A Functional Description of TEX's Formula Layout

REINHOLD HECKMANN, REINHARD WILHELM
Fachbereich Informatik, Universität des Saarlandes
Saarbrücken, Germany
{heckmann,wilhelm}@cs.uni-sb.de

Abstract

While the quality of the results of TEX's mathematical formula layout algorithm is convincing, its original description is hard to understand since it is presented as an imperative program with complex control flow and destructive manipulations of the data structures representing formulae. In this paper, we present a re-implementation of TEX's formula layout algorithm in the functional language SML, thereby providing a more readable description of the algorithm, extracted from the monolithical TEX system.

1 Introduction

The mathematical formula layout algorithm used by D. E. Knuth's TEX typesetting system generates remarkably good output. However, any attempt to understand the reasons for this success leads to deep frustration. The algorithm is informally described in Appendix G of the *TEXbook* (Knuth, 1986a) using English prose with some formal fragments. While this description provides a useful and welcome overview, the details are not completely correct. A complete and exact description of the whole TEX implementation is presented in *TEX: The Program* (Knuth, 1986b). It contains the full source code of the TEX system, structured into logical units, well commented, and documented by cross references. Nevertheless, anyone who wishes to gain a full understanding of TEX's algorithms from these descriptions must invest great efforts. The reason is that both the documented source code and the informal description are typical examples of imperative programs, involving complex control flow (including goto's) and complicated manipulations of the various data structures. In particular, the usage of global variables obscures the interdependencies between the subtasks. It is by no means obvious where certain information is produced, where it is changed, and where it is consumed.

In this paper, we present a new implementation of TEX's formula layout using the functional language SML (Paulson, 1991). The purpose of this re-implementation effort is twofold: First, it provides a novel and hopefully more understandable description of TEX's formula layout algorithm. This description was developed as part of a textbook (Wilhelm and Heckmann, 1996) on document processing. Second, it extracts this particular subtask of TEX from the monolithically designed TEX system, leading to the possibility to study it independently and to potentially use it in systems other than TEX. Some remarks on status and availability of the implementation are contained in Section 2.1 and in the conclusion (Section 7).

The main task of the formula layout algorithm consists in translating formula terms (a kind of abstract syntax for formulae) into box terms, which describe the sizes and relative positions of the formula constituents in the final layout. Knuth's original data structure for formula terms was designed to achieve space and time efficiency. In our opinion, it is misconceived from a more logical point of view, as it mixes semantic concepts with details concerning spacing. Hence, we propose a new data structure for formula terms with a clean and simple design. This change does not cause harm since formula terms do not occur in other subtasks of TEX. In contrast, box terms are produced by nearly all subtasks of TEX. Thus, we tried to mimic the structure of Knuth's original box terms as close as possible in our SML code in order to keep a well-defined interface with other subtasks of the TEX system, each of which might be re-implemented in the future.

Concerning the algorithm itself, we also tried to catch exactly TEX's behavior, but still we cannot definitively claim that our algorithm is the 'same' as Knuth's original algorithm. The change in the structure of formula terms and the switch to the functional paradigm causes big changes in the structure of the code and the temporal order of operations. On the other hand, we claim that for equivalent input formulae, the resulting box terms are equivalent, apart from certain round-off errors (Knuth uses a sophisticated self-defined arithmetic, while we use SML's standard arithmetic functions). Our claim is confirmed by practical tests: After programming a translator from box terms to `.dvi` code (the standard output from TEX), we can print the results of our formula layout program and verify that they are visually indistinguishable from the results of TEX.

In Section 2, the overall view of formula layout is presented, together with some details that influence the typesetting: the styles of formulae and subformulae (Section 2.2), style parameters (2.3), and character dimensions (2.4). Knuth's account of these things is very concrete. In contrast, we present an abstract interface that hides the details of font table organization, and makes clear how the information is used.

Section 3 presents the output of the layout algorithm, *box terms*, and Section 4 the input, *formula terms*. In Section 5, a set of specialized functions is presented that translate subformulae of various kinds into box terms. Section 6 treats the translation of whole formulae, including the introduction of implicit spaces between adjacent entities of certain kinds.

A fair estimation of our achievements is contained in the conclusion (Section 7). Of course, our functional solution is not simpler than the problem admits: Formula layout is an inherently difficult problem, not in terms of computational, but of algorithmic complexity. There are many different kinds of mathematical formulae, whose layout is governed by tradition and aesthetics. Algorithms for formula layout must distinguish many cases and pay attention to many little details.

In the presentation of our implementation, we do not list the SML program in its natural ordering. SML programs must be written in a bottom-up style because everything must be defined prior to its usage. For some of the description in a paper like this, a top-down approach is more suitable. To save space, we do not present the arrangement of the code into signatures and structures; every type specification is in fact part of a signature, while every function definition is part of some structure.

2 An Overview of Formula Layout

2.1 Complete Processing of Formulae

In TEX, a formula is entered using a *formula description language*, which is a sublanguage of TEX's input language. The formula $\sum_{i=1}^n i = \frac{n(n+1)}{2}$, for instance, is described by the expression

```
$\sum_{i=1}^n i = {n (n+1) \over 2}$.
```

Formulae are processed as follows:

1. The string representing the formula is read and parsed into a *formula term*, which essentially corresponds to the logical structure of the formula, but additionally contains some spacing information.
2. The formula term is recursively translated into a *box term*, which describes exactly the layout of the formula.
3. The box term corresponding to the formula becomes a subterm of the box term for the page where the formula occurs.
4. The box terms for the pages of the document are translated into the `.dvi` language and written to some file. The `.dvi`-file essentially consists of commands *where* to print *which* symbols.

Before any formula is processed, a preprocessing step reads information about the size of characters, the thickness of fraction strokes, etc. This information comes from `.tfm` *files*, which exist for every combination of font and type size.

This paper is only concerned with point 2. The actual SML implementation also contains the preprocessing step, addresses point 3 in a trivial manner (the whole page is a sequence of formulae), and performs step 4 completely in order to obtain a visible output. It does not (yet) address point 1, i.e. formulae must be entered as formula terms in SML notation.

2.2 Formula Styles

Formulae may appear in two different contexts: as *inline formulae*, e.g., $\sum_{i=1}^n i = \frac{n(n+1)}{2}$, within a line of text, and as *displayed formulae* in a line of their own, e.g.,

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

As one can see, the layouts of inline and displayed formulae are quite different. In our example, this concerns the positions of the limits of the sum and the sizes of the sum symbol and of the parts of the fraction. These properties are influenced by the *layout style* (Knuth, 1986a, page 140). Displayed formulae are typeset in *display style* `D`, while *text style* `T` is used for inline formulae. There are two further styles, which apply to certain subformulae: *script style* `S` for 'scripts' (superscripts and subscripts) and other subformulae with small typesetting, and *script script style* `SS` for scripts of scripts and other subformulae with tiny typesetting.

In TEX, there are four more styles, `D`$'$, `T`$'$, `S`$'$, and `SS`$'$, which are called *cramped styles*.

They apply to subformulae that are placed under something else, e.g., denominators. In cramped styles, superscripts are less raised than in the corresponding uncramped styles. Analyzing the usage of the eight TEX styles, it turns out that they may be regarded as pairs of a main style and a Boolean value 'cramped' where the two components are independently calculated and used. Thus we decided to separate them completely, and define

```
datatype style  =  D | T | S | SS.
```

2.3 Style Parameters

Every style has its own rules where to place scripts, numerators, denominators, etc. These rules are given by *style parameters* read from the `.tfm` files during the preprocessing phase. In TEX, these style parameters are accessed via the style size, which is identical for `D` and `T`. In many cases, two different style parameters are used for the same purpose, one for style `D`, and the other one for the remaining styles. Our preprocessing, however, produces slightly more high level style parameters that depend on the style directly. The conversion from styles to style sizes and the distinction between `D` and the remaining styles is done internally.

Hence, nearly all style parameters are given by functions of type `style` → `dist`, where `dist` is the type used for horizontal and vertical distances and lengths. As in TEX, `dist` is an integer type where a `dist` value of 1 corresponds to 2^{-16} pt. Here, 'pt' is the abbreviation of 'point', a traditional unit of measure for printers and compositors in English-speaking countries. In TEX, one inch equals exactly 72.27 pt.

In this overview, we present a few important style parameters only. The remaining ones are introduced when needed. For example, the parameters pertaining to the positions of subscripts are only explained when the typesetting of subscripts is presented.

Style parameter `xHeight` provides the height of the letter x in the current style. Parameter `MathUnit` yields the size of a 'mathematical unit', which by TEX's definition is one eighteenth of the width of the letter M in the current font. Parameter `RuleThickness` contains the thickness of fraction strokes and of over- and underlines, as in $\underline{x}+\overline{y}$. In TEX's layout algorithm (and in our re-implementation), it is also used to control spacing; an abuse that makes it impossible to change the rule thickness separately from the spacing.

Parameter `AxisHeight` contains the vertical distance from the *axis* to the *baseline*. The latter is the line where most of the letters and all digits sit on, while the former is the line where to put fraction strokes (consider, for instance, $1+\frac{2}{3}$; the baseline is at the bottom of the digit 1). The axis also plays some role outside of fractions. In a well-designed mathematical font, many symbols are placed symmetrically to the axis (consider plus and equal in $1+\frac{2}{3}=\frac{5}{3}$).

Apart from the style parameters, the layout is influenced by some constants that do not depend on the style. While their role in layout is similar to that of the style parameters, they have a completely different origin: they are not read from the `.tfm` files, but explicitly specified in the code. A sample constant is `scriptSpace` of type `dist`, which specifies an additional white space of 0.5 pt to be inserted after superscripts and subscripts.

