ELECTRONIC PUBLISHING, VOL. 3(1), 3–28 (FEBRUARY 1990)

A function-based formatting model

BO STIG HANSEN

Department of Computer Science
Building 344
Technical University of Denmark
DK-2800 Lyngby, Denmark

SUMMARY

The work presented here concerns a document processing model accounting for aspects of an activity which is usually called formatting. The core of the model, an experimental formatting language called FFL, is the central topic.

FFL is a purely functional language in the style of FP and the applicative part of APL. Sequences, characters, and so-called boxes constitute the data types and among the build-in primitives are functions for aligning/spacing, breaking etc. Emphasis is put on presenting the language and exemplifying its use.

Also considered are issues in type checking of formatting function definitions and techniques for doing incremental formatting with FFL formatting functions.

FFL is currently being implemented by the BENEDICK project group led by the author.

KEY WORDS Text formatting Document processing models Functional programming Special-purpose languages

1 INTRODUCTION

An important stage in the development of a document preparation system is the design of a document processing model including a precise definition of the notion(s) of document and the kinds of processing considered. The creation of such a model may be a valuable mental design tool because it helps in identifying important design issues and provides a way of recording the design decisions made. If the model is sufficiently precise and sufficiently abstract, i.e., close to the concepts of the application domain and free from unnecessary implementation detail, it may in addition serve as the user's model of the system.

A classical example illustrating these points is the work by Knuth and Plass [1,2] on the boxes/glue model of document components and the model of paragraph breaking viewed as an optimization problem. The following informal—and incomplete—description of their models is intended for readers who are unfamiliar with the work:

> White space, e.g. between words, is modeled by so-called glue glops, each with a given normal length, a stretchability, and a shrinkability. In addition, each glue glop has an actual length which may differ from the normal length if the glop has been stretched or shrunk. Boxes are used to model rectangular, indivisible pieces of type such as a character from some font, a logo, or a

0894–3982/90/010003–26$13.00

Received 24 January 1989
Revised 20 November 1989

mathematical formula. A sequence of box and glue items may be *set* in a new box of a given length by adjusting the length of the sequence so it precisely fits the box. The length adjustment is achieved by stretching or shrinking the individual glue glops proportionally to their stretchability/shrinkability.

In connection with the adjustment of glue, a notion of work[1] may be defined. Without going into details, the idea is: (1) the less flexible a glue is, the more work is required to adjust it; and (2) the further a glue must be stretched/shrunk, the more work it takes to do it.

Modeling a paragraph as a sequence of boxes and glue glops, paragraph breaking can be viewed as a task of: (1) breaking the sequence into subsequences, and (2) setting each of the subsequences in a box of the desired line width. The details of where to break the sequence can then be specified indirectly by requiring that the total amount of work involved in doing the breaking must be minimal.

Usually, more control is wanted concerning where to end one line of a paragraph and begin another. For this purpose, a third kind of items, called penalty specifications, may be inserted in a sequence of boxes and glue glops. One may think of a penalty specification as reinforcing or weakening the spot where it occurs, thus affecting the amount of work necessary to divide the sequence at that point.

Among other things, the papers cited demonstrate how a model can provide a good basis for discussing questions about system functionality as seen from the user's point of view. That the model of paragraph breaking, moreover, is a valuable basis for understanding the complicated algorithm implemented is beyond question. Thirdly, it is most likely that the renowned algorithm would never have been developed, had it not been preceded by the invention of the model.

The trade-offs between functionality and efficiency of implementations may also be clarified by creating and analyzing document processing models. This is clearly illustrated in Plass's Ph.D. thesis [3] on optimal pagination techniques. In investigating abstract, mathematical models of page breaking viewed as an optimization problem, it is here shown how the choice of optimization measure seriously affects the computational complexity of the possible implementations.

The works referred to above demonstrate the versatility of clearly defined models for capturing fundamental design ideas and for documenting design decisions and their implications—both with respect to functionality and implementability. However, experience, e.g. with the ISO/ODA model [4,5], has also shown that obtaining sufficient clarity in document processing models is not always easy.

1.1 General modeling principles

One of the keys to the successful development of a model is the use of abstraction followed by stepwise refinement. By abstracting from unnecessary implementation detail, e.g. concerning the concrete representation of data types, one often gains clarity as in the case

[1] Knuth and Plass call it badness instead of work.

of the boxes/glue model. Here, the essential functional properties of a rather complicated paragraph breaking algorithm are easily described by use of suitable metaphors and a measure for modeling the quality of paragraph breaks (the notion of work mentioned above).

By viewing the paragraph breaking algorithm and the choice of concrete data structures as steps toward an implementation, an important separation of concerns is obtained. The abstract model describes *what* to do whereas the algorithm and data structures describe *how* to do it.

Restricting the attention to well-delimited subproblems of the systems design is another useful approach which is well exemplified by the boxes/glue model. Making one huge, monolithic model, accounting for all the functionality of a typesetting system like TEX, might be worth while but there is no doubt that smaller models focusing on limited design issues, such as the boxes/glue model, are far more comprehensible. (The boxes/glue model is implemented as a part of TEX [6].)

For general references to the use of abstraction and stepwise refinement in the modeling of software systems, see, for example, the work by Bjørner and Jones [7,8].

1.2 The problem domain

The work presented here concerns a document processing model accounting for aspects of an activity in document production which is usually called *formatting*. This and other basic notions are briefly reviewed in the following.

The production of a document can be viewed as a three-step process: editing followed by formatting and viewing as illustrated in Figure 1. The editing has the purpose of creating a *logical view* of the document by recording its logical structure, i.e. the hierarchical breakdown into *logical entities* such as chapters, sections, paragraphs, etc. with characters at the lowest level.

The formatting comprises *restructuring* activities such as the breaking of paragraphs into lines and *rendering* activities such as selecting character fonts and ensuring proper spacing and alignment. The resulting *physical view* of the document exhibits a structure almost similar to the logical one. Some *physical entities*, such as the ones named 'word' correspond to the logical entities with the same name, or fractions of these if they are broken. There are, however, also physical entities like 'line' which do not correspond to any logical entity. In addition to the structure, the physical view also includes the rendering information necessary to determine the exact physical appearance of the document.

With this distinction between the logical and physical view, the task of creating the logical structure and contents of a document can, to a large extent,[2] be separated from the task of designing the document's physical layout. Usually, the physical layout is then described, in general, for a whole class of documents by specifying how the different types of logical entities are to be formatted, i.e. how they are to be mapped to corresponding physical entities. Such general descriptions are usually called *document styles* or *generic documents*.

[2] In a discussion with the author, David M. Levy has pointed out that physical characteristics, such as the column width, may influence the writing style. In the context of narrow columns, shorter paragraphs may be desirable in order to limit the physical extent of the paragraphs.

