Package: r2c (via r-universe)

July 23, 2024

Title Fast Iterated Statistic Computation in R

Description Compiles a subset of R into machine code so that expressions composed with that subset can be applied repeatedly on varying data without interpreter overhead.

Version 0.3.0

License GPL-2 | GPL-3

URL https://github.com/brodieG/r2c

BugReports https://github.com/brodieG/r2c/issues

Suggests unitizer, stats

Imports utils, vetr (>= 0.2.14.9000)

RoxygenNote 7.2.3

Roxygen list(markdown = TRUE)

Encoding UTF-8

Collate 'util.R' 'preproc-copy.R' 'rename.R' 'optim.R' 'code-rep.R' 'code-concat.R' 'constants.R' 'code-unary.R' 'code-bin.R' 'code-mean.R' 'code-summary.R' 'code-numeric.R' 'code-subset.R' 'code-logical.R' 'code-pow.R' 'code-seq.R' 'code-loop.R' 'r2c-package.R' 'code-ifelse.R' 'code-assign-braces.R' 'code.R' 'preprocess.R' 'alloc.R' 'compile.R' 'group.R' 'load.R' 'run.R' 'size.R' 'window.R'

Repository https://fastverse.r-universe.dev

RemoteUrl https://github.com/brodieG/r2c

RemoteRef HEAD

RemoteSha aaa39300a572dee6d912604c2e5f0633c1bac7ad

Contents

`r2c-package` *Fast Iterated Statistic Computation in R*

Description

Compiles a subset of R into machine code so that expressions composed with that subset can be applied repeatedly on varying data without interpreter overhead.

Details

Quick Start:

- Look at the group statistic examples.

Basics:

- Supported functions: which R functions `r2c` can compile.
- Compilation facilities: how to compile R with `r2c`.
- Runners: how to execute your code iteratively by group or across windows.

Advanced:

- Performance: what makes `r2c` fast and what tasks it is best suited for.
- Memory: how `r2c` minimizes peak memory usage and fragmentation.
- Preprocessing: why `r2c` modifies R calls before translating them into C.
- Inspect: how to extract components of the "r2c_fun" objects.
- Control structures: why these are considered experimental, and how and why their semantics diverge from their base R counterparts.
- Expression types: why `r2c` distinguishes between iteration constant and varying calls, why some supported function parameters require constant calls, and why calls to unsupported functions are allowed in some circumstances.

`bsac` *Basic Split Apply Combine*

Description

Evaluates quoted expressions in the context of data split by group. Intended purely for testing against r2c calculations.

Usage

```
bsac(call, data, groups, MoreArgs = list(), enclos = parent.frame())
```

Arguments

`call`	quoted R call to apply to each group.
`data`	a numeric vector, or a list of equal length numeric vectors. If a named list, the vectors will be matched to `fun` parameters by those names. Elements without names are matched positionally. If a list must contain at least one vector. Conceptually, this parameter is used similarly to `envir` parameter to `base::eval` when that is a list.
`groups`	an integer, numeric, or factor vector. Alternatively, a list of equal-length such vectors, the interaction of which defines individual groups to organize the vectors in `data` into (multiple vectors not implemented yet). Numeric vectors are coerced to integer, thus copied. Vectors of integer type, but with different classes/attributes (other than factors) will be treated as integer vectors. The vectors must be the same length as those in `data`. NA values are considered one group. If a list, the result of the calculation will be returned as a "data.frame", otherwise as a named vector. Currently only one group vector is allowed, even when using list mode. Support for multiple group vectors and other types of vectors will be added in the future. Zero length groups are not computed on at all (e.g. missing factor levels, zero-length group vector).
`MoreArgs`	a list of R objects to pass on as iteration-constant arguments to `fun`. Unlike with `data`, each of the objects therein are passed in full to the native code for each iteration This is useful for arguments that are intended to remain constant across iterations. Matching of these objects to `fun` parameters is the same as for `data`, with positional matching occurring after the elements in `data` are matched.
`enclos`	environment to use as the enclosure to the data in the evaluation call (see `eval`).

Value

numeric vector

`first_vec` *Retrieve First Vector from Data*

Description

Designed to handle the case where `data` can be either a numeric vector or a list of numeric vectors.

Usage

```
first_vec(x)
```

Arguments

`x`	a numeric vector, or a (possibly empty) list of numeric vectors.

Value

if `data` is a list, the first element if it is a numeric vector, or an empty numeric vector if the list is empty. If `data` is a numeric vector, then `data`. Otherwise an error is thrown.

Examples

```
first_vec(1:5)
first_vec(runif(5))
first_vec(mtcars)
first_vec(matrix(1:4, 2))  # matrices treated as vectors
```

`group_exec` *Execute r2c Function Iteratively on Groups in Data*

Description

A runner that organizes `data` into groups as defined by `groups`, and executes the native code associated with `fun` iteratively with each group's portion of `data`.

Usage

```
group_exec(fun, data, groups, MoreArgs = list())
```

Arguments

`fun`	an "r2c_fun" function as produced by the compilation functions.
`data`	a numeric vector, or a list of equal length numeric vectors. If a named list, the vectors will be matched to `fun` parameters by those names. Elements without names are matched positionally. If a list must contain at least one vector. Conceptually, this parameter is used similarly to `envir` parameter to `base::eval` when that is a list.
`groups`	an integer, numeric, or factor vector. Alternatively, a list of equal-length such vectors, the interaction of which defines individual groups to organize the vectors in `data` into (multiple vectors not implemented yet). Numeric vectors are coerced to integer, thus copied. Vectors of integer type, but with different classes/attributes (other than factors) will be treated as integer vectors. The vectors must be the same length as those in `data`. NA values are considered one group. If a list, the result of the calculation will be returned as a "data.frame", otherwise as a named vector. Currently only one group vector is allowed, even when using list mode. Support for multiple group vectors and other types of vectors will be added in the future. Zero length groups are not computed on at all (e.g. missing factor levels, zero-length group vector).
`MoreArgs`	a list of R objects to pass on as iteration-constant arguments to `fun`. Unlike with `data`, each of the objects therein are passed in full to the native code for each iteration This is useful for arguments that are intended to remain constant across iterations. Matching of these objects to `fun` parameters is the same as for `data`, with positional matching occurring after the elements in `data` are matched.