2.4 Characters and Character Dimensions

The main result of the preprocessing phase is the information about the dimensions of every single character. Before we present how this information is provided to the remainder of the program, we have to consider how characters are coded internally.

Consider, say, the formula x^x, which is entered as `x^x`. In the formula description, there is no difference between the two occurrences of `x`. This is still the case in the formula term obtained by parsing the description: Both occurrences correspond to the same subterm, which is a pair consisting of a *font family* ("math italic" in our example) and a *character code* (120 in our case). After typesetting, however, the two occurrences of x are different in size; the font family has been replaced by a concrete font that is different for the two occurrences. The choice of some concrete font from a font family of course depends on the style, `T` for the first occurrence of x and `S` for the second.

The preprocessing phase of our algorithm provides the types that are needed for these encodings: `family` for font families, `fontNr` for font numbers that refer to actual fonts, and `charCode` for character codes. It also provides the function

```
fontNumber:   style -> family -> fontNr
```

for the style-dependent selection of a concrete font from a font family.

The `.tfm` files contain four basic size parameters for each character in every font. These parameters are read by the preprocessing phase and made available via the following functions:

```
charHeight, charDepth, charWidth, charItalic:
   fontNr * charCode -> dist.
```

The *height* of a character is the distance from its top end to the baseline; e.g., 'a' and 'g' have the same height, and 'f' is bigger. The *depth* is the distance from the baseline to the bottom end; e.g., 'a' has depth 0, whereas 'g' has positive depth. The *width* is the horizontal size as it is used in the composition of ordinary text. It does not take into account a possible extension of the upper right part of the character. This extra extension, i.e. the difference between the overall horizontal extension and the given width of the character, is the *italic correction* provided by function `charItalic`. It becomes visible in cases where both superscripts and subscripts are attached to a character; in f_1^1, for instance, the horizontal distance between the two scripts is the amount of the italic correction of the letter f.

It is a particularly painful subtask of formula layout to decide whether the italic correction should be added to the width of a particular occurrence of a character. In the original TEX program, this issue is complicated by the habit of taking advantage of the destructive nature of imperative programming, by subtracting some corrections that were added in earlier stages of the layout process. In our SML program, an added correction will never be removed again.

The `.tfm` files contain another kind of information which is made available by

```
larger : fontNr * charCode -> charCode.
```

For some characters, a larger version is available in the same font. In this case, the code of this larger version is returned, while the code is unchanged otherwise.

Internally, all character information is stored in vectors to guarantee efficient access.

2.5 Top Level Layout Functions

Recall that formulae may appear in two contexts: inline and displayed. Therefore, we define two different functions for the translation from formula terms to box terms:

```
inlineFormula, displayFormula:  mlist -> hlist
```

where `mlist` (mathematical list) is the type of formula terms (Section 4.2), and `hlist` (horizontal list) is the type of box terms (Section 3.4).

Of course, the two functions are closely related; we implement them by a more general function that will also be used for the recursive translation of subformulae. This function will of course depend on the style and the Boolean crampedness. Apart from the style, there is another difference between inline and displayed formulae: inline formulae may be broken across lines, while display formulae may not. Hence, inline formulae must be equipped with *penalty information* telling the line breaking algorithm where to break. On the other hand, penalties need not be inserted into displayed formulae. Thus, the more general function has a second Boolean argument, which indicates whether penalties should be inserted.

```
val MListToHList:  style -> bool -> bool -> mlist -> hlist
val displayFormula  =  MListToHList D false false
val inlineFormula   =  MListToHList T false true
```

Before we describe the implementation of `MListToHList`, we must present the data structures for formula terms (`mlist`) and box terms (`hlist`).

3 The Target Representation: Box Terms

The main task of formula layout is the translation of formula terms into box terms. In Sections 3.1 through 3.3, we describe these box terms informally, on a semantical level. The actual TEX and SML implementations follow in the remaining subsections. In particular, 3.4 contains the necessary type definitions, and 3.5 the computation of the dimensions of box terms. The remaining two subsections describe a selection of basic functions to build or manipulate boxes.

3.1 Boxes and their Dimensions

A *box* is a rectangle whose edges are parallel to the page edges. Each box comes with a horizontal *baseline* and a *reference point* that is situated at the point where the baseline meets the left margin of the box. The reference point is used to position the box within its context. For the outside world, a box is characterized by three size parameters (of type `dist`): *height* h, *depth* d, and *width* w. The height is the vertical distance from the top margin to the baseline, the depth is the vertical distance from the baseline to the bottom margin, and the width is the horizontal distance from the left to the right margin (see Figure 1).

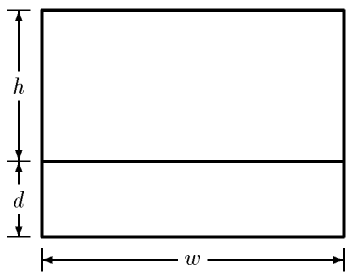

Fig. 1. Box with height h, depth d, and width w

Each of these dimensions may be negative. To understand what this means, we must clarify the meaning of 'distance from A to B'. Each page is equipped with a Cartesian coordinate system, whose x-axis points to the right as usual, while the y-axis points downward. These directions were chosen to follow the flow of text for scripts commonly used by European languages. Using these coordinates, we may define:

- *Horizontal distance* from A to B = x-coordinate of B − x-coordinate of A
- *Vertical distance* from A to B = y-coordinate of B − y-coordinate of A

Most boxes have positive width, which means that their left margin is to the left of their right margin. Boxes with negative width may, for instance, arise from negative horizontal spaces. They have the property that their left margin (which carries the reference point) is to the right of their right margin, which might appear surprising. Nevertheless, this does correspond to their unusual physical properties: they really do have negative width, which would be impossible for any concrete physical body.

Height and depth are positive if the baseline lies within the shape of the box, i.e. below the top margin and above the bottom margin. The height is negative if the baseline is above the top margin, and the depth is negative if it is below the bottom margin.

It is important to note that a box on its own does not have an absolute position on the page. The position of a box is only fixed relative to the reference point of a superbox once the superbox has been formed. Accordingly, every box fixes the positions of its subboxes relative to its own reference point. In fact, it is difficult in TEX to fix anything to a particular point on the page.

3.2 Horizontal Combination of Boxes

Boxes can be combined to form larger boxes either horizontally or vertically. In a standard horizontal combination of a list of boxes, the constituents are placed next to each other from left to right so that their baselines appear on the same horizontal line, and the right margin of each box is aligned with the left margin of the next box. The reference point of the combination is that of the leftmost constituent. Figure 2 shows a horizontal combination of three boxes. The baseline of the combined box is the bold line through the middle of the box. The small gaps between any two neighboring boxes, as well as between the subboxes and the surrounding box, were introduced to enhance the readability of

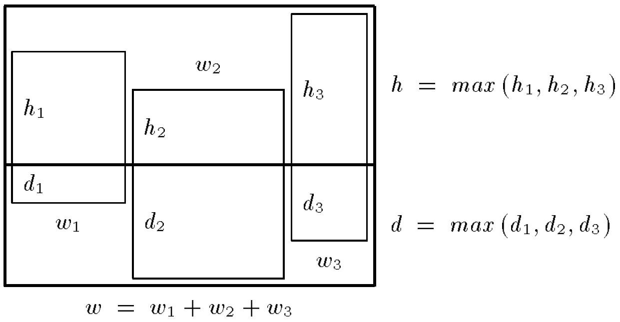

Fig. 2. A horizontal combination of three boxes

the figure. In reality, the three subboxes touch each other and sit tightly within the surrounding box. Hence the (natural) width of the combined box is the sum of the widths of the constituents.

There are two modifications of this standard procedure. First, the surrounding box may be given a width different from its natural width. This is done by stretching or shrinking *glue*, i.e. white space boxes whose width admits some variability. The main application is the adjustment of lines of text to a common width where the glue components are the interword spaces. As we shall see, there are some applications of this principle in formula layout as well.

The second modification consists in the possibility to vertically shift some boxes in the horizontal list. The amount of shift is added to the y-coordinate, and as the y-axis points downward, a positive amount means a downward shift. Note that there is some redundancy here: a box in a horizontal list may be shifted downward by an amount s using the method just described, or by simply adding s to the depth and subtracting s from the height. In the original TEX algorithm, both methods occur. We also use both methods, depending on when exactly the shift is performed: when the subbox is constructed (modify height and depth), or when the horizontal list is formed.

3.3 Vertical Combination of Boxes

A list of boxes may also be combined vertically. In a standard vertical combination, the boxes are placed in a vertical row from top to bottom so that their left margins are aligned, and the bottom margin of each box is aligned with the top margin of its successor. There is no canonical choice for the reference point of the surrounding box. Later, we shall meet cases where it is the reference point of the topmost box, of the box at the bottom, or of some box in between. In Figure 3, we present two examples. As in the previous figure, the small gaps only exist for better visibility.

In analogy to the case of horizontal combination, there are two modifications of the standard procedure. The first is stretching / shrinking to a predefined height, which does not occur in formula layout. The second is the possibility to shift some boxes to the right by a given amount, which might be negative. In contrast to the situation in horizontal lists, the right shift cannot be replaced by some manipulation of the box dimensions.