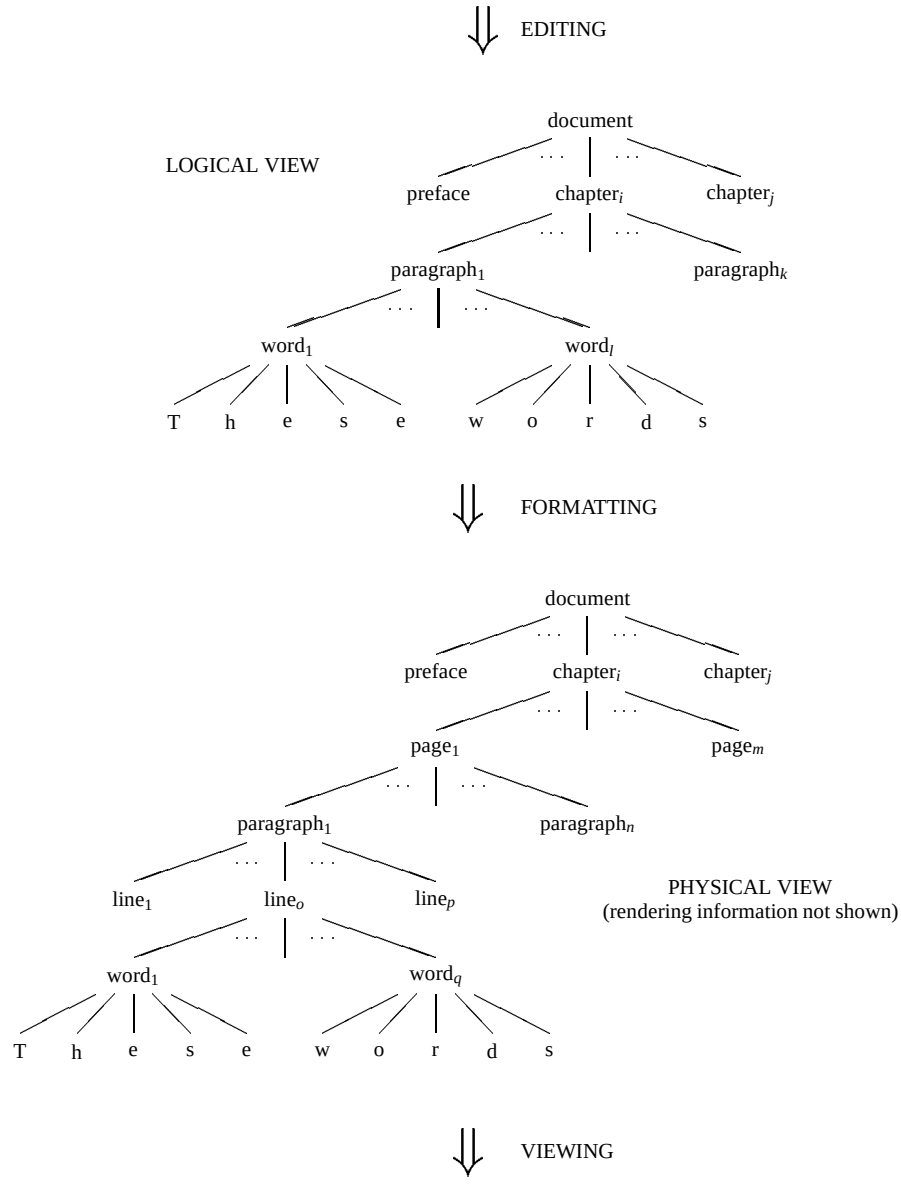


Figure 1. Different phases and document views in document production

Note that the logical and physical structures outlined in Figure 1 are just examples. Different classes of documents usually require different structures and different kinds of entities.

Based on the physical view, it is possible to derive other representations of the document's appearance such as a program in a page description language like PostScript [9], or a set of raster graphics pictures: one for each page. This changing of representation, which is called the viewing, allows the document to be printed on paper or displayed on a screen.

In integrated editing/formatting/viewing systems based on the WYSIWYG principle (What You See Is What You Get), the resulting document is presented on a screen just like it would appear on paper and the presentation is revised simultaneously with the editing. In such systems, the formatting and viewing must be very efficient in order to keep up with the editing. One of the objectives of the work presented here is to develop a processing model which allows integrated editing/formatting/viewing implementations with the necessary efficiency.

1.3 Model overview

Focusing on formatting, the question of how to model the logical and physical document views and the mapping from one to the other become the central topics. The approach taken here involves the definition of a *formatting language* which can be used by document style designers for defining how different classes of documents with different logical structures should be formatted.

One of the basic ideas of the approach is to model the logical view of a document by an *expression* in the formatting language. This is illustrated in Figure 2 where logical entity types, such as 'word', 'paragraph' etc., are viewed as function names. In the notation used, angle brackets enclose sequences and juxtaposition denotes function application.[3] As an example, the expression:

$$\text{word}\langle \text{T,h,e,s,e} \rangle$$

denotes the application of a function named 'word' to a sequence of 5 characters.

Given a set of function definitions, one for each kind of logical entity considered, the formatting is then modeled by the *evaluation* of the expression. The result of the evaluation should, therefore, be a *value* of some type appropriate for modeling the physical view of documents. It turns out that a notion of boxes, resembling the one introduced by Knuth and Plass can be used for that purpose.

Notice, that the above modeling of the logical view of documents is more permissive than necessary. Often it is possible and desirable to place restrictions on how the different kinds of logical entities may be embedded within each other. As discussed later, we (like many others) impose such restrictions by writing *grammars* for the different classes of documents considered. The logical view of a document in a given class must then conform with the grammar for that class.

Also notice that, with this basic model, the formatting of logical entities is *context independent*: for example, the formatting of a paragraph is if the same no matter whether it

[3] Parentheses may be inserted to resolve ambiguity or improve readability.

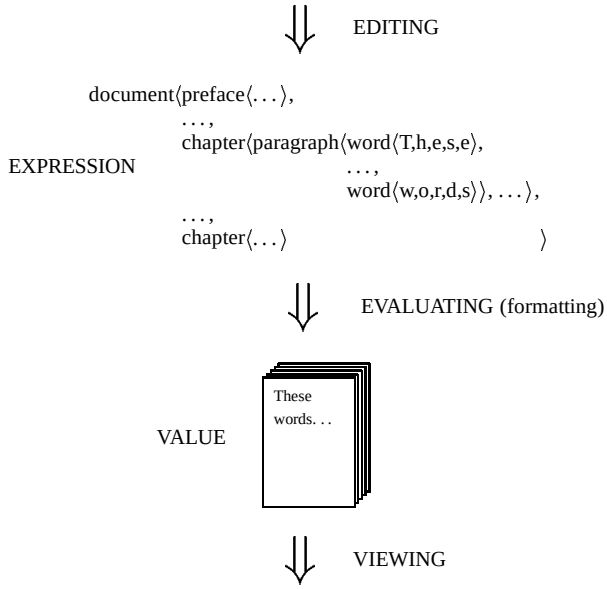


Figure 2. General view of a formatting model

occurs in a chapter, or perhaps in an abstract or a quotation. To allow for different formatting, different kinds of logical entities must be used, for example, paragraph-in-chapter and paragraph-in-abstract.

As discussed in work by Johnson and Beach [10], there are disadvantages to this requirement of context independence. Mainly, it does not support very well the introduction of changes to a document or changes to the way it should be formatted. For instance, wanting to move a paragraph, one may have to change it into a different kind of logical entity. Wanting to introduce a context-dependent difference in the formatting of a certain kind of entity, one must, in all contexts except one, change the entities of that kind into a new kind.

It turns out, however, that the handling of most context dependencies can be totally separated from the basic formatting. For example, one could envision a preformatting activity, changing paragraph entities into paragraph-in-X entities depending on the context. We will therefore in the following concentrate on the basic, context-independent formatting but our approach to handling context dependencies is briefly touched upon at the end of Section 5. Here it is outlined how the notion of *attributed grammars* may be combined with our basic formatting model in order to deal with context dependencies.

1.4 Contents overview

Following this introduction, some general design considerations regarding a new formatting language called FFL are presented. After this follows a section introducing and discussing a notion of boxes which constitutes the primary data type in the new language. Basic functions over boxes are then introduced and it is considered how to compose these in order to solve different formatting problems.

The subsequent sections deal with issues in the development of a document formatting system based on FFL. Among the problems considered is type checking of formatting function definitions in FFL, and incremental techniques[4] for implementing the language so it becomes efficient enough to be used with integrated editing/formatting/viewing systems.

The elements of FFL presented here are not sufficient for handling a number of interesting formatting problems, for instance the placement of floating figures and footnotes. Moreover, a straightforward, naive implementation will not give an acceptable performance. A number of language extensions and implementation techniques are therefore being considered by the BENEDICK project group; some of these are outlined in a section on further work.

