

The zref-clever package^{*}

Gustavo Barros[†]

2021-09-29

Abstract

`zref-clever` provides an user interface for making L^AT_EX cross-references which automates some of their typical features, thus easing their input in the document and improving the consistency of typeset results. A reference made with `\zceref` includes a “name” according to its “type” and lists of multiple labels can be automatically sorted and compressed into ranges when due. The reference format is highly and easily customizable, both globally and locally. `zref-clever` is based on `zref`’s extensible referencing system.

Contents

1	Introduction	2
2	Loading the package	2
3	Dependencies	2
4	User interface	2
5	Options	3
6	Reference types	7
7	Reference format	8
8	Internationalization	12
9	How tos	17
	9.1 <code>\newtheorem</code>	17
	9.2 <code>enumitem</code>	20
	9.3 <code>zref-xr</code>	21
10	Limitations	21
11	Acknowledgments	23
12	Change history	23

^{*}This file describes v0.1.0-alpha, released 2021-09-29.

[†]<https://github.com/gusbrs/zref-clever>

1 Introduction

2 Loading the package

As usual:

```
\usepackage[⟨options⟩]{zref-clever}
```

3 Dependencies

zref-clever requires zref, and L^AT_EX kernel 2021-06-01, or newer. It also needs l3keys2e and ifdraft. Some packages are leveraged by zref-clever if they are present, but are not loaded by default or required by it, namely: hyperref, zref-check, and zref's modules zref-titleref and zref-xr.

4 User interface

<code>\zcref</code>	<code>\zcref* [⟨options⟩] {⟨labels⟩}</code>
---------------------	---------------------------------------------

Typesets references to *⟨labels⟩*, given as a comma separated list. When hyperref support is enabled, references will be hyperlinked to their respective anchors, according to options. The starred version of the command does the same as the plain one, just does not form links. The *⟨options⟩* are (mostly) the same as those of the package, and can be given to local effect. The *⟨labels⟩* argument is protected by zref's `\zref@wrapper@babel`, so that it enjoys the same support for babel's active characters as zref itself does.

<code>\zcpageref</code>	<code>\zcpageref* [⟨options⟩] {⟨labels⟩}</code>
-------------------------	-------------------------------------------------

Typesets page references to *⟨labels⟩*, given as a comma separated list. It is equivalent to calling `\zcref` with the `ref=page` option: `\zcref* [⟨options⟩, ref=page] {⟨labels⟩}`.

<code>\zcsetup</code>	<code>\zcsetup {⟨options⟩}</code>
-----------------------	-----------------------------------

Sets zref-clever's general options (see Sections 5 and 7). The settings performed by `\zcsetup` are local, within the current group. But, of course, it can also be used to global effects if ungrouped, e.g. in the preamble.

<code>\zcRefTypeSetup</code>	<code>\zcRefTypeSetup {⟨type⟩} {⟨options⟩}</code>
------------------------------	---------------------------------------------------

Sets type-specific reference format options (see Section 7). Just as for `\zcsetup`, the settings performed by `\zcRefTypeSetup` are local, within the current group.

Besides these, user facing commands related to Internationalization are presented in Section 8. Note still that all user commands are defined with `\NewDocumentCommand`, which translates into the usual handling of arguments by it and/or processing by l3keys, particularly with regard to brace-stripping and space-trimming.

Furthermore, zref-clever loads zref's zref-user module by default. So you also have its user commands available out-of-the-box, notably `\zlabel`, but also `\zref` and `\zpageref` themselves.

5 Options

`zref-clever` is highly configurable, offering a lot of flexibility in typeset results of the references, but it also tries to keep these “handles” as convenient and user friendly as possible. To this end, most of what one can do with `zref-clever` (pretty much all of it), can be achieved directly through the standard and familiar “comma separated list of `key=value` options”.

There are two main groups of options in `zref-clever`: “general options”, which affect the overall behavior of the package, or the reference as a whole; and “reference format options”, which control the detail of reference formatting, including type-specific and language-specific settings.

This section covers the first group (for the second one, see Section 7). General options can be set globally either as package options at load-time (see Section 2) or by means of `\zcsetup` in the preamble (see Section 4). They can also be set locally with `\zcsetup` along the document or through the optional argument of `\zceref` (see Section 4). Most general options can be used in any of these contexts, but that is not necessarily true for all cases, some restrictions may apply, as described in each option’s documentation.

<code>ref</code>	The <code>ref</code> option controls the label property to which <code>\zceref</code> refers to. It can receive values <code>default</code> , <code>zc@thecnt</code> and <code>page</code> . If <code>zref-titleref</code> is loaded, <code>ref</code> also accepts the value <code>title</code> . The package’s default is, well, <code>default</code> , which is our standard reference. <code>zc@thecnt</code> is a property set by <code>zref-clever</code> and is similar to <code>zref</code> ’s <code>default</code> property, except that it is not affected by the kernel’s <code>\labelformat</code> . ¹ By default, reference formatting, sorting, and compression are done according to information inferred from the <i>current counter</i> (see <code>currentcounter</code> option below). Special treatment in these areas is provided for <code>page</code> , but not for <code>title</code> . The <code>page</code> option is a convenience alias for <code>ref=page</code> .
<code>typeset</code> <code>noname</code>	When <code>\zceref</code> typesets a set of references, each group of references of the same type can be, and by default are, preceded by the type’s “name”, and this is indeed an important feature of <code>zref-clever</code> . This is optional however, and the <code>typeset</code> option controls this behavior. It can receive values <code>ref</code> , in which case it typesets only the reference(s), <code>name</code> , in which case it typesets only the name(s), or <code>both</code> , in which case it typesets, well, both of them. Note that, when value <code>name</code> is used, the name is still typeset according to the set of references given to <code>\zceref</code> . For example, for multiple references, the plural form is used, capitalization options are honored, etc. Also hyperlinking behaves just <i>as if</i> the references were present and, depending on the corresponding options, the name may be linked to the first reference of the type group. The <code>noname</code> option is a convenience alias for <code>typeset=ref</code> .
<code>sort</code> <code>nosort</code>	The <code>sort</code> option controls whether the list of <i>labels</i> received as argument by <code>\zceref</code> should be sorted or not. It is a boolean option, and defaults to <code>true</code> . The <code>nosort</code> option is a convenience alias for <code>sort=false</code> .
<code>typesort</code> <code>notypesort</code>	Sorting references of the same type can be done with well defined logical criteria. They either have the same counter or their counters share a clear hierarchical relation (in

¹Technical note: the `default` property stores `\@currentlabel`, while the `zc@thecnt` property stores `\the\@currentcounter`. The later is exactly what `\refstepcounter` uses to build `\@currentlabel`, except for the `\labelformat` prefix and, hence, has the advantage of being unaffected by it. But the former is *more reliable*, since `\@currentlabel` is expected to be correct pretty much anywhere whereas, even though `\refstepcounter` does set `\@currentcounter`, it is not everywhere that uses `\refstepcounter` for the purpose. In the cases where the references from these two do diverge, `zref-clever` will likely misbehave (reference type, sorting and compression inevitably depend on a correct `currentcounter`), but using `default` at least ensures that the reference itself is correct. That said, if you do set `\labelformat` for some reason, `zc@thecnt` may be useful.

