Polynomial-Time Optimal Pretty-Printing Combinators with Choice

Anton Podkopaev[1(✉)] and Dmitri Boulytchev[2]

[1] Intellij Labs Co. Ltd, Universitetskaya emb., 7-9-11/5A,
199034 St.Petersburg, Russia
Anton.Podkopaev@jetbrains.com
[2] St.Petersburg State University, Universitetski pr., 28,
198504 St.Petersburg, Russia
dboulytchev@math.spbu.ru

Abstract. We describe pretty-printing combinators with choice which provide optimal document layout in polynomial time. Bottom-up tree rewriting and dynamic programming (BURS) is used to calculate a set of possible layouts for a given output width. We also present the results of suggested approach evaluation and discuss its application for the implementation of pretty-printers.

Keywords: Pretty-printing · Combinators · Bottom-up rewriting systems

1 Introduction

Pretty-printing is a transformation which provides a human-readable text from some internal program representation (for example, AST). The need for pretty-printing arises in a wide variety of applications, for example IDEs [7], reengineering tools [9] etc. In its utter form pretty-printer has to implement a reverse transformation to parsing, i.e. to generate text which can further be edited and parsed back.

The general requirement for pretty-printer to provide human-readable text decomposes into many additional requirements most of which are hard to formalize. For example, the resulting text must be *observable* (not *too wide* and not *too long*), it should reflect the structure of a program, it has to respect the coding style conventions for a given project, etc. As a result, pretty-printers, which try to fulfill these requirements, become hard to implement, maintain and reason about.

In the realm of functional programming the natural approach to pretty-printing is *pretty-printing combinators*: source representation of a program is transformed into a structured *document* using relatively small set of constructors. Then, this document is interpreted by a pretty-printing algorithm providing string representation.

A. Voronkov and I. Virbitskaite (Eds.): PSI 2014, LNCS 8974, pp. 257–265, 2015.
DOI: 10.1007/978-3-662-46823-4_21

The mainstream set of pretty-printer combinators originates from the works of Wadler [15] and Hughes [6], who in turn referred to the approach developed by Oppen [11]. The basic document constructors in this approach[1] are:

- atomic string which is printed as is;
- *separator*;
- sequential composition of two documents;
- scoping.

The definitive characteristic of this approach is the interpretation of separators: all separators within a given scope can *coherently* be turned into either spaces or newline symbols by a pretty-printing algorithm. The choice for separators is determined by the requirement to respect given line width using as few lines as possible (hence optimality property).

The original Oppen's algorithm is essentially imperative; it works in a time linear on an input program length and provides optimal result. Pretty-printer combinators of Wadler and Hughes use backtracking and therefore less efficient; in addition Hughes' combinators do not provide optimality. More elaborated versions with linear-time optimal implementation are presented in [4,10,12,13]. However, these works contribute more to functional programming than to pretty-printing as such since all of them provide more advanced functional implementations of the same approach.

The problem with mainstream pretty-printer combinators is their weak expressivity. They treat pretty-printing programs *too uniformly* which sometimes can result in undesirable (or even incorrect) output. For example, if we pretty-print Python programs, then we have to handle printing two operators on the same line essentially differently since in this case additional separator character (";") is required. However, mainstream pretty-printer combinators do not provide any means to specify such a behavior. Another example is layout-based syntax which is generally cannot be generated by mainstream pretty-printers since they treat vertical and horizontal spaces uniformly. It is impossible for mainstream pretty-printer combinators to adopt various project-specific layouts — they always generate text in a single hardcoded style. While some of these problems can be handled with other approaches [14] the resulting solutions utilize much more advanced machinery then a small set of high-order functions.

The aforementioned deficiencies can be alleviated if the set of pretty-printing primitives is extended by the notion of *choice* between various layouts. Thus, instead of making it possible to freely break line at arbitrary space, we may describe different *variants* of layouts and let pretty-printing algorithm choose the best one. The ability to choose between different layouts gives rise to the ability to support various code styles since each of them can be expressed as a set of patterns to choose from.

Pretty-printing frameworks with choice are already considered in the literature; however, none of them are optimal and efficient at the same time.

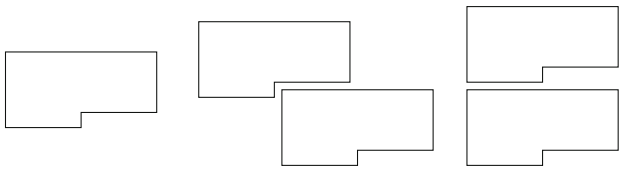
[1] The sets of combinators suggested by Hughes and Wadler slightly differ in details; however both of them share a similar relevant properties.