Value

If `groups` is an atomic vector, a named numeric or integer vector with the results of executing `fun` on each group and the names set to the groups. Otherwise, a "data.frame" with the group vectors as columns and the result of the computation as the last column. It is likely the mechanism to induce vector or data frame outputs will change in the future.

See Also

Compilation for more details on the behavior and constraints of "r2c_fun" functions, package overview for other `r2c` concepts.

Other runners: `rollby_exec()`, `rolli_exec()`

Examples

```
r2c_mean <- r2cq(mean(x))
with(mtcars, group_exec(r2c_mean, hp, groups=cyl))

r2c_slope <- r2cq(
  sum((x - mean(x)) * (y - mean(y))) / sum((x - mean(x)) ^ 2)
)
with(mtcars, group_exec(r2c_slope, list(hp, qsec), groups=cyl))

## Parameters are generated in the order they are encountered
```

```
str(formals(r2c_slope))

## Data frame output, re-order arguments
with(
  mtcars,
  group_exec(r2c_slope, list(y=hp, x=qsec), groups=list(cyl))
)

## We can provide iteration-constant parameters (na.rm here):
r2c_sum_add_na <- r2cq(sum(x * y, na.rm=na.rm) / sum(y))
str(formals(r2c_sum_add_na))
a <- runif(10)
a[8] <- NA
weights <- c(.1, .1, .2, .2, .4)
g <- rep(1:2, each=5)
group_exec(
  r2c_sum_add_na, a, groups=g,
  MoreArgs=list(y=weights, na.rm=TRUE)  ## MoreArgs for iter-constant
)
group_exec(
  r2c_sum_add_na, a, groups=g,
  MoreArgs=list(y=-weights, na.rm=FALSE)
)

## Groups known to be sorted can save substantial time
n <- 1e7
x <- runif(1e7)
g <- cumsum(sample(c(TRUE, rep(FALSE, 99)), n, replace=TRUE))
identical(g, sort(g))  # sorted already!
system.time(res1 <- group_exec(r2c_mean, x, g))
system.time(res2 <- group_exec(r2c_mean, x, process_groups(g, sorted=TRUE)))
identical(res1, res2)

## We can also group by runs by lying about `sorted` status
x <- 1:8
g <- rep(rep(1:2, each=2), 2)
g
group_exec(r2c_mean, x, groups=list(g))
group_exec(r2c_mean, x, groups=process_groups(list(g), sorted=TRUE))
```

`lcurry` *Pre-Set Function Parameters*

Description

Create a new function from an existing function, but with parameters pre-set. This is a function intended for testing to simplify complex expressions involving the _exec functions. It merely stores the function expression to execute in the lexical environment it was created in. All symbols will be resolved at evaluation time, not at creation time.

Usage

```
lcurry(FUN, ...)
```

Arguments

`FUN`	the function to pre-set parameters for
`...`	parameters to pre-set

Details

This is inspired by a function originally from Byron Ellis, adapted by Jamie F Olson, and discovered by me via Peter Danenberg's `{functional}` (see packages `?functional::Curry` and `functional::CurryL`). The implementation here is different, in particular it makes it easy to see what the intended call is by displaying the function contents (see examples).

Value

`FUN` wrapped with pre-set parameters

Examples

```
sum_nona <- lcurry(sum, na.rm=TRUE)
sum_nona(c(1, NA, 2))
sum_nona
```

`loaded_r2c_dynlibs` *Manage* `r2c` *Dynamic Libraries*

Description

List or unload `r2c` loaded dynamic libraries. These functions are helpful for managing situations where there is a sufficiently large number of `r2c` functions created (hundreds) that there is a risk of exhausting the number of allowed open dynamic libraries (see note for `base::dyn.load`).

Usage

```
loaded_r2c_dynlibs()

unload_r2c_dynlibs(except = character())
```

Arguments

`except`	character vector of dynamic library names as produced by `loaded_r2c_dynlibs` to exclude from unloading.

Details

"r2c_fun" functions are designed to unload their associated dynamic libraries when they are garbage collected, but in our experience garbage collection on them is difficult to predict or force. The intended use of these function is to record loaded `r2c` dynamic libraries that we wish to preserve prior to creating a set that we wish to use and discard. Once we are done with the discardable set, we drop all `r2c` dynamic libraries that were not part of the previously recorded list (see examples).

`r2c` dynamic libraries are recognized solely by matching their names against the regular expression pattern `"^r2c-[a-z0-9]{10}$"`. All such libraries missing from the `except` parameter will be unloaded by `unload_r2c_dynlibs`, even if they were not created by `r2c`. Any `r2c` function that has its associated dynamic library unloaded will cease to work, so it only makes sense to unload libraries for functions known to be deleted.

See Also

r2c-compile for details on "r2c_fun" functions, `get_so_loc` to retrieve original dynamic library file system location from and "r2c_fun" object, `base::dyn.unload`, `base::getLoadedDLLs`.

