The **polytable** package

Andres Löh

`polytable@andres-loeh.de`

2005/04/26

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.

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

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

Figure 3: Haskell code example

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.

`\\`   The command `\\` (or, now deprecated, `\nextline`) ends one line and begins
`\nextline`

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 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.

6.5   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   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

```
1 ⟨*package⟩
2 \NeedsTeXFormat{LaTeX2e}
3 \ProvidesPackage{polytable}%
4   [2005/04/26 v0.8.2 '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 . . .

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

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

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

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

— 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".

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

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

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

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

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

Figure 4: Commented Haskell code example

8

```
 5 \let\PT@original@And\And
 6 \RequirePackage{lazylist}
 7 \let\PT@And\And
 8 \def\PT@prelazylist
 9   {\let\And\PT@And}
10 \def\PT@postlazylist
11   {\let\And\PT@original@And}
12 \PT@postlazylist
13 \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.

```
14 \DeclareOption{debug} {\AtEndOfPackage\PT@debug}
15 \DeclareOption{info}  {\AtEndOfPackage\PT@info}
16 \DeclareOption{silent}{\AtEndOfPackage\PT@silent}
```

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

```
17 \newdimen\PT@colwidth
18 \newcount\PT@cols
19 \newcount\PT@table
20 \newtoks\PT@toks
21 \newif\ifPT@changed
22 \newread\PT@in
23 \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.

```
24 \def\PT@allcols{\Nil}
25 %\def\PT@allpubliccols{\Nil}
26 \let\PT@infromto\empty
```

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

```
27 \let\PT@currentwidths\empty
28 \def\PT@false{0}
29 \def\PT@true{1}
30 \let\PT@inrestore\PT@false
```

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

```
31 \newcommand{\defaultcolumn}[1]{\gdef\PT@defaultcolumnspec{#1}}
32 \newcommand{\nodefaultcolumn}{\global\let\PT@defaultcolumnspec\undefined}
33 \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.

```
34 \newcommand{\memorytables}{%
35   \let\PT@preparewrite\@gobble
```

```
36    \let\PT@add          \PT@addmem
37    \let\PT@prepareread  \PT@preparereadmem
38    \let\PT@split        \PT@splitmem
39    \let\PT@finalize     \relax
40 }
41 \newcommand{\disktables}{%
42    \let\PT@preparewrite\PT@preparewritefile
43    \let\PT@add          \PT@addfile
44    \let\PT@prepareread  \PT@preparereadfile
45    \let\PT@split        \PT@splitfile
46    \let\PT@finalize     \PT@finalizefile
47 }
48 \DeclareOption{memory}{\AtEndOfPackage\memorytables}
49 \ProcessOptions
```

\PT@debug Similar to the tabularx package, we add macros to print debugging information to
\PT@info the log. Depending on package options, we can set or unset them.
\PT@debug@
\PT@typeout@
```
50 \newcommand*{\PT@debug}
51    {\def\PT@debug@ ##1{\typeout{(polytable) ##1}}
```
\PT@silent
\PT@warning
```
52     \PT@info}
53 \newcommand*{\PT@info}
54    {\def\PT@typeout@ ##1{\typeout{(polytable) ##1}}}
55 \let\PT@debug@\@gobble
56 \let\PT@typeout@\@gobble
57 \def\PT@warning{\PackageWarning{polytable}}%
58 \def\PT@silent
59    {\let\PT@typeout@\@gobble\let\PT@warning\@gobble}
```

\PT@aligndim The first is (almost) stolen from the tabularx-package, to nicely align dimensions
\PT@aligncol in the log file. TODO: fix some issues. The other command is for column names.
```
60 \def\PT@aligndim#1#2#3\@@{%
61    \ifnum#1=0
62      \if #2p%
63        \PT@aligndim@0.0pt\space\space\space\space\space\@@
64      \else
65        \PT@aligndim@#1#2#3\space\space\space\space\space\space\space\space\@@
66      \fi
67    \else
68      \PT@aligndim@#1#2#3\space\space\space\space\space\space\space\space\@@
69    \fi}
70
71 \def\PT@aligndim@#1.#2#3#4#5#6#7#8#9\@@{%
72    \ifnum#1<10 \space\fi
73    \ifnum#1<100 \space\fi
74    \ifnum#1<\@m\space\fi
75    \ifnum#1<\@M\space\fi
76    #1.#2#3#4#5#6#7#8\space\space}
77
78 \def\PT@aligncol#1{%
```

```
79    \PT@aligncol@#1\space\space\space\space\space\space\space\space\@@}
80
81 \def\PT@aligncol@#1#2#3#4#5#6#7#8#9\@@{%
82    #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.

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

\PT@currentcol Both macros are used during typesetting to store the current column. The differ-
\PT@currentcolumn ences 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 This assumes that #2 is a list macro and #3 is a new list element. The macro
\PT@consmacro #2 should, after the call, expand to the list with the new element #1ed. Because
\PT@appendmacro we don't know the number of tokens in #3, we use a temporary macro **\PT@temp** (which is used frequently throughout the package).

