beamer named overlay specifications with beanoves

Jérôme Laurens

v1.0   2022/10/28

Abstract

This package allows the management of multiple named slide number sets in beamer documents. Named slide number sets are very handy both during edition and to manage complex and variable beamer overlay specifications. In particular, they allow to replace raw numbers in beamer `<...>` overlay specifications by logical identifiers. Demonstration files are available online as part of the development repository.

Contents

1   Minimal example

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

```
1  \documentclass {beamer}
2  \RequirePackage {beanoves}
3  \begin{document}
4  \Beanoves {
5      A = 1:2,
6      B = A.next:3,
7      C = B.next,
8    }
9  \begin{frame}
10 {\Large Frame \insertframenumber}
11 {\Large Slide \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.next)-> {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 overlay sets*. On line 5, we declare an overlay set named 'A', which is a range starting at slide 1 and with length 2. On line 12, the extended *named overlay specification* `?(A.1)` stands for 1 because 1 is the first index of the overlay set named A. On line 15, `?(A.2)` stands for 2 whereas on line 18, `?(A.next)` stands for 3. On line 6, we declare a second overlay set 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 overlay sets

2.1 Presentation

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

2.2 Named overlay reference

`A.1`, `C.2` are *named overlay references*, as well as `A` and `Y!C.2`. More precisely, they are string identifiers, each one representing a well defined static integer to be used in `beamer` overlay specifications. They can take one of the next forms.

$\langle$**short name**$\rangle$: like `A` and `C`,

$\langle$**frame id**$\rangle$!$\langle$**short name**$\rangle$: denoted by *qualified names*, like `X!A` and `Y!C`.

$\langle$**short name**$\rangle$$\langle$**dotted path**$\rangle$: denoted by *full names* like `A.1` and `C.2`,

$\langle$**frame id**$\rangle$!$\langle$**short name**$\rangle$$\langle$**dotted path**$\rangle$: denoted by *qualified full names* like `X!A.1` and `Y!C.2`.

The *short names* and *frame ids* are alphanumerical case sensitive identifiers, with possible underscores but no space nor leading digit. Unicode symbols above `U+00A0` are allowed if the underlying TeX engine supports it. Identifiers consisting only of lowercase letters and underscores are reserved by the package.

The *dotted path* is a string `.`$\langle$**component**$_1$$\rangle$`.`$\langle$**component**$_2$$\rangle$`.`...`.`$\langle$**component**$_n$$\rangle$, where each $\langle$*component$_i$*$\rangle$ denotes either an integer, eventually signed, or a $\langle$*short name*$\rangle$. The *dotted path* can be empty for which `n` is 0.

The mapping from *named overlay references* to integers is defined at the global TeX level to allow its use in `\begin{frame}<...>` and to share the same overlay sets between different frames. Hence the *frame id* due to the need to possibly target a particular frame.

2.3 Defining named overlay sets

In order to define *named overlay sets*s, we can either execute the next `\Beanoves` command 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`   `beanoves = {`$\langle$*ref$_1$*$\rangle$`=`$\langle$*spec$_1$*$\rangle$`, `$\langle$*ref$_2$*$\rangle$`=`$\langle$*spec$_2$*$\rangle$`,...,` $\langle$*ref$_n$*$\rangle$`=`$\langle$*spec$_n$*$\rangle$`}`

`\Beanoves`   `\Beanoves{`$\langle$*ref$_1$*$\rangle$`=`$\langle$*spec$_1$*$\rangle$`, `$\langle$*ref$_2$*$\rangle$`=`$\langle$*spec$_2$*$\rangle$`,...,` $\langle$*ref$_n$*$\rangle$`=`$\langle$*spec$_n$*$\rangle$`}`

Each $\langle$*ref*$\rangle$ key is a *named overlay reference* whereas each $\langle$*spec*$\rangle$ value is an *overlay set specifier*. When the same $\langle$*ref*$\rangle$ key is used multiple times, only the last one is taken

into account. Possible $\langle spec \rangle$ value are the *range specifiers*

$\quad$ `⟨first⟩`, `⟨first⟩:` and `⟨first⟩::`, `⟨first⟩:⟨length⟩`, `⟨first⟩::⟨last⟩`,

$\quad$ `:⟨length⟩::⟨last⟩` and `::⟨last⟩:⟨length⟩`.

Here $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$ are algebraic expression possibly involving any *named overlay reference* defined above. At least one of $\langle first \rangle$ or $\langle last \rangle$ must be provided.

$\quad$ When performed at the document level, the `\Beanoves` command starts by cleaning what was set by previous calls. When performed inside LaTeX environments, each call cumulates with the previous. Notice that the argument of this function can contain macros: they will be exhaustively expanded.

$\quad$ Also possible values are *list specifiers* which are comma separated lists of $\langle ref \rangle=\langle spec \rangle$ definitions. The definition

$\quad$ `⟨key⟩=[⟨ref₁⟩=⟨spec₁⟩, ⟨ref₂⟩=⟨spec₂⟩,..., ⟨refₙ⟩=⟨specₙ⟩]`

is a convenient shortcut for

$\quad$ `⟨key⟩.⟨ref₁⟩=⟨value₁⟩`,

$\quad$ `⟨key⟩.⟨ref₂⟩=⟨value₂⟩`,

$\quad$ `...`,

$\quad$ `⟨key⟩.⟨refₙ⟩=⟨valueₙ⟩`.

The rules above can apply individually to each line.

$\quad$ To support an array syntax, we can omit the $\langle ref \rangle$ key. The first missing key is replaced by 1, the second by 2, and so on.

3 Named overlay resolution

Turning a *named overlay reference* into the static integer it represents, as in `<?(A.1)>` above is denoted *named overlay resolution* or simply *resolution*. This section is devoted to *resolution rules* depending on the definition of the named overlay set. Here $\langle i \rangle$ denotes an integer whereas $\langle first \rangle$, $\langle length \rangle$ and $\langle last \rangle$ stand for integers, or integer valued expressions.

3.1 Simple definitions

`⟨name⟩ = ⟨first⟩` For an unlimited range

reference	resolution
`⟨name⟩.1`	$\langle first \rangle$
`⟨name⟩.2`	$\langle first \rangle + 1$
`⟨name⟩.⟨i⟩`	$\langle first \rangle + \langle i \rangle - 1$

`⟨name⟩ = ⟨first⟩:` as well as `⟨first⟩::`. For a range limited from below:

reference	resolution
`⟨name⟩.1`	$\langle first \rangle$
`⟨name⟩.2`	$\langle first \rangle + 1$
`⟨name⟩.⟨i⟩`	$\langle first \rangle + \langle i \rangle - 1$
`⟨name⟩.previous`	$\langle first \rangle - 1$

`⟨name⟩ = ::⟨last⟩` For a range limited from above:

reference	resolution
$\langle\mathit{name}\rangle$.1	$\langle\mathit{last}\rangle$
$\langle\mathit{name}\rangle$.0	$\langle\mathit{last}\rangle - 1$
$\langle\mathit{name}\rangle$.$\langle\mathit{i}\rangle$	$\langle\mathit{last}\rangle + \langle\mathit{i}\rangle - 1$
$\langle\mathit{name}\rangle$.next	$\langle\mathit{last}\rangle + 1$

$\langle\mathit{name}\rangle$ = $\langle\mathit{first}\rangle$:$\langle\mathit{length}\rangle$ as well as variants $\langle\mathit{first}\rangle$::$\langle\mathit{last}\rangle$, :$\langle\mathit{length}\rangle$::$\langle\mathit{last}\rangle$ or ::$\langle\mathit{last}\rangle$:$\langle\mathit{length}\rangle$, which are equivalent provided $\langle\mathit{first}\rangle + \langle\mathit{length}\rangle = \langle\mathit{last}\rangle + 1$.

For a range limited from both above and below:

reference	resolution
$\langle\mathit{name}\rangle$.1	$\langle\mathit{first}\rangle$
$\langle\mathit{name}\rangle$.2	$\langle\mathit{first}\rangle + 1$
$\langle\mathit{name}\rangle$.$\langle\mathit{i}\rangle$	$\langle\mathit{first}\rangle + \langle\mathit{i}\rangle - 1)$
$\langle\mathit{name}\rangle$.previous	$\langle\mathit{first}\rangle - 1$
$\langle\mathit{name}\rangle$.last	$\langle\mathit{last}\rangle$
$\langle\mathit{name}\rangle$.next	$\langle\mathit{last}\rangle + 1$
$\langle\mathit{name}\rangle$.length	$\langle\mathit{length}\rangle$
$\langle\mathit{name}\rangle$.range	$\max(0, \langle\mathit{first}\rangle)$ ''-'' $\max(0, \langle\mathit{last}\rangle)$

Notice that the resolution of $\langle\mathit{name}\rangle$.range is not an algebraic difference, and negative integers do not make sense there while in beamer context.

For example

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

3.2 Counters

For each named overlay set defined, we have an implicit index counter starting at 1, its actual value is an integer denoted $\langle n \rangle$. The $\langle\mathit{name}\rangle$.n *named counter reference* is resolved into $\langle\mathit{name}\rangle$.$\langle n \rangle$, which in turn is resolved according to the preceding rules.

Each named overlay set defined also has a dedicated value counter which is some kind of variable that can be used and incremented. A simple $\langle\mathit{name}\rangle$ reference is resolved into the position of this counter, which is the value of the counter bounded from below by the lowest value of the range and from above by the largest value of the range. For unbounded ranges, these values may be infinite.

Additionnaly, resolution rules are provided for the *named index references*:

$\langle \textbf{name} \rangle.\textbf{n+=}\langle \textit{integer expression} \rangle$: resolve $\langle \textit{integer expression} \rangle$ into $\langle \textit{integer} \rangle$, advance the implicit index counter associate to $\langle \textit{name} \rangle$ by $\langle \textit{integer} \rangle$ and use the resolution of $\langle \textbf{name} \rangle.\textbf{n}$.

Here $\langle \textit{integer expression} \rangle$ denotes the longest character sequence with no space[1].

$\langle \textbf{name} \rangle.\textbf{++n}$ as well as $\textbf{++}\langle \textbf{name} \rangle.\textbf{n}$: advance the implicit index counter associate to $\langle \textit{name} \rangle$ by 1 and use the resolution of $\langle \textbf{name} \rangle.\textbf{n}$,

$\langle \textbf{name} \rangle.\textbf{n++}$: use the resolution of $\langle \textbf{name} \rangle.\textbf{n}$ and increment the implicit index counter associate to $\langle \textit{name} \rangle$ by 1.

We have resolution rules as well for the *named value references*:

$\langle \textbf{name} \rangle\textbf{+=}\langle \textit{integer expression} \rangle$: resolve $\langle \textit{integer expression} \rangle$ into $\langle \textit{integer} \rangle$, advance the value counter by $\langle \textit{integer} \rangle$ and use the new position. Here again, $\langle \textit{integer expression} \rangle$ is the longest character sequence with no space.

$\textbf{++}\langle \textbf{name} \rangle$: advance the value counter for $\langle \textit{name} \rangle$ by 1 and use the new position.

$\langle \textbf{name} \rangle\textbf{++}$: use the actual position and advance the value counter for $\langle \textit{name} \rangle$ by 1.

For example both `?(A.next)`, `?(A.last+1)`, `?(A.1+A.length)` give the same result as soon as the slide range named '`A`' has been properly defined with a starting value and a length.

In order to decrement a counter, one can increment with a negative value, no dedicated syntax is provided yet.

3.3   Dotted paths

$\langle \textbf{name} \rangle.\langle \textbf{i} \rangle \textbf{ = } \langle \textbf{range spec} \rangle$ All the preceding rules are overriden by this particular one and $\langle \textbf{name} \rangle.\langle \textbf{i} \rangle$ resolves to the resolution of $\langle \textit{range spec} \rangle$.

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,
3    A.3 = 0,
4  }
5  \begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
6  \ttfamily
7  \BeanovesEval(A.1) == 3,
8  \BeanovesEval(A.2) == 4,
9  \BeanovesEval(A.-1)== 1,
10 \BeanovesEval(A.3) == 0,
11 \end{frame}
```

$\langle \textbf{name} \rangle.\langle \textbf{c}_1 \rangle.\langle \textbf{c}_2 \rangle\ldots\langle \textbf{c}_k \rangle \textbf{ = } \langle \textbf{range spec} \rangle$ When a dotted path has more than one component, a *named overlay reference* like `A.1.2` needs some well defined resolution rule to avoid ambiguity. To resolve one level of such a reference $\langle \textbf{name} \rangle.\langle \textbf{c}_1 \rangle.\langle \textbf{c}_2 \rangle\ldots\langle \textbf{c}_n \rangle$, we replace the longest $\langle \textbf{name} \rangle.\langle \textbf{c}_1 \rangle.\langle \textbf{c}_2 \rangle\ldots\langle \textbf{c}_k \rangle$ where

---

[1]The parser for algebraic expression is very rudimentary.

$0 \leq k \leq n$ by its definition $\langle \textbf{\textit{name'}} \rangle . \langle \textbf{\textit{c'}}_1 \rangle \ldots \langle \textbf{\textit{c'}}_p \rangle$ if any (the path can be empty). beanoves uses this one level resolution as many times as possible, but no more than a predefined limit to catch circular reference that would lead to an infinite TeX loop. One final resolution occurs with rules above if possible or an error is raised.

For a *named indexed reference* like $\langle \textbf{\textit{name}} \rangle . \langle \textbf{\textit{c}}_1 \rangle . \langle \textbf{\textit{c}}_2 \rangle \ldots \langle \textbf{\textit{c}}_n \rangle . \texttt{n}$, we must first resolve $\langle \textbf{\textit{name}} \rangle . \langle \textbf{\textit{c}}_1 \rangle . \langle \textbf{\textit{c}}_2 \rangle \ldots \langle \textbf{\textit{c}}_n \rangle$ into $\langle \textbf{\textit{name'}} \rangle$ with an empty dotted path, then retrieve the value of $\langle \textbf{\textit{name'}} \rangle . \texttt{n}$ denoted as $\langle n' \rangle$ and finally use the resolved $\langle \textbf{\textit{name}} \rangle . \langle \textbf{\textit{c}}_1 \rangle . \langle \textbf{\textit{c}}_2 \rangle \ldots \langle \textbf{\textit{c}}_n \rangle . \langle \textbf{\textit{n'}} \rangle$.

3.4 Frame id

Except for very special situations, the *frame ids* can be left unspecified. When no *frame id* was explicitly provided, beanoves uses the *last frame id*. At the beginning of each frame, the *last frame id* is set to the *frame id* of the current frame, which is denoted *current frame id* and defaults to `?`. Then it gets updated after each named reference resolution. For example, the first time `A.1` reference is resolved within a given frame, it is first translated to $\langle \textbf{\textit{current frame id}} \rangle \texttt{!A.1}$, but when used just after `Y!C.2`, it becomes a shortcut to `Y!A.1` because the *last frame id* was then `Y`.

