A New Approach to Optimal Code Formatting

Phillip M. Yelland

Google Inc.

1600 Amphitheatre Parkway, Mountain View, CA 94043, USA.

e-mail: phillip.yelland@gmail.com

Abstract

The rfmt code formatter incorporates a new algorithm that optimizes code layout with respect to an intuitive notion of layout cost. This note describes the foundations of the algorithm, and the programming abstractions used to facilitate its use with a variety of languages and code layout policies.

1 Introduction

rfmt (Yelland 2015) is a new source code formatter for the R programming language. Though the program itself is particular to R, it embodies a language-independent approach to source code layout that seeks an "optimal" rendering of a program with respect to an intuitively-appealing notion of layout cost. This paper describes the layout algorithm used in rfmt. In the next section, we note the prior work to which it is related. Subsequent sections detail the way in which alternate program layouts are provided to the algorithm, and the way in which the algorithm chooses the optimal layout.

1.1 Related work

Methods for ensuring that printed output is appealingly formatted date from the earliest days of computing (Harris 1956). With the widespread adoption of high-level programming languages, it is perhaps natural, therefore, that programs were devised to format software source code itself (Scowen et al. 1971). For the most part, such formatting—or *pretty printing*, as it came to be known—seeks to improve the readability of source code by breaking it into lines and indenting it by inserting whitespace characters.[1]

[1] Hughes (1995) draws a distinction between between *pretty printing*, which he reserves for the legible rendering of internal data structures, and *source code formatting*—improving the readability of program text. As he observes, the latter involves considerations such as the proper placement of comments (which are regarded as extra-syntactic contructs in most languages). In this paper, we use the terms interchangeably, since the intent is to describe the layout algorithm used by the source code formatter rfmt. It should be noted, however, that for the most part, the

LISP (McCarthy 1960) has provided a particularly fertile environment for code formatting; LISP programs are themselves lists (LISP is *homoiconic*, to use Kay's (1969) term), and without at least a modicum of formatting, their printed representation is all but illegible. An early survey paper by Goldstein (1973) examines the design space of LISP pretty printers, and describes a search algorithm with limited look ahead—which he calls the *recursive re-predictor*—implemented by one of the earliest, GRINDEF (Gosper 2015).

An influential paper by Oppen (1980) describes a language-independent pretty printing algorithm based—like GRINDEF—on a limited-lookahead search. Input to Oppen's algorithm takes the form of program source code, annotated so as to break it into (possibly nested) *logical blocks*, with separators that the algorithm may render either as spaces or as line breaks (accompanied by suitable indentation). Annotations are attached to the source code to reflect its syntactic structure—a conditional statement, for example, might constitute a logical block, with the positive and negative arms of conditional nested as logical blocks themselves. The structure of the logical blocks delineate the alternate formats that are open to exploration by the algorithm, which in the example, might chose to print the entire conditional statement on a single line, to split begin the statement and its arms on separate lines, and so on. Oppen's use of annotations to present possible formatting choices to a layout algorithm is mirrored in more recent language-independent code formatting systems, such as those described in (Jokinen 1989) and (van den Brand 1993).

Functional programming languages are heirs to the LISP tradition, and so the development of pretty printers in functional languages such as Haskell and ML may be unsurprising. In the main, however, from the pioneering work of Hughes (1995) in this area, through the developments of Wadler (1999) and recent contributions such as (Chitil 2005), the emphasis has been on the creative application of functional programming techniques to the problem, rather than on algorithms for pretty printing *per se*—Wadler's (1999) pretty printer, for example, is based on a lazy functional version of Oppen's (1980) algorithm. This paper has the contrary orientation: We concentrate less on programming techniques than on algorithms for code formatting. The work described here does nonetheless draw from this line of research in using an abstract data type with values created by a set of generating functions or *combinators* to describe alternate source code formats.

Closer to the actual formatting algorithm in this paper—in particular its reliance on dynamic programming to find a layout optimal with respect an explicit cost function—is the algorithm used by the TEX typesetting system. Here the objective is to produce well-justified paragraphs by breaking their constituent words into lines. Knuth's algorithm, later expanded upon by Knuth and Plass (1981), uses dynamic programming to minimize the sum of the squares of the width of the white space left at the end of each line.

A notable recent development has been the development of pretty printers—like the one described here—which employ an explicit cost function (in the manner of the TEX) that is optimized by a layout algorithm. The progenitor of these is **clang-format** (Jasper 2013), a source code formatter for C/C++. Though broadly similar in approach, **clang-format** and its

scope of this paper (which does not cover the treatment of comments, for example) is restricted to those aspects of `rfmt` that facilitate "pretty printing" in Hughes' narrower sense.

scion differ substantially from the formatter in this paper: **clang-format** begins by breaking source code into "unwrapped lines," usually corresponding to C/C++ language statements. **clang-format** assumes that output width is limited, and inserts potential line breaks into the each unwrapped line so as to fit it to the output width. Each potential line break is assigned a cost, and the program uses them to construct a weighted graph connecting the initial unwrapped state of the line with "solution" nodes that represent a broken version of the line with horizontal extent no greater than the output width. Dijkstra's (1959) algorithm is used to find a path from the source node to a solution that is of minimal cost. By contrast, here we use combinators to specify alternate candidate layouts directly—in `rfmt` these layouts are derived in a(n arguably more natural) syntax-directed fashion from a parsed representation of the language source. Rather than assuming a fixed output width and optimizing layout cost conditional on satisfying the width restriction, here output width is constrained by the cost function itself, which affords us a greater degree of flexibility (such as the ability to insert a "soft margin," as discussed later in section 7). Finally, we use dynamic programming directly to optimize cost, instead of the more indirect approach taken by **clang-format** (Dijkstra's algorithm itself involving a form of dynamic programming).

2 Code layout

As observed in the introduction, in its initial stages, `rfmt` takes the same approach to pretty printing as that taken by Hughes and his fellow functional programmers: To wit, data structures specifying alternative formats for a piece of source code are provided to the layout algorithm, which selects a format that is minimal with respect to a specified cost function. In this section, we describe these data structures—termed *layout expressions* here[2]—by way of the *combinator* functions that are used to construct them. The combinators introduced in this section constitute a set of "primitives," on which the more practical set of layout contructors offered in `rfmt` are based. In section 6, we show how the `rfmt` constructors may be built from these primitive combinators.

2.1 Layouts and Combinators

Four combinators are used in this paper, the first three of which are illustrated in figure 1. These three combinators are defined informally as follows:

'*txt***'** A layout expression consisting only of the text string *txt* (which is assumed not to contain formatting characters such as carriage returns, tabs and the like), to be output on a single line.

$l_1 \updownarrow l_2$ A layout expression comprising two layout expressions l_2 and l_2 "stacked" vertically, with l_1 above l_2. When output, the first character of both layouts fall into the same column, and the first line of l_2 is immediately below the last line of l_1.

[2]Layout expressions correspond roughly to the *pretty documents* or `Docs` in Hughes' paper. Where convenient, we will often use the term "layout" rather than "layout expression".

Figure 1: Three layout combinators

$l_1 \leftrightarrow l_2$ A layout expression that juxtaposes expressions l_1 and l_2, with l_2 to the right of l_1 when output. Note that in general, both components expression may contain multiple lines of unequal span, and we follow Hughes (1995, p. 19) and later Wadler (1999) in placing the first character of l_2 on the same line as and immediately to the right of the last character of l_1, translating l_2 bodily rightwards as shown in figure 1.

