On The Design, Implementation, and Use of Laziness in R

Northeastern University

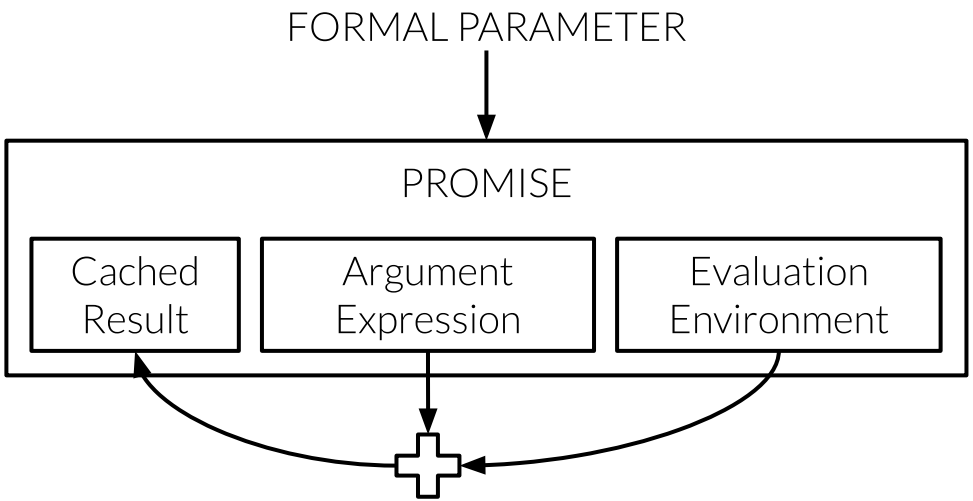
CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

Aviral Goel

Jan Vitek

We call an implementation of normal-order reduction in which recomputation is avoided ***lazy evaluation*** (or ***call by need***)
... arguments in function calls are evaluated at most once.
— *Paul Hudak*

Functional, Imperative & OO
Dynamically Typed
First-Class Environments
Reflection
Side-Effects
Lazy




```
mat <- matrix(c(1,1,1,1), nrow=2)
isSymmetric(mat, tol=100*epsilon)
```

These two lines generate
278 Promises!

Hypothesis: R could be strict ***by default***

Validation: Observational study of ***R usage***

Quantitative data: **dynamic analysis**

Qualitative data: **code inspection**

The Ecosystem

CRAN

Curated repository of general purpose packages

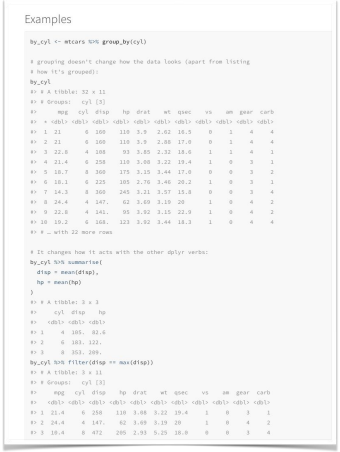
15,102 Packages

Bioconductor

Curated repository of high-throughput genomic data analysis packages

1,741 Packages

Over 25 years of use and 2 million users

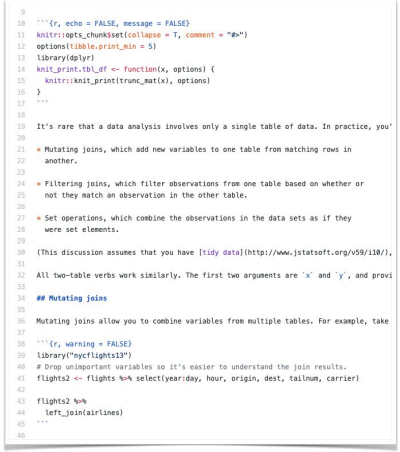


```
Examples

by_cyl <- mtcars %>% group_by(cyl)

# grouping doesn't change how the data looks (apart from listing
# how it's grouped):
by_cyl
#> # A tibble: 32 x 11
#> # Groups:   cyl [3]
#>    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  21     6   160   110  3.9  2.62  16.5     0     1     4     4
#>  2  21     6   160   110  3.9  2.88  17.0     0     1     4     4
#>  3  22.8    4   108    93  3.85  2.32  18.6     1     1     4     1
#>  4  21.4    6   258   110  3.08  3.22  19.4     1     0     3     1
#>  5  18.7    8   360   175  3.15  3.44  17.0     0     0     3     2
#>  6  18.1    6   225   105  2.76  3.46  20.2     1     0     3     1
#>  7  14.3    8   360   245  3.21  3.57  15.8     0     0     3     4
#>  8  24.4    4   147.   62  3.69  3.19  20       1     0     4     2
#>  9  22.8    4   141.   95  3.92  3.15  22.9     1     0     4     2
#> 10  19.2    6   168.  123  3.92  3.44  18.3     1     0     4     4
#> # … with 22 more rows

# It changes how it acts with the other dplyr verbs:
by_cyl %>% summarise(
  disp = mean(disp),
  hp = mean(hp)
)
#> # A tibble: 3 x 3
#>   cyl  disp    hp
#>   <dbl> <dbl> <dbl>
#> 1     4  105.  82.6
#> 2     6  183.  122.
#> 3     8  353.  209.
by_cyl %>% filter(disp == max(disp))
#> # A tibble: 3 x 11
#> # Groups:   cyl [3]
#>    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  21.4    6   258   110  3.08  3.22  19.4     1     0     3     1
#> 2  24.4    4   147.   62  3.69  3.19  20       1     0     4     2
#> 3  10.4    8   472   205  2.93  5.25  18.0     0     0     3     4
```

