

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	1
2	Named slide lists	2
2.1	Presentation	2
2.2	Defining named slide lists	2
3	Named overlay specifications	3
3.1	Named slide ranges	3
3.2	Named slide lists	4
4	?(...) query expressions	4
5	Implementation	5
5.1	Package declarations	5
5.2	logging and debugging facilities	5
5.3	Local variables	6
5.4	Infinite loop management	6
5.5	Overlay specification	7
5.5.1	In slide range definitions	7
5.5.2	Regular expressions	9
5.5.3	beamer.cls interface	12
5.5.4	Defining named slide ranges	12
5.5.5	Scanning named overlay specifications	17
5.5.6	Resolution	21
5.5.7	Evaluation bricks	30
5.5.8	Evaluation	40
5.5.9	Reseting slide ranges	50

1 Minimal example

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

```
1 \documentclass {beamer}
2 \RequirePackage {beanoves}
3 \begin{document}
4 \Beanoves {
5     A = 1:2,
6     B = A.next:3,
7     C = B.next,
8 }
9 \begin{frame}
10 {\Large Frame \insertframenumber}
11 {\Large Slide \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* `?(A.1)` stands for 1, on line 15, `?(A.2)` stands for 2 whereas on line 18, `?(A.3)` 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 `?(B.last)` 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 = {
  <name1>=<spec1>,
  <name2>=<spec2>,
  ...,
  <namen>=<specn>,
}

```

```

\Beanoves{
  <name1>=<spec1>,
  <name2>=<spec2>,
  ...,
  <namen>=<specn>,
}

```

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
$\langle name \rangle.range$	$i \text{ ' '-'' } j$	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

¹This is actually an experimental feature.

$\langle name \rangle.n+=\langle integer \rangle$: 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 ‘? $\langle queries \rangle$ ’. 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 $\langle name \rangle.range$

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 $?(\mathbf{A}.next)$, $?(\mathbf{A}.last+1)$, $?(\mathbf{A}.1+\mathbf{A}.length)$ 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.

¹ $\langle *package \rangle$

5 Implementation

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

² $\langle @@=bnvs \rangle$

5.1 Package declarations

```

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

```

5.2 logging and debugging facilities

Utility message.

```

9 \msg_new:nnn { beanoves } { :n } { #1 }
10 \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
11 \cs_set:Npn \__bnvs_DEBUG_:nn #1 #2 {
12   \msg_term:nnn { beanoves } { :n } { #1~#2 }
13 }
14 \cs_new:Npn \__bnvs_DEBUG_on: {
15   \cs_set:Npn \__bnvs_DEBUG:n {
16     \exp_args:Nx
17     \__bnvs_DEBUG_:nn
18     { \prg_replicate:nn {\l__bnvs_group_int} { } \space }
19   }
20 }
21 \cs_new:Npn \__bnvs_DEBUG_off: {
22   \cs_set_eq:NN \__bnvs_DEBUG:n \use_none:n
23 }
24 \__bnvs_DEBUG_off:
25 \cs_generate_variant:Nn \__bnvs_DEBUG:n { x, V }
26 \int_zero_new:N \l__bnvs_group_int
27 \cs_set:Npn \__bnvs_group_begin: {
28   \group_begin:
29   \int_incr:N \l__bnvs_group_int
30 }
31 \cs_set:Npn \__bnvs_group_end: {
32   \group_end:
33 }
34 \cs_new:Npn \__bnvs_LOG:nn #1 #2 {
35   \__bnvs_DEBUG:x { #1~#2 }
36 }
37 \cs_new:Npn \__bnvs_DEBUG:nn #1 {
38   \exp_args:Nx
39   \__bnvs_LOG:nn
40   { \prg_replicate:nn {\l__bnvs_group_int + 1} {#1} }
41 }
42 \cs_generate_variant:Nn \__bnvs_DEBUG:nn { nx, nV }

```

5.3 Local variables

We make heavy use of local variables and function scopes. Many functions are executed within a \TeX group, which ensures no name collision with the caller stack. In that case, variables need not follow exactly the \LaTeX 3 naming convention: we do not specialize with the module name. On execution, next initialization instructions declare the variables as side effect.

```

43 \int_if_exist:NF \l_depth_int {
44   \int_new:N \l_depth_int
45 }
46 \bool_new:N \l__bnvs_no_counter_bool
47 \bool_new:N \l__bnvs_no_range_bool
48 \bool_new:N \l__bnvs_continue_bool
49 \bool_new:N \l__bnvs_in_frame_bool
50 \bool_set_false:N \l__bnvs_in_frame_bool

```

5.4 Infinite loop management

Unending recursivity is managed here.

`\g__bnvs_call_int`

```

51 \int_zero_new:N \g__bnvs_call_int
52 \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.

```

53 \cs_set:Npn \__bnvs_call_reset: {
54   \int_gset:Nn \g__bnvs_call_int { \c__bnvs_max_call_int }
55 }

```

`__bnvs_call:TF`

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

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

```

56 \prg_new_conditional:Npnn \__bnvs_call: { T, F, TF } {
57   \int_gdecr:N \g__bnvs_call_int
58   \int_compare:nNnTF \g__bnvs_call_int > 0 {
59     \prg_return_true:
60   } {
61     \prg_return_false:
62   }
63 }

```

5.5 Overlay specification

5.5.1 In slide range definitions

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

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

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

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

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

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

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

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

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

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

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

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

64 `\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.

```

65 \cs_new:Npn \__bnvs_gput:nn #1 #2 {
66   \__bnvs_DEBUG:x {\string\__bnvs_gput:nn/key:#1/value:#2/}
67   \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
68 }
69 \cs_new:Npn \__bnvs_gprovide:nn #1 #2 {
70   \__bnvs_DEBUG:x {\string\__bnvs_gprovide:nn/key:#1/value:#2/}
71   \prop_if_in:NnF \g__bnvs_prop { #1 } {
72     \prop_gput:Nnn \g__bnvs_prop { #1 } { #2 }
73   }
74 }
75 \cs_new:Npn \__bnvs_item:n {
76   \prop_item:Nn \g__bnvs_prop
77 }
78 \cs_new:Npn \__bnvs_get:nN {
79   \prop_get:NnN \g__bnvs_prop
80 }
81 \cs_new:Npn \__bnvs_gremove:n {
82   \prop_gremove:Nn \g__bnvs_prop
83 }
84 \cs_new:Npn \__bnvs_gclear:n #1 {
85   \clist_map_inline:nn { A, L, Z, C, CO, /, /A, /L, /Z, /N } {
86     \__bnvs_gremove:n { #1 / ##1 }
87   }
88 }
89 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
90   \clist_map_inline:nn { /A, /L, /Z, /N } {
91     \__bnvs_gremove:n { #1 / ##1 }
92   }
93 }
94 \cs_new:Npn \__bnvs_gclear: {
95   \prop_gclear:N \g__bnvs_prop
96 }
97 \cs_generate_variant:Nn \__bnvs_gput:nn { nV }
98 \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.

```

99 \prg_new_conditional:Npnn \__bnvs_if_in:n #1 { p, T, F, TF } {
100   \prop_if_in:NnTF \g__bnvs_prop { #1 } {

```

```

101     \prg_return_true:
102   } {
103     \prg_return_false:
104   }
105 }
106 \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.

```

107 \prg_new_conditional:Npnn \__bnvs_get:nN #1 #2 { T, F, TF } {
108   \prop_get:NnNTF \g__bnvs_prop { #1 } #2 {
109     \__bnvs_DEBUG:x { \string\__bnvs_get:nN\space TRUE/
110       #1/\string#2:#2/ }
111     \prg_return_true:
112   } {
113     \__bnvs_DEBUG:x { \string\__bnvs_get:nN\space FALSE/#1/\string#2/ }
114     \prg_return_false:
115   }
116 }

```

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.

```

117 \regex_const:Nn \c__bnvs_name_regex {
118   [[:alpha:]]_ [[:alnum:]]_*
119 }

```

(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.

```

120 \regex_const:Nn \c__bnvs_id_regex {
121   (?: \ur{c__bnvs_name_regex} | [?]* ) ? !
122 }

```

(End definition for `\c__bnvs_id_regex`.)

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

```

123 \regex_const:Nn \c__bnvs_path_regex {
124   (?: \. [+-]? \d+ ) *
125 }

```

(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
`\c__bnvs_A_key_Z_regex` a dot syntax. The ‘A_key_Z’ variant matches the whole string.

```

126 \regex_const:Nn \c__bnvs_key_regex {
127   \ur{c__bnvs_id_regex} ?
128   \ur{c__bnvs_name_regex}
129   \ur{c__bnvs_path_regex}
130 }
131 \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

```

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

```

5: the path, if any.

```

133   ( \ur{c__bnvs_path_regex} ) \Z
134 }
135

```

(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.

```

136 \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.

```

137 \regex_const:Nn \c__bnvs_list_regex {
138   \A \[ \s*

```

Capture groups:

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

```

139   ( [^\] %[-~
140   ]*? )
141   \s* \] \Z
142 }

```

(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.

```

143 \regex_const:Nn \c__bnvs_split_regex {
144   \s* ( ? :

```

We start with ‘++’ instructions ².

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

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

145 $\backslash+ \backslash+ ((\backslash\{c_bnvs_id_regex\}?) \backslash\{c_bnvs_name_regex\})$

- 3: optionally followed by an integer path

146 $(\backslash\{c_bnvs_path_regex\}) (?: \backslash. n)?$

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 $\langle id \rangle$)

147 $| ((\backslash\{c_bnvs_id_regex\}?) \backslash\{c_bnvs_name_regex\})$

- 6: optionally followed by an integer path

148 $(\backslash\{c_bnvs_path_regex\})$

Next comes another branching

149 $(?:$

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

150 $\backslash. l(e)ngth$

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

151 $| \backslash. l(a)st$

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

152 $| \backslash. ne(x)t$

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

153 $| \backslash. (r)ange$

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

154 $| \backslash. (n)$

• 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.

155 $(?: \backslashs* \ += \ \backslashs* (\backslashS+))?$

156 $)?$

157 $) \backslashs*$

158 $\}$

(End definition for $\backslash c_bnvs_split_regex$.)

5.5.3 beamer.cls interface

Work in progress.

