The polytable package

Andres Löh
polytable@andres-loeh.de

2013/07/08

Abstract

This package implements a variant of tabular/tabbing-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 `\\`.

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{ptboxed}
\defaultcolumn{l|}\column{.}{|l|}
\> left
\=3 first of three \> second of three \> third of three
\=r right \\
\defaultcolumn{r|}\> left \=2 middle 1/2 \> middle 2/2 \=r right \\
\> left \=3 middle 1/3 \> middle 2/3 \> middle 3/3 \=r right \\
\> left
\=2 first of two middle columns \> second of two middle columns
\=r right \\
\end{ptboxed}
```

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 `\\`. 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 `tabular` 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.

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.

```
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

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 Setup

New in v0.8: We allow to implicitly define columns by just using the names during table content specification, without having declared them formally using `\column` (see below).

`\nodefaultcolumn` By default, though, this behaviour is turned off, because the use of a mis-

spelled column is often an error. Thus, by default or after the command `\nodefaultcolumn` is used, all columns must be declared.

`\defaultcolumn` Using a statement of the form `\defaultcolumn{`⟨*spec*⟩`}`, implicit columns can be activated. All undefined columns that are encountered will be assumed to have format string ⟨*spec*⟩.

6.2 The environments

`pboxed` `ptboxed` `pmboxed` `ptabular` `parray` New in v0.8: There are now five environments that this package provides: in addition to the former `ptabular`, `parray`, and `pboxed`, there is now also `ptboxed` and `pmboxed`. The 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 (probably non-indented) paragraph before beginning a `pboxed`. All lines that a `pboxed` produces are of equal length, so it should be possible to center or right-align the material.

Both `ptboxed` and `pbmoxed` are like `pboxed`, but pre-wrapped in a `tabular` or `array` environment, respectively, and thus not page-breakable but less fragile (or should I just say: easier to use) than the unwrapped `pboxed`. With those, there is no need for `ptabular` and `parray` anymore – which were directly based on (and translated into) usual `tabular` and `array` environments as provided by the `array` package. The environments are still supported, to ensure compatibility, but they are mapped to `ptboxed` and `pmboxed`, respectively.

The `pmboxed` and `parray` environments assume math mode, whereas `ptboxed` and `ptabular` assume text mode. The `pboxed` environment works in both text and math modes and adapts to whatever is currently active.

I wrote in previous versions that one additional environment, namely `plongtable`, a poly-version of the `longtable` environment, was planned. Page-breaking of `pboxed` works fine, and I do not see a real need for a `plongtable` environment anymore. If you would like it anyway, feel free to mail me and inform me about the advantages it would have over `pboxed`.

The interface is the same for all of the environments.

6.3 The interface

(Note: this has changed significantly in v0.8!)

In each of the environments, the following commands can be used. All other commands should be used with care if placed directly inside such an environment: the contents of a polytable are processed multiple times; hence, if your commands generate output, the output will end up multiple times in the final document, and destroy your layout.

`\column` With `\column[`⟨*dimen*⟩`]{`⟨*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. However, it must only contain the description for *one* column.

As long as the save/restore feature (explained below) is not used, `\column` definitions are 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.

Columns must be defined before use when implicit columns using `\defaultcolumn` are disabled.

A minimal horizontal position ⟨*dimen*⟩ (relative to the beginning of the table) can be specified for a column, which defaults to 0pt.

`\=` The command `\=`{⟨*fromid*⟩}`[`⟨*spec*⟩`]` instructs the package to prepare for a new entry starting at column ⟨*fromid*⟩. Everything that follows the command, up to the next interface-command (except `\column`) or the end of the environment is interpreted as contents of the entry and can be arbitrary LaTeX code that has balanced grouping. The column specifier ⟨*spec*⟩ can be used to define a different formatting for this particular entry. If the specifier starts with an exclamation mark (`!`), it will be used as specifier for all entries of that column. The use of multiple exclamation-mark specifiers for the same column name gives precedence to the last one, but you should not rely on this behaviour as it may change in the future.

Note that, contrary to normal LaTeX behaviour, the second argument is the optional argument. Therefore, if an entry starts with an opening bracket (`[`), and the optional argument is omitted, a `\relax` should be inserted between command and opening bracket, as otherwise the entry would be treated as the optional argument.

`\>` The command `\>[`⟨*fromid*⟩`] [`⟨*spec*⟩`]` is a variant of `\=` where both arguments are optional. If no column name is given then the current column name, postfixed by a dot (`.`), is assumed as column name. At the beginning of a line, a single dot (`.`) will be assumed as a name. The ⟨*spec*⟩ argument has the same behaviour as for `\=`. Note that if the specifier is given, the column name must be given as well, but may be left empty if the default choice is desired. For instance, `\>[] [r]`, will move to the next column, which will be typeset right-aligned.

