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.

```
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` With `\savecolumns[`⟨*setid*⟩`]`, one can save the information of the current table
`\restorecolumns` 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

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{polytable}%
   [2004/02/27 v0.7.2 'polytable' package (Andres Loeh)]
```

New in v0.7.2: The amsmath package clashes with lazylist: both define the command `\And`. Although it would certainly be better to find another name in lazylist, we take precautions for now. (Note that this will still fail if lazylist is already loaded – but then it's not our problem ...

```
\let\PT@original@And\And
\RequirePackage{lazylist}
\let\PT@And\And
\def\PT@prelazylist
  {\let\And\PT@And}
\def\PT@postlazylist
  {\let\And\PT@original@And}
\PT@postlazylist
\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.

```
\DeclareOption{debug}{\AtEndOfPackage\PT@debug}
\DeclareOption{silent}{\AtEndOfPackage\PT@silent}
\ProcessOptions
```

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

```
\newdimen\PT@colwidth
%\newdimen\PT@delta
\newcount\PT@cols
\newcount\PT@table
\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`. In v0.8, we will have a second list that only contains the public columns.

```
\def\PT@allcols{\Nil}
%\def\PT@allpubliccols{\Nil}
\let\PT@infromto\empty
```

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

```
\let\PT@currentwidths\empty
```

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

$$\textbf{class}\ (\mathsf{Eq}\ a) \Rightarrow \mathsf{Ord}\ a\ \textbf{where}$$

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

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

— 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".

$$\begin{array}{lll} \mathit{compare}\ x\ y & \mid x \equiv y & = \mathsf{EQ} \\ & \mid x \leq y & = \mathsf{LT} \\ & \mid \mathit{otherwise} & = \mathsf{GT} \end{array}$$

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

$$\begin{array}{ll} x \leq y & = \mathit{compare}\ x\ y \not\equiv \mathsf{GT} \\ x < y & = \mathit{compare}\ x\ y \equiv \mathsf{LT} \\ x \geq y & = \mathit{compare}\ x\ y \not\equiv \mathsf{LT} \\ x > y & = \mathit{compare}\ x\ y \equiv \mathsf{GT} \end{array}$$

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

$$\begin{array}{lll} \mathit{max}\ x\ y & \mid x \leq y & = y \\ & \mid \mathit{otherwise} & = x \\ \mathit{min}\ x\ y & \mid x \leq y & = x \\ & \mid \mathit{otherwise} & = y \end{array}$$

Figure 4: Commented Haskell code example

```
\def\PT@false{0}
\def\PT@true{1}
\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?)

```
%\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.

```
\def\PT@debug
  {\def\PT@typeout@ ##1{\typeout{(polytable) ##1}}}
\let\PT@typeout@\@gobble
\def\PT@warning{\PackageWarning{polytable}}%
\def\PT@silent
  {\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.

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

7.1 Macro definition tools

`\PT@listopmacro` `\PT@consmacro` `\PT@appendmacro`

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

```
\def\PT@listopmacro #1#2#3% #1 #3 to the list #2
  {\def\PT@temp{#1{#3}}%
   \expandafter\expandafter\expandafter
     \def\expandafter\expandafter\expandafter
     #2\expandafter\expandafter\expandafter
     {\expandafter\PT@temp\expandafter{#2}}}

\def\PT@consmacro{\PT@listopmacro\Cons}
\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.

```
\def\PT@addbeginmacro #1#2% add #2 to the beginning of #1
  {\def\PT@temp{#2}%
   \expandafter\expandafter\expandafter
     \def\expandafter\expandafter\expandafter
```

```
    #1\expandafter\expandafter\expandafter
    {\expandafter\PT@temp #1}}

\def\PT@gaddendmacro #1#2% add #2 to the end of #1
  {\expandafter\gdef\expandafter #1\expandafter{#1#2}}
```

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

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

`\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`.

```
\def\PT@adddeftomacroas#1#2#3%
  {\expandafter\expandafter\expandafter
     \def\expandafter\expandafter\expandafter\PT@temp
       \expandafter\expandafter\expandafter
         {\expandafter\expandafter\expandafter\def
            \expandafter\expandafter\csname #2\endcsname
            \expandafter{#3}}%
   \expandafter\expandafter\expandafter\PT@gaddendmacro
     \expandafter\expandafter\expandafter
       {\expandafter\expandafter\csname #1\endcsname
        \expandafter}\expandafter{\PT@temp}}
```

