

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   {beanoves}
5   {2022/10/28}
6   {1.0}
7   {Named overlay specifications for beamer}
```

5.2 logging and debugging facilities

Utility message.

```
8 \msg_new:nnn { beanoves } { :n } { #1 }
9 \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.

```
10 \int_new:N \l__bnvs_depth_int
11 \bool_new:N \l__bnvs_ask_bool
12 \bool_new:N \l__bnvs_query_bool
13 \bool_new:N \l__bnvs_no_counter_bool
14 \bool_new:N \l__bnvs_no_range_bool
15 \bool_new:N \l__bnvs_continue_bool
16 \bool_new:N \l__bnvs_in_frame_bool
17 \bool_set_false:N \l__bnvs_in_frame_bool
18 \tl_new:N \l__bnvs_id_current_tl
19 \tl_new:N \l__bnvs_a_tl
20 \tl_new:N \l__bnvs_b_tl
21 \tl_new:N \l__bnvs_c_tl
22 \tl_new:N \l__bnvs_id_tl
23 \tl_new:N \l__bnvs_ans_tl
24 \tl_new:N \l__bnvs_name_tl
25 \tl_new:N \l__bnvs_path_tl
26 \tl_new:N \l__bnvs_group_tl
```

```

27 \tl_new:N \l__bnvs_query_tl
28 \tl_new:N \l__bnvs_token_tl
29 \seq_new:N \l__bnvs_a_seq
30 \seq_new:N \l__bnvs_b_seq
31 \seq_new:N \l__bnvs_ans_seq
32 \seq_new:N \l__bnvs_match_seq
33 \seq_new:N \l__bnvs_split_seq
34 \seq_new:N \l__bnvs_path_seq
35 \seq_new:N \l__bnvs_query_seq
36 \seq_new:N \l__bnvs_token_seq

```

5.4 Infinite loop management

Unending recursivity is managed here.

`\g__bnvs_call_int`

```

37 \int_zero_new:N \g__bnvs_call_int
38 \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.

```

39 \cs_set:Npn \__bnvs_call_reset: {
40   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
41 }

```

`__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.

```

42 \prg_new_conditional:Npnn \__bnvs_call: { T, F, TF } {
43   \int_gdecr:N \g__bnvs_call_int
44   \int_compare:nNnTF \g__bnvs_call_int > 0 {
45     \prg_return_true:
46   } {
47     \prg_return_false:
48   }
49 }

```

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

*<id>!**<name>/L* for the length when provided

*<id>!**<name>/Z* for the last index when provided

*<id>!**<name>/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.

50 `\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.

```

51 \cs_new:Npn \__bnvs_gput:nn #1 #2 {
52   \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
53 }
54 \cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
55   \prop_if_in:NnF \g__bnvs_prop { #1 } {
56     \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
57   }
58 }
59 \cs_new:Npn \__bnvs_item:n {
60   \prop_item:Nn \g__bnvs_prop
61 }
62 \cs_new:Npn \__bnvs_get:nN {
63   \prop_get:NnN \g__bnvs_prop
64 }
65 \cs_new:Npn \__bnvs_gremove:n {
66   \prop_gremove:Nn \g__bnvs_prop
67 }
68 \cs_new:Npn \__bnvs_gclear:n #1 {
69   \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
70     \__bnvs_gremove:n { #1 / ##1 }
71   }
72 }
73 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
74   \clist_map_inline:nn { /A, /L, /Z, /N } {
75     \__bnvs_gremove:n { #1 / ##1 }
76   }
77 }
78 \cs_new:Npn \__bnvs_gclear: {
79   \prop_gclear:N \g__bnvs_prop
80 }
81 \cs_generate_variant:Nn \__bnvs_gput:nn { nV }
82 \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.

```

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

```



```

86   } {
87     \prg_return_false:
88   }
89 }
90 \prg_generate_conditional_variant:Nnn \__bnvs_if_in:n {V} { p, T, F, TF }

```

```

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

```

91 \prg_new_conditional:Npnn \__bnvs_get:nN #1 #2 { T, F, TF } {
92   \prop_get:NnNTF \g__bnvs_prop { #1 } #2 {
93     \prg_return_true:
94   } {
95     \prg_return_false:
96   }
97 }

```

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.

```

98 \regex_const:Nn \c__bnvs_name_regex {
99   [[[:alpha:]]_][[:alnum:]]_*
100 }

```

(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.

```

101 \regex_const:Nn \c__bnvs_id_regex {
102   (?: \ur{c__bnvs_name_regex} | [?]* ) ? !
103 }

```

(End definition for `\c__bnvs_id_regex`.)

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

```

104 \regex_const:Nn \c__bnvs_path_regex {
105   (?: \. [+-]? \d+ ) *
106 }

```

(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.

```

107 \regex_const:Nn \c__bnvs_key_regex {
108   \ur{c__bnvs_id_regex} ?
109   \ur{c__bnvs_name_regex}

```

```

110     \ur{c__bnvs_path_regex}
111 }
112 \regex_const:Nn \c__bnvs_A_key_Z_regex {

    2: slide <id>

    3: question mark, when <id> is empty

    4: The range name

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

    5: the path, if any.

114     ( \ur{c__bnvs_path_regex} ) \Z
115 }
116

```

(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.

```

117 \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.

```

118 \regex_const:Nn \c__bnvs_list_regex {
119     \A \[ \s*

Capture groups:

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

120     ( [^\] %[---
121     ]*? )
122     \s* \] \Z
123 }

```

(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.

```

124 \regex_const:Nn \c__bnvs_split_regex {
125     \s* ( ? :

```

We start with ‘++’ instrussions².

- 1: <name> of a slide range
- 2: <id> of a slide range plus the exclamation mark

```

126     \+ \+ ( ( \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

127 (\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 $\langle id \rangle$)

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

- 6: optionally followed by an integer path

129 (\ur{c__bnvs_path_regex})

Next comes another branching

130 (?:

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

131 \ . l(e)ngth

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

132 | \ . l(a)st

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

133 | \ . ne(x)t

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

134 | \ . (r)ange

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

135 | \ . (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.

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

137)?

138) \s*

139 }

(End definition for \c__bnvs_split_regex.)

5.5.3 beamer.cls interface

Work in progress.

