Package 'typed'

October 14, 2022

Title Support Types for Variables, Arguments, and Return Values

Version 0.0.1

Description A type system for R. It supports setting variable types in a script or the body of a function, so variables can't be assigned illegal values. Moreover it supports setting argument and return types for functions.

License GPL-3

Encoding UTF-8

Language en

LazyData true

RoxygenNote 7.1.1

URL https://github.com/moodymudskipper/typed

BugReports https://github.com/moodymudskipper/typed/issues

Imports waldo

Suggests testthat, covr

NeedsCompilation no

Author Antoine Fabri [aut, cre]

Maintainer Antoine Fabri <antoine.fabri@gmail.com>

Repository CRAN

Date/Publication 2021-03-19 10:50:02 UTC

R topics documented:

? *Set Variable Types, Argument Types and Return Types.*

Description

Use ? to set a function's return type, argument types, or variable types in the body of the function. `declare` is an alternative to set a variable's type.

Usage

```
`?`(lhs, rhs)

declare(x, assertion, value, const = FALSE)
```

Arguments

`lhs`	lhs
`rhs`	rhs
`x`	variable name as a string
`assertion`	a function
`value`	an optional value
`const`	whether to declare `x` as a constant

Value

`declare` (and ? when it maps to `declare`) returns `value` invisibly, it is called for side effects. `assertion ? function(<args>) {<body>}` returns a typed function, of class `c("typed", "function")`. `fun <- assertion ? function(<args>) {<body>}` returns a typed function and binds it to `fun` in the local environment.

Set A Variable's Type

When used to set a variable's type, ? maps to `declare` so that `assertion ? var` calls `declare("var", assertion)`, `assertion ? var <- value` calls `declare("var", assertion, value)`, and `assertion ? (var) <- value` calls `declare("var", assertion, value, const = TRUE)`

In those cases an active binding is defined so `var` returns `value` (or `NULL` if none was provided). If `const` is `FALSE` (the default), the returned value can then be altered if by assigning to `var`, but a value which doesn't satisfy the assertion will trigger an error.

Set A Function's Return Type

The syntaxes `assertion ? function(<args>) {<body>}` and `fun <- assertion ? function(<args>) {<body>}` can be used to create a function of class `c("typed", "function")`. The returned function will have its body modified so that return values are wrapped inside a `check_output()` call. Printing the function will display the return type.

Set A Function Argument's Type

When using the above syntax, or if we don't want to force a return type, the simpler `? function(<args>) {<body>}` or `fun <- ? function(<args>) {<body>}` syntax, we can set argument types by providing arguments as `arg = default_value ? assertion` or `arg = ? assertion`. When entering the function, argument types will be checked.

By default the arguments are only checked at the top, and might be assigned later in the function's body values that don't satisfy the assertion, to avoid this we can type `arg = default_value ? +assertion` or `arg = ? +assertion`.

Note that forgetting the `?` before `function` is an easy mistake to do!

If we'd rather check the quoted argument rather than the argument's value, we can type `arg = default_value ? ~assertion` or `arg = ? ~assertion`. A possible use case might be `arg = ? ~Symbol()`.

Dots can be checked too, `... = ? assertion` will make sure that every argument passed to dots satisfies the assertion.

The special assertion factory `Dots` can also be used, in that case the checks will apply to `list(...)` rather than to each element individually, for instance `function(... = ? Dots(2))` makes sure the dots were fed 2 values.

The returned function will have its body modified so the arguments are checked by `check_arg()` calls at the top. Printing the function will display the argument types.

Examples

```
Integer() ? function (x= ? Integer()) {
  Integer() ? y <- 2L
  res <- x + y
  res
}
```

Any *Assertion factories of package 'typed'*

Description

These functions are assertion factories, they produce assertions, which take an object, check conditions, and returns the input, usually unmodified (never modified with the functions documented on this page).