```
89 \def\PT@listopmacro #1#2#3% #1 #3 to the list #2
90    {\def\PT@temp{#1{#3}}%
91     \expandafter\expandafter\expandafter
92       \def\expandafter\expandafter\expandafter
93       #2\expandafter\expandafter\expandafter
94       {\expandafter\PT@temp\expandafter{#2}}}
95
96 \def\PT@consmacro{\PT@listopmacro\Cons}
97 \def\PT@appendmacro{\PT@listopmacro\Cat}
```

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

```
98 \def\PT@gaddendmacro #1#2% add #2 to the end of #1
99    {\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.

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

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

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

```
\PT@addoptargtomacro
    \PT@addargtomacro  104 \def\PT@addoptargtomacro
                       105    {\PT@add@argtomacro\PT@makeoptarg}
                       106 \def\PT@addargtomacro
                       107    {\PT@add@argtomacro\PT@makearg}
                       108
                       109 \def\PT@add@argtomacro#1#2#3%
                       110    {\PT@expanded{\PT@expanded{\gdef\PT@temp}}{\csname #3\endcsname}%
                       111     #1%
                       112     \PT@expanded{\PT@gaddendmacro{#2}}{\PT@temp}}
                       113
                       114 \def\PT@makeoptarg%
                       115    {\PT@expanded{\def\PT@temp}{\expandafter[\PT@temp]}}
                       116 \def\PT@makearg%
                       117    {\PT@expanded{\def\PT@temp}{\expandafter{\PT@temp}}}

  \PT@gobbleoptional   Gobbles one optional argument. Ignores spaces.
                       118 \newcommand*{\PT@gobbleoptional}[1][]{\ignorespaces}

        \PT@addmem     The following macros handle a simple queue of names.  With \PT@addmem, a
        \PT@splitmem   name is added to the end of the queue.  Using \PT@splitmem, the first element
        \PT@addfile    of the queue is bound to another command.  As a replacement, we also define
        \PT@splitfile  \PT@addfile and \PT@splitfile, that implement the queue in a file.
 \PT@preparereadmem   119 \def\PT@addmem#1#2{\PT@gaddendmacro #2{\PT@elt{#1}}}
 \PT@preparereadfile  120 \def\PT@splitmem#1#2{#1\PT@nil{#2}{#1}}
    \PT@preparewrite  121
    \PT@finalizefile  122 \def\PT@elt#1#2\PT@nil#3#4{\gdef #3{#1}\gdef #4{#2}}
   \PT@queuefilename  123
                       124 \def\PT@queuefilename{\jobname.ptb}
                       125
                       126 % the \empty at the end consumes the newline space
                       127 \def\PT@addfile#1#2{%
                       128    \immediate\write #2{\string\def\string\PTtemp{#1}\string\empty}}
                       129
                       130 \def\PT@splitfile#1#2{%
                       131    \ifeof #1%
                       132      \let #2=\empty
                       133    \else
                       134      \read #1 to#2%
                       135      %\show #2%
                       136      #2% hack, because it essentially ignores #2
                       137      \PT@expanded{\def #2}{\PTtemp}%
                       138      %\show #2%
                       139    \fi}
                       140
                       141 %\def\strip#1{\def #1{\expandafter\@strip #1\@dummy}}
                       142 %\def\@strip#1\@dummy{#1}
                       143
                       144 \def\PT@preparereadmem#1#2{%
```

12

```
145     \global\let #1=#2}
146
147 \def\PT@preparewritefile#1{%
148     \immediate\openout\PT@out\PT@queuefilename\relax
149     \let #1\PT@out}
150
151 \def\PT@preparereadfile#1#2{%
152     \immediate\closeout\PT@out
153     \openin\PT@in\PT@queuefilename\relax
154     \let #1\PT@in}
155
156 \def\PT@finalizefile{%
157     \closein\PT@in}
158
159 \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.

```
160 \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.

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

\endpolytable   This is just defined for convenience.

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

\PT@scantoend   Whenenver an \end is encountered, we check if it really ends the current environ-
ment. We store the tokens we have read. Once we have found the end of the envi-
ronment, 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.

```
168 \newcommand{\PT@scantoend}% LaTeX check
169 \long\def\PT@scantoend #1\end #2%
170   {\PT@toks\expandafter{\the\PT@toks #1}%
171    \def\PT@temp{#2}%
172    \ifx\PT@temp\PT@environment
173      \global\let\PT@columnqueue      \empty
174      \global\let\PT@columnreference \undefined
175      \PT@preparewrite\PT@columnqueue
176      \expandafter\PT@getwidths
177    \else
178      \PT@toks\expandafter{\the\PT@toks\end{#2}}%
179      \expandafter\PT@scantoend
180    \fi}
```

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

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

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

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

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

```
183    \let\savecolumns     \PT@savewidths
184    \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).