2 GENERAL DESIGN CONSIDERATIONS

There are many important issues to be dealt with in the design of a document formatting language; in this section we will present some of the more general considerations which underlie the Functional Formatting Language FFL.

Distinguishing the logical view of a document from the physical view has become a well established principle which is supported in various degrees by a number of pure document formatting systems as well as integrated editing/formatting/viewing systems. See, for example, the collection of papers edited by André, Furuta and Quint [11]. For a survey of document formatting systems, concepts and issues, see the work by Furuta, Scofield and Shaw [12].

The use of such systems is often based on *predefined* document styles. However, especially in scientific and technical writing, it is the experience that in many cases the predefined styles must be modified before the desired physical layout can be obtained. The modifications typically involve redefinition of the way certain logical entities are formatted, but may also require introduction of new types of logical entities which are to be formatted in a specific way.

With current approaches, the frequent need for new or differently formatted entities presents a serious problem. It is typical, that either there are too many limitations on the kind of formatting which can be prescribed, or else the formatting prescriptions are very difficult to write and comprehend. Referring to the previously defined subdivision of the formatting activity into restructuring and rendering, it is typical that the formatting languages used with integrated editing/formatting/viewing systems are almost exclusively oriented towards rendering. On the other hand, the existing pure formatting systems are typically based on rather powerful, but low level, procedural formatting languages only suitable for specialist programmers.

Fundamental to the approach presented here is the hypothesis that a properly designed document formatting language can be a very useful tool—also for semi-professional users such as authors and secretaries who do not have document style design as a primary task. When a formatting language for these users is being designed, it is important that the language constructs are oriented towards the problem domain, i.e. document structure and layout, and at the same time the language should be powerful enough to express

[4] As with incremental compilers, it is not necessary to redo all the processing (formatting), after a change in the input; often it is possible to reuse parts of the previously obtained results.

both restructuring and rendering tasks. As a third requirement, it should be possible to implement the language in an integrated editing/formatting/viewing system.

Considering the intended users, it is also worth noticing that, typically, they are not computer professionals with a background in programming and debugging of run-time errors. It is therefore rather important that the formatting language is designed so *all* inconsistent uses are automatically detectable by a static analysis. In other words, having used the language to define a document style for a class of documents, it should be possible to type check the definition thoroughly enough to ensure that no run-time errors will occur during the actual formatting of a specific document.

In general, computer language designs are also influenced by technical limitations relating to efficiency of the implementations. This is also the case for formatting languages, especially those intended for use with integrated editing/formatting/viewing systems. It is the experience from existing systems that the expressive power of the formatting language must be restricted in some way in order to ensure the necessary efficiency of the formatting process. Presumably, there are many ways to impose such restrictions, e.g. by only supporting very limited restructuring capabilities like in many existing editing/formatting systems. With the *incremental formatting technique* considered in this work (Section 6), the restrictions find expression in the requirement that the mapping from the logical to the physical document view must always be *invertible*.

As will appear from the following sections, the requirement of invertibility has influenced the language design very much. In order to ensure invertibility, the language is based on special-purpose primitives with restricted expressive power. As an implication of this design decision, the language is not complete in the sense of being able to describe all possible mappings from the logical to the physical document view. It is, however, our belief that the language with suitable, special-purpose extensions will be sufficient for handling most document formatting problems.

3 BOXES

A characteristic of our formatting model is its foundation on language theoretic notions (see Figure 2 on page 8). The core of the model, a formatting language named FFL, must be defined so its expressions can represent the logical view of documents and its values are adequate for representing the physical view. Considering first the values, these should be expressive enough that the precise placement of all characters in a document can be described.

If this was the only requirement, values could be sets of coordinate/character pairs. With 3-dimensional coordinates, one of the dimensions could be used for distinguishing between pages. However, wanting to support typographical problem solving, such a model based on absolute positions might not be the best choice. *Relative* properties, such as alignment of document components and space between components are better suited for this purpose because they focus on the *intention* of the typographer, not the details of how this intention is realized.

The same argument may be repeated when considering the modeling of space: a notion of white space with fixed sizes is sufficient for describing the results, but it cannot be used for describing intentions regarding relationships between the sizes of different spaces.

Consider, for example, the justification of lines in a paragraph where it is the intention that the size of the spaces on each line should be adjusted to make all lines have the same width. Moreover, the spaces between words on the same line should have equal size, whereas the spaces between sentences should be larger. Here it is not convenient to define the space sizes explicitly since they depend on, for example, the sizes of the words and the desired line width.

As described in the introduction, a notion of glue glops is used for modeling white space in the boxes/glue model by Knuth and Plass. In fact, this notion of glue, with its stretchability and shrinkability properties, allows for specification of the size relationships discussed above; this is accomplished by simply letting the glue between sentences be wider and more stretchable than the glue between words.

The values used in FFL for representing the physical view of documents are called FFL boxes or just boxes. In some aspects, they resemble Knuth's box concept but there are also significant differences, some of which will be discussed later. In our definition, a box embeds a sequence of components which are either characters or other boxes. Moreover, a box has a number of primary properties:

Direction An indication of the dimension in which the components should be aligned: horizontal, vertical or depth; and orientation: positive or negative. The positive orientation in each dimension is defined in Figure 3, which illustrates how boxes representing pages could be aligned in the depth dimension to form a box representing the whole document. The upper, leftmost, foremost corner of a box or character is called its *origin* whereas the diametrically opposite corner is called its *extreme*.

Gluing A description of spaces used when composing the box components in the given direction. Three kinds of spacing may be described: one to be used before the first box component, another to be used after the last component, and a third to be used between neighboring components. The spaces are described using the notion of glue introduced above, with a normal length, a stretchability, a shrinkability, and an actual length. Omitting the description of one of the three kinds of spacing means that non-adjustable glue with length zero should be applied at the place in question.

Reference point descriptions One or more descriptions of named, user-defined reference points for the box. The descriptions may refer to the reference points, origin and extreme of the first and last component. In addition, the extreme and the already specified reference points of the box itself may be used. All the points in question are measured relative to the origin of the box itself; when viewed as vectors, they

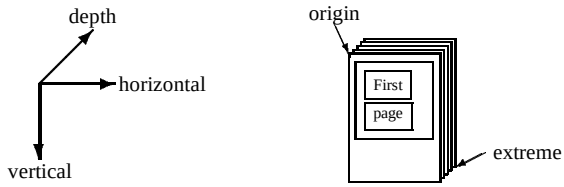


Figure 3. Composition of boxes representing pages

may be combined by vector addition, scalar multiplication, and vertical, horizontal, and depth projection.

Alignment A specification of how adjacent box components should be aligned when placed in the given direction with the given spacing. In general, it is possible to specify that each pair of adjacent components c_1 and c_2 should be aligned with respect to a given c_1 reference point and a given c_2 reference point.

In the following section, functions for creating boxes are considered and in this connection a syntax for expressing the above-mentioned box properties is introduced.

An example of the use of two reference points for each box is presented in Figure 4 where the two kinds of reference points are visualized as ► and ◄ (coinciding points are shown as ◄►). The box shown represents an entry in a table of contents, the three components constituting respectively the chapter number, title, and page number. In this example, the title occupies two lines; they are shown as dashed boxes and their reference points are not shown but the two reference points of the box containing the whole title are situated on the base lines of the two parts, respectively. Now, by specifying the alignment to be such that ► of each component is aligned with ◄ of the following component, the chapter number will be aligned with the first title line and the page number with the last.

Characters are assumed to have a reference point called 'ref' centered on the baseline. During the creation of boxes, their reference points must be defined so they can be used later to obtain the desired alignment.

Besides the above-mentioned primary properties, a box has two derived properties: its size and appearance. The size of a box is the size of its aligned and spaced sequence of components; similarly for the appearance. In the following, we shall use a number of different terms when discussing sizes: length is used for the size in an unspecified dimension; width, height, and depth are used for the size in the horizontal, vertical, and depth dimension respectively.

