

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>
```

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_new:N \l__bnvs_a_tl
12 \tl_new:N \l__bnvs_b_tl
13 \tl_new:N \l__bnvs_c_tl
14 \tl_new:N \l__bnvs_id_tl
15 \tl_new:N \l__bnvs_ans_tl
16 \tl_new:N \l__bnvs_name_tl
17 \tl_new:N \l__bnvs_path_tl
18 \tl_new:N \l__bnvs_group_tl
19 \tl_new:N \l__bnvs_query_tl
20 \tl_new:N \l__bnvs_token_tl
```

```

21 \int_new:N \g__bnvs_call_int
22 \int_new:N \l__bnvs_depth_int
23 \seq_new:N \l__bnvs_a_seq
24 \seq_new:N \l__bnvs_b_seq
25 \seq_new:N \l__bnvs_ans_seq
26 \seq_new:N \l__bnvs_match_seq
27 \seq_new:N \l__bnvs_split_seq
28 \seq_new:N \l__bnvs_path_seq
29 \seq_new:N \l__bnvs_query_seq
30 \seq_new:N \l__bnvs_token_seq
31 \bool_new:N \l__bnvs_no_counter_bool
32 \bool_new:N \l__bnvs_no_range_bool
33 \bool_new:N \l__bnvs_in_frame_bool
34 \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`.)

```

35 \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.

```

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

```

`__bnvs_call:TF` `__bnvs_call_do:TF` $\{ \langle \text{true code} \rangle \} \{ \langle \text{false code} \rangle \}$

Decrement the `\g__bnvs_call_int` counter globally and execute $\langle \text{true code} \rangle$ if we have not reached 0, $\langle \text{false code} \rangle$ otherwise.

```

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

```

5.6 Overlay specification

5.6.1 In slide range definitions

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

$\langle \text{id} \rangle$! $\langle \text{name} \rangle$ /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.

47 `\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.

```

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

```

```

\__bnvs_if_in_p:n ★
\__bnvs_if_in_p:V ★
\__bnvs_if_in:nTF ★
\__bnvs_if_in:VTF ★

```

```

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

```

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

```

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

```



```

83   } {
84     \prg_return_false:
85   }
86 }
87 \prg_generate_conditional_variant:Nnn \__bnvs_if_in:n {V} { p, T, F, TF }

```

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

```

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

```

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.

```

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

```

(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.

```

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

```

(End definition for `\c__bnvs_id_regex`.)

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

```

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

```

(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.

```

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

110 `\A ((\ur{c__bnvs_id_regex} ?) \ur{c__bnvs_name_regex})`

- 5: the path, if any.

111 `(\ur{c__bnvs_path_regex}) \Z`
 112 `}`
 113

(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.

114 `\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.

115 `\regex_const:Nn \c__bnvs_list_regex {`
 116 `\A \[\s*`

Capture groups:

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

117 `([^\]]%[---`
 118 `] *?)`
 119 `\s* \] \Z`
 120 `}`

(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.

121 `\regex_const:Nn \c__bnvs_split_regex {`
 122 `\s* (? :`

We start with ‘++’ instrussions².

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

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

- 3: optionally followed by an integer path

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

We continue with other expressions

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

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

125 | ((\ur{c__bnvs_id_regex}?) \ur{c__bnvs_name_regex})

- 6: optionally followed by an integer path

126 (\ur{c__bnvs_path_regex})

Next comes another branching

127 (?:

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

128 \. 1(e)ngth

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

129 | \. 1(a)st

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

130 | \. ne(x)t

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

131 | \. (r)ange

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

132 | (?: \. (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.

133 \s* \ += \s* (\S+)

- 13: the post increment

134 | (\+)\+)?

135)?

136) \s*

137 }

(End definition for `c__bnvs_split_regex`.)

5.6.3 beamer.cls interface

Work in progress.

```

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

`\l__bnvs_match_seq` Local storage for the match result.

(End definition for `\l__bnvs_match_seq`.)

<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 $\langle command \rangle$ { $\langle key \rangle$ }

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_do_parse:Nnn __bnvs_do_parse:Nnn $\langle command \rangle$ { $\langle full name \rangle$ }

Auxiliary function for __bnvs_parse:Nn. $\langle command \rangle$ is __bnvs_range:nVVV at run-time and must have signature nVVV.

```

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

```

This is not a list.

```

203   \tl_clear:N \l__bnvs_a_tl
204   \tl_clear:N \l__bnvs_b_tl
205   \tl_clear:N \l__bnvs_c_tl
206   \regex_split:NnN \c__bnvs_colons_regex { #3 } \l__bnvs_split_seq
207   \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_a_tl {

```

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

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

This is a one colon range.

```
210 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
```

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

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

A :: was expected:

```
213 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(1):~#3 }
214 } {
215 \int_compare:nNnT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
216 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(2):~#3 }
217 }
218 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
```

\l_c_tl may contain the $\langle end \rangle$.

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

This is a two colon range.

```
225 \int_compare:nNnT { \tl_count:N \l__bnvs_b_tl } > { 1 } {
226 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(4):~#3 }
227 }
228 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
```

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

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

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

```
232 \seq_if_empty:NF \l__bnvs_split_seq {
233 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(5):~#3 }
234 }
235 } {
236 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(6):~#3 }
237 }
238 } {
239 \tl_clear:N \l__bnvs_b_tl
240 }
241 }
242 }
243 }
```

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.

```
244 \bool_if:nF {
245 \tl_if_empty_p:N \l__bnvs_a_tl
246 || \tl_if_empty_p:N \l__bnvs_b_tl
```

```

247     || \tl_if_empty_p:N \l__bnvs_c_tl
248   } {
249 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(7):-#3 }
250   }
251   #1 { #2 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
252 }
253 \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>.

```

254 \prg_new_conditional:Npnn \__bnvs_id_name_set:nN #1 #2 #3 { T, F, TF } {
255   \__bnvs_group_begin:
256   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
257     #1
258   } \l__bnvs_match_seq {
259     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
260     \tl_if_empty:NTF #2 {
261       \exp_args:NNNx
262       \__bnvs_group_end:
263       \tl_set:Nn #3 { \l__bnvs_id_current_tl #1 }
264       \tl_set_eq:NN #2 \l__bnvs_id_current_tl
265     } {
266       \cs_set:Npn \:n ##1 {
267         \__bnvs_group_end:
268         \tl_set:Nn #2 { ##1 }
269         \tl_set:Nn \l__bnvs_id_current_tl { ##1 }
270       }
271       \exp_args:NV
272       \:n #2
273       \tl_set:Nn #3 { #1 }
274     }
275   \prg_return_true:
276 } {
277   \__bnvs_group_end:
278   \prg_return_false:
279 }
280 }

281 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
282   \__bnvs_group_begin:
283   \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
284     \regex_extract_once:NnNTF \c__bnvs_list_regex {
285       #3
286     } \l__bnvs_match_seq {

```

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

```

287     \exp_args:NNx
288     \seq_set_from_clist:Nn \l__bnvs_match_seq {
289       \seq_item:Nn \l__bnvs_match_seq { 2 }

```

```

290     }
291     \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
292       \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl.##1 } { ##2 }
293     }
294   } {
295     \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
296   }
297 } {
298   \msg_error:nnn { beanoves } { :n } { Invalid~key:~#2 }
299 }

```

We export \l__bnvs_id_tl:

```

300   \exp_args:NNNV
301   \__bnvs_group_end:
302   \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
303 }

```

\Beanoves \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.

