we introduce extra efficiency requirements, which make the problem harder. The solutions to these new problems form the core content of this paper.

2.2.1 Basic specification

We can produce our layout by a simple pre-order traversal of the document tree, i.e., the tree representing the *group* structure. During this traversal we keep track of space remaining on the current output line. At the end of each group we check whether it fits in the space available for this group.

To determine whether a group fits in the remaining space on a line, we compute its total length, i.e., the sum of all the lengths s_k of the *text* elements and the number of *line* markers contained in the group, as if the whole document has been formatted horizontally. Because we want to compute sizes s_l for many segments, we maintain the accumulated length p_l for which the following invariant holds:

$$\sum_{i \leqslant k < j} s_k = p_j - p_i$$

Since the accumulated lengths of preceding elements correspond to the position of an element if the complete document were laid out horizontally we will refer to such values as positions:

```
type Position = Int
```

We start by defining an instance *Spec* of *Doc*, which models a document as a function of four parameters and three results:

```
type Remaining = Int
type Horizontal = Bool
type Spec      =      (Indent, Width) → Horizontal
                  →  Position → Remaining
                  → (Position  ,  Remaining, Layout)
```

To reduce the number of arguments in the algorithms given later, we have tupled the indentation of the document with the global *Width*. The argument of type *Position* is the position at the beginning of the represented document, and the result *Position* is the position at the end. The *Horizontal* argument indicates whether the embracing group is to be formatted horizontally or vertically. The *Remaining* values keep track of the free space on the "current output line": the argument tells us how much is available at the beginning and the result tells how much is still left at the end of the "current document." Our basic specification now reads as follows:

```
instance Doc Spec where
  text t      iw h p r = (p + l, r − l, t) where l = length t
  line        iw h p r = (p + 1, r_l, l_l)    where (r_l, l_l) = newLine iw h r
  (d_l <> d_r) iw h p r = (p_r, r_r, l_l ++ l_r)
                          where (p_l, r_l, l_l) = d_l iw h p r
                                (p_r, r_r, l_r) = d_r iw h p_l r_l
  group d     iw h p r = let v@(p_d, _, _) = d iw (p_d − p ⩽ r) p r in v
  nest j d    (i, w)   = d (i + j, w)
  pretty w d           = let (_, _, l) = d (0, w) False 0 w in l
newLine (i, w) True  r = (r − 1, [' '])
newLine (i, w) False r = (w − i, '\n' : replicate i ' ')
```

This algorithm depends on lazy evaluation, because the definition of *group* uses a cyclic binding which both defines the endpoint p_d of group and uses it. This design pattern in which part of the result of a call is used to compute one of its arguments is also known from the famous *Repmin* problem (Bird, 1984). If the difference between the begin- and end-position of a group does not exceed the free space at the beginning of the group ($p_d - p \leqslant r$), we can format the group horizontally, otherwise we have to resort to vertical formatting; we say that in such a case the group extends beyond its *maximal endpoint* $p + r$. In the definition of *pretty* the whole document is applied to *False*, expressing that *line* markers appearing outside any group are always to be formatted as line breaks with an initial indentation of 0.

One might be tempted to combine the definition of *newline* with that of *line*, writing

```
line _      p True  r = (p + 1, r − 1, [' '])
line (i, w) p False r = (p + 1, w − i, '\n' : replicate i ' ')
```

This however fails. To choose between the two alternatives we need the value of the *Horizontal* argument; but this argument usually depends on an expression $p_d - p \leqslant r$, since this *line* may be a part of the group, so it depends on the final position p_d of this group, and this is a value which is returned by the call and thus cannot be used in deciding which alternative to take. Despite the fact that both alternatives

contribute the same value to the position $(p + 1)$, the last formulation of *line* forces us to make a choice between the two alternatives too early.

