Linear, Online, Functional Pretty Printing (corrected and extended version)

S. Doaitse Swierstra

institute of information and computing sciences,
utrecht university

technical report UU-CS-2004-025a

www.cs.uu.nl

Linear, Online, Functional Pretty Printing (corrected and extended version)

S. Doaitse Swierstra

August 24, 2004

Abstract

We present yet another implementation of Oppen's pretty printing solution in Haskell. The algorithm is linear in the size of the input, independent of the line width, uses limited look ahead, and has online behaviour.

1 Background

In 1980 Derek Oppen [5] published a paper in which he describes a pretty printing algorithm that attempts to nicely format a document within a given line width.

The essential ideas are that one specifies:

- *potential line breaks*, each to be formatted as either an actual line break or as a space character, depending on the amount of free space left on the current line;
- *a grouping mechanism* to indicate that a specific set of such potential breaks should all be formatted in the same way; thus either all as line breaks or all as space characters.

This problem has been studied by many, some of whom have attempted to derive an equivalent Haskell algorithm [3, 8, 9]. This work has been a basis for the pretty printing library that is currently being distributed as a Haskell library ([6]).

At the Haskell workshop in 2001 Olaf Chitil finally presented an algorithm that fulfills all the nice properties of Oppen's original algorithm [1, 2]. The solution uses a clever coupling between two data structures, in

which the structure of one data structure is used to update another, not yet existing data structure.

The purpose of this paper is to present a more straightforward solution than Chitil's, in which the two aspects of the formatting problem, i.e. the actual formatting and the computation of the choices to be made are clearly separated. We furthermore show why some of the claims made in Chitil's paper about making a change in the main data representation are unjustified.

Since the underlying problem has already been described several times, we will only give a very brief introduction to the formatting problem itself. Next we will discuss properties one might require from a solution, and finally we present, in a number of steps, our solution that has all these desirable properties.

2 Problem specification

Oppen's algorithm aims at formatting a document of the following type:

```
data Doc = NoDoc
         | Text String
         | Line
         | Doc ':<>' Doc
         | Group Doc
         | Nest Int Doc
```

The rôle of the alternatives is as follows:

- *NoDoc* is the unit element for :<>
- *Text String* holds a piece of text that is to be included in the document
- :<> concatenates two documents
- *Line* is to be either formatted as a space or as a line break
- *Group* indicates that all *Line*'s directly inside this group have to be formatted in the same way, i.e. either all as spaces or all as line breaks
- *Nest Int Doc* indicates that all line breaks inside the *Doc* are to be indented by *Int* spaces more than the current indentation

The specification requires that the length of all the lines in the formatted document should not exceed a given width w. If all the *Line* tokens

that directly or indirectly contribute to a group can be formatted as spaces without making the group extend beyond the end of the line, we say that a group *fits*. A direct consequence of this specification is that if a group *fits*, then all its contained groups fit too. Finally we are looking for an algorithm that takes a decision based on a limited look-ahead. A short moment of thought will show that in this way we may not get an optimal solution, in the sense that the the total number of lines in the formatted document in minimized.

Instead of first building a value of type *Doc* and then destructing it through pattern matching we will directly provide the functions constituing the formatting algebra. The domain of this algebra will be of type *Formatter*, a type that we will develop in a number of steps:

```
noDoc ::                                  Formatter
text  :: String      →                    Formatter
line  ::                                  Formatter
(<>)  :: Formatter → Formatter →          Formatter
group :: Formatter                →       Formatter
nest  :: Int         → Formatter →        Formatter
```

For the sake of completeness we will repeat here an example of the use of these functions, directly taken from from Wadler [9].

We want to compute a nice layout for values of the following structure:

```
data Tree = Tree String [Tree]
```

An example of such a structure being:

```
ex = Tree "aaa" [Tree "bbbb" [Tree "ccc" []
                             , Tree "dd" []
                             ]
                , Tree "eee"  []
                , Tree "ffff" [Tree "gg" []
                             , Tree "hhh" []
                             , Tree "ii" []
                             ]
                ]
```

