Package 'broadcast'

March 28, 2025

Title Simple Broadcasted Operations for Atomic and Recursive Arrays with Minimal Dependencies

Version 0.0.0.9

Description Implements simple broadcasted operations for atomic and recursive arrays.
Besides linking to 'Rcpp',
'broadcast' does not depend on, vendor, link to, include, or otherwise use any external libraries;
'broadcast' was essentially made from scratch and can be installed out-of-the-box.
The implementations available in 'broadcast' include, but are not limited to, the following.
1) A faster, more memory efficient, and broadcasted version of `abind()`,
for binding arrays along an arbitrary dimension;
2) A set of type-specific functions for broadcasted element-wise binary operations;
they support a large set of relational-, arithmetic-, Boolean-, and string operations.
3) A Broadcasted implementation of `ifelse()`;
4) A Broadcasted apply-like function;
5) The `acast()` function, for casting/pivoting an array into a new dimension.
The functions in the 'broadcast' package strive to minimize computation time and memory usage
(which is not just good for efficient computing, but also for the environment).

License MPL-2.0 | file LICENSE

Encoding UTF-8

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Depends R (>= 4.2.0)

Imports Rcpp (>= 1.0.14)

Suggests data.table, tinytest, roxygen2, abind, funr

Contents

`aaa00_broadcast_help` *broadcast: Simple Broadcasted Operations for Atomic and Recursive Arrays with Minimal Dependencies*

Description

broadcast:
Simple Broadcasted Binding and Binary Operations for Atomic and Recursive Arrays with Minimal Dependencies.

Implements simple broadcasted operations for atomic and recursive arrays.

Besides linking to 'Rcpp', 'broadcast' does not depend on, vendor, link to, include, or otherwise use any external libraries; 'broadcast' was essentially made from scratch and can be installed out-of-the-box.

The implementations available in 'broadcast' include, but are not limited to, the following:

1. A faster, more memory efficient, and broadcasted version of `abind()`, for binding arrays along an arbitrary dimension;
2. A set of type-specific functions for broadcasted element-wise binary operations; they support a large set of relational-, arithmetic-, Boolean-, and string operations.
3. A Broadcasted implementation of `ifelse()`;
4. A Broadcasted apply-like function;
5. The `acast()` function, for casting/pivoting an array into a new dimension.

The functions in the 'broadcast' package strive to minimize computation time and memory usage (which is not just good for efficient computing, but also for the environment).

Getting Started

An introduction and overview of the package can be found on the website.

Functions

Functions for broadcasted element-wise binary operations
'broadcast' provides a set of functions for broadcasted element-wise binary operations with broadcasting.
These functions use an API similar to the outer function.

The following functions for type-specific binary operations are available:

- bc.b: Boolean operations;
- bc.i: integer (53bit) arithmetic and relational operations;
- bc.d: decimal (64bit) arithmetic and relational operations;
- bc.cplx: complex arithmetic and (in)equality operations;
- bc.str: string (in)equality, concatenation, and distance operations;
- bc.list: apply any 'R' function to 2 recursive arrays with broadcasting.

Binding Implementations
'broadcast' provides 3 binding implementations:
bind_mat, bind_array, and bind_dt.

General functions
'broadcast' also comes with 2 general broadcasted functions:

- bc_ifelse: Broadcasted version of ifelse.
- bcapply: Broadcasted apply-like function.

Other functions
'broadcast' also provides type-casting functions, which preserve names and dimensions - convenient for arrays.

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_broadcast_bind` *Details on the Binding Implementations in 'broadcast'*

Description

This help page gives additional details on the binding implementations in the 'broadcast' package.

Empty inputs

If argument `input` has length `0`, or it contains exclusively objects where one or more dimensions are `0`, an error is returned.

If `input` has length `1`, these functions simply return `input[[1L]]`.