```

159 \RequirePackage{keyval}
160 \define@key{beamerframe}{beanoves~id}[] {
161   \tl_set:Nx \l__bnvs_id_tl { #1 ! }
162   \__bnvs_DEBUG_on:
163   \__bnvs_DEBUG:x {THIS_IS_KEY}
164   \__bnvs_DEBUG_off:
165 }
166 \AddToHook{env/beamer@frameslide/before}{
167   \bool_set_true:N \l__bnvs_in_frame_bool
168   \__bnvs_DEBUG_on:
169   \__bnvs_DEBUG:x {THIS_IS_BEFORE}
170   \__bnvs_DEBUG_off:
171 }
172 \AddToHook{env/beamer@frameslide/after}{
173   \bool_set_false:N \l__bnvs_in_frame_bool
174   \__bnvs_DEBUG_on:
175   \__bnvs_DEBUG:x {THIS_IS_BEFORE}
176   \__bnvs_DEBUG_off:
177 }
178 \AddToHook{cmd/frame/before}{
179   \tl_set:Nn \l__bnvs_id_tl { ?! }
180   \__bnvs_DEBUG_on:
181   \__bnvs_DEBUG:x {THIS_IS_FRAME}
182   \__bnvs_DEBUG_off:
183 }

```

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_match_seq` Local storage for the match result.

(End definition for `\l_match_seq`. This variable is documented on page ??.)

<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 function `__bnvs_range:Nnnnn` which first argument is `__bnvs_gput:nn` for `__bnvs_range:nnnn` and `__bnvs_gprovide:nn` for `__bnvs_range_alt:nnnn`.

```

184 \cs_new:Npn \__bnvs_range:Nnnnn #1 #2 #3 #4 #5 {
185   \__bnvs_DEBUG:x {\string\__bnvs_range:Nnnnn/\string#1/#2/#3/#4/#5/}
186   \tl_if_empty:nTF { #3 } {

```

```

187 \tl_if_empty:nTF { #4 } {
188 \tl_if_empty:nTF { #5 } {
189 \msg_error:nnn { beanoves } { :n } { Not~a~range::~~#2 }
190 } {
191 #1 { #2/Z } { #5 }
192 }
193 } {
194 #1 { #2/L } { #4 }
195 \tl_if_empty:nF { #5 } {
196 #1 { #2/Z } { #5 }
197 #1 { #2/A } { #2.last - (#2.length) + 1 }
198 }
199 }
200 } {
201 #1 { #2/A } { #3 }
202 \tl_if_empty:nTF { #4 } {
203 \tl_if_empty:nF { #5 } {
204 #1 { #2/Z } { #5 }
205 #1 { #2/L } { #2.last - (#2.1) + 1 }
206 }
207 } {
208 #1 { #2/L } { #4 }
209 #1 { #2/Z } { #2.1 + #2.length - 1 }
210 }
211 }
212 }
213 \cs_new:Npn \__bnvs_range:nnnn #1 {
214 \__bnvs_gclear:n { #1 }
215 \__bnvs_range:Nnnnn \__bnvs_gput:nn { #1 }
216 }
217 \cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
218 \cs_new:Npn \__bnvs_range_alt:nnnn #1 {
219 \__bnvs_gclear_cache:n { #1 }
220 \__bnvs_range:Nnnnn \__bnvs_gprovide:nn { #1 }
221 }
222 \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.

```

223 \cs_new:Npn \__bnvs_parse:Nn #1 #2 {
224 \__bnvs_group_begin:
225 \__bnvs_id_name_set:nNNTF { #2 } \l_id_tl \l_name_tl {
226 \exp_args:Nx \__bnvs_gput:nn { \l_name_tl/ } { }
227 \exp_args:NNNV
228 \__bnvs_group_end:
229 \tl_set:Nn \l__bnvs_id_tl \l__bnvs_id_tl
230 } {
231 \msg_error:nnn { beanoves } { :n } { Unexpected-key::~~#2 }
232 \__bnvs_group_end:
233 }
234 }

```

```

__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`.

```

235 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
236 \cs_new:Npn \__bnvs_do_parse:Nnn #1 #2 #3 {
237 \__bnvs_DEBUG:x {\string\__bnvs_do_parse:Nnn/\string#1/#2/#3}

```

This is not a list.

```

238 \tl_clear:N \l_a_tl
239 \tl_clear:N \l_b_tl
240 \tl_clear:N \l_c_tl
241 \regex_split:NnN \c__bnvs_colons_regex { #3 } \l_split_seq
242 \seq_pop_left:NNT \l_split_seq \l_a_tl {

```

`\l_a_tl` may contain the `<start>`.

```

243 \seq_pop_left:NNT \l_split_seq \l_b_tl {
244 \tl_if_empty:NNT \l_b_tl {

```

This is a one colon range.

```

245 \seq_pop_left:NN \l_split_seq \l_b_tl

```

`\l_b_tl` may contain the `<length>`.

```

246 \seq_pop_left:NNT \l_split_seq \l_c_tl {
247 \tl_if_empty:NNT \l_c_tl {

```

A `::` was expected:

```

248 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(1):~#3 }
249 } {
250 \int_compare:nNnT { \tl_count:N \l_c_tl } > { 1 } {
251 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(2):~#3 }
252 }
253 \seq_pop_left:NN \l_split_seq \l_c_tl

```

`\l_c_tl` may contain the `<end>`.

```

254 \seq_if_empty:NF \l_split_seq {
255 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(3):~#3 }
256 }
257 }
258 }
259 } {

```

This is a two colon range.

```

260 \int_compare:nNnT { \tl_count:N \l_b_tl } > { 1 } {
261 \msg_error:nnn { beanoves } { :n } { Invalid~range~expression(4):~#3 }
262 }
263 \seq_pop_left:NN \l_split_seq \l_c_tl

```

`\l_c_tl` contains the `<end>`.

```

264 \seq_pop_left:NNTF \l_split_seq \l_b_tl {
265 \tl_if_empty:NNTF \l_b_tl {
266 \seq_pop_left:NN \l_split_seq \l_b_tl

```

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

```

267         \seq_if_empty:NF \l_split_seq {
268 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(5):~#3 }
269     }
270     } {
271 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(6):~#3 }
272     }
273     } {
274         \tl_clear:N \l_b_tl
275     }
276     }
277 }
278 }

```

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.

```

279 \bool_if:nF {
280     \tl_if_empty_p:N \l_a_tl
281     || \tl_if_empty_p:N \l_b_tl
282     || \tl_if_empty_p:N \l_c_tl
283 } {
284 \msg_error:nnn { beanoves } { :n } { Invalid~range-expression(7):~#3 }
285 }
286 #1 { #2 } \l_a_tl \l_b_tl \l_c_tl
287 }
288 \cs_generate_variant:Nn \__bnvs_do_parse:Nnn { Nxn, Non }

289 \cs_new:Npn \__bnvs_parse_old:Nnn #1 #2 #3 {
290     \__bnvs_group_begin:
291     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #2 } {

```

We got a valid key.

```

292     \regex_extract_once:NnNTF \c__bnvs_list_regex { #3 } \l_match_seq {

```

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

```

293         \exp_args:NNx
294         \seq_set_from_clist:Nn \l_match_seq {
295             \seq_item:Nn \l_match_seq { 2 }
296         }
297         \seq_map_indexed_inline:Nn \l_match_seq {
298             \__bnvs_do_parse:Nnn #1 { #2.##1 } { ##2 }
299         }
300     } {
301         \__bnvs_do_parse:Nnn #1 { #2 } { #3 }
302     }
303 } {
304     \msg_error:nnn { beanoves } { :n } { Invalid~key:~#1 }
305 }
306 \__bnvs_group_end:
307 }

```

```

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

```

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

```

308 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
309   \__bnvs_group_begin:
310   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex { #1 } \l_match_seq {
311     \tl_set:Nx #2 { \seq_item:Nn \l_match_seq 3 }
312     \tl_if_empty:NTF #2 {
313       \exp_args:NNNx
314       \__bnvs_group_end:
315       \tl_set:Nn #3 { \l__bnvs_id_tl #1 }
316       \tl_set_eq:NN #2 \l__bnvs_id_tl
317     } {
318       \cs_set:Npn \:n ##1 {
319         \__bnvs_group_end:
320         \tl_set:Nn #2 { ##1 }
321         \tl_set:Nn \l__bnvs_id_tl { ##1 }
322       }
323       \exp_args:NV
324       \:n #2
325       \tl_set:Nn #3 { #1 }
326     }
327   \__bnvs_DEBUG:x { \string\__bnvs_id_name_set:nNN\space TRUE/#1/
328     \string#2:#2/\string#3:#3/\string\l__bnvs_id_tl:\l__bnvs_id_tl/ }
329   \prg_return_true:
330 } {
331   \__bnvs_group_end:
332   \__bnvs_DEBUG:x { \string\__bnvs_id_name_set:nNN\space FALSE/#1/
333     \string#2/\string#3/ }
334   \prg_return_false:
335 }
336 }

337 \cs_new:Npn \__bnvs_parse:Nnn #1 #2 #3 {
338   \__bnvs_DEBUG:x { \string\__bnvs_parse:Nnn/\string#1/#2/#3/ }
339   \__bnvs_group_begin:
340   \__bnvs_id_name_set:nNNTF { #2 } \l_id_tl \l_name_tl {
341     \__bnvs_DEBUG:x {key:#2/ID:\l_id_tl/NAME:\l_name_tl/}
342     \regex_extract_once:NnNTF \c__bnvs_list_regex { #3 } \l_match_seq {

```

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

```

343     \exp_args:NNx
344     \seq_set_from_clist:Nn \l_match_seq {
345       \seq_item:Nn \l_match_seq { 2 }
346     }
347     \seq_map_indexed_inline:Nn \l_match_seq {
348       \__bnvs_do_parse:Nxn #1 { \l_name_tl.##1 } { ##2 }
349     }
350   } {
351     \__bnvs_do_parse:Nxn #1 { \l_name_tl } { #3 }
352   }

```

```

353 } {
354   \msg_error:nnn { beanoves } { :n } { Invalid-key:~#2 }
355 }

```

We export `\l__bnvs_id_tl`:

```

356 \exp_args:NNNV
357 \__bnvs_group_end:
358 \tl_set:Nn \l__bnvs_id_tl \l__bnvs_id_tl
359 }

```

\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`.

```

360 \NewDocumentCommand \Beanoves { sm } {
361   \tl_if_eq:NnT \@currentenvir { document } {
362     \__bnvs_gclear:
363   }
364   \IfBooleanTF {#1} {
365     \keyval_parse:nnn {
366       \__bnvs_parse:Nn \__bnvs_range_alt:nVVV
367     } {
368       \__bnvs_parse:Nnn \__bnvs_range_alt:nVVV
369     }
370   } {
371     \keyval_parse:nnn {
372       \__bnvs_parse:Nn \__bnvs_range:nVVV
373     } {
374       \__bnvs_parse:Nnn \__bnvs_range:nVVV
375     }
376   }
377   { #2 }
378   \ignorespaces
379 }

```

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

```

380 \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 {<overlay specification>}</code> <code>\beamer@masterdecode {<overlay specification>}</code>
---	---

Preprocess `<overlay specification>` before `beamer` uses it.

