Profiling and Optimization

Günther Sawitzki
StatLab Heidelberg

gs@statlab.uni-heidelberg.de
http://www.statlab.uni-heidelberg.de/

16. Jan. 2012
Typeset: April 3, 2012

in preparation

/u/math/j40/cvsroot/lectures/src/insider/profile/beamer/main.tex,v 1.23 2012/01/26 19:48:05 j40 Exp

Computing Background

An Abstract Machine

Old times

In old times, computing time was computable

- register access: negligible
- integer: τ_{int}, approx. 1 unit
- reals: τ_{real}, approx. 10 units
- transcendental/non-polynomial: τ_{trans}, approx. 100 units

Total time

$$t = nr_{int} * \tau_{int} \times nr_{real} * \tau_{real} + nr_{trans} \times \tau_{trans}$$
$$\propto nr_{real} + 10 \times nr_{trans} \qquad \text{“complexity”}$$

An Abstract Machine

Old times . . . have passed.

Today, time “constants”

- vary by configuration (check yours!)
- do not differ by magnitude

Total time is not additive (operations may overlap)

Still “complexity” may be a useful notion, in particular if related to parameters of the computation.

Complexity

For your problem:

What is an appropriate measure for system size n?
What is the complexity, as a function of system size n?

Use a pragmatic definition, for your purposes only . . .

Optimisation has two possibilities:

- reduce complexity, e.g. by clever algorithms
- make efficient use of resources, e.g. by adapted implementation

The algorithm will be applied to data. Data management is a second field for optimisation.

An Abstract Machine

Old times

Memory access time was sizeable

- register access: negligible
- RAM measurable
- other storage: slow

Now:

Different media may have different access times.

Access may be dynamic (in a cloud).

Memory management may be an issue to take into account.

An Abstract Machine

Abstract...ignoring all implementation details

Computers just cannot understand.

For an abstract view, think of computing in terms of four states:

Source → intermediate code → evaluation → result

An Abstract Machine

Abstract...ignoring all implementation details

Source: the program code we enter

This may be a valid program or it may contain errors, it may be incomplete.

Intermediate code: a graph is a convenient representation

Each node represents an operation to be performed. The terminating nodes are special: their action is to fetch some data, resulting in a value. Each node can try to evaluate. If it succeeds, the result is again a value; if it does not succeed, we have an exception.

ToDo 1: Blackboard graphics: attributed variables and operators

An Abstract Machine

Abstract...ignoring all implementation details

What is missing in this picture is data. It was a major step towards programmable computing to understand that in principle data are not different from instructions.

The intended purpose makes the difference.

For our purpose, it may be helpful to think of data as a separate entity, at least available at the evaluation step.

Source	→ intermediate code	→ evaluation	→ result
	→ data structure	→ data	

An Abstract Machine

Pen and paper arithmetics

Note: operators are polymorphic even for pen and paper.
ToDo 2: Blackboard $2 + 3$ can be added by digit. $2.1 + 3.14$ needs adjustment for the exponent, i.e. alignment of the decimal.

R Specifics

An R function

```
area <- function(r)  pi * r ^2
area(1)
[1] 3.141593
```

If we enter the name, we get the definition

```
area
function(r)  pi * r ^2
> typeof(area)
[1] "closure"
> mode(area)
[1] "function"
```

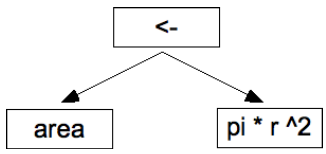
`closure` is short for function closure, the internal type for functions.
The `print()` function gives the form most appropriate for a closure.

