The Future of Document Formatting

Jeffrey H. Kingston
jeff@cs.usyd.edu.au

Basser Department of Computer Science
The University of Sydney 2006
Australia

Abstract

Document formatting systems have reached a plateau. Although existing systems are being steadily enhanced, the next major step forward will require a union of the best features of batch formatters, interactive document editors, and page description languages. This paper draws on its author's twelve years of experience designing, implementing, and enhancing the Lout document formatting system to identify the remaining problems in document formatting and explore some possible solutions.

Keywords: document formatting, computerized typesetting

The Future of Document Formatting

Jeffrey H. Kingston
jeff@cs.usyd.edu.au

Basser Department of Computer Science
The University of Sydney 2006
Australia

1. Introduction

Document formatting is one of the most widespread applications of computers. Improvements in document formatting software and the hardware on which it is based have revolutionized the production of documents and enlarged our conception of what a document might be.

Any attempt at this point to define 'document' would run a risk of being overtaken by events; already documents commonly include hyperlinks, moving images, sound, and dynamic updating as their sources of information change in real time. It is perhaps safe to say that a document is information arranged for presentation to a person; the information may be called the *content*, and the arrangement its *layout*. Document formatting is essentially about mapping content to layout, although functions that do not exactly fit this definition, such as spelling and grammar checking, or even creation and editing of content, are often found in document formatting systems.

Document formatting systems fall into two camps. In one camp are the interactive document editors, ranging from word processing systems such as Microsoft Word [19] up to desktop publishing systems such as FrameMaker [2] and Interleaf [10]. These offer an editable screen image of the document layout. In the other camp are the batch formatters, such as troff [20], Scribe [22], TeX [16], and Lout [14], which process text files with embedded markup to produce non-editable layout. In this paper the above names will stand for the entire software family; TeX includes LaTeX [18], FrameMaker includes FrameMaker+SGML, and so on. Somewhere in between are the hypertext [8] net browsers, based on HTML, which are primitive batch formatters offering limited interactivity such as the ability to click on a hyperlink or fill in a form.

All of these systems are being actively enhanced by their developers, with new versions appearing regularly. For example, FrameMaker and Interleaf have responded to the World-Wide Web phenomenon by adding support for SGML [7] and HTML. Nevertheless, viewed from a wider perspective, they all appear to have reached a plateau, in the sense that each has fundamental limitations that are not likely to be overcome. For example, troff, TeX and Lout are batch formatters and are not likely to become interactive; FrameMaker and Interleaf are not as extendable as the batch formatters and, again, are not likely to become so.

One frequently hears arguments for or against these systems, but the truth is that none of them is ideal yet all have something to offer to the future of document formatting. What is needed now is a synthesis of the best features of all of these systems.

Papers which reflect on document formatting seem to be very rare. The survey paper by

Furuta, Scofield and Shaw [6] is still well worth reading; Kernighan [12] reflects on the troff family; this author has described the design and implementation of Lout [13]. But for the most part one has to infer principles from the systems themselves, and to look among the specialized applications such as music formatting [5], graph drawing [11, 23, 17], or non-European languages for requirements.

This paper draws on its author's twelve years of experience in designing, implementing, and enhancing the Lout document formatting system, plus his more limited experience of the systems mentioned above, to identify a set of requirements for a document formatting system that would be a significant advance on all current systems, and to explore their interactions.

2. Requirements

This section identifies the most significant requirements for a document formatting system. Efficiency in space will cease to be a requirement in the next few years. Efficiency in time is of course essential, as are other requirements that apply to any large software system, such as robustness, openness, and an interface that permits users of varying levels of expertise to work productively.

The other requirements are editability, extendability, generality, and optimality. Each of these requirements is discussed in turn in the sections that follow, together with problems that it presents either alone or in conjunction with previous requirements.