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

(End definition for `\l_ans_tl`. This variable is documented on page ??.)

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

```

381 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
382 \cs_set:Npn \beamer@frame < #1 > {
383   \__bnvs_group_begin:
384   \tl_clear:N \l_ans_tl
385   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l_ans_tl
386   \exp_args:NNNV
387   \__bnvs_group_end:
388   \__bnvs_beamer@frame < \l_ans_tl >
389 }
390 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
391 \cs_set:Npn \beamer@masterdecode #1 {
392   \__bnvs_group_begin:
393   \tl_clear:N \l_ans_tl
394   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l_ans_tl
395   \exp_args:NNV
396   \__bnvs_group_end:
397   \__bnvs_beamer@masterdecode \l_ans_tl
398 }

```

__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_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_depth_int. This variable is documented on page ??.)

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

(End definition for \l_query_tl. This variable is documented on page ??.)

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

(End definition for \l_token_seq. This variable is documented on page ??.)

\l_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_ask_bool. This variable is documented on page ??.)

\l_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_query_bool. This variable is documented on page ??.)

\l_token_tl Storage for just one token.

(End definition for \l_token_tl. This variable is documented on page ??.)

```

399 \cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 {
400   \__bnvs_group_begin:
401   \tl_clear:N \l_ans_tl
402   \int_zero:N \l_depth_int
403   \seq_clear:N \l_token_seq

```

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

```

404   \regex_split:nnN {} { #1 } \l_token_seq

```

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

```

405   \bool_set_true:N \l_ask_bool
406   \bool_while_do:Nn \l_ask_bool {
407     \seq_pop_left:NN \l_token_seq \l_token_tl
408     \quark_if_no_value:NTF \l_token_tl {

```

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

```

409       \bool_set_false:N \l_ask_bool
410     } {

```

`\l_token_tl` contains a ‘normal’ token.

```

411       \tl_if_eq:NnTF \l_token_tl { ? } {

```

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

```

412         \bool_set_true:N \l_query_bool
413         \bool_while_do:Nn \l_query_bool {

```

Get next token.

```

414           \seq_pop_left:NN \l_token_seq \l_token_tl
415           \quark_if_no_value:NTF \l_token_tl {

```

No opening parenthesis found, raise.

```

416             \msg_fatal:nmx { beanoves } { :n } {Missing~'('%---)
417             ~after~a~?:~#1}
418           } {
419             \tl_if_eq:NnT \l_token_tl { ( %}
420           } {

```

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

```

421             \int_incr:N \l_depth_int

```

Record the forthcoming content in the `\l_query_tl` variable, up to the next balancing ‘)’:

```

422             \tl_clear:N \l_query_tl
423             \bool_while_do:Nn \l_query_bool {

```

Get next token.

```

424               \seq_pop_left:NN \l_token_seq \l_token_tl
425               \quark_if_no_value:NTF \l_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_depth_int` is 0 whenever `\l_query_bool` is false.

```

426               \msg_error:nmx { beanoves } { :n } {Missing~%((---
427               ~)'':~#1 }
428             \int_do_while:nNnn \l_depth_int > 1 {
429               \int_decr:N \l_depth_int
430               \tl_put_right:Nn \l_query_tl {%(---

```

```

431         })
432     }
433     \int_zero:N \l_depth_int
434     \bool_set_false:N \l_query_bool
435     \bool_set_false:N \l_ask_bool
436 } {
437     \tl_if_eq:NnTF \l_token_tl { ( %---)
438 } {

```

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

```

439         \int_incr:N \l_depth_int
440         \tl_put_right:NV \l_query_tl \l_token_tl
441     } {

```

This is not a ‘(’.

```

442         \tl_if_eq:NnTF \l_token_tl { %(
443         )
444     } {

```

We found a ‘)’, decrement the depth.

```

445         \int_decr:N \l_depth_int
446         \int_compare:nNnTF \l_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.

```

447     \exp_args:NV #2 \l_query_tl \l_ans_tl
448     \bool_set_false:N \l_query_bool
449 } {

```

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

```

450         \tl_put_right:NV \l_query_tl \l_token_tl
451     }

```

Above ends the code for a positive depth.

```

452 } {

```

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

```

453         \tl_put_right:NV \l_query_tl \l_token_tl
454     }
455 }
456 }

```

Above ends the code for Not a ‘(’

```

457     }
458 }

```

Above ends the code for: Found the ‘(’ after the ‘?’

```

459 }

```

Above ends the code for not a no value quark.

```

460 }

```

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.

```

461         \quark_if_no_value:NT \l_token_tl {
462             \bool_set_false:N \l_query_bool
463             \bool_set_false:N \l_ask_bool
464         }
465     } {

```

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

```

466         \tl_put_right:NV \l_ans_tl \l_token_tl
467     }

```

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

```

468     }
469 }

```

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

```

470     \exp_args:NNNV
471     \__bnvs_group_end:
472     \tl_put_right:Nn #3 \l_ans_tl
473 }

```

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.

```

474 \exp_args_generate:n { VVx }
475 \prg_new_conditional:Npnn \__bnvs_extract_key:NNN
476     #1 #2 #3 { T, F, TF } {
477     \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN/
478         \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn#3./}
479     \__bnvs_group_begin:
480     \exp_args:NNV
481     \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex #2 \l_match_seq {

```

This is a correct key, update the path sequence accordingly

```

482     \exp_args:Nx
483     \tl_if_empty:nT { \seq_item:Nn \l_match_seq 3 } {
484         \tl_put_left:NV #2 { #1 }
485     \__bnvs_DEBUG:x { VERIF~\tl_to_str:V #2 }
486     }
487     \exp_args:NNNx

```

```

488     \seq_set_split:Nnn \l_split_seq . { \seq_item:Nn \l_match_seq 4 }
489     \seq_remove_all:Nn \l_split_seq { }
490     \seq_pop_left:NN \l_split_seq \l_a_tl
491     \seq_if_empty:NTF \l_split_seq {

```

No new integer path component is added.

```

492     \cs_set:Npn \:nn ##1 ##2 {
493         \__bnvs_group_end:
494         \tl_set:Nn #1 { ##1 }
495         \tl_set:Nn #2 { ##2 }
496     }
497     \exp_args:NVV \:nn #1 #2
498     \__bnvs_DEBUG:x { END/\string#1:#1/\string#2:#2/ }
499     } {

```

Some new integer path components are added.

```

500     \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN/\string#1:#1/
501         \string#2:#2/\string#3:\seq_use:Nn#3./
502         \string\l_split_seq:\seq_use:Nn\l_split_seq./ }
503     \cs_set:Npn \:nnn ##1 ##2 ##3 {
504         \__bnvs_group_end:
505         \tl_set:Nn #1 { ##1 }
506         \tl_set:Nn #2 { ##2 }
507         \seq_set_split:Nnn #3 . { ##3 }
508         \seq_remove_all:Nn #3 { }
509     }
510     \__bnvs_DEBUG:n{1}
511     \exp_args:NVVx
512     \:nnn #1 #2 {
513         \seq_use:Nn \l_split_seq . . \seq_use:Nn #3 .
514     }
515     \__bnvs_DEBUG:x { END/\string#1:#1/\string#2:#2/
516         \string#3:\seq_use:Nn #3 . /
517         \string\l_split_seq:\seq_use:Nn \l_split_seq . / }
518     }
519     \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN\space TRUE/
520         \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3 . /}
521     \prg_return_true:
522     } {
523         \__bnvs_group_end:
524     \__bnvs_DEBUG:x { \string\__bnvs_extract_key:NNN\space FALSE/
525         \string#1/\string#2/\string#3/}
526     \prg_return_false:
527     }
528 }

```

```

\__bnvs_resolve:NNNTF \__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 $\backslash l_a_seq$, $\backslash l_b_seq$.

```

529 \prg_new_conditional:Npnn \__bnvs_resolve:NNN
530   #1 #2 #3 { T, F, TF } {
531   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNN/
532     \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./ }
533   \__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.

```

534 \seq_set_eq:NN \l_a_seq #3
535 \seq_clear:N \l_b_seq
536 \cs_set:Npn \loop: {
537   \__bnvs_call:TF {
538     \tl_set_eq:NN \l_a_tl #2
539     \seq_if_empty:NTF \l_a_seq {
540       \exp_args:Nx
541       \__bnvs_get:nNTF { \l_a_tl / L } \l_b_tl {
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     \tl_put_right:Nx \l_a_tl { . \seq_use:Nn \l_a_seq . }
550     \get_extract:F {
551         \seq_pop_right:NNT \l_a_seq \l_c_tl {
552             \seq_put_left:NV \l_b_seq \l_c_tl
553         }
554     }
555 }
556 \loop:
557 } {
558     \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNN\space~TOO~MANY~CALLS/
559         \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./ }
560     \__bnvs_group_end:
561     \prg_return_false:
562 }
563 }
564 \cs_set:Npn \get_extract:F ##1 {
565     \exp_args:Nx
566     \__bnvs_get:nNTF { \l_a_tl / A } \l_b_tl {
567     \__bnvs_DEBUG:x { RESOLUTION:~\l_a_tl / A=>\l_b_tl}
568         \__bnvs_extract_key:NNTF #1 \l_b_tl \l_b_seq {
569             \tl_set_eq:NN #2 \l_b_tl
570             \seq_set_eq:NN #3 \l_b_seq
571             \seq_set_eq:NN \l_a_seq \l_b_seq
572             \seq_clear:N \l_b_seq
573         } { ##1 }
574     } { ##1 }
575 }
576 \cs_set:Npn \return_true: {
577     \cs_set:Npn \:nnn #####1 #####2 #####3 {
578         \__bnvs_group_end:
579         \tl_set:Nn #1 { #####1 }
580         \tl_set:Nn #2 { #####2 }
581         \seq_set_split:Nnn #3 . { #####3 }
582         \seq_remove_all:Nn #3 { }
583     }
584     \exp_args:NVVx
585     \:nnn #1 #2 {
586         \seq_use:Nn #3 .
587     }
588     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNN\space TRUE/
589         \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./ }
590     \prg_return_true:
591 }
592 \loop:
593 }

```

```

__bnvs_resolve_n:NNNTF $\overline{TF}$  \__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