```

140 \RequirePackage{keyval}
141 \define@key{beamerframe}{beanoves~id}[]{}
142 \tl_set:Nx \l__bnvs_id_current_tl { #1 ! }
143 }
144 \AddToHook{env/beamer@frameslide/before}{
145   \bool_set_true:N \l__bnvs_in_frame_bool
146 }
147 \AddToHook{env/beamer@frameslide/after}{
148   \bool_set_false:N \l__bnvs_in_frame_bool
149 }
150 \AddToHook{cmd/frame/before}{
151   \tl_set:Nn \l__bnvs_id_current_tl { ?! }
152 }

```

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.

`\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`.

```

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

```

```

166     }
167   }
168   {
169     #1 { #2/A } { #3 }
170     \tl_if_empty:nTF { #4 } {
171       \tl_if_empty:nF { #5 } {
172         #1 { #2/Z } { #5 }
173         #1 { #2/L } { #2.last - (#2.1) + 1 }
174       }
175     } {
176       #1 { #2/L } { #4 }
177       #1 { #2/Z } { #2.1 + #2.length - 1 }
178     }
179   }
180 }
181 \cs_new:Npn \__bnvs_range:nnnn #1 {
182   \__bnvs_gclear:n { #1 }
183   \__bnvs_range:Nnnnn \__bnvs_gput:nn { #1 }
184 }
185 \cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
186 \cs_new:Npn \__bnvs_range_alt:nnnn #1 {
187   \__bnvs_gclear_cache:n { #1 }
188   \__bnvs_range:Nnnnn \__bnvs_gprovide:nn { #1 }
189 }
190 \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.

```

191 \cs_new:Npn \__bnvs_parse:Nn #1 #2 {
192   \__bnvs_group_begin:
193   \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {
194     \exp_args:Nx \__bnvs_gput:nn { \l__bnvs_name_tl/ } { }
195     \exp_args:NNNV
196     \__bnvs_group_end:
197     \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
198   } {
199     \msg_error:nnn { beanoves } { :n } { Unexpected~key:~#2 }
200     \__bnvs_group_end:
201   }
202 }

```

__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.

```

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

```

This is not a list.

```

205   \tl_clear:N \l__bnvs_a_tl
206   \tl_clear:N \l__bnvs_b_tl

```

```

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

```

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.

```

246   \bool_if:nF {
247     \tl_if_empty_p:N \l__bnvs_a_tl
248     || \tl_if_empty_p:N \l__bnvs_b_tl
249     || \tl_if_empty_p:N \l__bnvs_c_tl
250   } {
251 \msg_error:nnn { beanoves } { :n } { Invalid-range-expression(7):~#3 }
252   }
253   #1 { #2 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
254 }
255 \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 $\langle false \text{ code} \rangle$.

```

256 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNn #1 #2 #3 { T, F, TF } {
257   \__bnvs_group_begin:
258   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
259     #1
260   } \l__bnvs_match_seq {
261     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
262     \tl_if_empty:NTF #2 {
263       \exp_args:NNNx
264       \__bnvs_group_end:
265       \tl_set:Nn #3 { \l__bnvs_id_current_tl #1 }
266       \tl_set_eq:NN #2 \l__bnvs_id_current_tl
267     } {
268       \cs_set:Npn \:n ##1 {
269         \__bnvs_group_end:
270         \tl_set:Nn #2 { ##1 }
271         \tl_set:Nn \l__bnvs_id_current_tl { ##1 }
272       }
273       \exp_args:NV
274       \:n #2
275       \tl_set:Nn #3 { #1 }
276     }
277   \prg_return_true:
278 } {
279   \__bnvs_group_end:
280   \prg_return_false:
281 }
282 }

283 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
284   \__bnvs_group_begin:
285   \__bnvs_id_name_set:nNNTF { #2 } \l__bnvs_id_tl \l__bnvs_name_tl {

```

```

286 \regex_extract_once:NnNTF \c__bnvs_list_regex {
287   #3
288 } \l__bnvs_match_seq {
This is a comma separated list, extract each item and go recursive.
289 \exp_args:NNx
290 \seq_set_from_clist:Nn \l__bnvs_match_seq {
291   \seq_item:Nn \l__bnvs_match_seq { 2 }
292 }
293 \seq_map_indexed_inline:Nn \l__bnvs_match_seq {
294   \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl.##1 } { ##2 }
295 }
296 } {
297   \__bnvs_do_parse:Nxn #1 { \l__bnvs_name_tl } { #3 }
298 }
299 } {
300   \msg_error:nnn { beanoves } { :n } { Invalid~key:~#2 }
301 }
We export \l__bnvs_id_tl:
302 \exp_args:NNNV
303 \__bnvs_group_end:
304 \tl_set:Nn \l__bnvs_id_current_tl \l__bnvs_id_current_tl
305 }

```

\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.

```

306 \NewDocumentCommand \Beanoves { sm } {
307   \tl_if_eq:NnT \@currenvir { document } {
308     \__bnvs_gclear:
309   }
310   \IfBooleanTF {#1} {
311     \keyval_parse:nnn {
312       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
313     } {
314       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
315     }
316   } {
317     \keyval_parse:nnn {
318       \__bnvs_parse:Nn \__bnvs_range:nVVV
319     } {
320       \__bnvs_parse:Nnn \__bnvs_range:nVVV
321     }
322   }
323   { #2 }
324   \ignorespaces
325 }

```

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

```

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

```


5.5.5 Scanning named overlay specifications

Patch some beamer commands to support $\langle\text{overlay specification}\rangle$ instructions in overlay specifications.

$\backslash\text{beamer@frame}$ $\backslash\text{beamer@masterdecode}$	$\backslash\text{beamer@frame} \{\langle\text{overlay specification}\rangle\}$ $\backslash\text{beamer@masterdecode} \{\langle\text{overlay specification}\rangle\}$
---	---

Preprocess $\langle\text{overlay specification}\rangle$ before beamer uses it.

$\backslash\text{l_bnvs_ans_tl}$ Storage for the translated overlay specification, where $\langle\text{overlay specification}\rangle$ instructions are replaced by their static counterparts.

(End definition for $\backslash\text{l_bnvs_ans_tl}$.)

Save the original macro $\backslash\text{beamer@masterdecode}$ and then override it to properly preprocess the argument.