Differences with `abind()`, `rbind()`/`cbind()`

The API of `bind_array()` is inspired by the fantastic `abind::abind` function by Tony Plare & Richard Heiberger (2016).
But `bind_array()` differs considerably from `abind::abind` in the following ways:

- `bind_array()` differs from `abind::abind` in that it can handle recursive arrays properly (the `abind::abind` function would unlist everything to atomic arrays, ruining the structure).
- `bind_array()` allows for broadcasting, while `abind::abind` does not support broadcasting.
- `bind_array()` is generally faster than `abind::abind`, as `bind_array()` relies heavily on 'C' and 'C++' code.
- unlike `abind::abind`, `bind_array()` only binds (atomic/recursive) arrays and matrices. `bind_array()`does not attempt to convert things to arrays when they are not arrays, but will give an error instead.
  This saves computation time and prevents unexpected results.
- `bind_array()` has more streamlined naming options, compared to `abind::abind`.

`bind_mat()` is a modified version of base R's `rbind()`/`cbind()` functions.
`bind_mat()` differs from `rbind()`/`cbind()` in the following ways:

- it has more streamlined naming options/
- `bind_mat()` gives an error when fractional recycling is attempted (like binding `1:3` with `1:10`).

`acast` *Simple and Fast Casting/Pivoting of an Array*

Description

The `acast()` function spreads subsets of an array margin over a new dimension.
Written in 'C' and 'C++' for high speed and memory efficiency.

Roughly speaking, `acast()` can be thought of as the "array" analogy to `data.table::dcast`.
But note 2 important differences:

- `acast()` works on arrays instead of data.tables.
- `acast()` casts into a completely new dimension (namely `ndim(x) + 1`), instead of casting into new columns.

Usage

```
acast(x, margin, grp, fill = FALSE)
```

Arguments

x	an atomic or recursive array.
margin	a scalar integer, specifying the margin to cast from.
grp	a factor, where length(grp) == dim(x)[margin], with at least 2 unique values, specifying which indices of dim(x)[margin] belong to which group. Each group will be cast onto a separate index of dimension ndim(x) + 1. Unused levels of grp will be dropped. Any NA values or levels found in grp will result in an error.
fill	Boolean, indicating if missing values should be filled. This is used in case the levels of grp do not have equal frequencies, and thus additional values must be filled. If x is atomic but not raw, missing values are filled with NA. If x is recursive, missing values are filled with list(NULL). If x is of type raw, uneven groupings are not supported.

Details

For the sake of illustration, consider a matrix x and a grouping factor grp.
Let the integer scalar k represent a group in grp, such that k $\in$ 1:nlevels(grp).
Then the code
out = acast(x, margin = 1, grp = grp)
essentially performs the following for every group k:

- copy-paste the subset x[grp == k,] to the subset out[, , k].

Please see the examples section to get a good idea on how this function casts an array.
A more detailed explanation of the acast() function can be found on the website.

Value

An array with the following properties:

- the number of dimensions of the output array is equal to ndim(x) + 1;
- the dimensions of the output array is equal to c(dim(x), max(tabulate(grp)));
- the dimnames of the output array is equal to c(dimnames(x), list(levels(grp))).

Back transformation

From the casted array,
out = acast(x, margin, grp),
one can get the original x back by using
back = asplit(out, ndim(out)) |> bind_array(along = margin).
Note, however, the following about the back-transformed array back:

- back will be ordered by grp along dimension margin;

- if the levels of grp did not have equal frequencies, then dim(back)[margin] > dim(x)[margin], and back will have more missing values than x.

Examples

```
x <- cbind(id = c(rep(1:3, each = 2), 1), grp = c(rep(1:2, 3), 2), val = rnorm(7))
print(x)

grp <- as.factor(x[, 2])
levels(grp) <- c("a", "b")
margin <- 1L

acast(x, margin, grp, fill = TRUE)
```

---

bc.b *Broadcasted Boolean Operations*

Description

The bc.b() function performs broadcasted Boolean operations on 2 logical (or 32bit integer) arrays.

Please note that these operations will treat the input as Boolean.
Therefore, something like bc.b(1, 2, "==") returns TRUE, because both 1 and 2 are TRUE when cast as Boolean.

Usage

```
bc.b(x, y, op)
```

Arguments

x, y	conformable logical (or 32bit integer) arrays.
op	a single string, giving the operator. Supported Boolean operators: &, |, xor, nand, ==, !=, <, >, <=, >=.

