

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-debug}
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 \insertslidenumber}
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 <*<debug>
5   {beanoves}
6 </!debug>
7 <*<gubed>
8   {beanoves-debug}
9 </!gubed>
10 {2022/10/28}
11 {1.0}
12 {Named overlay specifications for beamer}
```

5.2 logging and debugging facilities

Utility message.

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

5.3 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.

```
15 \int_new:N \l__bnvs_depth_int
16 \bool_new:N \l__bnvs_ask_bool
17 \bool_new:N \l__bnvs_query_bool
18 \bool_new:N \l__bnvs_no_counter_bool
19 \bool_new:N \l__bnvs_no_range_bool
20 \bool_new:N \l__bnvs_continue_bool
21 \bool_new:N \l__bnvs_in_frame_bool
22 \bool_set_false:N \l__bnvs_in_frame_bool
23 \tl_new:N \l__bnvs_id_current_tl
24 \tl_new:N \l__bnvs_a_tl
25 \tl_new:N \l__bnvs_b_tl
26 \tl_new:N \l__bnvs_c_tl
```

```

27 \tl_new:N \l__bnvs_id_tl
28 \tl_new:N \l__bnvs_ans_tl
29 \tl_new:N \l__bnvs_name_tl
30 \tl_new:N \l__bnvs_path_tl
31 \tl_new:N \l__bnvs_group_tl
32 \tl_new:N \l__bnvs_query_tl
33 \tl_new:N \l__bnvs_token_tl
34 \seq_new:N \l__bnvs_a_seq
35 \seq_new:N \l__bnvs_b_seq
36 \seq_new:N \l__bnvs_ans_seq
37 \seq_new:N \l__bnvs_match_seq
38 \seq_new:N \l__bnvs_split_seq
39 \seq_new:N \l__bnvs_path_seq
40 \seq_new:N \l__bnvs_query_seq
41 \seq_new:N \l__bnvs_token_seq

```

5.4 Infinite loop management

Unending recursivity is managed here.

`\g__bnvs_call_int`

```

42 \int_zero_new:N \g__bnvs_call_int
43 \int_const:Nn \c__bnvs_max_call_int { 2048 }

```

(End definition for `\g__bnvs_call_int`.)

`__bnvs_call_reset:`

`__bnvs_call_reset:`

Reset the call stack counter.

```

44 \cs_set:Npn \__bnvs_call_reset: {
45   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
46 }

```

`__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.

```

47 \prg_new_conditional:Npnn \__bnvs_call: { T, F, TF } {
48   \int_gdecr:N \g__bnvs_call_int
49   \int_compare:nNnTF \g__bnvs_call_int > 0 {
50     \prg_return_true:
51   } {
52     \prg_return_false:
53   }
54 }

```

5.5 Overlay specification

5.5.1 In slide range definitions

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

`<id>!<name>/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.

55 `\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.

```

56 \cs_new:Npn \__bnvs_gput:nn #1 #2 {
57   \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
58 }
59 \cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
60   \prop_if_in:NnF \g__bnvs_prop { #1 } {
61     \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
62   }
63 }
64 \cs_new:Npn \__bnvs_item:n {
65   \prop_item:Nn \g__bnvs_prop
66 }
67 \cs_new:Npn \__bnvs_get:nN {
68   \prop_get:NnN \g__bnvs_prop
69 }
70 \cs_new:Npn \__bnvs_gremove:n {
71   \prop_gremove:Nn \g__bnvs_prop
72 }
73 \cs_new:Npn \__bnvs_gclear:n #1 {
74   \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
75     \__bnvs_gremove:n { #1 / ##1 }
76   }
77 }
78 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
79   \clist_map_inline:nn { /A, /L, /Z, /N } {
80     \__bnvs_gremove:n { #1 / ##1 }
81   }
82 }
83 \cs_new:Npn \__bnvs_gclear: {
84   \prop_gclear:N \g__bnvs_prop
85 }
86 \cs_generate_variant:Nn \__bnvs_gput:nn { nV }
87 \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.

```

88 \prg_new_conditional:Npnn \__bnvs_if_in:n #1 { p, T, F, TF } {
89   \prop_if_in:NnTF \g__bnvs_prop { #1 } {
90     \prg_return_true:

```



```

91   } {
92     \prg_return_false:
93   }
94 }
95 \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.

```

96 \prg_new_conditional:Npnn \__bnvs_get:nN #1 #2 { T, F, TF } {
97   \prop_get:NnNTF \g__bnvs_prop { #1 } #2 {
98     \prg_return_true:
99   } {
100     \prg_return_false:
101   }
102 }

```

5.5.2 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.

```

103 \regex_const:Nn \c__bnvs_name_regex {
104   [[:alpha:]]_ [[:alnum:]]_*
105 }

```

(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.

```

106 \regex_const:Nn \c__bnvs_id_regex {
107   (? \ur{c__bnvs_name_regex} | [?]* ) ? !
108 }

```

(End definition for `\c__bnvs_id_regex`.)

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

```

109 \regex_const:Nn \c__bnvs_path_regex {
110   (? \. [+-]? \d+ )*
111 }

```

(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.

`\c__bnvs_A_key_Z_regex`

```

112 \regex_const:Nn \c__bnvs_key_regex {
113   \ur{c__bnvs_id_regex} ?
114   \ur{c__bnvs_name_regex}

```

```

115     \ur{c__bnvs_path_regex}
116 }
117 \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

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

    5: the path, if any.

119     ( \ur{c__bnvs_path_regex} ) \Z
120 }
121

```

(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.

```

122 \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.

```

123 \regex_const:Nn \c__bnvs_list_regex {
124     \A \[ \s*

    Capture groups:

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

125     ( [^\] %[---
126     ]*? )
127     \s* \] \Z
128 }

```

(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.

```

129 \regex_const:Nn \c__bnvs_split_regex {
130     \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

```

131     \+ \+ ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_name_regex} )

```

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

- 3: optionally followed by an integer path

132 (\ur{c__bnvs_path_regex}) (?: \. n)?

We continue with other expressions

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

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

- 6: optionally followed by an integer path

134 (\ur{c__bnvs_path_regex})

Next comes another branching

135 (?:

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

136 \.(e)ngth

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

137 | \.(a)st

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

138 | \.(n)ext

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

139 | \.(r)ange

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

140 | \.(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.

141 (?: \s* \+= \s* (\S+))?

142)?

143) \s*

144 }

(End definition for `\c__bnvs_split_regex`.)

5.5.3 beamer.cls interface

Work in progress.

