

beamer named overlay specification with beanoves

Jérôme Laurens

v1.0 2022/10/28

Abstract

This package allows the management of multiple slide lists in **beamer** documents. Slide lists are very handy both during edition and to manage complex and variable beamer overlay specifications.

Contents

1 Minimal example

The document below is a contrived example to show how the **beamer** overlay specifications have been extended.

```
1 \documentclass {beamer}
2 \RequirePackage {beanoves}
3 \begin{document}
4 \Beanoves {
5     A = 1:2,
6     B = A.next:3,
7     C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \insertframenumber}
11 {\Large Slide \insertslidenumbers}
12 \visible<?(A.1)> {Only on slide 1}\\
13 \visible<?(B.1)-?(B.last)> {Only on slide 3 to 5}\\
14 \visible<?(C.1)> {Only on slide 6}\\
15 \visible<?(A.2)> {Only on slide 2}\\
16 \visible<?(B.2::B.last)> {Only on slide 4 to 5}\\
17 \visible<?(C.2)> {Only on slide 7}\\
18 \visible<?(A.3)-> {From slide 3}\\
19 \visible<?(B.3::B.last)> {Only on slide 5}\\
20 \visible<?(C.3)> {Only on slide 8}\\
21 \end{frame}
22 \end{document}
```

On line 4, we use the `\Beanoves` command to declare named slide ranges. On line 5, we declare a slide range named ‘A’, starting at slide 1 and with length 2. On line 12,

the extended *named overlay specification* $\langle A.1 \rangle$ stands for 1, on line 15, $\langle A.2 \rangle$ stands for 2 whereas on line 18, $\langle A.3 \rangle$ stands for 3. On line 6, we declare a second slide range named ‘B’, starting after the 2 slides of ‘A’ namely 3. Its length is 3 meaning that its last slide number is 5, thus each $\langle B.last \rangle$ is replaced by 5. The next slide number after slide range ‘B’ is 6 which is also the start of the third slide range due to line 7.

2 Named slide lists

2.1 Presentation

Within a `beamer` frame, there are different slides that appear in turn. The main slide list is a range of integers covering all the slide numbers, from one to the total amount of slides. In general, a slide list is a range of positive integers identified by a unique name. The main practical interest is that such lists may be defined relative to one another, we can even have lists of slide ranges. Finally, we can use these lists to organize `beamer` overlay specifications logically.

2.2 Defining named slide lists

In order to define named slide lists, we can either use the `\Beanoves` command below before a `beamer` frame environment, or use the `beanoves` option of this environment. The value of the `beanoves` option is similar to the argument of the `\Beanoves` commands, but the latter takes precedence on the former. This behaviour may be useful to input the very same source code into different frames and have different combinations of slides.

```
beanoves = {
  \langle name_1 \rangle = \langle spec_1 \rangle,
  \langle name_2 \rangle = \langle spec_2 \rangle,
  \dots,
  \langle name_n \rangle = \langle spec_n \rangle,
}
```

```
\Beanoves{
  \langle name_1 \rangle = \langle spec_1 \rangle,
  \langle name_2 \rangle = \langle spec_2 \rangle,
  \dots,
  \langle name_n \rangle = \langle spec_n \rangle,
}
```

The keys $\langle name_i \rangle$ are the slide lists names, they are case sensitive and must contain no spaces nor ‘/’ character. In order to avoid name conflicts with floating point functions, it is suggested to let them contain at least an uppercase letter or an underscore. When the same key is used multiple times, only the last one is taken into account. Possible values for $\langle spec_i \rangle$ are the *slide range specifiers* $\langle first \rangle$, $\langle first \rangle : \langle length \rangle$, $\langle first \rangle :: \langle last \rangle$, $: \langle length \rangle :: \langle last \rangle$ where $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$ are algebraic expression possibly involving any integer valued named overlay specifications defined below.

Also possible values are *slide list specifiers* which are comma separated list of *slide range specifiers* and *slide list specifier* between square brackets. The definition

$\langle name \rangle = [\langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_n \rangle]$,

is a convenient shortcut for

$$\begin{aligned}\langle name \rangle.1 &= \langle spec_1 \rangle, \\ \langle name \rangle.2 &= \langle spec_2 \rangle, \\ &\dots, \\ \langle name \rangle.n &= \langle spec_n \rangle.\end{aligned}$$

The rules above can apply individually to each

$$\langle name \rangle.i = \langle spec_i \rangle.$$

Moreover we can go deeper: the definition

$$\langle name \rangle = [[\langle spec_{1.1} \rangle, \langle spec_{1.2} \rangle], [[\langle spec_{2.1} \rangle, \langle spec_{2.2} \rangle]]]$$

happens to be a convenient shortcut for

$$\begin{aligned}\langle name \rangle.1.1 &= \langle spec_{1.1} \rangle, \\ \langle name \rangle.1.2 &= \langle spec_{1.2} \rangle, \\ \langle name \rangle.2.1 &= \langle spec_{2.1} \rangle, \\ \langle name \rangle.2.2 &= \langle spec_{2.2} \rangle\end{aligned}$$

and so on.

3 Named overlay specifications

3.1 Named slide ranges

When *slide range specifications* are used, the named overlay specifications are detailed in the tables below together with their replacement meaning value as `beamer` standard overlay specification.

$\langle name \rangle == [i, i + 1, i + 2, \dots]$	
syntax	meaning
$\langle name \rangle.1$	i
$\langle name \rangle.2$	$i + 1$
$\langle name \rangle.\langle integer \rangle$	$i + \langle integer \rangle - 1$

In the frame example below, we use the `\BeanovesEval` command for the demonstration. It is mainly used for debugging and testing purposes.

```

1 \Beanoves {
2   A = 3:6,
3 }
4 \begin{frame} {Frame \insertframenum} {Slide \insertslidenumber}
5 \ttfamily
6 \BeanovesEval(A.1) ==3,
7 \BeanovesEval(A.2) ==4,
8 \BeanovesEval(A.-1)==1,
9 \end{frame}
```

When the slide range has been given a length or an end, like in the frame example below, we also have

$\langle name \rangle == [i, i + 1, \dots, j]$			
syntax	meaning	example	output
$\langle name \rangle.length$	$j - i + 1$	A.length	6
$\langle name \rangle.last$	j	A.last	8
$\langle name \rangle.next$	$j + 1$	A.next	9
$\langle name \rangle.range$	$i \text{ '}' - \text{'}' j$	A.range	3-8

```

1 \Beanoves {
2   A = 3:6, % or equivalently A = 3::8 or A = :6::8,
3
4 }
5 \begin{frame} {Frame \insertframenum} {Slide \insertslidenumber}
6 \ttfamily
7 \BeanovesEval(A.1) == 3,
8 \BeanovesEval(A.length) == 6,
9 \BeanovesEval(A.last) == 8,
10 \BeanovesEval(A.next) == 9,
11 \BeanovesEval(A.range) == 3-8,
12 \end{frame}

```

Using these specifications on unfinite named slide ranges is unsupported. Finally each named slide range has a dedicated counter $\langle name \rangle.n$ which is some kind of variable that can be used and incremented¹.

$\langle name \rangle.n$: use the position of the counter

$\langle name \rangle.n += \langle integer \rangle$: advance the counter by $\langle integer \rangle$ and use the new position

$++\langle name \rangle.n$: advance the counter by 1 and use the new position

Notice that “.n” can generally be omitted.

3.2 Named slide lists

After the definition

$\langle name \rangle = [\langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_n \rangle]$

the rules of the previous section apply recursively to each individual declaration

$\langle name \rangle.i = \langle spec_i \rangle$.

4 ?(...) query expressions

This is the key feature of the `beanoves` package, extending `beamer overlay specifications` included between pointed brackets. Before the `overlay specifications` are processed by the `beamer` class, the `beanoves` package scans them for any occurrence of ‘ $\langle ?(\langle queries \rangle) \rangle$ ’. Each one is then evaluated and replaced by its static counterpart. The overall result is finally forwarded to the `beamer` class.

The $\langle queries \rangle$ argument is a comma separated list of individual $\langle query \rangle$ ’s of next table. Sometimes, using $\langle name \rangle.range$ is not allowed as it would lead to an algebraic difference instead of a range.

query	static value	limitation
:	–	
::	–	
$\langle first\ expr \rangle$	$\langle first \rangle$	
$\langle first\ expr \rangle :$	$\langle first \rangle -$	no $\langle name \rangle.range$
$\langle first\ expr \rangle ::$	$\langle first \rangle -$	no $\langle name \rangle.range$
$\langle first\ expr \rangle : \langle length\ expr \rangle$	$\langle first \rangle - \langle last \rangle$	no $\langle name \rangle.range$
$\langle first\ expr \rangle :: \langle end\ expr \rangle$	$\langle first \rangle - \langle last \rangle$	no $\langle name \rangle.range$

¹This is actually an experimental feature.

Here $\langle first\ expr \rangle$, $\langle length\ expr \rangle$ and $\langle end\ expr \rangle$ both denote algebraic expressions possibly involving named overlay specifications and counters. As integers, they respectively evaluate to $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$.

For example both $?(\mathbf{A.next})$, $?(\mathbf{A.last+1})$, $?(\mathbf{A.1+A.length})$ give the same result as soon as the slide range named ‘A’ has been properly defined with a starting value and a length.

Notice that nesting $?(\dots)$ expressions is not supported.

5 Implementation

Identify the internal prefix (L^AT_EX3 DocStrip convention).

```
1 <@@=bnvs>
```

Reserved namespace: identifiers containing the case insensitive string `beanoves` or the string `bnvs` delimited by two non characters.

5.1 Package declarations

```
2 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
3 \ProvidesExplPackage
4   {beanoves}
5   {2022/10/28}
6   {1.0}
7   {Named overlay specifications for beamer}
```

5.2 logging

Utility message.

```
8 \msg_new:nnn { beanoves } { :n } { #1 }
9 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
```

5.3 Debugging and testing facilities

Typesetting file `beanoves.dtx` creates both `beanoves` and `beanoves-debug` style files. The former is intended for everyday use whereas the latter contains supplemental debugging and testing facilities which are intentionally left undocumented.

5.4 Local variables

We make heavy use of local variables and function scopes. Many functions are executed within a T_EX group, which ensures no name collision with the caller stack. In that case, variables need not follow exactly the L^AT_EX3 naming convention: we do not specialize with the module name. On execution, next initialization instructions declare the variables as side effect.

```
10 \tl_new:N \l__bnvs_id_current_tl
11 \tl_set:Nn \l__bnvs_id_current_tl { ?! }
12 \tl_new:N \l__bnvs_a_tl
13 \tl_new:N \l__bnvs_b_tl
14 \tl_new:N \l__bnvs_c_tl
15 \tl_new:N \l__bnvs_id_tl
16 \tl_new:N \l__bnvs_ans_tl
```

```

17 \tl_new:N \l__bnvs_name_tl
18 \tl_new:N \l__bnvs_path_tl
19 \tl_new:N \l__bnvs_group_tl
20 \tl_new:N \l__bnvs_query_tl
21 \tl_new:N \l__bnvs_token_tl
22 \int_new:N \g__bnvs_call_int
23 \int_new:N \l__bnvs_depth_int
24 \seq_new:N \l__bnvs_a_seq
25 \seq_new:N \l__bnvs_b_seq
26 \seq_new:N \l__bnvs_ans_seq
27 \seq_new:N \l__bnvs_match_seq
28 \seq_new:N \l__bnvs_split_seq
29 \seq_new:N \l__bnvs_path_seq
30 \seq_new:N \l__bnvs_query_seq
31 \seq_new:N \l__bnvs_token_seq
32 \bool_new:N \l__bnvs_no_counter_bool
33 \bool_new:N \l__bnvs_no_range_bool
34 \bool_new:N \l__bnvs_in_frame_bool
35 \bool_set_false:N \l__bnvs_in_frame_bool

```

5.5 Infinite loop management

Unending recursivity is managed here.