```

327 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
328 \cs_set:Npn \beamer@frame < #1 > {
329   \__bnvs_group_begin:
330   \tl_clear:N \l__bnvs_ans_tl
331   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
332   \exp_args:NNNV
333   \__bnvs_group_end:
334   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
335 }
336 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
337 \cs_set:Npn \beamer@masterdecode #1 {
338   \__bnvs_group_begin:
339   \tl_clear:N \l__bnvs_ans_tl
340   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
341   \exp_args:NNNV
342   \__bnvs_group_end:
343   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
344 }
```

`_bnvs_scan:nNN` `_bnvs_scan:nNN {⟨named overlay expression⟩} ⟨eval⟩ ⟨tl variable⟩`

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

`\l_ans_tl`: is the token list that will be appended to *⟨tl variable⟩* on return.

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

(End definition for `\l__bnvs_depth_int`.)

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

(End definition for `\l__bnvs_query_tl`.)

`\l__bnvs_token_seq` The *⟨overlay expression⟩* is split into the sequence of its tokens.

(End definition for `\l__bnvs_token_seq`.)

`\l__bnvs_ask_bool` Whether a loop may continue. Controls the continuation of the main loop that scans the tokens of the *⟨named overlay expression⟩* looking for a question mark.

(End definition for `\l__bnvs_ask_bool`.)

`\l__bnvs_query_bool` Whether a loop may continue. Controls the continuation of the secondary loop that scans the tokens of the *⟨named overlay expression⟩* looking for an opening parenthesis follow the question mark. It then controls the loop looking for the balanced closing parenthesis.

(End definition for `\l__bnvs_query_bool`.)

`\l__bnvs_token_tl` Storage for just one token.

(End definition for `\l__bnvs_token_tl`.)

```

345 \cs_new:Npn \_bnvs_scan:nNN #1 #2 #3 {
346   \_bnvs_group_begin:
347   \tl_clear:N \l__bnvs_ans_tl
348   \int_zero:N \l__bnvs_depth_int
349   \seq_clear:N \l__bnvs_token_seq

```

Explode the *⟨named overlay expression⟩* into a list of tokens:

```

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

```

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

```

351   \bool_set_true:N \l__bnvs_ask_bool
352   \bool_while_do:Nn \l__bnvs_ask_bool {
353     \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
354     \quark_if_no_value:NTF \l__bnvs_token_tl {

```

We reached the end of the sequence (and the token list), we end the loop here.

```

355       \bool_set_false:N \l__bnvs_ask_bool
356     } {

```

`\l_token_tl` contains a ‘normal’ token.

```

357       \tl_if_eq:NnTF \l__bnvs_token_tl { ? } {

```

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

```
358         \bool_set_true:N \l__bnvs_query_bool
359         \bool_while_do:Nn \l__bnvs_query_bool {
```

Get next token.

```
360         \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
361         \quark_if_no_value:NTF \l__bnvs_token_tl {
```

No opening parenthesis found, raise.

```
362         \msg_fatal:nxx { beanoves } { :n } {Missing~'('%---)
363         ~after~a~?:~#1}
364     } {
365         \tl_if_eq:NnT \l__bnvs_token_tl { ( %)
366     } {
```

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

```
367         \int_incr:N \l__bnvs_depth_int
```

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

```
368         \tl_clear:N \l__bnvs_query_tl
369         \bool_while_do:Nn \l__bnvs_query_bool {
```

Get next token.

```
370         \seq_pop_left:NN \l__bnvs_token_seq \l__bnvs_token_tl
371         \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.

```
372         \msg_error:nxx { beanoves } { :n } {Missing~%((---
373         ~)~':~#1 }
374         \int_do_while:nNnn \l__bnvs_depth_int > 1 {
375             \int_decr:N \l__bnvs_depth_int
376             \tl_put_right:Nn \l__bnvs_query_tl {%(---
377         )}
378     }
379     \int_zero:N \l__bnvs_depth_int
380     \bool_set_false:N \l__bnvs_query_bool
381     \bool_set_false:N \l__bnvs_ask_bool
382 } {
383     \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
384 } {
```

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

```
385         \int_incr:N \l__bnvs_depth_int
386         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
387     } {
```

This is not a '('.

```
388         \tl_if_eq:NnTF \l__bnvs_token_tl { %(
389         )
390     } {
```

We found a ')', decrement the depth.

```

391         \int_decr:N \l__bnvs_depth_int
392         \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.

```

393     \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
394     \bool_set_false:N \l__bnvs_query_bool
395     } {

```

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

```

396         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
397     }

```

Above ends the code for a positive depth.

```

398     } {

```

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

```

399         \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
400     }
401 }
402 }

```

Above ends the code for Not a '('

```

403     }
404 }

```

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

```

405     }

```

Above ends the code for not a no value quark.

```

406     }

```

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.

```

407     \quark_if_no_value:NT \l__bnvs_token_tl {
408         \bool_set_false:N \l__bnvs_query_bool
409         \bool_set_false:N \l__bnvs_ask_bool
410     }
411 } {

```

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

```

412     \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
413 }

```

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

```

414     }
415 }

```

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

```

416     \exp_args:NNNV
417     \__bnvs_group_end:
418     \tl_put_right:Nn #3 \l__bnvs_ans_tl
419 }

```

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 \__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.

```

420 \exp_args_generate:n { VVx }
421 \prg_new_conditional:Npnn \__bnvs_extract_key:NNN
422   #1 #2 #3 { T, F, TF } {
423   \__bnvs_group_begin:
424   \exp_args:NNV
425   \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

```

426   \exp_args:Nx
427   \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
428     \tl_put_left:NV #2 { #1 }
429   }
430   \exp_args:NNnx
431   \seq_set_split:Nnn \l__bnvs_split_seq . {
432     \seq_item:Nn \l__bnvs_match_seq 4
433   }
434   \seq_remove_all:Nn \l__bnvs_split_seq { }
435   \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
436   \seq_if_empty:NTF \l__bnvs_split_seq {
```

No new integer path component is added.

```

437   \cs_set:Npn \:nn ##1 ##2 {
438     \__bnvs_group_end:
439     \tl_set:Nn #1 { ##1 }
440     \tl_set:Nn #2 { ##2 }
441   }
442   \exp_args:NVV \:nn #1 #2
443 } {
```

Some new integer path components are added.

```

444   \cs_set:Npn \:nnn ##1 ##2 ##3 {
445     \__bnvs_group_end:
446     \tl_set:Nn #1 { ##1 }
447     \tl_set:Nn #2 { ##2 }
448     \seq_set_split:Nnn #3 . { ##3 }
449     \seq_remove_all:Nn #3 { }
450   }
451   \exp_args:NVVx
452   \:nnn #1 #2 {
453     \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
454   }
```

```

455 </!gubed>
456 % \end{gobble}
457 % \begin{macrocode}
458 }
459 \prg_return_true:
460 } {
461 \__bnvs_group_end:
462 \prg_return_false:
463 }
464 }

```

__bnvs_resolve:NNN \overline{TF} __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 $\backslash l_bnvs_id_tl$ 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 $\backslash l_a_seq$, $\backslash l_b_seq$.

```

465 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
466 #1 #2 #3 { T, F, TF } {
467 \__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.