In order to set the *frame id* of the current frame to $\langle \textit{frame id} \rangle$, use the new beanoves `id` option of the beamer frame environment.

beanoves id

`beanoves id=`$\langle \textit{frame id} \rangle$`,`

We can use the same *frame id* for different frames to share named overlay sets.

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 '?($\langle \textbf{\textit{queries}} \rangle$)'. Each one is then evaluated and replaced by its resolved static counterpart. The overall result is finally forwarded to the beamer class.

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

query	resolution	limitation
$\langle \textbf{\textit{first expr}} \rangle$	$\langle \textit{first} \rangle$	
$\langle \textbf{\textit{first expr}} \rangle \texttt{:}$	$\langle \textit{first} \rangle$ –	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\langle \textbf{\textit{first expr}} \rangle \texttt{:} \langle \textbf{\textit{length expr}} \rangle$	$\langle \textit{first} \rangle$ – $\langle \textit{last} \rangle$	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\texttt{::} \langle \textbf{\textit{end expr}} \rangle \texttt{:} \langle \textbf{\textit{length expr}} \rangle$	$\langle \textit{first} \rangle$ – $\langle \textit{last} \rangle$	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\texttt{:}$	–	
$\langle \textbf{\textit{first expr}} \rangle \texttt{::}$	$\langle \textit{first} \rangle$ –	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\texttt{::} \langle \textbf{\textit{end expr}} \rangle$	– $\langle \textit{last} \rangle$	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\langle \textbf{\textit{first expr}} \rangle \texttt{::} \langle \textbf{\textit{end expr}} \rangle$	$\langle \textit{first} \rangle$ – $\langle \textit{last} \rangle$	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\texttt{:} \langle \textbf{\textit{length expr}} \rangle \texttt{::} \langle \textbf{\textit{end expr}} \rangle$	$\langle \textit{first} \rangle$ – $\langle \textit{last} \rangle$	no $\langle \textbf{\textit{name}} \rangle . \texttt{range}$
$\texttt{::}$	–	

Here ⟨*first expr*⟩, ⟨*length expr*⟩ and ⟨*end expr*⟩ both denote algebraic expressions possibly involving named slide references and counters. As integers, they are respectively resolved into ⟨*first*⟩, ⟨*length*⟩ and ⟨*last*⟩.

Notice that nesting `?(...)` query expressions is not supported.

5 Support

See .

6 Implementation

Identify the internal prefix (LaTeX3 DocStrip convention).

```
1 ⟨@@=bnvs⟩
```

Reserved namespace: identifiers containing the case insensitive string `beanoves` or the string `bnvs` delimited by two non characters. Not all the variables or functions names used by this package follow this convention, but in that case the global macro level is not polluted.

6.1 Package declarations

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

6.2 logging

Utility message.

```
8  \msg_new:nnn { beanoves } { :n } { #1 }
9  \msg_new:nnn { beanoves } { :nn } { #1~(#2) }
10 \cs_new:Npn \__bnvs_warning:n {
11   \msg_warning:nnn { beanoves } { :n }
12 }
13 \cs_new:Npn \__bnvs_error:n {
14   \msg_error:nnn { beanoves } { :n }
15 }
16 \cs_new:Npn \__bnvs_error:x {
17   \msg_error:nnx { beanoves } { :n }
18 }
19 \cs_new:Npn \__bnvs_fatal:n {
20   \msg_fatal:nnn { beanoves } { :n }
21 }
22 \cs_new:Npn \__bnvs_fatal:x {
23   \msg_fatal:nnx { beanoves } { :n }
24 }
```

6.3 Debugging and testing facilities

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

6.4 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 LaTeX3 naming convention: we do not specialize with the module name. On execution, next initialization instructions declare the variables as side effect.

```
25 \tl_new:N \l__bnvs_id_last_tl
26 \tl_set:Nn \l__bnvs_id_last_tl { ?! }
27 \tl_new:N \l__bnvs_a_tl
28 \tl_new:N \l__bnvs_b_tl
29 \tl_new:N \l__bnvs_c_tl
30 \tl_new:N \l__bnvs_id_tl
31 \tl_new:N \l__bnvs_ans_tl
32 \tl_new:N \l__bnvs_name_tl
33 \tl_new:N \l__bnvs_path_tl
34 \tl_new:N \l__bnvs_group_tl
35 \tl_new:N \l__bnvs_query_tl
36 \tl_new:N \l__bnvs_token_tl
37 \tl_new:N \l__bnvs_root_tl
38 \int_new:N \g__bnvs_call_int
39 \int_new:N \l__bnvs_int
40 \seq_new:N \g__bnvs_def_seq
41 \seq_new:N \l__bnvs_a_seq
42 \seq_new:N \l__bnvs_b_seq
43 \seq_new:N \l__bnvs_ans_seq
44 \seq_new:N \l__bnvs_match_seq
45 \seq_new:N \l__bnvs_split_seq
46 \seq_new:N \l__bnvs_path_seq
47 \seq_new:N \l__bnvs_query_seq
48 \seq_new:N \l__bnvs_token_seq
49 \bool_new:N \l__bnvs_in_frame_bool
50 \bool_new:N \l__bnvs_parse_bool
51 \bool_set_false:N \l__bnvs_in_frame_bool
```

6.5 Infinite loop management

Unending recursivity is managed here.

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

(*End definition for* `\g__bnvs_call_int`.)

```
52 \int_const:Nn \c__bnvs_max_call_int { 2048 }
```

__bnvs_call_greset:

__bnvs_call_greset:

Reset globally the call stack counter to its maximum value.

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

__bnvs_call:_TF_

__bnvs_call_do:TF {⟨ *true code* ⟩} {⟨ *false code* ⟩}

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

```
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 }
```

6.6 Overlay specification

6.6.1 In slide range definitions

\g__bnvs_prop

⟨*key*⟩–⟨*value*⟩ property list to store the named overlay sets. The basic keys are, assuming ⟨*id*⟩!⟨*name*⟩ is a fully qualified overlay set name,

⟨**id**⟩!⟨**name**⟩/A for the first index

⟨**id**⟩!⟨**name**⟩/L for the length when provided

⟨**id**⟩!⟨**name**⟩/Z for the last index when provided

⟨**id**⟩!⟨**name**⟩/V for the counter value, when used

⟨**id**⟩!⟨**name**⟩/n for the implicit index counter, when used.

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

⟨**id**⟩!⟨**name**⟩//A for the cached static value of the first index

⟨**id**⟩!⟨**name**⟩//Z for the cached static value of the last index

⟨**id**⟩!⟨**name**⟩//L for the cached static value of the length

⟨**id**⟩!⟨**name**⟩//P for the cached static value of the previous index

⟨**id**⟩!⟨**name**⟩//N for the cached static value of the next index

⟨**id**⟩!⟨**name**⟩//V for the real counter value

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.)

6.7 Basic functions

__bnvs_gput:nnn
__bnvs_gput:nnV
__bnvs_gprovide:nnn
__bnvs_gprovide:nVn
__bnvs_item:nn
__bnvs_gremove:nn
__bnvs_gremove:nV
__bnvs_gclear:n
__bnvs_gclear:

__bnvs_gput:nnn {⟨*subkey*⟩} {⟨*key*⟩} {⟨*value*⟩}
__bnvs_gprovide:nnn {⟨*subkey*⟩} {⟨*key*⟩} {⟨*value*⟩}
__bnvs_item:nn {⟨*subkey*⟩} {⟨*key*⟩}
__bnvs_get:nnN {⟨*subkey*⟩} {⟨*key*⟩} ⟨*tl variable*⟩
__bnvs_gremove:nn {⟨*subkey*⟩} {⟨*key*⟩}
__bnvs_clear:n {⟨*key*⟩}
__bnvs_clear:

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over LaTeX3 eponym functions, except __bnvs_gprovide:nn which meaning is straightforward. The key argument is ⟨*key*⟩/⟨*subkey*⟩.

```
65 \cs_new:Npn \__bnvs_gput:nnn #1 #2 {
66   \prop_gput:Nnn \g__bnvs_prop { #2 / #1 }
67 }
68 \cs_new:Npn \__bnvs_gprovide:nnn #1 #2 #3 {
69   \prop_if_in:NnF \g__bnvs_prop { #2 / #1 } {
70     \prop_gput:Nnn \g__bnvs_prop { #2 / #1 } { #3 }
71   }
72 }
73 \cs_new:Npn \__bnvs_item:nn #1 #2 {
74   \prop_item:Nn \g__bnvs_prop { #2 / #1 }
75 }
76 \cs_new:Npn \__bnvs_gremove:nn  #1 #2 {
77   \prop_gremove:Nn \g__bnvs_prop { #2 / #1 }
78 }
79 \cs_new:Npn \__bnvs_gclear:n #1 {
80   \__bnvs_gremove:nn {} { #1 }
81   \clist_map_inline:nn { A, L, Z, V, n } {
82     \__bnvs_gremove:nn { ##1 } { #1 }
83   }
84   \__bnvs_gclear_cache:n { #1 }
85 }
86 \cs_new:Npn \__bnvs_gclear: {
87   \prop_gclear:N \g__bnvs_prop
88 }
89 \cs_generate_variant:Nn \__bnvs_gput:nnn { nnV }
90 \cs_generate_variant:Nn \__bnvs_gremove:nn { nV }
```

__bnvs_if_in_p:nn ⋆
__bnvs_if_in_p:nV ⋆
__bnvs_if_in:nn*TF* ⋆
__bnvs_if_in:nV*TF* ⋆

__bnvs_if_in_p:nn {⟨*subkey*⟩} {⟨*key*⟩}
__bnvs_if_in:nnTF {⟨*subkey*⟩} {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Convenient shortcuts to test for the existence of ⟨*subkey*⟩/⟨*key*⟩, it makes the code more concise and readable.

```
91 \prg_new_conditional:Npnn \__bnvs_if_in:nn #1 #2 { p, T, F, TF } {
92   \prop_if_in:NnTF \g__bnvs_prop { #2 / #1 } {
93     \prg_return_true:
94   } {
95     \prg_return_false:
96   }
97 }
```

```
98 \prg_generate_conditional_variant:Nnn
99   \__bnvs_if_in:nn {nV} { p, T, F, TF }
```

__bnvs_get:nnN*TF*   __bnvs_get:nnNTF {⟨*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.

```
100 \prg_new_conditional:Npnn \__bnvs_get:nnN #1 #2 #3 { p, T, F, TF } {
101   \prop_get:NnNTF \g__bnvs_prop { #2 / #1 } #3 {
102     \prg_return_true:
103   } {
104     \prg_return_false:
105   }
106 }
107 \prg_generate_conditional_variant:Nnn
108   \__bnvs_get:nnN {nV} { p, T, F, TF }
```

6.8 Functions with cache

`\__bnvs_gput_cache:nnn`	`\__bnvs_gput_cache:nnn {⟨subkey⟩} {⟨key⟩} {⟨value⟩}`
`\__bnvs_gput_cache:(nnV|nVn)`	`\__bnvs_item_cache:nn {⟨subkey⟩} {⟨key⟩}`
`\__bnvs_item_cache:nn`	`\__bnvs_gremove_cache:nn {⟨subkey⟩} {⟨key⟩}`
`\__bnvs_gremove_cache:nn`	`\__bnvs_clear_cache:n {⟨key⟩}`
`\__bnvs_gremove_cache:nV`	
`\__bnvs_gclear_cache:n`	
`\__bnvs_v_gclear:`	

Wrapper over the functions above for ⟨*key*⟩/⟨*subkey*⟩.

```
109 \cs_new:Npn \__bnvs_gput_cache:nnn #1 {
110   \__bnvs_gput:nnn { / #1 }
111 }
112 \cs_new:Npn \__bnvs_item_cache:nn #1 #2 {
113   \prop_item:Nn \g__bnvs_prop { #2 / / #1 }
114 }
115 \cs_new:Npn \__bnvs_gremove_cache:nn  #1 {
116   \__bnvs_gremove:nn { / #1 }
117 }
118 \cs_new:Npn \__bnvs_gclear_cache:n #1 {
119   \clist_map_inline:nn { {}, A, L, Z, P, N, V } {
120     \__bnvs_gremove_cache:nn { ##1 } { #1 }
121   }
122 }
123 \cs_new:Npn \__bnvs_v_gclear: {
124   \__bnvs_group_begin:
125   \seq_clear:N \l__bnvs_a_seq
126   \prop_map_inline:Nn \g__bnvs_prop {
127     \regex_match:nnT { //V$ } { ##1 } {
128       \seq_put_right:Nn \l__bnvs_a_seq { ##1 }
129     }
130   }
131   \seq_map_inline:Nn \l__bnvs_a_seq {
132     \prop_gremove:Nn \g__bnvs_prop { ##1 }
133   }
134   \__bnvs_group_end:
135 }
136 \cs_generate_variant:Nn \__bnvs_gremove_cache:nn { nV }
137 \cs_generate_variant:Nn \__bnvs_gput_cache:nnn { nVn, nnV }
```

`\__bnvs_if_in_cache_p:nn` ⋆	`\__bnvs_if_in_cache_p:n {⟨subkey⟩} {⟨key⟩}`
`\__bnvs_if_in_cache_p:nV` ⋆	`\__bnvs_if_in_cache:nTF {⟨subkey⟩} {⟨key⟩} {⟨true code⟩} {⟨false code⟩}`
`\__bnvs_if_in_cache:nnTF` ⋆	
`\__bnvs_if_in_cache:nVTF` ⋆	

Convenient shortcuts to test for the existence of ⟨*subkey*⟩/⟨*key*⟩, it makes the code more concise and readable.

```
138 \prg_new_conditional:Npnn \__bnvs_if_in_cache:nn #1 #2 { p, T, F, TF } {
139   \__bnvs_if_in:nnTF { / #1 } { #2 } {
140     \prg_return_true:
141   } {
142     \prg_return_false:
143   }
```

```
144 }
145 \prg_generate_conditional_variant:Nnn
146   \__bnvs_if_in_cache:nn {nV} { p, T, F, TF }
```

__bnvs_get_cache:nnN*TF*

__bnvs_get_cache:nnNTF {⟨*subkey*⟩} {⟨*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.

```
147 \prg_new_conditional:Npnn \__bnvs_get_cache:nnN #1 #2 #3 { p, T, F, TF } {
148   \__bnvs_get:nnNTF { / #1 } { #2 } #3 {
149     \prg_return_true:
150   } {
151     \prg_return_false:
152   }
153 }
154 \prg_generate_conditional_variant:Nnn
155   \__bnvs_get_cache:nnN {nV} { p, T, F, TF }
```

6.8.1 Implicit index counter

The implicit index counter is local to the current frame. When used for the first time, it defaults to 1.

\g__bnvs_n_prop

⟨*key*⟩–⟨*value*⟩ property list to store the named slide lists. The keys are ⟨*id*⟩!⟨*name*⟩.

```
156 \prop_new:N \g__bnvs_n_prop
```

(*End definition for* \g__bnvs_n_prop.)