We can generate a wide variety of code layouts using these three combinators. As a simple illustration, the following expression:

$$(\text{`if (voltage[t] < LOW\_THRESHOLD)'} \updownarrow \text{` '}) \leftrightarrow \text{`LogLowVoltage(voltage[t])'}$$

specifies the formatted C conditional statement:

```
if (voltage[t] < LOW_THRESHOLD)
    LogLowVoltage(voltage[t])
```

2.2 Choosing layouts

Observe that the code above may also be formatted using the layout expression:

$$(\text{`if (voltage[t] < LOW\_THRESHOLD)'} \leftrightarrow \text{` '}) \leftrightarrow \text{`LogLowVoltage(voltage[t])'}$$

which specifies:

```
if (voltage[t] < LOW_THRESHOLD) LogLowVoltage(voltage[t])
```

Syntactically and semantically, of course, both forms of the code are indistinguishable, since C is largely oblivious to white space. They do, however, represent different trade-offs between the horizontal and vertical space occupied by a piece of code, since the first layout occupies one more line than does the latter, but it has a smaller horizontal extent. This sort of trade-off becomes particularly pointed when restrictions are placed on the total width of formatted code; prodigious modern-day screen widths notwithstanding, most code layout is still obliged to honor (where possible) a fixed right-hand margin that restricts the overall

output to 80 character widths, or *columns*. On the other hand, other things being equal, it is generally desirable to minimize the vertical space occupied by a piece of code. To capture the trade-off in quantitative terms, we associate a *cost* with a code layout: When code is output according to the layout, we incur a cost of β units for each character beyond the right margin, and a cost α for each line break output.[3] Given a collection of alternative layouts for the same piece of code, we should select one whose cost is on output is minimal.

Of course, on its own, the code fragment above is unlikely to breach an 80 character right margin if output according to any reasonable layout. Imagine it, however, nested inside other code structures, as illustrated in figure 2, for instance. In such circumstances, we might opt for the first layout for the conditional statement, trading off an extra line in order to avoid breaching the right margin might—that is, incurring a cost of α units so as to save $n\beta$ units, for some number $n \geq 1$ of characters which might otherwise lie beyond the margin.

```
for (t = 0; t < n; t++) if (voltage[t] < LOW_THRESHOLD) LogLowVoltage(voltage[t])
```

Figure 2: Conditional statement nested in another construct

To make such choices, we have a fourth layout combinator: Given layout expressions l_1 and l_2 (which normally represent alternate ways of formatting the same code), output of any layout expression containing $l_1 ? l_2$ results in the output whichever of l_1 and l_2 incurs the lowest overall cost. For example, in the case of the conditional statement, the following represents the choice between the alternate layouts given above:

$$\begin{aligned} l_{\text{if}} \quad = \; & [(\text{`if (voltage[t] < LOW\_THRESHOLD)'} \updownarrow \text{`} \quad \text{'}) ? \\ & (\text{`if (voltage[t] < LOW\_THRESHOLD)'} \leftrightarrow \text{` '})] \leftrightarrow \text{`LogLowVoltage(voltage[t])'} \end{aligned}$$

Effective implementation of the choice combinator crucial to efficient code layout. This is because in order to decide which of the component layout expressions l_1 and l_2 to select, it is necessary to consider the context in which the choice expression $l_1 ? l_2$ appears. To demonstrate, return to the nested code constructs presented in figure 2, and suppose that the `for` loop itself has two alternate formats:

$$l_{\text{for}} = (\text{`for (t = 0; t < n; t++)'} \updownarrow \text{`} \quad \text{'}) ? (\text{`for (t = 0; t < n; t++)'} \leftrightarrow \text{` '})$$

The layout of the code in figure 2 may then be expressed as $l_{\text{for}} \leftrightarrow l_{\text{if}}$, where the choices in l_{for} and l_{if} capture the different options involved in the layout of both the `for` loop and the `if` statement. Note, however, that the choice of component layout in l_{for} affects the horizontal position (or "column") at which the output of l_{if} begins. This in turn affects the relationship of l_{if} to the right margin, and thus the costs of the components of l_{if}. To decide on the lowest-cost component of l_{if}, therefore, we need to take into account the choices made in l_{for}.

[3] That is, one less than the number of lines occupied by the code.

Generalizing, in the composite layout:

$$\underbrace{(l_{11} \,?\, l_{12})}_{l_1} \leftrightarrow \underbrace{(l_{21} \,?\, l_{22})}_{l_2} \leftrightarrow \ldots \leftrightarrow \underbrace{(l_{n-1,1} \,?\, l_{n-1,2})}_{l_{n-1}} \leftrightarrow \underbrace{(l_{n1} \,?\, l_{n2})}_{l_n} \tag{1}$$

the choice of layout in sub-expression $l_n = l_{n1} \,?\, l_{n2}$ depends potentially on all of the choices made in sub-expressions l_1 through l_{n-1}. A naïve implementation of the "?" combinator would enumerate each of the n choices involved, entailing examination of 2^n layout combinations. Clearly, exponential complexity of this kind would lead to unacceptable performance for source programs of even moderate length.

2.3 Dynamic programming and optimal layouts

A more practical implementation of the "?" combinator starts with the observation that in expression (1), for $i = 1, \ldots, n-1$, the influence that the choice in layout l_i has on that in layout l_{i+1} and its successors is mediated entirely by the starting column for l_{i+1} determined by the choice in l_i. So the minimum overall cost for layout $l_i \leftrightarrow \ldots \leftrightarrow l_n$ can be calculated by computing: a) which of the component layouts l_{i1} and l_{i2} in l_i incurs the lesser cost at the current starting column, when added to b) the minimum cost choices for $l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$ given the new starting column fixed by the choice in a). This description of the layout problem suggests that it is amenable to solution using *dynamic programming*, first explored by Bellman (1957, ch. 3) and employed in a great variety of applications since (Skiena 2008, ch. 3).

Key to the use of dynamic programming in this application is the association of a layout expression with a function—call it the layout's *minimum cost function*—that maps a column to the minimum cost incurred by the layout when started at that column. By way of illustration, let us calculate the minimum cost function for $l_1 \leftrightarrow \ldots \leftrightarrow l_n$ by induction:

First, let f_{n+1} be the constant function $x \mapsto 0$ that maps any starting column to a cost of 0 units; this reflects the fact that the "empty layout" to the right of l_n incurs no cost, regardless of the column in which its output begins.

Then, for $i = n, \ldots, 1$, assume that we are given f_{i+1}, the minimum cost function corresponding to the layout expression $l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$. To calculate the minimum cost function for $l_i \leftrightarrow l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$, we need to know the horizontal extents or *spans* of the two component expressions in $l_i = l_{i1} \,?\, l_{i2}$. In general (as detailed later), these spans will depend on layout choices made for l_{i1} and l_{i2} themselves, but for simplicity, we will assume here that they are the constants s_{i1} and s_{i2} respectively. Therefore, if the output of $l_i \leftrightarrow l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$ starts at column x, and we choose component l_{i1}, output of $l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$ will begin at column $x + s_{i1}$. Similarly, if we choose l_{i2}, output will begin at column $x + s_{i2}$.

By assumption, the minimum cost of $l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$ when output at column $x + s_{i1}$ is $f_{i+1}(x + s_{i1})$, and $f_{i+1}(x + s_{i2})$ when output at $x + s_{i2}$. Finally, let the costs of outputting just the subcomponents l_{i1} and l_{i2} at column x be c_{i1} and c_{i2}, resp., which again are assumed to be constants for simplicity. The minimum cost associated with the output of $l_{i+1} \leftrightarrow \ldots \leftrightarrow l_n$ at column x reflects whichever choice of subcomponent results in the lowest overall cost, so its

minimum cost function, f_i, is:

$$x \mapsto \min\{c_{i1} + f_{i+1}(x + s_{i1}), c_{i2} + f_{i+1}(x + s_{i2})\}. \tag{2}$$

Once we have the minimum cost function for the entire expression $l_1 \leftrightarrow \ldots \leftrightarrow l_n$, the cost of outputting its optimal layout is simply $f_1(0)$, since the starting column for the entire expression is 0. And as we will see in the next section, by recording, for each x in (2), the component (l_{i1} or l_{i2}) that yields the minimum cost at x, we are able to reconstruct the optimal layout itself, as well.