`\<` The command `\<[`⟨*fromid*⟩`]` ends the current entry at the boundary specified by ⟨*fromid*⟩. This macro can be used instead of `\>` or `\=` if an entry should be ended without immediately starting a new one. Differences in behaviour occur if ⟨*fromid*⟩ is associated with a non-trivial column format string. TODO: Improve this explanation.

`\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 `\\` (or, now deprecated, `\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 `\\[`⟨*dimen*⟩`]`, that will add ⟨*dimen*⟩ extra space between the lines.

6.4 A warning

The contents of the table are processed multiple times because the widths of the entries are measured. Global assignments 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.

6.5 Saving column width information

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    [2013/07/18 v0.8.5 '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 ...

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

```
\let\PT@original@And\And
\let\PT@original@Not\Not
\RequirePackage{lazylist}
\let\PT@And\And
\let\PT@Not\Not
\def\PT@prelazylist
  {\let\And\PT@And
   \let\Not\PT@Not}
\def\PT@postlazylist
  {\let\And\PT@original@And
   \let\Not\PT@original@Not}
\PT@postlazylist
\RequirePackage{array}
```

The option debug will cause (a considerable amount of) debugging output to be printed. The option info is a compromise, printing some status information for each table, but no real debugging information. The option silent, on the other hand, will prevent certain warnings from being printed.

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

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

```
\newdimen\PT@colwidth
\newcount\PT@cols
\newcount\PT@table
\newtoks\PT@toks
\newif\ifPT@changed
\newread\PT@in
\newwrite\PT@out
```

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 v0.9, 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
\def\PT@false{0}
\def\PT@true{1}
\let\PT@inrestore\PT@false
```

`\defaultcolumn` `\nodefaultcolumn` `\PT@defaultcolumnspec`

The macro `\PT@defaultcolumnspec` contains, if defined, the current default specifier that is assumed for undefined columns. The other two commands can be used to set the specifier.

```
\newcommand{\defaultcolumn}[1]{\gdef\PT@defaultcolumnspec{#1}}
\newcommand{\nodefaultcolumn}{\global\let\PT@defaultcolumnspec\undefined}
\DeclareOption{defaultcolumns}{\defaultcolumn{l}}
```

`\memorytables` `\disktables` `\PT@add` `\PT@split` `\PT@prearewrite` `\PT@preareread` `\PT@finalize`

These macros steer where the end-column queue is stored, which can be either in

memory or on disk. The default is on disk, because that's more reliable for large tables. There is a package option `memory` to make `\memorytables` the default.

```
\newcommand{\memorytables}{%
  \let\PT@preparewrite\@gobble
  \let\PT@add         \PT@addmem
  \let\PT@prepareread \PT@prepareread mem
  \let\PT@split       \PT@splitmem
  \let\PT@finalize    \relax
}
\newcommand{\disktables}{%
  \let\PT@preparewrite\PT@preparewritefile
  \let\PT@add         \PT@addfile
  \let\PT@prepareread \PT@prepareread file
  \let\PT@split       \PT@splitfile
  \let\PT@finalize    \PT@finalizefile
}
\DeclareOption{memory}{\AtEndOfPackage\memorytables}
\ProcessOptions
```

`\PT@debug` `\PT@info` `\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.

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

`\PT@aligndim` `\PT@aligncol`

The first is (almost) stolen from the tabularx-package, to nicely align dimensions in the log file. TODO: fix some issues. The other command is for column names.

```
\def\PT@aligndim#1#2#3\@@{%
  \ifnum#1=0
    \if #2p%
      \PT@aligndim@0.0pt\space\space\space\space\space\@@
    \else
      \PT@aligndim@#1#2#3\space\space\space\space\space\space\space\space\@@
    \fi
  \else
    \PT@aligndim@#1#2#3\space\space\space\space\space\space\space\space\@@
  \fi}

\def\PT@aligndim@#1.#2#3#4#5#6#7#8#9\@@{%
  \ifnum#1<10 \space\fi
  \ifnum#1<100 \space\fi
  \ifnum#1<\@m\space\fi
```

```
  \ifnum#1<\@M\space\fi
  #1.#2#3#4#5#6#7#8\space\space}

\def\PT@aligncol#1{%
  \PT@aligncol@#1\space\space\space\space\space\space\space\space\@@}

\def\PT@aligncol@#1#2#3#4#5#6#7#8#9\@@{%
  #1#2#3#4#5#6#7#8\space\space}
```

`\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}
```

`\PT@currentcol` `\PT@currentcolumn` Both macros are used during typesetting to store the current column. The differences are subtle. TODO: remove one of the two, if possible. The `\PT@currentcol` variant contains the internal name, whereas the `\PT@currentcolumn` variant contains the external name.

