Formula Layout

R. Heckmann *R. Wilhelm*

Technischer Bericht Nr. A 07 / 95

Reinhold Heckmann, Reinhard Wilhelm

FB 14 – Informatik,
Universität des Saarlandes
Postfach 151150
D-66041 Saarbrücken
Germany

e-mail: {`heckmann,wilhelm`}`@cs.uni-sb.de`

Formula Layout

Reinhold Heckmann *Reinhard Wilhelm*

Fachbereich Informatik, Universität des Saarlandes
Saarbrücken, Germany

{heckmann,wilhelm}@cs.uni-sb.de

July 3, 1995

Abstract

Both the quality of the results of TEX's formula layout algorithm and the complexity of its description in the TEXbook [1] are hard to beat. The algorithm is (verbally) described as an imperative program with very complex control flow and complicated manipulations of the data structures representing formulae. In a forthcoming textbook [3], we describe TEX's formula layout algorithm as a functional program transforming *mlist*-terms into *box*-terms. This transformation is given in this paper.

1 Introduction

The quality of the results of TEX's formula layout algorithm are convincing. However, any attempt to understand the reasons for that leads to deep frustration when Knuth's description of the algorithm from the TEXbook [1] is used. In an attempt to understand this problem, one has to cleanly separate the reasons for the lack of understandability.

1. The problem may have a nature that does not allow for a solution which is easily described in some readable way. Not much can be done about that.
2. The algorithm used to solve the problem may not be the simplest possible, but may be tuned for efficiency or optimality of the result. Here, a clean separation between principles of a space of solutions and the optimizations applied would help the interested reader.
3. The context of the chosen algorithm may enforce a bad design. Here, a new describer may take the freedom to abstract from this context.
4. The description may not be the best possible for the given algorithm. This is a particularly favorable situation for an attempt to explain an interesting subject better.

Knuth's description of formula layout is an imperative program with very complex control flow and complicated manipulations of the data structures representing formulae. The 'programming language' is English prose with some formal fragments. In this paper, we present a new description using the pure functional language Miranda[1] [2]. The use of a functional language gives a completely new flavor to the description. Of course, the mere fact that we use a concrete programming language instead of English phrases adds rigor and exactness to the exposition.

Our criticism and attempt to improve the presentation of the formula layout algorithm of TEX mainly touches points 2 – 4 above. In Section 2, we consider the input of the layout algorithm, i.e., the internal representation of formulae. In 2.1, we discuss Knuth's original data structure. In our

[1] Miranda is a trademark of Research Software Ltd.

opinion, it is misconceived. Many difficulties in Knuth's description result from the design of this data structure. In 2.2, we propose a new data structure for formulae with a clean and simple design.

In Section 3 we present some more details which influence formula layout: the styles of formulae and subformulae (which communicate information about their context), the representation of characters, and the layout parameters which control the positions of subformulae. Knuth's account of these things is very concrete. In contrast, we present an abstract interface which hides the details of font table organization, and makes clear how the information is used.

In Section 4, we consider the output of the layout algorithm, *box terms*. Knuth's description of this data structure and its operations is particularly vague. We try to model Knuth's intentions by a Miranda data type and functions defined in Miranda.

In Section 5, we present a bunch of specialized functions which translate subformulae of various kinds into box terms. In Section 6, we deal with the translation of whole formulae, i.e., the recursive descent to subformulae and the selection of the appropriate specialized subformula functions.

We give an honest estimation of the improvements 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 have to distinguish many cases and pay attention to lots of little details.

2 Internal Representation of Formulae

2.1 The Original TEX-Representation

TEX reads a formula specification from the input and converts it into an internal representation, a *math list*. A math list is a sequence of *math items*.

According to the description in the TEXbook [1, page 157], a math item is an *atom*, a *horizontal space*, a *style command* (e.g., `\textstyle`), a *generalized fraction*, or some other material which we do not consider here for simplification.

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 with 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 instance is an overlined subformula.

The formula $(x_i + y)^{\overline{n+1}}$ for instance may be specified as `$(x_i+y)^{\overline{n+1}}$`. In internal form, it is represented by a math list consisting of five atoms: an 'Open'-atom with nucleus '(' (and empty superscript and subscript); an 'Ord'-atom with nucleus 'x', empty superscript, and subscript 'i'; a 'Bin'-atom with nucleus '$+$'; an 'Ord'-atom with nucleus 'y'; and finally a 'Close'-atom with nucleus ')', whose superscript is a math list consisting of a single 'Over'-atom, whose nucleus is a math list of three atoms corresponding to $n + 1$.

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. The overline rule is added to the corresponding subformula, and afterwards, it is transformed into an 'Ord' atom 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, 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, we see that the mixture of different concepts into the same notion leads to the need to destructively transform the data structure of formulae which makes TEX's layout algorithm hard to understand.

2.2 An Alternative Representation Defined in Miranda

To avoid the problems mentioned above, we completely redesigned the internal representation of formulae. The following definition is given in Miranda.

