

Development best practices

Hadley Wickham

Assistant Professor / Dobelman Family Junior Chair
Department of Statistics / Rice University

June 2012



Wednesday, June 27, 12

1. Correct code
2. Maintainable code
3. Fast code
4. Learning more

Correct code

Testing

- Will focus on systematic unit testing tomorrow afternoon
- Today we'll discuss debugging, and basic techniques for making your code more robust

Rules of thumb

- Use TRUE and FALSE, not T and F
- Avoid functions that have non-standard evaluation rules (no subset, with, transform)
- Avoid functions that can have different types of output (sapply, always use drop = FALSE)
- Be explicit about missings.

Check preconditions

Always best to fail early - as soon as you know something is wrong.

If your function expects certain types of input, it's a good idea to test that they are as expected. `stopifnot` is a quick and dirty way of doing so.

Your turn

Take the function on the next page and make it work more reliably, or at least give sensible error messages.

```
col_means <- function(df) {  
  numeric <- sapply(df, is.numeric)  
  numeric_cols <- df[, numeric]  
  
  data.frame(lapply(numeric_cols, mean))  
}
```

```
col_means(mtcars)  
col_means(mtcars[, 0])  
col_means(mtcars[0, ])  
col_means(mtcars[, "mpg", drop = F])  
col_means(1:10)  
col_means(as.matrix(mtcars))  
col_means(as.list(mtcars))
```

```
mtcars2 <- mtcars  
mtcars2[-1] <- lapply(mtcars2[-1], as.character)  
col_means(mtcars2)
```



```
col_means <- function(df) {  
  numeric <- vapply(df, is.numeric, logical(1))  
  numeric_cols <- df[, numeric, drop = FALSE]  
  
  data.frame(lapply(numeric_cols, mean))  
}
```

No peeking until you've
made an attempt!

My solution:

```
col_means <- function(df) {  
  # stopifnot(is.data.frame(df))  
  df <- as.data.frame(df)  
  
  numeric <- vapply(df, is.numeric, logical(1))  
  numeric_cols <- df[, numeric, drop = FALSE]  
  
  data.frame(lapply(numeric_cols, mean))  
}
```

Debugging

- `traceback()` tells you where the problem is
- `browser()` starts an interactive debugger where it's called
- `options(error = recover)` starts interactive debugger automatically on error
- `options(warn = 2)` turns warnings into errors so you can find them more easily

Trace

- Allows you to insert code into any function
- `debug()` automatically inserts `browser()`, `debugonce()` automatically removes it after it's called once.

```
# Also some tools for post-mortem debugging of  
# non-interactive scripts
```

```
options(error =  
  quote({dump.frames(to.file = TRUE); q()}))
```

```
# Saves debugging info to file last.dump.rda
```

```
# Then in an interactive R session:
```

```
print(load("last.dump.rda"))  
debugger("last.dump")
```

```
# Just like if you'd used recover()
```

Maintainable code

Tips

- Code gets faster as computers get faster. It never gets correct by itself, and it never gets more elegant.
- Pick a style guide and stick with it.
<https://github.com/hadley/devtools/wiki/Style>
- Use source code control
(more on that tomorrow)

More tips

- Rewrite important code - your first attempt will not usually be the best approach.
- Use comments to explain why, not what or how.

Fast code

Figure out what's slow.

Speed it up.

What's slow?

RProf

Every interval seconds, writes the call stack out to a file on disk.

```
library(ggplot2)
```

```
Rprof("5-profile-ggplot2.txt")
```

```
qplot(carat, price, data = diamonds)
```

```
Rprof(NULL)
```

- "ggplot.data.frame" "ggplot" "qplot"
- "<Anonymous>" "set_last_plot" "+.ggplot" "+" "qplot"
- "plot_clone" "+.ggplot" "+" "<Anonymous>" ".Call"
"mapply" "qplot"
- "plot_clone" "+.ggplot" "+" "<Anonymous>" ".Call"
"mapply" "qplot"
- "plot_clone" "+.ggplot" "+" "<Anonymous>" ".Call"
"mapply" "qplot"
- "unlist" "as.vector" "simplify2array" "mapply" "qplot"
- "<Anonymous>" "set_last_plot" "print.ggplot" "print"
- "c" "do.call" "transform.data.frame" "transform"
"facet_map_layout.null" "facet_map_layout" "FUN" "lapply"
"map_layout" "ggplot_build" "print.ggplot" "print"
- "data.frame" "do.call" "transform.data.frame" "transform"
"facet_map_layout.null" "facet_map_layout" "FUN" "lapply"
"map_layout" "ggplot_build" "print.ggplot" "print"