One of the differences between Knuth's box notion and the one used here regards the rôle of boxes in the formatting activity. In Knuth's model, a box is an unbreakable unit which, from a formatting point of view, only can be used as a building block in the construction of other boxes. As discussed in the following section on formatting functions, we consider boxes in general to be breakable; it will therefore in all practical applications be necessary to have further properties assigned to boxes, describing the more detailed rules (e.g. hyphenation rules) and preferences regarding their possible breaking. Such attributes can be defined but we will not go into the details here.

Other notable differences concern the dimensions and reference points. With multiple reference points, the combined top/bottom alignment illustrated in Figure 4 is easily

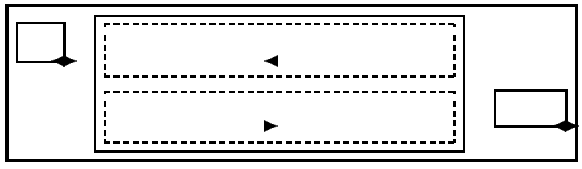


Figure 4. Alignment of three box components

specified. Knuth's boxes have only one reference point so he cannot obtain the desired alignment of the three boxes in the same straightforward manner.

Regarding dimensions, we use the depth dimension for placing the different pages of a document, whereas Knuth uses a sequence of two-dimensional boxes, each representing a page. The space of pages *is* discrete so using sequences may look like the natural choice; there are, however, formatting problems where alignment in the depth dimension is of major importance, for example, in connection with the preparation of overhead transparencies to be overlaid. Therefore, we propose the three-dimensional box model, however with the requirement that the three spacing descriptions are omitted when boxes are laid out in the depth dimension.

4 FORMATTING FUNCTIONS

In this section, we will consider a number of formatting primitives and some ways of combining these in order to define the formatting of different kinds of logical entities.

It is today widely accepted among programming language researchers that a clear, compositional semantics[5] is instrumental in understanding and using a language. One way of achieving this is by strictly adhering to a pure, applicative style, refraining from procedurality and side-effects. Taking this functional approach, it turns out convenient to adapt a combinator style similar to programming languages such as FP [13] and (the non-procedural part of) APL [14]. However, in contrast to the high number of combinators found in these languages, we will only consider two, one of which is the ordinary function composition.

4.1 Laying out

Given a sequence of characters or boxes representing, for example, words, lines, or whole pages, a sensible operation would be to embed these components in a box, thereby: (1) fixing the direction in which the components should be aligned; (2) determining which reference points should be used when aligning the components; and (3) optionally specifying the glue to be used before the first, after the last, and between all the components. For this purpose, the **Layout** family of functions is introduced. Besides the direction, alignment, and glue specifications, each **Layout** function is characterized by a specification of the user-defined reference points of the resulting box. As an example, consider the following definition of a function for formatting words:

```
word = Layout[dimension = horizontal
              orientation = positive
              alignment = ref with ref
              glue: between = inter-char-glue
              reference: ref = ref(first)        ]
```

Using this function, the characters of a word will be aligned horizontally by their 'ref' reference points with a certain kind of glue in between. In many cases, it will be appropriate to have non-adjustable, zero width glue at this place but in special documents

[5] With such a semantics, the meaning of a composite expression is an easily understood composition of the meanings of the expression components.

such as newspapers some stretchability might be desired. We abstract from this choice by introducing the symbolic constant *inter-char-glue* which at this point is left unspecified.

The reference point 'ref' of the word is in this example coinciding with that of the first character. Alternatively, it could have been set to the horizontal center on the baseline of the word, which may be expressed as:

$$\text{vertical}(\text{ref}(\text{first})) + 0.5\,\text{horizontal}(\text{extreme})$$

Note that a vertical projection yields a vector with the two other dimensions set to zero and vice versa for the horizontal and depth projections. The above expression is therefore a sum of vectors—not a sum of scalars.

Reconsidering the primary properties of boxes, the reader may have wondered why reference point *descriptions* were included rather than just reference points. The motive for this is to retain the intentional descriptions, i.e. the expressions defining the reference points, rather than the vectors which they denote. By saving the expressions together with the box they may later be reused, for example, for defining the reference points of the box after it has been stretched or shrunk.

4.2 Breaking and adjusting boxes

Having laid out words as specified above, let us consider larger units such as sentence fragments (e.g. separated by commas), sentences (e.g. ended by full stops), and paragraphs. These may all be laid out in a similar manner, but with increasing amounts of space between their components. The result of doing this will be a rather wide box which should be broken into appropriately sized pieces representing lines.

For the purpose of handling this and similar situations, a family of **Break** functions is introduced. Given a desired length, the functions use Knuth's optimization technique for determining exactly where to break a box. The result is a sequence of boxes, each having *approximately* the desired length. The lengths may vary because the **Break** functions do not perform any adjustment of glue glops; they only do the breaking so a later adjustment to the desired length will lead to the optimal solution.

The following example illustrates how **Layout** and **Break** functions may be composed to define a simple paragraph formatting function:

```
paragraph = Layout[dimension = horizontal
                   orientation = positive
                   alignment = ref with ref
                   glue: before     = par-indent
                         between = inter-word-glue
                   reference: ref = vertical(ref(first))] ∘
            Break[length = line-width] ∘
            Layout[dimension = vertical
                   orientation = positive
                   alignment = ref with ref
                   glue: between = inter-line-glue
                   reference: topref = ref(first)
                              botref = ref(last)      ]
```

Notice that we define function composition so it can be read from left to right:

$$(f \circ g)x = g(f(x))$$

The function 'paragraph' therefore first lays out the paragraph components (say words) horizontally with *par-indent* glue before the first word, then breaks the result into a sequence of lines, and finally lays out these lines vertically.

When a box is broken, its reference point descriptions are inherited by each of the boxes in the resulting sequence. In this example, the reference point 'ref' of the unbroken box is placed at the left edge, on the baseline; this is due to the vertical projection applied in the reference points' description. Each of the lines resulting from the breaking inherits this reference point description and thus has a reference point 'ref' placed at the left edge, on the baseline.

Therefore, when the lines are laid out vertically with the reference point of each line being aligned with that of the following, they will be left-justified (flushed left). If they should be flushed right instead, this could be obtained by placing the reference points of the unbroken box at the right edge:

$$\text{vertical}(\text{ref}(\text{first})) + \text{horizontal}(\text{extreme})$$

Notice that the resulting paragraph has two user-defined reference points situated at the baseline of the first, respectively the last, line in the paragraph.

In a simplified setting without floating figures, footnotes, etc., the family of **Break** functions may also be used for column and page breaking:

document = **Layout**[dimension = vertical . . .] ∘
Break[length = *column-height*] ∘
Layout[dimension = horizontal . . .] ∘
Break[length = *text-width*] ∘
Layout[dimension = depth . . .]

Note that, in this and the following examples, the focus is not so much on the details of the **Layout** functions so they will be left partially unspecified as indicated by the ellipsis. Now, the function 'document' first lays out the document components vertically; these components may simply be boxes resulting from the use of the 'paragraph' function defined above. The resulting box is then broken into a sequence of columns which is laid out horizontally. Finally, the box containing all the columns, side by side, is broken into a sequence of pages and these are laid out on top of each other.

Going a little bit more into the details of breaking, consider a box *b* embedding a sequence of components which are laid out in dimension *d* and glued with *g-before*, *g-between*, and *g-after*. Then:

1. Breaks are generally allowed between the individual components of *b* and also inside a component if this is itself a box whose components are laid out in dimension *d*.
2. The glue at a break point is discarded. The glue glops before and after a sequence of box components are never discarded. Considering, e.g., the glue *g-before* and *g-after*

of the outermost box, these will be appear in the front of the first, respectively at the end of the last box resulting from the break.