__bnvs_n_gput:nn
__bnvs_n_gput:(nV|Vn)
__bnvs_n_item:n
__bnvs_n_gremove:n
__bnvs_n_gclear:

__bnvs_n_gput:nn {⟨*key*⟩} {⟨*value*⟩}
__bnvs_n_item:n {⟨*key*⟩}
__bnvs_n_gremove:n {⟨*key*⟩}
__bnvs_n_gclear:

Convenient shortcuts to manage the storage, it makes the code more concise and readable. This is a wrapper over LaTeX3 eponym functions.

```
157 \cs_new:Npn \__bnvs_n_gput:nn {
158   \prop_gput:Nnn \g__bnvs_n_prop
159 }
160 \cs_new:Npn \__bnvs_n_item:n #1 {
161   \prop_item:Nn \g__bnvs_n_prop { #1 }
162 }
163 \cs_new:Npn \__bnvs_n_gremove:n {
164   \prop_gremove:Nn \g__bnvs_n_prop
165 }
166 \cs_new:Npn \__bnvs_n_gclear: {
167   \prop_gclear:N \g__bnvs_n_prop
168 }
```

14

__bnvs_n_get:nN*TF*   __bnvs_n_get:nNTF {⟨*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.

```
169 \prg_new_conditional:Npnn \__bnvs_n_get:nN #1 #2 { T, F, TF } {
170   \prop_get:NnNTF \g__bnvs_n_prop { #1 } #2 {
171     \prg_return_true:
172   } {
173     \prg_return_false:
174   }
175 }
176
```

6.8.2 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.

```
177 \regex_const:Nn \c__bnvs_name_regex {
178   [[:alpha:]_][[:alnum:]_]*
179 }
```

(*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.

```
180 \regex_const:Nn \c__bnvs_id_regex {
181   (?: \ur{c__bnvs_name_regex} | [?] )? !
182 }
```

(*End definition for* \c__bnvs_id_regex.)

\c__bnvs_path_regex   A sequence of .⟨*positive integer*⟩ items representing a path.

```
183 \regex_const:Nn \c__bnvs_path_regex {
184   (?: \. \ur{c__bnvs_name_regex} | \. [-+]? \d+ )*
185 }
```

(*End definition for* \c__bnvs_path_regex.)

\c__bnvs_A_key_Z_regex   A key is the name of an overlay set possibly followed by a dotted path. Matches the whole string.

(*End definition for* \c__bnvs_A_key_Z_regex.)

```
186 \regex_const:Nn \c__bnvs_A_key_Z_regex {
```

  1: The range name including the slide ⟨*id*⟩ and question mark if any

  2: slide ⟨*id*⟩ including the question mark

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

3: the path, if any.

```
188        ( \ur{c__bnvs_path_regex} ) \Z
189      }
```

\c__bnvs_colons_regex   For ranges defined by a colon syntax.

```
190 \regex_const:Nn \c__bnvs_colons_regex { :(:+)? }
```

(*End definition for* \c__bnvs_colons_regex.)

\c__bnvs_split_regex   Used to parse slide list overlay specifications in queries. Next are the 7 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.

```
191 \regex_const:Nn \c__bnvs_split_regex {
192   \s* ( ? :
```

We start with '++' instrussions[2].

```
193        \+\+
```

- 1: ⟨*name*⟩ of a slide range
- 2: ⟨*id*⟩ of a slide range including the exclamation mark

```
194      ( ( \ur{c__bnvs_id_regex}? ) \ur{c__bnvs_name_regex} )
```

- 3: optionally followed by a dotted path

```
195      ( \ur{c__bnvs_path_regex} )
```

- 4: ⟨*name*⟩ of a slide range
- 5: ⟨*id*⟩ of a slide range including the exclamation mark

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

- 6: optionally followed by a dotted path

```
197        ( \ur{c__bnvs_path_regex} )
```

We continue with other expressions

- 7: the ⟨*++n*⟩ attribute

```
198      (?: \.(\+)\+n
```

- 8: 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.

```
199    |  \s* \+= \s* ( \S+ )
```

- 9: the post increment

```
200    | (\+)\+
```

```
201    )?
202   ) \s*
203 }
```

(*End definition for* \c__bnvs_split_regex.)

---

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

16

6.8.3 beamer.cls interface

Work in progress.

```
204 \RequirePackage{keyval}
205 \define@key{beamerframe}{beanoves~id}[]{
206   \tl_set:Nx \l__bnvs_id_last_tl { #1 ! }
207 }
208 \AddToHook{env/beamer@frameslide/before}{
209   \__bnvs_n_gclear:
210   \__bnvs_v_gclear:
211   \bool_set_true:N \l__bnvs_in_frame_bool
212 }
213 \AddToHook{env/beamer@frameslide/after}{
214   \bool_set_false:N \l__bnvs_in_frame_bool
215 }
```

6.8.4 Defining named slide ranges

__bnvs_parse:nn   __bnvs_parse:nn {⟨key⟩} {⟨definition⟩}

Auxiliary function called within a group. ⟨key⟩ is the overlay reference key, including eventually a dotted path and a frame identifier, ⟨definition⟩ is the corresponding definition.

\l__bnvs_match_seq   Local storage for the match result.

(*End definition for* \l__bnvs_match_seq.)

__bnvs_range:nnnn   __bnvs_range:nnnn {⟨key⟩} {⟨first⟩} {⟨length⟩} {⟨last⟩}

Auxiliary function called within a group. Setup the model to define a range.

```
216 \cs_new:Npn \__bnvs_range:nnnn #1 {
217   \bool_if:NTF \l__bnvs_parse_bool {
218     \__bnvs_n_gremove:n { #1 }
219     \__bnvs_gclear:n { #1 }
220     \__bnvs_do_range:nnnn { #1 }
221   } {
222     \__bnvs_if_in:nnTF A { #1 } {
223       \use_none:nnn
224     } {
225       \__bnvs_if_in:nnTF L { #1 } {
226         \use_none:nnn
227       } {
228         \__bnvs_if_in:nnTF Z { #1 } {
229           \use_none:nnn
230         } {
231           \__bnvs_do_range:nnnn { #1 }
232         }
233       }
234     }
235   }
236 }
237 \cs_generate_variant:Nn \__bnvs_range:nnnn { nVVV }
238 \cs_new:Npn \__bnvs_do_range:nnnn #1 #2 #3 #4 {
```

```
239    \tl_if_empty:nTF { #3 } {
240      \tl_if_empty:nTF { #2 } {
241        \tl_if_empty:nTF { #4 } {
242          \__bnvs_error:n { Not~a~range:~:~#1 }
243        } {
244          \__bnvs_gput:nnn Z { #1 } { #4 }
245          \__bnvs_gput:nnn V { #1 } { \q_nil }
246        }
247      } {
248        \__bnvs_gput:nnn A { #1 } { #2 }
249        \__bnvs_gput:nnn V { #1 } { \q_nil }
250        \tl_if_empty:nF { #4 } {
251          \__bnvs_gput:nnn Z { #1 } { #4 }
252          \__bnvs_gput:nnn L { #1 } { \q_nil }
253        }
254      }
255    } {
256      \tl_if_empty:nTF { #2 } {
257        \__bnvs_gput:nnn L { #1 } { #3 }
258        \tl_if_empty:nF { #4 } {
259          \__bnvs_gput:nnn Z { #1 } { #4 }
260          \__bnvs_gput:nnn A { #1 } { \q_nil }
261          \__bnvs_gput:nnn V { #1 } { \q_nil }
262        }
263      } {
264        \__bnvs_gput:nnn A { #1 } { #2 }
265        \__bnvs_gput:nnn L { #1 } { #3 }
266        \__bnvs_gput:nnn Z { #1 } { \q_nil }
267        \__bnvs_gput:nnn V { #1 } { \q_nil }
268      }
269    }
270  }
```

__bnvs_parse:n __bnvs_parse:n {⟨key⟩}

A key with no value has been parsed by \keyval_parse.

```
271  \cs_new:Npn \__bnvs_parse:n #1 {
272    \peek_catcode_ignore_spaces:NTF \c_group_begin_token {
273      \tl_if_empty:NTF \l__bnvs_root_tl {
274        \__bnvs_error:n { Unexpected~list~at~top~level. }
275      }
276      \__bnvs_group_begin:
277      \int_incr:N \l__bnvs_int
278      \tl_set:Nx \l__bnvs_root_tl { \int_use:N \l__bnvs_int . }
279      \cs_set:Npn \bnvs:nw ####1 ####2 \s_stop  {
280        \regex_match:nnT { \S* } { ####2 } {
281          \__bnvs_error:n { Unexpected~####2 }
282        }
283        \keyval_parse:nnn {
284          \__bnvs_parse:n
285        } {
286          \__bnvs_parse:nn
287        } { ####1 }
```

```
288          \__bnvs_group_end:
289        }
290      \bnvs:nw
291    } {
292      \tl_if_empty:NTF \l__bnvs_root_tl {
293        \__bnvs_id_name_set:nNNTF { #1 } \l__bnvs_id_tl \l__bnvs_name_tl {
294          \__bnvs_parse_record:V \l__bnvs_name_tl
295        } {
296          \__bnvs_error:n { Unexpected~key:~#1 }
297        }
298      } {
299        \int_incr:N \l__bnvs_int
300        \__bnvs_parse_record:xn {
301          \l__bnvs_root_tl . \int_use:N \l__bnvs_int
302        } { #1 }
303      }
304      \use_none_delimit_by_s_stop:w
305    }
306    #1 \s_stop
307 }
```

__bnvs_parse_range:nNNN*TF*

__bnvs_parse_range:nNNN {⟨*input*⟩} ⟨*first tl*⟩ ⟨*length tl*⟩ ⟨*last tl*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Parse ⟨*input*⟩ as a range according to \c__bnvs_colons_regex.

```
308 \exp_args_generate:n { VVV }
309 \prg_new_conditional:Npnn \__bnvs_range_set:NNNn #1 #2 #3 #4 { T, F, TF } {
310    \__bnvs_group_begin:
```

This is not a list.

```
311    \tl_clear:N \l__bnvs_a_tl
312    \tl_clear:N \l__bnvs_b_tl
313    \tl_clear:N \l__bnvs_c_tl
314    \regex_split:NnNTF \c_bnvs_colons_regex { #4 } \l__bnvs_split_seq {
315      \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_a_tl {
```

\l__bnvs_a_tl may contain the ⟨*start*⟩.

```
316        \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_b_tl {
317          \tl_if_empty:NTF \l__bnvs_b_tl {
```

This is a one colon range.

```
318            \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
```

\l__bnvs_b_tl may contain the ⟨*length*⟩.

```
319            \seq_pop_left:NNT \l__bnvs_split_seq \l__bnvs_c_tl {
320              \tl_if_empty:NTF \l__bnvs_c_tl {
```

A :: was expected:

```
321                \__bnvs_error:n { Invalid~range~expression(1):~#4 }
322              } {
323                \int_compare:nNnT { \tl_count:N \l__bnvs_c_tl } > { 1 } {
324                  \__bnvs_error:n { Invalid~range~expression(2):~#4 }
325                }
326                \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
```

`\l__bnvs_c_tl` may contain the ⟨*end*⟩.

```
327              \seq_if_empty:NF \l__bnvs_split_seq {
328                \__bnvs_error:n { Invalid~range~expression(3):~#4 }
329              }
330            }
331          }
332        } {
```

This is a two colon range.

```
333          \int_compare:nNnT { \tl_count:N \l__bnvs_b_tl } > { 1 } {
334            \__bnvs_error:n { Invalid~range~expression(4):~#4 }
335          }
336          \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_c_tl
```

`\l__bnvs_c_tl` contains the ⟨*end*⟩.

```
337          \seq_pop_left:NNTF \l__bnvs_split_seq \l__bnvs_b_tl {
338            \tl_if_empty:NTF \l__bnvs_b_tl {
339              \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_b_tl
```

`\l_b_tl` may contain the ⟨*length*⟩.

```
340              \seq_if_empty:NF \l__bnvs_split_seq {
341                \__bnvs_error:n { Invalid~range~expression(5):~#4 }
342              }
343            } {
344              \__bnvs_error:n { Invalid~range~expression(6):~#4 }
345            }
346          } {
347            \tl_clear:N \l__bnvs_b_tl
348          }
349        }
350      }
351    }
```

Providing both the ⟨*start*⟩, ⟨*length*⟩ and ⟨*end*⟩ of a range is not allowed, even if they happen to be consistent.

```
352    \bool_if:nF {
353      \tl_if_empty_p:N \l__bnvs_a_tl
354      || \tl_if_empty_p:N \l__bnvs_b_tl
355      || \tl_if_empty_p:N \l__bnvs_c_tl
356    } {
357      \__bnvs_error:n { Invalid~range~expression(7):~#3 }
358    }
359    \cs_set:Npn \:nnn ##1 ##2 ##3 {
360      \__bnvs_group_end:
361      \tl_set:Nn #1 { ##1 }
362      \tl_set:Nn #2 { ##2 }
363      \tl_set:Nn #3 { ##3 }
364    }
365    \exp_args:NVVV \:nnn \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
366    \prg_return_true:
367  } {
368    \__bnvs_group_end:
369    \prg_return_false:
370  }
371 }
```

`\__bnvs_parse_record:n`
`\__bnvs_parse_record:nn`

`\__bnvs_parse_record:n` {⟨*full name*⟩}
`\__bnvs_parse_record:nn` {⟨*full name*⟩} {⟨*value*⟩}

Auxiliary function for `\__bnvs_parse:n` and `\__bnvs_parse:nn`.

```
372 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
373 \cs_new:Npn \__bnvs_parse_record:n #1 {
```

This is not a list.

```
374   \bool_if:NTF \l__bnvs_parse_bool {
375     \__bnvs_gclear:n { #1 }
376     \__bnvs_gput:nnn V  { #1 } { 1 }
377     \__bnvs_gput:nnn {} { #1 } { 1 }
378   } {
379     \__bnvs_gprovide:nnn V  { #1 } { 1 }
380     \__bnvs_gprovide:nnn {} { #1 } { 1 }
381   }
382 }
383 \cs_generate_variant:Nn \__bnvs_parse_record:n { V }
384 \cs_new:Npn \__bnvs_parse_record:nn #1 #2 {
```

This is not a list.

```
385   \__bnvs_range_set:NNNnTF \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl { #2 } {
386     \__bnvs_range:nVVV { #1 } \l__bnvs_a_tl \l__bnvs_b_tl \l__bnvs_c_tl
387   } {
388     \bool_if:NTF \l__bnvs_parse_bool {
389       \__bnvs_gclear:n { #1 }
390       \__bnvs_gput:nnn V  { #1 } { #2 }
391       \__bnvs_gput:nnn {} { #1 } { #2 }
392     } {
393       \__bnvs_gprovide:nnn V  { #1 } { #2 }
394       \__bnvs_gprovide:nnn {} { #1 } { #2 }
395     }
396   }
397 }
398 \cs_generate_variant:Nn \__bnvs_parse_record:nn { xn, Vn }
```

