Package ‘vetr’

January 6, 2023

Title Trust, but Verify

Description Declarative template-based framework for verifying that objects meet structural requirements, and auto-composing error messages when they do not.

Version 0.2.15

Depends R (>= 3.2.0)

License GPL (>= 2)

URL https://github.com/brodieG/vetr

BugReports https://github.com/brodieG/vetr/issues

VignetteBuilder knitr

Imports methods, stats, utils

Suggests knitr, rmarkdown, unitizer

RoxygenNote 7.2.3

NeedsCompilation yes

Author Brodie Gaslam [aut, cre],
Paxdiablo [cph] (Hash table implementation in src/pfhash.h),
R Core Team [cph] (Used/adapted several code snippets from R sources, see src/r-copied.c)

Maintainer Brodie Gaslam <brodie.gaslam@yahoo.com>

Repository CRAN

Date/Publication 2023-01-06 18:50:12 UTC

R topics documented:

`vetr-package` *Trust, but Verify*

Description

Declarative template-based framework for verifying that objects meet structural requirements, and auto-composing error messages when they do not.

`abstract` *Turn S3 Objects Into Templates*

Description

Create templates for use by `alike`. Currently somewhat experimental; behavior may change in future.

Usage

```
abstract(x, ...)

## S3 method for class 'data.frame'
abstract(x, ...)

## Default S3 method:
abstract(x, ...)

## S3 method for class 'array'
abstract(x, ...)

## S3 method for class 'matrix'
abstract(x, ...)

## S3 method for class 'list'
abstract(x, ...)

## S3 method for class 'lm'
abstract(x, ...)
```

```
## S3 method for class 'environment'
abstract(x, ...)

## S3 method for class 'ts'
abstract(x, what = c("start", "end", "frequency"), ...)
```

Arguments

`x`	the object to abstract
`...`	arguments for methods that require further arguments
`what,`	for time series which portion of the `ts` attribute to abstract, by default all three are abstracted, but you can select, any one, two, or all

Details

`abstract` is intended to create templates for use by `alike`. The result of abstraction is often a partially specified object. This type of object may not be suited for use in typical R computations and may cause errors (or worse) if you try to use them as normal R objects.

There is no guarantee that the `abstracted` object is suitable for use as a template to `alike` as is. You may need to modify it further so that it suits your purposes.

`abstract` is an S3 generic. The default method will dispatch on implicit classes, so if you attempt to `abstract` an object without an explicit `abstract` method, it will get abstracted based on its implicit class. If you define your own `abstract` method and do not wish further abstraction based on implicit classes do not use `NextMethod`.

S4 and RC objects are returned unchanged.

Value

abstracted object

Time Series

`alike` will treat time series parameter components with zero in them as wildcards. This function allows you to create these wild card time series attributes since R does not allow direct creation/modification of `ts` attributes with zero values.

Make sure you do not try to use the templates you create with this for anything other than as `alike` templates since the result is likely undefined given R expects non zero values for the `ts` attribute and attempts to prevent such attributes.

Examples

```
iris.tpl <- abstract(iris)
alike(iris.tpl, iris[1:10, ])
alike(iris.tpl, transform(iris, Species=as.character(Species)))

abstract(1:10)
abstract(matrix(1:9, nrow=3))
abstract(list(1:9, runif(10)))
```

alike *Compare Object Structure*

Description

Similar to all.equal, but compares object structure rather than value. The target argument defines a template that the current argument must match.

Usage

```
alike(target, current, env = parent.frame(), settings = NULL)
```

Arguments

target	the template to compare the object to
current	the object to determine alikeness of to the template
env	environment used internally when evaluating expressions; currently used only when looking up functions to match.call when testing language objects, note that this will be overridden by the environment specified in settings if any, defaults to the parent frame.
settings	a list of settings generated using vetr_settings, NULL for default

Value

TRUE if target and current are alike, character(1L) describing why they are not if they are not

alikeness

Generally speaking two objects are alike if they are of the same type (as determined by type_alike) and length. Attributes on the objects are required to be recursively alike, though the following attributes are treated specially: class, dim, dimnames, names, row.names, levels, tsp, and srcref.

Exactly what makes two objects alike is complex, but should be intuitive. The best way to understand "alikeness" is to review the examples. For a thorough exposition see the vignette.