To this end we define the function *showtree*:

$$
\begin{array}{lll}
showTree\ \ (Tree\ s\ ts) & = & group \\
 & & (\ \ text\ s <> nest\ (length\ s) \\
 & & \ \ (showBracket\ ts) \\
 & &) \\
showBracket\ [\,] & = & noDoc \\
showBracket\ ts & = & text\ \texttt{"["} <> nest\ 1\ (showTrees\ ts) \\
 & <> & text\ \texttt{"]"} \\
showTrees\ \ t & = & foldr\ (<>)\ noDoc \\
 & & (intersperse\ (text\ \texttt{","} <> line)\ t)
\end{array}
$$

Our solution will format the example tree within a maximum line lengths of 15 and 20:

```
aaa[bbbb[ccc,                    aaa[bbbb[ccc, dd],
         dd],                        eee,
    eee,                             ffff[gg,
    ffff[gg,                              hhh,
         hhh,                            ii]]
         ii]]
```

In order not to clutter our code too much, we assume that the line width in which the formatting has to take place is fixed by a global constant w.

We now introduce the first version of our type *Formatter*, followed by a description of the rôle of its various components:

$$
\begin{array}{lll}
\textbf{type}\ Fit & = & Bool \\
\textbf{type}\ Space & = & Int \\
\textbf{type}\ Hp & = & Int \\
\textbf{type}\ Indent & = & Int \\
\textbf{type}\ Formatter & = & Fit \to Space \to Hp \to Indent \to (Space, Hp, Result)
\end{array}
$$

The function takes a *Fit* value that indicates whether the current group fits, and thus determines how line breaks are to be formatted.

Furthermore we thread two values through the computation:

- a value of type *Space* that keeps track of the remaining free space on the current line
- a value of type *Hp* (horizontal position) that keeps track of the position of an element in an hypothetical formatting in which all potential line breaks are formatted as spaces

The latter value enables us to compute the *size* of a document by taking *the difference between the entry and exit value* of this threaded value. We might have computed this value in a more direct way, but later the approach taken here will come in handy.

Besides the two threaded values, a *Formatter* returns a *Result* for which type we take the usual *String* → *String* type, that enables us to concatenate partial results in linear time. In the program text variables will in general be chosen as to match with the first character of their type.

We can now easily define the formatting functions, which we also take as the logical specification of the formatting problem:

```
noDoc    _     s h _ = (s              , h              , id     )
text t   _     s h _ = (s − length t, h + length t, (t++))
line     f     s h i = (s'             , h + 1          , r      )
                       where (r, s') = newLine f s i
d_l <> d_r f   s h i = (s_r            , h_r            , r_l · r_r)
                       where (s_l, h_l, r_l)     = d_l f          s   h   i
                             (s_r, h_r, r_r)     = d_r f          s_l h_l i
group d  f     s h i = v
                       where mep = h + s
                             v@(s_d, h_d, r_d) = d  (h_d ⩽ mep) s   h   i
nest j d  f    s h i = d f s h (i + j)
newLine True  s    i = ((' ':)                        , s  − 1)
newLine False s    i = (((('\n' : replicate i ' ')++), w − i)
format   :: Formatter → String
format d = let (_, _, v) = d False w 0 0
           in toSeq v
```

For a group we will call the value $h + s$ the *maximal end point* for this group (*mep*), and a group apparently fits if a group does not extend beyond it's maximal end point: $h_d \leqslant mep$ (h_d is the horizontal position h returned by the call to d), which is equivalent to its size being less than the available free space: $h_d - h \leqslant s$. Note furthermore that we make essential use of lazy evaluation. The *Fits* value that is passed in the function *group* is expressed in terms of the result of this same call, i.e. in terms of h_d.

3 Requirements

The solution we are after computes the same layout as the functions just given, but also has all of the following properties:

1. a complexity that is *linear* in the size of the formatted object
2. a complexity that is *independent of the width w* in which the string has to be formatted
3. produces results with *at most w characters of look-ahead* , and
4. is *online*, i.e. it takes constant time to produce a next piece of output

Since all of our functions perform a constant amount of work for each kind of node, the first requirement is clearly met. It is also fairly obvious that the second part of the specification has been fulfilled already, so apparently the catch is to find a solution that has the last two properties, without losing any of the first two.

The third requirement unfortunately has not been met, since for each group we compute its full size, before we can decide ($h_d \leqslant mep$) whether the group fits. As a consequence, if the top constructor is a *Group*, the whole document has to be traversed (and thus be available and be stored in the heap) before any output can be produced. Obviously the last requirement is not met either, since the time needed before producing any output for a group depends on the size of the group.

