The zref-clever package*
User manual

Gustavo Barros†

2022-02-11

Abstract

zref-clever provides an user interface for making LaTeX 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 `\zcref` 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.

EXPERIMENTAL
Please read Section 2 carefully.

*This file describes v0.2.2-alpha, released 2022-02-11.
†`https://github.com/gusbrs/zref-clever`

Contents

1 Introduction

Cross-referencing is an area which lends itself quite naturally to automation. Not only for input convenience but also, and most importantly, for end results consistency. Indeed, the standard LaTeX cross-referencing system – with `\label`, `\ref`, and `\pageref` – is already a form of automation, by relieving us from checking the number of the referenced object, and the page where it lies.

But the plethora of existing features, packages and document classes which, in one way or another, extends this basic functionality is a clear indication of a demand for more automation. Just to name the most popular: cleveref, hyperref, titleref, nameref, varioref, fancyref, and the kernel's `\labelformat`.

However, the standard cross-referencing system stores two, and only two, properties with the label: the printed representation of the counter last incremented with `\refstepcounter` and the page. Of course, out of the mentioned desire to automate more, the need arose to store more information about the label to support this: the title or caption of the referenced object; its counter or, even better, its "type", that is, whether it is a section, chapter, figure, etc.; its hyperlink anchor, and so on. Thus those two property "fields" of the standard label became quite a disputed real state. And the packages in this area of functionality were bound to step on each other's toes as a result.

Out of this conundrum, Heiko Oberdiek eventually developed zref, which implements an extensible referencing system, making the labels store a property list of flexible length, so that new properties can be easily added and queried. However, even when zref can rightfully boast this powerful basic concept and is really quite featureful, with several different modules available, it is fair to say that, for the average user, the package may appear to be somewhat raw. Indeed, for someone who "just wants to make a cross-reference", the user interface of the zref-user module is akin to the standard LaTeX cross-referencing system, and even requires some extra work if you want to have a hyperlinked reference. In other words, zref seems to have focused on infrastructure and on performing a number of specialized tasks with different modules, and a large part of the landscape of automation features available for the standard referencing system was not carried over to zref, neither by the zref itself nor by other packages.

zref-clever tries to cover this gap, by bringing a number of existing features available for the standard referencing system to zref. And the package's name makes it clear that the core of the envisaged feature set is that of cleveref, even though the attempt was less one of replicating functionality per se than that of having it as a successful point of reference, from where we could then try to tap into zref's potential. Indeed, although there is a significant intersection, the features of zref-clever are neither a superset nor a subset of those of cleveref. There are things either of them can do that the other can't. There are also important differences in user interface design. In particular, zref-clever relies heavily on `key=value` interfaces both for general configuration and for centering in a single user command, `\zcref`, as the main entrance for reference making, whose behavior can be modulated by local options.

Considering that zref itself offers the zref-titleref module, and that zref-vario offers integration of zref-clever with varioref, a significant part of the most prominent automation features available to the standard referencing system is thus brought to zref, working under a single consistent underlying infrastructure and user interface. Alas, there are some limitations (see Section 11), and it may be your cup of tea or not. Still, all in

all, hopefully zref-clever can make zref more accessible to the average user, and more interesting to users in general.

2 Warning

This package is in its early days, and should be considered experimental. By this I don't mean I expect it to be "edgy", indeed quite a lot of effort has been put into it so that this is not the case. However, during the initial development, I had to make a number of calls for which I felt I had insufficient information: in relation to features, packages, or classes I don't use much, or to languages I don't know well, user needs I found hard to anticipate etc. Hence, the package needs some time, and some use by more adventurous people, until it can settle down with more conviction. In the meantime, polishing the user interface and the infrastructure have a clear priority over backward compatibility. So, if you choose to use this package, you should be ready to accommodate to eventual upstream changes.

3 Loading the package

As usual:

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

4 Dependencies

zref-clever requires zref, and LaTeX kernel 2021-11-15, 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 zref-titleref, zref-hyperref, and zref-xr modules.

5 User interface

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

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.

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

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⟩}`.

\zcsetup `\zcsetup{⟨options⟩}`

Sets zref-clever's general options (see Sections 6 and 8). 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.

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

Sets type-specific reference format options (see Section 8). 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 9. 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, including `\zref` and `\zpageref`, but notably:

\zlabel `\zlabel{⟨label⟩}`

Sets ⟨*label*⟩ for referencing with `\zref` and, thus, also `\zcref`. `\zlabel` is provided by zref-user and is the counterpart of `\label` for zref's referencing system.

6 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 8). General options can be set globally either as package options at load-time (see Section 3) or by means of `\zcsetup` in the preamble (see Section 5). They can also be set locally with `\zcsetup` along the document or through the optional argument of `\zcref` (see Section 5). 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.

ref
page

The `ref` option controls the label property to which `\zcref` refers to. It can receive zref properties, as long as they are declared, but notably `default`, `page`, `thecounter` and, if zref-titleref is loaded, `title`. The package's default is, well, `default`, which is our standard reference. `thecounter` is a property set by zref-clever and is similar to zref's

`default` property, except that it is not affected by the kernel's `\labelformat`.[1] By default, reference formatting, sorting, and compression are done according to information inferred from the *current counter* (see `currentcounter` option below). Special treatment in these areas is provided for `page`, but not for any other properties. The `page` option is a convenience alias for `ref=page`.