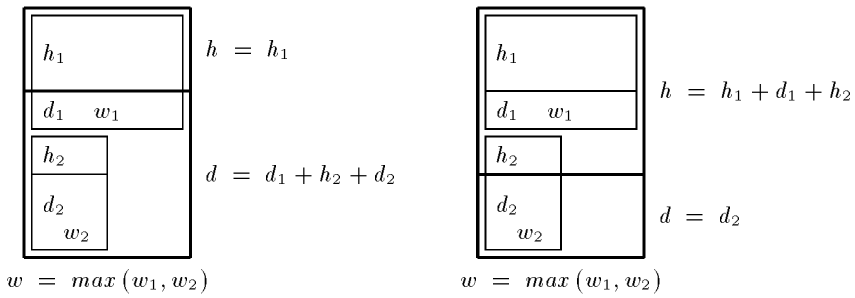

Fig. 3. Two examples for vertical combinations

3.4 Implementation of the Box Types

In the actual TEX implementation, the entities that were called boxes in the above informal description are known as nodes. There are several kinds of elementary nodes, the most prominent being character nodes, kern nodes (white space of fixed size), and glue nodes (variably sized white space). Kern nodes and glue nodes have a so-called width, which is their real width in a horizontal list, or the height in a vertical list. Compound nodes are called boxes; besides a description of their content (a list of nodes), they consist of an indicator telling whether the list is horizontal or vertical, fields containing the three dimensions, a field for the shift amount which applies to them, and some further information. Remember that the interpretation of the shift amount depends on whether the box is placed in a horizontal or a vertical list.

In our SML implementation, we basically copied this data structure. For obvious reasons, we renamed the 'width' of kern and glue nodes into size. The only more serious change is the decision to extract the shift field from a box. The logical reason is that the shift is not an intrinsic property of a box, but something that is applied to it by its context. The pragmatic reason is that the shift is usually unknown when the box is built, so that some dummy value must be assigned to the shift field in the box itself to be changed to the proper value at some later stage. While possible in principle, such a procedure is unelegant and inappropriate in a functional setting.

The data type `node` has the following cases:

`Char of fontNr * charCode`

A character node where the character is specified by its code and a font number.

`Kern of dist`

A kern, i.e. white space, of a given size.

`Glue of glueSpec`

A glue, i.e. white space of variable size. Type `glueSpec` is a record with a field `size` for the natural size of the glue, and fields that specify its stretchability and shrinkability.

`Rule of dim`

A rule, i.e. a black rectangle, with given dimensions. Type `dim` is a record type with fields height, depth, and width of type `dist`.

The dimensions of a rule in TEX may be *running*, i.e. undefined and to be determined by the context. We do not use running dimensions because they add algorithmic and logical complexity, and the dimensions of rules in formula layout are known at construction time, and cannot be changed later on.

`Penalty of penalty`

A penalty node, i.e. potential point of line breaking, with a description of the adequacy of a line break at this point.

`Box of dist * box`

A box together with the amount of shift applied to it. Type `box` is the following record type:

```
{kind: boxkind,  width: dist,  depth: dist,  height: dist,
 content: node list,  glueParam: glueParam}
```

where `boxkind` is `HBox` or `VBox`. The `glueParam` field indicates how glue nodes in the `content` node list must be modified so that the sum of the horizontal (`HBox`) or vertical (`VBox`) sizes of the components equals the given size of the box. The most important value of type `glueParam` is `natural` which means that every piece of glue gets its natural size.

In our system, the dimensions of a box are uniquely determined by its `kind`, its `content` list, and its `glueParam` field. Thus, the addition of explicit dimensions is merely a matter of efficiency. In original TEX, this is not true since Knuth often omits `Kern` nodes at the very end of the `content` list. While this saves some space, it leads to logical and algorithmic difficulties in certain situations.

Some operations apply to horizontal or vertical node lists without difference, while others are specific for one direction. To be able to express this at least in their type specification, we define

```
type hlist  =  node list;    type vlist  =  node list
```

Unfortunately, the type checker cannot enforce these distinctions, but we did not want to complicate things further by introducing constructors for the two kinds of lists. The type checker does enforce the distinction between boxes, nodes, and node lists, a difference that sometimes seems to be inappropriate.

Note that the original TEX implementation does not distinguish between nodes and node lists; type `node` contains a field `next` for chaining, and both nodes and node lists show up as entities of type 'pointer to node'.

Most rules have depth zero, most boxes are not shifted, and sometimes, we must construct a horizontal list from a single box. Hence, it proved to be useful to introduce the following abbreviations:

```
fun rule h w  =  Rule {height = h, depth = zero, width = w}
fun Box0 b  =  Box (zero, b)  (* creates node with zero shift *)
fun HL   b  =  [Box0 b]       (* creates hor. list from box *)
```

We admit that this data structure is not optimal and clumsy in parts, but it is close to the original TEX implementation, and a thorough refinement should only be done after experiences have been collected from considering more subtasks of TEX than just formula layout.

3.5 The Dimensions of Nodes and Node Lists

We need functions to compute the dimensions of nodes and node lists. Unfortunately, the dimensions of a node are context-sensitive; they depend on whether the node occurs in a horizontal or a vertical list. Thus, there are three functions, `width`, `height`, and `depth`, for nodes in a horizontal list, and four functions, `vwidth`, `vheight`, `vdepth`, and `vsize` (vheight plus vdepth), for nodes in a vertical list. All functions have the same type `node` $\rightarrow$ `dist`. Here, we only present the definition of the two width functions.

```
fun width (Char info)                   =  charWidth info
|   width (Box  (_, {width = w, ...}))  =  w
|   width (Rule {width = w, ...})       =  w
|   width (Glue {size, ...})            =  size
|   width (Kern size)                   =  size
|   width  _                            =  zero

fun vwidth (Char info)                       =  charWidth info
|   vwidth (Box  (shift, {width = w, ...}))  =  shift + w
|   vwidth (Rule {width = w, ...})           =  w
|   vwidth  _                                =  zero
```

The difference between the two functions lies in the handling of kern and glue sizes and of box shifts.

From the dimension functions for single nodes, those for node lists are derived. Three functions deal with horizontal lists, and two with vertical lists. Functions `vlistHeight` and `vlistDepth` are not needed.

```
hlistWidth,  hlistHeight,  hlistDepth:   hlist -> dist
vlistWidth,  vlistVsize:                 vlist -> dist
```

They are defined using a common auxiliary function.

```
fun compute f g nl  =  f (map g nl)
val hlistWidth   =  compute sum width
val hlistHeight  =  compute Max height
val hlistDepth   =  compute Max depth
val vlistWidth   =  compute Max vwidth
val vlistVsize   =  compute sum vsize
```

The dimensions of a node list are either the sum or the maximum of the corresponding node dimensions. Function `Max` is defined as `fold max 0`; hence, `Max []` is 0, and `Max` never yields a negative result. (This behavior is taken over from TEX.)

3.6 Basic Functions on Box Terms

In this subsection, we present several functions on box terms that are needed during formula layout.

Extension to the right. Often, a white space has to be added to the right of a given node, e.g., when the italic correction is added to a character. The following function performs the addition in an intelligent way, transforming a node to a node list.

```
val extend: dist -> node -> hlist      (* extends to the right *)
fun extend dist node  =
    let val extension  =  if dist = zero then [] else [Kern dist]
    in  node :: extension  end
```

From node lists to boxes. Next, we define an auxiliary function to produce a box with given dimensions from a node list.

```
val makebox: boxkind -> dim -> node list -> box
fun makebox boxkind  {height = h,  depth = d,  width = w}  nl  =
    {kind = boxkind,  height = h,  depth = d,  width = w,
     content = nl,  glueParam = natural}
```

The only purpose of this function is to prepare the following two definitions:

```
val hbox: dim -> hlist -> box  (* hbox with given dimensions *)
val vbox: dim -> vlist -> box  (* vbox with given dimensions *)
val hbox  =  makebox HBox;      val vbox  =  makebox VBox
```

The following function packs a horizontal list into a box with natural dimensions.

```
val hpackNat:  hlist -> box
fun hpackNat nl  =  hbox  {width  = hlistWidth  nl,
                           height = hlistHeight nl,
                           depth  = hlistDepth  nl}  nl
```

Formula layout does not use this function proper, but an optimized version: if the given node list consists of a single unshifted box, this box is returned.

```
val boxList:  hlist -> box
fun boxList [Box (0, b)]  =  b
|   boxList  nl           =  hpackNat nl
```

Centering around the axis. The *axis* was introduced in Section 2.3; it is the line where fraction strokes sit on. Sometimes, a box must be vertically centered around the axis. Assuming that the centered box will be part of a horizontal list, the centering can be performed by adding a suitable shift value.

```
val axisCenter:  style -> box -> node
fun axisCenter st box  =
let val axh  =  AxisHeight st
    val h  =  #height box   and  d  =  #depth box
    val shift  =  half (h - d)  -  axh
in  Box (shift, box)  end
```

In the beginning, the baseline of the box coincides with the overall baseline. Variable `axh` is bound to the distance from the axis to the baseline in the current style. This distance is given by style parameter `AxisHeight`. The total shift performed by this function can be considered as a shift by $s_1 = (\mathtt{h} - \mathtt{d})/2$, followed by a shift by $s_2 = -\mathtt{axh}$. The first shift results in an effective height of $\mathtt{h} - s_1 = (\mathtt{h} + \mathtt{d})/2$ and an effective depth of $\mathtt{d} + s_1 = (\mathtt{h} + \mathtt{d})/2$, i.e. yields a node that is vertically centered around the baseline. The second shift by $-\mathtt{axh}$ downward, i.e. by `axh` upward, yields a node centered around the axis as required.