4 A Pruning Solution

Apparently the last two problems are caused by the fact that we have to traverse a group completely before being able to decide whether it fits or not. This decision could be made earlier however, if we take all elements of a group into account separately. We change the above algorithm by removing the computation of the horizontal position, and make all functions return the list of horizontal sizes of the contained elements instead (again in the efficient list representation $[Int] \to [Int]$). The rephrased algorithm now reads:

$$
\begin{array}{lll}
nDoc & \_\ s\ i = (s & , id \quad , id \quad) \\
text\ t & \_\ s\ i = (s - length\ t, & (length\ t:), (t{+\!\!+})) \\
line & f\ s\ i = (s' & , (1:) \quad , r' \quad) \\
 & \textbf{where}\ (r', s') = newLine\ f\ s\ i & \\
d_l <\!\!> d_r & f\ s\ i = (s_r & , p_l \cdot p_r \quad , r_l \cdot r_r) \\
 & \textbf{where}\ (s_l, p_l, r_l) = d_l\ f & s\ \ i \\
 & \qquad\quad (s_r, p_r, r_r) = d_r\ f & s_l\ i \\
group\ d & f\ s\ i = v & \\
 & \textbf{where}\ v@(\_, p_d, \_) = d\ (prune\ s\ (toSeq\ p_d))\ s\ \ i &
\end{array}
$$

$$nest\ j\ d\ \ f\ s\ i = d\ f\ s \quad (i + j)$$

The function *prune* checks the elements of the returned sequence one by one in order to return *False* as soon as we have no free space left for the group under inspection.

```
prune  _ []       = True
prune  s (p : ps) = (p ⩽ s) ∧ prune (s − p) ps
format :: Formatter → String
format d = let (_, _, v) = d False w 0
           in v []
```

Although this modification makes the algorithm fulfill the third requirement, we no longer fulfill the second requirement: the time taken by the function *prune* is in general $O(w)$, and the work is done for each group individually. One can see this easily since the result p_d in the function *group* is both used in pruning, and is being returned as part of the value v to be made part of the list of pruning values of the embracing groups. So the question arises how we can reuse some of work done by a call to *prune*, when investigating whether an enclosed group fits.

5 A stepwise development

We now "derive" a solution that consists of two loosely coupled parts:

1. the computation of the layout, provided we know for each group whether it fits
2. the computation of this useful fitting information

Our final solution will turn out to be an intricate mixture between the specification and the pruning version.

5.1 Computing a layout

We start by building on top of the specification, by assuming that we have a list of values available that tells us, in a pre-order arrangement, for each group whether it fits. This value is threaded through the tree too: when visiting a child d of a *Group* node the head of this list indicates whether this group d fits, and the value returned from d has all values associated with its children removed. We show only those parts of our final solution that

do really change with respect to the earlier shown code. The functions that are not mentioned just return the list f unmodified as part of their result.

$$\textbf{type } \mathit{Fits} = [\mathit{Bool}]$$

$$\begin{aligned}\textbf{type } \mathit{Formatter} = \ & \mathit{Fits} \to \mathit{Space} \to \mathit{Hp} \to \quad \ldots \\ & \to (\mathit{Fits} \ , \ \mathit{Space} \ , \ \mathit{Hp}, \mathit{Result}, \ldots)\end{aligned}$$

$$\begin{aligned}
&\mathit{line} \quad f \quad \ldots = (f \quad , \ldots) \\
&\qquad\qquad\quad \textbf{where } (r', s') = \mathit{newLine}\ (\mathit{head}\ f)\ s\ i \\
&(d_l <> d_r)\ f \quad \ldots = (f_r \quad , \ldots) \\
&\qquad\qquad\quad \textbf{where } (f_l\ , \ldots) = d_l\ f \quad \ldots \\
&\qquad\qquad\qquad\quad\ (f_r\ , \ldots) = d_r\ f_l \quad \ldots \\
&\mathit{group}\ d \quad \sim(f : \mathit{ff}) \ \ldots = (f : \mathit{tail}\ f_d, \ldots) \\
&\qquad\qquad\quad \textbf{where } (f_d\ , \ldots) = d \quad \mathit{ff} \quad \ldots
\end{aligned}$$

The major differences here are that in the case of a *line* we look at the head of the list of fitting information, and that in case of a *group* we pass the tail *ff* of the argument list to the contained group d. This group now finds its corresponding element at the head of the list. In order to make sure that siblings of this group element again find the element of the common ancestor group at the head of the list, we reinsert the value f that was active when entering the group. The rest of the result is formed by the tail f_d of possibly updated list *ff*, returned by the call to group d. In this way each group effectively removes its corresponding element from the list of fitting information, once it has served its rôle. In the next section we concentrate on how to compute this quite useful list of fitting information.