```

594 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
595   #1 #2 #3 { T, F, TF } {
596 \__bnvs_DEBUG:x { \string\__bnvs_resolve_n:NNN/
597   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./ }
598   \__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.

```

599 \seq_set_eq:NN \l_a_seq #3
600 \seq_clear:N \l_b_seq
601 \cs_set:Npn \loop: {
602   \__bnvs_call:TF {
603     \tl_set_eq:NN \l_a_tl #2
604     \seq_if_empty:NTF \l_a_seq {
605       \exp_args:Nx
606       \__bnvs_get:nNTF { \l_a_tl / L } \l_b_tl {
607         \cs_set:Nn \loop: { \return_true: }
608       } {
609         \seq_if_empty:NTF \l_b_seq {
610           \cs_set:Nn \loop: { \return_true: }
611         } {
612           \get_extract:F {

```

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

```

613       \cs_set:Nn \loop: { \return_true: }
614     }
615   }
616 } {
617 } {
618   \tl_put_right:Nx \l_a_tl { . \seq_use:Nn \l_a_seq . }
619   \get_extract:F {
620     \seq_pop_right:NNT \l_a_seq \l_c_tl {
621       \seq_put_left:NV \l_b_seq \l_c_tl
622     }
623   }
624 }
625 \loop:
626 } {
627 \__bnvs_DEBUG:x { \string\__bnvs_resolve_n:NNN\space-TOO-MANY-CALLS/
628   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./ }
629   \__bnvs_group_end:
630   \prg_return_false:
631 }

```

```

632 }
633 \cs_set:Npn \get_extract:F ##1 {
634   \exp_args:Nx
635   \__bnvs_get:nNTF { \l_a_tl / A } \l_b_tl {
636   \__bnvs_DEBUG:x { RESOLUTION:~\l_a_tl / A=>\l_b_tl}
637   \__bnvs_extract_key:NNTF #1 \l_b_tl \l_b_seq {
638     \tl_set_eq:NN #2 \l_b_tl
639     \seq_set_eq:NN #3 \l_b_seq
640     \seq_set_eq:NN \l_a_seq \l_b_seq
641     \seq_clear:N \l_b_seq
642   } { ##1 }
643 } { ##1 }
644 }
645 \cs_set:Npn \return_true: {
646   \cs_set:Npn \:nnn #####1 #####2 #####3 {
647     \__bnvs_group_end:
648     \tl_set:Nn #1 { #####1 }
649     \tl_set:Nn #2 { #####2 }
650     \seq_set_split:Nnn #3 . { #####3 }
651     \seq_remove_all:Nn #3 { }
652   }
653   \exp_args:NVVx
654   \:nnn #1 #2 {
655     \seq_use:Nn #3 .
656   }
657   \__bnvs_DEBUG:x { ... \string\__bnvs_resolve_n:NNN\space TRUE/
658   \string#1:#1/\string#2:#2/\string#3:\seq_use:Nn #3./ }
659   \prg_return_true:
660 }
661 \loop:
662 }

```

```

\__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, $\{false\}$ 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.

```

663 \prg_new_conditional:Npnn \__bnvs_resolve:NNNN
664   #1 #2 #3 #4 { T, F, TF } {
665   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN/
666   \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./ }
667   #1 {
668     \__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.

```

669     \tl_set_eq:NN \l_a_tl #3
670     \seq_set_eq:NN \l_a_seq #4
671     \tl_clear:N \l_b_tl
672     \seq_clear:N \l_b_seq
673     \cs_set:Npn \return_true: {
674       \cs_set:Npn \:nnn #####1 #####2 #####3 {
675         \__bnvs_group_end:
676         \tl_set:Nn #2 { #####1 }
677         \tl_set:Nn #3 { #####2 }
678         \seq_set_split:Nnn #4 . { #####3 }
679         \seq_remove_all:Nn #4 { }
680       }
681       \exp_args:NVVx
682       \:nnn #2 #3 {
683         \seq_use:Nn #4 .
684       }
685     \__bnvs_DEBUG:x { ... \string\__bnvs_resolve:NNNN \space TRUE/
686       \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./ }
687     \prg_return_true:
688   }
689   \cs_set:Npn \branch:n ##1 {
690     \seq_pop_right:NNTF \l_a_seq \l_b_tl {
691       \seq_put_left:NV \l_b_seq \l_b_tl
692       \__bnvs_DEBUG:x {\string\__bnvs_resolve:NNNN \space POP~TRUE~##1}
693       \__bnvs_DEBUG:x {\string\l_b_tl:\l_b_tl }
694       \__bnvs_DEBUG:x {\string\l_a_seq:\seq_count:N\l_a_seq/\seq_use:Nn \l_a_seq ./ }
695       \__bnvs_DEBUG:x {\string\l_b_seq:\seq_count:N\l_b_seq/\seq_use:Nn \l_b_seq ./ }
696       \tl_set:Nn \l_a_tl { #3 . }
697       \tl_put_right:Nx \l_a_tl { \seq_use:Nn \l_a_seq . }
698     } {
699       \cs_set_eq:NN \loop: \return_true:
700     }
701   }
702   \cs_set:Npn \branch:FF ##1 ##2 {
703     \exp_args:Nx
704     \__bnvs_get:nNTF { \l_a_tl / A } \l_b_tl {
705       \__bnvs_extract_key:NNTF #2 \l_b_tl \l_b_seq {
706         \tl_set_eq:NN #3 \l_b_tl
707         \seq_set_eq:NN #4 \l_b_seq
708         \seq_set_eq:NN \l_a_seq \l_b_seq
709       } { ##1 }
710     } { ##2 }
711   }
712   \cs_set:Npn \extract_key:F {
713     \__bnvs_extract_key:NNTF #2 \l_b_tl \l_b_seq {
714       \tl_set_eq:NN #3 \l_b_tl
715       \seq_set_eq:NN #4 \l_b_seq
716       \seq_set_eq:NN \l_a_seq \l_b_seq
717     }
718   }
719   \cs_set:Npn \loop: {
720     \__bnvs_call:TF {
721       \exp_args:Nx
722       \__bnvs_get:nNTF { \l_a_tl / L } \l_b_tl {

```

If there is a length, no resolution occurs.

```

723     \branch:n { 1 }
724   } {
725     \seq_pop_right:NNTF \l_a_seq \l_c_tl {
726       \seq_clear:N \l_b_seq
727       \tl_set:Nn \l_a_tl { #3 . }
728       \tl_put_right:Nx \l_a_tl { \seq_use:Nn \l_a_seq . . }
729       \tl_put_right:NV \l_a_tl \l_c_tl
730     \branch:FF {

```

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

```

731 \seq_put_left:NV \l_b_seq \l_c_tl
732   } {

```

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

```

733 \seq_put_left:NV \l_b_seq \l_c_tl
734   }
735   } {
736     \branch:FF {
737       \cs_set_eq:NN \loop: \return_true:
738     } {
739       \cs_set:Npn \loop: {
740         \__bnvs_group_end:
741         \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space FALSE/
742           \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
743           \g__bnvs_call_int : \int_use:N \g__bnvs_call_int/
744         }
745         \prg_return_false:
746       }
747     }
748   }
749   } {
750     \cs_set:Npn \loop: {
751       \__bnvs_group_end:
752       \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space FALSE/
753         \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
754         \g__bnvs_call_int : \int_use:N \g__bnvs_call_int/
755       }
756     }
757     \prg_return_false:
758   }
759   }
760   \loop:
761 }
762 \loop:
763 } {
764   \prg_return_true:
765 }
766 }
767 \prg_new_conditional:Npnn \__bnvs_resolve_OLD:NNNN
768   #1 #2 #3 #4 { T, F, TF } {
769   \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN/
770     \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./ }
771   #1 {
772     \__bnvs_group_begin:

```

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

```

773     \tl_set_eq:NN \l_a_tl #3
774     \seq_set_eq:NN \l_a_seq #4
775     \cs_set:Npn \return_true: {
776       \cs_set:Npn \:nnn #####1 #####2 #####3 {
777         \__bnvs_group_end:
778         \tl_set:Nn #2 { #####1 }
779         \tl_set:Nn #3 { #####2 }
780         \seq_set_split:Nnn #4 . { #####3 }
781         \seq_remove_all:Nn #4 { }
782       }
783       \exp_args:NVVx
784       \:nnn #2 #3 {
785         \seq_use:Nn #4 .
786       }
787     \__bnvs_DEBUG:x { ...\string\__bnvs_resolve:NNNN\space TRUE/
788       \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./ }
789     \prg_return_true:
790   }
791   \cs_set:Npn \branch:n ##1 {
792     \seq_pop_left:NNTF \l_a_seq \l_b_tl {
793       \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space POP~TRUE~##1/
794         \string\l_b_tl:\l_b_tl/\string\l_a_seq:\seq_count:N\l_a_seq/
795         \seq_use:Nn \l_a_seq ./ }
796       \tl_put_right:Nn \l_a_tl { . }
797       \tl_put_right:NV \l_a_tl \l_b_tl
798     } {
799       \cs_set_eq:NN \loop: \return_true:
800     }
801   }
802   \cs_set:Npn \loop: {
803     \__bnvs_call:TF {
804       \exp_args:Nx
805       \__bnvs_get:nNTF { \l_a_tl / L } \l_b_tl {
806         \branch:n { 1 }
807       } {
808         \exp_args:Nx
809         \__bnvs_get:nNTF { \l_a_tl / A } \l_b_tl {
810           \__bnvs_extract_key:NNNTF #2 \l_b_tl \l_a_seq {
811             \tl_set_eq:NN \l_a_tl \l_b_tl
812             \tl_set_eq:NN #3 \l_b_tl
813             \seq_set_eq:NN #4 \l_a_seq
814           } {
815             \branch:n { 2 }
816           }
817         } {
818           \branch:n { 3 }
819         }
820       } {
821     } {
822       \cs_set:Npn \loop: {
823         \__bnvs_group_end:
824         \__bnvs_DEBUG:x { \string\__bnvs_resolve:NNNN\space FALSE/

```

```

825 \string#1/\string#2:#2/\string#3:#3/\string#4:\seq_use:Nn #4./
826 \g__bnvs_call_int : \int_use:N \g__bnvs_call_int/
827 }
828     \prg_return_false:
829 }
830 }
831 \loop:
832 }
833 \loop:
834 } {
835     \prg_return_true:
836 }
837 }