```
185    \column{@begin@}{@{}l@{}}%
186    \column{@end@}{}%
187    \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 `\\`.

```
188    \let\fromto          \PT@fromto
189    \let\PT@processentry \PT@checkwidth
190    \let\PT@scanbegin     \PT@scanbeginfree
191    \let\\=               \PT@resetcolumn
192    \let\nextline         \PT@resetcolumn
193    \let\>=               \PT@fromopt
194    \let\==               \PT@from
195    \let\<=               \PT@toopt
196    \global\PT@changedfalse % nothing has changed so far
197    \PT@resetcolumn % we are at the beginning of a line
```

14

Now we are ready for a test run.

```
198    \the\PT@toks
199    \@ifundefined{PT@scanning}%
200      {}{\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.

```
201    \ifx\column\PT@otherrun@column
202    \else
203      % we are in first run, print extra info
204      \PT@prelazylist
205      \PT@typeout@{\PT@environment: \the\PT@cols\space columns, %
206                              \PT@Print\PT@allcols}%
207      \PT@postlazylist
208    \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.

```
209    \let\PT@firstrun@column \PT@otherrun@column
210    \let\savecolumns        \PT@gobbleoptional
211    \let\restorecolumns     \PT@gobbleoptional
212    \let\PT@savewidths      \PT@gobbleoptional
213    \let\PT@restorewidths   \PT@gobbleoptional
```

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

```
214    \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`.

```
215    \ifPT@changed
216      % we need to rerun if something has changed
217      \PT@typeout@{There were changes; another trial run needed.}%
218      \expandafter\PT@getwidths
219    \else
220      % we are done and can do the sorting
221      \PT@typeout@{There were no changes; reached fixpoint.}%
222      \expandafter\PT@sortcols
223    \fi}
```

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

```
224 \def\PT@savecounters
225   {\begingroup
226     \def\@elt ##1%
227       {\global\csname c@##1\endcsname\the\csname c@##1\endcsname}%
228     \xdef\PT@restorecounters{\cl@@ckpt}%
229   \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.

```
230 \def\PT@sortcols
```

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

```
231   {\PT@prelazylist
232    \edef\PT@sortedlist
233      {\Foldr{\noexpand\Cons}{\noexpand\Nil}%
234          {\Insertsort\PT@ltmax\PT@allcols}}%
235    \PT@typeout@{Sorted columns:}%
236    \PT@PrintWidth\PT@sortedlist
237    \PT@postlazylist
```

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

```
238    \PT@cols=0\relax%
239    \PT@prelazylist
240    \PT@Execute{\Map\PT@numbercol\PT@sortedlist}%
241    \PT@postlazylist
242    \edef\PT@lastcol@{\PT@StripColumn\PT@lastcol}%
243    \PT@typeout@{Numbered successfully, %
244                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.

```
245      \ifx\PT@currentwidths\empty
246      \else
247        \PT@typeout@{Saving table information for \PT@currentwidths .}%
248        \PT@expanded\PT@saveinformation\PT@currentwidths
249      \fi
```

Finally, we can typeset the table.

```
250      \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 \\.

```
251 \def\PT@typeset
252   {\PT@typeout@{Typesetting the table ...}%
253    \let\PT@processentry        \PT@placeinbox
254    \let\PT@scanbegin           \PT@scanbeginwidth
255    \let\\=                     \PT@resetandcr
256    \let\nextline               \PT@resetandcr
```

```
257    \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.

```
258    \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.

```
259    \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.

```
260    \PT@begin%
```

Run, and this time, typeset, the contents.

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

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

```
263    \PT@finalize% finalize the queue (possibly close file)
264    \PT@end
265    \endgroup
266    \PT@typeout@{Finished.}%
267    \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 The macro `\PT@from` is bound to `\=` in a polytable environment, and used to move
\PT@fromopt to the named column specified by its argument. The previous column is closed.
\PT@toopt 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.

```
268 \newcommand{\PT@from}[1]%
269   {\PT@checkendentry{#1}\PT@dofrom{#1}}
270
271 \newcommand{\PT@fromopt}[1][]%
272   {\def\PT@temp{#1}%
273    \ifx\PT@temp\empty
274      % set default column name
```

```
275     \def\PT@temp{\PT@currentcolumn .}%
276    \fi
277    \PT@expanded\PT@from\PT@temp}
278
279 \newcommand{\PT@toopt}[1][]%
280   {\def\PT@temp{#1}%
281    \ifx\PT@temp\empty
282      % set default column name
283      \def\PT@temp{\PT@currentcolumn .}%
284    \fi
285    \PT@expanded\PT@checkendentry\PT@temp
286    \let\PT@scanning\undefined}
```