Pretty-printers described in [8,9] make use of the "ALT" operator which gives an opportunity to express non-trivial layout variations. However, provided implementation does not deliver optimal layout because the choice is based on the first document fitting in the given width. In contrast, pretty-printer combinators described in [2] have no lack in expressiveness and generate optimal layout but proposed algorithm has exponential complexity. Though authors later [3] discuss some heuristic, this doesn't change the worst-case behavior.

The contribution of this paper is optimal and polynomial-time re-implementation of combinators with choice described in [2]. We utilize bottom-up tree rewriting and dynamic programming (BURS) [1,5] to provide an optimal pretty-printing algorithm which has linear complexity on the number of nodes in the document, polynomial on the width of output and exponential only on the document tree degree (which we consider a constant). Our implementation does not make use of lazy evaluation and can be natively expressed in both strict and non-strict languages.

2 Pretty-Printing Combinators with Choice

Pretty-printer combinators with choice were introduced in [2]; the implementation we refer to (and compare with) is a part of Utrecht Tools Library[2] (there are some negligible differences between published and implemented versions).



(a) Format shape (b) Beside composition (c) Above composition

Fig. 1. Format primitives

Output text in this approach is built from blocks shaped as rectangles with possibly incomplete last line (Fig. 1a). In Haskell implementation blocks are represented by the values of type `Format`:

```
data Format = Elem {height        :: Int
                   , lastLineWidth :: Int
                   , width         :: Int
                   , txtstr        :: Int → String → String
                   }
```

The first three fields specify the geometric properties of the block; the last one is a content-generation function which is needed to provide a linear-time block-to-text conversion in a functional settings. The first integer argument of `txtstr` is an indentation of the whole block (so blocks can be moved right horizontally).

[2] http://www.cs.uu.nl/wiki/HUT/WebHome.

Table 1. Format and document manipulation primitives

`s2fmt`	`:: String → Format`	`text`	`:: String → Doc`
`indentFmt`	`:: Int → Format → Format`	`indent`	`:: Int → Doc → Doc`
`aboveFmt`	`:: Format → Format → Format`	`beside`	`:: Doc → Doc → Doc`
`besideFmt`	`:: Format → Format → Format`	`above`	`:: Doc → Doc → Doc`

There are four primitives working with formats (see Table 1, left column). Function `s2fmt` creates a single-line `Format` from an atomic string; `indentFmt` moves the whole block right by a given number of positions. Two composition primitives `besideFmt` and `aboveFmt` combine two layouts as shown on the Fig. 1b, 1c.

The next notion in the framework is *document*. We may consider document as a set of various layouts for the same text. Documents are represented by the values of type `Doc` which we leave abstract by now. The right column in the Table 1 shows four primitives for documents which are dual for those for formats.

In addition there is fifth operator for documents:

```
choice :: Doc → Doc → Doc
```

which denotes the *union* of two sets of layouts. Note that `choice` is the only primitive which can produce multi-variant layouts from the single-variant arguments.

From an end-user perspective, first, the document is created by means of these five combinators. Then, the document can be rendered using the function

```
pretty :: Int → Doc → String
```

which takes the output width and the document and provides optimal layout.

The original implementation essentially relies on lazy evaluation. In [2] `Doc` data type is represented as a (lazy) list of all possible layouts for a given width. This list is sorted so "better" layouts come first. In the case of "beside" or "above" document composition the complexity of the new document construction is $O(n \times m)$ where n and m are lengths of the first and the second layout lists. The new document also has length $O(n \times m)$.

The document rendering function just takes the head of the document layouts list. So, at the moment of rendering we need only its first element. Due to lazy evaluation this may reduce the overall complexity. However, the implementation of `beside` combinator in [2] triggers the full calculation of its both parameters which compromises the benefits of lazy evaluation. Thus, the calculation of layout in [2] has the worst-case exponential complexity on the number of combinators used to construct the document. We do not see any way to avoid this drawback while preserving list-based representation.

Despite a poor worst-case behavior optimal pretty-printing combinators with choice can be used in many practical cases.