```

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

```

```

519     \prg_return_true:
520   }
521   \loop:
522 }

```

```

\__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

```

523 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
524   #1 #2 #3 { T, F, TF } {
525   \__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.

```

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

```

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

```

540       \cs_set:Nn \loop: { \return_true: }
541     }
542   }
543 }
544 } {
545   \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
546   \get_extract:F {
547     \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
548       \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
549     }
550   }
551 }
552 \loop:
553 } {

```



```

554     \__bnvs_group_end:
555     \prg_return_false:
556   }
557 }
558 \cs_set:Npn \get_extract:F ##1 {
559   \exp_args:Nx
560   \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
561     \__bnvs_extract_key:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
562       \tl_set_eq:NN #2 \l__bnvs_b_tl
563       \seq_set_eq:NN #3 \l__bnvs_b_seq
564       \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
565       \seq_clear:N \l__bnvs_b_seq
566     } { ##1 }
567   } { ##1 }
568 }
569 \cs_set:Npn \return_true: {
570   \cs_set:Npn \:nnn #####1 #####2 #####3 {
571     \__bnvs_group_end:
572     \tl_set:Nn #1 { #####1 }
573     \tl_set:Nn #2 { #####2 }
574     \seq_set_split:Nnn #3 . { #####3 }
575     \seq_remove_all:Nn #3 { }
576   }
577   \exp_args:NVVx
578   \:nnn #1 #2 {
579     \seq_use:Nn #3 .
580   }
581   \prg_return_true:
582 }
583 \loop:
584 }

```

```

\__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.

```

585 \prg_new_conditional:Npnn \__bnvs_resolve:NNNN
586   #1 #2 #3 #4 { T, F, TF } {
587   #1 {
588     \__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.

```

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

```

If there is a length, no resolution occurs.

```

637   \branch:n { 1 }
638 } {
639   \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
640     \seq_clear:N \l__bnvs_b_seq

```

```

641         \tl_set:Nn \l__bnvs_a_tl { #3 . }
642         \tl_put_right:Nx \l__bnvs_a_tl {
643             \seq_use:Nn \l__bnvs_a_seq . .
644         }
645         \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_c_tl
646         \branch:FF {

```

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

```

647     \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
648     } {

```

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

```

649     \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
650     }
651     } {
652         \branch:FF {
653             \cs_set_eq:NN \loop: \return_true:
654         } {
655             \cs_set:Npn \loop: {
656                 \__bnvs_group_end:
657                 \prg_return_false:
658             }
659         }
660     }
661     } {
662         \cs_set:Npn \loop: {
663             \__bnvs_group_end:
664             \prg_return_false:
665         }
666     }
667     \loop:
668     }
669     \loop:
670     } {
671         \prg_return_true:
672     }
673     }
674     }
675     \prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
676     #1 #2 #3 #4 { T, F, TF } {
677     #1 {
678         \__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.

```

679     \tl_set_eq:NN \l__bnvs_a_tl #3
680     \seq_set_eq:NN \l__bnvs_a_seq #4
681     \cs_set:Npn \return_true: {
682         \cs_set:Npn \:nnn #####1 #####2 #####3 {
683             \__bnvs_group_end:
684             \tl_set:Nn #2 { #####1 }
685             \tl_set:Nn #3 { #####2 }
686             \seq_set_split:Nnn #4 . { #####3 }
687             \seq_remove_all:Nn #4 { }

```

```

688     }
689     \exp_args:NVVx
690     \:nnn #2 #3 {
691         \seq_use:Nn #4 .
692     }
693     \prg_return_true:
694 }
695 \cs_set:Npn \branch:n ##1 {
696     \seq_pop_left:NNTF \l__bnvs_a_seq \l__bnvs_b_tl {
697         \tl_put_right:Nn \l__bnvs_a_tl { . }
698         \tl_put_right:NV \l__bnvs_a_tl \l__bnvs_b_tl
699     } {
700         \cs_set_eq:NN \loop: \return_true:
701     }
702 }
703 \cs_set:Npn \loop: {
704     \__bnvs_call:TF {
705         \exp_args:Nx
706         \__bnvs_get:nNTF { \l__bnvs_a_tl / L } \l__bnvs_b_tl {
707             \branch:n { 1 }
708         } {
709             \exp_args:Nx
710             \__bnvs_get:nNTF { \l__bnvs_a_tl / A } \l__bnvs_b_tl {
711                 \__bnvs_extract_key:NNNTF #2 \l__bnvs_b_tl \l__bnvs_a_seq {
712                     \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_b_tl
713                     \tl_set_eq:NN #3 \l__bnvs_b_tl
714                     \seq_set_eq:NN #4 \l__bnvs_a_seq
715                 } {
716                     \branch:n { 2 }
717                 }
718             } {
719                 \branch:n { 3 }
720             }
721         }
722     } {
723         \cs_set:Npn \loop: {
724             \__bnvs_group_end:
725             \prg_return_false:
726         }
727     }
728     \loop:
729 }
730 \loop:
731 } {
732     \prg_return_true:
733 }
734 }

```

5.5.7 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.

```

735 \cs_new:Npn \__bnvs_fp_round:nN #1 #2 {
736   \tl_if_empty:nTF { #1 } {
737     {
738       \tl_put_right:Nx #2 {
739         \fp_eval:n { round(#1) }
740       }
741     }
742   }
743   \cs_generate_variant:Nn \__bnvs_fp_round:nN { VN, xN }
744   \cs_new:Npn \__bnvs_fp_round:N #1 {
745     \tl_if_empty:VTF #1 {
746       {
747         \tl_set:Nx #1 {
748           \fp_eval:n { round(#1) }
749         }
750       }
751     }