As in the original representation, formulae are math lists (`mlist`) consisting of math items (`mitem`).

```
mlist  ==  [mitem]
```

Mitems are defined as the elements of a constructor type. We do not distinguish between atoms and non-atoms, and restrict ourselves to semantically meaningful constructors.

```
mitem ::=
   Sym class mathchar       |  || a single symbol with its class
   MathSpace num            |  || space (in relative math units)
   Over   mlist             |  || overlined subformula
   Under  mlist             |  || underlined subformula
   Frac   mlist mlist       |  || fraction with numerator and denominator
   Sup    mitem mlist       |  || formula with superscript
   Sub    mitem       mlist |  || formula with subscript
   SupSub mitem mlist mlist |  || with superscript and subscript
   Class_cmd class mlist    |  || from \mathord{ }, \mathop{ } etc
   Style_cmd style          |  || from \displaystyle etc
   Group  mlist                || a nested group, indicated by {...}
```

For reasons of simplicity, we omitted some of TEX's possibilities. To cover the full power of TEX formulae, additional constructors would be needed for left and right big delimiters, for accented characters, for roots ($\sqrt{3}+\sqrt[3]{2}$), for vertically centered subformulae, etc. They don't offer principally new problems, although the treatment of accented characters and roots in [1] is particularly hard to grasp. The constructor `Frac` represents a special case of TEX's generalized fractions; for a full treatment, more argument fields would be needed.

To complete our description, we have to define the types `class`, `mathchar`, and `style`. The type `mathchar` is defined in Section 3.2, and `style` in Section 3.1. The type `class` enumerates nine constructors:

```
class ::= Ord | Op | Bin | Rel | Open | Close | Punct | Inner | None
```

The first eight classes correspond to those atom kinds which control spacing. The ninth class `None` is used for mitems which are not atoms in the original TEX representation, and are never transformed into atoms.

Using our representation, the formula `(x_i+y)^{\overline{n+1}}` is internally described as the following term:

```
[ Sym Open '(',
  Sub  (Sym Ord 'x')    [Sym Ord 'i'],
  Sym Bin '+',
  Sym Ord 'y',
```

```
  Sup  (Sym Close ')')  [Over [Sym Ord 'n', Sym Bin '+', Sym Ord '1']]
]
```

The internal representation is created by a parser starting from the external formula description. We have to assume that this parser is a bit more powerful than the one employed in the TEXbook. It has to correctly transform the input string into our data structure obeying the subformula structure. Note that the nucleus of `Sup` etc. needs grouping if it is not a single symbol. (The nucleus is an mitem instead of an mlist, since otherwise, class computation and spacing would fail.)

When reading a character or mathematical symbol, the parser knows about the pre-assigned class of this symbol, e.g., `Rel` for '=' and `Open` for '('. This class is stored in the internal representation as the first argument of the `Sym` constructor. As the biggest difference to the original TEX-situation, we assume that the parser is able to recognize binary symbols which are used in non-binary contexts, e.g., the plus symbol in f^+. The class of these symbols should be `Ord` instead of `Bin`. In the original TEX-algorithm, atoms of kind 'Bin' change their kind into 'Ord' depending on the kinds of neighboring atoms during the computation of inter-atom spaces. This solution could also be programmed in Miranda, but would make function `do_mlist` in Section 6.3 overly complex.

In contrast to the original description, classes are not stored with all mitems. The reason is that in almost all cases, the class of an mitem can be derived mechanically from its structure. The only exceptions are symbols which are classified by some external declarations, and the `Class_cmd` items which come from explicit class assertions in the formula description (by the commands `\mathord`, `\mathop` etc.).

The following function computes the class of every subformula:

```
class_ :: mitem -> class
class_ (Sym cl mc)          =  cl     || the class is a symbol property
class_ (MathSpace w)        =  None   || spaces are not atoms, and never will be
class_ (Over  ml)           =  Ord    || Over-atoms are changed into Ord-atoms
class_ (Under ml)           =  Ord    || Under-atoms are changed into Ord-atoms
class_ (Frac  num den)      =  Inner  || fractions become Inner-atoms
class_ (Sup    mi sup)      =  class_ mi
class_ (Sub    mi sub)      =  class_ mi
class_ (SupSub mi sup sub)  =  class_ mi
class_ (Class_cmd cl ml)    =  cl     || class is explicitly set
class_ (Style_cmd st)       =  None   || style commands are not atoms
class_ (Group  ml)          =  Ord    || this is an Ord-atom in TeX
```

In the comments, we tried to explain the reason for this rule. For instance, `Over`-items are classified as `Ord` because they are transformed into 'Ord'-atoms in the course of TEX's layout algorithm.

3 Additional Details

In this section, we present some additional detail information needed for the formula layout: the styles of formulae and subformulae, the representation of characters, and the layout parameters controlling the positions of subformulae.