```

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.

```

838 \cs_new:Npn \__bnvs_fp_round:nN #1 #2 {
839     \__bnvs_DEBUG:x { ROUND:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2}
840     \tl_if_empty:NTF { #1 } {
841         \__bnvs_DEBUG:x { ...ROUND:~EMPTY }
842     } {
843         \tl_put_right:Nx #2 {
844             \fp_eval:n { round(#1) }
845         }
846         \__bnvs_DEBUG:x { ...ROUND:~\tl_to_str:V #2 => \string#2}
847     }
848 }
849 \cs_generate_variant:Nn \__bnvs_fp_round:nN { VN, xN }
850 \cs_new:Npn \__bnvs_fp_round:N #1 {
851     \tl_if_empty:VTF #1 {
852         \__bnvs_DEBUG:x { ROUND:~EMPTY }
853     } {
854         \__bnvs_DEBUG:x { ROUND-IN:~\tl_to_str:V #1 }
855         \tl_set:Nx #1 {
856             \fp_eval:n { round(#1) }
857         }
858         \__bnvs_DEBUG:x { ROUND-OUT:~\tl_to_str:V #1 }
859     }
860 }

```

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

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.

```

861 \cs_set:Npn \__bnvs_return_true:nnN #1 #2 #3 {
862     \tl_if_empty:NTF \l_ans_tl {
863         \__bnvs_group_end:

```

```

864 \__bnvs_DEBUG:n { RETURN_FALSE/key=#1/type=#2/EMPTY }
865 \__bnvs_gremove:n { #1//#2 }
866 \prg_return_false:
867 } {
868 \__bnvs_fp_round:N \l_ans_tl
869 \__bnvs_gput:nV { #1//#2 } \l_ans_tl
870 \exp_args:NNNV
871 \__bnvs_group_end:
872 \tl_put_right:Nn #3 \l_ans_tl
873 \__bnvs_DEBUG:x { RETURN_TRUE/key=#1/type=#2/ans=\l_ans_tl/ }
874 \prg_return_true:
875 }
876 }
877 \cs_set:Npn \__bnvs_return_false:nn #1 #2 {
878 \__bnvs_DEBUG:n { RETURN_FALSE/key=#1/type=#2/ }
879 \__bnvs_group_end:
880 \__bnvs_gremove:n { #1//#2 }
881 \prg_return_false:
882 }
883 \prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
884 \__bnvs_DEBUG:x { RAW_FIRST/
885 key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2/}
886 \__bnvs_if_in:nTF { #1//A } {
887 \__bnvs_DEBUG:n { RAW_FIRST/#1/CACHED }
888 \tl_put_right:Nx #2 { \__bnvs_item:n { #1//A } }
889 \prg_return_true:
890 } {
891 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/NOT_CACHED }
892 \__bnvs_group_begin:
893 \tl_clear:N \l_ans_tl
894 \__bnvs_get:nNTF { #1/A } \l_a_tl {
895 \__bnvs_DEBUG:x { RAW_FIRST/key=#1/A=\l_a_tl }
896 \__bnvs_if_append:VNTF \l_a_tl \l_ans_tl {
897 \__bnvs_return_true:nnN { #1 } A #2
898 } {
899 \__bnvs_return_false:nn { #1 } A
900 }
901 } {
902 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/A/F }
903 \__bnvs_get:nNTF { #1/L } \l_a_tl {
904 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/L=\l_a_tl }
905 \__bnvs_get:nNTF { #1/Z } \l_b_tl {
906 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/Z=\l_b_tl }
907 \__bnvs_if_append:xNTF {
908 \l_b_tl - ( \l_a_tl ) + 1
909 } \l_ans_tl {
910 \__bnvs_return_true:nnN { #1 } A #2
911 } {
912 \__bnvs_return_false:nn { #1 } A
913 }
914 } {
915 \__bnvs_DEBUG:n { RAW_FIRST/key=#1/Z/F/ }
916 \__bnvs_return_false:nn { #1 } A
917 }

```



```

918     } {
919     \__bnvs_DEBUG:n { RAW_FIRST/key=#1/L/F/ }
920         \__bnvs_return_false:nn { #1 } A
921     }
922 }
923 }
924 }
925 \prg_generate_conditional_variant:Nnn
926     \__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.

```

927 \prg_new_conditional:Npnn \__bnvs_if_first:nN #1 #2 { T, F, TF } {
928 \__bnvs_DEBUG:x { IF_FIRST/\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2}
929 \__bnvs_raw_first:nNTF { #1 } #2 {
930     \prg_return_true:
931 } {
932     \__bnvs_get:nNTF { #1/C } \l_a_tl {
933 \__bnvs_DEBUG:n { IF_FIRST/#1/C/T/\l_a_tl }
934     \bool_set_true:N \l_no_counter_bool
935     \__bnvs_if_append:xNTF \l_a_tl \l_ans_tl {
936         \__bnvs_return_true:nnN { #1 } A #2
937     } {
938         \__bnvs_return_false:nn { #1 } A
939     }
940 } {
941     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
942         \__bnvs_gput:nn { #1/A } { 1 }
943         \tl_set:Nn #2 { 1 }
944 \__bnvs_DEBUG:x{IF_FIRST_MATCH:
945     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /}
946     \__bnvs_return_true:nnN { #1 } A #2
947 } {
948 \__bnvs_DEBUG:x{IF_FIRST_NO_MATCH:
949     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 /}
950     \__bnvs_return_false:nn { #1 } A
951 }
952 }
953 }
954 }

```

__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.

```

955 \cs_new:Npn \__bnvs_first:nN #1 #2 {
956     \__bnvs_if_first:nNF { #1 } #2 {
957         \msg_error:nnn { beanoves } { :n } { Range-with-no~first:~#1 }
958     }
959 }
960 \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.

```
961 \prg_new_conditional:Npnn \\_bnvs_raw_length:nN #1 #2 { T, F, TF } {
962   \\_bnvs_DEBUG:x { \string\\_bnvs_raw_length:nN/#1/\string#2/ }
963   \\_bnvs_if_in:nTF { #1//L } {
964     \tl_put_right:Nx #2 { \\_bnvs_item:n { #1//L } }
965   \\_bnvs_DEBUG:x { RAW_LENGTH/CACHED/key:#1/\\_bnvs_item:n { #1//L } }
966   \prg_return_true:
967 } {
968   \\_bnvs_DEBUG:x { RAW_LENGTH/NOT_CACHED/key:#1/ }
969   \\_bnvs_gput:nn { #1//L } { 0 }
970   \\_bnvs_group_begin:
971   \tl_clear:N \l_ans_tl
972   \\_bnvs_if_in:nTF { #1/L } {
973     \\_bnvs_if_append:xNTF {
974       \\_bnvs_item:n { #1/L }
975     } \l_ans_tl {
976       \\_bnvs_return_true:nnN { #1 } L #2
977     } {
978       \\_bnvs_return_false:nn { #1 } L
979     }
980   } {
981     \\_bnvs_get:nNTF { #1/A } \l_a_tl {
982       \\_bnvs_get:nNTF { #1/Z } \l_b_tl {
983         \\_bnvs_if_append:xNTF {
984           \l_b_tl - (\l_a_tl) + 1
985         } \l_ans_tl {
986           \\_bnvs_return_true:nnN { #1 } L #2
987         } {
988           \\_bnvs_return_false:nn { #1 } L
989         }
990       } {
991         \\_bnvs_return_false:nn { #1 } L
992       }
993     } {
994       \\_bnvs_return_false:nn { #1 } L
995     }
996   }
997 }
998 }
999 \prg_generate_conditional_variant:Nnn
1000   \\_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.

```
1001 \prg_new_conditional:Npnn \\_bnvs_raw_last:nN #1 #2 { T, F, TF } {
1002   \\_bnvs_DEBUG:n { RAW_LAST/#1 }
1003   \\_bnvs_if_in:nTF { #1//Z } {
1004     \tl_put_right:Nx #2 { \\_bnvs_item:n { #1//Z } }
1005   }
```

```

1005     \prg_return_true:
1006   } {
1007     \__bnvs_gput:nn { #1//Z } { 0 }
1008     \__bnvs_group_begin:
1009     \tl_clear:N \l_ans_tl
1010     \__bnvs_if_in:nTF { #1/Z } {
1011 \__bnvs_DEBUG:x { NORMAL_RAW_LAST:~\__bnvs_item:n { #1/Z } }
1012     \__bnvs_if_append:xNTF {
1013       \__bnvs_item:n { #1/Z }
1014     } \l_ans_tl {
1015       \__bnvs_return_true:nnN { #1 } Z #2
1016     } {
1017       \__bnvs_return_false:nn { #1 } Z
1018     }
1019   } {
1020     \__bnvs_get:nNTF { #1/A } \l_a_tl {
1021       \__bnvs_get:nNTF { #1/L } \l_b_tl {
1022         \__bnvs_if_append:xNTF {
1023           \l_a_tl + (\l_b_tl) - 1
1024         } \l_ans_tl {
1025           \__bnvs_return_true:nnN { #1 } Z #2
1026         } {
1027           \__bnvs_return_false:nn { #1 } Z
1028         }
1029       } {
1030         \__bnvs_return_false:nn { #1 } Z
1031       }
1032     } {
1033       \__bnvs_return_false:nn { #1 } Z
1034     }
1035   }
1036 }
1037 }
1038 \prg_generate_conditional_variant:Nnn
1039   \__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>

```

1040 \cs_new:Npn \__bnvs_last:nN #1 #2 {
1041   \__bnvs_raw_last:nNF { #1 } #2 {
1042     \msg_error:nnn { beanoes } { :n } { Range-with-no-last:~#1 }
1043   }
1044 }
1045 \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.