2.2.2 Normalizing documents

Although probably not immediately obvious the given specification may produce lines longer than w:

```
> prettyIO 6 (group (text "Hi" <> line <> text "you") <> text "!!!"))
Hi you!!!
```

whereas we would prefer:

```
Hi
you!!!
```

The cause of this behavior is that a group that still fits on a line may be followed by further text without a separating *line* marker, and thus will end up on the current line even if it extends beyond the end of the line; unfortunately the fact that a group fits does not imply that all its trailing *text* elements will also fit. In our example, formatting the preceding group vertically would have avoided lines becoming longer than w. A simple preprocessing step deals with this problem, by moving all *text* elements to the group to which their nearest preceding *line* marker belongs if it exists. There are two ways to look at our documents: either as sequences of *text* elements separated by *line* markers, or as tree structures built by *group* and *nest* operators, where each node contains additional *text* elements and *line* markers.

So before formatting we first apply a document transformation that moves all *text* elements such that no consecutive sequence of them extends beyond a group. The transformation is based on the following laws:

$$
\begin{aligned}
&\mathit{group}\ (\mathit{text}\ t <> d) = \mathit{text}\ t <> \mathit{group}\ d \\
&\mathit{group}\ d <> \mathit{text}\ t \quad = \mathit{group}\ (d <> \mathit{text}\ t) \\
&\mathit{nest}\ j\ (\mathit{text}\ t <> d) = \mathit{text}\ t <> \mathit{nest}\ j\ d \\
&\mathit{nest}\ j\ d <> \mathit{text}\ t \quad = \mathit{nest}\ j\ (d <> \mathit{text}\ t) \\
&(d_1 <> d_2) <> d_3 \quad = d_1 <> (d_2 <> d_3)
\end{aligned}
$$

We introduce $\mathit{Norm}\ d$ as a second instance of *Doc*. A $\mathit{Norm}\ d$ is a function returning two elements of type d: one containing the leading sequence of *text* elements to be included in a preceding group, and the other the rest of the document. Furthermore, each $\mathit{Norm}\ d$ takes an argument, containing the leading text of its successor:

$$\textbf{type}\ \mathit{Norm}\ d = d \to (d, d)$$

In this way we have introduced a backward traveling accumulating document, containing a sequence of text elements (passed to it predecessor in the argument tt, trailing text). At each *line* marker we insert these accumulated text elements. As a result, in a normalized document each group starts with a *line* marker if it contains elements at all.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Doc}\ d \Rightarrow \mathit{Doc}\ (\mathit{Norm}\ d)\ \textbf{where} \\
&\quad \mathit{text}\ t \qquad tt = (\mathit{text}\ t <> tt, \mathit{nil}\qquad\quad) \\
&\quad \mathit{line} \qquad\ tt = (\mathit{nil}\qquad\quad , \mathit{line} <> tt)
\end{aligned}
$$

$$
\begin{aligned}
(d_l <> d_r)\ tt &= \textbf{let}\ (td_l, sd_l) = d_l\ td_r \\
&\qquad\quad (td_r, sd_r) = d_r\ tt \\
&\quad\ \textbf{in}\ (td_l, sd_l <> sd_r) \\
group\ d \quad tt &= mapsnd\ group \quad (d\ tt) \\
nest\ j\ d \quad tt &= mapsnd\ (nest\ j)\ (d\ tt) \\
pretty\ w\ d \quad &= \textbf{let}\ (td, sd) = d\ nil\ \textbf{in}\ pretty\ w\ (td <> sd) \\
nil \qquad tt &= (tt, nil) \\
mapsnd\ f\ (x, y) &= (x, f\ y)
\end{aligned}
$$

2.3 Extra requirements

2.3.1 Optimally bounded

If the outer element of the document is a *group*, our straightforward algorithm *Spec* traverses the complete document tree before emitting any result; this is definitely undesirable for large documents. So we introduce some extra requirements.

Our straightforward algorithm *Spec* is fully strict, as the following computation demonstrates:

```
> prettyIO 4 (group (text "Hi" <> line <> text "you" <> undefined) :: Spec)
Program error: {undefined}
```

However, after having seen the strings `"Hi"` and `"you"` we can already conclude that together they do not fit in a line of width four. So output can be produced without inspecting any further elements, resulting in:

```
> prettyIO 4 (group (text "Hi" <> line <> text "you" <> undefined) :: ??? )
Hi
you
Program error: {undefined}
```

Based on a prefix of the document of size w we can always decide how to continue formatting, because any group wider than the width-limit has to be formatted vertically. We say that *pretty* is *bounded* if look-ahead into the input is limited by the width w. We require our final program to be even *optimally bounded*, i.e., any part of the output that can be produced without touching a $\perp$ element in the input has to be produced.[1] Our *Spec* and *Norm Spec* instances do not fulfill the boundedness requirements, because to determine the total horizontal size of a group it requires all group elements to be defined.