Value

A logical array as a result of the broadcasted Boolean operation.

Examples

```
x.dim <- c(4:2)
x.len <- prod(x.dim)
x.data <- sample(c(TRUE, FALSE, NA), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(4,1,1))

bc.b(x, y, "&")
bc.b(x, y, "|")
bc.b(x, y, "xor")
bc.b(x, y, "nand")
bc.b(x, y, "==")
bc.b(x, y, "!=")
```

`bc.cplx` *Broadcasted Complex Numeric Operations*

Description

The `bc.cplx()` function performs broadcasted complex numeric operations pairs of arrays.

Note that `bc.cplx()` uses more strict `NA` checks than base 'R':
If for an element of either `x` or `y`, either the real or imaginary part is `NA` or `NaN`, than the result of the operation for that element is necessarily `NA`.

Usage

```
bc.cplx(x, y, op)
```

Arguments

`x`, `y`	conformable atomic arrays of type `complex`.
`op`	a single string, giving the operator. Supported arithmetic operators: +, -, *, /. Supported relational operators: ==, !=.

Value

For arithmetic operators:
A complex array as a result of the broadcasted arithmetic operation.

For relational operators:
A logical array as a result of the broadcasted relational comparison.

Examples

```
x.dim <- c(4:2)
x.len <- prod(x.dim)
gen <- function() sample(c(rnorm(10), NA, NA, NaN, NaN, Inf, Inf, -Inf, -Inf))
x <- array(gen() + gen() * -1i, x.dim)
y <- array(gen() + gen() * -1i, c(4,1,1))

bc.cplx(x, y, "==")
bc.cplx(x, y, "!=")

bc.cplx(x, y, "+")

bc.cplx(array(gen() + gen() * -1i), array(gen() + gen() * -1i), "==")
bc.cplx(array(gen() + gen() * -1i), array(gen() + gen() * -1i), "!=")

x <- gen() + gen() * -1i
y <- gen() + gen() * -1i
out <- bc.cplx(array(x), array(y), "*")
cbind(x, y, x*y, out)
```

`bc.d` *Broadcasted Decimal Numeric Operations*

Description

The `bc.d()` function performs broadcasted decimal numeric operations on 2 numeric or logical arrays.
`bc.num()` is an alias for `bc.d()`.

Usage

```
bc.d(x, y, op, prec = sqrt(.Machine$double.eps))

bc.num(x, y, op, prec = sqrt(.Machine$double.eps))
```

Arguments

`x, y`	conformable logical or numeric arrays.
`op`	a single string, giving the operator. Supported arithmetic operators: +, -, *, /, ^, pmin, pmax. Supported relational operators: ==, !=, <, >, <=, >=, d==, d!=, d<, d>, d<=, d>=.
`prec`	a single number between 0 and 0.1, giving the machine precision to use. Only relevant for the following operators: d==, d!=, d<, d>, d<=, d>= See the %d==%, %d!=%, %d<%, %d>%, %d<=%, %d>=% operators from the 'tinycodet' package for details.

Value

For arithmetic operators:
A numeric array as a result of the broadcasted decimal arithmetic operation.

For relational operators:
A logical array as a result of the broadcasted decimal relational comparison.

Examples

```
x.dim <- c(4:2)
x.len <- prod(x.dim)
x.data <- sample(c(NA, 1.1:1000.1), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(4,1,1))

bc.d(x, y, "+")
bc.d(x, y, "-")
bc.d(x, y, "*")
bc.d(x, y, "/")
bc.d(x, y, "^")

bc.d(x, y, "==")
bc.d(x, y, "!=")
bc.d(x, y, "<")
bc.d(x, y, ">")
bc.d(x, y, "<=")
bc.d(x, y, ">=")
```

---

bc.i *Broadcasted Integer Numeric Operations with Extra Overflow Protection*

Description

The `bc.i()` function performs broadcasted integer numeric operations on 2 numeric or logical arrays.

Please note that these operations will treat the input as 53bit integers, and will efficiently truncate when necessary.
Therefore, something like `bc.i(1, 1.5, "==")` returns `TRUE`, because `trunc(1.5)` equals `1`.