`typeset` `noname` `noref`

When `\zcref` 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 zref-clever. This is optional however, and the `typeset` option controls this behavior. It can receive values `ref`, in which case it typesets only the reference(s), `name`, in which case it typesets only the name(s), or `both`, in which case it typesets, well, both of them. Note that, when value `name` is used, the name is still typeset according to the set of references given to `\zcref`. For example, for multiple references, the plural form is used, capitalization options are honored, etc. Also hyperlinking behaves just *as if* the references were present and, depending on the corresponding options, the name may be linked to the first reference of the type group. The `noname` and `noref` options are convenience aliases for `typeset=ref` and `typeset=name`, respectively.

`sort` `nosort`

The `sort` option controls whether the list of ⟨*labels*⟩ received as argument by `\zcref` should be sorted or not. It is a boolean option, and defaults to `true`. The `nosort` option is a convenience alias for `sort=false`.

`typesort` `notypesort`

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 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, double braced, one for l3keys, so that the second can make the list) can be placed anywhere along the list, to specify the sort priority of any type not included explicitly. 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.

[1] Technical note: the `default` property stores `\@currentlabel`, while the `thecounter` 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, although `\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, `thecounter` may be useful.

comp
nocomp

`\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.

endrange

The `endrange` option provides additional control over how the end reference of a range is typeset, so as to achieve terse ranges. The option can operate in two technically different ways. It may receive one of a number of predefined values, which can process the end reference of the range, comparing it with the beginning reference, to achieve a given end result.[2] Or, it can specify a label property to be used directly, without any processing. The available predefined values are: `ref`, `stripprefix`, `pagecomp`, and `pagecomp2`. `ref` corresponds to the default behavior, and instructs `\zcref` to use whatever property was set at the `ref` option for the end of range reference. `stripprefix` strips the common part at the start of each reference from the end one. `pagecomp` is the equivalent of `stripprefix` for page numbers, it does the same thing, but only if the references are comprised exclusively of Arabic numerals. `pagecomp2` is a variant of `pagecomp` that leaves at least two digits at the end reference (except for a leading zero). If values other than the predefined ones are given to `endrange` they are considered as label properties, as long as they are declared. This property is used to typeset the end of range reference if the label contains it, and if both references would be "compressible" according to the `comp` option, otherwise the property specified by the `ref` option is used. This is useful for things like sub-elements for which we can build a proper abbreviated sub-reference and populate the label with it (some compatibility modules already provide a number of such properties, but other ones can be built with zref, as needed).

range
rangetopair

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. 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`). `\zcref` is smart enough 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". But this behavior can be disabled by setting the `rangetopair` option to `false`.

cap
nocap
capfirst

The `cap` option controls whether the reference type names should be capitalized or not. It can receive values `true` or `false`, and it can also be set for specific reference types or languages (see Section 8). 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`.

[2] For the TEXnically inclined: those values that perform some processing – namely `stripprefix`, `pagecomp`, and `pagecomp2` – fully expand the references (x-type expansion) before comparing them, since it makes sense to perform this task as close as possible to the printed representation of the references. I don't expect this to be a problem in normal use cases, but it does represent a limitation on what the references can contain. In case some control over this is needed, check the `zref-clever/endrange-setup` hook in the code documentation.

abbrev
noabbrev
noabbrevfirst

The abbrev option controls whether to use abbreviated reference type names when they are available. It can receive values true or false, and it can also be set for specific reference types or languages (see Section 8). 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 \zcref 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 \zcref 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 \zcref 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 \zcref{chap:chapter1} and \zcref{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 \zcref when looking for language-specific reference format options (see Section 8). The default value, current, uses the current language, as defined by babel or polyglossia (or english if none of them is loaded). Value main uses the main document 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 9.

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 9.

nudge
nudgeif
nonudge
sg
g

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 \zcref. The nudge option is the main entrance to this feature and takes values true, false, ifdraft, or iffinal. The first two, respectively, enable or disable the "nudging" unconditionally. With ifdraft, nudge keeps quiet when option draft is given to \documentclass, while with iffinal, 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 8). 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 files can store the gender(s) of each type name (this is done for built-in language files, 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(s) for the type name in the language file. Both the comptosing and the gender nudges have a type block as its scope. See Section 9 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 namefont and reffont reference format options in Section 8). 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 LaTeX, 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
vario

The titleref option receives no value and, when given, loads zref's zref-titleref module. Similarly, the vario option loads the zref-vario package. These are a preamble only options. Note that zref-vario loads varioref, which has known load order interaction with other packages, prominently with hyperref. Hence, depending on your document, you may wish to load zref-vario separately, instead of through the 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 8).