Note that in deriving the optimal layout for the entire expression by this procedure, we were obliged to compute only n minimum cost functions. With a suitable means of calculating minimum cost functions, therefore, dynamic programming offers the prospect of avoiding the exponential complexity entailed by the naïve approach to the layout problem.

3 Calculating minimum cost functions

Since a minimum cost function simply maps a starting column to a cost and its corresponding optimal layout, an obvious implementation is simply a vector of costs and layouts, indexed by starting column. Unfortunately, there are drawbacks to this representation:

1) The number of elements in such a vector must equal the maximum starting column for any layout (call it $x_{\max}$). This is difficult to establish *a priori*, and to adjust it dynamically involves awkward reallocation and copying. Furthermore, once we have increased our estimate for $x_{\max}$, it is not obvious if and when we might decrease it.

2) For reasonably large values of $x_{\max}$, such a representation is likely to be inefficient in terms of space, and more importantly, in terms of time. To see the latter, observe that with this representation, the evaluating expression (2) above, for example, requires us to carry out at least $x_{\max}$ cost comparisons—one for each entry in the new minimum cost function.

In this section, we describe a more parsimonious and efficient means of deriving minimum cost function using piecewise constant functions, which we dub *layout functions*. To construct layout functions, we implement a set of combinators that parallel those in section 2.1.

3.1 Knots

Fundamentally, a *layout function* is simply a function defined on a set of *knots*. Here, a knot is a positive integer representing a starting column for a layout; the knots of a layout function represent starting columns at which the value of the function changes—between the knots, the value is assumed to remain constant.

More formally, a *knot set* K, is a (finite) set of positive integers (*knots*), such that $0 \in K$. We define two operations on knot sets that help locate knots associated with given column positions.

First, given a knot set K, for $x \in [0, \infty)$, define:

$$x_K^- = \max\{k \in K \mid k \leq x\}. \tag{3}$$

Intuitively speaking, x_K^- is the rightmost knot in K at or to the left of x.

Correspondingly, define:

$$x_K^+ = \begin{cases} \infty & \text{if } x \geq \max K, \\ \min\{k \in K \mid x < k\} & \text{otherwise.} \end{cases} \tag{4}$$

Informally, x_K^+ is the leftmost knot in K to the right of k, if such a member of K exists, and equals infinity otherwise.

Where the knot set is clear from the context, we will often drop the subscript on x_K^- and x_K^+, writing x^- for example, rather than x_K^-.

3.2 Layout functions

A layout function maps each knot in its knot set—which, to recall, represents output starting columns—to a tuple of four values:

1) A *layout expression* without any occurrences of the "?" operator. This denotes the layout—with all selections entailed by any "?" operators resolved—that is optimal for output beginning at the knot.

2) An integer giving the *span* of the optimal layout, i.e. the width of its last line in characters.

3) An *intercept*—a real number equal to the cost incurred by the output of the optimal layout at the knot.

4) A *gradient* specifying the amount by which the cost of output increases for each unit increase of the starting column beyond the knot.

Let the value of the layout function g at knot k be the tuple (l_k, s_k, a_k, b_k), comprising respectively the layout, span, intercept and gradient at knot k. We define accessor functions such that:

$$\begin{aligned} l_g(k) &= l_k, & s_g(k) &= s_k, \\ a_g(k) &= a_k, & b_g(k) &= b_k. \end{aligned}$$

If K is the knot set that constitutes the domain of g, using the "$.^-$" operator defined above, we can extend these accessors to (positive) starting column values in general. Thus for example, $a_g(x) = a_g(x_K^-)$ is the gradient that g associates with the knot immediately to the left of x; since the layout function is piecewise constant, $a_g(x)$ is also the value of the gradient from x_K^- up to (but not including) x_K^+.

With the intercept and gradient accessors defined for an arbitrary starting column, we recover the minimum cost function associated with a layout function by linear extrapolation from the closest knot; it is the function that maps x to the value $v_g(x)$, where:

$$v_g(x) = a_g(x) + b_g(x)[x - x^-]. \tag{5}$$

As before, when the layout function is evident from the context, we will often suppress the subscript on $l_g(k)$, etc. Furthermore, when dealing with an indexed collection of layout functions such as $g_1, \ldots, g_i, \ldots, g_n$, we will refer to $a_i(x)$, $b_i(x)$ and so on, rather than the more cumbersome $a_{g_i}(x)$, $b_{g_i}(x)$, etc.

When we need to define a particular layout function explicitly, we present a collection of entries of the form "$k \mapsto (l, s, a, b)$," where k is a knot value, and l, s, a and b are respectively the layout, span, intercept and gradient associated with that knot.

In terms of data structures, a layout function is expediently represented by an ordered vector containing its knots, and parallel vectors with the corresponding layouts, spans, etc. The operations defined above for layout functions, (namely the operators $.^-$, $.^+$, the accessor functions and v_g), may be implemented by scanning the knot vector,[4] retrieving such supplementary data as is required from the parallel vectors.

4 Combinators for layout functions

Having introduced the representation of layout functions, we move on to the definition of combinators constructing layout functions, analogous to those given for layout expressions.

4.1 A text string

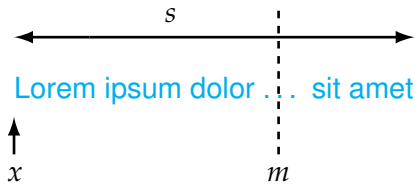

Figure 3: Output of a text string

We begin with the first kind of layout expression in section 2.1: Expressions of the form '*txt*', where *txt* is a text string (free of carriage returns, etc.). Assume that the text string *txt* consist of s characters—that is, it has a fixed span *span* s.[5] Consider the output of such

[4] Since knot sets are fairly small in practice, a linear scan suffices, though more efficient binary, hashed, etc. searches are also possibilities.

[5] For the sake of simplicity, we assume that $s \leq m$; the more general case is a fairly straightforward elaboration of that presented here.

a string, beginning in column x, as depicted in figure 3. Here, m denotes the right margin discussed in section 2.2.

Let us derive the minimum cost function[6] for this layout expression. Recall that output incurs a cost β for every character that projects beyond the margin—or equivalently, for every character width the end of a line falls to the right of the margin. Therefore, if when output, the first character of the text begins in column x, a cost of β units will be incurred for every character by which $x + s$ exceeds m. Since there are no choices involved in the output, this is also the minimum cost that might be incurred. Therefore the minimum cost function for this layout is:

$$x \mapsto \begin{cases} 0 & \text{if } x + s < m, \\ \beta[(x + s) - m] & \text{if } x + s \geq m. \end{cases} \tag{6}$$

Since x is required to be at least 0 (and finite), with little algebra we can restate this:

$$x \mapsto \begin{cases} 0 & \text{if } 0 \leq x < m - s, \\ \beta[x - (m - s)] & \text{if } m - s \leq x < \infty. \end{cases} \tag{7}$$

Inspecting the mapping in (7), it is not difficult to see that it constitutes the piecewise linear function illustrated in figure 4, consisting of a segment $[0, m - s)$ with gradient 0, followed by a segment $[m - s, \infty)$ with gradient β.[7]