It is not possible to prove that this list of requirements is complete, but the author has carefully compared it against the features of most of the document formatting systems listed earlier. The only major omission has been the convenience features commonly found in interactive systems, such as spelling and grammar checkers, input and output in a variety of data formats, version control, and so on. These are valuable features, but they have little to do with document formatting in the core sense of mapping content to layout.

2.1. Editability

Editability, the ability to edit content while viewing layout, is the strong suit of word processing and desktop publishing systems. Fairly or not fairly, many users will not accept batch formatting. Also, the batch formatting edit-format-view cycle is too slow when the layout rule is 'what pleases the eye,' such as in diagrams, or when content must be altered to achieve good layout, for example in paragraphs containing long unbreakable inline equations.

Interactive interfaces also have an advantage when the logical structure does not follow a tree pattern. A good example is the editing of graphs (the combinatorial kind). Users of an interactive system can click on any pair of nodes to indicate that they are to be joined by an edge. In a batch system, because the structure is not tree-like, it is necessary for the user to invent names for the nodes and use the names when creating edges, which is considerably more error-prone. By contrast, equations do follow a tree pattern and so there is never any need to attach names to subexpressions.

Critics of interactive systems typically complain about the lack of content structure in interactive editors, and also about their weakness as editors compared with good text editors. Neither problem would seem to be inherent, and in fact recent versions of high-end document editors (FrameMaker+SGML for example) are addressing the content structure problem.

Openness to such auxiliary applications as free-text search and retrieval and creation of documents by computer programs requires that an archive format based on marked-up text be included in any interactive system. It only takes a little care to make such a format readable by humans. Thus an interactive system is automatically also a batch system.

2.2. Extendability

Extendability in a document formatting system means the easy addition of new features. It is the strong suit of batch formatters. For example, this author's Lout system has no built-in knowledge of equations, tables, or page layout (not even the concept of a page is built-in); these are all added by means of packages of definitions written in the Lout language, which is sufficiently high-level to make them fairly easy to produce.

Extendability implies some initial kernel of primitive features upon which the extensions are built. These would include horizontal and vertical stacking, rotation, and so on. The most interesting such feature is the mechanism for getting floating figures and footnotes into their places: diversions and traps in troff, floating insertions in TEX, galleys in Lout. There must also be ways of combining and packaging the primitives into features useful to the end user.

Although a system not built on such a kernel is conceivable, it seems scarcely possible to this author that such a system could supply all the features demanded by end users. The list is so vast – equations, tables, graphs, chemical molecules, music, and so on – that some kind of high-level kernel language seems essential to achieving them in any reasonable time and with any consistency, just as high-level programming languages are essential to large software projects.

Ideally, the kernel language would be defined formally, so that it would behave as predictably as a good modern programming language; lack of predictability is a fequently-heard complaint about existing systems. An example of a highly predictable layout system exists for the simpler problem of pretty-printing of tree-structured data such as computer programs [9].

Typography generates requirements for many features, such as hyphenation, spacing and kerning, ligatures, and so on. A document formatting system must produce good typography, because end users cannot be expected to do it themselves. Many of these features are dependent on the current language, and many English or European-oriented systems have failed to extend to the typography of languages outside that sphere. A good source of features needed in world-wide typography is Apple Computer's QuickDraw GX [3], although its approach of implementing the features in C is relatively non-extendable since it requires recompilation.

When an interactive system is extended with a new feature, it must be possible to continue editing in the vicinity of the new feature.

Ultimately, the layout of a document is a function of its content, so we may identify features with functions. In extreme cases, such as optimal layout, a function may take the entire document as its parameter; but usually it has small, clearly delimited parameters as in

$$\mathit{built\_up\_fraction}(\mathit{numerator}, \mathit{denominator})$$

There may also be implicit parameters inherited from the context, such as the current font size.

It is quite reasonable to insist that within any editing session the collection of features be immutable. Thus it is not essential to be able to edit the definition of any function while viewing any layout. In some cases, such as simple abbreviations, editing of definitions is quite simple and could easily be supported. But more complex functions, such as optimal layout or graph layout, are defined by computer programs and so are not amenable to editing in this way.

