The zref-clever package implementation[*]

Gustavo Barros[†]

2021-09-29

Contents

[*]This file describes v0.1.0-alpha, released 2021-09-29.
[†]https://github.com/gusbrs/zref-clever

1 Initial setup

Start the DocStrip guards.

```
⟨*package⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention).

```
⟨@@=zrefclever⟩
```

Taking a stance on backward compatibility of the package. During initial development, we have used freely recent features of the kernel (albeit refraining from l3candidates, even though I'd have loved to have used `\bool_case_true:...`). We presume xparse (which made to the kernel in the 2020-10-01 release), and expl3 as well (which made to the kernel in the 2020-02-02 release). We also just use UTF-8 for the dictionaries (which became the default input encoding in the 2018-04-01 release). Hence, since we would not be able to go much backwards without special handling anyway, we make the cut with the inclusion of the new hook management system (ltcmdhooks), which is bound to be useful for our purposes, and was released with the 2021-06-01 kernel.

CHECK Should I just go ahead and bump this to 2021-11-15 considering the appendix case?

```
\providecommand\IfFormatAtLeastTF{\@ifl@t@r\fmtversion}
\IfFormatAtLeastTF{2021-06-01}
  {}
  {%
    \PackageError{zref-clever}{LaTeX kernel too old}
      {%
        'zref-clever' requires a LaTeX kernel newer than 2021-06-01.%
        \MessageBreak Loading will abort!%
      }%
    \endinput
  }%
```

Identify the package.

```
\ProvidesExplPackage {zref-clever} {2021-09-29} {0.1.0-alpha}
  {Clever LaTeX cross-references based on zref}
```

2 Dependencies

Required packages. Besides these, zref-hyperref may also be required depending on the presence of hyperref itself and on the `hyperref` option.

```
\RequirePackage { zref-base }
\RequirePackage { zref-user }
\RequirePackage { zref-abspage }
\RequirePackage { l3keys2e }
```

3 zref setup

For the purposes of the package, we need to store some information with the labels, some of it standard, some of it not so much. So, we have to setup zref to do so.

Some basic properties are handled by zref itself, or some of its modules. The page property is provided by zref-base, while zref-abspage provides the abspage property which gives us a safe and easy way to sort labels for page references.

The counter property, in most cases, will be just the kernel's \@currentcounter, set by \refstepcounter. However, not everywhere is it assured that \@currentcounter gets updated as it should, so we need to have some means to manually tell zref-clever what the current counter actually is. This is done with the currentcounter option, and stored in \l__zrefclever_current_counter_tl, whose default is \@currentcounter.

```
\zref@newprop { zc@counter } { \l__zrefclever_current_counter_tl }
\zref@addprop \ZREF@mainlist { zc@counter }
```

The reference itself, stored by zref-base in the default property, is somewhat a disputed real estate. In particular, the use of \labelformat (previously from varioref, now in the kernel) will include there the reference "prefix" and complicate the job we are trying to do here. Hence, we isolate \the⟨*counter*⟩ and store it "clean" in zc@thecnt for reserved use. Based on the definition of \@currentlabel done inside \refstepcounter in 'texdoc source2e', section 'ltxref.dtx'. We just drop the \p@... prefix.

```
\zref@newprop { zc@thecnt }
  { \use:c { the \l__zrefclever_current_counter_tl } }
\zref@addprop \ZREF@mainlist { zc@thecnt }
```

Much of the work of zref-clever relies on the association between a label's "counter" and its "type" (see the User manual section on "Reference types"). Superficially examined, one might think this relation could just be stored in a global property list, rather than in the label itself. However, there are cases in which we want to distinguish different types for the same counter, depending on the document context. Hence, we need to store the "type" of the "counter" for each "label". In setting this, the presumption is that the label's type has the same name as its counter, unless it is specified otherwise by the countertype option, as stored in \l__zrefclever_counter_type_prop.

```
\zref@newprop { zc@type }
  {
    \exp_args:NNe \prop_if_in:NnTF \l__zrefclever_counter_type_prop
      \l__zrefclever_current_counter_tl
      {
        \exp_args:NNe \prop_item:Nn \l__zrefclever_counter_type_prop
          { \l__zrefclever_current_counter_tl }
      }
      { \l__zrefclever_current_counter_tl }
  }
\zref@addprop \ZREF@mainlist { zc@type }
```

Since the zc@thecnt and page properties store the "*printed* representation" of their respective counters, for sorting and compressing purposes, we are also interested in their numeric values. So we store them in zc@cntval and zc@pgval. For this, we use \c@⟨*counter*⟩, which contains the counter's numerical value (see 'texdoc source2e', section 'ltcounts.dtx').

```
\zref@newprop { zc@cntval } [0]
  { \int_use:c { c@ \l__zrefclever_current_counter_tl } }
```

```
\zref@addprop \ZREF@mainlist { zc@cntval }
\zref@newprop* { zc@pgval } [0] { \int_use:c { c@page } }
\zref@addprop \ZREF@mainlist { zc@pgval }
```

However, since many counters (may) get reset along the document, we require more than just their numeric values. We need to know the reset chain of a given counter, in order to sort and compress a group of references. Also here, the "printed representation" is not enough, not only because it is easier to work with the numeric values but, given we occasionally group multiple counters within a single type, sorting this group requires to know the actual counter reset chain (the counters' names and values). Indeed, the set of counters grouped into a single type cannot be arbitrary: all of them must belong to the same reset chain, and must be nested within each other (they cannot even just share the same parent).

Furthermore, even if it is true that most of the definitions of counters, and hence of their reset behavior, is likely to be defined in the preamble, this is not necessarily true. Users can create counters, newtheorems mid-document, and alter their reset behavior along the way. Was that not the case, we could just store the desired information at `begindocument` in a variable and retrieve it when needed. But since it is, we need to store the information with the label, with the values as current when the label is set.

Though counters can be reset at any time, and in different ways at that, the most important use case is the automatic resetting of counters when some other counter is stepped, as performed by the standard mechanisms of the kernel (optional argument of `\newcounter`, `\@addtoreset`, `\counterwithin`, and related infrastructure). The canonical optional argument of `\newcounter` establishes that the counter being created (the mandatory argument) gets reset every time the "enclosing counter" gets stepped (this is called in the usual sources "within-counter", "old counter", "supercounter", "parent counter" etc.). This information is a little trickier to get. For starters, the counters which may reset the current counter are not retrievable from the counter itself, because this information is stored with the counter that does the resetting, not with the one that gets reset (the list is stored in `\cl@`⟨*counter*⟩ with format `\@elt{countera}\@elt{counterb}\@elt{counterc}`, see section 'ltcounts.dtx' in 'source2e'). Besides, there may be a chain of resetting counters, which must be taken into account: if 'counterC' gets reset by 'counterB', and 'counterB' gets reset by 'counterA', stepping the latter affects all three of them.

The procedure below examines a set of counters, those included in `\l__zrefclever_counter_resetters_seq`, and for each of them retrieves the set of counters it resets, as stored in `\cl@`⟨*counter*⟩, looking for the counter for which we are trying to set a label (`\l__zrefclever_current_counter_tl`, by default `\@currentcounter`, passed as an argument to the functions). There is one relevant caveat to this procedure: `\l__zrefclever_counter_resetters_seq` is populated by hand with the "usual suspects", there is no way (that I know of) to ensure it is exhaustive. However, it is not that difficult to create a reasonable "usual suspects" list which, of course, should include the counters for the sectioning commands to start with, and it is easy to add more counters to this list if needed, with the option `counterresetters`. Unfortunately, not all counters are created alike, or reset alike. Some counters, even some kernel ones, get reset by other mechanisms (notably, the `enumerate` environment counters do not use the regular counter machinery for resetting on each level, but are nested nevertheless by other means). Therefore, inspecting `\cl@`⟨*counter*⟩ cannot possibly fully account for all of the automatic counter resetting which takes place in the document. And there's also no other "general rule" we could grab on for this, as far as I know. So we provide a way to manually tell zref-clever of these cases, by means of the `counterresetby` option, whose information is stored in `\l__zrefclever_counter_resetby_prop`. This manual specification has precedence over the search through `\l__zrefclever_counter_resetters_seq`, and should be handled with care, since there is no possible verification mechanism for this.

`\__zrefclever_get_enclosing_counters:n`
`\__zrefclever_get_enclosing_counters_value:n`

Recursively generate a *sequence* of “enclosing counters” and values, for a given ⟨*counter*⟩ and leave it in the input stream. These functions must be expandable, since they get called from `\zref@newprop` and are the ones responsible for generating the desired information when the label is being set. Note that the order in which we are getting this information is reversed, since we are navigating the counter reset chain bottom-up. But it is very hard to do otherwise here where we need expandable functions, and easy to handle at the reading side.

```
\__zrefclever_get_enclosing_counters:n {⟨counter⟩}
\__zrefclever_get_enclosing_counters_value:n {⟨counter⟩}
```

```
\cs_new:Npn \__zrefclever_get_enclosing_counters:n #1
  {
    \cs_if_exist:cT { c@ \__zrefclever_counter_reset_by:n {#1} }
      {
        { \__zrefclever_counter_reset_by:n {#1} }
        \__zrefclever_get_enclosing_counters:e
          { \__zrefclever_counter_reset_by:n {#1} }
      }
  }
\cs_new:Npn \__zrefclever_get_enclosing_counters_value:n #1
  {
    \cs_if_exist:cT { c@ \__zrefclever_counter_reset_by:n {#1} }
      {
        { \int_use:c { c@ \__zrefclever_counter_reset_by:n {#1} } }
        \__zrefclever_get_enclosing_counters_value:e
          { \__zrefclever_counter_reset_by:n {#1} }
      }
  }
```

Both `e` and `f` expansions work for this particular recursive call. I’ll stay with the `e` variant, since conceptually it is what I want (`x` itself is not expandable), and this package is anyway not compatible with older kernels for which the performance penalty of the `e` expansion would ensue (see also https://tex.stackexchange.com/q/611370/#comment1529282_611385, thanks Enrico Gregorio, aka ‘egreg’).

```
\cs_generate_variant:Nn \__zrefclever_get_enclosing_counters:n { e }
\cs_generate_variant:Nn \__zrefclever_get_enclosing_counters_value:n { e }
```

(*End definition for* `\__zrefclever_get_enclosing_counters:n` *and* `\__zrefclever_get_enclosing_counters_value:n`.)

`\__zrefclever_counter_reset_by:n`

Auxiliary function for `\__zrefclever_get_enclosing_counters:n` and `\__zrefclever_get_enclosing_counters_value:n`. They are broken in parts to be able to use the expandable mapping functions. `\__zrefclever_counter_reset_by:n` leaves in the stream the “enclosing counter” which resets ⟨*counter*⟩.

```
\__zrefclever_counter_reset_by:n {⟨counter⟩}
```

```
\cs_new:Npn \__zrefclever_counter_reset_by:n #1
  {
    \bool_if:nTF
      { \prop_if_in_p:Nn \l__zrefclever_counter_resetby_prop {#1} }
      { \prop_item:Nn  \l__zrefclever_counter_resetby_prop {#1} }
      {
        \seq_map_tokens:Nn \l__zrefclever_counter_resetters_seq
          { \__zrefclever_counter_reset_by_aux:nn {#1} }
      }
  }
\cs_new:Npn \__zrefclever_counter_reset_by_aux:nn #1#2
  {
    \cs_if_exist:cT { c@ #2 }
      {
        \tl_if_empty:cF { cl@ #2 }
          {
            \tl_map_tokens:cn { cl@ #2 }
              { \__zrefclever_counter_reset_by_auxi:nnn {#2} {#1} }
          }
      }
  }
\cs_new:Npn \__zrefclever_counter_reset_by_auxi:nnn #1#2#3
  {
    \str_if_eq:nnT {#2} {#3}
      { \tl_map_break:n { \seq_map_break:n {#1} } }
  }
```

(*End definition for* `\__zrefclever_counter_reset_by:n`*.*)

Finally, we create the `zc@enclcnt` and `zc@enclval` properties, and add them to the `main` property list.

```
\zref@newprop { zc@enclcnt }
  { \__zrefclever_get_enclosing_counters:e \l__zrefclever_current_counter_tl }
\zref@newprop { zc@enclval }
  { \__zrefclever_get_enclosing_counters_value:e \l__zrefclever_current_counter_tl }
\zref@addprop \ZREF@mainlist { zc@enclcnt }
\zref@addprop \ZREF@mainlist { zc@enclval }
```

Another piece of information we need is the page numbering format being used by `\thepage`, so that we know when we can (or not) group a set of page references in a range. Unfortunately, `page` is not a typical counter in ways which complicates things. First, it does commonly get reset along the document, not necessarily by the usual counter reset chains, but rather with `\pagenumbering` or variations thereof. Second, the format of the page number commonly changes in the document (roman, arabic, etc.), not necessarily, though usually, together with a reset. Trying to "parse" `\thepage` to retrieve such information is bound to go wrong: we don't know, and can't know, what is within that macro, and that's the business of the user, or of the documentclass, or of the loaded packages. The technique used by cleveref, which we borrow here, is simple and smart: store with the label what `\thepage` would return, if the counter `\c@page` was "1". That does not allow us to *sort* the references, luckily however, we have `abspage` which solves this problem. But we can decide whether two labels can be compressed into a range or not based on this format: if they are identical, we can compress them, otherwise, we can't. To do so, we locally redefine `\c@page` to return "1", thus avoiding any global spillovers of this trick. Since this operation is not expandable we cannot run it directly from the property definition. Hence, we use a shipout hook, and set `\g__zrefclever_page_format_tl`, which can then be retrieved by the starred definition of `\zref@newprop*{zc@pgfmt}`.

```
\tl_new:N \g__zrefclever_page_format_tl
\cs_new_protected:Npx \__zrefclever_page_format_aux: { \int_eval:n { 1 } }
\AddToHook { shipout / before }
  {
    \group_begin:
    \cs_set_eq:NN \c@page \__zrefclever_page_format_aux:
    \exp_args:NNx \tl_gset:Nn \g__zrefclever_page_format_tl { \thepage }
    \group_end:
  }
\zref@newprop* { zc@pgfmt } { \g__zrefclever_page_format_tl }
\zref@addprop \ZREF@mainlist { zc@pgfmt }
```

Still another property which we don't need to handle at the data provision side, but need to cater for at the retrieval side, is the `url` property (or the equivalent `urluse`) from the zref-xr module, which is added to the labels imported from external documents, and needed to construct hyperlinks to them.

4 Plumbing

4.1 Messages

```
\msg_new:nnn { zref-clever } { option-not-type-specific }
  {
    Option~'#1'~is~not~type-specific~\msg_line_context:.~
    Set~it~in~'\iow_char:N\\zcLanguageSetup'~before~first~'type'
    ~switch~or~as~package~option.
  }
\msg_new:nnn { zref-clever } { option-only-type-specific }
  {
    No~type~specified~for~option~'#1'~\msg_line_context:.~
    Set~it~after~'type'~switch~or~in~'\iow_char:N\\zcRefTypeSetup'.
  }
\msg_new:nnn { zref-clever } { key-requires-value }
  { The~'#1'~key~'#2'~requires~a~value~\msg_line_context:. }
\msg_new:nnn { zref-clever } { language-declared }
  { Language~'#1'~is~already~declared.~Nothing~to~do. }
\msg_new:nnn { zref-clever } { unknown-language-alias }
  {
    Language~'#1'~is~unknown,~cannot~alias~to~it.~See~documentation~for~
    '\iow_char:N\\zcDeclareLanguage'~and~
    '\iow_char:N\\zcDeclareLanguageAlias'.
  }
\msg_new:nnn { zref-clever } { unknown-language-transl }
  {
    Language~'#1'~is~unknown,~cannot~declare~translations~to~it.~
    See~documentation~for~'\iow_char:N\\zcDeclareLanguage'~and~
    '\iow_char:N\\zcDeclareLanguageAlias'.
  }
\msg_new:nnn { zref-clever } { unknown-language-opt }
  {
    Language~'#1'~is~unknown~\msg_line_context:.~Using~default.~
    See~documentation~for~'\iow_char:N\\zcDeclareLanguage'~and~
    '\iow_char:N\\zcDeclareLanguageAlias'.
  }
\msg_new:nnn { zref-clever } { dict-loaded }
  { Loaded~'#1'~dictionary. }
\msg_new:nnn { zref-clever } { dict-not-available }
  { Dictionary~for~'#1'~not~available~\msg_line_context:. }
\msg_new:nnn { zref-clever } { unknown-language-load }
  {
    Language~'#1'~is~unknown~\msg_line_context:.~Unable~to~load~dictionary.~
    See~documentation~for~'\iow_char:N\\zcDeclareLanguage'~and~
    '\iow_char:N\\zcDeclareLanguageAlias'.
  }
\msg_new:nnn { zref-clever } { missing-zref-titleref }
  {
    Option~'ref=title'~requested~\msg_line_context:.~
    But~package~'zref-titleref'~is~not~loaded,~falling-back~to~default~'ref'.
  }
\msg_new:nnn { zref-clever } { hyperref-preamble-only }
  {
    Option~'hyperref'~only~available~in~the~preamble.~
    Use~the~starred~version~of~'\iow_char:N\\zcref'~instead.
  }
\msg_new:nnn { zref-clever } { missing-hyperref }
  { Missing~'hyperref'~package.~Setting~'hyperref=false'. }
\msg_new:nnn { zref-check } { check-document-only }
  { Option~'check'~only~available~in~the~document. }
\msg_new:nnn { zref-clever } { missing-zref-check }
  {
    Option~'check'~requested~\msg_line_context:.~
    But~package~'zref-check'~is~not~loaded,~can't~run~the~checks.
  }
\msg_new:nnn { zref-clever } { counters-not-nested }
  { Counters~not~nested~for~labels~'#1'~and~'#2'~\msg_line_context:. }
\msg_new:nnn { zref-clever } { missing-type }
  { Reference~type~undefined~for~label~'#1'~\msg_line_context:. }
\msg_new:nnn { zref-clever } { missing-name }
  { Name~undefined~for~type~'#1'~\msg_line_context:. }
\msg_new:nnn { zref-clever } { missing-string }
  {
    We~couldn't~find~a~value~for~reference~option~'#1'~\msg_line_context:.~
    But~we~should~have:~throw~a~rock~at~the~maintainer.
  }
\msg_new:nnn { zref-clever } { single-element-range }
  { Range~for~type~'#1'~resulted~in~single~element~\msg_line_context:. }
\msg_new:nnn { zref-clever } { compat-package }
  { Loaded~support~for~'#1'~package. }
\msg_new:nnn { zref-clever } { compat-class }
  { Loaded~support~for~'#1'~documentclass. }
```

4.2 Reference format

For a general discussion on the precedence rules for reference format options, see Section “Reference format” in the User manual. Internally, these precedence rules are handled / enforced in `\__zrefclever_get_ref_string:nN`, `\__zrefclever_get_ref_-font:nN`, and `\__zrefclever_type_name_setup:` which are the basic functions to retrieve proper values for reference format settings. The "fallback" settings are stored in `\g__zrefclever_fallback_dict_prop`.

\l__zrefclever_setup_type_tl
\l__zrefclever_dict_language_tl

Store "current" type and language in different places for option and translation handling, notably in `\__zrefclever_provide_dictionary:n`, `\zcRefTypeSetup`, and `\zcLanguageSetup`. But also for translations retrieval, in `\__zrefclever_get_type_-transl:nnnN` and `\__zrefclever_get_default_transl:nnN`.

```
\tl_new:N \l__zrefclever_setup_type_tl
\tl_new:N \l__zrefclever_dict_language_tl
```

(*End definition for* `\l__zrefclever_setup_type_tl` *and* `\l__zrefclever_dict_language_tl`*.*)

f_options_necessarily_not_type_specific_seq
ever_ref_options_possibly_type_specific_seq
r_ref_options_necessarily_type_specific_seq
\c__zrefclever_ref_options_font_seq
\c__zrefclever_ref_options_typesetup_seq
\c__zrefclever_ref_options_reference_seq

Lists of reference format related options in "categories". Since these options are set in different scopes, and at different places, storing the actual lists in centralized variables makes the job not only easier later on, but also keeps things consistent.

```
\seq_const_from_clist:Nn
  \c__zrefclever_ref_options_necessarily_not_type_specific_seq
  {
    tpairsep ,
    tlistsep ,
    tlastsep ,
    notesep ,
  }
\seq_const_from_clist:Nn
  \c__zrefclever_ref_options_possibly_type_specific_seq
  {
    namesep ,
    pairsep ,
    listsep ,
    lastsep ,
    rangesep ,
    refpre ,
    refpos ,
    refpre-in ,
    refpos-in ,
  }
```

Only "type names" are "necessarily type-specific", which makes them somewhat special on the retrieval side of things. In short, they don't have their values queried by `\__zrefclever_get_ref_string:nN`, but by `\__zrefclever_type_name_setup:`.

```
\seq_const_from_clist:Nn
  \c__zrefclever_ref_options_necessarily_type_specific_seq
  {
    Name-sg ,
    name-sg ,
    Name-pl ,
    name-pl ,
    Name-sg-ab ,
    name-sg-ab ,
    Name-pl-ab ,
    name-pl-ab ,
  }
```

`\c__zrefclever_ref_options_font_seq` are technically "possibly type-specific", but are not "language-specific", so we separate them.

```
\seq_const_from_clist:Nn
  \c__zrefclever_ref_options_font_seq
  {
    namefont ,
    reffont ,
    reffont-in ,
  }
\seq_new:N \c__zrefclever_ref_options_typesetup_seq
\seq_gconcat:NNN \c__zrefclever_ref_options_typesetup_seq
  \c__zrefclever_ref_options_possibly_type_specific_seq
  \c__zrefclever_ref_options_necessarily_type_specific_seq
\seq_gconcat:NNN \c__zrefclever_ref_options_typesetup_seq
  \c__zrefclever_ref_options_typesetup_seq
  \c__zrefclever_ref_options_font_seq
\seq_new:N \c__zrefclever_ref_options_reference_seq
\seq_gconcat:NNN \c__zrefclever_ref_options_reference_seq
  \c__zrefclever_ref_options_necessarily_not_type_specific_seq
  \c__zrefclever_ref_options_possibly_type_specific_seq
\seq_gconcat:NNN \c__zrefclever_ref_options_reference_seq
  \c__zrefclever_ref_options_reference_seq
  \c__zrefclever_ref_options_font_seq
```

(*End definition for* `\c__zrefclever_ref_options_necessarily_not_type_specific_seq` *and others.*)

4.3 Languages

\g__zrefclever_languages_prop

Stores the names of known languages and the mapping from "language name" to "dictionary name". Whether of not a language or alias is known to zref-clever is decided by its presence in this property list. A "base language" (loose concept here, meaning just "the name we gave for the dictionary in that particular language") is just like any other one, the only difference is that the "language name" happens to be the same as the "dictionary name", in other words, it is an "alias to itself".

