Package 'mutatomic'

April 20, 2025

Type Package

Title Tools for Safe Pass-by-Reference Modification Semantics on Atomic Objects

Version 0.0.0.9

Description Provides tools, including a new class ('mutatomic'),
for safe pass-by-reference modification semantics on atomic objects.
Primary purpose for this package is so the 'mutatomic' class and its related tools
are accessible for other packages that wish to implement safe pass-by-
reference semantics for atomic objects.

License MPL-2.0 | file LICENSE

Encoding UTF-8

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Suggests tinytest,
roxygen2

Depends R (>= 4.2.0)

Imports Rcpp (>= 1.0.11),
data.table (>= 1.14.8)

Language en-gb

Contents

`.internal_set_ma` *Exposed functions for Package developers*

Description

Functions for developers.
The functions in this list should NOT be called by regular users.
They are only to be used inside packages.

Usage

```
.internal_set_ma(x)

.internal_ma_set_DimsAndNames(x, names = NULL, dim = NULL, dimnames = NULL)
```

Arguments

`x`	atomic object
`names`	`NULL` or a character vector of the same length as `x`, giving the flat names.
`dim`	`NULL` or an integer vector giving the dimensions of `x`. Note that it must hold that `prod(dim(x)) == length(x)`.
`dimnames`	`NULL` or a list of `dimnames``.`

Details

`.internal_set_ma()` sets an object to class 'mutatomic' by reference.
`.internal_ma_set_DimsAndNames()` sets the dimensions and (dim)names of a 'mutatomic' object by reference.

Value

Returns: VOID. This function modifies the object by reference.
Do NOT use assignment like `x <- .internal_set_ma(x, ...)`.
Since this function returns void, you'll just get `NULL`.

Examples

```
testfun1 <- function(x) {
  .internal_set_ma(x)
}
testfun2 <- function(x, names = NULL, dim = NULL, dimnames = NULL) {
  .internal_ma_set_DimsAndNames(x, names = names, dim = dim, dimnames = dimnames)
}

x <- 1:10
```

```
is.mutatomic(x)

testfun1(x)
is.mutatomic(x)
print(x)

testfun2(
  x,
  letters[1:10],
  c(2, 5),
  list(month.abb[1:2], month.name[1:5])
)

print(x)
```

`aaa00_mutatomic_help` *mutatomic*

Description

Tools for Safe Pass-By-Reference Modification Semantics on Atomic Objects

What is 'mutatomic'?

'mutatomic' is an 'R' package that provides 2 things:

- For regular 'R' users:
  A new class of atomic vectors, matrices, and arrays, called mutatomic, that provides (safer) support for pass-by-reference semantics.
  It also provides an optional helper function, called currentBindings.
- For package back-end:
  A set of tools for safer pass-by-reference modification semantics for atomic objects, as 'R' does not natively provide pass-by-reference mechanics (except via it's internal C API or through a package like 'Rcpp').
  The developer tools provided by 'mutatomic' were primarily designed for the 'squarebrackets' and 'broadcast' packages, but authors of other packages are welcome to use 'mutatomic' also.

'mutatomic' does not come with pass-by-reference functions itself, except for 2 simple example functions (ma_set and setapply).

For Regular 'R' Users

Regular 'R' users can construct objects of class 'mutatomic', or convert objects to class 'mutatomic';
see mutatomic_class.

Depending on the situation end users may also use the currentBindings function.

The 'mutatomic' package also comes with a few help pages, that end users can refer to, to gain a better understanding of the pass-by-reference semantics supported by 'mutatomic'.
These help pages are the following:

- mutatomic_class:
  Explains the 'mutatomic' class.
- mutatomic_PassByReference:
  Explains Pass-by-Reference semantics, and its important consequences.
- mutatomic_coercion:
  Explains the difference in coercion rules between modification through Pass-by-Reference semantics and modification through copy (i.e. pass-by-value).

For Package Back-End

'mutatomic' provides tools for package authors to program with mutable atomic objects.
Arguably the most important function in 'mutatomic' for developers is the stopifnot_ma_safe2mutate function.
This function checks if an atomic object is safe to mutate, and gives an error otherwise.

What follows is technical information on 'mutatomic', and why a 'mutatomic' class is needed; this is of no interest for regular 'R' users.

Technical - Why is 'mutatomic' needed?