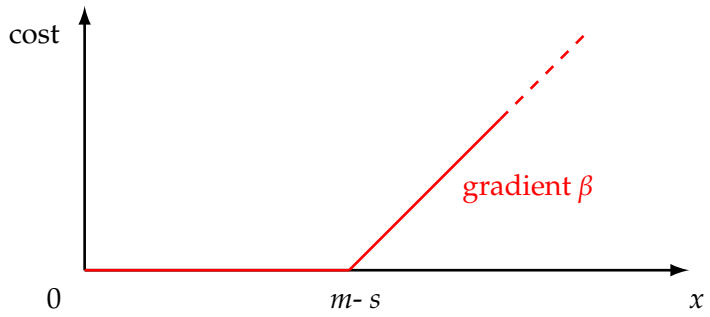

Figure 4: Costs of a single line of text as a piecewise linear function

This piecewise linear minimum cost function—along with other information needed to characterize the optimal layout—can be specified using a layout function on two knots, 0 and $m - s$. The beginning cost and gradient—both 0—in the segment $[0, m - s)$ of the minimum cost function are associated with knot 0, and knot $m - s$ is mapped to the beginning cost 0 and gradient β of the segment $[m - s, \infty)$. Since the span (namely s) and optimal layout ('*txt*') are the same in both segments, we have the full layout function:

[6] Not the *layout* function that *represents* this minimum cost function—that is derived later.

[7] Arguably, the figure should reflect the restriction of the starting column offset x to the integers, but for clarity we suppress this consideration here and throughout the paper.

Definition 1 *For text string txt consisting of s characters, let* $\langle$*'txt'*$\rangle$ *be the layout function:*

$$\left\{\begin{array}{r} 0 \mapsto (\text{'}txt\text{'}, s, 0, 0) \\ m - s \mapsto (\text{'}txt\text{'}, s, 0, \beta) \end{array}\right\} \tag{8}$$

4.2 Stacking

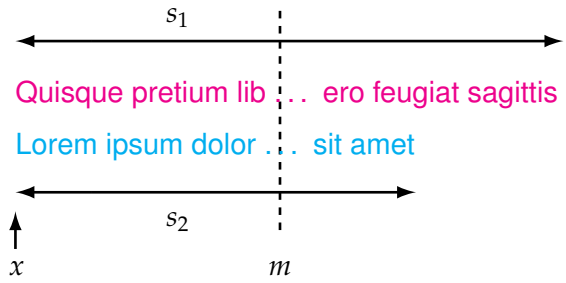

Figure 5: Two vertically-stacked lines of text

Moving on to the analog of the stacking combinator, consider by way of example the costs associated with the output depicted in figure 5—two lines of text with spans s_1 and s_2, stacked onto successive lines, both beginning at column x. Without loss of generality, we will assume that $s_1 \geq s_2$.[8] If we consider for the moment only the costs incurred by characters beyond the right margin, reasoning similar to that in the previous section yields the cost function:

$$x \mapsto \begin{cases} 0 & \text{if } 0 \leq x < m - s_1, \\ \beta[x - (m - s_1)] & \text{if } m - s_1 \leq x < m - s_2, \\ \beta[(m - s_2) - (m - s_1)] + 2\beta[x - (m - s_2)] & \text{if } m - s_2 \leq x < \infty. \end{cases} \tag{9}$$

Here, the three cases apply to starting positions for which resp. 0, 1, and 2 lines of text project beyond the margin at m (the reason for the rather stilted expression in case 3 will become apparent below).

This is not, however, a full account of the costs associated with the output depicted in figure 5. The discussion of costs in section 2.2 implies that since this output contains a line break, we need to add a constant penalty α to all values of the cost function. If we let $k_1 = m - s_1$

[8] Otherwise, simply reorder the subscripts associated with the lines.

and $k_2 = m - s_2$, we can write out the amended minimum cost function as:

$$x \mapsto \begin{cases} \alpha & \text{if } 0 \le x < k_1, \\ \alpha + \beta[x - k_1] & \text{if } k_1 \le x < k_2, \\ \alpha + \beta[k_2 - k_1] + 2\beta[x - k_1] & \text{if } k_2 \le x < \infty. \end{cases} \tag{10}$$

Note that the span of the stacked expression—the character width of its last line—is a constant s_2, and the layout expression output is the same regardless of the starting column. Let l_1 and l_2 denote the layouts of the two component lines in the figure, so that the stacked layout expression output is $l_1 \updownarrow l_2$. Reasoning as above we can derive the layout function for the layout expression depicted in figure 5:

$$\left\{ \begin{array}{l} 0 \mapsto (l_1 \updownarrow l_2, s_2, \alpha, 0) \\ k_1 \mapsto (l_1 \updownarrow l_2, s_2, \alpha, \beta) \\ k_2 \mapsto (l_1 \updownarrow l_2, s_2, \alpha + \beta[k_2 - k_1], 2\beta) \end{array} \right\} \tag{11}$$

Similar arguments apply to the derivation of layout functions for stacked layout expressions in general, but in the general case we work from the layout functions associated with each of the components expressions:

Definition 2 *For layout functions g_1 and g_2, with knot sets K_1 and K_2, let $g_1 \langle\updownarrow\rangle g_2$ be the layout function with knot set $K = K_1 \cup K_2$, and for each $k \in K$:*

$$\begin{aligned} l(k) &= l_1(k) \updownarrow l_2(k), \\ s(k) &= s_2(k), \\ a(k) &= v_1(k) + v_2(k) + \alpha, \\ b(k) &= b_1(k) + b_2(k). \end{aligned}$$

Verifying that this operation does indeed reflect the stacking operation in the case of the example in figure 5 is straightforward: Then we have $g_1 = \{0 \mapsto (l_1, s_1, 0, 0), k_1 \mapsto (l_1, s_1, 0, \beta)\}$ and $g_2 = \{0 \mapsto (l_2, s_2, 0, 0), k_2 \mapsto (l_2, s_2, 0, \beta)\}$, so that $g_1 \langle\updownarrow\rangle g_2$ has a combined knot set $\{0, k_1, k_2\}$, with corresponding tuples matching the result in (11).

4.3 Juxtaposition

Output of a layout expression involving the juxtaposition combinator "↔" is illustrated in figure 6. We can define an analogous juxtaposition operator on layout functions as follows:

Definition 3 *Given layout functions g_1 and g_2, with knot sets K_1 and K_2, let $g_1 \langle\leftrightarrow\rangle g_2$ be the cost function with knot set K:*

$$K = K_1 \cup \{k - t \mid k \in K_2 \text{ and } s_1(k - t) = t\},$$

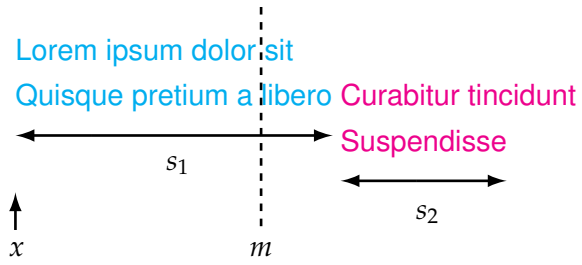

Figure 6: Juxtaposed outputs

and such that with $k' = k + s_1(k)$ *for each* $k \in K$*:*

$$\begin{aligned} l(k) &= l_1(k) \leftrightarrow l_2(k'), \\ s(k) &= s_1(k) + s_2(k'), \\ a(k) &= v_1(k) + v_2(k') - \beta \max(k' - m, 0), \\ b(k) &= b_1(k) + b_2(k') - \beta \mathcal{I}(k' \geq m), \end{aligned}$$

where the indicator expression $\mathcal{I}(k' \geq m)$ *is equal to 1 if* $k' \geq m$ *and 0 otherwise.*

The knot set of the juxaposed combination must contain all those offsets (the value x in figure 6) at which the gradient, intercept or span of the combination may change. Now any offset x of the combination places the left hand component at offset x, and the right hand component at that offset plus the span of the left hand component at that offset. Thus the knot set of the combination contains all the knots of the first operand, g_1, together with all those offsets which, added to the span of g_1 at that offset, coincide with a knot of g_2.