`\g__bnvs_call_int` Some functions calls, as well as some loop bodies, decrement this counter. When this counter reaches 0, an error is raised or a computation is aborted.

(End definition for `\g__bnvs_call_int`.)

```

36 \int_const:Nn \c__bnvs_max_call_int { 2048 }

```

`__bnvs_call_greset:`

`__bnvs_call_greset:`

Reset globally the call stack counter to its maximum value.

```

37 \cs_set:Npn \__bnvs_call_greset: {
38   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
39 }

```

`__bnvs_call:TF`

`__bnvs_call_do:TF` {*< true code >*} {*< false code >*}

Decrement the `\g__bnvs_call_int` counter globally and execute *< true code >* if we have not reached 0, *< false code >* otherwise.

```

40 \prg_new_conditional:Npnn \__bnvs_call: { T, F, TF } {
41   \int_gdecr:N \g__bnvs_call_int
42   \int_compare:nNnTF \g__bnvs_call_int > 0 {
43     \prg_return_true:
44   } {
45     \prg_return_false:
46   }
47 }

```

5.6 Overlay specification

5.6.1 In slide range definitions

`\g__bnvs_prop` $\langle key \rangle - \langle value \rangle$ property list to store the named slide lists. The basic keys are, assuming $\langle id \rangle! \langle name \rangle$ is a fully qualified slide list name,

$\langle id \rangle! \langle name \rangle / A$ for the first index

$\langle id \rangle! \langle name \rangle / L$ for the length when provided

$\langle id \rangle! \langle name \rangle / Z$ for the last index when provided

$\langle id \rangle! \langle name \rangle / C$ for the counter value, when used

$\langle id \rangle! \langle name \rangle / C0$ for initial value of the counter (when reset)

Other keys are eventually used to cache results when some attributes are defined from other slide ranges. They are characterized by a ‘//’.

$\langle id \rangle! \langle name \rangle // A$ for the cached static value of the first index

$\langle id \rangle! \langle name \rangle // Z$ for the cached static value of the last index

$\langle id \rangle! \langle name \rangle // L$ for the cached static value of the length

$\langle id \rangle! \langle name \rangle // N$ for the cached static value of the next index

The implementation is private, in particular, keys may change in future versions.

⁴⁸ `\prop_new:N \g__bnvs_prop`

(End definition for `\g__bnvs_prop`.)

```

\__bnvs_gput:nn
\__bnvs_gput:nV
\__bnvs_gprovide:nn
\__bnvs_gprovide:nV
\__bnvs_item:n
\__bnvs_get:nN
\__bnvs_gremove:n
\__bnvs_gclear:n
\__bnvs_gclear_cache:n
\__bnvs_gclear:

```

```

\__bnvs_gput:nn {<key>} {<value>}
\__bnvs_gprovide:nn {<key>} {<value>}
\__bnvs_item:n {<key>}
\__bnvs_get:n {<key>} {<tl variable>}
\__bnvs_gremove:n {<key>}
\__bnvs_gclear:n {<key>}
\__bnvs_gclear_cache:n {<key>}
\__bnvs_gclear:

```

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over L^AT_EX3 eponym functions, except `__bnvs_gprovide:nn` which meaning is straightforward.

```

49 \cs_new:Npn \__bnvs_gput:nn #1 #2 {
50   \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
51 }
52 \cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
53   \prop_if_in:NnF \g__bnvs_prop { #1 } {
54     \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
55   }
56 }
57 \cs_new:Npn \__bnvs_item:n {
58   \prop_item:Nn \g__bnvs_prop
59 }
60 \cs_new:Npn \__bnvs_get:nN {
61   \prop_get:NnN \g__bnvs_prop
62 }
63 \cs_new:Npn \__bnvs_gremove:n {
64   \prop_gremove:Nn \g__bnvs_prop
65 }
66 \cs_new:Npn \__bnvs_gclear:n #1 {
67   \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
68     \__bnvs_gremove:n { #1 / ##1 }
69   }
70 }
71 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
72   \clist_map_inline:nn { /A, /L, /Z, /N } {
73     \__bnvs_gremove:n { #1 / ##1 }
74   }
75 }
76 \cs_new:Npn \__bnvs_gclear: {
77   \prop_gclear:N \g__bnvs_prop
78 }
79 \cs_generate_variant:Nn \__bnvs_gput:nn { nV }
80 \cs_generate_variant:Nn \__bnvs_gprovide:nn { nV }

```

```

\__bnvs_if_in_p:n ★ \__bnvs_if_in_p:n {<key>}
\__bnvs_if_in_p:V ★ \__bnvs_if_in:nTF {<key>} {<true code>} {<false code>}
\__bnvs_if_in:nTF ★
\__bnvs_if_in:VTF ★

```

Convenient shortcuts to test for the existence of some key, it makes the code more concise and readable.

```

81 \prg_new_conditional:Npnn \__bnvs_if_in:n #1 { p, T, F, TF } {
82   \prop_if_in:NnTF \g__bnvs_prop { #1 } {
83     \prg_return_true:

```



```

84   } {
85     \prg_return_false:
86   }
87 }
88 \prg_generate_conditional_variant:Nnn \__bnvs_if_in:n {V} { p, T, F, TF }

```

<u>__bnvs_get:nNTF</u>	<u>__bnvs_get:nNTF</u> {<key>} <tl variable> {<true code>} {<false code>}
<u>__bnvs_get:nnNTF</u>	<u>__bnvs_get:nnNTF</u> {<id>} {<key>} <tl variable> {<true code>} {<false code>}

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute <true code> when the item is found, <false code> otherwise. In the latter case, the content of the <tl variable> is undefined. NB: the predicate won't work because \prop_get:NnNTF is not expandable.

```

89 \prg_new_conditional:Npnn \__bnvs_get:nN #1 #2 { T, F, TF } {
90   \prop_get:NnNTF \g__bnvs_prop { #1 } #2 {
91     \prg_return_true:
92   } {
93     \prg_return_false:
94   }
95 }

```

5.6.2 Regular expressions

`\c__bnvs_name_regex` The name of a slide range consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```

96 \regex_const:Nn \c__bnvs_name_regex {
97   [[[:alpha:]]_][[:alnum:]]_*
98 }

```

(End definition for `\c__bnvs_name_regex`.)

`\c__bnvs_id_regex` The name of a slide range consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```

99 \regex_const:Nn \c__bnvs_id_regex {
100   (?: \ur{c__bnvs_name_regex} | [?]* ) ? !
101 }

```

(End definition for `\c__bnvs_id_regex`.)

`\c__bnvs_path_regex` A sequence of .<positive integer> items representing a path.

```

102 \regex_const:Nn \c__bnvs_path_regex {
103   (?: \. [+-]? \d+ ) *
104 }

```

(End definition for `\c__bnvs_path_regex`.)

`\c__bnvs_key_regex` A key is the name of a slide range possibly followed by positive integer attributes using a dot syntax. The 'A_key_Z' variant matches the whole string.

```

105 \regex_const:Nn \c__bnvs_key_regex {
106   \ur{c__bnvs_id_regex} ?
107   \ur{c__bnvs_name_regex}
108   \ur{c__bnvs_path_regex}
109 }
110 \regex_const:Nn \c__bnvs_A_key_Z_regex {

```

2: slide $\langle id \rangle$

3: question mark, when $\langle id \rangle$ is empty

4: The range name

```

111   \A ( ( \ur{c__bnvs_id_regex} ? ) \ur{c__bnvs_name_regex} )

```

5: the path, if any.

```

112   ( \ur{c__bnvs_path_regex} ) \Z
113 }
114

```

(End definition for \c__bnvs_key_regex and \c__bnvs_A_key_Z_regex.)

`\c__bnvs_colons_regex` For ranges defined by a colon syntax.

```

115 \regex_const:Nn \c__bnvs_colons_regex { :(:+)? }

```

(End definition for \c__bnvs_colons_regex.)

`\c__bnvs_list_regex` A comma separated list between square brackets.

```

116 \regex_const:Nn \c__bnvs_list_regex {
117   \A \[ \s*

```

Capture groups:

- 2: the content between the brackets, outer spaces trimmed out

```

118   ( [^\] %[---
119   ]*? )
120   \s* \] \Z
121 }

```

(End definition for \c__bnvs_list_regex.)

`\c__bnvs_split_regex` Used to parse slide list overlay specifications in queries. Next are the 10 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

```

122 \regex_const:Nn \c__bnvs_split_regex {
123   \s* ( ? :

```

We start with ‘++’ instructions².

²At the same time an instruction and an expression... this is a synonym of expression

- 1: $\langle name \rangle$ of a slide range
- 2: $\langle id \rangle$ of a slide range plus the exclamation mark

124 `\+\+ ((\ur{c__bnvs_id_regex}?) \ur{c__bnvs_name_regex})`

- 3: optionally followed by an integer path

125 `(\ur{c__bnvs_path_regex}) (?: \. n)?`

We continue with other expressions

- 4: qualified $\langle name \rangle$ of a slide range,
- 5: $\langle id \rangle$ of a slide range plus the exclamation mark (to manage void $\langle id \rangle$)

126 `| ((\ur{c__bnvs_id_regex}?) \ur{c__bnvs_name_regex})`

- 6: optionally followed by an integer path

127 `(\ur{c__bnvs_path_regex})`

Next comes another branching

128 `(?:`

- 7: the $\langle length \rangle$ attribute

129 `\. l(e)ngth`

- 8: the $\langle last \rangle$ attribute

130 `| \. l(a)st`

- 9: the $\langle next \rangle$ attribute

131 `| \. ne(x)t`

- 10: the $\langle range \rangle$ attribute

132 `| \. (r)ange`

- 11: the $\langle n \rangle$ attribute

133 `| (?: \. (n))? (?:`

- 12: the poor man integer expression after ‘+=’, which is the longest sequence of black characters, which ends just before a space or at the very last character. This tricky definition allows quite any algebraic expression, even those involving parenthesis.

134 `\s* \+= \s* (\S+)`

- 13: the post increment

135 `| (\+)\+)?`

136 `)?`

137 `) \s*`

138 `}`

(End definition for `c__bnvs_split_regex`.)

5.6.3 beamer.cls interface

Work in progress.

```

139 \RequirePackage{keyval}
140 \define@key{beamerframe}{beanoves~id}[]{}
141 \tl_set:Nx \l__bnvs_id_current_tl { #1 ! }
142 }
143 \AddToHook{env/beamer@frameslide/before}{
144 \bool_set_true:N \l__bnvs_in_frame_bool
145 }
146 \AddToHook{env/beamer@frameslide/after}{
147 \bool_set_false:N \l__bnvs_in_frame_bool
148 }
149 \AddToHook{cmd/frame/before}{
150 }

```

5.6.4 Defining named slide ranges

<code>__bnvs_parse:Nnn</code>	<code>__bnvs_parse:Nnn <command> {<key>} {<definition>}</code>
--------------------------------	---

Auxiliary function called within a group. *<key>* is the slide range key, including eventually a dotted integer path and a slide identifier, *<definition>* is the corresponding definition. *<command>* is `__bnvs_range:nVVV` at runtime.

<code>\l__bnvs_match_seq</code>	Local storage for the match result.
	(End definition for <code>\l__bnvs_match_seq</code> .)

