Development best practices

Hadley Wickham

Assistant Professor / Dobelman Family Junior Chair

Department of Statistics / Rice University

August 2011

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

Correct code

Testing

- Focus on debugging, and basic techniques for making your code more robust
- Pointers to more info on testing

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 you 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)
```

No peeking until you've made an attempt!

My solution:

```
col_means <- function(df) {
  stopifnot(is.data.frame(df))
  if (nrow(df) == 0) return(df)

  numeric <- vapply(df, is.numeric, logical(1))
  numeric_cols <- df[, numeric, drop = FALSE]

  data.frame(lapply(numeric_cols, mean))
}
```

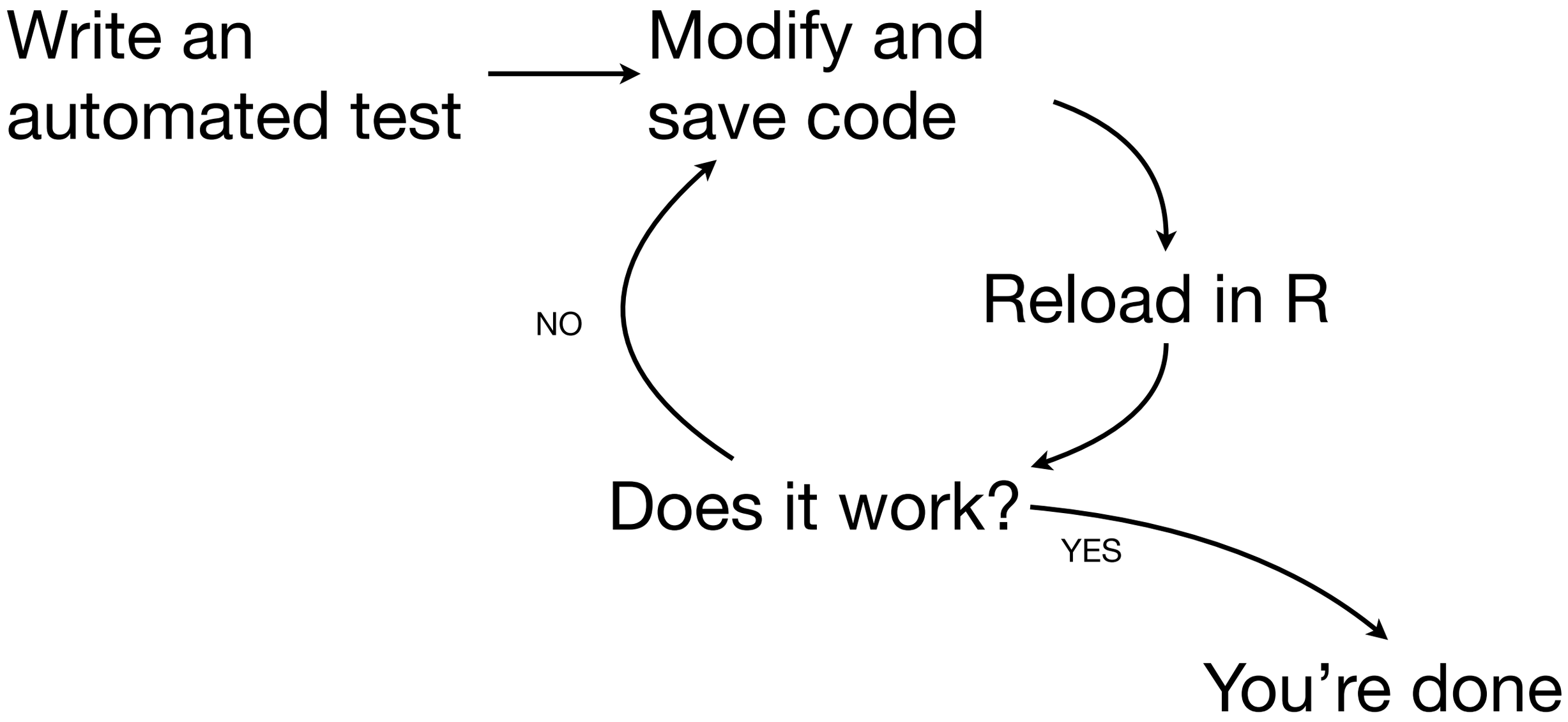
Debugging

- `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

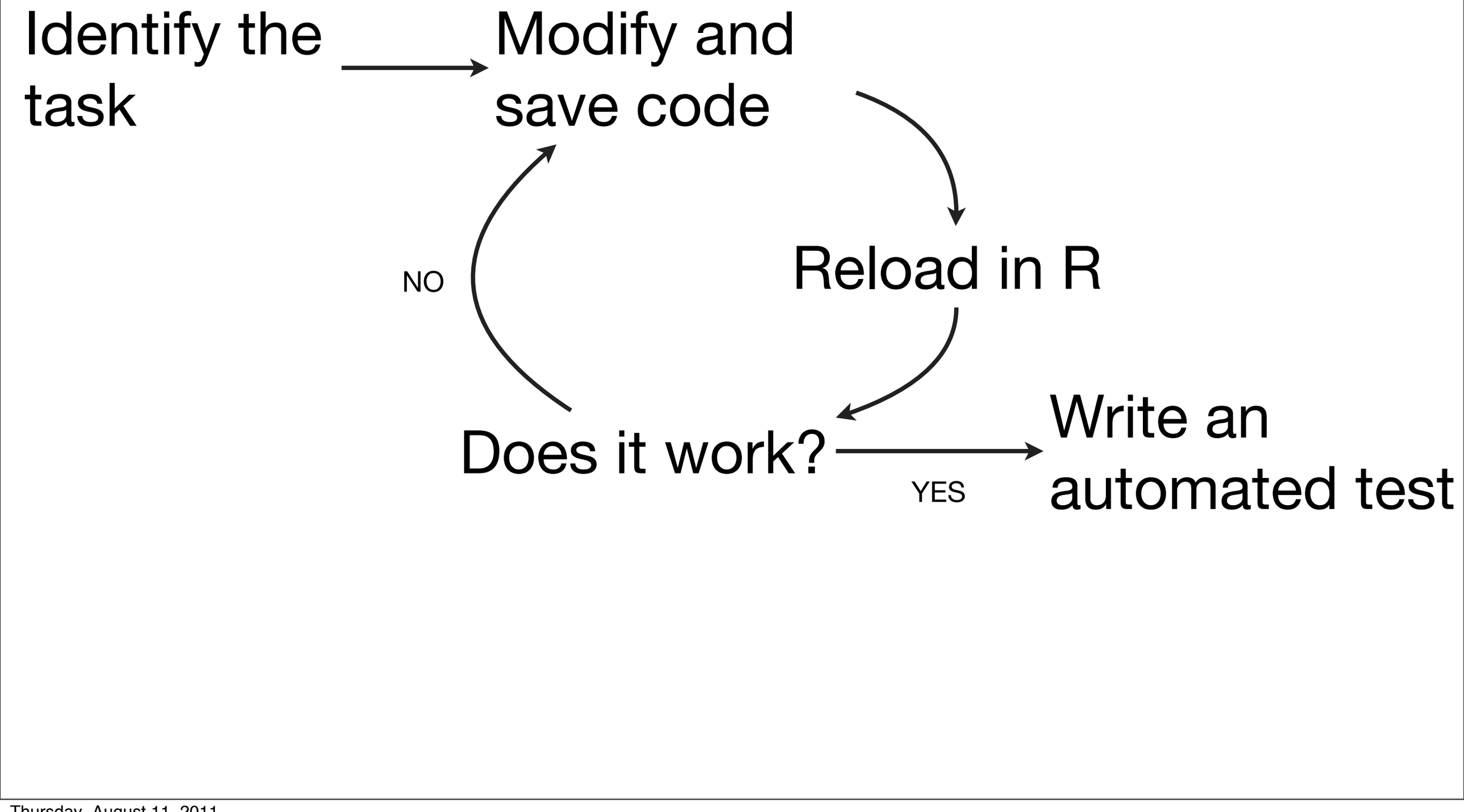
Trace

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

Confirmatory programming



Exploratory programming



Other benefits

- Code that can be tested easily, often has a better, more modular, design
- When you stop working, leave a test failing. You'll know what to work on when you come back
- Make big changes without fear of accidentally breaking anything