3.1 Formula Styles

The layout of formulae and subformulae in TEX documents depends on a *style* parameter. There are two kinds of basic styles: formulae may appear on a separate line by their own (display style) or as part of a line of text (text style). Consider the following displayed formula

$$\sum_{j=1}^{n} A^{i^j} + \frac{y^2}{y^2+z^2}$$

and its inline counterpart $\sum_{j=1}^{n} A^{i^j} + \frac{y^2}{y^2+z^2}$. We observe that in display style, the sum symbol is bigger, and the limits of the summation are placed vertically below and above it (this is called *limit position*). In text style however, the position of the limits is to the right of the symbol. All superscripts are set in styles with smaller characters and spaces. The same is true for the constituents of the fraction in text style. Notice also how the position of superscripts depends on their context, i.e., on the style of the corresponding subformula. In the denominator, their position is lower than in the numerator.

In the TEXbook [1], there are eight styles altogether: display style D, text style T, script style S, script-script style SS, and four 'cramped' styles D', T', S', and SS'. In cramped styles, which are used for denominators, superscripts are placed in a lower position than in the corresponding uncramped styles. In analyzing the usage of these styles, it turned out that they may be regarded as pairs of a main style and a Boolean value 'cramped'. The two components of the pairs are independently calculated and used, so that it is easy to separate them completely. This is done in our Miranda program. Hence, we have only four styles:

```
style  ::=  D | T | S | SS
```

Function `script` computes the style for subscripts and superscripts from the current style, and `fract` calculates the styles of numerators and denominators.

```
script, fract  ::  style -> style
script D = S;   script T = S;   script S = SS;   script SS = SS
fract  D = T;   fract  T = S;   fract  S = SS;   fract  SS = SS
```

3.2 Math Characters and Output Characters

Characters from a formula description do not yet completely determine the characters which appear in the printed document. The formula description `x^x` for instance yields the printed formula x^x, where the two occurrences of x appear in different sizes. The reason is that the first x is set in text style `T`, whereas for the second one, script style `S` is used.

In our description, we model this behavior by using two different types of characters and a style-dependent transfer function. For characters in the internal representation, whose appearance is not yet determined, we use type `mathchar`, whereas type `outchar` is used for characters in the result of the formula layout. The transfer function is `setchar :: style -> mathchar -> outchar`. Although the two types and `setchar` could be specified further following the hints in the TEXbook, we refrain from doing it since a complete definition would be difficult and hardly interesting.

For formula layout, we need some information about the size and form of characters. The *height* of a character is the distance from its top end to the base line; e.g., 'a' and 'g' have the same height, and 'f' has a bigger one. The *depth* is the distance from the base line to the bottom end; e.g., 'a' has depth 0, whereas 'g' has non-zero depth. The *width* is the horizontal size, and the *slant* gives information how far the character is slanted to the right.

These character informations are given by the four functions `char_height`, `char_depth`, `char_width`, and `char_slant`, all with type `outchar -> dim`, where `dim` is the type of dimensions, i.e., amounts of length, measured in basic units. We may simply assume `dim == num`. The four functions are left unspecified here; in practice, their values are read off from the appropriate font tables.

3.3 Layout Parameters

The exact layout of a formula depends on some *layout parameters*. They control the position of superscripts, the distance between numerator and fraction stroke, the thickness of the stroke, etc.

In the TEXbook, the layout parameters are attached to the fonts used to make formulae. Since the choice of the font depends on the style, we incorporate the layout parameters as functions of type `style -> dim`. The function names are (abbreviations of) the symbolical names given in the table in [1, page 447].

Height of 'x' in current font:	`x_height`
Width of 'M' in current font:	`quad`
Parameters for numerators:	`num1 num2`
for denominators:	`denom1 denom2`
for superscripts:	`sup_drop sup1 sup2 sup3`
for subscripts:	`sub_drop sub1 sub2`
for limits at large operators:	`big_op1` through `big_op5`
Default thickness of rules:	`rule_thickness`
Distance from 'axis' to base line:	`axis_height`

The axis is the line where fraction strokes sit on. Consider e.g., $x + \frac{y}{z}$. The base line is at the bottom end of the 'x' and the '+'.

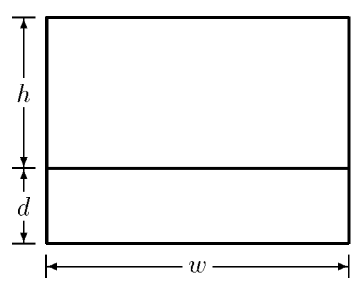
Some font parameters are used in special contexts only. This is realised by three auxiliary functions.

```
num_level, den_level  ::  style -> dim
num_level D  =  num1 D;     num_level st =  num2 st
den_level D  =  denom1 D;   den_level st =  denom2 st
```

```
sup_level :: style -> bool -> dim
sup_level D  False  =  sup1  D      || Display style, not cramped
sup_level st True   =  sup3 st      || all cramped styles
sup_level st  cr    =  sup2 st      || style T, S, or SS;  not cramped
```

In addition to the style-dependent layout parameters, there is a constant `scriptspace` of type `dim`.