<code>__bnvs_range:nnnn</code>	<code>__bnvs_range:nnnn {<key>} {<first>} {<length>} {<last>}</code>
<code>__bnvs_range:nVVV</code>	<code>__bnvs_range_alt:nnnn {<key>} {<first>} {<length>} {<last>}</code>
<code>__bnvs_range_alt:nnnn</code>	<code>__bnvs_range:Nnnnn <cmd> {<key>} {<first>} {<length>} {<last>}</code>
<code>__bnvs_range_alt:nVVV</code>	
<code>__bnvs_range:Nnnnn</code>	

Auxiliary function called within a group. Setup the model to define a range. The alt variant does not override an already existing value.

Implementation detail: the core functionality is implemented in the auxiliary function `__bnvs_range:Nnnnn` which first argument is `__bnvs_gput:nn` for `__bnvs_range:nnnn` and `__bnvs_gprovide:nn` for `__bnvs_range_alt:nnnn`.

```

151 \cs_new:Npn \__bnvs_range:Nnnnn #1 #2 #3 #4 #5 {
152   \tl_if_empty:nTF { #3 } {
153     \tl_if_empty:nTF { #4 } {
154       \tl_if_empty:nTF { #5 } {
155         \msg_error:nnn { beanoves } { :n } { Not~a~range::~~#2 }
156       } {
157         #1 { #2/Z } { #5 }
158       }
159     } {
160       #1 { #2/L } { #4 }
161       \tl_if_empty:nF { #5 } {
162         #1 { #2/Z } { #5 }
163         #1 { #2/A } { #2.last - (#2.length) + 1 }
164       }
165     }
166   } {

```

```

167     #1 { #2/A } { #3 }
168     \tl_if_empty:nTF { #4 } {
169         \tl_if_empty:nF { #5 } {
170             #1 { #2/Z } { #5 }
171             #1 { #2/L } { #2.last - (#2.1) + 1 }
172         }
173     } {
174         #1 { #2/L } { #4 }
175         #1 { #2/Z } { #2.1 + #2.length - 1 }
176     }
177 }
178 }
179 \cs_new:Npn \__bnvs_range:nnnn #1 {
180     \__bnvs_gclear:n { #1 }
181     \__bnvs_range:Nnnnn \__bnvs_gput:nn { #1 }
182 }
183 \cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
184 \cs_new:Npn \__bnvs_range_alt:nnnn #1 {
185     \__bnvs_gclear_cache:n { #1 }
186     \__bnvs_range:Nnnnn \__bnvs_gprovide:nn { #1 }
187 }
188 \cs_generate_variant:Nn \__bnvs_range_alt:nnnn { nVVV }

```

__bnvs_parse:Nn __bnvs_parse:Nn <command> {<key>}

Define a hidden range, for which slides are never shown. This is useful to conditionally show or hide a sequence of slides.

```

189 \cs_new:Npn \__bnvs_parse:Nn #1 #2 {
190     \__bnvs_group_begin:
191     \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
192         \exp_args:Nx \__bnvs_gput:nn { \l__bnvs_name_tl/ } { }
193         \exp_args:NNNV
194         \__bnvs_group_end:
195         \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
196     } {
197         \msg_error:nnn { beanoves } { :n } { Unexpected-key:~#2 }
198         \__bnvs_group_end:
199     }
200 }

```

__bnvs_parse_range:nNNNTF __bnvs_parse_range:nNNN {<input>} <first tl> <length tl> <last tl> {<true code>} {<false code>}

Parse <input> as a range according to \c__bnvs_colons_regex.

```

201 \exp_args_generate:n { VVV }
202 \cs_new:Npn \__bnvs_range_set:NNNn #1 #2 #3 #4 {
203     \__bnvs_group_begin:

```

This is not a list.

```

204     \tl_clear:N \l__bnvs_a_tl
205     \tl_clear:N \l__bnvs_b_tl
206     \tl_clear:N \l__bnvs_c_tl
207     \regex_split:NnN \c__bnvs_colons_regex { #4 } \l__bnvs_split_seq
208     \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_a_tl {

```

\l__bnvs_a_tl may contain the $\langle start \rangle$.

```
209 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_b_tl {
210 \tl_if_empty:NTF \l__bnvs_b_tl {
```

This is a one colon range.

```
211 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
\l__bnvs_b_tl may contain the  $\langle length \rangle$ .
```

```
212 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_c_tl {
213 \tl_if_empty:NTF \l__bnvs_c_tl {
```

A :: was expected:

```
214 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(1):~#4 }
215 } {
216 \int_compare:nNtT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
217 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(2):~#4 }
218 }
219 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
\l__bnvs_c_tl may contain the  $\langle end \rangle$ .
```

```
220 \seq_if_empty:NF \l__bnvs_split_seq {
221 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(3):~#4 }
222 }
223 }
224 }
225 } {
```

This is a two colon range.

```
226 \int_compare:nNtT { \tl_count:N \l__bnvs_b_tl } > { 1 } {
227 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(4):~#4 }
228 }
229 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
```

\l__bnvs_c_tl contains the $\langle end \rangle$.

```
230 \seq_pop_left:NNTF \l__bnvs_split_seq \l__bnvs_b_tl {
231 \tl_if_empty:NTF \l__bnvs_b_tl {
232 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
```

\l_b_tl may contain the $\langle length \rangle$.

```
233 \seq_if_empty:NF \l__bnvs_split_seq {
234 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(5):~#4 }
235 }
236 } {
237 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(6):~#4 }
238 }
239 } {
240 \tl_clear:N \l__bnvs_b_tl
241 }
242 }
243 }
244 }
```

Providing both the $\langle start \rangle$, $\langle length \rangle$ and $\langle end \rangle$ of a range is not allowed, even if they happen to be consistent.

```

245 \bool_if:nF {
246   \tl_if_empty_p:N \l__bnvs_a_tl
247   || \tl_if_empty_p:N \l__bnvs_b_tl
248   || \tl_if_empty_p:N \l__bnvs_c_tl
249 } {
250 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(7):~#3 }
251 }
252 \cs_set:Npn \:nnn ##1 ##2 ##3 {
253   \__bnvs_group_end:
254   \tl_set:Nn #1 { ##1 }
255   \tl_set:Nn #2 { ##2 }
256   \tl_set:Nn #3 { ##3 }
257 }
258 \exp_args:NVVV \:nnn \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
259 }

```

__bnvs_do_parse:Nnn __bnvs_do_parse:Nnn <command> {<full name>}

Auxiliary function for __bnvs_parse:Nn. <command> is __bnvs_range:nVVV at run-time and must have signature nVVV.

```

260 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
261 \cs_new:Npn \__bnvs_do_parse:Nnn #1 #2 #3 {

```

This is not a list.

```

262 \__bnvs_range_set:NNNn \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl { #3 }
263   #1 { #2 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
264 }
265 \cs_generate_variant:Nn \__bnvs_do_parse:Nnn { Nxn, Non }

```

__bnvs_id_name_set:nNNTF __bnvs_id_name_set:nNNTF {<key>} <id tl var> <full name tl var> {< true code>} {< false code>}

If the <key> is a key, put the name it defines into the <name tl var> with the current frame id prefix \l__bnvs_id_tl if none was given, then execute <true code>. Otherwise execute <false code>.

```

266 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
267   \__bnvs_group_begin:
268   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
269     #1
270   } \l__bnvs_match_seq {
271     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
272     \tl_if_empty:NTF #2 {
273       \exp_args:NNNx
274       \__bnvs_group_end:
275       \tl_set:Nn #3 { \l__bnvs_id_current_tl #1 }
276       \tl_set_eq:NN #2 \l__bnvs_id_current_tl
277     } {
278       \cs_set:Npn \:n ##1 {
279         \__bnvs_group_end:
280         \tl_set:Nn #2 { ##1 }
281         \tl_set:Nn \l__bnvs_id_current_tl { ##1 }

```

```

282     }
283     \exp_args:NV
284     \:n #2
285     \tl_set:Nn #3 { #1 }
286   }
287   \prg_return_true:
288 } {
289   \__bnvs_group_end:
290   \prg_return_false:
291 }
292 }

293 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
294   \__bnvs_group_begin:
295   \__bnvs_id_name_set:nNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
296     \regex_extract_once:NnNTF \c__bnvs_list_regex {
297       #3
298     } \l__bnvs_match_seq {

```

This is a comma separated list, extract each item and go recursive.

```

299     \exp_args:NNx
300     \seq_set_from_clist:Nn \l__bnvs_match_seq {
301       \seq_item:Nn \l__bnvs_match_seq { 2 }
302     }
303     \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
304       \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl.##1 } { ##2 }
305     }
306   } {
307     \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
308   }
309 } {
310   \msg_error:nnn { beanoves } { :n } { Invalid~key:~#2 }
311 }

```

We export \l__bnvs_id_tl:

```

312   \exp_args:NNNV
313   \__bnvs_group_end:
314   \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
315 }

```

`\Beanoves` `{⟨key--value list⟩}`

The keys are the slide range specifiers. When no value is provided, it defaults to 1. On the contrary, `⟨key-value⟩` items are parsed by `__bnvs_parse:Nnn`.

```

316 \NewDocumentCommand \Beanoves { sm } {
317   \tl_if_eq:NnT \@currenvir { document } {
318     \__bnvs_gclear:
319   }
320   \IfBooleanTF {#1} {
321     \keyval_parse:nnn {
322       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
323     } {
324       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
325     }
326   } {
327     \keyval_parse:nnn {
328       \__bnvs_parse:Nn \__bnvs_range:nVVV
329     } {
330       \__bnvs_parse:Nnn \__bnvs_range:nVVV
331     }
332   }
333   { #2 }
334   \ignorespaces
335 }

```

If we use the frame `beanoves` option, we can provide default values to the various name ranges.

```

336 \define@key{beamerframe}{beanoves}{\Beanoves*{#1}}

```

5.6.5 Scanning named overlay specifications

Patch some beamer commands to support `?(...)` instructions in overlay specifications.

`\beamer@frame` `\beamer@frame {⟨overlay specification⟩}`
`\beamer@masterdecode` `\beamer@masterdecode {⟨overlay specification⟩}`

Preprocess `⟨overlay specification⟩` before `beamer` reads it.

`\l__bnvs_ans_tl` Storage for the translated overlay specification, where `?(...)` instructions are replaced by their static counterparts.

(End definition for `\l__bnvs_ans_tl`.)

Save the original macro `\beamer@masterdecode` and then override it to properly preprocess the argument.

```

337 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
338 \cs_set:Npn \beamer@frame < #1 > {
339   \__bnvs_group_begin:
340   \tl_clear:N \l__bnvs_ans_tl
341   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
342   \exp_args:NNNV
343   \__bnvs_group_end:
344   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
345 }
346 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode

```

```

347 \cs_set:Npn \beamer@masterdecode #1 {
348   \__bnvs_group_begin:
349   \tl_clear:N \l__bnvs_ans_tl
350   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
351   \exp_args:NNV
352   \__bnvs_group_end:
353   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
354 }

```

__bnvs_scan:nNN __bnvs_scan:nNN {*<named overlay expression>*} *<eval>* *<tl variable>*

Scan the *<named overlay expression>* argument and feed the *<tl variable>* replacing ?(...) instructions by their static counterpart with help from the *<eval>* function, which is __bnvs_eval:nN. A group is created to use local variables:

\l__bnvs_ans_tl The token list that will be appended to *<tl variable>* on return.

(End definition for \l__bnvs_ans_tl.)