5.2 Maintaining the pruning state

The first problem we address now is what extra information to maintain while investigating whether a specific group fits; once we have discovered –when pruning– that the group under investigation does not fit, we want to continue cheaply with the investigation of the next group, i.e. without again inspecting values that we have seen before

Taking a closer look at the specification we see that we can take the nodes into account in a prefix order, since this is the order in which they will be consumed. Let us now draw a picture (figure 1) of the situation when the earlier pruning has reached a specific element. Each circle in the

picture corresponds to a *Group* node. This current position, which is either at a text element or a line element, has been indicated by the large arrow, pointing between the last two groups at the bottom of the picture. We can distinguish four different kinds of *Group* nodes:

1. the *top* node, that is currently being subjected to pruning
2. the *path* nodes on the path between the root node and the current position, for which we have not made a decision yet. Such nodes will either be moved to the next category (*traversed*) when we leave the group they correspond to, or they will eventually become a root node themselves, once their ancestors have been pruned away.
3. the *traversed* nodes to the left of this path, which have been traversed completely, and of which we can, just as in the original specification, cheaply decide whether they fit or not
4. nodes to the right of the path, of which we do not even know that they exist

The nodes on the path from the active node to the current point in this tree structure each correspond to a pending question, i.e. we still have to decide whether these nodes fit. Now, instead of computing a list of element sizes and passing that list to the function *prune*, we carry a representation of the pending questions through the tree, at the same time performing the pruning in an incremental way: we perform a little bit of the pruning work for the top node of this tree whenever we encounter a new element that takes up horizontal space. The rest of the tree representation contains extra information that we are deducing in the mean time. In the pruning version of our algorithm the values to be pruned were collected and moved to the call site of the function *prune*; now we move the new function *prune* to the values it uses for pruning!

Looking at the picture we see that each group we have entered, but not yet left, corresponds to a node on the path. For each such node we maintain a value of type Q, and the whole *Path* is a list of such Q's.

```
type TraversedChildren = Fits → Fits   -- efficient list implementation
type Mep               = Int
data Q                 = Q Mep TraversedChildren
type Path              = [Q]
```

The *Mep* stores the maximal end point for a group, and the *TraversedChildren* field contains a *Fits* value for each fully traversed child.

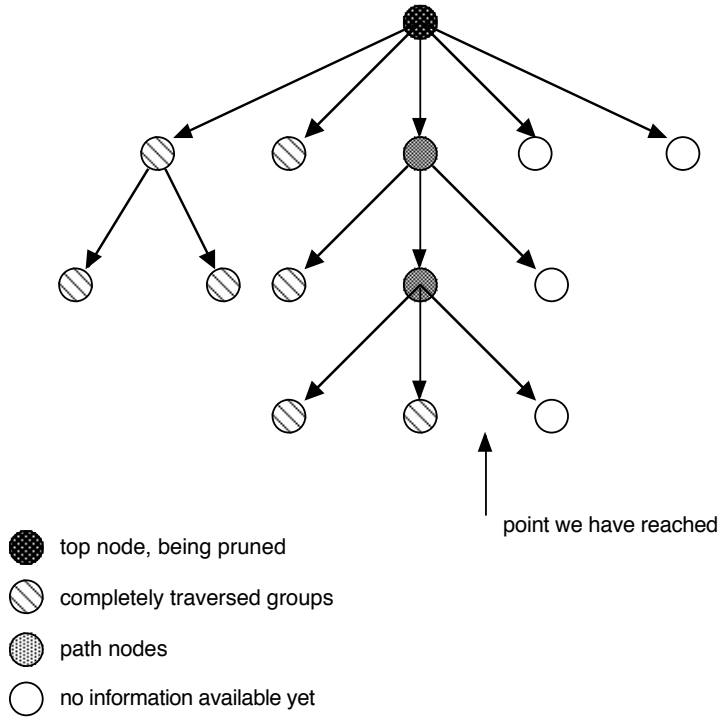

Figure 1: The situation when a pruning fails

In the next step we extend our algorithm so that it maintains this *Path* information in an extra threaded variable p. At the same time we introduce a threaded variable a (for *answers*) of type $Fits \to Fits$, again a functional list implementation. This variable however is threaded in the opposite direction through the tree, and carries all the fitting information we are discovering to the beginning of the computation tree. This is the value we finally use as the initial argument for the root of the whole document at the *Fits* position. In order to shorten the code we have decided to tuple the variables p and a and will indicate the that tuple in our code by pa.