`\__bnvs_id_name_set:nNN`*TF*

`\__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*⟩.

```
399 \prg_new_conditional:Npnn \__bnvs_id_name_set:nNN #1 #2 #3 { T, F, TF } {
400   \__bnvs_group_begin:
401   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex {
402     #1
403   } \l__bnvs_match_seq {
404     \tl_set:Nx #2 { \seq_item:Nn \l__bnvs_match_seq 3 }
405     \tl_if_empty:NTF #2 {
406       \exp_args:NNNx
407       \__bnvs_group_end:
408       \tl_set:Nn #3 { \l__bnvs_id_last_tl #1 }
409       \tl_set_eq:NN #2 \l__bnvs_id_last_tl
```

```
410    } {
411      \cs_set:Npn \:n ##1 {
412        \__bnvs_group_end:
413        \tl_set:Nn #2 { ##1 }
414        \tl_set:Nn \l__bnvs_id_last_tl { ##1 }
415      }
416      \exp_args:NV
417      \:n #2
418      \tl_set:Nn #3 { #1 }
419    }
420    \prg_return_true:
421  } {
422    \__bnvs_group_end:
423    \prg_return_false:
424  }
425 }

426 \cs_new:Npn \__bnvs_parse:nn #1 #2 {
427   \__bnvs_group_begin:
428   \tl_set:Nn \l__bnvs_a_tl { #1 }
429   \tl_put_left:NV \l__bnvs_a_tl \l__bnvs_root_tl
430   \exp_args:NV
431   \__bnvs_id_name_set:nNNTF \l__bnvs_a_tl \l__bnvs_id_tl \l__bnvs_name_tl {
432     \regex_match:nnTF { \S } { #2 } {
433       \peek_catcode_ignore_spaces:NTF \c_group_begin_token {
```

This is a comma separated list, go recursive.

```
434         \__bnvs_group_begin:
435         \tl_set:NV \l__bnvs_root_tl \l__bnvs_name_tl
436         \int_set:Nn \l__bnvs_int { 0 }
437         \cs_set:Npn \bnvs:nn ##1 ##2 \s_stop {
438           \regex_match:nnT { \S } { ##2 } {
439             \__bnvs_error:n { Unexpected~value~#2 }
440           }
441           \keyval_parse:nnn {
442             \__bnvs_parse:n
443           } {
444             \__bnvs_parse:nn
445           } { ##1 }
446           \__bnvs_group_end:
447         }
448         \bnvs:nn
449       } {
450         \__bnvs_parse_record:Vn \l__bnvs_name_tl { #2 }
451         \use_none_delimit_by_s_stop:w
452       } #2 \s_stop
453     } {
```

Empty value given: remove the reference.

```
454         \exp_args:NV
455         \__bnvs_gclear:n \l__bnvs_name_tl
456         \exp_args:NV
457         \__bnvs_n_gremove:n \l__bnvs_name_tl
458       }
```

```
459    } {
460      \__bnvs_error:n { Invalid~key:~#2 }
461    }
```

We export \l__bnvs_id_tl:

```
462    \exp_args:NNNV
463    \__bnvs_group_end:
464    \tl_set:Nn \l__bnvs_id_last_tl \l__bnvs_id_last_tl
465  }

466  \cs_new:Npn \__bnvs_parse_prepare:N #1 {
467    \tl_set:Nx #1 #1
468    \bool_set_false:N \l__bnvs_parse_bool
469    \bool_do_until:Nn \l__bnvs_parse_bool {
470      \tl_if_in:NnTF #1 {%---[
471      ]} {
472        \regex_replace_all:nnNF { \[ ([^\]]*) \] } { { { \1 } } } #1 {
473          \bool_set_true:N \l__bnvs_parse_bool
474        }
475      } {
476        \bool_set_true:N \l__bnvs_parse_bool
477      }
478    }
479    \tl_if_in:NnTF #1 {%---[
480    ]} {
481      \__bnvs_error:n { Unbalanced~%---[
482      ]}
483    } {
484      \tl_if_in:NnT #1 { [%---]
485      } {
486        \__bnvs_error:n { Unbalanced~[ %---]
487      }
488    }
489  }
490  }
```

\Beanoves   \Beanoves {⟨*key--value list*⟩}

The keys are the slide overlay references. When no value is provided, it defaults to 1.
On the contrary, ⟨*key–value*⟩ items are parsed by __bnvs_parse:nn.

```
491  \NewDocumentCommand \Beanoves { sm } {
492    \tl_if_empty:NTF \@currenvir {
```

We are most certainly in the preamble, record the definitions globally for later use.

```
493      \seq_gput_right:Nn \g__bnvs_def_seq { #2 }
494    } {
495      \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, clear everything.

```
496        \__bnvs_gclear:
497      }
498      \__bnvs_group_begin:
499      \tl_clear:N \l__bnvs_root_tl
500      \int_zero:N \l__bnvs_int
```

23

```
501     \tl_set:Nn \l__bnvs_a_tl { #2 }
502     \tl_if_eq:NnT \@currenvir { document } {
```

At the top level, use the global definitions.

```
503       \seq_if_empty:NF \g__bnvs_def_seq {
504         \tl_put_left:Nx \l__bnvs_a_tl {
505           \seq_use:Nn \g__bnvs_def_seq , ,
506         }
507       }
508     }
509     \__bnvs_parse_prepare:N \l__bnvs_a_tl
510     \IfBooleanTF {#1} {
511       \bool_set_false:N \l__bnvs_parse_bool
512     } {
513       \bool_set_true:N \l__bnvs_parse_bool
514     }
515     \exp_args:NnnV
516     \keyval_parse:nnn { \__bnvs_parse:n } { \__bnvs_parse:nn } \l__bnvs_a_tl
517     \exp_args:NNNV
518     \__bnvs_group_end:
519     \tl_set:Nn \l__bnvs_id_last_tl \l__bnvs_id_last_tl
520     \ignorespaces
521   }
522 }
```

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

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

6.8.5 Scanning named overlay specifications

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

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

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

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

(*End definition for* `\l__bnvs_ans_tl`.)

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

```
524 \cs_set_eq:NN \__bnvs_beamer@frame \beamer@frame
525 \cs_set:Npn \beamer@frame < #1 > {
526   \__bnvs_group_begin:
527   \tl_clear:N \l__bnvs_ans_tl
528   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
529   \exp_args:NNNV
530   \__bnvs_group_end:
531   \__bnvs_beamer@frame < \l__bnvs_ans_tl >
532 }
533 \cs_set_eq:NN \__bnvs_beamer@masterdecode \beamer@masterdecode
534 \cs_set:Npn \beamer@masterdecode #1 {
```

24

```
535   \__bnvs_group_begin:
536   \tl_clear:N \l__bnvs_ans_tl
537   \__bnvs_scan:nNN { #1 } \__bnvs_eval:nN \l__bnvs_ans_tl
538   \exp_args:NNV
539   \__bnvs_group_end:
540   \__bnvs_beamer@masterdecode \l__bnvs_ans_tl
541 }
```

__bnvs_scan:nNN   __bnvs_scan:nNN {⟨*named overlay expression*⟩} ⟨*eval*⟩ ⟨*tl variable*⟩

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

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

(*End definition for* \l__bnvs_ans_tl.)

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

(*End definition for* \l__bnvs_int.)

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

(*End definition for* \l__bnvs_query_tl.)

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

(*End definition for* \l__bnvs_token_seq.)

\l__bnvs_token_tl   Storage for just one token.

(*End definition for* \l__bnvs_token_tl.)

```
542 \cs_new:Npn \__bnvs_scan:nNN #1 #2 #3 {
543   \__bnvs_group_begin:
544   \tl_clear:N \l__bnvs_ans_tl
545   \seq_clear:N \l__bnvs_token_seq
```

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

```
546   \regex_split:nnN {} { #1 } \l__bnvs_token_seq
```

\scan_question: \scan_question:

At top level state, scan the tokens of the ⟨*named overlay expression*⟩ looking for a '?'
character.

```
547   \cs_set:Npn \scan_question: {
548     \seq_pop_left:NNT \l__bnvs_token_seq \l__bnvs_token_tl {
549       \tl_if_eq:NnTF \l__bnvs_token_tl { ? } {
550         \require_open:
551       } {
552         \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_token_tl
553         \scan_question:
```

```
554         }
555       }
556     }
```

\require_open:

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

```
557     \cs_set:Npn \require_open: {
```

Get next token.

```
558       \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
559         \tl_if_eq:NnTF \l__bnvs_token_tl { ( %)
560       } {
```

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

```
561         \int_set:Nn \l__bnvs_int { 1 }
```

Record the forthcomming content in the \l__bnvs_query_tl variable, up to the next balancing ')'.

```
562         \tl_clear:N \l__bnvs_query_tl
563         \require_close:
564       } {
```

Ignore this token and loop.

```
565         \require_open:
566       }
567     } {
```

End reached but no opening parenthesis found, raise.

```
568       \__bnvs_fatal:x {Missing~'('%---)
569         ~after~a~?:~#1}
570     }
571   }
```

\require_close:

We found a '?(', we record the forthcomming content in the \l__bnvs_query_tl variable, up to the next balancing ')'.

```
572     \cs_set:Npn \require_close: {
```

Get next token.

```
573     \seq_pop_left:NNTF \l__bnvs_token_seq \l__bnvs_token_tl {
574       \tl_if_eq:NnTF \l__bnvs_token_tl { ( %---)
575     } {
```

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

```
576       \int_incr:N \l__bnvs_int
577       \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
578       \require_close:
579     } {
```

This is not a '('.

```
580          \tl_if_eq:NnTF \l__bnvs_token_tl { %(---
581             )
582          } {
```

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

```
583             \int_decr:N \l__bnvs_int
584             \int_compare:nNnTF \l__bnvs_int = 0 {
```

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

```
585                \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
586                \scan_question:
587             } {
```

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

```
588                \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
589                \require_close:
590             }
591          } {
```

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

```
592             \tl_put_right:NV \l__bnvs_query_tl \l__bnvs_token_tl
593             \require_close:
594          }
595       }
596    } {
```

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

```
597       \__bnvs_error:x {Missing~%(---
598          `)':~#1 }
599       \tl_put_right:Nx \l__bnvs_query_tl {
600          \prg_replicate:nn { \l__bnvs_int } {%(---
601          )}
602       }
603       \exp_args:NV #2 \l__bnvs_query_tl \l__bnvs_ans_tl
604    }
605 }
```

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

```
606    \scan_question:
607    \exp_args:NNNV
608    \__bnvs_group_end:
609    \tl_put_right:Nn #3 \l__bnvs_ans_tl
610 }
```

I

6.8.6 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_inp:NNN`*TF*

__bnvs_inp: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 recorded key, on return, ⟨*name tl var*⟩ contains the resolved name, ⟨*path seq var*⟩ is prepended with new dotted path components, {⟨*true code*⟩} is executed, otherwise {⟨*false code*⟩} is executed.

```
611 \exp_args_generate:n { VVx }
612 \prg_new_conditional:Npnn \__bnvs_inp:NNN
613     #1 #2 #3 { T, F, TF } {
614   \__bnvs_group_begin:
615   \exp_args:NNV
616   \regex_extract_once:NnNTF \c__bnvs_A_key_Z_regex #2 \l__bnvs_match_seq {
```

This is a correct key, update the path sequence accordingly

```
617     \exp_args:Nx
618     \tl_if_empty:nT { \seq_item:Nn \l__bnvs_match_seq 3 } {
619       \tl_put_left:NV #2 { #1 }
620     }
621     \exp_args:NNnx
622     \seq_set_split:Nnn \l__bnvs_split_seq . {
623       \seq_item:Nn \l__bnvs_match_seq 4
624     }
625     \seq_remove_all:Nn \l__bnvs_split_seq { }
626     \seq_pop_left:NN \l__bnvs_split_seq \l__bnvs_a_tl
627     \seq_if_empty:NTF \l__bnvs_split_seq {
```

No new integer path component is added.

```
628       \cs_set:Npn \:nn ##1 ##2 {
629         \__bnvs_group_end:
630         \tl_set:Nn #1 { ##1 }
631         \tl_set:Nn #2 { ##2 }
632       }
633       \exp_args:NVV \:nn #1 #2
634     } {
```

Some new dotted path components are added.

```
635       \cs_set:Npn \:nnn ##1 ##2 ##3 {
636         \__bnvs_group_end:
637         \tl_set:Nn #1 { ##1 }
638         \tl_set:Nn #2 { ##2 }
639         \seq_set_split:Nnn #3 . { ##3 }
640         \seq_remove_all:Nn #3 { }
641       }
642       \exp_args:NVVx
643       \:nnn #1 #2 {
644         \seq_use:Nn \l__bnvs_split_seq . . \seq_use:Nn #3 .
645       }
646     }
```

```
647        \prg_return_true:
648      } {
649        \__bnvs_group_end:
650        \prg_return_false:
651      }
652  }
```

__bnvs_resolve_n:NNN*TF*
__bnvs_resolve_n:TFF*TF*
__bnvs_resolve_x:NNN*TF*
__bnvs_resolve_x:TFF*TF*

__bnvs_resolve_x:TF {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_resolve_x:NNNTF ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩}
{⟨*false code*⟩}
__bnvs_resolve_n:TF {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_resolve_n:NNNTF ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩}
{⟨*false code*⟩}

When too many nested calls occurred, {⟨*false code*⟩} is executed directly otherwise {⟨*true code*⟩} will be executed once resolution has occurred. The ⟨*id tl var*⟩, ⟨*name tl var*⟩ and ⟨*path seq var*⟩ are meant to contain proper information on input and on output as well. On input, {⟨*id tl var*⟩} contains a frame id, {⟨*name tl var*⟩} contains a slide range name and {⟨*path seq var*⟩} contains the components of an integer path, possibly empty. On return, ⟨*id tl var*⟩ contains the frame id used, ⟨*name tl var*⟩ contains the resolved range name and ⟨*path seq var*⟩ contains the sequence of integer path components that could not be resolved.

To resolve a level of a named one slide specification ⟨*qualified name*⟩.⟨i_1⟩.⟨i_2⟩...⟨i_n⟩, we replace the shortest ⟨*qualified name*⟩.⟨i_1⟩.⟨i_2⟩...⟨i_k⟩ where 0≤k≤n by its definition ⟨*qualified name'*⟩.⟨j_1⟩...⟨j_p⟩ if any. The __bnvs_resolve:NNNTF function uses this one level resolution as many times as possible, but no more than a predefined limit to catch circular reference that would lead to an infinite loop.

1. If ⟨*name tl var*⟩ content is the name of an unlimited range, and the first item of this range is exactly another name range with eventually a heading frame identifier or a trailing integer path, then ⟨*name tl var*⟩ is replaced by this name, the ⟨*id tl var*⟩ and \l__bnvs_id_tl are updates accordingly and the ⟨*path seq var*⟩ is prepended with the integer path.

2. If ⟨*path seq var*⟩ is not empty, append to the right of ⟨*name tl var*⟩ after a separating dot, all its left elements but the last one and loop. Otherwise return.

   In the _n variant, the resolution is driven only when there is a non empty dotted path.

   In the _x variant, the resolution is driven one step further: if ⟨*path seq var*⟩ is empty, ⟨*name tl var*⟩ can contain anything.