```

1046 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
1047   \__bnvs_if_in:nTF { #1//N } {
1048     \tl_put_right:Nx #2 { \__bnvs_item:n { #1//N } }

```

```

1049 \prg_return_true:
1050 } {
1051   \__bnvs_group_begin:
1052   \cs_set:Npn \__bnvs_return_true: {
1053     \tl_if_empty:NTF \l_ans_tl {
1054       \__bnvs_group_end:
1055       \prg_return_false:
1056     } {
1057       \__bnvs_fp_round:N \l_ans_tl
1058       \__bnvs_gput:nV { #1//N } \l_ans_tl
1059       \exp_args:NNNV
1060       \__bnvs_group_end:
1061       \tl_put_right:Nn #2 \l_ans_tl
1062       \prg_return_true:
1063     }
1064   }
1065   \cs_set:Npn \return_false: {
1066     \__bnvs_group_end:
1067     \prg_return_false:
1068   }
1069   \tl_clear:N \l_a_tl
1070   \__bnvs_raw_last:nNTF { #1 } \l_a_tl {
1071     \__bnvs_if_append:xNTF {
1072       \l_a_tl + 1
1073     } \l_ans_tl {
1074       \__bnvs_return_true:
1075     } {
1076       \return_false:
1077     }
1078   } {
1079     \return_false:
1080   }
1081 }
1082 }
1083 \prg_generate_conditional_variant:Nnn
1084 \__bnvs_if_next:nN { VN } { T, F, TF }

```

```

\__bnvs_next:nN
\__bnvs_next:VN

```

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

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

```

1085 \cs_new:Npn \__bnvs_next:nN #1 #2 {
1086   \__bnvs_if_next:nNF { #1 } #2 {
1087     \msg_error:nnn { beanoves } { :n } { Range-with-no~next:~#1 }
1088   }
1089 }
1090 \cs_generate_variant:Nn \__bnvs_next:nN { VN }

```

```

__bnvs_if_index:nnNTF
__bnvs_if_index:VVNTF
__bnvs_if_index:nnnNTF

```

```

__bnvs_if_index:nnNTF {<name>} {<integer>} <tl variable> {<true code>} {<false
code>}}

```

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

```

1091 \prg_new_conditional:Npnn __bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
1092   __bnvs_DEBUG:x { IF_INDEX:key=#1/index=#2/\string#3/ }
1093   __bnvs_group_begin:
1094   \tl_clear:N \l_ans_tl
1095   __bnvs_raw_first:nNTF { #1 } \l_ans_tl {
1096     \tl_put_right:Nn \l_ans_tl { + (#2) - 1 }
1097     \exp_args:NNV
1098     __bnvs_group_end:
1099     __bnvs_fp_round:nN \l_ans_tl #3
1100   __bnvs_DEBUG:x { IF_INDEX_TRUE:key=#1/index=#2/
1101     \string#3=\tl_to_str:N #3 }
1102     \prg_return_true:
1103   } {
1104   __bnvs_DEBUG:x { IF_INDEX_FALSE:key=#1/index=#2/ }
1105     \prg_return_false:
1106   }
1107 }
1108 \prg_generate_conditional_variant:Nnn
1109   __bnvs_if_index:nnN { VVN } { T, F, TF }

```

```

__bnvs_if_range:nNTF

```

```

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

```

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

```

1110 \prg_new_conditional:Npnn __bnvs_if_range:nN #1 #2 { T, F, TF } {
1111   __bnvs_DEBUG:x{ RANGE:key=#1/\string#2/}
1112   \bool_if:NTF \l__bnvs_no_range_bool {
1113     \prg_return_false:
1114   } {
1115     __bnvs_if_in:nTF { #1/ } {
1116       \tl_put_right:Nn { 0-0 }
1117     } {
1118       __bnvs_group_begin:
1119       \tl_clear:N \l_a_tl
1120       \tl_clear:N \l_b_tl
1121       \tl_clear:N \l_ans_tl
1122       __bnvs_raw_first:nNTF { #1 } \l_a_tl {
1123         __bnvs_raw_last:nNTF { #1 } \l_b_tl {
1124           \exp_args:NNNx
1125           __bnvs_group_end:
1126           \tl_put_right:Nn #2 { \l_a_tl - \l_b_tl }
1127       __bnvs_DEBUG:x{ RANGE_TRUE_A_Z:key=#1/\string#2=#2/}
1128         \prg_return_true:
1129       } {

```

```

1130         \exp_args:NNNx
1131         \__bnvs_group_end:
1132         \tl_put_right:Nn #2 { \l_a_tl - }
1133     \__bnvs_DEBUG:x{ RANGE_TRUE_A:key=#1/\string#2=#2/}
1134     \prg_return_true:
1135     }
1136     } {
1137         \__bnvs_raw_last:nNTF { #1 } \l_b_tl {
1138     \__bnvs_DEBUG:x{ RANGE_TRUE_Z:key=#1/\string#2=#2/}
1139         \exp_args:NNNx
1140         \__bnvs_group_end:
1141         \tl_put_right:Nn #2 { - \l_b_tl }
1142         \prg_return_true:
1143         } {
1144     \__bnvs_DEBUG:x{ RANGE_FALSE:key=#1/}
1145         \__bnvs_group_end:
1146         \prg_return_false:
1147         }
1148     }
1149     }
1150     }
1151     }
1152     \prg_generate_conditional_variant:Nnn
1153     \__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>.

```

1154     \cs_new:Npn \__bnvs_range:nN #1 #2 {
1155         \__bnvs_if_range:nNF { #1 } #2 {
1156             \msg_error:nnn { beanoves } { :n } { No~range~available:~#1 }
1157         }
1158     }
1159     \cs_generate_variant:Nn \__bnvs_range:nN { VN }

```

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

__bnvs_if_free_counter:VNTF code)}

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

```

1160     \prg_new_conditional:Npnn \__bnvs_if_free_counter:nN #1 #2 { T, F, TF } {
1161     \__bnvs_DEBUG:x { IF_FREE: key=\tl_to_str:n{#1}/
1162         value=\__bnvs_item:n {#1/C}/cs=\string #2/ }
1163     \__bnvs_group_begin:
1164     \tl_clear:N \l_ans_tl
1165     \__bnvs_get:nNF { #1/C } \l_ans_tl {
1166         \__bnvs_raw_first:nNF { #1 } \l_ans_tl {
1167             \__bnvs_raw_last:nNF { #1 } \l_ans_tl { }
1168         }
1169     }
1170     \__bnvs_DEBUG:x { IF_FREE_2:\string \l_ans_tl=\tl_to_str:V \l_ans_tl/}
1171     \tl_if_empty:NTF \l_ans_tl {
1172         \__bnvs_group_end:

```

```

1173     \regex_match:NnTF \c__bnvs_A_key_Z_regex { #1 } {
1174     \__bnvs_gput:nn { #1/C } { 1 }
1175     \tl_set:Nn #2 { 1 }
1176     \__bnvs_DEBUG:x { IF_FREE_MATCH_TRUE:
1177     key=\tl_to_str:n{#1}\string #2=\tl_to_str:V #2 / }
1178     \prg_return_true:
1179     } {
1180     \__bnvs_DEBUG:x { IF_FREE_NO_MATCH_FALSE:
1181     key=\tl_to_str:n{#1}\string #2=\tl_to_str:V #2/ }
1182     \prg_return_false:
1183     }
1184     } {
1185     \__bnvs_gput:nV { #1/C } \l_ans_tl
1186     \exp_args:NNNV
1187     \__bnvs_group_end:
1188     \tl_set:Nn #2 \l_ans_tl
1189     \__bnvs_DEBUG:x { IF_FREE_TRUE(2): /
1190     key=\tl_to_str:n{#1}/\string #2=\tl_to_str:V #2}
1191     \prg_return_true:
1192     }
1193     }
1194     \prg_generate_conditional_variant:Nnn
1195     \__bnvs_if_free_counter:nN { VN } { T, F, TF }

```

```

\__bnvs_if_counter:nNTF
\__bnvs_if_counter:VNTF

```

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

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.

```

1196 \prg_new_conditional:Npnn \__bnvs_if_counter:nN #1 #2 { T, F, TF } {
1197 \__bnvs_DEBUG:x { IF_COUNTER:key=
1198 \tl_to_str:n{#1}/\string #2=\tl_to_str:V #2 }
1199 \__bnvs_group_begin:
1200 \__bnvs_if_free_counter:nNTF { #1 } \l_ans_tl {

```

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

```

1201 \tl_clear:N \l_a_tl
1202 \__bnvs_raw_first:nNT { #1 } \l_a_tl {
1203 \fp_compare:nNnT { \l_ans_tl } < { \l_a_tl } {
1204 \tl_set:NV \l_ans_tl \l_a_tl
1205 }
1206 }

```

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

```

1207 \tl_clear:N \l_a_tl
1208 \__bnvs_raw_last:nNT { #1 } \l_a_tl {
1209 \fp_compare:nNnT { \l_ans_tl } > { \l_a_tl } {
1210 \tl_set:NV \l_ans_tl \l_a_tl
1211 }
1212 }
1213 \exp_args:NNV
1214 \__bnvs_group_end:
1215 \__bnvs_fp_round:nN \l_ans_tl #2
1216 \__bnvs_DEBUG:x {IF_COUNTER_TRUE:key=\tl_to_str:n{#1}/
1217 \string #2=\tl_to_str:V #2 }

```

```

1218     \prg_return_true:
1219   } {
1220   \__bnvs_DEBUG:x {IF_COUNTER_FALSE:key=\tl_to_str:n{#1}/
1221     \string #2=\tl_to_str:V #2 }
1222     \prg_return_false:
1223   }
1224 }
1225 \prg_generate_conditional_variant:Nnn
1226   \__bnvs_if_counter:nN { VN } { T, F, TF }

```

<pre> __bnvs_if_incr:nnTF __bnvs_if_incr:nnNTF __bnvs_if_incr:(VnN VVN)TF </pre>	<pre> __bnvs_if_incr:nnTF {<name>} {<offset>} {<true code>} {<false code>} __bnvs_if_incr:nnNTF {<name>} {<offset>} <tl variable> {<true code>} {<false code>} </pre>
---	---

Increment the free counter position accordingly. When requested, put the result in the *<tl variable>*. In the second version, the result will lay within the declared range.

```

1227 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {
1228   \__bnvs_DEBUG:x { IF_INCR:\tl_to_str:n{#1}/\tl_to_str:n{#2} }
1229   \__bnvs_group_begin:
1230   \tl_clear:N \l_a_tl
1231   \__bnvs_if_free_counter:nNTF { #1 } \l_a_tl {
1232     \tl_clear:N \l_b_tl
1233     \__bnvs_if_append:xNTF { \l_a_tl + (#2) } \l_b_tl {
1234       \__bnvs_fp_round:N \l_b_tl
1235       \__bnvs_gput:nV { #1/C } \l_b_tl
1236       \__bnvs_group_end:
1237     }
1238     \prg_return_true:
1239   } {
1240     \__bnvs_group_end:
1241     \__bnvs_DEBUG:x { IF_INCR_FALSE(1):#1/#2 }
1242     \prg_return_false:
1243   }
1244 } {
1245   \__bnvs_group_end:
1246   \__bnvs_DEBUG:x { IF_INCR_FALSE(2):#1/#2 }
1247   \prg_return_false:
1248 }
1249 }
1250 \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1251   \__bnvs_if_incr:nnTF { #1 } { #2 } {
1252     \__bnvs_if_counter:nNTF { #1 } #3 {
1253       \prg_return_true:
1254     } {
1255       \prg_return_false:
1256     }
1257   } {
1258     \prg_return_false:
1259   }
1260 }
1261 \prg_generate_conditional_variant:Nnn
1262   \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }

```


5.5.8 Evaluation

`_bnvs_if_append:nNTF`
`_bnvs_if_append:(VN|xN)TF`

`_bnvs_if_append:nNTF {⟨integer expression⟩} ⟨tl variable⟩ {⟨true code⟩} {⟨false code⟩}`

Evaluates the *⟨integer expression⟩*, replacing all the named specifications by their static counterpart then put the result to the right of the *⟨tl variable⟩*. Executed within a group. Heavily used by `_bnvs_eval_query:nN`, where *⟨integer expression⟩* was initially enclosed in `‘?(...)’`. Local variables:

`\l_ans_tl` To feed *⟨tl variable⟩* with.

