Page Makeup by Postprocessing Text Formatter Output

Brian W. Kernighan and
Christopher J. Van Wyk
AT&T Bell Laboratories

ABSTRACT: Automatic text formatters usually produce poor page layouts: pages may have different lengths, widow lines abound, and figure placement is hard to control and often wrong. This paper describes a different approach to page makeup. We add to the output of a text formatter extra information that tells how various elements of the document should appear on the page. A postprocessor uses this information to make all pages the same height, prevent the creation of widow and orphan lines, place footnotes, and float figures to a suitable position on an appropriate page.

We have implemented such a postprocessor for text processed by the TROFF text formatter. The current version handles these page-makeup tasks for one- and two-column text. Versions of the program have been used to produce camera-ready copy for at least six books and several journal articles.

The authors typeset this paper using the software it describes.

1. Introduction

Text formatters construct pieces of a document in many ways. They may break running text into lines, compose tables and mathematical expressions, and include various kinds of figures. The last thing they must do with these pieces is select which are to go on each output page and lay these pages out neatly. *The Chicago Manual of Style* notes that page makeup ''is a highly skilled procedure. If galley material is simply divided mechanically into portions of equal length, without regard to where the divisions fall, some of the pages that result are bound to be unacceptable logically or aesthetically'' [1, p. 603].

Printers have a variety of criteria for what makes pages look good. Here are four of the most basic properties they generally seek [2]:

- There are no *widows*, single lines of text (usually the last line of a paragraph) at the top of a page, or *orphans*, section headings not followed by text at the bottom of a page.
- Pages are *justified*; every pair of facing pages is the same length, and all pages are about the same length.
- No page contains too much *white space*.
- Figures appear near the text that references them.

Although these properties by no means exhaust the set of desirable properties of printed pages, they illustrate an important computational point. Given any reasonable definition of excessive white space and proximity of figures to their textual references, one can construct examples of input for which these properties cannot all be achieved at the same time. Thus, a program that simply enforces a set of rules will frequently fail to find *any* satisfactory solution to a page makeup problem. The skilled page maker cautiously breaks rules, choosing which to violate by considering the sense of the document.

Since it is impossible to satisfy all of the principles of good page makeup simultaneously, one must understand the document to know how it should best appear on paper [3, 4]. Good page makeup cannot be defined in terms of purely numerical properties of the input [5]. This observation inspires two of the principles that have guided our work. First, use the simplest possible

methods to do the job, so long as they usually work; since we cannot solve the problem completely, each proposed complication must be justified by substantially improving the program's performance. Second, it should be relatively easy to predict how a small change to the input will change the page layouts produced, so that a user can tailor the pagination where necessary.

This paper describes how one can separate page makeup from the other jobs done by text formatters. It presents a general model of the page-makeup problem and a relatively straightforward algorithm. We have implemented the algorithm; versions of it have been used to produce camera-ready copy for at least six books and several journal articles.

Section 2 explains how others have addressed page makeup; our work builds on this experience. Sections 3 through 5 describe our model of the page-makeup problem and the algorithms we use to solve it. Section 6 sets forth some of the details of our implementation. Section 7 concludes with lessons learned.

2. *Previous Work*

In the printing industry, page makeup is often done by cutting galley proofs with scissors and pasting them onto large sheets of paper. This manual approach can produce excellent page makeup, but is expensive because it requires human labor. Manual cut-and-paste can be performed with the help of a computer [6], so it need not preclude automating such clerical tasks as producing the table of contents, the index, or the running page headers.

Early batch text formatters took only rudimentary steps to improve page makeup. PUB, for example, prevented widow lines by leaving a blank line at the bottom of each page, which was filled only to avoid a widow on the subsequent page [7]; of course, this meant that the output pages might not be justified. Even the widely distributed SCRIBE system does ''a barely adequate job of page makeup'' [8].

The standard macro packages for TROFF (*−ms*, *−mm*, *−me*) will not create orphan lines: a section heading will never be isolated at the bottom of a page. They do not prevent the creation of widow lines, however, nor do they perform any vertical justification. Fig-

ures that do not fit on the current page when they are encountered float to the top of a subsequent page. Some of these macro packages also support two-column formats, again without vertical justification or much control over figure placement.

Although more elaborate macro packages have been written for TROFF, none appears to have been widely used. For example, a package by M. E. Lesk does vertical justification; this has been used for a variety of journal styles. Enough text to fill a page is collected internally (''diverted,'' in TROFF parlance), then reprocessed with paragraph breaks and other paddable spaces expanded appropriately. This is intricate and slow, and does not address the more complicated task of mixing single- and double-column text with figures of different widths, which is characteristic of technical conference proceedings and some scientific journals.

The TEX text formatter was designed with a great deal of attention to page-makeup issues. It treats a document as a collection of *boxes* that contain text and *glue* that specifies white space that can be padded. Each gob of glue in TEX is specified by three values: its nominal value, its minimum shrinkability and its maximum stretchability; arithmetic on these values is performed in terms of the standard integers augmented by three non-standard orders of infinity. *Penalties* are also inserted to affect the makeup of output pages by marking good or bad places to break; TEX computes a layout that minimizes a function of these penalties. While the model of boxes, glue, and penalties permits one to specify a remarkable variety of page layouts [9], it can be extremely subtle: ''Glue will never shrink more than its stated shrinkability. . . . But glue is allowed to stretch arbitrarily far, whenever it has a positive stretch component'' [10, p. 70]. TEX considers somewhat more than a page of output before it breaks off and sets a page. This lookahead can be a hindrance in the final stages of producing a document, when one no longer wants small changes in the input to cause dramatic changes in the output [11].

The LATEX macro package for TEX standardizes many idioms common in TEX programs [12]. For it, as for TEX, the acceptability of widow lines is an adjustable parameter. The popularity of LATEX attests to its usefulness, but its solution to the page-makeup problem is not one of its strongest points; the manual notes several lim-

itations on its page-breaking commands. Section 7 compares the page-makeup facilities provided by TEX, LATEX, and our system.

The difficulty with macro packages for page makeup in TROFF and TEX is that it is simply too hard to write page-makeup programs of the necessary complexity in the clumsy and incomplete macro languages provided by these formatters. This paper describes an alternative approach, which allows much of the page-makeup program to be written more naturally in a conventional programming language.