```

145 \RequirePackage{keyval}
146 \define@key{beamerframe}{beanoves~id}[] {
147   \tl_set:Nx \l__bnvs_id_current_tl { #1 ! }
148 }
149 \AddToHook{env/beamer@frameslide/before}{
150   \bool_set_true:N \l__bnvs_in_frame_bool
151 }
152 \AddToHook{env/beamer@frameslide/after}{
153   \bool_set_false:N \l__bnvs_in_frame_bool
154 }
155 \AddToHook{cmd/frame/before}{
156   \tl_set:Nn \l__bnvs_id_current_tl { ?! }
157 }

```

5.5.4 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`.

```

158 \cs_new:Npn \__bnvs_range:Nnnnn #1 #2 #3 #4 #5 {
159   \tl_if_empty:nTF { #3 } {
160     \tl_if_empty:nTF { #4 } {
161       \tl_if_empty:nTF { #5 } {
162         \msg_error:nnn { beanoves } { :n } { Not~a~range::~~#2 }
163       } {
164         #1 { #2/Z } { #5 }
165       }
166     } {
167       #1 { #2/L } { #4 }
168       \tl_if_empty:nF { #5 } {
169         #1 { #2/Z } { #5 }
170         #1 { #2/A } { #2.last - (#2.length) + 1 }

```

```

171     }
172   }
173   {
174     #1 { #2/A } { #3 }
175     \tl_if_empty:nTF { #4 } {
176       \tl_if_empty:nF { #5 } {
177         #1 { #2/Z } { #5 }
178         #1 { #2/L } { #2.last - (#2.1) + 1 }
179       }
180     } {
181       #1 { #2/L } { #4 }
182       #1 { #2/Z } { #2.1 + #2.length - 1 }
183     }
184   }
185 }
186 \cs_new:Npn \__bnvs_range:nnnn #1 {
187   \__bnvs_gclear:n { #1 }
188   \__bnvs_range:Nnnnn \__bnvs_gput:nn { #1 }
189 }
190 \cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
191 \cs_new:Npn \__bnvs_range_alt:nnnn #1 {
192   \__bnvs_gclear_cache:n { #1 }
193   \__bnvs_range:Nnnnn \__bnvs_gprovide:nn { #1 }
194 }
195 \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.

```

196 \cs_new:Npn \__bnvs_parse:Nn #1 #2 {
197   \__bnvs_group_begin:
198   \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
199     \exp_args:Nx \__bnvs_gput:nn { \l__bnvs_name_tl/ } { }
200     \exp_args:NNNV
201     \__bnvs_group_end:
202     \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
203   } {
204     \msg_error:nnn { beanoves } { :n } { Unexpected~key:~#2 }
205     \__bnvs_group_end:
206   }
207 }

```

__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.

```

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

```

This is not a list.

```

210   \tl_clear:N \l__bnvs_a_tl
211   \tl_clear:N \l__bnvs_b_tl

```

```

212 \tl_clear:N \l__bnvs_c_tl
213 \regex_split:NnN \c__bnvs_colons_regex { #3 } \l__bnvs_split_seq
214 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_a_tl {
\l_a_tl may contain the ⟨start⟩.
215 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_b_tl {
216 \tl_if_empty:NTF \l__bnvs_b_tl {
This is a one colon range.
217 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
\l_b_tl may contain the ⟨length⟩.
218 \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_c_tl {
219 \tl_if_empty:NTF \l__bnvs_c_tl {
A :: was expected:
220 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(1):~#3 }
221 } {
222 \int_compare:nNnT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
223 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(2):~#3 }
224 }
225 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
\l_c_tl may contain the ⟨end⟩.
226 \seq_if_empty:NF \l__bnvs_split_seq {
227 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(3):~#3 }
228 }
229 }
230 }
231 } {
This is a two colon range.
232 \int_compare:nNnT { \tl_count:N \l__bnvs_b_tl } > { 1 } {
233 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(4):~#3 }
234 }
235 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
\l_c_tl contains the ⟨end⟩.
236 \seq_pop_left:NNTF \l__bnvs_split_seq \l__bnvs_b_tl {
237 \tl_if_empty:NTF \l__bnvs_b_tl {
238 \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
\l_b_tl may contain the ⟨length⟩.
239 \seq_if_empty:NF \l__bnvs_split_seq {
240 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(5):~#3 }
241 }
242 } {
243 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(6):~#3 }
244 }
245 } {
246 \tl_clear:N \l__bnvs_b_tl
247 }
248 }
249 }
250 }

```

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.

```

251 \bool_if:nF {
252   \tl_if_empty_p:N \l__bnvs_a_tl
253   || \tl_if_empty_p:N \l__bnvs_b_tl
254   || \tl_if_empty_p:N \l__bnvs_c_tl
255 } {
256 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(7):~#3 }
257 }
258 #1 { #2 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
259 }
260 \cs_generate_variant:Nn \__bnvs_do_parse:Nnn { Nxn, Non }

```

$__bnvs_id_name_set:nNNTF$ $__bnvs_id_name_set:nNNTF \{ \langle key \rangle \} \langle id \text{ tl var} \rangle \langle full \text{ name tl var} \rangle \{ \langle true \text{ code} \rangle \} \{ \langle false \text{ code} \rangle \}$