For stretching or shrinking boxes to a desired length, a family of **Adjust** functions is provided. They map boxes into boxes by adjusting the actual length of the embedded glue glops (if possible). Glue glops of deeply nested component sequences are also considered—but only if these are laid out in the same dimension as all embedding sequences. Starting from the normal length of the glue glops, they are adjusted proportionally to their stretchability/shrinkability in order to obtain the desired total length.

The reference point descriptions are left unchanged by these functions; therefore, specifying a reference point of a box to be, for example, at its right edge, it will also be at the right edge after the box has been adjusted.

Notice that the **Adjust** functions cannot directly be applied to the lines resulting from a breaking because these are embedded in a sequence. To solve this and similar problems, a notion of sequence functions is introduced in the following.

4.3 Parameterized functions and sequence functions

Instead of writing large and complex function definitions, composed directly from the primitives, it is often convenient to introduce auxiliary, user-defined functions as building blocks. In order to make the auxiliary functions as generally useful as possible, it is in many cases desirable to make them parameterized in the same sense as, for example, the **Break** primitive is parameterized with a length. As with the primitives, we use square brackets to enclose parameters which, in this way, index a family of user-defined formatting functions. As an example, consider a parameterized version of the previously defined paragraph function (the right-hand side is the same):

$$\text{generic-par}[\textit{line-width}] = \ldots$$

Provisions are also needed for operating on each of the boxes in a sequence. We might, for example, want to adjust each of them to a desired length. There are also examples where the boxes resulting from a breaking should be handled on a more individual basis; consider, for example, the formatting of verso and recto pages. With this situation, it might be tempting to introduce a set of general list processing primitives like those found in programming languages such as LISP.

However, as mentioned in the section on general design considerations, our approach to incremental formatting requires the formatting functions to be automatically invertible. To fulfill the invertibility requirement, a more restricted notion of sequence functions must be considered. We have found that many interesting formatting problems may be solved with sequence functions which map arguments sequences s_1 into result sequences s_2 in the following way:

1. The number of elements in s_2 equals that in s_1.
2. An element at index i in s_2 depends only on the corresponding element at index i in s_1 and the index i itself.

This kind of mapping may be illustrated by the following picture:

$$\begin{array}{ccccccc} \langle & e_1 & , & e_2 & , \ldots & e_n & \rangle \\ & \downarrow & & \downarrow & & \downarrow & \\ \langle & f_1(e_1), & & f_2(e_2), & \ldots & f_n(e_n) & \rangle \end{array}$$

To define such restricted sequence functions, it suffices to define the sequence of element functions $f_1, f_2, \ldots f_n$. In accordance with this idea, we propose the following primitives for constructing and combining (partial) sequence functions:

all *i*: *f*[*i*]
: With this sequence function, the element function at each index *i* is defined as *f*[*i*].

first: *f*
: This sequence function has only defined the element function for the element at index 1.

last *i*: *f*[*i*]
: This sequence function has only defined the element function for the last index in the argument sequence.

every *i* **modulo** *j* **from** *k*: *f*[*i*]
: With this sequence function, the element functions are defined to be *f*[*i*] for all indices *i* which satisfy the equation: $i \bmod j = k - 1$ for some *k* in the range: 1 to $j + 1$. Element functions for elements at all other indices are left undefined.

s_1 **except** s_2
: Given two sequence functions s_1 and s_2, the resulting sequence function has the same element functions as s_1 except for the indices where an s_2 element function is defined. In these cases the element functions of s_2 are used. In other words, s_2 may be used both to extend and to overwrite the collection of element functions defined with s_1.

s_1 **and** s_2
: As for the above **except** combinator. For the result to be well-defined it is, however, required that the indices for which s_1 defines element functions are *not* overlapping the indices for which s_2 defines element functions.

With these primitives we may, for example, define a sequence function for adjusting all the lines of a broken paragraph except the last line (**Id** is the identity function):

adj-lines = **all** *i*: **Adjust**[length = *line-length*]
except
last *i*: **Id**

Similarly, a function for formatting a sequence of pages as alternating verso and recto pages may be defined, given the appropriately defined functions for formatting verso and recto pages:

verso-recto = **every** *i* **modulo** 2 **from** 1**:** verso[*i*]
and
every *i* **modulo** 2 **from** 2**:** recto[*i*]

Notice that ‘recto’ and ‘verso’ are parameterized formatting functions. For each number *i*, there is a ‘verso’ function which adds the decimal representation of *i*, flushed right, as the footer of a page. Similarly for the ‘recto’ functions, except that the page number is flushed left. We define these two functions in terms of a function ‘bottom-num’ which is parameterized with both the page number and a specification of which reference point to use in the vertical alignment of the number and the text:

verso[*i*] = bottom-num[*i*,extreme]
recto[*i*] = bottom-num[*i*,origin]

Now, before defining ‘bottom-num’, some new primitives must be introduced. The **Singleton** function maps an element, e.g. a box, into a sequence only containing this one element. The **Insert** functions accept sequences as arguments and inserts an additional element either first or last in the sequence. The **Arabic** function maps an integer into a sequence of characters (digits) constituting the decimal representation of the integer. In the following definition of ‘bottom-num’, the sequences of digits are formatted like words before they are inserted below the text as page numbers.

bottom-num[*i*,*r*] = **Singleton** ∘
Insert[place = last
element = word(**Arabic**(*i*))] ∘
Layout[dimension = vertical
orientation = positive
alignment = *r* **with** *r*
glue: between = *foot-sep-glue*
reference: ref = ref(first)]

4.4 Multi-dimensional alignment

Wanting to consider two-dimensional structures such as tables and matrices, we must, at the logical level, choose either a rowwise or a columnwise representation. Deciding on, for example, the columnwise representation, logical entity types ‘table’ and ‘column’ could be used:

$$\text{table}\langle \text{column}\langle\, .\,, \ldots \rangle, \\ \vdots \\ \text{column}\langle\, .\,, \ldots \rangle\, \rangle$$

However, it is not possible with the primitives introduced so far to define the corresponding formatting functions so both a rowwise and a columnwise alignment is obtained. By using the **Layout** primitive, a columnwise alignment of the elements is achievable, but there is no way to ensure the alignment of all elements in a row:

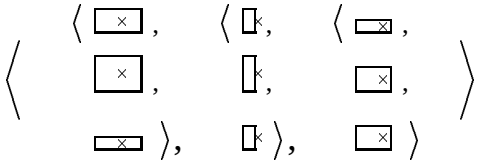
column = **Layout**[dimension = vertical . . .]
table = **Layout**[dimension = horizontal . . .]

Looking for new primitives to solve the problem, we might as a first step find a way to obtain the same potential for doing formatting no matter whether a rowwise or a columnwise representation is chosen at the logical level. A solution to this problem is to introduce the matrix transposition operation defined on matrices which are represented by sequences of sequences of elements.

Consider, for example, the case of having a columnwise representation at the logical level, but wanting to do some row-oriented formatting such as inserting the row number as the first element in each row (thus adding a column of numbers to the table). Now, letting the 'column' function be the identity function will leave a sequence of sequences to the 'table' function which may transpose it to a rowwise representation. At this point, sequence functions may be used to format all the rows, and finally the sequence of formatted rows may be laid out vertically:

```
column = Id
table  = Transpose ∘
           (all i: Insert[place = first
                          element = word(Arabic(i))]) ∘
           (all i: Layout[dimension = horizontal . . . ]) ∘
           Layout[dimension = vertical . . . ]
```

Having provided the ability to switch between columnwise and rowwise representations, let us now return to the problem of two-dimensional alignment. First, it is worth noticing that there are special cases where the problem is almost solved in advance. Consider, for example, a columnwise representation where all elements in each column have the same width and the same horizontal projection of the reference point (×) which should be used in a vertical alignment:



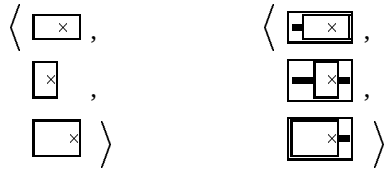
Note that this sequence of sequences of boxes does not by itself contain any layout information (except the ordering of the elements); the two-dimensional presentation displayed here has been chosen only to emphasize the similarities of those elements which we consider to be in the same column.