There are two components to the solution: a macro package and a postprocessor. The macros cause the document formatter to include in its output extra information in addition to the typeset text it would normally produce. For example, the macros might mark the beginning of each paragraph, as well as larger blocks like tables, figures, and displayed equations. The output from the document formatter under this macro package is a sequence of pieces of typeset text and commands to the postprocessor. The postprocessor reads this output, computes the dimensions of the typeset pieces of text, then uses the commands to rearrange and lay out the typeset pieces on pages.

We have implemented our solution as a TROFF macro package and a TROFF postprocessor written in C++ [13], as depicted in Figure 1. The idea applies to any document formatter, however, so long as it can pass information untouched to its output, and that output can be read by a program.

The first postprocessor was a program called *pj*, for ``page justifier.'' It let TROFF find page breaks as it usually does under the *−ms* macro package. The macros produced commands in the output to mark spaces that could be padded if necessary. The postprocessor padded every page to the length of the first page. Several books were produced with *pj*, including

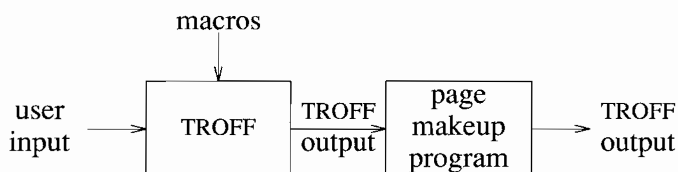

Figure 1: Page-makeup by a text-formatter postprocessor.

A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Mass. (1988).

J. L. Bentley, *More Programming Pearls: Confessions of a Coder*, Addison-Wesley, Reading, Mass. (1988).

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J. (1988). Second edition.

A. R. Koenig, *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass. (1989).

The program itself is about 500 lines of C++, much of it devoted to parsing TROFF output. It is fast: while TROFF requires 65 seconds on a DEC VAX 8550 to format the 22,000 words (49 pages) of Appendix A of the second edition of *The C Programming Language*, *pj* justifies the pages in only 4.5 seconds. Unfortunately, the simplicity of *pj* depends on the naive belief that TROFF finds good page breaks, and just needs a little help with justification (cf. [14]); *pj* really ignores most of the important problems of page makeup.

The next four sections describe PM (''page maker''), which addresses all of the page-makeup issues mentioned in Section 1.

3. Model of the Page-Makeup Problem

Our page-makeup algorithm and program are based on ideas from traditional printing. The input (that is, the output of the text formatter) is treated as a sequence of *slugs*. (The term in printing means ''line of type.'') Some of the slugs contain typeset material, while others contain instructions to guide the page-makeup algorithm. The program permutes the input slugs and partitions them into pages, then places them between fixed top and bottom margins on each output page.

The model expresses only geometric properties that slugs must obey as they are placed on the page. Its elements do not correspond directly to such logical pieces of a document as paragraph or footnote, though the macro package may well express those logical pieces in terms of elements of the model. For

instance, some groups of slugs are not allowed to split across a page boundary; tables and figures naturally fall into this category. Other groups of slugs may split so long as none of the resulting pieces is too small; for example, the first and last lines of a paragraph may not be split off from the rest of the paragraph. We shall describe how these examples fit into the model after we have presented the model itself.

The basic slugs are:

vbox A *vbox* contains printable material; it is the only kind of slug that produces visible output. It has a height that is determined by the extent and position of the text it contains. (The name *vbox*, for ''vertical box,'' is homage to the inspiration and instruction we have derived from TEX.)

sp An *sp* slug represents a paddable space; its parameter gives its nominal height. This height may be adjusted in two ways. When an *sp* slug is placed at the top or the bottom of an output page, or next to a slug of greater nominal height, the slug's height is set to zero. When a page is justified vertically, an *sp* slug whose height is positive may have its height increased.

These basic slugs belong to groups. Each group of slugs belongs to one of four possible types determined by its position on two axes. One axis determines whether the group can be split across a page boundary:

a *breakable* group may be split;
an *unbreakable* group must remain whole.

The other axis tells whether a group can float:

stream groups must appear in the output in the same order that they appear in the input;
float groups may move in the document.

Thus the four types are *breakable stream* (BS), *unbreakable stream* (US), *breakable float* (BF), and *unbreakable float* (UF).

Groups of three of these types can have parameters, whose role is defined by the following rules:

- A breakable stream has a single parameter k; if the group is split, at least k of the *vbox*es in the group must appear on each page where any do.
- A breakable float has a parameter set $\{v_i\}$, whose elements are desirable y-positions on the page for the center of the group. The group, or each piece if the group breaks across a page boundary, will float on the page so that its center is as close as possible to one of the values in $\{v_i\}$.
- An unbreakable float has a parameter set $\{v_i\}$, whose elements are desirable y-positions on the page for the center of the group. The group, which should not split across a page boundary, will float on the page so that its center is as close as possible to one of the values in $\{v_i\}$.

Groups may be nested within other groups. Indeed, it is common for a floating group to interrupt a breakable stream; its position in the input defines implicitly the place where the group belongs. Stream groups inside floating groups are also common. Slugs within floats or unbreakable streams do not float, however, either within or past the boundaries of their containing group.

The input may also include slugs and groups of the following kinds:

pt A "page title" group has a parameter k. The ith page title group is output at the beginning of page whose ordinal number is i; the logical page number is set to k. Only the *vbox* and *tm* slugs in a page title matter; any paddable space or grouping commands are ignored.

tm A "terminal message" slug has a string parameter s. When the *tm* slug is output, the string s is printed as part of the program's error log, prefixed by the current logical page number.

ne A "need" slug has a parameter h. When it is encountered during the processing of an unnested breakable stream, the page must be broken unless there is room on the current page for a *vbox* of height h. Once a need slug appears on a page, however, its height is zero.

The ith page printed has three page numbers. Its *ordinal* number is i, its position in the output sequence. Its *logical* number is the

value k defined by the ith *pt* group. Its *printed* number is the value that is produced by printing the *vbox* slugs in the ith *pt* group. It is the responsibility of the user to ensure that the logical and printed page numbers agree. They need have no relation to the ordinal page numbers.

Here ends the model. How does it apply to solve some basic problems?