If the $\langle key \rangle$ is a key, put the name it defines into the $\langle name \text{ tl var} \rangle$ with the current frame id prefix $\l__bnvs_id_tl$ if none was given, then execute $\langle true \text{ code} \rangle$. Otherwise execute $\langle false \text{ code} \rangle$.

```

261 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNNTF #1 #2 #3 { T, F, TF } {
262   \__bnvs_group_begin:
263   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
264     #1
265   } \l__bnvs_match_seq {
266     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
267     \tl_if_empty:NNTF #2 {
268       \exp_args:NNNx
269       \__bnvs_group_end:
270       \tl_set:Nn #3 { \l__bnvs_id_current_tl #1 }
271       \tl_set_eq:NN #2 \l__bnvs_id_current_tl
272     } {
273       \cs_set:Npn \:n ##1 {
274         \__bnvs_group_end:
275         \tl_set:Nn #2 { ##1 }
276         \tl_set:Nn \l__bnvs_id_current_tl { ##1 }
277       }
278       \exp_args:NV
279       \:n #2
280       \tl_set:Nn #3 { #1 }
281     }
282   \prg_return_true:
283   } {
284     \__bnvs_group_end:
285     \prg_return_false:
286   }
287 }

288 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
289   \__bnvs_group_begin:
290   \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {

```

```

291 \regex_extract_once:NnNTF \c__bnvs_list_regex {
292   #3
293 } \l__bnvs_match_seq {
This is a comma separated list, extract each item and go recursive.
294 \exp_args:NNx
295 \seq_set_from_clist:Nn \l__bnvs_match_seq {
296   \seq_item:Nn \l__bnvs_match_seq { 2 }
297 }
298 \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
299   \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl.##1 } { ##2 }
300 }
301 } {
302   \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
303 }
304 } {
305   \msg_error:nnn { beanoves } { :n } { Invalid-key:~#2 }
306 }
We export \l__bnvs_id_tl:
307 \exp_args:NNNV
308 \__bnvs_group_end:
309 \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
310 }

```

\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.

```

311 \NewDocumentCommand \Beanoves { sm } {
312   \tl_if_eq:NnT \@currenvir { document } {
313     \__bnvs_gclear:
314   }
315   \IfBooleanTF {#1} {
316     \keyval_parse:nnn {
317       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
318     } {
319       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
320     }
321   } {
322     \keyval_parse:nnn {
323       \__bnvs_parse:Nn \__bnvs_range:nVVV
324     } {
325       \__bnvs_parse:Nnn \__bnvs_range:nVVV
326     }
327   }
328   { #2 }
329   \ignorespaces
330 }

```

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

```

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

```


5.5.5 Scanning named overlay specifications

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

<code>\beamer@frame</code> <code>\beamer@masterdecode</code>	<code>\beamer@frame {⟨<i>overlay specification</i>⟩}</code> <code>\beamer@masterdecode {⟨<i>overlay specification</i>⟩}</code>
---	---

Preprocess `⟨overlay specification⟩` before beamer uses 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.

```

332 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
333 \cs_set:Npn \beamer@frame < #1 > {
334   \__bnvs_group_begin:
335   \tl_clear:N \l__bnvs_ans_tl
336   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
337   \exp_args:NNNV
338   \__bnvs_group_end:
339   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
340 }
341 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
342 \cs_set:Npn \beamer@masterdecode #1 {
343   \__bnvs_group_begin:
344   \tl_clear:N \l__bnvs_ans_tl
345   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
346   \exp_args:NNV
347   \__bnvs_group_end:
348   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
349 }
```

<u>_bnvs_scan:nnN</u>	<p>_bnvs_scan:nnN {<i>(named overlay expression)</i>} <i><eval></i> <i><tl variable></i></p> <p>Scan the <i><named overlay expression></i> argument and feed the <i><tl variable></i> replacing <i>?(...)</i> instructions by their static counterpart with help from the <i><eval></i> function, which is _bnvs_eval:nn. A group is created to use local variables:</p> <p>\l_ans_tl: is the token list that will be appended to <i><tl variable></i> on return.</p>
\l__bnvs_depth_int	<p>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 <i>?(...)</i> instruction.</p> <p>(End definition for \l__bnvs_depth_int.)</p>
\l__bnvs_query_tl	<p>Storage for the overlay query expression to be evaluated.</p> <p>(End definition for \l__bnvs_query_tl.)</p>
\l__bnvs_token_seq	<p>The <i><overlay expression></i> is split into the sequence of its tokens.</p> <p>(End definition for \l__bnvs_token_seq.)</p>
\l__bnvs_ask_bool	<p>Whether a loop may continue. Controls the continuation of the main loop that scans the tokens of the <i><named overlay expression></i> looking for a question mark.</p> <p>(End definition for \l__bnvs_ask_bool.)</p>
\l__bnvs_query_bool	<p>Whether a loop may continue. Controls the continuation of the secondary loop that scans the tokens of the <i><named overlay expression></i> looking for an opening parenthesis follow the question mark. It then controls the loop looking for the balanced closing parenthesis.</p> <p>(End definition for \l__bnvs_query_bool.)</p>
\l__bnvs_token_tl	<p>Storage for just one token.</p> <p>(End definition for \l__bnvs_token_tl.)</p> <pre> 350 \cs_new:Npn _bnvs_scan:nnN #1 #2 #3 { 351 _bnvs_group_begin: 352 \tl_clear:N \l__bnvs_ans_tl 353 \int_zero:N \l__bnvs_depth_int 354 \seq_clear:N \l__bnvs_token_seq </pre> <p>Explode the <i><named overlay expression></i> into a list of tokens:</p> <pre> 355 \regex_split:nnN {} { #1 } \l__bnvs_token_seq </pre> <p>Run the top level loop to scan for a ‘?’:</p> <pre> 356 \bool_set_true:N \l__bnvs_ask_bool 357 \bool_while_do:Nn \l__bnvs_ask_bool { 358 \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl 359 \quark_if_no_value:NTF \l__bnvs_token_tl { </pre> <p>We reached the end of the sequence (and the token list), we end the loop here.</p> <pre> 360 \bool_set_false:N \l__bnvs_ask_bool 361 } { </pre> <p>\l_token_tl contains a ‘normal’ token.</p> <pre> 362 \tl_if_eq:NnTF \l__bnvs_token_tl { ? } { </pre>

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

```
363         \bool_set_true:N \l__bnvs_query_bool
364         \bool_while_do:Nn \l__bnvs_query_bool {
```

