The **polytable** package

Andres Löh

`polytable@andres-loeh.de`

2003/12/26

Abstract

This package implements a variant of tabular-like environments where columns can be given a name and entries can flexibly be placed between arbitrary columns. Complex alignment-based layouts, for example for program code, are possible.

1 Introduction

This package implements a variant of tabular-like environments. We will call these environments the `poly`-environments to distinguish them from the standard ones as provided by the LaTeX kernel or the `array` package.

Other than in standard tables, each column has a name. For instance, the commands
```
\column{foo}{l}
\column{bar}{r}
```
– when used within a `poly`-environment – define a column with name `foo` that is left-aligned, and a column with name `bar` that is right-aligned.

Once a couple of columns have been defined, the text is specified in a series of `\fromto` commands. Instead of specifying text per column in order, separating columns with `&`, we give the name of the column where the content should start, and the name of the column before which the content should stop. To typeset the text "I'm aligned!" in the column `foo`, we could thus use the command
```
\fromto{foo}{bar}{I'm aligned}
```
Several `\fromto`-commands can be used to typeset a complete line of the table. A new line can be started with `\nextline`.

The strength of this approach is that it implicitly handles cases where different lines have different alignment properties. Not all column names have to occur in all lines.

2 A complete example

Figure 1 is an example that is designed to show the capabilities of this package. In particular, it is *not* supposed to look beautiful.

left	first of three	second of three	third of three		right
left		middle 1/2		middle 2/2	right
left	middle 1/3	middle 2/3	middle 3/3		right
left	first of two middle columns		second of two middle columns		right

Figure 1: Example table

The example table consists of four lines. All lines have some text on the left and on the right, but the middle part follows two different patterns: the first and the third line have three middle columnss that should be aligned, the second and the fourth line have two (right-aligned) middle columns that should be aligned, but otherwise independent of the three middle columns in the other lines.

Vertical bars are used to clarify where one column ends and the next column starts in a particular line. Note that the first and the third line are completely aligned. Likewise, the second and the fourth line are. However, the fact that the bar after the text "middle 1/2" ends up between the two bars delimiting the column with "second of three" in it is just determined by the length of the text "first of two middle columns" in the last line. This text fragment is wider than the first of the three middle columns, but not wider than the first two of the three middle columns.

Let's have a look at the input for the example table:

```
\begin{ptabular}
\column{left}{|l|}
\column{right}{l|}
\column{m13}{l|}
\column{m23}{l|}
\column{m33}{l|}
\column{m12}{r|}
\column{m22}{r|}
\column{end}{l}
\fromto{left}{m13}{left}
\fromto{m13}{m23}{first of three}
\fromto{m23}{m33}{second of three}
\fromto{m33}{right}{third of three}
\fromto{right}{end}{right}
\nextline
\fromto{left}{m12}{left}
\fromto{m12}{m22}{middle 1/2}
\fromto{m22}{right}{middle 2/2}
\fromto{right}{end}{right}
\nextline
\fromto{left}{m13}{left}
\fromto{m13}{m23}{middle 1/3}
\fromto{m23}{m33}{middle 2/3}
\fromto{m33}{right}{middle 3/3}
\fromto{right}{end}{right}
\nextline
```

```
\fromto{left}{m12}{left}
\fromto{m12}{m22}{first of two middle columns}
\fromto{m22}{right}{second of two middle columns}
\fromto{right}{end}{right}
\end{ptabular}
```

First, columns are declared, including the vertical lines. Note that there is a final column `end` being declared that is only used as the end column in the `\fromto` statements. A future version of this package will probably get rid of the need to define such a column. After the column definitions, the lines are typeset by a series of `\fromto` commands, separated by `\nextline`. Note that the first and third column do not use `m12`, `m22`. Similarly, the second and fourth column do not use `m13`, `m23`, and `m33`.

So far, one could achieve the same with an ordinary `table` environment. The table would have 6 columns. One left and right, the other four for the middle: the first and third line would use the first of the four columns, then place the second entry in a `\multicolumn` of length 2, and then use the fourth column for the third entry. Likewise, the other lines would place both their entries in a `\multicolumn` of length 2. In fact, this procedure is very similar to the way the `ptabular` environment is implemented.

The problem is, though, that we need the information that the first of the two middle columns ends somewhere in the middle of the second of the three columns, as observed above. If we slightly modify the texts to be displayed in the middle columns, this situation changes. Figure 2 shows two variants of the example table. The input is the same, only that the texts contained in some columns have slightly changed. As you can see, the separator between the first and second middle column in the second and fourth lines of the tables now once ends up within the first, once within the third of the three middle columns of the other lines.

left	first of three	second of three	third of three	right
left	middle 1/2		middle 2/2	right
left	middle 1/3	middle 2/3	middle 3/3	right
left	first of two		second of two	right

left	first of three	second of three	third of three	right
left		middle 1/2	middle 2/2	right
left	middle 1/3	middle 2/3	middle 3/3	right
left	the first of two middle columns		2/2	right

Figure 2: Variants of the example table

If one wants the general case using the `\multicolumn` approach, one thus has to measure the widths of the entries of the columns to compute their relative position. In essence, this is what the package does for you.

```haskell
class (Eq a) ⇒ Ord a where
    compare                    ::  a → a → Ordering
    (<), (≤), (≥), (>)         ::  a → a → Bool
    max, min                   ::  a → a → Bool

    — Minimal complete definition: (≤) or compare
    — using compare can be more efficient for complex types
    compare x y  |  x ≡ y       = EQ
                 |  x ≤ y       = LT
                 |  otherwise  = GT
    x ≤ y                       =  compare x y ≢ GT
    x < y                       =  compare x y ≡ LT
    x ≥ y                       =  compare x y ≢ LT
    x > y                       =  compare x y ≡ GT
    max x y      |  x ≤ y       = y
                 |  otherwise  = x
    min x y      |  x ≤ y       = x
                 |  otherwise  = y
```

Figure 3: Haskell code example

3   Haskell code example

I have written this package mainly for one purpose: to be able to beautifully align Haskell source code. Haskell is a functional programming language where definitions are often grouped into several declarations. I've seen programmers exhibit symmetric structures in different lines by adding spaces in their source code files in such a way that corresponding parts in different definitions line up. On the other hand, as Haskell allows user-defined infix operators, some programmers like their symbols to be typeset as LATEX symbols, not as typewriter code. But using LATEX symbols and a beautiful proportional font usually destroys the carefully crafted layout and alignment.

With lhs2TEX, there is now a preprocessor available that preserves the source code's internal alignment by mapping the output onto polytable's environments. Figure 3 is an example of how the output of lhs2TEX might look like.

Of course, this could be useful for other programming languages as well. In fact, lhs2TEX can be tweaked to process several experimental languages that are based on Haskell, but I can imagine that this package could generally prove useful to typeset program code.

4 Other applications

Although I have written this package for a specific purpose, I am very much interested to hear of other potential application areas. Please tell me if you found a use for this package and do not hesitate to ask for additional features that could convince you to use the package for something.

5 The lazylist package

Internally, this package makes use of Alan Jeffrey's excellent lazylist package, which provides an implementation of the lambda calculus using fully expandable control sequences. Unfortunately, `lazylist.sty` is not included in most common TeX distributions, so you might need to fetch it from CTAN separately.