the resetting behavior), such that a definite sorting rule can be inferred from the label's data. The same is not true for sorting of references of different types. Should “tables” come before or after “figures”? The `typesort` option allows to specify the sorting priority of different reference types. It receives as value a comma separated list of reference types, specifying that their sorting is to be done in the order of that list. But `typesort` does not need to receive *all* possible reference types. The special value `{othertypes}` (yes, braced) can be placed anywhere along the list, to specify the sort priority of any type not included explicitly in the list. If `{othertypes}` is not present in the list, it is presumed to be at the end of it. Any unspecified types (that is, those falling implicitly or explicitly into the `{othertypes}` category) get sorted between themselves in the order of their first appearance in the label list given as argument to `\zcref`. I presume the common use cases will not need to specify `{othertypes}` at all but, for the sake of example, if you just really dislike equations, you could use `typesort={{othertypes}, equation}`. `typesort`'s default value is `{part, chapter, section, paragraph}`, which places the sectioning reference types first in the list, in their hierarchical order, and leaves everything else to the order of appearance of the labels. The `notypesort` option behaves like `typesort={{othertypes}}` would do, that is, it sorts all types in the order of the first appearance in the labels' list.

`comp` `\zcref` can automatically compress a set of references of the same type into a range, when they occur in immediate sequence. The `comp` controls whether this compression should take place or not. It is a boolean option, and defaults to `true`. The `nocomp` option is a convenience alias for `comp=false`. Of course, for better compression results the `sort` is recommended, but the two options are technically independent.

`range` By default (that is, when the `range` option is not given), `\zcref` typesets a complete list of references according to the `\labels` it received as argument, and only compresses some of them into ranges if the `comp` option is enabled and if references of the same type occur in immediate sequence. The `range` option makes `\zcref` behave differently. Sorting is implied by this option (the `sort` option is disregarded) and, for each reference type group in `\labels`, `\zcref` builds a range from the first to the last reference in it, even if references in between do not occur in immediate sequence. `\zcref` is smart enough, though, to recognize when the first and last references of a type do happen to be contiguous, in which case it typesets a “pair”, instead of a “range”. It is a boolean option, and the package's default is `range=false`. The option given without a value is equivalent to `range=true` (in the l3keys' jargon, the *option*'s default is `true`).

`cap` The `cap` option controls whether the reference type names should be capitalized or not. It is a boolean option, and the package's default is `cap=false`. The option given without a value is equivalent to `cap=true`. The `nocap` option is a convenience alias for `cap=false`. The `capfirst` option ensures that the reference type name of the *first* type block is capitalized, even when `cap` is set to `false`.

`abbrev` The `abbrev` option controls whether to use abbreviated reference type names when they are available. It is a boolean option, and the package's default is `abbrev=false`.
`noabbrev` The option given without a value is equivalent to `abbrev=true`. The `noabbrev` option is a convenience alias for `abbrev=false`. The `noabbrevfirst` ensures that the reference type name of the *first* type block is never abbreviated, even when `abbrev` is set to `true`.

`S` `S` for “Sentence”. The `S` option is a convenience alias for `capfirst=true`, `noabbrevfirst=true`, and is intended to be used in references made at the beginning of a sentence. It is highly recommended that you make a habit of using the `S` option for beginning of sentence references. Even if you do happen to be currently using `cap=true`, `abbrev=false`, proper semantic markup will ensure you get expected results even if you

change your mind in that regard later on. For that reason, it was made short and mnemonic, it can't get any easier.

hyperref The **hyperref** option controls the use of **hyperref** by **zref-clever** and takes values **auto**, **true**, **false**. The default value, **auto**, makes **zref-clever** use **hyperref** if it is loaded, meaning that references made with `\zceref` get hyperlinked to the anchors of their respective *<labels>*. **true** does the same thing, but warns if **hyperref** is not loaded (**hyperref** is never loaded for you). In either of these cases, if **hyperref** is loaded, module **zref-hyperref** is also loaded by **zref-clever**. **false** means not to use **hyperref** regardless of its availability. This is a preamble only option, but `\zceref` provides granular control of hyperlinking by means of its starred version.

nameinlink The **nameinlink** option controls whether the type name should be included in the reference hyperlink or not (provided there is a link, of course). Naturally, the name can only be included in the link of the *first* reference of each type block. **nameinlink** can receive values **true**, **false**, **single**, and **tsingle**. When the value is **true** the type name is always included in the hyperlink. When it is **false** the type name is never included in the link. When the value is **single**, the type name is included in the link only if `\zceref` is typesetting a single reference (not necessarily having received a single label as argument, as they may have been compressed), otherwise, the name is left out of the link. When the value is **tsingle**, the type name is included in the link for each type block with a single reference, otherwise, it isn't. An example: suppose you make a couple of references to something like `\zceref{chap:chapter1}` and `\zceref{chap:chapter1, sec:section1, fig:figure1, fig:figure2}`. The "figure" type name will only be included in the hyperlink if **nameinlink** option is set to **true**. If it is set to **tsingle**, the first reference will include the name in the link for "chapter", as expected, but also in the second reference the "chapter" and "section" names will be included in their respective links, while that of "figure" will not. If the option is set to **single**, only the name for "chapter" in the first reference will be included in the link, while in the second reference none of them will. The package's default is **nameinlink=tsingle**, and the option given without a value is equivalent to **nameinlink=true**.

lang The **lang** option controls the language used by `\zceref` when looking for language-specific reference format options (see Section 7). The default value, **main**, uses the main document language, as defined by **babel** or **polyglossia** (or **english** if none of them is loaded). Value **current** uses the current language, as defined by **babel** or **polyglossia** (or **english** if none of them is loaded). The **lang** option also accepts that the language be specified directly by its name, as long as it's a language known by **zref-clever**. For more details on Internationalization, see Section 8.

d The **d** option sets the declension case, and affects the type name used for typesetting the reference. Whether this option is operative, and which values it accepts, depends on the declared setup for each language. For details, see Section 8.