Get next token.

```
365         \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
366         \quark_if_no_value:NTF \l__bnvs_token_tl {
```

No opening parenthesis found, raise.

```
367         \msg_fatal:nxx { beanoves } { :n } {Missing~'('%---)
368         ~after~a~?:~#1}
369     } {
370         \tl_if_eq:NnT \l__bnvs_token_tl { ( %)
371     } {
```

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

```
372         \int_incr:N \l__bnvs_depth_int
```

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

```
373         \tl_clear:N \l__bnvs_query_tl
374         \bool_while_do:Nn \l__bnvs_query_bool {
```

Get next token.

```
375         \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
376         \quark_if_no_value:NTF \l__bnvs_token_tl {
```

We reached the end of the sequence and the token list with no closing ')'. We raise and end both bool while loops. As recovery we feed \l_query_tl with the missing ')'. \l__bnvs_depth_int is 0 whenever \l__bnvs_query_bool is false.

```
377         \msg_error:nxx { beanoves } { :n } {Missing~%((---
378         ~)~':~#1 }
379         \int_do_while:nNnn \l__bnvs_depth_int > 1 {
380             \int_decr:N \l__bnvs_depth_int
381             \tl_put_right:Nn \l__bnvs_query_tl {%(---
382             ~)~}
383         }
384         \int_zero:N \l__bnvs_depth_int
385         \bool_set_false:N \l__bnvs_query_bool
386         \bool_set_false:N \l__bnvs_ask_bool
387     } {
388         \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
389     } {
```

We found a '(', increment the depth and append the token to \l_query_tl.

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

This is not a '('.

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

We found a ')', decrement 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 stop the inner loop.

```

398     \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
399     \bool_set_false:N \l__bnvs_query_bool
400   } {

```

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

```

401             \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
402           }

```

Above ends the code for a positive depth.

```

403   } {

```

The scanned token is not a '(' nor a ')', we append it as is to \l_query_tl.

```

404             \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
405           }
406         }
407       }

```

Above ends the code for Not a '('

```

408     }
409   }

```

Above ends the code for: Found the '(' after the '?'

```

410   }

```

Above ends the code for not a no value quark.

```

411   }

```

Above ends the code for the bool while loop to find the '(' after the '?'.

If we reached the end of the token list, then end both the current loop and its containing loop.

```

412     \quark_if_no_value:NT \l__bnvs_token_tl {
413       \bool_set_false:N \l__bnvs_query_bool
414       \bool_set_false:N \l__bnvs_ask_bool
415     }
416   } {

```

This is not a '?', append the token to right of \l_ans_tl and continue.

```

417       \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
418     }

```

Above ends the code for the bool while loop to find a '(' after the '?'

```

419   }
420 }

```

Above ends the outer bool while loop to find '?' characters. We can append our result to *<tl variable>*

```

421 \exp_args:NNNV
422 \__bnvs_group_end:
423 \tl_put_right:Nn #3 \l__bnvs_ans_tl
424 }

```

I

5.5.6 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 <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.

```

425 \exp_args_generate:n { VVx }
426 \prg_new_conditional:Npnn __bnvs_extract_key:NNN
427   #1 #2 #3 { T, F, TF } {
428     __bnvs_group_begin:
429     \exp_args:NNV
430     \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

```

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

```

No new integer path component is added.

```

442     \cs_set:Npn \:nn ##1 ##2 {
443       __bnvs_group_end:
444       \tl_set:Nn #1 { ##1 }
445       \tl_set:Nn #2 { ##2 }
446     }
447     \exp_args:NVV \:nn #1 #2
448   } {

```

Some new integer path components are added.

```

449     \cs_set:Npn \:nnn ##1 ##2 ##3 {
450       __bnvs_group_end:
451       \tl_set:Nn #1 { ##1 }
452       \tl_set:Nn #2 { ##2 }
453       \seq_set_split:Nnn #3 . { ##3 }
454       \seq_remove_all:Nn #3 { }
455     }
456     \exp_args:NVVx
457     \:nnn #1 #2 {
458       \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
459     }

```

```

460 </!gubed>
461 % \end{gobble}
462 % \begin{macrocode}
463 }
464 \prg_return_true:
465 } {
466 \__bnvs_group_end:
467 \prg_return_false:
468 }
469 }

```

```

\__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, $\{\langle false\ code\rangle\}$ 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 $\backslash l_bnvs_id_tl$ 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 $\backslash l_a_tl$, $\backslash l_b_tl$ or $\backslash l_c_tl$. None of the seq variables must be one of $\backslash l_a_seq$, $\backslash l_b_seq$.

```

470 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
471 #1 #2 #3 { T, F, TF } {
472 \__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.

```

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

```

```

524     \prg_return_true:
525   }
526   \loop:
527 }

```

```

\__bnvs_resolve_n:NNNTF TF \__bnvs_resolve_n:NNNTF <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

```

528 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
529   #1 #2 #3 { T, F, TF } {
530   \__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.

```

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

```

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

```

545       \cs_set:Nn \loop: { \return_true: }
546     }
547   }
548 }
549 } {
550   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
551   \get_extract:F {
552     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
553       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
554     }
555   }
556 }
557 \loop:
558 } {

```



```

559     \__bnvs_group_end:
560     \prg_return_false:
561 }
562 }
563 \cs_set:Npn \get_extract:F ##1 {
564   \exp_args:Nx
565   \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
566     \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
567       \tl_set_eq:NN #2 \l__bnvs_b_tl
568       \seq_set_eq:NN #3 \l__bnvs_b_seq
569       \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
570       \seq_clear:N \l__bnvs_b_seq
571     } { ##1 }
572   } { ##1 }
573 }
574 \cs_set:Npn \return_true: {
575   \cs_set:Npn \:nnn #####1 ####2 ####3 {
576     \__bnvs_group_end:
577     \tl_set:Nn #1 { #####1 }
578     \tl_set:Nn #2 { #####2 }
579     \seq_set_split:Nnn #3 . { #####3 }
580     \seq_remove_all:Nn #3 { }
581   }
582   \exp_args:NVVx
583   \:nnn #1 #2 {
584     \seq_use:Nn #3 .
585   }
586   \prg_return_true:
587 }
588 \loop:
589 }

```

```

\__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.

```

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

```

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

```

If there is a length, no resolution occurs.

