Package 'lobstr'

October 13, 2022

Title Visualize R Data Structures with Trees

Version 1.1.2

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 MIT + file LICENSE

URL https://lobstr.r-lib.org/, https://github.com/r-lib/lobstr

BugReports https://github.com/r-lib/lobstr/issues

Depends R (>= 3.2)

Imports crayon, methods, prettyunits, rlang (>= 1.0.0)

Suggests covr, pillar, pkgdown, testthat (>= 3.0.0)

LinkingTo cpp11 (>= 0.4.2)

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.2.0

SystemRequirements C++11

NeedsCompilation yes

Author Hadley Wickham [aut, cre],
RStudio [cph]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2022-06-22 23:20:02 UTC

R topics documented:

`ast` *Display the abstract syntax tree*

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 } )

# Operator precedence
ast(1 * 2 + 3)
ast(!1 + !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 away that it avoids taking references to an object.

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 realise 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

```
# 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)
```

`ref` *Display tree of references*

Description

This tree display focusses 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

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

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

e <- new.env()
e$e <- e
e$x <- x
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 hashtables. • "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 environmnets 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")
```

`tree` *Pretty tree-like object printing*

Description

A cleaner and easier to read replacement for `str` for nested list-like objects

Usage

```
tree(
  x,
  ...,
  index_unnamed = FALSE,
  max_depth = 10L,
  max_length = 1000L,
  show_environments = TRUE,
  hide_scalar_types = TRUE,
  val_printer = crayon::blue,
  class_printer = crayon::silver,
  show_attributes = FALSE,
  remove_newlines = TRUE,
  tree_chars = box_chars()
)
```

Arguments

`x`	A tree like object (list, etc.)
`...`	Ignored (used to force use of names)
`index_unnamed`	Should children of containers without names have indices used as stand-in?
`max_depth`	How far down the tree structure should be printed. E.g. `1` means only direct children of the root element will be shown. Useful for very deep lists.

`max_length` How many elements should be printed? This is useful in case you try and print an object with 100,000 items in it.

`show_environments`
Should environments be treated like normal lists and recursed into?

`hide_scalar_types`
Should atomic scalars be printed with type and length like vectors? E.g. `x <- "a"` would be shown as `x<char [1]>: "a"` instead of `x: "a"`.

`val_printer` Function that values get passed to before being drawn to screen. Can be used to color or generally style output.

`class_printer` Same as `val_printer` but for the the class types of non-atomic tree elements.

`show_attributes`
Should attributes be printed as a child of the list or avoided?

`remove_newlines`
Should character strings with newlines in them have the newlines removed? Not doing so will mess up the vertical flow of the tree but may be desired for some use-cases if newline structure is important to understanding object state.

`tree_chars` List of box characters used to construct tree. Needs elements `$h` for horizontal bar, `$hd` for dotted horizontal bar, `$v` for vertical bar, `$vd` for dotted vertical bar, `$l` for l-bend, and `$j` for junction (or middle child).

Value

console output of structure

Examples

```
x <- list(
  list(id = "a", val = 2),
  list(
    id = "b",
    val = 1,
    children = list(
      list(id = "b1", val = 2.5),
      list(
        id = "b2",
        val = 8,
        children = list(
          list(id = "b21", val = 4)
        )
      )
    )
  ),
  list(
    id = "c",
    val = 8,
    children = list(
      list(id = "c1"),
      list(id = "c2", val = 1)
```

```
    )
  )
)

# Basic usage
tree(x)

# Even cleaner output can be achieved by not printing indices
tree(x, index_unnamed = FALSE)

# Limit depth if object is potentially very large
tree(x, max_depth = 2)

# You can customize how the values and classes are printed if desired
tree(x, val_printer = function(x) {
  paste0("_", x, "_")
})
```

`tree_label` *Build element or node label in tree*

Description

These methods control how the value of a given node is printed. New methods can be added if support is needed for a novel class

Usage

```
tree_label(x, opts)
```

Arguments

`x`	A tree like object (list, etc.)
`opts`	A list of options that directly mirrors the named arguments of tree. E.g. `list(val_printer = crayon::red)` is equivalent to `tree(..., val_printer = crayon::red)`.

Index