6 Reference

6.1 The environments

ptabular
parray
pboxed

There are currently three environments that this package provides: `ptabular` and `parray` are based on (and translated into) the usual `tabular` and `array` environments as provided by the `array` package. In particular, `parray` assumes math mode, whereas `ptabular` assumes text mode. The third environment, `pboxed`, typesets the material in boxes of the calculated length, but in normal paragraph mode. The advantage is that there can be page breaks within the table. Note that you should start a new, nonindented paragraph before beginning a `pboxed`. All lines in a `pboxed` should be of equal length, so it might be possible to center or right-align the material, although this has not been extensively tested.

One more environment is planned: `plongtable`, a poly-version of the `longtable` environment.

The interface is the same for all of the environments.

6.2 The commands

In each of the environments, the following commands can be used (and *only* these commands should be used):

\column

With `\column{⟨columnid⟩}{⟨spec⟩}`, a new column ⟨columnid⟩ is specified. The name of the column can be any sequence of alphanumerical characters. The ⟨spec⟩ is a format string for that particular column, and it can contain the same constructs that can be used in format strings of normal tables or arrays (this also holds for the `pboxed` environment). However, it should only contain the description for *one* column. (I've never tested what happens if you do something else, but you have been warned ...)

If the save/restore feature (explained below) is not used, `\column` definitions are always local to one table. One can define a column multiple times within one

table. A warning will be produced, and the second format string will be used for the complete table.

\fromto   The call `\fromto{⟨fromid⟩}{⟨toid⟩}{⟨text⟩}` will typeset ⟨text⟩ in the current line, starting at column ⟨fromid⟩ and ending before column ⟨toid⟩, using the format string specified for ⟨fromid⟩.

A line of a table usually consists of multiple `\fromto` statements. Each statement's starting column should be either the same as the end column of the previous statement, or it will be assumed that the start column is located somewhere to the right of the previous end column. The user is responsible to not introduce cycles in the (partial) order of columns. If such a cycle is specified, the current algorithm will loop, causing a `dimension too large` error ultimately. TODO: catch this error.

\nextline   The command `\nextline` ends one line and begins the next. There is no need to end the last line. One can pass an optional argument, as in `\nextline[⟨dimen⟩]`, that will add ⟨dimen⟩ extra space between the lines. TODO: make this command available as `\\`.

6.3   A warning

The contents of the table are processed multiple times because the widths of the entries are measured. Global assigments that modify registers and similar things can thus result in unexpected behaviour. New in v0.7: LaTeX counters (i.e. counters defined by `\newcounter`, are protected now. They will be reset after each of the trial runs.

6.4   Saving column width information

WARNING: this feature does *only* work correctly with the `pboxed` environment right now. TODO: make this work with the other environments (this essentially amounts to implementing a `tabbing`-like `\kill` statement for `tabular` and `array`; does that already exist somewhere?).

Sometimes, one might want to reuse not only the same column, but exactly the same alignment as in a previous table. An example would be a fragment of program code, which has been broken into several pieces, with documentation paragraphs added in between.

\savecolumns
\restorecolumns   With `\savecolumns[⟨setid⟩]`, one can save the information of the current table for later reuse. The name `setid` can be an arbitrary sequence of alphanumeric characters. It does *not* share the same namespace as the column names. The argument is optional; if it is omitted, a default name is assumed. Later, one can restore the information (multiple times, if needed) in other tables, by issuing a `\restorecolumns[⟨setid⟩]`.

This feature requires to pass information backwards in the general case, as column widths in later environments using one specific column set might influence the layout of earlier environments. Therefore, information is written into the `.aux` file, and sometimes, a warning is given that a rerun is needed. Multiple reruns might be required to get all the widths right.

I have tried very hard to avoid producing rerun warnings infinitely except if there are really cyclic dependencies between columns. Still, if it happens or something seems to be broken, it often is a good idea to remove the `.aux` file and start over. Be sure to report it as a bug, though.

Figure 4 is an example of the Haskell code example with several comments inserted. The source of this file shows how to typeset the example.

7   The Code

```
1 ⟨*package⟩
2 \NeedsTeXFormat{LaTeX2e}
3 \ProvidesPackage{polytable}%
4   [2003/12/26 v0.7.1 'polytable' package (Andres Loeh)]
5 \RequirePackage{lazylist}
6 \RequirePackage{array}
```

The option debug will cause (a considerable amount of) debugging output to be printed. The option silent, on the other hand, will prevent certain warnings from being printed.

```
7 \DeclareOption{debug}{\AtEndOfPackage\PT@debug}
8 \DeclareOption{silent}{\AtEndOfPackage\PT@silent}
9 \ProcessOptions
```

First, we declare a couple of registers that we will need later.

```
10 \newdimen\PT@colwidth
11 %\newdimen\PT@delta
12 \newcount\PT@cols
13 \newcount\PT@table
14 \newif\ifPT@changed
```

In `\PT@allcols`, we will store the list of all columns, as a list as provided by the lazylist package. We initialise it to the empty list, which is represented by `\Nil`. New in v0.8, we have a second list that only contains the public columns.

```
15 \def\PT@allcols{\Nil}
16 \def\PT@allpubliccols{\Nil}
17 \let\PT@infromto\empty
```

These are flags and truth values. TODO: Reduce and simplify.

```
18 \let\PT@currentwidths\empty
19 \def\PT@false{0}
20 \def\PT@true{1}
21 \let\PT@inrestore\PT@false
```

The dimension `\PT@delta` is currently not used. The dimension comparisons should probably have a small tolerance, to prevent infinite loops due to rounding errors. (Can this really happen?)

```
22 %\PT@delta\hfuzz
```

`\PT@debug`
`\PT@typeout@`
`\PT@silent`
`\PT@warning`

Similar to the tabularx package, we add macros to print debugging information to the log. Depending on package options, we can set or unset them.

```
23 \def\PT@debug
```

We introduce a new type class Ord for objects that admit an ordering. It is based on the Eq class:

class (Eq a) $\Rightarrow$ Ord a **where**

The next three lines give the type signatures for all the methods of the class.

$$
\begin{array}{lll}
compare & :: & a \to a \to \textsf{Ordering} \\
(<), (\leq), (\geq), (>) & :: & a \to a \to \textsf{Bool} \\
max, min & :: & a \to a \to \textsf{Bool}
\end{array}
$$

— Minimal complete definition: $(\leq)$ or *compare*
— using *compare* can be more efficient for complex types

As the comment above says, it is sufficient to define either $(\leq)$ or *compare* to get a complete instance. All of the class methods have default definitions. First, we can define *compare* in terms of $(\leq)$. The result type of *compare* is an Ordering, a type consisting of only three values: EQ for "equality", LT for "less than", and GT for "greater than".

$$
\begin{array}{llll}
compare \; x \; y & | \; x \equiv y & = & \textsf{EQ} \\
& | \; x \leq y & = & \textsf{LT} \\
& | \; otherwise & = & \textsf{GT}
\end{array}
$$

All the other comparison operators can be defined in terms of compare:

$$
\begin{array}{lll}
x \leq y & = & compare \; x \; y \not\equiv \textsf{GT} \\
x < y & = & compare \; x \; y \equiv \textsf{LT} \\
x \geq y & = & compare \; x \; y \not\equiv \textsf{LT} \\
x > y & = & compare \; x \; y \equiv \textsf{GT}
\end{array}
$$