nudge This set of options revolving around **nudge** aims to offer some guard against mischievous automation on the part of **zref-clever** by providing a number of "nudges" (compilation time messages) for cases in which you may wish to revise material *surrounding* the reference – an article, a preposition – according to the reference typeset results. Useful mainly for languages which inflect the preceding article to gender and/or number, but may be used generally to fine-tune the language and style around the cross-references made with `\zceref`. The **nudge** option is the main entrance to this feature and takes values **true**, **false**, **obeydraft**, or **obeyfinal**. The first two, respectively, enable or disable the "nudging" unconditionally. With **obeydraft**, **nudge** keeps quiet when option **draft** is given to `\documentclass`, while with **obeyfinal**, nudging is only enabled when option **final** is (explicitly) passed to `\documentclass`. The option given without a value

is equivalent to `nudge=true` and the package’s default is `nudge=false`. `nonudge` is a convenience alias for `nudge=false`, and can be used to silence individual references. The `nudgeif` option controls the events which may trigger a nudge. It takes a comma separated list of elements, and recognizes values `multitype`, `comptosing`, `gender`, and `all`. The `multitype` nudge warns when the reference is composed by multiple type blocks (see Section 7). The `comptosing` nudge let’s you know when multiple labels of the same type have been compressed to a singular type name form. It can be combined with the `sg` option, which is the way to tell `\zcref` you know it’s a singular and so not to nudge if a compression to singular occurs, but to nudge if the contrary occurs, that is, when a plural type name form is employed. The `gender` nudge must be combined with option `g`, and depends on the language having support for it. In essence language dictionaries can store the gender of each type name (this is done for built-in dictionaries, but can also be done with `\zcLanguageSetup` for languages declared to support it). The `g` option let’s you specify the gender you expect for that particular reference and the nudge is triggered if there is a mismatch between `g` and the gender for the type name in the dictionary. Both the `comptosing` and the `gender` nudges have a type block as its scope. See Section 8 for more details and intended use cases of the “nudging” feature.

font The `font` option can receive font styling commands to change the appearance of the whole reference list (see also the reference format options, `namefont`, `reffont`, and `reffont-in` in Section 7). It does not affect the content of the `note`, however. The option is intended exclusively for commands that only change font attributes: style, family, shape, weight, size, color, etc. Anything else, particularly commands that may generate typeset output, is not supported. Given how package options are handled by L^AT_EX, the fact that this option receives commands as value means this option *can’t* be set at load time, as a package option. If you want to set it globally, use `\zcsetup` instead.

titleref The `titleref` option receives no value and, when given, loads `zref`’s `zref-titleref` module. This is a preamble only option.

note The `note` option receives as value some text to be typeset at the end of the whole reference list. It is separated from it by `notesep` (see Section 7).

check Provides integration of `zref-clever` with the `zref-check` package. In the preamble, the `check` option receives no value and, when given, loads `zref-check`. In the document body, `check` requires a value, which works exactly like the optional argument of `\zcheck`, and can receive both checks and `\zcheck`’s options. And the checks are performed for each label in `{\labels}` received as argument by `\zcref`. See the User manual of `zref-check` for details. The checks done by the `check` option in `\zcref` comprise the complete reference, including the `note` (see Section 7). If `zref-check` was not loaded in the preamble, at begin document the option is made no-op and issues a warning.

countertype The `countertype` option allows to specify the “reference type” of each counter, which is stored as a label property when the label is set. This “reference type” is what determines how a reference to this label will eventually be typeset when it is referred to (see Section 6). A value like `countertype = {foo = bar}` sets the `foo` counter to use the reference type `bar`. There’s only need to specify the `countertype` for counters whose name differs from that of their type, since `zref-clever` presumes the type has the same name as the counter, unless otherwise specified. Also, the default value of the option already sets appropriate types for basic L^AT_EX counters, including those from the standard classes. Setting a counter type to an empty value removes any (explicit) type association for that counter, in practice, this means it then uses a type equal to its name. Since this option only affects how labels are set, it is not available in `\zcref`.



counterresetters
counterresetby

The sorting and compression of references done by `\zcref` requires that we know the counter whose `\refstepcounter` is being stored by `\zlabel` but also information on

any counter whose stepping may trigger its resetting, or its “enclosing counters”. This information is not easily retrievable from the counter itself but is (normally) stored with the counter that does the resetting. The `counterresetters` option adds counter names, received as a comma separated list, to the list of counters `zref-clever` uses to search for “enclosing counters” of the counter for which a label is being set. Unfortunately, not every counter gets reset through the standard machinery for this, including some L^AT_EX kernel ones (e.g. the `enumerate` environment counters). For those, there is really no way to retrieve this information directly, so we have to just tell `zref-clever` about them. And that’s what the `counterresetby` option is made for. It receives a comma separated list of `key=value` pairs, in which `key` is the counter, and `value` is its “enclosing counter”, that is, the counter whose stepping results in its resetting. This is not really an “option” in the sense of “user choice”, it is more of a way to inform `zref-clever` of something it cannot know or automatically find in general. One cannot place arbitrary information there, or `zref-clever` can be thoroughly confused. The setting must correspond to the actual resetting behavior of the involved counters. `counterresetby` has precedence over the search done in the `counterresetters` list. The default value of `counterresetters` includes the counters for sectioning commands of the standard classes which, in most cases, should be the relevant ones for cross-referencing purposes. The default value of `counterresetby` includes the `enumerate` environment counters. So, hopefully, you don’t need to ever bother with either of these options. But, if you do, they are here. Use them with caution though. Since these options only affect how labels are set, they are not available in `\zcref`.



`currentcounter`

L^AT_EX’s `\refstepcounter` sets two variables which potentially affect the `\zlabel` set after it: the current label (`\@currentlabel`), and the current counter (`\@currentcounter`). Actually, traditionally, only the current label was thus stored, the current counter was added to `\refstepcounter` somewhat recently (with the 2020-10-01 kernel release). But, since `zref-clever` relies heavily on the information of what the current counter is, it must set `zref` to store that information with the label, as it does. As long as the document element we are trying to refer to uses the standard machinery of `\refstepcounter` we are on solid ground and can retrieve the correct information. However, it is not always ensured that `\@currentcounter` is kept up to date. For example, packages which handle labels specially, for one reason or another, may or may not set `\@currentcounter` as required. Considering the addition of `\@currentcounter` to `\refstepcounter` itself is not that old, it is likely that in a good number of places a reliable `\@currentcounter` is not really in place. Therefore, it may happen we need to tell `zref-clever` what the current counter is in certain circumstances, and that’s what `currentcounter` does. The same as with the previous two options, this is not really an “user choice” kind of option, but a way to tell `zref-clever` a piece of information it has no means to retrieve automatically. The setting must correspond to the actual “current counter”, meaning here “the counter underlying `\@currentlabel`” in a given situation. Its default value is, quite naturally, `\@currentcounter`. Since this option only affects how labels are set, it is not available in `\zcref`.

6 Reference types

A “reference type” is the basic `zref-clever` setup unit for specifying how a cross-reference group of a certain kind is to be typeset. Though, usually, it will have the same name as the underlying L^AT_EX *counter*, they are conceptually different. `zref-clever` sets up *reference types* and an association between each *counter* and its *type*, it does not define

the counters themselves, which are defined by your document. One *reference type* can be associated with one or more *counters*, and a *counter* can be associated with different *types* at different points in your document. But each label is stored with only one *type*, as specified by the counter-type association at the moment it is set, and that determines how the reference to that label is typeset. References to different *counters* of the same *type* are grouped together, and treated alike by `\zcref`. A *reference type* may be known to `zref-clever` when the *counter* it is associated with is not actually defined, and this is inconsequential. In practice, the contrary may also happen, a *counter* may be defined but we have no *type* for it, but this must be handled by `zref-clever` as an error (at least, if we try to refer to it), usually a “missing name” error.