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 macro can be used to add something to the end of a control structure.

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

`\PT@expanded` This macro expands its second argument once before passing it to the first argument. It is like `\expandafter`, but works on grouped arguments instead of tokens.

```
\def\PT@expanded #1#2%
  {\expandafter\Twiddle\expandafter\Identity\expandafter{#2}{#1}}
```

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

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

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

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

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

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

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

\PT@addmem
\PT@splitmem
\PT@addfile
\PT@splitfile
\PT@preparereadmem
\PT@preparereadfile
\PT@preparewrite
\PT@finalizefile
\PT@queuefilename

The following macros handle a simple queue of names. With \PT@addmem, a name is added to the end of the queue. Using \PT@splitmem, the first element of the queue is bound to another command. As a replacement, we also define \PT@addfile and \PT@splitfile, that implement the queue in a file.

```
\def\PT@addmem#1#2{\PT@gaddendmacro #2{\PT@elt{#1}}}
\def\PT@splitmem#1#2{#1\PT@nil{#2}{#1}}

\def\PT@elt#1#2\PT@nil#3#4{\gdef #3{#1}\gdef #4{#2}}

\def\PT@queuefilename{\jobname.ptb}

% the \empty at the end consumes the newline space
\def\PT@addfile#1#2{%
  \immediate\write #2{\string\def\string\PTtemp{#1}\string\empty}}

\def\PT@splitfile#1#2{%
  \ifeof #1%
    \let #2=\empty
  \else
    \read #1 to#2%
    %\show #2%
    #2% hack, because it essentially ignores #2
    \PT@expanded{\def #2}{\PTtemp}%
    %\show #2%
  \fi}

```

```
%\def\strip#1{\def #1{\expandafter\@strip #1\@dummy}}
%\def\@strip#1\@dummy{#1}

\def\PT@preparereadmem#1#2{%
  \global\let #1=#2}

\def\PT@preparewritefile#1{%
  \immediate\openout\PT@out\PT@queuefilename\relax
  \let #1\PT@out}

\def\PT@preparereadfile#1#2{%
  \immediate\closeout\PT@out
  \openin\PT@in\PT@queuefilename\relax
  \let #1\PT@in}

\def\PT@finalizefile{%
  \closein\PT@in}

\disktables
```

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 table is typeset by producing boxes of appropriate size.

`\beginpolytable` This macro starts the environment. It should, however, not be called directly, but rather in a LaTeX environment. We 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
   \PT@toks{}% initialise token register
   \PT@scantoend}
```

`\endpolytable` This is just defined for convenience.

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

`\PT@scantoend` Whenenver an `\end` is encountered, we check if it really ends the current environment. We store the tokens we have read. Once we have found the end of the environment, we initialize the column queue and column reference, `\PT@columnqueue` and `\PT@columnreference`. The new interface commands build the queue during

the first test run, containing the end columns of specific entries of the table. Later, the queue is copied to be the reference, and in the typesetting run, the information is used to compute the correct box widths. We start with an empty queue, and set the reference to `\undefined`, because it is not yet needed. TODO: queue and reference must be global variables at the moment; try to change that.

```
\newcommand{\PT@scantoend}% LaTeX check
\long\def\PT@scantoend #1\end #2%
  {\PT@toks\expandafter{\the\PT@toks #1}%
   \def\PT@temp{#2}%
   \ifx\PT@temp\PT@environment
     \global\let\PT@columnqueue     \empty
     \global\let\PT@columnreference \undefined
     \PT@preparewrite\PT@columnqueue
     \expandafter\PT@getwidths
   \else
     \PT@toks\expandafter{\the\PT@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. We also define a pseudo-column `@end@` denoting the end of a row (as of v0.8; and I'm not sure if `@begin@` is still needed).