The layout and span of the combination at each knot are reasonably easy to calculate, provided we draw from the second component function at the appropriate offset (i.e. the sum of the knot and the span of g_1 at that knot). However, in calculating the gradients and intercepts of the combined cost function at each knot, we also need to account for the fact (illustrated in figure 6) that the end of the last line of g_1 is no longer a line end in the composition, since it becomes a prefix of the first line of g_2. This means that we must eliminate any contributions it makes to the gradients and intercepts of the combined function. As we saw in section 4.1, these contributions are only positive in as far as the end of the line projects beyond the right margin, m. For each knot, the end of the last line of g_1 (which is also the starting offset for g_2) is given by the quantity k' in definition (3), and so in the calculations of $b(k)$ and $a(k)$, we compare this quantity with the right margin on order to make the appropriate adjustments.

4.4 The choice operator

The analog of the final combinator—the choice operator, "?"—may be derived as a generalization of the motivating example given in section 2.3. An intuitive explanation follows its definition below.

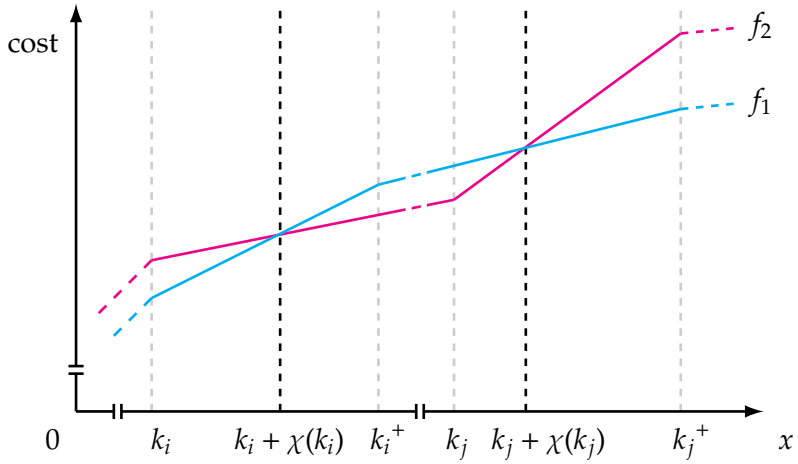

Figure 7: Costs associated with component layout functions

Definition 4 *Let g_1 and g_2 be given layout functions with knot sets K_1 and K_2. Let $L = K_1 \cup K_2$, and for each $k \in L$, let:*

$$\chi(k) = \frac{v_2(k) - v_1(k)}{b_1(k) - b_2(k)}. \tag{12}$$

Now, recalling the definition of k_L^+ from equation (4)*, let K be the knot set:*

$$K = L \cup \{\lceil k + \chi(k) \rceil \mid k \in L,\ \lceil k + \chi(k) \rceil < k_L^+\}, \tag{13}$$

where $\lceil x \rceil$ is the largest integer greater than or equal to x. Now for each $k \in K$, let:

$$\mu(k) = \begin{cases} 1 & \text{if } v_1(k) < v_2(k) \text{ or } [v_1(k) = v_2(k) \text{ and } b_1(k) \leq b_2(k)], \\ 2 & \text{otherwise.} \end{cases}$$

Finally, define $g_1 \langle ? \rangle g_2$ as the layout function with knot set K, such that for all $k \in K$:

$$l(k) = l_{\mu(k)}(k), \qquad s(k) = s_{\mu(k)}(k),$$
$$a(k) = v_{\mu(k)}(k), \qquad b(k) = b_{\mu(k)}(k).$$

Thus the layout function $g_1 \langle ? \rangle g_2$ associates a starting column x with the layout (and cost thereof) of either g_1 or g_2, depending on which layout yields the lowest cost when output at x. Since the layouts, spans, intercepts or gradients of g_1 and g_2 may change at their knots, the knot set of $g_1 \langle ? \rangle g_2$ contains at least the union of the component knot sets. In addition to these points, however, we must also consider those offsets between knots at which the costs of the constituent functions may cross. Two such instances are depicted in figure 7: There, knots k_i and k_j, as well as their immediate successors, k_i^+ and k_j^+, are taken from the knot sets

of the constituent functions.[9] Observe that while at k_i the cost of g_1 is less than that of g_2, the costs cross at a point between k_i and $k_i{}^+$. The distance of this latter point from k_i, $\chi(k_i)$, may be calculated from the values and gradients of g_1 and g_2 at k_i, as shown in equation (12). The upshot is that while the value (i.e. the tuple comprising layout, span, intercept and gradient) of $g_1 \langle ? \rangle g_2$ is that of g_1 at k_i, it must be set to the value of g_2 for all (integer) column offsets greater than or equal to $k_i + \chi(k_i)$, thus requiring another knot at $\lceil k_i + \chi(k_i) \rceil$; a similar situation pertains at k_j, with the rôles of g_1 and g_2 reversed. The full knot set K for $g_1 \langle ? \rangle g_2$, defined in equation (13), adds such intermediate knots as required, and the values in the new layout function are set accordingly.

5 Deriving layout functions from layout expressions

Given the analogs of the layout combinators defined in the previous section, it is straightforward to use structural recursion to define a function $C(\cdot)$, mapping a layout expression to its corresponding layout function; for layout expressions l_1, l_2, and text string t:

$$C(l_1 \leftrightarrow l_2) = C(l_1) \langle \leftrightarrow \rangle C(l_2) \tag{14}$$

$$C(l_1 \updownarrow l_2) = C(l_1) \langle \updownarrow \rangle C(l_2) \tag{15}$$

$$C(l_1 \,?\, l_2) = C(l_1) \langle ? \rangle C(l_2) \tag{16}$$

$$C(\text{`}t\text{'}) = \langle \text{`}t\text{'} \rangle. \tag{17}$$

Unfortunately, though the definition of $C(\cdot)$ is appealingly straightforward, it does not yield a practical solution to the code formatting problem. Again, the source of the difficulty is the "?" operator—in particular, its interaction with the juxtaposition operator "↔". To illlustrate why this is, return to expression (1), and note that if we are to decide on the optimal choice in subexpression $l_{11} \,?\, l_{12}$, the effect of the choice on the juxtaposed layouts $l_{i+1} \ldots l_n$ must be taken into account. This means that the layout function for $l_{11} \,?\, l_{12}$ (which must reflect this optimal choice) depends on $l_{i+1} \ldots l_n$. Equation (16), however, stipulates that the layout function $C(l_{11} \,?\, l_{12}) = C(l_{11}) \langle ? \rangle C(l_{12})$ depends only on l_{11} and l_{12}.

To address this problem, we restrict ourselves to "expanded" expressions in which the only subexpressions involving horizontal juxtaposition operator "↔" are of the form $\text{`}t\text{'} \leftrightarrow l$, where l is also an expanded layout expression. This means in particular that there are no occurrences of subexpressions involving the choice operator "?" in the left operand of a juxtaposition. Thus all information about juxtaposed layouts required to make the choice implied by the "?" operator are present in its operands, in keeping with the requirements of equation (16).

No loss of generality is entailed by the restriction to expanded expressions because we can define a function $E(\cdot)$, which transforms any layout expression to its expanded equivalent. First, let $\square$ denote the empty layout expression; note that it is the right identity element of

[9] It should be noted that in general, cost functions are not required to be continuous at their knots, though in the figure they are portrayed so for clarity's sake.