Note that the semantics of alikeness for language objects, formulas, and functions may change in the future.

See Also

type_alike, type_of, abstract, vetr_settings for more control of settings

Examples

```
## Type comparison
alike(1L, 1.0)         # TRUE, because 1.0 is integer-like
alike(1L, 1.1)         # FALSE, 1.1 is not integer-like
alike(1.1, 1L)         # TRUE, by default, integers are always considered real

alike(1:100, 1:100 + 0.0)  # TRUE

## We do not check numerics for integerness if longer than 100
alike(1:101, 1:101 + 0.0)

## Scalarness can now be checked at same time as type
alike(integer(1L), 1)           # integer-like and length 1?
alike(logical(1L), TRUE)        # logical and length 1?
alike(integer(1L), 1:3)
alike(logical(1L), c(TRUE, TRUE))

## Zero length match any length of same type
alike(integer(), 1:10)
alike(1:10, integer())   # but not the other way around

## Recursive objects compared recursively
alike(
  list(integer(), list(character(), logical(1L))),
  list(1:10, list(letters, TRUE))
)
alike(
  list(integer(), list(character(), logical(1L))),
  list(1:10, list(letters, c(TRUE, FALSE)))
)

## `NULL` is a wild card when nested within recursive objects
alike(list(NULL, NULL), list(iris, mtcars))
alike(NULL, mtcars)    # but not at top level

## Since `data.frame` are lists, we can compare them recursively:
iris.fake <- transform(iris, Species=as.character(Species))
alike(iris, iris.fake)

## we even check attributes (factor levels must match)!
iris.fake2 <- iris
levels(iris.fake2$Species) <- c("setosa", "versicolor", "africana")
alike(iris, iris.fake2)

## We can use partially specified objects as templates
iris.tpl <- abstract(iris)
str(iris.tpl)
alike(iris.tpl, iris)
## any row sample of iris matches our iris template
alike(iris.tpl, iris[sample(1:nrow(iris), 10), ])
## but column order matters
alike(iris.tpl, iris[c(2, 1, 3, 4, 5)])
```

```
## 3 x 3 integer
alike(matrix(integer(), 3, 3), matrix(1:9, nrow=3))
## 3 x 3, but not integer!
alike(matrix(integer(), 3, 3), matrix(runif(9), nrow=3))
## partial spec, any 3 row integer matrix
alike(matrix(integer(), 3), matrix(1:12, nrow=3))
alike(matrix(integer(), 3), matrix(1:12, nrow=4))
## Any logical matrix (but not arrays)
alike(matrix(logical()), array(rep(TRUE, 8), rep(2, 3)))

## In order for objects to be alike, they must share a family
## tree, not just a common class
obj.tpl <- structure(TRUE, class=letters[1:3])
obj.cur.1 <-  structure(TRUE, class=c("x", letters[1:3]))
obj.cur.2 <-  structure(TRUE, class=c(letters[1:3], "x"))

alike(obj.tpl, obj.cur.1)
alike(obj.tpl, obj.cur.2)

## You can compare language objects; these are alike if they are self
## consistent; we don't care what the symbols are, so long as they are used
## consistently across target and current:

## TRUE, symbols are consistent (adding two different symbols)
alike(quote(x + y), quote(a + b))
## FALSE, different function
alike(quote(x + y), quote(a - b))
## FALSE, inconsistent symbols
alike(quote(x + y), quote(a + a))
```

`all_bw` *Verify Values in Vector are Between Two Others*

Description

Similar to `isTRUE(all(x >= lo & x <= hi))` with default settings, except that it is substantially faster and returns a string describing the first encountered violation rather than FALSE on failure.

Usage

```
all_bw(x, lo = -Inf, hi = Inf, na.rm = FALSE, bounds = "[]")
```

Arguments

`x`	vector logical (treated as integer), integer, numeric, or character. Factors are treated as their underlying integer vectors.
`lo`	scalar vector of type coercible to the type of `x`, cannot be NA, use `-Inf` to indicate unbounded (default).

hi
: scalar vector of type coercible to the type of x, cannot be NA, use Inf to indicate unbounded (default), must be greater than or equal to lo.