`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 8). 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 7). 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 LaTeX 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 associated with a particular label 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 LaTeX 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` LaTeX's `\refstepcounter` sets two variables which potentially affect the `\zlabel` set after it: `\@currentlabel` and `\@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. Also, when using the `currentcounter` option, make sure the setting is duly grouped because, if set, it has precedence over `\@currentcounter` and, contrary to the later, the former is not reset the next time `\refstepcounter` runs. Its default value is, quite naturally, `\@currentcounter`. Since this option only affects how labels are set, it is not available in `\zcref`.

`nocompat` Some packages, document classes, or LaTeX features may require specific support to work with zref-clever (see Section 11). zref-clever tries to make things smoother by covering some of them. Depending on the case, this can take the form of some simple setup for zref-clever, or may involve the use of hooks to external environments or commands and, eventually, a patch or redefinition. By default, all the available compatibility modules are enabled. Should this be undesired or cause any problems in your environment, the option `nocompat` can selectively or completely inhibit their loading. `nocompat` receives a comma separated list of compatibility modules to disable (for the list of available modules and details about each of them, see Section 12). You can disable all modules by setting `nocompat` without a value (or an empty one). This is a preamble only option.

7 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 LaTeX *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 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`,

endnote, note, equation, theorem, lemma, corollary, proposition, definition, proof, result, remark, example, algorithm, listing, exercise, and solution. Therefore, if you are using a language for which zref-clever has built-in support (see Section 9), these reference types are available for use out of the box.[3] And, in any case, it is always easy to setup custom reference types with \zcRefTypeSetup or \zcLanguageSetup (see Sections 5, 8 and 9).

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 LaTeX'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 "name" which zref-clever uses to look for a whole set of type-specific reference format options (see Section 8). 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.

8 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 friendliness". 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.

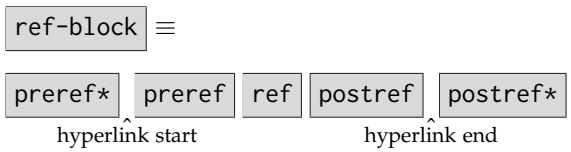
With that in mind, we have settled with an user interface for reference formatting which allows settings to be done in different scopes, with more or less overarching effects, and some precedence rules to regulate the relation of settings given in each of

[3] There 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 language files.

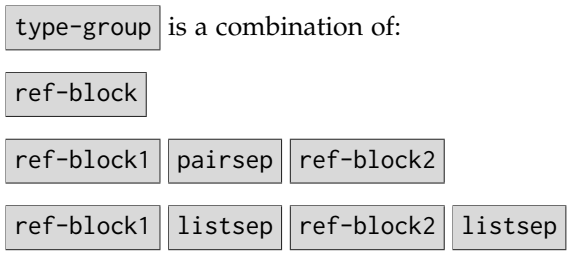
these scopes. There are four scopes in which reference formatting can be specified by the user, in the following precedence order: i) as *general options*; ii) as *type-specific options*; iii) as *language- and type-specific options*; and iv) as *language-specific default options*. Besides those, there's 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. 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.

"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 6).[4] "Type" specific options (ii) are handled by `\zcRefTypeSetup` (see Section 5). "Language" options, whether "type" specific (iii) or "default" (iv) have their user interface in `\zcLanguageSetup`, and have their values populated by the package's built-in language files (see Section 9). 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.

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 `refbounds` option, which receives as value a comma separated list of four items in the form `{preref*,preref,postref,postref*}`, sets the surrounding elements to `ref`.[5] A `ref-block` is built for *each* label given as argument to `\zcref`. When the ⟨*labels*⟩ argument is comprised of multiple labels, each "reference type group", or `type-group` is 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:



[4] The 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`.

[5] As usual, if each of the items contains start or end spaces, or commas anywhere, they must be protected by a pair of braces. However, care is taken that empty items don't need such protection. So you can set, for example, something like `refbounds={(,,,)}` to get parentheses around your references, outside the hyperlink.

		General	Type	Language	
				Type	Default
		(i)	(ii)	(iii)	(iv)
Typesetting (necessarily not type-specific)	`tpairsep`	•			•
	`tlistsep`	•			•
	`tlastsep`	•			•
	`notesep`	•			•
Typesetting (possibly type-specific)	`namesep`	•	•	•	•
	`pairsep`	•	•	•	•
	`listsep`	•	•	•	•
	`lastsep`	•	•	•	•
	`rangesep`	•	•	•	•
	`refbounds`	•	•	•	•
Typesetting (necessarily type-specific)	`Name-sg`		•	•	
	`name-sg`		•	•	
	`Name-pl`		•	•	
	`name-pl`		•	•	
	`Name-sg-ab`		•	•	
	`name-sg-ab`		•	•	
	`Name-pl-ab`		•	•	
	`name-pl-ab`		•	•	
Font	`namefont`	•	•	•	•
	`reffont`	•	•	•	•
Other	`cap`	•	•	•	•
	`abbrev`	•	•	•	•
	`endrange`	•	•	•	•
	`rangetopair`	•	•	•	•

Table 1: Reference format options and their scopes

… `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":

`type-block` ≡

`type-name` `namesep` `type-group`

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:

`type-block-group` is one of:

`type-block`

`type-block1` `tpairsep` `type-block2`

`type-block1` `tlistsep` `type-block2` `tlistsep` `type-block3` ...