We define three functions that update the *Path* structure, two of which may return newly found fitting information as a side effect:

$$\begin{array}{lll} enterGroup :: & Mep \to Path & \to Path \\ leaveGroup :: & Hp \to (Path, Fits) & \to (Path, Fits) \\ prune :: & Hp \to (Path, Fits) & \to (Path, Fits) \end{array}$$

Each of these functions possibly updates the tree structure of type *Path* and may return newly acquired fitting information by updating the component *Fits*.

When descending into a new group we update the tree structure by appending the question associated with this new group to the end of the path, with *id* indicating that we know nothing about its children yet:

$$enterGroup\ mep\ p = p +\!\!+ [\,Q\ mep\ id\,]$$

When we leave a group we distinguish three cases for the path p:

length $p \equiv 0$ the length of the path equals 0, i.e. we have already discovered that none of the groups we are investigating fits, and so we learn nothing new

length $p \equiv 1$ we are leaving the top node and this node was not pruned away yet, so we can now test (just as in our initial specification) whether the group fits. We also incorporate the elements c to be consumed by the children into the list of answers.

length $p \geqslant 1$ we leave an inner group, i.e. we can test whether the group at the end of the path fits. 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.

Based on these cases we define the function *leaveGroup*, that updates the path p and possibly returns newly found fitting information by updating. We will use c to refer to the information associated with the traversed children. In order to describe easy access to the final two elements of the path sequence we have taken some notational liberty in the pattern matching:

$$\begin{array}{l} leaveGroup\ hp\ pa@(p, a) = \\ \quad \textbf{case}\ p\ \textbf{of} \\ \quad [\,] \qquad\qquad\qquad \rightarrow pa \\ \quad [\,Q\ mep\ c\,] \qquad\quad \rightarrow ([\,], (mep \leqslant hp) : c\ a) \\ \quad pp +\!\!+ [\,Q\ mep_1\ c_1\,] +\!\!+ [\,Q\ mep_2\ c_2\,] \\ \qquad\qquad\qquad\quad \rightarrow (pp +\!\!+ [\,Q\ mep_1\ (c_1 \cdot ((hp \leqslant mep_2):) \cdot c_2)\,] \\ \qquad\qquad\qquad\qquad , a \\ \qquad\qquad\qquad\qquad) \end{array}$$

The third function, *prune*, is called when we visit a *text* or a *line* node, since these are the only points where some output is produced. *Prune* compares

the current position *hp* with the maximal end point of the top node (if present). If this node still fits on the line we do nothing, and return the path p unmodified; if we have reached the point where we can conclude that the top node does not fit we report this by prefixing a *False* to the sequence of answers, append any pending information about fitting children (c), and continue pruning the new top node.

```
prune hp pa@(p, a) =
  case p of
  []                          → pa
  (Q mep c : p') | hp ⩽ mep   → pa
                 | True       → let (p'',          rest) = prune hp (p', a)
                                in  (p'', False : c rest)
```

As we can see *leaveGroup* updates the end of the path and and *prune* the beginning, so the type we have to use actually is a dequeue, which Chris Okasaki showed how to implement with linear amortized cost [4].

