Kent Academic Repository

Chitil, Olaf (2006) *Pretty Printing with Delimited Continuations.* Technical report. University of Kent, Kent, UK

Downloaded from
https://kar.kent.ac.uk/14464/ The University of Kent's Academic Repository KAR

The version of record is available from

This document version
UNSPECIFIED

DOI for this version

Licence for this version
UNSPECIFIED

Additional information

Versions of research works

Versions of Record
If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts
If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries
If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Computer Science at Kent

Pretty Printing with Delimited Continuations

Olaf Chitil

Technical Report No. 4-06
June 2006

Abstract Pretty printing is the task of nicely formatting tree structured data within a given line width limit. In 1980 Oppen published a pretty printing algorithm that takes time linear in the size of the input, independent of the line width, and uses only limited look-ahead. This work inspired the development of a number of purely functional pretty printing libraries in Haskell. Here I present a new functional pretty printing algorithm that has all the nice properties of Oppen's and is surprisingly simple. A double-ended queue of delimited continuations is the key to addressing all aspects of the problem explicitly.

Copyright © 2006 University of Kent
Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

A pretty printer converts tree structured data, for example the syntax tree of a program or an XML document, into nicely formatted text with a given line width limit. A pretty printing library provides the functionality common to a large class of pretty printers, thus enabling a programmer to easily implement a specific pretty printer. A pretty printing library enables the programmer to *compositionally* describe alternative layouts for components of the data to be printed. The pretty printing algorithm then chooses an *optimal* layout from this set of layouts. There is a trade-off between the available choice of alternative layouts, the optimality criterion and the efficiency of the pretty printing algorithm.

Oppen [8] published an imperative pretty printer that allows the description of nice layouts, adequate for many purposes, and that is very efficient. The algorithm takes time linear in the size of the input, independent of the line-width limit. Furthermore, the algorithm is *bounded*, that is, it produces parts of the output already after having processed only limited parts of its input. Oppen's work inspired numerous pretty printing libraries, in particular several Haskell libraries [4, 9, 12, 1, 2, 11], because lazy evaluation seems to be a natural basis for implementing boundedness.

Here I present a new purely functional algorithm that is both linear-time and bounded. It is faster than any previous functional implementation; it is relatively simple and explicitly expresses all issues in pretty printing, many of which are hidden in previous implementations based on implicit lazy evaluation. The new algorithm uses delimited continuations to explicitly control the flow of the computation. I will start with a concise formal specification and then develop the efficient implementation in several steps.

2 Specification

Interface. We will implement a library that provides the following interface:

```
type Width = Int
type Layout = String

text   :: String -> Doc
line   :: Doc
(<>)   :: Doc -> Doc -> Doc
nest   :: Int -> Doc -> Doc
group  :: Doc -> Doc
pretty :: Width -> Doc -> Layout
```

The first five functions construct documents whereas the last one produces, for a given line-width, a pretty layout from a document.

The function `nest` increases the indentation for all line breaks within its document argument. Although indentation is necessary for obtaining pretty output, indentation does not change the essential problem of selecting the optimal layout. Hence we omit

the function `nest` in the subsequent sections, but Section 7 will outline how `nest` can easily be added to the final implementation.

Alternative layouts. A document may be formatted *horizontally* or *vertically*. The former means that its layout is a single line, without any line breaks, whereas the latter allows line breaks. A document has many different layouts and we represent it as follows:

```
type Horizontal = Bool
type Doc = Horizontal -> [Layout]
```

For convenience we use a list of `Layout`s, although it is actually a set. We represent a document as a function from `Horizontal` to `Layout`s to describe both the (actually singleton) set of layouts when the document is formatted horizontally and the set of layouts when the document is formatted vertically.[1] We can concisely specify the semantics of all document constructors:

```
(text t) _ = [t]
line True = [" "]
line False = ["\n"]
(d1 <> d2) h = [l1 ++ l2 | l1 <- d1 h, l2 <- d2 h]
(group d) True = d True
(group d) False = d False ++ d True
```

The document `line` denotes a line break, if it is formatted vertically, and a single space, if it is formatted horizontally. The concatenation `(<>)` of two documents yields the cross product of the two sets. The function `group` marks a document as a unit which may be formatted either *horizontally* or *vertically*. The two equations for `group` express that within a horizontally formatted document any nested group has to be formatted horizontally as well, but within a vertically formatted document a group can be formatted horizontally or vertically. This is why a (vertically formatted) document has many different layouts.