```
\prop_new:N \g__zrefclever_languages_prop
```

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

\zcDeclareLanguage

Declare a new language for use with zref-clever. ⟨*language*⟩ is taken to be both the "language name" and the "dictionary name". If ⟨*language*⟩ is already known, just warn. `\zcDeclareLanguage` is preamble only.

```
\zcDeclareLanguage {⟨language⟩}
```

```
\NewDocumentCommand \zcDeclareLanguage { m }
  {
    \tl_if_empty:nF {#1}
      {
        \prop_if_in:NnTF \g__zrefclever_languages_prop {#1}
          { \msg_warning:nnn { zref-clever } { language-declared } {#1} }
          { \prop_gput:Nnn \g__zrefclever_languages_prop {#1} {#1} }
      }
  }
\@onlypreamble \zcDeclareLanguage
```

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

`\zcDeclareLanguageAlias` Declare ⟨*language alias*⟩ to be an alias of ⟨*aliased language*⟩. ⟨*aliased language*⟩ must be already known to zref-clever, as stored in `\g__zrefclever_languages_prop`. `\zcDeclareLanguageAlias` is preamble only.

```
\zcDeclareLanguageAlias {⟨language alias⟩} {⟨aliased language⟩}
```

```
\NewDocumentCommand \zcDeclareLanguageAlias { m m }
  {
    \tl_if_empty:nF {#1}
      {
        \prop_if_in:NnTF \g__zrefclever_languages_prop {#2}
          {
            \exp_args:NNnx
              \prop_gput:Nnn \g__zrefclever_languages_prop {#1}
                { \prop_item:Nn \g__zrefclever_languages_prop {#2} }
          }
          { \msg_warning:nnn { zref-clever } { unknown-language-alias } {#2} }
      }
  }
\@onlypreamble \zcDeclareLanguageAlias
```

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

4.4 Dictionaries

Contrary to general options and type options, which are always *local*, "dictionaries", "translations" or "language-specific settings" are always *global*. Hence, the loading of built-in dictionaries, as well as settings done with `\zcLanguageSetup`, should set the relevant variables globally.

The built-in dictionaries and their related infrastructure are designed to perform "on the fly" loading of dictionaries, "lazily" as needed. Much like babel does for languages not declared in the preamble, but used in the document. This offers some convenience, of course, and that's one reason to do it. But it also has the purpose of parsimony, of "loading the least possible". My expectation is that for most use cases, users will require a single language of the functionality of zref-clever – the main language of the document –, even in multilingual documents. Hence, even the set of babel or polyglossia "loaded languages", which would be the most tenable set if loading were restricted to the preamble, is bound to be an overshoot in typical cases. Therefore, we load at `begindocument` one single language (see `lang` option), as specified by the user in the preamble with the `lang` option or, failing any specification, the main language of the document, which is the default. Anything else is lazily loaded, on the fly, along the document.

This design decision has also implications to the *form* the dictionary files assumed. As far as my somewhat impressionistic sampling goes, dictionary or localization files of the most common packages in this area of functionality, are usually a set of commands which perform the relevant definitions and assignments in the preamble or at `begindocument`. This includes translator, translations, but also babel's `.ldf` files, and biblatex's `.lbx` files. I'm not really well acquainted with this machinery, but as far as I grasp, they all rely on some variation of `\ProvidesFile` and `\input`. And they can be safely `\input` without generating spurious content, because they rely on being loaded before the document has actually started. As far as I can tell, babel's "on the fly" functionality is not based on the `.ldf` files, but on the `.ini` files, and on `\babelprovide`. And the `.ini` files are not in this form, but actually resemble "configuration files" of sorts, which means they are read and processed somehow else than with just `\input`. So we do the more or less the same here. It seems a reasonable way to ensure we can load dictionaries on the fly robustly mid-document, without getting paranoid with the last bit of white-space in them, and without introducing any undue content on the stream when we cannot afford to do it. Hence, zref-clever's built-in dictionary files are a set of *key-value options* which are read from the file, and fed to `\keys_set:nn{zref-clever/dictionary}` by `\__zrefclever_provide_dictionary:n`. And they use the same syntax and options as `\zcLanguageSetup` does. The dictionary file itself is read with `\ExplSyntaxOn` with the usual implications for white-space and catcodes.

`\__zrefclever_provide_dictionary:n` is only meant to load the built-in dictionaries. For languages declared by the user, or for any settings to a known language made with `\zcLanguageSetup`, values are populated directly to a variable `\g__zrefclever_-dict_⟨language⟩_prop`, created as needed. Hence, there is no need to "load" anything in this case: definitions and assignments made by the user are performed immediately.

Provide

\g__zrefclever_loaded_dictionaries_seq

Used to keep track of whether a dictionary has already been loaded or not.

```
\seq_new:N \g__zrefclever_loaded_dictionaries_seq
```

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

\l__zrefclever_load_dict_verbose_bool

Controls whether `\__zrefclever_provide_dictionary:n` fails silently or verbosely in case of unknown languages or dictionaries not found.

```
\bool_new:N \l__zrefclever_load_dict_verbose_bool
```

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

__zrefclever_provide_dictionary:n

Load dictionary for known ⟨*language*⟩ if it is available and if it has not already been loaded.

```
    \__zrefclever_provide_dictionary:n {⟨language⟩}
```

```
\cs_new_protected:Npn \__zrefclever_provide_dictionary:n #1
  {
    \group_begin:
    \prop_get:NnNTF \g__zrefclever_languages_prop {#1}
      \l__zrefclever_dict_language_tl
      {
        \seq_if_in:NVF
          \g__zrefclever_loaded_dictionaries_seq
          \l__zrefclever_dict_language_tl
          {
            \exp_args:Nx \file_get:nnNTF
              { zref-clever- \l__zrefclever_dict_language_tl .dict }
              { \ExplSyntaxOn }
              \l_tmpa_tl
              {
                \prop_if_exist:cF
                  {
                    g__zrefclever_dict_
```

```
                      \l__zrefclever_dict_language_tl _prop
                    }
                    {
                      \prop_new:c
                        {
                          g__zrefclever_dict_
                          \l__zrefclever_dict_language_tl _prop
                        }
                    }
                  \tl_clear:N \l__zrefclever_setup_type_tl
                  \exp_args:NnV
                    \keys_set:nn { zref-clever / dictionary } \l_tmpa_tl
                  \seq_gput_right:NV \g__zrefclever_loaded_dictionaries_seq
                    \l__zrefclever_dict_language_tl
                  \msg_note:nnx { zref-clever } { dict-loaded }
                    { \l__zrefclever_dict_language_tl }
                }
                {
                  \bool_if:NT \l__zrefclever_load_dict_verbose_bool
                    {
                      \msg_warning:nnx { zref-clever } { dict-not-available }
                        { \l__zrefclever_dict_language_tl }
                    }
```

Even if we don't have the actual dictionary, we register it as "loaded". At this point, it is a known language, properly declared. There is no point in trying to load it multiple times, because users cannot really provide the dictionary files (well, technically they could, but we are working so they don't need to, and have better ways to do what they want). And if the users had provided some translations themselves, by means of `\zcLanguageSetup`, everything would be in place, and they could use the `lang` option multiple times, and the `dict-not-available` warning would never go away.

```
                  \seq_gput_right:NV \g__zrefclever_loaded_dictionaries_seq
                    \l__zrefclever_dict_language_tl
                }
            }
        }
        {
          \bool_if:NT \l__zrefclever_load_dict_verbose_bool
            { \msg_warning:nnn { zref-clever } { unknown-language-load } {#1} }
        }
      \group_end:
  }
\cs_generate_variant:Nn \__zrefclever_provide_dictionary:n { x }
```

(*End definition for* `\__zrefclever_provide_dictionary:n`.)

`\__zrefclever_provide_dictionary_verbose:n` Does the same as `\__zrefclever_provide_dictionary:n`, but warns if the loading of the dictionary has failed.

```
\__zrefclever_provide_dictionary_verbose:n {⟨language⟩}
```

```
\cs_new_protected:Npn \__zrefclever_provide_dictionary_verbose:n #1
  {
    \group_begin:
    \bool_set_true:N \l__zrefclever_load_dict_verbose_bool
    \__zrefclever_provide_dictionary:n {#1}
    \group_end:
  }
\cs_generate_variant:Nn \__zrefclever_provide_dictionary_verbose:n { x }
```

(*End definition for* `\__zrefclever_provide_dictionary_verbose:n`*.*)

__zrefclever_provide_dict_type_transl:nn
__zrefclever_provide_dict_default_transl:nn

A couple of auxiliary functions for the of `zref-clever/dictionary` keys set in `\__zrefclever_provide_dictionary:n`. They respectively "provide" (i.e. set if it value does not exist, do nothing if it already does) "type-specific" and "default" translations. Both receive ⟨*key*⟩ and ⟨*translation*⟩ as arguments, but `\__zrefclever_provide_dict_type_transl:nn` relies on the current value of `\l__zrefclever_setup_type_tl`, as set by the `type` key.

```
\__zrefclever_provide_dict_type_transl:nn {⟨key⟩} {⟨translation⟩}
\__zrefclever_provide_dict_default_transl:nn {⟨key⟩} {⟨translation⟩}
```

```
\cs_new_protected:Npn \__zrefclever_provide_dict_type_transl:nn #1#2
  {
    \exp_args:Nnx \prop_gput_if_new:cnn
      { g__zrefclever_dict_ \l__zrefclever_dict_language_tl _prop }
      { type- \l__zrefclever_setup_type_tl - #1 } {#2}
  }
\cs_new_protected:Npn \__zrefclever_provide_dict_default_transl:nn #1#2
  {
    \prop_gput_if_new:cnn
      { g__zrefclever_dict_ \l__zrefclever_dict_language_tl _prop }
      { default- #1 } {#2}
  }
```

(*End definition for* `\__zrefclever_provide_dict_type_transl:nn` *and* `\__zrefclever_provide_dict_default_transl:nn`*.*)

The set of keys for `zref-clever/dictionary`, which is used to process the dictionary files in `\__zrefclever_provide_dictionary:n`. The no-op cases for each category have their messages sent to "info". These messages should not occur, as long as the dictionaries are well formed, but they're placed there nevertheless, and can be leveraged in regression tests.

```
\keys_define:nn { zref-clever / dictionary }
  {
    type .code:n =
      {
        \tl_if_empty:nTF {#1}
          { \tl_clear:N \l__zrefclever_setup_type_tl }
          { \tl_set:Nn \l__zrefclever_setup_type_tl {#1} }
      } ,
  }
\seq_map_inline:Nn
  \c__zrefclever_ref_options_necessarily_not_type_specific_seq
  {
    \keys_define:nn { zref-clever / dictionary }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          {
            \tl_if_empty:NTF \l__zrefclever_setup_type_tl
              { \__zrefclever_provide_dict_default_transl:nn {#1} {##1} }
              {
                \msg_info:nnn { zref-clever }
                  { option-not-type-specific } {#1}
              }
          } ,
      }
  }
\seq_map_inline:Nn
  \c__zrefclever_ref_options_possibly_type_specific_seq
  {
    \keys_define:nn { zref-clever / dictionary }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          {
            \tl_if_empty:NTF \l__zrefclever_setup_type_tl
              { \__zrefclever_provide_dict_default_transl:nn {#1} {##1} }
              { \__zrefclever_provide_dict_type_transl:nn {#1} {##1} }
          } ,
      }
  }
\seq_map_inline:Nn
  \c__zrefclever_ref_options_necessarily_type_specific_seq
  {
    \keys_define:nn { zref-clever / dictionary }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          {
            \tl_if_empty:NTF \l__zrefclever_setup_type_tl
              {
                \msg_info:nnn { zref-clever }
                  { option-only-type-specific } {#1}
              }
              { \__zrefclever_provide_dict_type_transl:nn {#1} {##1} }
          } ,
      }
  }
```

Fallback

All "strings" queried with `\__zrefclever_get_ref_string:nN` – in practice, those in either `\c__zrefclever_ref_options_necessarily_not_type_specific_seq` or `\c__zrefclever_ref_options_possibly_type_specific_seq` – must have their values set for "fallback", even if to empty ones, since this is what will be retrieved in the absence of a proper translation, which will be the case if babel or polyglossia is loaded and sets a language which zref-clever does not know. On the other hand, "type names" are not looked for in "fallback", since it is indeed impossible to provide any reasonable value for them for a "specified but unknown language". Also "font" options – those in `\c__zrefclever_ref_options_font_seq`, and queried with `\__zrefclever_get_ref_font:nN` – do not need to be provided here, since the later function sets an empty value if the option is not found.

TODO Add regression test to ensure all fallback “translations” are indeed present.

```
\prop_new:N \g__zrefclever_fallback_dict_prop
\prop_gset_from_keyval:Nn \g__zrefclever_fallback_dict_prop
  {
    tpairsep  = {,~} ,
    tlistsep  = {,~} ,
    tlastsep  = {,~} ,
    notesep   = {~} ,
    namesep   = {\nobreakspace} ,
    pairsep   = {,~} ,
    listsep   = {,~} ,
    lastsep   = {,~} ,
    rangesep  = {\textendash} ,
    refpre    = {} ,
    refpos    = {} ,
    refpre-in = {} ,
    refpos-in = {} ,
  }
```

Get translations

__zrefclever_get_type_transl:nnnNF

Get type-specific translation of ⟨*key*⟩ for ⟨*type*⟩ and ⟨*language*⟩, and store it in ⟨*tl variable*⟩ if found. If not found, leave the ⟨*false code*⟩ on the stream, in which case the value of ⟨*tl variable*⟩ should not be relied upon.

```
\__zrefclever_get_type_transl:nnnNF {⟨language⟩} {⟨type⟩} {⟨key⟩}
  ⟨tl variable⟩ {⟨false code⟩}
```

```
\prg_new_protected_conditional:Npnn
  \__zrefclever_get_type_transl:nnnN #1#2#3#4 { F }
  {
    \prop_get:NnNTF \g__zrefclever_languages_prop {#1}
      \l__zrefclever_dict_language_tl
      {
        \prop_get:cnNTF
          { g__zrefclever_dict_ \l__zrefclever_dict_language_tl _prop }
          { type- #2 - #3 } #4
          { \prg_return_true:  }
          { \prg_return_false: }
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn
  \__zrefclever_get_type_transl:nnnN { xxxN , xxnN } { F }
```

(*End definition for* `\__zrefclever_get_type_transl:nnnNF`.)

__zrefclever_get_default_transl:nnNF

Get default translation of ⟨*key*⟩ for ⟨*language*⟩, and store it in ⟨*tl variable*⟩ if found. If not found, leave the ⟨*false code*⟩ on the stream, in which case the value of ⟨*tl variable*⟩ should not be relied upon.

```
\__zrefclever_get_default_transl:nnNF {⟨language⟩} {⟨key⟩}
  ⟨tl variable⟩ {⟨false code⟩}
```

```
\prg_new_protected_conditional:Npnn
  \__zrefclever_get_default_transl:nnN #1#2#3 { F }
  {
    \prop_get:NnNTF \g__zrefclever_languages_prop {#1}
      \l__zrefclever_dict_language_tl
      {
        \prop_get:cnNTF
          { g__zrefclever_dict_ \l__zrefclever_dict_language_tl _prop }
          { default- #2 } #3
          { \prg_return_true:  }
          { \prg_return_false: }
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn
  \__zrefclever_get_default_transl:nnN { xnN } { F }
```

(*End definition for* `\__zrefclever_get_default_transl:nnNF`.)

`\__zrefclever_get_fallback_transl:nNF` Get fallback translation of ⟨*key*⟩, and store it in ⟨*tl variable*⟩ if found. If not found, leave the ⟨*false code*⟩ on the stream, in which case the value of ⟨*tl variable*⟩ should not be relied upon.

```
\__zrefclever_get_fallback_transl:nNF {⟨key⟩}
  ⟨tl variable⟩ {⟨false code⟩}
```

```
% {<key>}<tl var to set>
\prg_new_protected_conditional:Npnn
  \__zrefclever_get_fallback_transl:nN #1#2 { F }
  {
    \prop_get:NnNTF \g__zrefclever_fallback_dict_prop
      { #1 } #2
      { \prg_return_true:  }
      { \prg_return_false: }
  }
```

(*End definition for* `\__zrefclever_get_fallback_transl:nNF`.)

4.5 Options

Auxiliary

`\__zrefclever_prop_put_non_empty:Nnn` If ⟨*value*⟩ is empty, remove ⟨*key*⟩ from ⟨*property list*⟩. Otherwise, add ⟨*key*⟩ = ⟨*value*⟩ to ⟨*property list*⟩.

```
\__zrefclever_prop_put_non_empty:Nnn ⟨property list⟩ {⟨key⟩} {⟨value⟩}
```

```
\cs_new_protected:Npn \__zrefclever_prop_put_non_empty:Nnn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      { \prop_remove:Nn #1 {#2} }
      { \prop_put:Nnn #1 {#2} {#3} }
  }
```

(*End definition for* `\__zrefclever_prop_put_non_empty:Nnn`.)

ref option

`\l__zrefclever_ref_property_tl` stores the property to which the reference is being made. Currently, we restrict `ref=` to these two (or three) alternatives – `zc@thecnt`, `page`, and `title` if zref-titleref is loaded –, but there might be a case for making this more flexible. The infrastructure can already handle receiving an arbitrary property, as long as one is satisfied with sorting and compressing from the default counter. If more flexibility is granted, one thing *must* be handled at this point: the existence of the property itself, as far as `zref` is concerned. This because typesetting relies on the check `\zref@ifrefcontainsprop`, which *presumes* the property is defined and silently expands the *true* branch if it is not (see https://github.com/ho-tex/zref/issues/13, thanks Ulrike Fischer). Therefore, before adding anything to `\l__zrefclever_ref_property_tl`, check if first here with `\zref@ifpropundefined`: close it at the door.

```
\tl_new:N \l__zrefclever_ref_property_tl
\keys_define:nn { zref-clever / reference }
  {
    ref .choice: ,
    ref / zc@thecnt .code:n =
      { \tl_set:Nn \l__zrefclever_ref_property_tl { zc@thecnt } } ,
    ref / page .code:n =
      { \tl_set:Nn \l__zrefclever_ref_property_tl { page } } ,
    ref / title .code:n =
      {
        \AddToHook { begindocument }
          {
            \@ifpackageloaded { zref-titleref }
              { \tl_set:Nn \l__zrefclever_ref_property_tl { title } }
              {
                \msg_warning:nn { zref-clever } { missing-zref-titleref }
                \tl_set:Nn \l__zrefclever_ref_property_tl { zc@thecnt }
              }
          }
      } ,
    ref .initial:n = zc@thecnt ,
    ref .default:n = zc@thecnt ,
    page .meta:n = { ref = page },
    page .value_forbidden:n = true ,
  }
\AddToHook { begindocument }
  {
    \@ifpackageloaded { zref-titleref }
      {
        \keys_define:nn { zref-clever / reference }
          {
            ref / title .code:n =
              { \tl_set:Nn \l__zrefclever_ref_property_tl { title } }
          }
      }
      {
        \keys_define:nn { zref-clever / reference }
          {
            ref / title .code:n =
              {
                \msg_warning:nn { zref-clever } { missing-zref-titleref }
                \tl_set:Nn \l__zrefclever_ref_property_tl { zc@thecnt }
              }
          }
      }
  }
```

typeset option

```
\bool_new:N \l__zrefclever_typeset_ref_bool
\bool_new:N \l__zrefclever_typeset_name_bool
\keys_define:nn { zref-clever / reference }
  {
    typeset .choice: ,
    typeset / both .code:n =
      {
        \bool_set_true:N \l__zrefclever_typeset_ref_bool
        \bool_set_true:N \l__zrefclever_typeset_name_bool
      } ,
    typeset / ref .code:n =
      {
        \bool_set_true:N \l__zrefclever_typeset_ref_bool
        \bool_set_false:N \l__zrefclever_typeset_name_bool
      } ,
    typeset / name .code:n =
      {
        \bool_set_false:N \l__zrefclever_typeset_ref_bool
        \bool_set_true:N \l__zrefclever_typeset_name_bool
      } ,
    typeset .initial:n = both ,
    typeset .value_required:n = true ,

    noname .meta:n = { typeset = ref },
    noname .value_forbidden:n = true ,
  }
```

sort option

```
\bool_new:N \l__zrefclever_typeset_sort_bool
\keys_define:nn { zref-clever / reference }
  {
    sort .bool_set:N = \l__zrefclever_typeset_sort_bool ,
    sort .initial:n = true ,
    sort .default:n = true ,
    nosort .meta:n = { sort = false },
    nosort .value_forbidden:n = true ,
  }
```

typesort option

`\l__zrefclever_typesort_seq` is stored reversed, since the sort priorities are computed in the negative range in `\__zrefclever_sort_default_different_types:nn`, so that we can implicitly rely on '0' being the "last value", and spare creating an integer variable using `\seq_map_indexed_inline:Nn`.