In this very special situation, the table may simply be transposed to the rowwise representation and the rows laid out horizontally with fixed space between the elements; owing to the sizes of the elements and the placement of their reference points, the vertical alignment is already there. In practice, such special cases are rare; usually the elements have different sizes. However, looking for a way of obtaining two-dimensional alignment, we may introduce a new primitive which pads each element with glue glops in order to produce this special case.

The padding of, for example, a sequence of column elements can be realized by embedding each column element as the only element of a box; the box must be laid out

horizontally with appropriate glue before and after the element, so the desired width and vertical alignment with the other padded elements is obtained. In addition, the embedding box must have reference points coinciding with those of the embedded element. This is the way the family of **Pad** functions work; each function is indexed by the dimension of the boxes used for padding and an identification of the reference points which are to be aligned.

As an example, consider the following picture which illustrates horizontal padding of a sequence of column elements. Each element has a single reference point (×) which should be used in a vertical alignment. The padded sequence is the one to the right; thick horizontal bars are used for symbolizing the inserted glue:



With the **Pad** and **Transpose** primitives, it is thus possible to define the 'column' and 'table' functions so the desired two-dimensional alignment is obtained:

```
column = Pad[dimension = horizontal
             alignment = ... with ... ]
table  = Transpose ∘
         (all i: Layout[dimension = horizontal ... ]) ∘
         Layout[dimension = vertical ... ]
```

Together with the sequence functions, the primitives are, in fact, powerful enough that they can be used for solving three-dimensional alignment problems. Such problems may occur in connection with the preparation of overhead transparencies. Besides this, the primitives have good uses by themselves, especially **Transpose**. It should be noted, however, that the primitives only provide for the very basics of high-quality tabular typography as, for example, considered in Beach's work [15]. Table heads which span more than one column and rules which separate rows or columns are some of the features which are not supported so far. To solve these problems, it is necessary to introduce additional primitives.

5 DOCUMENT STYLES AND TYPE CHECKING

In the foregoing, it has been demonstrated how essential aspects of the mapping from the logical to the physical view of documents may be modeled by the evaluation of simple expressions in a functional language. The expressions, which represent the logical document view, have been exemplified in Figure 2, page 8 and on page 18.

There, it was implicitly assumed that certain rules were obeyed regarding which kinds of logical entities could be embedded in which. For instance, in the table formatting example, a table was expected to contain a sequence of columns. There are good reasons for requiring such rules to be explicitly stated, e.g. to make it possible for the editing/formatting

system to ensure uniformity of the documents. Here, we will investigate another task made possible by such rules, namely the type checking of formatting function definitions.

The formatting primitives introduced are only well-defined for certain types of arguments. For example, the **Layout** functions only work on sequences of boxes or characters, the **Break** functions require boxes as arguments, and the **Transpose** function is well-defined only for sequences of elements which, by themselves, all are sequences with the same length. It would therefore be desirable to have a notion of type, allowing one to identify bad uses of the formatting functions before doing any actual formatting.

Dealing with *specific* expressions, which represent the logical view of specific documents, this type checking problem is comparable to type checking in other functional languages (see Cardelli and Wegner's work [16] for an introduction and further references). There are differences, due, for example, to the special box data type, but to discuss the details of a type system for FFL is not within the scope of this presentation. Instead, we will look at a more general problem which is relevant to the type checking of formatting function definitions as opposed to the type checking of function definitions in general-purpose languages.

Defining a set of formatting functions, it is the aim that the functions be well-defined not just for a given, specific document, i.e. a specific expression, but for a whole class of documents. Usually, such a document class is defined by a context-free grammar describing the possible logical structures of documents belonging to the class. Consider, for example, the following very simple grammar where the notation l^* is used to denote zero, one, or more repetitions of the kind of logical entities which are described by l.

$$\begin{aligned} \text{document} &::= \text{paragraph}^* \\ \text{paragraph} &::= \text{sentence}^* \\ \text{sentence} &::= \text{word}^* \\ \text{word} &::= \text{CHAR}^* \end{aligned}$$

With the FFL primitives, formatting functions corresponding to each kind of logical entity may be defined; let us name these: f_{document}, etc. Now, assume that the types of these functions may be inferred by use of a type system which defines the type of each FFL primitive. The problem is then to establish a connection which will allow utilization of the grammar in the type checking.

It turns out that each production in the above grammar may be viewed as a *type assertion* concerning the type of the corresponding formatting function. Let us assume that, in the type system considered, 'char' is the type of characters and '$\mathsf{seq\ of}\ t$' is the type of sequences of elements which themselves are of type 't'.Then, the grammar asserts:

$$\begin{aligned} f_{\text{document}} &: \mathsf{seq\ of}\ \textit{paragraph} \rightarrow \textit{document} \\ f_{\text{paragraph}} &: \mathsf{seq\ of}\ \textit{sentence} \rightarrow \textit{paragraph} \\ f_{\text{sentence}} &: \mathsf{seq\ of}\ \textit{word} \rightarrow \textit{sentence} \\ f_{\text{word}} &: \mathsf{seq\ of\ char} \rightarrow \textit{word} \end{aligned}$$

Here, the names '*document*', '*paragraph*', '*sentence*', and '*word*' may be viewed as type variables denoting arbitrary types, but with the restriction that a type variable which is used in different assertions must denote the same type in all of these.

In this setting, the type checking task reduces to the problem of checking whether there exists an assignment of types to the type variables such that all assertions become correct with respect to the inferred types. Consider, for example, a type system where ‘box’ is the type of boxes and **Layout** functions have the type: ‘seq of $t \rightarrow$ box’ for arbitrary types ‘t’. If f_{word} then is defined just to lay out the characters, it also receives this type and, in that case, the type variable ‘*word*’ must stand for the type ‘box’ in order for the last assertion to be correct. With this information, the second last assertion may be checked, thereby determining the type corresponding to the variable ‘*sentence*’ and so forth.

Grammars are generally not as simple as the one shown here; there may, for example, be productions which allow a given kind of logical entity to be composed in a number of alternative ways. The grammar may also be circular, thus reflecting that a certain logical entities may appear inside other entities of the same kind. To cope with those situations, sum types[6] and recursively defined types must be included in the type system; these kinds of types can then be used to express how a grammar should be viewed as a set of type assertions.

5.1 Context-dependent formatting

Having combined the notion of formatting functions with that of document grammars, it is a natural step to go from simple, context-free grammars to the more expressive attributed grammars originally introduced by Donald Knuth [17]. Attributed grammars could, for example, be used for synthesizing the table of contents or for systematic production of section and subsection numbers. They may also be used to propagate information about the context, such as the current font and line width, down in the logical structure. This topic is considered further in work by the author [18] and a master’s thesis by M. N. Jakobsen and J. H. Hansen [19].

As a link between the attributed grammar and formatting functions, the attributes could then be used as parameters in the function definitions, for instance for specifying the break length used with a **Break** function or for specifying an element to be inserted with an **Insert** function.

Recent work in attributed grammars has concentrated on incremental evaluation of the attributes in connection with changes to an expression (in our case a document) satisfying the grammar; for a survey, see the work by Deransart, Jourdan and Lorho [20]. This means that formatting function parameters, such as a break length or a synthesized table of contents, may be very efficiently re-evaluated after a change to the document. Efficient re-evaluation of the formatting functions themselves is, however, not covered by this scheme. The following section addresses this problem.

6 INCREMENTAL FORMATTING

With integrated editing/formatting/viewing systems, it is the main idea that the effect of making a change to the logical view of the document, such as adding or deleting a

[6] Sum types are also called union types or variant types.

character, should be immediately observable. This implies that the formatting and viewing must occur simultaneously with the editing.