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

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

```
   \let\fromto          \PT@fromto
   \let\PT@processentry \PT@checkwidth
   \let\PT@scanbegin    \PT@scanbeginfree
   \let\\=              \PT@resetcolumn
   \let\nextline        \PT@resetcolumn
   \let\>=              \PT@fromopt
   \let\==              \PT@from
   \let\<=              \PT@toopt
```

```
  \global\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\PT@toks
  \@ifundefined{PT@scanning}%
    {}{\PT@resetcolumn\relax}%
```

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@prelazylist
    \PT@typeout@{\PT@environment: \the\PT@cols\space columns, %
                                  \PT@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
    \PT@typeout@{There were changes; another trial run needed.}%
    \expandafter\PT@getwidths
  \else
    % we are done and can do the sorting
    \PT@typeout@{There were no changes; 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:}%
   \PT@PrintWidth\PT@sortedlist
   \PT@postlazylist
```

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

```
   \PT@cols=0\relax%
   \PT@prelazylist
   \PT@Execute{\Map\PT@numbercol\PT@sortedlist}%
   \PT@postlazylist
   \edef\PT@lastcol@{\PT@StripColumn\PT@lastcol}%
   \PT@typeout@{Numbered successfully, %
                last column is \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 .}%
         \PT@expanded\PT@saveinformation\PT@currentwidths
      \fi
```

Finally, we can typeset the table.

```
   \PT@typeset}
```

\PT@typeset We redefine `\fromto` and `\\` to their final meaning in the typesetting process. The `\fromto` statements will be replaced by appropriate calls to `\multicolumn`, whereas the `\\` will again reset the counter for the current column, but also call the table environment's newline macro. Again, `\nextline` is allowed as an old name for `\\`.

```
\def\PT@typeset
  {\PT@typeout@{Typesetting the table ...}%
   \let\PT@processentry          \PT@placeinbox
   \let\PT@scanbegin             \PT@scanbeginwidth
   \let\\=                       \PT@resetandcr
   \let\nextline                 \PT@resetandcr
```

```
\PT@prepareread\PT@columnreference\PT@columnqueue
```

The environment that will be opened soon, can, if if happens to be `tabular` or `array`, redefines `\\` once more, and will redefine it to `\@arraycr`. To prevent this, we also set `\@arraycr` to our command.

```
\let\@arraycr                    \PT@resetandcr
```

The array environments keep each line in a group; therefore `\PT@resetcolumn`, when executed during the linebreak, will not affect the current column counters. By having `\PT@resetcolumn` before entering the environment, we ensure that the group reset will have the correct effect anyway.

```
\PT@resetcolumn % we are at the beginning of a line
```

Now we start the tabular environment with the computed preamble. We redefine the `\\` to whatever the environment dictates.

```
\PT@begin%
```

Run, and this time, typeset, the contents.

```
\the\PT@toks
\PT@fill% new in 0.7.3: balance the last line
```

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

```
\PT@finalize% finalize the queue (possibly close file)
\PT@end
\endgroup
\PT@typeout@{Finished.}%
\expandafter\end\expandafter{\PT@environment}}%
```

7.3 New interface

From v0.8 on, we offer a more convenient user interface for the programming of the columns.

`\PT@from` `\PT@fromopt` `\PT@toopt`

The macro `\PT@from` is bound to `\=` in a polytable environment, and used to move to the named column specified by its argument. The previous column is closed. The variant `\PT@fromopt` is bound to `\>` and takes an optional argument instead of a mandatory, which defaults to the current column name followed by a single dot `.`. We use an empty default which is then redefined to make it easier for the user to use the default definition (TODO: explain better). Otherwise, it is like `\PT@from`.

The macro `\PT@toopt` is bound to `\<`. Where `\PT@from` ends an entry if one is active, and starts a new one, the `\PT@toopt` variant only ends the currently active entry.

```
\newcommand{\PT@from}[1]%
  {\PT@checkendentry{#1}\PT@dofrom{#1}}

\newcommand{\PT@fromopt}[1][]%
  {\def\PT@temp{#1}%
   \ifx\PT@temp\empty
     % set default column name
```

```
      \def\PT@temp{\PT@currentcolumn .}%
    \fi
    \PT@expanded\PT@from\PT@temp}

\newcommand{\PT@toopt}[1][]%
  {\def\PT@temp{#1}%
   \ifx\PT@temp\empty
     % set default column name
     \def\PT@temp{\PT@currentcolumn .}%
   \fi
   \PT@expanded\PT@checkendentry\PT@temp
   \let\PT@scanning\undefined}
```

\PT@dofrom

```
\newcommand*{\PT@dofrom}[1]%
  {\edef\PT@currentcolumn{#1}%
   \let\PT@scanning\PT@currentcolumn
   \let\PT@currentpreamble\relax% necessary for preparescan
   \@ifnextchar[%]
     {\PT@expanded\PT@dospecfrom\PT@currentcolumn}%
     {\PT@expanded\PT@dodofrom  \PT@currentcolumn}}

\newcommand*{\PT@dospecfrom}{}% LaTeX check
\def\PT@dospecfrom #1[#2]%
  {\PT@checkglobalfrom #2\PT@nil{#1}%
   \PT@dodofrom{#1}}