```
\seq_new:N \l__zrefclever_typesort_seq
```

```
\keys_define:nn { zref-clever / reference }
  {
    typesort .code:n =
      {
        \seq_set_from_clist:Nn \l__zrefclever_typesort_seq {#1}
        \seq_reverse:N \l__zrefclever_typesort_seq
      } ,
    typesort .initial:n =
      { part , chapter , section , paragraph },
    typesort .value_required:n = true ,
    notypesort .code:n =
      { \seq_clear:N \l__zrefclever_typesort_seq } ,
    notypesort .value_forbidden:n = true ,
  }
```

comp option

```
\bool_new:N \l__zrefclever_typeset_compress_bool
\keys_define:nn { zref-clever / reference }
  {
    comp .bool_set:N = \l__zrefclever_typeset_compress_bool ,
    comp .initial:n = true ,
    comp .default:n = true ,
    nocomp .meta:n = { comp = false },
    nocomp .value_forbidden:n = true ,
  }
```

range option

```
\bool_new:N \l__zrefclever_typeset_range_bool
\keys_define:nn { zref-clever / reference }
  {
    range .bool_set:N = \l__zrefclever_typeset_range_bool ,
    range .initial:n = false ,
    range .default:n = true ,
  }
```

cap and capfirst options

```
\bool_new:N \l__zrefclever_capitalize_bool
\bool_new:N \l__zrefclever_capitalize_first_bool
\keys_define:nn { zref-clever / reference }
  {
    cap .bool_set:N = \l__zrefclever_capitalize_bool ,
    cap .initial:n = false ,
    cap .default:n = true ,
    nocap .meta:n = { cap = false },
    nocap .value_forbidden:n = true ,

    capfirst .bool_set:N = \l__zrefclever_capitalize_first_bool ,
    capfirst .initial:n = false ,
    capfirst .default:n = true ,
  }
```

abbrev and noabbrevfirst options

```
\bool_new:N \l__zrefclever_abbrev_bool
```

```
\bool_new:N \l__zrefclever_noabbrev_first_bool
\keys_define:nn { zref-clever / reference }
  {
    abbrev .bool_set:N = \l__zrefclever_abbrev_bool ,
    abbrev .initial:n = false ,
    abbrev .default:n = true ,
    noabbrev .meta:n = { abbrev = false },
    noabbrev .value_forbidden:n = true ,

    noabbrevfirst .bool_set:N = \l__zrefclever_noabbrev_first_bool ,
    noabbrevfirst .initial:n = false ,
    noabbrevfirst .default:n = true ,
  }
```

S option

```
\keys_define:nn { zref-clever / reference }
  {
    S .meta:n =
      { capfirst = true , noabbrevfirst = true },
    S .value_forbidden:n = true ,
  }
```

hyperref option

```
\bool_new:N \l__zrefclever_use_hyperref_bool
\bool_new:N \l__zrefclever_warn_hyperref_bool
\keys_define:nn { zref-clever / reference }
  {
    hyperref .choice: ,
    hyperref / auto .code:n =
      {
        \bool_set_true:N \l__zrefclever_use_hyperref_bool
        \bool_set_false:N \l__zrefclever_warn_hyperref_bool
      } ,
    hyperref / true .code:n =
      {
        \bool_set_true:N \l__zrefclever_use_hyperref_bool
        \bool_set_true:N \l__zrefclever_warn_hyperref_bool
      } ,
    hyperref / false .code:n =
      {
        \bool_set_false:N \l__zrefclever_use_hyperref_bool
        \bool_set_false:N \l__zrefclever_warn_hyperref_bool
      } ,
    hyperref .initial:n = auto ,
    hyperref .default:n = auto
  }
\AddToHook { begindocument }
  {
    \@ifpackageloaded { hyperref }
      {
        \bool_if:NT \l__zrefclever_use_hyperref_bool
          { \RequirePackage { zref-hyperref } }
      }
      {
        \bool_if:NT \l__zrefclever_warn_hyperref_bool
          { \msg_warning:nn { zref-clever } { missing-hyperref } }
        \bool_set_false:N \l__zrefclever_use_hyperref_bool
      }
    \keys_define:nn { zref-clever / reference }
      {
        hyperref .code:n =
          { \msg_warning:nn { zref-clever } { hyperref-preamble-only } }
      }
  }
```

nameinlink option

```
\str_new:N \l__zrefclever_nameinlink_str
\keys_define:nn { zref-clever / reference }
  {
    nameinlink .choice: ,
    nameinlink / true .code:n =
      { \str_set:Nn \l__zrefclever_nameinlink_str { true } } ,
    nameinlink / false .code:n =
      { \str_set:Nn \l__zrefclever_nameinlink_str { false } } ,
    nameinlink / single .code:n =
      { \str_set:Nn \l__zrefclever_nameinlink_str { single } } ,
    nameinlink / tsingle .code:n =
      { \str_set:Nn \l__zrefclever_nameinlink_str { tsingle } } ,
    nameinlink .initial:n = tsingle ,
    nameinlink .default:n = true ,
  }
```

lang option

\l__zrefclever_current_language_tl is an internal alias for babel's \languagename or polyglossia's \mainbabelname and, if none of them is loaded, we set it to english. \l__zrefclever_main_language_tl is an internal alias for babel's \bbl@main@language or for polyglossia's \mainbabelname, as the case may be. Note that for polyglossia we get babel's language names, so that we only need to handle those internally. \l__zrefclever_ref_language_tl is the internal variable which stores the language in which the reference is to be made.

The overall setup here seems a little roundabout, but this is actually required. In the preamble, we (potentially) don't yet have values for the "main" and "current" document languages, this must be retrieved at a begindocument hook. The begindocument hook is responsible to get values for \l__zrefclever_main_language_tl and \l__zrefclever_current_language_tl, and to set the default for \l__zrefclever_ref_language_tl. Package options, or preamble calls to \zcsetup are also hooked at begindocument, but come after the first hook, so that the pertinent variables have been set when they are executed. Finally, we set a third begindocument hook, at begindocument/before, so that it runs after any options set in the preamble. This hook redefines the lang option for immediate execution in the document body, and ensures the main language's dictionary gets loaded, if it hadn't been already.

For the babel and polyglossia variables which store the "main" and "current" languages, see https://tex.stackexchange.com/a/233178, including comments, particularly the one by Javier Bezos. For the babel and polyglossia variables which store the list of loaded languages, see https://tex.stackexchange.com/a/281220, including comments, particularly PLK's. Note, however, that languages loaded by \babelprovide, either directly, “on the fly”, or with the `provide` option, `do not` get included in `\bbl@loaded`.

```
\tl_new:N \l__zrefclever_ref_language_tl
\tl_new:N \l__zrefclever_main_language_tl
\tl_new:N \l__zrefclever_current_language_tl
\AddToHook { begindocument }
  {
    \@ifpackageloaded { babel }
      {
        \tl_set:Nn \l__zrefclever_current_language_tl { \languagename }
        \tl_set:Nn \l__zrefclever_main_language_tl { \bbl@main@language }
      }
      {
        \@ifpackageloaded { polyglossia }
          {
            \tl_set:Nn \l__zrefclever_current_language_tl { \babelname }
            \tl_set:Nn \l__zrefclever_main_language_tl { \mainbabelname }
          }
          {
            \tl_set:Nn \l__zrefclever_current_language_tl { english }
            \tl_set:Nn \l__zrefclever_main_language_tl { english }
          }
      }
```

Provide default value for `\l__zrefclever_ref_language_tl` corresponding to option `main`, but do so outside of the l3keys machinery (that is, instead of using `.initial:n`), so that we are able to distinguish when the user actually gave the option, in which case the dictionary loading is done verbosely, from when we are setting the default value (here), in which case the dictionary loading is done silently.

```
    \tl_set:Nn \l__zrefclever_ref_language_tl
      { \l__zrefclever_main_language_tl }
  }
```

```
\keys_define:nn { zref-clever / reference }
  {
    lang .code:n =
      {
        \AddToHook { begindocument }
          {
            \str_case:nnF {#1}
              {
                { main }
                {
                  \tl_set:Nn \l__zrefclever_ref_language_tl
                    { \l__zrefclever_main_language_tl }
                  \__zrefclever_provide_dictionary_verbose:x
                    { \l__zrefclever_ref_language_tl }
                }

                { current }
                {
                  \tl_set:Nn \l__zrefclever_ref_language_tl
                    { \l__zrefclever_current_language_tl }
                  \__zrefclever_provide_dictionary_verbose:x
```

```
                    { \l__zrefclever_ref_language_tl }
                }
              }
              {
                \prop_if_in:NnTF \g__zrefclever_languages_prop {#1}
                  {
                    \tl_set:Nn \l__zrefclever_ref_language_tl {#1}
                  }
                  {
                    \msg_warning:nnn { zref-clever }
                      { unknown-language-opt } {#1}
                    \tl_set:Nn \l__zrefclever_ref_language_tl
                      { \l__zrefclever_main_language_tl }
                  }
                \__zrefclever_provide_dictionary_verbose:x
                  { \l__zrefclever_ref_language_tl }
              }
          }
      } ,
    lang .value_required:n = true ,
  }
```

```
\AddToHook { begindocument / before }
  {
    \AddToHook { begindocument }
      {
```

If any `lang` option has been given by the user, the corresponding language is already loaded, otherwise, ensure the default one (`main`) gets loaded early, but not verbosely.

```
        \__zrefclever_provide_dictionary:x { \l__zrefclever_ref_language_tl }
```

Redefinition of the `lang` key option for the document body. Also, drop the verbose dictionary loading in the document body, as it can become intrusive depending on the use case, and does not provide much "juice" anyway: in `\zcref` missing names warnings will already ensue.

```
        \keys_define:nn { zref-clever / reference }
          {
            lang .code:n =
              {
                \str_case:nnF {#1}
                  {
                    { main }
                    {
                      \tl_set:Nn \l__zrefclever_ref_language_tl
                        { \l__zrefclever_main_language_tl }
                      \__zrefclever_provide_dictionary:x
                        { \l__zrefclever_ref_language_tl }
                    }

                    { current }
                    {
                      \tl_set:Nn \l__zrefclever_ref_language_tl
                        { \l__zrefclever_current_language_tl }
                      \__zrefclever_provide_dictionary:x
                        { \l__zrefclever_ref_language_tl }
                    }
                  }
                  {
                    \prop_if_in:NnTF \g__zrefclever_languages_prop {#1}
                      {
                        \tl_set:Nn \l__zrefclever_ref_language_tl {#1}
                      }
                      {
                        \msg_warning:nnn { zref-clever }
                          { unknown-language-opt } {#1}
                        \tl_set:Nn \l__zrefclever_ref_language_tl
                          { \l__zrefclever_main_language_tl }
                      }
                    \__zrefclever_provide_dictionary:x
                      { \l__zrefclever_ref_language_tl }
                  }
              } ,
            lang .value_required:n = true ,
          }
      }
  }
```

font option

`font` *can't be used as a package option*, since the options get expanded by LaTeX before being passed to the package (see https://tex.stackexchange.com/a/489570). It can't be set in `\zcref` and, for global settings, with `\zcsetup`.

```
\tl_new:N \l__zrefclever_ref_typeset_font_tl
\keys_define:nn { zref-clever / reference }
  { font .tl_set:N = \l__zrefclever_ref_typeset_font_tl }
```

note option

```
\tl_new:N \l__zrefclever_zcref_note_tl
\keys_define:nn { zref-clever / reference }
  {
    note .tl_set:N = \l__zrefclever_zcref_note_tl ,
    note .value_required:n = true ,
  }
```

check option

Integration with zref-check.

```
\bool_new:N \l__zrefclever_zrefcheck_available_bool
\bool_new:N \l__zrefclever_zcref_with_check_bool
\keys_define:nn { zref-clever / reference }
  {
    check .code:n =
      { \msg_warning:nn { zref-clever } { check-document-only } } ,
  }
\AddToHook { begindocument }
  {
    \@ifpackageloaded { zref-check }
      {
        \bool_set_true:N \l__zrefclever_zrefcheck_available_bool
        \keys_define:nn { zref-clever / reference }
          {
            check .code:n =
              {
                \bool_set_true:N \l__zrefclever_zcref_with_check_bool
                \keys_set:nn { zref-check / zcheck } {#1}
              }
          }
      }
      {
        \bool_set_false:N \l__zrefclever_zrefcheck_available_bool
        \keys_define:nn { zref-clever / reference }
          {
            check .code:n =
              { \msg_warning:nn { zref-clever } { missing-zref-check } }
          }
      }
  }
```

countertype option

`\l__zrefclever_counter_type_prop` is used by `zc@type` property, and stores a mapping from "counter" to "reference type". Only those counters whose type name is different from that of the counter need to be specified, since `zc@type` presumes the counter as the type if the counter is not found in `\l__zrefclever_counter_type_prop`.

```
\prop_new:N \l__zrefclever_counter_type_prop
\keys_define:nn { zref-clever / label }
  {
    countertype .code:n =
      {
        \keyval_parse:nnn
          {
            \msg_warning:nnnn { zref-clever }
              { key-requires-value } { countertype }
          }
          {
            \__zrefclever_prop_put_non_empty:Nnn
              \l__zrefclever_counter_type_prop
          }
          {#1}
      } ,
    countertype .value_required:n = true ,
    countertype .initial:n =
      {
        subsection    = section ,
        subsubsection = section ,
        subparagraph  = paragraph ,
        enumi         = item ,
        enumii        = item ,
        enumiii       = item ,
        enumiv        = item ,
      } ,
  }
```

counterresetters option

\l__zrefclever_counter_resetters_seq is used by __zrefclever_counter_reset_-by:n to populate the zc@enclcnt and zc@enclval properties, and stores the list of counters which are potential "enclosing counters" for other counters. This option is constructed such that users can only *add* items to the variable. There would be little gain and some risk in allowing removal, and the syntax of the option would become unnecessarily more complicated. Besides, users can already override, for any particular counter, the search done from the set in \l__zrefclever_counter_resetters_seq with the counterresetby option.

```
\seq_new:N \l__zrefclever_counter_resetters_seq
\keys_define:nn { zref-clever / label }
  {
    counterresetters .code:n =
      {
        \clist_map_inline:nn {#1}
          {
            \seq_if_in:NnF \l__zrefclever_counter_resetters_seq {##1}
              {
                \seq_put_right:Nn
                  \l__zrefclever_counter_resetters_seq {##1}
              }
          }
      } ,
    counterresetters .initial:n =
      {
        part ,
        chapter ,
        section ,
        subsection ,
        subsubsection ,
        paragraph ,
        subparagraph ,
      },
    counterresetters .value_required:n = true ,
  }
```

counterresetby option

\l__zrefclever_counter_resetby_prop is used by __zrefclever_counter_reset_-by:n to populate the zc@enclcnt and zc@enclval properties, and stores a mapping from counters to the counter which resets each of them. This mapping has precedence in __zrefclever_counter_reset_by:n over the search through \l__zrefclever_-counter_resetters_seq.

```
\prop_new:N \l__zrefclever_counter_resetby_prop
\keys_define:nn { zref-clever / label }
  {
    counterresetby .code:n =
      {
        \keyval_parse:nnn
```

```
          {
            \msg_warning:nnn { zref-clever }
              { key-requires-value } { counterresetby }
          }
          {
            \__zrefclever_prop_put_non_empty:Nnn
              \l__zrefclever_counter_resetby_prop
          }
          {#1}
      } ,
    counterresetby .value_required:n = true ,
    counterresetby .initial:n =
      {
```

The counters for the `enumerate` environment do not use the regular counter machinery for resetting on each level, but are nested nevertheless by other means, treat them as exception.

```
        enumii  = enumi   ,
        enumiii = enumii  ,
        enumiv  = enumiii ,
      } ,
  }
```

`currentcounter` option

`\l__zrefclever_current_counter_tl` is pretty much the starting point of all of the data specification for label setting done by `zref` with our setup for it. It exists because we must provide some "handle" to specify the current counter for packages/features that do not set `\@currentcounter` appropriately.

```
\tl_new:N \l__zrefclever_current_counter_tl
\keys_define:nn { zref-clever / label }
  {
    currentcounter .tl_set:N = \l__zrefclever_current_counter_tl ,
    currentcounter .value_required:n = true ,
    currentcounter .initial:n = \@currentcounter ,
  }
```

Reference options

This is a set of options related to reference typesetting which receive equal treatment and, hence, are handled in batch. Since we are dealing with options to be passed to `\zcref` or to `\zcsetup` or at load time, only "not necessarily type-specific" options are pertinent here. However, they *may* either be type-specific or language-specific, and thus must be stored in a property list, `\l__zrefclever_ref_options_prop`, in order to be retrieved from the option *name* by `\__zrefclever_get_ref_string:nN` and `\__zrefclever_get_ref_font:nN` according to context and precedence rules.

The keys are set so that any value, including an empty one, is added to `\l__zrefclever_ref_options_prop`, while a key with *no value* removes the property from the list, so that these options can then fall back to lower precedence levels settings. For discussion about the used technique, see Section 5.2.

```
\prop_new:N \l__zrefclever_ref_options_prop
\seq_map_inline:Nn
```

```
  \c__zrefclever_ref_options_reference_seq
  {
    \keys_define:nn { zref-clever / reference }
      {
        #1 .default:V = \c_novalue_tl ,
        #1 .code:n =
          {
            \tl_if_novalue:nTF {##1}
              { \prop_remove:Nn \l__zrefclever_ref_options_prop {#1} }
              { \prop_put:Nnn \l__zrefclever_ref_options_prop {#1} {##1} }
          } ,
      }
  }
```

Package options

The options have been separated in two different groups, so that we can potentially apply them selectively to different contexts: `label` and `reference`. Currently, the only use of this selection is the ability to exclude label related options from `\zcref`'s options. Anyway, for load-time package options and for `\zcsetup` we want the whole set, so we aggregate the two into `zref-clever/zcsetup`, and use that here.

```
\keys_define:nn { }
  {
    zref-clever / zcsetup .inherit:n =
      {
        zref-clever / label ,
        zref-clever / reference ,
      }
  }
```

Process load-time package options (https://tex.stackexchange.com/a/15840).

```
\ProcessKeysOptions { zref-clever / zcsetup }
```

5 Configuration

5.1 `\zcsetup`

`\zcsetup` Provide `\zcsetup`.

`\zcsetup{⟨options⟩}`

```
\NewDocumentCommand \zcsetup { m }
  { \keys_set:nn { zref-clever / zcsetup } {#1} }
```

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

5.2 `\zcRefTypeSetup`

`\zcRefTypeSetup` is the main user interface for "type-specific" reference formatting. Settings done by this command have a higher precedence than any translation, hence they override any language-specific setting, either done at `\zcLanguageSetup` or by the package's dictionaries. On the other hand, they have a lower precedence than non type-specific general options. The ⟨*options*⟩ should be given in the usual `key=val` format. The ⟨*type*⟩

does not need to pre-exist, the property list variable to store the properties for the type gets created if need be.

\zcRefTypeSetup   `\zcRefTypeSetup {⟨type⟩} {⟨options⟩}`

```
\NewDocumentCommand \zcRefTypeSetup { m m }
  {
    \prop_if_exist:cF { l__zrefclever_type_ #1 _options_prop }
      { \prop_new:c { l__zrefclever_type_ #1 _options_prop } }
    \tl_set:Nn \l__zrefclever_setup_type_tl {#1}
    \keys_set:nn { zref-clever / typesetup } {#2}
  }
```

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

Inside `\zcRefTypeSetup` any of the options *can* receive empty values, and those values, if they exist in the property list, will override translations, regardless of their emptiness. In principle, we could live with the situation of, once a setting has been made in `\l__zrefclever_type_<type>_options_prop` or in `\l__zrefclever_ref_options_prop` it stays there forever, and can only be overridden by a new value at the same precedence level or a higher one. But it would be nice if an user can "unset" an option at either of those scopes to go back to the lower precedence level of the translations at any given point. So both in `\zcRefTypeSetup` and in setting reference options (see Section 4.5), we leverage the distinction of an "empty valued key" (`key=` or `key={}`) from a "key with no value" (`key`). This distinction is captured internally by the lower-level key parsing, but must be made explicit at `\keys_set:nn` by means of the `.default:V` property of the key in `\keys_define:nn`. For the technique and some discussion about it, see https://tex.stackexchange.com/q/614690 (thanks Jonathan P. Spratte, aka 'Skillmon', and Phelype Oleinik) and https://github.com/latex3/latex3/pull/988.

```
\seq_map_inline:Nn
  \c__zrefclever_ref_options_necessarily_not_type_specific_seq
  {
    \keys_define:nn { zref-clever / typesetup }
      {
        #1 .code:n =
          {
            \msg_warning:nnn { zref-clever }
              { option-not-type-specific } {#1}
          } ,
      }
  }
```

```
\seq_map_inline:Nn
  \c__zrefclever_ref_options_typesetup_seq
  {
    \keys_define:nn { zref-clever / typesetup }
      {
        #1 .default:V = \c_novalue_tl ,
        #1 .code:n =
          {
            \tl_if_novalue:nTF {##1}
              {
                \prop_remove:cn
                  {
                    l__zrefclever_type_
                    \l__zrefclever_setup_type_tl _options_prop
                  }
                  {#1}
              }
              {
                \prop_put:cnn
                  {
                    l__zrefclever_type_
                    \l__zrefclever_setup_type_tl _options_prop
                  }
                  {#1} {##1}
              }
          } ,
      }
  }
```

5.3 \zcLanguageSetup

`\zcLanguageSetup` is the main user interface for "language-specific" reference formatting, be it "type-specific" or not. The difference between the two cases is captured by the `type` key, which works as a sort of a "switch". Inside the ⟨*options*⟩ argument of `\zcLanguageSetup`, any options made before the first `type` key declare "default" (non type-specific) translations. When the `type` key is given with a value, the options following it will set "type-specific" translations for that type. The current type can be switched off by an empty `type` key. `\zcLanguageSetup` is preamble only.

\zcLanguageSetup

```
\zcLanguageSetup{⟨language⟩}{⟨options⟩}
```

```
\NewDocumentCommand \zcLanguageSetup { m m }
  {
    \group_begin:
    \prop_get:NnNTF \g__zrefclever_languages_prop {#1}
      \l__zrefclever_dict_language_tl
      {
        \tl_clear:N \l__zrefclever_setup_type_tl
        \keys_set:nn { zref-clever / langsetup } {#2}
      }
      { \msg_warning:nnn { zref-clever } { unknown-language-transl } {#1} }
    \group_end:
  }
\@onlypreamble \zcLanguageSetup
```

(*End definition for \zcLanguageSetup.*)

__zrefclever_declare_type_transl:nnnn
__zrefclever_declare_default_transl:nnn

A couple of auxiliary functions for the of `zref-clever/translation` keys set in `\zcLanguageSetup`. They respectively declare (unconditionally set) "type-specific" and "default" translations.

```
\__zrefclever_declare_type_transl:nnnn {⟨language⟩} {⟨type⟩}
  {⟨key⟩} {⟨translation⟩}
\__zrefclever_declare_default_transl:nnn {⟨language⟩}
  {⟨key⟩} {⟨translation⟩}
```

```
\cs_new_protected:Npn \__zrefclever_declare_type_transl:nnnn #1#2#3#4
  {
    \prop_gput:cnn { g__zrefclever_dict_ #1 _prop }
      { type- #2 - #3 } {#4}
  }
\cs_generate_variant:Nn \__zrefclever_declare_type_transl:nnnn { VVnn }
\cs_new_protected:Npn \__zrefclever_declare_default_transl:nnn #1#2#3
  {
    \prop_gput:cnn { g__zrefclever_dict_ #1 _prop }
      { default- #2 } {#3}
  }
\cs_generate_variant:Nn \__zrefclever_declare_default_transl:nnn { Vnn }
```

(*End definition for* `\__zrefclever_declare_type_transl:nnnn` *and* `\__zrefclever_declare_default_transl:nnn`*.*)

The set of keys for `zref-clever/langsetup`, which is used to set language-specific translations in `\zcLanguageSetup`.