A paragraph is an example of a breakable stream; when the parameter k is 2, the model forbids splitting off a single line from its end or its beginning, and thus prevents the creation of widow lines. A displayed equation or a table or a picture is an unbreakable stream. A footnote is a breakable float with a single parameter $v_1 = pageht$, which says that it should float to the bottom of the page. (The distance between the top and bottom margins is *pageht*; as is traditional in typesetting, the origin of the coordinate system is at the top left corner of the page and y increases down the page.) A numbered figure is an unbreakable float, which usually has two parameters, $v_1 = 0$ and $v_2 = pageht$, so that it will float to the top or bottom of the page. The rule that all output slugs must lie between the top and bottom margins means that such floats will bump into, but not cross, those margins.

Figure 2 shows how a paragraph in the input could be partitioned into a sequence of slugs and groups. It also illustrates how a macro package might generate adjacent *sp* slugs: the macros that make displayed equations, figures, and pictures each surround their contents with *sp* slugs; this causes the unbreakable float to start with two *sp* slugs. We naturally think of an unnested breakable stream as part of the *running text* of the document. As Figure 2 shows, the running text may be interrupted frequently by interpolations of various types.

Need slugs are ignored except when the running text is being processed. A need can be used to prevent orphans by forcing a page break when a page is too full for a section heading and two lines of following text.

Terminal message slugs are used to produce page number data for indexes and cross references. For instance, a *tm* slug might be associated with each figure to help produce a table of figures.

The *vbox*es in page title groups are placed on the output page in the same absolute positions they had on input. They may place

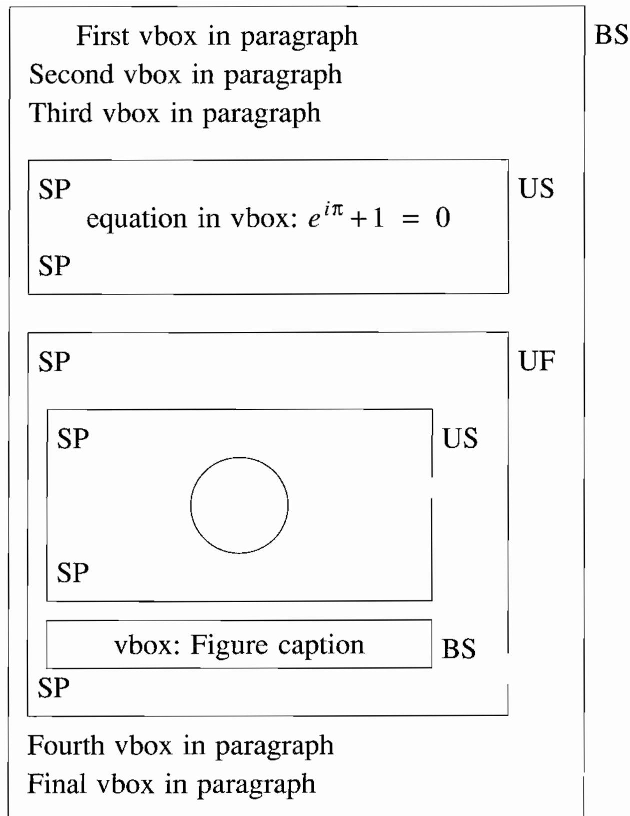

Figure 2: How a paragraph with a displayed equation and a floating figure might appear as a sequence of slugs and groups of slugs.

material anywhere on the page, and often contain text like running headers and footers. If there are more *pt* groups than output pages, the extra ones are ignored; if there are too few, the last one is re-used. Thus, the macro package can generate *pt* groups automatically, providing services like page numbering. When typesetting a book, one might want more elaborate headers that contain the name of the current section. This can be done in two passes through the document. During the first, strategically located *tm* slugs report the page numbers on which each section begins. An intermediate step uses these slugs to prepare *pt* groups that contain the running heads, and these groups are placed at the beginning of the input sequence. When the document is processed again, these *pt* groups are used to title the output pages. Both SCRIBE and LaTeX use

analogous two-pass procedures to produce bibliographic citations and cross references.

4. Single-column Layout

This section describes the algorithm PM uses to produce single-column output. The fundamentally greedy behavior of the algorithm is tempered by its adherence to the rules of the model. The slugs and groups on the page currently being filled are stored in list *currpage*. Algorithm 1 shows the outer loop. The details of step (4) depend on how one turns *vbox* slugs back into formatter output; Section 6 describes our implementation. Now we consider the other three steps, from simplest to hardest.

Step (3) takes as input a permutation of the slugs and groups on a page. It pads each *sp* slug of positive height proportionally so that the total height of the page is *pageht*. It does not justify the last page of a document, which is usually the desired behavior.

The input to step (2) is a sequence of slugs and groups. The output is a permutation of this sequence in which the centers of floating groups are as close as possible to a desirable *y*-position (a member of the parameter set $\{v_i\}$) and the heights of some *sp* slugs may have been set to zero. To give the greedy pagination the best possible chance to pack the page, the output of step (2) should be the permutation that yields the minimum total height of the resulting sequence. Short of trying all possible permutations, however, it is not obvious how to compute such an optimum permutation. PM uses the following approach:

(2) Each slug or group has a vertical position on the page, which is determined by the heights of its predecessors in the input

```
while slugs remain to be output
(1)     fill currpage with enough eligible slugs
(2)     compose currpage into a page
        if this is not the last page
(3)             justify currpage to height pageht
(4)     output the next pt group and currpage
```

Algorithm 1: Pagination

sequence. The vertical position of each slug and group is maintained throughout the execution of the following two steps:

($2a$) For each float on the page, choose v, the member of its parameter set $\{v_i\}$ that is closest to its current vertical position. Considering in turn each float on the page, move the float over its neighbors so long as this brings the center of the float closer to v and does not interchange two floats with the same v. Each float is considered only once.

($2b$) Proceeding through the slugs in order (including slugs nested inside groups) as they now appear from top to bottom on the page, set to zero the heights of *sp* slugs above the first *vbox* and below the last *vbox* on the page, and coalesce adjacent *sp* slugs so that only the one with maximum height survives.

In the most common case, the parameter set $\{v_i\}$ of an unbreakable float contains two values that correspond to the top and bottom of the page. Step ($2a$) chooses the closer of these possible locations as each float's goal, then moves each float as close as possible to its goal. The parameter set for a footnote will contain only one value, the bottom of the page. The rule on interchanges means that two floats whose goal is the same will not be permuted on the page.