```

304 \NewDocumentCommand \Beanoves { sm } {
305   \tl_if_eq:NnT \@currenvir { document } {
306     \__bnvs_gclear:
307   }
308   \IfBooleanTF {#1} {
309     \keyval_parse:nnn {
310       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
311     } {
312       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
313     }
314   } {
315     \keyval_parse:nnn {
316       \__bnvs_parse:Nn \__bnvs_range:nVVV
317     } {
318       \__bnvs_parse:Nnn \__bnvs_range:nVVV
319     }
320   }
321   { #2 }
322   \ignorespaces
323 }

```

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

```

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

```

5.6.5 Scanning named overlay specifications

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

<hr/>	<hr/>
<code>\beamer@frame</code>	<code>\beamer@frame {\langle overlay specification \rangle}</code>
<code>\beamer@masterdecode</code>	<code>\beamer@masterdecode {\langle overlay specification \rangle}</code>
	Preprocess $\langle overlay specification \rangle$ before beamer reads it.
<code>\l__bnvs_ans_tl</code>	Storage for the translated overlay specification, where $\langle ?(\dots) \rangle$ instructions are replaced by their static counterparts.
	(End definition for <code>\l__bnvs_ans_tl</code> .)
	Save the original macro <code>\beamer@masterdecode</code> and then override it to properly preprocess the argument.
	<pre> 325 \cs_set_eq:NN __bnvs_beamer@frame \beamer@frame 326 \cs_set:Npn \beamer@frame < #1 > { 327 __bnvs_group_begin: 328 \tl_clear:N \l__bnvs_ans_tl 329 __bnvs_scan:nNN { #1 } __bnvs_eval:nN \l__bnvs_ans_tl 330 \exp_args:NNNV 331 __bnvs_group_end: 332 __bnvs_beamer@frame < \l__bnvs_ans_tl > 333 } 334 \cs_set_eq:NN __bnvs_beamer@masterdecode \beamer@masterdecode 335 \cs_set:Npn \beamer@masterdecode #1 { 336 __bnvs_group_begin: 337 \tl_clear:N \l__bnvs_ans_tl 338 __bnvs_scan:nNN { #1 } __bnvs_eval:nN \l__bnvs_ans_tl 339 \exp_args:NNV 340 __bnvs_group_end: 341 __bnvs_beamer@masterdecode \l__bnvs_ans_tl 342 }</pre>
<hr/>	<hr/>
<code>__bnvs_scan:nNN</code>	<code>__bnvs_scan:nNN {\langle named overlay expression \rangle} \langle eval \rangle \langle tl variable \rangle</code>
	Scan the $\langle named overlay expression \rangle$ argument and feed the $\langle tl variable \rangle$ replacing $\langle ?(\dots) \rangle$ instructions by their static counterpart with help from the $\langle eval \rangle$ function, which is <code>__bnvs_eval:nN</code> . A group is created to use local variables:
<code>\l__bnvs_ans_tl</code>	The token list that will be appended to $\langle tl variable \rangle$ on return.
	(End definition for <code>\l__bnvs_ans_tl</code> .)
<code>\l__bnvs_depth_int</code>	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 $\langle ?(\dots) \rangle$ instruction.
	(End definition for <code>\l__bnvs_depth_int</code> .)
<code>\l__bnvs_query_tl</code>	Storage for the overlay query expression to be evaluated.
	(End definition for <code>\l__bnvs_query_tl</code> .)
<code>\l__bnvs_token_seq</code>	The $\langle overlay expression \rangle$ is split into the sequence of its tokens.
	(End definition for <code>\l__bnvs_token_seq</code> .)
<code>\l__bnvs_token_tl</code>	Storage for just one token.
	(End definition for <code>\l__bnvs_token_tl</code> .)

```

343 \cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 {
344   \__bnvs_group_begin:
345   \tl_clear:N \l__bnvs_ans_tl
346   \seq_clear:N \l__bnvs_token_seq

Explode the <named overlay expression> into a list of tokens:
347   \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.

```

348   \cs_set:Npn \scan_question: {
349     \seq_pop_left:NNT \l__bnvs_token_seq \l__bnvs_token_tl {
350       \tl_if_eq:NnTF \l__bnvs_token_tl { ? } {
351         \require_open:
352       } {
353         \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
354         \scan_question:
355       }
356     }
357   }

```

\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.

```

358   \cs_set:Npn \require_open: {
Get next token.
359     \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
360       \tl_if_eq:NnTF \l__bnvs_token_tl { ( % )
361     } {

```

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

```

362     \int_set:Nn \l__bnvs_depth_int { 1 }

```

Record the forthcoming content in the \l__bnvs_query_tl variable, up to the next balancing ‘)’.

```

363     \tl_clear:N \l__bnvs_query_tl
364     \require_close:
365   } {

```

Ignore this token and loop.

```

366     \require_open:
367   }
368 } {

```

End reached but no opening parenthesis found, raise.

```

369     \msg_fatal:nxx { beanoves } { :n } {Missing~'(%---)
370       ~after~a~?:~#1}
371   }
372 }

```

`\require_close:`

`\require_close:`

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

```
373 \cs_set:Npn \require_close: {  
Get next token.  
374 \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {  
375 \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)  
376 } {
```

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

```
377 \int_incr:N \l__bnvs_depth_int  
378 \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl  
379 \require_close:  
380 } {
```

This is not a '('.

```
381 \tl_if_eq:NnTF \l__bnvs_token_tl { %(---  
382 )  
383 } {
```

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

```
384 \int_decr:N \l__bnvs_depth_int  
385 \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 ?.

```
386 \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl  
387 \scan_question:  
388 } {
```

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.

```
389 \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl  
390 \require_close:  
391 }  
392 } {
```

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

```
393 \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl  
394 \require_close:  
395 }  
396 }  
397 } {
```

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 ')':

```
398 \msg_error:nnx { beanoves } { :n } {Missing~%(---  
399 `)'::~#1 }  
400 \tl_put_right:Nx \l__bnvs_query_tl {  
401 \prg_replicate:nn { \l__bnvs_depth_int } {%(---  
402 )}
```

```

403     }
404     \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
405   }
406 }

```

Run the top level loop to scan for a ‘?’:

```

407   \scan_question:
408   \exp_args:NNNV
409   \__bnvs_group_end:
410   \tl_put_right:Nn #3 \l__bnvs_ans_tl
411 }

```

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. $\langle id\ tl\ var \rangle$ contains a frame id whereas $\langle name\ tl\ var \rangle$ contains a range name. If we recognize a key, on return, $\langle name\ tl\ var \rangle$ contains the resolved name, $\langle path\ seq\ var \rangle$ is prepended with new integer path components, $\{ \langle true\ code \rangle \}$ is executed, otherwise $\{ \langle false\ code \rangle \}$ is executed.