Similarly it is correct to insist that those parts of the layout originating within definitions be immutable. For example, the bar in a built-up fraction should not be editable. This does not preclude the addition of parameters to *built_up_fraction* to control the appearance of the bar if desired, but to allow the user to arbitrarily change the bar would produce a layout whose origin as a built-up fraction must be lost.

Thus, editability of features really only means editability of their parameters.

The most favourable case occurs when a function displays a parameter in a form similar to that which it would have taken if it had been entered outside the function. For example, *built_up_fraction* displays both its parameters, changing their appearance only slightly (by squeezing vertical spacing within them, and possibly changing the font size). The user can edit such a parameter as though it was not a parameter at all, and so (inductively) can edit parameters of parameters and so on without limit. This is essentially how equation editors work, and the Lilac system [4] has demonstrated it in an extendable framework, although using a kernel language too incomplete to support the full range of features required by users. A function may display a parameter more than once, in which case editing one display must change them all.

Preserving editability of displayed parameters is a difficult problem when the function is implemented externally to the document editing system. For example, if an external graph layout program [17] is employed, the result cannot be returned as a bitmap or PostScript file; rather a set of coordinate pairs or something similar is required so that the document formatter can place the nodes itself and hence understand where they ended up.

It has been suggested that a non-editable result is acceptable in such cases if a mouse click in the region it occupies signals the opening of a separate editor that does undertand what is going on in that region. This is the interactive equivalent of the preprocessor approach used by troff, and it has the same drawbacks of lack of consistency, duplication of features, and loss of generality (since even if every editor may invoke every other editor, the communication channels between them typically cannot convey such information as the current font, paragraph breaking style, available space, and so on). An architecture based on a single master editor with slave non-interactive formatting programs is preferable.

Parameters which are not displayed are a nightmare, and are responsible for much of what is ad-hoc in existing interactive systems. Two main approaches are in use. The first is the ‘style sheet’ or ‘dialogue box’ approach, in which the user who selects a feature with non-displayed parameters is presented with a box listing them and asked to supply values: a font name, a location to place a figure, a style of numbering, or whatever. This is the most general method, easily adapted for use in an extendable system. It works particularly well when the parameters have sensible default values, for then use of the box is optional, and when they have only a small range of possible values, for then the values may be displayed in a menu.

Second is the ‘inference’ method. Every parameter has some effect on layout, otherwise it would be useless. So the user is offered a means of manipulating layout, and the parameter’s value is inferred from it. For example, most editors permit an included graphic to be clipped by clicking on its boundary and moving the mouse; scaling and even rotation may be set by such means. Drawing programs allow nodes to be dragged about in the drawing area. ‘Master pages’ or ‘template pages,’ which allow the user to specify entire page layouts involving many parameters simultaneously, demonstrate the value of the inference method.

The great drawback of the inference method is that an inference interface has to be invented for every non-displayed parameter, and this is difficult in an extendable system. However, it should at least be possible to implement an inference interface for all suitable non-displayed parameters of kernel features, such as the *boundary* parameter of *clip*(), and in cases such as

define user_level_feature(…, *boundary*, …) =
    … *clip*(…, *boundary*, …) …

to propagate this interface upwards from kernel features to user level features. Then every user level feature that offers clipping as a parameter, for example, will do so in the same way.

2.3. Generality

By generality we will mean the absence of illogical restrictions on the use of features, either in the contexts in which they may be used, or in the values that may be assigned to their parameters. (These are formally the same thing, but the distinction is useful.)

Examples of illogical context restrictions are extremely common in document formatting systems. FrameMaker permits objects to be rotated in certain contexts (when they are table entries, for example) but not others. In troff it is very easy to include an equation within a table, but very much harder to include a table in an equation. Not all context restrictions are illogical, of course: a chapter should not begin within a figure, for example.