The response time requirements, which follow from this approach, rule out the possibility of implementing such systems by simply re-evaluating the expression representing the logical view of the whole document for each change made to the expression. To meet the requirements, it it necessary to use techniques for incremental formatting. Rather than mapping the whole expression to its physical representation (a box), incremental techniques map *changes* to the expression into changes which can be efficiently applied to the corresponding box.

We will in the following restrict ourselves to only consider one kind of editing operation: replacement of character instances in expressions. However, owing to the potential difference in character widths, such replacements call for a rather general treatment which can be extended straightforwardly to cover insertions and deletions of characters and subexpressions representing whole logical entities.

Consider a notion of *logical* and *physical indices*, which uniquely identify character instances in respectively expressions and boxes. Given an expression *e* and the box *b* which results from evaluating *e*, there is then a relation between those logical and physical indices which identify the same character instance in respectively *e* and *b*. Disregarding characters which have been introduced with the **Insert** functions, this relation is one-to-one. In the following, we will use the symbols i and i' to denote such related logical and physical indices.

The result of replacing the character instance at logical index i in the expression e with the new character c is written $\Delta_{c,i}(e)$. When replacing the same character instance in the corresponding box b, the result is expressed as $\Gamma_{c,i'}(b)$.

Regarding the mapping of changes to the expression e into changes to the corresponding box b, it would be desirable to show that character replacement in e could be mapped directly to character replacement in b, i.e. to show that the following diagram commutes:

$$\begin{array}{ccc} e & \xrightarrow{\Delta_{c,i}} & e' \\ {\scriptstyle\mathcal{E}}\downarrow & & \downarrow{\scriptstyle\mathcal{E}} \\ b & \xrightarrow[\Gamma_{c,i'}]{} & b' \end{array}$$

In this diagram, $\mathcal{E}$ is the function which evaluates an expression into a box, given a definition of all formatting functions used.

We have not presented any definitions precise enough to serve in a formal argument concerning this property. However, the assumption that "real" typographical breaking, e.g. of paragraphs, can be modeled by the **Break** primitive on boxes gives a sufficient basis for concluding that, in general, *the above diagram cannot be expected to commute.* Considering the breaking of paragraphs, it is clear that replacing a character instance in the expression representing the paragraph implies more than just replacing the corresponding instance in the box. If the old and new characters differ in width, the new paragraph may in addition have to be broken in a different way.

6.1 Two approaches

Having rejected the simple relationship between updates of expressions and boxes, we will in the following consider two other approaches to incremental document formatting.

The first of these is based on a subsumed property of $\mathcal{E}$: that the result of evaluating an expression *e* is some function of the evaluation of its subexpressions. This means that after replacing a character instance in *e*, it is only necessary to apply $\mathcal{E}$ to the subexpressions of *e* which embed the new instance. The application of $\mathcal{E}$ to the other subexpressions is not affected by the replacement, so the previous results can be reused in these cases. We call this "the reuse approach".

The second approach is based on a somewhat broad interpretation of the requirement that "the effects of changes to the logical view of a document should be immediately observable". Rather than re-evaluating the expression representing the whole document, it is the idea to obtain a first approximation by *locally re-evaluating* the subexpression which immediately embeds the changed character; this could, for example, be an expression representing a word. The effects, which are immediately observable in this first approximation, only concern the very local context of the changed character, but a series of better approximations may be obtained by locally re-evaluating the embedding expressions which represent the paragraph, section, etc., until the whole document has been reformatted.

The key to this approach is a property which the mapping $\mathcal{E}$ from expressions to boxes may possess (depending on the formatting function definitions). As mentioned above, the result $\mathcal{E}(e)$ of evaluating an expression *e* is some function of the boxes resulting from evaluating the subexpressions of *e*. For some subexpressions, this function may simply have the effect of embedding the corresponding boxes as components of $\mathcal{E}(e)$. Consider a subexpression *s* and the corresponding box $\mathcal{E}(s)$ which can be identified as a component *c* of $\mathcal{E}(e)$. This situation is illustrated in Figure 5. Assume further that a character in *s* is replaced, so we obtain a modified expression e' with the modified subexpression s'. Now, the correspondence between *s* and *c* motivates a local re-evaluation by simply replacing *c* with $\mathcal{E}(s')$. The result of this replacement can be shown to be an approximation to the correct result $\mathcal{E}(e')$ in the sense that only the breaking is incorrect. For example, the line

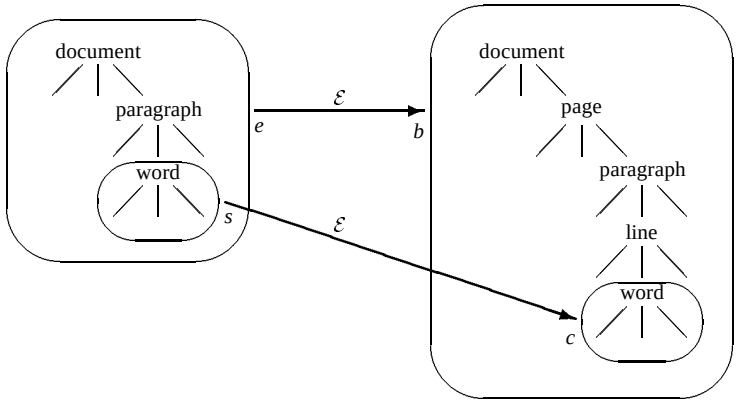


Figure 5. Preservation of the result of evaluating a subexpression

containing the changed character may be too long or too short and the page may not have the correct height.

The strong point of the local reformatting approach is the efficient construction of the first approximations, and the possibility of postponing the later approximation steps if necessary. Postponing the later steps allows for fast response in the case of multiple successive changes to the logical view of the document.

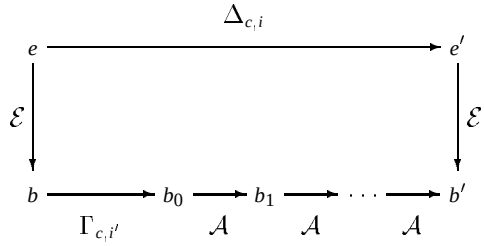
6.2 Construction of approximations

Keeping track of the relationship between subexpressions and box components illustrated in Figure 5 is not straightforward. In work by the author [21] this is done by extending the notions of boxes and sequences to allow for annotations. Different annotations are used to inform about:

- The kind of logical entity which a box or sequence represents (if any). This gives information about the formatting functions which have been used to generate the object.
- The fact that a box is a fraction resulting from a breaking.
- The annotations which were attached to an immediately embedding sequence prior to a transposition. Transposing, for example, a table from rowwise to columnwise representation, the annotations previously attached to each row are "saved" as annotations attached to each element of the row.

By defining how to invert formatting functions composed from the build-in primitives, these annotations make possible a retrieval of the expression from which a box stems (when this has a meaning). We call this to *invert* the box.

Now, the character replacement function Γ working on boxes can be used to obtain the first approximation and subsequent approximations may then be constructed by searching outwards to the nearest enclosing box which corresponds to a logical entity. This box is first inverted, then re-evaluated and the result is inserted in its place. Continuing the construction of approximations, the box representing the whole document will be reached in a finite number of steps. Having inverted and re-evaluated this, the resulting box should be the correct physical representation of the whole changed document. This strategy is illustrated by the following diagram where $\mathcal{A}$ denotes the function for constructing new approximations:



One of the interesting aspects of this approach is the possibility of working only with the extended box representation of documents. As the bottom line of the above diagram shows,

the initial change to the expression e may instead be applied to the corresponding box b and after a series of approximations, the correct box b' is obtained.