R Specifics

```
area <- function(r)  pi * r ^2
```

We can look at the internal structure

```
> print(as.list(area))
$r

[[2]]
pi * r^2

> names(as.list(area))
[1] "r" ""

> typeof(as.list(area)[[2]])
[1] "language"
```

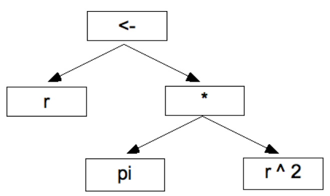


R Specifics

```
area <- function(r)  pi * r ^2
```

Internal structure

```
> as.list(as.list(area)[[2]])
[[1]]
`*`
[[2]]
pi
[[3]]
r^2
```

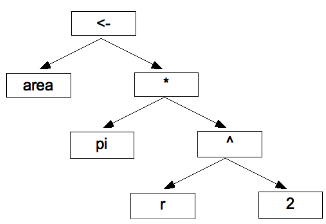


R Specifics

```
area <- function(r)  pi * r ^2
```

Internal structure

```
> as.list(as.list(area)[[2]][[3]])
[[1]]
`^`
[[2]]
r
[[3]]
[1] 2
```



R is using the call graph as intermediate code, stored in form of a variable of type `list`. Convention: first list element is the node name.

Computing Background

Intermediate Code

“Intermediate code” is the level to think of a program.

In general, the most appropriate representation of a pragram is a graph that has to be worked through. .

Code optimisation amounts to replacing the graph by an equivalent graph, with a more efficient execution.

R Specifics

Type Coercion

To evaluate a node, type conversion may be applied. Usually, the amout of effort can only be seen from the implementaion.

ToDo 3: Blackboard: Guarded operator evaluation at run time

R Specifics

Vector Expansion

R is vectorized. To evaluate a node, scalars may be extended to vectors, or vectors may be recycled.

ToDo 4: Operator control over vectorization

R Specifics

Storage

R follows FORTRAN conventions, that is, in arrays the first (highest) index varies first. Matrices are stored column-wise.

```
> matrix(1:6, nrow=3)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

If an algorithm follows this convention, it can be most efficient.

R Specifics

Storage

Check your algorithms whether they use row-first or column-first. For R, they better be column first.

R Specifics

R is interpreted	Advantage: you can work line by line. Downside: no look ahead optimisation
R is untyped	Advantage: flexible use of types. Downside: type has to be determined at evaluation time.
R is vectorised	Advantage: obvious. Downside: possibly unexpected vectorization or collapse to scalar.
	Note: R is *not* a matrix language.

R Specifics

R specifics

Be aware: there is overhead by implicit type conversion or vectorization.

Vectorize, where sensible.

But: there are pitfalls ahead.

R Specifics

Exercise

ToDo 5: Vectorization exercise

R Specifics

Search Paths and Environments

In any programming environment, at some points symbols must be resolved.

In R, a symbol has a name.

Names need not be unique (e.g. global names, names in variables).

Names have a scope where they are valid.

Scopes form a hierarchy, implemented in R as a chain of `environments`.

ToDo 6: lexical scope

R Specifics

Function Arguments and Environments

An R `environment` is a frame, a list of symbol/value references. Each environment has a reference to its enclosing environment.

The most common example is the frame of variables local to a function call; its enclosure is the environment where the function was defined.

The global environment `.GlobalEnv`, more often known as the user's workspace, is the first item on the search path.

R Specifics

Function Arguments and Environments

Environments are special objects in R. By default, arguments are passed to functions by value in R, that is a copy of generated. As an exception, environments are not copied. They are passed by reference. Wrapping information in an environment is an important possibility to optimize access to large data sets.

R Specifics

Function Arguments and Promises

R uses a lazy evaluation for arguments passed to functions.

When a function is called, the arguments are passed as parsed expression, together with a reference to the environment to be used for their evaluation.

Evaluation only takes place when the value of the argument is actually needed.

See *R Optimisation: Function Arguments*. Go to Function Arguments

Performance Measurement: Time

First clause of control theory

Without measurement, there is no control...

Performance Measurement: Time: `system.time()`

`system.time` *CPU Time Used*

Description

Return CPU (and other) times that `expr` used.

Usage