```
\keys_define:nn { zref-clever / langsetup }
  {
    type .code:n =
      {
        \tl_if_empty:nTF {#1}
          { \tl_clear:N \l__zrefclever_setup_type_tl }
          { \tl_set:Nn \l__zrefclever_setup_type_tl {#1} }
      } ,
  }
\seq_map_inline:Nn
  \c__zrefclever_ref_options_necessarily_not_type_specific_seq
  {
    \keys_define:nn { zref-clever / langsetup }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          {
            \tl_if_empty:NTF \l__zrefclever_setup_type_tl
              {
                \__zrefclever_declare_default_transl:Vnn
                  \l__zrefclever_dict_language_tl
                  {#1} {##1}
              }
              {
                \msg_warning:nnn { zref-clever }
                  { option-not-type-specific } {#1}
              }
          } ,
      }
  }
\seq_map_inline:Nn
  \c__zrefclever_ref_options_possibly_type_specific_seq
  {
    \keys_define:nn { zref-clever / langsetup }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          {
            \tl_if_empty:NTF \l__zrefclever_setup_type_tl
              {
                \__zrefclever_declare_default_transl:Vnn
                  \l__zrefclever_dict_language_tl
                  {#1} {##1}
              }
              {
                \__zrefclever_declare_type_transl:VVnn
                  \l__zrefclever_dict_language_tl
                  \l__zrefclever_setup_type_tl
                  {#1} {##1}
              }
          } ,
      }
  }
\seq_map_inline:Nn
  \c__zrefclever_ref_options_necessarily_type_specific_seq
  {
    \keys_define:nn { zref-clever / langsetup }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          {
            \tl_if_empty:NTF \l__zrefclever_setup_type_tl
              {
                \msg_warning:nnn { zref-clever }
                  { option-only-type-specific } {#1}
              }
              {
                \__zrefclever_declare_type_transl:VVnn
                  \l__zrefclever_dict_language_tl
                  \l__zrefclever_setup_type_tl
                  {#1} {##1}
              }
          } ,
      }
  }
```

6 User interface

6.1 \zcref

\zcref The main user command of the package.

`\zcref⟨*⟩[⟨options⟩]{⟨labels⟩}`

```
\NewDocumentCommand \zcref { s O { } m }
  { \zref@wrapper@babel \__zrefclever_zcref:nnn {#3} {#1} {#2} }
```

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

\__zrefclever_zcref:nnnn An intermediate internal function, which does the actual heavy lifting, and places `{`⟨*labels*⟩`}` as first argument, so that it can be protected by `\zref@wrapper@babel` in `\zcref`.

```
\__zrefclever_zcref:nnnn {⟨labels⟩} {⟨*⟩} {⟨options⟩}
```

```
\cs_new_protected:Npn \__zrefclever_zcref:nnn #1#2#3
  {
    \group_begin:
```

Set options.

```
      \keys_set:nn { zref-clever / reference } {#3}
```

Store arguments values.

```
      \seq_set_from_clist:Nn \l__zrefclever_zcref_labels_seq {#1}
      \bool_set:Nn \l__zrefclever_link_star_bool {#2}
```

Ensure dictionary for reference language is loaded, if available. We cannot rely on `\keys_set:nn` for the task, since if the `lang` option is set for `current`, the actual language may have changed outside our control. `\__zrefclever_provide_dictionary:x` does nothing if the dictionary is already loaded.

```
      \__zrefclever_provide_dictionary:x { \l__zrefclever_ref_language_tl }
```

Integration with zref-check.

```
      \bool_lazy_and:nnT
        { \l__zrefclever_zrefcheck_available_bool }
        { \l__zrefclever_zcref_with_check_bool }
        { \zrefcheck_zcref_beg_label: }
```

Sort the labels.

```
      \bool_lazy_or:nnT
        { \l__zrefclever_typeset_sort_bool }
        { \l__zrefclever_typeset_range_bool }
        { \__zrefclever_sort_labels: }
```

Typeset the references. Also, set the reference font, and group it, so that it does not leak to the note.

```
      \group_begin:
      \l__zrefclever_ref_typeset_font_tl
      \__zrefclever_typeset_refs:
      \group_end:
```

Typeset `note`.

```
      \tl_if_empty:NF \l__zrefclever_zcref_note_tl
        {
          \__zrefclever_get_ref_string:nN { notesep } \l_tmpa_tl
          \l_tmpa_tl
          \l__zrefclever_zcref_note_tl
        }
```

Integration with zref-check.

```
      \bool_lazy_and:nnT
        { \l__zrefclever_zrefcheck_available_bool }
        { \l__zrefclever_zcref_with_check_bool }
        {
          \zrefcheck_zcref_end_label_maybe:
          \zrefcheck_zcref_run_checks_on_labels:n
            { \l__zrefclever_zcref_labels_seq }
        }
    \group_end:
  }
```

(*End definition for* `\__zrefclever_zcref:nnnn`*.*)

`\l__zrefclever_zcref_labels_seq`
`\l__zrefclever_link_star_bool`

```
\seq_new:N \l__zrefclever_zcref_labels_seq
\bool_new:N \l__zrefclever_link_star_bool
```

(*End definition for* `\l__zrefclever_zcref_labels_seq` *and* `\l__zrefclever_link_star_bool`*.*)

6.2 `\zcpageref`

`\zcpageref` A `\pageref` equivalent of `\zcref`.

```
\zcpageref⟨*⟩[⟨options⟩]{⟨labels⟩}
```

```
\NewDocumentCommand \zcpageref { s O { } m }
  {
    \IfBooleanTF {#1}
      { \zcref*[#2, ref = page] {#3} }
      { \zcref [#2, ref = page] {#3} }
  }
```

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

7 Sorting

Sorting is certainly a "big task" for zref-clever but, in the end, it boils down to "carefully done branching", and quite some of it. The sorting of "page" references is very much lightened by the availability of `abspage`, from the zref-abspage module, which offers "just what we need" for our purposes. The sorting of "default" references falls on two main cases: i) labels of the same type; ii) labels of different types. The first case is sorted according to the priorities set by the `typesort` option or, if that is silent for the case, by the order in which labels were given by the user in `\zcref`. The second case is the most involved one, since it is possible for multiple counters to be bundled together in a single reference type. Because of this, sorting must take into account the whole chain of "enclosing counters" for the counters of the labels at hand.

`\l__zrefclever_label_type_a_tl`
`\l__zrefclever_label_type_b_tl`
`\l__zrefclever_label_enclcnt_a_tl`
`\l__zrefclever_label_enclcnt_b_tl`
`\l__zrefclever_label_enclval_a_tl`
`\l__zrefclever_label_enclval_b_tl`
`\l__zrefclever_label_extdoc_a_tl`
`\l__zrefclever_label_extdoc_b_tl`

Auxiliary variables, for use in sorting, and some also in typesetting. Used to store reference information – label properties – of the "current" (`a`) and "next" (`b`) labels.

```
\tl_new:N \l__zrefclever_label_type_a_tl
\tl_new:N \l__zrefclever_label_type_b_tl
\tl_new:N \l__zrefclever_label_enclcnt_a_tl
\tl_new:N \l__zrefclever_label_enclcnt_b_tl
\tl_new:N \l__zrefclever_label_enclval_a_tl
\tl_new:N \l__zrefclever_label_enclval_b_tl
\tl_new:N \l__zrefclever_label_extdoc_a_tl
\tl_new:N \l__zrefclever_label_extdoc_b_tl
```

(*End definition for* `\l__zrefclever_label_type_a_tl` *and others.*)

`\l__zrefclever_sort_decided_bool` Auxiliary variable for `\__zrefclever_sort_default_same_type:nn`, signals if the sorting between two labels has been decided or not.

```
\bool_new:N \l__zrefclever_sort_decided_bool
```

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

`\l__zrefclever_sort_prior_a_int`
`\l__zrefclever_sort_prior_b_int`

Auxiliary variables for `\__zrefclever_sort_default_different_types:nn`. Store the sort priority of the "current" and "next" labels.

```
\int_new:N \l__zrefclever_sort_prior_a_int
\int_new:N \l__zrefclever_sort_prior_b_int
```

(*End definition for* `\l__zrefclever_sort_prior_a_int` *and* `\l__zrefclever_sort_prior_b_int`.)

`\l__zrefclever_label_types_seq`

Stores the order in which reference types appear in the label list supplied by the user in `\zcref`. This variable is populated by `\__zrefclever_label_type_put_new_right:n` at the start of `\__zrefclever_sort_labels:`. This order is required as a "last resort" sort criterion between the reference types, for use in `\__zrefclever_sort_default_different_types:nn`.

```
\seq_new:N \l__zrefclever_label_types_seq
```

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

`\__zrefclever_sort_labels:`

The main sorting function. It does not receive arguments, but it is expected to be run inside `\__zrefclever_zcref:nnnn` where a number of environment variables are to be set appropriately. In particular, `\l__zrefclever_zcref_labels_seq` should contain the labels received as argument to `\zcref`, and the function performs its task by sorting this variable.

```
\cs_new_protected:Npn \__zrefclever_sort_labels:
  {
```

Store label types sequence.

```
    \seq_clear:N \l__zrefclever_label_types_seq
    \tl_if_eq:NnF \l__zrefclever_ref_property_tl { page }
      {
        \seq_map_function:NN \l__zrefclever_zcref_labels_seq
          \__zrefclever_label_type_put_new_right:n
      }
```

Sort.

```
    \seq_sort:Nn \l__zrefclever_zcref_labels_seq
      {
        \zref@ifrefundefined {##1}
          {
            \zref@ifrefundefined {##2}
              {
                % Neither label is defined.
                \sort_return_same:
              }
              {
                % The second label is defined, but the first isn't, leave the
                % undefined first (to be more visible).
                \sort_return_same:
              }
          }
          {
            \zref@ifrefundefined {##2}
              {
                % The first label is defined, but the second isn't, bring the
```

```
                  % second forward.
                  \sort_return_swapped:
                }
                {
                  % The interesting case: both labels are defined.  References
                  % to the "default" property or to the "page" are quite
                  % different with regard to sorting, so we branch them here to
                  % specialized functions.
                  \tl_if_eq:NnTF \l__zrefclever_ref_property_tl { page }
                    { \__zrefclever_sort_page:nn {##1} {##2} }
                    { \__zrefclever_sort_default:nn {##1} {##2} }
                }
            }
        }
    }
```

(*End definition for* `\__zrefclever_sort_labels:`.)

`\__zrefclever_label_type_put_new_right:n`

Auxiliary function used to store the order in which reference types appear in the label list supplied by the user in `\zcref`. It is expected to be run inside `\__zrefclever_sort_labels:`, and stores the types sequence in `\l__zrefclever_label_types_seq`. I have tried to handle the same task inside `\seq_sort:Nn` in `\__zrefclever_sort_labels:` to spare mapping over `\l__zrefclever_zcref_labels_seq`, but it turned out it not to be easy to rely on the order the labels get processed at that point, since the variable is being sorted there. Besides, the mapping is simple, not a particularly expensive operation. Anyway, this keeps things clean.

`\__zrefclever_label_type_put_new_right:n {⟨label⟩}`

```
\cs_new_protected:Npn \__zrefclever_label_type_put_new_right:n #1
  {
    \tl_set:Nx \l__zrefclever_label_type_a_tl
      { \zref@extractdefault {#1} { zc@type } { \c_empty_tl } }
    \seq_if_in:NVF \l__zrefclever_label_types_seq
      \l__zrefclever_label_type_a_tl
      {
        \seq_put_right:NV \l__zrefclever_label_types_seq
          \l__zrefclever_label_type_a_tl
      }
  }
```

(*End definition for* `\__zrefclever_label_type_put_new_right:n`.)

`\__zrefclever_sort_default:nn`

The heavy-lifting function for sorting of defined labels for "default" references (that is, a standard reference, not to "page"). This function is expected to be called within the sorting loop of `\__zrefclever_sort_labels:` and receives the pair of labels being considered for a change of order or not. It should *always* "return" either `\sort_return_same:` or `\sort_return_swapped:`.

`\__zrefclever_sort_default:nn {⟨label a⟩} {⟨label b⟩}`

```
\cs_new_protected:Npn \__zrefclever_sort_default:nn #1#2
  {
    \tl_set:Nx \l__zrefclever_label_type_a_tl
      { \zref@extractdefault {#1} { zc@type } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_type_b_tl
      { \zref@extractdefault {#2} { zc@type } { \c_empty_tl } }

    \bool_if:nTF
      {
        % The second label has a type, but the first doesn't, leave the
        % undefined first (to be more visible).
        \tl_if_empty_p:N \l__zrefclever_label_type_a_tl &&
        ! \tl_if_empty_p:N \l__zrefclever_label_type_b_tl
      }
      { \sort_return_same: }
      {
        \bool_if:nTF
          {
            % The first label has a type, but the second doesn't, bring the
            % second forward.
            ! \tl_if_empty_p:N \l__zrefclever_label_type_a_tl &&
            \tl_if_empty_p:N \l__zrefclever_label_type_b_tl
          }
          { \sort_return_swapped: }
          {
            \bool_if:nTF
              {
                % The interesting case: both labels have a type...
                ! \tl_if_empty_p:N \l__zrefclever_label_type_a_tl &&
                ! \tl_if_empty_p:N \l__zrefclever_label_type_b_tl
              }
              {
                \tl_if_eq:NNTF
                  \l__zrefclever_label_type_a_tl
                  \l__zrefclever_label_type_b_tl
                  % ...and it's the same type.
                  { \__zrefclever_sort_default_same_type:nn {#1} {#2} }
                  % ...and they are different types.
                  { \__zrefclever_sort_default_different_types:nn {#1} {#2} }
              }
              {
                % Neither label has a type.  We can't do much of meaningful
                % here, but if it's the same counter, compare it.
                \exp_args:Nxx \tl_if_eq:nnTF
                  { \zref@extractdefault {#1} { zc@counter } { } }
                  { \zref@extractdefault {#2} { zc@counter } { } }
                  {
                    \int_compare:nNnTF
                      { \zref@extractdefault {#1} { zc@cntval } { -1 } }
                        >
                      { \zref@extractdefault {#2} { zc@cntval } { -1 } }
                      { \sort_return_swapped: }
                      { \sort_return_same:    }
                  }
                  { \sort_return_same: }
              }
          }
      }
```

```
  }
```

(*End definition for* `\__zrefclever_sort_default:nn`.)

Variant not provided by the kernel, for use in `\__zrefclever_sort_default_same_type:nn`.

```
\cs_generate_variant:Nn \tl_reverse_items:n { V }
```

`\__zrefclever_sort_default_same_type:nn`

`\__zrefclever_sort_default_same_type:nn {⟨label a⟩} {⟨label b⟩}`

```
\cs_new_protected:Npn \__zrefclever_sort_default_same_type:nn #1#2
  {
    \tl_set:Nx \l__zrefclever_label_enclcnt_a_tl
      { \zref@extractdefault {#1} { zc@enclcnt } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_enclcnt_a_tl
      { \tl_reverse_items:V \l__zrefclever_label_enclcnt_a_tl }
    \tl_set:Nx \l__zrefclever_label_enclcnt_b_tl
      { \zref@extractdefault {#2} { zc@enclcnt } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_enclcnt_b_tl
      { \tl_reverse_items:V \l__zrefclever_label_enclcnt_b_tl }
    \tl_set:Nx \l__zrefclever_label_enclval_a_tl
      { \zref@extractdefault {#1} { zc@enclval } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_enclval_a_tl
      { \tl_reverse_items:V \l__zrefclever_label_enclval_a_tl }
    \tl_set:Nx \l__zrefclever_label_enclval_b_tl
      { \zref@extractdefault {#2} { zc@enclval } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_enclval_b_tl
      { \tl_reverse_items:V \l__zrefclever_label_enclval_b_tl }
    \tl_set:Nx \l__zrefclever_label_extdoc_a_tl
      { \zref@extractdefault {#1} { externaldocument } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_extdoc_b_tl
      { \zref@extractdefault {#2} { externaldocument } { \c_empty_tl } }

    \bool_set_false:N \l__zrefclever_sort_decided_bool

    % First we check if there's any "external document" difference (coming
    % from 'zref-xr') and, if so, sort based on that.
    \tl_if_eq:NNF
      \l__zrefclever_label_extdoc_a_tl
      \l__zrefclever_label_extdoc_b_tl
      {
        \bool_if:nTF
          {
            \tl_if_empty_p:V \l__zrefclever_label_extdoc_a_tl &&
            ! \tl_if_empty_p:V \l__zrefclever_label_extdoc_b_tl
          }
          {
            \bool_set_true:N \l__zrefclever_sort_decided_bool
            \sort_return_same:
          }
          {
            \bool_if:nTF
              {
                ! \tl_if_empty_p:V \l__zrefclever_label_extdoc_a_tl &&
                \tl_if_empty_p:V \l__zrefclever_label_extdoc_b_tl
              }
            {
              \bool_set_true:N \l__zrefclever_sort_decided_bool
              \sort_return_swapped:
            }
            {
              \bool_set_true:N \l__zrefclever_sort_decided_bool
              % Two different "external documents": last resort, sort by the
              % document name itself.
              \str_compare:eNeTF
                { \l__zrefclever_label_extdoc_b_tl } <
                { \l__zrefclever_label_extdoc_a_tl }
                { \sort_return_swapped: }
                { \sort_return_same:    }
            }
        }
    }

  \bool_until_do:Nn \l__zrefclever_sort_decided_bool
    {
      \bool_if:nTF
        {
          % Both are empty: neither label has any (further) "enclosing
          % counters" (left).
          \tl_if_empty_p:V \l__zrefclever_label_enclcnt_a_tl &&
          \tl_if_empty_p:V \l__zrefclever_label_enclcnt_b_tl
        }
        {
          \exp_args:Nxx \tl_if_eq:nnTF
            { \zref@extractdefault {#1} { zc@counter } { } }
            { \zref@extractdefault {#2} { zc@counter } { } }
            {
              \bool_set_true:N \l__zrefclever_sort_decided_bool
              \int_compare:nNnTF
                { \zref@extractdefault {#1} { zc@cntval } { -1 } }
                  >
                { \zref@extractdefault {#2} { zc@cntval } { -1 } }
                { \sort_return_swapped: }
                { \sort_return_same:    }
            }
            {
              \msg_warning:nnnn { zref-clever }
                { counters-not-nested } {#1} {#2}
              \bool_set_true:N \l__zrefclever_sort_decided_bool
              \sort_return_same:
            }
        }
        {
          \bool_if:nTF
            {
              % ‘a’ is empty (and ‘b’ is not): ‘b’ may be nested in ‘a’.
              \tl_if_empty_p:V \l__zrefclever_label_enclcnt_a_tl
            }
            {
              \int_zero:N \l_tmpb_int
          \tl_map_inline:Nn \l__zrefclever_label_enclcnt_b_tl
            {
              \int_incr:N \l_tmpb_int
              \exp_args:Nnx \tl_if_eq:nnT {##1}
                { \zref@extractdefault {#1} { zc@counter } { } }
                {
                  \tl_map_break:n
                    {
                      \int_compare:nNnTF
                        { \zref@extractdefault {#1} { zc@cntval } { } }
                          >
                        {
                          \tl_item:Nn \l__zrefclever_label_enclval_b_tl
                            { \l_tmpb_int }
                        }
                        { \sort_return_swapped: }
                        { \sort_return_same:    }
                      \bool_set_true:N \l__zrefclever_sort_decided_bool
                    }
                }
            }
          \bool_if:NF \l__zrefclever_sort_decided_bool
            {
              \msg_warning:nnnn { zref-clever }
                { counters-not-nested } {#1} {#2}
              \bool_set_true:N \l__zrefclever_sort_decided_bool
              \sort_return_same:
            }
        }
        {
          \bool_if:nTF
            {
              % ‘b’ is empty (and ‘a’ is not): ‘a’ may be nested in ‘b’.
              \tl_if_empty_p:V \l__zrefclever_label_enclcnt_b_tl
            }
            {
              \int_zero:N \l_tmpa_int
              \tl_map_inline:Nn \l__zrefclever_label_enclcnt_a_tl
                {
                  \int_incr:N \l_tmpa_int
                  \exp_args:Nnx \tl_if_eq:nnT {##1}
                    { \zref@extractdefault {#2} { zc@counter } { } }
                    {
                      \tl_map_break:n
                        {
                          \int_compare:nNnTF
                            {
                              \tl_item:Nn
                                \l__zrefclever_label_enclval_a_tl
                                { \l_tmpa_int }
                            }
                              <
                            {
                              \zref@extractdefault {#2}
          { zc@cntval } { }
        }
        { \sort_return_same:    }
        { \sort_return_swapped: }
       \bool_set_true:N
        \l__zrefclever_sort_decided_bool
      }
    }
  }
 \bool_if:NF \l__zrefclever_sort_decided_bool
  {
   \msg_warning:nnnn { zref-clever }
    { counters-not-nested } {#1} {#2}
   \bool_set_true:N \l__zrefclever_sort_decided_bool
   \sort_return_same:
  }
}
{
 % Neither is empty: we can (possibly) compare the values
 % of the current enclosing counter in the loop, if they
 % are equal, we are still in the loop, if they are not, a
 % sorting decision can be made directly.
 \exp_args:Nxx \tl_if_eq:nnTF
  { \tl_head:N \l__zrefclever_label_enclcnt_a_tl }
  { \tl_head:N \l__zrefclever_label_enclcnt_b_tl }
  {
   \int_compare:nNnTF
    { \tl_head:N \l__zrefclever_label_enclval_a_tl }
     =
    { \tl_head:N \l__zrefclever_label_enclval_b_tl }
    {
     \tl_set:Nx \l__zrefclever_label_enclcnt_a_tl
      { \tl_tail:N \l__zrefclever_label_enclcnt_a_tl }
     \tl_set:Nx \l__zrefclever_label_enclcnt_b_tl
      { \tl_tail:N \l__zrefclever_label_enclcnt_b_tl }
     \tl_set:Nx \l__zrefclever_label_enclval_a_tl
      { \tl_tail:N \l__zrefclever_label_enclval_a_tl }
     \tl_set:Nx \l__zrefclever_label_enclval_b_tl
      { \tl_tail:N \l__zrefclever_label_enclval_b_tl }
    }
    {
     \bool_set_true:N \l__zrefclever_sort_decided_bool
     \int_compare:nNnTF
      { \tl_head:N \l__zrefclever_label_enclval_a_tl }
       >
      { \tl_head:N \l__zrefclever_label_enclval_b_tl }
      { \sort_return_swapped: }
      { \sort_return_same:    }
    }
  }
  {
   \msg_warning:nnnn { zref-clever }
    { counters-not-nested } {#1} {#2}
   \bool_set_true:N \l__zrefclever_sort_decided_bool
```

```
                              \sort_return_same:
                            }
                        }
                    }
                }
            }
        }
```

(*End definition for* `\__zrefclever_sort_default_same_type:nn`.)

_zrefclever_sort_default_different_types:nn

`\__zrefclever_sort_default_different_types:nn {⟨label a⟩} {⟨label b⟩}`

```
\cs_new_protected:Npn \__zrefclever_sort_default_different_types:nn #1#2
  {
```

Retrieve sort priorities for ⟨*label a*⟩ and ⟨*label b*⟩. `\l__zrefclever_typesort_seq` was stored in reverse sequence, and we compute the sort priorities in the negative range, so that we can implicitly rely on '0' being the "last value".