Finally, there are default definitions for *max* and *min* in terms of $(\leq)$.

$$
\begin{array}{llll}
max \; x \; y & | \; x \leq y & = & y \\
& | \; otherwise & = & x \\
min \; x \; y & | \; x \leq y & = & x \\
& | \; otherwise & = & y
\end{array}
$$

Figure 4: Commented Haskell code example

```
24    {\def\PT@typeout@ ##1{\typeout{(polytable) ##1}}}
25 \let\PT@typeout@\@gobble
26 \def\PT@warning{\PackageWarning{polytable}}%
27 \def\PT@silent
28    {\let\PT@typeout@\@gobble\let\PT@warning\@gobble}
```

\PT@rerun This macro can be called at a position where we know that we have to rerun LaTeX
to get the column widths right. It issues a warning at the end of the document.

```
29 \def\PT@rerun
30    {\PT@typeout@{We have to rerun LaTeX ...}%
31     \AtEndDocument
32       {\PackageWarning{polytable}%
33          {Column widths have changed. Rerun LaTeX.\@gobbletwo}}%
34     \global\let\PT@rerun\relax}
```

7.1 Macro definition tools

\PT@listopmacro This assumes that #2 is a list macro and #3 is a new list element. The macro
\PT@consmacro #2 should, after the call, expand to the list with the new element #1ed. Because
\PT@appendmacro we don't know the number of tokens in #3, we use a temporary macro \PT@temp
(which is used frequently throughout the package).

```
35 \def\PT@listopmacro #1#2#3% #1 #3 to the list #2
36    {\def\PT@temp{#1{#3}}%
37     \expandafter\expandafter\expandafter
38       \def\expandafter\expandafter\expandafter
39       #2\expandafter\expandafter\expandafter
40       {\expandafter\PT@temp\expandafter{#2}}}
41
42 \def\PT@consmacro{\PT@listopmacro\Cons}
43 \def\PT@appendmacro{\PT@listopmacro\Cat}
```

The follwing two macros can be used to add something to the beginning or the
end of a constrol structure.

```
44 \def\PT@addbeginmacro #1#2% add #2 to the beginning of #1
45    {\def\PT@temp{#2}%
46     \expandafter\expandafter\expandafter
47       \def\expandafter\expandafter\expandafter
48       #1\expandafter\expandafter\expandafter
49       {\expandafter\PT@temp #1}}
50
51 \def\PT@gaddendmacro #1#2% add #2 to the end of #1
52    {\expandafter\gdef\expandafter #1\expandafter{#1#2}}
```

\PT@enamedef This is much like \@namedef, but it expands #2 once.

```
53 \def\PT@enamedef #1#2% sets name #1 to the expansion of #2
54    {\expandafter\Twiddle\expandafter\@namedef\expandafter{#2}{#1}}
```

9

\PT@adddeftomacroas Given the name of a control structure `#1` and a name of another control structure `#2` and an expression `#3`, we add the definition of `#2` to the expansion of `#3` to the macro `#1`.

```
55 \def\PT@adddeftomacroas#1#2#3%
56   {\expandafter\expandafter\expandafter
57      \def\expandafter\expandafter\expandafter\PT@temp
58        \expandafter\expandafter\expandafter
59          {\expandafter\expandafter\expandafter\def
60            \expandafter\expandafter\csname #2\endcsname
61            \expandafter{#3}}%
62    \expandafter\expandafter\expandafter\PT@gaddendmacro
63      \expandafter\expandafter\expandafter
64        {\expandafter\expandafter\csname #1\endcsname
65         \expandafter}\expandafter{\PT@temp}}
```

\PT@adddeftomacro This is a special case of `\PT@adddeftomacroas` where `#3` is the expansion of `#2`.

```
66 \def\PT@adddeftomacro#1#2%
67   {\def\PT@temp{\PT@adddeftomacroas{#1}{#2}}%
68    \expandafter\PT@temp\csname #2\endcsname}
```

\PT@addoptargtomacro

```
69 \def\PT@addoptargtomacro
70   {\PT@add@argtomacro\PT@makeoptarg}
71 \def\PT@addargtomacro
72   {\PT@add@argtomacro\PT@makearg}
73
74 \def\PT@add@argtomacro#1#2#3%
75   {\expandafter\expandafter\expandafter\gdef
76      \expandafter\expandafter\expandafter\PT@temp
77        \expandafter\expandafter\expandafter{\csname #3\endcsname}%
78    #1%
79    \expandafter\PT@gaddendmacro\expandafter
80      {\expandafter#2\expandafter}\expandafter{\PT@temp}}
81
82 \def\PT@makeoptarg%
83   {\expandafter\def\expandafter\PT@temp\expandafter
84      {\expandafter[\PT@temp]}}
85 \def\PT@makearg%
86   {\expandafter\def\expandafter\PT@temp\expandafter
87      {\expandafter{\PT@temp}}}
88
89 %
90 % \begin{macro}{\PT@mtimesn}
91 % Expands to |#1| times |#2|. (Work in progress.)
92 %    \begin{macrocode}
93 % \def\PT@mtimesn #1#2%
94 %   {\expandafter\PT@mtimtesn@\romannumeral #1011{#2}}
95 % \def\PT@mtimesn@ #1i#2%
96 %   {\if#1m%
```

10

```
97 %        #2\expandafter\PT@mtimesn@#1i#3}
```

\PT@gobbleoptional Gobbles one optional argument. Ignores spaces.

```
98 \newcommand*{\PT@gobbleoptional}[1][]{\ignorespaces}
```

\PT@origomit Save the original definition of omit.

```
99 \let\PT@origomit\omit
```

\PT@disableomitonce Undefines the next use of omit.

```
100 \def\PT@disableomitonce
101    {\def\omit
102       {\let\omit\PT@origomit}}
```

7.2   The environment

The general idea is to first scan the contents of the environment and store them in a token register. In a few test runs, the positions of the column borders are determined. After that, the columns are sorted and the table is typeset, translating the named ranges into appropriate calls to \multicolumn.

\beginpolytable This macro starts the environment. It should, however, not be called directly, but rather in a LaTeX environment. We just initialize the token register to the empty string and then start scanning.

```
103 \newcommand{\beginpolytable}%
```

We save the current enclosing LaTeX environment in \PT@environment. This will be the \end we will be looking for, and this will be the environment we manually close in the end.

```
104    {\edef\PT@environment{\@currenvir}%
105     \begingroup
106     % new in v0.7: save counters
107     \PT@savecounters
108     \toks@{}% initialise token register
109     \PT@scantoend}
```

\endpolytable This is just defined for convenience.

```
110 \let\endpolytable=\relax
```

\PT@scantoend We scan until the next occurence of \endpolytable and store the tokens. Then we continue with determining the column widths.

```
111 \long\def\PT@scantoend #1\end #2%
112    {\toks@\expandafter{\the\toks@ #1}%
113     \def\PT@temp{#2}%
114     \ifx\PT@temp\PT@environment
115       \expandafter\PT@getwidths
116     \else
117       \toks@\expandafter{\the\toks@\end{#2}}%
118       \expandafter\PT@scantoend
119     \fi}
```

\PT@getwidths   Here, we make as many test runs as are necessary to determine the correct column widths.

120 `\def\PT@getwidths`

We let the `\column` command initialize a column in the first run.