`zref-clever` provides default settings for the following reference types: `part`, `chapter`, `section`, `paragraph`, `appendix`, `subappendix`, `page`, `line`, `figure`, `table`, `item`, `footnote`, `note`, `equation`, `theorem`, `lemma`, `corollary`, `proposition`, `definition`, `proof`, `result`, `remark`, `example`, `algorithm`, `listing`, `exercise`, `solution`. Therefore, if you are using a language for which `zref-clever` has built-in support (see Section 8), these reference types are available for use out-of-the-box.² And, in any case, it is always easy to setup custom reference types with `\zcRefTypeSetup` or `\zcLanguageSetup` (see Sections 4, 7 and 8).

The association of a *counter* to its *type* is controlled by the `countertype` option. As seen in its documentation, `zref-clever` presumes the *type* to be the same as the *counter* unless instructed otherwise by that option. This association, as determined by the local value of the option, affects how the *label* is set, which stores the type among its properties. However, when it comes to typesetting, that is from the perspective of `\zcref`, only the *type* matters. In other words, how the reference is supposed to be typeset is determined at the point the *label* gets set. In sum, they may be namesakes (or not), but type is type and counter is counter.

Indeed, a reference type can be associated with multiple counters because we may want to refer to different document elements, with different *counters*, as a single *type*, with a single name. One prominent case of this are sectioning commands. `\section`, `\subsection`, and `\subsubsection` have each their counter, but we’d like to refer to all of them by “sections” and group them together. The same for `\paragraph` and `\subparagraph`.

There are also cases in which we may want to use different *reference types* to refer to document objects sharing the same *counter*. Notably, the environments created with L^AT_EX’s `\newtheorem` command and the `\appendix`.

One more observation about “reference types” is due here. A *type* is not really “defined” in the sense a variable or a function is. It is more of a “string” which `zref-clever` uses to look for a whole set of type-specific reference format options (see Section 7). Each of these options individually may be “set” or not, “defined” or not. And, depending on the setup and the relevant precedence rules for this, some of them may be required and some not. In practice, `zref-clever` uses the *type* to look for these options when it needs one, and issues a compilation warning when it cannot find a suitable value.

7 Reference format

Formatting how the reference is to be typeset is, quite naturally, a big part of the user interface of `zref-clever`. In this area, we tried to balance “flexibility” and “user friendli-

²There may be slight availability differences depending on the language, but `zref-clever` strives to keep this complete list available for the languages it has built-in dictionaries.

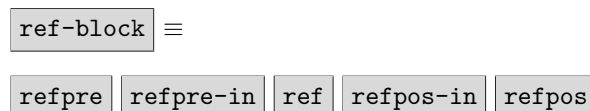
ness”. But the former does place a big toll overall, since there are indeed many places where tweaking may be desired, and the settings may depend on at least two important dimensions of variation: the reference type and the language. Combination of those necessarily makes for a large set of possibilities. Hence, the attempt here is to provide a rich set of “handles” for fine tuning the reference format but, at the same time, do not *require* detailed setup by the users, unless they really want it.

With that in mind, we have settled with an user interface for reference formatting which allows settings to be done in different scopes, with more or less overarching effects, and some precedence rules to regulate the relation of settings given in each of these scopes. There are four scopes in which reference formatting can be specified by the user, in the following precedence order: i) as general *options*; ii) as *type-specific options*; iii) as *language-specific and type-specific translations*; and iv) as *default translations* (that is, language-specific but not type-specific). Besides those, there’s a fifth *internal* scope, with the least priority of all, a “fallback”, for the cases where it is meaningful to provide some value, even for an unknown language.

General “options” (i) can be given by the user in the optional argument of `\zcref`, but also set through `\zcsetup` or even, depending on the case, as package options at load-time (see Section 5).³ “Type-specific options” (ii) are handled by `\zcRefTypeSetup` (see Section 4). “Language-specific translations”, be they “type-specific” (iii) or “default” (iv) have their user interface in `\zcLanguageSetup`, and have their values populated by the package’s built-in dictionaries (see Section 8). Not all reference format specifications can be given in all of these scopes, though. Some of them can’t be type-specific, others must be type-specific, so the set available in each scope depends on the pertinence of the case. Table 1 introduces the available reference format options, which will be discussed in more detail soon, and lists the scopes in which each is available.

The package itself places the default setup for reference formatting at low precedence levels, and the users can easily and conveniently override them as desired. Indeed, I expect most of the users’ needs to be normally achievable with the general options and type-specific options, since references will normally be typeset in a single language (the document’s main language) and, hence, multiple translations don’t need to be provided.

Understanding the role of each of these reference format options is likely eased by some visual schemes of how `zref-clever` builds a reference based on the labels’ data and the value of these options. Take a `ref` to be that which a standard \LaTeX `\ref` would typeset. A `zref-clever` “reference block”, or `ref-block`, is constructed as:



Where the difference between `refpre` and `refpos`, with regard to `refpre-in` and `refpos-in` is that the later are included in the reference hyperlink, whereas the former are not.

A `ref-block` is built for *each* label given as argument to `\zcref`. When the $\langle labels \rangle$ argument is comprised of multiple labels, each “reference type group”, or `type-group` is

³The use of `\zcsetup` for global reference format settings is recommended though. Whether you can use load-time options or not depends on the values of the options: due to how \LaTeX handles package options, if the values of the options you are setting include *commands* you can’t set them at load-time, and rather *must* use `\zcsetup`.

		General options (i)	Type options (ii)	Type-specific translations (iii)	Default translations (iv)
Necessarily not type-specific	<code>tpairsep</code>	•			•
	<code>tlistsep</code>	•			•
	<code>tlastsep</code>	•			•
	<code>notesep</code>	•			•
Possibly type-specific	<code>namesep</code>	•	•	•	•
	<code>pairsep</code>	•	•	•	•
	<code>listsep</code>	•	•	•	•
	<code>lastsep</code>	•	•	•	•
	<code>rangesep</code>	•	•	•	•
	<code>refpre</code>	•	•	•	•
	<code>refpos</code>	•	•	•	•
	<code>refpre-in</code>	•	•	•	•
	<code>refpos-in</code>	•	•	•	•
Necessarily type-specific	<code>Name-sg</code>		•	•	
	<code>name-sg</code>		•	•	
	<code>Name-pl</code>		•	•	
	<code>name-pl</code>		•	•	
	<code>Name-sg-ab</code>		•	•	
	<code>name-sg-ab</code>		•	•	
	<code>Name-pl-ab</code>		•	•	
	<code>name-pl-ab</code>		•	•	
Font options	<code>namefont</code>	•	•		
	<code>reffont</code>	•	•		
	<code>reffont-in</code>	•	•		

Table 1: Reference format options and their scopes

potentially made from the combination of single reference blocks, “reference block pairs”, “reference block lists”, or “reference block ranges”, where each is respectively built as:

`type-group` is a combination of:

`ref-block`

`ref-block1` `pairsep` `ref-block2`

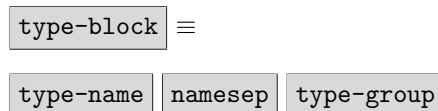
`ref-block1` `listsep` `ref-block2` `listsep` `ref-block3` ...