Summarising

SummaryRProf summarises in a format that I don't find very helpful. I wrote the profr package to do better.

```
library(profr)
```

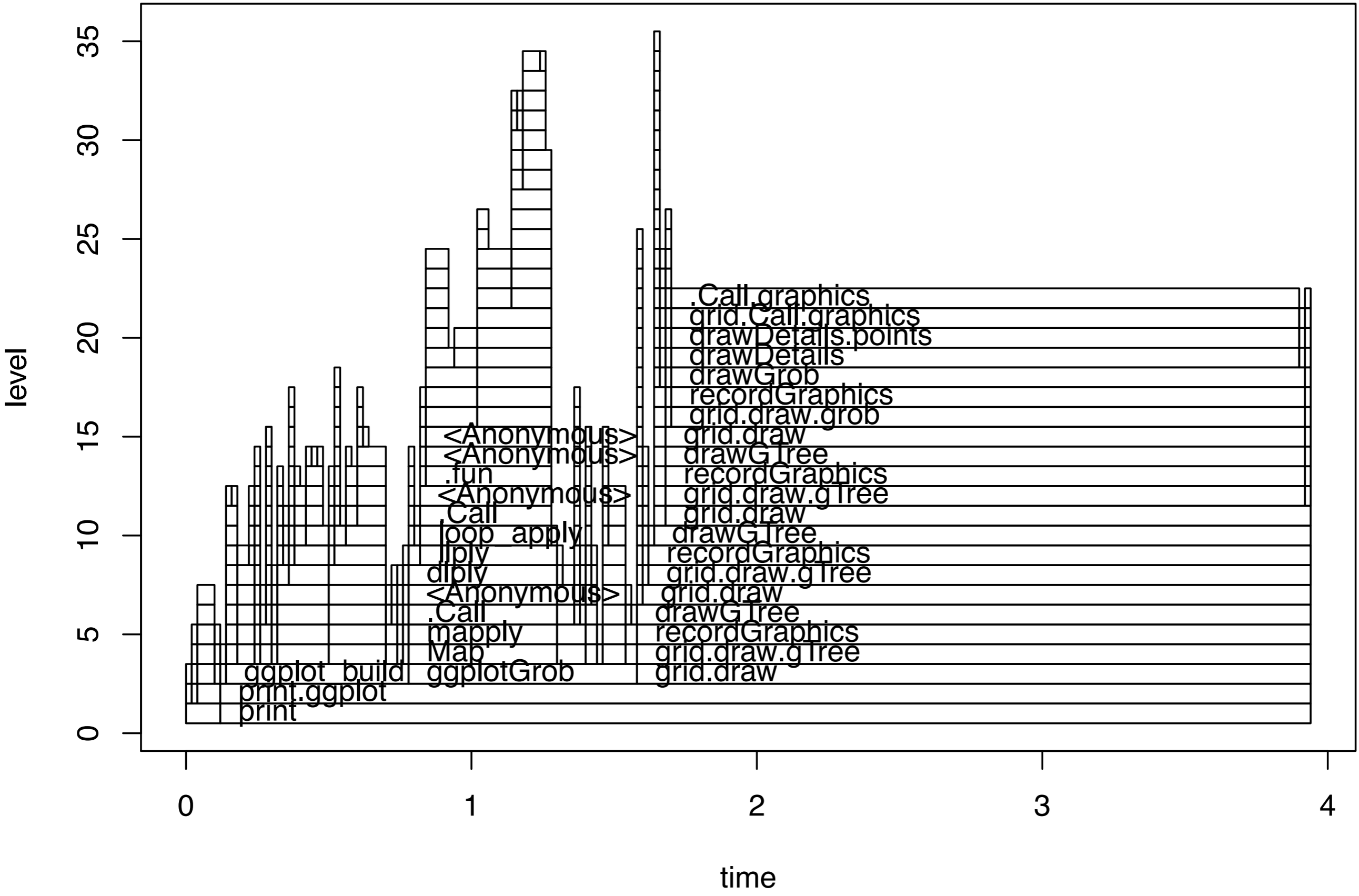
```
p <- parse_rprof("5-profile-ggplot2.txt")
```