```
    \int_zero:N \l__zrefclever_sort_prior_a_int
    \int_zero:N \l__zrefclever_sort_prior_b_int
    \seq_map_indexed_inline:Nn \l__zrefclever_typesort_seq
      {
        \tl_if_eq:nnTF {##2} {{othertypes}}
          {
            \int_compare:nNnT { \l__zrefclever_sort_prior_a_int } = { 0 }
              { \int_set:Nn \l__zrefclever_sort_prior_a_int { - ##1 } }
            \int_compare:nNnT { \l__zrefclever_sort_prior_b_int } = { 0 }
              { \int_set:Nn \l__zrefclever_sort_prior_b_int { - ##1 } }
          }
          {
            \tl_if_eq:NnTF \l__zrefclever_label_type_a_tl {##2}
              { \int_set:Nn \l__zrefclever_sort_prior_a_int { - ##1 } }
              {
                \tl_if_eq:NnT \l__zrefclever_label_type_b_tl {##2}
                  { \int_set:Nn \l__zrefclever_sort_prior_b_int { - ##1 } }
              }
          }
      }
```

Then do the actual sorting.

```
    \bool_if:nTF
      {
        \int_compare_p:nNn
          { \l__zrefclever_sort_prior_a_int } <
          { \l__zrefclever_sort_prior_b_int }
      }
      { \sort_return_same: }
      {
        \bool_if:nTF
          {
            \int_compare_p:nNn
              { \l__zrefclever_sort_prior_a_int } >
              { \l__zrefclever_sort_prior_b_int }
          }
          { \sort_return_swapped: }
          {
            % Sort priorities are equal: the type that occurs first in
            % ‘labels’, as given by the user, is kept (or brought) forward.
            \seq_map_inline:Nn \l__zrefclever_label_types_seq
              {
                \tl_if_eq:NnTF \l__zrefclever_label_type_a_tl {##1}
                  { \seq_map_break:n { \sort_return_same: } }
                  {
                    \tl_if_eq:NnT \l__zrefclever_label_type_b_tl {##1}
                      { \seq_map_break:n { \sort_return_swapped: } }
                  }
              }
          }
      }
  }
```

(*End definition for* `\__zrefclever_sort_default_different_types:nn`.)

`\__zrefclever_sort_page:nn` The sorting function for sorting of defined labels for references to "page". This function is expected to be called within the sorting loop of `\__zrefclever_sort_labels:` and receives the pair of labels being considered for a change of order or not. It should *always* "return" either `\sort_return_same:` or `\sort_return_swapped:`. Compared to the sorting of default labels, this is a piece of cake (thanks to `abspage`).

```
\__zrefclever_sort_page:nn {⟨label a⟩} {⟨label b⟩}
```

```
\cs_new_protected:Npn \__zrefclever_sort_page:nn #1#2
  {
    \int_compare:nNnTF
      { \zref@extractdefault {#1} { abspage } {-1} }
        >
      { \zref@extractdefault {#2} { abspage } {-1} }
      { \sort_return_swapped: }
      { \sort_return_same:    }
  }
```

(*End definition for* `\__zrefclever_sort_page:nn`.)

8 Typesetting

"Typesetting" the reference, which here includes the parsing of the labels and eventual compression of labels in sequence into ranges, is definitely the "crux" of zref-clever. This because we process the label set as a stack, in a single pass, and hence "parsing", "compressing", and "typesetting" must be decided upon at the same time, making it difficult to slice the job into more specific and self-contained tasks. So, do bear this in mind before you curse me for the length of some of the functions below, or before a more orthodox "docstripper" complains about me not sticking to code commenting conventions to keep the code more readable in the `.dtx` file.

While processing the label stack (kept in `\l__zrefclever_typeset_labels_seq`), `\__zrefclever_typeset_refs:` "sees" two labels, and two labels only, the "current" one (kept in `\l__zrefclever_label_a_tl`), and the "next" one (kept in `\l__zrefclever_label_b_tl`). However, the typesetting needs (a lot) more information than just these two immediate labels to make a number of critical decisions. Some examples: i) We cannot know if labels “current” and “next” of the same type are a “pair”, or just “elements in a list”, until we examine the label after “next”; ii) If the “next” label is of the same type as the “current”, and it is in immediate sequence to it, it potentially forms a “range”, but we cannot know if “next” is actually the end of the range until we examined an arbitrary number of labels, and found one which is not in sequence from the previous one; iii) When processing a type block, the “name” comes first, however, we only know if that name should be plural, or if it should be included in the hyperlink, after processing an arbitrary number of labels and find one of a different type. One could naively assume that just examining “next” would be enough for this, since we can know if it is of the same type or not. Alas, “there be ranges”, and a compression operation may boil down to a single element, so we have to process the whole type block to know how its name should be typeset; iv) Similar issues apply to lists of type blocks, each of which is of arbitrary length: we can only know if two type blocks form a “pair” or are “elements in a list” when we finish the block. Etc. etc. etc.

We handle this by storing the reference “pieces” in “queues”, instead of typesetting them immediately upon processing. The “queues” get typeset at the point where all the information needed is available, which usually happens when a type block finishes (we see something of a different type in “next”, signaled by `\l__zrefclever_last_of_type_bool`), or the stack itself finishes (has no more elements, signaled by `\l__zrefclever_typeset_last_bool`). And, in processing a type block, the type “name” gets added last (on the left) of the queue. The very first reference of its type always follows the name, since it may form a hyperlink with it (so we keep it stored separately, in `\l__zrefclever_type_first_label_tl`, with `\l__zrefclever_type_first_label_type_tl` being its type). And, since we may need up to two type blocks in storage before typesetting, we have two of these “queues”: `\l__zrefclever_typeset_queue_curr_tl` and `\l__zrefclever_typeset_queue_prev_tl`.

Some of the relevant cases (e.g., distinguishing “pair” from “list”) are handled by counters, the main ones are: one for the “type” (`\l__zrefclever_type_count_int`) and one for the “label in the current type block” (`\l__zrefclever_label_count_int`).

Range compression, in particular, relies heavily on counting to be able do distinguish relevant cases. `\l__zrefclever_range_count_int` counts the number of elements in the current sequential “streak”, and `\l__zrefclever_range_same_count_int` counts the number of *equal* elements in that same “streak”. The difference between the two allows us to distinguish the cases in which a range actually “skips” a number in the sequence, in which case we should use a range separator, from when they are after all just contiguous, in which case a pair separator is called for. Since, as usual, we can only know this when a arbitrary long “streak” finishes, we have to store the label which (potentially) begins a range (kept in `\l__zrefclever_range_beg_label_tl`). `\l__zrefclever_next_maybe_range_bool` signals when “next” is potentially a range with “current”, and `\l__zrefclever_next_is_same_bool` when their values are actually equal.

One further thing to discuss here – to keep this “on record” – is inhibition of compression for individual labels. It is not difficult to handle it at the infrastructure side, what gets sloppy is the user facing syntax to signal such inhibition. For some possible alternatives for this (and good ones at that) see https://tex.stackexchange.com/q/611370 (thanks Enrico Gregorio, Phelype Oleinik, and Steven B. Segletes). Yet another alternative would be an option receiving the label(s) not to be compressed, this would be a repetition, but would keep the syntax clean. All in all, probably the best is simply not to allow individual inhibition of compression. We can already control compression of each `\zcref` call with existing options, this should be enough. I don’t think the small extra flexibility individual label control for this would grant is worth the syntax disruption it would entail. Anyway, it would be easy to deal with this in case the need arose, by just adding another condition (coming from whatever the chosen syntax was) when we check for `\__zrefclever_labels_in_sequence:nn` in `\__zrefclever_typeset_refs_not_-last_of_type:`. But I remain unconvinced of the pertinence of doing so.

Variables

\l__zrefclever_typeset_labels_seq
\l__zrefclever_typeset_last_bool
\l__zrefclever_last_of_type_bool

Auxiliary variables for `\__zrefclever_typeset_refs:` main stack control.

```
\seq_new:N \l__zrefclever_typeset_labels_seq
\bool_new:N \l__zrefclever_typeset_last_bool
\bool_new:N \l__zrefclever_last_of_type_bool
```

(*End definition for* `\l__zrefclever_typeset_labels_seq`*,* `\l__zrefclever_typeset_last_bool`*, and* `\l__zrefclever_last_of_type_bool`*.*)

\l__zrefclever_type_count_int
\l__zrefclever_label_count_int

Auxiliary variables for `\__zrefclever_typeset_refs:` main counters.

```
\int_new:N \l__zrefclever_type_count_int
\int_new:N \l__zrefclever_label_count_int
```

(*End definition for* `\l__zrefclever_type_count_int` *and* `\l__zrefclever_label_count_int`*.*)

\l__zrefclever_label_a_tl
\l__zrefclever_label_b_tl
\l__zrefclever_typeset_queue_prev_tl
\l__zrefclever_typeset_queue_curr_tl
\l__zrefclever_type_first_label_tl
\l__zrefclever_type_first_label_type_tl

Auxiliary variables for `\__zrefclever_typeset_refs:` main "queue" control and storage.

```
\tl_new:N \l__zrefclever_label_a_tl
\tl_new:N \l__zrefclever_label_b_tl
\tl_new:N \l__zrefclever_typeset_queue_prev_tl
\tl_new:N \l__zrefclever_typeset_queue_curr_tl
\tl_new:N \l__zrefclever_type_first_label_tl
\tl_new:N \l__zrefclever_type_first_label_type_tl
```

(*End definition for* `\l__zrefclever_label_a_tl` *and others.*)

\l__zrefclever_type_name_tl
\l__zrefclever_name_in_link_bool
\l__zrefclever_name_format_tl
\l__zrefclever_name_format_fallback_tl

Auxiliary variables for `\__zrefclever_typeset_refs:` type name handling.

```
\tl_new:N \l__zrefclever_type_name_tl
\bool_new:N \l__zrefclever_name_in_link_bool
\tl_new:N \l__zrefclever_name_format_tl
\tl_new:N \l__zrefclever_name_format_fallback_tl
```

(*End definition for* `\l__zrefclever_type_name_tl` *and others.*)

\l__zrefclever_range_count_int
\l__zrefclever_range_same_count_int
\l__zrefclever_range_beg_label_tl
\l__zrefclever_next_maybe_range_bool
\l__zrefclever_next_is_same_bool

Auxiliary variables for `\__zrefclever_typeset_refs:` range handling.

```
\int_new:N \l__zrefclever_range_count_int
\int_new:N \l__zrefclever_range_same_count_int
\tl_new:N \l__zrefclever_range_beg_label_tl
\bool_new:N \l__zrefclever_next_maybe_range_bool
\bool_new:N \l__zrefclever_next_is_same_bool
```

(*End definition for* `\l__zrefclever_range_count_int` *and others.*)

\l__zrefclever_tpairsep_tl
\l__zrefclever_tlistsep_tl
\l__zrefclever_tlastsep_tl
\l__zrefclever_namesep_tl
\l__zrefclever_pairsep_tl
\l__zrefclever_listsep_tl
\l__zrefclever_lastsep_tl
\l__zrefclever_rangesep_tl
\l__zrefclever_refpre_out_tl
\l__zrefclever_refpos_out_tl
\l__zrefclever_refpre_in_tl
\l__zrefclever_refpos_in_tl
\l__zrefclever_namefont_tl
\l__zrefclever_reffont_out_tl
\l__zrefclever_reffont_in_tl

Auxiliary variables for `\__zrefclever_typeset_refs:` separators, refpre/pos and font options.

```
\tl_new:N \l__zrefclever_tpairsep_tl
\tl_new:N \l__zrefclever_tlistsep_tl
\tl_new:N \l__zrefclever_tlastsep_tl
\tl_new:N \l__zrefclever_namesep_tl
\tl_new:N \l__zrefclever_pairsep_tl
\tl_new:N \l__zrefclever_listsep_tl
\tl_new:N \l__zrefclever_lastsep_tl
\tl_new:N \l__zrefclever_rangesep_tl
\tl_new:N \l__zrefclever_refpre_out_tl
\tl_new:N \l__zrefclever_refpos_out_tl
\tl_new:N \l__zrefclever_refpre_in_tl
\tl_new:N \l__zrefclever_refpos_in_tl
\tl_new:N \l__zrefclever_namefont_tl
\tl_new:N \l__zrefclever_reffont_out_tl
\tl_new:N \l__zrefclever_reffont_in_tl
```

(*End definition for* `\l__zrefclever_tpairsep_tl` *and others.*)

Main functions

__zrefclever_typeset_refs:

Main typesetting function for `\zcref`.

```
\cs_new_protected:Npn \__zrefclever_typeset_refs:
  {
    \seq_set_eq:NN \l__zrefclever_typeset_labels_seq
      \l__zrefclever_zcref_labels_seq
    \tl_clear:N \l__zrefclever_typeset_queue_prev_tl
    \tl_clear:N \l__zrefclever_typeset_queue_curr_tl
    \tl_clear:N \l__zrefclever_type_first_label_tl
    \tl_clear:N \l__zrefclever_type_first_label_type_tl
    \tl_clear:N \l__zrefclever_range_beg_label_tl
    \int_zero:N \l__zrefclever_label_count_int
    \int_zero:N \l__zrefclever_type_count_int
    \int_zero:N \l__zrefclever_range_count_int
    \int_zero:N \l__zrefclever_range_same_count_int

    % Get type block options (not type-specific).
    \__zrefclever_get_ref_string:nN { tpairsep }
      \l__zrefclever_tpairsep_tl
    \__zrefclever_get_ref_string:nN { tlistsep }
      \l__zrefclever_tlistsep_tl
    \__zrefclever_get_ref_string:nN { tlastsep }
      \l__zrefclever_tlastsep_tl

    % Process label stack.
    \bool_set_false:N \l__zrefclever_typeset_last_bool
    \bool_until_do:Nn \l__zrefclever_typeset_last_bool
      {
        \seq_pop_left:NN \l__zrefclever_typeset_labels_seq
          \l__zrefclever_label_a_tl
        \seq_if_empty:NTF \l__zrefclever_typeset_labels_seq
          {
            \tl_clear:N \l__zrefclever_label_b_tl
            \bool_set_true:N \l__zrefclever_typeset_last_bool
          }
          {
            \seq_get_left:NN \l__zrefclever_typeset_labels_seq
              \l__zrefclever_label_b_tl
          }

        \tl_if_eq:NnTF \l__zrefclever_ref_property_tl { page }
          {
            \tl_set:Nn \l__zrefclever_label_type_a_tl { page }
            \tl_set:Nn \l__zrefclever_label_type_b_tl { page }
          }
          {
            \tl_set:Nx \l__zrefclever_label_type_a_tl
              {
                \zref@extractdefault
                  { \l__zrefclever_label_a_tl } { zc@type } { \c_empty_tl }
              }
            \tl_set:Nx \l__zrefclever_label_type_b_tl
              {
                \zref@extractdefault
                  { \l__zrefclever_label_b_tl } { zc@type } { \c_empty_tl }
              }
          }

        % First, we establish whether the "current label" (i.e. ‘a’) is the
        % last one of its type.  This can happen because the "next label"
        % (i.e. ‘b’) is of a different type (or different definition status),
        % or because we are at the end of the list.
        \bool_if:NTF \l__zrefclever_typeset_last_bool
          { \bool_set_true:N \l__zrefclever_last_of_type_bool }
          {
            \zref@ifrefundefined { \l__zrefclever_label_a_tl }
              {
                \zref@ifrefundefined { \l__zrefclever_label_b_tl }
                  { \bool_set_false:N \l__zrefclever_last_of_type_bool }
                  { \bool_set_true:N \l__zrefclever_last_of_type_bool  }
              }
              {
                \zref@ifrefundefined { \l__zrefclever_label_b_tl }
                  { \bool_set_true:N \l__zrefclever_last_of_type_bool }
                  {
                    % Neither is undefined, we must check the types.
                    \bool_if:nTF
                      {
                        % Both empty: same "type".
                        \tl_if_empty_p:N \l__zrefclever_label_type_a_tl &&
                        \tl_if_empty_p:N \l__zrefclever_label_type_b_tl
                      }
                      { \bool_set_false:N \l__zrefclever_last_of_type_bool }
                      {
                        \bool_if:nTF
                          {
                            % Neither empty: compare types.
                        ! \tl_if_empty_p:N \l__zrefclever_label_type_a_tl
                        &&
                        ! \tl_if_empty_p:N \l__zrefclever_label_type_b_tl
                      }
                      {
                        \tl_if_eq:NNTF
                          \l__zrefclever_label_type_a_tl
                          \l__zrefclever_label_type_b_tl
                          {
                            \bool_set_false:N
                              \l__zrefclever_last_of_type_bool
                          }
                          {
                            \bool_set_true:N
                              \l__zrefclever_last_of_type_bool
                          }
                      }
                      % One empty, the other not: different "types".
                      {
                        \bool_set_true:N
                          \l__zrefclever_last_of_type_bool
                      }
                  }
              }
          }
      }

    % Handle warnings in case of reference or type undefined.
    \zref@refused { \l__zrefclever_label_a_tl }
    \zref@ifrefundefined { \l__zrefclever_label_a_tl }
      {}
      {
        \tl_if_empty:NT \l__zrefclever_label_type_a_tl
          {
            \msg_warning:nnx { zref-clever } { missing-type }
              { \l__zrefclever_label_a_tl }
          }
      }

    % Get type-specific separators, refpre/pos and font options, once per
    % type.
    \int_compare:nNnT { \l__zrefclever_label_count_int } = { 0 }
      {
        \__zrefclever_get_ref_string:nN { namesep    }
          \l__zrefclever_namesep_tl
        \__zrefclever_get_ref_string:nN { rangesep   }
          \l__zrefclever_rangesep_tl
        \__zrefclever_get_ref_string:nN { pairsep    }
          \l__zrefclever_pairsep_tl
        \__zrefclever_get_ref_string:nN { listsep    }
          \l__zrefclever_listsep_tl
        \__zrefclever_get_ref_string:nN { lastsep    }
          \l__zrefclever_lastsep_tl
        \__zrefclever_get_ref_string:nN { refpre     }
```

```
              \l__zrefclever_refpre_out_tl
            \__zrefclever_get_ref_string:nN { refpos     }
              \l__zrefclever_refpos_out_tl
            \__zrefclever_get_ref_string:nN { refpre-in  }
              \l__zrefclever_refpre_in_tl
            \__zrefclever_get_ref_string:nN { refpos-in  }
              \l__zrefclever_refpos_in_tl
            \__zrefclever_get_ref_font:nN   { namefont   }
              \l__zrefclever_namefont_tl
            \__zrefclever_get_ref_font:nN   { reffont    }
              \l__zrefclever_reffont_out_tl
            \__zrefclever_get_ref_font:nN   { reffont-in }
              \l__zrefclever_reffont_in_tl
          }

        % Here we send this to a couple of auxiliary functions.
        \bool_if:NTF \l__zrefclever_last_of_type_bool
          % There exists no next label of the same type as the current.
          { \__zrefclever_typeset_refs_last_of_type: }
          % There exists a next label of the same type as the current.
          { \__zrefclever_typeset_refs_not_last_of_type: }
      }
  }
```

(*End definition for* `\__zrefclever_typeset_refs:`*.*)

This is actually the one meaningful "big branching" we can do while processing the label stack: i) the "current" label is the last of its type block; or ii) the "current" label is *not* the last of its type block. Indeed, as mentioned above, quite a number of things can only be decided when the type block ends, and we only know this when we look at the "next" label and find something of a different "type" (loose here, maybe different definition status, maybe end of stack). So, though this is not very strict, `\__zrefclever_typeset_refs_last_of_type:` is more of a "wrapping up" function, and it is indeed the one which does the actual typesetting, while `\__zrefclever_typeset_refs_not_last_of_type:` is more of an "accumulation" function.

__zrefclever_typeset_refs_last_of_type:

Handles typesetting when the current label is the last of its type.