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

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

`\PT@addoptargtomacro`

```
\def\PT@addoptargtomacro
  {\PT@add@argtomacro\PT@makeoptarg}
\def\PT@addargtomacro
  {\PT@add@argtomacro\PT@makearg}

\def\PT@add@argtomacro#1#2#3%
  {\expandafter\expandafter\expandafter\gdef
     \expandafter\expandafter\expandafter\PT@temp
       \expandafter\expandafter\expandafter{\csname #3\endcsname}%
   #1%
   \expandafter\PT@gaddendmacro\expandafter
     {\expandafter#2\expandafter}\expandafter{\PT@temp}}

\def\PT@makeoptarg%
  {\expandafter\def\expandafter\PT@temp\expandafter
     {\expandafter[\PT@temp]}}
\def\PT@makearg%
  {\expandafter\def\expandafter\PT@temp\expandafter
```

```
    {\expandafter{\PT@temp}}}

%
% \begin{macro}{\PT@mtimesn}
% Expands to |#1| times |#2|. (Work in progress.)
%    \begin{macrocode}
% \def\PT@mtimesn #1#2%
%   {\expandafter\PT@mtimtesn@\romannumeral #1011{#2}}
% \def\PT@mtimesn@ #1i#2%
%   {\if#1m%
%      #2\expandafter\PT@mtimesn@#1i#3}
```

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

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

`\PT@origomit` Save the original definition of omit.

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

`\PT@disableomitonce` Undefines the next use of omit.

```
\def\PT@disableomitonce
  {\def\omit
     {\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.

```
\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.

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

`\endpolytable` This is just defined for convenience.

```
\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.

```
\long\def\PT@scantoend #1\end #2%
  {\toks@\expandafter{\the\toks@ #1}%
   \def\PT@temp{#2}%
   \ifx\PT@temp\PT@environment
     \expandafter\PT@getwidths
   \else
     \toks@\expandafter{\the\toks@\end{#2}}%
     \expandafter\PT@scantoend
   \fi}
```

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

```
\def\PT@getwidths
```

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

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

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

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

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

```
   \column{@begin@}{@{}l@{}}
   \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`.

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

Now we are ready for a test run.

```
   \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.

```
   \ifx\column\PT@otherrun@column
   \else
      % we are in first run, print extra info
      \PT@typeout@{Number of columns: \the\PT@cols}%
      \PT@prelazylist
      \PT@typeout@{Column list: \Print\PT@allcols}%
      \PT@postlazylist
   \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.

```
\let\PT@firstrun@column\PT@otherrun@column
\let\savecolumns\PT@gobbleoptional
\let\restorecolumns\PT@gobbleoptional
\let\PT@savewidths\PT@gobbleoptional
\let\PT@restorewidths\PT@gobbleoptional
```

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

```
\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`.

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

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

```
\def\PT@savecounters
  {\begingroup
     \def\@elt ##1%
       {\global\csname c@##1\endcsname\the\csname c@##1\endcsname}%
     \xdef\PT@restorecounters{\cl@@ckpt}%
  \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.

```
\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 `\noexpand`ed versions of the list constructors.

```
  {\PT@prelazylist
   \edef\PT@sortedlist
     {\Foldr{\noexpand\Cons}{\noexpand\Nil}%
        {\Insertsort\PT@ltmax\PT@allcols}}%
   \PT@typeout@{Sorted columns: \Print\PT@sortedlist}%
   \PT@postlazylist
```

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

```
\PT@cols=0\relax%
\PT@prelazylist
\Execute{\Map\PT@numbercol\PT@sortedlist}%
\PT@postlazylist
\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.

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

Finally, we can typeset the table.

```
\PT@typeset}
```

`\PT@typeset`

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

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

```
 {\PT@typeout@{Typesetting the table ...}%
  \PT@prelazylist
  \edef\PT@temp{@{}\Execute{\Map\PT@preamble\PT@sortedlist}}%
  \PT@postlazylist
  %\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.

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

Now we start the tabular environment with the computed preamble.

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

Run, and this time, typeset, the contents.