na.rm
: TRUE, or FALSE (default), whether NAs are considered to be in bounds. Unlike with all(), for all_bw na.rm=FALSE returns an error string if there are NAs instead of NA. Arguably NA, but not NaN, should be considered to be in [-Inf,Inf], but since NA < Inf is NA we treat them as always being out of bounds.

bounds
: character(1L) for values between lo and hi:
  - "[]" include lo and hi
  - "()" exclude lo and hi
  - "(]" exclude lo, include hi
  - "[)" include lo, exclude hi

Details

You can modify the comparison to be strictly greater/less than via the bounds parameter, and the treatment of NAs with na.rm. Note that NAs are considered to be out of bounds by default. While technically incorrect since we cannot know whether an NA value is in or out of bounds, this assumption is both conservative and convenient. Zero length x will always succeed.

If x and lo/hi are different types, lo/hi will be coerced to the type of x. When lo/hi are numeric and x is integer, if lo/hi values are outside of the integer range then that side will be treated as if you had used -Inf/Inf. -Inf and Inf mean lo and hi will be unbounded for all data types.

Value

TRUE if all values in x conform to the specified bounds, a string describing the first position that fails otherwise

Examples

```
all_bw(runif(100), 0, 1)
all_bw(runif(100) * 2, 0, 1)
all_bw(NA, 0, 1)              # This is does not return NA
all_bw(NA, 0, 1, na.rm=TRUE)

vec <- c(runif(100, 0, 1e12), Inf, 0)
all_bw(vec, 0)      # All +ve numbers
all_bw(vec, hi=0)   # All -ve numbers
all_bw(vec, 0, bounds="(]") # All strictly +ve nums
all_bw(vec, 0, bounds="[)") # All finite +ve nums
```

`bench_mark` *Lightweight Benchmarking Function*

Description

Evaluates provided expression in a loop and reports mean evaluation time. This is inferior to `microbenchmark` and other benchmarking tools in many ways except that it has zero dependencies or suggests which helps with package build and test times. Used in vignettes.

Usage

```
bench_mark(..., times = 1000L, deparse.width = 40)
```

Arguments

`...`	expressions to benchmark, are captured unevaluated
`times`	how many times to loop, defaults to 1000
`deparse.width`	how many characters to deparse for labels

Details

Runs `gc()` before each expression is evaluated. Expressions are evaluated in the order provided. Attempts to estimate the overhead of the loop by running a loop that evaluates `NULL` the `times` times.

Unfortunately because this computes the average of all iterations it is very susceptible to outliers in small sample runs, particularly with fast running code. For that reason the default number of iterations is one thousand.

Value

NULL, invisibly, reports timings as a side effect as screen output

Examples

```
bench_mark(runif(1000), Sys.sleep(0.001), times=10)
```

`nullify` *Set Element to NULL Without Removing It*

Description

This function is required because there is no straightforward way to over-write a value in a list with NULL without completely removing the entry from the list as well.

Usage

```
nullify(obj, index)

## Default S3 method:
nullify(obj, index)
```

Arguments

`obj`	the R object to NULL a value in
`index`	an indexing vectors of values to NULL

Details

This returns a copy of the object modified with null slots; it does not modify the input argument.

Default method will attempt to convert non-list objects to lists with `as.list`, and then back to whatever they were by using a function with name `paste0("as.", class(obj)[[1L]])` if it exists and works. If the object cannot be coerced back to its original type the corresponding list will be returned.

If this is not appropriate for your object type you can write an S3 method for it.

Value

object with selected values NULLified

Note

attributes are copied from original object and re-applied to final object before return, which may not make sense in some circumstances.

Examples

```
nullify(list(1, 2, 3), 2)
nullify(call("fun", 1, 2, 3), 2)
```

`type_alike` *Compare Types of Objects*

Description

By default, checks `type_of()` objects and two objects are considered `type_alike` if they have the same type. There is special handling for integers, numerics, and functions.

Usage

```
type_alike(target, current, settings = NULL)
```

Arguments

`target`	the object to test type alikeness against
`current`	the object to test the type alikeness of
`settings`	NULL, or a list as produced by `vetr_settings()`

Details

For integers and numerics, if `current` is integer or integer-like (e.g. 1.0) it will match real or integer `target` values. Closures, built-ins, and specials are all treated as type function.

Specific behavior can be tuned with the `type.mode` parameter to the `vetr_settings()` object passed as the `settings` parameter to this function.