Horizontal centering. Numerator and denominator of a fraction are usually centered within the horizontal extension of the fraction, e.g., $\frac{1}{n+1}$. Hence, we need a function `rebox: dist` $\rightarrow$ `box` $\rightarrow$ `box` that centers a given box within a space of given width. This is not done by simply adding white space (kerns) at both sides of the box, but by adding glue of great flexibility to both ends of the horizontal list within the box, and then creating a new box of the specified width by glue adaptation. The details of the process are complex, but it is fully implemented in our SML program. We do not include the code here, but only note that nothing is done to a box that already has the desired width:

```
fun rebox  newWidth
           (b as {kind, width, height, depth, content, ...})  =
if  newWidth = width  then  b  else  <not specified here>
```

The reason for this complexity is to allow TEX users to modify the position of numerator and denominator by means of explicit glue, as is done in $\frac{1}{n+1}$.

3.7 Making Vertical Boxes

General vertical boxes. We start with a fairly general function that creates a box out of a vertical list of nodes. The reference point of the resulting box will be the reference point of one of the nodes, which we call the reference node. This reference node is not necessarily the node at the top or at the bottom of the vertical list. This function is needed for big operators with limits (e.g., $\sum\limits_{i=1}^{n}$) where the reference node is the box with the operator symbol, and for fractions (e.g., $\frac{1}{2}$) where the reference node is the fraction stroke (a rule node).
The type of the considered function is

```
makeVBox:  dist -> node -> vlist -> vlist -> box
```

where the distance `dist` is the width of the box being built, `node` is the reference node, the first `vlist` contains the nodes above the reference node, and the second `vlist` the nodes below.

The width is added as an parameter since it is known anyway when `makeVBox` is called; thus, its recalculation as the maximum of the widths of all involved nodes is avoided. There is another design decision which seems to be odd at first glance: when `makeVBox` is called, the first `vlist` has to be enumerated from bottom to top, and the second `vlist`

from top to bottom. This causes some trouble within `makeVBox` since the first `vlist` must be reversed. There is no loss in efficiency, however, since the two vertical lists must be concatenated anyway. On the other hand, the decision to enumerate the two lists symmetrically, i.e. from the reference node toward the top and bottom edges, allows for maximum exploitation of the inherent symmetry of the formulae that are typeset using `makeVBox`. In fact, using higher order functions, usually only one piece of code is needed to compose both argument lists of `makeVBox`.
After these preliminaries, here is the definition of `makeVBox`:

```
fun makeVBox  w  node  upList  dnList  =
let val h  =  vlistVsize upList + vheight node
    val d  =  vlistVsize dnList + vdepth  node
    val nodeList  =  revAppend upList (node :: dnList)
in  vbox  {width = w,  height = h,  depth = d}  nodeList  end
```

where the auxiliary function `revAppend` is given by

```
fun revAppend    []      yl  =  yl
|   revAppend (x :: xl) yl  =  revAppend xl (x :: yl).
```

Special instances of vertical boxes. From the general function `makeVBox`, two special instances are derived where the reference node is at the top or bottom end of the complete vertical list.

```
upVBox, dnVBox:  dist -> box -> vlist -> box
```

As in `makeVBox`, the first argument is the width of the box being built. In `upVBox`, the reference node (of type `box`) is the bottom node, and the whole `vlist` goes above it. Function `dnVBox` works the other way round: the reference node is at the top, and all other nodes are placed below it. The `vlist` of `upVBox` is enumerated from bottom to top, while the `vlist` of `dnVBox` is enumerated from top to bottom.

```
fun upVBox  w  box  upList   =   makeVBox w (Box0 box) upList []
fun dnVBox  w  box  dnList   =   makeVBox w (Box0 box) [] dnList
```

There is a subtle difference between `makeVBox` and the two new functions: in `makeVBox`, the reference node is of type `node`, while it is of type `box` here. This decision was made by observing how the functions are called; `upVBox` and `dnVBox` are always called with a box, while `makeVBox` is called with a box or a rule node. It is simpler to use the transfer function `Box0` in the bodies of `upVBox` and `dnVBox`, than to repeat it in all their calls.

Putting two things above each other. Sometimes, two nodes must be placed above each other with some white space in between, for instance the two scripts in x_1^1. In this case, neither of the two nodes can be used as reference node, so that none of the functions defined above is easily usable. Thus, we define yet another function

```
above:  node -> (dist * dist) -> node -> node.
```

By a call `above` n_1 (s_1, s) n_2, node n_1 is placed above node n_2 with white space of size s in between. Parameter s_1 is the distance from the bottom edge of n_1 to the baseline of

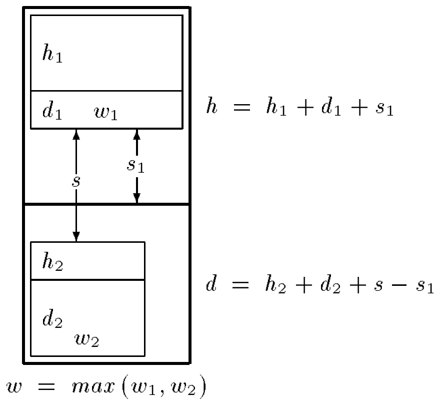

Fig. 4. Two nodes placed above each other

the combined box (see Figure 4). Pairing of the two distances simplifies the calls of this function because the two distances are returned as a pair by some other function. The result of `above` is a node since all callers expect to see a node.

```
fun above n1 (s1, s) n2  =
let val w  =  max (vwidth n1, vwidth n2)
    val h  =  vsize n1 + s1
    val d  =  vsize n2 + s - s1
    val nodeList  =  [n1, Kern s, n2]
in  Box0 (vbox {width = w, height = h, depth = d} nodeList)  end
```

4 Formula Terms

The main task of formula layout consists of translating formula terms into box terms. Below, we first present the TEX implementation of formula terms, highlight its weaknesses, and introduce our redesigned implementation.

4.1 The Original TEX-Representation

In TEX, formula terms are called *math lists*. Math lists are sequences of *math items*. According to the description in the *TEXbook* (Knuth, 1986a, page 157), a math item is an *atom*, a *horizontal space*, a *style command* (e.g., `\textstyle`), a *generalized fraction* (see Section 5.3), or some other material which we do not consider here for simplification. The description in *TEX: The Program* uses different names (Knuth, 1986b, Par. 680). There, the 'atoms' plus generalized fractions and several other entities are called *noads* (according to Knuth, this word should be pronounced as 'no-ads').

Atoms have (at least) three parts: a *nucleus*, a *superscript*, and a *subscript*. Each of these fields may be empty, a math symbol, or a math list. There are thirteen kinds of atoms, some of which have additional parts. Eight atom kinds mainly regulate the spacing between two adjacent atoms: a *relation atom* such as '=' is surrounded by some amount of space, a *binary atom* such as '+' by less space, and an *ordinary atom* such as 'x' by no extra space at all. The remaining five kinds of atoms have a more serious semantics. An *overline atom*, for example, is an overlined subformula.

This internal representation deserves some criticism. The superscript and subscript fields are empty in most cases; there should really be superscript and subscript constructors. The thirteen kinds of atoms combine two completely different aspects: a classification needed to control spacing, and the adjunction of meaningful constructors. These two aspects should not be mixed into a single concept. Interestingly, TEX's layout algorithm internally tries hard to distinguish these aspects, as we explain by two examples.

Overline atoms are handled during a first pass through the formula. After addition of the overline rule, they are transformed into 'Ord' atoms since the spacing of overline atoms and 'Ord' atoms is identical. The actual inter-atom spaces are added in a second pass through the formula.

Fractions are math items (noads), but not atoms. Their layout is computed during the first pass of the algorithm, and afterwards, they are transformed into 'Inner' atoms. The kind 'Inner' controls the spacing around fractions in the second pass of the algorithm.

Thus, the mixture of different concepts into the same notion entails destructive transformations of the formula data structure, thereby making TEX's layout algorithm hard to understand.

4.2 An Alternative Representation Defined in SML

To avoid the problems mentioned above, we completely redesigned the internal representation of formulae. Our formula terms merely reflect the logical structure of the formula. They do not contain spacing information except for characters. The spacing attributes of larger subformulae are explicitly computed during typesetting.

A formula term is an object of type `mlist = noad list`, i.e. a list of noads (we borrow this word from Knuth). Type `noad` is a datatype with as many cases as there are sorts of subformulae. There are for instance cases for atomic symbols, for overlined and underlined subformulae, for superscripted and subscripted subformulae, for operator symbols with their limits, and for generalized fractions. We do not describe all these cases here at once, but instead describe them together with their translation to box terms. We hope that this presentation method avoids redundancies in the description and enhances readability of the paper.

5 Translating Noads into Horizontal Lists

Recall that typesetting of a complete formula is done by the function

```
MListToHList:  style -> bool -> bool -> mlist -> hlist
```

where the first Boolean indicates crampedness, and the second Boolean requests the insertion of penalties (line breaking information).
Function `MListToHList` uses the function

```
NoadToHList:   style -> bool -> noad -> hlist
```

to translate the noads in its `mlist` argument into horizontal lists. Insertion of penalty nodes is handled by `MListToHList` itself. Hence, the penalty Boolean need not be passed to `NoadToHList`; the Boolean argument of this function is the crampedness.

In the course of its work, `NoadToHList` recursively calls function `MListToHList` on subformulae. All these recursive calls are done via an auxiliary function

```
val cleanBox:  style -> bool -> mlist -> box
fun cleanBox st cr ml  =
    boxList (MListToHList st cr false (* no penalties! *) ml)
```

which accounts for the fact that subformulae do not need penalty nodes, and packs the resulting horizontal list into a single box.

Below, we present some sample cases of type `noad`, and define how `NoadToHList` acts on these cases. The definition of `MListToHList` follows in Section 6. The overall format of the definition of `NoadToHList` is

```
fun NoadToHList st cr  =  fn  <list of cases>
```

so that the function name and the first two parameters need not be repeated with every case; occurrences of `st` and `cr` in the description below should be understood as being the formal parameters of this function definition.