Since this computation floats the groups on the page in one step and trims the paddable spaces in a separate step, the vertical positions it uses for the slugs and groups are only approximately their true vertical positions on the page as it ultimately will be printed. Although the approximation works well in practice, this algorithm certainly is not guaranteed to find the permutation that leads to the smallest total height.

Step (1) chooses the slugs that are input to step (2). It does its job greedily, filling the page while obeying the rules of the model, which does not specify how to fill pages as full as possible. We describe in top-down fashion how PM organizes the computation of step (1).

Each slug has a *serial number* that reflects its position in the input sequence. The serial number of a group is the smallest serial number of the slugs it contains. As a primitive data type, the algorithm uses *blocking priority queues*, which are priority queues of

slugs and groups ordered by serial number, augmented by a bit that tells whether the queue is blocked to prevent reading its head.

The algorithm maintains five blocking priority queues. Initially, all of the slugs and groups reside on queue *Input*. They pass through one of four queues, *BSqueue*, *USqueue*, *BFqueue*, and *UFqueue*, on their way to a page. The slugs and groups on the current page are stored on list *currpage* in the order in which they were added to the page. To simplify the presentation of the algorithm, we shall assume that these queues are maintained by a coroutine that ensures that slugs and groups are placed on queues as soon as they are eligible and that the following properties hold:

- Each of *USqueue*, *BFqueue*, and *UFqueue* includes slug groups of the type corresponding to its name.
- Whenever the head of *Input*, *BSqueue*, or *USqueue* has a smaller serial number than the head of *BFqueue* or *UFqueue*, the latter queue is blocked.
- At any given time, only one of *BSqueue* and *USqueue* is nonempty.
- Whenever either *BSqueue* or *USqueue* is blocked, *Input* is blocked as well.
- When *BSqueue* is nonempty, it contains the minimum possible number of slugs that one can add to the current page consistent with the stream parameter k.

BSqueue is a staging area for slugs from the running text. As an example of its application, consider a paragraph for which $k = 2$. Ordinarily, the process that maintains *BSqueue* will put just one *vbox* at a time onto *BSqueue*. At the start of a new page or of a new paragraph, however, it will add two *vbox*es to *BSqueue*. It will also ensure that the last two *vbox*es from the paragraph are placed onto *BSqueue* at the same time.

Algorithm 2 shows how PM performs step (1) of the Pagination Algorithm. When the contents of *BSqueue* do not fit, they must go back onto *Input* so that subsequent processing can treat them separately. For example, in the middle of a paragraph, it might be acceptable to add just one more line to the page. If that line does not fit, however, then one must add at least two lines from the paragraph to the next page.

```
unblock all queues
while there is a queue that is neither empty nor blocked
        while USqueue is neither empty nor blocked
                try to add its head
        while BFqueue is neither empty nor blocked
                try to add all or part of its head
        while UFqueue is neither empty nor blocked
                try to add its head
        try to add all of BSqueue
        if BSqueue did not fit
                empty BSqueue back onto Input
                block Input
```

Algorithm 2: Step (1) of Pagination

The key step in the Pagination Algorithm is the trial computation: what does it mean to *try to add* a slug, set of slugs, or group of slugs to the page, and how does one tell whether the trial succeeded? PM uses step (2) of the algorithm, shown in Algorithm 3.

Our statement of the pagination algorithm now omits only a discussion of how to decide where to break a breakable float. We postpone the point to discuss the experience that led us to some of the details of PM.

Placement of Floats. The position of a nested floating group within the running text gives the only clue to where the float belongs. Since there is no explicit tie between stream slugs and

```
remove the trial item from its queue and add it to
        the end of currpage
compose the slugs and groups of a copy of currpage
        using the step (2) computation
if the composed copy of currpage is taller than pageht
        remove the trial item from the end of currpage
        replace it on the queue from which it came
        block that queue
        advise caller that the trial failed
else
        advise caller that the trial succeeded
```

Algorithm 3: Try to add an item

floating groups, it is vital that floating groups be allowed to interrupt the running text; otherwise, floats would be forced to occur in the input sequence only between paragraphs, and a footnote or figure would be less likely to appear on the same page as its citation in the text. The queues are maintained so that a floating group appears on the relevant queue as soon as it is needed. Even so, there is no guarantee that a footnote will be printed on the same page as its citation.

The greedy strategy and use of queues prevent floats of the same type from being permuted across pages, and also prevent a float from appearing on an earlier page than a stream that precedes it in the input. Therefore a figure will never move from a right-hand page to the facing left-hand page.

From these limitations it is clear that the algorithm does not always place floats where one might want them to appear. One can, however, state its behavior concisely. Stream groups are never permuted, as required by the model. If i and j are the serial numbers of two groups, group j is a float, group i is either a stream or of the same floating type as group j, and $i < j$, then group j does not appear on an earlier page than group i; group j could, however, appear earlier than group *i on the same page*. Together with the greedy strategy, this rule means that the user can alter the output position of a float simply by moving it ahead or back a few lines in the input.

Queue Management. When a queue is blocked, the item at its head has failed once to fit on the current page, and no further attempts will be made to place the item on that page. While this simple strategy can cause the algorithm to miss a good page makeup, it prevents endless oscillation between trying to add the items at the heads of two different queues.

The order in which the queues are examined is important. To see why we check *UFqueue* before *BSqueue*, consider the fate of an unbreakable float of height *pageht* if *BSqueue* were checked first: at the top of each page, a slug from *BSqueue* would be placed on the page, preventing the tall float from appearing on that page; thus, the tall float, and all unbreakable floats that appeared after it in the input, would appear in the output only after all other queues were empty. We check *BFqueue* before *UFqueue* because we assume that the user wants the head of *BFqueue* to win any com-

petition between them; after all, the user is willing for that item to break in order that some of it may fit.

Exact and Approximate Trial Computation. The trial computation alters a copy of *currpage*, rather than the original. Thus, each trial computation begins with all *sp* slugs at their original heights, and the slugs and groups in the order in which they were originally added. This is the only way to ensure that previous trial computations do not affect the result of the current trial computation, since the computation in step (2) of the pagination algorithm depends on the order in which slugs and groups are presented.

As presented, the page-filling algorithm has complexity at least quadratic in the number of items on the page, since the page is composed every time an item is added, and the complexity of composition is surely superlinear. In general, the trial computation must be at least as "conservative" as the computation in step (2); that is, its estimate of the page height must never be lower. Otherwise, the trial could succeed in composing a page where the main loop would fail. We describe our experience with three conservative shortcuts to the trial computations.