121 `  {\let\column\PT@firstrun@column`

There is the possibility to save or restore columns. This is new in v0.4.

122 `    \let\savecolumns\PT@savewidths`
123 `    \let\restorecolumns\PT@restorewidths`

We *always* define a pseudo-column `@begin@`. This denotes the begin of a row.

124 `    \column{@begin@}{@{}l@{}}`
125 `    \PT@cols=0\relax%`

The two other commands that are allowed inside of the environment, namely `\fromto` and `\nextline` are initialized. The `\fromto` command may increase the current widths of some columns, if necessary, whereas `\nextline` just resets the counter that keeps track of the "current" column, to 0.

126 `    \let\fromto\PT@fromto`
127 `    \let\nextline\PT@resetcolumn`
128 `    \PT@changedfalse % nothing has changed so far`
129 `    \PT@resetcolumn % we are at the beginning of a line`

Now we are ready for a test run.

130 `    \the\toks@`

After the first run, we print extra information. We use the contents of the macro `\column` to check whether we are in the first run, because it will be reset below for all other runs to do nothing.

131 `    \ifx\column\PT@otherrun@column`
132 `    \else`
133 `        % we are in first run, print extra info`
134 `        \PT@typeout@{Number of columns: \the\PT@cols}%`
135 `        \PT@typeout@{Column list: \Print\PT@allcols}%`
136 `    \fi`

The columns are initialised after the first run. Therefore, we make sure that the `\column` command won't do much in the other runs. Also, saving and restoring columns is no longer needed.

137 `    \let\PT@firstrun@column\PT@otherrun@column`
138 `    \let\savecolumns\PT@gobbleoptional`
139 `    \let\restorecolumns\PT@gobbleoptional`
140 `    \let\PT@savewidths\PT@gobbleoptional`
141 `    \let\PT@restorewidths\PT@gobbleoptional`

New in v0.7.1: restore counters after each trial run.

142 `    \PT@restorecounters`

If some column widths have indeed changed in the test run, this will be indicated by the flag `\ifPT@changed`. Depending on this flag, we will either loop and rerun, or we will continue in `\PT@sortcols`.

12

```
143     \ifPT@changed
144        % we need to rerun if something has changed
145        \expandafter\PT@getwidths
146     \else
147        % we are done and can do the sorting
148        \PT@typeout@{Reached fixpoint.}%
149        \expandafter\PT@sortcols
150     \fi}
```

\PT@savecounters Save all LaTeX counters so that they can be restored after a trial run.

```
151 \def\PT@savecounters
152   {\begingroup
153      \def\@elt ##1%
154        {\global\csname c@##1\endcsname\the\csname c@##1\endcsname}%
155      \xdef\PT@restorecounters{\cl@@ckpt}%
156   \endgroup}
```

\PT@sortcols The column borders are sorted by their horizontal position on the page (width). The they get numbered consecutively. After that, we are well prepared to typeset the table.

```
157 \def\PT@sortcols
```

First, we sort the list. To make sure that the computation is only executed once, we save the sorted list by means of an \edef. Sorting happens with lazylist's \Insertsort which expects an order and a list. As order, we provide \PT@ltwidth, which compares the widths of the columns. To prevent expansion of the list structure, given by \Cons and \Nil, we fold the list with the \noexpanded versions of the list constructors.

```
158   {\edef\PT@sortedlist
159      {\Foldr{\noexpand\Cons}{\noexpand\Nil}%
160        {\Insertsort\PT@ltmax\PT@allcols}}%
161   \PT@typeout@{Sorted columns: \Print\PT@sortedlist}%
```

Now, each column is assigned a number, starting from zero.

```
162   \PT@cols=0\relax%
163   \Execute{\Map\PT@numbercol\PT@sortedlist}%
164   \PT@typeout@{Numbered successfully, last column is \StripColumn\PT@lastcol}%
```

Now is a good time to save table information, if needed later. We will also compare our computed information with the restored maximum widths.

```
165      \ifx\PT@currentwidths\empty
166      \else
167        \PT@typeout@{Saving table information for \PT@currentwidths .}%
168        \expandafter\PT@saveinformation\expandafter{\PT@currentwidths}%
169      \fi
```

Finally, we can typeset the table.

```
170   \PT@typeset}
```

\PT@typeset

```
171 \def\PT@typeset
```

13

As a first step, we generate the table's preamble and print it for debugging purposes.

```
172  {\PT@typeout@{Typesetting the table ...}%
173   \edef\PT@temp{@{}\Execute{\Map\PT@preamble\PT@sortedlist}}%
174   %\PT@typeout@{Preamble: \PT@temp}%
```

Now, we redefine \fromto and \nextline to their final meaning in the typesetting process. The \fromto statements will be replaced by appropriate calls to \multicolumn, whereas the \nextline will again reset the counter for the current column, but also call the table environment's newline macro.

```
175   \let\fromto\PT@multicolumn
176   \PT@resetcolumn % we are at the beginning of a line
177   \let\nextline=\PT@resetandcr
```

Now we start the tabular environment with the computed preamble.

```
178   \expandafter\PT@begin\expandafter{\PT@temp}%
```

Run, and this time, typeset, the contents.

```
179   \the\toks@
```

End the array, close the group, close the environment. We are done!

```
180   \PT@end
181   \endgroup
182   \PT@typeout@{Finished.}%
183   \expandafter\end\expandafter{\PT@environment}}%
```

7.3   The trial runs

For each column, we store information in macros that are based on the column name. We store a column's type (i.e. its contribution to the table's preamble), its current width (i.e. its the horizontal position where the column will start on the page), and later, its number, which will be used for the \multicolumn calculations.

\PT@firstrun@column During the first trial run, we initialise all the columns. We store their type, as declared in the \column command inside the environment, and we set their initial width to 0pt. Furthermore, we add the column to the list of all available columns, increase the column counter, and tell TeX to ignore spaces that might follow the \column command. New in v0.4.1: We make a case distinction on an empty type field to prevent warnings for columns that have been defined via \PT@setmaxwidth – see there for additional comments. New in v0.4.2: We allow redefinition of width if explicitly specified, i.e. not equal to 0pt.

```
184  \newcommand\PT@firstrun@column[3][0pt]%
185   {\@ifundefined{PT@col@#2.type}%
186      {\PT@typeout@{Defining column #2 at #1.}%
187       \@namedef{PT@col@#2.type}{#3}%
188       \@namedef{PT@col@#2.width}{#1}% initialize the width of the column
189       % add the new column to the (sortable) list of all columns
190       \PT@consmacro\PT@allcols{PT@col@#2}%
191       \advance\PT@cols by 1\relax}%
```

```
192        {\expandafter\ifx\csname PT@col@#2.type\endcsname\empty
193          \relax % will be defined in a later table of the same set
194        \else
195          \PT@warning{Redefining column #2}%
196        \fi
197        \@namedef{PT@col@#2.type}{#3}%
198        \expandafter\ifdim#1>0pt\relax
199          \PT@typeout@{Redefining column #2 at #1.}%
200          \@namedef{PT@col@#2.width}{#1}%
201        \fi
202      }%
```

For the case that we are saving and there is not yet information from the `.aux` file, we define the `.max` and `.trusted` fields if they are undefined. If information becomes available later, it will overwrite these definitions.

```
203      \@ifundefined{PT@col@#2.max}%
204        {\@namedef{PT@col@#2.max}{#1}%
205          \expandafter\let\csname PT@col@#2.trusted\endcsname\PT@true}{}%
206      \ignorespaces}
```

\PT@otherrun@column   In all but the first trial run, we do not need any additional information about the columns any more, so we just gobble the two arguments, but still ignore spaces.

```
207 \newcommand\PT@otherrun@column[3][]%
208   {\ignorespaces}
```

\PT@checkcoldefined   This macro verifies that a certain column is defined and produces an error message if it is not.

```
209 \def\PT@checkcoldefined #1%
210   {\@ifundefined{PT@col@#1.type}%
211      {\PackageError{polytable}{Undefined column #1}{}}{}%
```

\PT@fromto   Most of the work during the trial runs is done here. We increase the widths of certain columns, if necessary. Note that there are two conditions that have to hold if \fromto{A}{B} is encountered:

- the width of A has to be at least the width of the current (i.e. previous) column.