\l__bnvs_depth_int Store the depth level in parenthesis grouping used when finding the proper closing parenthesis balancing the opening parenthesis that follows immediately a question mark in a ?(...) instruction.

(End definition for \l__bnvs_depth_int.)

\l__bnvs_query_tl Storage for the overlay query expression to be evaluated.

(End definition for \l__bnvs_query_tl.)

\l__bnvs_token_seq The *<overlay expression>* is split into the sequence of its tokens.

(End definition for \l__bnvs_token_seq.)

\l__bnvs_token_tl Storage for just one token.

(End definition for \l__bnvs_token_tl.)

```

355 \cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 {
356   \__bnvs_group_begin:
357   \tl_clear:N \l__bnvs_ans_tl
358   \seq_clear:N \l__bnvs_token_seq

```

Explode the *<named overlay expression>* into a list of tokens:

```

359   \regex_split:nnN {} { #1 } \l__bnvs_token_seq

```

\scan_question: \scan_question:

At top level state, scan the tokens of the *<named overlay expression>* looking for a ‘?’ character.

```

360   \cs_set:Npn \scan_question: {
361     \seq_pop_left:NNT \l__bnvs_token_seq \l__bnvs_token_tl {
362       \tl_if_eq:NnTF \l__bnvs_token_tl { ? } {
363         \require_open:
364       } {
365         \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl

```

```

366         \scan_question:
367     }
368 }
369 }

```

\require_open: \require_open:

We just found a '?', we first gobble tokens until the next '(', whatever they may be. In general, no tokens should be silently ignored.

```

370     \cs_set:Npn \require_open: {
    Get next token.
371     \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
372         \tl_if_eq:NnTF \l__bnvs_token_tl { ( %}
373     } {

```

We found the '(' after the '?'. Set the parenthesis depth to 1 (on first passage).

```

374         \int_set:Nn \l__bnvs_depth_int { 1 }

```

Record the forthcoming content in the \l__bnvs_query_tl variable, up to the next balancing ')':

```

375         \tl_clear:N \l__bnvs_query_tl
376         \require_close:
377     } {

```

Ignore this token and loop.

```

378         \require_open:
379     }
380 } {

```

End reached but no opening parenthesis found, raise.

```

381     \msg_fatal:nxx { beanoves } { :n } {Missing~'('%---)
382         ~after~a~?:~#1}
383 }
384 }

```

\require_close: \require_close:

We found a '(', we record the forthcoming content in the \l__bnvs_query_tl variable, up to the next balancing ')':

```

385     \cs_set:Npn \require_close: {
    Get next token.
386     \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
387         \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
388     } {

```

We found a '(', increment the depth and append the token to \l__bnvs_query_tl, then scan again for a).

```

389         \int_incr:N \l__bnvs_depth_int
390         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
391         \require_close:
392     } {

```

This is not a '('.

```

393         \tl_if_eq:NnTF \l__bnvs_token_tl { %(---
394         )
395     } {

```

We found a ')', we decrement and test the depth.

```

396         \int_decr:N \l__bnvs_depth_int
397         \int_compare:nNnTF \l__bnvs_depth_int = 0 {

```

The depth level has reached 0: we found our balancing parenthesis of the ?(...) instruction. We can append the evaluated slide ranges token list to \l__ans_tl and look for the next ?.

```

398         \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
399         \scan_question:
400     } {

```

The depth has not yet reached level 0. We append the ')' to \l__bnvs_query_tl because it is not yet the end of sequence marker.

```

401         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
402         \require_close:
403     }
404     } {

```

The scanned token is not a '(' nor a ')', we append it as is to \l__bnvs_query_tl and look for a).

```

405         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
406         \require_close:
407     }
408     }
409     } {

```

Above ends the code for Not a '(' We reached the end of the sequence and the token list with no closing ')'. We raise and terminate. As recovery we feed \l__bnvs_query_tl with the missing ')'.

```

410         \msg_error:nnx { beanoves } { :n } {Missing~%(---
411         `)'':~#1 }
412         \tl_put_right:Nx \l__bnvs_query_tl {
413             \prg_replicate:nn { \l__bnvs_depth_int } {%(---
414             )}
415         }
416         \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
417     }
418 }

```

Run the top level loop to scan for a '?':

```

419     \scan_question:
420     \exp_args:NNNV
421     \__bnvs_group_end:
422     \tl_put_right:Nn #3 \l__bnvs_ans_tl
423 }

```

I

5.6.6 Resolution

Given a frame id, a name and an integer path, we resolve any intermediate standalone reference. For example, with A=B and B=C, A is resolved in C. But with A=B+1 and B=C, A is not resolved in C+1. With A=B:D and B=C, A is not resolved in C:D as well.

```

__bnvs_extract_key:NNNTF  \__bnvs_extract_key:NNNTF <id tl var> <name tl var> <path seq var> {<true code>}
                               {<false code>}}

```

Auxiliary function. *<id tl var>* contains a frame id whereas *<name tl var>* contains a range name. If we recognize a key, on return, *<name tl var>* contains the resolved name, *<path seq var>* is prepended with new integer path components, *{<true code>}* is executed, otherwise *{<false code>}* is executed.

```

424 \exp_args_generate:n { VVx }
425 \prg_new_conditional:Npnn \__bnvs_extract_key:NNN
426   #1 #2 #3 { T, F, TF } {
427   \__bnvs_group_begin:
428   \exp_args:NNV
429   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex #2 \l__bnvs_match_seq {

```

This is a correct key, update the path sequence accordingly

```

430   \exp_args:Nx
431   \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
432     \tl_put_left:NV #2 { #1 }
433   }
434   \exp_args:NNnx
435   \seq_set_split:Nnn \l__bnvs_split_seq . {
436     \seq_item:Nn \l__bnvs_match_seq 4
437   }
438   \seq_remove_all:Nn \l__bnvs_split_seq { }
439   \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
440   \seq_if_empty:NTF \l__bnvs_split_seq {

```

No new integer path component is added.

```

441   \cs_set:Npn \:nn ##1 ##2 {
442     \__bnvs_group_end:
443     \tl_set:Nn #1 { ##1 }
444     \tl_set:Nn #2 { ##2 }
445   }
446   \exp_args:NVV \:nn #1 #2
447 } {

```

Some new integer path components are added.

```

448   \cs_set:Npn \:nnn ##1 ##2 ##3 {
449     \__bnvs_group_end:
450     \tl_set:Nn #1 { ##1 }
451     \tl_set:Nn #2 { ##2 }
452     \seq_set_split:Nnn #3 . { ##3 }
453     \seq_remove_all:Nn #3 { }
454   }
455   \exp_args:NVVx
456   \:nnn #1 #2 {
457     \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
458   }

```

```

459 </!gubed>
460 % \end{bnvs.gobble}
461 % \begin{macrocode}
462 }
463 \prg_return_true:
464 } {
465 \__bnvs_group_end:
466 \prg_return_false:
467 }
468 }

```

```

\__bnvs_resolve:NNN $\overline{TF}$  \__bnvs_resolve:NNNTF <id tl var> <name tl var> <path seq var> {<true code>}
{<false code>}

```

When too many nested calls occurred, $\{<false code>\}$ is executed directly. $<id tl var>$, $<name tl var>$ and $<path seq var>$ are meant to contain proper information. On input, $\{<id tl var>\}$ contains a frame id, $\{<name tl var>\}$ contains a range name and $\{<path seq var>\}$ contains the components of an integer path, possibly empty. On return, $<id tl var>$ contains the frame id used, $<name tl var>$ contains the resolved range name and $<path seq var>$ contains the sequence of integer path components that could not be resolved. To resolve a path, $<name_0>.\langle i_1 \rangle.\langle i_2 \rangle \dots \langle i_n \rangle$ is turned into $<name_1>.\langle i_2 \rangle \dots \langle i_n \rangle$ where $<name_0>.\langle i_1 \rangle$ is $<name_1>$, then $<name_2>.\langle i_3 \rangle \dots \langle i_n \rangle$ where $<name_1>.\langle i_2 \rangle$ is $<name_2>$... If the above rule does not apply, $<name_0>.\langle i_1 \rangle.\langle i_2 \rangle \dots \langle i_n \rangle$ may turn into $<name_2>.\langle i_3 \rangle \dots \langle i_n \rangle$ when $<name_0>.\langle i_1 \rangle.\langle i_2 \rangle$ is $<name_2>$... The algorithm is not yet more clever. The resolution algorithm is quite straightforward:

1. If $<name tl var>$ content is the name of an unlimited range, and the first item of this range is exactly another name range with eventually a heading frame identifier or a trailing integer path, then $<name tl var>$ is replaced by this name, the $<id tl var>$ and $\backslash l_bnvs_id_tl$ are updates accordingly and the $<path seq var>$ is prepended with the integer path.
2. If $<path seq var>$ is not empty, append to the right of $<name tl var>$ after a separating dot, all its left elements but the last one and loop. Otherwise return. None of the tl variables must be one of $\backslash l_a_tl$, $\backslash l_b_tl$ or $\backslash l_c_tl$. None of the seq variables must be one of $\backslash l_a_seq$, $\backslash l_b_seq$.

```

469 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
470 #1 #2 #3 { T, F, TF } {
471 \__bnvs_group_begin:

```

Local variables:

- $\backslash l_a_tl$ contains the name with a partial index path currently resolved.
- $\backslash l_a_seq$ contains the index path components currently resolved.
- $\backslash l_b_tl$ contains the resolution.
- $\backslash l_b_seq$ contains the index path components to be resolved.

```

472 \seq_set_eq:NN \l__bnvs_a_seq #3
473 \seq_clear:N \l__bnvs_b_seq
474 \cs_set:Npn \loop: {

```

```

475     \__bnvs_call:TF {
476         \tl_set_eq:NN \l__bnvs_a_tl #2
477         \seq_if_empty:NTF \l__bnvs_a_seq {
478             \exp_args:Nx
479             \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
480                 \cs_set:Nn \loop: { \return_true: }
481             } {
482                 \get_extract:F {
Unknown key <\l_a_tl>/A or the value for key <\l_a_tl>/A does not fit.
483                     \cs_set:Nn \loop: { \return_true: }
484                 }
485             } {
486                 \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
487                 \get_extract:F {
488                     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
489                         \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
490                     }
491                 }
492             }
493         }
494         \loop:
495     } {
496         \__bnvs_group_end:
497         \prg_return_false:
498     }
499 }
500 \cs_set:Npn \get_extract:F ##1 {
501     \exp_args:Nx
502     \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
503         \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
504             \tl_set_eq:NN #2 \l__bnvs_b_tl
505             \seq_set_eq:NN #3 \l__bnvs_b_seq
506             \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
507             \seq_clear:N \l__bnvs_b_seq
508         } { ##1 }
509     } { ##1 }
510 }
511 \cs_set:Npn \return_true: {
512     \cs_set:Npn \:nnn #####1 #####2 #####3 {
513         \__bnvs_group_end:
514         \tl_set:Nn #1 { #####1 }
515         \tl_set:Nn #2 { #####2 }
516         \seq_set_split:Nnn #3 . { #####3 }
517         \seq_remove_all:Nn #3 { }
518     }
519     \exp_args:NVVx
520     \:nnn #1 #2 {
521         \seq_use:Nn #3 .
522     }
523     \prg_return_true:
524 }
525 \loop:
526 }

```

```

__bnvs_resolve_n:NNNTF TF
    <id tl var> <name tl var> <path seq var> {< true code>} {<
    >} false code