... `ref-blockN-1` `lastsep` `ref-blockN`

`ref-block1` `rangesep` `ref-blockN`

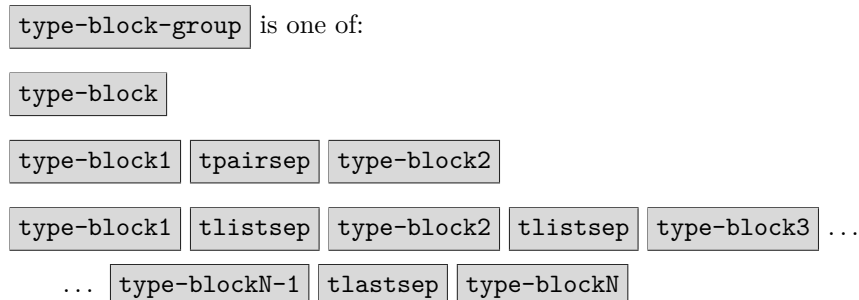
To complete a “type-block”, a `type-group` only needs to be accompanied by the

“type name”:

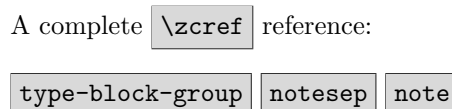


The `type-name` is determined not by one single reference format option but by the appropriate one among the `[Nn]ame-` options according to the composition of `type-group` and the general options. The reference format name options are eight in total: `Name-sg`, `name-sg`, `Name-pl`, `name-pl`, `Name-sg-ab`, `name-sg-ab`, `Name-pl-ab`, and `name-pl-ab`. The initial uppercase “N” signals the capitalized form of the type name. The `-sg` suffix stands for singular, while `-pl` for plural. The `-ab` is appended to the abbreviated type name form options. When setting up a type, not necessarily all forms need to be provided. `zref-clever` will always use the non-abbreviated form as a fallback to the abbreviated one, if the later is not available. Hence, if a reference type is not intended to be used with abbreviated names (the most common case), only the basic four forms are needed. Besides that, if you are using the `cap` option, only the capitalized forms will ever be required by `\zcref`, so you can get away setting only `Name-sg` and `Name-pl`. You should not do the contrary though, and provide only the non-capitalized forms because, even if you are using the `nocap` option, the capitalized forms will be still required for `capfirst` and `S` options to work. Whatever the case may be, you need not worry too much about being remiss in this area: if `\zcref` does lack a name form in any given reference, it will let you know with a compilation warning (and will typeset the usual missing reference sign: “??”).

A complete reference typeset by `\zcref` may be comprised of multiple `type-blocks`, in which case the “type-block-group” can also be made of single type blocks, “type block pairs” or “type block lists”, where each is respectively built as:



Finally, since `\zcref` can also receive an optional `note`, its full typeset output is built as:



Reference format options can yet be divided in two general categories: i) “string” options, the ones which we have seen thus far, as “building blocks” of the reference; and ii) “font” options, which control font attributes of parts of the reference, namely `namefont`, `reffont`, and `reffont-in`. These options set the font, respectively, for the `type-name`, for the whole `ref-block`, and for the part of `ref-block` which is included in

the hyperlink (to set the font for the whole reference, see the `font` option in Section 5). “String” options is not really a strict denomination for the first category, but this set of options is intended exclusively for typesetting material: things you expect to see in the output of your references. The “font” options, on the other hand, are intended exclusively for commands that only change font attributes: style, family, shape, weight, size, color, etc. In either case, anything other than their intended uses is not supported.

Finally, a comment about the internal “fallback” reference format values mentioned above. These “last resort” option values are required by `zref-clever` for a clear particular case: if the user loads either `babel` or `polyglossia`, or explicitly sets a language, with a language that `zref-clever` does not know and has no dictionary for, it cannot guess what language that is, and thus has to provide some reasonable “language agnostic” default, at least for the options for which this makes sense (all the “string” options, except for the `[Nn]ame-` ones). Users do not need to have access to this scope, since they know the language of their document, or know the values they want for those options, and can set them as general options, type-specific options, or language options through the user interface provided for the purpose. But the “fallback” options are documented here so that you can recognize when you are getting these values and change them appropriately as desired. Though hopefully reasonable, they may not be what you want. The “fallback” option values are the following:

```
tpairsep = {, } ,
tlistsep = {, } ,
tlastsep = {, } ,
notesep  = { } ,
namesep  = {\nobreakspace} ,
pairsep   = {, } ,
listsep   = {, } ,
lastsep   = {, } ,
rangesep  = {\textendash} ,
refpre    = {} ,
refpos    = {} ,
refpre-in = {} ,
refpos-in = {} ,
```

8 Internationalization

`zref-clever` provides internationalization facilities and integrates with `babel` and `polyglossia` to adapt to the languages in use by either of these language packages, or to a language specified directly by the user. This is primarily relevant for reference format options, particularly reference type *names* (though not only, since most reference format options can have language-specific values, or “translations”, see Section 7). But other features of the package also cater for language specific needs.

As far as language selection is concerned, if the language is declared and `zref-clever` has a built-in “dictionary” for it, most use cases will likely be covered by the `lang` option (see Section 5), and its values `main` and `current`. When the `lang` option is set to `main` or `current`, `zref-check` will use, respectively, the *main* or *current* language of the document, as defined by `babel` or `polyglossia`.⁴ Users can also set `lang` to a specific

⁴Technically, `zref-clever` uses `\bb1@main@language` and `\language` for `babel`, and `\mainbabelname` and `\babelname` for `polyglossia`, which boils down to `zref-clever` always using *babel names* internally,

Language	Aliases	Language	Aliases
english	american	german	austrian
	australian		germanb
	british		ngerman
	canadian		naustrian
	newzealand		nswissgerman
	UKenglish		swissgerman
french	USenglish	portuguese	brazilian
	acadian		brazil
	canadien		portuges
	francais	spanish	
	frenchb		

Table 2: Declared languages and aliases

language directly, in which case `babel` and `polyglossia` are disregarded. `zref-clever` provides a number of built-in “dictionaries”, for the languages listed in Table 2, which also includes the declared aliases to those languages.

`zref-clever`’s “dictionaries” are loaded sparingly and lazily. A dictionary for a single language – that specified by user options in the preamble, which by default is the main document language – is loaded at `\begin{document}`. If any other dictionary is needed, it is loaded on the fly, if and when required. Of course, in either case, conditioned on availability. This is done because the presumed common use case for `zref-clever` is to use a single language for cross-references (the “main” one), even in many multi-language documents scenarios. Hence, even the “loaded languages” set, from `babel` or `polyglossia`, would tend to be an overshoot of the actual needs. So, `zref-clever` loads as little as possible, but allows for convenient on the fly loading of dictionaries if the values are indeed required, without users having to worry about it at all.