```
653  \cs_new:Npn \__bnvs_resolve_x:TFF #1 #2 {
654    \__bnvs_resolve_x:NNNTF
655      \l__bnvs_id_tl
656      \l__bnvs_name_tl
657      \l__bnvs_path_seq {
658        \seq_if_empty:NTF \l__bnvs_path_seq { #1 } { #2 }
659    }
660  }
661  \prg_new_conditional:Npnn \__bnvs_resolve_x:NNN
662      #1 #2 #3 { T, F, TF } {
```

```
663     \__bnvs_group_begin:
```

Local variables:

- `\l__bnvs_a_tl` contains the name with a partial index path currently resolved.

- `\l__bnvs_a_seq` contains the index path components currently resolved.

- `\l__bnvs_b_tl` contains the resolution.

- `\l__bnvs_b_seq` contains the index path components to be resolved.

```
664     \seq_set_eq:NN \l__bnvs_a_seq #3
665     \seq_clear:N \l__bnvs_b_seq
666     \cs_set:Npn \loop: {
667       \__bnvs_call:TF {
668         \tl_set_eq:NN \l__bnvs_a_tl #2
669         \seq_if_empty:NTF \l__bnvs_a_seq {
670           \__bnvs_get:nVNTF L \l__bnvs_a_tl \l__bnvs_b_tl {
671             \cs_set:Nn \loop: { \return_true: }
672           } {
673             \resolve:F {
```

Unknown key ⟨`\l_a_tl`⟩`/A` or the value for key ⟨`\l_a_tl`⟩`/A` does not fit.

```
674               \cs_set:Nn \loop: { \return_true: }
675             }
676           }
677         } {
678           \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
679           \resolve:F {
680             \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
681               \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
682             }
683           }
684         }
685         \loop:
686       } {
687         \__bnvs_group_end:
688         \prg_return_false:
689       }
690     }
691     \cs_set:Npn \resolve:F ##1 {
692       \__bnvs_get:nVNTF A \l__bnvs_a_tl \l__bnvs_b_tl {
693         \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
694           \tl_set_eq:NN #2 \l__bnvs_b_tl
695           \seq_set_eq:NN #3 \l__bnvs_b_seq
696           \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
697           \seq_clear:N \l__bnvs_b_seq
698         } {
699           \seq_if_empty:NTF \l__bnvs_b_seq {
700             \tl_set_eq:NN #2 \l__bnvs_b_tl
701             \seq_clear:N #3
702             \seq_clear:N \l__bnvs_a_seq
703           } {
704             ##1
705           }
```

```
706        }
707      } {
708      \__bnvs_get:nVNTF V \l__bnvs_a_tl \l__bnvs_b_tl {
709        \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
710          \tl_set_eq:NN #2 \l__bnvs_b_tl
711          \seq_set_eq:NN #3 \l__bnvs_b_seq
712          \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
713          \seq_clear:N \l__bnvs_b_seq
714        } {
715          \seq_if_empty:NTF \l__bnvs_b_seq {
716            \tl_set_eq:NN #2 \l__bnvs_b_tl
717            \seq_clear:N #3
718            \seq_clear:N \l__bnvs_a_seq
719          } {
720            ##1
721          }
722        }
723      } { ##1 }
724    }
725    }
726    \cs_set:Npn \return_true: {
727      \seq_pop_left:NNTF #3 \l__bnvs_a_tl {
728        \seq_if_empty:NTF #3 {
729          \tl_clear:N \l__bnvs_b_tl
730          \__bnvs_can_index:VTF #2 {
731            \__bnvs_if_index:VVNTF #2 \l__bnvs_a_tl \l__bnvs_b_tl {
732              \tl_set:NV #2 \l__bnvs_b_tl
733            } {
734              \tl_set:NV #2 \l__bnvs_a_tl
735            }
736          } {
737            \tl_set:NV #2 \l__bnvs_a_tl
738          }
739        } {
740          \__bnvs_error:x { Path~too~long:~#2.\l__bnvs_a_tl
741            .\seq_use:Nn\l__bnvs_path_seq .}
742        }
743      } {
744        \tl_clear:N \l__bnvs_b_tl
745        \__bnvs_raw_value:VNT #2 \l__bnvs_b_tl {
746          \tl_set:NV #2 \l__bnvs_b_tl
747        }
748      }
749      \cs_set:Npn \:nnn ####1 ####2 ####3 {
750        \__bnvs_group_end:
751        \tl_set:Nn #1 { ####1 }
752        \tl_set:Nn #2 { ####2 }
753        \seq_set_split:Nnn #3 . { ####3 }
754        \seq_remove_all:Nn #3 { }
755      }
756      \exp_args:NVVx
757      \:nnn #1 #2 {
758        \seq_use:Nn #3 .
759      }
```

```
760      \prg_return_true:
761    }
762    \loop:
763 }
764 \cs_new:Npn \__bnvs_resolve_n:TFF #1 #2 {
765    \__bnvs_resolve_n:NNNTF
766    \l__bnvs_id_tl
767    \l__bnvs_name_tl
768    \l__bnvs_path_seq {
769      \seq_if_empty:NTF \l__bnvs_path_seq { #1 } { #2 }
770    }
771 }
772 \prg_new_conditional:Npnn \__bnvs_resolve_n: { T, F, TF } {
773    \__bnvs_resolve_n:NNNTF
774    \l__bnvs_name_tl
775    \l__bnvs_id_tl
776    \l__bnvs_path_seq {
777      \prg_return_true:
778    } {
779      \prg_return_false:
780    }
781 }
782 \prg_new_conditional:Npnn \__bnvs_resolve_n_old:NNN
783      #1 #2 #3 { T, F, TF } {
784    \__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.

```
785    \seq_set_eq:NN \l__bnvs_a_seq #3
786    \seq_clear:N \l__bnvs_b_seq
787    \cs_set:Npn \loop: {
788      \__bnvs_call:TF {
789        \tl_set_eq:NN \l__bnvs_a_tl #2
790        \seq_if_empty:NTF \l__bnvs_a_seq {
791          \__bnvs_get:nVNTF L \l__bnvs_a_tl \l__bnvs_b_tl {
792            \cs_set:Npn \loop: { \return_true: }
793          } {
794            \seq_if_empty:NTF \l__bnvs_b_seq {
795              \cs_set:Npn \loop: { \return_true: }
796            } {
797              \:F {
```

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

```
798                \cs_set:Npn \loop: { \return_true: }
799              }
800            }
801          }
```

```
802    } {
803      \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
804      \:F {
805        \seq_pop_right:NNT \l__bnvs_a_seq \l__bnvs_c_tl {
806          \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
807        }
808      }
809    }
810    \loop:
811  } {
812    \__bnvs_group_end:
813    \prg_return_false:
814  }
815 }
816 \cs_set:Npn \:F ##1 {
817   \__bnvs_get:nVNTF A \l__bnvs_a_tl \l__bnvs_b_tl {
818     \__bnvs_inp:NNNTF #1 \l__bnvs_b_tl \l__bnvs_b_seq {
819       \tl_set_eq:NN #2 \l__bnvs_b_tl
820       \seq_set_eq:NN #3 \l__bnvs_b_seq
821       \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
822       \seq_clear:N \l__bnvs_b_seq
823     } { ##1 }
824   } { ##1 }
825 }
826 \cs_set:Npn \return_true: {
827   \cs_set:Npn \:nnn ####1 ####2 ####3 {
828     \__bnvs_group_end:
829     \tl_set:Nn #1 { ####1 }
830     \tl_set:Nn #2 { ####2 }
831     \seq_set_split:Nnn #3 . { ####3 }
832     \seq_remove_all:Nn #3 { }
833   }
834   \exp_args:NVVx
835   \:nnn #1 #2 { \seq_use:Nn #3 . }
836   \prg_return_true:
837 }
838 \loop:
839 }
840 \prg_new_conditional:Npnn \__bnvs_resolve_n:NNN
841     #1 #2 #3 { T, F, TF } {
842   \__bnvs_group_begin:
```

Local variables:

- `\l__bnvs_a_tl` contains the name with a partial index path currently resolved.

- `\l__bnvs_id_tl`, `\l__bnvs_name_tl`, `\l__bnvs_path_seq` contains the resolution.

- `\l__bnvs_a_seq` contains the dotted path components to be resolved. Initially empty.

```
843   \tl_set_eq:NN \l__bnvs_id_tl #1
844   \tl_set_eq:NN \l__bnvs_name_tl #2
845   \seq_set_eq:NN \l__bnvs_path_seq #3
846   \seq_set_eq:NN \l__bnvs_a_seq #3
847   \seq_clear:N \l__bnvs_b_seq
```

```
848  \cs_set:Npn \loop: {
849    \__bnvs_call:TF {
850      \tl_set_eq:NN \l__bnvs_a_tl \l__bnvs_name_tl
851      \seq_if_empty:NTF \l__bnvs_a_seq {
852        \seq_if_empty:NTF \l__bnvs_b_seq {
853          \group_end_return_true:
854        } {
855          \resolve:nF A {
856            \resolve:nF V {
857              \may_loop:
858            }
859          }
860        }
861      } {
862        \tl_put_right:Nx \l__bnvs_a_tl { . \seq_use:Nn \l__bnvs_a_seq . }
863        \resolve:nF A {
864          \resolve:nF V {
865            \may_loop:
866          }
867        }
868      }
869    } {
870      \__bnvs_group_end:
871      \prg_return_false:
872    }
873  }
874  \cs_set:Npn \may_loop: {
875    \seq_pop_right:NNTF \l__bnvs_a_seq \l__bnvs_c_tl {
876      \seq_put_left:NV \l__bnvs_b_seq \l__bnvs_c_tl
877      \loop:
878    } {
879      \group_end_return_true:
880    }
881  }
882  \cs_set:Npn \resolve:nF ##1 ##2 {
883    \__bnvs_get:nVNTF ##1 \l__bnvs_a_tl \l__bnvs_b_tl {
884      \__bnvs_inp:NNNTF \l__bnvs_id_tl \l__bnvs_b_tl \l__bnvs_b_seq {
885        \tl_set_eq:NN \l__bnvs_name_tl \l__bnvs_b_tl
886        \seq_set_eq:NN \l__bnvs_path_seq \l__bnvs_b_seq
887        \seq_set_eq:NN \l__bnvs_a_seq \l__bnvs_b_seq
888        \seq_clear:N \l__bnvs_b_seq
889        \loop:
890      } {
891        \may_loop:
892      }
893    } {
894      ##2
895    }
896  }
897  \cs_set:Npn \group_end_return_true: {
898    \cs_set:Npn \:nnn ####1 ####2 ####3 {
899      \__bnvs_group_end:
900      \tl_set:Nn #1 { ####1 }
901      \tl_set:Nn #2 { ####2 }
```

34

```
902        \seq_set_split:Nnn #3 . { ####3 }
903        \seq_remove_all:Nn #3 { }
904      }
905      \exp_args:NVVx
906      \:nnn \l__bnvs_id_tl \l__bnvs_name_tl { \seq_use:Nn \l__bnvs_path_seq . }
907      \prg_return_true:
908    }
909    \loop:
910  }
```

__bnvs_resolve_n:NNN*TF*
__bnvs_resolve_x:NNN*TF*

__bnvs_resolve_n:NNNTF ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩}
{⟨*false code*⟩}
__bnvs_resolve_x:NNNTF ⟨*id tl var*⟩ ⟨*name tl var*⟩ ⟨*path seq var*⟩ {⟨*true code*⟩}
{⟨*false code*⟩}

When too many nested calls occurred, {⟨*false code*⟩} is executed directly. ⟨*id tl var*⟩,
⟨*name tl var*⟩ and ⟨*path seq var*⟩ are meant to contain proper information. On input,
{⟨*id tl var*⟩} contains a frame id, {⟨*name tl var*⟩} contains a slide range name and {⟨*path
seq var*⟩} contains the components of an integer path, possibly empty. On return, ⟨*id tl
var*⟩ contains the frame id used, ⟨*name tl var*⟩ contains the resolved range name and ⟨*path
seq var*⟩ contains the sequence of integer path components that could not be resolved.

To resolve a level of a named one slide specification ⟨*qualified name*⟩.⟨c_1⟩.⟨c_2⟩...⟨c_n⟩,
we replace the shortest ⟨*qualified name*⟩.⟨c_1⟩.⟨c_2⟩...⟨c_k⟩ where 0≤k≤n by its definition
⟨*qualified name'*⟩.⟨c'_1⟩...⟨c'_p⟩ if any. The __bnvs_resolve:NNNTF function uses this one
level resolution as many times as possible, but no more than a predefined limit to catch
circular reference that would lead to an infinite loop.

1. If ⟨*name tl var*⟩ content is the name of an unlimited range, and the first item of this
   range is exactly another name range with eventually a heading frame identifier or
   a trailing integer path, then ⟨*name tl var*⟩ is replaced by this name, the ⟨*id tl var*⟩
   and \l__bnvs_id_tl are updates accordingly and the ⟨*path seq var*⟩ is prepended
   with the integer path.

2. If ⟨*path seq var*⟩ is not empty, append to the right of ⟨*name tl var*⟩ after a separating
   dot, all its left elements but the last one and loop. Otherwise return.

   NOTA BENE: Implementation details. None of the tl variables must be one of
   \l__bnvs_a_tl, \l__bnvs_b_tl or \l__bnvs_c_tl. None of the seq variables
   must be one of \l__bnvs_a_seq, \l__bnvs_b_seq.

   In the _x variant, the resolution is driven one step further: if ⟨*path seq var*⟩ is
   empty, ⟨*name tl var*⟩ can contain anything, including an integer for example.

6.8.7 Evaluation bricks

We start by helpers.