Examples

```
except <- loaded_r2c_dynlibs()
tmp.r2c.fun <- r2cq(sum(x))
tmp.r2c.fun(1:10)
rm(tmp.r2c.fun)
gc()                         # gc should unload lib, but it often doesn't
unload_r2c_dynlibs(except)  # force unload
```

`mean1` *Single Pass Mean Calculation*

Description

`base::mean` does a two pass calculation that additional handles cases where `sum(x)` overflows doubles but `sum(x/n)` does not. This version is a single pass one that does not protect against the overflow case.

Usage

```
mean1(x, na.rm = FALSE)
```

Arguments

`x`	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`na.rm`	a logical evaluating to `TRUE` or `FALSE` indicating whether `NA` values should be stripped before the computation proceeds.

Value

scalar numeric

Examples

```
mean1(runif(10))

## Overflow even 80 bit long double:
mean1(rep(.Machine$double.xmax, 2^4))
mean(rep(.Machine$double.xmax, 2^4))

## Reduced precision
x <- runif(10) ^ 2
mean(x) - mean1(x)
```

`numeric_along` *Initialize a Numeric Vector Sized to Match Input*

Description

Generates a numeric vector of the same size as the input. Equivalent to `numeric(length(x))`. `numeric_alongn` supports multiple vectors in the input, for which the result size is the product of the lengths of the inputs.

Usage

```
numeric_along(along.with)

numeric_alongn(...)
```

Arguments

`along.with`	vector to use for sizing the result.
`...`	vectors to use for sizing the result.

Value

a zero numeric vector the same length as the product of the lengths of all the vectors in provided as inputs.

See Also

base::numeric

Examples

```
numeric_along(1:3)
numeric_alongn(1:3, 1:2)
```

`process_groups` *Compute Group Meta Data*

Description

`group_exec` sorts data by groups prior to iterating through them. When running `group_exec` multiple times on the same data, it is better to pre-sort the data and tell `group_exec` as much so it does not sort the data again. We can do the latter with `process_groups`, which additionally computes group information we can re-use across calls.

Usage

```
process_groups(groups, sorted = FALSE)
```

Arguments

`groups`	an integer, numeric, or factor vector. Alternatively, a list of equal-length such vectors, the interaction of which defines individual groups to organize the vectors in `data` into (multiple vectors not implemented yet). Numeric vectors are coerced to integer, thus copied. Vectors of integer type, but with different classes/attributes (other than factors) will be treated as integer vectors. The vectors must be the same length as those in `data`. NA values are considered one group. If a list, the result of the calculation will be returned as a "data.frame", otherwise as a named vector. Currently only one group vector is allowed, even when using list mode. Support for multiple group vectors and other types of vectors will be added in the future. Zero length groups are not computed on at all (e.g. missing factor levels, zero-length group vector).
`sorted`	TRUE or FALSE (default), whether the vectors in `groups` are already sorted. If set to TRUE, no sorting will be done on the groups, nor later on the `data` by `group_exec`. If the data is truly sorted this produces the same results while avoiding the cost of sorting. If the data is not sorted by groups, g will produce groups corresponding to equal-value runs it contains, which might be useful in some circumstances.

Value

an "r2c.groups" object, which is a list containing group sizes, labels, and group count, along with other meta data such as the group ordering vector.

Note

The structure and content of the return value may change in the future.

See Also

`group_exec`

Examples

```
## Use same group data for different but same length data.
## (alternatively, could use two functions on same data).
n <- 10
dat <- data.frame(x=runif(n), y=runif(n), g=sample(1:3, n, replace=TRUE))

## Pre-sort by group and compute grouping data
dat <- dat[order(dat[['g']]),]
g.r2c <- process_groups(dat[['g']], sorted=TRUE) # note sorted=TRUE

## Re-use pre-computed group data
f <- r2cq(sum(x))
with(dat, group_exec(f, x, groups=g.r2c))
with(dat, group_exec(f, y, groups=g.r2c))

## Claim unsorted data is sorted to implement RLE
g <- c(1, 2, 2, 1, 1, 1, 2, 2)
group_exec(f, rep(1, length(g)), process_groups(g, sorted=TRUE))
rle(g)$values
rle(g)$lengths
```

---

r2c-compile — *Compile Eligible R Calls Into Native Instructions*

Description

The `r2c*` compilation functions translate supported R function calls into C, compile them into native instructions using `R CMD SHLIB`, and return an interface to that code in the form of an "r2c_fun" function. This function will carry out out numerical calculations with `r2c` native instructions instead of with the standard R routines, with the exception of some iteration-constant calls. "r2c_fun" functions are intended to be run with the `r2c` runners for fast iterated calculations. Look at the examples here and those of the runners to get started.

Usage

```
r2cf(
  x,
  dir = NULL,
  check = getOption("r2c.check.result", FALSE),
  quiet = getOption("r2c.quiet", TRUE),
  clean = is.null(dir),
  optimize = getOption("r2c.optimize", TRUE),
  envir = environment(x)
)

r2cl(
  x,
  formals = NULL,
  dir = NULL,
  check = getOption("r2c.check.result", FALSE),
  quiet = getOption("r2c.quiet", TRUE),
  clean = is.null(dir),
  optimize = getOption("r2c.optimize", TRUE),
  envir = parent.frame()
)

r2cq(
  x,
  formals = NULL,
  dir = NULL,
  check = getOption("r2c.check.result", FALSE),
  quiet = getOption("r2c.quiet", TRUE),
  clean = is.null(dir),
  optimize = getOption("r2c.optimize", TRUE),
  envir = parent.frame()
)
```