4 The Target Representation: Box Terms



During formula layout, an input term of type `mlist` is translated into a term of type `box`. Boxes are rectangles whose edges are parallel to the page edges. Compound boxes are built from smaller boxes, and atomic boxes contain symbols or are filled with black. Each box has a horizontal *base line*. It has the *reference point* of the box at its left end. Boxes have *heights*, h, *depths*, d, and *widths*, w. These dimensions may be negative. This is the case for boxes which are shifted upwards or downwards beyond their base line and for boxes which represent negative distances.

In the TEXbook, boxes and their properties are described verbally. At first glance, the size attributes of a compound box seem to be totally determined by the sizes of its constituents. Later however, it seems as if the size dimensions of a box may be arbitrarily changed. For, the description of the formula layout contains phrases such as "increase the depth of the box by", "add ... to the width of the box", or "construct a box with depth ... and height ...".

Here, we represent boxes as a Miranda data type. The operations on boxes are formalized. The size dimensions of our boxes are determined by their structure. We tried to catch the intended meaning of the size manipulations in [1] by adding space boxes without visible content.

```
box  ::=  HSpace dim         | || horizontal space with width
          VSpace dim dim     | || vertical space with height and depth
          Rule   dim dim dim | || black box with height, depth, and width
          Chr    outchar     | || character box
          HBox   [box]       | || horizontal list of boxes
          Vdn    [box]       | || vertical list of boxes, downward
          Vup    [box]         || vertical list of boxes, upward
```

Their are four kinds of atomic boxes and three kinds of compound boxes. An `HBox` is the horizontal concatenation of a list of boxes, ordered from left to right. The boxes are concatenated such that their base lines become adjacent. The reference point of an `HBox` is the one of its leftmost constituent. An `HBox` may be empty.

Both `Vdn` and `Vup` boxes represent vertical concatenations of boxes. In both cases, the concatenation is done so that the reference points of the constituent boxes are vertically aligned. In `Vdn` boxes, the constituents are ordered from top to bottom. The reference point of a `Vdn` box is the reference point of its topmost component. In contrast, the components of a `Vup` list are ordered from bottom to top. The reference point of a `Vup` box is the one of its lowest component. Thus, in both cases, the reference point of the compound box is the one of the head of its list of components. Both `Vdn` and `Vup` lists should never be empty.

The Dimensions of a Box

Height, depth, and width of a box are uniquely defined from its structure. We call the sum of height and depth `vsize`.

```
height, depth, width, vsize  ::  box -> dim
vsize box  =  height box + depth box

height (HSpace w) = 0;    height (VSpace h d) = h;    height (Rule h d w) = h
height (Chr  ch)    =  char_height ch
height (HBox boxl)  =  max0 (map height boxl)     || as max, but  max0 [] = 0
height (Vdn (top : rest))  =  height top
height (Vup (bot : rest))  =  height bot + sum (map vsize rest)

depth (HSpace w) = 0;     depth (VSpace h d) = d;     depth (Rule h d w) = d
depth (Chr  ch)     =  char_depth ch
depth (HBox boxl)   =  max0 (map depth boxl)
depth (Vdn (top : rest))  =  depth top + sum (map vsize rest)
depth (Vup (bot : rest))  =  depth bot

width (HSpace w) = w;     width (VSpace h d) = 0;     width (Rule h d w) = w
width (Chr  ch)     =  char_width ch
width (HBox boxl)   =  sum (map width boxl)
width (Vdn  boxl)   =  max (map width boxl)
width (Vup  boxl)   =  max (map width boxl)
```

For the sake of efficiency, all three dimensions could be stored at `HBox`, `Vdn`, and `Vup` constructors, in order to avoid costly recomputations (memoization).

Some Operations on Boxes

`hconc` concatenates two boxes to form an `HBox`. If one of them is an `HBox` already, nesting of `HBox`es is avoided.

```
hconc :: box -> box -> box
hconc (HBox boxl1) (HBox boxl2) = HBox (boxl1 ++ boxl2) || list concatenation
hconc       box1   (HBox boxl2) = HBox (box1  :  boxl2)
hconc (HBox boxl1)       box2   = HBox (boxl1 ++ [box2])
hconc       box1         box2   = HBox [box1  ,   box2]
```

`right` moves a box to the right by putting an `HSpace` box in front of it.

```
right :: dim -> box -> box
right 0 box  =  box
right l box  =  (HSpace l)  $hconc  box     || $hconc = hconc as infix operator
```

`center` centers a given box inside a space of given width. It uses `right`.

```
center :: dim -> box -> box;      center w box  =  right ((w - width box)/2) box
```

`center` is only called with `w` $\geq$ `width box`. It does not matter that there is no `HSpace` to the right of the box since centered boxes are placed in vertical lists where widths are maximized.