Usage

```
bc.i(x, y, op)
```

Arguments

x, y	conformable logical or numeric arrays.
op	a single string, giving the operator. Supported arithmetic operators: +, -, *, gcd, %%, ^, pmin, pmax. Supported relational operators: ==, !=, <, >, <=, >=. The "gcd" operator performs the Greatest Common Divisor" operation, using the Euclidean algorithm..

Value

For arithmetic operators:
A numeric array of whole numbers, as a result of the broadcasted arithmetic operation.
Base 'R' supports 53 bit integers, which thus range from approximately `-9` quadrillion to `+9` quadrillion.
Values outside of this range will be returned as `-Inf` or `Inf`, as an extra protection against integer overflow.

For relational operators:
A logical array as a result of the broadcasted integer relational comparison.

Examples

```
x.dim <- c(4:2)
x.len <- prod(x.dim)
x.data <- sample(c(NA, 1.1:1000.1), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(4,1,1))

bc.i(x, y, "+")
bc.i(x, y, "-")
bc.i(x, y, "*")
bc.i(x, y, "gcd") # greatest common divisor
bc.i(x, y, "^")

bc.i(x, y, "==")
bc.i(x, y, "!=")
bc.i(x, y, "<")
bc.i(x, y, ">")
bc.i(x, y, "<=")
bc.i(x, y, ">=")
```

---

bc.list *Broadcasted Operations for Recursive Arrays*

Description

The `bc.list()` function performs broadcasted operations on 2 Recursive arrays.

Usage

```
bc.list(x, y, f)
```

Arguments

x, y	conformable Recursive arrays (i.e. arrays of type `list`).
f	a function that takes in exactly **2** arguments, and **returns** a result that can be stored in a single element of a list.

Value

A recursive array.

Examples

```
x.dim <- c(c(10, 2,2))
x.len <- prod(x.dim)

gen <- function(n) sample(list(letters, month.abb, 1:10), n, TRUE)

x <- array(gen(10), x.dim)
y <- array(gen(10), c(10,1,1))

bc.list(
  x, y,
  \(x, y)c(length(x) == length(y), typeof(x) == typeof(y))
)
```

---

bc.str *Broadcasted String Operations*

Description

The `bc.str()` function performs broadcasted string operations on pairs of arrays.

Usage

```
bc.str(x, y, op)
```

Arguments

x, y	conformable atomic arrays of type `character`.
op	a single string, giving the operator. Supported concatenation operators: +. Supported relational operators: ==, !=. Supported distance operators: levenshtein.

Value

For concatenation operation:
A character array as a result of the broadcasted concatenation operation.

For relational operation:
A logical array as a result of the broadcasted relational comparison.

For distance operation:
An integer array as a result of the broadcasted distance measurement.

References

The 'C++' code for the Levenshtein edit string distance is based on the code found in https://rosettacode.org/wiki/Levenshtein_distance#C++

Examples

```
# string concatenation:
x <- array(letters, c(10, 2, 1))
y <- array(letters, c(10,1,1))
bc.str(x, y, "+")


# string (in)equality:
bc.str(array(letters), array(letters), "==")
bc.str(array(letters), array(letters), "!=")


# string distance (Levenshtein):
x <- array(month.name, c(12, 1))
y <- array(month.abb, c(1, 12))
out <- bc.str(x, y, "levenshtein")
dimnames(out) <- list(month.name, month.abb)
print(out)
```

---

bcapply *Apply a Function to 2 Broadcasted Arrays*

Description

The `bcapply()` function applies a function to 2 arrays element-wise with broadcasting.

Usage

```
bcapply(x, y, f, v = NULL)
```

Arguments

x, y	conformable atomic or recursive arrays.
f	a function that takes in exactly **2** arguments, and **returns** a result that can be stored in a single element of a recursive or atomic array.
v	either NULL, or single string, giving the scalar type for a single iteration. If NULL (default) or "list", the result will be a recursive array. If it is certain that, for every iteration, f() always results in a **single atomic scalar**, the user can specify the type in v to pre-allocate the result. Pre-allocating the results leads to slightly faster and more memory efficient code. NOTE: Incorrectly specifying v leads to undefined behaviour; when unsure, leave v at its default value.