But if the built-in dictionaries do not cover your language, or if you’d like to adjust some of the default language-specific options, this can be done with `\zcDeclareLanguage`, `\zcDeclareLanguageAlias`, and `\zcLanguageSetup`.⁵

regardless of which language package is in use. Indeed, an acquainted user will note that Table 2 contains only `babel` language names.

⁵Needless to say, if you’d like to contribute a dictionary or improve an existing one, that is much welcome at <https://github.com/gusbrs/zref-clever/issues>.

Language	declension	gender	allcaps
english	-	-	-
french	-	f,m	-
german	N,A,D,G	f,m,n	yes
portuguese	-	f,m	-
spanish	-	f,m	-

Table 3: Options for declared languages

`\zcDeclareLanguage`

`\zcDeclareLanguage` [*options*] {*language*}

Declare a new language for use with `zref-clever`. If *language* has already been declared, just warn. The *options* argument receives the usual **key=value** list and recognizes three keys: **declension**, **gender**, and **allcaps**. **declension** receives a coma separated list of valid declension cases for *language*. The first element of the list is considered to be the default case, both for the **d** option in `\zcref` and for the **case** option in `\zcLanguageSetup`. Similarly, **gender** receives a comma separated list of genders for *language*. The elements in this list are those which are recognized as valid for the language for both the **g** option in `\zcref` and the **gender** option in `\zcLanguageSetup`. There is no default presumed in this case. Finally, **allcaps** can be used with languages for which nouns must be always capitalized for grammatical reasons. For a language declared with the **allcaps** option, the **cap** reference option (see Section 5) is disregarded, and `\zcref` always uses the capitalized type name forms. This means that dictionaries for languages with such a trait can be halved in size, and that user customization for them is simplified, only requiring the capitalized name forms. On the other hand, the non-capitalized **name-...** reference format options are rendered no-op for the language in question. Table 3 presents an overview of the options in effect for the languages declared by `zref-clever`. `\zcDeclareLanguage` is preamble only.

`\zcDeclareLanguageAlias`

`\zcDeclareLanguageAlias` {*language alias*} {*aliased language*}

Declare *language alias* to be an alias of *aliased language*. *aliased language* must be already known to `zref-clever`. Once set, the *language alias* is treated by `zref-clever` as completely equivalent to the *aliased language* for any language specification by the user. `\zcDeclareLanguageAlias` is preamble only.

\zcLanguageSetup**\zcLanguageSetup** {*<language>*} {*<options>*}

Sets language-specific reference format options for *<language>* (see Section 7), be they type-specific or not. *<language>* must be already known to **zref-clever**. Besides reference format options, **\zcLanguageSetup** knows three other keys: **type**, **case**, and **gender**. The first two work like a “switch” affecting the options *following* it. For example, if **type=foo** is given in *<options>* the options following it will be set as type-specific options for reference type **foo**. Similarly, after **case=X** (provided **X** is a valid declension case for *<language>*), the following **[Nn]ame-...** options will set values for the **X** declension case (other reference format options are not affected by **case**). Before the first occurrence of either **type** or **case** default values are set. For **case** this means the default declension case, which is the first element of the list provided to the **declension** option in **\zcDeclareLanguage**. For **type** this means “default translations”, which are language-specific but not type-specific option values (see Section 7). An empty valued **type=** key can also “unset” the type. The **gender** key sets the gender of the current **type** (provided the value it receives is one of the declared genders for *<language>*). **\zcLanguageSetup** is preamble only.

A couple of examples to illustrate the syntax of **\zcLanguageSetup**:

```
\zcLanguageSetup{french}{
  type = section ,
  gender = f ,
  Name-sg = Section ,
  name-sg = section ,
  Name-pl = Sections ,
  name-pl = sections ,
}
\zcLanguageSetup{german}{
  type = section ,
  gender = m ,
  case = N ,
  Name-sg = Abschnitt ,
  Name-pl = Abschnitte ,
  case = A ,
  Name-sg = Abschnitt ,
  Name-pl = Abschnitte ,
  case = D ,
  Name-sg = Abschnitt ,
  Name-pl = Abschnitten ,
  case = G ,
  Name-sg = Abschnitts ,
  Name-pl = Abschnitte ,
}
```

As already noted, **zref-clever** has some support for languages with declension. This means mainly the declension of *nouns*, which is used for the reference type names. But some tools are also provided to support the user in getting better results for the text surrounding a reference, particularly for numbered and gendered articles, even if those don’t have their typeset output automated.

For reference type names, the declension cases for each language must be declared with **\zcDeclareLanguage**, and the name reference format options must be provided for

each case, which is done for built-in dictionaries of languages which have noun declension, and can be done by the user with `\zcLanguageSetup`, as we’ve seen. `zref-clever` does not try to guess or infer the case though, you must tell it to `\zcref`. And this is done by means of the `d` option (see Section 5). So you may write something like “nach den `\zcref[d=D]{sec:section-1,sec:section-2}`” to get “nach den Abschnitten 1 und 2”. Or “trotz des `\zcref[d=G]{eq:theorem-1}`” to get “trotz des Theorems 1”.

Regarding the text surrounding the reference – the inflected article, the passing preposition, etc. –, the issue is more delicate. `zref-clever` cannot and intends not to typeset those for you. But, depending on the language, it is true that the kind of automation provided by `zref-clever` may betray your best efforts to get a proper surrounding text. Multiple labels passed to `\zcref` may result in singular type names, either because the labels are of different types, or because they got compressed into a single reference. References comprised of multiple type blocks may have each a name with a different gender. Or, worse, `tpairsep`, `tpairsep`, and `tlastsep` may not provide a general enough way to separate different type blocks in your language altogether. You may change something in your document that causes a label to change its type, and hence the gender of the type name. A page reference to a couple of floats which were by chance on the same page and all of a sudden no longer are. And so on.

In this area, the approach taken by `zref-clever` is to identify some typical situations in which your attention may be required in reviewing the surrounding text, and signal it at compilation time. Just like bad boxes, for example. This feature can be enabled by the `nudge` option (which is opt-in, see Section 5). There are three “nudges” available for this purpose which trigger messages at different events: `multitype`, `comptosing`, and `gender`. `multitype` nudges when a reference is comprised of multiple type blocks. `comptosing` when multiple labels of the same type block have been compressed into a single one and, hence, the type name used is singular. Finally, `gender` nudges when there is a mismatch between the gender specified in `\zcref` with the `g` option and the gender of the type name, as stored in the dictionary or language settings, for each type block. Which nudges to use is configurable with the option `nudgeif`. And, if you’re sure of the results for a particular `\zcref` call, you can always silence the nudges locally with the `nonudge` option.