See Also

type_of, alike, `vetr_settings()`, in particular the section about the `type.mode` parameter which affects how this function behaves.

`type_of` *A Fuzzier Version of* `typeof()`

Description

Numerics that are equivalent to integers (e.g `x == floor(x)`) are classified as integers, and builtin and special functions are reported as closures.

Usage

```
type_of(object)
```

Arguments

`object`	the object to check the type of

Value

character(1L) the type of the object

Examples

```
type_of(1.0001)         # numeric
type_of(1.0)            # integer (`typeof` returns numeric)
type_of(1)              # integer (`typeof` returns numeric)
type_of(sum)            # closure (`typeof` returns builtin)
type_of(`$`)            # closure (`typeof` returns special)
```

---

vet — *Verify Objects Meet Structural Requirements*

Description

Use vetting expressions to enforce structural requirements for objects. `tev` is a version of `vet` compatible with `magrittr` pipes.

Usage

```
vet(
  target,
  current,
  env = parent.frame(),
  format = "text",
  stop = FALSE,
  settings = NULL
)

tev(
  current,
  target,
  env = parent.frame(),
  format = "text",
  stop = FALSE,
  settings = NULL
)
```

Arguments

`target`	a template, a vetting expression, or a compound expression
`current`	an object to vet
`env`	the environment to match calls and evaluate vetting expressions in; will be ignored if an environment is also specified via `vetr_settings()`. Defaults to calling frame.

format
: character(1L), controls the format of the return value for vet, in case of failure. One of:
  - "text": (default) character(1L) message for use elsewhere in code
  - "full": character(1L) the full error message used in "stop" mode, but actually returned instead of thrown as an error
  - "raw": character(N) least processed version of the error message with none of the formatting or surrounding verbiage

stop
: TRUE or FALSE whether to call `stop()` on failure or not (default)

settings
: a settings list as produced by `vetr_settings()`, or NULL to use the default settings

Details

`tev` just reverses the `target` and `current` arguments for better integration with `magrittr`. There are two major caveats:

- error messages will be less useful since you will get `.` instead of the deparsed call
- `x \\%>\\% tev(y)` is much slower than `vet(y, x)` (or even `tev(x, y)`)

Value

TRUE if validation succeeds, otherwise varies according to value chosen with parameter `stop`

Vetting Expressions

Vetting expressions can be template tokens, standard tokens, or any combination of template and standard tokens combined with `&&` and/or `||`. Template tokens are R objects that define the required structure, much like the `FUN.VALUE` argument to `vapply()`. Standard tokens are tokens that contain the `.` symbol and are used to vet values.

If you do use the `.` symbol in your vetting expressions in your packages, you will need to include `utils::globalVariables(".")` as a top-level call to avoid the "no visible binding for global variable '.'" R CMD check NOTE.

See `vignette('vetr', package='vetr')` and examples for details on how to craft vetting expressions.

See Also

`vetr()` for a version optimized to vet function arguments, `alike()` for how templates are used, `vet_token()` for how to specify custom error messages and also for predefined validation tokens for common use cases, `all_bw()` for fast bounds checks.