Examples

```
context("Group by")

df <- data.frame(x = rep(1:3, each = 10), y = rep(1:6, each = 5))

test_that("group_by with add = TRUE adds groups", {
  add_groups1 <- function(tbl) group_by(tbl, x, y, add = TRUE)
  add_groups2 <- function(tbl) group_by(group_by(tbl, x, add = TRUE), y, add = TRUE)

  expect_groups(add_groups1(df), c("x", "y"))
  expect_groups(add_groups2(df), c("x", "y"))
})

test_that("group_by_ backwards compatibility with add = TRUE adds groups", {
  scoped_lifecycle_silence()
  add_groups_extendedclass <- function(tbl) {
    grouped <- group_by(tbl, x)
    group_by.default(grouped, y, add = TRUE)
  }

  expect_groups(add_groups_extendedclass(df), c("x", "y"))
})

test_that("joins preserve grouping", {
  g <- group_by(df, x)

  expect_groups(inner_join(g, g, by = c("x", "y")), "x")
  expect_groups(left_join(g, g, by = c("x", "y")), "x")
  expect_groups(semi_join(g, g, by = c("x", "y")), "x")
  expect_groups(anti_join(g, g, by = c("x", "y")), "x")
})

test_that("constructors drops groups", {
  df <- data.frame(x = 1:3) %>% group_by(x)
  expect_no_groups(tbl_df(df))
})

test_that("grouping by constant adds column (#410)", {
  grouped <- group_by(mtcars, "cyl") %>% summarise(foo = n())
  expect_equal(names(grouped), c("""cyl"", "foo"))
  expect_equal(nrow(grouped), 1L)
})
```

Tests



```
 9
10 ```{r, echo = FALSE, message = FALSE}
11 knitr::opts_chunk$set(collapse = T, comment = "#>")
12 options(tibble.print_min = 5)
13 library(dplyr)
14 knit_print.tbl_df <- function(x, options) {
15   knitr::knit_print(trunc_mat(x), options)
16 }
17 ```
18
19 It's rare that a data analysis involves only a single table of data. In practice, you'
20
21 * Mutating joins, which add new variables to one table from matching rows in
22   another.
23
24 * Filtering joins, which filter observations from one table based on whether or
25   not they match an observation in the other table.
26
27 * Set operations, which combine the observations in the data sets as if they
28   were set elements.
29
30 (This discussion assumes that you have [tidy data](http://www.jstatsoft.org/v59/i10/),
31
32 All two-table verbs work similarly. The first two arguments are `x` and `y`, and provi
33
34 ## Mutating joins
35
36 Mutating joins allow you to combine variables from multiple tables. For example, take
37
38 ```{r, warning = FALSE}
39 library("nycflights13")
40 # Drop unimportant variables so it's easier to understand the join results.
41 flights2 <- flights %>% select(year:day, hour, origin, dest, tailnum, carrier)
42
43 flights2 %>%
44   left_join(airlines)
45 ```
46
```