```
\cs_new_protected:Npn \__zrefclever_typeset_refs_last_of_type:
  {
    % Process the current label to the current queue.
    \int_case:nnF { \l__zrefclever_label_count_int }
      {
        % It is the last label of its type, but also the first one, and that's
        % what matters here: just store it.
        { 0 }
        {
          \tl_set:NV \l__zrefclever_type_first_label_tl
            \l__zrefclever_label_a_tl
          \tl_set:NV \l__zrefclever_type_first_label_type_tl
            \l__zrefclever_label_type_a_tl
        }

        % The last is the second: we have a pair (if not repeated).
        { 1 }
        {
```

```
      \int_compare:nNnF { \l__zrefclever_range_same_count_int } = { 1 }
       {
        \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
         {
          \exp_not:V \l__zrefclever_pairsep_tl
          \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
         }
       }
     }
    }
    % Last is third or more of its type: without repetition, we'd have the
    % last element on a list, but control for possible repetition.
    {
     \int_case:nnF { \l__zrefclever_range_count_int }
      {
       % There was no range going on.
       { 0 }
       {
        \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
         {
          \exp_not:V \l__zrefclever_lastsep_tl
          \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
         }
       }
       % Last in the range is also the second in it.
       { 1 }
       {
        \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
         {
          % We know 'range_beg_label' is not empty, since this is the
          % second element in the range, but the third or more in the
          % type list.
          \exp_not:V \l__zrefclever_listsep_tl
          \__zrefclever_get_ref:V \l__zrefclever_range_beg_label_tl
          \int_compare:nNnF
           { \l__zrefclever_range_same_count_int } = { 1 }
           {
            \exp_not:V \l__zrefclever_lastsep_tl
            \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
           }
         }
       }
      }
      % Last in the range is third or more in it.
      {
       \int_case:nnF
        {
         \l__zrefclever_range_count_int -
         \l__zrefclever_range_same_count_int
        }
        {
         % Repetition, not a range.
         { 0 }
         {
                % If ‘range_beg_label’ is empty, it means it was also the
                % first of the type, and hence was already handled.
                \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                  {
                    \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                      {
                        \exp_not:V \l__zrefclever_lastsep_tl
                        \__zrefclever_get_ref:V
                          \l__zrefclever_range_beg_label_tl
                      }
                  }
              }
              % A ‘range’, but with no skipped value, treat as list.
              { 1 }
              {
                \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                  {
                    % Ditto.
                    \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                      {
                        \exp_not:V \l__zrefclever_listsep_tl
                        \__zrefclever_get_ref:V
                          \l__zrefclever_range_beg_label_tl
                      }
                    \exp_not:V \l__zrefclever_lastsep_tl
                    \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
                  }
              }
            }
            {
              % An actual range.
              \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                {
                  % Ditto.
                  \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                    {
                      \exp_not:V \l__zrefclever_lastsep_tl
                      \__zrefclever_get_ref:V
                        \l__zrefclever_range_beg_label_tl
                    }
                  \exp_not:V \l__zrefclever_rangesep_tl
                  \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
                }
            }
        }
    }

  % Handle "range" option.  The idea is simple: if the queue is not empty,
  % we replace it with the end of the range (or pair).  We can still
  % retrieve the end of the range from ‘label_a’ since we know to be
  % processing the last label of its type at this point.
  \bool_if:NT \l__zrefclever_typeset_range_bool
    {
      \tl_if_empty:NTF \l__zrefclever_typeset_queue_curr_tl
```

```
          {
            \zref@ifrefundefined { \l__zrefclever_type_first_label_tl }
              { }
              {
                \msg_warning:nnx { zref-clever } { single-element-range }
                  { \l__zrefclever_type_first_label_type_tl }
              }
          }
          {
            \bool_set_false:N \l__zrefclever_next_maybe_range_bool
            \zref@ifrefundefined { \l__zrefclever_type_first_label_tl }
              { }
              {
                \__zrefclever_labels_in_sequence:nn
                  { \l__zrefclever_type_first_label_tl }
                  { \l__zrefclever_label_a_tl }
              }
            \tl_set:Nx \l__zrefclever_typeset_queue_curr_tl
              {
                \bool_if:NTF \l__zrefclever_next_maybe_range_bool
                  { \exp_not:V \l__zrefclever_pairsep_tl }
                  { \exp_not:V \l__zrefclever_rangesep_tl }
                \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
              }
          }
      }

    % Now that the type block is finished, we can add the name and the first
    % ref to the queue.  Also, if "typeset" option is not "both", handle it
    % here as well.
    \__zrefclever_type_name_setup:
    \bool_if:nTF
      { \l__zrefclever_typeset_ref_bool && \l__zrefclever_typeset_name_bool }
      {
        \tl_put_left:Nx \l__zrefclever_typeset_queue_curr_tl
          { \__zrefclever_get_ref_first: }
      }
      {
        \bool_if:nTF
          { \l__zrefclever_typeset_ref_bool }
          {
            \tl_put_left:Nx \l__zrefclever_typeset_queue_curr_tl
              { \__zrefclever_get_ref:V \l__zrefclever_type_first_label_tl }
          }
          {
            \bool_if:nTF
              { \l__zrefclever_typeset_name_bool }
              {
                \tl_set:Nx \l__zrefclever_typeset_queue_curr_tl
                  {
                    \bool_if:NTF \l__zrefclever_name_in_link_bool
                      {
                        \exp_not:N \group_begin:
                        \exp_not:V \l__zrefclever_namefont_tl
                    % It’s two ’@s’, but escaped for DocStrip.
                    \exp_not:N \hyper@@link
                      {
                        \__zrefclever_extract_url:V
                          \l__zrefclever_type_first_label_tl
                      }
                      {
                        \zref@extractdefault
                          { \l__zrefclever_type_first_label_tl }
                          { anchor } {}
                      }
                      { \exp_not:V \l__zrefclever_type_name_tl }
                    \exp_not:N \group_end:
                  }
                  {
                    \exp_not:N \group_begin:
                    \exp_not:V \l__zrefclever_namefont_tl
                    \exp_not:V \l__zrefclever_type_name_tl
                    \exp_not:N \group_end:
                  }
              }
          }
          {
            % Logically, this case would correspond to "typeset=none", but
            % it should not occur, given that the options are set up to
            % typeset either "ref" or "name".  Still, leave here a
            % sensible fallback, equal to the behavior of "both".
            \tl_put_left:Nx \l__zrefclever_typeset_queue_curr_tl
              { \__zrefclever_get_ref_first: }
          }
      }
  }

% Typeset the previous type, if there is one.
\int_compare:nNnT { \l__zrefclever_type_count_int } > { 0 }
  {
    \int_compare:nNnT { \l__zrefclever_type_count_int } > { 1 }
      { \l__zrefclever_tlistsep_tl }
    \l__zrefclever_typeset_queue_prev_tl
  }

% Wrap up loop, or prepare for next iteration.
\bool_if:NTF \l__zrefclever_typeset_last_bool
  {
    % We are finishing, typeset the current queue.
    \int_case:nnF { \l__zrefclever_type_count_int }
      {
        % Single type.
        { 0 }
        { \l__zrefclever_typeset_queue_curr_tl }
        % Pair of types.
        { 1 }
        {
          \l__zrefclever_tpairsep_tl
```

```
              \l__zrefclever_typeset_queue_curr_tl
            }
          }
          {
            % Last in list of types.
            \l__zrefclever_tlastsep_tl
            \l__zrefclever_typeset_queue_curr_tl
          }
      }
      {
        % There are further labels, set variables for next iteration.
        \tl_set_eq:NN \l__zrefclever_typeset_queue_prev_tl
          \l__zrefclever_typeset_queue_curr_tl
        \tl_clear:N \l__zrefclever_typeset_queue_curr_tl
        \tl_clear:N \l__zrefclever_type_first_label_tl
        \tl_clear:N \l__zrefclever_type_first_label_type_tl
        \tl_clear:N \l__zrefclever_range_beg_label_tl
        \int_zero:N \l__zrefclever_label_count_int
        \int_incr:N \l__zrefclever_type_count_int
        \int_zero:N \l__zrefclever_range_count_int
        \int_zero:N \l__zrefclever_range_same_count_int
      }
  }
```

(*End definition for* `\__zrefclever_typeset_refs_last_of_type:`*.*)

__zrefclever_typeset_refs_not_last_of_type: Handles typesetting when the current label is not the last of its type.

```
\cs_new_protected:Npn \__zrefclever_typeset_refs_not_last_of_type:
  {
    % Signal if next label may form a range with the current one (only
    % considered if compression is enabled in the first place).
    \bool_set_false:N \l__zrefclever_next_maybe_range_bool
    \bool_set_false:N \l__zrefclever_next_is_same_bool
    \bool_if:NT \l__zrefclever_typeset_compress_bool
      {
        \zref@ifrefundefined { \l__zrefclever_label_a_tl }
          { }
          {
            \__zrefclever_labels_in_sequence:nn
              { \l__zrefclever_label_a_tl } { \l__zrefclever_label_b_tl }
          }
      }

    % Process the current label to the current queue.
    \int_compare:nNnTF { \l__zrefclever_label_count_int } = { 0 }
      {
        % Current label is the first of its type (also not the last, but it
        % doesn't matter here): just store the label.
        \tl_set:NV \l__zrefclever_type_first_label_tl
          \l__zrefclever_label_a_tl
        \tl_set:NV \l__zrefclever_type_first_label_type_tl
          \l__zrefclever_label_type_a_tl

        % If the next label may be part of a range, we set 'range_beg_label'
```

```
          % to "empty" (we deal with it as the "first", and must do it there, to
          % handle hyperlinking), but also step the range counters.
          \bool_if:NT \l__zrefclever_next_maybe_range_bool
            {
              \tl_clear:N \l__zrefclever_range_beg_label_tl
              \int_incr:N \l__zrefclever_range_count_int
              \bool_if:NT \l__zrefclever_next_is_same_bool
                { \int_incr:N \l__zrefclever_range_same_count_int }
           }
        }
        {
          % Current label is neither the first (nor the last) of its type.
          \bool_if:NTF \l__zrefclever_next_maybe_range_bool
            {
              % Starting, or continuing a range.
              \int_compare:nNnTF
                { \l__zrefclever_range_count_int } = { 0 }
                {
                  % There was no range going, we are starting one.
                  \tl_set:NV \l__zrefclever_range_beg_label_tl
                    \l__zrefclever_label_a_tl
                  \int_incr:N \l__zrefclever_range_count_int
                  \bool_if:NT \l__zrefclever_next_is_same_bool
                    { \int_incr:N \l__zrefclever_range_same_count_int }
                }
                {
                  % Second or more in the range, but not the last.
                  \int_incr:N \l__zrefclever_range_count_int
                  \bool_if:NT \l__zrefclever_next_is_same_bool
                    { \int_incr:N \l__zrefclever_range_same_count_int }
                }
            }
            {
              % Next element is not in sequence: there was no range, or we are
              % closing one.
              \int_case:nnF { \l__zrefclever_range_count_int }
                {
                  % There was no range going on.
                  { 0 }
                  {
                    \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                      {
                        \exp_not:V \l__zrefclever_listsep_tl
                        \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
                      }
                  }
                  % Last is second in the range: if ‘range_same_count’ is also
                  % ‘1’, it’s a repetition (drop it), otherwise, it’s a "pair
                  % within a list", treat as list.
                  { 1 }
                  {
                    \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                      {
                        \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                        {
                          \exp_not:V \l__zrefclever_listsep_tl
                          \__zrefclever_get_ref:V
                            \l__zrefclever_range_beg_label_tl
                        }
                      \int_compare:nNnF
                        { \l__zrefclever_range_same_count_int } = { 1 }
                        {
                          \exp_not:V \l__zrefclever_listsep_tl
                          \__zrefclever_get_ref:V
                            \l__zrefclever_label_a_tl
                        }
                    }
                }
              }
              {
                % Last is third or more in the range: if ‘range_count’ and
                % ‘range_same_count’ are the same, its a repetition (drop it),
                % if they differ by ‘1’, its a list, if they differ by more,
                % it is a real range.
                \int_case:nnF
                  {
                    \l__zrefclever_range_count_int -
                    \l__zrefclever_range_same_count_int
                  }
                  {
                    { 0 }
                    {
                      \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                        {
                          \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                            {
                              \exp_not:V \l__zrefclever_listsep_tl
                              \__zrefclever_get_ref:V
                                \l__zrefclever_range_beg_label_tl
                            }
                        }
                    }
                    { 1 }
                    {
                      \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                        {
                          \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                            {
                              \exp_not:V \l__zrefclever_listsep_tl
                              \__zrefclever_get_ref:V
                                \l__zrefclever_range_beg_label_tl
                            }
                          \exp_not:V \l__zrefclever_listsep_tl
                          \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
                        }
                    }
                  }
                  {
                    \tl_put_right:Nx \l__zrefclever_typeset_queue_curr_tl
                      {
                        \tl_if_empty:VF \l__zrefclever_range_beg_label_tl
                          {
                            \exp_not:V \l__zrefclever_listsep_tl
                            \__zrefclever_get_ref:V
                              \l__zrefclever_range_beg_label_tl
                          }
                        \exp_not:V \l__zrefclever_rangesep_tl
                        \__zrefclever_get_ref:V \l__zrefclever_label_a_tl
                      }
                  }
              }
            % Reset counters.
            \int_zero:N \l__zrefclever_range_count_int
            \int_zero:N \l__zrefclever_range_same_count_int
          }
      }
    % Step label counter for next iteration.
    \int_incr:N \l__zrefclever_label_count_int
  }
```

(*End definition for* `\__zrefclever_typeset_refs_not_last_of_type:`.)

Aux functions

`\__zrefclever_get_ref:n` and `\__zrefclever_get_ref_first:` are the two functions which actually build the reference blocks for typesetting. `\__zrefclever_get_ref:n` handles all references but the first of its type, and `\__zrefclever_get_ref_first:` deals with the first reference of a type. Saying they do “typesetting” is imprecise though, they actually prepare material to be accumulated in `\l__zrefclever_typeset_queue_curr_tl` inside `\__zrefclever_typeset_refs_last_of_type:` and `\__zrefclever_typeset_refs_not_last_of_type:`. And this difference results quite crucial for the TEXnical requirements of these functions. This because, as we are processing the label stack and accumulating content in the queue, we are using a number of variables which are transient to the current label, the label properties among them, but not only. Hence, these variables *must* be expanded to their current values to be stored in the queue. Indeed, `\__zrefclever_get_ref:n` and `\__zrefclever_get_ref_first:` get called, as they must, in the context of `x` type expansions. But we don't want to expand the values of the variables themselves, so we need to get current values, but stop expansion after that. In particular, reference options given by the user should reach the stream for its final typesetting (when the queue itself gets typeset) *unmodified* (“no manipulation”, to use the `n` signature jargon). We also need to prevent premature expansion of material that can't be expanded at this point (e.g. grouping, `\zref@default` or `\hyper@@link`). In a nutshell, the job of these two functions is putting the pieces in place, but with proper expansion control.

`\__zrefclever_ref_default:`
`\__zrefclever_name_default:`

Default values for undefined references and undefined type names, respectively. We are ultimately using `\zref@default`, but calls to it should be made through these internal functions, according to the case. As a bonus, we don't need to protect them with `\exp_not:N`, as `\zref@default` would require, since we already define them protected.

```
\cs_new_protected:Npn \__zrefclever_ref_default:
  { \zref@default }
\cs_new_protected:Npn \__zrefclever_name_default:
  { \zref@default }
```

(*End definition for* `\__zrefclever_ref_default:` *and* `\__zrefclever_name_default:`*.*)

`\__zrefclever_get_ref:n` Handles a complete reference block to be accumulated in the "queue", including "pre" and "pos" elements, and hyperlinking. For use with all labels, except the first of its type, which is done by `\__zrefclever_get_ref_first:`.

`\__zrefclever_get_ref:n {⟨label⟩}`

```
\cs_new:Npn \__zrefclever_get_ref:n #1
  {
    \zref@ifrefcontainsprop {#1} { \l__zrefclever_ref_property_tl }
      {
        \bool_if:nTF
          {
            \l__zrefclever_use_hyperref_bool &&
            ! \l__zrefclever_link_star_bool
          }
          {
            \exp_not:N \group_begin:
            \exp_not:V \l__zrefclever_reffont_out_tl
            \exp_not:V \l__zrefclever_refpre_out_tl
            \exp_not:N \group_begin:
            \exp_not:V \l__zrefclever_reffont_in_tl
            % It's two '@s', but escaped for DocStrip.
            \exp_not:N \hyper@@link
              { \__zrefclever_extract_url:n {#1} }
              { \zref@extractdefault {#1} { anchor } { } }
              {
                \exp_not:V \l__zrefclever_refpre_in_tl
                \zref@extractdefault {#1}
                  { \l__zrefclever_ref_property_tl } { }
                \exp_not:V \l__zrefclever_refpos_in_tl
              }
            \exp_not:N \group_end:
            \exp_not:V \l__zrefclever_refpos_out_tl
            \exp_not:N \group_end:
          }
          {
            \exp_not:N \group_begin:
            \exp_not:V \l__zrefclever_reffont_out_tl
            \exp_not:V \l__zrefclever_refpre_out_tl
            \exp_not:N \group_begin:
            \exp_not:V \l__zrefclever_reffont_in_tl
            \exp_not:V \l__zrefclever_refpre_in_tl
            \zref@extractdefault {#1} { \l__zrefclever_ref_property_tl } { }
            \exp_not:V \l__zrefclever_refpos_in_tl
            \exp_not:N \group_end:
            \exp_not:V \l__zrefclever_refpos_out_tl
            \exp_not:N \group_end:
          }
      }
      { \__zrefclever_ref_default: }
  }
\cs_generate_variant:Nn \__zrefclever_get_ref:n { V }
```

(*End definition for* `\__zrefclever_get_ref:n`*.*)

`\__zrefclever_get_ref_first:` Handles a complete reference block for the first label of its type to be accumulated in the "queue", including "pre" and "pos" elements, hyperlinking, and the reference type "name". It does not receive arguments, but relies on being called in the appropriate place in `\__zrefclever_typeset_refs_last_of_type:` where a number of variables are expected to be appropriately set for it to consume. Prominently among those is `\l__zrefclever_type_first_label_tl`, but it also expected to be called right after `\__zrefclever_type_name_setup:` which sets `\l__zrefclever_type_name_tl` and `\l__zrefclever_name_in_link_bool` which it uses.

```
\cs_new:Npn \__zrefclever_get_ref_first:
  {
    \zref@ifrefundefined { \l__zrefclever_type_first_label_tl }
      { \__zrefclever_ref_default: }
      {
        \bool_if:NTF \l__zrefclever_name_in_link_bool
          {
            \zref@ifrefcontainsprop
              { \l__zrefclever_type_first_label_tl }
              { \l__zrefclever_ref_property_tl }
              {
                % It's two '@s', but escaped for DocStrip.
                \exp_not:N \hyper@@link
                  {
                    \__zrefclever_extract_url:V
                      \l__zrefclever_type_first_label_tl
                  }
                  {
                    \zref@extractdefault
                      { \l__zrefclever_type_first_label_tl }
                      { anchor } { }
                  }
                  {
                    \exp_not:N \group_begin:
                    \exp_not:V \l__zrefclever_namefont_tl
                    \exp_not:V \l__zrefclever_type_name_tl
                    \exp_not:N \group_end:
                    \exp_not:V \l__zrefclever_namesep_tl
                    \exp_not:N \group_begin:
                    \exp_not:V \l__zrefclever_reffont_out_tl
                    \exp_not:V \l__zrefclever_refpre_out_tl
                    \exp_not:N \group_begin:
                    \exp_not:V \l__zrefclever_reffont_in_tl
                    \exp_not:V \l__zrefclever_refpre_in_tl
                    \zref@extractdefault
                      { \l__zrefclever_type_first_label_tl }
                      { \l__zrefclever_ref_property_tl } { }
                    \exp_not:V \l__zrefclever_refpos_in_tl
                    \exp_not:N \group_end:
                    % hyperlink makes it's own group, we'd like to close the
                % ‘refpre-out’ group after ‘refpos-out’, but... we close
                % it here, and give the trailing ‘refpos-out’ its own
                % group.  This will result that formatting given to
                % ‘refpre-out’ will not reach ‘refpos-out’, but I see no
                % alternative, and this has to be handled specially.
                \exp_not:N \group_end:
              }
            \exp_not:N \group_begin:
            % Ditto: special treatment.
            \exp_not:V \l__zrefclever_reffont_out_tl
            \exp_not:V \l__zrefclever_refpos_out_tl
            \exp_not:N \group_end:
          }
          {
            \exp_not:N \group_begin:
            \exp_not:V \l__zrefclever_namefont_tl
            \exp_not:V \l__zrefclever_type_name_tl
            \exp_not:N \group_end:
            \exp_not:V \l__zrefclever_namesep_tl
            \__zrefclever_ref_default:
          }
      }
      {
        \tl_if_empty:NTF \l__zrefclever_type_name_tl
          {
            \__zrefclever_name_default:
            \exp_not:V \l__zrefclever_namesep_tl
          }
          {
            \exp_not:N \group_begin:
            \exp_not:V \l__zrefclever_namefont_tl
            \exp_not:V \l__zrefclever_type_name_tl
            \exp_not:N \group_end:
            \exp_not:V \l__zrefclever_namesep_tl
          }
        \zref@ifrefcontainsprop
          { \l__zrefclever_type_first_label_tl }
          { \l__zrefclever_ref_property_tl }
          {
            \bool_if:nTF
              {
                \l__zrefclever_use_hyperref_bool &&
                ! \l__zrefclever_link_star_bool
              }
              {
                \exp_not:N \group_begin:
                \exp_not:V \l__zrefclever_reffont_out_tl
                \exp_not:V \l__zrefclever_refpre_out_tl
                \exp_not:N \group_begin:
                \exp_not:V \l__zrefclever_reffont_in_tl
                % It’s two ’@s’, but escaped for DocStrip.
                \exp_not:N \hyper@@link
                  {
                    \__zrefclever_extract_url:V
```

```
                              \l__zrefclever_type_first_label_tl
                          }
                          {
                            \zref@extractdefault
                              { \l__zrefclever_type_first_label_tl }
                              { anchor } { }
                          }
                          {
                            \exp_not:V \l__zrefclever_refpre_in_tl
                            \zref@extractdefault
                              { \l__zrefclever_type_first_label_tl }
                              { \l__zrefclever_ref_property_tl } { }
                            \exp_not:V \l__zrefclever_refpos_in_tl
                          }
                        \exp_not:N \group_end:
                        \exp_not:V \l__zrefclever_refpos_out_tl
                        \exp_not:N \group_end:
                      }
                      {
                        \exp_not:N \group_begin:
                        \exp_not:V \l__zrefclever_reffont_out_tl
                        \exp_not:V \l__zrefclever_refpre_out_tl
                        \exp_not:N \group_begin:
                        \exp_not:V \l__zrefclever_reffont_in_tl
                        \exp_not:V \l__zrefclever_refpre_in_tl
                        \zref@extractdefault
                          { \l__zrefclever_type_first_label_tl }
                          { \l__zrefclever_ref_property_tl } { }
                        \exp_not:V \l__zrefclever_refpos_in_tl
                        \exp_not:N \group_end:
                        \exp_not:V \l__zrefclever_refpos_out_tl
                        \exp_not:N \group_end:
                      }
                  }
                  { \__zrefclever_ref_default: }
              }
          }
      }
```

(*End definition for* `\__zrefclever_get_ref_first:`.)

`\__zrefclever_type_name_setup:` Auxiliary function to `\__zrefclever_typeset_refs_last_of_type:`. It is responsible for setting the type name variable `\l__zrefclever_type_name_tl` and `\l__zrefclever_name_in_link_bool`. If a type name can't be found, `\l__zrefclever_type_name_tl` is cleared. The function takes no arguments, but is expected to be called in `\__zrefclever_typeset_refs_last_of_type:` right before `\__zrefclever_get_ref_first:`, which is the main consumer of the variables it sets, though not the only one (and hence this cannot be moved into `\__zrefclever_get_ref_first:` itself). It also expects a number of relevant variables to have been appropriately set, and which it uses, prominently `\l__zrefclever_type_first_label_type_tl`, but also the queue itself in `\l__zrefclever_typeset_queue_curr_tl`, which should be "ready except for the first label", and the type counter `\l__zrefclever_type_count_int`.