The next operation extends a box to the right ("increases its width").

```
extend :: dim -> box -> box
extend 0 box  =  box;             extend l box  =  box  $hconc  (HSpace l)
```

A box is raised by increasing its height and decreasing its depth; the `vsize` does not change. This is done by vertically adjoining an empty box of `vsize` 0, but non-zero height and depth (one of these must be negative).

```
raise  ::  dim -> box -> box
raise 0 box  =  box
raise l box  =  Vup [VSpace (l - d) (d - l),  box]  where  d  =  depth box
```

To verify `raise`, show the two equations

```
  height (raise l box) = height box + l      depth (raise l box) = depth box - l.
```

Instead of `Vup`, `Vdn` could be used equally well (with a different argument).

Finally, we define an operation `vlist` which takes three arguments: a box B, a list of boxes in upward order which goes above B, and a list of boxes in downward order which goes below B. The reference point of the whole thing is that of B.

```
vlist :: box -> [box] -> [box]    -> box
vlist    box    up_list  dn_list  =  Vdn ( Vup (box : up_list) : dn_list )
```

`vlist` could equally well be specified the other way round: … `Vup (Vdn (box:dn_list) : up_list)`.

5 Setting of Subformulae

In the sequel, we show how subformulae of the various kinds are translated into box terms. Later, we combine these functions to a function that computes the layout of arbitrary mitems.

5.1 Symbols and Spaces

Symbols (mathchars) are transformed into character boxes by choosing the appropriate output character (function `setchar` of Section 3.2) and putting it into a box (`Chr`) which is vertically centered around the axis in some cases (`vcenter`). The result is not only the box, but also the slant ('italic correction') of the produced character. This information is needed later.

```
set_sym  ::  style -> class -> mathchar -> (box, dim)
set_sym  st  cl  mc  =  (vcenter (Chr ch), char_slant ch),  if  cl = Op
                     =  (         Chr ch , char_slant ch),  otherwise
                        where  ch  =  setchar st mc

vcenter :: style -> box -> box
vcenter st box  =  raise (axis_height st - (height box - depth box)/2) box
```

When spaces are set, their size has to be transformed from style-dependent mathematical units into an absolute dimension.

```
set_space :: style -> num -> box;     set_space st ml  =  HSpace (ml * quad st / 18)
```

5.2 Setting Overlined and Underlined Subformulae

We assume that the subformula is already translated into a box. The thickness of the line, `th`, depends on the style. Between the line and the formula, there is a gap of size 3`th`, and above the overline / below the underline, there is white space of size `th`. Since the reference point of the whole thing should be that of the subformula, we use `Vup` for overlines and `Vdn` for underlines. In both cases, the list of constituent boxes starts with the subformula box, followed by the distance to the line, the line itself, and the white space beyond it.

```
set_over, set_under  ::  style -> box -> box
set_over  st  box  =  Vup  [box, VSpace (3 * th) 0, Rule th 0 w, VSpace th 0]
                      where  w = width box;   th  =  rule_thickness st
set_under st  box  =  Vdn  [box, VSpace (3 * th) 0, Rule th 0 w, VSpace th 0]
                      where  w = width box;   th  =  rule_thickness st
```

5.3 Setting of Fractions

Numerator and denominator are already given as boxes. The desired vertical position of the numerator is given by `num_level` relative to the base line. However, the fraction stroke will be positioned at the axis, an invisible line somewhere above the base line. Thus, we compute the position `num_pos` of the reference point of the numerator relative to the axis. From this, the actual distance `num_dist` between the bottom edge of the numerator and the top edge of the stroke is calculated. There is a style-dependent minimal distance `min_dist`. If `num_dist` is too small, it is increased up to `min_dist`. The denominator is handled analogously. Next, both numerator and denominator are centered to the maximum of their width. Then, we form a vertical list whose reference point is at the middle of the fraction stroke using `vlist`, and finally raise the resulting box to the level of the axis.

```
set_frac :: style -> box -> box -> box
set_frac    st       num    den    =
   raise  ax  fracbox
   where ax  =  axis_height st;                  th  =  rule_thickness st
         num_pos   =  num_level st - ax;          den_pos  =  den_level st + ax
         num_dist  =  num_pos - depth  num - th/2
         den_dist  =  den_pos - height den - th/2
         min_dist  =  3 * th,  if  st = D
                   =      th,  otherwise
         num_dist' =  num_dist  $max2  min_dist    || maximum as infix operator
         den_dist' =  den_dist  $max2  min_dist
         w  =  (width num)  $max2  (width den)
         num_list  =  [VSpace num_dist' 0,  center w num]
         den_list  =  [VSpace den_dist' 0,  center w den]
         fracbox   =  vlist  (Rule (th/2) (th/2) w)  num_list  den_list
```

5.4 Superscripts and Subscripts in Limit Position

The following functions deal with superscripts and subscripts in the limit position, i.e., vertically above and below the nucleus as in $\sum\limits_{i=1}^{\infty}$. (In the nolimit position, they are to the right of the nucleus.) Function `lim_sup` deals with the case of superscripts only, `lim_sub` is called if there are only subscripts, and `lim_supsub` is for joint superscript / subscript combinations. We assume that nucleus, superscript, and subscript are already given as boxes.