\PT@dofrom

```
287 \newcommand*{\PT@dofrom}[1]%
288   {\edef\PT@currentcolumn{#1}%
289    \let\PT@scanning\PT@currentcolumn
290    \let\PT@currentpreamble\relax% necessary for preparescan
291    \@ifnextchar[%
292      {\PT@expanded\PT@dospecfrom\PT@currentcolumn}%
293      {\PT@expanded\PT@dodofrom  \PT@currentcolumn}}
294
295 \newcommand*{\PT@dospecfrom}{}% LaTeX check
296 \def\PT@dospecfrom #1[#2]%
297   {\PT@checkglobalfrom #2\PT@nil{#1}%
298    \PT@dodofrom{#1}}
299
300 \newcommand*{\PT@checkglobalfrom}{}% LaTeX check
301 \def\PT@checkglobalfrom
302   {\@ifnextchar!\PT@getglobalfrom\PT@ignorefrom}
303
304 \newcommand*{\PT@getglobalfrom}{}% LaTeX check
305 \def\PT@getglobalfrom!#1\PT@nil#2%
306   {\column{#2}{#1}}
307
308 \newcommand*{\PT@ignorefrom}{}% LaTeX check
309 \def\PT@ignorefrom #1\PT@nil#2%
310   {\def\PT@currentpreamble{#1}}
311
312 \newcommand*{\PT@dodofrom}[1]%
313   {\@ifundefined{PT@columnreference}%
314      {% trial run
315       \ifx\column\PT@otherruncolumn
316       \else
317         % first run
318         \let\PT@storeendcolumn\PT@add
319       \fi
320       \def\PT@temp{@end@}}%
321      {% final run
322       \PT@split\PT@columnreference\PT@temp
```

18

```
323      %\PT@typeout@{splitted: \PT@temp}
324      }%
325    \PT@expanded{\PT@expanded\PT@preparescan\PT@currentcolumn}\PT@temp
326    \PT@scanbegin}
327
328 \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.

```
329 \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.

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

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

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

Here, the real work is done:

```
335    \let\PT@currentpreamble\relax% necessary for preparescan
336    \PT@preparescan{#1}{#2}%
337    \PT@scanbegin #3\PT@scanend% defines \@curfield
338    \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.

```
339    \let\PT@infromto\empty
340    \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.

```
341 \newcommand*{\PT@checkendentry}% takes one argument
342    {\@ifundefined{PT@scanning}%
343       {\let\PT@temp\@gobble}%
344       {\let\PT@temp\PT@endentry}%
345    \PT@temp}
346 %\newcommand*{\PT@checkendentry}% takes one argument
347 %   {\@ifundefined{PT@post@preamble}%
348 %      {\let\PT@temp\PT@discardentry}%
349 %      {\let\PT@temp\PT@endentry}%
350 %   \PT@temp}
351
```

```
352 %\newcommand*{\PT@discardentry}[1]%
353 %  {\let\PT@postpreamble=\empty\PT@scanend}
354
355 \newcommand*{\PT@endentry}[1]%
356   {\PT@scanend
357    \edef\PT@temp{#1}%
358    \PT@expanded\PT@storeendcolumn\PT@temp\PT@columnqueue
359    \let\PT@storeendcolumn\@gobbletwo
360    \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.

```
361 \newcommand\PT@firstrun@column[3][0pt]%
362   {\@ifundefined{PT@col@#2.type}%
363     {\PT@typeout@{Defining column \PT@aligncol{#2} at #1.}%
364      \@namedef{PT@col@#2.type}{#3}%
365      \@namedef{PT@col@#2.width}{#1}% initialize the width of the column
366      % add the new column to the (sortable) list of all columns
367      \PT@consmacro\PT@allcols{PT@col@#2}%
368      \advance\PT@cols by 1\relax}%
369     {\expandafter\ifx\csname PT@col@#2.type\endcsname\empty
370        \relax % will be defined in a later table of the same set
371      \else
372        \begingroup
373        \def\PT@temp{PT@col@#2}%
374        \ifx\PT@temp\PT@endcol
375          \relax % end column is always redefined
376        \else
377          \PT@warning{Redefining column #2}%
378        \fi
379        \endgroup
380      \fi
381      \@namedef{PT@col@#2.type}{#3}%
382      \expandafter\ifdim#1>0pt\relax
383        \PT@typeout@{Redefining column #2 at #1.}%
384        \@namedef{PT@col@#2.width}{#1}%
```

```
385        \fi
386      }%
```

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.

```
387      \@ifundefined{PT@col@#2.max}%
388        {\@namedef{PT@col@#2.max}{#1}%
389          \expandafter\let\csname PT@col@#2.trusted\endcsname\PT@true}{}%
390      \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.

```
391 \newcommand\PT@otherrun@column[3][]%
392    {\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.