```

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

```

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

```

```

772  \__bnvs_if_in:nTF { #1//A } {
773      \tl_put_right:Nx #2 { \__bnvs_item:n { #1//A } }
774      \prg_return_true:
775  } {
776      \__bnvs_group_begin:
777      \tl_clear:N \l__bnvs_ans_tl
778      \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
779          \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
780              \__bnvs_return_true:nn { #1 } A #2
781          } {
782              \__bnvs_return_false:nn { #1 } A
783          }
784      } {
785          \__bnvs_get:nNTF { #1/L } \l__bnvs_a_tl {
786              \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {
787                  \__bnvs_if_append:xNTF {
788                      \l__bnvs_b_tl - ( \l__bnvs_a_tl ) + 1
789                  } \l__bnvs_ans_tl {
790                      \__bnvs_return_true:nn { #1 } A #2
791                  } {
792                      \__bnvs_return_false:nn { #1 } A
793                  }
794              } {
795                  \__bnvs_return_false:nn { #1 } A
796              }
797          } {
798              \__bnvs_return_false:nn { #1 } A
799          }
800      }
801  }
802 }
803 \prg_generate_conditional_variant:Nnn
804   \__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.

```

805 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
806     \__bnvs_raw_first:nNTF { #1 } #2 {
807         \prg_return_true:
808     } {
809         \__bnvs_get:nNTF { #1/C } \l__bnvs_a_tl {

```

```

810 \bool_set_true:N \l_no_counter_bool
811 \__bnvs_if_append:xNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
812 \__bnvs_return_true:nnN { #1 } A #2
813 } {
814 \__bnvs_return_false:nn { #1 } A
815 }
816 } {
817 \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
818 \__bnvs_gput:nn { #1/A } { 1 }
819 \tl_set:Nn #2 { 1 }
820 \__bnvs_return_true:nnN { #1 } A #2
821 } {
822 \__bnvs_return_false:nn { #1 } A
823 }
824 }
825 }
826 }

```

$\backslash_bnvs_first:nN$
 $\backslash_bnvs_first:VN$

$\backslash_bnvs_first:nN$ { $\langle name \rangle$ } $\langle tl\ variable \rangle$

Append the start of the $\langle name \rangle$ slide range to the $\langle tl\ variable \rangle$. Cache the result.

```

827 \cs_new:Npn \__bnvs_first:nN #1 #2 {
828 \__bnvs_if_first:nNF { #1 } #2 {
829 \msg_error:nnn { beanoves } { :n } { Range-with-no-first:~#1 }
830 }
831 }
832 \cs_generate_variant:Nn \__bnvs_first:nN { VN }

```

$\backslash_bnvs_raw_length:nNTF$

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

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

```

833 \prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
834 \__bnvs_if_in:nTF { #1//L } {
835 \tl_put_right:Nx #2 { \__bnvs_item:n { #1//L } }
836 \prg_return_true:
837 } {
838 \__bnvs_gput:nn { #1//L } { 0 }
839 \__bnvs_group_begin:
840 \tl_clear:N \l__bnvs_ans_tl
841 \__bnvs_if_in:nTF { #1/L } {
842 \__bnvs_if_append:xNTF {
843 \__bnvs_item:n { #1/L }
844 } \l__bnvs_ans_tl {
845 \__bnvs_return_true:nnN { #1 } L #2
846 } {
847 \__bnvs_return_false:nn { #1 } L
848 }
849 } {
850 \__bnvs_get:nNTF { #1/A } \l__bnvs_a_tl {
851 \__bnvs_get:nNTF { #1/Z } \l__bnvs_b_tl {

```

```

852     \__bnvs_if_append:xNTF {
853         \l__bnvs_b_tl = (\l__bnvs_a_tl) + 1
854     } \l__bnvs_ans_tl {
855         \__bnvs_return_true:nnN { #1 } L #2
856     } {
857         \__bnvs_return_false:nn { #1 } L
858     }
859 } {
860     \__bnvs_return_false:nn { #1 } L
861 }
862 } {
863     \__bnvs_return_false:nn { #1 } L
864 }
865 }
866 }
867 }
868 \prg_generate_conditional_variant:Nnn
869   \__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.

```

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

```



```

900     \_bnvs_return_false:nn { #1 } Z
901   }
902 }
903 }
904 }
905 \prg_generate_conditional_variant:Nnn
906   \_bnvs_raw_last:nN { VN } { T, F, TF }

```

_bnvs_last:nN
 _bnvs_last:VN

_bnvs_last:nN {<name>} <tl variable>
 Append the last index of the fully qualified <name> slide range to <tl variable>

```

907 \cs_new:Npn \_bnvs_last:nN #1 #2 {
908   \_bnvs_raw_last:nNF { #1 } #2 {
909     \msg_error:nnn { beanoves } { :n } { Range-with-no-last:~#1 }
910   }
911 }
912 \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.

```

913 \prg_new_conditional:Npnn \_bnvs_if_next:nN #1 #2 { T, F, TF } {
914   \_bnvs_if_in:nTF { #1//N } {
915     \tl_put_right:Nx #2 { \_bnvs_item:n { #1//N } }
916     \prg_return_true:
917   } {
918     \_bnvs_group_begin:
919     \cs_set:Npn \_bnvs_return_true: {
920       \tl_if_empty:NTF \l__bnvs_ans_tl {
921         \_bnvs_group_end:
922         \prg_return_false:
923       } {
924         \_bnvs_fp_round:N \l__bnvs_ans_tl
925         \_bnvs_gput:nV { #1//N } \l__bnvs_ans_tl
926         \exp_args:NNNV
927         \_bnvs_group_end:
928         \tl_put_right:Nn #2 \l__bnvs_ans_tl
929         \prg_return_true:
930       }
931     }
932     \cs_set:Npn \return_false: {
933       \_bnvs_group_end:
934       \prg_return_false:
935     }
936     \tl_clear:N \l__bnvs_a_tl
937     \_bnvs_raw_last:nNTF { #1 } \l__bnvs_a_tl {
938       \_bnvs_if_append:xNTF {
939         \l__bnvs_a_tl + 1
940       } \l__bnvs_ans_tl {
941         \_bnvs_return_true:
942       } {
943         \return_false:

```

```

944     }
945   } {
946     \return_false:
947   }
948 }
949 }
950 \prg_generate_conditional_variant:Nnn
951   \__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 $\langle tl \ variable \rangle$.

```

952 \cs_new:Npn \__bnvs_next:nN #1 #2 {
953   \__bnvs_if_next:nNF { #1 } #2 {
954     \msg_error:nnn { beanoves } { :n } { Range-with-no-next:~#1 }
955   }
956 }
957 \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.