Vignettes

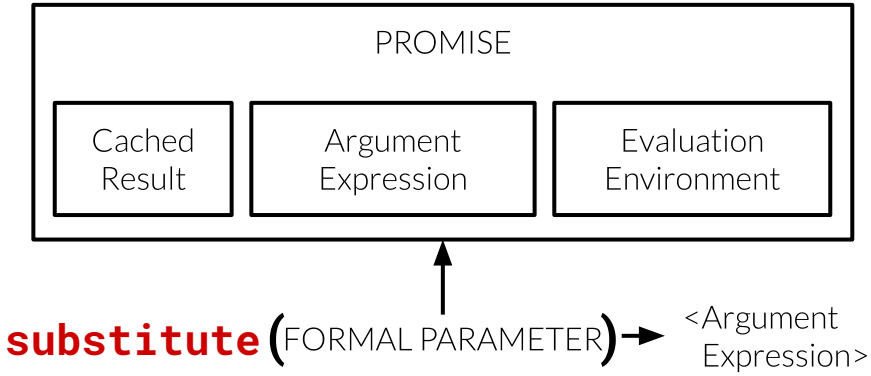
Avg. 60% code coverage

F. Krikava, J.Vitek. *Tests from Traces: Automated Unit Test Extraction for R.* ISSTA'18

270.9 B

Metaprogramming with Promises

Reflectively accessing argument AST from promises



2B (0.7%) Promises Metaprogrammed in ~2K R Packages.

Non-Standard Evaluation

```
local <- function(x) {
    eval.parent(substitute(eval(x, new.env())))
}
```

Syntactic Convenience

```
`::` <- function(pk,nm) {
    get(package = as.character(substitute(pk)),
        name    = as.character(substitute(nm)))
}
```

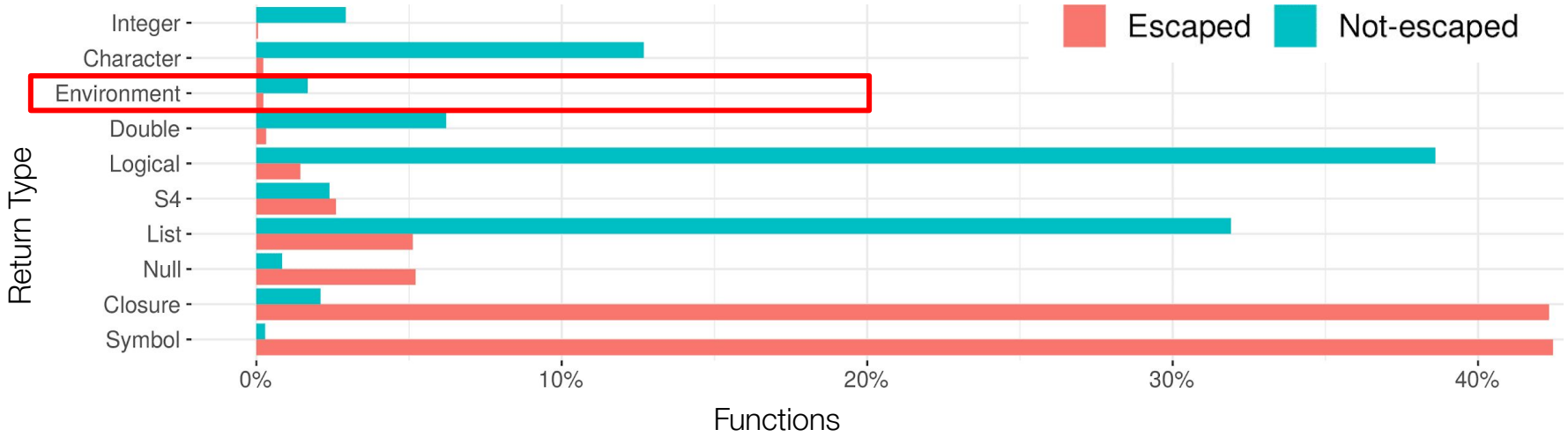
Lazy Data Structures

Are they used?

```
nil   <- list2env(list(tag="nil"))
head  <- function(l) l$h
empty<- function(l) l$tag=="nil"

cons <- function(h,t) environment()
```

Promise captured in returned environment escape and outlive creating call

Only 11M out of 270B Promises Escape



Laziness and Side-Effects

Interaction of laziness, side effects and higher-order functions confuses users

```
f <- function(x) {
         function() { x }
     }
y <- 2
g <- f(y)
y <- 4
g()  ## 4
y <- 6
g()  ## 4
```

*On **numerous** occasions programmers require strictness to control side effects*

```
function(a=1, b=0, e=1, d=1) {
    force_all(a,b,e,d)
    function(n) {
        viridis(n,a,b,e,d)
    }
}
```