(End definition for `\l_ans_tl`. This variable is documented on page ??.)

`\l_split_seq` The sequence of caught query groups and non queries.

(End definition for `\l_split_seq`. This variable is documented on page ??.)

`\l_split_int` Is the index of the non queries, before all the caught groups.

(End definition for `\l_split_int`. This variable is documented on page ??.)

```
1263 \int_if_exist:NF \l_split_int {
1264   \int_new:N \l_split_int
1265 }
```

`\l_name_tl` Storage for `\l_split_seq` items that represent names.

(End definition for `\l_name_tl`. This variable is documented on page ??.)

`\l_path_tl` Storage for `\l_split_seq` items that represent integer paths.

(End definition for `\l_path_tl`. This variable is documented on page ??.)

Catch circular definitions.

```
1266 \prg_new_conditional:Npnn \_bnvs_if_append:nN #1 #2 { T, F, TF } {
1267   \_bnvs_DEBUG:x { \string\_bnvs_if_append:nNTF/
1268     \tl_to_str:n { #1 } / \string #2/
1269 }
1270   \_bnvs_call:TF {
1271     \_bnvs_DEBUG:x { IF_APPEND...}
1272     \_bnvs_group_begin:
```

Local variables:

```
1273   \int_zero:N \l_split_int
1274   \seq_clear:N \l_split_seq
1275   \tl_clear:N \l_id_tl
1276   \tl_clear:N \l_name_tl
1277   \tl_clear:N \l_path_tl
1278   \tl_clear:N \l_group_tl
1279   \tl_clear:N \l_ans_tl
1280   \tl_clear:N \l_a_tl
```

Implementation:

```
1281   \regex_split:NnN \c__bnvs_split_regex { #1 } \l_split_seq
1282   \_bnvs_DEBUG:x { IF_APPEND_SPLIT_SEQ: /
1283     \#=\seq_count:N \l_split_seq /
```

```

1284 \seq_use:Nn \l_split_seq / /
1285 }
1286 \int_set:Nn \l_split_int { 1 }
1287 \tl_set:Nx \l_ans_tl {
1288   \seq_item:Nn \l_split_seq { \l_split_int }
1289 }
1290 \__bnvs_DEBUG:x { ANS: \l_ans_tl }

```

\switch:nTF `\switch:nTF {<capture group number>} {<black code>} {<white code>}`

Helper function to locally set the `\l_group_tl` variable to the captured group *<capture group number>* and branch.

```

1291 \cs_set:Npn \switch:nTF ##1 ##2 ##3 ##4 {
1292   \tl_set:Nx ##2 {
1293     \seq_item:Nn \l_split_seq { \l_split_int + ##1 }
1294   }
1295   \__bnvs_DEBUG:x { IF_APPEND_SWITCH/##1/
1296     \int_eval:n { \l_split_int + ##1 } /
1297     \string##2=\tl_to_str:N##2/
1298   }
1299   \tl_if_empty:NTF ##2 {
1300     \__bnvs_DEBUG:x { IF_APPEND_SWITCH_WHITE/##1/
1301       \int_eval:n { \l_split_int + ##1 }
1302     }
1303     ##4 } {
1304     \__bnvs_DEBUG:x { IF_APPEND_SWITCH_BLACK/##1/
1305       \int_eval:n { \l_split_int + ##1 }
1306     }
1307     ##3
1308   }
1309 }

```

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

```

1310 \cs_set:Npn \return_true: {
1311   \fp_round:
1312   \exp_args:NNNV
1313   \__bnvs_group_end:
1314   \tl_put_right:Nn #2 \l_ans_tl
1315   \__bnvs_DEBUG:x { IF_APPEND_TRUE:\tl_to_str:n { #1 } /
1316     \string #2=\tl_to_str:V #2 /}
1317   \log_g_prop:
1318   \prg_return_true:
1319 }
1320 \cs_set:Npn \fp_round: {
1321   \__bnvs_fp_round:N \l_ans_tl
1322 }
1323 \cs_set:Npn \return_false: {
1324   \__bnvs_group_end:
1325   \__bnvs_DEBUG:x { IF_APPEND_FALSE:\tl_to_str:n { #1 } /
1326     \string #2=\tl_to_str:V #2 /}
1327   \prg_return_false:
1328 }
1329 \cs_set:Npn \:NnnT ##1 ##2 ##3 ##4 {

```

```

1330     \switch:nNTF { ##2 } \l_id_tl { } {
1331         \tl_set_eq:NN \l_id_tl \l__bnvs_id_tl
1332         \tl_put_left:NV \l_name_tl \l_id_tl
1333     }
1334     \switch:nNTF { ##3 } \l_path_tl {
1335         \seq_set_split:NnV \l_path_seq { . } \l_path_tl
1336         \seq_remove_all:Nn \l_path_seq { }
1337     } \l__bnvs_DEBUG:x { PATH_SEQ:\l_path_tl==\seq_use:Nn\l_path_seq .}
1338     } {
1339         \seq_clear:N \l_path_seq
1340     }
1341     \l__bnvs_DEBUG:x { PATH_SEQ:\l_path_tl==\seq_use:Nn\l_path_seq .}
1342     \l__bnvs_DEBUG:x { \string ##1 }
1343     ##1 \l_id_tl \l_name_tl \l_path_seq {
1344         \cs_set:Npn \: {
1345             ##4
1346         } \l__bnvs_DEBUG:x {AFTER_ANS~::~\l_ans_tl}
1347     }
1348     } {
1349         \cs_set:Npn \: { \cs_set_eq:NN \loop: \return_false: }
1350     }
1351     \:
1352     \l__bnvs_DEBUG:x {AFTER_AFTER_ANS~::~\l_ans_tl}
1353     }
1354     \cs_set:Npn \:T ##1 {
1355         \seq_if_empty:NTF \l_path_seq { ##1 } {
1356             \cs_set_eq:NN \loop: \return_false:
1357         }
1358     }

```

Main loop.

```

1359     \cs_set:Npn \loop: {
1360         \l__bnvs_DEBUG:x { IF_APPEND_LOOP:\int_use:N\l_split_int /
1361             \seq_count:N \l_split_seq / }
1362         \int_compare:nNnTF {
1363             \l_split_int } < { \seq_count:N \l_split_seq
1364         } {
1365             \switch:nNTF 1 \l_name_tl {

```

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

```

1366         \:NnnT \l__bnvs_resolve_n:NNNTF 2 3 {
1367             \l__bnvs_if_incr:VnNF \l_name_tl 1 \l_ans_tl {
1368                 \cs_set_eq:NN \loop: \return_false:
1369             }
1370         }
1371     } {
1372         \switch:nNTF 4 \l_name_tl {

```

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

```

1373         \switch:nNTF 7 \l_a_tl {
1374             \:NnnT \l__bnvs_resolve:NNNTF 5 6 {
1375                 \:T {
1376                     \l__bnvs_raw_length:VNF \l_name_tl \l_ans_tl {

```

```

1377         \cs_set_eq:NN \loop: \return_false:
1378     }
1379 }
1380 }

    • Case ...length.

1381     } {
1382         \switch:nNTF 8 \l_a_tl {

    • Case ...last.

1383         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1384             \:T {
1385                 \__bnvs_raw_last:VNF \l_name_tl \l_ans_tl {
1386                     \cs_set_eq:NN \loop: \return_false:
1387                 }
1388             }
1389         }

1390     } {
1391         \switch:nNTF 9 \l_a_tl {

    • Case ...next.

1392         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1393             \:T {
1394                 \__bnvs_if_next:VNF \l_name_tl \l_ans_tl {
1395                     \cs_set_eq:NN \loop: \return_false:
1396                 }
1397             }
1398         }
1399     } {
1400         \switch:nNTF { 10 } \l_a_tl {

    • Case ...range.

1401         \:NnnT \__bnvs_resolve:NNNTF 5 6 {
1402             \:T {
1403                 \__bnvs_if_range:VNTF \l_name_tl \l_ans_tl {
1404                     \cs_set_eq:NN \fp_round: \prg_do_nothing:
1405                 } {
1406                     \cs_set_eq:NN \loop: \return_false:
1407                 }
1408             }
1409         }
1410     } {
1411         \switch:nNTF { 11 } \l_a_tl {

    • Case ...n.

1412         \switch:nNTF { 12 } \l_a_tl {

```

- Case ...+=*<integer>*.

```

1413 \:NnnT \_bnvs_resolve_n:NNNTF 5 6 {
1414 \:T {
1415 \_bnvs_DEBUG:x {NAME=\l_name_t1}
1416 \_bnvs_DEBUG:x {INCR=\l_a_t1}
1417 \_bnvs_if_incr:VVNF \l_name_t1 \l_a_t1 \l_ans_t1 {
1418 \cs_set_eq:NN \loop: \return_false:
1419 }
1420 \_bnvs_DEBUG:x {ANS~=\l_ans_t1}
1421 }
1422 \_bnvs_DEBUG:x {ANS~=\l_ans_t1}
1423 }
1424 \_bnvs_DEBUG:x {ANS~=\l_ans_t1}
1425 } {
1426 \:NnnT \_bnvs_resolve_n:NNNTF 5 6 {
1427 \seq_if_empty:NTF \l_path_seq {
1428 \_bnvs_if_counter:VNF \l_name_t1 \l_ans_t1 {
1429 \cs_set_eq:NN \loop: \return_false:
1430 }
1431 } {
1432 \seq_pop_left:NN \l_path_seq \l_a_t1
1433 \seq_if_empty:NTF \l_path_seq {
1434 \_bnvs_if_incr:VVNF \l_name_t1 \l_a_t1 \l_ans_t1 {
1435 \_bnvs_DEBUG:x { INCR~FALSE }
1436 \cs_set_eq:NN \loop: \return_false:
1437 }
1438 \_bnvs_DEBUG:x { INCR~ANS=\l_ans_t1 }
1439 } {
1440 \msg_error:nx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
1441 \cs_set_eq:NN \loop: \return_false:
1442 }
1443 }
1444 }
1445 }
1446 \_bnvs_DEBUG:x {ANS~=\l_ans_t1}

1447 } {
1448 \:NnnT \_bnvs_resolve_n:NNNTF 5 6 {
1449 \seq_if_empty:NTF \l_path_seq {
1450 \_bnvs_if_counter:VNF \l_name_t1 \l_ans_t1 {
1451 \cs_set_eq:NN \loop: \return_false:
1452 }
1453 } {
1454 \seq_pop_left:NN \l_path_seq \l_a_t1
1455 \seq_if_empty:NTF \l_path_seq {
1456 \_bnvs_if_index:VVNF \l_name_t1 \l_a_t1 \l_ans_t1 {
1457 \cs_set_eq:NN \loop: \return_false:
1458 }
1459 } {
1460 \msg_error:nx { beanoves } { :n } { Too~many~.<integer>~components:~#1 }
1461 \cs_set_eq:NN \loop: \return_false:
1462 }
1463 }