Arguments

`x`	an object to compile into an "r2c_fun", for `r2cf` an R function, for `r2cq` an expression that will be captured unevaluated, for `r2cl` an R expression escaped with `quote`. See details.
`dir`	NULL (default), or character(1L) name of a file system directory to store the shared object file in. If NULL a temporary directory will be used. The shared object will also be loaded, and if `dir` is NULL the directory with the file will be removed after loading. Currently the capability to re-use generated shared objects across R sessions is not formally supported, but can likely be arranged for by preserving the directory.
`check`	TRUE or FALSE (default), if TRUE will evaluate the R expression with the input data and compare that result to the one obtained from the `r2c` C code evaluation, marking the result with attributes that indicate that the result was identical, and if not, also with an attribute with the result of an `all.equal` comparison. The check is only carried out when an `r2c` function is invoked directly (see example).
`quiet`	whether to suppress the compilation output.
`clean`	TRUE or FALSE, whether to remove the `dir` folder containing the generated C code and the shared object file after the shared object is `dyn.load`ed. Normally this is an auto-generated temporary folder. This will only delete folders that have the same directory root as one generated by `tempfile()` to avoid accidents. If you manually provide `dir` you will need to manually delete the directory yourself.
`optimize`	TRUE (default) or FALSE whether to enable "compiler" optimizations. Currently it is just the automatic re-use of repeated computation results. You can use `get_r_code` to see if optimizations were applied.
`envir`	environment to use as the enclosure of the function evaluation environment. It defaults to the environment from which the compilation function is called, or for `r2cf` the environment of `fun`. See details.

`formals`	character vector of the names of the parameters for the resulting "r2c_fun", a list of formals as generated with e.g. `alist`, or NULL (default). NULL causes all free symbols in `x` to become parameters to the result "r2c_fun" in the order they appear in `x`'s call tree (see details). Non-default values can be used to specify different parameter order, and in the list form also to specify default values for parameters. Symbols in `x` not in `formals` will be resolved against the evaluation environment at run time.
`TRUE`	FALSE, or an integer setting optimization levels. Currently applies `reuse_calls` if not FALSE or 0.

Value

an "r2c_fun" function; this is an unusual function so please see details.

r2c Generated Functions

While "r2c_fun" functions can be called in the same way as normal R functions, there is limited value in doing so. "r2c_fun" functions are optimized to be invoked invoked indirectly with runners. In many common cases it is likely that using an "r2c_fun" directly instead of with a runner will be slower than evaluating the corresponding R expression.

The lifecycle of an `r2c` function has two stages.

1. Compilation, with `r2cq` or similar.
2. Execution, either direct or via runners, which comprises:
    - A one time memory allocation sized to largest iteration (this memory is re-used for every iteration).
    - Iterative execution over groups/windows.

Each of the `r2c*` functions addresses different types of input:

- `r2cf` generates an "r2c_fun" function from a regular R function.
- `r2cq` captures an unquoted R expression and turns it into an "r2c_fun" function (e.g. `r2cq(a + b)`).
- `r2cl` turns quoted R language (e.g. as generated by `quote`) into an "r2c_fun" function (e.g. `r2cl(quote(a + b))`).

For `r2cl` and `r2cq`, free symbols used as parameters to `call` and its constituent sub-calls (e.g. the `x` and `y` in `sum(x) + y`) will become parameters to the output "r2c_fun" function. There must be at least one such symbol in `call`. Parameter order follows that of appearance in the call tree after everything is `match.call`ed. Symbols beginning with `.R2C` are reserved for use by `r2c` and thus disallowed in `call`. You may also directly set the parameter list with the `formals` parameter, or with `r2cf`.

As with regular R functions, unbound symbols are resolved in the lexical environment of the function. You can set a different environment on creation of the function with the `envir` parameter, but currently there is no way to change it afterwards (`environment(r2c_fun) <- x` will likely just break the function).

Details

r2c will preprocess the provided call either to apply optimizations (see optimize parameter), or because a call needs to be modified to work correctly with r2c. The processing leaves call semantics unchanged. If r2c modified a call, get_r_code will show a "processed" member with the modified call.

r2c requires a C99 or later compatible implementation with floating point infinity defined and the R_xlen_t range representable without precision loss as double precision floating point. Platforms that support R and fail this requirement are likely rare.

Interrupts are supported at the runner level, e.g. *between* groups or windows, each time a preset number of elements has been processed since the last interrupt check. There is infrastructure to support within iteration-interrupts, but it adds overhead when dealing with many iterations with few elements each and thus is disabled at the moment.

Users should not rely on specifics of the internal structure of "r2c_fun" functions; these are subject to change without notice in future r2c releases. The only supported uses of "r2c_fun" functions are use with the runners, standard invocation with the (operator, and other r2c functions that accept "r2c_fun" functions as arguments.

See Also

runners to iterate "r2c_fun" functions on varying data, inspection functions to retrieve meta data from the function including the generated C code and the compiler output, preprocessing for how r2c modifies R calls before translation to C, package overview for other r2c concepts.

Examples

```
r2c_mean_area <- r2cq(mean(x * y))
## Not run:
## Equivalently with `r2cl` or `r2cf`:
r2c_mean_area <- r2cl(quote(mean(x * y)))
mean_area <- function(x, y) mean(x * y)
r2c_mean_area <- r2cf(mean_area)

## End(Not run)
## Intended use is with runners
with(
  iris,
  group_exec(r2c_mean_area, list(Sepal.Width, Sepal.Length), Species)
)
## Standard invocation supported but, it is of limited value.
## We'll use standard invocation in the examples for clarity.
r2c_mean_area(iris[['Sepal.Width']], iris[['Sepal.Length']])

## Set parameter order for r2cq
r2c_sum_sub2 <- r2cq(sum(x - y), formals=c('y', 'x'))
r2c_sum_sub2(-1, c(1, 2, 3))

## Multi-line statements with assignments are supported (but
## `r2c` automatically optimizes re-used calls, so intermediate
## assignments may be unnecessary (see `?reuse_calls`):
```

```
slope <- function(x, y) {
  mux <- mean(x)
  x_mux <- x - mux
  sum(x_mux * (y - mean(y))) / sum(x_mux^2)
}
r2c_slope <- r2cf(slope)
u <- runif(10)
v <- 3/4*u + 1/4*runif(10)
r2c_slope(u, v)
```

---

r2c-inspect *Extract Data from "r2c_fun" Objects*

Description

"r2c_fun" functions contain embedded data used by the runners to call the compiled native code associated with the functions. The functions documented here extract various aspects of this data.