```

642     \branch:n { 1 }
643   } {
644     \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
645       \seq_clear:N \l__bnvs_b_seq

```

```

646         \tl_set:Nn \l__bnvs_a_tl { #3 . }
647         \tl_put_right:Nx \l__bnvs_a_tl {
648             \seq_use:Nn \l__bnvs_a_seq . .
649         }
650         \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
651         \branch:FF {

```

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

```

652 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
653     } {

```

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

```

654 \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
655     }
656     } {
657         \branch:FF {
658             \cs_set_eq:NN \loop: \return_true:
659         } {
660             \cs_set:Npn \loop: {
661                 \__bnvs_group_end:
662                 \prg_return_false:
663             }
664         }
665     }
666 } {
667     \cs_set:Npn \loop: {
668         \__bnvs_group_end:
669         \prg_return_false:
670     }
671 }
672 }
673 \loop:
674 }
675 \loop:
676 } {
677     \prg_return_true:
678 }
679 }
680 \prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
681     #1 #2 #3 #4 { T, F, TF } {
682     #1 {
683         \__bnvs_group_begin:

```

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

```

684     \tl_set_eq:NN \l__bnvs_a_tl #3
685     \seq_set_eq:NN \l__bnvs_a_seq #4
686     \cs_set:Npn \return_true: {
687         \cs_set:Npn \:nnn #####1 #####2 #####3 {
688             \__bnvs_group_end:
689             \tl_set:Nn #2 { #####1 }
690             \tl_set:Nn #3 { #####2 }
691             \seq_set_split:Nnn #4 . { #####3 }
692             \seq_remove_all:Nn #4 { }

```

```

693     }
694     \exp_args:NVVx
695     \:nnn #2 #3 {
696         \seq_use:Nn #4 .
697     }
698     \prg_return_true:
699 }
700 \cs_set:Npn \branch:n ##1 {
701     \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
702         \tl_put_right:Nn \l__bnvs_a_tl { . }
703         \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
704     } {
705         \cs_set_eq:NN \loop: \return_true:
706     }
707 }
708 \cs_set:Npn \loop: {
709     \__bnvs_call:TF {
710         \exp_args:Nx
711         \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
712             \branch:n { 1 }
713         } {
714             \exp_args:Nx
715             \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
716                 \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
717                     \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
718                     \tl_set_eq:NN #3 \l__bnvs_b_tl
719                     \seq_set_eq:NN #4 \l__bnvs_a_seq
720                 } {
721                     \branch:n { 2 }
722                 }
723             } {
724                 \branch:n { 3 }
725             }
726         }
727     } {
728         \cs_set:Npn \loop: {
729             \__bnvs_group_end:
730             \prg_return_false:
731         }
732     }
733     \loop:
734 }
735 \loop:
736 } {
737     \prg_return_true:
738 }
739 }

```

5.5.7 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.

```

740 \cs_new:Npn \__bnvs_fp_round:nN #1 #2 {
741   \tl_if_empty:nTF { #1 } {
742     } {
743       \tl_put_right:Nx #2 {
744         \fp_eval:n { round(#1) }
745       }
746     }
747   }
748 \cs_generate_variant:Nn \__bnvs_fp_round:nN { VN, xN }
749 \cs_new:Npn \__bnvs_fp_round:N #1 {
750   \tl_if_empty:VTF #1 {
751     } {
752       \tl_set:Nx #1 {
753         \fp_eval:n { round(#1) }
754       }
755     }
756   }

```

<code>__bnvs_raw_first:nNTF</code>	<code>__bnvs_raw_first:nNTF {<name>} <tl variable> {<true code>} {<false code>}</code>
<code>__bnvs_raw_first:(xN VN)TF</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.

```

757 \cs_set:Npn \__bnvs_return_true:nnN #1 #2 #3 {
758   \tl_if_empty:NTF \l__bnvs_ans_tl {
759     \__bnvs_group_end:
760     \__bnvs_gremove:n { #1//#2 }
761     \prg_return_false:
762   } {
763     \__bnvs_fp_round:N \l__bnvs_ans_tl
764     \__bnvs_gput:nV { #1//#2 } \l__bnvs_ans_tl
765     \exp_args:NNNV
766     \__bnvs_group_end:
767     \tl_put_right:Nn #3 \l__bnvs_ans_tl
768     \prg_return_true:
769   }
770 }
771 \cs_set:Npn \__bnvs_return_false:nn #1 #2 {
772   \__bnvs_group_end:
773   \__bnvs_gremove:n { #1//#2 }
774   \prg_return_false:
775 }
776 \prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {

```

```

777  \__bnvs_if_in:nTF { #1//A } {
778      \tl_put_right:Nx #2 { \__bnvs_item:n { #1//A } }
779      \prg_return_true:
780  } {
781      \__bnvs_group_begin:
782      \tl_clear:N \l__bnvs_ans_tl
783      \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
784          \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
785              \__bnvs_return_true:nn { #1 } A #2
786          } {
787              \__bnvs_return_false:nn { #1 } A
788          }
789      } {
790          \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
791              \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
792                  \__bnvs_if_append:xNTF {
793                      \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
794                  } \l__bnvs_ans_tl {
795                      \__bnvs_return_true:nn { #1 } A #2
796                  } {
797                      \__bnvs_return_false:nn { #1 } A
798                  }
799              } {
800                  \__bnvs_return_false:nn { #1 } A
801              }
802          } {
803              \__bnvs_return_false:nn { #1 } A
804          }
805      }
806  }
807 }
808 \prg_generate_conditional_variant:Nnn
809   \__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.

```

810 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
811     \__bnvs_raw_first:nNTF { #1 } #2 {
812         \prg_return_true:
813     } {
814         \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {
815             \bool_set_true:N \l_no_counter_bool
816             \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
817                 \__bnvs_return_true:nn { #1 } A #2
818             } {
819                 \__bnvs_return_false:nn { #1 } A

```

```

820     }
821   } {
822     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
823       \__bnvs_gput:nn { #1/A } { 1 }
824       \tl_set:Nn #2 { 1 }
825
826       \__bnvs_return_true:nnN { #1 } A #2
827     } {
828       \__bnvs_return_false:nn { #1 } A
829     }
830   }
831 }

```

__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.

```

832 \cs_new:Npn \__bnvs_first:nN #1 #2 {
833   \__bnvs_if_first:nNF { #1 } #2 {
834     \msg_error:nnn { beanoves } { :n } { Range-with-no-first:~#1 }
835   }
836 }
837 \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.