2.3.2 Complexity

Of course, we want our algorithms to be in the class $\mathcal{O}(n)$, where n is the number of elements in the input. However, with increased line width the length of the output, when seen as a single long string, may increase, because we will generally have deeper nestings and thus we may generate more white space. We could avoid this problem

[1] To be precise, we do not consider partial strings. The argument of *text* is considered as an atomic value that is added to the layout in one step.

by representing a layout as a list of indentation–line pairs, $[(Indent, String)]$, as in Hughes (1995). However, for simplicity and practical applicability we produce a single string and just consider generation of a sequence of white spaces as a constant time operation. Given this caveat our specification is linear and we have to ensure that this linearity remains fulfilled once we find optimally bounded solutions.

One may try to transform *Spec* into an optimally bounded version by having a document return a (lazily constructed) list $(ls :: [Int])$ containing the sizes of the *text* elements and *line* markers in a group in which the lengths of earlier elements come first, and by replacing the test $p_d - p \leqslant r$ by an incremental test ls `` `pruning` `` r, which fails as soon as the accumulated sizes exceed the available free space:

$$
\begin{aligned}
&\mathit{pruning} :: [\mathit{Int}] \to \mathit{Remaining} \to \mathit{Bool}\\
&(s : ss)\ \text{`}\mathit{pruning}\text{`}\ r = (s \leqslant r) \wedge ss\ \text{`}\mathit{pruning}\text{`}\ (r - s)\\
&[\,]\qquad \text{`}\mathit{pruning}\text{`}\ r = \mathit{True}
\end{aligned}
$$

Although this modification makes the algorithm optimally bounded, its complexity now suddenly depends on w, because the pruning is done for each group individually: because the result ls of a document in a group will be traversed by *pruning* when deciding whether this group fits, and is returned as part of the ls of the embracing groups, pruning with nested groups will traverse the same (parts of) lists. Especially with deeply nested groups this becomes a problem. For a document of the shape

$$\mathit{group}\ (\mathit{group}\ (\mathit{group}\ (....$$

the lists of the inner groups will be prefixes of their embracing groups. The pruning process will become a linear search for the first one that passes the *pruning* test. Thus the complexity of our solution becomes dependent on w.

At this point we may point out a subtlety. One might be inclined to think that if the function *pruning*, as part of the pruning process of the father group, consumes all the elements contributed by a subgroup without failing that subgroup will fit irrespective of the decisions taken for its ancestors. Unfortunately, this is not the case as the following example demonstrates:

```
prettyIO 15
  (group   (  text "this"
           <> nest 9 (line <> group (text "takes" <> line <> text "four"))
           <> line <> text "lines"))
```

which results in:

```
this
        takes
        four
lines
```

Because the outer group does not fit, the inner group is suddenly indented by nine spaces. As a consequence the inner group does not fit either! So in order to take a decision for an inner group, first we always have to decide whether its embracing group fits. Only then we will precisely know how much free space is still left on the line for this inner group.