- the width of B has to be at least the width of A, plus the width of the entry.

```
212 \def\PT@fromto #1#2#3%
```

We start by checking a switch.

```
213   {\PT@infromto
214   \def\PT@infromto{%
215      \PackageError{polytable}{Nested fromto}{}}%
```

Next, we check that both columns are defined.

```
216   \PT@checkcoldefined{#1}%
217   \PT@checkcoldefined{#2}%
```

Here, we check the first condition.

```
218    \def\PT@temp{PT@col@#1}%
219    \ifx\PT@currentcol\PT@temp
220      \PT@typeout@{No need to skip columns.}%
221    \else
222      \PT@colwidth=\expandafter\@nameuse\expandafter
223                    {\PT@currentcol.width}\relax
224      \ifdim\PT@colwidth>\csname PT@col@#1.width\endcsname\relax
225        % we need to change the width
226        \PT@typeout@{s #1: old=\@nameuse{PT@col@#1.width} new=\the\PT@colwidth}%
227        \PT@changedtrue
228        \PT@enamedef{PT@col@#1.width}{\the\PT@colwidth}%
229      \fi
```

The same for the untrusted .max values.

```
230      \PT@colwidth=\expandafter\@nameuse\expandafter
231                    {\PT@currentcol.max}\relax
232      \ifdim\PT@colwidth>\csname PT@col@#1.max\endcsname\relax
233        % we need to change the width
234        \PT@typeout@{S #1: old=\@nameuse{PT@col@#1.max} new=\the\PT@colwidth}%
235        \PT@changedtrue
236        \PT@checkrerun
237        \PT@enamedef{PT@col@#1.max}{\the\PT@colwidth}%
238      \fi
239      \ifnum\csname PT@col@#1.trusted\endcsname=\PT@false\relax
240        \ifdim\PT@colwidth=\csname PT@col@#1.max\endcsname\relax
241          \PT@typeout@{#1=\the\PT@colwidth\space is now trusted}%
242          \expandafter\let\csname PT@col@#1.trusted\endcsname\PT@true%
243        \fi
244      \fi
245    \fi
```

To test the second condition, we have to test-typeset the contents of the column, contained in #3. We prepare a "safe environment" for these contents. We determine whether we are in math mode or not, put the contents into an hbox in the same mode, and we are typesetting the contents in the same environment as we will typeset the table in the end.

```
246    \begingroup
247    \ifmmode
248      \let\d@llarbegin=$%$
249      \let\d@llarend=$%$
250      \let\col@sep=\arraycolsep
251    \else
252      \let\d@llarbegin=\begingroup
253      \let\d@llarend=\endgroup
254      \let\col@sep=\tabcolsep
255    \fi
256    %\def\PT@currentcol{PT@col@#1}%
257    %\ifx\PT@currentcol\PT@nullcol
258    %\else
```

16

```
259    %  \PT@addbeginmacro\PT@currentpreamble{@{}}%
260    %\fi
261    \expandafter\expandafter\expandafter
262      \def\expandafter\expandafter\expandafter\PT@currentpreamble
263        \expandafter\expandafter\expandafter
264          {\csname PT@col@#1.type\endcsname}%
265    \setbox0=\hbox{%
266      \expandafter\@mkpream\expandafter{\PT@currentpreamble}%
267      \def\@sharp{\strut #3}%
268      %\show\@preamble
269      \@preamble}%
270    \expandafter\gdef\expandafter\PT@temp\expandafter{\the\wd0}%
271    \endgroup
```

Now begins the real comparison.

```
272    \global\PT@colwidth=\@nameuse{PT@col@#1.width}%
273    \global\advance\PT@colwidth by \PT@temp\relax%
274    \ifdim\PT@colwidth>\csname PT@col@#2.width\endcsname\relax
275      % we need to change the width
276      \PT@typeout@{c #2:
277                  old=\@nameuse{PT@col@#2.width}
278                  new=\the\PT@colwidth}%
279      \PT@changedtrue
280      \PT@enamedef{PT@col@#2.width}{\the\PT@colwidth}%
281    \fi
```

And again, we have to do the same for the untrusted maximums.

```
282    \global\PT@colwidth=\@nameuse{PT@col@#1.max}%
283    \global\advance\PT@colwidth by \PT@temp\relax%
284    \ifdim\PT@colwidth>\csname PT@col@#2.max\endcsname\relax
285      % we need to change the width
286      \PT@typeout@{C #2:
287                  old=\@nameuse{PT@col@#2.max}
288                  new=\the\PT@colwidth}%
289      \PT@changedtrue
290      \PT@checkrerun
291      \PT@enamedef{PT@col@#2.max}{\the\PT@colwidth}%
292    \fi
293    \ifnum\csname PT@col@#2.trusted\endcsname=\PT@false\relax
294      \ifdim\PT@colwidth=\csname PT@col@#2.max\endcsname\relax
295        \PT@typeout@{#2=\the\PT@colwidth\space is now trusted}%
296        \expandafter\let\csname PT@col@#2.trusted\endcsname\PT@true%
297      \fi
298    \fi
```

Finally, we update the current column to #2, and, of course, we ignore spaces after the \fromto command.

```
299    \def\PT@currentcol{PT@col@#2}%
300    \let\PT@infromto\empty
301    \ignorespaces}%
```

\PT@checkrerun If we have changed something with the trusted widths, we have to check whether we are in a situation where we are using previously defined columns. If so, we have to rerun LaTeX.

```
302 \def\PT@checkrerun
303   {\ifnum\PT@inrestore=\PT@true\relax
304     \PT@rerun
305   \fi}
```

\PT@resetcolumn At the end of a line, we reset the current column to the special column `@begin@`.

```
306 \newcommand*{\PT@resetcolumn}[1][]%
307   {\let\PT@currentcol\PT@nullcol}
```

\PT@nullcol The name of the `@begin@` column as a macro, to be able to compare to it with `\ifx`.

```
308 \def\PT@nullcol
309   {PT@col@@begin@}
```

7.4   Sorting and numbering the columns

Not much needs to be done here, all the work is done by the macros supplied by the lazylist package. We just provide a few additional commands to facilitate their use.