\newcommand*{\PT@checkglobalfrom}{}% LaTeX check
\def\PT@checkglobalfrom
  {\@ifnextchar!\PT@getglobalfrom\PT@ignorefrom}

\newcommand*{\PT@getglobalfrom}{}% LaTeX check
\def\PT@getglobalfrom!#1\PT@nil#2%
  {\column{#2}{#1}}

\newcommand*{\PT@ignorefrom}{}% LaTeX check
\def\PT@ignorefrom #1\PT@nil#2%
  {\def\PT@currentpreamble{#1}}

\newcommand*{\PT@dodofrom}[1]%
  {\@ifundefined{PT@columnreference}%
     {% trial run
      \ifx\column\PT@otherruncolumn
      \else
        % first run
        \let\PT@storeendcolumn\PT@add
      \fi
      \def\PT@temp{@end@}}%
     {% final run
      \PT@split\PT@columnreference\PT@temp
```

```
      %\PT@typeout@{splitted: \PT@temp}
     }%
   \PT@expanded{\PT@expanded\PT@preparescan\PT@currentcolumn}\PT@temp
   \PT@scanbegin}

\let\PT@storeendcolumn\@gobbletwo
```

Here, `\PT@scanbegin` will scan free or using the width, depending on the run we are in.

`\PT@fromto` This is the implementation for the old explicit `\fromto` macro. It takes the start and end columns, and the contents. It can be used for all runs.

```
\newcommand*{\PT@fromto}[3]%
```

We allow a `\fromto` to follow a new-style command, but we reset the current column to undefined, so no text may immediately follow a `\fromto` command.

```
  {\PT@checkendentry{#1}%
   \let\PT@scanning\undefined
```

We check a switch to prevent nested `\fromto`s.

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

Here, the real work is done:

```
   \let\PT@currentpreamble\relax% necessary for preparescan
   \PT@preparescan{#1}{#2}%
   \PT@scanbegin #3\PT@scanend% defines \@curfield
   \PT@processentry{#1}{#2}%
```

The commands `\PT@scanbegin` and `\PT@processentry` will perform different tasks depending on which run we are in.

We ignore spaces after the `\fromto` command.

```
   \let\PT@infromto\empty
   \ignorespaces}
```

`\PT@checkendentry` This macro checks if we are currently scanning an entry. If so (this is detected by checking if `\PT@scanning` is defined), we close the entry and handle it (in `\PT@endentry`) before returning. The argument is the name of the column from which this macro is called.

```
\newcommand*{\PT@checkendentry}% takes one argument
  {\@ifundefined{PT@scanning}%
     {\let\PT@temp\@gobble}%
     {\let\PT@temp\PT@endentry}%
   \PT@temp}
%\newcommand*{\PT@checkendentry}% takes one argument
%  {\@ifundefined{PT@post@preamble}%
%     {\let\PT@temp\PT@discardentry}%
%     {\let\PT@temp\PT@endentry}%
%   \PT@temp}

```

```
%\newcommand*{\PT@discardentry}[1]%
%  {\let\PT@postpreamble=\empty\PT@scanend}

\newcommand*{\PT@endentry}[1]%
  {\PT@scanend
   \edef\PT@temp{#1}%
   \PT@expanded\PT@storeendcolumn\PT@temp\PT@columnqueue
   \let\PT@storeendcolumn\@gobbletwo
   \PT@expanded{\PT@expanded\PT@processentry\PT@currentcolumn}\PT@temp}
```

7.4 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 \PT@aligncol{#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
         \begingroup
         \def\PT@temp{PT@col@#2}%
         \ifx\PT@temp\PT@endcol
           \relax % end column is always redefined
         \else
           \PT@warning{Redefining column #2}%
         \fi
         \endgroup
       \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. New in v0.8: this macro implicitly defines the column if we have a default column specifier.

```
\def\PT@checkcoldefined #1%
  {\@ifundefined{PT@col@#1.type}%
      {\@ifundefined{PT@defaultcolumnspec}%
          {\PackageError{polytable}{Undefined column #1}{}}
          {\PT@debug@{Implicitly defining column #1}%
           \PT@expanded{\column{#1}}{\PT@defaultcolumnspec}}}{}%
```

We also have to define columns with empty specifiers. This situation can occur if save/restoring columns that are defined by default specifiers.