6 Assembling the fragments

Merging the two different parts we have developed, i.e. the formatting part and the computation of the fitting information and the maintenance of the path we get for the new *Formatter* type:

```
type Formatter =  Fits     → Space → Hp → (Path, Fits) → Indent
                → (Fits,      Space  ,  Hp  , (Path, Fits) ,  Result)
```

```
noDoc       f        s h pa    _ = (f,  s     , h     ,                pa, id    )
text t      f        s h pa    _ = (f,  s − l, h + l , prune (h + l)  pa, (t++))
  where l = length t
line        f        s h pa    i = (f,  s'    , h + 1, prune (h + 1)  pa, r     )
  where (r, s') = newLine (head f) s i
dl <> dr    f        s h (p, a) i = (f_r, s_r,          h_r, (p_r, a_l)          , r_l · r_r)
  where (f_l, s_l, h_l, (p_l, a_l ), r_l          ) = d_l f   s       h       (p, a_r) i
        (f_r, s_r, h_r, (p_r, a_r), r_r           ) = d_r f_l s_l     h_l     (p_l, a ) i
group d     ∼(f : ff) s h (p, a) i = (f : tail f_d, s_d, h_d, (p'', a_d)          , r_d    )
  where (p'                                       ) = enterGroup (h + s) p
        (f_d, s_d,   h_d, (    p_d,    a_d), r_d  ) = d  ff s       h       (p', a') i
        (                     p'',    a'    )      = leaveGroup h_d     (p_d, a )
```

```
nest j d        f          s h pa     i = (f_d,          s_d, h_d, pa_d              , r_d    )
  where (f_d,       s_d,  h_d, pa_d, r_d        ) = d  f   s        h         pa       (i + j)
```

The actual formatting function, in which we feed back the computed list of answers back into the tree, now becomes:

```
format :: Formatter → String
format d = let (_, _, _, (_, ans), v) = d (False : ans) w 0 ([], []) 0
           in toSeq v
```

7 Variations

7.1 A Process View

In our derivation we can distinguish two different processes:

- one process that traverses the tree, consumes fitting information, and produces "pruning and grouping events"
- one process that consumes these pruning events, and produces the fitting information.

We can make this view more explicit in the code if we remove the actual computation of the pruning information from the tree catamorphism. In order to do so we return a list of pruning steps:

```
data Prune = E Mep   -- enter
           | L Hp    -- leave
           | P Hp    -- prune
```

The function *ans* converts a sequence of such *Prune* steps into a sequence of type *Fits*:

```
type Formatter =      Fits → Space          →  Indent
                    → (Fits  ,  Space, Hp      ,  [Prune], Result)
ans :: Path → [Prune] → Fits
ans p          (E mep : r) =  ans (p ++ [Q mep []]) r
ans []         (L hp  : r) =  ans []                r
ans [Q mep c] (L hp  : r) =  (hp ⩽ mep) : c (ans [] r)
ans (qs ++ [Q mep1 c1] ++ [Q mep2 c2])
               (L hp  : r) =  ans (qs ++ [Q mep1 (c1 · ((hp ⩽ mep2):) · c2)])
                                  r
```

```
ans []          (P _    : r) =  ans []                  r
ans q@(Q mep c : qs)
                (P m    : r) =  if m ⩽ mep
                                then ans q r
                                else False : c (ans qs r))
ans _           []           =  []
```

The functions d and *ans* now function as two individual processes, communicating with each other over the channels *pe* and *fits*:

```
format d = let (_, _, _, pe, v) = d (False : fits) w 0 [] 0   -- process 1
               fits             = ans [] pe                   -- process 2
           in  v
```

If we proceed along this line, we end up in a situation where we have two sequential processes expressed by means of state monads, synchronized with each other through the channels *fits* and *pe*.

7.2 Extending groups

The above algorithms may find solutions that extend beyond the end of the line, because all the *text* elements following a group up to the next *line* in the sequence, effectively form part of the group. The obvious thing to do is thus to check whether at a *Line* not only space is left for the `' '`-character resulting from the line itself, but to prune also for all the *Text* items that follow upto the next *Line* node. So we decide to thread yet another value *future* through the computation, which is the list of sizes of these following *Text* items. It will be clear that at a *Text* item itself we do not have to prune anymore. We sketch the changes to the code:

```
line   ... pa future = (..., foldr prune pa (1 : pend), r', [])
text t ... pa future = (...,                pa, (t++)     , length t : future)
```

8 Discussion

As mentioned in the introduction many have tried to derive a backtrack free implementation of Oppen's algorithm. Especially Hughes [3] and Wadler [8, 9] tried so by employing algebraic techniques, and one may wonder why they did not come up with a solution satisfying all four properties. We think the answer lies in the fact that we are dealing here with two mutual recursive

processes, that run asynchronously. This is not easy to express in purely algebraic style.

Since Chitil [1] already presented a solution to this problem the question arises what are the differences between his solution and the solution presented here. If we look at the way we deal with the data structure of type *Path* we see that our prime interest is what is happening at the root side of the path: will it be pruned away or will this node make it to the end of the group before the end of the line is reached? What we discover about contained nodes is happily accepted and stored in this data structure for later inclusion in the list of answers. There is a subtlety involved here: we may have all the variables available here to construct the expression, but lay evaluation is again essential to make sure that this expression is only evaluated when all values are indeed available! As such it is a straightforward extension of the version using the function *prune*.