```

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

```

```

861         } {
862             \__bnvs_return_false:nn { #1 } L
863         }
864     } {
865         \__bnvs_return_false:nn { #1 } L
866     }
867 } {
868     \__bnvs_return_false:nn { #1 } L
869 }
870 }
871 }
872 }
873 \prg_generate_conditional_variant:Nnn
874   \__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.

```

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

```



```

909 }
910 \prg_generate_conditional_variant:Nnn
911   \__bnvs_raw_last:nN { VN } { T, F, TF }

```

$\underline{\underline{\text{_bnvs_last:nN}}}$	$\underline{\underline{\text{_bnvs_last:VN}}}$	$\text{_bnvs_last:nN } \{ \langle name \rangle \} \langle tl \ variable \rangle$ Append the last index of the fully qualified $\langle name \rangle$ slide range to $\langle tl \ variable \rangle$
--	--	--

```

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

```

$\underline{\underline{\text{_bnvs_if_next:nNTF}}}$	$\text{_bnvs_if_next:nNTF } \{ \langle name \rangle \} \langle tl \ variable \rangle \{ \langle true \ code \rangle \} \{ \langle false \ code \rangle \}$ Append the index after the $\langle name \rangle$ slide range to the $\langle tl \ variable \rangle$. Execute $\langle true \ code \rangle$ when there is a $\langle next \rangle$ index, $\langle false \ code \rangle$ otherwise.
--	---

```

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

```

```

953 }
954 }
955 \prg_generate_conditional_variant:Nnn
956   \__bnvs_if_next:nN { VN } { T, F, TF }

```

$\backslash_bnvs_next:nN$ $\backslash_bnvs_next:VN$	$\backslash_bnvs_next:nN \{ \langle name \rangle \} \langle tl \ variable \rangle$ Append the index after the $\langle name \rangle$ slide range to the $\langle tl \ variable \rangle$.
--	--

```

957 \cs_new:Npn \__bnvs_next:nN #1 #2 {
958   \__bnvs_if_next:nNF { #1 } #2 {
959     \msg_error:nnn { beanoves } { :n } { Range-with-no-next:~#1 }
960   }
961 }
962 \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.
--	---

```

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

```

978 \prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {
979   \bool_if:NTF \l__bnvs_no_range_bool {
980     \prg_return_false:
981   } {
982     \__bnvs_if_in:nTF { #1/ } {
983       \tl_put_right:Nn { 0-0 }

```

```

984   } {
985     \__bnvs_group_begin:
986     \tl_clear:N \l__bnvs_a_tl
987     \tl_clear:N \l__bnvs_b_tl
988     \tl_clear:N \l__bnvs_ans_tl
989     \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
990       \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
991         \exp_args:NNNx
992         \__bnvs_group_end:
993         \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
994
995         \prg_return_true:
996       } {
997         \exp_args:NNNx
998         \__bnvs_group_end:
999         \tl_put_right:Nn #2 { \l__bnvs_a_tl - }
1000
1001         \prg_return_true:
1002       }
1003     } {
1004       \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
1005         \exp_args:NNNx
1006         \__bnvs_group_end:
1007         \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1008         \prg_return_true:
1009       } {
1010         \__bnvs_group_end:
1011         \prg_return_false:
1012       }
1013     }
1014   }
1015   \prg_generate_conditional_variant:Nnn
1016   \__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 $\langle tl\ variable \rangle$.
--	--

```

1017 \cs_new:Npn \__bnvs_range:nN #1 #2 {
1018   \__bnvs_if_range:nNF { #1 } #2 {
1019     \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1020   }
1021 }
1022 \cs_generate_variant:Nn \__bnvs_range:nN { VN }

```

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

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

```

1023 \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {

```

```

1024  \__bnvs_group_begin:
1025  \tl_clear:N \l__bnvs_ans_tl
1026  \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
1027    \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
1028      \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
1029    }
1030  }
1031  \tl_if_empty:NTF \l__bnvs_ans_tl {
1032    \__bnvs_group_end:
1033    \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1034      \__bnvs_gput:nn { #1/C } { 1 }
1035      \tl_set:Nn #2 { 1 }
1036
1037      \prg_return_true:
1038    } {
1039
1040      \prg_return_false:
1041    } {
1042      \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl
1043      \exp_args:NNNV
1044      \__bnvs_group_end:
1045      \tl_set:Nn #2 \l__bnvs_ans_tl
1046
1047      \prg_return_true:
1048    }
1049  }
1050  \prg_generate_conditional_variant:Nnn
1051  \__bnvs_if_free_counter:nN { VN } { T, F, TF }

```

```

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

```

`__bnvs_if_counter:VNTF` Append the value of the counter associated to the `{<name>}` slide range to the right of `<tl variable>`. The value always lays in between the range, whenever possible.

```

1050 \prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {

```

```

1051  \__bnvs_group_begin:
1052  \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_ans_tl {

```

If there is a `<first>`, use it to bound the result from below.

```

1053    \tl_clear:N \l__bnvs_a_tl
1054    \__bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
1055      \fp_compare:nNtT { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
1056        \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1057      }
1058    }

```

If there is a `<last>`, use it to bound the result from above.