Testing packages

- RUnit
- svUnit
- **testthat**

Why test that?

- Easy transition from informal to formal tests. Can be used in wide variety of situations
- Wide range of expectations/assertions
- Fun, colourful output that keeps you motivated

https://github.com/hadley/devtools/wiki/Testing

http://bit.ly/testthat

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 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("4-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

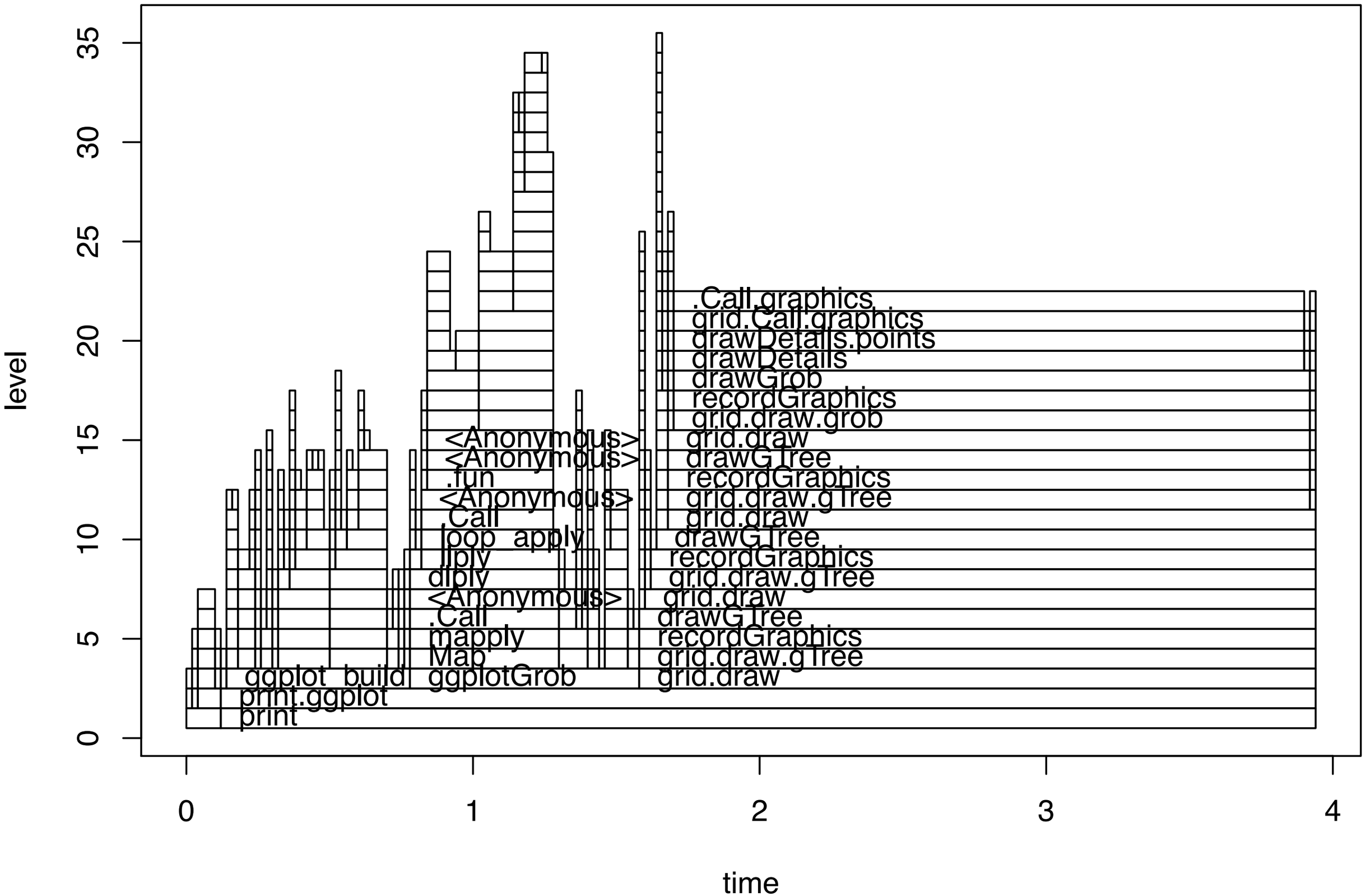
`SummaryRProf` summarises in a format that I don't find very helpful. I wrote the `profr` package to do better.

