Package ‘lobstr’

July 3, 2019

Title  Visualize R Data Structures with Trees
Version  1.1.1
Description  A set of tools for inspecting and understanding R data structures inspired by str(). Includes ast() for visualizing abstract syntax trees, ref() for showing shared references, cst() for showing call stack trees, and obj_size() for computing object sizes.
License  GPL-3
URL  https://github.com/r-lib/lobstr
BugReports  https://github.com/r-lib/lobstr/issues
Depends  R (>= 3.2)
Imports  crayon, Rcpp, rlang (>= 0.3.0)
Suggests  covr, pillar, pkgdown, testthat
LinkingTo  Rcpp
Encoding  UTF-8
LazyData  true
RoxygenNote  6.1.1
NeedsCompilation  yes
Author  Hadley Wickham [aut, cre],
        RStudio [cph]
Maintainer  Hadley Wickham <hadley@rstudio.com>
Repository  CRAN
Date/Publication  2019-07-02 23:10:04 UTC

R topics documented:

  ast ................................................................. 2
  cst ................................................................. 3
  mem_used ......................................................... 4
  obj_addr ......................................................... 4
Description

This is a useful alternative to `str()` for expression objects.

Usage

```
ast(x)
```

Arguments

```
x
```
An expression to display. Input is automatically quoted, use `!!` to unquote if
you have already captured an expression object.

See Also

Other object inspectors: `ref`, `sxp`

Examples

```
# Leaves
ast(1)
ast(x)

# Simple calls
ast(f())
ast(f(x, 1, g(), h(i())))
ast(f())
ast(f(x)(y))
ast((x + 1))

# Displaying expression already stored in object
x <- quote(a + b + c)
ast(x)
ast(!!x)

# All operations have this same structure
ast(if (TRUE) 3 else 4)
ast(y <- x * 10)
ast(function(x = 1, y = 2) ( x + y ))
```
COperator precedence

ast(1 * 2 + 3)
ast(!1 + !!)

---

cst

Call stack tree

Description

Shows the relationship between calls on the stack. This function combines the results of `sys.calls()` and `sys.parents()` yielding a display that shows how frames on the call stack are related.

Usage

cst()

Examples

# If all evaluation is eager, you get a single tree
f <- function() g()
g <- function() h()
h <- function() cst()
f()

# You get multiple trees with delayed evaluation
try(f())

# Pay attention to the first element of each subtree: each
# evaluates the outermost call
f <- function(x) g(x)
g <- function(x) h(x)
h <- function(x) x
try(f(cst()))

# With a little ingenuity you can use it to see how NSE
# functions work in base R
with(mtcars, {cst(); invisible()})
invisible(subset(mtcars, {cst(); cyl == 0}))

# You can also get unusual trees by evaluating in frames
# higher up the call stack
f <- function() g()
g <- function() h()
h <- function() eval(quote(cst()), parent.frame(2))
f()
mem_used

How much memory is currently used by R?

Description

mem_used() wraps around gc() and returns the exact number of bytes currently used by R. Note that changes will not match up exactly to obj_size() as session specific state (e.g. .Last.value) adds minor variations.

Usage

mem_used()

Examples

prev_m <- 0; m <- mem_used(); m - prev_m

x <- 1:1e6
prev_m <- m; m <- mem_used(); m - prev_m
obj_size(x)

rm(x)
prev_m <- m; m <- mem_used(); m - prev_m

prev_m <- m; m <- mem_used(); m - prev_m

obj_addr

Find memory location of objects and their children.

Description

obj_addr() gives the address of the value that x points to; obj_addrs() gives the address of the components the list, environment, and character vector x point to.

Usage

obj_addr(x)

obj_addrs(x)

Arguments

x	An object

Details

obj_addr() has been written in such a way that it avoids taking references to an object.
obj_size

Examples

# R creates copies lazily
x <- 1:10
y <- x
obj_addr(x) == obj_addr(y)

y[1] <- 2L
obj_addr(x) == obj_addr(y)

y <- runif(10)
obj_addr(y)
z <- list(y, y)
obj_addrs(z)

y[2] <- 1.0
obj_addrs(z)
obj_addr(y)

# The address of an object is different every time you create it:
obj_addr(1:10)
obj_addr(1:10)
obj_addr(1:10)

obj_size

Calculate the size of an object.

Description

obj_size() computes the size of an object or set of objects; obj_sizes() breaks down the individual contribution of multiple objects to the total size.

Usage

obj_size(..., env = parent.frame())

obj_sizes(..., env = parent.frame())

Arguments

... Set of objects to compute size.

env Environment in which to terminate search. This defaults to the current environment so that you don’t include the size of objects that are already stored elsewhere.
Regardless of the value here, obj_size() never looks past the global or base environments.