```

412 \exp_args_generate:n { VVx }
413 \prg_new_conditional:Npnn \__bnvs_extract_key:NNN
414   #1 #2 #3 { T, F, TF } {
415   \__bnvs_group_begin:
416   \exp_args:NNV
417   \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

```

418   \exp_args:Nx
419   \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
420     \tl_put_left:NV #2 { #1 }
421   }
422   \exp_args:NNnx
423   \seq_set_split:Nnn \l__bnvs_split_seq . {
424     \seq_item:Nn \l__bnvs_match_seq 4
425   }
426   \seq_remove_all:Nn \l__bnvs_split_seq { }
427   \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
428   \seq_if_empty:NNTF \l__bnvs_split_seq {

```

No new integer path component is added.

```

429   \cs_set:Npn \:nn ##1 ##2 {
430     \__bnvs_group_end:
431     \tl_set:Nn #1 { ##1 }
432     \tl_set:Nn #2 { ##2 }
433   }
434   \exp_args:NVV \:nn #1 #2
435 } {

```

Some new integer path components are added.

```

436     \cs_set:Npn \:nnn ##1 ##2 ##3 {
437         \__bnvs_group_end:
438         \tl_set:Nn #1 { ##1 }
439         \tl_set:Nn #2 { ##2 }
440         \seq_set_split:Nnn #3 . { ##3 }
441         \seq_remove_all:Nn #3 { }
442     }
443     \exp_args:NVVx
444     \:nnn #1 #2 {
445         \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
446     }
447     \!gubed
448     % \end{gobble}
449     % \begin{macrocode}
450     }
451     \prg_return_true:
452 } {
453     \__bnvs_group_end:
454     \prg_return_false:
455 }
456 }
```

__bnvs_resolve:NNNTF __bnvs_resolve:NNNTF $\langle id\ tl\ var \rangle$ $\langle name\ tl\ var \rangle$ $\langle path\ seq\ var \rangle$ $\{\langle true\ code \rangle\}$
 $\{\langle false\ code \rangle\}$

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. On input, $\{\langle id\ tl\ var \rangle\}$ contains a frame id, $\{\langle name\ tl\ var \rangle\}$ contains a range name and $\{\langle path\ seq\ var \rangle\}$ contains the components of an integer path, possibly empty. On return, $\langle id\ tl\ var \rangle$ contains the frame id used, $\langle name\ tl\ var \rangle$ contains the resolved range name and $\langle path\ seq\ var \rangle$ contains the sequence of integer path components that could not be resolved. 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 $\langle name_2 \rangle...$ The algorithm is not yet more clever. The resolution algorithm is quite straightforward:

1. If $\langle name\ tl\ var \rangle$ 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 $\langle name\ tl\ var \rangle$ is replaced by this name, the $\langle id\ tl\ var \rangle$ and $\l_1_bnvs_id_tl$ are updates accordingly and the $\langle path\ seq\ var \rangle$ is prepended with the integer path.
2. If $\langle path\ seq\ var \rangle$ is not empty, append to the right of $\langle name\ tl\ var \rangle$ 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 $\l_1_a_tl$, $\l_1_b_tl$ or $\l_1_c_tl$. None of the seq variables must be one of $\l_1_a_seq$, $\l_1_b_seq$.

```

457 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
458     #1 #2 #3 { T, F, TF } {
```

459 __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.

```
460 \seq_set_eq:NN \l__bnvs_a_seq #3
461 \seq_clear:N \l__bnvs_b_seq
462 \cs_set:Npn \loop: {
463   \__bnvs_call:TF {
464     \tl_set_eq:NN \l__bnvs_a_tl #2
465     \seq_if_empty:NNTF \l__bnvs_a_seq {
466       \exp_args:Nx
467       \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
468         \cs_set:Nn \loop: { \return_true: }
469       } {
470         \get_extract:F {
```

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

```
471       \cs_set:Nn \loop: { \return_true: }
472     }
473   } {
474     \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
475     \get_extract:F {
476       \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
477         \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
478       }
479     }
480   }
481 }
482 \loop:
483 } {
484   \__bnvs_group_end:
485   \prg_return_false:
486 }
487 }
488 \cs_set:Npn \get_extract:F ##1 {
489   \exp_args:Nx
490   \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
491     \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
492       \tl_set_eq:NN #2 \l__bnvs_b_tl
493       \seq_set_eq:NN #3 \l__bnvs_b_seq
494       \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
495       \seq_clear:N \l__bnvs_b_seq
496     } { ##1 }
497   } { ##1 }
498 }
499 \cs_set:Npn \return_true: {
500   \cs_set:Npn \:nnn #####1 ####2 ####3 {
501     \__bnvs_group_end:
```

```

502     \tl_set:Nn #1 { ####1 }
503     \tl_set:Nn #2 { ####2 }
504     \seq_set_split:Nnn #3 . { ####3 }
505     \seq_remove_all:Nn #3 { }
506   }
507   \exp_args:NVVx
508   \:nnn #1 #2 {
509     \seq_use:Nn #3 .
510   }
511   \prg_return_true:
512 }
513 \loop:
514 }

```

_bnvs_resolve_n:NNNTF \overline{TF} _bnvs_resolve_n:NNNTF $\langle id\ tl\ var \rangle$ $\langle name\ tl\ var \rangle$ $\langle path\ seq\ var \rangle$ { $\langle true\ code \rangle$ } { $\langle \rangle$ } false code

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

```

515 \prg_new_conditional:Npnn \_bnvs_resolve_n:NNN
516   #1 #2 #3 { T, F, TF } {
517   \_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.

```

518   \seq_set_eq:NN \l__bnvs_a_seq #3
519   \seq_clear:N \l__bnvs_b_seq
520   \cs_set:Npn \loop: {
521     \_bnvs_call:TF {
522       \tl_set_eq:NN \l__bnvs_a_tl #2
523       \seq_if_empty:NTF \l__bnvs_a_seq {
524         \exp_args:Nx
525         \_bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
526           \cs_set:Nn \loop: { \return_true: }
527         } {
528           \seq_if_empty:NTF \l__bnvs_b_seq {
529             \cs_set:Nn \loop: { \return_true: }
530           } {
531             \get_extract:F {

```

Unknown key $\langle \l_a_tl \rangle/A$ or the value for key $\langle \l_a_tl \rangle/A$ does not fit.

```

532       \cs_set:Nn \loop: { \return_true: }
533     }
534   }
535 } {
536   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
537 }

```

```

538     \get_extract:F {
539         \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
540             \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
541         }
542     }
543 }
544 \loop:
545 } {
546     \__bnvs_group_end:
547     \prg_return_false:
548 }
549 }
550 \cs_set:Npn \get_extract:F ##1 {
551     \exp_args:Nx
552     \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
553         \__bnvs_extract_key:NNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
554             \tl_set_eq:NN #2 \l__bnvs_b_tl
555             \seq_set_eq:NN #3 \l__bnvs_b_seq
556             \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
557             \seq_clear:N \l__bnvs_b_seq
558         } { ##1 }
559     } { ##1 }
560 }
561 \cs_set:Npn \return_true: {
562     \cs_set:Npn \:nnn ####1 ####2 ####3 {
563         \__bnvs_group_end:
564         \tl_set:Nn #1 { ####1 }
565         \tl_set:Nn #2 { ####2 }
566         \seq_set_split:Nnn #3 . { ####3 }
567         \seq_remove_all:Nn #3 { }
568     }
569     \exp_args:NVVx
570     \:nnn #1 #2 {
571         \seq_use:Nn #3 .
572     }
573     \prg_return_true:
574 }
575 \loop:
576 }

```

`__bnvs_resolve:NNNTF` `__bnvs_resolve:NNNTF <cs:nn> <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. To resolve a path, `<name0>.<i1>.<i2>...<in>` is turned into `<name1>.<i2>...<in>` where `<name0>.<i1>` is `<name1>`, then `<name2>.<i3>...<in>` where `<name1>.<i2>` is `<name2>...`. If the above rule does not apply, `<name0>.<i1>.<i2>...<in>` may turn into `<name2>.<i3>...<in>` when `<name0>.<i1>.<i2>` is `<name2>...`. We try to match the longest sequence of components first. The algorithm is not yet more clever. In general, `<cs:nn>` is just `\use_i:nn` but for in place incrementation, we must resolve only when there is an integer path. See the implementation of the `__bnvs_if_append:...` conditionals.