Value

An atomic or recursive array with dimensions bc_dim(x, y).

Examples

```
x.dim <- c(c(10, 2,2))
x.len <- prod(x.dim)

gen <- function(n) sample(list(letters, month.abb, 1:10), n, TRUE)

x <- array(gen(10), x.dim)
y <- array(gen(10), c(10,1,1))

f <- function(x, y) list(x, y)
bcapply(x, y, f)
```

---

bc_dim *Predict Broadcasted dimensions*

Description

bc_dim(x, y) gives the dimensions an array would have, as the result of an broadcasted binary element-wise operation between 2 arrays x and y.

Usage

```
bc_dim(x, y)
```

Arguments

x, y	an atomic or recursive array.

Value

Returns the recycled array.

Examples

```
x.dim <- c(4:2)
x.len <- prod(x.dim)
x.data <- sample(c(TRUE, FALSE, NA), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(4,1,1))

dim(bc.b(x, y, "&")) == bc_dim(x, y)
dim(bc.b(x, y, "|")) == bc_dim(x, y)
```

`bc_ifelse` *Broadcasted Ifelse*

Description

The `bc_ifelse()` function performs a broadcasted form of ifelse.

Usage

```
bc_ifelse(cond, yes, no)
```

Arguments

cond	`logical` vector or array with the length equal to `prod(bc_dim(yes, no))`.
yes, no	conformable arrays of the same type. All atomic types are supported except for the type of raw. Recursive arrays of type list are also supported.

Value

The ouput, here referred to as `out`, will be an array of the same type as `yes` and `no`.
After broadcasting `yes` against `no`, given any element index `i`, the following will hold for the output:

- when `cond[i] == TRUE`, `out[i]` is `yes[i]`;
- when `cond[i] == FALSE`, `out[i]` is `no[i]`;
- when `cond[i]` is `NA`, `out[i]` is `NA` when `yes` and `no` are atomic, and `out[i]` is `list(NULL)` when `yes` and `no` are recursive.

Examples

```
x.dim <- c(c(10, 2,2))
x.len <- prod(x.dim)

gen <- function(n) sample(list(letters, month.abb, 1:10), n, TRUE)

x <- array(gen(10), x.dim)
```

```
y <- array(gen(10), c(10,1,1))

cond <- bc.list(
  x, y,
  \(x, y)c(length(x) == length(y) && typeof(x) == typeof(y))
) |> as_bool()

bc_ifelse(cond, yes = x, no = y)
```

`bind` *Dimensional Binding of Objects*

Description

The `bind_`implementations provide dimensional binding functionalities.

The following implementations are available:

- `bind_mat()` binds dimensionless (atomic/recursive) vectors and (atomic/recursive) matrices row- or column-wise.
  Returns a matrix.
  Allows for linear/vector recycling.
- `bind_array()` binds (atomic/recursive) arrays and (atomic/recursive) matrices.
  Returns an array.
  Allows for broadcasting.
- `bind_dt()` binds data.tables and other data.frame-like objects.
  Returns a `data.table`.
  This function is only available if the 'data.table' package is installed.
  The `bind_dt()` function is faster than `do.call(cbind, ...)` or `do.call(rbind, ...)` for regular `data.frame` objects.

Usage

```
bind_mat(input, along, name_deparse = TRUE, comnames_from = 1L)

bind_array(
  input,
  along,
  rev = FALSE,
  ndim2bc = 1L,
  name_along = TRUE,
  comnames_from = 1L
)

bind_dt(input, along, ...)
```

Arguments

`input`	a list of only the appropriate objects. If `input` is named, its names will be used for the names of dimension `along` of the output, as far as possible.

`along` — a single integer, indicating the dimension along which to bind the dimensions.
I.e. use `along = 1` for row-binding, `along = 2` for column-binding, etc.
For arrays, additional flexibility is available:

- Specifying `along = 0` will bind the arrays on a new dimension before the first, making `along` the new first dimension.
- Specifying `along = N + 1`, with `N = max(lst.ndim(input))`, will create an additional dimension (`N + 1`) and bind the arrays along that new dimension.