```
  \the\toks@
```

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

```
  \PT@end
  \endgroup
  \PT@typeout@{Finished.}%
  \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.

```
\newcommand\PT@firstrun@column[3][0pt]%
  {\@ifundefined{PT@col@#2.type}%
      {\PT@typeout@{Defining column #2 at #1.}%
       \@namedef{PT@col@#2.type}{#3}%
       \@namedef{PT@col@#2.width}{#1}% initialize the width of the column
       % add the new column to the (sortable) list of all columns
       \PT@consmacro\PT@allcols{PT@col@#2}%
       \advance\PT@cols by 1\relax}%
      {\expandafter\ifx\csname PT@col@#2.type\endcsname\empty
         \relax % will be defined in a later table of the same set
       \else
         \PT@warning{Redefining column #2}%
       \fi
       \@namedef{PT@col@#2.type}{#3}%
       \expandafter\ifdim#1>0pt\relax
         \PT@typeout@{Redefining column #2 at #1.}%
         \@namedef{PT@col@#2.width}{#1}%
       \fi
      }%
```

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.

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

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

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

```
\def\PT@checkcoldefined #1%
  {\@ifundefined{PT@col@#1.type}%
     {\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.

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

We start by checking a switch.

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

Next, we check that both columns are defined.

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

Here, we check the first condition.

```
   \def\PT@temp{PT@col@#1}%
   \ifx\PT@currentcol\PT@temp
     \PT@typeout@{No need to skip columns.}%
   \else
     \PT@colwidth=\expandafter\@nameuse\expandafter
                    {\PT@currentcol.width}\relax
     \ifdim\PT@colwidth>\csname PT@col@#1.width\endcsname\relax
       % we need to change the width
       \PT@typeout@{s #1: old=\@nameuse{PT@col@#1.width} new=\the\PT@colwidth}%
       \PT@changedtrue
       \PT@enamedef{PT@col@#1.width}{\the\PT@colwidth}%
     \fi
```

The same for the untrusted `.max` values.

```
     \PT@colwidth=\expandafter\@nameuse\expandafter
                    {\PT@currentcol.max}\relax
     \ifdim\PT@colwidth>\csname PT@col@#1.max\endcsname\relax
       % we need to change the width
       \PT@typeout@{S #1: old=\@nameuse{PT@col@#1.max} new=\the\PT@colwidth}%
       \PT@changedtrue
       \PT@checkrerun
       \PT@enamedef{PT@col@#1.max}{\the\PT@colwidth}%
     \fi
     \ifnum\csname PT@col@#1.trusted\endcsname=\PT@false\relax
       \ifdim\PT@colwidth=\csname PT@col@#1.max\endcsname\relax
         \PT@typeout@{#1=\the\PT@colwidth\space is now trusted}%
         \expandafter\let\csname PT@col@#1.trusted\endcsname\PT@true%
```

```
      \fi
    \fi
  \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.

```
  \begingroup
  \ifmmode
    \let\d@llarbegin=$%$
    \let\d@llarend=$%$
    \let\col@sep=\arraycolsep
  \else
    \let\d@llarbegin=\begingroup
    \let\d@llarend=\endgroup
    \let\col@sep=\tabcolsep
  \fi
  %\def\PT@currentcol{PT@col@#1}%
  %\ifx\PT@currentcol\PT@nullcol
  %\else
  %  \PT@addbeginmacro\PT@currentpreamble{@{}}%
  %\fi
  \expandafter\expandafter\expandafter
    \def\expandafter\expandafter\expandafter\PT@currentpreamble
      \expandafter\expandafter\expandafter
        {\csname PT@col@#1.type\endcsname}%
  \setbox0=\hbox{%
    \expandafter\@mkpream\expandafter{\PT@currentpreamble}%
    \def\@sharp{\strut #3}%
    %\show\@preamble
    \@preamble}%
  \expandafter\gdef\expandafter\PT@temp\expandafter{\the\wd0}%
  \endgroup
```