Consider the following code:

```
x <- base::letters
collapse::setv(x, "a", "xxx")
```

the above code modifies `base::letters` by reference, and nothing is stopping the user from changing base 'R' while 'R' is still running!
Now, obviously `collapse::setv()` was meant for internal programming purposes, and not to be called by amateurs.
But what if one wishes to design an 'R' package that provides pass-by-reference mechanics for atomic objects in a somewhat safe way?

This is where the 'mutatomic' package comes in.
'mutatomic' provides a new class which can be considered the atomic version of the 'data.table' class, and through this class package authors can protect the user from changing things like `base:letters`.

Technical - How does 'mutatomic' solve the issue?

'mutatomic' first and fore-most provides a new class, called 'mutatomic'.
If a function demands an atomic object is of class 'mutatomic' to allow pass-by-reference semantics, the above issue is (mostly) prevented.
The issue is prevented through the following means:

- Creating an object of class 'mutatomic', at least when using the functions provided by this package, will copy the original object.
  So when calling `x <- mutatomic(base::letters)`, `x` no longer refers to `base::letters`.
- When a function demands an object is of class 'mutatomic', it is (almost) guaranteed it does not refer to an base 'R' object, so something like `base::letters` will never be modified by reference.
- 'mutatomic' stores a list of most base 'R' (atomic) object addresses when it is loaded.
  The is.mutatomic and stopifnot_ma_safe2mutate functions check this list, creating extra certainty that base 'R' is never modified by reference.
- The 'mutatomic' class is not just defined by a class name attribute.
  The class comes with additional attributes to ensure it truly has been created by the functions of this package.
  Although these attributes can be mimicked with enough effort, it is very unlikely for an object to **accidentally** have these attributes.
  The is.mutatomic and stopifnot_ma_safe2mutate functions check for these attributes, creating extra security.

- 'mutatomic' respects the lock of a binding, if there is any.

Author(s)

Author, Maintainer: Tony Wilkes <tony_a_wilkes@outlook.com> (ORCID)

References

The badges shown in the documentation of this R-package were made using the services of: https://shields.io/

`aaa01_mutatomic_PassByReference`

Regarding Modification By Reference

Description

This help page describes how modification using "pass-by-reference" semantics is handled by the 'mutatomic' package.

"Pass-by-reference" refers to modifying a mutable object, or a subset of a mutable object, without making any copies at all.

This help page does not explain all the basics of pass-by-reference semantics, as this is treated as prior knowledge.
All functions/methods in the 'mutatomic' package (and 'data.table', 'squarebrackets', 'broadcast', and more) with the word "set" in the name use pass-by-reference semantics.

Advantages and Disadvantages

The main advantage of pass-by-reference is that much less memory is required to modify objects, and modification is also generally faster.
But it does have several disadvantages.

First, the coercion rules are slightly different: see mutatomic_coercion.

Second, if 2 or more variables refer to exactly the same object (i.e. have the same address), changing one variable also changes the other ones.
I.e. the following code,

```
x <- y <- mutatomic(1:16)
ma_set(x, 1:6, 8)
```

modifies not just `x`, but also `y`.
This is true even if one of the variables is locked (see bindingIsLocked).
I.e. the following code,

```
x <- mutatomic(1:16)
y <- x
lockBinding("y", environment())
ma_set(x, i = 1:6, rp = 8)
```

modifies both `x` and `y` without error, even though `y` is a locked constant.

Mutable vs Immutable Classes

With the exception of environments, most of base R's S3 classes are treated as immutable:
Modifying an object in 'R' will make a copy of the object, something called 'copy-on-modify' semantics.

A prominent mutable S3 class is the `data.table` class, which is a mutable data.frame class, and supported by 'mutatomic'.
Similarly, 'mutatomic' adds a class for mutable atomic objects:
mutatomic.

Material vs Immaterial objects

Most objects in 'R' are material objects:
the values an object contains are actually stored in memory.
For example, given `x <- rnorm(1e6)`, `x` is a material object:
1 million values (decimal numbers, in this case) are actually stored in memory.

In contrast, ActiveBindings are immaterial:
They are objects that, when accessed, call a function to generate values on the fly, rather than actually storing values.

Since immaterial objects do not actually store the values in memory, the values obviously also

cannot be changed in memory.
Therefore, Pass-by-Reference semantics don't work on immaterial objects.

ALTREP

The mutatomic constructors (i.e. mutatomic, as.mutatomic, etc.) will automatically materialize ALTREP objects, to ensure consistent behaviour for 'pass-by-reference' semantics.

A `data.table` can have ALTREP columns.
A `data.tables` will coerce the column to a materialized column when it is modified, even by reference.