```
393 \def\PT@checkcoldefined #1%
394    {\@ifundefined{PT@col@#1.type}%
395        {\@ifundefined{PT@defaultcolumnspec}%
396          {\PackageError{polytable}{Undefined column #1}{}}
397          {\PT@debug@{Implicitly defining column #1}%
398           \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.

```
399      \expandafter\ifx\csname PT@col@#1.type\endcsname\empty\relax
400        \@ifundefined{PT@defaultcolumnspec}{}%
401        {\PT@debug@{Implicitly defining column #1}%
402          \PT@expanded{\column{#1}}{\PT@defaultcolumnspec}}%
403      \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.

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

Here, we check the first condition.

```
406      \def\PT@temp{PT@col@#1}%
407      \ifx\PT@currentcol\PT@temp
408        \PT@debug@{No need to skip columns.}%
```

```
409     \else
410       \PT@colwidth=\expandafter\@nameuse\expandafter
411                     {\PT@currentcol.width}\relax
412       \ifdim\PT@colwidth>\csname PT@col@#1.width\endcsname\relax
413         % we need to change the width
414         \PT@debug@{s \PT@aligncol{#1}: %
415                       old=\expandafter\expandafter\expandafter
416                           \PT@aligndim\csname PT@col@#1.width\endcsname\@@%
417                       new=\expandafter\PT@aligndim\the\PT@colwidth\@@}%
418         \PT@changedtrue
419         \PT@enamedef{PT@col@#1.width}{\the\PT@colwidth}%
420       \fi
```

The same for the untrusted `.max` values.

```
421       \PT@colwidth=\expandafter\@nameuse\expandafter
422                     {\PT@currentcol.max}\relax
423       \ifdim\PT@colwidth>\csname PT@col@#1.max\endcsname\relax
424         % we need to change the width
425         \PT@debug@{S \PT@aligncol{#1}: %
426                       old=\expandafter\expandafter\expandafter
427                           \PT@aligndim\csname PT@col@#1.max\endcsname\@@%
428                       new=\expandafter\PT@aligndim\the\PT@colwidth\@@}%
429         \PT@changedtrue
430         \PT@checkrerun
431         \PT@enamedef{PT@col@#1.max}{\the\PT@colwidth}%
432       \fi
433       \ifnum\csname PT@col@#1.trusted\endcsname=\PT@false\relax
434         \ifdim\PT@colwidth=\csname PT@col@#1.max\endcsname\relax
435           \PT@debug@{#1=\the\PT@colwidth\space is now trusted}%
436           \expandafter\let\csname PT@col@#1.trusted\endcsname\PT@true%
437         \fi
438       \fi
439     \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.

```
440     \PT@expanded{\def\PT@temp}{\the\wd\@curfield}%
441     \global\PT@colwidth=\@nameuse{PT@col@#1.width}%
442     \global\advance\PT@colwidth by \PT@temp\relax%
443     \ifdim\PT@colwidth>\csname PT@col@#2.width\endcsname\relax
444       % we need to change the width
445       \PT@debug@{#2 (width \PT@temp) starts after #1 (at \csname PT@col@#1.width\endcsname)}%
446       \PT@debug@{c \PT@aligncol{#2}: %
447                     old=\expandafter\expandafter\expandafter
448                         \PT@aligndim\csname PT@col@#2.width\endcsname\@@%
449                     new=\expandafter\PT@aligndim\the\PT@colwidth\@@}%
450       \PT@changedtrue
451       \PT@enamedef{PT@col@#2.width}{\the\PT@colwidth}%
452     \fi
```

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

```
453    \global\PT@colwidth=\@nameuse{PT@col@#1.max}%
454    \global\advance\PT@colwidth by \PT@temp\relax%
455    \ifdim\PT@colwidth>\csname PT@col@#2.max\endcsname\relax
456      % we need to change the width
457      \PT@debug@{C \PT@aligncol{#2}: %
458                old=\expandafter\expandafter\expandafter
459                   \PT@aligndim\csname PT@col@#2.max\endcsname\@@%
460                new=\expandafter\PT@aligndim\the\PT@colwidth\@@}%
461      \PT@changedtrue
462      \PT@checkrerun
463      \PT@enamedef{PT@col@#2.max}{\the\PT@colwidth}%
464    \fi
465    \ifnum\csname PT@col@#2.trusted\endcsname=\PT@false\relax
466      \ifdim\PT@colwidth=\csname PT@col@#2.max\endcsname\relax
467        \PT@debug@{#2=\the\PT@colwidth\space is now trusted}%
468        \expandafter\let\csname PT@col@#2.trusted\endcsname\PT@true%
469      \fi
470    \fi
```

Finally, we update the current column to #2.

```
471    \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.