```
  \expandafter\ifx\csname PT@col@#1.type\endcsname\empty\relax
     \@ifundefined{PT@defaultcolumnspec}{}%
     {\PT@debug@{Implicitly defining column #1}%
      \PT@expanded{\column{#1}}{\PT@defaultcolumnspec}}%
  \fi}
```

`\PT@checkwidth` 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@checkwidth #1#2%
  {\PT@checkcoldefined{#2}% first column should have been checked before
```

Here, we check the first condition.

```
  \def\PT@temp{PT@col@#1}%
  \ifx\PT@currentcol\PT@temp
    \PT@debug@{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@debug@{s \PT@aligncol{#1}: %
                 old=\expandafter\expandafter\expandafter
                       \PT@aligndim\csname PT@col@#1.width\endcsname\@@%
                 new=\expandafter\PT@aligndim\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@debug@{S \PT@aligncol{#1}: %
                 old=\expandafter\expandafter\expandafter
                       \PT@aligndim\csname PT@col@#1.max\endcsname\@@%
                 new=\expandafter\PT@aligndim\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@debug@{#1=\the\PT@colwidth\space is now trusted}%
        \expandafter\let\csname PT@col@#1.trusted\endcsname\PT@true%
      \fi
    \fi
  \fi
```

We assume that the current field is typeset in `\@curfield`; we can thus measure the width of the box and then test the second condition.

```
  \PT@expanded{\def\PT@temp}{\the\wd\@curfield}%
  \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@debug@{#2 (width \PT@temp) starts after #1 (at \csname PT@col@#1.width\endcsname)}%
    \PT@debug@{c \PT@aligncol{#2}: %
               old=\expandafter\expandafter\expandafter
                     \PT@aligndim\csname PT@col@#2.width\endcsname\@@%
               new=\expandafter\PT@aligndim\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@debug@{C \PT@aligncol{#2}: %
               old=\expandafter\expandafter\expandafter
                     \PT@aligndim\csname PT@col@#2.max\endcsname\@@%
               new=\expandafter\PT@aligndim\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@debug@{#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`.

```
  \def\PT@currentcol{PT@col@#2}}
```

`\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` If the end of a line is encountered, we stop scanning the current entry, and reset the current column.

```
\newcommand*{\PT@resetcolumn}[1][]%
  {\PT@checkendentry{@end@}%
   \let\PT@currentcolumn\empty%
   \let\PT@scanning\undefined
   \let\PT@currentcol\PT@nullcol
   % TODO: remove these lines if they don't work
   %\let\PT@pre@preamble\empty
   %\PT@scanbeginfree
  }
```

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

```
\def\PT@nullcol{PT@col@@begin@}
\def\PT@endcol{PT@col@@end@}
```

7.5 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.

`\PT@Execute` `\PT@Sequence` With `\PT@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\PT@Execute{\Foldr\PT@Sequence\empty}
\def\PT@Sequence #1#2{#1#2}
```

`\PT@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\PT@ShowColumn #1#2%
  {\PT@ShowColumn@{#1}#2\PT@ShowColumn@}
\def\PT@ShowColumn@ #1PT@col@#2\PT@ShowColumn@
  {#1{#2} }
\def\PT@ShowColumnWidth #1%
  {\PT@typeout@{%
     \PT@ShowColumn\PT@aligncol{#1}:
     \expandafter\expandafter\expandafter
       \PT@aligndim\csname #1.max\endcsname\@@}}
\def\PT@StripColumn #1%
  {\expandafter\PT@StripColumn@#1\PT@StripColumn@}
\def\PT@StripColumn@ PT@col@#1\PT@StripColumn@
  {#1}
```

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

```
\def\PT@Print#1{\PT@Execute{\Map{\PT@ShowColumn\Identity}#1}}
\def\PT@PrintWidth#1{\PT@Execute{\Map\PT@ShowColumnWidth#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%
  {\Not{\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}
```

7.6 Typesetting the table

Remember that there are three important macros that occur in the body of the polytable: \column, \fromto, and \\. 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 \\ does in the typesetting phase. It resets the current column, but it also calls the surrounding environment's newline macro \PT@cr ... 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}%
  {\PT@expanded\PT@checkendentry\PT@lastcol@%
   \ifx\PT@currentcol\PT@lastcol
   \else
     \ifx\PT@currentcol\PT@nullcol
       \edef\PT@currentcol{\Head{\Tail\PT@sortedlist}}%
     \fi
     \edef\PT@currentcol@{\PT@StripColumn\PT@currentcol}%
     \PT@typeout@{adding implicit fromto at eol from \PT@currentcol@
                                          \space to \PT@lastcol@}%
     \PT@expanded{\PT@expanded\fromto\PT@currentcol@}\PT@lastcol@
   \fi
   \PT@typeout@{Next line ...}%
   \let\PT@scanning\undefined% needed for resetcolumn
   \PT@resetcolumn\PT@cr}
```