```

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

```

973 \prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {

```

```

974 \bool_if:NTF \l__bnvs_no_range_bool {
975   \prg_return_false:
976 } {
977   \__bnvs_if_in:nTF { #1/ } {
978     \tl_put_right:Nn { 0-0 }
979   } {
980     \__bnvs_group_begin:
981     \tl_clear:N \l__bnvs_a_tl
982     \tl_clear:N \l__bnvs_b_tl
983     \tl_clear:N \l__bnvs_ans_tl
984     \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
985       \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
986         \exp_args:NNNx
987         \__bnvs_group_end:
988         \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
989         \prg_return_true:
990       } {
991         \exp_args:NNNx
992         \__bnvs_group_end:
993         \tl_put_right:Nn #2 { \l__bnvs_a_tl - }
994         \prg_return_true:
995       }
996     } {
997       \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
998         \exp_args:NNNx
999         \__bnvs_group_end:
1000         \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1001         \prg_return_true:
1002       } {
1003         \__bnvs_group_end:
1004         \prg_return_false:
1005       }
1006     }
1007   }
1008 }
1009 }
1010 \prg_generate_conditional_variant:Nnn
1011 \__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>.

```

1012 \cs_new:Npn \__bnvs_range:nN #1 #2 {
1013   \__bnvs_if_range:nNF { #1 } #2 {
1014     \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1015   }
1016 }
1017 \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:VNNTF code>}

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

```

1018 \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
1019   \__bnvs_group_begin:
1020   \tl_clear:N \l__bnvs_ans_tl
1021   \__bnvs_get:nNF { #1/C } \l__bnvs_ans_tl {
1022     \__bnvs_raw_first:nNF { #1 } \l__bnvs_ans_tl {
1023       \__bnvs_raw_last:nNF { #1 } \l__bnvs_ans_tl { }
1024     }
1025   }
1026   \tl_if_empty:NTF \l__bnvs_ans_tl {
1027     \__bnvs_group_end:
1028     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1029       \__bnvs_gput:nn { #1/C } { 1 }
1030       \tl_set:Nn #2 { 1 }
1031       \prg_return_true:
1032     } {
1033       \prg_return_false:
1034     }
1035   } {
1036     \__bnvs_gput:nV { #1/C } \l__bnvs_ans_tl
1037     \exp_args:NNNV
1038     \__bnvs_group_end:
1039     \tl_set:Nn #2 \l__bnvs_ans_tl
1040     \prg_return_true:
1041   }
1042 }
1043 \prg_generate_conditional_variant:Nnn
1044   \__bnvs_if_free_counter:nN { VN } { T, F, TF }

```

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

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

```

1045 \prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {
1046   \__bnvs_group_begin:
1047   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_ans_tl {
1048     \tl_clear:N \l__bnvs_a_tl
1049     \__bnvs_raw_first:nNT { #1 } \l__bnvs_a_tl {
1050       \fp_compare:nNnT { \l__bnvs_ans_tl } < { \l__bnvs_a_tl } {
1051         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1052       }
1053     }
1054     \tl_clear:N \l__bnvs_a_tl
1055     \__bnvs_raw_last:nNT { #1 } \l__bnvs_a_tl {
1056       \fp_compare:nNnT { \l__bnvs_ans_tl } > { \l__bnvs_a_tl } {
1057         \tl_set:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1058       }
1059     }
1060     \exp_args:NNV
1061     \__bnvs_group_end:
1062     \__bnvs_fp_round:nN \l__bnvs_ans_tl #2

```

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

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

```

1063     \prg_return_true:
1064   } {

1065     \prg_return_false:
1066   }
1067 }

1068 \prg_generate_conditional_variant:Nnn
1069   \__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.

```

1070 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {

1071   \__bnvs_group_begin:
1072   \tl_clear:N \l__bnvs_a_tl
1073   \__bnvs_if_free_counter:nNTF { #1 } \l__bnvs_a_tl {
1074     \tl_clear:N \l__bnvs_b_tl
1075     \__bnvs_if_append:xNTF { \l__bnvs_a_tl + (#2) } \l__bnvs_b_tl {
1076       \__bnvs_fp_round:N \l__bnvs_b_tl
1077       \__bnvs_gput:nV { #1/C } \l__bnvs_b_tl
1078     }
1079     \prg_return_true:
1080   } {
1081     \__bnvs_group_end:
1082     \prg_return_false:
1083   }
1084   } {
1085     \__bnvs_group_end:
1086     \prg_return_false:
1087   }
1088 }

1089 \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1090   \__bnvs_if_incr:nnTF { #1 } { #2 } {
1091     \__bnvs_if_counter:nNTF { #1 } #3 {
1092       \prg_return_true:
1093     } {
1094       \prg_return_false:
1095     }
1096   } {
1097     \prg_return_false:
1098   }
1099 }

1100 \prg_generate_conditional_variant:Nnn
1101   \__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>	<u>_bnvs_if_append:nNTF {⟨integer expression⟩} ⟨tl variable⟩ {⟨true code⟩} {⟨false code⟩}</u> Evaluates the ⟨integer expression⟩, replacing all the named specifications by their static counterpart then put the result to the right of the ⟨tl variable⟩. Executed within a group. Heavily used by _bnvs_eval_query:nN, where ⟨integer expression⟩ was initially enclosed in ‘?(...)’. Local variables:
\l__bnvs_ans_tl	To feed ⟨tl variable⟩ with. (End definition for \l__bnvs_ans_tl.)
\l__bnvs_split_seq	The sequence of caught query groups and non queries. (End definition for \l__bnvs_split_seq.)
\l__bnvs_split_int	Is the index of the non queries, before all the caught groups. (End definition for \l__bnvs_split_int.)
1102 \int_new:N \l__bnvs_split_int	
\l__bnvs_name_tl	Storage for \l_split_seq items that represent names. (End definition for \l__bnvs_name_tl.)
\l__bnvs_path_tl	Storage for \l_split_seq items that represent integer paths. (End definition for \l__bnvs_path_tl.)
	Catch circular definitions.
1103 \prg_new_conditional:Npnn _bnvs_if_append:nN #1 #2 { T, F, TF } {	
1104 _bnvs_call:TF {	
1105 _bnvs_group_begin:	
	Local variables:
1106 \int_zero:N \l__bnvs_split_int	
1107 \seq_clear:N \l__bnvs_split_seq	
1108 \tl_clear:N \l__bnvs_id_tl	
1109 \tl_clear:N \l__bnvs_name_tl	
1110 \tl_clear:N \l__bnvs_path_tl	
1111 \tl_clear:N \l__bnvs_group_tl	
1112 \tl_clear:N \l__bnvs_ans_tl	
1113 \tl_clear:N \l__bnvs_a_tl	
	Implementation:
1114 \regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq	
1115 \int_set:Nn \l__bnvs_split_int { 1 }	
1116 \tl_set:Nx \l__bnvs_ans_tl {	
1117 \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }	
1118 }	