```
system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)
```

skip help

Performance Measurement: Time: `system.time()`

`system.time` *CPU Time Used*

Usage

```
system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)
```

Arguments

`expr`	Valid R expression to be timed.
`gcFirst`	Logical - should a garbage collection be performed immediately before the timing? Default is `TRUE`.

skip help

Performance Measurement: Time: `system.time()`

Details

`system.time` calls the function `proc.time`, evaluates `expr`, and then calls `proc.time` once more, returning the difference between the two `proc.time` calls.

`unix.time` is an alias of `system.time`, for compatibility with S.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When `gcFirst` is `TRUE` a garbage collection (gc) will be performed immediately before the evaluation of `expr`. This will usually produce more consistent timings.

Performance Measurement: Time

Timing Functions

`system.time(expr, gcFirst = TRUE)`	Elapsed time
`unix.time(expr, gcFirst = TRUE)`	Elapsed time
`proc.time()`	R process up time
`Sys.sleep(time)`	Suspend execution of R expressions for a given number of seconds.
`gc.time(on = TRUE)`	Reports the time spent in garbage collection so far in the R session while GC timing was enabled.

Performance Measurement: Time

`system.time` often is used to compare timings of a series of conditions.

Benchmark is a simple wrapper around `system.time` for this purpose.

```
library(rbenchmark)
help(benchmark)
```

Example:

```
> benchmark(1:10^4, log(1:10^4), sin(1:10^4),
columns=c('test', 'elapsed', 'replications', 'relative'))

        test elapsed replications relative
1      1:10^4   0.003          100  1.00000
2 log(1:10^4)   0.050          100 16.66667
3 sin(1:10^4)   0.081          100 27.00000
```


Performance Measurement: Time

Timer Resolution

The timing resolution is limited.

Working range on all nowadays system should be 10ms or better.

To find e.g. the number of iterations to be above the resolution threshold, use

```
library(itertools)
timeout(iterable, time)
```

For example: See how high we can count in a tenth of a second

```
length(as.list(timeout(icount(), 0.1)))
```


Performance Measurement: Time

Timing: Sequential Numbers

Simple example

```
> n<- 100
> system.time({vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)})
   user  system elapsed
  0.045   0.001   0.046
```

Simple example with pre-allocation

```
> system.time({vec <- numeric(n); for (i in 1:n) vec[i] <- i})
   user  system elapsed
  0.001   0.000   0.000
```

Pre-allocation reduces time.

Note: Time is recorded with limited precision that is machine dependent.
On all machines, you can expect timing resolution in the order of 1ms.

Performance Measurement: Time

Timing: Sequential Numbers

```
> n<- 1000
> system.time({vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)})
   user  system elapsed
  0.048   0.001   0.051
```

Simple example with pre-allocation

```
> system.time({vec <- numeric(n); for (i in 1:n) vec[i] <- i})
   user  system elapsed
  0.002   0.000   0.002
```

Vectorized

```
> system.time( vec <- 1:n)
   user  system elapsed
      0       0       0
```


Performance Measurement: Time

Timing: Sequential Numbers

Total elapsed times [s]

$n =$	[1]	[2]	[3]
100	0.046	0.001	0
1000	0.051	0.002	0
10000	0.298	0.018	0
100000	40.474	0.231	0.001
1000000	> 3659.686	2.090	0.001

```
[1]vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)
[2]vec <- numeric(n); for (i in 1:n) vec[i] <- i
[3]vec <- 1:n
```

Improvement by pre-allocation[2]. Drastic improvement by using vectorized version [3].

Performance Measurement: Time

Case Study: Sequential Numbers

`vec <- c(vec,i)`

Internal steps:

- copy argument `vec,i`
- check type. Adjust type if necessary
- calculate requested length `length(vec)+length(i)`
- allocate space of requested length
- copy `vec` to result space
- append `i` to result space
- return result as new value for `vec`

Note: the storage space previously allocated for `vec` is now free.

skip case study

Performance Measurement: Time

Case study: Sequential Numbers

`vec <- c(vec,i)`

Internal steps:

- copy argument `vec,i`
- check type. Adjust type if necessary
- calculate requested length `length(vec)+length(i)`
- allocate space of requested length
- copy `vec` to result space
- append `i` to result space
- return result as new value for `vec`

Note: the storage space previously allocated for `vec` is now free.

skip case study

Performance Measurement: Time

Case study: Sequential Numbers