"$\leftrightarrow$", so that $l \leftrightarrow \square = l$. Now for layout expressions l, l_1, l_2 and r, and text string t let:

$$E(l) = E'(l, \square) \tag{18}$$

where:

$$E'(l_1 \leftrightarrow l_2, r) = E'(l_1, E'(l_2, r)) \tag{19}$$
$$E'(l_1 \updownarrow l_2, r) = E'(l_1, r) \updownarrow E'(l_2, r) \tag{20}$$
$$E'(l_1 \,?\, l_2, r) = E'(l_1, r) \,?\, E'(l_2, r) \tag{21}$$
$$E'(\text{`}t\text{'}, r) = \text{`}t\text{'} \leftrightarrow r. \tag{22}$$

For example, we have:

$$E((\text{`}a\text{'} \,?\, \text{`}b\text{'}) \leftrightarrow (\text{`}c\text{'} \,?\, \text{`}d\text{'})) = (\text{`}a\text{'} \leftrightarrow (\text{`}c\text{'} \,?\, \text{`}d\text{'})) \,?\, (\text{`}b\text{'} \leftrightarrow (\text{`}c\text{'} \,?\, \text{`}d\text{'})).$$

Note that as we require, in the expression on the right hand side of the above, the operands of the "?" operators contain all the layouts needed to make the choice of component, given a starting column.

But though the expanded form of a layout expression defines the same layout as the original expression, without special provision, the size of the expanded term may grow exponentially. Again, to illustrate, take an instance of the form in (1):

$$(\text{`}t_{11}\text{'} \,?\, \text{`}t_{12}\text{'}) \leftrightarrow (\text{`}t_{21}\text{'} \,?\, \text{`}t_{22}\text{'}) \leftrightarrow \ldots \leftrightarrow (\text{`}t_{n1}\text{'} \,?\, \text{`}t_{n2}\text{'}), \tag{23}$$

where $t_{11}, t_{21}, \ldots, t_{n1}, t_{n2}$ are text strings. While this expression contains only $3n - 1$ combinators,[10] is not too hard to show that its expanded form contains some $2^{n+2} - 5$ combinators. Thus if we were to derive the layout function of the layout expression in (23) by applying the function $C(\cdot)$ defined by equations (14 - 17) to the expanded term computed by $E(\cdot)$ in (19 - 22), performance would again become unacceptable, even for fairly small values of n.

This difficulty can be circumvented by performing expansion and translation of layout expressions simultaneously, "expanding out" the composition of $E(\cdot)$ and $C(\cdot)$ beforehand. To do this, we arrange for a distinguished "empty" layout function $\blacksquare$, requiring in addition that $C(\square) = \blacksquare$, and for any g_i, $g_i \langle\leftrightarrow\rangle \blacksquare = g_i$.[11] Next define a function $C'(\cdot, \cdot)$, such that for layouts l and r:

$$C'(l, C(r)) = C(E'(l, r)). \tag{24}$$

Finally, let $[\![l]\!]$, the layout function associated with the layout expression l, be $C'(l, \blacksquare)$.

[10] To be pedantic, combinator *instances*.

[11] The latter is most expediently arranged simply by adding a clause to the definition of "$\langle\leftrightarrow\rangle$"checking for the distinguished value "$\blacksquare$" in the arguments.

Now we can show that as required:

$$\begin{aligned}
[\![l]\!] &= C'(l, \blacksquare) && \text{definition of } [\![\cdot]\!], \\
&= C'(l, C(\square)) && \text{property of } \blacksquare, \\
&= C(E'(l, \square)) && \text{equation (24)}, \\
&= C(E(l)) && \text{equation (18)}.
\end{aligned}$$

Furthermore, expanding the right hand side of equation (24) using equations (19 - 22), we can derive a recursive (constructive) definition of $C'(\cdot, \cdot)$. From equations (24) and (19):

$$\begin{aligned}
C'(l_1 \leftrightarrow l_2, C(r)) &= C(E'(l_1 \leftrightarrow l_2, r)) \\
&= C(E'(l_1, E'(l_2, r))) \\
&= C'(l_1, C(E'(l_2, r))) \\
&= C'(l_1, C'(l_2, C(r))).
\end{aligned}$$

We can arrange to satisfy this equality by defining, for layout function g:

$$C'(l_1 \leftrightarrow l_2, g) = C'(l_1, C'(l_2, g)). \tag{25}$$

Similarly, we derive:

$$C'(l_1 \updownarrow l_2, g) = C'(l_1, g) \langle \updownarrow \rangle C'(l_2, g) \tag{26}$$

$$C'(l_1 \,?\, l_2, g) = C'(l_1, g) \langle ? \rangle C'(l_2, g) \tag{27}$$

$$C'(\text{`}t\text{'}, g) = \langle \text{`}t\text{'} \rangle \langle \leftrightarrow \rangle g. \tag{28}$$

It may appear that little has been gained by this exercise; after all, the equations defining $C'(\cdot, \cdot)$ mirror those for $E'(\cdot, \cdot)$ exactly. Note, however, that the argument g to $C'(\cdot, \cdot)$ is a layout *function*, not a layout *expression*, as are the objects on the right hand sides of equations (25 - 28). In particular, the two occurrences of g on the right hand sides of equations (26) and (27) refer to a *single* a layout function that is computed only once, while deriving the same layout function by application of $C(\cdot)$ to the expanded term from $E(\cdot)$ involves computing the same function *twice*.

It is the ability of a single layout function to characterize the optimum layout of a layout expression for any given starting column that facilitates this sharing. Figure 8 illustrates this point. In the figure, the layout function $\langle$`'e'`$\rangle\langle ? \rangle\langle$`'ff'`$\rangle$ appears twice in the expansion of the subexpression (`'c' ? 'dd'`)$\leftrightarrow$(`'e' ? 'ff'`), and the latter itself appears twice in the expansion of the entire expression. Depending on the choices between the text strings made in the two leftmost choice expressions, therefore, this single layout function must determine the optimal choice between the strings "`e`" and "`ff`" starting 2, 3 or 4 character widths[12] beyond the starting column of the entire expression. As Skiena (2008, chp. 3) notes, this capacity to

[12] Coresponding to choices "`a`", "`c`" (2 character widths), "`a`", "`dd`" or "`bb`", "`c`" (3 character widths) and "`bb`", "`dd`" (4 character widths) in the two leftmost choices.

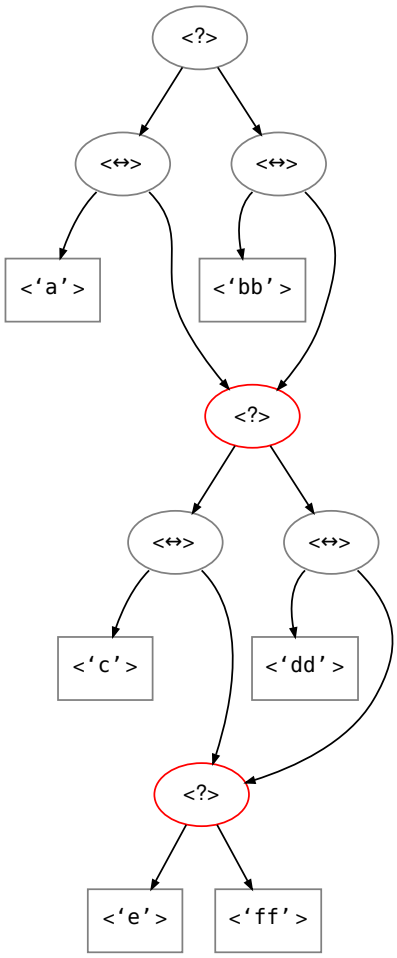

Figure 8: Sharing in the layout function calculation for (‘a’ ? ‘bb’) ↔(‘c’ ? ‘dd’) ↔(‘e’ ? ‘ff’).

identify and effectively address *overlapping subproblems* (in this case, the optimal layouts in subexpressions) is key to the effective application of dynamic programming.