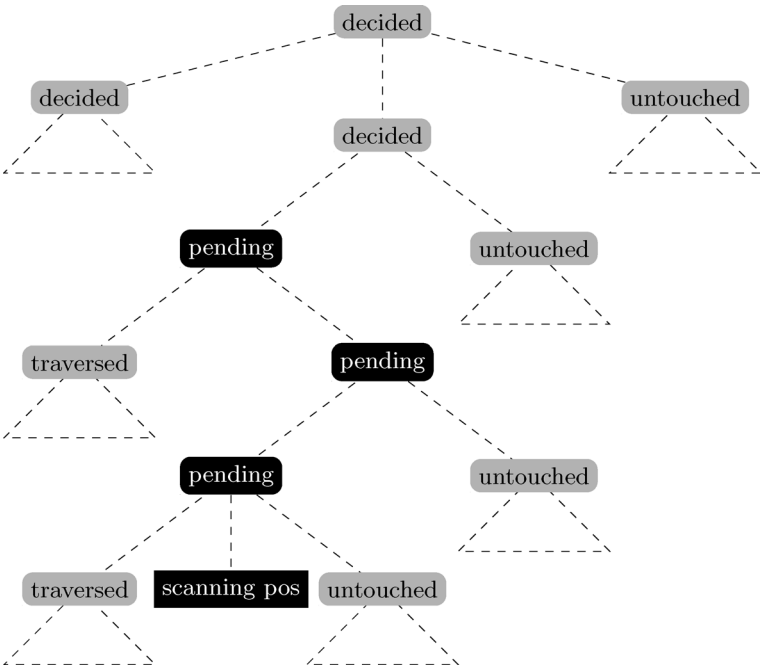

Fig. 1. The situation while pruning.

3 Solutions

In this section, we will present a sequence of solutions to the pretty-printing problem. Before going into the actual solutions we will explain why we will use double-ended queues in all our solutions.

3.1 Double-ended queues

We have seen how the idea of pruning avoids always scanning a complete group before deciding whether it fits or not. The problem is how to share the scanning of a group with the enclosed groups and thus to avoid the observed recomputations, because it is this aspect that makes the pruning solution depend on the width w. The fundamental idea, due to Oppen (1980), is to have two processes traverse the document: a *scanning* process determines for all groups whether they fit or not and a *printing* process uses that information to produce the pretty layout. Pruning ensures that the scanning process never goes far ahead of the printing process.

To explain more precisely what the two processes do, we refer to Figure 1. In this figure, we have sketched the *group structure* (without the *line* and *text* elements) of some example input. We distinguish four kinds of nodes: *decided*, *pending*, *traversed,* and *untouched*. The following are the kinds of nodes change while both processes traverse the tree in prefix order:

1. Nodes that form part of a group which we know how to format. We indicate these as *decided* nodes. A decided group can be traversed by the printing process up to the first *pending* or *traversed* subnode.
2. Nodes that have been traversed by the scanning process but are not decided yet; they come in two kinds:
   a. Nodes in groups that have been completely *traversed* by the scanning process.
   b. Nodes corresponding to the root of a subgroup we have entered, but not yet left. We call these nodes *pending*.
3. Nodes that have not been inspected yet (*untouched*).

In this figure we have indicated the path consisting of pending *group* nodes that leads from the top pending group to the point up to where the scanning process has proceeded. When traversing the tree, this path will grow and shrink as a result of pruning, and entering and leaving groups. We will refer to these nodes as the *dequeue*, as it is a double-ended queue that can grow and shrink at the end and shrink at the top. Each node of the second category corresponds to an open question: for the pending node we expect an answer from the scanning process, and for the traversed node we can decide once we know the decision for its father, so we can compute how much space is available for the group. In the latter case the situation is similar to the situation as handled in our original specification; *traversed* implies that the total horizontal space needed for the group is known.

The first problem we address is what extra information we have to maintain while investigating whether the top pending group fits, such that when we discover—when pruning—that it does not fit, we can continue with the investigation of the next pending group without reinspecting any values.

We focus on the path with nodes labeled *pending* in the figure. When scanning following things may happen:

- We may conclude that the top pending group does not fit. Then we can take the decision for all its traversed children in the same way as we did in *Spec*: we know their horizontal sizes. So we can print all the elements up to the group node that is next in the dequeue, without any further scanning. This next group becomes the new top pending group.
- We may have traversed the innermost group, i.e., the group corresponding to the last element of the dequeue; in this case we can mark the group as *traversed* and remove it from the end of the dequeue.
- We encounter a new group, in which we extend the dequeue at the end with an extra element.

We introduce the following pseudo-data-type for double-ended queues.[2] Okasaki's banker's dequeue implementation (Okasaki, 1998) supports all operations of $\mathcal{O}(1)$ amortized time.