`name_deparse` — Boolean, for `bind_mat()`.
Indicates if dimension `along` should be named.
Uses the naming method from rbind/cbind itself.

`comnames_from` — either integer scalar or `NULL`, for `bind_mat()` and `bind_array()`.
Indicates which object in `input` should be used for naming the shared dimension.
If `NULL`, no communal names will be given.
For example:
When binding columns of matrices, the matrices will share the same rownames.
Using `comnames_from = 10` will then result in `bind_array()` using `rownames(input[[10]])` for the rownames of the output.

`rev` — Boolean, for `bind_array()` only.
Indicates if `along` should be reversed, counting backwards.
If `FALSE` (default), `along` works like normally; if `TRUE`, `along` is reversed.
I.e. `along = 0, rev = TRUE` is equivalent to `along = N+1, rev = FALSE`;
and `along = N+1, rev = TRUE` is equivalent to `along = 0, rev = FALSE`;
with `N = max(lst.ndim(input))`.

`ndim2bc` — non-negative integer, for `bind_array` only.
Specify here the maximum number of dimensions that are allowed to be broadcasted when binding arrays.
If `ndim2bc = 0L`, **no** broadcasting will be allowed at all.

`name_along` — Boolean, for `bind_array()`.
Indicates if dimension `along` should be named.
The examples section illustrates the naming behaviour.

`...` — arguments to be passed to `data.table::rbindlist()`.

Details

For in-depth information about the binding implentations in the 'broadcast' package, please refer to broadcast_bind.

Value

The bound object.

References

Plate T, Heiberger R (2016). *abind: Combine Multidimensional Arrays*. R package version 1.4-5, https://CRAN.R-project.org/package=abind.

Examples

```
# bind_array ====

# here, atomic and recursive arrays are mixed,
# resulting in recursive arrays

# creating the arrays
x <- c(
  lapply(1:3, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:3, \(x)sample(1:10)),
  lapply(1:3, \(x)rnorm(10)),
  lapply(1:3, \(x)sample(letters))
)
x <- matrix(x, 4, 3, byrow = TRUE)
dimnames(x) <- list(letters[1:4], LETTERS[1:3])
print(x)

y <- matrix(1:12, 4, 3)
print(y)

# binding the arrays
input <- list(x = x, y = y)
bind_array(input, along = 0L) # binds on new dimension before first
bind_array(input, along = 1L) # binds on first dimension (i.e. rows)
bind_array(input, along = 2L)
bind_array(input, along = 3L) # bind on new dimension after last

bind_array(input, along = 0L, TRUE) # binds on new dimension after last
bind_array(input, along = 1L, TRUE) # binds on last dimension (i.e. columns)
bind_array(input, along = 2L, TRUE)
bind_array(input, along = 3L, TRUE) # bind on new dimension before first


# binding, with empty arrays
emptyarray <- array(numeric(0L), c(0L, 3L))
dimnames(emptyarray) <- list(NULL, paste("empty", 1:3))
print(emptyarray)
input <- list(x = x, y = emptyarray)
bind_array(input, along = 1L, comnames_from = 2L) # row-bind


# Illustrating `name_along`

x <- array(1:20, c(5, 3), list(NULL, LETTERS[1:3]))
y <- array(-1:-20, c(5, 3))
z <- array(-1:-20, c(5, 3))

bind_array(list(a = x, b = y, z), 2L)

bind_array(list(x, y, z), 2L)

bind_array(list(a = unname(x), b = y, c = z), 2L)

bind_array(list(x, a = y, b = z), 2L)
```

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

# bind_mat ====

# here, atomic and recursive matrices are mixed,
# resulting in a recursive matrix

x <- c(
  lapply(1:3, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:3, \(x)sample(1:10)),
  lapply(1:3, \(x)rnorm(10)),
  lapply(1:3, \(x)sample(letters))
)
x <- matrix(x, 4, 3, byrow = TRUE)
dimnames(x) <- list(letters[1:4], LETTERS[1:3])
print(x)

y <- matrix(1:12, 4, 3)
print(y)

bind_mat(list(x = x, y = y), 2L)