```
library(profr)
p <- parse_rprof("4-profile-ggplot2.txt")
# OR
p <- profr(print(qplot(carat, price,
  data = diamonds)))
```

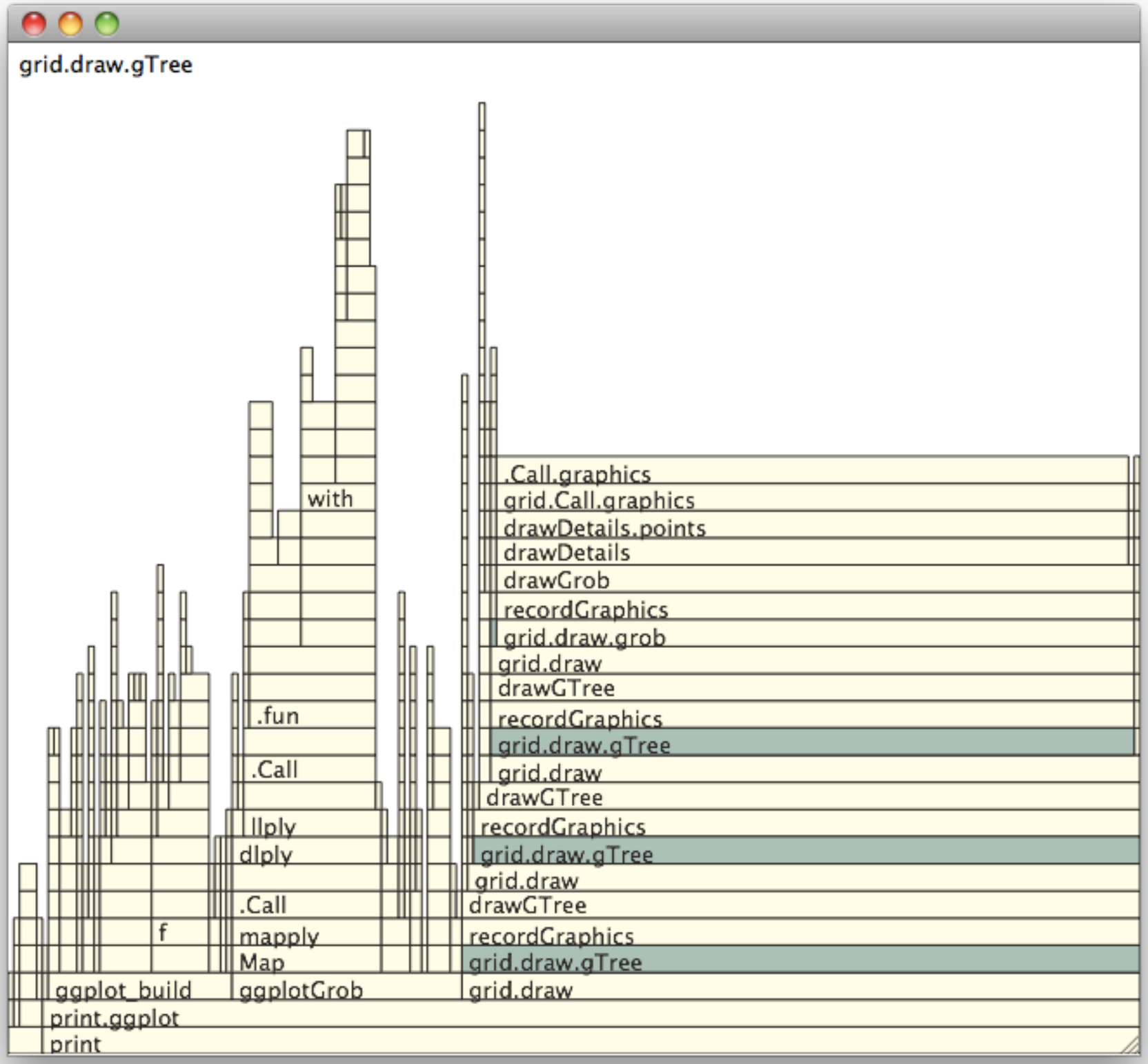
```
head(p, 20)
```

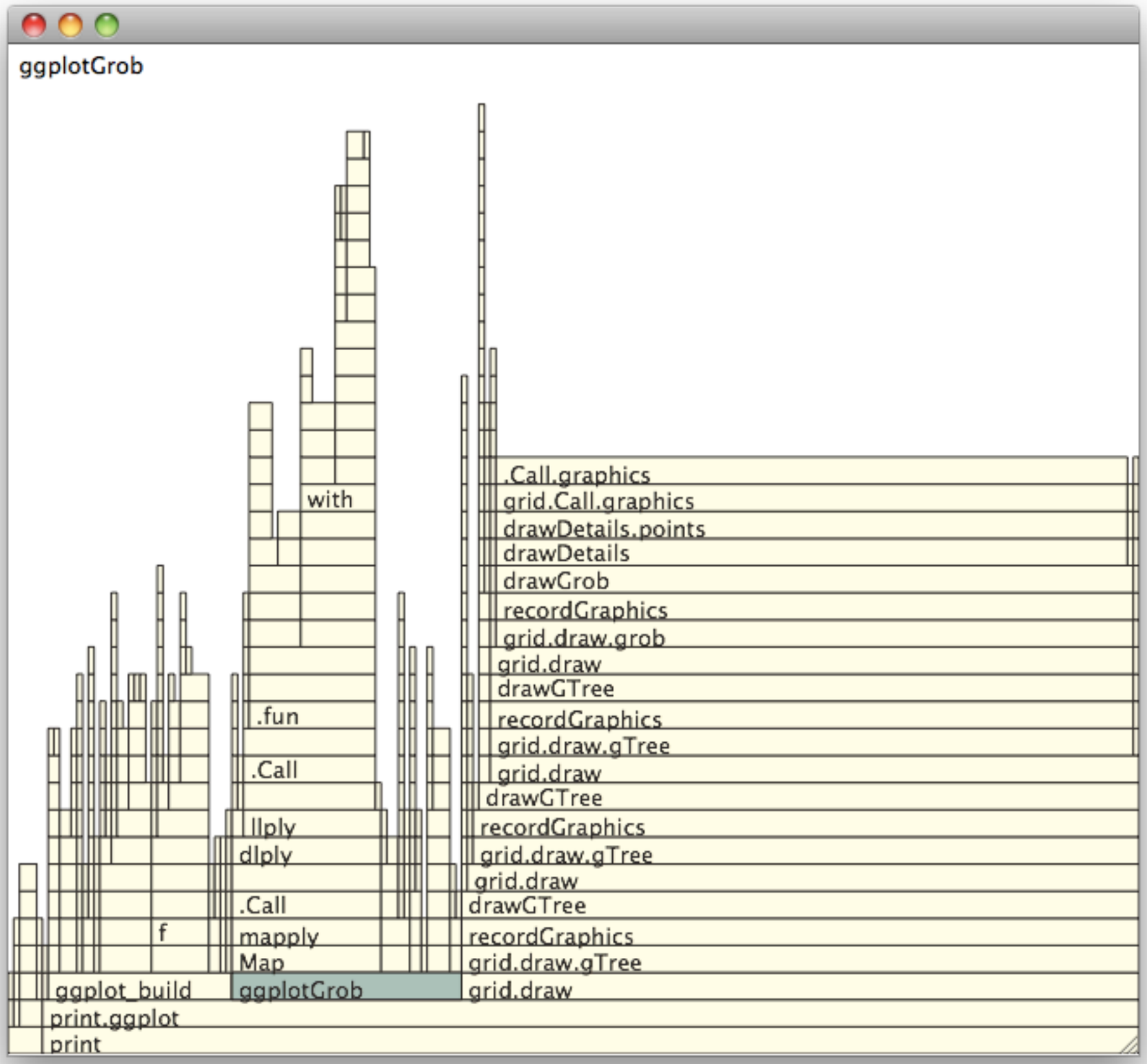
	level	f	start	end	time	source
1	1	qplot	0.00	0.12	0.12	<NA>
2	1	print	0.12	3.94	3.82	base
3	2	ggplot	0.00	0.02	0.02	<NA>
4	2	+	0.02	0.04	0.02	base
5	2	mapply	0.04	0.12	0.08	base
6	2	print.ggplot	0.12	3.94	3.82	<NA>
7	3	ggplot.data.frame	0.00	0.02	0.02	<NA>
8	3	+.ggplot	0.02	0.04	0.02	<NA>
9	3	.Call	0.04	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>

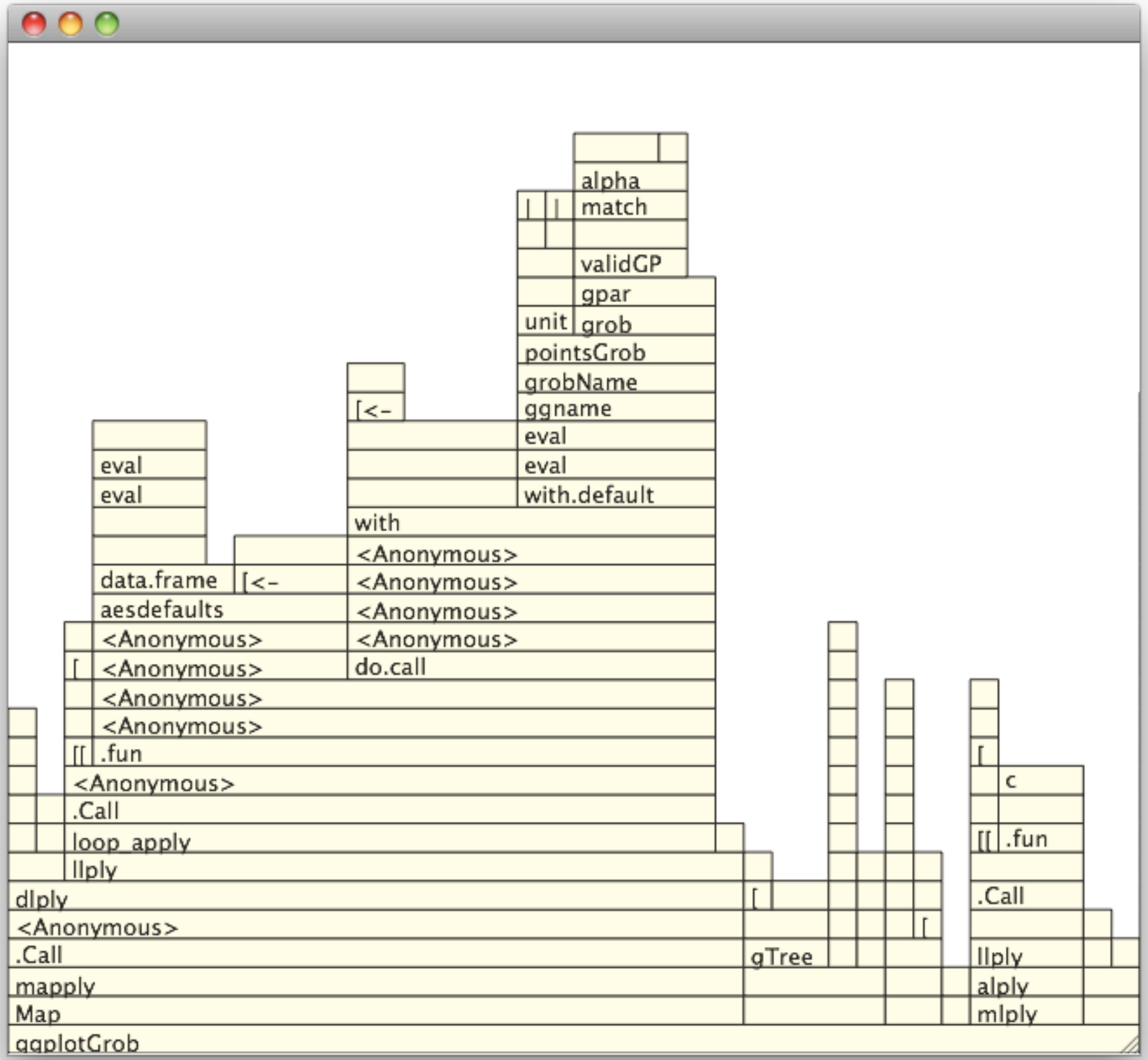
plot(p)


level
0
5
10
15
20
25
30
35
0
1
2
3
4
time
.Call.graphics
grid.Call.graphics
drawDetails.points
drawDetails
drawGrob
recordGraphics
grid.draw.grob