As one might expect, the definition of the approximation function is quite complex. This does not, however, worry us since it is just a part of an efficient implementation for which the abstract specification is very clear:

$$\mathcal{E}(\Delta_{c,i}(e))$$

In this way, there is a firm reference against which to validate the implementation. With the formal definitions given [21], there is even a basis for a mathematical proof of the correctness of the stepwise approximation technique, i.e. a proof that the diagram above actually does commute.

7 FURTHER WORK

The elements of FFL presented in the previous are not sufficient for handling a number of interesting formatting problems. Some of these are listed in the following and it is outlined how they may be dealt with by introducing suitable extensions.

Consider, for example, the placement of footnotes, margin notes and floating figures. It is common for these problems that they involve nontrivial constraints on the relative placement of two boxes representing respectively a reference and the material referred to. By extending the notion of boxes to allow for temporarily invisible components, new families of functions could be introduced thereby enabling problems of this kind to be solved. The functions could: (1) make a box invisible and classify it by giving it a certain name; (2) extract from a box, the sequence of all embedded, invisible boxes of a certain class; and (3) break a box by extracting and placing certain classes of its embedded invisible boxes according to specified rules.

Another class of problems concern box distances which depend not only on the box shape enclosing the box contents but also on other features of the contents. Examples include kerning and line spacing with respect to baselines. Like in TEX, the problems may be solved by: (1) dropping the restriction that the same kind of glue must be used between all adjacent components in a box; and (2) introducing new functions for inserting or replacing glue glops. Kerning must then be handled in an *ad hoc* manner, by recognizing the individual cases where the use of box shapes leads to unacceptable spacing. Alternatively, more general geometric shapes may be introduced in order to allow for a general handling of kerning-like problems.

Regarding implementation issues, the notion of three-dimensional boxes with multiple reference points in the three-dimensional space will lead to unrealistic space and time requirements in a straightforward, naive implementation. Using compilation and optimization techniques there is, however, a good basis for overcoming this problem. Given a document grammar and a set of formatting function definitions, it will, for example, be possible with a static analysis to identify the dimensions and reference points actually utilized in the different kinds of boxes which may be constructed during the formatting. Based on such an analysis, individual, space-optimal box representations may be generated automatically and, correspondingly, the code generated for formatting functions may be individually tailored to fit the different box representations.

Our desire to have the **Break** primitives produce optimal breaks, following the ideas of Knuth and Plass [1], presents us with another implementation problem. Compared with traditional, suboptimal, one-line-at-a-time paragraph breaking, Knuth's algorithm needs only about twice as much computation for finding the optimal breakpoints of a paragraph. However, in combined editing/formatting/viewing systems, the problem is that paragraphs may be changed at speeds of 10 operations per second and when considering efficient re-computation of breakpoints after a change, Knuth's approach looks less attractive.

The simpler, suboptimal approaches invite efficient, incremental implementations which start from the previously computed breakpoints, "fixing" exceedingly long or short lines, one at a time, and allowing new changes to be introduced in between. The fact that optimal breaking requires a global view of the whole paragraph rather than a local view of a single line has kept researchers and designers from considering it in connection with combined editing/formatting/viewing systems.

In a recent master's thesis by Ulrikka Thyssen [22] it is, however, shown that the search tree needed for finding an optimal solution may be efficiently constructed from previously computed search trees. We plan to incorporate this technique in a prototype implementation of an FFL interpreter.

8 CONCLUSION AND DISCUSSION

A document formatting model has been proposed and the development of its central component, a functional language, has been presented. The model, as it appears here, only accounts for some of the many formatting aspects which must be considered in "real" document production. To this, we have two comments to add.

First, we have done some experiments with the model and believe that it may be extended in a consistent and natural way to cope with many of the relevant formatting problems. New primitives must be added to the functional language and the notion of boxes may have to be extended.

Secondly, it should be emphasized that we consider the process of developing the model as important as the model itself. The paper does describe a formatting model and a language, but the aim has also been to illuminate the different rôles of document models and the advantages which can be obtained by recognizing these rôles.

Compared to previous work, the formatting language presented as part of the model is distinctive in several ways: it is founded on simple mathematical concepts, such as sequences, functions and function compositions, it may be type checked to ensure that no run-time errors can occur, and it is designed to allow for an efficient incremental implementation.

9 ACKNOWLEDGEMENTS

The work reported here was sponsored by the Danish Technical Research Council with grants 16-4322.E and 16-4535.E to the BENEDICK project. Sincere thanks for many valuable contributions go to the present and former members of the BENEDICK project group: Flemming M. Damm, Jesper H. Hansen, Michael N. Jacobsen, Peter Grønning, Thomas Q. Nielsen, Ulrikka Thyssen, Henrik Bach and Johan P. Møller.

REFERENCES

1. Donald E. Knuth and Michael F. Plass, 'Breaking paragraphs into lines', *Software—Practice and Experience*, **11**:1119–1184 (1981).
2. Michael F. Plass and Donald E. Knuth, 'Choosing better line breaks', In Nievergelt *et al*., editor, *Document Preparation Systems*. North-Holland Publishing Company, 1982.
3. Michael F. Plass, *Optimal Pagination Techniques for Automatic Typesetting Systems.* PhD thesis, Department of Computer Science, Stanford University, June 1981. Report no. STAN-CS-81-870.
4. ISO. *Information processing – Text and office systems – Office Document Architecture (ODA) and interchange format – Part 1–6.* International Standard no. 8613.
5. Andrzej Borzyszkowski and Stefan Sokołowski, 'Understanding an informal description: Office documents architecture, an ISO standard', In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM88: VDM – The Way Ahead*, pages 48–63. VDM-Europe, Springer-Verlag, 1988. Lecture Notes in Computer Science, vol. 328.
6. Donald E. Knuth, *The TEXbook*. Addison-Wesley Publishing Company, 1986.
7. Cliff B. Jones, *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall International, 2nd edn, 1990.
8. Dines Bjørner and Cliff B. Jones, *Formal Specification and Software Development.* Series in Computer Science. Prentice-Hall International, 1982.
9. Adobe Systems Inc. *PostScript Language Reference Manual*, Addison-Wesley Publishing Company, 1985.
10. Jeff Johnson and Richard J. Beach, Styles in document editing systems. *IEEE Computer*, pages 32–43, January 1988.
11. Jacques André, Richard Furuta, and Vincent Quint, editors. *Structured Documents*. Cambridge University Press, 1989.
12. Richard Furuta, Jeffrey Scofield, and Alan Shaw, 'Document formatting systems: Survey, concepts, and issues', *Computing Surveys*, **14**(3):417–472 (1982).
13. J. Backus, 'Can programming be liberated from the von Neumann style? A functional style and its algebra of programs', *Communications of the ACM*, **21**(8) (1978).
14. K. E. Iverson, *A Programming Language*. John Wiley and Sons, Inc., New York, 1962.
15. Richard J. Beach, 'Tabular typography', in J. C. van Vliet, editor, *Text processing and document manipulation, Proceedings of the International Conference*, pages 19–33. British Computer Society, Cambridge University Press, April 1986.
16. Luca Cardelli and Peter Wegner, 'On understanding types, data abstraction, and polymorphism', *Computing Surveys*, **17**(4):471–522 (1985).
17. Donald E. Knuth, 'Semantics of context-free languages', *Math. Systems Theory*, **2**(2):127–145 (1968).
18. Bo Stig Hansen, 'Attributed document grammars', Technical Report ID-TR 1990-68, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, February 1990.
19. Jesper H. Hansen and Michael Jakobsen, 'Incremental evaluation of attributed grammars for documents', Master's thesis, Department of Computer Science, Technical University of Denmark, August 1989.
20. Pierre Deransart, Martin Jourdan, and Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.
21. Bo Stig Hansen, 'Towards a theory of incremental document formatting', Technical Report ID-TR 1990-67, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, February 1990.
22. Ulrikka Thyssen, 'Optimal, incremental text breaking', Master's thesis, Department of Computer Science, Technical University of Denmark, January 1990.