One possibility is to use the total height of *currpage* as an estimate of the space left on the page, and to add the item if it would not cause the total height to exceed *pageht*. This is almost certain to cause pages to contain too much white space. For example, floating figures usually have space at top and bottom to separate them from the text; one of these spaces disappears when the figure is finally placed.

A second possibility is to use the total height of *currpage*, but to trim some of the space as the slugs are added to the page. For example, it is relatively easy to discard leading spaces and coalesce consecutive spaces on *currpage*. This approach leads to acceptable results; it was used to print

> R. E. Tarjan and C. J. Van Wyk, "An $O(n \log \log n)$-time algorithm for triangulating a simple polygon," *SIAM Journal on Computing* 17(1), pp. 143-178 (1988).
> C. J. Van Wyk, *Data Structures and C Programs*, Addison-Wesley, Reading, Mass. (1988).

The third approach is cleaner than trimming space on input. We use the total height of the items currently on *currpage* to esti-

mate the space left on the page. If it leaves room for the next item, then the model guarantees that we can add that item to the page; otherwise, we perform a complete trial computation. This method is usually fast, since the shortcut usually reports that a slug fits, and the expensive full computation is needed only for a few slugs at the end of each page. It also produces pages that are more tightly packed than the other two shortcuts. It was used to print

> R. Sethi, *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading, Mass. (1989).

The appealing shortcut of using the composed height of *currpage* to estimate whether to add the next item is not conservative. If a page has a stream *sp* slug at the bottom, that space is trimmed when the page is composed. Were a *vbox* added at the bottom, however, the resurrection of the height of that *sp* slug could result in a page that is taller than *pageht*.

Breaking Floats. Now we return to the lone omitted detail, how to break a breakable float. Our program breaks off only as much of the float as the height of *currpage* guarantees will fit on the page. The residue becomes a smaller breakable float whose $\{v_i\}$ parameter includes only the minimum value from its parent's $\{v_i\}$ parameter set. Should one need better packing, one could add pieces of the breakable float to the page group by group, performing a trial computation each time to decide when to quit. We have not found this to be necessary.

In principle, this completes the description of our input model and algorithm for single-column makeup. In practice, we need to provide for unbreakable groups that are taller than a page. There is no right thing to do—what does it mean to break something that is unbreakable?—but a program needs a sensible response that is less draconian than aborting all processing. We arrived at the following rule. If an unnested unbreakable group is taller than a page, then the group will be broken into page-sized pieces, and each will begin at the top of a page. If a *nested* unbreakable group is taller than a page, however, the program exits with an error message.

The first condition deals with the common case where an unbreakable stream like a program listing is longer than a page; the unbreakable stream will start at the top of a page and be broken

into page-sized chunks. The second condition handles less common situations that are more likely to be errors. For example, if an unbreakable float that is too tall is composed of a nested unbreakable stream (a figure) and a nested breakable stream (its caption), the program will allow the figure to appear on a different page from its caption; if the figure itself does not fit onto a page, however, the program will not try to break it.

The splitting of unbreakable streams is the final condition that determines the order in which the page-filling algorithm checks queues. The algorithm is stated in terms of two queues of stream items, *BSqueue* and *USqueue*, rather than a single stream queue, because of the possibility that an unbreakable stream might need to split. Thus, it checks the *USqueue* first in case it contains the second part of a broken unbreakable stream.

5. *Multi-column Layout*

It is much harder to lay out pages in multiple columns, especially in the presence of floating groups that span several columns. One of our principal goals has been to keep the mechanism for producing multiple columns separate from the rest of the page-makeup program as much as possible. This localizes the complications of multi-column makeup, both keeping the rest of the program clean and making it easier to experiment with different approaches.

We have restricted PM to just one- and two-column makeup. The term ''one-column material'' refers to *vbox*es that are to be set in a single column the full width of the page; the term ''two-column material'' refers to *vbox*es that are to be set in two columns each about half the width of the page. As each page is filled, it has a list of one-column material and another list of two-column material. Whenever we need to make up the page, the two-column material is formatted into a chunk containing two columns of approximately equal height; this chunk is then treated like an ordinary group in the single-column makeup problem.

Thus, ordinarily the input stream can mix one-column

figures with two-column text material; each page will contain a single chunk of two-column material. It is also possible, however, to freeze the current two-column chunk and establish a new, empty, list for subsequent two-column material. With this mechanism one can compose pages on which two-column material alternates with one-column material, such as wide equations, that spans both columns. This freezing operation is a PM command.

The height of each chunk of two-column material is determined as follows: compute the total height h of all slugs and groups, then put as many slugs and groups into the first column as possible without that column's height exceeding $h/2$ (obeying need slugs and widow-suppression parameters); perform step (2) of the pagination algorithm on the slugs and groups in each column; then justify the shorter column to the height of the taller.

Since this algorithm operates on untrimmed spaces, it tends to pack right columns fuller than left columns. The virtue of this approach is its simplicity: it uses a greedy algorithm to divide into columns, then uses subroutines that are already available to produce single-column layouts. It also generalizes readily to three or more columns. (A more sophisticated algorithm would find the shortest layout among all consistent allocations of the floats to each column. Such an algorithm would be much more expensive than the greedy approach for two-column makeup, and not obviously practical for three or more columns.)

When a page contains both one- and two-column material, the trial computation is very expensive because it uses step (2) three times—once on each column and once on the whole page. We know of no better shortcut than the third conservative one suggested in Section 4, however.

The following paper was produced by PM in two-column format:*

S. P. Morgan, ''Queueing Disciplines and Passive Congestion Control in Byte-Stream Networks,'' IEEE INFOCOM '89 (Ottawa, April 23-27, 1989).

*Notice the switch from two columns back to one.

6. *Implementation*

Section 3 describes the input to our page-makeup algorithm in terms of ideal slugs that contain text to be printed or other information to guide the makeup. The model tacitly assumes that each slug is readily identifiable, as are the dimensions, size, font, and contents of *vbox* slugs. It also assumes that each slug is independent, self-contained, and ''relocatable'': it can be moved relative to or separated from its neighbors in the input sequence of slugs, and placed anywhere on an output page. Finally, the algorithm as stated implicitly assumes that slugs should not overlap when they are printed on the page.