Prettiest layout. The function `pretty` selects the "best" layout from the set of layouts described by a document, given a line-width limit. One might consider the "best" layout to be the one with the least number of lines that has all lines within the line-width limit. 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 (cf. [4]).

Hence we take the same optimality criterion that Hughes and Wadler use. We compare two layouts lexically line by line, if both lines are within the line-width limit, then the layout with the longer line is better, if one line is beyond the line-width limit, then the layout with the shorter line is better:[2]

[1] An alternative representation as a tuple is more awkward to use.

[2] The standard function `lines :: String -> [String]` breaks a text into lines based on the occurrences of line break characters (`\n`).

```
pretty w d = minimumBy (compareLayout w) (d False)

compareLayout :: Width -> Layout -> Layout -> Ordering
compareLayout w l1 l2 = compareLines w (lines l1) (lines l2)

compareLines :: Width -> [String] -> [String] -> Ordering
compareLines w []        []        = EQ
compareLines w []        _         = LT
compareLines w _         []        = GT
compareLines w (l1:l1s) (l2:l2s) = compareLine w l1 l2 `lexical`
                                     compareLines w l1s l2s

lexical :: Ordering -> Ordering -> Ordering
lexical EQ o = o
lexical LT _ = LT
lexical GT _ = GT

compareLine :: Width -> String -> String -> Ordering
compareLine w l1 l2 = if len1 <= w && len2 <= w then compare len2 len1
                                                else compare len1 len2
  where
  len1 = length l1
  len2 = length l2
```

In the definition of `pretty` the whole document is applied to `False`, expressing that `lines` appearing outside any group are always formatted as line breaks.

Example. We can define a simple pretty printer for lists

```
toDoc :: [Int] -> Doc
toDoc xs = text "[" <>
             foldr (<>) (text "]")
               (intersperse (group (text "," <> line))
                 (map (text.show) xs))
```

such that `pretty 60 (toDoc [1..50])` yields

```
[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, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50]
```

Independently, each `line` can be formatted horizontally or vertically, because each is in a separate `group`. A `line` is formatted horizontally, if and only if the text up to the next `line` still fits on the current line. Many more examples of using the library are given in [12].

Linear time. The exponential time complexity of the functional specification is irrelevant. We just state separately that we demand our implementation to take time linear in the size of the expression constructing the document, independent of the line-width limit of `pretty`.

Optimal Boundedness. The given functional specification is hyper-strict, as the following computation of the Haskell interpreter Hugs [5] demonstrates:

```
Main> pretty 4 (group (text "Hi" <> line <> text "you" <> undefined))
"
Program error: undefined
```

However, we see that the strings `"Hi"` and `"you"` already do not fit together in a line of width `4`, so the vertical format has to be chosen for the group and hence we desire the following partial output:

```
Main> pretty 4 (group (text "Hi" <> line <> text "you" <> undefined))
"Hi\nyou
Program error: undefined
```

In general, for any partial input we consider all completions of this partial input to total inputs. For each of these total inputs `pretty` selects a layout. The common prefix of all these layouts is the output that we demand our implementation of `pretty` to yield for the original partial input. So we specify `pretty` to be the least-strict extension of the functional specification defined in Haskell here.

Any group with a width larger than the width-limit has to be formatted vertically. Hence the output of the least-strict `pretty` is at most width-limit characters behind the input already processed. We say that `pretty` is *bounded*, because the look-ahead into the input is *bounded* by the width limit. It is even *optimally bounded*, because as the least-strict extension it produces output with the minimal look-ahead possible.[3]

3 Algorithm Outline, Normalisation and Document Representation

There exists a basic pretty printing algorithm that meets our specification for many but not all documents. Transforming a document before that pretty printing algorithm is applied ensures that the output always meets our specification.

The basic pretty printing algorithm works as follows. First determine for each group in the document its width, that is, the space it requires for printing *if* it was printed horizontally, all in one line. Given this information we can produce the optimal layout

[3]To be precise, we do not consider a partial string as argument of `text`. The argument of `text` is considered as an atomic value that is added to the layout in one step. Considering partial strings in our implementation is straightforward, but the high granularity would increase run-time by a substantial constant factor.

by a simple in-order traversal of the document tree. In the traversal we keep track of the free space remaining in the current output line. Every time we come across the start of a group we just compare the remaining space with the width of the group. If the width is smaller or equal, the group is formatted horizontally, otherwise vertically.