... `type-blockN-1` `tlastsep` `type-blockN`

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

A complete `\zcref` reference:

`type-block-group` `notesep` `note`

Reference format options can yet be divided in three general categories: i) "typesetting" options, the ones which we have seen thus far, as "building blocks" of the reference; ii) "font" options, which control font attributes of parts of the reference, namely `namefont` and `reffont`; and iii) "other" options. "Typesetting" options are intended exclusively for typesetting material: things you expect to see in the output of your references. "Font" options set the font, respectively, for the `type-name` and for `ref` (to set the font for the whole reference, see the `font` option in Section 6). These options 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 language file 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. 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} ,
```

8.1 Advanced reference formatting

The reference format options discussed above and presented in Table 1 should suffice for most needs. However, if more fine-grained control of the reference format is needed, this can be achieved through a more detailed specification of `refbounds` for the different cases in which they may occur when a reference is processed. The options available for this purpose are presented in Table 2.

The "base" options are the actually operative ones, while the "derived" options are convenience aliases to set multiple base options in one go. In the naming scheme of these options, as is easy to presume, "`first`" refers to the first reference of a type-block, "`mid`" to the middle ones, and "`last`" to the last reference of the type-block. Less obviously, but hopefully still mnemonic enough, "`sg`" stands for "single", "`pb`" and "`pe`" for "pair begin" and "pair end", and finally "`rb`" and "`re`" for "range begin" and "range end". Each of them which receives as value a comma separated list of four items in the form `{preref*,preref,postref,postref*}`, just like `refbounds`.

The base options are mutually exclusive, which means, for example, that it is not sufficient to set `refbounds-first` to define the behavior of all first references of a type block. `refbounds-first` is the value used for the first reference when not single, not the beginning of a pair, and not the beginning of a range. Setting a group of them is the purpose of the derived options. Each of these sets all options under it. Some examples. `+refbounds-first` sets `refbounds-first`, `refbounds-first-sg`, `refbounds-first-pb`, and `refbounds-first-rg`. In turn, `+refbounds-first-rb` sets `refbounds-first-rb` and `refbounds-first-rb`. And quite conveniently, `+refbounds` sets `+refbounds-first`, `+refbounds-mid`, and `+refbounds-last`, it is hence sufficient to set it to define the behavior of what is typeset around all references for the whole type-block. As you probably guessed by now, the `refbounds` option presented in Table 1 is an alias of `+refbounds`.

		General	Type	Language	
				Type	Default
		(i)	(ii)	(iii)	(iv)
Base options	`refbounds-first`	•	•	•	•
	`refbounds-first-sg`	•	•	•	•
	`refbounds-first-pb`	•	•	•	•
	`refbounds-first-rb`	•	•	•	•
	`refbounds-mid`	•	•	•	•
	`refbounds-mid-rb`	•	•	•	•
	`refbounds-mid-re`	•	•	•	•
	`refbounds-last`	•	•	•	•
	`refbounds-last-pe`	•	•	•	•
	`refbounds-last-re`	•	•	•	•
Derived options (groups)	`+refbounds-first`	•	•	•	•
	`+refbounds-mid`	•	•	•	•
	`+refbounds-last`	•	•	•	•
	`+refbounds-rb`	•	•	•	•
	`+refbounds-re`	•	•	•	•
	`+refbounds`	•	•	•	•

Table 2: Advanced reference format options and their scopes

Given that base and derived options are actually setting the same group of underlying options (the base ones), the order in which they are given is relevant: the last one prevails. The idea here is to use first the derived options to set some general defaults, and then change one or another base option to handle exceptions as needed. Of course, how best to use them depends on the case.

9 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 see Section 8). 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 "language file" for it, most use cases will likely be covered by the `lang` option (see Section 6), and its values `current` and `main`. When the `lang` option is set to `current` or `main`, zref-check will use, respectively, the *current* or *main* language of the document, as defined by babel or polyglossia.[6] Users can also set `lang` to a specific

[6]Technically, zref-clever uses `\languagename` and `\bbl@main@language` for babel, and `\babelname` and `\mainbabelname` for polyglossia, which boils down to zref-clever always using *babel names* internally, regardless of which language package is in use. Indeed, an acquainted user will note that Table 3 contains only babel language names.

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

Table 3: Declared languages and aliases

language directly, in which case babel and polyglossia are disregarded. zref-clever provides a number of built-in "language files", for the languages listed in Table 3, which also includes the declared aliases to those languages.

zref-clever's "language files" are loaded sparingly and lazily. A language file for a single language – that specified by user options in the preamble, which by default is the current document language – is loaded at `begindocument`. If any other language file is needed, it is loaded on the fly, if and when required. Of course, in either case, conditioned on availability. In sum, zref-clever loads as little as possible, but allows for convenient on the fly loading of language files if the values are indeed required, without users having to worry about it at all.

But if the built-in language files 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`.[7]

`\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 6) is disregarded, and `\zcref` always uses the capitalized type name forms. This means that language files for languages with such a trait can be halved in size, and that user customization

[7] Needless to say, if you'd like to contribute a language file 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	-
dutch	-	f,m,n	-

Table 4: Options for declared languages

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 4 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 8), 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 language-specific but not type-specific option values (see Section 8). 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*⟩). For `types` which have multiple valid genders for a given language, the option can also receive a comma separated list. `\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 language files 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 6). 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 6). 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 set in the language file or with \zcLanguageSetup, 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 "poor 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 sounds ugly, at best. 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 6), 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 language files 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 couse, this means that the gender markup has to be supplied in the document at each reference. And given 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.

10 How-tos

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

10.1 Extended page references (varioref)

Task Make cross-references to pages which are sensitive to the relative position between the reference and the label being referred to using varioref.

zref-vario, which can be enabled with package option `vario`, offers a layer of compatibility with varioref and provides `\z...` counterparts for the latter's main reference commands.

How-to 1: zref-vario