3 Bottom-Up Rewrite Systems

Bottom-up rewrite systems (BURS) [1] is a dynamic programming framework initially developed in the context of research on instruction selection problem for machine code generation. The core notion of BURS is a weighted regular tree grammar [5] i.e. a grammar with the following two kinds of rules:

$$N : \alpha\ [c] \text{ and } N : \alpha\ (K_1, \ldots, K_n)\ [c]$$

Here N, K_i are nonterminals, α — terminal, c — cost functions (one per rule, see below). Similar to the ordinary *"linear"* (or *"word"*) grammar we distinguish certain starting nonterminal S and say that terminal-labeled *tree* is derivable in the given grammar if it can be constructed from a single node labeled by S using repetitive substitutions. Each substitution replaces (arbitrary) leaf labeled by a nonterminal N with a tree $\alpha\ (K_1, \ldots, K_n)$ if there is a rule $N : \alpha\ (K_1, \ldots, K_n)$ in the grammar. The cost functions are used to calculate the overall cost of a certain derivation. Each cost function can depend on the terminal label (α) and derivation costs for each subtree.

In the context of BURS we are interested in (arbitrary) least-cost derivation of a certain tree provided by a certain grammar. This derivation can be found by a two-pass algorithm.

The first pass (*labeling*) traverses the subject tree bottom-up and calculates for each its node the set of all triplets $(K,\ R,\ c)$, where K — nonterminal from which the subtree rooted at the given node can be derived, R — the first rule of the minimal derivation from K, c — the cost of this derivation. The labeling process is performed as follows:

- for each leaf node labeled by a terminal α we add into the set for this node a triplet $(K,\ R,\ c\,(\alpha))$ for each rule $R = K :\ \alpha\ [c]$;
- for an intermediate node labeled by a terminal α with immediate successors $v_1, \ldots, v_n$ we add into the set for this node a triplet $(K,\ R,\ c(\alpha, c_1, \ldots, c_n))$ for each rule $R = K :\ \alpha\ (K_1, \ldots, K_n)\ [c]$ such that there is a triplet $(K_i,\ R_i,\ c_i)$ in the labeling for the node v_i; if there are different suitable rules for the same nonterminal K then we choose that delivering minimal cost.

The second pass (*reduce*) is a top-down traversal which makes use of the constructed labeling. The first rule of minimal derivation is that from the triplet $(S,\ R,\ c)$ for the root node (if there is no such a triplet, then there is no derivation from S). This rule unambiguously determines nonterminals K_i for each direct subtree of the root node and the process repeats.

To perform labeling we potentially need to try each rule of the grammar for each node of the tree; given the fixed grammar this results in $O\,(|R|)$ complexity, where $|R|$ is the number of rules (the size of the set of triples is limited by the number of nonterminals which in turn is not greater than the number of rules). Reduce is linear as well.

4 Pretty-Printing via BURS

The reduction of the optimal pretty-printing problem to a BURS is based on the following observations. Let w be the output width. Since the approach in question deals with formats (rectangular boxes of a certain shape, see Sect. 2), the rendered text is in turn always shaped as a box. Let parameters of this box are n, k, h, where n — its width, $k \leq n$ — the length of its last line, h — its height. Under these considerations an *optimal* rendering is that with the minimal h over all pairs (n, k) such that $k \leq n \leq w$. So for a fixed width w we may try to render the text as no more than w^2 boxes and then simply choose the best one.

The document for pretty-printing can be considered as a tree built of primitives `text`, `indent`, `beside`, `above`, and `choice`. The main question is whether the rendering can be done *compositionally* by the tree structure (i.e. by reusing renderings for the subtrees of each node). It can be done if we memoize for each node and each pair (n, k), where $k \leq n \leq w$, the minimal h such that the subtree rooted at that node can be rendered as a box with parameters n, k and h. Having optimal renderings for each possible box shape for each subtree of some node we can in turn calculate optimal rendering for each possible box shape for the node itself *et cetera*. For each node of the tree we thus need to calculate no more then w^2 renderings which means that (under the assumption that w is fixed) the number of renderings is linear on the number of nodes in the tree.

Once all these renderings are calculated in a bottom-up manner we may then reconstruct the optimal one by a top-down traversal. For the root of the tree we choose the rendering with the minimal height. This choice immediately provides us with the renderings for immediate subtrees *et cetera*. Note that generally speaking the optimal rendering for a tree is not necessarily combined from optimal renderings for its subtrees.

These considerations boil down to the following BURS specification. Given output width w we introduce a family of nonterminals T_n^k for all $k \leq n \leq w$. We are going to define a BURS grammar in such a way that a derivation of cost h of some document tree from the nonterminal T_n^k corresponds to the optimal rendering of that document into a box with parameters n, k and h. Once we have a grammar with this property the labeling stage will calculate all (interesting) renderings while reduce stage will provide the optimal one.