The main reason to watch for multiple type references with the `multitype` nudge is that bundling together automatically a list of type blocks is less smooth an operation than it is for a single reference type. While it arguably works reasonably well for English, even there it is not always flawless, and depending on the language, results may range from “lack of style” to outright wrong. A typical case would be of that of a language with inflected articles and a reference with multiple types of different genders or numbers. For example, in French, with a standard “au `\zcref{cha:chapter-3,sec:section-3.1}`” we get “au chapitre 3 et section 3.1” which just sounds like a shrieking chalkboard. So we may be better off writing instead “au `\zcref{cha:chapter-3}` et à la `\zcref{sec:section-3.1}`”. Or something else, of course. But the general point is that, depending on circumstances and on the language, the results of automating the grouping of multiple reference types, as `zref-clever` is able to do, may leave things to be desired for. Hence it lets you know when one such case occurs, so that you can review it for best results.

The case of the `comptosing` and `gender` nudges is more objective in nature, they respectively signal mismatches of number and gender. When a reference is made with `\zcref` to a single label we are sure the type name will be a singular form. However, when `\zcref` receives multiple labels of the same type, the type name will normally be a plural, but not necessarily so, since the labels may be compressed into a single one (see

the `comp` option in Section 5), in which case the singular is used. The compression of multiple labels into a single reference should be an exception for default references, but not so for `page` references, where it is easy to conceive practical situations where it may occur. Suppose, for example, you have two contiguous floats in your document and make a page reference to both of them. Will they end up in the same page or not? Maybe we know what the current state is, but we cannot know what may happen as the document keeps being edited. As a consequence, we don't know whether that reference will end up having a plural or a singular type name. That given, the logic of the `comptosing` nudge is the following. If we are giving multiple labels to `\zcref` we can *presume* a plural type name, but we get a nudge in case the compression of the labels results in a singular type name form. If one such compression did happen to one of your references you can use a singular article and then tell `\zcref` you did so with option `sg`. The effect of the `sg` option is to inhibit the nudge when a compression to singular occurs, but to do it instead when the compression *ceases* to occur, that is, if we get a plural type name again at some point.

The `gender` nudge aims to guard against one particular situation: possible changes of a reference's type. This does not occur by reason of any internal behavior of `zref-clever`, but it may be caused by changes in the document. You may wish to change one `theorem` into a `proposition` and, if you're writing in French or Portuguese, for example, that implies that the reference to it changes gender and the likely preceding article will no longer pass to the reference. The `gender` nudge requires that the gender of each type name and of each reference be explicitly specified. For the type names, this is done for the built-in dictionaries of languages were this matters, and can be done with `\zcLanguageSetup` as well. For the references, that is the purpose of the `g` option. When there is a mismatch between the two for any type block, the nudge is triggered. Of course, this means that the gender markup has to be supplied in the document at each reference. And considering such type changes may not be frequent for you, or considered not particularly problematic, you'll have to balance if doing so is worth it. Still, the feature is available, and it's up to you.

9 How tos

This section gathers some usage examples, or “how tos”, of cases which may require some `zref-clever` setup, and each item is set around a cross-reference “task” we'd like to perform with `zref-clever`.

9.1 `\newtheorem`

Since \LaTeX 's `\newtheorem` allows users to create arbitrary numbered environments, with respective arbitrary counters, the most `zref-clever` can do in this regard is to provide some “typical” built-in reference types to smooth user setup but, in the general case, some user setup may be indeed required. The examples below are equally valid for `amsthm`'s `\newtheorem` since, even it provides features beyond those available in the kernel, its syntax and underlying relation with counters is pretty much the same. The same for `nttheorem`. For `thmtools`' `\declaretheorem` the situation is also similar. Though some adjustments to the examples below may be required, the basic logic is the same. There is no integration with the `Refname`, `refname`, and `label` options, which are targeted to the standard reference system, but you don't actually need them to get things working conveniently.

Simple case

Task Setup up a new theorem environment created with `\newtheorem` to be referred to with `\zceref`. The theorem environment does not share its counter with other theorem environments, and one of `zref-clever` built-in reference types is adequate for my needs.

Suppose you set a “Lemma” environment with:

```
\newtheorem{lemma}{Lemma}
```

Or with:

```
\newtheorem{lemma}{Lemma}[section]
```

In this case, since `zref-clever` provides a built-in `lemma` type (for supported languages) and presumes the reference type to be the same name as the counter, there is no need for setup, and things just work out-of-the-box. So, you can go ahead with:

```
\documentclass{article}
\usepackage{zref-clever}
\newtheorem{lemma}{Lemma}[section]
\begin{document}
\section{Section 1}
\begin{lemma}\zlabel{lemma-1}
  A lemma.
\end{lemma}
\zceref{lemma-1}
\end{document}
```

If, however, you had chosen an environment name which did not happen to coincide with the built-in reference type, all you’d need to do is instruct `zref-clever` to associate the counter for your environment to the desired type with the `countertype` option:

```
\documentclass{article}
\usepackage{zref-clever}
\zcsetup{countertype={lem=lemma}}
\newtheorem{lem}{Lemma}[section]
\begin{document}
\section{Section 1}
\begin{lem}\zlabel{lemma-1}
  A lemma.
\end{lem}
\zceref{lemma-1}
\end{document}
```

Shared counter

Task Setup up two new theorem environments created with `\newtheorem` to be referred to with `\zceref`. The theorem environments share the same counter, and the available `zref-clever` built-in reference types are adequate for my needs.

In this case, we need to set the `countertype` option in the appropriate contexts, so that the labels of each environment get set with the expected reference type. As we’ve seen (at Section 4), `\zcsetup` has local effects, so it can be issued inside the respective environments for the purpose. Even better, we can leverage the kernel’s new hook management system and just set it for all occurrences with `\AddToHook{env}/myenv/begin}`.

```
\documentclass{article}
\usepackage{zref-clever}
\AddToHook{env/mytheorem/begin}{%
  \zcsetup{countertype={mytheorem=theorem}}}
\AddToHook{env/myproposition/begin}{%
  \zcsetup{countertype={mytheorem=proposition}}}
\newtheorem{mytheorem}{Theorem}[section]
\newtheorem{myproposition}[mytheorem]{Proposition}
\begin{document}
\section{Section 1}
\begin{mytheorem}\zlabel{theorem-1}
  A theorem.
\end{mytheorem}
\begin{myproposition}\zlabel{proposition-1}
  A proposition.
\end{myproposition}
\zcref{theorem-1,proposition-1}
\end{document}
```

Custom type

Task Setup up a new theorem environment created with `\newtheorem` to be referred to with `\zcref`. The theorem environment does not share its counter with other theorem environments, but none of `zref-clever` built-in reference types is adequate for my needs.

In this case, we need to provide `zref-clever` with settings pertaining to the custom reference type we’d like to use. Unless you need to typeset your cross-references in multiple languages, in which case you’d require `\zcLanguageSetup`, the most convenient way to setup a reference type is `\zcRefTypeSetup`. In most cases, what we really need to provide for a custom type are the “type names” and other reference format options can rely on default translations already provided by the package (assuming the language is supported).

```
\documentclass{article}
\usepackage{zref-clever}
\newtheorem{myconjecture}{Conjecture}[section]
\zcRefTypeSetup{myconjecture}{
  Name-sg = Conjecture ,
  name-sg = conjecture ,
  Name-pl = Conjectures ,
  name-pl = conjectures ,
}
\begin{document}
\section{Section 1}
```

```

\begin{myconjecture}\zlabel{conjecture-1}
  A conjecture.
\end{myconjecture}
\zcref{conjecture-1}
\end{document}

```

9.2 enumitem

Task Setup a custom enumerate environment created with `enumitem` to be referred to with `\zcref`.