Lack of context generality takes a severe toll, because it means that implementation code, possibly highly sophisticated and with a great deal to offer, is locked into a few limited contexts. For example, most word processing systems now have interactive equation editors, but there seems to be no hope that their code can be re-used for such tasks as editing tree diagrams or diagrams of chemical molecules, despite the technical similarities among these tasks.

Examples of illogical domain restrictions are particularly common among geometrical functions. For example, LaTeX formerly produced lines only at certain fixed angles, and most systems only really understand rectangular shapes. The PostScript page description language [1] is far ahead of everything else in geometrical generality: in PostScript, arbitrary curves (even disconnected ones) made of lines, arcs, and Bezier curves may be drawn and filled, and arbitrary combinations of rotation, scaling and translation may be applied to arbitrarily complex fragments of documents lying within one page.

The abandonment of rectangles in favour of arbitrary shapes would have widespread beneficial effects if done in full generality. Text could fill arbitrary shapes and run around arbitrary graphics. Fonts could be defined (as they are in PostScript) as collections of arbitrary shapes, permitting kerning of arbitrary pairs of glyphs, not just glyphs of equal font and font size, thus solving the subscript kerning problem. Line spacing could reflect the true appearance of lines, not be crudely based on the highest ascender and lowest descender. Optimizations based on bounding boxes and caching should be able to solve the efficiency problems.

2.4. Optimality

By optimality is meant the ability to find the best possible layout for the given content. An optimal layout is not necessarily a good layout, because some documents have no good layout.

Optimal layout thus cannot remove the burden of rewriting content to achieve good layout, but in practice it does greatly reduce that burden, and this is why it is has been included.

The idea that layout could be optimal seems to be due to Knuth and Plass [15], who presented an algorithm for the optimal breaking of a paragraph into lines which is used in Knuth's TEX system. Research work was done on more general optimality as well [21], although this author is unsure how much of this work was incorporated into TEX.

Suitably generalized, their paragraph breaking algorithm is as follows. The first step is to deduce from the content a sequence of atomic formatting steps. For example, the content

The cat sat on the mat

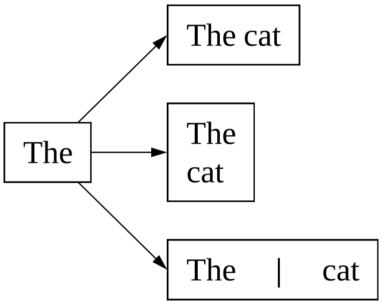
might have sequence

create_empty_paragraph
add_word_to_paragraph(*The*)
add_word_to_paragraph(*cat*)
. . .

Every prefix of this sequence should define a legal document in its own right; the whole sequence defines the document we wish to format. The question as to what constitutes an atomic operation is not of fundamental importance; one could choose to add one letter at a time, or an entire paragraph.

Define a *badness* function from layouts to integers. Small values indicate good layouts, large values indicate poor ones. There are no restrictions on how this function is defined, except the practical one of being computable in a reasonable time. Ideally, this function would be expressed entirely in the kernel language, and hence would be editable by the expert user.

Now there will be several ways in which each atomic step may be performed. For example, *add_word_to_paragraph* could add its word to the end of the current line, or it could start a new line, or it could even start a new page or column. This leads to a tree structure:



Each node is a layout of a partial document, each edge is one atomic operation.

The next atomic operation is applied to each leaf node, creating more partial documents, and so on until the sequence ends and the leaf nodes represent all layouts of the document of interest. The leaf node of minimum badness is the optimal layout.