In reality, none of these idealizations is even remotely correct. The TROFF output language [15] represents in ASCII where to print characters in what size and font. (See Figure 3.) In general, TROFF does not produce redundant information. Thus, point size and type font appear in the output in only two places: when they change from a previous value, and at the top of each page. (The latter stipulation makes it easy to print only selected pages from TROFF output.) A sequence of horizontal or vertical position changes in TROFF input is compressed on output into single motions. All vertical positions appear in the output as absolute positions relative to the top of the page. Each output line starts at horizontal position zero, but its vertical position is not specified unless it has changed from the previous line. The end of each output line is marked, as is the break between each output word.

Since the gap between ideal and reality is large, the macro package or other TROFF input must insert at appropriate places in the output information about significant events, and must force TROFF to produce state information as often as possible. At the same time, PM itself must devote significant effort (about 25 percent of the code) to parsing TROFF output, extracting as much information as possible, and converting it into something closer to the idealized structures needed by the rest of the program. (Converting back to TROFF for output is trivial by comparison.)

To insert information into TROFF output, one uses a general ''escape'' mechanism: the input

\X'*text*'

```
sn     set size to n
fn     set font to n
cx     print ascii character x
Cxy    print character named xy
Hn     go to absolute horizontal position n
Vn     go to absolute vertical position n (down is positive)
hn     go n units horizontally (relative)
vn     go n units vertically (relative)
nnc    move right nn (exactly 2 digits), print character c
Dt ...\n     draw graphic of type t:
       Dl x y           line from here by x,y
       Dc d             circle of diameter d, leftmost point here
       De x y           ellipse of axes x,y, leftmost point here
       Da x y x1 y1     arc counter-clockwise,
                        center at here+x,y; end at center+x1,y1
       D~ x y x1 y1 ...    spline by x,y then by x1,y1 ...
nb a   end of line; b = space before line, a = after
w      paddable word space
pn     begin new page n -- set V to 0
#...\n       comment
x ...\n      device control functions:
       x i          initialize
       x T s        name of typesetting device is s
       x r n h v    resolution is n/inch, h = min horiz motion,
                                  v = min vert
       x p          pause
       x s          stop -- done for ever
       x t          generate trailer
       x f n s      font number n contains font named s
       x anything else uninterpreted
```

Figure 3: Summary of TROFF Output Language

causes *text* to be interpolated into the TROFF output as a "device control function," in the form

    x X *text*

The device control function X is not defined, so TROFF output processors can attach any desired meaning to it. (\X is approximately the same as \special in TEX.) As used by PM, the *text* line contains the name of a PM command and any parameters.

Macro Definitions. Consider the .SP macro, which requests paddable space. The invocation .SP 1, which requests one line of paddable space, expands into \X'SP *n*', which in turn becomes x X SP *n* in the TROFF output, where *n* is the height of one line in typesetter units. In a similar way, a .PP (paragraph) macro might generate

.PP (paragraph)	SP
	BS
First vbox in paragraph...	VBOX
Second vbox in paragraph...	VBOX
Third vbox in paragraph...	VBOX
.EQ (equation)	US
	SP
%e sup {i pi} + 1 ~=~ 0%	VBOX
.EN (end of eqn)	SP
	END US
.KF (floating keep)	UF
	SP
.PS (picture start)	US
	SP
circle	VBOX
.PE (picture end)	SP
	END US
Figure caption...	VBOX
.KE (end keep)	SP
	END UF
Fourth vbox in paragraph...	VBOX
Final vbox in paragraph...	VBOX

Figure 4. The paragraph from Figure 2 in the left column is decomposed into the hierarchy of slugs shown in the right column.

```
x X SP 50        inter-paragraph spacing of 50 units
x X BS 2         bs parameter 2 to avoid widows
```

Our macro package, *−mpm*, is compatible with *−ms*. In *−ms* and *−mpm*, the beginning of each paragraph is marked by a `.PP` command; this ends the previous paragraph. Both the beginning and end of larger blocks such as displayed equations, tables, and pictures are marked (by `.EQ`/`.EN`, `.TS`/`.TE`, and `.PS`/`.PE` pairs, respectively). The output of TROFF under *−mpm* includes PM commands that correspond to these macros. Each new *bs* command terminates the preceding *bs* group, while all of the larger blocks produce slug sequences bracketed by *us* and *end* slugs. The explicit *end* markers make it possible for groups to nest. Figure 4 illustrates how the short piece of text from Figure 2 might produce a sequence of slugs.

The macro package starts a new output page whenever a group—*bs*, *us*, *bf*, or *uf*—begins; it usually generates a *pt* group as

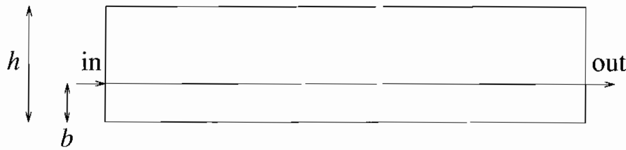

Figure 5: An ideal *vbox*.

well. This means that the TROFF output that is input to PM contains many more ''pages'' than the final document, and a surplus of page headers as well. We wrote the macros this way because the top of the page serves as a convenient fiducial mark against which to measure the heights of *vbox*es.

Converting Input Lines into Slugs. PM first parses its input into slugs, where each slug is either a line of text—a *vbox*—or one of the other commands like *sp* or *bs*. The ideal *vbox* (Figure 5) conceptually is a rectangle whose height h and base b are known. Printing begins on the left side at the baseline (for normal text lines like those of this paragraph, the baseline is zero). All motions are relative to the baseline so the *vbox* can be relocated, and nothing extends beyond the rectangle. It is typographically self-sufficient, so it can produce the right characters in the right sizes and fonts. After printing, the current position is at the baseline on the right side.

Given a sequence of such *vbox*es, each can be placed at the proper vertical position after the previous one, or placed on a new page at the proper distance from the top margin. If h_i and b_i are the height and base of *vbox* v_i, then v_i and v_{i+1} are placed so that the baseline of v_{i+1} is $b_i + h_{i+1} - b_{i+1}$ below the baseline of v_i.