```
472 \def\PT@checkrerun
473   {\ifnum\PT@inrestore=\PT@true\relax
474      \PT@rerun
475   \fi}
```

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

```
476 \newcommand*{\PT@resetcolumn}[1][]%
477   {\PT@checkendentry{@end@}%
478    \let\PT@currentcolumn\empty%
479    \let\PT@scanning\undefined
480    \let\PT@currentcol\PT@nullcol
481    % TODO: remove these lines if they don't work
482    %\let\PT@pre@preamble\empty
483    %\PT@scanbeginfree
484   }
```

\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.

```
485 \def\PT@nullcol{PT@col@@begin@}
486 \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.

```
487 \def\PT@Execute{\Foldr\PT@Sequence\empty}
488 \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.

```
489 \def\PT@ShowColumn #1#2%
490    {\PT@ShowColumn@{#1}#2\PT@ShowColumn@}
491 \def\PT@ShowColumn@ #1PT@col@#2\PT@ShowColumn@
492    {#1{#2} }
493 \def\PT@ShowColumnWidth #1%
494    {\PT@typeout@{%
495       \PT@ShowColumn\PT@aligncol{#1}:
496       \expandafter\expandafter\expandafter
497          \PT@aligndim\csname #1.max\endcsname\@@}}
498 \def\PT@StripColumn #1%
499    {\expandafter\PT@StripColumn@#1\PT@StripColumn@}
500 \def\PT@StripColumn@ PT@col@#1\PT@StripColumn@
501    {#1}
```

\PT@Print

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

```
502 \def\PT@Print#1{\PT@Execute{\Map{\PT@ShowColumn\Identity}#1}}
503 \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.

```
504 \def\PT@TeXif #1%
505    {\expandafter\@gobble#1\relax
506       \PT@gobblefalse
507    \else\relax
508       \gobbletrue
509    \fi}
510 \def\PT@gobblefalse\else\relax\gobbletrue\fi #1#2%
511    {\fi #1}
```

\PT@ltmax

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

```
512 \def\PT@ltmax #1#2%
513    {\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.

```
514 \def\PT@numbercol #1%
515  {%\PT@typeout@{numbering #1 as \the\PT@cols}%
516   \PT@enamedef{#1.num}{\the\PT@cols}%
517   \def\PT@lastcol{#1}%
518   \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.

```
519 \newcommand{\PT@resetandcr}%
520  {\PT@expanded\PT@checkendentry\PT@lastcol@%
521   \ifx\PT@currentcol\PT@lastcol
522   \else
523     \ifx\PT@currentcol\PT@nullcol
524       \edef\PT@currentcol{\Head{\Tail\PT@sortedlist}}%
525     \fi
526     \edef\PT@currentcol@{\PT@StripColumn\PT@currentcol}%
527     \PT@typeout@{adding implicit fromto at eol from \PT@currentcol@
528                                     \space to \PT@lastcol@}%
529     \PT@expanded{\PT@expanded\fromto\PT@currentcol@}\PT@lastcol@
530   \fi
531   \PT@typeout@{Next line ...}%
532   \let\PT@scanning\undefined% needed for resetcolumn
533   \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.

```
534 \newcommand{\PT@fill}%
535  {\PT@expanded\PT@checkendentry\PT@lastcol@%
536   \ifx\PT@currentcol\PT@lastcol
537   \else
538     \ifx\PT@currentcol\PT@nullcol
539     \else
540       \edef\PT@currentcol@{\PT@StripColumn\PT@currentcol}%
541       \PT@typeout@{adding implicit fromto from \PT@currentcol@
```

```
542                                              \space to \PT@lastcol@}%
543        \PT@expanded{\PT@expanded\fromto\PT@currentcol@}\PT@lastcol@
544     \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.

```
545 \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.

```
546    {\PT@colwidth=\@nameuse{PT@col@#1.max}%
547     \advance\PT@colwidth by -\expandafter\csname\PT@currentcol.max\endcsname
548     \leavevmode
549     \edef\PT@temp{\PT@StripColumn\PT@currentcol}%
550     \PT@typeout@{adding space of width %
551                 \expandafter\PT@aligndim\the\PT@colwidth\@@
552                 (\expandafter\PT@aligncol\expandafter{\PT@temp} %
553                 -> \PT@aligncol{#1})}%
554     \hb@xt@\PT@colwidth{%
555       {\@mkpream{@{}l@{}}\@addtopreamble\@empty}%
556       \let\CT@row@color\relax% colortbl compatibility
557       \let\@sharp\empty%
558       %\show\@preamble
559       \@preamble}%
```

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

```
560     \PT@typeout@{adding box \space\space of width %
561                 \expandafter\PT@aligndim\the\wd\@curfield\@@
562                 (\PT@aligncol{#1} -> \PT@aligncol{#2})}%
563     \box\@curfield
```

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