Why does this algorithm not always produce the desired output? For

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

this algorithm yields

```
Hi you!
```

whereas our specification says that the output should be

```
Hi
you!
```

A group that still fits on a line may be followed by further text, without a separating `line`. Because there is no `line`, the text has to be added to the current line, even if it does not fit. Formatting the group vertically might have avoided the problem.

We say that a document is *group-closed* if between the end of every `group` and the next `text` document there is always a `line` document. In that case the end of every group can be chosen to be the end of the line. Hence for group-closed documents the algorithm produces the same layout as is selected by our functional specification.

To use the pretty printing algorithm for any document, we first apply a normalisation transformation which transforms any document into an equivalent group-closed document. To enable a simple transformation we represent a document as a token sequence:

```
data Tokens = Text String Tokens
            | Line Tokens
            | Open Tokens
            | Close Tokens
            | Empty
```

A group is represented as an `Open` token, the sequence of the grouped document and a final `Close` token. To construct the token sequence in linear time we actually represent a document as a function on token sequences, and function composition performs concatenation [3].

```
newtype Doc = Doc (Tokens -> Tokens)

text s          = Doc (Text s)
line            = Doc (Line)
Doc l1 <> Doc l2 = Doc (l1 . l2)
group (Doc l)   = Doc (Open . l . Close)
```

To transform a document into a group-closed document, we normalise the token list with respect to the following (confluent) rewriting rules:

```
Close (Text s ts) ⇒  Text s (Close ts)
Open (Text s ts)  ⇒  Text s (Open ts)
Open (Close ts)   ⇒  ts
```

Rewriting only moves `Text` tokens in and out of groups. Therefore the set of `line`s "belonging" to each group, which are either all formatted as new lines or all as spaces, is unchanged. So normalisation leaves the set of layouts denoted by a document unchanged. Only the representation of the document is changed, so that our pretty printing algorithm (possibly) selects a different set element as output. Normalised token lists even have the additional property that there is no `Text` token between an `Open` token and the next `Line` token. Hence making the decision whether a group is formatted horizontally or vertically when we come across an `Open` token is not premature, but indeed just in time.

We can implement normalisation by a linear traversal of the token list, which collects `Open` and `Close` tokens until the next `Line` token is reached:

```
normalise :: Tokens -> Tokens
normalise = collect id
  where
  collect :: (Tokens -> Tokens) -> Tokens -> Tokens
  collect co Empty       = co Empty
  collect co (Open ts)   = collect (co . open) ts
  collect co (Close ts)  = collect (co . Close) ts
  collect co (Line ts)   = (co . Line . collect id) ts
  collect co (Text s ts) = Text s (collect co ts)

  open (Close ts) = ts
  open ts         = Open ts
```

This normalisation reminds of the context passing representation of John Hughes [4]. The argument `co` of the function `collect` is a context of `Open` and `Close` tokens.

The construction of the token sequence and its normalisation happen in linear time and are least-strict. In particular, for a group the `Open` token will always be produced if tokens for some part of its content can be produced:

```
Main> (\(Doc d) -> normalise (d Empty)) (group (text "hi" <> line <> undefined))
Text "hi" (Open (Line
Program error: Prelude.undefined
```

4 A Linear Unbounded Algorithm

We have a group-closed document represented by a token list. Now we have to determine the details of the actual pretty printing algorithm.

The algorithm outline suggests that we first determine the widths of all groups and then in a subsequent traversal of the document produce the actual layout. However, considering our final aim of a bounded algorithm we realise that determining group widths and producing output have to be interleaved. Hence we endeavour straight away to define a pretty printing algorithm that only traverses the token list a single time.