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
<Anonymous>
<Anonymous>
.fun
<Anonymous>
.Call
loop_apply
llply
dlply
<Anonymous>
.Call
mapply
Map
ggplotGrob
ggplot_build
print.ggplot
print

explore(p)


grid.draw.gTree
with
.fun
.Call
llply
dlply
.Call
f
mapply
Map
ggplot_build
ggplotGrob
print.ggplot
print
.Call.graphics
grid.Call.graphics
drawDetails.points
drawDetails
drawGrob
recordGraphics
grid.draw.grob
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw


ggplotGrob
with
.fun
.Call
llply
dlply
.Call
f
mapply
Map
ggplot_build
ggplotGrob
print.ggplot
print
.Call.graphics
grid.Call.graphics
drawDetails.points
drawDetails
drawGrob
recordGraphics
grid.draw.grob
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw
drawGTree
recordGraphics
grid.draw.gTree
grid.draw


alpha
match
validGP
gpar
unit
grob
pointsGrob
grobName
[<-
ggname
eval
eval
eval
eval
with.default
with
<Anonymous>
data.frame
[<-
<Anonymous>
aesdefaults
<Anonymous>
<Anonymous>
<Anonymous>
[
<Anonymous>
do.call
<Anonymous>
<Anonymous>
[[
.fun
<Anonymous>
.Call
loop_apply
llply
dlply
[
[
<Anonymous>
.Call
gTree
llply
mapply
alply
Map
mlply
ggplotGrob
c
[[
.fun
.Call

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
- Use the byte code compiler
- 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? (`system.time`)

```r
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")
```

```
vectorise <- function() {
  mtcars$cyl <- 2 * mtcars$cyl
  mtcars
}
apply2 <- function() {
  mtcars$cyl <- vapply(mtcars$cyl, function(x) x *
2, numeric(1))
  mtcars
}

b <- microbenchmark(df(), vector(), apply2(),
vectorise())
print(b, unit = "eps")
```

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.

Byte code compiler

- New core package by Luke Tierney
- “Compilation” for R code
- 2-4x speed-up for best case, 20% on average
- Next version, in 2.14, even better

Byte code compiler

```
library(compiler)

grow_c <- cmpfun(grow)
preallocate_c <- cmpfun(preallocate)

b <- microbenchmark(grow(), grow_c(), preallocate(),
preallocate_c(), times = 10)
print(b, unit = "eps")

grow2_c <- cmpfun(grow2)
double_c <- cmpfun(double)
system.time(g <- grow2_c())
system.time(d <- double_c())
```

Rcpp

- Package developed by Dirk Eddelbuettel and Romain Francois
- Makes it easy to connect C++ to R
- http://dirk.eddelbuettel.com/code/rcpp.html

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.

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