```
564     \def\PT@currentcol{PT@col@#2}%
565     \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.

```
566 \def\PT@preparescan#1#2%
567 % First, we check that both columns are defined. This will
568 % actually define the columns if implicit column definitions are
569 % enabled.
570 %    \begin{macrocode}
```

26

```
571    {\PT@checkcoldefined{#1}%
572     \PT@checkcoldefined{#2}%
573     \PT@colwidth=\@nameuse{PT@col@#2.max}%
574     \advance\PT@colwidth by -\@nameuse{PT@col@#1.max}\relax%
575     \ifmmode
576       \PT@debug@{*math mode*}%
577       \let\d@llarbegin=$%$
578       \let\d@llarend=$%$
579       \let\col@sep\arraycolsep
580     \else
581       \PT@debug@{*text mode*}%
582       \let\d@llarbegin=\begingroup
583       \let\d@llarend=\endgroup
584       \let\col@sep\tabcolsep
585     \fi
586     \ifx\PT@currentpreamble\relax
587       \PT@expanded{\PT@expanded{\def\PT@currentpreamble}}%
588                   {\csname PT@col@#1.type\endcsname}%
589     \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.

```
590    {\PT@expanded\@mkpream\PT@currentpreamble%
591     \@addtopreamble\@empty}%
592    \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.

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

We now define the splitting of the preamble.

```
594 \def\PT@splitpreamble #1\@sharp #2\PT@nil{%
595   \let\@sharp=\relax% needed for the following assignment
596   \def\PT@terp{#2}%
597   \ifx\PT@terp\empty%
598     \PackageError{polytable}{Illegal preamble (no columns)}{}%
599   \fi
600   \PT@splitsplitpreamble{#1}#2\PT@nil}
601
602 \def\PT@splitsplitpreamble #1#2\@sharp #3\PT@nil{%
603   \def\PT@temp{#3}%
604   \ifx\PT@temp\empty%
605   \else
606     \PackageError{polytable}{Illegal preamble (multiple columns)}{}%
```

27

```
607    \fi
608    \def\PT@pre@preamble{#1}%
609    \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.

```
610 \def\PT@scanbeginwidth
611    {\PT@scanbegin@{\hbox to \PT@colwidth}}
612
613 \def\PT@scanbeginfree
614    {\PT@scanbegin@{\hbox}}
615
616 \def\PT@scanbegin@#1%
617    {\setbox\@curfield #1%
618      \bgroup
619      \PT@pre@preamble\strut\ignorespaces}
620
621 \def\PT@scanend
622    {\PT@post@preamble
623      \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.

```
624 \newcommand*{\PT@setmaxwidth}[3][\PT@false]% #2 column name, #3 maximum width
625   {\@namedef{PT@col@#2.max}{#3}%
626     \ifdim#3=0pt\relax
627       \expandafter\let\csname PT@col@#2.trusted\endcsname=\PT@true%
628     \else
629       \expandafter\let\csname PT@col@#2.trusted\endcsname=#1%
630     \fi
631     \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.

```
632 \def\PT@loadtable#1% #1 table id number
633   {%\expandafter\show\csname PT@restore@\romannumeral #1\endcsname
634     %\show\column
635     \PT@typeout@
636       {Calling \expandafter\string
637               \csname PT@restore@\romannumeral #1\endcsname.}%
638     \let\maxcolumn\PT@setmaxwidth
639     %\expandafter\show\csname PT@load@\romannumeral #1\endcsname
640     \csname PT@restore@\romannumeral #1\endcsname}
```

\PT@loadtablebyname Often, we want to access table information by a column width set name.

```
641 \def\PT@loadtablebyname#1% #1 set name
642   {\PT@typeout@{Loading table information for column width set #1.}%
643     \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.

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

29

```
650     \expandafter\PT@Execute\expandafter{\expandafter
651       \Map\expandafter{\expandafter\PT@savecolumn
652         \expandafter{\PT@temp}}\PT@sortedlist}}
```

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

```
653 \def\PT@savecolumn#1#2% #1 macro name, #2 column name
654   {\PT@typeout@{saving column #2 in \string #1 ...}%
655    \def\PT@temp{#2}%
656    \ifx\PT@temp\PT@nullcol
657      \PT@typeout@{skipping nullcol ...}%
658 % This was a bug: end column cannot be skipped, because
659 % it can change.
660 %    \else\ifx\PT@temp\PT@endcol
661 %      \PT@typeout@{skipping endcol ...}%
662    \else
663      \PT@typeout@{max=\csname #2.max\endcsname, %
664                   width=\csname #2.width\endcsname, %
665                   trusted=\csname #2.trusted\endcsname}%
666      % we need the column command in here
667      % we could do the same in \column, but then the location of
668      % \save / \restore matters ...
669      \PT@gaddendmacro{#1}{\maxcolumn}%
670      \ifnum\csname #2.trusted\endcsname=\PT@true\relax
671        \PT@gaddendmacro{#1}{[\PT@true]}%
672      \fi
673      \edef\PT@temp{\PT@StripColumn{#2}}%
674      \PT@addargtomacro{#1}{PT@temp}%
675      \PT@addargtomacro{#1}{#2.max}%
676      \PT@gaddendmacro{#1}{\column}%
677      \PT@addoptargtomacro{#1}{#2.width}%
678      \edef\PT@temp{\PT@StripColumn{#2}}%
679      \PT@addargtomacro{#1}{PT@temp}%
680      \PT@addargtomacro{#1}{#2.type}%
681      %\show#1%
682 %  \fi
683    \fi
684   }
```