```
vec <- c(vec,i)
```

- Note: repeated allocation and copy steps are time consuming.
- Note: the storage space previously allocated for `vec` is now free.
- Bad news: this slot size is just two items less than the slot needed in the next step.
- It cannot be re-used now.
- Garbage collection is necessary eventually.

skip case study

Performance Measurement: Time

Memory Management

Think of two processes

- the computing process
- the memory management process

You influence the memory management in two ways

- by allocation. This is controlled by the program flow.
- by garbage management. This is on demand and influenced by the program state.

Have mercy with your garbage collector.

- If a variable is not used, consider marking it for release using `rm()`.
- If a slot is not used, consider marking it for release by setting it to `NULL`.
- Consider calling the garbage collector explicitly using `gc()` before critical program segments.

Performance Measurement: Time

Case Study: Click-Example

```
click1 <- function(){
    x <- runif(1);y <- runif(1)
    plot(x = x, y = y, xlim = c(0, 1), ylim = c(0, 1),
        main = "Please click on the point",
        xlab = '', ylab = '',
        axes = FALSE, frame.plot = TRUE)
    clicktime <- system.time(xyclick <- locator(1))
    list(timestamp = Sys.time(),
        x = x, y = y,
        xclick = xyclick$x, yclick = xyclick$y,
        tclick = clicktime[3])
}
```


Performance Measurement: Time

Click-Example

```
click <- function(runs = 1){
    dx <- data.frame(click1()) # start up
    for (i in (1:runs)){dx <- rbind(dx, data.frame(click1()))}
    dx <- dx[-1, ] #discard startup
    plot(0, 0,
        main = paste(runs, " clicks registered"),
        xlab = '', ylab = '',
        axes = FALSE, frame.plot = TRUE)# clean up plotting area
    dx
}
```


Performance Measurement: Time

Click-Example: Side Remark

`rbind(dx, data.frame(click1()))` is worse than `vec <- c(vec,i)`.

In `vec <- c(vec,i)` `i` can be appended to the new copy of `vec`.

In `rbind(dx, data.frame(click1()))`, the old information has to be split to make space and the new information has to be inserted.

$dx[1,1]$
$\vdots$
$dx[n-1,1]$ insert $dx[n,1]$
$dx[1,2]$
$\vdots$
$dx[n-1,2]$ insert $dx[n,2]$
$\vdots$
insert $dx[n,k-1]$
$dx[1,k]$
$\vdots$

Performance Measurement: Time

Avoid Growing Variables

If you must use growing variables: remember, R stores variables in 'column first' order.

Use memory preallocation, if possible.

Consider using a `list` variable while accumulating data, and reshape finally as simple as possible.

Files are made for sequential extension. Consider using a file.

Performance Measurement: Time

Avoid Storage Type Conversion

Avoid explicit or implicit type conversions. Keep it as simple as possible, but not simpler.

Prefer using a vector over using a list.

Prefer using a matrix over using a data frame.

But of course if a list or a data frame is needed, don't use a simpler type.

Performance Measurement: Time

Exercise

Write/select a small example function to time.
Use the `click()` model to write a timing function.

Write/select a small scalable example function to time.
Use the `click()` model to write a timing function for a small number of scale levels.

Display the results.

But on the other side: don't try working with a vector when a list is needed. Don't try working with a matrix when a data frame is needed.

Performance Measurement: Memory

Memory Usage

Memory usage is dynamic. Memory may be allocated or released during evaluation, for example to store intermediate results.

In R: Information about memory usage is provided as a side effect of garbage calculation.

Part of the memory management is garbage collection, and the time used for garbage collection may be a major concern.

Garbage collection is a second process which is influenced, but not controlled by your program process.

`system.time()` does not include specific information about the time for garbage collection.

Performance Measurement: Memory

Garbage collection levels

There are three levels of collections.

- Level 0 collects only the youngest generation.
- Level 1 collects the two youngest generations.
- Level 2 collects all generations.

After 20 level-0 collections the next collection is at level 1, and after 5 level-1 collections at level 2.

Further, if a level-n collection fails to provide 20% free space (for each of nodes and the vector heap), the next collection will be at level $n+1$.

(The R-level function `gc()` performs a level-2 collection.)

Performance Measurement: Memory

Exercise

Use `gc.time()` to extend `system.time()` to provide the time spent on garbage collection during the process.

Provide a modified function as `systemwgc.time()` .

Repeat the previous exercise.

Performance Measurement: Memory

Memory Usage: gc()

`gc()`	Trigger garbage collection.
`gcinfo(verbose=TRUE)`	Make garbage collection verbose

See `help(Memory)`.

Performance Measurement: Memory: gc()

gc *Garbage Collection*

Description

A call of gc causes a garbage collection to take place. `gcinfo` sets a flag so that automatic collection is either silent (`verbose=FALSE`) or prints memory usage statistics (`verbose=TRUE`).

Usage