This model can incorporate diverging operation sequences caused by layout dependencies. For example, suppose the word *abacus* has an index entry attached to it, and that along one path in the tree this word appears on page 99, while along another it appears on page 100. Then, in the sequence of operations defining the index, we will find

```
…
add_word_to_paragraph(abacus)
add_word_to_paragraph(99)
…
```

along one path, and

```
…
add_word_to_paragraph(abacus)
add_word_to_paragraph(100)
…
```

along the other. However, forward references create cyclic dependencies which cannot be handled in this way. For them, it seems to be necessary to add operations which change the value of words that have already been laid out, and to propagate the resulting changes until they die out. In rare cases this method will cycle forever, but in practice it is probably not difficult to avoid this problem using tricks such as refusing to allow a revision to reduce the number of lines allocated to a paragraph.

The algorithm as expressed has exponential time complexity. In practice, however, the number of different layouts of a document that are close enough to optimal to deserve examination is likely to be quite small. The challenge, then, is to find ways to prune the layout tree severely while retaining enough of it to discover, for example, that setting a sequence of paragraphs tight or loose will avoid a bad page break further on. This is an area needing detailed research; we can only glance at a few obvious possibilities here.

If the badness function is monotone increasing along every operation sequence, then a bad node can only have worse successors, and this justifies pruning its entire subtree. Monotonicity is not guaranteed (for example, adding one word to a paragraph which has a widow word will reduce its badness) but it is probable that tricks such as ignoring widow words in incomplete paragraphs can bring us near enough to monotonicity to justify pruning bad nodes.

One immediate application is to prune nodes whose layouts are obviously terrible, such as nodes containing clearly premature line endings or page endings. Indeed, it should be possible to avoid even generating such nodes.

When it can be established that two nodes are equivalent, in the sense that they lay out the same subsequence and their layouts occupy the same space, their future careers must be identical and the worst of the two may be pruned. The tree structure becomes a graph, and the optimal layout algorithm may be viewed as a shortest path algorithm, as described by Knuth [16].

Establishing the equivalence of two nodes may not be easy. There certainly is not time for complex comparisons of all pairs of layouts of a given subsequence. Knuth and Plass's algorithm recognises that two nodes are equivalent when they lay out the same subsequence and the most recent choice on the path to each was to start a new line. This same idea may be used to equivalence all paths into one at the new-page operation preceding a new chapter.

Another useful idea is to group operations together, find optimal layouts for the group separately, then introduce an atomic operation at a higher level which represents the entire group. Grouping the operations that define one paragraph in this way is very beneficial, for example. In isolation, optimal pragraph breaking explores many options, but in the end it is likely to return only at most two reasonable distinct results, of n and $n+1$ lines respectively for some n, and these become the only choices for the atomic *add_paragraph* operation that represents the whole group at the higher level. Furthermore, these two results may be cached and used without recalculation on every path containing that particular *add_paragraph* operation whenever the margins have the same width.

With care, suppressing tiny variations introduced by ascenders and descenders on letters, the layout tree might be induced to contain only as many paths as the difference in the total number of lines between the loosest and tightest settings of the paragraphs inserted so far, and over the course of one chapter this might be a manageable number. For safety, a fixed upper limit could be placed on the number of nodes kept, producing a beam search [24] which would definitely bound the time complexity to a fixed multiple of the cost of non-optimal layout, while sacrificing guaranteed optimality.

There do not seem to be any extra problems in incorporating optimality into an extendable system. Users would certainly welcome options to user-level features such as ‘insert this figure either following the current line, or at the top of the next page, whichever looks best.’ Whether an editable system can offer optimal layout without exceeding response time bounds is a matter for further research. There should be time to maintain optimality of the current paragraph at least, and if the current chapter is set within constant-width margins, it should be no more time-consuming to maintain optimal layout in a twenty page chapter than it is in a twenty line paragraph, provided the two alternative paragraph breaks of each non-current paragraph of the chapter are cached. If the cost does prove too great, optimality could be relegated to a button that the user can press just before going for coffee.

3. Conclusion

This paper has demonstrated that a next-generation document formatting system, incorporating the best features of current systems in full generality, is neither logically inconsistent nor likely to be infeasibily slow.