\Execute
\Sequence With `\Execute`, a list of commands (with sideeffects) can be executed in sequence. Usually, first a command will be mapped over a list, and then the resulting list will be executed.

```
310 \def\Execute{\Foldr\Sequence\empty}
311 \def\Sequence #1#2{#1#2}
```

\ShowColumn This is a debugging macro, that is used to output the list of columns in a pretty way. The columns internally get prefixes to their names, to prevent name conflicts with normal commands. In the debug output, we gobble this prefix again.

```
312 \def\ShowColumn #1%
313   {\ShowColumn@#1\ShowColumn@}
314 \def\ShowColumn@ PT@col@#1\ShowColumn@
315   {#1 }
316 \def\StripColumn #1%
317   {\expandafter\StripColumn@#1\StripColumn@}
318 \def\StripColumn@ PT@col@#1\StripColumn@
319   {#1}
```

\Print Prints a list of columns, using `\ShowColumn`.

```
320 \def\Print#1{\Execute{\Map\ShowColumn#1}}
```

\PT@TeXif This is an improved version of lazylist's `\TeXif`. It does have an additional `\relax` to terminate the condition. The `\relax` is gobbled again to keep it fully expandable.

```
321 \def\PT@TeXif #1%
322   {\expandafter\@gobble#1\relax
323       \PT@gobblefalse
324     \else\relax
325       \gobbletrue
326     \fi}
327 \def\PT@gobblefalse\else\relax\gobbletrue\fi #1#2%
328   {\fi #1}
```

\PT@ltmax The order by which the columns are sorted is given by the order on their (untrusted) widths.

```
329 \def\PT@ltmax #1#2%
330   {\PT@TeXif{\ifdim\csname #1.max\endcsname<\csname #2.max\endcsname}}
```

\PT@numbercol This assigns the next consecutive number to a column. We also reassign PT@lastcol to remember the final column.

```
331 \def\PT@numbercol #1%
332   {%\PT@typeout@{numbering #1 as \the\PT@cols}%
333     \PT@enamedef{#1.num}{\the\PT@cols}%
334     \def\PT@lastcol{#1}%
335     \advance\PT@cols by 1\relax}
```

7.5 Typesetting the table

\PT@preamble The table's preamble is created by mapping this function over the column list and then \Executeing ...New: We always use l, as the specific type is always given by the \multicolumn. Yet new: We use @{}l@{}, to prevent column separation space from being generated.

```
336 \def\PT@preamble #1%
337 % {\csname #1.type\endcsname}
338 % {l}
339   {l@{}}
```

Remember that there are three important macros that occur in the body of the polytable: \column, \fromto, and \nextline. The \column macro is only really used in the very first trial run, so there is nothing new we have to do here, but the other two have to be redefined.

\PT@resetandcr This is what \nextline does in the typesetting phase. It resets the current column, but it also calls the table environment's newline macro \\ ...If we are *not* in the last column, we insert an implicit fromto. This is needed for the boxed environment to make each column equally wide. Otherwise, if the boxed environment is typeset in a centered way, thing will go wrong.

```
340 \newcommand{\PT@resetandcr}[1][0pt]%
341   {\ifx\PT@currentcol\PT@lastcol
342     \else
343       \ifx\PT@currentcol\PT@nullcol
344         \edef\PT@currentcol{\Head{\Tail\PT@sortedlist}}%
```

19

```
345      \fi
346      \edef\PT@currentcol@{\StripColumn\PT@currentcol}%
347      \edef\PT@lastcol@
348        {\StripColumn\PT@lastcol}%
349      \PT@typeout@{adding implicit fromto from \PT@currentcol@
350                              \space to \PT@lastcol@}%
351      \expandafter\expandafter\expandafter\fromto
352        \expandafter\expandafter\expandafter{%
353          \expandafter\expandafter\expandafter\PT@currentcol@
354            \expandafter}\expandafter{\PT@lastcol@}{}%
355    \fi
356    \PT@typeout@{Next line ...}%
357    \PT@resetcolumn\\[#1]}
```

\PT@multicolumn   All the \fromtos are expanded into \multicolumn calls, which is achieved by
this quite tricky macro. Part of the trickyness stems from the fact that a
\multicolumn's expansion starts with \omit which is a plain TEX primitive that
causes the template for a table column to be ignored. But \omit has to be the
first token (after expansion) in a column to be valid, which is why the alignment
tabs & and the \multicolumn calls have to be close to each other. It would maybe
be better to call \omit manually and hack \multicolumn later!!

   This macro gets three arguments. The first is the column in which the entry
begins, the second is the column *before* which the entry stops, and the third
contains the contents that should be typeset in this range.

```
358  \def\PT@multicolumn #1#2#3%
```

We start by producing an \omit to indicate that we want to ignore the column
format that has been specified in the table header. After that, we disable the
\omit command, because we will later call \multicolumn which contains another
one. A second \omit would usually cause an error. TODO: Make this work to
simplify the rest. For now, we don't use this.

```
359    {%\omit\PT@disableomitonce
```

We skip ahead until we are in the column in which the entry should start. For
this, we store the number of the column we want to start in and subtract the
current columns number. If the current column is the null column, we have to
adjust by -1 which is not nice, but necessary . . . In 0.4.3: added missing relax
after \global\advance.

```
360      % skip to current position
361      \global\PT@cols=\@nameuse{PT@col@#1.num}%
362      \global\advance\PT@cols
363        by -\expandafter\csname\PT@currentcol.num\endcsname\relax
364      \ifx\PT@currentcol\PT@nullcol
365        \global\advance\PT@cols by -1\relax%
366      \fi
```

We now skip by inserting alignment tabs and using a multicolumn with no content.
It might be nicer to just use as many tabs as necessary, because we could do with
less case distinctions. The current value of \PT@cols indicates how many tabs we

have to insert, minus one. We will insert that one tab (which is the minimum we have to insert) just before we insert the content, and first deal with the extra tabs.

```
367    \PT@typeout@{skipping:
368            nf=\expandafter\ShowColumn
369                \expandafter{\PT@currentcol}nt=#1 %
370            from=\expandafter
371                \csname\PT@currentcol.num\endcsname\space
372            to=\@nameuse{PT@col@#1.num}}%
373    \ifnum\PT@cols>0\relax
```

So there are extra tabs necessary.

```
374        \ifnum\PT@cols>1\relax
```

We can use a multicolumn to save time.

```
375        \global\advance\PT@cols by -1\relax
376        \PT@typeout@{after next &, multicolumn \the\PT@cols\space blank}%
377        \PT@NextCol
378        \multicolumn{\the\PT@cols}{@{}l@{}}{}%
379      \fi
380      \PT@NextCol
381    \fi
```

We now are in the correct column and can print the contents. Again, we have to check if we have to use a \multicolumn. If we do, we will use the formatting type of the first column that it spans, in contrast to normal \multicolumns which always take an extra parameter to determine how to format their contents. An optional parameter should be introduced here to make overriding the default template possible!! New: we always use a \multicolumn, otherwise spacing will be inconsistent sometimes.

```
382    \global\PT@cols=\@nameuse{PT@col@#2.num}%
383    \global\advance\PT@cols by -\@nameuse{PT@col@#1.num}\relax%
384    %\ifnum\PT@cols>1\relax
385      % we always skip one column
386      \PT@typeout@{after next &,
387                putting text in \the\PT@cols\space multicol}%
388      \PT@typeout@{nf=#1 nt=#2 %
389                from=\@nameuse{PT@col@#1.num}
390                to=\@nameuse{PT@col@#2.num}}%
391      \expandafter\global\expandafter\let\expandafter\PT@temp
392        \csname PT@col@#1.type\endcsname%
393      \PT@NextCol
394      % use multicolumn
395      \expandafter\multicolumn
396        \expandafter{\expandafter\the\expandafter\PT@cols
397        \expandafter}\expandafter{\PT@temp}{#3}%
398      %\PT@typeout@{!!!!}%
399    %\else
400    %    \PT@NextCol
401    %    #3%
402    %\fi
```