At the outermost level of a document — outside any group — pretty printing is straightforward: When we come across a `Text` token we immediately output its string. When we come across a `Line` token we immediately output a line break. All the time we can also keep track of the space remaining on the current line. However, what do we do when we come across an `Open` token? When going along each subsequent token we can compute the width of the group, but we cannot yet output the tokens. However, we can construct a function for outputting these tokens! We can construct a function that given the information whether the group content should be formatted horizontally or vertically and the remaining space at the beginning of the group will output the formatted contents of the group. So we define the types

```
type Remaining = Int
type Out = Remaining -> Layout
type OutGroup = Horizontal -> Out -> Out
```

A group output function takes an argument of type `Horizontal` and an argument of type `Out`. The later is the continuation of the function. This continuation outputs the parts of the document that come after the group.

Groups can be nested. First of all, this complicates the computation of the width of a group. When processing a token we do not want to update a width value for each surrounding group; their number is unbounded. Hence we introduce an absolute measure of a token's position. The *(absolute) position* gives the column in which a token would start, if the *whole* document that is passed to `pretty` was formatted in a single line. In the traversal of the token list we only need to keep track of the position of the current token. The width of a group is the difference between the position of its `Close` token and the position of its `Open` token.

```
type Position = Int
```

The second consequence of the nesting of groups is that our algorithm cannot just defer outputting a group by passing along the token list a single group output function. Why? Because each group has its own `Horizontal` value and when we extend the group output function we can only refer to its `Horizontal` argument, which would be for the outermost deferred group. We need access to the `Horizontal` value of the innermost deferred group. We can best express this scoping by having a separate group output function for each surrounding group. Only the output function for the innermost group is extended while traversing the document of the innermost group whereas the other functions are passed on unchanged. So each function does not represent the deferred output of a group up to the currently processed token, but only the deferred output up to another deferred nested group. When we come across a `Close` token we merge the

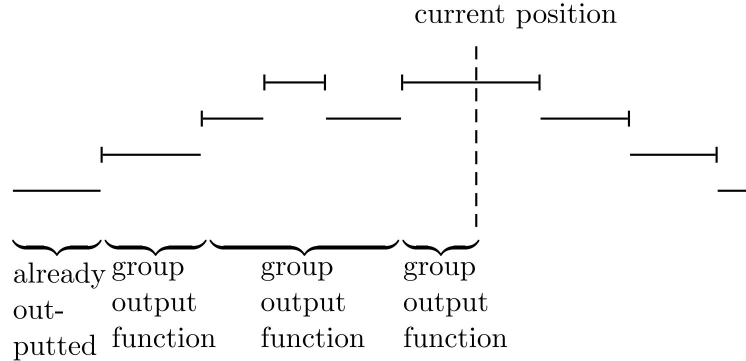

Figure 1: Groups and group output functions in token list traversal.

function for the innermost group with the function for the next inner group. If there is no other group we can apply the function to produce the output for the group.

Figure 1 illustrates for a point in the computation which part of the token list each group output function is responsible for. A horizontal line symbolises a group, with inner nested groups further up. The dashed line indicates the current point of the token list traversal. There are three surrounding groups, hence three group output functions.

In addition to its output function we need for each surrounding group also the position of its `Open` token, so that when coming to its `Close` token we can determine the width of the group. Altogether we keep all information about surrounding groups in a sequence of tuples: $\langle$`(Position,OutGroup)`$\rangle$.

Here $\langle$`a`$\rangle$ denotes some abstract sequence type with elements of any type `a`. Additionally $\langle\rangle$ denotes both the empty sequence and the pattern that matches the empty sequence, $\langle e \rangle$ denotes the singleton sequence with element e and the pattern that matches the singleton sequence, and $e \lhd es$ denotes the expression that puts e in front of the sequence es and the pattern that matches a non-empty sequence. Using an abstract sequence type might seem overkill — we could just use a list instead — but when we refine the algorithm in the next section the sequence will have to become a double ended queue. Okasaki [7] demonstrated that premature commitment to a representation can make functional programmers blind to a natural implementation solution.