The grammar in question can be constructed by the case analysis:

1. For a terminal node [`text s`][3] we have two cases:
   - if $|s| \leq w$ (where $|s|$ is the length of the string s) we introduce the single rule $T_{|s|}^{|s|} : [\texttt{text}\ \ \texttt{s}]$ with cost 1; for all other $k, n \neq |s|$ we have $T_n^k : [\texttt{text}\ \ \texttt{s}]$ with cost ∞;
   - if $|s| > w$ then we have $T_n^k : [\texttt{text s}]$ with cost ∞ for all k, n.

   Indeed, a (single-line) string of length $|s|$ can only be rendered as a box with parameters $|s|$ (width), $|s|$ (length of the last line) and 1 (height). All other renderings are not possible — hence "∞" cost.

[3] We use square brackets to denote multi-symbol terminals.

2. For a node [`indent m`] we introduce two sets of rules:
   (a) $T^{k+m}_{n+m} : [\texttt{indent m}]\,(T^k_n)$ with identity cost function for each n and k such that $n+m \le w$ and $k \le n$;
   (b) $T^k_n : [\texttt{indent m}]\,(T^i_j)$ with cost ∞ for all other cases.
   Clearly, shifting a box with parameters n, k and h by m positions to the right transforms it into the box with parameters $n+m$, $k+m$, h. This box represents an admissible rendering if $n+m \le w$ (and hence $k+m \le w$).
3. For a node [`above`] we have the rule $T^{k_2}_{\max(n_1,n_2)} : [\texttt{above}]\,(T^{k_1}_{n_1},\ T^{k_2}_{n_2})$ with the cost function which sums the costs of both subtree derivations for each $k_1 \le n_1 \le w$ and $k_2 \le n_2 \le w$. Indeed, when we combine boxes with parameters n_1, k_1, h_1 and n_2, k_2, h_2 we obtain the box with parameters $\max(n_1, n_2)$, k_2, $h_1 + h_2$. Vertical combination of two admissible boxes is always admissible.
4. For a node [`beside`] we have the rule $T^{k_1+k_2}_{\max(n_1,\,k_1+n_2)} : [\texttt{beside}]\,(T^{k_1}_{n_1},\ T^{k_2}_{n_2})$ for each combination of n_1, n_2, k_1, k_2 such that $k_1 + k_2 \le \max(n_1,\ k_1 + n_2) \le w$. The cost function for these rules calculates the sum of costs for subtree derivations minus 1. This can be validated by elementary geometric considerations.
5. Finally, for [`choice`] we have the rule $T^k_n : [\texttt{choice}](T^k_n, T^k_n)$ for all $k \le n \le w$. The cost function is minimum between two derivations for subtrees. Clearly, between two layouts with the same shape but different height we have to choose the shortest one.

To complete the construction we have to provide rules for the starting nonterminal S. We can either add a rule $S :\ r$ with identity cost function for each right-hand side r of each rule constructed so far or introduce a chain rule $S : T^k_n$ with identity cost function for each nonterminal T^k_n (the latter requires a trivial extension of BURS description presented in Sect. 3).

The number of nonterminals in the constructed grammar is $O(w^2)$; the number of rules, however, is $O(w^4)$ since there are nodes of degree 2 in the tree. So our BURS implementation of the optimal pretty-printer works in linear time on the number of nodes in the document tree for fixed width; the complexity on width is of fourth degree. Clearly, given reduction can be scaled to document construction primitives of arbitrary degree at the cost of exponential growth.

5 Implementation and Evaluation

We implemented our approach in Haskell as a pretty-printing combinator library[4] Our implementation borrows some basic underlying types and functions from the original library [2] with top-level types and combinators re-implemented.

In our implementation we do not follow BURS reduction literally; we do not make any use of BURS grammar, sets of nonterminals or standard algorithm. Instead, we calculate for each node of the document a map from pairs of integers (n, k) to the best ("shortest") format with the parameters n and k (if any). Thus, an entry (n, k) in the map corresponds to the cost of optimal derivation from T^k_n

[4] http://github.com/anlun/polynomialPPCombinators.

Table 2. Time of layout calculation (in seconds)

Height	Nodes	W=25		W=50		W=100		W=150	
7	10921	0.07	0.18	0.12	0.69	0.06	0.97	0.06	1.01
8	43689	1.28	0.73	287.55	4.03	97.79	14.92	38.99	20.75
9	174761	6.47	2.86	294	17.93	179	88.71	59	204.51
10	699049	18.46	11.58	284	71.70	172	390.45	58.25	1026.44

and the first rule for that derivation. At the top level we choose the least-cost element from the map.