Usage

```
get_c_code(fun, all = TRUE)

get_r_code(fun, raw = FALSE)

get_so_loc(fun)

get_compile_out(fun)

show_c_code(fun, all = FALSE)
```

Arguments

`fun`	an "r2c_fun" function as produced by the compilation functions.
`all`	TRUE or FALSE (default) whether to retrieve all of the C code, or just the portion directly corresponding to the translated R expression.
`raw`	TRUE or FALSE (default) whether to display the processed R code exactly as `r2c` will use it, or to simplify it to make easier to read. If a simplification occurred the processed member name will be "processed*".

Details

- `get_so_loc` the file system location of the shared object file which can be used to identify the corresponding loaded dynamic library (see details).
- `get_c_code` the generated C code used to produce the shared object, but for quick inspection `show_c_code` is best.
- `show_c_code` retrieves code with `get_c_code` and outputs to screen the portion corresponding to the compiled expression, or optionally all of it.

- `get_r_code` the R call that was translated into the C code; if processing modified the original call the processed version will also be shown (see `r2cq`).
- `get_compile_out` the "stdout" produced during the compilation of the shared object.

Most calls seen in the raw version of what `get_r_code` returns will have a C level counterpart labeled with the R call in a comment. This includes calls that are nested as arguments to other calls, which will appear before the outer call. Due to how how control structures are implemented the R calls and the C level counterparts will not match up exactly.

When `r2c` creates a dynamic library, by default it immediately deletes the file system object after loading into memory. Thus, `get_so_loc` is only useful to help identify the loaded version of the library. See `r2c-compile` for how to preserve the file system version of loaded libraries.

Value

For `get_r_code` a list with on or two members, the first "original" is the R language object provided to the compilation functions, the second "processed" (or "processed*", see `raw` parameter) is the version that the C code is based on. For all other functions a character vector, invisibly for `show_c_code`.

See Also

`r2c-compile`, `r2c-preprocess`, `loaded_r2c_dynlibs`.

Examples

```
r2c_sum_sub <- r2cq(sum(x + y))
get_r_code(r2c_sum_sub)
show_c_code(r2c_sum_sub)
```

`reuse_calls` *Identify Repeated Calls and Reuse First Instance*

Description

Complex statistics often re-use a simpler statistic multiple times, providing the opportunity to optimize them by storing the result of the simple statistic instead of recomputing it. This function detects reuses of sub-calls and modifies the call tree to implement the store-and-reuse optimization.

Usage

```
reuse_calls(x)
```

Arguments

`x`	a call

Details

While this function is intended primarily for use as a pre-processing optimization for `r2c`, it can be applied to R expressions generally with some important caveats. `r2c` should work correctly with all `r2c` compatible R expressions, but is not guaranteed to do so with all R expressions. In particular, any expressions with side effects are likely to cause problems. Simple assignments like `<-` or `=` are safe in cases where there is no ambiguity about the order in which they would be made (e.g. `{a <- b; b <- a}` is safe, but `fun(a <- b, b <- a)` might not be). `r2c` allows only the unambiguous cases for the "r2c_fun" functions, but that is enforced by the compilation functions, not by `reuse_calls`. Nested scopes will also trick `reuse_calls` (e.g. use of `local`, in-lined functions).

Sub-calls are compared by equality of their deparsed forms after symbols are disambiguated to account for potential re-assignment. `reuse_calls` is conservative about branches, assuming that variables might be set to different values by different branches. This might void substitutions that at run-time would have turned out to be valid, and even some that could have been known to be valid with more sophisticated static analysis (e.g. `if(TRUE) ... else ...` will be treated as a branch even though it is not really). Substitutions involving loop modified variables are also limited due to the possibility of their value changing during e.g. the first and nth loop iteration, or the possibility of the loop not running at all.

`reuse_calls` relies on functions being bound to their original symbols, so do not expect it to work correctly if e.g. you rebind `<-` to some other function. `r2c` checks this in its own use, but if you use `reuse_calls` directly you are responsible for this.

Value

a list with elements:

- x: the call with the re-used expressions substituted
- reuse: named list with names the variables that reference the expressions that are substituted, and values those expressions.

Examples

```
x <- runif(100)
y <- runif(100)
slope <- quote(((x - mean(x)) * (y - mean(y))) / (x - mean(x))^2)
(slope.r <- reuse_calls(slope))
identical(eval(slope), eval(slope.r))

intercept <- quote(
  mean(y) - mean(x) * ((x - mean(x)) * (y - mean(y))) / (x - mean(x))^2
)
(intercept.r <- reuse_calls(intercept))
identical(eval(intercept), eval(intercept.r))
```

`rollby_exec` *Compute on Sequential Windows on Data*

Description

A runner that calls the native code associated with `fun` on sequential windows along `data` vector(s) with "elements" positioned on the real line. Data element positions can be specified and irregular, so equal sized windows may contain different number of elements. Window positions may be specified independent of data element positions. Each `roll*_exec` function provides a different mechanism for defining the space covered by each window. All of them will compute `fun` for each iteration with the set of data "elements" that fall within that window.