Now begins the real comparison.

```
  \global\PT@colwidth=\@nameuse{PT@col@#1.width}%
  \global\advance\PT@colwidth by \PT@temp\relax%
  \ifdim\PT@colwidth>\csname PT@col@#2.width\endcsname\relax
    % we need to change the width
    \PT@typeout@{c #2:
                 old=\@nameuse{PT@col@#2.width}
                 new=\the\PT@colwidth}%
    \PT@changedtrue
    \PT@enamedef{PT@col@#2.width}{\the\PT@colwidth}%
  \fi
```

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

```
  \global\PT@colwidth=\@nameuse{PT@col@#1.max}%
  \global\advance\PT@colwidth by \PT@temp\relax%
```

```
  \ifdim\PT@colwidth>\csname PT@col@#2.max\endcsname\relax
    % we need to change the width
    \PT@typeout@{C #2:
                 old=\@nameuse{PT@col@#2.max}
                 new=\the\PT@colwidth}%
    \PT@changedtrue
    \PT@checkrerun
    \PT@enamedef{PT@col@#2.max}{\the\PT@colwidth}%
  \fi
  \ifnum\csname PT@col@#2.trusted\endcsname=\PT@false\relax
    \ifdim\PT@colwidth=\csname PT@col@#2.max\endcsname\relax
      \PT@typeout@{#2=\the\PT@colwidth\space is now trusted}%
      \expandafter\let\csname PT@col@#2.trusted\endcsname\PT@true%
    \fi
  \fi
```

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

```
  \def\PT@currentcol{PT@col@#2}%
  \let\PT@infromto\empty
  \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.

```
\def\PT@checkrerun
  {\ifnum\PT@inrestore=\PT@true\relax
     \PT@rerun
  \fi}
```

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

```
\newcommand*{\PT@resetcolumn}[1][]%
  {\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`.

```
\def\PT@nullcol
  {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.

```
\def\Execute{\Foldr\Sequence\empty}
\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.

```
\def\ShowColumn #1%
  {\ShowColumn@#1\ShowColumn@}
\def\ShowColumn@ PT@col@#1\ShowColumn@
  {#1 }
\def\StripColumn #1%
  {\expandafter\StripColumn@#1\StripColumn@}
\def\StripColumn@ PT@col@#1\StripColumn@
  {#1}
```

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

```
\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.

```
\def\PT@TeXif #1%
  {\expandafter\@gobble#1\relax
     \PT@gobblefalse
   \else\relax
     \gobbletrue
   \fi}
\def\PT@gobblefalse\else\relax\gobbletrue\fi #1#2%
  {\fi #1}
```

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

```
\def\PT@ltmax #1#2%
  {\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.

```
\def\PT@numbercol #1%
  {%\PT@typeout@{numbering #1 as \the\PT@cols}%
   \PT@enamedef{#1.num}{\the\PT@cols}%
   \def\PT@lastcol{#1}%
   \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 `\Execute`ing ... 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.

```
\def\PT@preamble #1%
%  {\csname #1.type\endcsname}
```

```
% {l}
  {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, things will go wrong.

```
\newcommand{\PT@resetandcr}[1][0pt]%
 {\ifx\PT@currentcol\PT@lastcol
  \else
    \ifx\PT@currentcol\PT@nullcol
      \edef\PT@currentcol{\Head{\Tail\PT@sortedlist}}%
    \fi
    \edef\PT@currentcol@{\StripColumn\PT@currentcol}%
    \edef\PT@lastcol@
      {\StripColumn\PT@lastcol}%
    \PT@typeout@{adding implicit fromto from \PT@currentcol@
                                  \space to \PT@lastcol@}%
    \expandafter\expandafter\expandafter\fromto
      \expandafter\expandafter\expandafter{%
        \expandafter\expandafter\expandafter\PT@currentcol@
          \expandafter}\expandafter{\PT@lastcol@}{}%
  \fi
  \PT@typeout@{Next line ...}%
  \PT@resetcolumn\\[#1]}
```

`\PT@multicolumn` All the `\fromto`s 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.

```
\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.

```
{%\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`.

```
  % skip to current position
  \global\PT@cols=\@nameuse{PT@col@#1.num}%
  \global\advance\PT@cols
    by -\expandafter\csname\PT@currentcol.num\endcsname\relax
  \ifx\PT@currentcol\PT@nullcol
    \global\advance\PT@cols by -1\relax%
  \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.

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

So there are extra tabs necessary.

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

We can use a multicolumn to save time.