[2] We will use the $\lhd$ only for matching, but never as a constructor.

```
data Dequeue e = ⟨⟩                -- the empty dequeue
               | e ◁ Dequeue e     -- prepend an element
               | Dequeue e ▷ e     -- append an element
```

When we introduced the function *pruning*, the values to be pruned were collected and moved to the start of the group; instead of this we build solutions in which we carry along a dequeue containing the current state of the scanning process, and we update this dequeue based on the result of pruning the values we encounter. We perform a little bit of the pruning work for the *top pending* node whenever we encounter a new element that takes up space.

We will now present a sequence of solutions, with increasing efficiency, but also an increasing intricacy.

3.2 Bringing the arguments to the printing functions

Let us suppose for a moment that the scanning process manages to compute the *Horizontal* (i.e., the Boolean indicating how to format a group) information efficiently by carrying the dequeue along when traversing the tree. Then this creates a new problem: the *Horizontal* values that become available while scanning have to be made available to the printing process.

Our first step is to extend the algorithm with the computation of the complete list of all needed *Horizontal* values (i.e., the Boolean values for all the groups), listed in prefix order. In the function *pretty* at the root of the overall computation we pass this list back as an argument to the root document, so it can be consumed in the printing process; this design pattern was also used in the *group* function in the *Spec* instance. The tricky part however is here that we also have an information flow in the other direction:

1. The printing process computes the layout. It consumes the list of *Horizontal* values and additionally returns *Remaining* values.
2. The pruning process computes the positions p, reads the *Remaining* values, and thus produces the list of *Horizontal* values.

We use lazy evaluation to schedule the two parallel processes, each producing output for the other.

Looking at the figure we see that for each group we have entered but not yet left the node we have in the dequeue. In this node we store the relevant information for each pending group: its maximal endpoint $p + r :: Position$ and its (lazily evaluated) *Horizontal* values for its *traversed* descendant groups.

```
type Horizontals = [Horizontal]
type P           = Horizontals   -- Produced by the pruning process
type C           = Horizontals   -- Consumed by the printing process
type Dq          = DeQueue (Position, Horizontals)
```

We extend our algorithm such that it carries along two extra threaded variables: a *Dq* on behalf of the pruning process and the list of unconsumed *Horizontals* on

behalf of the printing process. Furthermore, we return the global list of *Horizontals* to be used in *pretty* as part of the following result:

$$\textbf{type}\ State = (Position, Dq, C, Remaining)$$
$$\textbf{type}\ Lazy = (Indent, Width) \to State \to (State, Layout, P)$$

We define three functions that update the Dq structure, two of which may additionally return newly found *Horizontal* information:

$$enter\ ::Position \to Dq \to\ Dq$$
$$leave\ ::Position \to Dq \to (Dq, P)$$
$$prune\ ::Position \to Dq \to (Dq, P)$$

When descending into a new group we update the Dq by appending the maximal endpoint of this group, while at the same time recording that we have no information on its traversed children yet (the empty list $[\,]$):

$$enter\ mep\ q = q \rhd (mep, [\,])$$

The function *leave* updates the Dq and possibly returns newly found P. The function distinguishes three cases, based on the length of the dequeue:

0: We have already discovered that the group we are leaving does not fit, and so we learn nothing new.
$$leave\ p\ \langle\rangle = (\ \langle\rangle, [\,])$$

1: We are leaving the current group. Since this node was not pruned away yet, we conclude that the group fits. We also incorporate the *Horizontal*s computed for its children into the list of answers, so they show up in their correct position.
$$leave\ p\ (\ \langle\rangle \rhd (mep, hs)) = (\ \langle\rangle, True : hs)$$

>1: The last *pending* group changes status to *traversed*. We incorporate this information, together with the information about its children into the node of the group one level up, to be included later in the list of answers we are constructing:

$$leave\ p\ \ (pp \rhd (mep_2, hs_2) \rhd (mep_1, hs_1)) =$$
$$\qquad ((pp \rhd (mep_2, hs_2 +\!\!+ [p \leqslant mep_1\,] +\!\!+ hs_1), [\,])$$