__bnvs_round:nN
__bnvs_round:N

__bnvs_round:nN {⟨*expression*⟩} ⟨*tl variable*⟩
__bnvs_round:N ⟨*tl variable*⟩

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

```
911 \cs_new:Npn \__bnvs_round:nN #1 #2 {
912   \tl_if_empty:nTF { #1 } {
913     \tl_put_right:Nn #2 { 0 }
914   } {
915     \tl_put_right:Nx #2 { \fp_eval:n { round(#1) } }
916   }
917 }
918 \cs_new:Npn \__bnvs_round:N #1 {
919   \tl_if_empty:VTF #1 {
920     \tl_set:Nn #1 { 0 }
921   } {
922     \tl_set:Nx #1 { \fp_eval:n { round(#1) } }
923   }
924 }
```

__bnvs_group_end_return_true:nnN	__bnvs_group_end_return_true:nnN {⟨subkey⟩} {⟨key⟩} ⟨tl variable⟩
__bnvs_group_end_return_false:nn	__bnvs_group_end_return_false:nn {⟨subkey⟩} {⟨key⟩}

End a group and calls \prg_return_true: or \prg_return_false:. Before returning, the first one appends the content of \l__bvs_ans_tl to the ⟨tl variable⟩ and cache this content under ⟨subkey⟩ whereas the second one cleans the canche for that ⟨subkey⟩.

```
925 \cs_set:Npn \__bnvs_group_end_return_true:nnN #1 #2 #3 {
926   \tl_if_empty:NTF \l__bnvs_ans_tl {
927     \__bnvs_group_end:
928     \__bnvs_gremove_cache:nn { #1 } { #2 }
929     \prg_return_false:
930   } {
931     \__bnvs_round:N \l__bnvs_ans_tl
932     \__bnvs_gput_cache:nnV { #1 } { #2 } \l__bnvs_ans_tl
933     \exp_args:NNNV
934     \__bnvs_group_end:
935     \tl_put_right:Nn #3 \l__bnvs_ans_tl
936     \prg_return_true:
937   }
938 }
939 \cs_set:Npn \__bnvs_group_end_return_false:nn #1 #2 {
940   \__bnvs_group_end:
941   \__bnvs_gremove_cache:nn { #1 } { #2 }
942   \prg_return_false:
943 }
```

__bnvs_raw_first:nN*TF*	__bnvs_raw_first:nNTF {⟨name⟩} ⟨tl variable⟩ {⟨true code⟩} {⟨false code⟩}
__bnvs_raw_first:(xN|VN)*TF*	

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

```
944 \prg_new_conditional:Npnn \__bnvs_raw_first:nN #1 #2 { T, F, TF } {
945   \__bnvs_group_begin:
946   \__bnvs_get_cache:nnNTF A { #1 } #2 {
947     \exp_args:NNNV
948     \__bnvs_group_end:
949     \tl_put_right:Nn #2 #2
```

```
950     \prg_return_true:
951   } {
952     \__bnvs_get:nnNTF A { #1 } \l__bnvs_a_tl {
953       \tl_clear:N \l__bnvs_ans_tl
954       \quark_if_nil:NTF \l__bnvs_a_tl {
955         \__bnvs_gput:nnn A { #1 } { \q_no_value }
```

The first index must be computed separately from the length and the last index.

```
956         \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
957           \tl_put_right:Nn \l__bnvs_ans_tl { - }
958           \tl_clear:N \l__bnvs_a_tl
959           \__bnvs_raw_length:nNTF { #1 } \l__bnvs_a_tl {
960             \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
961             \tl_put_right:Nn \l__bnvs_ans_tl { + 1 }
962             \__bnvs_group_end_return_true:nnN A { #1 } #2
963           } {
964             \__bnvs_error:n { Unavailable~length~for~#1~(\__bnvs_raw_first:nNTF/2) }
965             \__bnvs_group_end_return_false:nn A { #1 }
966           }
967         } {
968           \__bnvs_error:n { Unavailable~last~for~#1~(\__bnvs_raw_first:nNTF/1) }
969           \__bnvs_group_end_return_false:nn A { #1 }
970         }
971       } {
972         \quark_if_no_value:NTF \l__bnvs_a_tl {
973           \__bnvs_fatal:n {Circular~definition:~#1}
974         } {
975           \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
976             \__bnvs_group_end_return_true:nnN A { #1 } #2
977           } {
978             \__bnvs_group_end_return_false:nn A { #1 }
979           }
980         }
981       }
982     } {
983       \__bnvs_group_end_return_false:nn A { #1 }
984     }
985   }
986 }
987 \prg_generate_conditional_variant:Nnn
988   \__bnvs_raw_first:nN { VN, xN } { T, F, TF }
```

__bnvs_raw_length:nN*TF* __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.

```
989 \prg_new_conditional:Npnn \__bnvs_raw_length:nN #1 #2 { T, F, TF } {
990   \__bnvs_group_begin:
991   \__bnvs_get_cache:nnNTF L { #1 } #2 {
992     \exp_args:NNNV
993     \__bnvs_group_end:
994     \tl_put_right:Nn #2 #2
```

```
995        \prg_return_true:
996      } {
997        \__bnvs_get:nnNTF L { #1 } \l__bnvs_a_tl {
998          \tl_clear:N \l__bnvs_ans_tl
999          \quark_if_nil:NTF \l__bnvs_a_tl {
1000           \__bnvs_gput:nnn L { #1 } { \q_no_value }
```

The length must be computed separately from the start and the last index.

```
1001           \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
1002             \tl_put_right:Nn \l__bnvs_ans_tl { - }
1003             \tl_clear:N \l__bnvs_a_tl
1004             \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
1005               \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1006               \tl_put_right:Nn \l__bnvs_ans_tl { + 1 }
1007               \__bnvs_group_end_return_true:nnN L { #1 } #2
1008             } {
1009               \__bnvs_error:n { Unavailable~first~for~#1~(\__bnvs_raw_length:nNTF/2) }
1010               \__bnvs_group_end_return_false:nn L { #1 }
1011             }
1012           } {
1013             \__bnvs_error:n { Unavailable~last~for~#1~(\__bnvs_raw_length:nNTF/1) }
1014             \__bnvs_group_end_return_false:nn L { #1 }
1015           }
1016         } {
1017           \quark_if_no_value:NTF \l__bnvs_a_tl {
1018             \__bnvs_fatal:n {Circular~definition:~#1}
1019           } {
1020             \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1021               \__bnvs_group_end_return_true:nnN L { #1 } #2
1022             } {
1023               \__bnvs_group_end_return_false:nn L { #1 }
1024             }
1025           }
1026         }
1027       } {
1028         \__bnvs_group_end_return_false:nn L { #1 }
1029       }
1030   }
1031 }
1032 \prg_generate_conditional_variant:Nnn
1033   \__bnvs_raw_length:nN { VN } { T, F, TF }
```

__bnvs_raw_last:nN*TF*   __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.

```
1034 \prg_new_conditional:Npnn \__bnvs_raw_last:nN #1 #2 { T, F, TF } {
1035   \__bnvs_group_begin:
1036   \__bnvs_get_cache:nnNTF Z { #1 } #2 {
1037     \exp_args:NNNV
1038     \__bnvs_group_end:
1039     \tl_put_right:Nn #2 #2
```

```
1040      \prg_return_true:
1041    } {
1042      \__bnvs_get:nnNTF Z { #1 } \l__bnvs_a_tl {
1043        \tl_clear:N \l__bnvs_ans_tl
1044        \quark_if_nil:NTF \l__bnvs_a_tl {
1045          \__bnvs_gput:nnn Z { #1 } { \q_no_value }
```

The last index must be computed separately from the start and the length.

```
1046          \tl_clear:N \l__bnvs_a_tl
1047          \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1048            \tl_put_right:Nn \l__bnvs_ans_tl { + }
1049            \tl_clear:N \l__bnvs_b_tl
1050            \__bnvs_raw_length:nNTF { #1 } \l__bnvs_b_tl {
1051              \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_b_tl
1052              \tl_put_right:Nn \l__bnvs_ans_tl { - 1 }
1053              \__bnvs_group_end_return_true:nnN Z { #1 } #2
1054            } {
1055              \__bnvs_error:n { Unavailable~length~for~#1~(\__bnvs_raw_last:nNTF/1) }
1056              \__bnvs_group_end_return_false:nn Z { #1 }
1057            }
1058          } {
1059            \__bnvs_error:n { Unavailable~start~for~#1~(\__bnvs_raw_last:nNTF/1) }
1060            \__bnvs_group_end_return_false:nn Z { #1 }
1061          }
1062        } {
1063          \quark_if_no_value:NTF \l__bnvs_a_tl {
1064            \__bnvs_fatal:n {Circular~definition:~#1}
1065          } {
1066            \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1067              \__bnvs_group_end_return_true:nnN Z { #1 } #2
1068            } {
1069              \__bnvs_group_end_return_false:nn Z { #1 }
1070            }
1071          }
1072        }
1073      } {
1074        \__bnvs_group_end_return_false:nn Z { #1 }
1075      }
1076    }
1077 }
1078 \prg_generate_conditional_variant:Nnn
1079    \__bnvs_raw_last:nN { VN } { T, F, TF }
```

$\underline{\texttt{\\__bnvs_if_range:nN}\textit{TF}}$ __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.

```
1080 \prg_new_conditional:Npnn \__bnvs_if_range:nN #1 #2 { T, F, TF } {
1081    \__bnvs_group_begin:
1082    \tl_clear:N \l__bnvs_a_tl
1083    \tl_clear:N \l__bnvs_b_tl
1084    \tl_clear:N \l__bnvs_ans_tl
1085    \__bnvs_raw_first:nNTF { #1 } \l__bnvs_a_tl {
```

```
1086        \int_compare:nNnT { \l__bnvs_a_tl } < 0 {
1087          \tl_set:Nn \l__bnvs_a_tl { 0 }
1088        }
1089        \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
```

Limited from above and below.

```
1090          \int_compare:nNnT { \l__bnvs_b_tl } < 0 {
1091            \tl_set:Nn \l__bnvs_b_tl { 0 }
1092          }
1093          \exp_args:NNNx
1094          \__bnvs_group_end:
1095          \tl_put_right:Nn #2 { \l__bnvs_a_tl - \l__bnvs_b_tl }
1096          \prg_return_true:
1097        } {
```

Limited from below.

```
1098          \exp_args:NNNV
1099          \__bnvs_group_end:
1100          \tl_put_right:Nn #2 \l__bnvs_a_tl
1101          \tl_put_right:Nn #2 { - }
1102          \prg_return_true:
1103        }
1104      } {
1105        \__bnvs_raw_last:nNTF { #1 } \l__bnvs_b_tl {
```

Limited from above.

```
1106          \int_compare:nNnT { \l__bnvs_b_tl } < 0 {
1107            \tl_set:Nn \l__bnvs_b_tl { 0 }
1108          }
1109          \exp_args:NNNx
1110          \__bnvs_group_end:
1111          \tl_put_right:Nn #2 { - \l__bnvs_b_tl }
1112          \prg_return_true:
1113        } {
1114          \__bnvs_raw_value:nNTF { #1 } \l__bnvs_b_tl {
```

Unlimited range.

```
1115            \exp_args:NNNx
1116            \__bnvs_group_end:
1117            \tl_put_right:Nn #2 { - }
1118            \prg_return_true:
1119          } {
1120            \__bnvs_group_end:
1121            \prg_return_false:
1122          }
1123        }
1124      }
1125  }
1126  \prg_generate_conditional_variant:Nnn
1127    \__bnvs_if_range:nN { VN } { T, F, TF }
```

__bnvs_if_previous:nN*TF*   __bnvs_if_previous: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.

```
1128 \prg_new_conditional:Npnn \__bnvs_if_previous:nN #1 #2 { T, F, TF } {
1129   \__bnvs_group_begin:
1130   \__bnvs_get_cache:nnNTF P { #1 } #2 {
1131     \exp_args:NNNV
1132     \__bnvs_group_end:
1133     \tl_put_right:Nn #2 #2
1134     \prg_return_true:
1135   } {
1136     \tl_clear:N \l__bnvs_ans_tl
1137     \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1138       \tl_put_right:Nn \l__bnvs_ans_tl { -1 }
1139       \__bnvs_group_end_return_true:nnN P { #1 } #2
1140     } {
1141       \__bnvs_group_end_return_false:nn P { #1 }
1142     }
1143   }
1144 }
1145 \prg_generate_conditional_variant:Nnn
1146   \__bnvs_if_previous:nN { VN } { T, F, TF }
```

__bnvs_if_next:nN*TF* __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.

```
1147 \prg_new_conditional:Npnn \__bnvs_if_next:nN #1 #2 { T, F, TF } {
1148   \__bnvs_group_begin:
1149   \__bnvs_get_cache:nnNTF N { #1 } #2 {
1150     \exp_args:NNNV
1151     \__bnvs_group_end:
1152     \tl_put_right:Nn #2 #2
1153     \prg_return_true:
1154   } {
1155     \tl_clear:N \l__bnvs_ans_tl
1156     \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
1157       \tl_put_right:Nn \l__bnvs_ans_tl { +1 }
1158       \__bnvs_group_end_return_true:nnN P { #1 } #2
1159     } {
1160       \__bnvs_group_end_return_false:nn P { #1 }
1161     }
1162   }
1163 }
1164 \prg_generate_conditional_variant:Nnn
1165   \__bnvs_if_next:nN { VN } { T, F, TF }
```

__bnvs_raw_value:nN*TF* __bnvs_raw_value:nNTF {⟨name⟩} ⟨tl variable⟩ {⟨true code⟩} {⟨false code⟩}

Append the value of the ⟨name⟩ overlay set to the ⟨tl variable⟩. Cache the result under subkey V. Execute ⟨true code⟩ when there is a ⟨value⟩, ⟨false code⟩ otherwise.

```
1166 \prg_new_conditional:Npnn \__bnvs_raw_value:nN #1 #2 { T, F, TF } {
```

```
1167     \__bnvs_group_begin:
1168     \__bnvs_get_cache:nnNTF V { #1 } #2 {
1169       \exp_args:NNNV
1170       \__bnvs_group_end:
1171       \tl_put_right:Nn #2 #2
1172       \prg_return_true:
1173     } {
1174       \__bnvs_get:nnNTF V { #1 } \l__bnvs_a_tl {
1175         \tl_clear:N \l__bnvs_ans_tl
1176         \quark_if_nil:NTF \l__bnvs_a_tl {
1177           \__bnvs_gput:nnn V { #1 } { \q_no_value }
1178           \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
1179             \__bnvs_group_end_return_true:nnN V { #1 } #2
1180           } {
1181             \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
1182               \__bnvs_group_end_return_true:nnN V { #1 } #2
1183             } {
1184               \__bnvs_group_end_return_false:nn V { #1 }
1185             }
1186           }
1187         } {
1188           \quark_if_no_value:NTF \l__bnvs_a_tl {
1189             \__bnvs_fatal:n {Circular~definition:~#1}
1190           } {
1191             \__bnvs_if_append:VNTF \l__bnvs_a_tl \l__bnvs_ans_tl {
1192               \__bnvs_group_end_return_true:nnN V { #1 } #2
1193             } {
1194               \__bnvs_group_end_return_false:nn V { #1 }
1195             }
1196           }
1197         }
1198       } {
1199         \__bnvs_group_end_return_false:nn V { #1 }
1200       }
1201     }
1202 }
1203 \prg_generate_conditional_variant:Nnn
1204     \__bnvs_raw_value:nN{ V } { T, F, TF }
```