```
\documentclass{article}
\usepackage[vario]{zref-clever}
\begin{document}
\section{Section 1}
\zlabel{sec:section-1}
\begin{figure}
  A figure.
  \caption{Figure 1}
  \zlabel{fig:figure-1}
\end{figure}
\begin{figure}
  Another figure.
  \caption{Figure 2}
  \zlabel{fig:figure-2}
\end{figure}
\zvref[S]{sec:section-1}
\zvpageref{fig:figure-1}
\zvrefrange{fig:figure-1}{fig:figure-2}
\zvpagerefrange{fig:figure-1}{fig:figure-2}
\zfullref{fig:figure-1}
\end{document}
```

10.2 \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 equaly 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 ntheorem. For thmtools' \declaretheorem, 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 \zcref. 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}[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:

How-to 2: \newtheorem, simple case

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

How-to 3: \newtheorem, simple case

```
\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}
\zcref{lemma-1}
\end{document}
```

Shared counter

Task Setup up two new theorem environments created with `\newtheorem` to be referred to with `\zcref`. 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 5), `\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}`.

How-to 4: `\newtheorem`, shared counter

```
\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 language options already provided by the package (assuming the language is supported).

How-to 5: `\newtheorem`, custom type

```
\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}
```

10.3 newfloat

Task Setup a new float environment created with newfloat to be referred to with `\zcref`. None of zref-clever built-in reference types is adequate for my needs.

The case here is pretty much the same as that for `\newtheorem` with a custom type. Hence, we need to setup a corresponding type, for which providing the "type names" should normally suffice. Note that, as far as zref-clever is concerned, there's nothing specific to the newfloat package in the setup, the same procedure can be used with memoir's `\newfloat` command or with the float, floatrow, and trivfloat packages.

How-to 6: newfloat

```
\documentclass{article}
\usepackage{newfloat}
\DeclareFloatingEnvironment{diagram}
\usepackage{zref-clever}
\zcRefTypeSetup{diagram}{
  Name-sg = Diagram ,
  name-sg = diagram ,
  Name-pl = Diagrams ,
  name-pl = diagrams ,
}
\begin{document}
\section{Section 1}
\begin{diagram}
  A diagram.
  \caption{A diagram}
  \zlabel{diagram-1}
\end{diagram}
```

```
\zcref{diagram-1}
\end{document}
```

10.4 amsmath

Task Make references to amsmath display math environments with `\zcref`.

Given how amsmath's display math environments work, they need to handle `\label` specially, and this support is quite hard-wired into the environments, but it is not extended to `\zlabel`. zref-clever's `amsmath` compatibility module provides that a `\label` used inside these environments set both a regular `\label` and a `\zlabel`, so that we can refer to the equations with both referencing systems. Note the use of `\zlabel` for `subequations` though. For more details, see the description of the `amsmath` compatibility module at Section 12.

How-to 7: amsmath

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{zref-clever}
\usepackage{hyperref}
\begin{document}
\section{Section 1}
\begin{equation}\label{eq:1}
  A^{(1)}_1 =\begin{cases} n!,&\text{if }1 =1\\
    0,&\text{otherwise}.\end{cases}
\end{equation}
\begin{equation*} \tag{foo}\label{eq:2}
  A^{(1)}_1 =\begin{cases} n!,&\text{if }1 =1\\
    0,&\text{otherwise}.\end{cases}
\end{equation*}
\begin{subequations}\zlabel{eq:3}
  \begin{align}
    A+B&=B+A\\
    C&=D+E\label{eq:3b}\\
    E&=F
  \end{align}
\end{subequations}
\zcref{eq:1, eq:2, eq:3, eq:3b}
\end{document}
```

10.5 listings

Task Make references to a `lstlisting` environment from the listings package with `\zcref`.

Being `lstlisting` a verbatim environment, setting labels inside it requires special treatment. zref-clever's `listings` compatibility module provides that a label given to the `label` option gets set with both a regular `\label` and a `\zlabel`, so that we can refer to it with both referencing systems. Setting labels for specific lines of the environment can

be done with \zlabel directly, subject to the same escaping as for the standard \label. For more details, see the description of the listings compatibility module at Section 12.

How-to 8: listings