5.1 Ordinary Symbols and Characters

In principle, there is no difference in the treatment of specific mathematical symbols such as '$\oplus$' or '∞' and ordinary characters such as 'x'. Nevertheless, there are two exceptions: Big operator symbols such as '$\sum$' are handled in a specific way (see Section 5.4), and after selection of a text font such as `\rm` or `\it`, a sequence of characters is joined together as in ordinary text. Here and in the SML implementation, we did not include the handling of text characters.

All mathematical symbols are described by a pair consisting of a character code and a font family. Within a formula term, an additional `kind` field carrying spacing information is attached to each symbol. Thus, the case of data type `noad` that corresponds to a single symbol has format

```
MathChar of kind * family * charCode
```

where `kind` is defined by

```
datatype kind  =  Ord | Op | Bin | Rel | Open |
                  Close | Punct | Inner | None.
```

Kind `Rel`, for example, is assigned to relation symbols such as '$=$' or '$<$', and kind `Open` is used for opening delimiters such as '(' and '{'. The last kind, `None`, does not occur in the original TEX implementation. We use it as the kind of those entities that do not have a kind in original TEX.

The translation of `MathChar` noads to horizontal lists ignores the kind information. It is used in a later stage of formula layout when the implicit spacing between adjacent entities is computed. Also the information about crampedness is not needed. Thus the `MathChar` case of `NoadToHList st cr` looks as follows:

```
MathChar(_, fam, ch)  =>  makeChar st fam ch
```

where auxiliary function `makeChar` is given by

```
val makeChar:  style -> family -> charCode -> hlist
fun makeChar st fam ch  =
let val (charNode, itCorr)  =  basicChar st false fam ch
in  extend itCorr charNode  end
```

A further auxiliary function `basicChar` returns a character node `charNode` and the italic correction `itCorr` of the character. By the last line of code, a white space of length `itCorr` is added to the right of the character node. Function `extend` is defined in Section 3.6.

Function `basicChar` is also called when big operators are typeset. The Boolean parameter indicates whether the given character is to be enlarged. This does not happen to ordinary `MakeChar` symbols, but may occur in the case of big operators. The function selects a concrete font from the given family, forms a character node, and returns the amount of italic correction with it. Callers of `basicChar` may then decide whether the italic correction is added to the character node or not.

```
val basicChar: style -> bool -> family -> charCode -> node * dist
fun basicChar st enlarge fam ch  =
let val fontNr  =  fontNumber st fam
    val ch'     =  if enlarge  then larger (fontNr, ch)  else ch
    val info    =  (fontNr, ch')
in  (Char info, charItalic info)  end
```

For `fontNumber` and `larger`, see Section 2.4.

5.2 Overlined and Underlined Subformulae

Subformulae may be overlined or underlined as in $\overline{x+y} - \underline{z}$. The corresponding cases of datatype `noad` are `Overline of mlist` and `Underline of mlist`. These constructors are handled by function `NoadToHList st cr` as follows:

```
Overline   ml  =>  HL (makeOver  st (cleanBox st true ml))
Underline  ml  =>  HL (makeUnder st (cleanBox st cr   ml))
```

First, subformula `ml` is typeset into a box by function `cleanBox`, using the same style `st` as the context. Underlined subformulae are cramped iff their context is cramped (parameter `cr`), while overlined subformulae are always cramped (parameter `true`). Being cramped always means being cramped from above; there is no notion of being cramped from below in TeX. The subformula box resulting from calling `cleanBox` is passed to function `makeOver` or `makeUnder`, which adds the line to the box. Both functions return a box that must be transformed into a horizontal list by `HL`.

Before we describe what `makeOver` does, let us consider an overlined subformula such as $\overline{x}$ more closely. It is a vertical combination of the box containing x with a rule node of appropriate width. The rule does not sit immediately at the top edge of x; there is vertical space in between. Finally, there is also white space above the rule that is invisible in this example, but affects the layout of a formula where an overlined subformula occurs below something else. The structure of underlined subformulae such as $\underline{x}$ is symmetrical: there is a space between x and the rule, and another space below the rule. The reference point of $\overline{x}$ and $\underline{x}$ is that of x in both cases. These are exactly the situations handled by `upVBox` and `dnVBox` (Section 3.7). Now, it pays off that their behavior is symmetrical; we may define

```
val makeOver = makeLine upVBox;  val makeUnder = makeLine dnVBox
```

using a common implementation `makeLine` for both cases. This was not done in the original TEX-program.

```
fun makeLine constrVBox st box  =
let val w  =  #width (box: box)
    val line  =  rule (RuleThickness st) w
in  constrVBox w box
          [Kern (lineDist st), line, Kern (linePad st)]   end
```

Function `RuleThickness` is a style parameter returning the default rule thickness of the current style. The first `Kern` is the one between the subformula and the rule, and the second `Kern` is that beyond the rule. Their sizes are derived from the current style by functions

```
fun lineDist st  =  3 * RuleThickness st
fun linePad  st  =      RuleThickness st
```

5.3 Generalized Fractions

A generalized fraction in TEX is, as the name indicates, a very general concept that includes ordinary fractions and binomial coefficients as special cases. Because of the generality of the concept, a generalized fraction is described by five components, together organized as a record:

```
GenFraction of genfraction
genfraction = {num: mlist,  den: mlist,  thickness: dist option,
               left: delim,  right: delim}
```

The core of a generalized fraction consists of two subformulae to be placed above each other. The upper one, the *numerator*, is contained in the `num` field, and the lower one, the *denominator*, resides in the `den` field.

The desired thickness of the fraction stroke may be explicitly specified in the `thickness` field. If the thickness is missing (`dist option`!), the default rule thickness of the current style is used. If the thickness is specified as zero, there will be no fraction stroke at all, and the rules for the placement of the numerator and the denominator are quite different from the case with a stroke.

A generalized fraction may be surrounded by delimiters, such as parentheses, that are specified in fields `left` and `right`. Delimiters may be null, i.e. non-existent; this is a special value of type `delim`. We shall say something more about delimiters later on in this subsection.

Examples for generalized fractions are ordinary fractions such as $\frac{1}{2}$ with a stroke of default thickness and null delimiters, and binomial coefficients such as $\binom{1}{2}$ without stroke, and with parentheses as delimiters.

Typesetting generalized fractions. Generalized fractions are handled by the following code:

```
GenFraction genFract  =>  HL (doGenFraction st cr genFract)
val doGenFraction: style -> bool -> genfraction -> box
fun doGenFraction st cr {left, right, thickness, num, den}  =
let val  st'      =  fract st
    val  numBox   =  cleanBox st' cr   num
    val  denBox   =  cleanBox st' true den
in  makeGenFraction st thickness left right numBox denBox  end
```

Numerator and denominator are typeset in a smaller style `st'` than the style `st` of the context. This smaller style is computed by function `fract`:

```
val fract:  style -> style
fun fract D  =  T   |   fract T  =  S   |   fract _  =  SS
```

The crampedness of the numerator is inherited from the context (`cr`), while the denominator is always cramped (`true`). The boxes resulting from numerator and denominator together with the style and the remaining components of the generalized fraction are passed to function `makeGenFraction`.

```
val makeGenFraction:
    style -> dist option -> delim -> delim -> box -> box -> box
fun makeGenFraction st thickness left right numBox denBox  =
```

The width of the central part of the generalized fraction (without the surrounding delimiters) is the maximum of the widths of the numerator and the denominator.

```
let val width    =  max (#width numBox, #width denBox)
```

Next, the numerator and the denominator are centered within this width by adding glue on both sides. See Section 3.6 for `rebox`.

```
    val numBox'  =  rebox width numBox
    val denBox'  =  rebox width denBox
```

Now, the thickness of the fraction stroke is calculated. Remember that parameter `thickness` is of type `dist option`. If no thickness is specified, the default rule thickness of the current style is used.

```
    val th  =  case  thickness  of  NONE    =>  RuleThickness st
                                  |  SOME t  =>  t
```

Next, the middle part of the generalized fraction is formed. The positioning of the components heavily depends on the existence of the fraction stroke; there is no stroke if `th` is zero.

```
    val middle  =  if th = zero
                      then  makeAtop  st numBox' denBox'
                      else  makeFract st th width numBox' denBox'
```

Function `makeAtop` is not presented in this paper; for `makeFract` see below.
Left and right delimiter nodes are constructed:

```
    val leftNode   =  makeDelimiter st left
    val rightNode  =  makeDelimiter st right
```

and finally, the delimiter nodes are placed around the middle part, and the resulting horizontal list is packed into a box:

```
in  boxList [leftNode, middle, rightNode]  end
```

Delimiters. In formula terms, delimiters are included as values of type `delim`. We do not describe this type here; it contains information so that function

```
varDelimiter:  style -> dist -> delim -> node
```

can produce delimiter symbols of given vertical size (parameter `dist`), depending on the current style. If the `delim` argument contains a null value, function `varDelimiter` returns a `Kern` node of size `nullDelimiterSpace`.

This function is used for two purposes: First, to make large delimiters around subformulae whose size depends on the size of the subformula; this specific task is not described here. Second, the function is used to make delimiters around generalized fractions. In TEX, the size of the delimiters does not depend on the size of the middle part of the generalized fraction; it merely depends on the style. Hence, we use the following function for generalized fractions:

```
val makeDelimiter: style -> delim -> node
fun makeDelimiter st del  =  varDelimiter st (Delim st) del
```

where `Delim` is a style parameter.

Proper fractions. Now, we present the layout of proper fractions, i.e. those with a fraction stroke. It is performed using function

```
makeFract:  style -> dist -> dist -> box -> box -> node.
```

In a call `makeFract st th w numBox denBox`, argument `st` is the current style, `th` is the thickness of the stroke, `w` is the width of the whole fraction and also the width of the two boxes `numBox` and `denBox`, which contain the numerator and the denominator of the fraction.