```

577 \prg_new_conditional:Npnn \__bnvs_resolve:NNNN
578   #1 #2 #3 #4 { T, F, TF } {
579     #1 {
580       \__bnvs_group_begin:

```

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

```

581       \tl_set_eq:NN \l__bnvs_a_tl #3
582       \seq_set_eq:NN \l__bnvs_a_seq #4
583       \tl_clear:N \l__bnvs_b_tl
584       \seq_clear:N \l__bnvs_b_seq
585       \cs_set:Npn \return_true: {
586         \cs_set:Npn \:nnn #####1 #####2 #####3 {
587           \__bnvs_group_end:
588           \tl_set:Nn #2 { #####1 }
589           \tl_set:Nn #3 { #####2 }
590           \seq_set_split:Nnn #4 . { #####3 }
591           \seq_remove_all:Nn #4 { }
592         }
593         \exp_args:NVVx
594         \:nnn #2 #3 {
595           \seq_use:Nn #4 .
596         }
597         \prg_return_true:
598       }
599       \cs_set:Npn \branch:n ##1 {
600         \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
601           \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_b_tl
602           \tl_set:Nn \l__bnvs_a_tl { #3 . }
603           \tl_put_right:Nx \l__bnvs_a_tl { \seq_use:Nn \l__bnvs_a_seq . }
604         } {
605           \cs_set_eq:NN \loop: \return_true:
606         }
607       }
608       \cs_set:Npn \branch:FF ##1 ##2 {
609         \exp_args:Nx
610         \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
611           \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
612             \tl_set_eq:NN #3 \l__bnvs_b_tl
613             \seq_set_eq:NN #4 \l__bnvs_b_seq
614             \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
615           } { ##1 }
616         } { ##2 }
617       }
618       \cs_set:Npn \extract_key:F {
619         \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_b_seq {
620           \tl_set_eq:NN #3 \l__bnvs_b_tl
621           \seq_set_eq:NN #4 \l__bnvs_b_seq
622           \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
623         }
624       }
625       \cs_set:Npn \loop: {
626         \__bnvs_call:TF {

```

```

627     \exp_args:Nx
628     \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {

```

If there is a length, no resolution occurs.

```

629         \branch:n { 1 }
630     } {
631         \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
632             \seq_clear:N \l__bnvs_b_seq
633             \tl_set:Nn \l__bnvs_a_tl { #3 . }
634             \tl_put_right:Nx \l__bnvs_a_tl {
635                 \seq_use:Nn \l__bnvs_a_seq . .
636             }
637             \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
638             \branch:FF {

```

The value for key $\langle \l_a_tl \rangle / L$ is not just a (qualified) name.

```

639     \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
640     } {

```

Unknown key $\langle \l_a_tl \rangle / L$.

```

641     \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
642     }
643     } {
644         \branch:FF {
645             \cs_set_eq:NN \loop: \return_true:
646         } {
647             \cs_set:Npn \loop: {
648                 \__bnvs_group_end:
649                 \prg_return_false:
650             }
651         }
652     }
653     } {
654         \cs_set:Npn \loop: {
655             \__bnvs_group_end:
656             \prg_return_false:
657         }
658     }
659     }
660     \loop:
661     }
662     \loop:
663     } {
664         \prg_return_true:
665     }
666 }
667 \prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
668 #1 #2 #3 #4 { T, F, TF } {
669     #1 {
670         \__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.

```

671     \tl_set_eq:NN \l__bnvs_a_tl #3
672     \seq_set_eq:NN \l__bnvs_a_seq #4

```

```

673 \cs_set:Npn \return_true: {
674   \cs_set:Npn \:nnn #####1 #####2 #####3 {
675     \__bnvs_group_end:
676     \tl_set:Nn #2 { #####1 }
677     \tl_set:Nn #3 { #####2 }
678     \seq_set_split:Nnn #4 . { #####3 }
679     \seq_remove_all:Nn #4 { }
680   }
681   \exp_args:NVVx
682   \:nnn #2 #3 {
683     \seq_use:Nn #4 .
684   }
685   \prg_return_true:
686 }
687 \cs_set:Npn \branch:n ##1 {
688   \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
689     \tl_put_right:Nn \l__bnvs_a_tl { . }
690     \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
691   } {
692     \cs_set_eq:NN \loop: \return_true:
693   }
694 }
695 \cs_set:Npn \loop: {
696   \__bnvs_call:TF {
697     \exp_args:Nx
698     \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
699       \branch:n { 1 }
700     } {
701       \exp_args:Nx
702       \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
703         \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
704           \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
705           \tl_set_eq:NN #3 \l__bnvs_b_tl
706           \seq_set_eq:NN #4 \l__bnvs_a_seq
707         } {
708           \branch:n { 2 }
709         }
710       } {
711         \branch:n { 3 }
712       }
713     }
714   } {
715     \cs_set:Npn \loop: {
716       \__bnvs_group_end:
717       \prg_return_false:
718     }
719   }
720   \loop:
721 }
722 \loop:
723 } {
724   \prg_return_true:
725 }
726 }

```

5.6.7 Evaluation bricks

$\backslash_bnvs_fp_round:nN$ $\backslash_bnvs_fp_round:N$	$\backslash_bnvs_fp_round:nN \{ \langle expression \rangle \} \langle tl \ variable \rangle$ $\backslash_bnvs_fp_round:N \langle tl \ variable \rangle$
---	--

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

```

727 \cs_new:Npn \__bnvs_fp_round:nN #1 #2 {
728   \tl_if_empty:nTF { #1 } {
729     } {
730     \tl_put_right:Nx #2 {
731       \fp_eval:n { round(#1) }
732     }
733   }
734 }
735 \cs_generate_variant:Nn \__bnvs_fp_round:nN { VN, xN }
736 \cs_new:Npn \__bnvs_fp_round:N #1 {
737   \tl_if_empty:VTF #1 {
738     } {
739     \tl_set:Nx #1 {
740       \fp_eval:n { round(#1) }
741     }
742   }
743 }

```

$\backslash_bnvs_raw_first:nNTF$ $\backslash_bnvs_raw_first:(xN VN)TF$	$\backslash_bnvs_raw_first:nNTF \{ \langle name \rangle \} \langle tl \ variable \rangle \{ \langle true \ code \rangle \} \{ \langle false \ code \rangle \}$
---	---

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

```

744 \cs_set:Npn \__bnvs_return_true:nnN #1 #2 #3 {
745   \tl_if_empty:NTF \l__bnvs_ans_tl {
746     \__bnvs_group_end:
747     \__bnvs_gremove:n { #1//#2 }
748     \prg_return_false:
749   } {
750     \__bnvs_fp_round:N \l__bnvs_ans_tl
751     \__bnvs_gput:nV { #1//#2 } \l__bnvs_ans_tl
752     \exp_args:NNNV
753     \__bnvs_group_end:
754     \tl_put_right:Nn #3 \l__bnvs_ans_tl
755     \prg_return_true:
756   }
757 }
758 \cs_set:Npn \__bnvs_return_false:nn #1 #2 {
759   \__bnvs_group_end:
760   \__bnvs_gremove:n { #1//#2 }
761   \prg_return_false:
762 }
763 \prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
764   \__bnvs_if_in:nTF { #1/A } {