Chitil also computes a list of boolean values, but the main emphasis of his solution starts with trying to find out whether the deepest nested group fits, with a pruning process merged with this. In order to store information about pruned away nodes he has to refer to positions in a sequence that does not exist yet. In short: we extend the pruning version of the algorithm, and carry along fitting information, whereas Chitil starts from the original specification, and has to insert pruning results into a second dequeue. Due to the fortunate presence of the dequeue of pending questions, which has the same structure as the queue of results to be produced, he is nevertheless able to address elements of this not yet existing sequence. By keeping the list in the reversed order we can address the head of the list of answers easily, and do not have to revert to the trick used by Chitil.

We furthermore have shown that the overall computation can be nicely expressed as a catamorphism over the input data structure, and that there is no need to revert to a transformation of the tree structure into a sequential one, as claimed by Chitil. Also the change in specification, caused by the inclusion of succeeding text elements into a group, was easily added.

Finally we want to remark that if we convert the formatting functions into continuation passing style, thus making the <> operator implicit, we will get highly imperative code, that, together with the function *ans* we have given, resembles the two cooperating, sequential processes even more. If one compares our code with the original code of Oppen, one sees that it is the implicit scheduling caused by the lazy evaluation that makes the functional formulation so much shorter.

9 Acknowledgements

I want to thank Markus Lauer for spotting a bug in an earlier version of this paper and Andres Löh fur ample support with lhs2TEX.

References

[1] Olaf Chitil. Pretty printing with lazy dequeues. In Ralf Hinze, editor, *ACM Sigplan Haskell Workshop*, number 23 in UU-CS, pages 183–201, September 2001.

[2] Olaf Chitil. Pretty printing with lazy dequeues. *accepted for Transaction on Programming Languages and Systems*, 2004.

[3] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.

[4] Chris Okasaki. Catenable double-ended queues. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 66–74. ACM Press, 1997.

[5] Dereck C. Oppen. Pretty-printing. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.

[6] Simon L. Peyton Jones. Haskell pretty-printing library.

[7] S. D. Swierstra, P. R. Azero Alocer, and J. Saraiava. Designing and implementing combinator languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

[8] Philip Wadler. A prettier printer. *Journal of Functional Programming*, 1999.

[9] Philips Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The fun of programming*, pages 223–244. Palgrave Macmillan, 2003.

A The equivalent attribute grammar definitions

If the reader would argue that the code of our final solution can hardly be called a *Pearl*, let us argue that it is actually an intricate composition of a large number of different aspects. In this appendix we show how we can describe the final algorithm using an attribute grammar system [7], and in such a way that the individual aspects clearly stand out.

We start by defining the underlying context free grammar:

```
DATA Doc
   | NoDoc
   | Text  t : String
   | Line
   | Besides l : Doc r : Doc
   | Group   d : Doc
   | Nest    j : Int  d : Doc
```

The idea is that we describe each of the *aspects* of the algorithm by a separate piece of code. This allows for a nice incremental development. We start by the definition of how the overall result is to be computed in an attribute *res* of type *String*. The notation [| *res* : *String* |] indicates that the non-terminal has both an inherited and a synthesized attribute with the same name *res* that is of type *String*.

```
ATTR Doc [| res : String |]
```

We take a somewhat different approach here than before, by building the result by threading an attribute value backwards through the tree. Had we done so before then we would have introduced yet another argument to all our formatting functions. There are three places where we have to do something; for a *Text* element we prepend the value of the text to the result, in case of a *Line* we refer to a local attribte value *lr* (to be introduced later) that is to be prepended, and in the case of a *Besides* we have to make sure that the value is propagated backwards by making explicit how the value is passed around. If we had omitted these rules, the default copy rules of the UAG system would have made the value being passed forward instead of backwards through the *Besides*. We will see this pattern showing up a few times more.

```
SEM Doc
  | Text     lhs.res = @t ++ @lhs.res
  | Line     lhs.res = @lr
  | Besides  r.res   =        @lhs.res
             l.res   =        @r.res
             lhs.res =        @l.res
```