Figure 2 gives the whole implementation. The token interpreter `inter` implements the algorithm just described. The functions `outText` and `outLine` serve as building blocks from which the group output functions are constructed. When `inter` comes across a `Close` token, the expression `p <= s1+r` is used to decide whether the innermost group is formatted horizontally or vertically — when the group output function is finally applied. If there is no other surrounding group, then the output function is applied and output is produced immediately, otherwise the output function is merged with the function for the next inner group. The interleaving of outputting a group and traversing the token list with `inter` is clearly expressed by the continuations.

```
pretty :: Width -> Doc -> Layout
pretty w (Doc d) = inter (normalise (d Empty)) w 1 ⟨⟩ w

inter :: Tokens -> Width -> Position -> ⟨(Position,OutGroup)⟩ -> Out
inter (Empty) _ _ _ = const ""
inter (Text t ts) w p ⟨⟩ =
  outText t (inter ts w (p + length t) ⟨⟩)
inter (Text t ts) w p ((s,outGrp) ◁ qs) =
  inter ts w (p + length t) ((s,\h c -> outGrp h (outText t c)) ◁ qs)
inter (Line ts) w p ⟨⟩ =
  outLine w False (inter ts w (p+1) ⟨⟩)
inter (Line ts) w p ((s,outGrp) ◁ qs) =
  inter ts w (p+1) ((s,\h c -> outGrp h (outLine w h c)) ◁ qs)
inter (Open ts) w p qs =
  inter ts w p ((p,\h c -> c) ◁ qs)
inter (Close ts) w p ⟨(s1,outGrp1)⟩ =
  \r -> outGrp1 (p<=s1+r) (inter ts w p ⟨⟩) r
inter (Close ts) w p ((s1,outGrp1) ◁ (s2,outGrp2) ◁ qs) =
  inter ts w p ((s2,\h c -> outGrp2 h (\r1 -> outGrp1 (p<=s1+r1) c r1)) ◁ qs)

outText :: String -> Out -> Out
outText t c r = t ++ c (r - length t)

outLine :: Width -> Horizontal -> Out -> Out
outLine w h c r = if h then ' ' : c (r-1) else '\n' : c w
```

Figure 2: The linear unbounded pretty printing algorithm

5 A Linear Bounded Algorithm

The algorithm in Figure 2 is still unbounded, because only when an outermost group has been completely traversed, the group and all its inner groups are outputted. Nonetheless, turning this algorithm into a bounded one requires few modifications.

While traversing the token list we know the start position of the outermost surrounding group, the remaining space at the start of the outermost surrounding group and the position of the current token. Thus we see when the width of the outermost group is definitely larger than the remaining space. At that point in the traversal of the token list we can already output the outermost group.

Figure 3 gives the implementation of the linear bounded algorithm. The new function `prune` checks whether the outermost group still fits. If it does not fit, then it applies its output function and continues checking whether the next outermost group fits. The function `prune` uses the pattern `es ▷ e` to access the last element of the sequence. So the sequence is no longer a simple stack but a double-ended queue. Okasaki [6] describes a functional implementation of double-ended queues with amortised constant

```
inter :: Tokens -> Width -> Position -> ⟨(Position,OutGroup)⟩ -> Out
inter (Empty) _ _ _ = const ""
inter (Text t ts) w p ⟨⟩ =
  outText t (inter ts w (p + length t) ⟨⟩)
inter (Text t ts) w p ((s,outGrp) ◁ qs) =
  prune ts w (p + length t) ((s,\h c -> outGrp h (outText t c)) ◁ qs)
inter (Line ts) w p ⟨⟩ =
  outLine False (inter ts w (p+1) ⟨⟩)
inter (Line ts) w p ((s,outGrp) ◁ qs) =
  prune ts w (p+1) ((s,\h c -> outGrp h (outLine h c)) ◁ qs)
inter (Open ts) w p qs =
  inter ts w p ((p,\h c -> c) ◁ qs)
inter (Close ts) w p ⟨⟩ = inter ts w p ⟨⟩
inter (Close ts) w p ⟨(s1,outGrp1)⟩ =
  \r -> outGrp1 (p<=s1+r) (inter ts w p ⟨⟩) r
inter (Close ts) w p ((s1,outGrp1) ◁ (s2,outGrp2) ◁ qs) =
  inter ts w p ((s2,\h c -> outGrp2 h (\r1 -> outGrp1 (p<=s1+r1) c r1)) ◁ qs)

prune :: Tokens -> Width -> Position -> ⟨(Position,OutGroup)⟩ -> Out
prune ts w p ⟨⟩ = inter ts w p ⟨⟩
prune ts w p qs@(qs' ▷ (s,outGrp)) =
  \r -> if p>s+r then outGrp False (prune ts w p qs') r else inter ts w p qs r
```

Figure 3: The linear bounded pretty printing algorithm

time complexity for every operation.