Mutability Rules With Respect To Recursive Objects

Lists are difficult objects in that they do not contain elements, they simply point to other objects, that one can access via a list.
When a recursive object is of a mutable class, all its subsets are treated as mutable, as long as they are part of the object.
On the other hand, When a recursive object is of an immutable class, its recursive subsets retain their original mutability.

Example 1: Mutable data.tables
A `data.table` is a mutable class.
So all columns of the `data.table` are treated as mutable;
There is no requirement to, for instance, first change all columns into the class of mutatomic to modify these columns by reference.

Example 2: Immutable lists
A regular `list` is an immutable class.
So the list itself is immutable, but the recursive subsets of the list retain their mutability.
If you have a list of `mutatomic` objects, for example, the mutatomic objects themselves remain mutable.
Therefore, the following pass-by-reference modification will work without issue:

```
x <- list(
 a = mutatomic(letters[1:10]),
 b = mutatomic(letters[11:20])
)
myref <- x$a
ma_set(myref, 1, "xxx")
```

Notice in the above code that `myref` has the same address as `x$a`, and is therefore not a copy of `x$a`.
Thus changing `myref` also changes `x$a`.
In other words: `myref` is what could be called a "**View**" of `x$a`.

Input Variable

Methods/functions that perform in-place modification by reference only works on objects that actually exist as an actual variable, similar to functions in the style of `some_function(x, ...) <- value`.
Thus things like any of the following,
`ma_set(1:10, ...)`, `ma_set(x$a, ...)`, or `ma_set(base::letters)`,
will not and should not work.

Lock Binding

Mutable classes are, as the name suggests, meant to be mutable.
Locking the binding of a mutable object is **mostly** fruitless (but not completely; see the currentBindings function).
To ensure an object cannot be modified by any of the methods/functions from 'mutatomic', 2 things must be true:

- the object must be an immutable class.
- the binding must be **locked** (see lockBinding).

Protection

Due to the properties described above in this help page, 'mutatomic' protects the user from do something like the following:

```
# letters = base::letters
ma_set(letters, i = 1, rp = "XXX")
```

'mutatomic' will give an error when running the code above, because:

1. most addresses in `baseenv()` are protected;
2. immutable objects are disallowed (you'll have to create a mutable object, which will create a copy of the original, thus keeping the original object safe from modification by reference);
3. locked bindings are disallowed.

Examples

```
# the following code demonstrates how locked bindings,
# such as `base::letters`,
# are being safe-guarded

x <- list(a = base::letters)
myref <- x$a # view of a list
address(myref) == address(base::letters) # TRUE: point to the same memory
bindingIsLocked("letters", baseenv()) # base::letters is locked ...
bindingIsLocked("myref", environment()) # ... but this pointer is not!
```

```
if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    ma_set(myref, 1, "XXX") # this still gives an error though ...
  )
}

is.mutatomic(myref) # ... because it's not of class `mutatomic`


x <- list(
  a = as.mutatomic(base::letters) # `as.mutatomic()` makes a copy
)
myref <- x$a # view of a list
address(myref) == address(base::letters) # FALSE: it's a copy
ma_set(
  myref, i = 26, rp = "XXX"  # modifies x, does NOT modify `base::letters`
)
print(x) # x is modified
base::letters # but this still the same
```

`aaa02_mutatomic_coercion`

Auto-Coercion Rules for Mutable Objects

Description

This help page describes the auto-coercion rules of the mutable classes, as they are handled by the 'mutatomic' package.
This useful information for users who wish to intend to employ Pass-by-Reference semantics as provided by 'mutatomic'.

mutatomic

coercion_through_copy: YES
coercion_by_reference: NO
Mutable atomic objects are automatically coerced to fit the modified subset values, when modifying through copy, just like regular atomic classes.
For example, replacing one or multiple values in an integer vector (type `int`) with a decimal number (type `dbl`) will coerce the entire vector to type `dbl`.