```

```

765     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//A } }
766     \prg_return_true:
767   } {
768     \__bnvs_group_begin:
769     \tl_clear:N \l__bnvs_ans_tl
770     \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
771       \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
772         \__bnvs_return_true:nn { #1 } A #2
773       } {
774         \__bnvs_return_false:nn { #1 } A
775       }
776     } {
777       \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
778         \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
779           \__bnvs_if_append:xNTF {
780             \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
781           } \l__bnvs_ans_tl {
782             \__bnvs_return_true:nn { #1 } A #2
783           } {
784             \__bnvs_return_false:nn { #1 } A
785           }
786         } {
787           \__bnvs_return_false:nn { #1 } A
788         }
789       } {
790         \__bnvs_return_false:nn { #1 } A
791       }
792     }
793   }
794 }
795 \prg_generate_conditional_variant:Nnn
796   \__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.

```

797 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
798   \__bnvs_raw_first:nNTF { #1 } #2 {
799     \prg_return_true:
800   } {
801     \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {
802       \bool_set_true:N \l_no_counter_bool
803       \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
804         \__bnvs_return_true:nn { #1 } A #2
805       } {
806         \__bnvs_return_false:nn { #1 } A
807       }
808     } {
809       \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
810         \__bnvs_gput:nn { #1/A } { 1 }
811         \tl_set:Nn #2 { 1 }

```

```

812     \_bnvs_return_true:nn { #1 } A #2
813   } {
814     \_bnvs_return_false:nn { #1 } A
815   }
816 }
817 }
818 }

```

_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.

```

819 \cs_new:Npn \_bnvs_first:nN #1 #2 {
820   \_bnvs_if_first:nNF { #1 } #2 {
821     \msg_error:nnn { beanoves } { :n } { Range-with-no-first:~#1 }
822   }
823 }
824 \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.

```

825 \prg_new_conditional:Npnn \_bnvs_raw_length:nN #1 #2 { T, F, TF } {
826   \_bnvs_if_in:nTF { #1/L } {
827     \tl_put_right:Nx #2 { \_bnvs_item:n { #1/L } }
828     \prg_return_true:
829   } {
830     \_bnvs_gput:nn { #1/L } { 0 }
831     \_bnvs_group_begin:
832     \tl_clear:N \l__bnvs_ans_tl
833     \_bnvs_if_in:nTF { #1/L } {
834       \_bnvs_if_append:xNTF {
835         \_bnvs_item:n { #1/L }
836       } \l__bnvs_ans_tl {
837         \_bnvs_return_true:nnN { #1 } L #2
838       } {
839         \_bnvs_return_false:nn { #1 } L
840       }
841     } {
842       \_bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
843         \_bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
844           \_bnvs_if_append:xNTF {
845             \l__bnvs_b_tl - (\l__bnvs_a_tl) + 1
846           } \l__bnvs_ans_tl {
847             \_bnvs_return_true:nnN { #1 } L #2
848           } {
849             \_bnvs_return_false:nn { #1 } L
850           }
851         } {
852           \_bnvs_return_false:nn { #1 } L
853         }
854       } {

```

```

855     \_bnvs_return_false:nn { #1 } L
856   }
857 }
858 }
859 }
860 \prg_generate_conditional_variant:Nnn
861   \_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.

```

862 \prg_new_conditional:Npnn \_bnvs_raw_last:nN #1 #2 { T, F, TF } {
863   \_bnvs_if_in:nTF { #1//Z } {
864     \tl_put_right:Nx #2 { \_bnvs_item:n { #1//Z } }
865     \prg_return_true:
866   } {
867     \_bnvs_gput:nn { #1//Z } { 0 }
868     \_bnvs_group_begin:
869     \tl_clear:N \l__bnvs_ans_tl
870     \_bnvs_if_in:nTF { #1/Z } {
871       \_bnvs_if_append:xNTF {
872         \_bnvs_item:n { #1/Z }
873       } \l__bnvs_ans_tl {
874         \_bnvs_return_true:nnN { #1 } Z #2
875       } {
876         \_bnvs_return_false:nn { #1 } Z
877       }
878     } {
879       \_bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
880         \_bnvs_get:nNTF { #1/L } \l__bnvs_b_tl {
881           \_bnvs_if_append:xNTF {
882             \l__bnvs_a_tl + (\l__bnvs_b_tl) - 1
883           } \l__bnvs_ans_tl {
884             \_bnvs_return_true:nnN { #1 } Z #2
885           } {
886             \_bnvs_return_false:nn { #1 } Z
887           }
888         } {
889           \_bnvs_return_false:nn { #1 } Z
890         }
891       } {
892         \_bnvs_return_false:nn { #1 } Z
893       }
894     }
895   }
896 }
897 \prg_generate_conditional_variant:Nnn
898   \_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>

```

899 \cs_new:Npn \__bnvs_last:nN #1 #2 {
900   \__bnvs_raw_last:nNF { #1 } #2 {
901     \msg_error:nnn { beanoves } { :n } { Range-with-no-last:~#1 }
902   }
903 }
904 \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.

```

905 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
906   \__bnvs_if_in:nTF { #1//N } {
907     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//N } }
908     \prg_return_true:
909   } {
910     \__bnvs_group_begin:
911     \cs_set:Npn \__bnvs_return_true: {
912       \tl_if_empty:NTF \l__bnvs_ans_tl {
913         \__bnvs_group_end:
914         \prg_return_false:
915       } {
916         \__bnvs_fp_round:N \l__bnvs_ans_tl
917         \__bnvs_gput:nV { #1//N } \l__bnvs_ans_tl
918         \exp_args:NNNV
919         \__bnvs_group_end:
920         \tl_put_right:Nn #2 \l__bnvs_ans_tl
921         \prg_return_true:
922       }
923     }
924     \cs_set:Npn \return_false: {
925       \__bnvs_group_end:
926       \prg_return_false:
927     }
928     \tl_clear:N \l__bnvs_a_tl
929     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_a_tl {
930       \__bnvs_if_append:xNTF {
931         \l__bnvs_a_tl + 1
932       } \l__bnvs_ans_tl {
933         \__bnvs_return_true:
934       } {
935         \return_false:
936       }
937     } {
938       \return_false:
939     }
940   }
941 }
942 \prg_generate_conditional_variant:Nnn
943   \__bnvs_if_next:nN { VN } { T, F, TF }

```

__bnvs_next:nN __bnvs_next:nN {<name>} <tl variable>
__bnvs_next:VN Append the index after the <name> slide range to the <tl variable>.