Value

An estimate of the size of the object, in bytes.
Compared to `object.size()`

Compared to `object.size()`, `obj_size()`:

- Accounts for all types of shared values, not just strings in the global string pool.
- Includes the size of environments (up to `env`)
- Accurately measures the size of ALTREP objects.

Environments

`obj_size()` attempts to take into account the size of the environments associated with an object. This is particularly important for closures and formulas, since otherwise you may not realize that you’ve accidentally captured a large object. However, it’s easy to over count: you don’t want to include the size of every object in every environment leading back to the `emptyenv()`. `obj_size()` takes a heuristic approach: it never counts the size of the global environment, the base environment, the empty environment, or any namespace.

Additionally, the `env` argument allows you to specify another environment at which to stop. This defaults to the environment from which `obj_size()` is called to prevent double-counting of objects created elsewhere.

Examples

```r
# obj_size correctly accounts for shared references
x <- runif(1e4)
obj_size(x)

z <- list(a = x, b = x, c = x)
obj_size(z)

# this means that object size is not transitive
obj_size(x)
obj_size(z)
obj_size(x, z)

# use obj_size() to see the unique contribution of each component
obj_sizes(x, z)
obj_sizes(z, x)
obj_sizes(!!z)

# obj_size() also includes the size of environments
f <- function() {
  x <- 1:1e4
  a ~ b
}
obj_size(f())

#' # In R 3.5 and greater, `:` creates a special "ALTREP" object that only
#' stores the first and last elements. This will make some vectors much
#' smaller than you'd otherwise expect
obj_size(1:1e6)
```
Description

This tree display focuses on the distinction between names and values. For each reference-type object (lists, environments, and optional character vectors), it displays the location of each component. The display shows the connection between shared references using a locally unique id.

Usage

`ref(..., character = FALSE)`

Arguments

... One or more objects
character If TRUE, show references from character vector in to global string pool

See Also

Other object inspectors: `ast`, `sxp`

Examples

```r
x <- 1:100
ref(x)

y <- list(x, x, x)
ref(y)
ref(x, y)

e <- new.env()
e$x <- e
e$y <- list(x, e)
ref(e)

# Can also show references to global string pool if requested
ref(c("x", "x", "y"))
ref(c("x", "x", "y"), character = TRUE)
```
sxp

Inspect an object

Description

sxp(x) is similar to .Internal(inspect(x)), recursing into the C data structures underlying any R object. The main difference is the output is a little more compact, it recurses fully, and avoids getting stuck in infinite loops by using a depth-first search. It also returns a list that you can compute with, and carefully uses colour to highlight the most important details.

Usage

sxp(x, expand = character(), max_depth = 5L)

Arguments

x          Object to inspect
expand     Optionally, expand components of the true that are usually suppressed. Use:
          • "character" to show underlying entries in the global string pool.
          • "environment" to show the underlying hashables.
          • "altrep" to show the underlying data.
          • "call" to show the full AST (but ast() is usually superior)
          • "bytecode" to show generated bytecode.
max_depth  Maximum depth to recurse. Use max_depth = Inf (with care!) to recurse as deeply as possible. Skipped elements will be shown as . . . .

Details

The name sxp comes from SEXP, the name of the C data structure that underlies all R objects.

See Also

Other object inspectors: ast, ref

Examples

x <- list(
  TRUE,
  1L,
  runif(100),
  "3"
)
sxp(x)

# Expand "character" to see underlying CHARSXP entries in the global
# string pool
x <- c("banana", "banana", "apple", "banana")
sxp(x)
sxp(x, expand = "character")

# Expand altrep to see underlying data
x <- 1:10
sxp(x)
sxp(x, expand = "altrep")

# Expand environments to see the underlying implementation details
e1 <- new.env(hash = FALSE, parent = emptyenv(), size = 3L)
e2 <- new.env(hash = TRUE, parent = emptyenv(), size = 3L)
e1$x <- e2$x <- 1:10
sxp(e1)
sxp(e1, expand = "environment")
sxp(e2, expand = "environment")
Index

.Last.value, 4
.ast, 2, 7, 8
.ast(), 8
.cst, 3
.emptyenv(), 6
.mem_used, 4
.obj_addr, 4
.obj_addrs (obj_addr), 4
.obj_size, 5
.obj_size(), 4
.obj_sizes (obj_size), 5
.object.size(), 6
.ref, 2, 7, 8
.sxp, 2, 7, 8
.sys.calls(), 3
.sys.parents(), 3