We reset the current column to `#2` and ignore spaces after the command. Then we are done . . .

```
403     % set current column
404     \def\PT@currentcol{PT@col@#2}%
405     \ignorespaces}%
```

\PT@NextCol   We hide the tab `&` in a macro, mostly to be able to add debugging output.

```
406 \def\PT@NextCol
407   {\PT@typeout@{ & }%
408    &}%
```

\PT@placeinbox   This macro is an alternative for `\PT@multicolumn`. It can be used to produce a simple box-based output instead of a table. We use the precomputed width information to typeset the contents of the table in aligned boxes. The arguments are the same as for `\PT@multicolumn`, i.e. the start and the end columns, plus the contents.

```
409 \def\PT@placeinbox#1#2#3%
```

We start by computing the amount of whitespace that must be inserted before the entry begins. We then insert that amount of space.

```
410   {\PT@colwidth=\@nameuse{PT@col@#1.max}%
411    \advance\PT@colwidth by -\expandafter\csname\PT@currentcol.max\endcsname
412    \leavevmode
413    \hb@xt@\PT@colwidth{%
414      \expandafter\@mkpream\expandafter{@{}l@{}}%
415      \let\@sharp\empty%
416      %\show\@preamble
417      \@preamble}%
418 % We continue by computing the width of the current entry.
419 %     \begin{macrocode}
420    \PT@colwidth=\@nameuse{PT@col@#2.max}%
421    \advance\PT@colwidth by -\@nameuse{PT@col@#1.max}\relax%
```

In the previous version, we really generated a hbox at this place. However, this is not so nice with respect to spacing and tabular specifiers. Therefore, we now use either an array or a tabular environment that can reuse the given specifier.

```
422    \ifmmode
423      \PT@typeout@{*math mode*}%
424      \let\d@llarbegin=$%$
425      \let\d@llarend=$%$
426      \let\col@sep\arraycolsep
427    \else
428      \PT@typeout@{*text mode*}%
429      \let\d@llarbegin=\begingroup
430      \let\d@llarend=\endgroup
431      \let\col@sep\tabcolsep
432    \fi
433    %\def\PT@currentcol{PT@col@#1}%
434    \expandafter\expandafter\expandafter
```

```
435    \def\expandafter\expandafter\expandafter\PT@currentpreamble
436      \expandafter\expandafter\expandafter
437        {\csname PT@col@#1.type\endcsname}%
438    %\ifx\PT@currentcol\PT@nullcol
439    %\else
440    %  \PT@addbeginmacro\PT@currentpreamble{@{}}%
441    %\fi
```

Now we proceed very much like in the test run(s), but we really output the box, and we use a specific width.

```
442    \hb@xt@\PT@colwidth{%
443      \expandafter\@mkpream\expandafter{\PT@currentpreamble}%
444      \def\@sharp{\strut #3}%
445      %\show\@preamble
446      \@preamble}%
```

Finally, we have to reset the current column and ignore spaces.

```
447    \def\PT@currentcol{PT@col@#2}%
448    \ignorespaces}%
```

7.6 Saving and restoring column widths

Column width information can be saved under a name and thus be reused in other tables. The idea is that the command \savecolumns can be issued inside a polytable to save the current column information, and \restorecolumns can be used to make that information accessible in a later table. All tables using the same information should have the same column widths, which means that some information might need to be passed back. Therefore, we need to write to an auxiliary file. TODO: As implemented now, this only really works in conjunction with the pboxed environment.

Both \savecolumns and \restorecolumns are mapped to the internal commands \PT@savewidths and \PT@restorewidths. Both take an optional argument specifying a name for the column width information. Thereby, multiple sets of such information can be used simultaneously.

One important thing to consider is that the widths read from the auxiliary file must not be trusted. The user may have edited the source file before the rerun, and therefore, the values read might actually be too large (or too small, but this is less dangerous).

The way we solve this problem is to distinguish two width values per column: the trusted width, only using information from the current run, and the untrusted width, incorporating information from the .aux file. An untrusted width can become (conditionally) trusted if it is reached in the computation with respect to an earlier column. (Conditionally, because its trustworthiness still depends on the earlier columns being trustworthy.) In the end, we can check whether all untrusted widths are conditionally trusted.

We write the final, the maximum widths, into the auxiliary file. We perform the write operation when we are sure that a specific set is no longer used. This is the case when we save a new set under the same name, or at the end of the

document. The command `\PT@verifywidths` takes care of this procedure. This command will also check if a rerun is necessary, and issue an appropriate warning if that should be the case.

`\PT@setmaxwidth` First, we need a macro to help us interpreting the contents of the `.aux` file. New v0.4.1: We need to define the restored columns with the `\column` command, because otherwise we will have problems in the case that later occurences of tables in the document that belong to the same set, but define additional columns. (Rerun warnings appear ad infinitum.) In v0.4.2: columns with width 0.0 are now always trusted.

```
449 \newcommand*{\PT@setmaxwidth}[3][\PT@false]% #2 column name, #3 maximum width
450   {\@namedef{PT@col@#2.max}{#3}%
451    \ifdim#3=0pt\relax
452      \expandafter\let\csname PT@col@#2.trusted\endcsname=\PT@true%
453    \else
454      \expandafter\let\csname PT@col@#2.trusted\endcsname=#1%
455    \fi
456    \column{#2}{}}
```

`\PT@loadtable` Now, we can load table information that has been read from the `.aux` file. Note that a `\csname` construct expands to `\relax` if undefined.

```
457 \def\PT@loadtable#1% #1 table id number
458   {%\expandafter\show\csname PT@restore@\romannumeral #1\endcsname
459    %\show\column
460    \PT@typeout@
461      {Calling \expandafter\string
462              \csname PT@restore@\romannumeral #1\endcsname.}%
463    \let\maxcolumn\PT@setmaxwidth
464    %\expandafter\show\csname PT@load@\romannumeral #1\endcsname
465    \csname PT@restore@\romannumeral #1\endcsname}
```

`\PT@loadtablebyname` Often, we want to access table information by a column width set name. We make the maximum column widths accessible, but also the information from the previous table that has been using the same column width set.

```
466 \def\PT@loadtablebyname#1% #1 set name
467   {\PT@typeout@{Loading table information for column width set #1.}%
468    \expandafter\PT@loadtable\expandafter{\csname PT@widths@#1\endcsname}}%
469 %  \advance\PT@cols by \PT@restoredcols\relax
470 %  \expandafter\PT@appendmacro\expandafter\PT@allcols
471 %    \expandafter{\PT@restoredallcols}}
```

`\PT@saveinformation` In each table for which the widths get reused (i.e., in all tables that use either `\savecolumns` or `\restorecolumns`, we have to store all important information for further use.

```
472 \def\PT@saveinformation#1% #1 set name
473   {\expandafter\def\expandafter\PT@temp\expandafter
474      {\csname PT@widths@#1\endcsname}%
475    \expandafter\def\expandafter\PT@temp\expandafter
```