Examples

```
## template vetting
vet(numeric(2L), runif(2))
vet(numeric(2L), runif(3))
vet(numeric(2L), letters)
try(vet(numeric(2L), letters, stop=TRUE))
```

```
## `tev` just reverses target and current for use with maggrittr
## Not run:
if(require(magrittr)) {
  runif(2) %>% tev(numeric(2L))
  runif(3) %>% tev(numeric(2L))
}

## End(Not run)
## Zero length templates are wild cards
vet(numeric(), runif(2))
vet(numeric(), runif(100))
vet(numeric(), letters)

## This extends to data.frames
iris.tpl <- iris[0,]   # zero row matches any # of rows
iris.1 <- iris[1:10,]
iris.2 <- iris[1:10, c(1,2,3,5,4)]  # change col order
vet(iris.tpl, iris.1)
vet(iris.tpl, iris.2)

## Short (<100 length) integer-like numerics will
## pass for integer
vet(integer(), c(1, 2, 3))
vet(integer(), c(1, 2, 3) + 0.1)

## Nested templates; note, in packages you should consider
## defining templates outside of `vet` or `vetr` so that
## they are computed on load rather that at runtime
tpl <- list(numeric(1L), matrix(integer(), 3))
val.1 <- list(runif(1), rbind(1:10, 1:10, 1:10))
val.2 <- list(runif(1), cbind(1:10, 1:10, 1:10))
vet(tpl, val.1)
vet(tpl, val.2)

## See `example(alike)` for more template examples

## Standard tokens allow you to check values
vet(. > 0, runif(10))
vet(. > 0, -runif(10))

## Zero length token results are considered TRUE,
## as is the case with `all(logical(0))`
vet(. > 0, numeric())

## `all_bw` is like `isTRUE(all(. >= x & . <= y))`, but
## ~10x faster for long vectors:
vet(all_bw(., 0, 1), runif(1e6) + .1)

## You can combine templates and standard tokens with
## `&&` and/or `||`
vet(numeric(2L) && . > 0, runif(2))
vet(numeric(2L) && . > 0, runif(10))
```

```
vet(numeric(2L) && . > 0, -runif(2))

## Using pre-defined tokens (see `?vet_token`)
vet(INT.1, 1)
vet(INT.1, 1:2)
vet(INT.1 && . %in% 0:1 || LGL.1, TRUE)
vet(INT.1 && . %in% 0:1 || LGL.1, 1)
vet(INT.1 && . %in% 0:1 || LGL.1, NA)

## Vetting expressions can be assembled from previously
## defined tokens
scalar.num.pos <- quote(numeric(1L) && . > 0)
foo.or.bar <- quote(character(1L) && . %in% c('foo', 'bar'))
vet.exp <- quote(scalar.num.pos || foo.or.bar)

vet(vet.exp, 42)
vet(scalar.num.pos || foo.or.bar, 42)  # equivalently
vet(vet.exp, "foo")
vet(vet.exp, "baz")
```

`vetr` *Verify Function Arguments Meet Structural Requirements*

Description

Use vetting expressions to enforce structural requirements for function arguments. Works just like `vet()`, except that the formals of the enclosing function automatically matched to the vetting expressions provided in `...`.

Usage

```
vetr(..., .VETR_SETTINGS = NULL)
```

Arguments

`...`	vetting expressions, each will be matched to the enclosing function formals as with `match.call()` and will be used to validate the value of the matching formal.
`.VETR_SETTINGS`	a settings list as produced by `vetr_settings()`, or NULL to use the default settings. Note that this means you cannot use `vetr` with a function that takes a `.VETR_SETTINGS` argument

Details

Only named arguments may be vetted; in other words it is not possible to vet arguments passed via `...`.

Value

TRUE if validation succeeds, otherwise `stop` with error message detailing nature of failure.

Vetting Expressions

Vetting expressions can be template tokens, standard tokens, or any combination of template and standard tokens combined with `&&` and/or `||`. Template tokens are R objects that define the required structure, much like the `FUN.VALUE` argument to `vapply()`. Standard tokens are tokens that contain the `.` symbol and are used to vet values.

If you do use the `.` symbol in your vetting expressions in your packages, you will need to include `utils::globalVariables(".")` as a top-level call to avoid the "no visible binding for global variable '.'" R CMD check NOTE.

See `vignette('vetr', package='vetr')` and examples for details on how to craft vetting expressions.

Note

`vetr` will force evaluation of any arguments that are being checked (you may omit arguments that should not be evaluate from `vetr`)

See Also

`vet()`, in particular `example(vet)`.

Examples

```
fun1 <- function(x, y) {
  vetr(integer(), LGL.1)
  TRUE   # do some work
}
fun1(1:10, TRUE)
try(fun1(1:10, 1:10))

## only vet the second argument
fun2 <- function(x, y) {
  vetr(y=LGL.1)
  TRUE   # do some work
}
try(fun2(letters, 1:10))

## Nested templates; note, in packages you should consider
## defining templates outside of `vet` or `vetr` so that
## they are computed on load rather that at runtime
tpl <- list(numeric(1L), matrix(integer(), 3))
val.1 <- list(runif(1), rbind(1:10, 1:10, 1:10))
val.2 <- list(runif(1), cbind(1:10, 1:10, 1:10))
fun3 <- function(x, y) {
  vetr(x=tpl, y=tpl && ncol(.[[2]]) == ncol(x[[2]]))
  TRUE   # do some work
}
fun3(val.1, val.1)
try(fun3(val.1, val.2))
val.1.a <- val.1
val.1.a[[2]] <- val.1.a[[2]][, 1:8]
try(fun3(val.1, val.1.a))
```

`vetr_settings` *Generate Control Settings For vetr and alike*

Description

Utility function to generate setting values. We strongly recommend that you generate the settings outside of function calls so that setting generation does not become part of the `vet/vetr/alike` evaluation as that could add noticeable overhead to the function evaluation.