In limit position, superscripts are placed above the nucleus in some distance, and white space is added above them. Superscript and nucleus are first centered to their maximum width. Afterwards, the superscript is shifted to the right by some amount `shift` which depends on the slant of the nucleus. This is visible in e.g., $\int\limits_1^1$. Subscripts are handled symmetrically.

To partially reduce the three functions to two, we use two auxiliary functions `mksup` and `mksub` which transform superscripts and subscripts into a list of boxes. For superscripts, the list is ordered upward, and for subscripts downward.

```
mksup :: style -> dim -> box -> [box]
mksup    st       shift  sup  =
  [VSpace dist 0, right shift sup, VSpace space 0]      || upward list
   where  dist  =  (big_op1 st) $max2 (big_op3 st - depth sup)
          space =  big_op5  st

mksub :: style -> dim -> box -> [box]
mksub    st       shift  sub  =
  [VSpace dist 0, right (-shift) sub, VSpace space 0]   || downward list
   where  dist  =  (big_op2 st) $max2 (big_op4 st - height sub)
          space =  big_op5  st
```

In the actual functions, the appropriate auxiliary functions are called and their results are vertically combined.

```
lim_sup :: style -> dim -> box -> box -> box
lim_sup    st       shift  nuc    sup    =
   Vup  (nuc' : sup_list)
   where  w  =  (width sup) $max2 (width nuc)
          sup'  =  center w sup;   nuc'  =  center w nuc
          sup_list  =  mksup st shift sup'

lim_sub :: style -> dim -> box -> box -> box
lim_sub    st       shift  nuc    sub    =
   Vdn  (nuc' : sub_list)
   where  w  =  (width nuc) $max2 (width sub)
          nuc'  =  center w nuc;   sub'  =  center w sub
          sub_list  =  mksub st shift sub'

lim_supsub :: style -> dim -> box -> box -> box -> box
lim_supsub    st       shift  nuc    sup    sub    =
   vlist  nuc'  sup_list  sub_list
   where  w  =  (width sup) $max2 (width nuc) $max2 (width sub)
          sup' = center w sup;  nuc' = center w nuc;  sub' = center w sub
          sup_list  =  mksup st shift sup'
          sub_list  =  mksub st shift sub'
```

5.5 Superscripts and Subscripts in Nolimit Position

Here, the superscripts and subscripts are put to the right of the nucleus as in $\sum_{i=1}^{\infty}$. Their exact position depends on the fact whether the nucleus is a "character box, possibly followed by a kern". This information is passed as a Boolean to the functions `nolim_sup`, `nolim_sub`, and `nolim_supsub`. The third function has an additional argument: the slant of the nucleus, which is used to move the superscript to the right. This is visible in e.g., P_2^2. The functions involving superscripts need the information whether the style is 'cramped'.

There are several auxiliary functions to compute the positions of superscripts and subscripts. For a (partial) motivation for the formulae appearing in these functions, we refer to the TeXbook [1].

```
sup_position :: style -> bool  -> bool  -> dim -> dim -> dim
sup_position    st       cramped  is_char  hnuc   dsup   =
    sup_pos   $max2   sup_level st cramped              || hnuc = height nuc
              $max2   dsup + abs (x_height st)/4        || dsup = depth  sup
    where  sup_pos  =  0,                              if  is_char
                    =  hnuc + sup_drop (script st),  otherwise
```

```
sub_position0 :: style -> bool -> dim -> dim
sub_position0    st       True    dnuc    =   0          || dnuc = depth nuc
sub_position0    st       False   dnuc    =   dnuc + sub_drop (script st)
```

```
sub_position1 :: style -> bool  -> dim -> dim -> dim
sub_position1    st       is_char  dnuc   hsub   =       || hsub = height sub
    sub_position0 st is_char dnuc
        $max2   sub1 st
        $max2   hsub - 4 * abs (x_height st) / 5
```

```
sub_position2 :: style -> bool  -> dim -> dim
sub_position2    st       is_char  dnuc   =
    sub_position0 st is_char dnuc   $max2   sub2 st
```

The cases where there are only superscripts or only subscripts are relatively simple. Note that every 'script' is extended to the right by `scriptspace`.

```
nolim_sup :: style -> bool ->  bool ->  box -> box -> box
nolim_sup    st       cramped  is_char  nuc    sup    =
   nuc  $hconc  (raise  sup_pos  sup')
   where  dsup  =  depth sup;   hnuc  =  height nuc
          sup_pos  =  sup_position  st  cramped  is_char  hnuc  dsup
          sup'     =  extend  scriptspace  sup
```

```
nolim_sub :: style -> bool ->  box -> box -> box
nolim_sub    st       is_char  nuc    sub    =
   nuc  $hconc  (raise  (-sub_pos)  sub')
   where  dnuc  =  depth nuc;   hsub  =  height sub
          sub_pos  =  sub_position1  st  is_char  dnuc  hsub
          sub'     =  extend  scriptspace  sub
```