Replacing or transforming subsets of mutable atomic objects **by reference** does not support coercion. Thus, for example, the following code,

```
x <- mutatomic(1:16)
ma_set(x, i = 1:6, rp = 8.5)
#> coercing replacement to integer
print(x)
```

```
#>  [1]  8  8  8  8  8  8  7  8  9 10 11 12 13 14 15 16
#> mutatomic
#> typeof:  integer
```

gives `c(rep(8, 6) 7:16)` instead of `c(rep(8.5, 6), 7:16)`, because `x` is of type `integer`, so `rp` is interpreted as type `integer` also.

data.table, when replacing/transforming whole columns

coercion_through_copy: YES
coercion_by_reference: YES
A data.table is actually a list made mutable, where each column is itself a list. As such, replacing/transforming whole columns using `data.table::set()`, without specifying rows (not even `i = 1:nrow(x)`), allows completely changing the type of the column.

data.table, when partially replacing/transforming columns

coercion_through_copy: YES
coercion_by_reference: NO
If rows are specified in the `data.table::set()` function (and functions that internally use `data.table::set()`), and thus not all values of columns but parts(i.e. rows) of columns are replaced, no auto-coercion takes place.
I.e.: replacing/transforming a value in an integer (`int`) column to become `1.5`, will not coerce the column to the decimal type (`dbl`); instead, the replacement value `1.5` is coerced to integer `1`.

Using R's native copy-on-modify semantics (for example by changing a `data.table` into a `data.frame`) allows for coercion even when partially replacing/transforming columns.

Views of Lists

Regular lists are treated as immutable by 'mutatomic'.
But remember that a list is a (potentially hierarchical) structure of references to other objects.
Thus, even if a list itself is not treated as mutable, subsets of a list which are themselves mutable classes, are mutable.
For example, if you have a list of `data.table` objects, the data.tables themselves are mutable.
Therefore, the following will work:

```
x <- list(
 a = mutatomic(letters[1:10]),
 b = mutatomic(letters[11:20])
)
myref <- x$a
ma_set(myref, 1, "xxx")
```

Notice in the above code that `myref` is not a copy of `x$a`, since they have the same address.
Thus changing `myref` also changes `x$a`.
In other words: `myref` is what could be called a "view" of `x$a`.
Notice also that ma_set`(x$a, ...)` will not work.
This is because stopifnot_ma_safe2mutate will give an error if `x` is not an **actual variable**, similar to in-place functions in the style of `` `myfun()<-` ``.

The auto-coercion rules of Views of Lists, depends entirely on the object itself.
Thus if the View is a data.table, coercion rules of data.tables apply.
And if the View is a mutatomic object, coercion rules of mutatomic objects apply, etc.

Examples

```
# Coercion examples - mutatomic ====

x <- as.mutatomic(1:16)
ma_set(x, i = 1:6, rp = 8.5) # 8.5 coerced to 8, because `x` is of type `integer`
print(x)

################################################################################

# Coercion examples - data.table - whole columns ====

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)

data.table::set(
  obj, j = c("a", "c"),
  value = -1 # SAFE: row=NULL & obs = NULL, so coercion performed
)
str(obj)

################################################################################


# Coercion examples - data.table - partial columns ====

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
data.table::set(
  obj, i = which(with(obj, (a >= 2) & (c <= 17))), j = c("a", "c"),
  value = -1
  # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
print(obj)



################################################################################
```

```
# View of List ====

x <- list(
  a = mutatomic(letters[1:10]),
  b = mutatomic(letters[11:20])
)
myref <- x$a
ma_set(myref, 1L, "xxx")
print(x)
myref <- x$a
address(myref) == address(x$a) # they are the same
print(x) # notice x has been changed
```

`currentBindings` *List or Lock All Currently Existing Bindings Pointing To Same Address*

Description

`currentBindings(x, action = "list")`
lists all **currently existing** objects sharing the same **address** as `x`, in a given environment.

`currentBindings(x, action = "checklock")`
searches all **currently existing** objects sharing the same **address** as `x`, in a given environment, and reports which of these are locked and which are not locked.

`currentBindings(x, action = "lock")`
searches all **currently existing** objects sharing the same **address** as `x`, in a given environment, and locks them using lockBinding.

See also mutatomic_PassByReference for information regarding the relation between locked bindings and pass-by-reference modifications.

Usage

```
currentBindings(x, action = "list", env = NULL)
```

Arguments

`x`	the existing variable whose address to use when searching for bindings.
`action`	a single string, giving the action to perform. Must be one of the following: • `"list"` (default). • `"checklock"`. • `"lock"`.
`env`	the environment where to look for objects. If `NULL` (default), the caller environment is used.

Details

The lockBinding function locks a binding of an object, preventing modification.
'R' also uses locked bindings to prevent modification of objects from package namespaces.
'mutatomic in principle respect this, and disallows modification of objects by reference.