The few places in which the definition of `inter` had to be adapted are underlined. Only processing the tokens `Text` and `Line` increases the current position and hence only there calls of `prune` have to be inserted. Additionally, `inter` may now come across a `Close` token even when the sequence of start positions and group output functions is empty, because groups may have been pruned before their `Close` token was reached.

6 Optimisation by Specialisation

Nearly every call of `inter` and every call of `prune` pattern matches on the queue. This distinction between empty and non-empty queue and projection of components costs substantial time. Processing the majority of tokens, `Text` and `Line`, updates the front element of the queue but leaves the rest unchanged. Only the `Open` and `Close` token and sometimes `prune` modify the queue. Hence we specialise `inter` into three and `prune` into two mutually recursive function definitions:

```
noGroup    :: Tokens -> Width -> Position -> Out
oneGroup   :: Tokens -> Width -> Position -> Position -> OutGroup -> Out
multiGroup :: Tokens -> Width -> Position -> Position -> OutGroup ->
```

```
              ⟨(Position,OutGroup)⟩ -> Position -> OutGroup -> Out
pruneOne   :: Tokens -> Width -> Position -> Position -> OutGroup -> Out
pruneMulti :: Tokens -> Width -> Position -> Position -> OutGroup ->
              ⟨(Position,OutGroup)⟩ -> Position -> OutGroup -> Out
```

The function `noGroup` is used when there is no deferred group, the function `oneGroup` when there is one deferred group and the function `multiGroup` when there are at least two deferred groups. The function `multiGroup` takes a queue as argument, but it takes the information for the innermost and the outermost deferred group separately, so that accessing or updating that information requires no queue operations.

Specialisation makes the algorithm approximately 35% faster. However, the specialised algorithm is substantially longer and contains duplicated code.

The output function itself takes a continuation to pass control. This continuation is not necessary. We can also define our algorithm with a type

```
type OutGroup = Horizontal -> Remaining ->
                (Remaining,String -> String)
```

This type makes even more explicit that a group output function produces a partial output (of type `String -> String` to enable constant time concatenation [3]) and an updated remaining space. Using this type is just inconvenient: merging an output group with the output of a single `Text` or `Line` token or with another output group is more awkward than with continuations, because of the required pattern matching on tuples.

7 Indentation

To complete the pretty printing library we have to add the function `nest`. There are different interpretations of the expression `nest` n. In Wadler's library it increases the current left margin by n columns whereas in Oppen's pretty printer (and other libraries) it sets the left margin to the current column position plus n. We can implement both variants by adding two new tokens:

```
data Tokens =  ...
            | OpenNest (Margin -> Remaining -> Width -> Margin) Tokens
            | CloseNest Tokens

type Margin = Int
```

We extend the implementation to interpret the new tokens such that the function of the first token takes the current margin to determine a new margin and the second token resets the margin to its previous value. Just as we keep track of the space remaining on the current line we keep track of old left margins; we extend the type `Out`:

```
type Out = Remaining -> [Margin] -> Layout
```

8 Related Work

The basic idea of the algorithm presented here dates back to Oppen [8]. In particular he also represents a document as a token list. In his Section 2 Oppen assumes that any document is group-closed; he hints at the end of his Section 5 that other input can be transformed. Oppen updates a mutable array in a complex pattern to keep track of the information required for switching between processing input and producing output.

Oppen's work inspired numerous pretty printing libraries for Haskell. The libraries of John Hughes [4], Simon Peyton Jones [9] and Phil Wadler [12] all use some forms of backtracking and hence are less efficient than Oppen's algorithm. The delimited continuation pretty printer implements the interface designed by Phil Wadler and the specification is based on the less formal specifications of John Hughes and Phil Wadler. The semantics of the delimited continuation pretty printer agrees with that of Phil Wadler, except that Wadler's is bounded but not optimally bounded (Section 9 of [2] demonstrates the difference).

In [1, 2] I presented the first purely functional algorithm that has all the nice efficiency properties of Oppen's algorithm. This algorithm uses an intricate lazy coupling of two double-ended queues. Thus it demonstrates the power of laziness but is rather complex and requires a specially modified implementation of double-ended queues.