```
\documentclass{article}
\usepackage{listings}
\usepackage{zref-clever}
\usepackage{hyperref}
\begin{document}
\section{Section 1}
\lstset{escapeinside={(*@}{@*)}, numbers=left, numberstyle=\tiny}
\begin{lstlisting}[caption={Useless code}, label=lst:1]
  for i:=maxint to 0 do
  begin
      { do nothing }(*@\zlabel{ln:1.1}@*)
  end;
\end{lstlisting}
\zcref{lst:1, ln:1.1}
\end{document}
```

10.6 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 standard 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.

How-to 9: enumitem

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

10.7 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.

How-to 10: zref-xr

`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}
```

11 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 LaTeX 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 potencial lack of support for zref itself. Broadly speaking, what zref-clever does is setup zref so that its `\zref@newlabels` contains 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. As far as my experience goes, the following label setting requirements can be potentially problematic and are not necessarily granted for `\zlabel`:

1. One must be able to call `\zlabel`, directly or indirectly, at the appropriate scope/location so as to set the label.

2. When `\zlabel` is set, it must see a correct value of `\@currentcounter`.

As to 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 other places may be problematic too, for example, amsmath display math environments also handle `\label` specially and the same work is not done for `\zlabel`.

Regarding the second, a correctly set `\@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. Since the 2020-10-01 LaTeX release, `\@currentcounter` is set by `\refstepcounter` alongside `\@currentlabel` and, since the 2021-11-15 release, the support for `\@currentcounter` has been further extended in the kernel. Hence, as long as kernel features are involved, or as long as `\refstepcounter` is the tool used for the purpose of reference setting, `\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 work-arounds when the value of `\@currentcounter` 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 is provided by the relevant packages and classes or not. Or, failing that, whether zref-clever is able to provide some specific support when a reasonable way to do so is within reach.

12 Compatibility modules

This section gives a description of each compatibility module provided by zref-clever. These modules intend to smooth the interaction of LaTeX features, document classes, and packages with zref-clever and zref, and they can be selectively or completely disabled with the option `nocompat` (see Section 6). This set is not to be confused with "the list of packages or classes supported by zref-clever". In most circumstances, things should just work out of the box, and need no specific handling. These are just the ones for which some special treatment was required. Of course, this effort is bound to be incomplete (see Section 11).

The purpose of outlining to some extend what the compatibility modules do is twofold. First, some of them require usage adjustments for label setting, which must be somehow conveyed in this documentation. Second, the kind and degree of intervention in external code varies significantly for each module, and since this is an area of potential friction, a minimum of information for the users to judge whether they want to leave these modules enabled or not is due. For this reason, this is also a little technical, but for full details, see the code documentation.

`appendix` The `\appendix` command provided by many document classes is normally used to change the behavior of the sectioning commands after that point. Usually, depending on the class, the changes that interest us involve using `\@Alph` for numbering and `\appendixname` for chapter's names. In sum, we'd like to refer to the appendix sectioning commands as "appendices" rather than "chapters" or "sections". Since the sectioning commands are the same as before `\appendix`, and so are their underlying counters, we must configure the counter type of the sectioning counters to `appendix`. And this is what this compatibility module does, and it uses a ltcmdhooks hook on `\appendix` for the purpose. Hence, this module applies to any document class or package which provides that command.

`appendices` This module implements support for the `appendices` and `subappendices` environments provided by the appendix package, and also by memoir. The task is the same as for the `appendix` module: set proper counter types for the sectioning counters. This module

employs environment hooks to appendices and subappendices and a command hook to \appendix for the purpose.

memoir The memoir class implements several features with one or another implication for cross-referencing, usually bearing just the standard referencing system in mind, and related mainly to captions, subfloats, and notes. This compatibility module tries to adjust zref-clever to these features with support for the following: i) set counter types for counters subfigure, subtable, and poemline (used in the verse environment); ii) configure resetting behavior (counterresetby option) for subfigure and subtable counters; iii) provide that the ⟨*label*⟩ arguments to environments sidecaption and sidecontcaption and commands \bitwonumcaption, \bionenumcaption, and \bicaption *also* set a \zlabel of the same name; iv) provide the zref property "subcaption" so that we can refer to, for example, \zcref[ref=subcaption]{subcap-1} to emulate the functionality of memoir's \subcaptionref; v) provide that \footnote, \verbfootnote, \sidefootnote, and \pagenote get a proper currentcounter set; and v) set counter types for counters sidefootnote and pagenote. Naturally, the sheer number of features that required some specific support implies that some gymnastics were needed here. But the most sensitive changes are: i) a local redefinition of \label inside the environments sidecaption and sidecontcaption and the commands \bitwonumcaption, \bionenumcaption, and \bicaption; and ii) the use of ltcmdhooks command hooks on \bitwonumcaption, \bionenumcaption, \bicaption, \@memsubcaption, \@makefntext, and \@makesidefntext.