In a fraction such as $\frac{1}{2}$, the stroke does not sit on the baseline, but on the *axis* of the formula. As one can see, numerator and denominator do not touch the stroke; there is some white space between the three visible components of the fraction. The fraction is formed as a vertical box from its five components (three visible ones and two kerns) so that the reference point is at the left end of the stroke. Then, the resulting box is shifted upward so that the stroke moves from the baseline to the axis. This vertical shift can be safely attached to the fraction box since we know that it will be placed into the context of a horizontal list by function `makeGenFraction`.

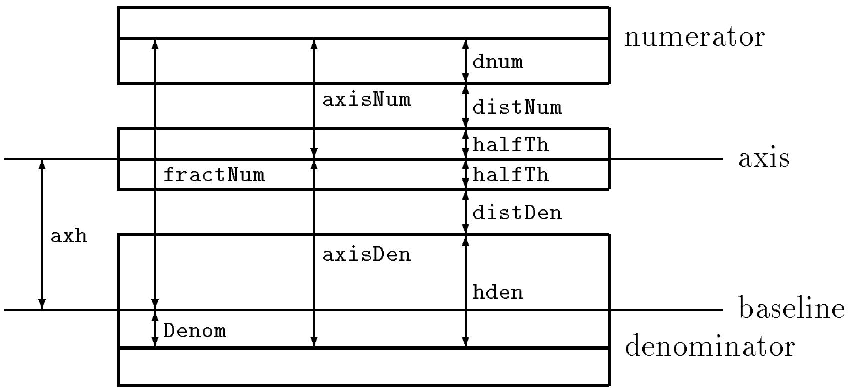

Fig. 5. Distances in a (proper) fraction

```
fun makeFract  st  th  w  numBox  denBox  =
```

To evaluate this call, we first compute half the thickness of the stroke, and the distance `axh` from the axis to the baseline which is given by style parameter `AxisHeight`.

```
let val halfTh  =  half th
    val axh     =  AxisHeight st
```

Then, a node for the stroke is formed. Height and depth are half the thickness so that later, the stroke will be vertically centered around the axis.

```
    val stroke  =  Rule {height = halfTh,
                         depth  = halfTh,  width = w}
```

Now, the distances `distNum` between the bottom edge of the numerator and the top edge of the stroke, and `distDen` between the bottom edge of the stroke and the top edge of the denominator (see Figure 5) are computed by calling function `distances` with all the relevant parameters.

```
    val (distNum, distDen)  =
         distances st axh halfTh (#depth numBox) (#height denBox)
```

The fraction box is formed from the stroke, which will be the reference node, two kerns, and the two boxes using `makeVBox` (Section 3.7). Some common work is abstracted out to the auxiliary function `makeList`.

```
    fun makeList dist box  =  [Kern dist, Box0 box]
    val fractBox  =  makeVBox w stroke (makeList distNum numBox)
                                       (makeList distDen denBox)
```

Finally, box `fractBox` is shifted from the baseline to the axis. The amount of shift is negative since the axis is above the baseline.

```
in  Box (~axh, fractBox)  end
```

Distances within a fraction. Unfortunately, the distances within a fraction are not given directly by style parameters. Instead, there is a style parameter `fractNum`, which specifies the desired distance from the baseline of the numerator to the overall baseline. By subtracting `axh`, the distance `axisNum` from the baseline of the numerator to the axis results. Subtracting the height of the stroke, which is `halfTh`, and the depth of the numerator yields the distance `distNum` from the bottom edge of the numerator to the top edge of the stroke (see Figure 5). The calculations for the denominator are similar, starting with style parameter `Denom`. Hence, the code for `distances` begins as follows:

```
fun distances st axh halfTh dnum hden  =
let val axisNum  =  fractNum st - axh
    and axisDen  =  Denom    st + axh
    val distNum  =  axisNum - halfTh - dnum
    and distDen  =  axisDen - halfTh - hden
```

The distances computed so far may be too small or even negative if the numerator is very deep or if the denominator is very high. In this case, the distances are increased to a minimum value given by `fractMinDist`.

```
    fun correct dist  =  max (dist, fractMinDist st halfTh)
in  (correct distNum,  correct distDen)  end
```

Function `fractMinDist` computes the minimum distance depending on the style and the thickness of the stroke.

```
fun fractMinDist D halfTh  =  6 * halfTh
|   fractMinDist _ halfTh  =  2 * halfTh
```

5.4 Subscripts, Superscripts, and Limits

Superscripts and subscripts. Formulae with subscripts and superscripts are, for instance, x_i, x^n, or x_i^n. They are represented by the following case of `noad`:

```
Script of script
```

where `script` is the following record type:

```
{nucleus: mlist,  supOpt: mlist option,  subOpt: mlist option}
```

The `nucleus` field contains the main formula, and the two remaining fields are the two scripts, where one of them may be missing.

Using two constructors `Sup` and `Sub` instead of `Script` is unsuitable since the case of both scripts together is typeset differently from a mere combination of the two scripts. Compare for instance the two formula specifications `{x_i}^n` and `x_i^n` which yield, respectively, ${x_i}^n$ and x_i^n. There is also a subtle difference between an empty script (specified by e.g., `x_{}`) and an entirely missing script. Hence, we must type the script fields with `mlist option`; a simple `mlist` with the empty list denoting a missing script does not suffice.

Big operators and limits. *Limits* are the scripts attached to big operators such as in `\sum_{i=1}^n`. While their textual representation in TEX's input language is identical with that of scripts, we use a different constructor for their internal representation. The reason is twofold: first, the big operator itself is handled differently from ordinary mathematical symbols, and second, the layout of limits differs from that of scripts in some cases. Scripts are always attached to the right of the nucleus, but limits may be placed above and below the operator symbol, as in $\sum\limits_{i=1}^{n}$, which is called *limit position*, or to the right of the operator, as in $\sum_{i=1}^{n}$ (*nolimit position*). The position can be influenced by writing `\sum\limits` and `\sum\nolimits`, respectively. If nothing is specified (just `\sum`), the position depends on the style; it is limit position in display style and nolimit position otherwise.
The constructor for big operators is the following:

```
BigOp  of  limits * script
```

where `script` is the record type introduced above, and `limits` is a data type of three values: `yes` means that limit position is chosen, while `no` means nolimit position, and `default` is the case where the position depends on the style.

Typesetting scripts and limits: Overview. Since the layout of limits in nolimit position coincides with that of scripts, we use the same general function `doGenScripts` for both scripts and limits. The behavior of this function depends on some Boolean flags:

```
doGenScripts:  style -> bool -> bool -> bool -> script -> hlist
```

The first Boolean is the usual crampedness flag, the second indicates whether the attachments are to be placed in limit position, and the third keeps the information whether the function was called while handling `BigOp` or `Script`.
When `Script` is handled, `doGenScripts` can be called directly:

```
Script script  =>  doGenScripts st cr false false script
```

When handling `BigOp`, we must first decide about the limit position:

```
BigOp (lim, script)  =>
   let val limits  =  (st = D  andalso  lim = default)
                      orelse  lim = yes
   in  doGenScripts st cr limits true script  end
```

Function `doGenScripts` performs the tasks common to limits and scripts:

```
fun doGenScripts st cr limits isOp {nucleus, supOpt, subOpt}  =
```

First, it calls `doNucleus` (see below) to typeset the nucleus. The result is a node `nucNode`, a flag `isChar` indicating whether this node is a character node, and the italic correction `itCorr` of this character.

```
let val (nucNode, itCorr, isChar) = doNucleus st cr isOp nucleus
```

Then, it determines the style of the attachments,

```
    val st'  =  script st
```

recursively typesets the subformulae; note that the subscripts are cramped:

```
    val supOptBox  =  optMap (cleanBox st' cr)   supOpt
    val subOptBox  =  optMap (cleanBox st' true) subOpt
    fun applyToArgs f  =  f itCorr nucNode supOptBox subOptBox
```

and finally calls `makeLimOp` in case of limits in limit position, or `makeScripts` in case of limits in nolimit position and scripts:

```
in  if  limits  then  HL (applyToArgs (makeLimOp st))
                else      applyToArgs (makeScripts st cr isChar)
end
```

The choice of the style for the attachments is done by function `script`:

```
fun script D  =  S  |  script T  =  S  |  script _  =  SS
```

Function `optMap f` maps `NONE` to `NONE`, and `SOME x` to `SOME (f x)`.

Typesetting the nucleus. The task of typesetting the nucleus is handled by

```
doNucleus:  style -> bool -> bool -> mlist -> node * dist * bool
```

where the first Boolean is crampedness, and the second indicates whether the function is called from a big operator context. There are two entirely different cases: If the nucleus is a single character, it is typeset by function `makeNucChar`, while otherwise, the usual formatting function `cleanBox` is employed.

```
fun doNucleus st _ isOp [MathChar (_, fam, ch)]
                         = makeNucChar st isOp fam ch
|   doNucleus st cr _ l = (Box0 (cleanBox st cr l), zero, false)
```

The result of `doNucleus` is a triple. The third component indicates whether the first component is a character node; this is never the case if `cleanBox` is used. The second component is the italic correction of the character in the nucleus, or zero if the nucleus is not a character.