Subsequently Doaitse Swierstra [11] showed that the two special double-ended queues can be replaced by one double-ended queue and a lazy list, thus giving a simpler solution that also reuses a standard queue implementation. The delimited continuation implementation is even simpler and does not rely on laziness. Unfortunately Swierstra's method for handling documents that are not group-closed is faulty.

Swierstra's algorithm does not use any intermediate data structures. We can easily remove the token list from the delimited continuation algorithm as well. The token list interpreter `inter` is clearly in the image of defunctionalisation [10]. Applying the inverse of defunctionalisation we replace the token list type by a functional type and obtain a function for every token:

```
type PrettyTokens = Width -> Position -> ⟨(Position,OutGroup)⟩ -> Out

pEmpty :: PrettyTokens
pText  :: String -> PrettyTokens -> PrettyTokens
pLine  :: PrettyTokens -> PrettyTokens
pOpen  :: PrettyTokens -> PrettyTokens
pClose :: PrettyTokens -> PrettyTokens
```

Similarly we can replace the use of the token list in normalisation by another function type and a set of functions. We just have to give up the minor optimisation `Open (Close ts)` ⇒ `ts`, because it requires matching a `Close` token more than once. The resulting algorithm has no intermediate data structures but replaces the token list type by two different functional types. I find this functional variant hard to read. There is no noticeable difference in runtime between the variants, because at a low level closures and algebraic data types are implemented rather similarly.

Surprisingly the algorithms of [4, 9, 12, 1, 2] are all first-order. Are higher-order functions essential for our algorithm? Naturally not, because defunctionalisation [10] can replace a continuation by a data structure and a first-order interpreter. Here this means replacing the types `Out` and `OutGroup` by two complex mutually recursive algebraic data types. Again we obtain an implementation that is harder to read.

9 Conclusions

I presented a new linear bounded pretty printing algorithm that is shorter and simpler than previous ones. Delimited continuations express explicitly the necessary switching between producing output and processing the input. For a partially traversed group, for which we do not yet have enough information to decide whether it is formatted horizontally or vertically, we produce a group output function that when we have the information will output the group contents. A group output function is a delimited continuation. The function that traverses the token list passes control to an output function to continue the pretty printing. However, the output function does not compute the whole remaining output, but passes control back to another function to continue. For every deferred surrounding group we keep a group output function and the start position of the group in a double-ended queue. It has to be a double ended queue, because processed input relates to the innermost surrounding group but the outermost group has to be checked continuously for whether it fits. Finally, specialisation reduces the number of costly operations on the double-ended queue and thus improves performance.

Correctness and linear runtime of the algorithm do not rely on lazy evaluation. However, lazy evaluation ensures that input and output are only evaluated as demanded and thus the algorithm only requires a small bounded amount of space.

Hughes [4] and Wadler [12] used algebraic techniques to derive their pretty printing implementations that are based on backtracking. In contrast, no formal derivation or correctness proof exists for any of the linear pretty printing implementations by Swierstra [11] or myself [2]. I agree with Section 8 of Swierstra [11] that the linear implementations employ programming techniques that are “not easy to express in purely algebraic style”. So there is a challenge: This report gives a concise formal specification and a highly efficient implementation, both of which are fairly short. Can anybody give a readable proof of their equivalence?

Acknowledgements

Thanks to Bernd Braßel and Michael Hanus for discussions about how logical variables could simplify the implementation of pretty printing. The need for deferring output until logical variables are bound gave me the idea of using continuations.

References

[1] Olaf Chitil. Pretty printing with lazy dequeues. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 183–201. Universiteit Utrecht, 2001. UU-CS-2001-23.

[2] Olaf Chitil. Pretty printing with lazy dequeues. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):163–184, January 2005.

[3] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141—144, 1986.

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

[5] The Haskell User's Gofer System. `http://www.haskell.org/hugs/`.

[6] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[7] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming*, pages 131–136, 2000.

[8] Dereck C Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[9] Simon L Peyton Jones. A pretty printer library in Haskell. Part of the GHC distribution at http://www.haskell.org/ghc, 1997.

[10] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, 1972.

[11] S. Doaitse Swierstra. Linear, online, functional pretty printing (corrected and extended version). Technical Report UU-CS-2004-025a, Utrecht University, 2004.

[12] Philip Wadler. A prettier printer. In *The Fun of Programming*, chapter 11, pages 223–244. Palgrave Macmillan, 2003.