```

```

1464         }
1465     }
1466 }
1467 \__bnvs_DEBUG:x {ANS~~\l_ans_tl}
1468 }
1469 \__bnvs_DEBUG:x {ANS~~\l_ans_tl}
1470 }
1471 \__bnvs_DEBUG:x {ANS~~\l_ans_tl}
1472 }
1473 \__bnvs_DEBUG:x {ANS~~\l_ans_tl}
1474 } {

```

No name.

```

1475     }
1476 \__bnvs_DEBUG:x {ANS~~\l_ans_tl}
1477 }
1478 \__bnvs_DEBUG:x {ITERATE~ANS=\l_ans_tl }
1479 \int_add:Nn \l_split_int { 13 }
1480 \tl_put_right:Nx \l_ans_tl {
1481 \seq_item:Nn \l_split_seq { \l_split_int }
1482 }
1483 \__bnvs_DEBUG:x {ITERATE~ANS=\l_ans_tl }
1484 \loop:
1485 } {
1486 \__bnvs_DEBUG:x {END_OF_LOOP~ANS=\l_ans_tl }
1487 \return_true:
1488 }
1489 }
1490 \loop:
1491 } {
1492 \msg_error:nxx { beanoves } { :n } { Too~many~calls:~ #1 }
1493 \prg_return_false:
1494 }
1495 }
1496 \prg_generate_conditional_variant:Nnn
1497 \__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_a_tl</code> Storage for the first index of a range.</p> <p>(End definition for <code>\l_a_tl</code>. This variable is documented on page ??.)</p> <p><code>\l_b_tl</code> Storage for the last index of a range, or its length.</p> <p>(End definition for <code>\l_b_tl</code>. This variable is documented on page ??.)</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> 1498 \regex_const:Nn \c__bnvs_A_cln_Z_regex { 1499 \A \s* (? </pre> <ul style="list-style-type: none"> • 2: <i><first></i> <pre> 1500 ([^:]*) \s* : </pre> <ul style="list-style-type: none"> • 3: second optional colon <pre> 1501 (:)? \s* </pre> <ul style="list-style-type: none"> • 4: <i><length></i> <pre> 1502 ([^:]*) </pre> <ul style="list-style-type: none"> • 5: standalone <i><first></i> <pre> 1503 ([^:]+) 1504) \s* \Z 1505 } </pre> <pre> 1506 \prg_new_conditional:Npnn _bnvs_if_eval_query:nN #1 #2 { T, F, TF } { 1507 _bnvs_DEBUG:x { EVAL_QUERY:#1/ 1508 \tl_to_str:n{#1}/\string#2=\tl_to_str:N #2} 1509 _bnvs_call_reset: 1510 \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex { 1511 #1 1512 } \l_match_seq { 1513 _bnvs_DEBUG:x { EVAL_QUERY:#1/ 1514 \string\l_match_seq/\seq_use:Nn \l_match_seq //} 1515 \bool_set_false:N \l__bnvs_no_counter_bool 1516 \bool_set_false:N \l__bnvs_no_range_bool </pre> <hr/> <p><u>\switch:nNTF</u></p> <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>
----------------------------------	--

```

1517     \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1518     \__bnvs_DEBUG:x { EQ_SWITCH:##1/ }
1519         \tl_set:Nx ##2 {
1520             \seq_item:Nn \l_match_seq { ##1 }
1521         }
1522     \__bnvs_DEBUG:x { \string ##2/ \tl_to_str:N ##2/}
1523         \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1524     }
1525     \switch:nNTF 5 \l_a_tl {

```

Single expression

```

1526     \bool_set_false:N \l__bnvs_no_range_bool
1527     \__bnvs_if_append:VNTF \l_a_tl #2 {
1528         \prg_return_true:
1529     } {
1530         \prg_return_false:
1531     }
1532 } {
1533     \switch:nNTF 2 \l_a_tl {
1534         \switch:nNTF 4 \l_b_tl {
1535             \switch:nNTF 3 \l_c_tl {

```

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

```

1536         \__bnvs_if_append:VNTF \l_a_tl #2 {
1537             \tl_put_right:Nn #2 { - }
1538             \__bnvs_if_append:VNTF \l_b_tl #2 {
1539                 \prg_return_true:
1540             } {
1541                 \prg_return_false:
1542             }
1543         } {
1544             \prg_return_false:
1545         }
1546     } {

```

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

```

1547         \__bnvs_if_append:VNTF \l_a_tl #2 {
1548             \tl_put_right:Nx #2 { - }
1549             \tl_put_right:Nx \l_a_tl { + ( \l_b_tl ) - 1 }
1550             \__bnvs_if_append:VNTF \l_a_tl #2 {
1551                 \prg_return_true:
1552             } {
1553                 \prg_return_false:
1554             }
1555         } {
1556             \prg_return_false:
1557         }
1558     }
1559 } {

```

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

```

1560         \__bnvs_if_append:VNTF \l_a_tl #2 {
1561             \tl_put_right:Nn #2 { - }
1562             \prg_return_true:
1563         } {

```



```

1564         \prg_return_false:
1565     }
1566 }
1567 } {
1568     \switch:nNTF 4 \l_b_tl {
1569         \switch:nNTF 3 \l_c_tl {
1570             \tl_put_right:Nn #2 { - }
1571             \__bnvs_if_append:VNTF \l_a_tl #2 {
1572                 \prg_return_true:
1573             } {
1574                 \prg_return_false:
1575             }
1576         } {
1577             \msg_error:nxx { beanoves } { :n } { Syntax-error(Missing-first):~#1 }
1578         }
1579     } {
1580         \seq_put_right:Nn #2 { - }
1581     }
1582 }
1583 }
1584 } {
Error
1585     \msg_error:nnn { beanoves } { :n } { Syntax-error:~#1 }
1586 }
1587 }

```

`__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_query_seq` Storage for a sequence of *<query>*'s obtained by splitting a comma separated list.

(End definition for `\l_query_seq`. This variable is documented on page ??.)

`\l_ans_seq` Storage of the evaluated result.

(End definition for `\l_ans_seq`. This variable is documented on page ??.)

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

```
1588 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }
```

(End definition for `\c__bnvs_comma_regex`.)

No other variable is used.

```

1589 \cs_new:Npn \__bnvs_eval:nN #1 #2 {
1590     \__bnvs_DEBUG:x {\string\__bnvs_eval:nN:\tl_to_str:n{#1}/\string#2=\tl_to_str:V #2}
1591     \__bnvs_group_begin:

```

Local variables declaration

```

1592 \_bnvs_DEBUG:x { WILL_CLEAN }
1593 \seq_if_exist:NTF \l_query_seq {
1594   \seq_clear:N \l_query_seq
1595 } {
1596   \seq_new:N \l_query_seq
1597 }
1598 \seq_if_exist:NTF \l_ans_seq {
1599   \seq_clear:N \l_ans_seq
1600 } {
1601   \seq_new:N \l_ans_seq
1602 }

```

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 $\backslash l_query_seq$.

```

1603 \regex_split:NnN \c__bnvs_comma_regex { #1 } \l_query_seq

```

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

```

1604 \seq_map_inline:Nn \l_query_seq {
1605   \tl_clear:N \l_ans_tl
1606   \_bnvs_if_eval_query:nNTF { ##1 } \l_ans_tl {
1607     \seq_put_right:NV \l_ans_seq \l_ans_tl
1608   } {
1609     \seq_map_break:n {
1610       \msg_fatal:nnn { beanoves } { :n } { Circular~dependency~in~#1}
1611     }
1612   }
1613 }

```

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

```

1614 \exp_args:NNNx
1615 \_bnvs_group_end:
1616 \tl_put_right:Nn #2 { \seq_use:Nn \l_ans_seq , }
1617 }
1618 \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.

```

1619 \NewDocumentCommand \BeanovesEval { s o m } {
1620   \_bnvs_group_begin:
1621   \tl_clear:N \l_ans_tl
1622   \IfBooleanTF { #1 } {
1623     \bool_set_true:N \l__bnvs_no_counter_bool
1624   } {
1625     \bool_set_false:N \l__bnvs_no_counter_bool

```

```

1626 }
1627 \__bnvs_eval:nN { #3 } \l_ans_tl
1628 \IfValueTF { #2 } {
1629   \exp_args:NNNV
1630   \__bnvs_group_end:
1631   \tl_set:Nn #2 \l_ans_tl
1632 } {
1633   \exp_args:NV
1634   \__bnvs_group_end: \l_ans_tl
1635 }
1636 }

```

5.5.9 Resetting slide ranges

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

```

1637 \NewDocumentCommand \BeanovesReset { 0{1} m } {
1638   \__bnvs_reset:nn { #1 } { #2 }
1639   \ignorespaces
1640 }

```

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.

```

1641 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1642   \bool_if:nTF {
1643     \__bnvs_if_in_p:n { #2/A } || \__bnvs_if_in_p:n { #2/Z }
1644   } {
1645     \__bnvs_gremove:n { #2/C }
1646     \__bnvs_gremove:n { #2//A }
1647     \__bnvs_gremove:n { #2//L }
1648     \__bnvs_gremove:n { #2//Z }
1649     \__bnvs_gremove:n { #2//N }
1650     \__bnvs_gput:nn { #2/C0 } { #1 }
1651   } {
1652     \msg_warning:nnn { beanoves } { :n } { Unknown~name:~#2 }
1653   }
1654 }

1655 \makeatother
1656 \ExplSyntaxOff
1657 </package>

```