A nucleus consisting of a single character is typeset by `makeNucChar` which slightly differs from `makeChar` that is used for characters in all other contexts. The result of `makeNucChar` is a triple of the same format as the result of `doNucleus`.

```
val makeNucChar:  style -> bool -> family -> charCode
                                      -> node * dist * bool
fun makeNucChar st isOp fam ch  =
```

A big operator in display style is enlarged.

```
let val enlarge  =  isOp  andalso  st = D
```

Function `basicChar` which we already met in Section 5.1 returns a character node and the italic correction of this character.

```
      val (charNode, itCorr)  =  basicChar st enlarge fam ch
```

Big operators are vertically centered around the axis by `axisCenter` (see Section 3.6).

```
      val nucNode = if isOp then axisCenter st (boxList [charNode])
                                 else charNode
   in  (nucNode, itCorr, not isOp)  end
```

Note that the result of `axisCenter` is not a character node any more, and thus, the third component of the result is `false` in this case. Thus, this component does *not* indicate whether the nucleus is specified as a single character, but whether it is a character node after typesetting (cf. Paragraph 18a in Appendix G of the *TEXbook* (Knuth, 1986a): "If the *translation* of the nucleus is a character box ..."). Nevertheless, the second component holds the italic correction even in the case of big operators where the third component is `false`.

Attaching the scripts. We do not include code for function `makeLimOp` in this paper, but concentrate on function

```
makeScripts:  style -> bool -> bool -> dist ->
                         node -> box option -> box option -> hlist
```

which attaches the scripts (the two `box option` arguments) to the nucleus (the `node` argument). The first Boolean is the crampedness, and the second tells whether the nucleus is a character node. The `dist` argument is the italic correction of the nucleus. The function distinguishes between four cases, depending on the existence or non-existence of the two scripts:

```
fun makeScripts st cr isChar itCorr nucNode  =
 (fn NONE        =>
    (fn NONE        => extend itCorr nucNode
     |  SOME subBox => makeSub st isChar nucNode subBox)
  |  SOME supBox =>
    (fn NONE        => makeSup st cr isChar itCorr nucNode supBox
     |  SOME subBox => makeSupSub st cr isChar itCorr
                                          nucNode supBox subBox)
  )
```

If there are no scripts at all, nothing happens but adding the italic correction to the nucleus (for `extend`, see Section 3.6). The other three cases are handled by specialized functions `makeSub`, `makeSup`, and `makeSupSub`. Let us consider these three functions together:

```
fun makeSup st cr isChar itCorr nucNode supBox  =
let val hnuc   =  height nucNode  and  dsup  =  #depth supBox
    val shift  =  SupPos st cr isChar hnuc dsup
    val scriptNode  =  Box (~shift, supBox)
in  extend itCorr nucNode  @  extendScript scriptNode  end
```

```
fun makeSub st isChar nucNode subBox  =
let val dnuc   =  depth nucNode  and  hsub  =  #height subBox
    val shift  =  SubAlonePos st isChar dnuc hsub
    val scriptNode  =  Box (shift, subBox)
in  nucNode :: (extendScript scriptNode)  end
```

```
fun makeSupSub st cr isChar itCorr nucNode supBox subBox  =
let val dnuc  =   depth nucNode  and  hnuc  =   height nucNode
    val dsup  =  #depth supBox   and  hsub  =  #height subBox
    val d     =  SupSubDistances st cr isChar hnuc dsup dnuc hsub
    val scriptNode = above (Box (itCorr, supBox)) d (Box0 subBox)
in  nucNode :: (extendScript scriptNode)  end
```

In all three cases, the result is a horizontal list essentially consisting of the nucleus node followed by a node for the script(s). In `makeSup` and `makeSub`, the script node is the script box shifted by some amount `shift`, while in `makeSupSub`, it consists of the two script boxes put above each other with some white space in between (see Section 3.7 for the definition of `above`). In all three cases, some white space of constant size `scriptSpace` is added to the right of the script node. This is done by

```
val extendScript  =  extend scriptSpace.
```

Finally, note the different usage of the italic correction: In `makeSub`, it is not used at all; the subscript is added to the uncorrected nucleus to avoid that it appears too far away from it (consider f_1). In `makeSup`, the correction is added to the nucleus lest the superscript runs into it (consider f^1). Consequently, the correction is not added to the nucleus in `makeSupSub`, but used to shift the superscript to the right (consider f_1^1).

Positioning the scripts. The positions of the scripts are computed by `SupPos`, `SubAlonePos`, and `SupSubDistances`. Let us concentrate on `SupSubDistances`. This function must return a pair of `dist` values: the distance `supDist` from the bottom of the superscript to the baseline, and the distance `Dist` from the bottom of the superscript to the top of the subscript.

```
fun SupSubDistances st cr isChar hnuc dsup dnuc hsub  =
let val supDist  =  SupPos st cr isChar hnuc dsup  -  dsup
```

Function `SupPos` is used to compute the desired value for the distance from the baseline of the superscript to the overall baseline. By subtracting the depth of the superscript, a first estimate for `supDist` results.

```
    val subDist  =  SubWithSupPos st isChar dnuc  -  hsub
    val Dist     =  supDist + subDist
```

Analogously, the distance `subDist` from the baseline to the top of the subscript is computed. The sum of `supDist` and `subDist` is a first estimate for `Dist`.

```
    val supDist' =  max (supDist,  minSupDist st)
    val Dist'    =  max (Dist,  minSupSubDist st)
in  (supDist', Dist')  end
```

If supDist and Dist are too small, they are increased to some style-dependent minimum values derived from style parameters:

```
fun minSupDist    st  =  (xHeight st * 4) div 5
fun minSupSubDist st  =  4 * RuleThickness st
```

Now we present function SupPos, which is used in makeSup and SupSubDistances. It defines the desired distance from the baseline of the superscript to the overall baseline as the maximum of three numbers:

```
fun SupPos0 st isChar hnuc  =
    if  isChar  then  zero  else  (hnuc - SupDrop (script st))
fun SupPos st cr isChar hnuc dsup  =
    Max [SupPos0 st isChar hnuc,
         Sup cr st,  dsup + (xHeight st) div 4]
```

The first number depends on the fact whether the nucleus is a single character node. If not, the number provides an upper bound SupDrop for the distance from the top of the nucleus to the baseline of the superscript. Function SupDrop is a style parameter that does not depend on the current style as usual, but on the style of the superscript.

The second number is a lower bound for SupPos given by style parameter Sup. This is the only style parameter that not only depends on the style, but also on the crampedness cr.

The third number provides a lower bound for the distance between the bottom of the superscript and the baseline of one fourth of the xHeight of the current style.

The definition of the two remaining functions SubAlonePos and SubWithSupPos is of similar complexity and not presented here.

6 Translating Formula Terms into Horizontal Lists

In the previous section, we defined function NoadToHList, which translates single noads into horizontal lists. Here, we define MListToHList, which handles complete mathematical lists. This involves applying NoadToHList to nearly all noads in the mathematical list, but there is much more than this.

As in the original TEX implementation, we split the job of MListToHList in two passes. The first pass translates formula terms (type mlist) into intermediate terms (type ilist), and the second pass proceeds by translating intermediate terms into box terms. The distribution of work among the two passes is slightly different from that in TEX. The first pass recursively handles subformulae and builds fractions, scripts, and the like. The second pass handles explicit spacing, inserts implicit spacing, e.g., around binary operators, and inserts line break points by adding penalties when requested. Because of this distribution, the cramping information is only needed in the first pass, while the information whether to add penalties is needed in the second pass only. This is a nice example of localizing information that is held in global variables in the original TEX implementation.

```
val MListToIList:  style -> bool -> mlist -> ilist
val IListToHList:  style -> bool -> ilist -> hlist
```

```
fun MListToHList st cr pen ml  =
let val il  =  MListToIList st cr  ml
    val hl  =  IListToHList st pen il
in  hl  end
```

6.1 First Pass: Mathematical Lists into Intermediate Form

Function `MListToIList` applies `NoadToHList` to nearly all noads of the mathematical list. There are a few exceptions; four kinds of noads which we did not yet present are handled by `MListToIList` directly or by the second pass `IListToHList`. The corresponding cases of data type `noad` are:

```
|  MPen   of  penalty
|  MSpace of  mathSpace
|  Style  of  style
|  Choice of  style -> mlist
```

Cases `MPen` and `MSpace` provide explicit penalties and spaces (kern or glue), respectively. A noad `Style` s results from an explicit style command such as `\textstyle`. It has effect from the place where it occurs until the end of the current subformula. Noads `Choice` f represent *four way choices*, i.e. lists of four subformulae, from which one is selected according to the current style.

Intermediate lists `ilist` are lists of entities of type `item`. Data type `item` contains representations for explicit penalties and spaces, which are handled in the second pass; for style commands, which have to be known in both passes; and for the horizontal lists, which result from translating noads by function `NoadToHList`. These lists must not yet be concatenated since they form units that will be separated by implicit spaces in the second pass. With every list, information about the kind of its origin is needed. Hence, type `item` is defined as follows:

```
datatype  item  =
  INoad   of  kind * hlist
| IPen    of  penalty
| ISpace  of  mathSpace
| IStyle  of  style
```

After defining the relevant types, we present the code for function `MListToIList`:

```
fun MListToIList st cr  =
fn []  =>  []
|  MPen   p     :: rest => IPen   p   :: MListToIList st  cr rest
|  MSpace s     :: rest => ISpace s   :: MListToIList st  cr rest
|  Style  st'   :: rest => IStyle st' :: MListToIList st' cr rest
|  Choice chfun :: rest => MListToIList st cr (chfun st  @  rest)
|  noad         :: rest =>
                       INoad (noadKind noad, NoadToHList st cr noad)
                       :: MListToIList st cr rest
```

Essentially, it is one run through the argument list. Penalty and space noads are copied into the intermediate form. Style commands take effect on the rest of the argument list, and are copied into the result to take effect again in the second pass. Four way choices are replaced by the appropriate case. Note that the result of a choice is not grouped into a subformula, but spliced into the argument list without grouping. (This is why it could not be handled by `NoadToHList`.) Ordinary noads are translated into horizontal lists, which are still grouped together. The kind of the original noad is kept in mind for the introduction of implicit spaces in the second pass. Function `noadKind` is defined as follows:

```
val noadKind  =
fn MathChar (k, _, _)  =>  k
|  Overline      _  =>  Ord
|  Underline     _  =>  Ord
|  GenFraction   _  =>  Inner
|  Script {nucleus, ...}  =>  nucKind nucleus
|  BigOp     _   =>  Op
|  <several more cases>

fun nucKind [MathChar (k, _, _)]  =  k
|   nucKind  ml                   =  Ord
```

6.2 The Second Pass: Spacing and Penalties

In the second pass through a formula, function

```
IListToHList:  style -> bool -> ilist -> hlist
```

translates an intermediate term into a single horizontal list. It must insert penalty nodes if the Boolean argument is true, handle explicit penalties and spaces, and introduce implicit spaces between adjacent entities. Implicit spaces are the horizontal spaces in formulae such as $x = y$ or $x - y$ in contrast to $x@y$ or $-y$. They are not specified in the formula input; the four formulae above are entered as `x=y`, `x-y`, `x@y`, and `-y`, respectively.