\PT@fill This variant of \PT@resetandcr is used at the end of the environment, to insert a blank box for the pboxed environment to balance the widths of all lines. It does not start a new line, and does nothing if the current column is @begin@. TODO: extract commonalities with \PT@resetandcr into a separate macro.

```
\newcommand{\PT@fill}%
  {\PT@expanded\PT@checkendentry\PT@lastcol@%
   \ifx\PT@currentcol\PT@lastcol
   \else
     \ifx\PT@currentcol\PT@nullcol
     \else
       \edef\PT@currentcol@{\PT@StripColumn\PT@currentcol}%
       \PT@typeout@{adding implicit fromto from \PT@currentcol@
```

```
                                        \space to \PT@lastcol@}%
    \PT@expanded{\PT@expanded\fromto\PT@currentcol@}\PT@lastcol@
  \fi\fi}
```

\PT@placeinbox This macro is the final-run replacement for \PT@checkwidth. We use the pre-computed width information to typeset the contents of the table in aligned boxes. The arguments are the same as for \PT@checkwidth, i.e., the start and the end columns, and the assumption that the entry is contained in the box \@curfield.

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

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
 \edef\PT@temp{\PT@StripColumn\PT@currentcol}%
 \PT@typeout@{adding space of width %
              \expandafter\PT@alignd im\the\PT@colwidth\@@
              (\expandafter\PT@aligncol\expandafter{\PT@temp} %
              -> \PT@aligncol{#1})}%
 \hb@xt@\PT@colwidth{%
   {\@mkpream{@{}l@{}}\@addtopreamble\@empty}%
   \let\CT@row@color\relax% colortbl compatibility
   \let\@sharp\empty%
   %\show\@preamble
   \@preamble}%
```

The important part is to use the pre-typeset box \@curfield. This produces real output!

```
 \PT@typeout@{adding box \space\space of width %
              \expandafter\PT@alignd im\the\wd\@curfield\@@
              (\PT@aligncol{#1} -> \PT@aligncol{#2})}%
 \box\@curfield
```

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

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

\PT@preparescan The macro \PT@preparescan sets the two macros \PT@scanbegin and \PT@scanend in such a way that they scan the input between those two macros and place it in a box. The width of the box is determined from the given column names. The name @end@ can be used as a column name is a free scan (a scan without knowing the real end column) is desired. To allow redefinition of the preamble, we assume that \PT@currentpreamble is defined to \relax if we want it set normally dugin \PT@preparescan.

```
\def\PT@preparescan#1#2%
% First, we check that both columns are defined. This will
% actually define the columns if implicit column definitions are
% enabled.
%    \begin{macrocode}
```

```
 {\PT@checkcoldefined{#1}%
  \PT@checkcoldefined{#2}%
  \PT@colwidth=\@nameuse{PT@col@#2.max}%
  \advance\PT@colwidth by -\@nameuse{PT@col@#1.max}\relax%
  \ifmmode
    \PT@debug@{*math mode*}%
    \let\d@llarbegin=$%$
    \let\d@llarend=$%$
    \let\col@sep=\arraycolsep
  \else
    \PT@debug@{*text mode*}%
    \let\d@llarbegin=\begingroup
    \let\d@llarend=\endgroup
    \let\col@sep=\tabcolsep
  \fi
  \ifx\PT@currentpreamble\relax
    \PT@expanded{\PT@expanded{\def\PT@currentpreamble}}%
                {\csname PT@col@#1.type\endcsname}%
  \fi
```

Now, we make a preamble using the macro `\@mkpream` from the array package. This macro takes a format string as argument, and defines `\@preamble` as a result, where `\@sharp` occurs in the positions of the column contents. We perform the operation in a group to prevent certain redefinitions from escaping. The `\@preamble` is set globally anyway.

```
  {\PT@expanded\@mkpream\PT@currentpreamble%
   \@addtopreamble\@empty}%
  \let\CT@row@color\relax% colortbl compatibility
```

We split the preamble at the position of the `\@sharp`, using some tricks to make sure that there really is precisely one occurrence of `\@sharp` in the resulting preamble code, and producing an error otherwise. The splitting defines `\PT@pre@preamble` and `\PT@post@preamble`. With those and the computed `\PT@colwidth`, the scan is successfully prepared.

```
  \expandafter\PT@splitpreamble\@preamble\@sharp\PT@nil}
```

We now define the splitting of the preamble.