```
      \global\advance\PT@cols by -1\relax
      \PT@typeout@{after next &, multicolumn \the\PT@cols\space blank}%
      \PT@NextCol
      \multicolumn{\the\PT@cols}{@{}l@{}}{}%
    \fi
    \PT@NextCol
  \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 `\multicolumn`s 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.

```
  \global\PT@cols=\@nameuse{PT@col@#2.num}%
  \global\advance\PT@cols by -\@nameuse{PT@col@#1.num}\relax%
  %\ifnum\PT@cols>1\relax
```

```
      % we always skip one column
      \PT@typeout@{after next &,
                   putting text in \the\PT@cols\space multicol}%
      \PT@typeout@{nf=#1 nt=#2 %
                   from=\@nameuse{PT@col@#1.num}
                   to=\@nameuse{PT@col@#2.num}}%
      \expandafter\global\expandafter\let\expandafter\PT@temp
        \csname PT@col@#1.type\endcsname%
      \PT@NextCol
      % use multicolumn
      \expandafter\multicolumn
        \expandafter{\expandafter\the\expandafter\PT@cols
        \expandafter}\expandafter{\PT@temp}{#3}%
      %\PT@typeout@{!!!!}%
    %\else
    %   \PT@NextCol
    %   #3%
    %\fi
```

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

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

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

```
\def\PT@NextCol
  {\PT@typeout@{ & }%
   &}%
```

`\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.

```
\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.

```
  {\PT@colwidth=\@nameuse{PT@col@#1.max}%
   \advance\PT@colwidth by -\expandafter\csname\PT@currentcol.max\endcsname
   \leavevmode
   \hb@xt@\PT@colwidth{%
     \expandafter\@mkpream\expandafter{@{}l@{}}%
     \let\@sharp\empty%
     %\show\@preamble
     \@preamble}%
% We continue by computing the width of the current entry.
%    \begin{macrocode}
   \PT@colwidth=\@nameuse{PT@col@#2.max}%
```

```
    \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.

```
    \ifmmode
      \PT@typeout@{*math mode*}%
      \let\d@llarbegin=$%$
      \let\d@llarend=$%$
      \let\col@sep=\arraycolsep
    \else
      \PT@typeout@{*text mode*}%
      \let\d@llarbegin=\begingroup
      \let\d@llarend=\endgroup
      \let\col@sep=\tabcolsep
    \fi
    %\def\PT@currentcol{PT@col@#1}%
    \expandafter\expandafter\expandafter
      \def\expandafter\expandafter\expandafter\PT@currentpreamble
        \expandafter\expandafter\expandafter
          {\csname PT@col@#1.type\endcsname}%
    %\ifx\PT@currentcol\PT@nullcol
    %\else
    %  \PT@addbeginmacro\PT@currentpreamble{@{}}%
    %\fi
```

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

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

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

```
    \def\PT@currentcol{PT@col@#2}%
    \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 argu-

ment 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, incorportating 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.

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

```
\def\PT@loadtable#1% #1 table id number
  {%\expandafter\show\csname PT@restore@\romannumeral #1\endcsname
   %\show\column
   \PT@typeout@
     {Calling \expandafter\string
                 \csname PT@restore@\romannumeral #1\endcsname.}%
   \let\maxcolumn\PT@setmaxwidth
   %\expandafter\show\csname PT@load@\romannumeral #1\endcsname
   \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.

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

```
\def\PT@saveinformation#1% #1 set name
  {\expandafter\def\expandafter\PT@temp\expandafter
     {\csname PT@widths@#1\endcsname}%
   \expandafter\def\expandafter\PT@temp\expandafter
     {\csname PT@restore@\romannumeral\PT@temp\endcsname}%
   \expandafter\gdef\PT@temp{}% start empty
   % this is: \Execute{\Map{\PT@savecolumn{\PT@temp}}\PT@sortedlist}
   \expandafter\Execute\expandafter{\expandafter
     \Map\expandafter{\expandafter\PT@savecolumn
       \expandafter{\PT@temp}}\PT@sortedlist}}
```