```

944 \cs_new:Npn \__bnvs_next:nN #1 #2 {
945   \__bnvs_if_next:nNF { #1 } #2 {
946     \msg_error:nnn { beanoves } { :n } { Range-with-no-next::~#1 }
947   }
948 }
949 \cs_generate_variant:Nn \__bnvs_next:nN { VN }

```

$\backslash_bnvs_if_index:nnNTF$ $\backslash_bnvs_if_index:VVNTF$ $\backslash_bnvs_if_index:nnnNTF$	$\backslash_bnvs_if_index:nnNTF \{ \langle name \rangle \} \{ \langle integer \rangle \} \langle tl\ variable \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
--	--

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

```

950 \prg_new_conditional:Npnn \__bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
951   \__bnvs_group_begin:
952   \tl_clear:N \l__bnvs_ans_tl
953   \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
954     \tl_put_right:Nn \l__bnvs_ans_tl { + (#2) - 1 }
955     \exp_args:NNV
956     \__bnvs_group_end:
957     \__bnvs_fp_round:nN \l__bnvs_ans_tl #3
958     \prg_return_true:
959   } {
960     \prg_return_false:
961   }
962 }
963 \prg_generate_conditional_variant:Nnn
964   \__bnvs_if_index:nnN { VVN } { T, F, TF }

```

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

Append the range of the $\langle name \rangle$ slide range to the $\langle tl\ variable \rangle$. Execute $\langle true\ code \rangle$ when there is a $\langle range \rangle$, $\langle false\ code \rangle$ otherwise.

```

965 \prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {
966   \bool_if:NTF \l__bnvs_no_range_bool {
967     \prg_return_false:
968   } {
969     \__bnvs_if_in:nTF { #1/ } {
970       \tl_put_right:Nn { 0-0 }
971     } {
972       \__bnvs_group_begin:
973       \tl_clear:N \l__bnvs_a_tl
974       \tl_clear:N \l__bnvs_b_tl
975       \tl_clear:N \l__bnvs_ans_tl
976       \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
977         \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
978           \exp_args:NNNx
979           \__bnvs_group_end:
980           \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }

```

```

981         \prg_return_true:
982     } {
983         \exp_args:NNNx
984         \__bnvs_group_end:
985         \tl_put_right:Nn #2 { \l__bnvs_a_tl - }
986         \prg_return_true:
987     }
988 } {
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_b_tl }
993         \prg_return_true:
994     } {
995         \__bnvs_group_end:
996         \prg_return_false:
997     }
998 }
999 }
1000 }
1001 }
1002 \prg_generate_conditional_variant:Nnn
1003   \__bnvs_if_range:nN { VN } { T, F, TF }

```

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

`__bnvs_range:VN` Append the range of the <name> slide range to the <tl variable>.

```

1004 \cs_new:Npn \__bnvs_range:nN #1 #2 {
1005     \__bnvs_if_range:nNF { #1 } #2 {
1006         \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1007     }
1008 }
1009 \cs_generate_variant:Nn \__bnvs_range:nN { VN }

```

`__bnvs_if_free_counter:nNTF` `__bnvs_if_free_counter:nNTF {<name>} <tl variable> {(true code)} {(false code)}`
`__bnvs_if_free_counter:VNTF` `code}`

Set the <tl variable> to the value of the counter associated to the {<name>} slide range.

```

1010 \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
1011     \__bnvs_group_begin:
1012     \tl_clear:N \l__bnvs_ans_tl
1013     \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
1014         \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
1015             \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
1016         }
1017     }
1018     \tl_if_empty:NTF \l__bnvs_ans_tl {
1019         \__bnvs_group_end:
1020         \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1021             \__bnvs_gput:nn { #1/C } { 1 }
1022             \tl_set:Nn #2 { 1 }

```

```

1023     \prg_return_true:
1024 } {
1025     \prg_return_false:
1026 }
1027 } {
1028     \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl
1029     \exp_args:NNNV
1030     \__bnvs_group_end:
1031     \tl_set:Nn #2 \l__bnvs_ans_tl
1032     \prg_return_true:
1033 }
1034 }
1035 \prg_generate_conditional_variant:Nnn
1036     \__bnvs_if_free_counter:nN { VN } { T, F, TF }

```

__bnvs_if_counter:nNTF
__bnvs_if_counter:VNTF

$\langle name \rangle$ $\langle tl\ variable \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$

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.

```

1037 \prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {
1038     \__bnvs_group_begin:
1039     \__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.

```

1040     \tl_clear:N \l__bnvs_a_tl
1041     \__bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
1042         \fp_compare:nNt { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
1043             \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1044         }
1045     }

```

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

```

1046     \tl_clear:N \l__bnvs_a_tl
1047     \__bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
1048         \fp_compare:nNt { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
1049             \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1050         }
1051     }
1052     \exp_args:NNV
1053     \__bnvs_group_end:
1054     \__bnvs_fp_round:nN \l__bnvs_ans_tl #2
1055     \prg_return_true:
1056 } {
1057     \prg_return_false:
1058 }
1059 }
1060 \prg_generate_conditional_variant:Nnn
1061     \__bnvs_if_counter:nN { VN } { T, F, TF }

```

__bnvs_if_incr:nnTF
__bnvs_if_incr:nnNTF
__bnvs_if_incr:(VnN|VVN)TF

$\langle name \rangle$ $\langle offset \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$
 $\langle name \rangle$ $\langle offset \rangle$ $\langle tl\ variable \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$

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.

```

1062 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {
1063   \__bnvs_group_begin:
1064   \tl_clear:N \l__bnvs_a_tl
1065   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_a_tl {
1066     \tl_clear:N \l__bnvs_b_tl
1067     \__bnvs_if_append:xNTF { \l__bnvs_a_tl + (#2) } \l__bnvs_b_tl {
1068       \__bnvs_fp_round:N \l__bnvs_b_tl
1069       \__bnvs_gput:nV { #1/C } \l__bnvs_b_tl
1070       \__bnvs_group_end:
1071       \prg_return_true:
1072     } {
1073       \__bnvs_group_end:
1074       \prg_return_false:
1075     }
1076   } {
1077     \__bnvs_group_end:
1078     \prg_return_false:
1079   }
1080 }
1081 \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1082   \__bnvs_if_incr:nnTF { #1 } { #2 } {
1083     \__bnvs_if_counter:nNTF { #1 } #3 {
1084       \prg_return_true:
1085     } {
1086       \prg_return_false:
1087     }
1088   } {
1089     \prg_return_false:
1090   }
1091 }
1092 \prg_generate_conditional_variant:Nnn
1093   \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }

```

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

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.

```

1094 \prg_new_conditional:Npnn \__bnvs_if_post:nnN #1 #2 #3 { T, F, TF } {
1095   \__bnvs_if_counter:nNTF { #1 } #3 {
1096     \__bnvs_if_incr:nnTF { #1 } { #2 } {
1097       \prg_return_true:
1098     } {
1099       \prg_return_false:
1100     }
1101   } {
1102     \prg_return_false:
1103   }
1104 }
1105 \prg_generate_conditional_variant:Nnn
1106   \__bnvs_if_post:nnN { VnN, VVN } { T, F, TF }