```

1059    \tl_clear:N \l__bnvs_a_tl
1060    \__bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
1061      \fp_compare:nNtT { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
1062        \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1063      }
1064    }
1065    \exp_args:NNV
1066    \__bnvs_group_end:
1067    \__bnvs_fp_round:nN \l__bnvs_ans_tl #2

```

```

1068     \prg_return_true:
1069   } {

1070     \prg_return_false:
1071   }
1072 }
1073 \prg_generate_conditional_variant:Nnn
1074   \__bnvs_if_counter:nN { VN } { T, F, TF }

```

$\backslash_bnvs_if_incr:nn\overline{TF}$ $\backslash_bnvs_if_incr:nn\overline{NTF}$ $\backslash_bnvs_if_incr:(VnN VVN)\overline{TF}$	$\backslash_bnvs_if_incr:nnTF \{ \langle name \rangle \} \{ \langle offset \rangle \} \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$ $\backslash_bnvs_if_incr:nnNTF \{ \langle name \rangle \} \{ \langle offset \rangle \} \langle tl\ variable \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
--	--

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.

```

1075 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {

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

```

5.5.8 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>
1107 \int_new:N \l__bnvs_split_int	
\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:	
	<p>Local variables:</p>
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	
	<p>Implementation:</p>
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:NNTF \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 ++ $\langle name \rangle \langle integer path \rangle .n$.

```

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 $\langle name \rangle \langle integer path \rangle \dots$

```

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:NNTF 5 6 {
1204             \:T {
1205                 \__bnvs_if_next:VNF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1206                     \cs_set_eq:NN \loop: \return_false:
1207                 }
1208             }
1209         }
1210     } {
1211         \switch:nNTF { 10 } \l__bnvs_a_t1 {

```

- Case ...range.

```

1212 \:NnnT \__bnvs_resolve:NNTF 5 6 {
1213     \:T {
1214         \__bnvs_if_range:VNTF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1215             \cs_set_eq:NN \fp_round: \prg_do_nothing:
1216         } {
1217             \cs_set_eq:NN \loop: \return_false:
1218         }
1219     }
1220 }
1221     } {
1222         \switch:nNTF { 11 } \l__bnvs_a_t1 {

```

- Case ...n.

```

1223         \switch:nNTF { 12 } \l__bnvs_a_t1 {

```

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

```

1224 \:NnnT \__bnvs_resolve_n:NNTF 5 6 {
1225     \:T {
1226         \__bnvs_if_incr:VVNF \l__bnvs_name_t1 \l__bnvs_a_t1 \l__bnvs_ans_t1 {
1227             \cs_set_eq:NN \loop: \return_false:
1228         }
1229     }
1230 }
1231     } {
1232         \:NnnT \__bnvs_resolve_n:NNTF 5 6 {
1233             \seq_if_empty:NTF \l__bnvs_path_seq {
1234 \__bnvs_if_counter:VNF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1235     \cs_set_eq:NN \loop: \return_false:
1236 }
1237         } {
1238 \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_t1
1239 \seq_if_empty:NTF \l__bnvs_path_seq {
1240     \__bnvs_if_incr:VVNF \l__bnvs_name_t1 \l__bnvs_a_t1 \l__bnvs_ans_t1 {
1241         \cs_set_eq:NN \loop: \return_false:
1242     }
1243 } {
1244     \msg_error:nxx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
1245     \cs_set_eq:NN \loop: \return_false:
1246 }

```

```

1247         }
1248     }
1249 }

1250     } {
1251         \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1252             \seq_if_empty:NTF \l__bnvs_path_seq {
1253 \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1254     \cs_set_eq:NN \loop: \return_false:
1255 }
1256         } {
1257             \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1258             \seq_if_empty:NTF \l__bnvs_path_seq {
1259 \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1260     \cs_set_eq:NN \loop: \return_false:
1261 }
1262         } {
1263 \msg_error:nx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1264 \cs_set_eq:NN \loop: \return_false:
1265         }
1266     }
1267 }
1268 }
1269 }
1270 }
1271 }
1272 }
1273 } {

```

No name.

```

1274     }
1275 }

1276     \int_add:Nn \l__bnvs_split_int { 13 }
1277     \tl_put_right:Nx \l__bnvs_ans_tl {
1278         \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1279     }

1280     \loop:
1281 } {

1282     \return_true:
1283 }
1284 }
1285     \loop:
1286 } {
1287     \msg_error:nx { beanoves } { :n } { Too-many~calls:~ #1 }
1288     \prg_return_false:
1289 }
1290 }
1291 \prg_generate_conditional_variant:Nnn
1292 \__bnvs_if_append:nN { VN, xN } { T, F, TF }

```

<u>_bnvs_if_eval_query:nNTF</u>	<p><code>_bnvs_if_eval_query:nNTF {<overlay query>} <tl variable> {<true code>} {<false code>}</code></p> <p>Evaluates the single <i><overlay query></i>, which is expected to contain no comma. Extract a range specification from the argument, replaces all the <i>named overlay specifications</i> by their static counterparts, make the computation then append the result to the right of the <i><seq variable></i>. Ranges are supported with the colon syntax. This is executed within a local group. Below are local variables and constants.</p> <p><code>\l__bnvs_a_tl</code> Storage for the first index of a range.</p> <p>(End definition for <code>\l__bnvs_a_tl</code>.)</p> <p><code>\l__bnvs_b_tl</code> Storage for the last index of a range, or its length.</p> <p>(End definition for <code>\l__bnvs_b_tl</code>.)</p> <p><code>\c__bnvs_A_cln_Z_regex</code> Used to parse slide range overlay specifications. Next are the capture groups.</p> <p>(End definition for <code>\c__bnvs_A_cln_Z_regex</code>.)</p> <pre> 1293 \regex_const:Nn \c__bnvs_A_cln_Z_regex { 1294 \A \s* (? </pre> <ul style="list-style-type: none"> • 2: <i><first></i> <pre> 1295 ([^:]*) \s* : </pre> <ul style="list-style-type: none"> • 3: second optional colon <pre> 1296 (:)? \s* </pre> <ul style="list-style-type: none"> • 4: <i><length></i> <pre> 1297 ([^:]*) </pre> <ul style="list-style-type: none"> • 5: standalone <i><first></i> <pre> 1298 ([^:]+) 1299) \s* \Z 1300 } </pre> <pre> 1301 \prg_new_conditional:Npnn _bnvs_if_eval_query:nN #1 #2 { T, F, TF } { 1302 _bnvs_call_reset: 1303 \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex { 1304 #1 1305 } \l__bnvs_match_seq { 1306 \bool_set_false:N \l__bnvs_no_counter_bool 1307 \bool_set_false:N \l__bnvs_no_range_bool </pre> <tr> <td><u>\switch:nNTF</u></td><td> <p><code>\switch:nNTF {<capture group number>} <tl variable> {<black code>} {<white code>}</code></p> <p>Helper function to locally set the <i><tl variable></i> to the captured group <i><capture group number></i> and branch depending on the emptiness of this variable.</p> <pre> 1308 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 { </pre> </td></tr>	<u>\switch:nNTF</u>	<p><code>\switch:nNTF {<capture group number>} <tl variable> {<black code>} {<white code>}</code></p> <p>Helper function to locally set the <i><tl variable></i> to the captured group <i><capture group number></i> and branch depending on the emptiness of this variable.</p> <pre> 1308 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 { </pre>
<u>\switch:nNTF</u>	<p><code>\switch:nNTF {<capture group number>} <tl variable> {<black code>} {<white code>}</code></p> <p>Helper function to locally set the <i><tl variable></i> to the captured group <i><capture group number></i> and branch depending on the emptiness of this variable.</p> <pre> 1308 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 { </pre>		

```

1309     \tl_set:Nx ##2 {
1310       \seq_item:Nn \l__bnvs_match_seq { ##1 }
1311     }
1312     \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1313   }
1314   \switch:nNTF 5 \l__bnvs_a_tl {