KOMA The KOMA-Script document classes are much more strict than memoir in using standard mechanisms for cross-reference related features, so that very little adjustment is needed here. Only the environments captionbeside and captionofbeside require special treatment. And, though they do use \refstepcounter under the hood, since they are environments, we don't get to see reference variables outside them. \@currentlabel is smuggled out of the group, but not \@currentcounter, so we must arrange for the later to have a correct value outside the caption environments too. Environment hooks are used for the purpose, essentially setting \@currentcounter to \@captype, which would be the value the captioning infrastructure would set for it.

amsmath amsmath's display math environments have their contents processed twice, once for measuring and the second does the final typesetting. Hence, amsmath needs to handle \label specially inside these environments, otherwise we'd have duplicate labels all around, and indeed it does redefine \label locally inside them. Alas, the same treatment is not granted to \zlabel. However, \label is set basically to *store* the value of the argument inside the environments, and its global meaning is kept in \ltx@label which is called at the appropriate time to actually set the label. So, what zref-clever does is redefine \ltx@label to set both a regular \label and a \zlabel for the same ⟨*label*⟩ (which does not generate a duplicate label problem since the referencing systems are independent). Therefore, you must use \label (not \zlabel) inside amsmath's display math environments, and this compatibility module arranges that you can also refer to that label with zref and, thus, with \zcref as well. The following environments are subject to this usage restriction: equation, align, alignat, flalign, xalignat, gather, multline, and their respective starred versions. In particular, the same is not the case for the subequations environment, inside which \zlabel works as usual (to refer to the main equation number, that is, right after \begin{subequations}). See the How-to 7 for an usage example. The module also ensures proper currentcounter values are in

place for the display math environments, for which it uses environment hooks, and sets the font of equation references to `\upshape`, following amsmath's `\eqref`. Finally, the module also provides a `subeq` property, for display math environments used inside the `subequations` environment, which can be used to refer to them directly with the `ref` option, or to build terse ranges with the `endrange` option. Note that zref-clever is not the only package to redefine `\ltx@label`, and compatibility problems may arise if this module is used with such packages or with document classes that do the same. In case of trouble, you can load zref-clever with option `nocompat=amsmath` and either use the standard referencing system's facilities to refer to amsmath's equations or check the code documentation for the technique used for this (which is pretty standard) and adapt it to your case. Given that any trouble that may arise here is one of the proper "timing" of the redefinition, it should not be particularly complicated to make such adjustments.

`mathtools` mathtools has a feature to show the numbers only for those equations actually referenced (with `\eqref` or `\refeq`), which is enabled by the `showonlyrefs` option. This compatibility module adds support for this feature, such that equation references made with `\zcref` also get marked as "referenced" for mathtools, when the option is active, of course. The module uses a couple of mathtools functions, but does not need to redefine or hook into anything, everything is handled on zref-clever's side.

`breqn` This compatibility module only sets proper `currentcounter` values for the environments `dgroup`, `dmath`, `dseries`, and `darray`, and uses environment hooks for the purpose. Note that these environments offer a `label` option for label setting, but implements it in a way that is somewhat tricky to grab for our purposes, so we don't do it. However, breqn's documentation says the following about the use of `\label` inside its environments: "Use of the normal `\label` command instead of the `label` option works, I think, most of the time (untested)". My light testing suggests the same is true for `\zlabel`, which can then be used directly in these environments. But the "second class" status for this use is the one granted by breqn, there's not much we can do here. If you have any trouble with this, you may wish to check Work-around 1.

`listings` Being `lstlistings` environments what they are, one cannot simply place a label inside them without special treatment. listings arranges to this by providing a `label` option for setting a label for the whole environment, and a properly escaped label command inside the environment can be used to refer to the line number. While the later can also be used to set a `\zlabel` to make line number references, the `label` option is catered to the standard labels exclusively. Hence, this compatibility module uses the same label name – which luckily and atypically is stored in a variable – and sets a `\zlabel` as well. This is done by using listings' `PreInit` hook, se we don't have to redefine or tamper with anything for the purpose. Besides this, the module also sets appropriate `countertype`, `counterresetby` and `currentcounter` values for the listings' counters: `lstlisting` and `lstnumber`. See the How-to 8 for an usage example.

`enumitem` LaTeX's `enumerate` environment requires some special treatment from zref-clever, since its resetting behavior is not stored in the standard way, and the counters' names, given they are numbered by level, do not map to the reference type naturally. This is done by default for the up to four levels of nested `enumerate` environments the kernel offers. enumitem, though, allows one to increase this maximum list depth for `enumerate` and, if this is done, setup for these deeper nesting levels have also to be taken care of, and that's what this compatibility module does. All settings here are internal to zref-clever, no hooks or redefinitions are needed, we just check the existing pertinent

counters at `begindocument`, and supply settings for them. Of course, this means that only enumitem's settings done in the preamble will be visible to the module and provided for.