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

```
\def\PT@savecolumn#1#2% #1 macro name, #2 column name
  {\PT@typeout@{saving column #2 in \string #1 ...}%
   \def\PT@temp{#2}%
   \ifx\PT@temp\PT@nullcol
     \PT@typeout@{skipping nullcol ...}%
   \else
     \PT@typeout@{max=\csname #2.max\endcsname, %
                  width=\csname #2.width\endcsname, %
                  trusted=\csname #2.trusted\endcsname}%
     % we need the column command in here
     % we could do the same in \column, but then the location of
     % \save / \restore matters ...
     \PT@gaddendmacro{#1}{\maxcolumn}%
     \ifnum\csname #2.trusted\endcsname=\PT@true\relax
       \PT@gaddendmacro{#1}{[\PT@true]}%
     \fi
     \edef\PT@temp{\StripColumn{#2}}%
     \PT@addargtomacro{#1}{PT@temp}%
     \PT@addargtomacro{#1}{#2.max}%
     \PT@gaddendmacro{#1}{\column}%
     \PT@addoptargtomacro{#1}{#2.width}%
     \edef\PT@temp{\StripColumn{#2}}%
     \PT@addargtomacro{#1}{PT@temp}%
     \PT@addargtomacro{#1}{#2.type}%
```

```
      %\show#1%
    \fi
  }
```

\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.

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

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

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

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

\PT@comparewidths

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

\PT@trustedmax

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

\PT@equalwidths

```
\def\PT@equalwidths#1% #1 full column name
  {\@ifundefined{#1.max}{}%
```

```
      {\ifdim\csname #1.max\endcsname=\csname #1.width\endcsname\relax
         \PT@typeout@{col #1 is okay ...}%
       \else
         \PT@rerun% a rerun is needed
       \fi}}

\PT@verifywidths
\def\PT@verifywidths#1% #1 column width set name
  {\@ifundefined{PT@widths@#1}%
     {\PT@typeout@{Nothing to verify yet for set #1.}%
      \PT@typeout@{Scheduling set #1 for verification at end of document.}%
      \AtEndDocument{\PT@verifywidths{#1}}}%
     {\PT@typeout@{Verifying column width set #1.}%
      \expandafter\PT@verify@widths\expandafter
        {\csname PT@widths@#1\endcsname}{#1}}}

\def\PT@verify@widths#1#2% #1 set id number, #2 set name
  {\@ifundefined{PT@restore@\romannumeral #1}{}%
     {\begingroup
        \let\column\PT@firstrun@column
        \PT@cols=0\relax%
        \def\PT@allcols{\Nil}%
        \PT@loadtablebyname{#2}%
        \PT@table=#1\relax
        % nullcolumn is not loaded, therefore:
        \expandafter\def\csname\PT@nullcol .width\endcsname{0pt}%
        % checking trust
        \PT@prelazylist
        \All{\PT@trustedmax}{\PT@allcols}%
           {\PT@typeout@{All maximum widths can be trusted -- writing .max!}%
            \PT@save@table{.max}}%
           {\PT@typeout@{Untrustworthy maximums widths -- writing .width!}%
            \PT@rerun
            \PT@save@table{.width}}%
        \PT@postlazylist
      \endgroup}%
   \PT@typeout@{Verification for #2 successful.}}
```

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

```
\def\PT@save@table#1%
  {\PT@typeout@{Saving column width information.}%
   \if@filesw
     \PT@prelazylist
     {\immediate\write\@auxout{%
        \gdef\expandafter\noexpand
          \csname PT@restore@\romannumeral\PT@table\endcsname
            {\Execute{\Map{\PT@write@column{#1}}\PT@allcols}}}}%
     \PT@postlazylist
   \fi}
```

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

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

7.7 The user environments

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

```
\newenvironment{ptabular}[1][c]%
  {\def\PT@begin{\tabular[#1]}%
   \let\PT@end\endtabular
   \beginpolytable}
  {\endpolytable}

\newenvironment{parray}[1][c]%
  {\def\PT@begin{\array[#1]}%
   \let\PT@end\endarray
   \beginpolytable}
  {\endpolytable}

\def\pboxed{%
   \let\PT@begin\@gobble
   \let\PT@end\empty
   \let\PT@multicolumn\PT@placeinbox
   \expandafter\beginpolytable\ignorespaces}

\let\endpboxed\endpolytable
```

That is all.

⟨/package⟩