6 Layout constructors in `rfmt`

As we pointed out in section 2, the `rfmt` program itself does not directly expose implementations of the primitive layout expression combinators. Instead, more convenient constructors—called *blocks*, with slight abuse of terminology—are provided, the implementations of which are composed from the primitive combinators. The `rfmt` blocks are described below.

6.1 Blocks and their implementations

TextBlock(*txt*) A layout consisting of a single line of unbroken text. This block is essentially a renaming of the "'.'" combinator, with the trivial implementation:

$$\mathsf{TextBlock}(txt) \triangleq \text{`}txt\text{'}$$

LineBlock($l_1, l_2, \ldots, l_n$) A layout consisting of the horizontal juxtaposition of layouts $l_1, \ldots, l_n$. The implementation of this block is simply a composition of the appropriate number of "↔" combinators:

$$\mathsf{LineBlock}(l_1, l_2, \ldots, l_n) \triangleq ((l_1 \leftrightarrow l_2) \leftrightarrow \ldots) \leftrightarrow l_n$$

StackBlock($l_1, l_2, \ldots, l_n$) A vertical stack comprising $l_1, \ldots, l_n$. Implemented by composition of "↕" combinators:

$$\mathsf{StackBlock}(l_1, l_2, \ldots, l_n) \triangleq ((l_1 \updownarrow l_2) \updownarrow \ldots) \updownarrow l_n$$

ChoiceBlock($l_1, \ldots, l_n$) A layout choosing one of $l_1, \ldots, l_n$, according to which layout has minimum cost on output. Again, we simply compose "?" combinators to implement this block:

$$\mathsf{ChoiceBlock}(l_1, l_2, \ldots, l_n) \triangleq ((l_1 \,?\, l_2) \,?\, \ldots) \,?\, l_n$$

IndentBlock(n, l) This block "indents" the layout l by n spaces. Its implementation juxtaposes a string of spaces of the requisite length (here denoted "*spaces*(n)") on the left of l:

$$\mathsf{IndentBlock}(b) \triangleq \text{`}spaces(n)\text{'} \leftrightarrow l$$

WrapBlock($l_1, l_2, l_3 \ldots, l_{n-1}, l_n$) The WrapBlock packs constituent layouts $l_1, l_2, \ldots, l_{n-1}, l_n$ horizontally, inserting line breaks between them so as to minimize the total cost of output, in a manner analogous to the composition of words in paragraph. (As with paragraphs, output after line breaks begins at the starting column of the entire WrapBlock.) When placed next to each other on the same line (i.e. where a line break does not intervene), the constituent layouts are separated by single spaces.

The implementation of this constructor is rather more involved than those above. Its expansion in terms of the primitive combinators is assembled by working from the final constituent layouts backwards, at each stage composing a choice whose alternatives entail placing increasing numbers of constituent layouts on the first line of the composite. For the sake of convenience, we use the blocks defined above, and the abbreviation "␣" for the layout

expression TextBlock(' ')—the text block containing a single space:

$$
\begin{aligned}
&\mathsf{WrapBlock}(l_1, l_2, l_3 \ldots, l_{n-1}, l_n) \triangleq \\
&\quad \textbf{let } b_n = l_n \textbf{ in} \\
&\qquad \textbf{let } b_{n-1} = \mathsf{ChoiceBlock}(l_{n-1} \updownarrow b_n, \\
&\qquad\qquad\qquad\qquad \mathsf{LineBlock}(l_{n-1}, \sqcup, l_n)) \textbf{ in} \\
&\qquad\quad \textbf{let } b_{n-2} = \mathsf{ChoiceBlock}(l_{n-2} \updownarrow b_{n-1}, \\
&\qquad\qquad\qquad\qquad \mathsf{LineBlock}(l_{n-2}, \sqcup, l_{n-1}) \updownarrow b_n, \\
&\qquad\qquad\qquad\qquad \mathsf{LineBlock}(l_{n-2}, \sqcup, l_{n-1}, \sqcup, l_n)) \textbf{ in} \\
&\qquad\quad \ldots \\
&\qquad\quad \mathsf{ChoiceBlock}(l_1 \updownarrow b_2, \\
&\qquad\qquad\qquad\qquad \mathsf{LineBlock}(l_1, \sqcup, l_2) \updownarrow b_3, \\
&\qquad\qquad\qquad\qquad \ldots, \\
&\qquad\qquad\qquad\qquad \mathsf{LineBlock}(l_1, \sqcup, l_2, \sqcup, \ldots, \sqcup, l_{n-1}) \updownarrow b_n, \\
&\qquad\qquad\qquad\qquad \mathsf{LineBlock}(l_1, \sqcup, l_2, \sqcup, \ldots, \sqcup, l_{n-1}, \sqcup, l_n))
\end{aligned}
$$

The layout functions corresponding to simpler blocks may be derived directly by application of the function defined in equations (25) – (28) of section 5. The WrapBlock, however, poses a further challenge, because if we expand out the definitions of $b_1, \ldots, b_n$ in the implementation above, the size of the resulting layout expression is exponential in n. We can address this problem by deriving the layout function corresponding to each b_i only once, reusing them in the derivation of subsequent layout functions, similar to the sharing of layout functions illustrated in figure 8.

To do this, we extend the definitions of $[\![.]\!]$ and C' to accommodate an addition argument, namely a tuple $\rho = (h_1, \ldots, h_n)$, whose elements are initially set (arbitrarily) to the empty layout function, $\blacksquare$:[13]

$$[\![l]\!] = C'(l, (\blacksquare, \ldots, \blacksquare), \blacksquare)$$

And trivially:

$$
\begin{aligned}
C'(l_1 \leftrightarrow l_2, \rho, g) &= C'(l_1, \rho, C'(l_2, \rho, g)) \\
C'(l_1 \updownarrow l_2, \rho, g) &= C'(l_1, \rho, g) \langle\updownarrow\rangle C'(l_2, \rho, g) \\
C'(l_1 \,?\, l_2, \rho, g) &= C'(l_1, \rho, g) \langle ? \rangle C'(l_2, \rho, g) \\
C'(\text{`}t\text{'}, \rho, g) &= \langle \text{`}t\text{'} \rangle \langle\leftrightarrow\rangle g.
\end{aligned}
$$

[13]The account of C' given here is somewhat simplified for pedagogical purposes—in the actual implementation of `rfmt` itself, sharing of layout functions is arranged by tagging both expressions and layout functions, and memoising the results of C'.

Rather than expanding out "**let** ... **in**" clauses prior to calculation of layout functions, the extended version of C' is applied to them directly. Definition of intermediate result b_i stores the corresponding layout function to be saved in ρ, from where it is retrieved each time the layout function of b_i is required:

$$C'(\textbf{let } b_i = e_1 \textbf{ in } e_2, \rho, g) = C'(e_2, \rho[i \mapsto C'(e_1, \rho, g)], g),$$
$$C'(b_i, (h_1, \ldots, h_i, \ldots, h_n), g) = h_i.$$

Here, the notation $\rho[i \mapsto h']$ denotes a tuple with its i^{th} element updated:

$$(h_1, \ldots, h_i, \ldots, h_n)[i \mapsto h'] = (h_1, \ldots, h', \ldots, h_n).$$

6.2 An example layout

In section 2, we introduced the primitive layout combinators with a selection of simple motivating examples. Here, we tackle a more realistic code layout problem, taking advantage of the relative convenience and clarity afforded by the blocks defined in the previous section.