Since we are interested in the worst-case behavior we evaluate our implementation on the number of artificial automatically-generated documents. Given a tree of type `Ast`, we then generate a document in a bottom-up manner. For each intermediate node we combine its subtrees' layouts both vertically and horizontally and generate a choice between them in the following manner:

```
data Ast = T String | N String Ast Ast
astToDoc (T s)     = text s
astToDoc (N s l r) = make beside `choice` make above where
   make f = foldl f (text s) (map astToDoc [l, r])
```

The results of comparison of our implementation against the original one are shown on the Table 2. Here "Height" stands for the height of the initial tree, "Nodes" — for the number of nodes in the generated document, "W" — for the output width. For each width the left sub-column shows the running time of the original implementation, while the right — of ours (in seconds). We can see that starting from some combination of width/number of nodes the original implementation was not able to calculate the layout. Table entries like 59 show the time when a stack overflow occurred.

Our implementation sometimes does not demonstrate linear behavior (as it is expected since the number of nodes is virtually quadrupled from line to line). We performed additional experiments and found that this phenomenon is due to the irregular sparsity of calculated layouts for the larger widths. In other words, for a small tree the number of non-empty entries in the layout maps is far below the upper bound. As the tree grows this number also grows non-linearly until the upper bound is reached.

6 Conclusions and Future Work

Despite our approach is better in asymptotic sense, the constant factor of w^4 makes it still unusable in a direct form for large widths. Several ways to reduce this factor may be considered as directions for future research. For example, we may try to factorize output width into smaller number of values or to perform auxiliary heuristic search to prevent too many layouts from being considered.

On the other hand, the presented approach has an interesting "relocation" property: if we once calculated layouts for some tree, then we can instantly pretty-print it in arbitrary context (e.g. from arbitrary position or as a subtree

of arbitrary tree). This property opens some perspectives for incremental pretty-printing in the context of IDEs.

Another issue which we have to mention is a conversion from AST into documents. Generally speaking the direct conversion might provide a document of an exponential size since at each node we might try to choose from various compositions ("beside" or "above") of layouts of its descendants. While this does not compromise our approach, it still has practical impact. To cope with this issue an additional level of memoization is needed to prevent shared document nodes from being processed several times. The original set of combinators [2] seem to face the same problem.

References

1. Aho, A.V., Ganapathi, M., Tjiang, S.W.K.: Code generation using tree matching and dynamic programming. ACM Trans. Program. Lang. Syst. **11**(4), 491–516 (1989)
2. Azero, P., Swierstra, S.D.: Optimal Pretty-Printing Combinators. http://www.cs.ruu.nl/groups/ST/Software/PP, (1998)
3. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and implementing combinator languages. In: Swierstra, S.D., Oliveira, J.N., Henriques, P.R. (eds.) AFP 1998. LNCS, vol. 1608. Springer, Heidelberg (1999)
4. Chitil, O.: Pretty printing with lazy dequeues. ACM Trans. Program. Lang. Syst. **27**, 163–184 (2005)
5. Comon, H., Dauchet, M., Gilleron, R. et al.: Tree Automata Techniques and Applications. http://www.grappa.univ-lille3.fr/tata, (2007)
6. Hughes, J.: The design of a pretty-printing library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925. Springer, Heidelberg (1995)
7. Jackson, S., Devanbu, P., Ma, K.: Stable, flexible, peephole pretty-printing. J. Sci. Comput. Program. **72**(1–2), 40–51 (2008)
8. De Jonge, M.: A pretty-printer for every occasion. In: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (2000)
9. De Jonge, M.: Pretty-printing for software reengineering. In: Proceedings of the International Conference On Software Maintenance (2002)
10. Costanzo, D., Shao, Z.: A case for behavior-preserving actions in separation logic. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 332–349. Springer, Heidelberg (2012)
11. Oppen, D.C.: Pretty-printing. ACM Transact. Program. Lang. Syst. **2(4)**, 465–483 (1980)
12. Swierstra, S.D.: Linear, Online, Functional Pretty Printing (corrected and extended version). Technical report, UU-CS-2004-025a. Institute of Information and Computing Sciences, Utrecht University (2004)
13. Swierstra, S.D., Chitil, O.: Linear, bounded, functional pretty-printing. J. Funct. Program. **19**, 1–16 (2009)
14. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Trans. Softw. Eng. Methodol. **5**(1), 1–41 (1996)
15. Wadler, P.: A Prettier Printer: The Fun of Programming. Palgrave MacMillan (2003)