```

The difference with the function above without `_n` is that resolution is performed only when there is an integer path afterwards

```

527 \prg_new_conditional:Npnn __bnvs_resolve_n:NNN
528     #1 #2 #3 { T, F, TF } {
529     __bnvs_group_begin:

```

Local variables:

- `\l_a_tl` contains the name with a partial index path currently resolved.
- `\l_a_seq` contains the index path components currently resolved.
- `\l_b_tl` contains the resolution.
- `\l_b_seq` contains the index path components to be resolved.

```

530 \seq_set_eq:NN \l__bnvs_a_seq #3
531 \seq_clear:N \l__bnvs_b_seq
532 \cs_set:Npn \loop: {
533     __bnvs_call:TF {
534         \tl_set_eq:NN \l__bnvs_a_tl #2
535         \seq_if_empty:NTF \l__bnvs_a_seq {
536             \exp_args:Nx
537             __bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
538                 \cs_set:Nn \loop: { \return_true: }
539             } {
540                 \seq_if_empty:NTF \l__bnvs_b_seq {
541                     \cs_set:Nn \loop: { \return_true: }
542                 } {
543                     \get_extract:F {

```

Unknown key `<\l_a_tl>/A` or the value for key `<\l_a_tl>/A` does not fit.

```

544         \cs_set:Nn \loop: { \return_true: }
545     }
546 }
547 }
548 } {
549     \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
550     \get_extract:F {
551         \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
552             \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
553         }
554     }
555 }
556 \loop:
557 } {
558     __bnvs_group_end:
559     \prg_return_false:
560 }
561 }
562 \cs_set:Npn \get_extract:F ##1 {
563     \exp_args:Nx
564     __bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {

```



```

565     \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
566         \tl_set_eq:NN #2 \l__bnvs_b_tl
567         \seq_set_eq:NN #3 \l__bnvs_b_seq
568         \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
569         \seq_clear:N \l__bnvs_b_seq
570     } { ##1 }
571 } { ##1 }
572 }
573 \cs_set:Npn \return_true: {
574     \cs_set:Npn \:nnn #####1 #####2 #####3 {
575         \__bnvs_group_end:
576         \tl_set:Nn #1 { #####1 }
577         \tl_set:Nn #2 { #####2 }
578         \seq_set_split:Nnn #3 . { #####3 }
579         \seq_remove_all:Nn #3 { }
580     }
581     \exp_args:NVVx
582     \:nnn #1 #2 {
583         \seq_use:Nn #3 .
584     }
585     \prg_return_true:
586 }
587 \loop:
588 }

```

```

\__bnvs_resolve:NNNTF TF \__bnvs_resolve:NNNTF <cs:nn> <id tl var> <name tl var> <path seq var> {< true
code>} {< >} false code

```

When too many nested calls occurred, $\{\langle false\ code\rangle\}$ is executed directly. $\langle id\ tl\ var\rangle$, $\langle name\ tl\ var\rangle$ and $\langle path\ seq\ var\rangle$ are meant to contain proper information. To resolve a path, $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ is turned into $\langle name_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ where $\langle name_0\rangle.\langle i_1\rangle$ is $\langle name_1\rangle$, then $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$ where $\langle name_1\rangle.\langle i_2\rangle$ is $\langle name_2\rangle...$. If the above rule does not apply, $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle...\langle i_n\rangle$ may turn into $\langle name_2\rangle.\langle i_3\rangle...\langle i_n\rangle$ when $\langle name_0\rangle.\langle i_1\rangle.\langle i_2\rangle$ is $\langle name_2\rangle...$. We try to match the longest sequence of components first. The algorithm is not yet more clever. In general, $\langle cs:nn\rangle$ is just $\backslash use_i:nn$ but for in place incrementation, we must resolve only when there is an integer path. See the implementation of the $\backslash_bnvs_if_append:...$ conditionals.

```

589 \prg_new_conditional:Npnn \__bnvs_resolve:NNNN
590     #1 #2 #3 #4 { T, F, TF } {
591     #1 {
592         \__bnvs_group_begin:

```

$\backslash l_a_tl$ contains the name with a partial index path currently resolved. $\backslash l_a_seq$ contains the remaining index path components to be resolved. $\backslash l_b_seq$ contains the current index path components to be resolved.

```

593     \tl_set_eq:NN \l__bnvs_a_tl #3
594     \seq_set_eq:NN \l__bnvs_a_seq #4
595     \tl_clear:N \l__bnvs_b_tl
596     \seq_clear:N \l__bnvs_b_seq
597     \cs_set:Npn \return_true: {
598         \cs_set:Npn \:nnn #####1 #####2 #####3 {
599             \__bnvs_group_end:
600             \tl_set:Nn #2 { #####1 }

```

```

601     \tl_set:Nn #3 { ####2 }
602     \seq_set_split:Nnn #4 . { ####3 }
603     \seq_remove_all:Nn #4 { }
604 }
605 \exp_args:NVVx
606 \:nnn #2 #3 {
607     \seq_use:Nn #4 .
608 }
609 \prg_return_true:
610 }
611 \cs_set:Npn \branch:n ##1 {
612     \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
613         \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_b_tl
614         \tl_set:Nn \l__bnvs_a_tl { #3 . }
615         \tl_put_right:Nx \l__bnvs_a_tl { \seq_use:Nn \l__bnvs_a_seq . }
616     } {
617         \cs_set_eq:NN \loop: \return_true:
618     }
619 }
620 \cs_set:Npn \branch:FF ##1 ##2 {
621     \exp_args:Nx
622     \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
623         \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
624             \tl_set_eq:NN #3 \l__bnvs_b_tl
625             \seq_set_eq:NN #4 \l__bnvs_b_seq
626             \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
627         } { ##1 }
628     } { ##2 }
629 }
630 \cs_set:Npn \extract_key:F {
631     \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
632         \tl_set_eq:NN #3 \l__bnvs_b_tl
633         \seq_set_eq:NN #4 \l__bnvs_b_seq
634         \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
635     }
636 }
637 \cs_set:Npn \loop: {
638     \__bnvs_call:TF {
639         \exp_args:Nx
640         \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {

```

If there is a length, no resolution occurs.

```

641     \branch:n { 1 }
642 } {
643     \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
644         \seq_clear:N \l__bnvs_b_seq
645         \tl_set:Nn \l__bnvs_a_tl { #3 . }
646         \tl_put_right:Nx \l__bnvs_a_tl {
647             \seq_use:Nn \l__bnvs_a_seq . .
648         }
649         \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
650         \branch:FF {

```

The value for key $\langle \l_a_{tl} \rangle / L$ is not just a (qualified) name.

```

651 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
652 } {
Unknown key <\l_a_tl>/L.
653 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
654 }
655 } {
656 \branch:FF {
657 \cs_set_eq:NN \loop: \return_true:
658 } {
659 \cs_set:Npn \loop: {
660 \__bnvs_group_end:
661 \prg_return_false:
662 }
663 }
664 }
665 }
666 } {
667 \cs_set:Npn \loop: {
668 \__bnvs_group_end:
669 \prg_return_false:
670 }
671 }
672 \loop:
673 }
674 \loop:
675 } {
676 \prg_return_true:
677 }
678 }
679 \prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
680 #1 #2 #3 #4 { T, F, TF } {
681 #1 {
682 \__bnvs_group_begin:
\l_a_tl contains the name with a partial index path to be resolved. \l_a_seq contains
the remaining index path components to be resolved.
683 \tl_set_eq:NN \l__bnvs_a_tl #3
684 \seq_set_eq:NN \l__bnvs_a_seq #4
685 \cs_set:Npn \return_true: {
686 \cs_set:Npn \:nnn ####1 ####2 ####3 {
687 \__bnvs_group_end:
688 \tl_set:Nn #2 { ####1 }
689 \tl_set:Nn #3 { ####2 }
690 \seq_set_split:Nnn #4 . { ####3 }
691 \seq_remove_all:Nn #4 { }
692 }
693 \exp_args:NVVx
694 \:nnn #2 #3 {
695 \seq_use:Nn #4 .
696 }

```

```

697     \prg_return_true:
698   }
699   \cs_set:Npn \branch:n ##1 {
700     \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
701       \tl_put_right:Nn \l__bnvs_a_tl { . }
702       \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
703     } {
704       \cs_set_eq:NN \loop: \return_true:
705     }
706   }
707   \cs_set:Npn \loop: {
708     \__bnvs_call:TF {
709       \exp_args:Nx
710       \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
711         \branch:n { 1 }
712       } {
713         \exp_args:Nx
714         \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
715           \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
716             \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
717             \tl_set_eq:NN #3 \l__bnvs_b_tl
718             \seq_set_eq:NN #4 \l__bnvs_a_seq
719           } {
720             \branch:n { 2 }
721           }
722         } {
723           \branch:n { 3 }
724         }
725       } {
726       } {
727         \cs_set:Npn \loop: {
728           \__bnvs_group_end:
729           \prg_return_false:
730         }
731       }
732     } \loop:
733   }
734   \loop:
735 } {
736   \prg_return_true:
737 }
738 }

```

5.6.7 Evaluation bricks

<code>__bnvs_fp_round:nN</code>	<code>__bnvs_fp_round:nN {<expression>} <tl variable></code>
<code>__bnvs_fp_round:N</code>	<code>__bnvs_fp_round:N <tl variable></code>

Shortcut for `\fp_eval:n{round(<expression>)}` appended to `<tl variable>`. The second variant replaces the variable content with its rounded floating point evaluation.

```

739 \cs_new:Npn \__bnvs_fp_round:nN #1 #2 {
740   \tl_if_empty:nTF { #1 } {

```

```

741 } {
742   \tl_put_right:Nx #2 {
743     \fp_eval:n { round(#1) }
744   }
745 }
746 }
747 \cs_generate_variant:Nn \__bnvs_fp_round:nN { VN, xN }
748 \cs_new:Npn \__bnvs_fp_round:N #1 {
749   \tl_if_empty:VTF #1 {
750   } {
751     \tl_set:Nx #1 {
752       \fp_eval:n { round(#1) }
753     }
754   }
755 }

```

$\backslash_bnvs_raw_first:nNTF$
 $\backslash_bnvs_raw_first:(xN|VN)TF$

$\backslash_bnvs_raw_first:nNTF$ {<name>} <tl variable> {<true code>} {<false code>}

Append the first index of the <name> slide range to the <tl variable>. Cache the result.
 Execute <true code> when there is a <first>, <false code> otherwise.

```

756 \cs_set:Npn \__bnvs_return_true:nnN #1 #2 #3 {
757   \tl_if_empty:NTF \l__bnvs_ans_tl {
758     \__bnvs_group_end:
759     \__bnvs_gremove:n { #1//#2 }
760     \prg_return_false:
761   } {
762     \__bnvs_fp_round:N \l__bnvs_ans_tl
763     \__bnvs_gput:nV { #1//#2 } \l__bnvs_ans_tl
764     \exp_args:NNNV
765     \__bnvs_group_end:
766     \tl_put_right:Nn #3 \l__bnvs_ans_tl
767     \prg_return_true:
768   }
769 }
770 \cs_set:Npn \__bnvs_return_false:nn #1 #2 {
771   \__bnvs_group_end:
772   \__bnvs_gremove:n { #1//#2 }
773   \prg_return_false:
774 }
775 \prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
776   \__bnvs_if_in:nTF { #1//A } {
777     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//A } }
778     \prg_return_true:
779   } {
780     \__bnvs_group_begin:
781     \tl_clear:N \l__bnvs_ans_tl
782     \__bnvs_get:nNTF { #1//A } \l__bnvs_a_tl {
783       \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
784         \__bnvs_return_true:nnN { #1 } A #2
785       } {
786         \__bnvs_return_false:nn { #1 } A
787       }
788     } {