The third function, *prune*, is called when we visit a *text* element or a *line* marker, because these are the only points where *Layout* is produced. The function *prune* compares the current position p with the maximal endpoint of the top pending group (if present). If this node still fits on the line, then we do nothing and return the dequeue q unmodified; if we have reached a point where we can conclude that the top pending group does not fit anymore once we include the next node, then we insert *False* in the list of *Horizontal*s we are producing, together with the information of the traversed groups (together in C); of course we have to continue pruning for the next pending group, which has become the topmost:

$$prune\ p\ \ \langle\rangle \qquad = \quad (\ \langle\rangle, [\,])$$
$$prune\ p\ q@((mep, hs) \lhd qq)$$
$$\qquad \mid p \leqslant mep = \quad (q, [\,])$$

```
  | True      =  let (q',                hs_new) = prune p qq
               in  (q', False : hs ++ hs_new)
```

Using these auxiliary functions we can now formulate a solution which fulfills all requirements. In the case of a group we remember the current head of the list of horizontal information, which tells us how the parent group is to be formatted, and put this back at the head of the *tail* of the returned value, from which the children have all taken away their *Horizontal* elements. The *tail* removes the *Horizontal* value for the current group, which has served its purpose and is thus no longer needed.[3]

```
instance Doc Lazy where
  text t      iw (p, dq, hs, r)
    = ((p + l, dq', hs, r − l), t, as) where l          = length t
                                            (dq', as)   = prune (p + l) dq
  line        iw (p, dq, hs, r)
    = ((p + 1, dq', hs, r'), l', as)    where (dq', as) = prune (p + 1) dq
                                            (r', l')    = newLine iw (head hs) r
  (d_l <> d_r) iw state
    = (state_r, l_l ++ l_r, as_l ++ as_r)  where (state_l, l_l, as_l) = d_l iw state
                                                 (state_r, l_r, as_r) = d_r iw state_l
  group d     iw (p, dq, ~(h : hs), r)
    = ((pe, dq', h : tail hsd, r_d), l_d, as_d ++ as')
      where (~(pe, dq_d, hsd, r_d), l_d, as_d)  = d iw (p, (enter (p + r) dq), hs, r)
            (dq', as') = leave pe dq_d
  nest j d    (i, w) = d (i + j, w)
  pretty w d  = let (_, l, as) = d (0, w) (0, ⟨⟩, (False : as), w)
                in l
```

For the function *group* lazily accessing *head hs* and *tail hs* is essential. When encountering a new *group* we may still be scanning for one of its remote ancestors, and thus the constructor (:) of *hs* cannot be matched upon, because this part of the list has not been produced yet. One might find this code quite elaborate. It was originally written using an attribute grammar, in which all the different aspects are described separately. The attribute grammar-based definition can be found in a technical report by Swierstra (2004).

We implemented two co-operating sequential processes that are coupled through lazy evaluation. The *P* list that is produced in the functions *leave* and *prune* is passed as an argument to the function *pretty* and is being consumed in the actual construction of the *Layout* and thus serves as a synchronizing buffer. The communication from the printing process to the computation of the *P* list is a bit more subtle: when storing the maximal endpoints $p + r$ in the dequeue, the value of r will in general not be known yet; only when we have concluded whether the parent groups fit and have produced the output up to the beginning of the group, this value gets known. Lazy evaluation enables us to refer to this yet unknown value.

[3] For the sake of clarity we have encoded all list concatenations explicitly. In the actual implementation these have to be replaced by more efficient versions.

3.3 Bringing the printing functions to the arguments

The question arises whether we can make the exchange of information between the scanning and the printing processes more explicitly, thus changing from a parallel processes view to a co-routine view. The answer is affirmative: instead of computing the *P* list and bringing it to the printing part, the scanning co-routine can build for each group a function that constructs the actual layout (*prints* in our terminology), based on its *Horizontal* parameter. So instead of storing *Horizontal* values in the dequeue we store printing functions, to be called once we know their "horizontality." Thus evaluation of the printing and scanning co-routines is interleaved.

We introduce the following types:

```
type Out      = Remaining → Layout
type OutGroup = Horizontal → Out → Out
```

The type *Out* is the type of a function that prints a suffix of a document, that is, from a given point to its end; the function takes as argument the remaining free space on a line and produces a *Layout*. The type *OutGroup* is used to represent the postponed construction of the layout corresponding to the traversed part of a group. It takes three arguments, one indicating how the group is to be formatted, a continuation for the rest of the document, and the remaining free space at the beginning of this group. The latter value remains synchronized with the actual output produced, and the updated value is passed on to the continuation.