The implicit space between two entities basically depends on their kinds. There is an additional non-local dependency for binary operator symbols; in $x - y$ as opposed to $-y$, the space between '$-$' and 'y' depends on the presence of 'x'. In any case, we need the kind of the previous entity when `IListToHList` processes some entity in the intermediate term. Hence, we define

```
fun IListToHList st insertPenalty iList  =
let fun trans st prevKind  =  fn  <see below>
in  trans st None iList  end.
```

Function `trans` runs through the `iList`, keeping in mind the kind `prevKind` of the previous entity. In the beginning, `prevKind` is `None`.

Style commands, explicit penalties and spaces. The first four cases of `trans st prevKind` are

```
fn []  =>  []
|  IStyle st' :: il =>                        trans st' prevKind il
|  IPen  pen  :: il => Penalty pen      ::  trans st  prevKind il
|  ISpace sp  :: il => makeSpace st sp   @  trans st  prevKind il
```

Style commands influence the style used for the rest of the input. Explicit penalties are put into the result without any change except for the constructor; this is a must in a functional language like SML. Explicit spaces (kern or glue) are processed by function

```
makeSpace:  style -> mathSpace -> hlist
```

and the resulting `hlist`, which is an empty list or a singleton, is spliced into the result. Note that explicit penalties and spaces are transparent for kinds; they have no kind and do not influence `prevKind`.

An explicit space (kern or glue) in a formula may be conditional or unconditional. Conditional spaces are suppressed in the styles `S` and `SS`, while unconditional spaces show up in all styles. The size of the spaces may be given in absolute units such as `pt`, or in *mathematical units* `mu`. The size of the latter depends on the style. Hence, function `makeSpace`, whose code we do not present here, must perform two tasks: the suppression of conditional spaces in small styles, and the conversion from mathematical units into an absolute size.

Handling translated noads. The last case of function `trans` handles already translated noads and inserts implicit spaces before them, and implicit penalties behind them. Remember that the following code is situated in a context where `prevKind` is the kind of the previous entity, `st` is the current style, and `insertPenalty` is a Boolean telling whether implicit penalties are needed.

```
| INoad (actKind, hList) :: il  =>
  let val newKind  =  changeKind  prevKind  actKind  il
      val spaceList  =  makeSpaceOpt st
                            (mathSpacing (prevKind, newKind))
      val penaltyList  =  mathPenalty  insertPenalty  newKind  il
  in  spaceList @ hList @ penaltyList @  trans st newKind il  end
```

In the next three paragraphs, we discuss the three `let` bindings in the code above. The chain of append operations in the last line is reasonably efficient since the first three operands usually are extremely short lists.

The kind of binary operators. The call of function `changeKind` deals with the non-local dependencies in the computation of implicit space. If `actKind` is `Bin` and the current item occurs in a non-binary context, then `actKind` is changed into `newKind` = `Ord`; otherwise, `newKind` equals `actKind`. Non-binary contexts can be detected by inspecting `prevKind` and the kind of the first `INoad`-item in `rest`. In $-y$, for instance, '$-$' occurs in a non-binary context and is assigned kind `Ord`, while its kind remains `Bin` in $x - y$. Thus, `changeKind` is defined as follows:

```
val changeKind:  kind -> kind -> ilist -> kind
```

```
fun changeKind  prevKind  Bin  rest  =
      if  checkPrev prevKind  orelse  checkNext (listKind rest)
      then  Ord  else  Bin
|   changeKind  _  k  _   =   k
```

Functions `checkPrev` and `checkNext` check whether their argument is in a certain list of kinds:

```
val checkPrev  =  contains [Bin, Op, Rel, Open, Punct, None]
val checkNext  =  contains [Rel, Close, Punct, None]
```

Function `listKind` returns the kind of the first `INoad` item in its argument list:

```
val rec listKind  =
fn  []                  =>   None
|   INoad (k, _) :: _   =>   k
|   _            :: t   =>   listKind t
```

Implicit spacing. The implicit space between two adjacent `INoad` items of given kind is computed by function

```
mathSpacing:  kind * kind -> mathSpace option.
```

This function implements the table in (Knuth, 1986a, page 170) that specifies the amount of space to be inserted between two adjacent items of given kind. Some combinations of kinds are not separated by additional space. To handle this case efficiently, the result type of function `mathSpacing` is `mathSpace option`.

Within function `trans`, the result of `mathSpacing` is further processed in the same manner as explicit spaces using function `makeSpace`:

```
fun makeSpaceOpt st  =  fn NONE     =>  []
                        |  SOME sp  =>  makeSpace st sp
```

Implicit penalties. Inline formulae contain implicit penalties allowing for line breaking. They are computed by function

```
mathPenalty:  bool -> kind -> ilist -> hlist
```

the result of which is either an empty list or a singleton containing a penalty node. If the Boolean argument is false, there are no penalties at all. Otherwise, penalties are added behind items of `kind Bin` or `Rel`, but only if the following `ilist` is not empty and does not start with a relation operator or an explicit penalty item.

7 Conclusion

Our formula layout system has been implemented in the language Standard ML of New Jersey, version 109.2. This implementation is far more comprehensive than the algorithms described in this paper. The core system, i.e. function `MListToHList` and all its auxiliary functions, completely covers the typesetting of all kinds of formulae except for math accents (e.g., $\dot{x}$), roots (e.g., $\sqrt{2} + \sqrt[3]{3}$), the construction of big delimiters from small pieces, and ordinary text occurring in formulae. The addition of these features is planned for the future.

Besides the core system, we have implemented the preprocessing phase, which provides style parameters and font information such as character dimensions. There are also functions which translate box terms into `.dvi` code, and construct a `.dvi` file of correct global format. Thus, the results of our algorithm can be considered on the screen or printed.

The system is still evolving. The current version is available in directory **`formulae`** of the ftp server **`ftp.cs.uni-sb.de`** of the University of the Saarland.

The motivation for this reimplementation effort originated from an attempt to understand and teach TEX's formula layout algorithm. The primary source, the *TEXbook* (Knuth, 1986a), did not provide sufficient clues about the method despite long and desperate attempts to understand it. Its description reflects the structure of the program with very complex control flow and destructive manipulations of global, ill designed data structures. This data structure, the internal representation of mathematical formulae, combines several orthogonal properties in an unintelligible way using some fields in parallel and consecutively for different purposes. It occurred that a formulation in a functional language clarified the method and enabled us to effectively teach about this subject.

The resulting SML program will be extended to cover the full set of formulae. Due to the modular structure it can be included in new contexts in need of a module for formula layout.

Some more specific properties of our implementation are as follows:

- Defining an adequate data type of formula terms separates concerns, i.e. spacing aspects from structure aspects. This is of great help for a better understanding of the algorithm.
- Though not optimal, the data structure of boxes and nodes represents a reasonable compromise between the original TEX types and the needs of a functional language.
- Using a functional description language forced us to transform the updatable global variables of Knuth's description into explicit function parameters. On the one hand, this adds complexity to the description, but on the other hand, the flow of information becomes visible: it can be seen where information comes from, where it is updated, and where it is used. Thus, it becomes apparent which subtasks depend on others, and which are independent from each other.
- The functional programming style discourages changing or removing entities that have already been constructed. This in particular concerns the white space carrying the italic correction, which is not built by our system before it is definitely known that it is required.
- Some constructs, e.g., limits in limit position, were not treated here for space reasons. They do not offer fundamentally new problems, and are included in the actual implementation.
- Some postprocessing parts of the algorithm look somewhat 'imperative'. These are those where some subformulae are positioned independently of each other only to detect afterwards, that certain minimal distances between them are not satisfied (see for instance function **`distances`** used for fractions, and function

`SupSubDistances` for superscripts and subscripts occurring together). It would be nice to have a declarative manner for stating such constraints, with automatic means for building a formula from its subformulae, or more generally, a two-dimensional diagram from its components.

Acknowledgements

We appreciated the detailed and constructive reviews we received. These have contributed to a greatly improved presentation. Our thanks go to the reviewers.

References

Knuth, D. E. 1986a. *The TEXbook.* Addison Wesley.

Knuth, D. E. 1986b. *TEX: The Program.* Addison Wesley.

Paulson, L. C. 1991. *ML for the Working Programmer.* Cambridge University Press.

Wilhelm, R. and Heckmann, R. 1996. *Grundlagen der Dokumentenverarbeitung.* Addison Wesley.