However, lockBinding does not lock the address/pointer of an object, only one particular binding of an object.
This problematic; consider the following example:

```
x <- mutatomic(1:16)
y <- x
lockBinding("y", environment())
ma_set(x, i = 1:6, rp = 8)
```

In the above code, `x` and `y` share the same address, thus pointing to the same memory, yet only `y` is actually locked.
Since `x` is not locked, modifying `x` is allowed.
But since `ma_set()` performs modification by reference, `y` will still be modified, despite being locked.

The `currentBindings()` function allows to user to: find all **currently existing** bindings in the **caller environment** sharing the same address as `x`, and locking all these bindings.

Value

For `currentBindings(x, action = "list")`:
Returns a character vector.

For `currentBindings(x, action = "checklock")`:
Returns a named logical vector.
The names give the names of the bindings,
and each associated value indicates whether the binding is locked (`TRUE`) or not locked (`FALSE`).

For `currentBindings(x, action = "lock")`:
Returns VOID. It just locks the currently existing bindings.
To unlock the bindings, remove the objects (see rm).

Warning

The `currentBindings()` function only locks **currently existing** bindings in the **specified environment**;
bindings that are created **after** calling `currentBindings()` will not automatically be locked.
Thus, every time the user creates a new binding of the same object, and the user wishes it to be locked, `currentBindings()` must be called again.

Examples

```
x <- as.mutatomic(1:10)
y <- x
lockBinding("y", environment())
currentBindings(x)
currentBindings(x, "checklock") # only y is locked


# since only y is locked, we can still modify y through x by reference:
ma_set(x, i = 1, rp = -1)
print(y) # modified!
rm(list= c("y")) # clean up


# one can fix this by locking ALL bindings:
y <- x
currentBindings(x, "lock") # lock all
currentBindings(x, "checklock") # all bindings are locked, including y
# the 'mutatomic' package respects the lock of a binding,
# provided all bindings of an address are locked;
# so this will give an error, as it should:

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    ma_set(x, i = 1, rp = -1),
    pattern = "cannot change value of locked binding for"
  )
}

# creating a new variable will NOT automatically be locked:
z <- y # new variable; will not be locked!
currentBindings(x, "checklock") # z is not locked
currentBindings(x, "lock") # we must re-run this
currentBindings(x, "checklock") # now z is also locked

if(requireNamespace("tinytest")) {
  tinytest::expect_error( # now z is also protected
    ma_set(z, i = 1, rp = -1),
    pattern = "cannot change value of locked binding for"
  )
}


rm(list= c("x", "y", "z")) # clean up
```

`ma_set` *Example Pass-By-Reference Function*

Description

The `ma_set()` function modifies a subset of a mutatomic object by reference.
It is an example function, used to illustrate various examples in the help pages.

For full pass-by-reference functionality, please see - for example - the 'squarebrackets' package.

Usage

```
ma_set(x, i, rp)
```

Arguments

`x`	a **mutatomic** vector (also works on arrays).
`i`	a vector of strictly positive numbers, providing vector indices.
`rp`	the replacement value. Must be of the same type as `x`, and the same length or lenght of 1.

Value

Returns: VOID. This method modifies the object by reference.
Do not use assignments like `x <- ma_set(x, ...)`.
Since this function returns void, you'll just get `NULL`.

Examples

```
x <- as.mutatomic(1:16)
ma_set(x, i = 1:6, rp = 8.5) # 8.5 coerced to 8, because `x` is of type `integer`
print(x)
```

---

mutatomic_class *A Class of Mutable Atomic Objects*

Description

For the package overview, see mutatomic_help.

The `mutatomic` class is a mutable version of atomic classes.
It works exactly the same in all aspects as regular atomic classes.
There is only one real difference:
Pass-by-reference functions in (primarily) the 'squarebrackets' and 'broadcast' packages only accept atomic objects when they are of class `mutatomic`, for greater safety.
In all other aspects, `mutatomic` objects are the same as R's regular atomic objects, including the behaviour of the `[<-` operator .

Exposed functions (beside the S3 methods):

- `mutatomic()`: create a `mutatomic` object from given data.

- `couldb.mutatomic()`: checks if an object could become `mutatomic`.
  An objects can become `mutatomic` if it is one of the following types:
  logical, integer, double, character, complex, raw.
  Factors can never be `mutatomic`.
- `typecast.mutatomic()` type-casts and possibly reshapes a (mutable) atomic object, and returns a `mutatomic` object.
  Does not preserve dimension names if dimensions are changed.