Instead of storing *Horizontal* values in the dequeue, we now store values of the type *OutGroup*, which represent postponed printing:

```
type Dq      = DeQueue (Position, OutGroup)
type TreeCont = Position → Dq → Out
type Cont    = (Indent, Width) → TreeCont → TreeCont
```

The algorithm mainly initializes and then combines delimited continuations:

```
instance Doc Cont where
  text t      iw    = scan l outText
                      where
                      l              = length t
                      outText _ c r  = t ++ c (r − l)
  line        (i, w) = scan 1 outLine
                      where
                      outLine True  c r = ' ' :                    c (r − 1)
                      outLine False c r = '\n' : replicate i ' ' ++ c (w − i)
  (dl <> dr) iw     = dl iw ∘ dr iw
  group d     iw    = λc p dq → d iw (leave c) p (dq ▷ (p, λh c → c))
  nest j d    (i, w) = d (i + j, w)
  pretty w d        = d (0, w) (λp dq r → "") 0 ⟨⟩w
  nil         iw    = λc → c
```

When scanning *text* and *line* documents we distinguish between the following cases:

- We already know that the group they belong to does not fit (represented by the dequeue being empty). In this case we can immediately print the element.
- We are still waiting for this information to become available. In this case we remember the printing obligation in the last element of the dequeue.

$$
\begin{aligned}
&scan :: Width \to OutGroup \to TreeCont \to TreeCont\\
&scan\ l\ out\ c\ p\ \langle\rangle = out\ False\ (c\ (p+l)\ \langle\rangle)\\
&scan\ l\ out\ c\ p\ (dq \rhd (s, grp)) = prune\ c\ (p+l)\ (dq \rhd (s, \lambda h \to grp\ h \circ out\ h))
\end{aligned}
$$

Every time scanning increases the current position and the dequeue may be nonempty, *prune* checks whether a pending dequeue element can be printed:

$$
\begin{aligned}
&prune :: TreeCont \to TreeCont\\
&prune\ c\ p\ \langle\rangle \qquad\qquad\qquad r \qquad\quad = c\ p\ \langle\rangle r\\
&prune\ c\ p\ dq@((s, grp) \lhd dq')\ r \mid p > s + r = grp\ False\ (prune\ c\ p\ dq')\ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid True \qquad = c\ p\ dq\ r
\end{aligned}
$$

Finally, at the end of a group the last dequeue element—if it has not already been pruned away—is printed or the print obligation is merged with the element for the surrounding group.

$$
\begin{aligned}
&leave :: TreeCont \to TreeCont\\
&leave\ c\ p\ \ \langle\rangle \qquad\qquad\qquad\qquad = \qquad\qquad c\ p\ \langle\rangle\\
&leave\ c\ p\ (\langle\rangle \rhd \qquad\qquad (s_1, grp_1)) = grp_1\ True\ (c\ p\ \langle\rangle)\\
&leave\ c\ p\ (pp \rhd (s_2, grp_2) \rhd (s_1, grp_1)) =\\
&\qquad c\ p\ (pp \rhd (s_2, \lambda h\ c \to grp_2\ h\ (\lambda r \to grp_1\ (p \leqslant s_1 + r)\ c\ r)))
\end{aligned}
$$

In contrast to our previous solution, how the scanning co-routine treats the elements of a group depends on whether the dequeue is empty. This distinction is required for our more explicit scheduling of the computation, which does not use lazy evaluation anymore. This solution also works in a strict setting.

4 Conclusions

As mentioned in the introduction many have tried to derive a backtrack-free implementation of Oppen's algorithm. Especially Hughes (1995) and Wadler (2003) employed algebraic techniques, and one may wonder why they did not come up with a solution satisfying all nice properties. We think the answer is that we are dealing here with two mutually recursive processes, which run asynchronously. This is not easy to express in a purely algebraic style.