Since the `enumerate` environment's counters are reset at each nesting level, but not with the usual `\refstepcounter` machinery, we have to inform `zref-clever` of this resetting behavior with the `counterresetby` option. Also, given the naming of the underlying counters is tied with the environment's name and the level's number, we cannot really rely on an implicit counter-type association, and have to set it explicitly with the `countertype` option.

```

\documentclass{article}
\usepackage{zref-clever}
\zcsetup{
  countertype = {
    myenumeratei   = item ,
    myenumerateii  = item ,
    myenumerateiii = item ,
    myenumerateiv  = item ,
  } ,
  counterresetby = {
    myenumerateii = myenumeratei ,
    myenumerateiii = myenumerateii ,
    myenumerateiv = myenumerateiii ,
  }
}
\usepackage{enumitem}
\newlist{myenumerate}{enumerate}{4}
\setlist[myenumerate,1]{label=(\arabic*)}
\setlist[myenumerate,2]{label=(\Roman*)}
\setlist[myenumerate,3]{label=(\Alph*)}
\setlist[myenumerate,4]{label=(\roman*)}
\begin{document}
\begin{myenumerate}
\item An item.\zlabel{item-1}
  \begin{myenumerate}
\item An item.\zlabel{item-2}
    \begin{myenumerate}
\item An item.\zlabel{item-3}
      \begin{myenumerate}
\item An item.\zlabel{item-4}
        \end{myenumerate}
      \end{myenumerate}
    \end{myenumerate}
  \end{myenumerate}
\end{document}

```

```

\end{myenumerate}
\end{myenumerate}
\zcref{item-1,item-2,item-3,item-4}
\end{document}

```

9.3 zref-xr

Task Make references to labels set in an external document with `\zcref`.

`zref` itself offers this functionality with module `zref-xr`, and `zref-clever` is prepared to make use of it. Just a couple of details have to be taken care of, for it to work as intended: i) `zref-clever` must be loaded in both the main document and the external document, so that the imported labels also contain the properties required by `zref-clever`; ii) since `\zexternaldocument` defines any properties it finds in the labels from the external document when it imports them, it must be called after `zref-clever` is loaded, otherwise the later will find its own internal properties already defined when it does get loaded, and will justifiably complain. Note as well that the starred version of `\zexternaldocument*`, which imports the standard labels from the external document, is not sufficient for `zref-clever`, since the imported labels will not contain all the required properties.

Assuming here `documentA.tex` as the main file and `documentB.tex` as the external one, and also assuming we just want to refer in “A” to the labels from “B”, and not the contrary, a minimum setup would be the following.

```

documentA.tex:

\documentclass{article}
\usepackage{zref-clever}
\usepackage{zref-xr}
\zexternaldocument[B-]{documentB}
\usepackage{hyperref}
\begin{document}
\section{Section A1}
\zlabel{sec:section-a1}
\zcref{sec:section-a1,B-sec:section-b1}
\end{document}

documentB.tex:

\documentclass{article}
\usepackage{zref-clever}
\usepackage{hyperref}
\begin{document}
\section{Section B1}
\zlabel{sec:section-b1}
\end{document}

```

10 Limitations

Being based on `zref` entails one quite sizable advantage for `zref-clever`: the extensible referencing system of the former allows `zref-clever` to store and retrieve the information it

needs to work without having to redefine some core L^AT_EX commands. This alone makes for reduced compatibility problems and less load order issues than the average package in this functionality area. On the other hand, being based on `zref` also does impair the supported scope of `zref-clever`. Not because of any particular limitation of either, but because any class or package which implements some special handling for reference labels universally does so aiming at the standard referencing system, and whether specific support for `zref` is included, or whether things work by spillover of the particular technique employed, is not guaranteed.

The limitation here is less one of `zref-clever` than that of a potential lack of support for `zref` itself. Broadly speaking, what `zref-clever` does is setup `zref` so that its `\zref@newlabels` contain the information we need using `zref`'s API. Once the `\zlabel` is set correctly, there is little in the way of `zref-clever`, it can just extract the label's information, again using `zref`'s API, and do its job. Therefore, the problems that may arise are really in *label setting*.

For `\zlabel` to be able to set a label with everything `zref-clever` needs, some conditions must be fulfilled, most of which are pretty much the same as that of a regular label, but not only. In particular, the `\@currentcounter` is critical for the task of `zref-clever`: the reference type will depend on that and, consequently, sorting and compression as well, counter resetting behavior information is also retrieved based on it, and so on. `zref` provides the `zref-counter` module which stores it in the `counter` property.

Despite that, as far as I've been experimenting, the following label setting requirements seem to be potentially problematic and are not necessarily granted for `\zlabel`:

1. One must be able to call `\zlabel` at the appropriate scope/location so as to set the label. Alternatively, `\zlabel` must be called with the desired label by some other means.
2. When `\zlabel` is set, it must see the proper value of `\@currentcounter`.

As for the first, it is not everywhere we technically can set a (z)label. On verbatim-like environments it depends on how they are defined and whether they provide a proper place or option to do so. But not only those, `amsmath` environments also handle `\label` specially and the same work is not done for `\zlabel`.

Regarding `\@currentcounter`, since the 2020-10-01 kernel release, it is set by `\refstepcounter` alongside `\@currentlabel`. Hence, as long as `\refstepcounter` is the tool used for the purpose, `\zlabel` will tend to have all information within its grasp at label setting time. But that's not always the case. For this reason `zref-clever` has the option `currentcounter` which at least allows for some viable workarounds when the kernel's `\@currentcounter` itself cannot be relied upon. Whether we have a proper opening to set it, depends on the case. Still, `\refstepcounter` is ubiquitous enough a tool that we can count on `\@currentcounter` most of the time.

All in all, most things work, but some things don't. And if the later will eventually work depends essentially on whether support for `zref` (`zref-counter` included) is provided by these packages/features or not. In what follows of this section, I list cases of known lack of support. Even if, formally, this should probably belong to `zref` rather than here, `zref-clever` is more of a "user side" kind of package than `zref` itself, so hopefully it is not inappropriate to do so.

11 Acknowledgments

TODO Include: `cleveref`, `zref`, Ulrike Fischer, Jonathan P. Spratte, aka ‘Skillmon’, Phe-lype Oleinik, Enrico Gregorio, aka ‘egreg’, Steven B. Segletes, @samcarter, Alan Munn, Florent Rougon, David Carlisle.

12 Change history

A change log with relevant changes for each version, eventual upgrade instructions, and upcoming changes, is maintained in the package’s repository, at <https://github.com/gusbrs/zref-clever/blob/main/CHANGELOG.md>.