`subcaption` This compatibility module sets appropriate `countertype` and `counterresetby` for the `subfigure` and `subtable` counters, and provides the zref property "`subref`" so that we can refer to, for example, `\zcref[ref=subref]{⟨label⟩}` to emulate the functionality of subcaption's `\subref`. The later feature uses the `\caption@subtypehook` provided by caption to locally add the `subref` property to zref's main property list.

`subfig` This module just sets appropriate `countertype` and `counterresetby` for the `subfigure` and `subtable` counters.

13 Work-arounds

As should be clear by now, the use of zref's `\zlabel` and thus of zref-clever may occasionally require some adjustments, since it does not enjoy the universal support the standard referencing system does. The compatibility modules presented in Section 12 go a long way in ensuring the user has to worry very little about it, but they cannot hopefully go all the way. Not only because this kind of support will never be exhaustive, but also since, sometimes, given the way certain features are implemented by packages or document classes, there may not be a reasonable way to provide this support, from our side. But, still, most of the time, it is still "viable" to get there, if one really wishes to do so. So, this section keeps track of some known recipes, which I don't think belong in zref-clever itself, but which you may choose to use. Note that this list is intended to spare users from having to reinvent the wheel every time someone needs something of the sort, but from zref-clever's perspective, their status is "not supported".

breqn

As mentioned in `breqn`'s compatibility module (Section 12), breqn's math environments `dgroup`, `dmath`, `dseries`, and `darray` offer a `label` option (plus `labelprefix`) for the purpose of label setting. Setting a `\zlabel` alongside with a regular `\label` based on that option requires redefining some of breqn internals. Also, breqn does not use `\refstepcounter` to increment the equation counters and, as a result, fails to set hyperref anchors for the equations (thus affecting standard labels too). The example below also provides to that.[8]

Work-around 1: breqn

```
\documentclass{article}
\usepackage{zref-clever}
\usepackage{breqn}
\makeatletter
\define@key{breqn}{label}{%
  \edef\next@label{%
    \noexpand\label{\next@label@pre#1}%
    \noexpand\zlabel{\next@label@pre#1}}%
  \let\next@label@pre\@empty}
```

[8]The later work-around thanks to Heiko Oberdiek, at `https://tex.stackexchange.com/a/241150`.

```
\makeatother
\usepackage{hyperref}
\usepackage{etoolbox}
\makeatletter
\patchcmd\eq@setnumber{\stepcounter}{\refstepcounter}{}{%
  \errmessage{Patching \noexpand\eq@setnumber failed}}
\makeatother
\begin{document}
\section{Section 1}
\begin{dmath}[label={eq:1}]
  f(x)=\frac{1}{x} \condition{for $x\neq 0$}
\end{dmath}
\begin{dmath}[labelprefix={eq:},label={2}]
  H_2^2 = x_1^2 + x_1 x_2 + x_2^2 - q_1 - q_2
\end{dmath}
\zcref{eq:1, eq:2}
\end{document}
```

Of course, you can always adapt things to your needs and, e.g., make the `label` option set just `\zlabel` instead of both labels, or create a separate `zlabel` option.

14 Acknowledgments

zref-clever would not be possible without other people's previous work and help.

Heiko Oberdiek's zref, now maintained by the Oberdiek Package Support Group, is the underlying infrastructure of this package. The potential of its basic concept and the solid implementation were certainly among the reasons I've chosen to venture into these waters, to start with. And I believe they will remain one of the main assets of zref-clever as it matures.

The name of the package makes no secret that a major inspiration for the kind of "feel" I strove to achieve has been Toby Cubitt's cleveref. Indeed, I have been an user of cleveref for several years, and a happy one at that. But the role cleveref played in the development of zref-clever extends beyond the visible influence in the design of user facing functionality. Some technical solutions and, specially, the handling of support for other packages were a valuable reference. Hence, the accumulated experience of cleveref allowed for zref-clever to start on a more solid foundation than would otherwise be the case.

The long term efforts of the LaTeX3 `\Team` around expl3 and xparse have also left their marks in this package. By implementing powerful tools and smoothing several regular programming tasks, they have certainly reduced my entry barrier to LaTeX programming and enabled me to develop this package with a significantly reduced effort. And, given the constraints of my abilities, the result is no doubt much better than it would be in their absence.

Besides these more general acknowledgments, a number of people have contributed to zref-clever, whether they are aware of it or not. Suggestions, ideas, solutions to problems, bug reports or even encouragement were generously provided by (in chronological order): Ulrike Fischer, Phelype Oleinik, Enrico Gregorio, Steven B. Segletes, Jonathan P. Spratte, David Carlisle, Frank Mittelbach, 'samcarter', Alan Munn, Florent Rougon, Denis Bitouzé, Marcel Krüger, Jürgen Spitzmüller, and 'niluxv'.

The package's language files have been provided or improved thanks to: Denis Bitouzé (French), François Lagarde (French), and 'niluxv' (Dutch).

If I have inadvertently left anyone off the list I apologize, and please let me know, so that I can correct the oversight.

Thank you all very much!

15 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`. An archive of historical versions of the package is also kept at `https://github.com/gusbrs/zref-clever/releases`.