Usage

```
vetr_settings(
  type.mode = 0L,
  attr.mode = 0L,
  lang.mode = 0L,
  fun.mode = 0L,
  rec.mode = 0L,
  suppress.warnings = FALSE,
  fuzzy.int.max.len = 100L,
  width = -1L,
  env.depth.max = 65535L,
  symb.sub.depth.max = 65535L,
  symb.size.max = 15000L,
  nchar.max = 65535L,
  track.hash.content.size = 63L,
  env = NULL,
  result.list.size.init = 64L,
  result.list.size.max = 1024L
)
```

Arguments

`type.mode` integer(1L) in 0:2, defaults to 0, determines how object types (as in `typeof`) are compared:

- 0: integer like numerics (e.g. `1.0`) can match against integer templates, and integers always match real templates; all function types are considered of the same type
- 1: integers always match against numeric templates, but not vice versa, and integer-like numerics are treated only as numerics; functions only match same function type (i.e. closures only match closures, builtins builtins, and specials specials)
- 2: types must be equal for all objects types (for functions, this is unchanged from 1)

`attr.mode` integer(1L) in 0:2, defaults to 0, determines strictness of attribute comparison:

- `0` only checks attributes that are present in target, and uses special comparisons for the special attributes (`class`, `dim`, `dimnames`, `names`, `row.names`, `levels`, `srcref`, and `tsp`) while requiring other attributes to be `alike`
- `1` is like `0`, except all atributes must be `alike`
- `2` requires all attributes to be present in `target` and `current` and to be alike

`lang.mode` integer(1L) in 0:1, defaults to 0, controls language matching, set to `1` to turn off use of `match.call()`

`fun.mode` NOT IMPLEMENTED, controls how functions are compared

`rec.mode` integer(1L) `0` currently unused, intended to control how recursive structures (other than language objects) are compared

`suppress.warnings`
logical(1L) suppress warnings if TRUE

`fuzzy.int.max.len`
max length of numeric vectors to consider for integer likeness (e.g. `c(1, 2)` can be considered "integer", even though it is numeric); currently we limit this check to vectors shorter than 100 to avoid a potentially expensive computation on large vectors, set to -1 to apply to all vectors irrespective of length

`width` to use when deparsing expressions; default `-1` equivalent to `getOption("width")`

`env.depth.max` integer(1L) maximum number of nested environments to recurse through, defaults to 65535L; these are tracked to make sure we do not get into an infinite recursion loop, but because they are tracked we keep a limit on how many we will go through, set to -1 to allow unlimited recursion depth. You should not need to change this unless you are running into the recursion limit.

`symb.sub.depth.max`
integer(1L) maximum recursion depth when recursively substituting symbols in vetting expression, defaults to 65535L

`symb.size.max` integer(1L) maximum number of characters that a symbol is allowed to have in vetting expressions, defaults to 15000L.

`nchar.max` integer(1L) defaults to 65535L, threshold after which strings encountered in C code are truncated. This is the read limit. In theory `vetr` can produce strings longer than that by combining multiple shorter pieces.

`track.hash.content.size`
integer(1L) (advanced) used to set the initial size of the symbol tracking vector used with the hash table that detects recursive symbol substitution. If the tracking vector fills up it will be grown by 2x. This parameter is exposed mostly for developer use.

`env` what environment to use to match calls and evaluate vetting expressions, although typically you would specify this with the `env` argument to `vet`; if NULL will use the calling frame to `vet/vetr/alike`.

`result.list.size.init`
initial value for token tracking. This will be grown by a factor of two each time it fills up until we reach `result.list.size.max`.

`result.list.size.max`
maximum number of tokens we keep track of, intended mostly as a safeguard in case a logic error causes us to keep allocating memory. Set to 1024 as a default

value since it should be exceedingly rare to have vetting expressions with such a large number of tokens, enough so that if we reach that number it is more likely something went wrong.