For the "horizontal position" aspect, all we have to do is to increment this value when encountering a *Line* or a *Text* element. In the case of a *Text* element we compute the length of the text in a local attribute l, so we can refer to it later. We have taken the short notation here, combining the introduction of the new attribute with its associated semantic rules.

```
SEM Doc [| h : Int |]
  | Text lhs.h = @l + @lhs.h
         loc.l = length  @t
  | Line lhs.h = 1   + @lhs.h
```

We have made use of the possibility to introduce a new attribute (h), together with its associated semantics. Next we introduce the attribute s, for keeping track of the free space left on the line; this value is only updated in case of a *Text* or a *Line* element. Here we also place the joint computation of the local attributes lr (line result) and ls (line space) values. The attribute i will be introduced later.

```
SEM Doc [| s : Int |]
  | Text lhs.s      =    @lhs.s − @l
  | Line lhs.s      =    @lhs.s − @ls
         loc.(lr, ls) = newLine (head@lhs.f)@lhs.s  @lhs.i@lhs.res
{
newLine True  s i rest = ((' ' : rest)                          , s − 1 )
newLine False s i rest = (((' \n' : replicate i ' ') ++ rest), w − i)
}
```

The values indicating whether groups fit is threaded through the tree in a forward way, the copy rules taking care of most of the work. Only in case there is something interesting to do, i.e. in case of a *Group* we have to provide some definitions. An interesting observation can be made now: the code given thus far is complete, i.e. if we were generating code from this we get a complete program that we can test provided we are prepared to provide the initial value of the attribute f. In contrast to the case in the paper we do not have to refer to ..., and when adding further aspects we do

not have to touch existing code! We can extend the code by just introducing further aspect by introducing new attributes and their associated semantic functions.

```
SEM Doc [| f : Fits |]
  | Group d.f   = tail (@lhs.f)
          lhs.f = head@lhs.f : tail@d.f
```

The indentation is very easy to take care of:

```
SEM Doc [i : Int||]
  | Nest d.i = @lhs.i + @j
```

Since we are going for the solution with the final adaptation we introduce now a backwards threaded attribute:

```
SEM Doc [| future : {[Int]} |]
  | Text     lhs.future = length@t : @lhs.future
  | Line     lhs.future = []
  | Besides r.future    = @lhs.future
            l.future    = @r.future
            lhs.future  = @l.future
```

The only aspect now missing is the joint computation of the *Path* and the list of answers.

```
ATTR Doc [| p : Path  a : Fits |]
```

In the case of a *text* or a a *Line* we have to prune:

```
SEM Doc
  | Line     loc.(p, a) = foldr prune (@lhs.p, @lhs.a) (1 : @lhs.future)
```

In the case of a *Besides* we have to make sure that the answers are threaded backwards:

```
  | Besides r.a       = @lhs.a
            l.a       = @r.a
            lhs.a     = @l.a
```

In the case of a *Group* we have to add the computations for entering and leaving the group:

```
  | Group   d.p       = enterGroup (@lhs.h + @lhs.s)@lhs.p
            loc.(p, a) = leaveGroup  @d.h                (@d.p, @lhs.a)
            lhs.a     = @d.a
```

Finally we have to initialize the various attributes. For this we introduce a starting symbol *Root*, with only a single production *Root*. Here we see clearly how the synthesized attribute a of child d is used to initialize the inherited attribute f of the same child d.

$$
\begin{array}{l}
DATA\ Root\ [||res : String] \\
\quad |\ Root\ d : Doc \\
SEM\ Root \\
\quad |\ Root\ d.s \quad = w \\
\qquad\quad d.res \quad = [\,] \\
\qquad\quad d.f \quad = False : @d.a \\
\qquad\quad d.h \quad = 0 \\
\qquad\quad d.i \quad = 0 \\
\qquad\quad d.future = [\,] \\
\qquad\quad d.p \quad = [\,] \\
\qquad\quad d.a \quad = [\,]
\end{array}
$$

To be compatible with the original version we add some code to redefine the generated semantic functions:

$$
\begin{array}{ll}
noDoc & = sem\_Doc\_NoDoc \\
text & = sem\_Doc\_Text \\
line & = sem\_Doc\_Line \\
(<>) & = sem\_Doc\_Besides \\
group & = sem\_Doc\_Group \\
nest & = sem\_Doc\_Nest \\
format & = sem\_Root\_Root
\end{array}
$$