Usage

```
mutatomic(data, names = NULL, dim = NULL, dimnames = NULL)

as.mutatomic(x, ...)

## Default S3 method:
as.mutatomic(x, ...)

is.mutatomic(x)

couldb.mutatomic(x)

typecast.mutatomic(x, type = typeof(x), dims = dim(x))

## S3 method for class 'mutatomic'
c(..., use.names = TRUE)

## S3 method for class 'mutatomic'
x[...]

## S3 replacement method for class 'mutatomic'
x[...] <- value

## S3 method for class 'mutatomic'
format(x, ...)

## S3 method for class 'mutatomic'
print(x, ...)
```

Arguments

`data`	atomic vector giving data to fill the `mutatomic` object.
`names`, `dim`, `dimnames`	see setNames and array.
`x`	an atomic object.
`...`	method dependent arguments.
`type`	a string giving the type; see typeof.
`dims`	integer vector, giving the new dimensions.
`use.names`	Boolean, indicating if names should be preserved.
`value`	see Extract.

Value

For `mutatomic()`, `as.mutatomic()`, `typecast.mutatomic()`:
Returns a `mutatomic` object.

For `is.mutatomic()`:
Returns `TRUE` if the object is `mutatomic`, and returns `FALSE` otherwise.

For `couldb.mutatomic()`:
Returns `TRUE` if the object is one of the following types:
logical, integer, double, character, complex, raw.
Returns `FALSE` otherwise.

Warning

Always use the exported functions given by 'mutatomic' to create a `mutatomic` object, as they make necessary checks.
Circumventing these checks may break things!

Examples

```
x <- mutatomic(
  1:20, dim = c(5, 4), dimnames = list(letters[1:5], letters[1:4])
)
x
typecast.mutatomic(x, "character")

x <- matrix(1:10, ncol = 2)
x <- as.mutatomic(x)
is.mutatomic(x)
print(x)
x[, 1]
x[] <- as.double(x)
print(x)
is.mutatomic(x)
```

---

setapply *Apply Functions Over mutatomic Matrix Margins By Reference*

Description

The `setapply()` function applies a functions over the rows or columns of a mutatomic matrix, through pass-by-reference semantics.

The `setapply()` is a bit faster and uses less memory than apply.

Usage

```
setapply(x, MARGIN, FUN)
```

Arguments

x	a mutatomic 2-dimensional array (i.e. a matrix). Arrays of other than 2 dimensions are not supported.
MARGIN	a single integer scalar, giving the subscript to apply the function over. 1 indicates rows, 2 indicates columns.
FUN	the function to be applied. The function must return a vector of the same type of x, and the appropriate length (i.e. length ncol(x) when MARGIN == 1 or length nrow(x) when MARGIN == 2).

Value

Returns: VOID. This function modifies the object by reference.
Do NOT use assignment like x <- setapply(x, ...).
Since this function returns void, you'll just get NULL.

Examples

```
# re-order elements matrix by reference ====
x <- mutatomic::mutatomic(1:20, dim = c(5,4))
print(x)
setapply(x, 1, FUN = \(x)x[c(4,1,3,2)])
print(x)


# sort elements of matrix by reference ====
x <- mutatomic::mutatomic(20:1, dim = c(5,4))
print(x)
setapply(x, 2, FUN = sort)
print(x)
```

---

stopifnot_ma_safe2mutate

Check If an Atomic Object is Safe to Mutate

Description

Arguably the most important function of 'mutatomic' for package development is the stopifnot_ma_safe2mutate() function, which checks if an atomic object is actually safe to mutate.
Package authors who wish to use 'mutatomic' for pass-by-reference semantics ought to check if an atomic object is safe to mutate using the stopifnot_ma_safe2mutate() function;
otherwise things might break in 'R'.

Usage

```
stopifnot_ma_safe2mutate(sym, envir, .abortcall)
```

Arguments

sym	the symbol of the object; i.e. substitute(x).
envir	the environment where the object resides; i.e. parent.frame(n = 1).
.abortcall	environment where the error message is passed to.

Value

Nothing. Only gives an error if the object is not safe to mutate.

Examples

```
x <- 1:16

testfun <- function(x) {
  stopifnot_ma_safe2mutate(substitute(x), parent.frame(n = 1), sys.call())
}

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    testfun(x),
    pattern = "not a 'mutatomic' object"
  )
}

mylist <- list(
  a = mutatomic(1:10)
)

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    testfun(mylist$a),
    pattern = "only objects that exist as variables can be modified by reference"
  )
}

lockBinding("x", environment())

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    testfun(x),
    pattern = "cannot change value of locked binding for"
  )
}


rm(list = "x")
```

Index