The elements of each input *vbox* have vertical positions relative to the top of the page, either explicitly with a V*n* command or implicitly. To make a relocatable *vbox* slug, we compute the vertical position of the first printing object in the *vbox*, then convert any subsequent changes in vertical positions within the line of text to relative vertical motions. Similar conversions are made for horizontal positions, in case the slug is part of two-column material. We also compute Δv, the amount by which the baseline of this *vbox* lies below the bottom of the previous *vbox*.

If each *vbox* matches the ideal, this is easy. Unfortunately, since there is no guarantee that TROFF output will have any of the ideal properties, it is hard to determine the parameters of a *vbox*. Here are some of the complications that can arise.

A *vbox* may exit below its nominal baseline; for example, a line may end with a subscript in an equation. If this is not accounted for, the Δv computed for the next input line will not be correct. Related problems arise if the exit is above the baseline, or if the next line begins printing above or below its nominal baseline.

TROFF does not generally provide the height or base of a line of text, so PM must compute this explicitly, both for lines of text containing motions and for graphical objects (the `\D` commands). This means computing the high-water mark of text positions within the slug. Even that computation is suspect, however, since in some cases, the proper value is not the high-water mark. Sorting out these ambiguities was the most difficult part of the input processing.

In a picture or table, each graphical element—line segment, circle—appears as a separate *vbox*. Obviously, these *vbox*es may overlap. The height of a sequence of *vbox*es is not the sum of their heights, but the maximum vertical position they attain on the page. If the user does not enclose a sequence of overlapping *vbox*es in an unbreakable group, nonsensical page breaks may be introduced.

The order in which items appear in TROFF output is unpredictable and uncontrollable. A footnote, floating figure, or *tm* slug that appears between two words in the input may appear in the output either before or after the line containing the two words.

To summarize, the notion of output line is an imperfect match for the model notion of *vbox*. Moreover, because TROFF output does not contain explicitly much of the information we need to compute the dimensions of each line, the process of turning that output into a sequence of *vbox*es is subtle and painstaking.

Output. Fortunately, output is elementary compared to input. Each *vbox* carries with it the size and font that should be in force when it is ultimately printed. This information, along with the vertical position determined by page composition, is prefixed to the rest of the slug. When multiple column text is being produced, the slug is also offset horizontally.

Other Features. PM includes a few features that are not described in Sections 3 through 5. Users can insert two more commands to guide makeup decisions. One simply forces a page break. The other forces all queued floating material to be output before any more stream material; this is useful when one wants all figures associated with a section to appear before the next section starts. Users can also specify that pages should not be justified if they contain more than a certain amount of white space; by default, pages less than 90% full are not justified.

Implementation Details. TROFF is often criticized, with considerable justice, for its irregular, complicated, and constraining input syntax. Similar criticism might well be leveled at its output side. One of our ground rules was that we would not change TROFF, both to keep PM as portable as possible, and to avoid changing an old, complicated, and poorly documented program. Nevertheless, at times we were very tempted to modify TROFF to produce redundant information that PM could use.

We wrote the postprocessor in C++, originally to learn the language, and subsequently because it made programming easier than it would have been in C. The strong type-checking provided by C++ saved us from many programming errors. As the program evolved, the internal data structures changed naturally and smoothly from arrays to lists and priority queues. Derived classes and virtual functions made it easy to add new kinds of objects without having changes ripple throughout the program. The program uses many sanity checks to detect inconsistencies and errors as soon as possible. For example, the program checks that every input slug appears in the output exactly once, and that only one *BSqueue* and *USqueue* is nonempty at any time. This careful checking frequently helped us to find logical errors and misunderstandings about how slugs would pass from *Input* through the queues to *currpage*.

The −*mpm* macro package is somewhat shorter, and substantially simpler, than −*ms*. It does not use TROFF's trap mechanism, which invokes a macro when a page has accumulated a specified amount of text. It also does not use diversions to re-order output. Both of these mechanisms are difficult to control [16]. It is shorter than −*ms*, 1000 lines compared to 1700. Nevertheless, it might have been better to write another program to break TROFF input

into chunks, feed the chunks to TROFF in a constrained order, then produce output in an order suitable as PM input. So long as users refrain from constructing macro names by calls to other macros, such a program could be fairly naive about the syntax of TROFF.

The version of PM that does only single-column makeup is 1800 lines long; the complete version that also does two-column makeup is 2000 lines long. To lay out material in single-column pages imposes a running-time overhead of about 10% over simply running TROFF. The overhead for laying out material in double-column pages is closer to 30%, thanks to the superquadratic algorithm described in Section 5.

7. *Retrospective*

In 1978, D. E. Knuth noted that greedy algorithms and simple proportional space-padding do not lead to excellent typesetting [17]. On the other hand, TEX's complicated algebra of padding space and its dynamic programming algorithms do not solve the page-makeup problem either [18]. In fact, the manual itself notes that "if you are fussy about the appearance of pages, you can expect to do some rewriting of the manuscript until you achieve an appropriate balance, or you might need to fiddle ...; no automated system will be able to do this as well as you can" [10, p. 109]. TEX users have even asked us whether they could use our system to lay out TEX-generated slugs on pages.

PM offers features comparable to those available in TEX and LATEX. The basic TEX macro package forces the user to specify whether each floating group should appear at the top or the bottom of a page. The LATEX macro package lets users specify that either the top or the bottom of a page is an acceptable placement for a floating group. A PM user can specify several acceptable placements, and is not restricted to the tops and bottoms of pages: for example, one can specify that figures should appear in the middle of each page.

Both basic TEX and LATEX can produce two-column output. Neither, however, can alternate between one- and two-column processing, as PM can. Nor will either balance the two columns on the last page of a two-column document, as PM does.

Both TEX and LATEX let the user specify by a parameter how urgent it is that widow lines not be created. In contrast, when one uses the *–mpm* macro package, PM will never leave a single line of a multi-line paragraph on a page by itself; this not only avoids creating widow lines, but also prevents leaving the first line of a paragraph at the bottom of a page. We like the appearance of pages that obey this rule. On the other hand, the freedom of TEX and LATEX to create widows in the face of hard page-makeup choices means that they might produce better pages overall.

Although we completed work on the input model before we started to write the program, our work on the algorithms progressed largely through experience. There is no other good way to proceed, since it is easy to pose computationally intractable page-makeup problems [19].