```

5.6.8 Evaluation

<u>__bnvs_if_append:nNTF</u> <u>__bnvs_if_append:(VN xN)TF</u>	<p>__bnvs_if_append:nNTF {$\langle integer\ expression \rangle$} $\langle tl\ variable \rangle$ {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</p> <p>Evaluates the $\langle integer\ expression \rangle$, replacing all the named specifications by their static counterpart then put the result to the right of the $\langle tl\ variable \rangle$. Executed within a group. Heavily used by __bnvs_eval_query:nN, where $\langle integer\ expression \rangle$ was initially enclosed in ‘?(...)’. Local variables:</p>
\l__bnvs_ans_tl	<p>To feed $\langle tl\ variable \rangle$ with.</p> <p>(End definition for \l__bnvs_ans_tl.)</p>
\l__bnvs_split_seq	<p>The sequence of caught query groups and non queries.</p> <p>(End definition for \l__bnvs_split_seq.)</p>
\l__bnvs_split_int	<p>Is the index of the non queries, before all the caught groups.</p> <p>(End definition for \l__bnvs_split_int.)</p>
\l__bnvs_name_tl	<p>Storage for \l_split_seq items that represent names.</p> <p>(End definition for \l__bnvs_name_tl.)</p>
\l__bnvs_path_tl	<p>Storage for \l_split_seq items that represent integer paths.</p> <p>(End definition for \l__bnvs_path_tl.)</p> <p>Catch circular definitions.</p>
1108	\prg_new_conditional:Npnn __bnvs_if_append:nN #1 #2 { T, F, TF } {
1109	__bnvs_call:TF {
1110	__bnvs_group_begin:
	Local variables:
1111	\int_zero:N \l__bnvs_split_int
1112	\seq_clear:N \l__bnvs_split_seq
1113	\tl_clear:N \l__bnvs_id_tl
1114	\tl_clear:N \l__bnvs_name_tl
1115	\tl_clear:N \l__bnvs_path_tl
1116	\tl_clear:N \l__bnvs_group_tl
1117	\tl_clear:N \l__bnvs_ans_tl
1118	\tl_clear:N \l__bnvs_a_tl
	Implementation:
1119	\regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq
1120	\int_set:Nn \l__bnvs_split_int { 1 }
1121	\tl_set:Nx \l__bnvs_ans_tl {
1122	\seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1123	}

`\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.

```

1124 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1125   \tl_set:Nx ##2 {
1126     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1127   }
1128   \tl_if_empty:NTF ##2 {
1129     ##4 } {
1130     ##3
1131   }
1132 }

```

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

```

1133 \cs_set:Npn \return_true: {
1134   \fp_round:
1135   \exp_args:NNNV
1136   \__bnvs_group_end:
1137   \tl_put_right:Nn #2 \l__bnvs_ans_tl
1138   \prg_return_true:
1139 }
1140 \cs_set:Npn \fp_round: {
1141   \__bnvs_fp_round:N \l__bnvs_ans_tl
1142 }
1143 \cs_set:Npn \return_false: {
1144   \__bnvs_group_end:
1145   \prg_return_false:
1146 }
1147 \cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {
1148   \switch:nNTF { ##2 } \l__bnvs_id_tl { } {
1149     \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_current_tl
1150     \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1151   }
1152   \switch:nNTF { ##3 } \l__bnvs_path_tl {
1153     \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
1154     \seq_remove_all:Nn \l__bnvs_path_seq { }
1155   } {
1156     \seq_clear:N \l__bnvs_path_seq
1157   }
1158   ##1 \l__bnvs_id_tl \l__bnvs_name_tl \l__bnvs_path_seq {
1159     \cs_set:Npn \: {
1160       ##4
1161     }
1162   } {
1163     \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
1164   }
1165   \:
1166 }
1167 \cs_set:Npn \:T ##1 {
1168   \seq_if_empty:NTF \l__bnvs_path_seq { ##1 } {
1169     \cs_set_eq:NN \loop: \return_false:
1170   }
1171 }

```

Main loop.

```

1172     \cs_set:Npn \loop: {
1173       \int_compare:nNnTF {
1174         \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1175       } {
1176         \switch:nNTF 1 \l__bnvs_name_tl {

• Case ++⟨name⟩⟨integer path⟩.n.

1177         \:NnnT \__bnvs_resolve_n:NNNTF 2 3 {
1178           \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1179             \cs_set_eq:NN \loop: \return_false:
1180           }
1181         }
1182       } {
1183         \switch:nNTF 4 \l__bnvs_name_tl {

• Cases ⟨name⟩⟨integer path⟩....

1184         \switch:nNTF 7 \l__bnvs_a_tl {
1185           \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1186             \:T {
1187               \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1188                 \cs_set_eq:NN \loop: \return_false:
1189               }
1190             }
1191           }

• Case ...length.

1192       } {
1193         \switch:nNTF 8 \l__bnvs_a_tl {

• Case ...last.

1194         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1195           \:T {
1196             \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1197               \cs_set_eq:NN \loop: \return_false:
1198             }
1199           }
1200         }

1201       } {
1202         \switch:nNTF 9 \l__bnvs_a_tl {

• Case ...next.

1203         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1204           \:T {
1205             \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1206               \cs_set_eq:NN \loop: \return_false:
1207             }
1208           }
1209         }
1210       } {
1211         \switch:nNTF { 10 } \l__bnvs_a_tl {

```

- Case ...range.

```

1212 \:NnnT \__bnvs_resolve:NNTF 5 6 {
1213   \:T {
1214     \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
1215       \cs_set_eq:NN \fp_round: \prg_do_nothing:
1216     } {
1217       \cs_set_eq:NN \loop: \return_false:
1218     }
1219   }
1220 }
1221   } {

```

- Case ...n.

```

1222   \switch:nNTF { 12 } \l__bnvs_a_tl {

```

- Case ...+= $\langle integer \rangle$.

```

1223 \:NnnT \__bnvs_resolve_n:NNTF 5 6 {
1224   \:T {
1225     \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1226       \cs_set_eq:NN \loop: \return_false:
1227     }
1228   }
1229 }
1230   } {

```

- Case ...n++.

```

1231   \switch:nNTF { 13 } \l__bnvs_a_tl {
1232     \:NnnT \__bnvs_resolve_n:NNTF 5 6 {
1233       \seq_if_empty:NTF \l__bnvs_path_seq {
1234         \__bnvs_if_post:VnNF \l__bnvs_name_tl { 1 } \l__bnvs_ans_tl {
1235           \cs_set_eq:NN \loop: \return_false:
1236         }
1237       } {
1238         \msg_error:nnx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
1239         \cs_set_eq:NN \loop: \return_false:
1240       }
1241     }
1242   } {
1243     \switch:nNTF { 11 } \l__bnvs_a_tl {

```

- Case ...n++.

```

1244     \:NnnT \__bnvs_resolve_n:NNTF 5 6 {
1245       \seq_if_empty:NTF \l__bnvs_path_seq {
1246         \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1247           \cs_set_eq:NN \loop: \return_false:
1248         }
1249       } {
1250         \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1251         \seq_if_empty:NTF \l__bnvs_path_seq {
1252           \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1253             \cs_set_eq:NN \loop: \return_false:

```



```

1254 }
1255 } {
1256   \msg_error:nnx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1257   \cs_set_eq:NN \loop: \return_false:
1258 }
1259   }
1260   }
1261   } {
1262     \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1263       \seq_if_empty:NTF \l__bnvs_path_seq {
1264         \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1265           \cs_set_eq:NN \loop: \return_false:
1266         }
1267       } {
1268         \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1269         \seq_if_empty:NTF \l__bnvs_path_seq {
1270           \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1271             \cs_set_eq:NN \loop: \return_false:
1272           }
1273         } {
1274           \msg_error:nnx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1275           \cs_set_eq:NN \loop: \return_false:
1276         }
1277       }
1278     }
1279   }
1280 }
1281 }
1282 }
1283 }
1284 }
1285 }
1286 } {

```

No name.

```

1287   }
1288 }
1289   \int_add:Nn \l__bnvs_split_int { 14 }
1290   \tl_put_right:Nx \l__bnvs_ans_tl {
1291     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1292   }
1293   \loop:
1294 } {
1295   \return_true:
1296 }
1297 }
1298   \loop:
1299 } {
1300   \msg_error:nnx { beanoves } { :n } { Too-many~calls:~ #1 }
1301   \prg_return_false:
1302 }

```

```

1303 }
1304 \prg_generate_conditional_variant:Nnn
1305   \__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.)