The major design problem is the identification of a suitable kernel of primitive features. Given the massive superstructure that this kernel will support, its design quality must be of the highest. This design was not attempted in this paper, but the author believes that the kernel of the Lout document formatting system would make a good starting point, although it is too incomplete, insufficiently general, too large, and occasionally too imprecisely defined to serve as the kernel of a next-generation system as it stands.

The major implementation problem is to find optimizations that preserve generality yet achieve the required response time. This paper has pointed out optimizations that seem quite likely to be adequate on hardware that will be widely available in a few years.

Acknowledgements. The author gratefully acknowledges comments on the first draft of this paper received from Mike Dowling, Ted Harding, Marnix Klooster, Robert Marsa, John Murdie, and Basile Starynkevitch.

References

[1] Adobe Systems, Inc. *PostScript Language Reference Manual, Second Edition.* Addison-Wesley, 1990.

[2] Adobe Systems, Inc. *Using FrameMaker+SGML.* Adobe Systems, Inc., 1995.

[3] Apple Computer, Inc. *Quickdraw GX*, 1996. URL http://support.info.apple.com/gx/gx.html.

[4] Kenneth P. Brooks. Lilac: a two-view document editor. *IEEE Computer*, 7–19 (1991).

[5] Eric Foxley. Music—a language for typesetting music scores. *Software—Practice and Experience* **17**, 485–502 (1987).

[6] Richard Furuta, Jeffrey Scofield, and Alan Shaw. Document formatting systems: survey, concepts, and issues. *Computing Surveys* **14**, 417–472 (1982).

[7] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[8] Charles F. Goldfarb. HyTime: a standard for structured hypermedia interchange. *IEEE Computer* **24**, 81–84 (1991).

[9] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, ed., *Advanced Functional Programming (Lecture Notes in Computer Science 925)*, pages 53–96. Springer-Verlag, 1995.

[10] Interleaf, Inc. *Interleaf 6 for Motif: next generation document creation, composition and assembly*, 1996. URL http://www.interleaf.com/i6motifds.html.

[11] Brian W. Kernighan. PIC – A language for typesetting graphics. *Software—Practice and Experience* **12**, 1–21 (1982).

[12] Brian W. Kernighan. The UNIX system document preparation tools: a retrospective. *AT&T Technical Journal* **68**, 5–20 (1989).

[13] Jeffrey H. Kingston. The design and implementation of the Lout document formatting language. *Software—Practice and Experience* **23**, 1001–1041 (1993).

[14] Jeffrey H. Kingston. *The Lout Document Formatting System (Version 3)*, 1995. URL ftp://ftp.cs.usyd.edu.au/jeff/lout/.

[15] D. E. Knuth and M. E. Plass. Breaking paragraphs into lines. *Software—Practice and Experience* **11**, 1119–1184 (1981).

[16] Donald E. Knuth. *The TEXBook*. Addison-Wesley, 1984.

[17] Balachander Krishnamurthy (ed.). *Practical Reusable UNIX Software.* John Wiley, 1995.

[18] Leslie Lamport. *LATEX User's Guide and Reference Manual.* Addison-Wesley. Second Edition, 1994.

[19] Microsoft, Inc. *Microsoft Word*. Microsoft, Inc., 1996. URL http://www.microsoft.com/msword/.

[20] Joseph F. Ossanna. Nroff/Troff User's Manual. Tech. Rep. 54 (1976), Bell Laboratories, Murray Hill, NJ 07974.

[21] Michael F. Plass. *Optimal pagination techniques for automatic typesetting systems.* Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA, 1981.

[22] Brian K. Reid. A High-Level Approach to Computer Document Production. In *Proceedings of the 7th Symposium on the Principles of Programming Languages (POPL), Las Vegas NV*, pages 24–31, 1980.

[23] Christopher J. Van Wyk. *A language for typesetting graphics.* Ph.D. thesis, Stanford University, Stanford, CA, 1980.

[24] P. H. Winston. *Artificial Intelligence.* Addison-Wesley. Third Edition, 1992.