Usage

```
Any(length, ...)

Logical(length, null_ok = FALSE, ...)

Integer(length, null_ok = FALSE, ...)
```

```
Double(length, null_ok = FALSE, ...)

Character(length, null_ok = FALSE, ...)

Raw(length, null_ok = FALSE, ...)

List(length, each, data_frame_ok, null_ok = FALSE, ...)

Null(...)

Closure(null_ok = FALSE, ...)

Special(null_ok = FALSE, ...)

Builtin(null_ok = FALSE, ...)

Environment(null_ok = FALSE, ...)

Symbol(null_ok = FALSE, ...)

Pairlist(length, each, null_ok = TRUE, ...)

Language(null_ok = FALSE, ...)

Expression(length, null_ok = FALSE, ...)

Function(null_ok = FALSE, ...)

Factor(length, levels, null_ok = FALSE, ...)

Matrix(nrow, ncol, null_ok = FALSE, ...)

Array(dim, null_ok = FALSE, ...)

Data.frame(nrow, ncol, each, null_ok = FALSE, ...)

Date(length, null_ok = FALSE, ...)

Time(length, null_ok = FALSE, ...)

Dots(length, each, ...)

Logical(length, null_ok = FALSE, ...)

Integer(length, null_ok = FALSE, ...)

Double(length, null_ok = FALSE, ...)
```

```
Character(length, null_ok = FALSE, ...)

Raw(length, null_ok = FALSE, ...)

List(length, each, data_frame_ok = TRUE, null_ok = FALSE, ...)

Null(...)

Closure(null_ok = FALSE, ...)

Special(null_ok = FALSE, ...)

Builtin(null_ok = FALSE, ...)

Environment(null_ok = FALSE, ...)

Symbol(null_ok = FALSE, ...)

Pairlist(length, each, null_ok = TRUE, ...)

Language(null_ok = FALSE, ...)

Expression(length, null_ok = FALSE, ...)

Function(null_ok = FALSE, ...)

Factor(length, levels, null_ok = FALSE, ...)

Data.frame(nrow, ncol, each, null_ok = FALSE, ...)

Matrix(nrow, ncol, null_ok = FALSE, ...)

Array(dim, null_ok = FALSE, ...)

Date(length, null_ok = FALSE, ...)

Time(length, null_ok = FALSE, ...)

Dots(length, each, ...)
```

Arguments

length	length of the object
...	additional conditions, see details.
null_ok	whether NULL values should be accepted, and not subjected to any further check.
each	assertion that every item must satisfy
data_frame_ok	whether data frames are to be considered as lists

levels	factor levels
nrow	number of rows
ncol	number of columns
dim	dimensions

Details

Additional conditions can be provided :

- If they are named, the name should be the name of a function to use on our object, and the value should be the expected value.
- If they are unnamed, they should be formulas, the right hand side should be a condition, using `value` or `.` as a placeholder for the latter, and the optional `lhs` an error message.

`Any` is the most general assertion factory, it doesn't check anything unless provided additional conditions through `...`. Others use the base `is.<type>` function if available, or check that the object is of the relevant type with `typeof` for atomic types, or check that the class of the checked value contains the relevant class.

`Dots` should only be used to check the dots using `check_arg` on `list(...)` or `substitute(...())`, which will be the case when it's called respectively with `function(... = ? Dots())` and `function(... = ?~ Dots())`

Value

A function, and more specifically, an assertion as defined above.

Examples

```
## Not run:
# fails
Integer() ? x <- 1
# equivalent to
declare("x", Integer(), value = 1)

Integer(2) ? x <- 1L

# we can use additional conditions in `...`
Integer(anyNA = FALSE) ? x <- c(1L, NA, 1L)
Integer(anyDuplicated = FALSE) ? x <- c(1L, NA, 1L)

## End(Not run)

Integer(2) ? x <- 11:12

## Not run:
# We can also use it directly to test assertions
Integer() ? x <- 1
# equivalent to
declare("x", Integer(), value = 1)
```

```
Integer(2) ? x <- 1L

## End(Not run)
```

`as_assertion_factory` *Build a new type*

Description

Build a new type

Usage

```
as_assertion_factory(f)
```

Arguments

`f`	a function

Value

a function with class `assertion_factory`

`check_output` *Check Argument Types and Return Type*

Description

These functions are not designed to be used directly, we advise to use the syntaxes described in `?declare` instead. `check_arg` checks that arguments satisfy an assertion, and if relevant make them into active bindings to make sure they always satisy it. `check_output` checks that the value, presumably a return value, satisfies an assertion,

Usage

```
check_output(.output, .assertion, ...)

check_arg(.arg, .assertion, ..., .bind = FALSE)
```

Arguments

`.output`	function output
`.assertion`	an assertion
`...`	additional arguments passed to assertion
`.arg`	function argument
`.bind`	whether to actively bind the argument so it cannot be modified unless it satisfies the assertion

Value

`.output`if it satisfies the assertion, fails otherwise.

returns `NULL` invisibly, called for side effects.

`process_assertion_factory_dots`
Process assertion factory dots

Description

This needs to be exported, but shouldn't be called by the user

Usage

```
process_assertion_factory_dots(...)
```

Arguments

`...`	dots

Value

a { expression

Index