```

1306 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1307   \A \s* (?

```

- 2: <first>

```

1308     ( [^:]* ) \s* :

```

- 3: second optional colon

```

1309     (:)? \s*

```

- 4: <length>

```

1310     ( [^:]* )

```

- 5: standalone <first>

```

1311     | ( [^:]+ )

```

```

1312   ) \s* \Z

```

```

1313 }

```

```

1314 \prg_new_conditional:Npnn \__bnvs_if_eval_query:nN #1 #2 { T, F, TF } {

```

```

1315   \__bnvs_call_greset:

```

```

1316   \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {

```

```

1317     #1

```

```

1318   } \l__bnvs_match_seq {

```

```

1319     \bool_set_false:N \l__bnvs_no_counter_bool

```

```

1320     \bool_set_false:N \l__bnvs_no_range_bool

```

```
\switch:nNTF {<capture group number>} <tl variable> {(black code)} {(white code)}
```

Helper function to locally set the *<tl variable>* to the captured group *<capture group number>* and branch depending on the emptiness of this variable.

```
1321 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1322   \tl_set:Nx ##2 {
1323     \seq_item:Nn \l__bnvs_match_seq { ##1 }
1324   }
1325   \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1326 }
1327 \switch:nNTF 5 \l__bnvs_a_tl {
```

☛ Single expression

```
1328   \bool_set_false:N \l__bnvs_no_range_bool
1329   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1330     \prg_return_true:
1331   } {
1332     \prg_return_false:
1333   }
1334 } {
1335   \switch:nNTF 2 \l__bnvs_a_tl {
1336     \switch:nNTF 4 \l__bnvs_b_tl {
1337       \switch:nNTF 3 \l__bnvs_c_tl {
```

☛ *<first>::<last>* range

```
1338   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1339     \tl_put_right:Nn #2 { - }
1340     \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1341       \prg_return_true:
1342     } {
1343       \prg_return_false:
1344     }
1345   } {
1346     \prg_return_false:
1347   }
1348 } {
```

☛ *<first>:<length>* range

```
1349   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1350     \tl_put_right:Nx #2 { - }
1351     \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1 }
1352     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1353       \prg_return_true:
1354     } {
1355       \prg_return_false:
1356     }
1357   } {
1358     \prg_return_false:
1359   }
1360 }
1361 } {
```

☛ *<first>*: and *<first>::* range

```
1362   \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1363     \tl_put_right:Nn #2 { - }
```

```

1364         \prg_return_true:
1365     } {
1366         \prg_return_false:
1367     }
1368 }
1369 } {
1370     \switch:nNTF 4 \l__bnvs_b_tl {
1371         \switch:nNTF 3 \l__bnvs_c_tl {
1372             ::⟨last⟩ range
1373             \tl_put_right:Nn #2 { - }
1374             \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1375                 \prg_return_true:
1376             } {
1377                 \prg_return_false:
1378             }
1379 \msg_error:nnx { beanoves } { :n } { Syntax-error(Missing-first):~#1 }
1380         }
1381     } {
1382         :: or :: range
1383         \seq_put_right:Nn #2 { - }
1384     }
1385 }
1386 } {
Error
1387     \msg_error:nnn { beanoves } { :n } { Syntax-error:~#1 }
1388 }
1389 }

```

__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.

```
1390 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }
```

(End definition for \c__bnvs_comma_regex.)

No other variable is used.

```
1391 \cs_new:Npn \__bnvs_eval:nN #1 #2 {
```

```
1392 \__bnvs_group_begin:
```

Local variables declaration

```
1393 \seq_clear:N \l__bnvs_query_seq
```

```
1394 \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 $\langle first \rangle :: \langle last \rangle$ range expressions as well. We first split the expression around commas, into $\backslash l_query_seq$.

```
1395 \regex_split:NnN \c__bnvs_comma_regex { #1 } \l__bnvs_query_seq
```

Then each component is evaluated and the result is stored in $\backslash l_bnvs_ans_seq$ that we have clear before use.

```
1396 \seq_map_inline:Nn \l__bnvs_query_seq {
1397   \tl_clear:N \l__bnvs_ans_tl
1398   \__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {
1399     \seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl
1400   } {
1401     \seq_map_break:n {
1402       \msg_fatal:nnn { beanoves } { :n } { Circular~dependency~in~#1}
1403     }
1404   }
1405 }
```

We have managed all the comma separated components, we collect them back and append them to $\langle tl\ variable \rangle$.

```
1406 \exp_args:NNNx
```

```
1407 \__bnvs_group_end:
```

```
1408 \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
```

```
1409 }
```

```
1410 \cs_generate_variant:Nn \__bnvs_eval:nN { VN, xN }
```

```
\BeanovesEval \BeanovesEval [⟨tl variable⟩] {⟨overlay queries⟩}
```

$\langle overlay\ queries \rangle$ is the argument of $?(\dots)$ instructions. This is a comma separated list of single $\langle overlay\ query \rangle$'s.

This function evaluates the $\langle overlay\ queries \rangle$ and store the result in the $\langle tl\ variable \rangle$ when provided or leave the result in the input stream. Forwards to $\backslash _bnvs_eval:nN$ within a group. $\backslash l_ans_tl$ is used locally to store the result.

```
1411 \NewDocumentCommand \BeanovesEval { s o m } {
1412   \__bnvs_group_begin:
1413   \tl_clear:N \l__bnvs_ans_tl
1414   \IfBooleanTF { #1 } {
1415     \bool_set_true:N \l__bnvs_no_counter_bool
1416   } {
1417     \bool_set_false:N \l__bnvs_no_counter_bool
1418   }
1419   \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
1420   \IfValueTF { #2 } {
1421     \exp_args:NNNV
1422     \__bnvs_group_end:
1423     \tl_set:Nn #2 \l__bnvs_ans_tl
1424   } {
```

```

1425 \exp_args:NV
1426 \__bnvs_group_end: \l__bnvs_ans_tl
1427 }
1428 }

```

5.6.9 Reseting slide ranges

\BeanovesReset [*⟨first value⟩*] {*⟨Slide range name⟩*}

```

1429 \NewDocumentCommand \BeanovesReset { 0{1} m } {
1430   \__bnvs_reset:nn { #1 } { #2 }
1431   \ignorespaces
1432 }

```

Forwards to __bnvs_reset:nn.

__bnvs_reset:nn [*⟨first value⟩*] {*⟨slide range name⟩*}

Reset the counter to the given *⟨first value⟩*. Clean the cached values also.

```

1433 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1434   \bool_if:nTF {
1435     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1436   } {
1437     \__bnvs_gremove:n { #2/C }
1438     \__bnvs_gremove:n { #2//A }
1439     \__bnvs_gremove:n { #2//L }
1440     \__bnvs_gremove:n { #2//Z }
1441     \__bnvs_gremove:n { #2//N }
1442     \__bnvs_gput:nn { #2/C0 } { #1 }
1443   } {
1444     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1445   }
1446 }

1447 \makeatother
1448 \ExplSyntaxOff
1449 </package>

```