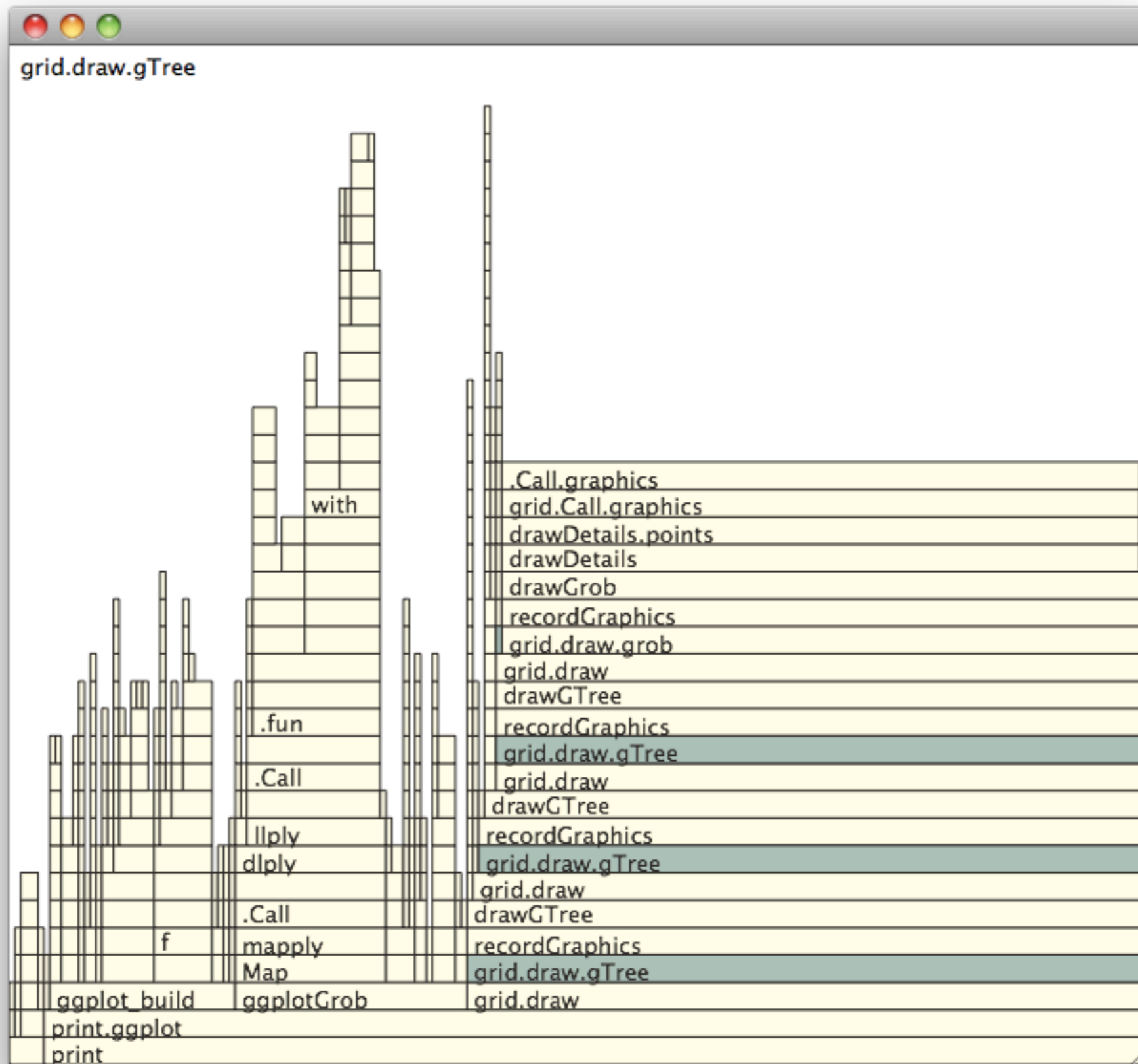
```
# OR
```

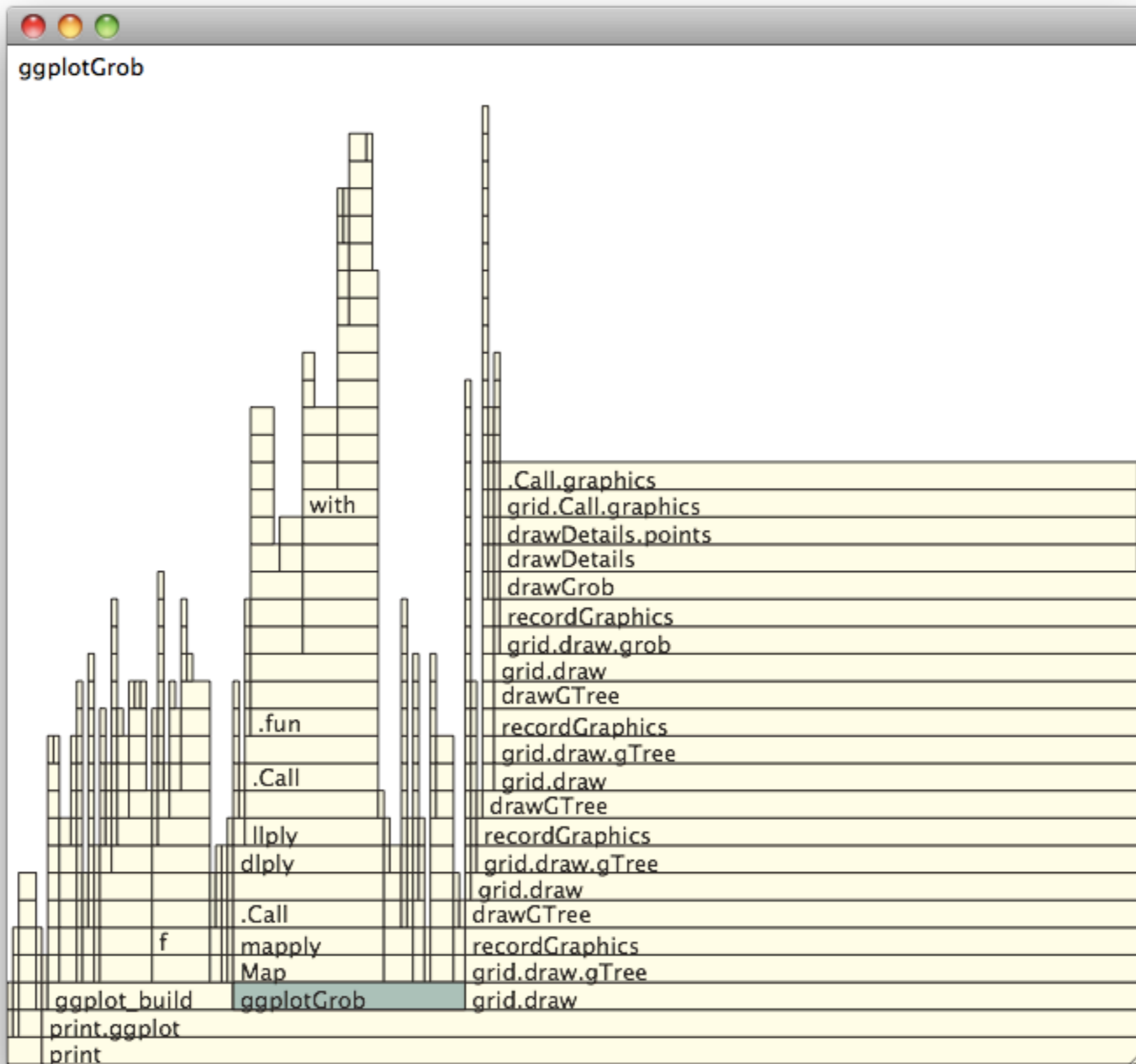
```
p <- profr(print(qplot(carat, price,  
  data = diamonds)))
```

head(p, 20)

	level	f	start	end	time	source
1	1	qplot	0.00	0.12	0.12	<NA>
2	1	print	0.12	3.94	3.82	base
3	2	ggplot	0.00	0.02	0.02	<NA>
4	2	+	0.02	0.04	0.02	base
5	2	mapply	0.04	0.12	0.08	base
6	2	print.ggplot	0.12	3.94	3.82	<NA>
7	3	ggplot.data.frame	0.00	0.02	0.02	<NA>
8	3	+.ggplot	0.02	0.04	0.02	<NA>
9	3	.Call	0.04	0.10	0.06	<NA>
10	3	simplify2array	0.10	0.12	0.02	base
11	3	set_last_plot	0.12	0.14	0.02	<NA>
12	3	ggplot_build	0.14	0.78	0.64	<NA>
13	3	ggplotGrob	0.78	1.58	0.80	<NA>
14	3	grid.draw	1.58	3.94	2.36	<NA>
15	4	set_last_plot	0.02	0.04	0.02	<NA>
16	4	<Anonymous>	0.04	0.10	0.06	<NA>
17	4	as.vector	0.10	0.12	0.02	base
18	4	<Anonymous>	0.12	0.14	0.02	<NA>
19	4	map_layout	0.14	0.18	0.04	<NA>
20	4	dlapply	0.18	0.24	0.06	<NA>







Memory profiling

- `gcTorture(T) + RProf(memory.profiling = T)` - gives minimum memory usage
- `Rprofmem()` - gives maximum memory usages
- `tracemem(x)` - prints message whenever `x` is duplicated

How can you make it
faster?