We used the interface designed by Wadler; his implementation is bounded, but not optimally bounded (Section 9 of Chitil, 2005 demonstrates the difference).

What is the difference between Chitil's (2001, 2005) first pretty-printing solution and the solutions presented here? It is the way in which the scanning process informs the printing process about whether a group is to be printed horizontally or vertically. Like our *Lazy* solution, Chitil's first solution passes for every group a Boolean *Horizontal* from the scanning process to the printing process. However, Chitil's first solution is based on the idea that the printing process has passed the information what the *Remaining* space at the beginning of the group is to the scanning process

and hence the *Horizontal* information should be passed backward along the same way. The *Remaining* value of a group travels as part of the start position of the group through the dequeue to the point where the scanning process uses it to decide whether the group is to be formatted horizontally. Hence, Chitil's first solution uses a second dequeue with the same structure as the dequeue of pending group nodes but with reversed data flow to pass a *Horizontal* value back to its group in the printing process. In the middle of pretty-printing parts of the second dequeue do not yet exist, but the defined elements can still be accessed using the identical and fully defined structure of the first dequeue. The required close relationship between the two dequeues implies that no standard dequeue implementation can be reused, the special dequeue implementation is quite complex, and operations have a constant but high time cost.

Our solutions presented here use a single standard dequeue. The *Lazy* solution passes *Horizontal* information in a simple list to the printing process and the *Cont* solution directly constructs printing functions. The latter corresponds to the co-routine equivalent to our parallel processes view, which makes the scheduling of all the computations explicitly visible. All lazy evaluation is gone. Ideally, we would have liked to derive the second solution from the first; we did not manage to do so. We hope this pearl will inspire others to investigate the transformation from the parallel view into the co-routine view in other (less tricky) contexts.

Acknowledgments

Doaitse Swierstra thanks Markus Lauer for spotting a bug in an earlier version of this pearl, Andres Löh for ample support with lhs2TEX and members of the Software Technology group in Utrecht for many useful comments on the presentation. Olaf Chitil thanks Bernd Braßel and Michael Hanus for discussions about how logical variables could simplify the implementation of the pretty printing.

References

Bird, Richard S. (1984) Using circular programs to eliminate multiple traversals of data. *Acta Inf.* **21**, 239–250.

Chitil, Olaf. (2001) Pretty printing with lazy dequeues. In *ACM Sigplan Haskell Workshop*, Hinze, Ralf (ed). Utrecht University Utrecht, pp. 183–201. UU-CS, no. 23.

Chitil, Olaf. (2005) Pretty printing with lazy dequeues. *Trans. Prog. Lang. Syst.* **27**(1), 163–184.

Chitil, Olaf. (2006) *Pretty Printing with Delimited Continuations.* Technical Report 4-06. Computing Laboratory, University of Kent.

Hughes, John. (1995) The design of a pretty-printing library. In *Advanced Functional Programming*, Jeuring, J. & Meijer, E. (eds), LNCS, vol. 925. Berlin: Springer-Verlag.

Okasaki, Chris. (1998) *Purely Functional Data Structures.* Cambridge, UK: Cambridge University Press.

Oppen, Dereck C. (1980) Pretty-printing. *ACM Trans. Prog. Lang. Syst.* **2**(4), 465–483.

Peyton Jones, Simon L. (1997) *A Pretty Printer Library in Haskell.* Part of the GHC distribution at http://www.haskell.org/ghc.

Swierstra, S. D. (2004) *Linear, Online, Functional Pretty Printing (Corrected and Extended Version)*. Technical Report UU-CS-2004-025a. Institute of Information and Computing Sciences, Utrecht University.

Swierstra, S. D., Azero Alocer, P. R. & Saraiva, J. (1999) Designing and implementing combinator languages. In *Advanced Functional Programming, Third International School, AFP'98*, Swierstra, Doaitse, Henriques, Pedro, & Oliveira, José (eds), LNCS, vol. 1608. Berlin: Springer-Verlag, pp. 150–206.

Wadler, Philip. (2003) A prettier printer. In *The Fun of Programming*, Gibbons, Jeremy & Moor, Oege de (eds). Hampshire: Palgrave Macmillan, pp. 223–244.