```

Single expression

```

1315     \bool_set_false:N \l__bnvs_no_range_bool
1316     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1317       \prg_return_true:
1318     } {
1319       \prg_return_false:
1320     }
1321   } {
1322     \switch:nNTF 2 \l__bnvs_a_tl {
1323       \switch:nNTF 4 \l__bnvs_b_tl {
1324         \switch:nNTF 3 \l__bnvs_c_tl {

```

$\langle first \rangle :: \langle last \rangle$ range

```

1325     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1326       \tl_put_right:Nn #2 { - }
1327       \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1328         \prg_return_true:
1329       } {
1330         \prg_return_false:
1331       }
1332     } {
1333       \prg_return_false:
1334     }
1335   } {

```

$\langle first \rangle : \langle length \rangle$ range

```

1336     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1337       \tl_put_right:Nx #2 { - }
1338       \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1 }
1339       \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1340         \prg_return_true:
1341       } {
1342         \prg_return_false:
1343       }
1344     } {
1345       \prg_return_false:
1346     }
1347   }
1348   } {

```

$\langle first \rangle :$ and $\langle first \rangle ::$ range

```

1349     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1350       \tl_put_right:Nn #2 { - }
1351       \prg_return_true:
1352     } {
1353       \prg_return_false:
1354     }
1355   }

```

```

1356     } {
1357         \switch:nNTF 4 \l__bnvs_b_tl {
1358             \switch:nNTF 3 \l__bnvs_c_tl {
1359                 \tl_put_right:Nn #2 { - }
1360                 \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1361                     \prg_return_true:
1362                 } {
1363                     \prg_return_false:
1364                 }
1365             } {
1366 \msg_error:nnx { beanoves } { :n } { Syntax~error(Missing~first):~#1 }
1367             }
1368         } {
1369             : or :: range
1370             \seq_put_right:Nn #2 { - }
1371         }
1372     }
1373 } {
Error
1374 \msg_error:nnn { beanoves } { :n } { Syntax~error:~#1 }
1375 }
1376 }

```

`__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.

```

1377 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

(End definition for \c__bnvs_comma_regex.)
No other variable is used.

```

1378 \cs_new:Npn \__bnvs_eval:nN #1 #2 {
1379     \__bnvs_group_begin:

```

Local variables declaration

```

1380     \seq_clear:N \l__bnvs_query_seq
1381     \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 $\backslash l_query_seq$.

```
1382 \regex_split:Nn \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.

```
1383 \seq_map_inline:Nn \l__bnvs_query_seq {
1384   \tl_clear:N \l__bnvs_ans_tl
1385   \__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {
1386     \seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl
1387   } {
1388     \seq_map_break:n {
1389       \msg_fatal:nnn { beanoves } { :n } { Circular-dependency-in~#1}
1390     }
1391   }
1392 }
```

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

```
1393 \exp_args:NNNx
1394 \__bnvs_group_end:
1395 \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
1396 }
1397 \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.

```
1398 \NewDocumentCommand \BeanovesEval { s o m } {
1399   \__bnvs_group_begin:
1400   \tl_clear:N \l__bnvs_ans_tl
1401   \IfBooleanTF { #1 } {
1402     \bool_set_true:N \l__bnvs_no_counter_bool
1403   } {
1404     \bool_set_false:N \l__bnvs_no_counter_bool
1405   }
1406   \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
1407   \IfValueTF { #2 } {
1408     \exp_args:NNNV
1409     \__bnvs_group_end:
1410     \tl_set:Nn #2 \l__bnvs_ans_tl
1411   } {
1412     \exp_args:NV
1413     \__bnvs_group_end: \l__bnvs_ans_tl
1414   }
1415 }
```

5.5.9 Resetting slide ranges

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

```

1416 \NewDocumentCommand \BeanovesReset { 0{1} m } {
1417   \__bnvs_reset:nn { #1 } { #2 }
1418   \ignorespaces
1419 }

```

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.

```

1420 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1421   \bool_if:nTF {
1422     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1423   } {
1424     \__bnvs_gremove:n { #2/C }
1425     \__bnvs_gremove:n { #2//A }
1426     \__bnvs_gremove:n { #2//L }
1427     \__bnvs_gremove:n { #2//Z }
1428     \__bnvs_gremove:n { #2//N }
1429     \__bnvs_gput:nn { #2/CO } { #1 }
1430   } {
1431     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1432   }
1433 }

1434 \makeatother
1435 \ExplSyntaxOff

1436 \</package>

```