24

```
476        {\csname PT@restore@\romannumeral\PT@temp\endcsname}%
477        \expandafter\gdef\PT@temp{}% start empty
478        % this is: \Execute{\Map{\PT@savecolumn{\PT@temp}}\PT@sortedlist}
479        \expandafter\Execute\expandafter{\expandafter
480          \Map\expandafter{\expandafter\PT@savecolumn
481            \expandafter{\PT@temp}}\PT@sortedlist}}
```

\PT@savecolumn   A single column is saved by this macro.

```
482 \def\PT@savecolumn#1#2% #1 macro name, #2 column name
483    {\PT@typeout@{saving column #2 in \string #1 ...}%
484     \def\PT@temp{#2}%
485     \ifx\PT@temp\PT@nullcol
486       \PT@typeout@{skipping nullcol ...}%
487     \else
488       \PT@typeout@{max=\csname #2.max\endcsname, %
489                    width=\csname #2.width\endcsname, %
490                    trusted=\csname #2.trusted\endcsname}%
491       % we need the column command in here
492       % we could do the same in \column, but then the location of
493       % \save / \restore matters ...
494       \PT@gaddendmacro{#1}{\maxcolumn}%
495       \ifnum\csname #2.trusted\endcsname=\PT@true\relax
496         \PT@gaddendmacro{#1}{[\PT@true]}%
497       \fi
498       \edef\PT@temp{\StripColumn{#2}}%
499       \PT@addargtomacro{#1}{PT@temp}%
500       \PT@addargtomacro{#1}{#2.max}%
501       \PT@gaddendmacro{#1}{\column}%
502       \PT@addoptargtomacro{#1}{#2.width}%
503       \edef\PT@temp{\StripColumn{#2}}%
504       \PT@addargtomacro{#1}{PT@temp}%
505       \PT@addargtomacro{#1}{#2.type}%
506       %\show#1%
507     \fi
508    }
```

\PT@savewidths   If we really want to save column width information, then the first thing we should
worry about is that there might already have been a set with the name in question.
Therefore, we will call \PT@verifywidths for that set. In the case that there is
no set of this name yet, we will schedule the set for verification at the end of
document.

```
509 \newcommand*{\PT@savewidths}[1][default@]
510    {\PT@typeout@{Executing \string\savecolumns [#1].}%
511     \def\PT@currentwidths{#1}%
512     \PT@verifywidths{#1}%
```

We now reserve a new unique number for this column width set by increasing the
\PT@table counter. We then associate the given name (or default@) with the
counter value and restore the widths from the .aux file if they are present.

```
513    \global\advance\PT@table by 1\relax
514    \expandafter\xdef\csname PT@widths@#1\endcsname
515      {\the\PT@table}%
516    \PT@loadtable{\PT@table}%
517    \ignorespaces}
```

\PT@restorewidths   Restoring information is quite simple. We just load all information available.

```
518  \newcommand*{\PT@restorewidths}[1][default@]
519    {\PT@typeout@{Executing \string\restorecolumns [#1].}%
520    \def\PT@currentwidths{#1}%
521    \let\PT@inrestore\PT@true
522    \PT@loadtablebyname{#1}%
523    \ignorespaces}
```

\PT@comparewidths

```
524  \def\PT@comparewidths#1% #1 full column name
525    {\@ifundefined{#1.max}%
526      {\PT@typeout@{computed width for #1 is fine ...}}%
527      {\ifdim\csname #1.max\endcsname>\csname #1.width\endcsname\relax
528        \PT@typeout@{Preferring saved width for \StripColumn{#1}.}%
529        \PT@changedtrue
530        \PT@colwidth=\@nameuse{#1.max}\relax
531        \PT@enamedef{#1.width}{\the\PT@colwidth}%
532      \fi}}
```

\PT@trustedmax

```
533  \def\PT@trustedmax#1%
534    {\PT@TeXif{\ifnum\csname #1.trusted\endcsname=\PT@true}}
```

\PT@equalwidths

```
535  \def\PT@equalwidths#1% #1 full column name
536    {\@ifundefined{#1.max}{}%
537      {\ifdim\csname #1.max\endcsname=\csname #1.width\endcsname\relax
538        \PT@typeout@{col #1 is okay ...}%
539      \else
540        \PT@rerun% a rerun is needed
541      \fi}}
```

\PT@verifywidths

```
542  \def\PT@verifywidths#1% #1 column width set name
543    {\@ifundefined{PT@widths@#1}%
544      {\PT@typeout@{Nothing to verify yet for set #1.}%
545      \PT@typeout@{Scheduling set #1 for verification at end of document.}%
546      \AtEndDocument{\PT@verifywidths{#1}}}%
547      {\PT@typeout@{Verifying column width set #1.}%
548      \expandafter\PT@verify@widths\expandafter
549        {\csname PT@widths@#1\endcsname}{#1}}}
550
551  \def\PT@verify@widths#1#2% #1 set id number, #2 set name
```

```
552    {\@ifundefined{PT@restore@\romannumeral #1}{}%
553      {\begingroup
554        \let\column\PT@firstrun@column
555        \PT@cols=0\relax%
556        \def\PT@allcols{\Nil}%
557        \PT@loadtablebyname{#2}%
558        \PT@table=#1\relax
559        % nullcolumn is not loaded, therefore:
560        \expandafter\def\csname\PT@nullcol .width\endcsname{0pt}%
561        % checking trust
562        \All{\PT@trustedmax}{\PT@allcols}%
563          {\PT@typeout@{All maximum widths can be trusted -- writing .max!}%
564           \PT@save@table{.max}}%
565          {\PT@typeout@{Untrustworthy maximums widths -- writing .width!}%
566           \PT@rerun
567           \PT@save@table{.width}}%
568        \endgroup}%
569      \PT@typeout@{Verification for #2 successful.}}
```

\PT@save@table   Here we prepare to write maximum column widths to the .aux file.

```
570 \def\PT@save@table#1%
571   {\PT@typeout@{Saving column width information.}%
572    \if@filesw
573      {\immediate\write\@auxout{%
574        \gdef\expandafter\noexpand
575          \csname PT@restore@\romannumeral\PT@table\endcsname
576            {\Execute{\Map{\PT@write@column{#1}}\PT@allcols}}}}%
577    \fi}
```

\PT@write@column   We define the column command to write to the file.

```
578 \def\PT@write@column #1#2%
579   {\noexpand\maxcolumn^^J%
580     {\StripColumn{#2}}%
581     {\@nameuse{#2#1}}}%
```

7.7   The user environments

It remains to define the three environments to be called by the user.

```
582 \newenvironment{ptabular}[1][c]%
583   {\def\PT@begin{\tabular[#1]}%
584    \let\PT@end\endtabular
585    \beginpolytable}
586   {\endpolytable}
587
588 \newenvironment{parray}[1][c]%
589   {\def\PT@begin{\array[#1]}%
590    \let\PT@end\endarray
591    \beginpolytable}
592   {\endpolytable}
```

```
593
594 \def\pboxed{%
595     \let\PT@begin\@gobble
596     \let\PT@end\empty
597     \let\PT@multicolumn\PT@placeinbox
598     \expandafter\beginpolytable\ignorespaces}
599
600 \let\endpboxed\endpolytable
```

That is all.

```
601 ⟨/package⟩
```