- `rollby_exec`: equal width windows spaced by apart.
- `rollat_exec`: equal width windows at specific positions given in `at`.
- `rollbw_exec`: windows with ends defined explicitly in `left` and `right`.

Additionally, `rolli_exec` is available for variable integer-width windows spaced by apart, but `data` elements are rank-positioned only.

Usage

```
rollby_exec(
  fun,
  data,
  width,
  by,
  offset = 0,
  position = seq(1, length(first_vec(data)), 1),
  start = position[1L],
  end = position[length(position)],
  bounds = "[)",
  MoreArgs = list()
)

rollat_exec(
  fun,
  data,
  width,
  at = position,
  offset = 0,
  position = seq(1, length(first_vec(data)), 1),
  bounds = "[)",
  MoreArgs = list()
)

rollbw_exec(
  fun,
  data,
  left,
  right,
  position = seq(1, length(first_vec(data)), 1),
  bounds = "[)",
  MoreArgs = list()
)
```

Arguments

`fun`	an "r2c_fun" function as produced by the compilation functions.
`data`	a numeric vector, or a list of equal length numeric vectors. If a named list, the vectors will be matched to `fun` parameters by those names. Elements without names are matched positionally. If a list must contain at least one vector. Conceptually, this parameter is used similarly to `envir` parameter to `base::eval` when that is a list.
`width`	scalar positive numeric giving the width of the window interval. Unlike with `rolli_exec`'s `n`, `width` must be scalar.
`by`	strictly positive, finite, non-NA scalar numeric, interpreted as the stride to increment the anchor by after each `fun` application.
`offset`	finite, non-na, scalar numeric representing the offset of the window from its "anchor". Defaults to 0, which means the left end of the window is aligned with the anchor (i.e. conceptually equivalent to `align="left"` for `rolli_exec`). Use `-width/2` for center aligned, and `-width` for right aligned. See "Intervals". Note this default is different to that for `rolli_exec`.
`position`	finite, non-NA, monotonically increasing numeric vector with as many elements as `data`. Each element in `position` is the position on the real line of the corresponding `data` element (see notes). Integer vectors are coerced to numeric and thus copied.
`start`	non-na, finite scalar numeric position on real line of first "anchor". Windows may extend to the left of `start` (or to the right of `end`) based on `offset`, and will include all data elements inside the window, even if they are outside `[start,end]`.
`end`	non-na, finite scalar numeric position on real line of last "anchor", see `start`.
`bounds`	scalar character to determine whether elements positions on a window boundary are included or excluded from the window: • "[)": include elements on left boundary, exclude those on right (default). • "(]": include elements on right boundary, exclude those on left. • "[]": include elements on either boundary. • "()": exclude elements on either boundary. [)": include elements on left boundary, exclude those on right (default). • "(]: R:)%22:%20include%20elements%20on%20left%20boundary,%20exclude%20those%20on%2
`MoreArgs`	a list of R objects to pass on as iteration-constant arguments to `fun`. Unlike with `data`, each of the objects therein are passed in full to the native code for each iteration This is useful for arguments that are intended to remain constant across

	iterations. Matching of these objects to `fun` parameters is the same as for `data`, with positional matching occurring after the elements in `data` are matched.
`at`	non-NA, finite, monotonically increasing numeric vector of anchor positions on the real line for each window (see notes). Integer vectors are coerced to numeric and thus copied.
`left`	non-NA, finite, monotonically increasing numeric positions of the left end of each window on the real line (see notes). Integer vectors are coerced to numeric and thus copied.
`right`	non-NA, finite, monotonically increasing numeric positions of the left end of each window on the real line, where `right >= left` (see notes). Integer vectors are coerced to numeric and thus copied.

Value

A numeric vector of length:

- `(end - start) %/% by + 1` for `rollby_exec`.
- `length(at)` for `rollat_exec`.
- `length(left)` for `rollbw_exec`.

Data Elements

`data` is made up of "elements", where an "element" is a vector element if `data` is an atomic vector, or a "row" if it is a "data.frame" / list of equal-length atomic vectors. Elements of `data` are arrayed on the real line by `position`. The default is for each element to be located at its integer rank, i.e. the first element is at 1, the second at 2, and so on. Rank position is the sole and implicit option for `rolli_exec`, which will be more efficient for that case, slightly so for `by = 1`, and more so for larger values of `by`.

Windows

Windows are intervals on the real line aligned (adjustably) relative to an "anchor" point given by `at` for `rollat_exec`, or derived from `start` and `by` for `rollby_exec`. `rollbw_exec` defines the ends of each window explicitly via `left` and `right`. Interval bounds are closed on the left and open on the right by default.

As an illustration for `rollby_exec` and `rollat_exec`, consider the case of `width = 3` windows at the fourth iteration, with various `offset` values. The offset is the distance from the left end of the window to the anchor. We use the letters a through g to reference the first seven elements of a numeric vector.

```
## rollby_exec(..., by=1, width=3)
                  +------------- 4th iteration, anchor is 4.0
                  V
1.0   2.0   3.0   4.0   5.0   6.0   7.0 | < Real Line
 a     b     c     d     e     f     g  | < Elements
                  |
                  |                       Offset     In-window Elements
                  [-----------------)  |     0      {d, e, f}
```

```
         [-----------------)           |  -w/2      {c, d, e}
[-----------------)                    |    -w      {a, b, c}
```

In each case we get three elements in the window, although this is only because the positions of the elements are on the integers by default. Since the windows are open on the right, elements that align exactly on the right end of the window are excluded. With irregularly spaced elements, e.g. with `position = c(1, 1.25, 2.5, 5.3, 7, ...)`, we might see (positions approximate):

```
## rollby_exec(..., by=1, width=3, position=c(1, 1.25, 2.5, 5.3, 7))
                   +------------- 4th iteration, base index is 4.0
                   V
1.0   2.0   3.0   4.0   5.0   6.0   7.0 | < Real Line
 a b      c        |       d         e  | < Elements
                   |
                   |                      Offset     In-window
                   [-----------------)  |     0      {d}
         [-----------------)            |  -w/2      {c, d}
 [-----------------)                    |    -w      {a, b, c}
```

Unlike with `rolli_exec` there is no `partial` parameter as there is no expectation of a fixed number of elements in any given window.