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

# bind_dt ====
if(require(data.table)) {
  x <- data.frame(a = 1:12, b = month.abb) # data.frame
  y <- data.table::data.table(a = 1:12, b = month.abb) # data.table

  bind_dt(list(x = x, y = y), 2L) # column bind

  bind_dt(list(x = x, y = y), 1L) # row bind

}
```

`properties` *Small property functions*

Description

`ndim()` returns the number of dimensions of an object.
`lst.ndim()` returns the number of dimensions of every list-element.
`lst.typeof()` returns the (internal) type of every list-element.

Usage

```
ndim(x)
```

```
lst.ndim(x)

lst.typeof(x)
```

Arguments

x	an object. For functions starting with `lst.`, `x` must be a list (i.e. recursive vector or recursive array).

Value

For `ndim()`: an integer scalar.
For `lst.ndim()`: an integer vector, with the same length, names and dimensions as `x`.
For `lst.typeof()`: a character vector, with the same length, names and dimensions as `x`.

Examples

```
# matrix example ====
x <- list(
  array(1:10, 10),
  array(1:10, c(2, 5)),
  array(c(letters, NA), c(3,3,3))
)
lst.ndim(x)
lst.typeof(x)
```

---

rep_dim *Replicate Array Dimensions*

Description

The `rep_dim()` function replicates array dimensions until the specified dimension sizes are reached, and returns the array.

The various broadcasting functions recycle array dimensions virtually, meaning little to no additional memory is needed.
The `rep_dim()` function, however, physically replicates the dimensions of an array (and thus actually occupies additional memory space).

Usage

```
rep_dim(x, tdim)
```

Arguments

x	an atomic or recursive array or matrix.
tdim	an integer vector, giving the target dimension to reach.

Value

Returns the replicated array.

Examples

```
x <- matrix(1:9, 3,3)
colnames(x) <- LETTERS[1:3]
rownames(x) <- letters[1:3]
names(x) <- month.abb[1:9]
print(x)

rep_dim(x, c(3,3,2)) # replicate to larger size
```

`typecast` *Atomic and List Type Casting With Names and Dimensions Preserved*

Description

Type casting usually strips away attributes of objects.
The functions provided here preserve dimensions, dimnames, and names, which may be more convenient for arrays and array-like objects.

The functions are as follows:

- `as_bool()`: converts object to atomic type `logical` (`TRUE`, `FALSE`, `NA`).
- `as_int()`: converts object to atomic type `integer`.
- `as_dbl()`: converts object to atomic type `double` (AKA numeric).
- `as_chr()`: converts object to atomic type `character`.
- `as_cplx()`: converts object to atomic type `complex`.
- `as_raw()`: converts object to atomic type `raw`.
- `as_list()`: converts object to recursive type `list`.

`as_num()` is an alias for `as_dbl()`.
`as_str()` is an alias for `as_chr()`.

See also typeof.

Usage

```
as_bool(x, ...)

as_int(x, ...)

as_dbl(x, ...)

as_num(x, ...)
```

```
as_chr(x, ...)

as_str(x, ...)

as_cplx(x, ...)

as_raw(x, ...)

as_list(x, ...)
```

Arguments

x	an R object.
...	further arguments passed to or from other methods.

Value

The converted object.

Examples

```
# matrix example ====
x <- matrix(sample(-1:28), ncol = 5)
colnames(x) <- month.name[1:5]
rownames(x) <- month.abb[1:6]
names(x) <- c(letters[1:20], LETTERS[1:10])
print(x)

as_bool(x)
as_int(x)
as_dbl(x)
as_chr(x)
as_cplx(x)
as_raw(x)


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

# factor example ====
x <- factor(month.abb, levels = month.abb)
names(x) <- month.name
print(x)

as_bool(as_int(x) > 6)
as_int(x)
as_dbl(x)
as_chr(x)
as_cplx(x)
as_raw(x)
```

Index