```

```

789     \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
790     \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
791     \__bnvs_if_append:xNTF {
792     \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
793     } \l__bnvs_ans_tl {
794     \__bnvs_return_true:nnN { #1 } A #2
795     } {
796     \__bnvs_return_false:nn { #1 } A
797     }
798     } {
799     \__bnvs_return_false:nn { #1 } A
800     }
801     } {
802     \__bnvs_return_false:nn { #1 } A
803     }
804     }
805     }
806     }
807 \prg_generate_conditional_variant:Nnn
808     \__bnvs_raw_first:nN { VN, xN } { T, F, TF }

```

__bnvs_if_first:nNTF __bnvs_if_first:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Append the first index of the <name> slide range to the <tl variable>. If no first index was explicitly given, use the counter when available and 1 hen not. Cache the result. Execute <true code> when there is a <first>, <false code> otherwise.

```

809 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
810   \__bnvs_raw_first:nNTF { #1 } #2 {
811     \prg_return_true:
812     } {
813     \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {
814     \bool_set_true:N \l_no_counter_bool
815     \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
816     \__bnvs_return_true:nnN { #1 } A #2
817     } {
818     \__bnvs_return_false:nn { #1 } A
819     }
820     } {
821     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
822     \__bnvs_gput:nn { #1/A } { 1 }
823     \tl_set:Nn #2 { 1 }
824     \__bnvs_return_true:nnN { #1 } A #2
825     } {
826     \__bnvs_return_false:nn { #1 } A
827     }
828     }
829     }
830     }

```

__bnvs_first:nN __bnvs_first:nN {<name>} <tl variable>
__bnvs_first:VN

Append the start of the <name> slide range to the <tl variable>. Cache the result.

```

831 \cs_new:Npn \__bnvs_first:nN #1 #2 {
832   \__bnvs_if_first:nNF { #1 } #2 {
833     \msg_error:nnn { beanoves } { :n } { Range-with-no-first:~#1 }
834   }
835 }
836 \cs_generate_variant:Nn \__bnvs_first:nN { VN }

```

__bnvs_raw_length:nNTF __bnvs_raw_length:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Append the length of the <name> slide range to <tl variable> Execute <true code> when there is a <length>, <false code> otherwise.

```

837 \prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
838   \__bnvs_if_in:nTF { #1//L } {
839     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//L } }
840     \prg_return_true:
841   } {
842     \__bnvs_gput:nn { #1//L } { 0 }
843     \__bnvs_group_begin:
844     \tl_clear:N \l__bnvs_ans_tl
845     \__bnvs_if_in:nTF { #1/L } {
846       \__bnvs_if_append:xNTF {
847         \__bnvs_item:n { #1/L }
848       } \l__bnvs_ans_tl {
849         \__bnvs_return_true:nnN { #1 } L #2
850       } {
851         \__bnvs_return_false:nn { #1 } L
852       }
853     } {
854       \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
855         \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
856           \__bnvs_if_append:xNTF {
857             \l__bnvs_b_tl - (\l__bnvs_a_tl) + 1
858           } \l__bnvs_ans_tl {
859             \__bnvs_return_true:nnN { #1 } L #2
860           } {
861             \__bnvs_return_false:nn { #1 } L
862           }
863         } {
864           \__bnvs_return_false:nn { #1 } L
865         }
866       } {
867         \__bnvs_return_false:nn { #1 } L
868       }
869     }
870   }
871 }
872 \prg_generate_conditional_variant:Nnn
873   \__bnvs_raw_length:nN { VN } { T, F, TF }

```

__bnvs_raw_last:nNTF __bnvs_raw_last:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Put the last index of the fully qualified <name> range to the right of the <tl variable>, when possible. Execute <true code> when a last index was given, <false code> otherwise.

```

874 \prg_new_conditional:Npnn \__bnvs_raw_last:nN #1 #2 { T, F, TF } {
875   \__bnvs_if_in:nTF { #1//Z } {
876     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//Z } }
877     \prg_return_true:
878   } {
879     \__bnvs_gput:nn { #1//Z } { 0 }
880     \__bnvs_group_begin:
881     \tl_clear:N \l__bnvs_ans_tl
882     \__bnvs_if_in:nTF { #1/Z } {
883       \__bnvs_if_append:xNTF {
884         \__bnvs_item:n { #1/Z }
885       } \l__bnvs_ans_tl {
886         \__bnvs_return_true:nnN { #1 } Z #2
887       } {
888         \__bnvs_return_false:nn { #1 } Z
889       }
890     } {
891       \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
892         \__bnvs_get:nNTF { #1/L } \l__bnvs_b_tl {
893           \__bnvs_if_append:xNTF {
894             \l__bnvs_a_tl + (\l__bnvs_b_tl) - 1
895           } \l__bnvs_ans_tl {
896             \__bnvs_return_true:nnN { #1 } Z #2
897           } {
898             \__bnvs_return_false:nn { #1 } Z
899           }
900         } {
901           \__bnvs_return_false:nn { #1 } Z
902         }
903       } {
904         \__bnvs_return_false:nn { #1 } Z
905       }
906     }
907   }
908 }
909 \prg_generate_conditional_variant:Nnn
910   \__bnvs_raw_last:nN { VN } { T, F, TF }

```

`__bnvs_last:nN` `__bnvs_last:nN {<name>} <tl variable>`

`__bnvs_last:VN` Append the last index of the fully qualified `<name>` slide range to `<tl variable>`

```

911 \cs_new:Npn \__bnvs_last:nN #1 #2 {
912   \__bnvs_raw_last:nNF { #1 } #2 {
913     \msg_error:nnn { beanoves } { :n } { Range-with-no-last:~#1 }
914   }
915 }
916 \cs_generate_variant:Nn \__bnvs_last:nN { VN }

```

`__bnvs_if_next:nNTF` `__bnvs_if_next:nNTF {<name>} <tl variable> {<true code>} {<false code>}`

Append the index after the `<name>` slide range to the `<tl variable>`. Execute `<true code>` when there is a `<next>` index, `<false code>` otherwise.


```

917 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
918   \__bnvs_if_in:nTF { #1//N } {
919     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//N } }
920     \prg_return_true:
921   } {
922     \__bnvs_group_begin:
923     \cs_set:Npn \__bnvs_return_true: {
924       \tl_if_empty:NTF \l__bnvs_ans_tl {
925         \__bnvs_group_end:
926         \prg_return_false:
927       } {
928         \__bnvs_fp_round:N \l__bnvs_ans_tl
929         \__bnvs_gput:nV { #1//N } \l__bnvs_ans_tl
930         \exp_args:NNNV
931         \__bnvs_group_end:
932         \tl_put_right:Nn #2 \l__bnvs_ans_tl
933         \prg_return_true:
934       }
935     }
936     \cs_set:Npn \return_false: {
937       \__bnvs_group_end:
938       \prg_return_false:
939     }
940     \tl_clear:N \l__bnvs_a_tl
941     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_a_tl {
942       \__bnvs_if_append:xNTF {
943         \l__bnvs_a_tl + 1
944       } \l__bnvs_ans_tl {
945         \__bnvs_return_true:
946       } {
947         \return_false:
948       }
949     } {
950       \return_false:
951     }
952   }
953 }
954 \prg_generate_conditional_variant:Nnn
955   \__bnvs_if_next:nN { VN } { T, F, TF }

```

```

\__bnvs_next:nN \__bnvs_next:nN {<name>} <tl variable>

```

Append the index after the <name> slide range to the <tl variable>.

```

956 \cs_new:Npn \__bnvs_next:nN #1 #2 {
957   \__bnvs_if_next:nNF { #1 } #2 {
958     \msg_error:nnn { beanoves } { :n } { Range-with-no-next:~#1 }
959   }
960 }
961 \cs_generate_variant:Nn \__bnvs_next:nN { VN }

```

<code>__bnvs_if_index:nnNTF</code> <code>__bnvs_if_index:VVNTF</code> <code>__bnvs_if_index:nnnNTF</code>	<code>__bnvs_if_index:nnNTF {<name>} {<integer>} <tl variable> {<true code>} {<false code>}</code>
--	---

Append the index associated to the {<name>} and {<integer>} slide range to the right of <tl variable>. When <integer shift> is 1, this is the first index, when <integer shift> is 2, this is the second index, and so on. When <integer shift> is 0, this is the index, before the first one, and so on. If the computation is possible, <true code> is executed, otherwise <false code> is executed. The computation may fail when too many recursion calls are made.

```

962 \prg_new_conditional:Npnn \__bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
963   \__bnvs_group_begin:
964   \tl_clear:N \l__bnvs_ans_tl
965   \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
966     \tl_put_right:Nn \l__bnvs_ans_tl { + (#2) - 1 }
967     \exp_args:NNV
968     \__bnvs_group_end:
969     \__bnvs_fp_round:nN \l__bnvs_ans_tl #3
970     \prg_return_true:
971   } {
972     \prg_return_false:
973   }
974 }
975 \prg_generate_conditional_variant:Nnn
976   \__bnvs_if_index:nnN { VVN } { T, F, TF }

```

<code>__bnvs_if_range:nNTF</code>	<code>__bnvs_if_range:nNTF {<name>} <tl variable> {<true code>} {<false code>}</code>
------------------------------------	--

Append the range of the <name> slide range to the <tl variable>. Execute <true code> when there is a <range>, <false code> otherwise.

```

977 \prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {
978   \bool_if:NNTF \l__bnvs_no_range_bool {
979     \prg_return_false:
980   } {
981     \__bnvs_if_in:nTF { #1/ } {
982       \tl_put_right:Nn { 0-0 }
983     } {
984       \__bnvs_group_begin:
985       \tl_clear:N \l__bnvs_a_tl
986       \tl_clear:N \l__bnvs_b_tl
987       \tl_clear:N \l__bnvs_ans_tl
988       \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
989         \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
990           \exp_args:NNNx
991           \__bnvs_group_end:
992           \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
993           \prg_return_true:
994         } {
995           \exp_args:NNNx
996           \__bnvs_group_end:
997           \tl_put_right:Nn #2 { \l__bnvs_a_tl - }

```

```

998         \prg_return_true:
999     }
1000 } {
1001     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
1002         \exp_args:NNNx
1003         \__bnvs_group_end:
1004         \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1005         \prg_return_true:
1006     } {
1007         \__bnvs_group_end:
1008         \prg_return_false:
1009     }
1010 }
1011 }
1012 }
1013 }
1014 \prg_generate_conditional_variant:Nnn
1015   \__bnvs_if_range:nN { VN } { T, F, TF }

```

$\backslash_bnvs_range:nN$ $\backslash_bnvs_range:VN$	$\backslash_bnvs_range:nN \{ \langle name \rangle \} \langle tl\ variable \rangle$ Append the range of the $\langle name \rangle$ slide range to the $\langle tl\ variable \rangle$.
--	--

```

1016 \cs_new:Npn \__bnvs_range:nN #1 #2 {
1017     \__bnvs_if_range:nNF { #1 } #2 {
1018         \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1019     }
1020 }
1021 \cs_generate_variant:Nn \__bnvs_range:nN { VN }

```

$\backslash_bnvs_if_free_counter:nNTF$ $\backslash_bnvs_if_free_counter:VNNTF$	$\backslash_bnvs_if_free_counter:nNTF \{ \langle name \rangle \} \langle tl\ variable \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
---	---