A restriction is that both ends of a window must increase monotonically relative to their counterparts in the prior window. This restriction might be relaxed for `rollbw_exec` in the future, likely at the cost of performance.

Equivalence

The `roll*_exec` functions can be ordered by increasing generality:

`rolli_exec` < `rollby_exec` < `rollat_exec` < `rollbw_exec`

Each of the functions can replicate the semantics of any of the less general functions, but with increased generality come efficiency decreases (see "Performance"). One exception is that `rolli_exec` supports fully variable width windows. `rollbw_exec` supports variable width windows, with the constraint that window bounds must monotonically increase with each iteration.

`rolli_exec` has semantics similar to the simple use case for `zoo::rollapply`, `data.table::froll*`, `RcppRoll::roll*`, and `slider::slide_<fun>`. `rollat_exec(..., position=x, at=x)` has semantics similar to `slider::slide_index`, but is more flexible because `at` need not be `position`.

Performance

In general `{r2c}` should perform better than most alternate window functions for "arbitrary" statistics (i.e. those that can be composed from `{r2c}` supported functions). Some packages implement algorithms that will outperform `{r2c}` on wide windows for a small set of simple predefined statistics. For example, for rolling means `{data.table}` and `{roll}` offer the "on-line" algorithm, and `{slider}` the "segment tree" algorithm, each with different performance and precision trade-offs.

Recall that the less general the `roll*_` function is, the better performance it will have (see "Equivalence"). The differences are slight between the by/at/bw implementations, and also for `rolli_exec` if `by << n`. If `by >> n`, `rolli_exec` can be much faster.

See README for more details.

Note

For the purposes of this documentation, the first value in a set or the lowest value in a range are considered to be the "leftmost" values. We think of vectors as starting on the "left" and ending on the "right", and of the real line as having negative infinity to the "left" of positive infinity.

Position vectors are expected to be monotonically increasing and devoid of NA and non-finite values. Additionally it is expected that `right` >= `left`. It is the user's responsibility to ensure these expectations are met. Window bounds are compared to element positions sequentially using by LT, LTE, GT, GTE relational operators in C, the exact set of which depending on `bounds`. If any of the position vectors are out of order, or contain NAs, or non-finite values, some, or all windows may not contain the elements they should. Further, if there are any NAs the result may depend on the C implementation used to compile this package. Future versions may check for and disallow disordered, NA, and/or non-finite values in the position vectors.

See Also

Compilation for more details on the behavior and constraints of "r2c_fun" functions, `first_vec` to retrieve first atomic vector, package overview for other `r2c` concepts.

Other runners: `group_exec()`, `rolli_exec()`

Examples

```
## Simulate transactions occurring ~4 days
old.opt <- options(digits=3)
set.seed(1)
count <- 150
frequency <- 1/(3600 * 24 * 4)
time <- as.POSIXct('2022-01-01') - rev(cumsum(rexp(count, frequency)))
revenue <- runif(count) * 100
data.frame(time, revenue)[c(1:3,NA,seq(count-2, count)),]

r2c_mean <- r2cq(mean(x))

## Mean trailing quarter revenue, computed/reported "monthly"
month <- 3600 * 24 * 30  # more or less
by30 <- rollby_exec(
  r2c_mean, revenue, position=time, width=3 * month, by=month,
  start=as.POSIXct('2021-01-01'),
  offset=-3 * month   # trailing three months
)
by30

## Same, but explicit times via `at`; notice these are not exactly monthly
timeby30 <- seq(as.POSIXct('2021-01-01'), to=max(time), by=month)
timeby30[1:10]
at30 <- rollat_exec(
  r2c_mean, revenue, position=time, width=3 * month,
  at=timeby30, offset=-3 * month
)
at30
identical(by30, at30)
```

```
## Use exact monthly times with `at`
timebymo <- seq(as.POSIXct('2021-01-01'), to=max(time), by="+1 month")
timebymo[1:10]
atmo <- rollat_exec(
  r2c_mean, revenue, position=time, width=3 * month,
  at=timebymo, offset=-3 * month
)
(rev.90 <- data.frame(time=timebymo, prev.90=atmo))[1:5,]

## Exact intervals with `rollexec_bw`.
months <- seq(as.POSIXct('2020-10-01'), to=max(time), by="+1 month")
left <- head(months, -3)
right <- tail(months, -3)
bwmo <- rollbw_exec(r2c_mean, revenue, position=time, left=left, right=right)
(rev.qtr <- data.frame(time=right, prev.qtr=bwmo))[1:5,]

## These are not exactly the same because -90 days is not always 3 months
atmo - bwmo
## Confirm bwmo is what we think it is (recall, right bound open)
identical(bwmo[1], mean(revenue[time >= '2020-10-01' & time < '2021-01-01']))

## Compare current month to trail quarter
months2 <- seq(
  as.POSIXct('2021-01-01'), length.out=nrow(rev.qtr) + 1, by="+1 month"
)
left <- months2[-length(months2)]
right <- months2[-1]
thismo <- rollbw_exec(r2c_mean, revenue, position=time, left=left, right=right)
transform(
  rev.qtr,
  this.month=thismo,
  pct.change=round((thismo - prev.qtr)/prev.qtr * 100, 1)
)
options(old.opt)
```

---

rolli_exec *Compute on Sequential Regular Windows on Equidistant Data*

Description

A runner that calls the native code associated with `fun` on sequential regularly spaced windows along the `data` vector(s). Each window is aligned relative to a specific data "element" (anchor), and the set of window size `n` contiguous elements around and including the "anchor" are computed on. This is a special case of `rollby_exec` intended to mimic the semantics of `zoo::rollapply` where `width` is a scalar integer, and implicitly the data elements are equally spaced.