\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.

```
685 \newcommand*{\PT@savewidths}[1][default@]
686   {\PT@typeout@{Executing \string\savecolumns [#1].}%
687    \def\PT@currentwidths{#1}%
688    \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.

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

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

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

\PT@comparewidths

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

\PT@trustedmax

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

\PT@equalwidths

```
711 \def\PT@equalwidths#1% #1 full column name
712    {\@ifundefined{#1.max}{}%
713      {\ifdim\csname #1.max\endcsname=\csname #1.width\endcsname\relax
714        \PT@typeout@{col #1 is okay ...}%
715      \else
716        \PT@rerun% a rerun is needed
717      \fi}}
```

\PT@verifywidths

```
718 \def\PT@verifywidths#1% #1 column width set name
719    {\@ifundefined{PT@widths@#1}%
720      {\PT@typeout@{Nothing to verify yet for set #1.}%
721       \PT@typeout@{Scheduling set #1 for verification at end of document.}%
722       \AtEndDocument{\PT@verifywidths{#1}}}%
723      {\PT@typeout@{Verifying column width set #1.}%
724       \PT@expanded\PT@verify@widths{\csname PT@widths@#1\endcsname}{#1}}}
725
726 \def\PT@verify@widths#1#2% #1 set id number, #2 set name
```

```
727    {\@ifundefined{PT@restore@\romannumeral #1}{}%
728        {\begingroup
729           \let\column\PT@firstrun@column
730           \PT@cols=0\relax%
731           \def\PT@allcols{\Nil}%
732           \PT@loadtablebyname{#2}%
733           \PT@table=#1\relax
734           % nullcolumn is not loaded, therefore:
735           \@namedef{\PT@nullcol .width}{0pt}%
736           % checking trust
737           \PT@prelazylist
738           \All{\PT@trustedmax}{\PT@allcols}%
739             {\PT@typeout@{All maximum widths can be trusted -- writing .max!}%
740              \PT@save@table{.max}}%
741             {\PT@typeout@{Untrustworthy maximums widths -- writing .width!}%
742              \PT@rerun
743              \PT@save@table{.width}}%
744           \PT@postlazylist
745         \endgroup}%
746     \PT@typeout@{Verification for #2 successful.}}
```

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

```
747 \def\PT@save@table#1%
748    {\PT@typeout@{Saving column width information.}%
749     \if@filesw
750       \PT@prelazylist
751       {\immediate\write\@auxout{%
752          \gdef\expandafter\noexpand
753            \csname PT@restore@\romannumeral\PT@table\endcsname
754              {\PT@Execute{\Map{\PT@write@column{#1}}\PT@allcols}}}}%
755       \PT@postlazylist
756     \fi}
```

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

```
757 \def\PT@write@column #1#2%
758    {\noexpand\maxcolumn^^J%
759       {\PT@StripColumn{#2}}%
760       {\@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.

```
761 \def\pboxed{%
762    \let\PT@begin       \empty
763    \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.

```
764     \ifx\\\PT@arraycr
765       \let\PT@cr          \PT@normalcr
766     \else
767       \let\PT@cr          \\%
768     \fi
769     \expandafter\beginpolytable\ignorespaces}
770
771 \let\endpboxed\endpolytable
772
773 \def\ptboxed{%
774     \def\PT@begin       {\tabular{@{}l@{}}}%
775     \let\PT@end         \endtabular
776     \let\PT@cr          \@arraycr
777     \expandafter\beginpolytable\ignorespaces}
778
779 \let\endptboxed\endpolytable
780
781 \def\pmboxed{%
782     \def\PT@begin       {\array{@{}l@{}}}%
783     \let\PT@end         \endarray
784     \let\PT@cr          \@arraycr
785     \expandafter\beginpolytable\ignorespaces}
786
787 \let\endpmboxed\endpolytable
788
789 \let\ptabular      \ptboxed
790 \let\endptabular   \endptboxed
791 \let\parray        \pmboxed
792 \let\endparray     \endpmboxed
793
```

That is all.

```
794 ⟨/package⟩
```