```
\def\PT@splitpreamble #1\@sharp #2\PT@nil{%
 \let\@sharp=\relax% needed for the following assignment
 \def\PT@terp{#2}%
 \ifx\PT@terp\empty%
   \PackageError{polytable}{Illegal preamble (no columns)}{}%
 \fi
 \PT@splitsplitpreamble{#1}#2\PT@nil}

\def\PT@splitsplitpreamble #1#2\@sharp #3\PT@nil{%
 \def\PT@temp{#3}%
 \ifx\PT@temp\empty%
 \else
   \PackageError{polytable}{Illegal preamble (multiple columns)}{}%
```

```
  \fi
  \def\PT@pre@preamble{#1}%
  \def\PT@post@preamble{#2}}%
```

Finally, we can define the scan environment, which depends on all the other macros being defined correctly. The macro `\PT@scanbegin` is not defined directly, but will be set to `\PT@scanbeginfree` during the trial runs and to `\PT@scanbeginwidth` during the final run.

```
\def\PT@scanbeginwidth
   {\PT@scanbegin@{\hbox to \PT@colwidth}}

\def\PT@scanbeginfree
   {\PT@scanbegin@{\hbox}}

\def\PT@scanbegin@#1%
   {\setbox\@curfield #1%
    \bgroup
    \PT@pre@preamble\strut\ignorespaces}

\def\PT@scanend
   {\PT@post@preamble
    \egroup}
```

7.7 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.

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, 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.

```
\def\PT@loadtablebyname#1% #1 set name
  {\PT@typeout@{Loading table information for column width set #1.}%
   \PT@loadtable{\csname PT@widths@#1\endcsname}}%
```

`\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
  {\PT@expanded{\def\PT@temp}{\csname PT@widths@#1\endcsname}%
   \PT@expanded{\def\PT@temp}%
               {\csname PT@restore@\romannumeral\PT@temp\endcsname}%
   \expandafter\gdef\PT@temp{}% start empty
   % this is: \PT@Execute{\Map{\PT@savecolumn{\PT@temp}}{\Reverse\PT@allcols}}
```

```
    \expandafter\PT@Execute\expandafter{\expandafter
      \Map\expandafter{\expandafter\PT@savecolumn
        \expandafter{\PT@temp}}{\Reverse\PT@allcols}}}
```

\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 ...}%
% This was a bug: end column cannot be skipped, because
% it can change.
%   \else\ifx\PT@temp\PT@endcol
%     \PT@typeout@{skipping endcol ...}%
   \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{\PT@StripColumn{#2}}%
     \PT@addargtomacro{#1}{PT@temp}%
     \PT@addargtomacro{#1}{#2.max}%
     \PT@gaddendmacro{#1}{\column}%
     \PT@addoptargtomacro{#1}{#2.width}%
     \edef\PT@temp{\PT@StripColumn{#2}}%
     \PT@addargtomacro{#1}{PT@temp}%
     \PT@addargtomacro{#1}{#2.type}%
     %\show#1%
%  \fi
   \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 \PT@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.}%
      \PT@expanded\PT@verify@widths{\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:
      \@namedef{\PT@nullcol .width}{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
            {\PT@Execute{\Map{\PT@write@column{#1}}{\Reverse\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%
    {\PT@StripColumn{#2}}%
    {\@nameuse{#2#1}}}%
```

7.8 The user environments

It remains to define the environments to be called by the user. New in v0.8: we add the environments `ptboxed` and `pmboxed` for text-mode and math-mode boxed environments. In turn, we remove `ptabular` and `parray`, and make the point to their new counterparts.

```
\def\pboxed{%
   \let\PT@begin       \empty
   \let\PT@end         \empty
```

The following assignment is a hack. If `pboxed` is called from within another `tabular`- or `array`-environment, then this sometimes does the right thing.

```
    \ifx\\\PT@arraycr
      \let\PT@cr        \PT@normalcr
    \else
      \let\PT@cr        \\%
    \fi
    \expandafter\beginpolytable\ignorespaces}

\let\endpboxed\endpolytable

\def\ptboxed{%
    \def\PT@begin       {\tabular{@{}l@{}}}%
    \let\PT@end         \endtabular
    \let\PT@cr          \@arraycr
    \expandafter\beginpolytable\ignorespaces}

\let\endptboxed\endpolytable

\def\pmboxed{%
    \def\PT@begin       {\array{@{}l@{}}}%
    \let\PT@end         \endarray
    \let\PT@cr          \@arraycr
    \expandafter\beginpolytable\ignorespaces}

\let\endpmboxed\endpolytable

\let\ptabular     \ptboxed
\let\endptabular  \endptboxed
\let\parray       \pmboxed
\let\endparray    \endpmboxed

```

That is all.

⟨/package⟩