Experience with the program also guided our choice of features *not* to implement. For example, we considered guaranteeing that facing pages would be the same height but permitting facing pairs to run a line shorter or longer than other pairs. But we have not yet seen a document for which this technique from traditional typesetting would improve the makeup, perhaps because the documents we have produced contain enough floating figures to sop up white space. As another example, we considered adding another parameter to paddable space to indicate that some spaces could grow more than others, but the page-filling algorithm of Section 4 usually packs pages tight enough that this is unnecessary.

The treatment of footnotes is one of the most obvious places that we have not improved. We noted in Section 4 that a footnote might not stay on the same page with its citation. It will only float off, however, when the citation is near the bottom of the page, a difficult situation for any page-makeup program. Our program treats footnotes the same way the first version of TEX did [20].

In fact, footnotes present many problems besides their placement. Some document styles call for footnotes to be separated from the body of text by a line. The closest our macro package can come to this behavior is to precede each footnote by a line, which is wrong when two footnotes appear on the same page; moreover, if the footnote splits, the second half will not have a separating line. The manual for the current version of TEX devotes ten pages to the processing of footnotes (TEX does cope correctly

with separating lines). To provide a similar facility in PM, we would have to clutter the general model with constructs specific to footnote processing.

Our solution fits well into the UNIX® system tradition of solving problems by breaking them into independent pieces. Of course, laying out pages independently of other formatting makes it hard to attain some desirable properties. For example, we cannot prevent a word from being hyphenated at a page boundary. The page-makeup problem seems hard enough, however, to justify treating it in a separate program.

It is tempting, perhaps especially for computer scientists, to seek the optimum solution to a problem. We believe, however, that it is much more important that the user be able to force a formatter to produce a desired makeup. In the early stages of a document, there is no need to seek the best layout. As a document nears completion, a user needs to be able to freeze the makeup of its early pages. Greedy algorithms offer this behavior almost free, and also make it easier to predict the layout after a small change in the input. When the formatter optimizes a function of the whole document, however, it is difficult or impossible to freeze the early pages.

We designed the input model before we started writing any programs, and it has not changed. The model is robust and simple; one of its most appealing features is that it defines the properties of text chunks in geometric terms (''breakable stream'') rather than in content-related terms (''paragraph''). Our paper design convinced us that we could express many of the important notions of page makeup in the model. An appealing alternative approach is to start with a simple model that accommodates running text and add features gradually [21]. Our experience suggests, however, that this approach quickly founders in the face of unexpected interactions between features; it is easy to suggest several heuristics, each reasonable when considered in isolation, that cause a page-makeup program to stall in a loop trying to choose among possibilities.

As it stands, PM commands have only a few parameters, and the main control the user has is to move figures in the input to change their position in the output. An attractive goal would be to provide users a general programming facility with which they could define the treatment of various situations. Unfortunately, we have no idea

how to make PM programmable without forcing users to read or duplicate much of the program.

Acknowledgements

We learned new things about page makeup from each of our early users. We are particularly grateful to Sam Morgan, Ravi Sethi, and Tom Szymanski for their perseverance and willingness to gamble their schedules on experimental software. We also thank Jon Bentley, John Hobby, Doug McIlroy, and Howard Trickey for valuable comments on this paper.

References

1. *The Chicago Manual of Style,* University of Chicago Press, Chicago (1982). Thirteenth edition.

2. *Words Into Type,* Prentice-Hall, Englewood Cliffs, N. J. (1974). Third edition.

3. R. Rubinstein, *Digital Typography: An Introduction to Type and Composition for Computer System Design,* Addison-Wesley, Reading, Mass. (1988).

4. J. W. Seybold, ''Document preparation systems and commercial typesetting,'' pp. 243-264 in *Document Preparation Systems*, ed. J. Nievergelt, G. Coray, J.-D. Nicoud, and A. C. Shaw, North-Holland, Amsterdam (1982).

5. H. L. Dreyfus, *What Computers Can't Do: The Limits of Artificial Intelligence,* Harper and Row, New York (1979). Revised edition.

6. Y. Sundblad, ''Computer based tools for skilled page make-up work,'' pp. 227-233 in *Protext I—Proceedings of the First International Conference on Text Processing Systems*, ed. J. J. H. Miller, Boole Press, Ltd., Dublin (1984).

7. R. Furuta, J. Scofield, and A. Shaw, ''Document formatting systems: survey, concepts, and issues,'' *Computing Surveys* **14**(3), pp. 417-472 (1982).

8. B. K. Reid, *Scribe: A Document Specification Language and its Compiler,* Carnegie-Mellon University (1980). Ph.D. dissertation.

9. M. F. Plass and D. E. Knuth, ''Breaking paragraphs into lines,'' *Software—Practice and Experience* **11**(11), pp. 1119-1184 (1981).

10. D. E. Knuth, *The TEXbook,* Addison-Wesley, Reading, Mass. (1986). Volume A of *Computers and Typesetting.*

11. D. F. Rogers, ''A page make-up challenge,'' *TUGboat* **9**(3), pp. 292-3 (1988).

12. L. Lamport, LATEX*: A Document Preparation System,* Addison-Wesley, Reading, Mass. (1986).

13. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading, Mass. (1986).

14. M. Luther, ''Justification by faith alone,'' *Die Funfundneunzig Thesen*, Church Door Publications, Wittenberg (1517).

15. B. W. Kernighan, ''A typesetter-independent TROFF,'' Bell Laboratories Comp. Sci. Tech. Rept. 97 (1981).

16. I. H. Witten, M. Bonham, and E. Strong, ''On the power of traps and diversions in a document preparation language,'' *Software—Practice and Experience* **12**, pp. 1119-1131 (1982).

17. D. E. Knuth, ''Mathematical typography,'' *Bulletin (New Series) of the American Mathematical Society* **1**(2), pp. 337-372 (1979).

18. F. H. Bartlett, ''Automatic page balancing macros wanted,'' *TUGboat* **9**(1), p. 83 (1988).

19. M. F. Plass, *Optimal Pagination Techniques for Automatic Typesetting Systems,* Stanford University (1981). Ph.D. dissertation.

20. D. E. Knuth, TEX *and* METAFONT*: New Directions in Typesetting,* Digital Press, Bedford, Mass. (1979).

21. A. W. Luniewski, ''Intent-based page modelling using blocks in the Quill document editor,'' *Document Manipulation and Typography—Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*, pp. 205-221, Cambridge University Press (1988).

[*submitted March 10, 1989; accepted April 6, 1989*]