Speeding up code

- Avoid common mistakes (see chapters 2-4 on Patrick Burn's "R inferno" for good advice)
- Vectorise (vocab)
- Re-think your approach
- Rewrite in C, Fortran or C++

```
# If you know how long your result will be,  
# preallocate the storage  
grow <- function() {  
  output <- c()  
  for(i in 1:10000) {  
    output <- c(output, i ^ 2)  
  }  
  output  
}  
  
preallocate <- function() {  
  
  output <- rep(NA, 10000)  
  for(i in 1:10000) {  
    output[i] <- i ^ 2  
  }  
  output  
}  
  
library(microbenchmark)  
b <- microbenchmark(grow(), preallocate(), times = 10)  
print(b, unit = "eps")
```

```
# But you should always vectorise (i.e.  
# push loops into pre-written C) where possible  
  
vectorise <- function() (1: 10000) ^ 2  
b <- microbenchmark(grow(), preallocate(),  
  vectorise(), times = 10)  
print(b, unit = "eps")  
  
# Key to this technique is building up a good  
# R vocabulary
```


Your turn

Compare the two methods for growing a vector on the next slide.

How do they work?

Do they return the same results?

Which is faster?

```
grow2 <- function() {
  set.seed(1000)

  output <- numeric()
  while(sample(1e5, 1) > 1) {
    output <- c(output, 1)
  }
  output
}
double <- function() {
  set.seed(1000)

  output <- rep(NA, 10)
  n <- 10
  i <- 0

  while(sample(1e5, 1) > 1) {
    i <- i + 1
    if (i > n) {
      output <- c(output, rep(NA, n))
      n <- 2 * n
    }
    output[i] <- 1
  }
  output[seq_len(i)]
}
```

```
system.time(g <- grow2())  
system.time(d <- double())  
all.equal(d, g)
```

```
df <- function() {  
  for(i in nrow(mtcars)) {  
    mtcars[i, "cyl"] <- mtcars[i, "cyl"] * 2  
  }  
}
```

```
mtcars  
}
```

```
vector <- function() {  
  var <- mtcars$cyl  
  
  for(i in nrow(mtcars)) {  
    var[i] <- var[i] * 2  
  }  
  mtcars$cyl <- var  
  mtcars  
}
```

```
b <- microbenchmark(df(), vector())  
print(b, unit = "eps")
```

```
df <- function() {  
  
  result <- matrix(nrow(mtcars), ncol(mtcars))  
  for(i in nrow(mtcars)) {  
    result[i, ] <- as.numeric(mtcars[i, ])  
  }  
  
  result  
}
```

Byte code compiler

Compiler package available since 2.13

10-20% speedups on most functions.

More dramatic on a few special cases.

Still a work in progress – will only get better with time.

Worth a shot!

```
library(compiler)
```

```
df2 <- cmpfun(df)
```

```
microbenchmark(df(), df2())
```

```
vector2 <- cmpfun(vector)
```

```
microbenchmark(vector(), vector2())
```

Caution

These are microbenchmarks, which test a very very small specific piece of code. You must have correctly identified what is slow before they can be useful.

**Learning
more**

Within R

Subscribe to R-devel.

Read the source, particularly of the code and packages that you use most commonly

Never be satisfied. Concentrated and reflective practice is key to mastery.

Invest time now to save time later.

Manuals

<http://cran.r-project.org/manuals.html>

R language definition

R internals

Build your vocab

<https://github.com/hadley/devtools/wiki/vocabulary>.

Read R help.

Read R release notes.

Read stackoverflow

<http://stackoverflow.com/tags/r>

Read the R Journal

Outside R

The structure and interpretation of computer programs by Harold Abelson and Gerald Jay Sussman. <http://mitpress.mit.edu/sicp/full-text/book/book.html>

Concepts, Techniques and Models of Computer Programming by Peter van Roy and Sef Haridi. <http://amzn.com/0262220695>

The pragmatic programmer, by Andrew Hunt and David Thomas. <http://amzn.com/020161622X>

Tomorrow

Please make sure you have winzip (or equivalent) installed if you're on windows

```
# This should work  
library(devtools)  
has_devel()
```

```
# if it doesn't make sure, you have
```

```
# * R 2.15
```

```
# * devtools 0.7
```

```
# * rtools from (windows only)
```

```
# http://cran.r-project.org/bin/windows/Rtools/
```

```
# Also check you have testthat and roxygen2 installed
```

```
library(testthat)
```

```
library(roxygen2)
```


This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.