The case of a joint superscript / subscript combination is much more difficult. First, the desired position `sup_pos` of the superscript is computed. It is the distance from the reference point of the superscript to the base line. From it, the distance `sup_dist` between the bottom edge of the superscript and the base line is derived. The subscript is handled analogously. There is a minimum value `min_sup` for `sup_dist`, and a minimum value `min_dist` for the total distance `sup_dist + sub_dist` between superscript and subscript. There are correction values `corr_sup` and `corr_dist` if these minimum values are not reached. The correction of `sup_dist` is done by raising both superscript and subscript, i.e., adding `corr_dist` to `sup_dist` and subtracting it from `sub_dist`. The correction of `sup_dist + sub_dist` is done by lowering the subscript, i.e., adding `corr_dist` to `sub_dist`.

```
nolim_supsub :: style -> bool ->  bool ->  dim  -> box -> box -> box -> box
nolim_supsub    st       cramped  is_char  slant   nuc    sup    sub    =
   nuc  $hconc  sup_sub
   where  dsup  =  depth sup;                  hsub  =  height sub
          hnuc  =  height nuc;                 dnuc  =  depth nuc
          sup_pos   =  sup_position   st  cramped  is_char  hnuc  dsup
          sub_pos   =  sub_position2  st  is_char  dnuc
          sup_dist  =  sup_pos - dsup;         sub_dist  =  sub_pos - hsub
```

```
min_dist  =  4 * rule_thickness st
corr_dist =  correction  (sup_dist + sub_dist)  min_dist
min_sup   =  4 * abs (x_height st) / 5
corr_sup  =  correction  sup_dist  min_sup
sup_dist' =  sup_dist + corr_sup
sub_dist' =  sub_dist + corr_dist - corr_sup
sup'      =  right slant (extend  scriptspace  sup)
sub'      =              extend  scriptspace  sub
sup_sub   =  vlist (VSpace sup_dist' sub_dist') [sup'] [sub']
```

The amount of the necessary correction values is computed by the following function:

```
correction :: num -> num -> num
correction  value  min_value  =  min_value - value,  if  value < min_value
                              =  0,                  otherwise
```

The definitions of `sup_dist'` and `sub_dist'` can be algebraically simplified to

```
sup_dist'  =  sup_dist  $max2  min_sup
sub_dist'  =  ((sup_dist + sub_dist)  $max2  min_dist) - sup_dist'
```

After that, `corr_sup`, `corr_dist`, and the function `correction` are no longer needed. We did not directly introduce the simplified definitions since they are hard to explain by themselves.

6 From Subformulae to Whole Formulae

6.1 Some Auxiliary Functions

In some cases, white space is appended to a symbol to compensate for its slant (italic correction).

```
it_corr :: (box, dim) -> box;         it_corr (box, slant)  =  extend slant box
```

The function `lim_` computes whether superscripts and subscripts are placed in limit position.

```
lim_  ::  style -> class -> bool
lim_  D  Op  =  True;                 lim_  st  cl  =  False
```

Actually, this is a bit simplified since it only realizes TEX's default rule. In full TEX, there are commands `\limits` and `\nolimits` which may be appended to large operators. The default rule is only applied if none of these commands is issued.

6.2 Translation of Math Items

Math items are translated by the function `set_mitem`. For the nucleus of formulae with superscripts or subscripts, we need a special version of `set_mitem`, called `set_nuc`, which not only returns a box, but also the information whether its argument was a single character, and if so, its slant (the value needed for italic correction).

```
set_nuc :: style -> bool -> mitem -> ((box, dim), bool)
set_nuc st cr (Sym cl mc)  =  ( set_sym st cl mc,          True )
set_nuc st cr  mitem       =  ((set_mitem st cr mitem, 0), False)
```

Function `set_mitem` deals with the various cases of mitems. It handles the recursive setting of subformulae and then passes control to specialized functions. Its Boolean parameter is the bit indicating cramped styles. Notice that denominators, overlined formulae, and subscripts are always cramped. Other subformulae inherit the cramp status of their context.

```
set_mitem  ::  style -> bool -> mitem -> box
set_mitem st cr (Sym cl mc)     =  it_corr (set_sym st cl mc)
set_mitem st cr (MathSpace mw)  =  set_space st mw
set_mitem st cr (Over  ml)      =  set_over  st (set_mlist st True ml)
set_mitem st cr (Under ml)      =  set_under st (set_mlist st  cr  ml)
set_mitem st cr (Frac num den)
   =  set_frac st numbox denbox
      where  st' =  fract st;      numbox  =  set_mlist st'  cr  num
                                   denbox  =  set_mlist st' True den
set_mitem st cr (Sup nuc sup)
   =    lim_sup  st  (slant/2)    boxnuc'  boxsup,  if lim
   =  nolim_sup  st  cr  is_char  boxnuc'  boxsup,  otherwise
      where  lim      =  lim_  st  (class_ nuc)
             ((boxnuc, slant), is_char)  =  set_nuc  st  cr  nuc
             boxnuc'  =  it_corr (boxnuc, slant)
             boxsup   =  set_mlist  (script st)  cr  sup
set_mitem st cr (Sub nuc sub)
   =    lim_sub  st  (slant/2)  boxnuc'  boxsub,  if lim
   =  nolim_sub  st  is_char    boxnuc   boxsub,  otherwise
      where  lim      =  lim_  st  (class_ nuc)
             ((boxnuc, slant), is_char)  =  set_nuc  st  cr  nuc
             boxnuc'  =  it_corr (boxnuc, slant)
             boxsub   =  set_mlist  (script st)  True  sub
set_mitem st cr (SupSub nuc sup sub)
   =    lim_supsub  st  (slant/2)          boxnuc'  boxsup  boxsub,  if lim
   =  nolim_supsub  st  cr  is_char  slant  boxnuc   boxsup  boxsub,  otherwise
      where  lim      =  lim_  st  (class_ nuc)
             st'      =  script st
             ((boxnuc, slant), is_char)  =  set_nuc  st  cr  nuc
             boxnuc'  =  it_corr (boxnuc, slant)
             boxsup   =  set_mlist  st'   cr   sup
             boxsub   =  set_mlist  st'  True  sub
set_mitem st cr (Class_cmd cl ml)  =  set_mlist st cr ml
set_mitem st cr (Group  ml)        =  set_mlist st cr ml
```