Set the $\langle tl\ variable \rangle$ to the value of the counter associated to the $\{ \langle name \rangle \}$ slide range.

```

1022 \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
1023     \__bnvs_group_begin:
1024     \tl_clear:N \l__bnvs_ans_tl
1025     \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
1026         \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
1027             \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
1028         }
1029     }
1030     \tl_if_empty:NNTF \l__bnvs_ans_tl {
1031         \__bnvs_group_end:
1032         \regex_match:NnNTF \c__bnvs_A_key_Z_regex { #1 } {
1033             \__bnvs_gput:nn { #1/C } { 1 }
1034             \tl_set:Nn #2 { 1 }
1035             \prg_return_true:
1036         } {
1037             \prg_return_false:
1038         }
1039     } {
1040         \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl

```

```

1041     \exp_args:NNNV
1042     \__bnvs_group_end:
1043     \tl_set:Nn #2 \l__bnvs_ans_tl
1044     \prg_return_true:
1045   }
1046 }
1047 \prg_generate_conditional_variant:Nnn
1048   \__bnvs_if_free_counter:nN { VN } { T, F, TF }

```

$\backslash_bnvs_if_counter:nNTF$
 $\backslash_bnvs_if_counter:VNTF$

$\backslash_bnvs_if_counter:nNTF$ $\langle name \rangle$ $\langle tl\ variable \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

Append the value of the counter associated to the $\langle name \rangle$ slide range to the right of $\langle tl\ variable \rangle$. The value always lays in between the range, whenever possible.

```

1049 \prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {
1050   \__bnvs_group_begin:
1051   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_ans_tl {

```

If there is a $\langle first \rangle$, use it to bound the result from below.

```

1052     \tl_clear:N \l__bnvs_a_tl
1053     \__bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
1054       \fp_compare:nNt { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
1055         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1056       }
1057     }

```

If there is a $\langle last \rangle$, use it to bound the result from above.

```

1058     \tl_clear:N \l__bnvs_a_tl
1059     \__bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
1060       \fp_compare:nNt { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
1061         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1062       }
1063     }
1064     \exp_args:NNV
1065     \__bnvs_group_end:
1066     \__bnvs_fp_round:nN \l__bnvs_ans_tl #2
1067     \prg_return_true:
1068   } {
1069     \prg_return_false:
1070   }
1071 }

```

```

1072 \prg_generate_conditional_variant:Nnn
1073   \__bnvs_if_counter:nN { VN } { T, F, TF }

```

$\backslash_bnvs_if_incr:nnTF$
 $\backslash_bnvs_if_incr:nnNTF$
 $\backslash_bnvs_if_incr:(VnN|VVN)TF$

$\backslash_bnvs_if_incr:nnTF$ $\{ \langle name \rangle \}$ $\{ \langle offset \rangle \}$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
 $\backslash_bnvs_if_incr:nnNTF$ $\{ \langle name \rangle \}$ $\{ \langle offset \rangle \}$ $\langle tl\ variable \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

Increment the free counter position accordingly. When requested, put the result in the $\langle tl\ variable \rangle$. In the second version, the result will lay within the declared range.

```

1074 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {

```

```

1075 \__bnvs_group_begin:
1076 \tl_clear:N \l__bnvs_a_tl
1077 \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_a_tl {
1078   \tl_clear:N \l__bnvs_b_tl
1079   \__bnvs_if_append:xNTF { \l__bnvs_a_tl + (#2) } \l__bnvs_b_tl {
1080     \__bnvs_fp_round:N \l__bnvs_b_tl
1081     \__bnvs_gput:nV { #1/C } \l__bnvs_b_tl
1082     \__bnvs_group_end:
1083     \prg_return_true:
1084   } {
1085     \__bnvs_group_end:
1086     \prg_return_false:
1087   }
1088 } {
1089   \__bnvs_group_end:
1090   \prg_return_false:
1091 }
1092 }
1093 \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1094   \__bnvs_if_incr:nnTF { #1 } { #2 } {
1095     \__bnvs_if_counter:nNTF { #1 } #3 {
1096       \prg_return_true:
1097     } {
1098       \prg_return_false:
1099     }
1100   } {
1101     \prg_return_false:
1102   }
1103 }
1104 \prg_generate_conditional_variant:Nnn
1105   \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }

```

$\underline{\underline{\text{_bnvs_if_post:nnNTF}}}$ $\underline{\underline{\text{_bnvs_if_post:(VnN VVN)TF}}}$	$\text{_bnvs_if_post:nnNTF } \{ \langle name \rangle \} \{ \langle offset \rangle \} \langle tl \ variable \rangle \{ \langle true \ code \rangle \} \{ \langle false \ code \rangle \}$
--	---

Put the value of the free counter for the given $\langle name \rangle$ in the $\langle tl \ variable \rangle$ then increment this free counter position accordingly.

```

1106 \prg_new_conditional:Npnn \__bnvs_if_post:nnN #1 #2 #3 { T, F, TF } {
1107   \__bnvs_if_counter:nNTF { #1 } #3 {
1108     \__bnvs_if_incr:nnTF { #1 } { #2 } {
1109       \prg_return_true:
1110     } {
1111       \prg_return_false:
1112     }
1113   } {
1114     \prg_return_false:
1115   }
1116 }
1117 \prg_generate_conditional_variant:Nnn
1118   \__bnvs_if_post:nnN { VnN, VVN } { T, F, TF }

```

5.6.8 Evaluation

<u><code>__bnvs_if_append:nNTF</code></u> <u><code>__bnvs_if_append:(VN xN)TF</code></u>	<code>__bnvs_if_append:nNTF {<integer expression>} <tl variable> {<true code>} {<false code>}</code> Evaluates the <integer expression>, replacing all the named specifications by their static counterpart then put the result to the right of the <tl variable>. Executed within a group. Heavily used by <code>__bnvs_eval_query:nN</code> , where <integer expression> was initially enclosed in ‘?(...)’. Local variables:
<code>\l__bnvs_ans_tl</code>	To feed <tl variable> with. (End definition for <code>\l__bnvs_ans_tl</code> .)
<code>\l__bnvs_split_seq</code>	The sequence of caught query groups and non queries. (End definition for <code>\l__bnvs_split_seq</code> .)
<code>\l__bnvs_split_int</code>	Is the index of the non queries, before all the caught groups. (End definition for <code>\l__bnvs_split_int</code> .)
<code>\int_new:N</code>	<code>\l__bnvs_split_int</code>
<code>\l__bnvs_name_tl</code>	Storage for <code>\l_split_seq</code> items that represent names. (End definition for <code>\l__bnvs_name_tl</code> .)
<code>\l__bnvs_path_tl</code>	Storage for <code>\l_split_seq</code> items that represent integer paths. (End definition for <code>\l__bnvs_path_tl</code> .)
	Catch circular definitions.
<code>\prg_new_conditional:Npnn</code>	<code>__bnvs_if_append:nN #1 #2 { T, F, TF } {</code>
<code>__bnvs_call:TF</code>	{
<code>__bnvs_group_begin:</code>	
	Local variables:
<code>\int_zero:N</code>	<code>\l__bnvs_split_int</code>
<code>\seq_clear:N</code>	<code>\l__bnvs_split_seq</code>
<code>\tl_clear:N</code>	<code>\l__bnvs_id_tl</code>
<code>\tl_clear:N</code>	<code>\l__bnvs_name_tl</code>
<code>\tl_clear:N</code>	<code>\l__bnvs_path_tl</code>
<code>\tl_clear:N</code>	<code>\l__bnvs_group_tl</code>
<code>\tl_clear:N</code>	<code>\l__bnvs_ans_tl</code>
<code>\tl_clear:N</code>	<code>\l__bnvs_a_tl</code>
	Implementation:
<code>\regex_split:NnN</code>	<code>\c__bnvs_split_regex { #1 } \l__bnvs_split_seq</code>
<code>\int_set:Nn</code>	<code>\l__bnvs_split_int { 1 }</code>
<code>\tl_set:Nx</code>	<code>\l__bnvs_ans_tl {</code>
<code>\seq_item:Nn</code>	<code>\l__bnvs_split_seq { \l__bnvs_split_int }</code>
	}

`\switch:nTF` `{(capture group number)}` `{(black code)}` `{(white code)}`

Helper function to locally set the `\l__bnvs_group_tl` variable to the captured group `(capture group number)` and branch.

```

1136 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1137   \tl_set:Nx ##2 {
1138     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1139   }
1140   \tl_if_empty:NTF ##2 {
1141     ##4 } {
1142     ##3
1143   }
1144 }

```

`\prg_return_true:` and `\prg_return_false:` are wrapped locally to close the group and return the proper value.

```

1145 \cs_set:Npn \return_true: {
1146   \fp_round:
1147   \exp_args:NNNV
1148   \__bnvs_group_end:
1149   \tl_put_right:Nn #2 \l__bnvs_ans_tl
1150   \prg_return_true:
1151 }
1152 \cs_set:Npn \fp_round: {
1153   \__bnvs_fp_round:N \l__bnvs_ans_tl
1154 }
1155 \cs_set:Npn \return_false: {
1156   \__bnvs_group_end:
1157   \prg_return_false:
1158 }
1159 \cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {
1160   \switch:nNTF { ##2 } \l__bnvs_id_tl { } {
1161     \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_current_tl
1162     \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1163   }
1164   \switch:nNTF { ##3 } \l__bnvs_path_tl {
1165     \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
1166     \seq_remove_all:Nn \l__bnvs_path_seq { }
1167   } {
1168     \seq_clear:N \l__bnvs_path_seq
1169   }
1170   ##1 \l__bnvs_id_tl \l__bnvs_name_tl \l__bnvs_path_seq {
1171     \cs_set:Npn \: {
1172       ##4
1173     }
1174   } {
1175     \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
1176   }
1177   \:
1178 }
1179 \cs_set:Npn \:T ##1 {
1180   \seq_if_empty:NTF \l__bnvs_path_seq { ##1 } {
1181     \cs_set_eq:NN \loop: \return_false:
1182   }

```

```

1183     }
Main loop.
1184     \cs_set:Npn \loop: {
1185         \int_compare:nNnTF {
1186             \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1187         } {
1188             \switch:nNTF 1 \l__bnvs_name_tl {

• Case ++⟨name⟩⟨integer path⟩.n.

1189             \:NnnT \__bnvs_resolve_n:NNNTF 2 3 {
1190                 \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1191                     \cs_set_eq:NN \loop: \return_false:
1192                 }
1193             }
1194         } {
1195             \switch:nNTF 4 \l__bnvs_name_tl {

• Cases ⟨name⟩⟨integer path⟩....

1196             \switch:nNTF 7 \l__bnvs_a_tl {
1197                 \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1198                     \:T {
1199                         \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1200                             \cs_set_eq:NN \loop: \return_false:
1201                         }
1202                     }
1203                 }

• Case ...length.

1204             } {
1205                 \switch:nNTF 8 \l__bnvs_a_tl {

• Case ...last.

1206                 \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1207                     \:T {
1208                         \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1209                             \cs_set_eq:NN \loop: \return_false:
1210                         }
1211                     }
1212                 }

1213             } {
1214                 \switch:nNTF 9 \l__bnvs_a_tl {

• Case ...next.