Details

Settings after `fuzzy.int.max.len` are fairly low level and exposed mostly for testing purposes. You should generally not need to use them.

Note that a successful evaluation of this function does not guarantee a correct settings list. Those checks are carried out internally by `vet/vetr/alike`.

Value

list with all the setting values

See Also

`type_alike`, `alike`, `vetr`

Examples

```
type_alike(1L, 1.0, settings=vetr_settings(type.mode=2))
## better if you are going to re-use settings to reduce overhead
set <- vetr_settings(type.mode=2)
type_alike(1L, 1.0, settings=set)
```

`vet_token` *Vetting Tokens With Custom Error Messages*

Description

Utility function to generate vetting tokens with attached error messages. You should only need to use this if the error message produced naturally by `vetr` is unclear. Several predefined tokens created by this function are also documented here.

Usage

```
vet_token(exp, err.msg = "%s")

NO.NA

NO.INF

GTE.0

LTE.0

GT.0
```

```
LT.0

INT.1

INT.1.POS

INT.1.NEG

INT.1.POS.STR

INT.1.NEG.STR

INT

INT.POS

INT.NEG

INT.POS.STR

INT.NEG.STR

NUM.1

NUM.1.POS

NUM.1.NEG

NUM

NUM.POS

NUM.NEG

CHR.1

CHR

CPX

CPX.1

LGL

LGL.1
```

Arguments

`exp`	an expression which will be captured but not evaluated.
`err.msg`	character(1L) a message that tells the user what the expected value should be, should contain a "%s" for `sprintf` to use (e.g. "%s should be greater than 2").

Format

An object of class `call` of length 2.

An object of class `call` of length 2.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

An object of class `call` of length 3.

Details

Allows you to supply error messages for vetting to use for each error token. Your token should not contain top level `&&` or `||`. If it does your error message will not be reported because `vetr` looks for error messages attached to atomic tokens. If your token must involve top level `&&` or `||`, use `I(x && y)` to ensure that your error message is used by `vet`, but beware than in doing so you do not use templates within the `I` call as everything therein will be interpreted as a vetting expression rather than a template.

Error messages are typically of the form "%s should be XXX".

This package ships with many predefined tokens for common use cases. They are listed in the `Usage` section of this documentation. The tokens are named in format `TYPE[.LENGTH][.OTHER]`. For example `INT` will vet an integer vector, `INT.1` will vet a scalar integer vector, and `INT.1.POS.STR` will vet a strictly positive integer vector. At this time tokens are predefined for the basic types as scalars or any-length vectors. Some additional checks are available (e.g. positive only values).

Every one of the predefined vetting tokens documented here implicitly disallows NAs. Numeric tokens also disallow infinite values. If you wish to allow NAs or infinite values just use a template object (e.g. `integer(1L)`).

Value

a quoted expressions with `err.msg` attribute set

Note

This will only work with standard tokens containing `.`. Anything else will be interpreted as a template token.

See Also

`vet()`

Examples

```
## Predefined tokens:
vet(INT.1, 1:2)
vet(INT.1 || LGL, 1:2)
vet(INT.1 || LGL, c(TRUE, FALSE))

## Check squareness
mx <- matrix(1:3)
SQR <- vet_token(nrow(.) == ncol(.), "%s should be square")
vet(SQR, mx)

## Let `vetr` make up error message; note `quote` vs `vet_token`
## Often, `vetr` does fine without explictly specified err msg:
SQR.V2 <- quote(nrow(.) == ncol(.))
vet(SQR.V2, mx)

## Combine some tokens, notice how we use `quote` at the combining
## step:
```

```
NUM.MX <- vet_token(matrix(numeric(), 0, 0), "%s should be numeric matrix")
SQR.NUM.MX <- quote(NUM.MX && SQR)
vet(SQR.NUM.MX, mx)

## If instead we used `vet_token` the overall error message
## is not used; instead it falls back to the error message of
## the specific sub-token that fails:
NUM.MX <- vet_token(matrix(numeric(), 0, 0), "%s should be numeric matrix")
SQR.NUM.MX.V2 <-
  vet_token(NUM.MX && SQR, "%s should be a square numeric matrix")
vet(SQR.NUM.MX.V2, mx)
```

Index