`\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.

```

1119 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1120   \tl_set:Nx ##2 {
1121     \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1122   }
1123   \tl_if_empty:NTF ##2 {
1124     ##4 } {
1125     ##3
1126   }
1127 }

```

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

```

1128 \cs_set:Npn \return_true: {
1129   \fp_round:
1130   \exp_args:NNNV
1131   \__bnvs_group_end:
1132   \tl_put_right:Nn #2 \l__bnvs_ans_tl
1133   \prg_return_true:
1134 }
1135 \cs_set:Npn \fp_round: {
1136   \__bnvs_fp_round:N \l__bnvs_ans_tl
1137 }
1138 \cs_set:Npn \return_false: {
1139   \__bnvs_group_end:
1140   \prg_return_false:
1141 }
1142 \cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {
1143   \switch:nNTF { ##2 } \l__bnvs_id_tl { } {
1144     \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_current_tl
1145     \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1146   }
1147   \switch:nNTF { ##3 } \l__bnvs_path_tl {
1148     \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
1149     \seq_remove_all:Nn \l__bnvs_path_seq { }
1150   } {
1151     \seq_clear:N \l__bnvs_path_seq
1152   }
1153   ##1 \l__bnvs_id_tl \l__bnvs_name_tl \l__bnvs_path_seq {
1154     \cs_set:Npn \: {
1155       ##4
1156     }
1157   } {
1158     \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
1159   }
1160   \:
1161 }
1162 \cs_set:Npn \:T ##1 {

```

```

1163     \seq_if_empty:NNTF \l__bnvs_path_seq { ##1 } {
1164         \cs_set_eq:NN \loop: \return_false:
1165     }
1166 }

```

Main loop.

```

1167     \cs_set:Npn \loop: {
1168         \int_compare:nNnTF {
1169             \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1170         } {
1171             \switch:nNTF 1 \l__bnvs_name_tl {

```

- Case ++ $\langle name \rangle \langle integer\ path \rangle .n$.

```

1172         \:NnnT \__bnvs_resolve_n:NNNTF 2 3 {
1173             \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1174                 \cs_set_eq:NN \loop: \return_false:
1175             }
1176         }
1177     } {
1178         \switch:nNTF 4 \l__bnvs_name_tl {

```

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

```

1179         \switch:nNTF 7 \l__bnvs_a_tl {
1180             \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1181                 \:T {
1182                     \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1183                         \cs_set_eq:NN \loop: \return_false:
1184                     }
1185                 }
1186             }

```

- Case ...length.

```

1187     } {
1188         \switch:nNTF 8 \l__bnvs_a_tl {

```

- Case ...last.

```

1189         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1190             \:T {
1191                 \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1192                     \cs_set_eq:NN \loop: \return_false:
1193                 }
1194             }
1195         }

1196     } {
1197         \switch:nNTF 9 \l__bnvs_a_tl {

```


- Case ...next.

```

1198         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1199             \:T {
1200                 \__bnvs_if_next:VNF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1201                     \cs_set_eq:NN \loop: \return_false:
1202                 }
1203             }
1204         }
1205     } {
1206         \switch:nNTF { 10 } \l__bnvs_a_t1 {

```

- Case ...range.

```

1207 \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1208     \:T {
1209         \__bnvs_if_range:VNTF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1210             \cs_set_eq:NN \fp_round: \prg_do_nothing:
1211         } {
1212             \cs_set_eq:NN \loop: \return_false:
1213         }
1214     }
1215 }
1216     } {
1217         \switch:nNTF { 11 } \l__bnvs_a_t1 {

```

- Case ...n.

```

1218         \switch:nNTF { 12 } \l__bnvs_a_t1 {

```

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

```

1219 \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1220     \:T {
1221         \__bnvs_if_incr:VVNF \l__bnvs_name_t1 \l__bnvs_a_t1 \l__bnvs_ans_t1 {
1222             \cs_set_eq:NN \loop: \return_false:
1223         }
1224     }
1225 }
1226     } {
1227         \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1228             \seq_if_empty:NTF \l__bnvs_path_seq {
1229                 \__bnvs_if_counter:VNF \l__bnvs_name_t1 \l__bnvs_ans_t1 {
1230                     \cs_set_eq:NN \loop: \return_false:
1231                 }
1232             } {
1233                 \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_t1
1234                 \seq_if_empty:NTF \l__bnvs_path_seq {
1235                     \__bnvs_if_incr:VVNF \l__bnvs_name_t1 \l__bnvs_a_t1 \l__bnvs_ans_t1 {
1236                         \cs_set_eq:NN \loop: \return_false:
1237                     }
1238                 } {
1239                     \msg_error:nmx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1240                     \cs_set_eq:NN \loop: \return_false:
1241                 }

```

```

1242         }
1243     }
1244 }

1245     } {
1246         \:NnnT \__bnvs_resolve_n:NNNTF 5 6 {
1247             \seq_if_empty:NTF \l__bnvs_path_seq {
1248 \__bnvs_if_counter:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1249     \cs_set_eq:NN \loop: \return_false:
1250 }
1251         } {
1252             \seq_pop_left:NN \l__bnvs_path_seq \l__bnvs_a_tl
1253             \seq_if_empty:NTF \l__bnvs_path_seq {
1254 \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1255     \cs_set_eq:NN \loop: \return_false:
1256 }
1257         } {
1258 \msg_error:nxx { beanoves } { :n } { Too-many~.<integer>~components:~#1 }
1259 \cs_set_eq:NN \loop: \return_false:
1260         }
1261     }
1262 }
1263 }
1264 }
1265 }
1266 }
1267 }
1268 } {