1.3B calls to `forceAndCall`

101M calls to `force` in 60% of packages

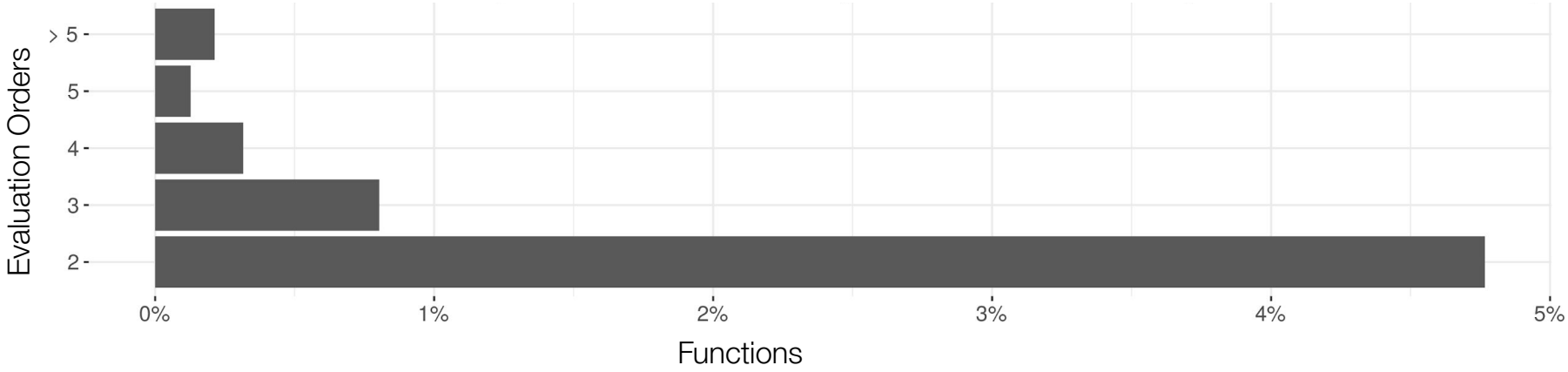
Argument
Evaluation Orders

Multiple argument orders and side effects

```
f <- function(x,A,B) {
    if (x) { A; B }
    else   { B; A }
}
```

94% functions have a **single evaluation order**

16.5M promises out of 270.9B perform side-effects



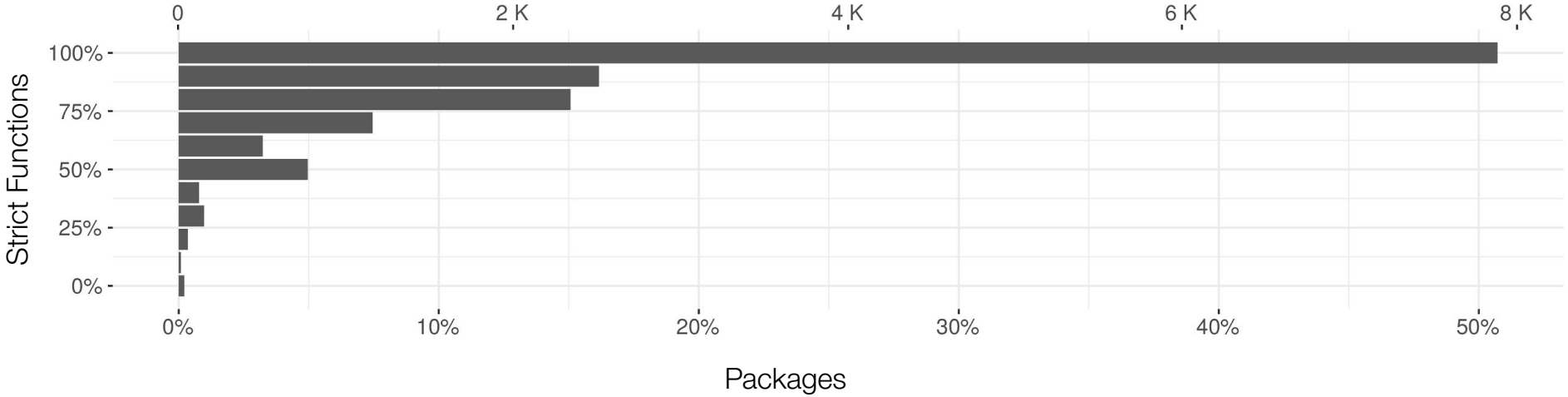
Function **Strictness**

Functions evaluating all arguments in a single order

```
f <- function(x, @lazy y) {
      if(x) y
}
```

```
g <- function(@meta y) {
      substitute(…y…)
}
```

84% strict functions, 17% packages less than 75% strict



Conclusions

Laziness is used in R but does not have to be the default

It is needed for meta-programming and extensibility, but little else

The combination of laziness and side-effects complicates life

We are designing an annotation system for a non-lazy R

Northeastern University

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

Aviral Goel

Jan Vitek