Usage

```
rolli_exec(
    fun,
    data,
    n,
    by = 1L,
    align = "center",
    partial = FALSE,
    MoreArgs = list()
  )
```

Arguments

`fun`	an "r2c_fun" function as produced by the compilation functions.
`data`	a numeric vector, or a list of equal length numeric vectors. If a named list, the vectors will be matched to `fun` parameters by those names. Elements without names are matched positionally. If a list must contain at least one vector. Conceptually, this parameter is used similarly to `envir` parameter to `base::eval` when that is a list.
`n`	integer number of adjacent data "elements" to compute `fun` on. It is called `n` and not `width` to emphasize it is a discrete count instead of an interval width as in `rollby_exec` and friends. Must be scalar, or have as many elements as data (see "Data Elements"). For the latter, specifies the element counts of each window. Coerced to integer if numeric, and thus copied.
`by`	strictly positive scalar integer interpreted as the stride to increment the "anchor" after each `fun` application. Coerced to integer if numeric.
`align`	scalar character one of "center" (default), "left", or "right", indicating what part of the window should align to the base index. Alternatively, a scalar integer where `0` is equivalent to "left", `1 - n` equivalent to "right", and `(1 - n) %/% 2` is equivalent to "center" (i.e. represents the offset of the window relative to its anchor).
`partial`	TRUE or FALSE (default), whether to allow computation on partial windows that extent out of either end of the data.
`MoreArgs`	a list of R objects to pass on as iteration-constant arguments to `fun`. Unlike with `data`, each of the objects therein are passed in full to the native code for each iteration This is useful for arguments that are intended to remain constant across iterations. Matching of these objects to `fun` parameters is the same as for `data`, with positional matching occurring after the elements in `data` are matched.

Value

a numeric vector of length `length(first_vec(data)) %/% by`.

Window Alignment

`align` specifies which end of the window aligns with the anchor. Here we illustrate on the fourth iteration of a call to `rolli_exec`:

```
## rolli_exec(..., data=1:7,  n=4)
```

```
      +--------- On the 4th iteration, anchor is 4
      v
1 2 3 4 5 6 7    | seq_along(first_vec(data))
      |
      |            Align     In-Window Elements
      * * * *    | "left"    {4, 5, 6, 7}
    * * * *      | "center"  {3, 4, 5, 6}  <- default
* * * *          | "right"   {1, 2, 3, 4}
```

For the case of "center" with even sized windows more elements will be to the right than to the left of the anchor.

Correspondence to rollby_exec

rolli_exec is a slightly more efficient implementation of:

```
function(fun, data, n, align, ...)
  roll_by_exec(
    fun, data,
    width=n - 1,
    offset=((match(align, c('left', 'center', 'right')) - 1) / 2) * (1 - n)
    bounds="[]",
    ...
   )
```

Window element counts correspond to an interval width as n - 1, e.g.:

```
1  2  3    |  n = 3
[     ]    |  width = 3 - 1 = 2 = n - 1
```

Unlike rolli_exec, rollby_exec only supports fixed width windows.

The align values correspond to numeric values as follows: "left" to 0, "center" to -width/2, and "right" to -width. The default window alignment is equivalent to "left" for rollby_exec, which is different than for this function.

[ ]: R:%20%20%20%20%20 rollby_exec: R:%60rollby_exec%60 rollby_exec: R:%60rollby_exec%60

Data Elements

data is made up of "elements", where an "element" is a vector element if data is an atomic vector, or a "row" if it is a "data.frame" / list of equal-length atomic vectors. Elements of data are arrayed on the real line by position. The default is for each element to be located at its integer rank, i.e. the first element is at 1, the second at 2, and so on. Rank position is the sole and implicit option for rolli_exec, which will be more efficient for that case, slightly so for by = 1, and more so for larger values of by.

See Also

Compilation for more details on the behavior and constraints of "r2c_fun" functions, package overview for other r2c concepts.

Other runners: group_exec(), rollby_exec()

Examples

```
r2c_mean <- r2cq(mean(x))
with(
  mtcars,
  rolli_exec(r2c_mean, hp, n=5)
)

## Effect of align and partial
r2c_len <- r2cq(length(x))
dat <- runif(5)
rolli_exec(r2c_len, dat, n=5, align='left', partial=TRUE)
rolli_exec(r2c_len, dat, n=5, align='center', partial=TRUE)
rolli_exec(r2c_len, dat, n=5, align='right', partial=TRUE)
rolli_exec(r2c_mean, dat, n=5, align='left')

## Variable length windows
rolli_exec(r2c_len, dat, n=c(1,3,1,3,1), align='left', partial=TRUE)
```

`square` *Raise a Vector to the Power of Two*

Description

Implemented as `x * x` to match what R does with `x ^ 2`.

Usage

```
square(x)
```

Arguments

`x`	a numeric vector

Value

`x`, squared

Index