```
\cs_new_protected:Npn \__zrefclever_type_name_setup:
```

```
{
  \zref@ifrefundefined { \l__zrefclever_type_first_label_tl }
    { \tl_clear:N \l__zrefclever_type_name_tl }
    {
      \tl_if_empty:nTF \l__zrefclever_type_first_label_type_tl
        { \tl_clear:N \l__zrefclever_type_name_tl }
        {
          % Determine whether we should use capitalization, abbreviation,
          % and plural.
          \bool_lazy_or:nnTF
            { \l__zrefclever_capitalize_bool }
            {
              \l__zrefclever_capitalize_first_bool &&
              \int_compare_p:nNn { \l__zrefclever_type_count_int } = { 0 }
            }
            { \tl_set:Nn \l__zrefclever_name_format_tl {Name} }
            { \tl_set:Nn \l__zrefclever_name_format_tl {name} }
          % If the queue is empty, we have a singular, otherwise, plural.
          \tl_if_empty:NTF \l__zrefclever_typeset_queue_curr_tl
            { \tl_put_right:Nn \l__zrefclever_name_format_tl { -sg } }
            { \tl_put_right:Nn \l__zrefclever_name_format_tl { -pl } }
          \bool_lazy_and:nnTF
            { \l__zrefclever_abbrev_bool }
            {
              ! \int_compare_p:nNn
                  { \l__zrefclever_type_count_int } = { 0 } ||
              ! \l__zrefclever_noabbrev_first_bool
            }
            {
              \tl_set:NV \l__zrefclever_name_format_fallback_tl
                \l__zrefclever_name_format_tl
              \tl_put_right:Nn \l__zrefclever_name_format_tl { -ab }
            }
            { \tl_clear:N \l__zrefclever_name_format_fallback_tl }

          \tl_if_empty:NTF \l__zrefclever_name_format_fallback_tl
            {
              \prop_get:cVNF
                {
                  l__zrefclever_type_
                  \l__zrefclever_type_first_label_type_tl _options_prop
                }
                \l__zrefclever_name_format_tl
                \l__zrefclever_type_name_tl
                {
                  \__zrefclever_get_type_transl:xxxNF
                    { \l__zrefclever_ref_language_tl }
                    { \l__zrefclever_type_first_label_type_tl }
                    { \l__zrefclever_name_format_tl }
                    \l__zrefclever_type_name_tl
                    {
                      \tl_clear:N \l__zrefclever_type_name_tl
                      \msg_warning:nnx { zref-clever } { missing-name }
                        { \l__zrefclever_type_first_label_type_tl }
```

```
         }
       }
     }
     {
      \prop_get:cVNF
       {
        l__zrefclever_type_
        \l__zrefclever_type_first_label_type_tl _options_prop
       }
       \l__zrefclever_name_format_tl
       \l__zrefclever_type_name_tl
       {
        \prop_get:cVNF
         {
          l__zrefclever_type_
          \l__zrefclever_type_first_label_type_tl _options_prop
         }
         \l__zrefclever_name_format_fallback_tl
         \l__zrefclever_type_name_tl
         {
          \__zrefclever_get_type_transl:xxxNF
           { \l__zrefclever_ref_language_tl }
           { \l__zrefclever_type_first_label_type_tl }
           { \l__zrefclever_name_format_tl }
           \l__zrefclever_type_name_tl
           {
            \__zrefclever_get_type_transl:xxxNF
             { \l__zrefclever_ref_language_tl }
             { \l__zrefclever_type_first_label_type_tl }
             { \l__zrefclever_name_format_fallback_tl }
             \l__zrefclever_type_name_tl
             {
              \tl_clear:N \l__zrefclever_type_name_tl
              \msg_warning:nnx { zref-clever }
               { missing-name }
               { \l__zrefclever_type_first_label_type_tl }
             }
           }
         }
       }
     }
   }
 }

% Signal whether the type name is to be included in the hyperlink or not.
\bool_lazy_any:nTF
 {
  { ! \l__zrefclever_use_hyperref_bool }
  { \l__zrefclever_link_star_bool }
  { \tl_if_empty_p:N \l__zrefclever_type_name_tl }
  { \str_if_eq_p:Vn \l__zrefclever_nameinlink_str { false } }
 }
 { \bool_set_false:N \l__zrefclever_name_in_link_bool }
 {
        \bool_lazy_any:nTF
          {
            { \str_if_eq_p:Vn \l__zrefclever_nameinlink_str { true } }
            {
              \str_if_eq_p:Vn \l__zrefclever_nameinlink_str { tsingle } &&
              \tl_if_empty_p:N \l__zrefclever_typeset_queue_curr_tl
            }
            {
              \str_if_eq_p:Vn \l__zrefclever_nameinlink_str { single } &&
              \tl_if_empty_p:N \l__zrefclever_typeset_queue_curr_tl &&
              \l__zrefclever_typeset_last_bool &&
              \int_compare_p:nNn { \l__zrefclever_type_count_int } = { 0 }
            }
          }
          { \bool_set_true:N \l__zrefclever_name_in_link_bool }
          { \bool_set_false:N \l__zrefclever_name_in_link_bool }
      }
  }
```

(*End definition for* `\__zrefclever_type_name_setup:`.)

`\__zrefclever_extract_url:n` A convenience auxiliary function for extraction of the `url` / `urluse` property, provided by the zref-xr module.

```
\cs_new:Npn \__zrefclever_extract_url:n #1
  {
    \zref@ifpropundefined { urluse }
      { \zref@extractdefault {#1} { url } { \c_empty_tl } }
      {
        \zref@ifrefcontainsprop {#1} { urluse }
          { \zref@extractdefault {#1} { urluse } { \c_empty_tl } }
          { \zref@extractdefault {#1} { url } { \c_empty_tl } }
      }
  }
\cs_generate_variant:Nn \__zrefclever_extract_url:n { V }
```

(*End definition for* `\__zrefclever_extract_url:n`.)

`\__zrefclever_labels_in_sequence:nn` Auxiliary function to `\__zrefclever_typeset_refs_not_last_of_type:`. Sets `\l__zrefclever_next_maybe_range_bool` to true if ⟨*label b*⟩ comes in immediate sequence from ⟨*label a*⟩. And sets both `\l__zrefclever_next_maybe_range_bool` and `\l__zrefclever_next_is_same_bool` to true if the two labels are the "same" (that is, have the same counter value). These two boolean variables are the basis for all range and compression handling inside `\__zrefclever_typeset_refs_not_last_of_type:`, so this function is expected to be called at its beginning, if compression is enabled.

```
\__zrefclever_labels_in_sequence:nn {⟨label a⟩} {⟨label b⟩}
```

```
\cs_new_protected:Npn \__zrefclever_labels_in_sequence:nn #1#2
  {
    \tl_set:Nx \l__zrefclever_label_extdoc_a_tl
      { \zref@extractdefault {#1} { externaldocument } { \c_empty_tl } }
    \tl_set:Nx \l__zrefclever_label_extdoc_b_tl
      { \zref@extractdefault {#2} { externaldocument } { \c_empty_tl } }

    \tl_if_eq:NNT
```

```
      \l__zrefclever_label_extdoc_a_tl
      \l__zrefclever_label_extdoc_b_tl
      {
        \tl_if_eq:NnTF \l__zrefclever_ref_property_tl { page }
          {
            \exp_args:Nxx \tl_if_eq:nnT
              { \zref@extractdefault {#1} { zc@pgfmt } { } }
              { \zref@extractdefault {#2} { zc@pgfmt } { } }
              {
                \int_compare:nNnTF
                  { \zref@extractdefault {#1} { zc@pgval } { -2 } + 1 }
                    =
                  { \zref@extractdefault {#2} { zc@pgval } { -1 } }
                  { \bool_set_true:N \l__zrefclever_next_maybe_range_bool }
                  {
                    \int_compare:nNnT
                      { \zref@extractdefault {#1} { zc@pgval } { -1 } }
                        =
                      { \zref@extractdefault {#2} { zc@pgval } { -1 } }
                      {
                        \bool_set_true:N
                          \l__zrefclever_next_maybe_range_bool
                        \bool_set_true:N
                          \l__zrefclever_next_is_same_bool
                      }
                  }
              }
          }
          {
            \exp_args:Nxx \tl_if_eq:nnT
              { \zref@extractdefault {#1} { zc@counter } { } }
              { \zref@extractdefault {#2} { zc@counter } { } }
              {
                \exp_args:Nxx \tl_if_eq:nnT
                  { \zref@extractdefault {#1} { zc@enclval } { } }
                  { \zref@extractdefault {#2} { zc@enclval } { } }
                  {
                    \int_compare:nNnTF
                      { \zref@extractdefault {#1} { zc@cntval } { -2 } + 1 }
                        =
                      { \zref@extractdefault {#2} { zc@cntval } { -1 } }
                      { \bool_set_true:N \l__zrefclever_next_maybe_range_bool }
                      {
                        \int_compare:nNnT
                          { \zref@extractdefault {#1} { zc@cntval } { -1 } }
                            =
                          { \zref@extractdefault {#2} { zc@cntval } { -1 } }
                          {
                            \bool_set_true:N
                              \l__zrefclever_next_maybe_range_bool
                            \bool_set_true:N
                              \l__zrefclever_next_is_same_bool
                          }
                      }
                    }
                }
            }
        }
    }
```

(*End definition for* `\__zrefclever_labels_in_sequence:nn`.)

Finally, a couple of functions for retrieving options values, according to the relevant precedence rules. They both receive an ⟨*option*⟩ as argument, and store the retrieved value in ⟨*tl variable*⟩. Though these are mostly general functions (for a change…), they are not completely so, they rely on the current state of `\l__zrefclever_label_type_a_tl`, as set during the processing of the label stack. This could be easily generalized, of course, but I don't think it is worth it, `\l__zrefclever_label_type_a_tl` is indeed what we want in all practical cases. The difference between `\__zrefclever_get_ref_string:nN` and `\__zrefclever_get_ref_font:nN` is the kind of option each should be used for. `\__zrefclever_get_ref_string:nN` is meant for the general options, and attempts to find values for them in all precedence levels (four plus "fallback"). `\__zrefclever_get_ref_font:nN` is intended for "font" options, which cannot be "language-specific", thus for these we just search general options and type options.

__zrefclever_get_ref_string:nN

`\__zrefclever_get_ref_string:nN {`⟨*option*⟩`} {`⟨*tl variable*⟩`}`

```
\cs_new_protected:Npn \__zrefclever_get_ref_string:nN #1#2
  {
    % First attempt: general options.
    \prop_get:NnNF \l__zrefclever_ref_options_prop {#1} #2
      {
        % If not found, try type specific options.
        \bool_lazy_all:nTF
          {
            { ! \tl_if_empty_p:N \l__zrefclever_label_type_a_tl }
            {
              \prop_if_exist_p:c
                {
                  l__zrefclever_type_
                  \l__zrefclever_label_type_a_tl _options_prop
                }
            }
            {
              \prop_if_in_p:cn
                {
                  l__zrefclever_type_
                  \l__zrefclever_label_type_a_tl _options_prop
                }
                {#1}
            }
          }
          {
            \prop_get:cnN
              {
                l__zrefclever_type_
                \l__zrefclever_label_type_a_tl _options_prop
              }
              {#1} #2
          }
          {
            % If not found, try type specific translations.
            \__zrefclever_get_type_transl:xxnNF
              { \l__zrefclever_ref_language_tl }
              { \l__zrefclever_label_type_a_tl }
              {#1} #2
              {
                % If not found, try default translations.
                \__zrefclever_get_default_transl:xnNF
                  { \l__zrefclever_ref_language_tl }
                  {#1} #2
                  {
                    % If not found, try fallback.
                    \__zrefclever_get_fallback_transl:nNF {#1} #2
                      {
                        \tl_clear:N #2
                        \msg_warning:nnn { zref-clever }
                          { missing-string } {#1}
                      }
                  }
              }
          }
      }
  }
```

(*End definition for* `\__zrefclever_get_ref_string:nN`*.*)

__zrefclever_get_ref_font:nN

`\__zrefclever_get_ref_font:nN {⟨option⟩} {⟨tl variable⟩}`

```
\cs_new_protected:Npn \__zrefclever_get_ref_font:nN #1#2
  {
    % First attempt: general options.
    \prop_get:NnNF \l__zrefclever_ref_options_prop {#1} #2
      {
        % If not found, try type specific options.
        \bool_lazy_and:nnTF
          { ! \tl_if_empty_p:N \l__zrefclever_label_type_a_tl }
          {
            \prop_if_exist_p:c
              {
                l__zrefclever_type_
                \l__zrefclever_label_type_a_tl _options_prop
              }
          }
          {
            \prop_get:cnNF
              {
                l__zrefclever_type_
                \l__zrefclever_label_type_a_tl _options_prop
              }
              {#1} #2
              { \tl_clear:N #2 }
          }
          { \tl_clear:N #2 }
      }
  }
```

(*End definition for* `\__zrefclever_get_ref_font:nN`.)

9 Compatibility

This section is meant to aggregate any "special handling" needed for LaTeX kernel features, document classes, and packages, needed for zref-clever to work properly with them. It is not meant to be a "kitchen sink of workarounds". Rather, I intend to keep this as lean as possible, trying to add things selectively when they are safe and reasonable. And, hopefully, doing so by proper setting of zref-clever's options, not by messing with other packages' code. In particular, I do not mean to compensate for "lack of support for zref" by individual packages here, unless there is really no alternative.

9.1 `\appendix`

One relevant case of different reference types sharing the same counter is the `\appendix` which in some document classes, including the standard ones, change the sectioning commands looks but, of course, keep using the same counter. `book.cls` and `report.cls` reset counters `chapter` and `section` to 0, change `\@chapapp` to use `\appendixname` and use `\@Alph` for `\thechapter`. `article.cls` resets counters `section` and `subsection` to 0, and uses `\@Alph` for `\thesection`. `memoir.cls`, `scrbook.cls` and `scrarticle.cls` do the same as their corresponding standard classes, and sometimes a little more, but what interests us here is pretty much the same. See also the appendix package.

The standard `\appendix` command is a one way switch, in other words, it cannot be reverted (see https://tex.stackexchange.com/a/444057). So, even if the fact that it is a "switch" rather than an environment complicates things, because we have to make ungrouped settings to correspond to its effects, in practice this is not a big deal, since these settings are never really reverted (by default, at least). Hence, hooking into `\appendix` is a viable and natural alternative. The memoir class and the appendix package define the `appendices` and `subappendices` environments, which provide for a way for the appendix to "end", but in this case, of course, we can hook into the environment instead.

```
\AddToHook { cmd / appendix / before }
  {
    \zcsetup
      {
        countertype =
          {
            chapter       = appendix ,
            section       = appendix ,
            subsection    = appendix ,
            subsubsection = appendix ,
          }
      }
  }
```

Depending on the definition of `\appendix`, using the hook may lead to trouble with the first released version of ltcmdhooks (the one released with the 2021-06-01 kernel). Particularly, if the definition of the command being hooked at contains a double hash mark (`##`) the patch to add the hook, if it needs to be done with the `\scantokens` method, may fail noisily (see https://tex.stackexchange.com/q/617905, thanks Phelype Oleinik). The 2021-11-15 kernel release should already handle this gracefully. In the meantime, given we cannot really expect to know what `\appendix` may contain in general, since it potentially gets redefined in quite a number of classes and packages, a user facing workaround may be needed in case of trouble. Phelype Oleinik recommends activating/providing the generic hook in question, so that ltcmdhooks considers the patch as already done, and do the patch ourselves with etoolbox (https://tex.stackexchange.com/a/617998). Like so:

```
\IfFormatAtLeastTF{2021-11-15}%
  {\ActivateGenericHook}%
  {\ProvideHook}%
    {cmd/appendix/before}
\usepackage{etoolbox}
\pretocmd\appendix
  {\UseHook{cmd/appendix/before}}
  {}{\FAILED}
```

9.2 appendix package

These settings also apply to the memoir class, since it "emulates" the loading of the appendix package.

```
\AddToHook { begindocument }
  {
    \@ifpackageloaded { appendix }
      {
        \AddToHook { env / appendices / begin }
          {
            \zcsetup
              {
                countertype =
                  {
                    chapter       = appendix ,
                    section       = appendix ,
                    subsection    = appendix ,
                    subsubsection = appendix ,
                  }
              }
          }
        \AddToHook { env / subappendices / begin }
          {
            \zcsetup
              {
                countertype =
                  {
                    chapter       = subappendix ,
                    section       = subappendix ,
                    subsection    = subappendix ,
                    subsubsection = subappendix ,
                  }
              }
          }
        \msg_info:nnn { zref-clever } { compat-package } { appendix }
      }
      {}
  }
```

9.3 listings package

```
\AddToHook { begindocument }
  {
    \@ifpackageloaded { listings }
      {
        \zcsetup
          {
            countertype =
              {
                lstlisting = listing ,
                lstnumber = line ,
              } ,
            counterresetby = { lstnumber = lstlisting } ,
          }
        \lst@AddToHook { Init }
          {
```

Set (also) a `\zlabel` with the label received in the `label=` option from the `lstlisting` environment.

```
            \tl_if_empty:NF \lst@label
              { \zlabel { \lst@label } }
```

The correct place to set `currentcounter` to `lstnumber` is indeed the `Init` hook, since listings itself sets `\@currentlabel` to `\thelstnumber` in the same hook. See section "Line numbers" of '`texdoc listings-devel`' (the `.dtx`), and search for the definition of macro `\c@lstnumber`. Note that listings *does use* `\refstepcounter{lstnumber}`, but does so in the `EveryPar` hook, and there must be some grouping involved such that `\@currentcounter` ends up not being visible to the label. Indeed, the fact that listings manually sets `\@currentlabel` to `\thelstnumber` is a signal that the work of `\refstepcounter` is being restrained somehow.

```
            \zcsetup { currentcounter = lstnumber }
          }
        \msg_info:nnn { zref-clever } { compat-package } { listings }
      }
      {}
  }
```

9.4 enumitem package

The procedure below will "see" any changes made to the `enumerate` environment (made with enumitem's `\renewlist`) as long as it is done in the preamble. Though, technically, `\renewlist` can be issued anywhere in the document, this should be more than enough for the purpose at hand. Besides, trying to retrieve this information "on the fly" would be much overkill.

The only real reason to "renew" `enumerate` itself is to change `{`⟨*max-depth*⟩`}`. `\renewlist` *hard-codes* `max-depth` in the environment's definition (well, just as the kernel does), so we cannot retrieve this information from any sort of variable. But `\renewlist` also creates any needed missing counters, so we can use their existence to make the appropriate settings. In the end, the existence of the counters is indeed what matters from zref-clever's perspective. Since the first four are defined by the kernel and already setup for zref-clever by default, we start from 5, and stop at the first non-existent `\c@enumN` counter.

```
\AddToHook { begindocument }
  {
    \@ifpackageloaded { enumitem }
      {
        \int_set:Nn \l_tmpa_int { 5 }
        \bool_while_do:nn
          {
            \cs_if_exist_p:c
              { c@ enum \int_to_roman:n { \l_tmpa_int } }
          }
          {
            \exp_args:Nx \zcsetup
              {
                counterresetby =
                  {
                    enum \int_to_roman:n { \l_tmpa_int } =
                    enum \int_to_roman:n { \l_tmpa_int - 1 }
                  } ,
                countertype =
                  { enum \int_to_roman:n { \l_tmpa_int } = item } ,
              }
            \int_incr:N \l_tmpa_int
          }
        \int_compare:nNnT { \l_tmpa_int } > { 5 }
          { \msg_info:nnn { zref-clever } { compat-package } { enumitem } }
      }
      {}
  }
⟨/package⟩
```

10 Dictionaries

10.1 English

```
⟨package⟩\zcDeclareLanguage { english }
⟨package⟩\zcDeclareLanguageAlias { american   } { english }
⟨package⟩\zcDeclareLanguageAlias { australian } { english }
⟨package⟩\zcDeclareLanguageAlias { british    } { english }
⟨package⟩\zcDeclareLanguageAlias { canadian   } { english }
⟨package⟩\zcDeclareLanguageAlias { newzealand } { english }
⟨package⟩\zcDeclareLanguageAlias { UKenglish  } { english }
⟨package⟩\zcDeclareLanguageAlias { USenglish  } { english }
⟨*dict-english⟩
namesep   = {\nobreakspace} ,
pairsep   = {~and\nobreakspace} ,
listsep   = {,~} ,
lastsep   = {~and\nobreakspace} ,
tpairsep  = {~and\nobreakspace} ,
tlistsep  = {,~} ,
tlastsep  = {,~and\nobreakspace} ,
notesep   = {~} ,
rangesep  = {~to\nobreakspace} ,

type = part ,
  Name-sg = Part ,
  name-sg = part ,
  Name-pl = Parts ,
  name-pl = parts ,

type = chapter ,
  Name-sg = Chapter ,
  name-sg = chapter ,
  Name-pl = Chapters ,
  name-pl = chapters ,

type = section ,
  Name-sg = Section ,
  name-sg = section ,
  Name-pl = Sections ,
  name-pl = sections ,

type = paragraph ,
  Name-sg = Paragraph ,
  name-sg = paragraph ,
  Name-pl = Paragraphs ,
  name-pl = paragraphs ,
  Name-sg-ab = Par. ,
  name-sg-ab = par. ,
  Name-pl-ab = Par. ,
  name-pl-ab = par. ,

type = appendix ,
  Name-sg = Appendix ,
  name-sg = appendix ,
  Name-pl = Appendices ,
  name-pl = appendices ,

type = subappendix ,
  Name-sg = Appendix ,
  name-sg = appendix ,
  Name-pl = Appendices ,
  name-pl = appendices ,

type = page ,
  Name-sg = Page ,
  name-sg = page ,
  Name-pl = Pages ,
  name-pl = pages ,
  name-sg-ab = p. ,
  name-pl-ab = pp. ,

type = line ,
  Name-sg = Line ,
  name-sg = line ,
  Name-pl = Lines ,
  name-pl = lines ,

type = figure ,
  Name-sg = Figure ,
  name-sg = figure ,
  Name-pl = Figures ,
  name-pl = figures ,
  Name-sg-ab = Fig. ,
  name-sg-ab = fig. ,
  Name-pl-ab = Figs. ,
  name-pl-ab = figs. ,

type = table ,
  Name-sg = Table ,
  name-sg = table ,
  Name-pl = Tables ,
  name-pl = tables ,

type = item ,
  Name-sg = Item ,
  name-sg = item ,
  Name-pl = Items ,
  name-pl = items ,

type = footnote ,
  Name-sg = Footnote ,
  name-sg = footnote ,
  Name-pl = Footnotes ,
  name-pl = footnotes ,

type = note ,
  Name-sg = Note ,
  name-sg = note ,
  Name-pl = Notes ,
  name-pl = notes ,

type = equation ,
  Name-sg = Equation ,
  name-sg = equation ,
  Name-pl = Equations ,
  name-pl = equations ,
  Name-sg-ab = Eq. ,
  name-sg-ab = eq. ,
  Name-pl-ab = Eqs. ,
  name-pl-ab = eqs. ,
  refpre-in = {(} ,
  refpos-in = {)} ,

type = theorem ,
  Name-sg = Theorem ,
  name-sg = theorem ,
  Name-pl = Theorems ,
  name-pl = theorems ,

type = lemma ,
  Name-sg = Lemma ,
  name-sg = lemma ,
  Name-pl = Lemmas ,
  name-pl = lemmas ,

type = corollary ,
  Name-sg = Corollary ,
  name-sg = corollary ,
  Name-pl = Corollaries ,
  name-pl = corollaries ,

type = proposition ,
  Name-sg = Proposition ,
  name-sg = proposition ,
  Name-pl = Propositions ,
  name-pl = propositions ,

type = definition ,
  Name-sg = Definition ,
  name-sg = definition ,
  Name-pl = Definitions ,
  name-pl = definitions ,

type = proof ,
  Name-sg = Proof ,
  name-sg = proof ,
  Name-pl = Proofs ,
  name-pl = proofs ,

type = result ,
  Name-sg = Result ,
  name-sg = result ,
  Name-pl = Results ,
  name-pl = results ,

type = remark ,
  Name-sg = Remark ,
  name-sg = remark ,
  Name-pl = Remarks ,
  name-pl = remarks ,

type = example ,
  Name-sg = Example ,
  name-sg = example ,
  Name-pl = Examples ,
  name-pl = examples ,

type = algorithm ,
  Name-sg = Algorithm ,
  name-sg = algorithm ,
  Name-pl = Algorithms ,
  name-pl = algorithms ,

type = listing ,
  Name-sg = Listing ,
  name-sg = listing ,
  Name-pl = Listings ,
  name-pl = listings ,

type = exercise ,
  Name-sg = Exercise ,
  name-sg = exercise ,
  Name-pl = Exercises ,
  name-pl = exercises ,

type = solution ,
  Name-sg = Solution ,
  name-sg = solution ,
  Name-pl = Solutions ,
  name-pl = solutions ,
⟨/dict-english⟩
```

10.2 German