```

No name.

```

1269     }
1270 }

1271     \int_add:Nn \l__bnvs_split_int { 13 }
1272     \tl_put_right:Nx \l__bnvs_ans_tl {
1273         \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1274     }

1275     \loop:
1276 } {

1277     \return_true:
1278 }
1279 }
1280 \loop:
1281 } {
1282     \msg_error:nxx { beanoves } { :n } { Too-many~calls:~ #1 }
1283     \prg_return_false:
1284 }
1285 }
1286 \prg_generate_conditional_variant:Nnn
1287 \__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> 1288 \regex_const:Nn \c__bnvs_A_cln_Z_regex { 1289 \A \s* (? 1290 1291 • 2: <first> 1292 ([^:]*) \s* : 1293 1294 • 3: second optional colon 1295 (:)? \s* 1296 1297 • 4: <length> 1298 ([^:]*) 1299 1300 • 5: standalone <first> 1301 ([^:]+) 1302) \s* \Z 1303 }</pre> <pre> 1296 \prg_new_conditional:Npnn _bnvs_if_eval_query:nN #1 #2 { T, F, TF } { 1297 _bnvs_call_reset: 1298 \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex { 1299 #1 1300 } \l__bnvs_match_seq { 1301 \bool_set_false:N \l__bnvs_no_counter_bool 1302 \bool_set_false:N \l__bnvs_no_range_bool </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> 1303 \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 { </pre>

```

1304     \tl_set:Nx ##2 {
1305       \seq_item:Nn \l__bnvs_match_seq { ##1 }
1306     }
1307     \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1308   }
1309   \switch:nNTF 5 \l__bnvs_a_tl {

```

Single expression

```

1310     \bool_set_false:N \l__bnvs_no_range_bool
1311     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1312       \prg_return_true:
1313     } {
1314       \prg_return_false:
1315     }
1316   } {
1317     \switch:nNTF 2 \l__bnvs_a_tl {
1318       \switch:nNTF 4 \l__bnvs_b_tl {
1319         \switch:nNTF 3 \l__bnvs_c_tl {

```

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

```

1320     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1321       \tl_put_right:Nn #2 { - }
1322       \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1323         \prg_return_true:
1324       } {
1325         \prg_return_false:
1326       }
1327     } {
1328       \prg_return_false:
1329     }
1330   } {

```

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

```

1331     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1332       \tl_put_right:Nx #2 { - }
1333       \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1 }
1334       \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1335         \prg_return_true:
1336       } {
1337         \prg_return_false:
1338       }
1339     } {
1340       \prg_return_false:
1341     }
1342   }
1343   } {

```

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

```

1344     \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1345       \tl_put_right:Nn #2 { - }
1346       \prg_return_true:
1347     } {
1348       \prg_return_false:
1349     }
1350   }

```

```

1351     } {
1352         \switch:nNTF 4 \l__bnvs_b_tl {
1353             \switch:nNTF 3 \l__bnvs_c_tl {
1354                 \tl_put_right:Nn #2 { - }
1355                 \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1356                     \prg_return_true:
1357                 } {
1358                     \prg_return_false:
1359                 }
1360             } {
1361 \msg_error:nnx { beanoves } { :n } { Syntax~error(Missing~first):~#1 }
1362             }
1363         } {
1364             : or :: range
1365             \seq_put_right:Nn #2 { - }
1366         }
1367     }
1368 } {
Error
1369 \msg_error:nnn { beanoves } { :n } { Syntax~error:~#1 }
1370 }
1371 }

```

__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.

```

1372 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }

```

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

```

1373 \cs_new:Npn \__bnvs_eval:nN #1 #2 {
1374     \__bnvs_group_begin:

```

Local variables declaration

```

1375     \seq_clear:N \l__bnvs_query_seq
1376     \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 $\backslash l_query_seq$.

```
1377 \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.

```
1378 \seq_map_inline:Nn \l__bnvs_query_seq {
1379   \tl_clear:N \l__bnvs_ans_tl
1380   \__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {
1381     \seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl
1382   } {
1383     \seq_map_break:n {
1384       \msg_fatal:nnn { beanoves } { :n } { Circular-dependency-in~#1}
1385     }
1386   }
1387 }
```

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

```
1388 \exp_args:NNNx
1389 \__bnvs_group_end:
1390 \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
1391 }
1392 \cs_generate_variant:Nn \__bnvs_eval:nN { VN, xN }
```

$\backslash BeanovesEval$ $\backslash BeanovesEval$ [$\langle tl\ variable \rangle$] [$\langle overlay\ queries \rangle$]

$\langle overlay\ queries \rangle$ 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.

```
1393 \NewDocumentCommand \BeanovesEval { s o m } {
1394   \__bnvs_group_begin:
1395   \tl_clear:N \l__bnvs_ans_tl
1396   \IfBooleanTF { #1 } {
1397     \bool_set_true:N \l__bnvs_no_counter_bool
1398   } {
1399     \bool_set_false:N \l__bnvs_no_counter_bool
1400   }
1401   \__bnvs_eval:nN { #3 } \l__bnvs_ans_tl
1402   \IfValueTF { #2 } {
1403     \exp_args:NNNV
1404     \__bnvs_group_end:
1405     \tl_set:Nn #2 \l__bnvs_ans_tl
1406   } {
1407     \exp_args:NV
1408     \__bnvs_group_end: \l__bnvs_ans_tl
1409   }
1410 }
```

5.5.9 Resetting slide ranges

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

```

1411 \NewDocumentCommand \BeanovesReset { 0{1} m } {
1412   \__bnvs_reset:nn { #1 } { #2 }
1413   \ignorespaces
1414 }

```

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.

```

1415 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1416   \bool_if:nTF {
1417     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1418   } {
1419     \__bnvs_gremove:n { #2/C }
1420     \__bnvs_gremove:n { #2//A }
1421     \__bnvs_gremove:n { #2//L }
1422     \__bnvs_gremove:n { #2//Z }
1423     \__bnvs_gremove:n { #2//N }
1424     \__bnvs_gput:nn { #2/CO } { #1 }
1425   } {
1426     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1427   }
1428 }

1429 \makeatother
1430 \ExplSyntaxOff
1431 \</package>

```