Function calls of the form "$f(a_1, a_2, \ldots, a_m)$"—where f is a function name, and $a_1, \ldots, a_m$ are argument expressions—are found in most programming languages. A common way of formatting such calls is exemplified by the following:

```
FnName(argument1, argument2, argument3, argument4,
       argument5, argument6, argument7, argument8,
       argument9, argument10)
```

In this example, the right margin has been set at column 50, to highlight the effect of restricted output width. The function name appears on the first line, immediately followed by the arguments. Where it is necessary to insert line breaks so as to avoid breaching the right margin, arguments are wrapped to align with the initial character of the first argument. Finally, the closing parenthesis is placed immediately after the final argument.

To specify this formatting strategy using a layout expression, let us assume that we are given layout expressions $a_1, \ldots, a_m$ for each of the arguments, and that the string f names the function. Then the layout expression for the function call is given:

$$\begin{aligned}&\mathsf{LineBlock}(\mathsf{LineBlock}(\mathsf{TextBlock}(f), \mathsf{TextBlock}(\text{`('}))),\\ &\qquad \mathsf{WrapBlock}(a_1, \ldots, a_m),\\ &\qquad \mathsf{TextBlock}(\text{`)'})\end{aligned}$$

This layout is a LineBlock with three components:

1) Another LineBlock containing the name of the function and the opening parenthesis of the call,

2) A WrapBlock that packs successive arguments to the call into lines, wrapping where necessary, and

3) The call's closing parenthesis.

Note that since the WrapBlock containing the arguments is placed immediately to the right of the opening parenthesis, any wrapped arguments will also begin immediately to the right of the parenthesis (though on different lines).

The inclusion of a WrapBlock in the layout above enables it to adjust dynamically, according to the requirements of the context. For example, if in the example, we reduce the output width to 30 characters from the original 50, fewer arguments are placed on each line:

```
FnName(argument1, argument2,
       argument3, argument4,
       argument5, argument6,
       argument7, argument8,
       argument9, argument10)
```

Such adjustments are limited, however. For example, let us revert to a 50 character output width, and consider the following call expression, output using the some layout:

```
AVeryLongAndDescriptiveFunctionName(argument1,
                                    argument2,
                                    argument3,
                                    argument4,
                                    argument5,
                                    argument6,
                                    argument7,
                                    argument8,
                                    argument9,
                                    argument10)
```

It is clear that with a lengthy function name, a strategy that involves wrapping immediately after the opening parenthesis makes for rather unappealing output. In such circumstances, we might wish to use a different layout:

$$\begin{aligned}&\mathsf{StackBlock}(\mathsf{LineBlock}(\mathsf{TextBlock}(f), \mathsf{TextBlock}(\text{`('}))),\\ &\qquad\mathsf{IndentBlock}(4, \mathsf{WrapBlock}(a_1, \ldots, a_m)),\\ &\qquad\mathsf{TextBlock}(\text{`)'})\end{aligned}$$

In this layout, the arguments begin on the line after the function name an opening parenthesis, and are wrapped not to the column immediately to the right of the parenthesis, but at an indent of 4 characters from the beginning of the function name. In addition, the closing parenthesis appears on a line by itself, immediately below the beginning of the function name. Output of the second example is much improved with this layout:

```
AVeryLongAndDescriptiveFunctionName(
    argument1, argument2, argument3, argument4,
    argument5, argument6, argument7, argument8,
    argument9, argument10
)
```

Finally, to allow for short or long function names, we can use a ChoiceBlock to switch layout strategies as required:

ChoiceBlock(
  LineBlock(LineBlock(TextBlock(f), TextBlock('('))),
    WrapBlock($a_1, \ldots, a_m$),
    TextBlock(')'),
  StackBlock(LineBlock(TextBlock(f), TextBlock('('))),
    IndentBlock(4, WrapBlock($a_1, \ldots, a_m$)),
    TextBlock(')')).

Since for short function names the first alternative in this combined layout will occupy fewer lines than the second alternative (and thus incur a lower cost), it will be preferred by the ChoiceBlock. With increasingly long names, however, and the consequent wrapping of arguments over increasingly many lines, the ChoiceBlock is more likely to select the second alternative, in spite of the additional fixed cost its two mandatory line breaks involve.

7 Conclusion

This paper has given a detailed overview of the algorithms embodied in the `rfmt` source code formatter. Though `rfmt` joins a rich tradition of pretty printers and code formatters with several decades of history, we feel that it does make a unique contribution, in that it marries the convenience of the combinator-oriented approach widely employed for functional programming languages with the rigor of optimization-based formatters such as TEX and **clang-format**.

Furthermore, the layout function representation employed in `rfmt` affords great flexibility: A simple embellishment of the cost function for text strings described in section 4.1 allows `rfmt` to incorporate a "soft margin" like that described by Hughes (1995), which favors shorter lines over longer ones, without imposing mandatory line breaks.[14] Alternative cost functions may

[14] In detail, we have two margins, m_0 ("soft") and m_1 ("hard") with associated costs β_0 and β_1, where $m_0 \leq m_1$ and $\beta_0 \ll \beta_1$. The layout function in (8) is amended:

$$\left\{\begin{array}{c} 0 \mapsto (\text{`}txt\text{'}, s, 0, 0) \\ m_0 - s \mapsto (\text{`}txt\text{'}, s, 0, \beta_0) \\ m_1 - s \mapsto (\text{`}txt\text{'}, s, \beta_0(m_1 - m_0), \beta_0 + \beta_1) \end{array}\right\}.$$

also be accommodated—piecewise quadratic, for example, rather than piecewise linear—with minor alterations to the calculations in section 3. Currently, practical experience with `rfmt` has been limited to the R langugage, with the option of two layout styles. As it is applied to a wider range of languages and formatting styles, it should be possible to assess which—if any—such developments are most desirable.

All the other layout functions and associated calculations from section 3 remain unchanged.

References

R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957. Republished Dover, 2003.

O. Chitil. Pretty printing with lazy dequeues. *ACM Transactions on Programming Langages and Systems (TOPLAS)*, 27(1):163–184, January 2005. ISSN 0164-0925.

Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

I. Goldstein. Pretty printing: Converting list to linear structure. Technical report, Massachusetts Institute of Technology, February 1973.

R. W. Gosper. Employment, 2015. URL http://gosper.org/bill.html.

R. W. Harris. Keyboard standardization. *Western Union Technical Review*, 10(1):37–42, 1956.

John Hughes. The design of a pretty-printing library. In *First International Spring School on Advanced Functional Programming Techniques*, pages 53–96, London, UK, 1995. Springer-Verlag.

D. Jasper. `clang-format`: Automatic formatting for C++, 2013. URL http://llvm.org/devmtg/2013-04/jasper-slides.pdf.

M. O. Jokinen. A language-independent pretty printer. *Software Practice and Experience*, 19(9): 839–856, September 1989.

Alan Curtis Kay. *The Reactive Engine*. PhD thesis, University of Utah, 1969.

D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–185, 1960.

D. C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):456–483, October 1980.

R. S. Scowen, D. Allin, A. L. Hillman, and M. Shimell. SOAP—A program which documents and edits ALGOL 60 programs. *The Computer Journal*, 14(2):133–135, 1971.

Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, 2008.

M.G.J. van den Brand. Generation of language independent modular prettyprinters, 1993.

Philip Wadler. A prettier printer. *Journal of Functional Programming*, pages 223–244, 1999.

P. Yelland. `rfmt` – R source code formatter, 2015. URL https://github.com/google/rfmt.