The case of the `Style_cmd` constructor is missing since it is handled by `set_list` presented below.

6.3 Translation of Math Lists

In the TEXbook, there is a second pass during formula layout where appropriate spaces are inserted between adjacent atoms, ignoring any non-atoms in between.

Here, insertion of inter-atom spaces is done by the function `set_mlist` which translates math lists into boxes. It does its job by calling an auxiliary function `do_mlist` with an additional class argument. This argument remembers the class of the previously set item, ignoring items of class `None`. At the beginning of the math list, the remembered class is `None`.

```
set_mlist :: style -> bool -> mlist -> box
set_mlist    st       cr      ml    =  do_mlist  st  cr  None  ml
```

Function `do_mlist` handles style commands, inserts inter-atom spaces (`set_space`), and calls `set_mitem` to translate items into boxes.

```
do_mlist  :: style -> bool -> class -> mlist -> box
```

```
do_mlist st cr old  []                   =  HBox []
do_mlist st cr old  (Style_cmd st' : ml)  =  do_mlist  st'  cr  old  ml
do_mlist st cr old  (mi : ml)  =
   boxmi  $hconc  rest
   where  boxmi  =  set_mitem  st  cr  mi;     new  =  class_  mi
          rest   =  do_mlist  st  cr  old  ml,               if  new  =  None
                 =  set_space st (space st old new)  $hconc
                    do_mlist  st  cr  new  ml,               otherwise
```

The auxiliary function `space :: style -> class -> class -> num` computes the space between two 'atoms' according to the table in the TEXbook [1, page 170]. The output is assumed to be in relative mathematical units. It depends on the style, and the conversion to an absolute dimension by `set_space` is again style dependent.

6.4 Setting of Whole Formulae

Function `set_display` handles displayed formulae, and `set_inline` formulae within text lines.

```
set_display, set_inline   ::   mlist -> box
set_display  ml  =  set_mlist D False ml   || Display style, not cramped
set_inline   ml  =  set_mlist T False ml   || Text style, not cramped
```

Actually, the difference between these two functions should be bigger. In TEX, displayed formulae require some postprocessing for positioning, and potential line breaks are computed within inline formulae.

7 Conclusion

Let us summarize what we achieved by our description.

- Defining an adequate data type separates concerns, i.e., spacing aspects from structure aspects. This is of great help for a better understanding of the algorithm.
- Using a powerful parser instead of a macro expansion mechanism avoids some postprocessing of the input on the data structure representing math formulae. Hence, we can translate mlists to box terms in a single pass, whereas Knuth needs two passes.
- Defining the result data type (box terms) makes many aspects of the algorithm explicit, which are implicit or at most verbally described in Knuth's description.
- 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.
- Some constructs, e.g., roots, were not treated here for space reasons. They don't offer principally new problems, although their treatment in [1] is particularly hard to grasp.
- Some postprocessing parts of the algorithm look somewhat 'imperative'. These are those, where some subformulae are set independently of each other only to detect afterwards, that certain minimal distances between them are not satisfied (`set_frac` and `nolim_supsub`). Miranda is not the best language to describe this, but it is possible as you can see.
- In our presentation, we preferred clarity over efficiency. The main problem is the computation of the size attributes `height`, `depth`, and `width` of box terms which is repeated many terms. The attributes should already be computed when box terms are constructed, and stored at

their constructors so that e.g., `HBox [box]` becomes `HBox dim dim dim [box]`. The `class_` attribute of mitems should be handled similarly. The necessary program transformations are not difficult, but afterwards, it is no longer obvious that the size attributes of a compound box are fully determined by its constituent boxes.

References

[1] D.E. Knuth. *The TEXbook*. Addison Wesley, 1986.

[2] D. Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.

[3] R. Wilhelm and R. Heckmann. *Dokumentenverarbeitung*. Addison Wesley, 1996.