```
⟨package⟩\zcDeclareLanguage { german }
⟨package⟩\zcDeclareLanguageAlias { austrian     } { german }
⟨package⟩\zcDeclareLanguageAlias { germanb      } { german }
⟨package⟩\zcDeclareLanguageAlias { ngerman      } { german }
⟨package⟩\zcDeclareLanguageAlias { naustrian    } { german }
⟨package⟩\zcDeclareLanguageAlias { nswissgerman } { german }
⟨package⟩\zcDeclareLanguageAlias { swissgerman  } { german }
⟨*dict-german⟩
namesep  = {\nobreakspace} ,
pairsep  = {~und\nobreakspace} ,
listsep  = {,~} ,
lastsep  = {~und\nobreakspace} ,
tpairsep = {~und\nobreakspace} ,
tlistsep = {,~} ,
tlastsep = {~und\nobreakspace} ,
notesep  = {~} ,
rangesep = {~bis\nobreakspace} ,

type = part ,
  Name-sg = Teil ,
  name-sg = Teil ,
  Name-pl = Teile ,
  name-pl = Teile ,

type = chapter ,
  Name-sg = Kapitel ,
  name-sg = Kapitel ,
  Name-pl = Kapitel ,
  name-pl = Kapitel ,

type = section ,
  Name-sg = Abschnitt ,
  name-sg = Abschnitt ,
  Name-pl = Abschnitte ,
  name-pl = Abschnitte ,

type = paragraph ,
  Name-sg = Absatz ,
  name-sg = Absatz ,
  Name-pl = Absätze ,
  name-pl = Absätze ,

type = appendix ,
  Name-sg = Anhang ,
  name-sg = Anhang ,
  Name-pl = Anhänge ,
  name-pl = Anhänge ,

type = subappendix ,
  Name-sg = Anhang ,
  name-sg = Anhang ,
  Name-pl = Anhänge ,
  name-pl = Anhänge ,

type = page ,
  Name-sg = Seite ,
  name-sg = Seite ,
  Name-pl = Seiten ,
  name-pl = Seiten ,

type = line ,
  Name-sg = Zeile ,
  name-sg = Zeile ,
  Name-pl = Zeilen ,
  name-pl = Zeilen ,

type = figure ,
  Name-sg = Abbildung ,
  name-sg = Abbildung ,
  Name-pl = Abbildungen ,
  name-pl = Abbildungen ,
  Name-sg-ab = Abb. ,
  name-sg-ab = Abb. ,
  Name-pl-ab = Abb. ,
  name-pl-ab = Abb. ,

type = table ,
  Name-sg = Tabelle ,
  name-sg = Tabelle ,
  Name-pl = Tabellen ,
  name-pl = Tabellen ,

type = item ,
  Name-sg = Punkt ,
  name-sg = Punkt ,
  Name-pl = Punkte ,
  name-pl = Punkte ,

type = footnote ,
  Name-sg = Fußnote ,
  name-sg = Fußnote ,
  Name-pl = Fußnoten ,
  name-pl = Fußnoten ,

type = note ,
  Name-sg = Anmerkung ,
  name-sg = Anmerkung ,
  Name-pl = Anmerkungen ,
  name-pl = Anmerkungen ,

type = equation ,
  Name-sg = Gleichung ,
  name-sg = Gleichung ,
  Name-pl = Gleichungen ,
  name-pl = Gleichungen ,
  refpre-in = {(} ,
  refpos-in = {)} ,

type = theorem ,
  Name-sg = Theorem ,
  name-sg = Theorem ,
  Name-pl = Theoreme ,
  name-pl = Theoreme ,

type = lemma ,
  Name-sg = Lemma ,
  name-sg = Lemma ,
  Name-pl = Lemmata ,
  name-pl = Lemmata ,

type = corollary ,
  Name-sg = Korollar ,
  name-sg = Korollar ,
  Name-pl = Korollare ,
  name-pl = Korollare ,

type = proposition ,
  Name-sg = Satz ,
  name-sg = Satz ,
  Name-pl = Sätze ,
  name-pl = Sätze ,

type = definition ,
  Name-sg = Definition ,
  name-sg = Definition ,
  Name-pl = Definitionen ,
  name-pl = Definitionen ,

type = proof ,
  Name-sg = Beweis ,
  name-sg = Beweis ,
  Name-pl = Beweise ,
  name-pl = Beweise ,

type = result ,
  Name-sg = Ergebnis ,
  name-sg = Ergebnis ,
  Name-pl = Ergebnisse ,
  name-pl = Ergebnisse ,

type = remark ,
  Name-sg = Bemerkung ,
  name-sg = Bemerkung ,
  Name-pl = Bemerkungen ,
  name-pl = Bemerkungen ,

type = example ,
  Name-sg = Beispiel ,
  name-sg = Beispiel ,
  Name-pl = Beispiele ,
  name-pl = Beispiele ,

type = algorithm ,
  Name-sg = Algorithmus ,
  name-sg = Algorithmus ,
  Name-pl = Algorithmen ,
  name-pl = Algorithmen ,

type = listing ,
  Name-sg = Listing ,
  name-sg = Listing ,
  Name-pl = Listings ,
  name-pl = Listings ,

type = exercise ,
  Name-sg = Übungsaufgabe ,
  name-sg = Übungsaufgabe ,
  Name-pl = Übungsaufgaben ,
  name-pl = Übungsaufgaben ,

type = solution ,
  Name-sg = Lösung ,
  name-sg = Lösung ,
  Name-pl = Lösungen ,
  name-pl = Lösungen ,
⟨/dict-german⟩
```

10.3 French

```
⟨package⟩\zcDeclareLanguage { french }
⟨package⟩\zcDeclareLanguageAlias { acadian  } { french }
```

```
⟨package⟩\zcDeclareLanguageAlias { canadien } { french }
⟨package⟩\zcDeclareLanguageAlias { francais } { french }
⟨package⟩\zcDeclareLanguageAlias { frenchb  } { french }
⟨*dict-french⟩
namesep  = {\nobreakspace} ,
pairsep  = {~et\nobreakspace} ,
listsep  = {,~} ,
lastsep  = {~et\nobreakspace} ,
tpairsep = {~et\nobreakspace} ,
tlistsep = {,~} ,
tlastsep = {~et\nobreakspace} ,
notesep  = {~} ,
rangesep = {~à\nobreakspace} ,

type = part ,
  Name-sg = Partie ,
  name-sg = partie ,
  Name-pl = Parties ,
  name-pl = parties ,

type = chapter ,
  Name-sg = Chapitre ,
  name-sg = chapitre ,
  Name-pl = Chapitres ,
  name-pl = chapitres ,

type = section ,
  Name-sg = Section ,
  name-sg = section ,
  Name-pl = Sections ,
  name-pl = sections ,

type = paragraph ,
  Name-sg = Paragraphe ,
  name-sg = paragraphe ,
  Name-pl = Paragraphes ,
  name-pl = paragraphes ,

type = appendix ,
  Name-sg = Annexe ,
  name-sg = annexe ,
  Name-pl = Annexes ,
  name-pl = annexes ,

type = subappendix ,
  Name-sg = Annexe ,
  name-sg = annexe ,
  Name-pl = Annexes ,
  name-pl = annexes ,

type = page ,
  Name-sg = Page ,
  name-sg = page ,
  Name-pl = Pages ,
  name-pl = pages ,

type = line ,
  Name-sg = Ligne ,
  name-sg = ligne ,
  Name-pl = Lignes ,
  name-pl = lignes ,

type = figure ,
  Name-sg = Figure ,
  name-sg = figure ,
  Name-pl = Figures ,
  name-pl = figures ,

type = table ,
  Name-sg = Table ,
  name-sg = table ,
  Name-pl = Tables ,
  name-pl = tables ,

type = item ,
  Name-sg = Point ,
  name-sg = point ,
  Name-pl = Points ,
  name-pl = points ,

type = footnote ,
  Name-sg = Note ,
  name-sg = note ,
  Name-pl = Notes ,
  name-pl = notes ,

type = note ,
  Name-sg = Note ,
  name-sg = note ,
  Name-pl = Notes ,
  name-pl = notes ,

type = equation ,
  Name-sg = Équation ,
  name-sg = équation ,
  Name-pl = Équations ,
  name-pl = équations ,
  refpre-in = {(} ,
  refpos-in = {)} ,

type = theorem ,
  Name-sg = Théorème ,
  name-sg = théorème ,
  Name-pl = Théorèmes ,
  name-pl = théorèmes ,

type = lemma ,
  Name-sg = Lemme ,
  name-sg = lemme ,
  Name-pl = Lemmes ,
  name-pl = lemmes ,

type = corollary ,
  Name-sg = Corollaire ,
  name-sg = corollaire ,
  Name-pl = Corollaires ,
  name-pl = corollaires ,

type = proposition ,
  Name-sg = Proposition ,
  name-sg = proposition ,
  Name-pl = Propositions ,
  name-pl = propositions ,

type = definition ,
  Name-sg = Définition ,
  name-sg = définition ,
  Name-pl = Définitions ,
  name-pl = définitions ,

type = proof ,
  Name-sg = Démonstration ,
  name-sg = démonstration ,
  Name-pl = Démonstrations ,
  name-pl = démonstrations ,

type = result ,
  Name-sg = Résultat ,
  name-sg = résultat ,
  Name-pl = Résultats ,
  name-pl = résultats ,

type = remark ,
  Name-sg = Remarque ,
  name-sg = remarque ,
  Name-pl = Remarques ,
  name-pl = remarques ,

type = example ,
  Name-sg = Exemple ,
  name-sg = exemple ,
  Name-pl = Exemples ,
  name-pl = exemples ,

type = algorithm ,
  Name-sg = Algorithme ,
  name-sg = algorithme ,
  Name-pl = Algorithmes ,
  name-pl = algorithmes ,

type = listing ,
  Name-sg = Liste ,
  name-sg = liste ,
  Name-pl = Listes ,
  name-pl = listes ,

type = exercise ,
  Name-sg = Exercice ,
  name-sg = exercice ,
  Name-pl = Exercices ,
  name-pl = exercices ,

type = solution ,
  Name-sg = Solution ,
  name-sg = solution ,
  Name-pl = Solutions ,
  name-pl = solutions ,
⟨/dict-french⟩
```

10.4 Portuguese

```
⟨package⟩\zcDeclareLanguage { portuguese }
⟨package⟩\zcDeclareLanguageAlias { brazilian } { portuguese }
⟨package⟩\zcDeclareLanguageAlias { brazil    } { portuguese }
⟨package⟩\zcDeclareLanguageAlias { portuges  } { portuguese }
⟨*dict-portuguese⟩
namesep  = {\nobreakspace} ,
pairsep  = {~e\nobreakspace} ,
listsep  = {,~} ,
lastsep  = {~e\nobreakspace} ,
tpairsep = {~e\nobreakspace} ,
tlistsep = {,~} ,
tlastsep = {~e\nobreakspace} ,
notesep  = {~} ,
rangesep = {~a\nobreakspace} ,

type = part ,
  Name-sg = Parte ,
  name-sg = parte ,
  Name-pl = Partes ,
  name-pl = partes ,

type = chapter ,
  Name-sg = Capítulo ,
  name-sg = capítulo ,
  Name-pl = Capítulos ,
  name-pl = capítulos ,

type = section ,
  Name-sg = Seção ,
  name-sg = seção ,
  Name-pl = Seções ,
  name-pl = seções ,

type = paragraph ,
  Name-sg = Parágrafo ,
  name-sg = parágrafo ,
  Name-pl = Parágrafos ,
  name-pl = parágrafos ,
  Name-sg-ab = Par. ,
  name-sg-ab = par. ,
  Name-pl-ab = Par. ,
  name-pl-ab = par. ,

type = appendix ,
  Name-sg = Apêndice ,
  name-sg = apêndice ,
  Name-pl = Apêndices ,
  name-pl = apêndices ,

type = subappendix ,
  Name-sg = Apêndice ,
  name-sg = apêndice ,
  Name-pl = Apêndices ,
  name-pl = apêndices ,

type = page ,
  Name-sg = Página ,
  name-sg = página ,
  Name-pl = Páginas ,
  name-pl = páginas ,
  name-sg-ab = p. ,
  name-pl-ab = pp. ,

type = line ,
  Name-sg = Linha ,
  name-sg = linha ,
  Name-pl = Linhas ,
  name-pl = linhas ,

type = figure ,
  Name-sg = Figura ,
  name-sg = figura ,
  Name-pl = Figuras ,
  name-pl = figuras ,
  Name-sg-ab = Fig. ,
  name-sg-ab = fig. ,
  Name-pl-ab = Figs. ,
  name-pl-ab = figs. ,

type = table ,
  Name-sg = Tabela ,
  name-sg = tabela ,
  Name-pl = Tabelas ,
  name-pl = tabelas ,

type = item ,
  Name-sg = Item ,
  name-sg = item ,
  Name-pl = Itens ,
  name-pl = itens ,

type = footnote ,
  Name-sg = Nota ,
  name-sg = nota ,
  Name-pl = Notas ,
  name-pl = notas ,

type = note ,
  Name-sg = Nota ,
  name-sg = nota ,
  Name-pl = Notas ,
  name-pl = notas ,

type = equation ,
  Name-sg = Equação ,
  name-sg = equação ,
  Name-pl = Equações ,
  name-pl = equações ,
  Name-sg-ab = Eq. ,
  name-sg-ab = eq. ,
  Name-pl-ab = Eqs. ,
  name-pl-ab = eqs. ,
  refpre-in = {(} ,
  refpos-in = {)} ,

type = theorem ,
  Name-sg = Teorema ,
  name-sg = teorema ,
  Name-pl = Teoremas ,
  name-pl = teoremas ,

type = lemma ,
  Name-sg = Lema ,
  name-sg = lema ,
  Name-pl = Lemas ,
  name-pl = lemas ,

type = corollary ,
  Name-sg = Corolário ,
  name-sg = corolário ,
  Name-pl = Corolários ,
  name-pl = corolários ,

type = proposition ,
  Name-sg = Proposição ,
  name-sg = proposição ,
  Name-pl = Proposições ,
  name-pl = proposições ,

type = definition ,
  Name-sg = Definição ,
  name-sg = definição ,
  Name-pl = Definições ,
  name-pl = definições ,

type = proof ,
  Name-sg = Demonstração ,
  name-sg = demonstração ,
  Name-pl = Demonstrações ,
  name-pl = demonstrações ,

type = result ,
  Name-sg = Resultado ,
  name-sg = resultado ,
  Name-pl = Resultados ,
  name-pl = resultados ,

type = remark ,
  Name-sg = Observação ,
  name-sg = observação ,
  Name-pl = Observações ,
  name-pl = observações ,

type = example ,
  Name-sg = Exemplo ,
  name-sg = exemplo ,
  Name-pl = Exemplos ,
  name-pl = exemplos ,

type = algorithm ,
  Name-sg = Algoritmo ,
  name-sg = algoritmo ,
  Name-pl = Algoritmos ,
  name-pl = algoritmos ,

type = listing ,
  Name-sg = Listagem ,
  name-sg = listagem ,
  Name-pl = Listagens ,
  name-pl = listagens ,

type = exercise ,
  Name-sg = Exercício ,
  name-sg = exercício ,
  Name-pl = Exercícios ,
  name-pl = exercícios ,

type = solution ,
  Name-sg = Solução ,
  name-sg = solução ,
  Name-pl = Soluções ,
  name-pl = soluções ,
```
⟨/dict-portuguese⟩

10.5 Spanish

```
⟨package⟩\zcDeclareLanguage { spanish }
⟨*dict-spanish⟩
namesep  = {\nobreakspace} ,
pairsep  = {~y\nobreakspace} ,
listsep  = {,~} ,
lastsep  = {~y\nobreakspace} ,
tpairsep = {~y\nobreakspace} ,
tlistsep = {,~} ,
tlastsep = {~y\nobreakspace} ,
notesep  = {~} ,
rangesep = {~a\nobreakspace} ,

type = part ,
  Name-sg = Parte ,
  name-sg = parte ,
  Name-pl = Partes ,
  name-pl = partes ,

type = chapter ,
  Name-sg = Capítulo ,
  name-sg = capítulo ,
  Name-pl = Capítulos ,
  name-pl = capítulos ,

type = section ,
  Name-sg = Sección ,
  name-sg = sección ,
  Name-pl = Secciones ,
  name-pl = secciones ,

type = paragraph ,
  Name-sg = Párrafo ,
  name-sg = párrafo ,
  Name-pl = Párrafos ,
  name-pl = párrafos ,

type = appendix ,
  Name-sg = Apéndice ,
  name-sg = apéndice ,
  Name-pl = Apéndices ,
  name-pl = apéndices ,

type = subappendix ,
  Name-sg = Apéndice ,
  name-sg = apéndice ,
  Name-pl = Apéndices ,
  name-pl = apéndices ,

type = page ,
  Name-sg = Página ,
  name-sg = página ,
  Name-pl = Páginas ,
  name-pl = páginas ,

type = line ,
  Name-sg = Línea ,
  name-sg = línea ,
  Name-pl = Líneas ,
  name-pl = líneas ,

type = figure ,
  Name-sg = Figura ,
  name-sg = figura ,
  Name-pl = Figuras ,
  name-pl = figuras ,

type = table ,
  Name-sg = Cuadro ,
  name-sg = cuadro ,
  Name-pl = Cuadros ,
  name-pl = cuadros ,

type = item ,
  Name-sg = Punto ,
  name-sg = punto ,
  Name-pl = Puntos ,
  name-pl = puntos ,

type = footnote ,
  Name-sg = Nota ,
  name-sg = nota ,
  Name-pl = Notas ,
  name-pl = notas ,

type = note ,
  Name-sg = Nota ,
  name-sg = nota ,
  Name-pl = Notas ,
  name-pl = notas ,

type = equation ,
  Name-sg = Ecuación ,
  name-sg = ecuación ,
  Name-pl = Ecuaciones ,
  name-pl = ecuaciones ,
  refpre-in = {(} ,
  refpos-in = {)} ,

type = theorem ,
  Name-sg = Teorema ,
  name-sg = teorema ,
  Name-pl = Teoremas ,
  name-pl = teoremas ,

type = lemma ,
  Name-sg = Lema ,
  name-sg = lema ,
  Name-pl = Lemas ,
  name-pl = lemas ,

type = corollary ,
  Name-sg = Corolario ,
  name-sg = corolario ,
  Name-pl = Corolarios ,
  name-pl = corolarios ,

type = proposition ,
  Name-sg = Proposición ,
  name-sg = proposición ,
  Name-pl = Proposiciones ,
  name-pl = proposiciones ,

type = definition ,
  Name-sg = Definición ,
  name-sg = definición ,
  Name-pl = Definiciones ,
  name-pl = definiciones ,

type = proof ,
  Name-sg = Demostración ,
  name-sg = demostración ,
  Name-pl = Demostraciones ,
  name-pl = demostraciones ,

type = result ,
  Name-sg = Resultado ,
  name-sg = resultado ,
  Name-pl = Resultados ,
  name-pl = resultados ,

type = remark ,
  Name-sg = Observación ,
  name-sg = observación ,
  Name-pl = Observaciones ,
  name-pl = observaciones ,

type = example ,
  Name-sg = Ejemplo ,
  name-sg = ejemplo ,
  Name-pl = Ejemplos ,
  name-pl = ejemplos ,

type = algorithm ,
  Name-sg = Algoritmo ,
  name-sg = algoritmo ,
  Name-pl = Algoritmos ,
  name-pl = algoritmos ,

type = listing ,
  Name-sg = Listado ,
  name-sg = listado ,
  Name-pl = Listados ,
  name-pl = listados ,

type = exercise ,
  Name-sg = Ejercicio ,
  name-sg = ejercicio ,
  Name-pl = Ejercicios ,
  name-pl = ejercicios ,

type = solution ,
  Name-sg = Solución ,
  name-sg = solución ,
  Name-pl = Soluciones ,
  name-pl = soluciones ,
⟨/dict-spanish⟩
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.