__bnvs_can_index:n*TF*
__bnvs_can_index:V*TF*
__bnvs_if_index:nnN*TF*
__bnvs_if_index:VVN*TF*

__bnvs_can_index:nTF {⟨*name*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_if_index:nnNTF {⟨*name*⟩} {⟨*integer*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_can_index:nTF {⟨*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.

```
1205 \prg_new_conditional:Npnn \__bnvs_can_index:n #1 { p, T, F, TF } {
1206   \bool_if:nTF {
```

```
1207        \__bnvs_if_in_p:nn A { #1 }
1208     || \__bnvs_if_in_p:nn Z { #1 }
1209     || \__bnvs_if_in_p:nn V { #1 }
1210  } {
1211     \prg_return_true:
1212  } {
1213     \prg_return_false:
1214  }
1215 }
1216 \prg_generate_conditional_variant:Nnn
1217    \__bnvs_can_index:n { V } { p, T, F, TF }
1218 \prg_new_conditional:Npnn \__bnvs_if_index:nnN #1 #2 #3 { T, F, TF } {
1219    \__bnvs_group_begin:
1220    \cs_set:Npn \group_end_return_true:n ##1 {
1221       \tl_put_right:Nn \l__bnvs_ans_tl { + #2 - 1 }
1222       \exp_args:NNV
1223       \__bnvs_group_end:
1224    \__bnvs_round:nN \l__bnvs_ans_tl #3
1225       \prg_return_true:
1226    }
1227    \tl_clear:N \l__bnvs_ans_tl
1228    \__bnvs_raw_first:nNTF { #1 } \l__bnvs_ans_tl {
```

Limited overlay set.

```
1229       \group_end_return_true:n { A }
1230    } {
1231       \__bnvs_raw_last:nNTF { #1 } \l__bnvs_ans_tl {
```

Right limited overlay set.

```
1232          \group_end_return_true:n { Z }
1233       } {
1234          \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
```

Unlimited overlay set.

```
1235             \group_end_return_true:n { V }
1236          } {
1237             \__bnvs_group_end:
1238             \prg_return_false:
1239          }
1240       }
1241    }
1242 }
1243 \prg_generate_conditional_variant:Nnn
1244    \__bnvs_if_index:nnN { VVN } { T, F, TF }
```

__bnvs_if_n_value:nN*TF*
__bnvs_if_n_value:VN*TF*

__bnvs_if_n_value:nNTF {⟨*name*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Append the value of the n counter associated to the {⟨*name*⟩} overlay set to the right of ⟨*tl variable*⟩. Initialize this counter to 1 on the first use. ⟨*false code*⟩ is never executed.

```
1245 \prg_new_conditional:Npnn \__bnvs_if_n_value:nN #1 #2 { T, F, TF } {
1246    \__bnvs_n_get:nNF { #1 } #2 {
1247       \tl_set:Nn #2 { 1 }
1248       \__bnvs_n_gput:nn { #1 } { 1 }
1249    }
```

```
1250     \prg_return_true:
1251 }
1252 \prg_generate_conditional_variant:Nnn
1253     \__bnvs_if_n_value:nN { VN } { T, F, TF }
```

__bnvs_if_n_value:nNTF {⟨*name*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Append the value of the n counter associated to the {⟨*name*⟩} overlay set to the right of ⟨*tl variable*⟩. Initialize this counter to 1 on the first use.

```
1254 \prg_new_conditional:Npnn \__bnvs_if_n_index:nN #1 #2 { T, F, TF } {
1255     \__bnvs_group_begin:
1256     \__bnvs_if_n_value:nNF { #1 } \l__bnvs_a_tl { }
1257     \exp_args:NNnV
1258     \__bnvs_group_end:
1259     \__bnvs_if_index:nnNTF { #1 } \l__bnvs_a_tl #2 {
1260         \prg_return_true:
1261     } {
1262         \__bnvs_group_begin:
1263         \__bnvs_raw_value:nNTF {#1} \l__bnvs_ans_tl {
1264             \tl_put_right:Nn \l__bnvs_ans_tl { + #2 - 1 }
1265             \exp_args:NNV
1266             \__bnvs_group_end:
1267             \__bnvs_round:Nn \l__bnvs_ans_tl
1268             \prg_return_true:
1269         } {
1270             \__bnvs_group_end:
1271             \prg_return_false:
1272         }
1273     }
1274 }
1275 \prg_generate_conditional_variant:Nnn
1276     \__bnvs_if_n_index:nN { VN } { T, F, TF }
```

__bnvs_if_incr:nnTF   {⟨*name*⟩} {⟨*offset*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
__bnvs_if_incr:nnNTF {⟨*name*⟩} {⟨*offset*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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.

```
1277 \prg_new_conditional:Npnn \__bnvs_if_incr:nn #1 #2 { T, F, TF } {
1278     \__bnvs_group_begin:
1279     \tl_clear:N \l__bnvs_ans_tl
1280     \__bnvs_if_append:nNTF { #2 } \l__bnvs_ans_tl {
1281         \int_compare:nNnTF \l__bnvs_ans_tl = 0 {
1282             \tl_clear:N \l__bnvs_ans_tl
1283             \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
1284                 \__bnvs_group_end:
1285                 \prg_return_true:
1286             } {
1287                 \__bnvs_group_end:
```

```
1288              \prg_return_false:
1289            }
1290          } {
1291            \tl_put_right:Nn \l__bnvs_ans_tl { + }
1292            \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
1293              \__bnvs_round:N \l__bnvs_ans_tl
1294              \__bnvs_gput_cache:nnV V { #1 } \l__bnvs_ans_tl
1295              \__bnvs_group_end:
1296              \prg_return_true:
1297            } {
1298              \__bnvs_group_end:
1299              \prg_return_false:
1300            }
1301          }
1302        } {
1303          \__bnvs_group_end:
1304          \prg_return_false:
1305        }
1306    }
1307    \prg_new_conditional:Npnn \__bnvs_if_incr:nnN #1 #2 #3 { T, F, TF } {
1308      \__bnvs_if_incr:nnTF { #1 } { #2 } {
1309        \__bnvs_raw_value:nNTF { #1 } #3 {
1310          \prg_return_true:
1311        } {
1312          \prg_return_false:
1313        }
1314      } {
1315        \prg_return_false:
1316      }
1317    }
1318    \prg_generate_conditional_variant:Nnn
1319      \__bnvs_if_incr:nnN { VnN, VVN } { T, F, TF }
```

__bnvs_if_n_incr:nn*TF*	__bnvs_if_n_incr:nnTF {⟨name⟩} {⟨offset⟩} {⟨true code⟩} {⟨false code⟩}
__bnvs_if_n_incr:nnN*TF*	__bnvs_if_n_incr:nnNTF {⟨name⟩} {⟨offset⟩} ⟨tl variable⟩ {⟨true code⟩}
__bnvs_if_n_incr:(VnN|VVN)*TF*	{⟨false code⟩}

Increment the implicit index counter accordingly. When requested, put the result in the ⟨tl variable⟩.

```
1320    \prg_new_conditional:Npnn \__bnvs_if_n_incr:nn #1 #2 { T, F, TF } {
1321      \__bnvs_group_begin:
1322      \tl_clear:N \l__bnvs_ans_tl
1323      \__bnvs_n_get:nNF { #1 } \l__bnvs_ans_tl {
1324        \tl_set:Nn \l__bnvs_ans_tl { 1 }
1325      }
1326      \tl_clear:N \l__bnvs_a_tl
1327      \__bnvs_if_append:nNTF { #2 } \l__bnvs_a_tl {
1328        \tl_put_right:Nn \l__bnvs_ans_tl { + }
1329        \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_a_tl
1330        \__bnvs_round:N \l__bnvs_ans_tl
1331        \__bnvs_n_gput:nV { #1 } \l__bnvs_ans_tl
1332        \__bnvs_group_end:
```

```
1333        \prg_return_true:
1334    } {
1335        \__bnvs_group_end:
1336        \prg_return_false:
1337    }
1338 }
1339 \prg_new_conditional:Npnn \__bnvs_if_n_incr:nnN #1 #2 #3 { T, F, TF } {
1340    \__bnvs_if_n_incr:nnTF { #1 } { #2 } {
1341        \__bnvs_n_get:nNTF { #1 } #3 {
1342            \prg_return_true:
1343        } {
1344            \prg_return_false:
1345        }
1346    } {
1347        \prg_return_false:
1348    }
1349 }
1350 \prg_generate_conditional_variant:Nnn
1351    \__bnvs_if_n_incr:nnN { VnN, VVN } { T, F, TF }
```

__bnvs_if_post:nnN*TF*
__bnvs_if_post:(VnN|VVN)*TF*

__bnvs_if_post:nnNTF {⟨*name*⟩} {⟨*offset*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Put the value of the free counter for the given ⟨*name*⟩ in the ⟨*tl variable*⟩ then increment this free counter position accordingly.

```
1352 \prg_new_conditional:Npnn \__bnvs_if_post:nnN #1 #2 #3 { T, F, TF } {
1353    \__bnvs_group_begin:
1354    \tl_clear:N \l__bnvs_ans_tl
1355    \__bnvs_raw_value:nNTF { #1 } \l__bnvs_ans_tl {
1356        \__bnvs_if_incr:nnTF { #1 } { #2 } {
1357            \exp_args:NNNV
1358            \__bnvs_group_end:
1359            \tl_put_right:Nn #3 \l__bnvs_ans_tl
1360            \prg_return_true:
1361        } {
1362            \__bnvs_group_end:
1363            \prg_return_false:
1364        }
1365    } {
1366        \__bnvs_group_end:
1367        \prg_return_false:
1368    }
1369 }
1370 \prg_generate_conditional_variant:Nnn
1371    \__bnvs_if_post:nnN { VnN, VVN } { T, F, TF }
```

46

6.8.8 Evaluation

`\__bnvs_if_append:nN` *TF*	`\__bnvs_if_append:nNTF {⟨integer expression⟩} ⟨tl variable⟩ {⟨true code⟩} {⟨false`
`\__bnvs_if_append:(VN|xN)` *TF*	`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__bnvs_ans_tl` To feed ⟨*tl variable*⟩ with.

(End definition for `\l__bnvs_ans_tl`*.)*

`\l__bnvs_split_seq` The sequence of catched query groups and non queries.

(End definition for `\l__bnvs_split_seq`*.)*

`\l__bnvs_split_int` Is the index of the non queries, before all the catched groups.

(End definition for `\l__bnvs_split_int`*.)*

```
1372 \int_new:N \l__bnvs_split_int
```

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

(End definition for `\l__bnvs_name_tl`*.)*

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

(End definition for `\l__bnvs_path_tl`*.)*

Catch circular definitions. Open a main TEX group to define local functions and variables, sometimes another grouping level is used. The main TEX group is closed in the `\return_-...` functions.

```
1373 \prg_new_conditional:Nnn \__bnvs_if_append:nN { T, F, TF } {
1374   \__bnvs_call:TF {
1375     \__bnvs_group_begin:
```

This TEX group is closed just before returning. Local variables:

```
1376     \int_zero:N  \l__bnvs_split_int
1377     \seq_clear:N \l__bnvs_split_seq
1378     \tl_clear:N  \l__bnvs_id_tl
1379     \tl_clear:N  \l__bnvs_name_tl
1380     \tl_clear:N  \l__bnvs_path_tl
1381     \tl_clear:N  \l__bnvs_group_tl
1382     \tl_clear:N  \l__bnvs_ans_tl
1383     \tl_clear:N  \l__bnvs_a_tl
```

Implementation:

```
1384     \regex_split:NnN \c__bnvs_split_regex { #1 } \l__bnvs_split_seq
1385     \int_set:Nn \l__bnvs_split_int { 1 }
1386     \tl_set:Nx \l__bnvs_ans_tl {
1387       \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
1388     }
```

47

\switch:nNTF {⟨*capture group number*⟩} {⟨*tl variable*⟩} {⟨*black code*⟩} {⟨*white code*⟩}

Helper function to locally set the ⟨*tl variable*⟩ to the captured group ⟨*capture group number*⟩ and branch.

```
1389    \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1390      \tl_set:Nx ##2 {
1391        \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int + ##1 }
1392      }
1393      \tl_if_empty:NTF ##2 {
1394        ##4 } {
1395        ##3
1396      }
1397    }
1398    \cs_set:Npn \fp_round: {
1399      \__bnvs_round:N \l__bnvs_ans_tl
1400    }
```

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

```
1401    \cs_set:Npn \group_end_return_false: {
1402      \cs_set:Npn \loop: {
1403        \__bnvs_group_end:
1404        \prg_return_false:
1405      }
1406    }
1407    \cs_set:Npn \group_end_return_false:x ##1 {
1408      \__bnvs_error:x { ##1 }
1409      \group_end_return_false:
1410    }
1411    \cs_set:Npn \resolve_n:T ##1 {
1412      \__bnvs_resolve_n:TFF {
1413        ##1
1414      } {
1415        \group_end_return_false:x { Too~many~dotted~components:~#1 }
1416      } {
1417        \group_end_return_false:x { Unknown~dotted~path:~#1 }
1418      }
1419    }
1420    \cs_set:Npn \resolve_x:T ##1 {
1421      \__bnvs_resolve_x:TFF {
1422        ##1
1423      } {
1424        \group_end_return_false:x { Too~many~dotted~components:~#1 }
1425      } {
1426        \group_end_return_false:x { Unknown~dotted~path:~#1 }
1427      }
1428    }
1429    \cs_set:Npn \:nn ##1 ##2 {
1430      \switch:nNTF { ##1 } \l__bnvs_id_tl { } {
1431        \tl_set_eq:NN \l__bnvs_id_tl \l__bnvs_id_last_tl
1432        \tl_put_left:NV \l__bnvs_name_tl \l__bnvs_id_tl
1433      }
1434      \switch:nNTF { ##2 } \l__bnvs_path_tl {
1435        \seq_set_split:NnV \l__bnvs_path_seq { . } \l__bnvs_path_tl
```

```
1436            \seq_remove_all:Nn \l__bnvs_path_seq { }
1437        } {
1438            \seq_clear:N \l__bnvs_path_seq
1439        }
1440     }
1441     \cs_set:cpn {.n?:TF} ##1 ##2 {
1442        \seq_get_right:NNTF \l__bnvs_path_seq \l__bnvs_b_tl {
1443            \exp_args:NV
1444            \str_if_eq:nnTF \l__bnvs_b_tl { n } {
1445                \seq_pop_right:NN \l__bnvs_path_seq \l__bnvs_b_tl
1446                ##1
1447            } { ##2 }
1448        } { ##2 }
1449     }
1450     \cs_set:cpn {...++n:} {
1451        \__bnvs_group_begin:
1452        \__bnvs_resolve_n:TFF {
1453            \tl_clear:N \l__bnvs_b_tl
1454            \__bnvs_if_n_incr:VnNTF \l__bnvs_name_tl { 1 } \l__bnvs_b_tl {
1455                \exp_args:NNNV
1456                \__bnvs_group_end:
1457                \tl_set:Nn \l__bnvs_b_tl \l__bnvs_b_tl
1458                \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
1459                \resolve_x:T {
1460                    \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
1461                }
1462            } {
1463                \__bnvs_group_end:
1464            }
1465        } {
1466            \__bnvs_group_end:
1467            \group_end_return_false:x { Too~many~dotted~components:~#1 }
1468        } {
1469            \__bnvs_group_end:
1470            \group_end_return_false:
1471        }
1472     }
```