```



```

1215         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1216             \:T {
1217                 \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1218                     \cs_set_eq:NN \loop: \return_false:
1219                 }
1220             }
1221         }
1222     } {
1223         \switch:nNTF { 10 } \l__bnvs_a_tl {

```

• Case ...range.

```

1224 \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1225     \:T {
1226         \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
1227             \cs_set_eq:NN \fp_round: \prg_do_nothing:
1228         } {
1229             \cs_set_eq:NN \loop: \return_false:
1230         }
1231     }
1232 }
1233     } {

```

• Case ...n.

```

1234         \switch:nNTF { 12 } \l__bnvs_a_tl {

```

• Case ...+=*<integer>*.

```

1235 \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1236     \:T {
1237         \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1238             \cs_set_eq:NN \loop: \return_false:
1239         }
1240     }
1241 }
1242     } {

```

• Case ...n++.

```

1243         \switch:nNTF { 13 } \l__bnvs_a_tl {
1244             \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1245                 \seq_if_empty:NTF \l__bnvs_path_seq {
1246                     \__bnvs_if_post:VnNF \l__bnvs_name_tl { 1 } \l__bnvs_ans_tl {
1247                         \cs_set_eq:NN \loop: \return_false:
1248                     }
1249                 } {
1250                     \msg_error:nnx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1251                     \cs_set_eq:NN \loop: \return_false:
1252                 }
1253             }
1254         } {
1255             \switch:nNTF { 11 } \l__bnvs_a_tl {

```

• Case ...n++.

```

1256             \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1257                 \seq_if_empty:NTF \l__bnvs_path_seq {
1258 \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1259     \cs_set_eq:NN \loop: \return_false:
1260 }
1261             } {
1262 \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1263 \seq_if_empty:NTF \l__bnvs_path_seq {
1264     \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1265         \cs_set_eq:NN \loop: \return_false:
1266     }
1267 } {
1268     \msg_error:nxx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1269     \cs_set_eq:NN \loop: \return_false:
1270 }
1271         }
1272     }
1273 } {
1274     \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1275         \seq_if_empty:NTF \l__bnvs_path_seq {
1276 \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1277     \cs_set_eq:NN \loop: \return_false:
1278 }
1279         } {
1280 \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1281 \seq_if_empty:NTF \l__bnvs_path_seq {
1282     \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1283         \cs_set_eq:NN \loop: \return_false:
1284     }
1285 } {
1286     \msg_error:nxx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1287     \cs_set_eq:NN \loop: \return_false:
1288 }
1289         }
1290     }
1291 }
1292 }
1293 }
1294 }
1295 }
1296 }
1297 }
1298 } {

```

No name.

```

1299     }
1300 }
1301 \int_add:Nn \l__bnvs_split_int { 14 }
1302 \tl_put_right:Nx \l__bnvs_ans_tl {
1303     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1304 }
1305 \loop:
1306 } {

```

```

1307         \return_true:
1308     }
1309 }
1310 \loop:
1311 } {
1312     \msg_error:nx { beanoves } { :n } { Too-many-calls:~ #1 }
1313     \prg_return_false:
1314 }
1315 }
1316 \prg_generate_conditional_variant:Nnn
1317 \__bnvs_if_append:nN { VN, xN } { T, F, TF }

```

__bnvs_if_eval_query:nNTF __bnvs_if_eval_query:nNTF {<overlay query>} <tl variable> {<true code>} {<false code>}

Evaluates the single <overlay query>, which is expected to contain no comma. Extract a range specification from the argument, replaces all the *named overlay specifications* by their static counterparts, make the computation then append the result to the right of the <seq variable>. Ranges are supported with the colon syntax. This is executed within a local group. Below are local variables and constants.

\l__bnvs_a_tl Storage for the first index of a range.

(End definition for \l__bnvs_a_tl.)

\l__bnvs_b_tl Storage for the last index of a range, or its length.

(End definition for \l__bnvs_b_tl.)

\c__bnvs_A_cln_Z_regex Used to parse slide range overlay specifications. Next are the capture groups.

(End definition for \c__bnvs_A_cln_Z_regex.)

```

1318 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1319     \A \s* (?

```

- 2: <first>

```

1320         ( [^:]* ) \s* :

```

- 3: second optional colon

```

1321         (:)? \s*

```

- 4: <length>

```

1322         ( [^:]* )

```

- 5: standalone <first>

```

1323         | ( [^:]+ )

```

```

1324     ) \s* \Z

```

```

1325 }

```

```

1326 \prg_new_conditional:Npnn \__bnvs_if_eval_query:n #1 #2 { T, F, TF } {

```

```

1327  \__bnvs_call_greset:
1328  \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
1329      #1
1330  } \l__bnvs_match_seq {
1331      \bool_set_false:N \l__bnvs_no_counter_bool
1332      \bool_set_false:N \l__bnvs_no_range_bool

```

\switch:nNTF $\langle\text{capture group number}\rangle$ $\langle\text{tl variable}\rangle$ $\langle\text{black code}\rangle$ $\langle\text{white code}\rangle$

Helper function to locally set the $\langle\text{tl variable}\rangle$ to the captured group $\langle\text{capture group number}\rangle$ and branch depending on the emptiness of this variable.

```

1333  \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1334      \tl_set:Nx ##2 {
1335          \seq_item:Nn \l__bnvs_match_seq { ##1 }
1336      }
1337      \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1338  }
1339  \switch:nNTF 5 \l__bnvs_a_tl {

```

● Single expression

```

1340      \bool_set_false:N \l__bnvs_no_range_bool
1341      \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1342          \prg_return_true:
1343      } {
1344          \prg_return_false:
1345      }
1346  } {
1347      \switch:nNTF 2 \l__bnvs_a_tl {
1348          \switch:nNTF 4 \l__bnvs_b_tl {
1349              \switch:nNTF 3 \l__bnvs_c_tl {

```

● $\langle\text{first}\rangle::\langle\text{last}\rangle$ range

```

1350          \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1351              \tl_put_right:Nn #2 { - }
1352              \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1353                  \prg_return_true:
1354              } {
1355                  \prg_return_false:
1356              }
1357          } {
1358              \prg_return_false:
1359          }
1360      } {

```

● $\langle\text{first}\rangle:\langle\text{length}\rangle$ range

```

1361          \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1362              \tl_put_right:Nx #2 { - }
1363              \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1 }
1364              \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1365                  \prg_return_true:
1366              } {
1367                  \prg_return_false:
1368              }
1369          } {

```

```

1370         \prg_return_false:
1371     }
1372 }
1373 } {
    • <first>: and <first>:: range
1374     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1375         \tl_put_right:Nn #2 { - }
1376         \prg_return_true:
1377     } {
1378         \prg_return_false:
1379     }
1380 }
1381 } {
1382     \switch:nNTF 4 \l__bnvs_b_tl {
1383         \switch:nNTF 3 \l__bnvs_c_tl {
    • ::<last> range
1384         \tl_put_right:Nn #2 { - }
1385         \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1386             \prg_return_true:
1387         } {
1388             \prg_return_false:
1389         }
1390     } {
1391 \msg_error:nnx { beanoves } { :n } { Syntax-error(Missing~first):~#1 }
1392     }
1393 } {
    • : or :: range
1394         \seq_put_right:Nn #2 { - }
1395     }
1396 }
1397 }
1398 } {

Error
1399 \msg_error:nnn { beanoves } { :n } { Syntax~error:~#1 }
1400 }
1401 }

```

`__bnvs_eval:nN` `__bnvs_eval:nN {⟨overlay query list⟩} ⟨tl variable⟩`

This is called by the *named overlay specifications* scanner. Evaluates the comma separated list of *⟨overlay query⟩*'s, replacing all the named overlay specifications and integer expressions by their static counterparts by calling `__bnvs_eval_query:nN`, then append the result to the right of the *⟨tl variable⟩*. This is executed within a local group. Below are local variables and constants used throughout the body of this function.

`\l__bnvs_query_seq` Storage for a sequence of *⟨query⟩*'s obtained by splitting a comma separated list.

(End definition for `\l__bnvs_query_seq`.)

`\l__bnvs_ans_seq` Storage of the evaluated result.

(End definition for `\l__bnvs_ans_seq`.)

`\c__bnvs_comma_regex` Used to parse slide range overlay specifications.

1402 `\regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }`

(End definition for `\c__bnvs_comma_regex`.)

No other variable is used.

1403 `\cs_new:Npn __bnvs_eval:nN #1 #2 {`

1404 `__bnvs_group_begin:`

Local variables declaration

1405 `\seq_clear:N \l__bnvs_query_seq`

1406 `\seq_clear:N \l__bnvs_ans_seq`

In this main evaluation step, we evaluate the integer expression and put the result in a variable which content will be copied after the group is closed. We authorize comma separated expressions and *⟨first⟩::⟨last⟩* range expressions as well. We first split the expression around commas, into `\l_query_seq`.

1407 `\regex_split:NnN \c__bnvs_comma_regex { #1 } \l__bnvs_query_seq`

Then each component is evaluated and the result is stored in `\l__bnvs_ans_seq` that we have clear before use.

1408 `\seq_map_inline:Nn \l__bnvs_query_seq {`

1409 `\tl_clear:N \l__bnvs_ans_tl`

1410 `__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {`

1411 `\seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl`

1412 `} {`

1413 `\seq_map_break:n {`

1414 `\msg_fatal:nnn { beanoves } { :n } { Circular-dependency-in~#1}`

1415 `}`

1416 `}`

1417 `}`

We have managed all the comma separated components, we collect them back and append them to *⟨tl variable⟩*.

1418 `\exp_args:NNNx`

1419 `__bnvs_group_end:`

1420 `\tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }`

1421 `}`

1422 `\cs_generate_variant:Nn __bnvs_eval:nN { VN, xN }`

\BeanovesEval

\BeanovesEval [*<tl variable>*] {*<overlay queries>*}

<overlay queries> is the argument of ?(...) instructions. This is a comma separated list of single *<overlay query>*'s.

This function evaluates the *<overlay queries>* and store the result in the *<tl variable>* when provided or leave the result in the input stream. Forwards to `__bnvs_eval:nN` within a group. `\l_ans_tl` is used locally to store the result.

```
1423 \NewDocumentCommand \BeanovesEval { s o m } {
1424   \__bnvs_group_begin:
1425   \tl_clear:N \l__bnvs_ans_tl
1426   \IfBooleanTF { #1 } {
1427     \bool_set_true:N \l__bnvs_no_counter_bool
1428   } {
1429     \bool_set_false:N \l__bnvs_no_counter_bool
1430   }
1431   \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
1432   \IfValueTF { #2 } {
1433     \exp_args:NNNV
1434     \__bnvs_group_end:
1435     \tl_set:Nn #2 \l__bnvs_ans_tl
1436   } {
1437     \exp_args:NV
1438     \__bnvs_group_end: \l__bnvs_ans_tl
1439   }
1440 }
```

5.6.9 Resetting slide ranges

\BeanovesReset

\beanovesReset [*<first value>*] {*<Slide range name>*}

```
1441 \NewDocumentCommand \BeanovesReset { O{1} m } {
1442   \__bnvs_reset:nn { #1 } { #2 }
1443   \ignorespaces
1444 }
```

Forwards to `__bnvs_reset:nn`.

__bnvs_reset:nn

__bnvs_reset:nn {*<first value>*} {*<slide range name>*}

Reset the counter to the given *<first value>*. Clean the cached values also.

```
1445 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1446   \bool_if:nTF {
1447     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1448   } {
1449     \__bnvs_gremove:n { #2/C }
1450     \__bnvs_gremove:n { #2//A }
1451     \__bnvs_gremove:n { #2//L }
1452     \__bnvs_gremove:n { #2//Z }
1453     \__bnvs_gremove:n { #2//N }
1454     \__bnvs_gput:nn { #2/C0 } { #1 }
1455   } {
1456     \msg_warning:nnn { beanoves } { :n } { Unknown-name:~#2 }
1457   }
```

```
1458 }  
1459 \makeatother  
1460 \ExplSyntaxOff
```