Main loop. The explanations given here apply to quite every case. We start by recovering the frame id and the dotted path. Then we resolve the slide range name and path to the last possible name and a void integer path. We raise if we cannot obtain a void integer path.

```
1473     \cs_set:Npn \loop: {
1474        \int_compare:nNnTF {
1475            \l__bnvs_split_int } < { \seq_count:N \l__bnvs_split_seq
1476        } {
1477            \switch:nNTF { 1 } \l__bnvs_name_tl {
1478                \:nn { 2 } { 3 }
1479                \use:c {.n?:TF} {
```

- Case `++...n`.

```
1480                    \use:c { ...++n: }
1481                } {
```

49

- Case ++⟨*name*⟩⟨*integer path*⟩.

```
1482              \resolve_n:T {
1483                \tl_clear:N \l__bnvs_ans_tl
1484                \__bnvs_if_incr:VnNF \l__bnvs_name_tl 1 \l__bnvs_ans_tl {
1485                  \group_end_return_false:
1486                }
1487              }
1488            }
1489          } {
1490            \switch:nNTF 4 \l__bnvs_name_tl {
1491              \:nn { 5 } { 6 }
1492              \switch:nNTF 7 \l__bnvs_a_tl {
```

- Case ...++n.

```
1493                  \use:c { ...++n: }
1494                } {
1495                  \switch:nNTF 8 \l__bnvs_a_tl {
1496                    \use:c { .n?:TF } {
```

- Case ....n+=⟨*integer*⟩.

```
1497 \__bnvs_group_begin:
1498 \__bnvs_resolve_n:TFF {
1499   \tl_clear:N \l__bnvs_b_tl
1500   \__bnvs_if_n_incr:VVNTF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_b_tl {
1501     \exp_args:NNNV
1502     \__bnvs_group_end:
1503     \tl_set:Nn \l__bnvs_b_tl \l__bnvs_b_tl
1504     \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
1505     \resolve_x:T {
1506       \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
1507     }
1508   } {
1509     \__bnvs_group_end:
1510   }
1511 } {
1512     \__bnvs_group_end:
1513     \group_end_return_false:x { Too~many~dotted~components:~#1 }
1514 } {
1515   \__bnvs_group_end:
1516   \group_end_return_false:x { Unknown~dotted~path:~#1 }
1517 }
1518                  } {
```

- Case A+=⟨*integer*⟩.

```
1519 \resolve_n:T {
1520   \__bnvs_if_incr:VVNF \l__bnvs_name_tl \l__bnvs_a_tl \l__bnvs_ans_tl {
1521     \group_end_return_false:
1522   }
1523 }
1524                  }
1525                } {
1526                  \switch:nNTF 9 \l__bnvs_a_tl {
```

- Case ...++.

```
1527 \resolve_n:T {
1528   \__bnvs_if_post:VnNF \l__bnvs_name_tl { 1 } \l__bnvs_ans_tl {
1529     \return_false:
1530   }
1531 }
1532             } {
```

Only the path, branch according to the last component.

```
1533 \seq_pop_right:NNTF \l__bnvs_path_seq \l__bnvs_b_tl {
1534   \exp_args:NV
1535   \str_case:nnF \l__bnvs_b_tl {
1536     { n } {
```

- Case ...n.

```
1537       \__bnvs_group_begin:
1538       \resolve_n:T {
1539         \exp_args:NNV
1540         \__bnvs_group_end:
1541         \__bnvs_if_n_value:nNTF \l__bnvs_name_tl \l__bnvs_b_tl {
1542           \seq_put_right:NV \l__bnvs_path_seq \l__bnvs_b_tl
1543           \resolve_x:T {
1544             \tl_put_right:NV \l__bnvs_ans_tl \l__bnvs_name_tl
1545           }
1546         } {
1547 \group_end_return_false:x { Undefined~dotted~path:~#1 }
1548         }
1549       }
1550     }
1551     { length } {
```

- Case ...length.

```
1552       \resolve_n:T {
1553         \__bnvs_raw_length:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1554           \group_end_return_false:
1555         }
1556       }
1557     }
1558     { last } {
```

- Case ...last.

```
1559       \resolve_n:T {
1560         \__bnvs_raw_last:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1561           \group_end_return_false:
1562         }
1563       }
1564     }
1565     { range } {
```

- Case ...range.

```
1566          \resolve_n:T {
1567            \__bnvs_if_range:VNTF \l__bnvs_name_tl \l__bnvs_ans_tl {
1568              \cs_set_eq:NN \fp_round: \prg_do_nothing:
1569            } {
1570              \group_end_return_false:
1571            }
1572          }
1573        }
1574      { previous } {
```

- Case ...previous.

```
1575          \resolve_n:T {
1576            \__bnvs_if_previous:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1577              \group_end_return_false:
1578            }
1579          }
1580        }
1581      { next } {
```

- Case ...next.

```
1582          \resolve_n:T {
1583            \__bnvs_if_next:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1584              \group_end_return_false:
1585            }
1586          }
1587        }
1588    } {
```

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

```
1589      \resolve_n:T {
1590        \__bnvs_if_index:VVNF \l__bnvs_name_tl \l__bnvs_b_tl \l__bnvs_ans_tl {
1591          \group_end_return_false:
1592        }
1593      }
1594    }
1595  } {
```

- Case ....

```
1596  \resolve_n:T {
1597    \__bnvs_raw_value:VNF \l__bnvs_name_tl \l__bnvs_ans_tl {
1598      \group_end_return_false:
1599    }
1600  }
1601 }
1602              }
1603            }
1604          }
1605        } {
```

No name. Unreachable code.

```
            }
          }
        \int_add:Nn \l__bnvs_split_int { 10 }
        \tl_put_right:Nx \l__bnvs_ans_tl {
          \seq_item:Nn \l__bnvs_split_seq { \l__bnvs_split_int }
        }
        \loop:
      } {
        \fp_round:
        \exp_args:NNNV
        \__bnvs_group_end:
        \tl_put_right:Nn #2 \l__bnvs_ans_tl
        \prg_return_true:
      }
    }
    \loop:
  } {
    \__bnvs_error:x { Too~many~calls:~ #1 }
    \prg_return_false:
  }
}
\prg_generate_conditional_variant:Nnn
  \__bnvs_if_append:nN { VN } { T, F, TF }
```

__bnvs_if_eval_query:nNTF {⟨*overlay query*⟩} ⟨*tl variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\l__bnvs_a_tl Storage for the first index of a range.

(*End definition for* \l__bnvs_a_tl.)

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

(*End definition for* \l__bnvs_b_tl.)

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

(*End definition for* \c__bnvs_A_cln_Z_regex.)

```
1629 \regex_const:Nn \c__bnvs_A_cln_Z_regex {
1630   \A \s* (?:
```

- 2: ⟨*first*⟩

```
1631       ( [^:]* ) \s* :
```

- 3: second optional colon

```
1632       (:)? \s*
```

- 4: ⟨*length*⟩

```
1633       ( [^:]* )
```

- 5: standalone ⟨*first*⟩

```
1634     | ( [^:]+ )
1635   ) \s* \Z
1636 }
```

```
1637 \prg_new_conditional:Npnn \__bnvs_if_eval_query:nN #1 #2 { T, F, TF } {
1638   \__bnvs_call_greset:
1639   \cs_set:Npn \return_true: {
1640     \prg_return_true:
1641   }
1642   \cs_set:Npn \return_false: {
1643     \prg_return_false:
1644   }
1645   \regex_extract_once:NnNTF \c__bnvs_A_cln_Z_regex {
1646     #1
1647   } \l__bnvs_match_seq {
```

\switch:nNTF {⟨*capture group number*⟩} ⟨*tl variable*⟩ {⟨*black code*⟩} {⟨*white code*⟩}

Helper function to locally set the ⟨*tl variable*⟩ to the captured group ⟨*capture group number*⟩ and branch depending on the emptyness of this variable.

```
1648      \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 {
1649        \tl_set:Nx ##2 {
1650          \seq_item:Nn \l__bnvs_match_seq { ##1 }
1651        }
1652        \tl_if_empty:NTF ##2 { ##4 } { ##3 }
1653      }
1654      \switch:nNTF 5 \l__bnvs_a_tl {
```

🔍 Single expression

```
1655        \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1656          \return_true:
1657        } {
1658          \return_false:
1659        }
1660      } {
1661        \switch:nNTF 2 \l__bnvs_a_tl {
1662          \switch:nNTF 4 \l__bnvs_b_tl {
1663            \switch:nNTF 3 \l__bnvs_c_tl {
```

🔍 ⟨*first*⟩::⟨*last*⟩ range

```
1664              \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1665                \tl_put_right:Nn #2 { - }
1666                \__bnvs_if_append:VNTF \l__bnvs_b_tl #2 {
1667                  \return_true:
1668                } {
1669                  \return_false:
1670                }
1671              } {
1672                \return_false:
1673              }
1674            } {
```

🔍 ⟨*first*⟩:⟨*length*⟩ range

```
1675              \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1676                \tl_put_right:Nx #2 { - }
1677                \tl_put_right:Nx \l__bnvs_a_tl { + ( \l__bnvs_b_tl ) - 1}
1678                \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1679                  \return_true:
1680                } {
1681                  \return_false:
1682                }
1683              } {
1684                \return_false:
1685              }
1686            }
1687          } {
```

🔍 ⟨*first*⟩: and ⟨*first*⟩:: range

```
1688          \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1689            \tl_put_right:Nn #2 { - }
1690            \return_true:
1691          } {
1692            \return_false:
1693          }
1694        }
1695      } {
1696        \switch:nNTF 4 \l__bnvs_b_tl {
1697          \switch:nNTF 3 \l__bnvs_c_tl {
```

🗨 ::⟨*last*⟩ range

```
1698            \tl_put_right:Nn #2 { - }
1699            \__bnvs_if_append:VNTF \l__bnvs_a_tl #2 {
1700              \return_true:
1701            } {
1702              \return_false:
1703            }
1704          } {
1705            \__bnvs_error:x { Syntax~error(Missing~first):~#1 }
1706          }
1707        } {
```

🗨 : or :: range

```
1708            \seq_put_right:Nn #2 { - }
1709          }
1710        }
1711      }
1712    } {
```

Error

```
1713      \__bnvs_error:n { Syntax~error:~#1 }
1714      \return_false:
1715    }
1716 }
```

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

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

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

(*End definition for* `\l__bnvs_query_seq`.)

`\l__bnvs_ans_seq` Storage of the evaluated result.

(*End definition for* `\l__bnvs_ans_seq`.)

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

```
1717 \regex_const:Nn \c__bnvs_comma_regex { \s* , \s* }
```

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

```
1718 \cs_new:Npn \__bnvs_eval:nN #1 #2 {
1719   \__bnvs_group_begin:
```

Local variables declaration

```
1720   \seq_clear:N \l__bnvs_query_seq
1721   \seq_clear:N \l__bnvs_ans_seq
```

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

```
1722   \regex_split:NnN \c__bnvs_comma_regex { #1 } \l__bnvs_query_seq
```

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

```
1723   \seq_map_inline:Nn \l__bnvs_query_seq {
1724     \tl_clear:N \l__bnvs_ans_tl
1725     \__bnvs_if_eval_query:nNTF { ##1 } \l__bnvs_ans_tl {
1726       \seq_put_right:NV \l__bnvs_ans_seq \l__bnvs_ans_tl
1727     } {
1728       \seq_map_break:n {
1729         \__bnvs_fatal:n { Circular/Undefined~dependency~in~#1}
1730       }
1731     }
1732   }
```

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

```
1733   \exp_args:NNNx
1734   \__bnvs_group_end:
1735   \tl_put_right:Nn #2 { \seq_use:Nn \l__bnvs_ans_seq , }
1736 }
1737 \cs_generate_variant:Nn \__bnvs_eval:nN { VN, xN }
```

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

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

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

```
1738 \NewDocumentCommand \BeanovesEval { o m } {
1739   \__bnvs_group_begin:
1740   \tl_clear:N \l__bnvs_ans_tl
1741   \__bnvs_eval:nN { #2 } \l__bnvs_ans_tl
1742   \IfValueTF { #1 } {
1743     \exp_args:NNNV
1744     \__bnvs_group_end:
1745     \tl_set:Nn #1 \l__bnvs_ans_tl
1746   } {
1747     \exp_args:NV
1748     \__bnvs_group_end: \l__bnvs_ans_tl
1749   }
1750 }
```

6.8.9 Reseting counters

\BeanovesReset   \beanovesReset [⟨*first value*⟩] {⟨*key*⟩}
\BeanovesReset*   \beanovesReset* [⟨*first value*⟩] {⟨*key*⟩}

Forwards to `\__bnvs_reset:nn` or `\__bnvs_reset_all:nn` when starred.

```
1751 \NewDocumentCommand \BeanovesReset { s O{} m } {
1752   \__bnvs_id_name_set:nNNTF { #3 } \l__bnvs_id_tl \l__bnvs_name_tl {
1753     \IfBooleanTF { #1 } {
1754       \exp_args:NV \__bnvs_reset_all:nn
1755     } {
1756       \exp_args:NV \__bnvs_reset:nn
1757     }
1758     \l__bnvs_name_tl { #2 }
1759   } {
1760     \__bnvs_warning:n { Unknown~name:~#1 }
1761   }
1762   \ignorespaces
1763 }
```

__bnvs_reset:nn   __bnvs_reset:nn {⟨*key*⟩} {⟨*first value*⟩}
__bnvs_reset_all:nn

The key must include the frame id. Reset the value counter to the given ⟨*first value*⟩. The _all version also cleans the cached values.

```
1764 \cs_new:Npn \__bnvs_reset_all:nn #1 #2 {
1765   \bool_if:nTF {
1766       \__bnvs_if_in_p:nn A { #1 }
1767     || \__bnvs_if_in_p:nn Z { #1 }
1768     || \__bnvs_if_in_p:nn V { #1 }
1769   } {
```

```
1770    \__bnvs_gremove_cache:nn A { #1 }
1771    \__bnvs_gremove_cache:nn L { #1 }
1772    \__bnvs_gremove_cache:nn Z { #1 }
1773    \__bnvs_gremove_cache:nn P { #1 }
1774    \__bnvs_gremove_cache:nn N { #1 }
1775    \__bnvs_gremove_cache:nn V { #1 }
1776    \tl_if_empty:nF { #2 } {
1777      \__bnvs_gput_cache:nnn V { #1 } { #2 }
1778    }
1779  } {
1780    \__bnvs_warning:n { Unknown~name:~#1 }
1781  }
1782 }
1783 \cs_new:Npn \__bnvs_reset:nn #1 #2 {
1784  \__bnvs_if_in:nnTF V { #1 } {
1785    \__bnvs_gremove_cache:nn V { #1 }
1786    \tl_if_empty:nF { #2 } {
1787      \__bnvs_gput_cache:nnn V { #1 } { #2 }
1788    }
1789  } {
1790    \__bnvs_warning:n { Unknown~name:~#1 }
1791  }
1792 }
1793 \makeatother
1794 \ExplSyntaxOff
```