```
gc(verbose = getOption("verbose"), reset=FALSE)
gcinfo(verbose)
```

skip help

Performance Measurement: Memory: gc()

Usage

```
gc(verbose = getOption("verbose"), reset=FALSE)
gcinfo(verbose)
```

Arguments

`verbose`	logical; if `TRUE`, the garbage collection prints statistics about cons cells and the space allocated for vectors.
`reset`	logical; if `TRUE` the values for maximum space used are reset to the current values.

skip help

Performance Measurement: Memory: gc()

Details

A call of gc causes a garbage collection to take place. This will also take place automatically without user intervention, and the primary purpose of calling gc is for the report on memory usage.

However, it can be useful to call gc after a large object has been removed, as this may prompt R to return memory to the operating system.

R allocates space for vectors in multiples of 8 bytes: hence the report of `"Vcells"`, a relict of an earlier allocator (that used a vector heap).

skip help

Performance Measurement: Memory: gc()

Details

When `gcinfo(TRUE)` is in force, messages are sent to the message connection at each garbage collection of the form

```
Garbage collection 12 = 10+0+2 (level 0) ...
6.4 Mbytes of cons cells used (58%)
2.0 Mbytes of vectors used (32%)
```

Here the last two lines give the current memory usage rounded up to the next 0.1Mb and as a percentage of the current trigger value. The first line gives a breakdown of the number of garbage collections at various levels (for an explanation see the 'R Internals' manual).

Code Level Tools

Instrumenting the code

```
debug(fun, text="",
 condition=NULL)
...
```

Set, unset or query the debugging flag on a function.

```
trace(what, tracer,
 exit, at, print, signature,
 where = topenv(parent.frame()),
 edit = FALSE)
...
```

A call to trace allows you to insert debugging code (e.g., a call to browser or recover) at chosen places in any function.

Do not forget the global possibility: `options(error=browser)`

Vectorization

Vectorized Functions

The basic functions in R are vectorized.

In some instances, vectorization is required, for example for functions passed as argument to integrators and optimisers.

A function can be vectorized using `Vectorize()`. `Vectorize()` wraps a call using `mapply()`,

Vectorization by itself does not bring any performance advantage. In particular, `Vectorize()` only puts a loop around the code of a function. Formal vectorization must be accompanied by an access optimisation.

Special Libraries, and Other Ways to Avoid the Problems

THINK !

Consider avoiding the problem.

- Think ! If you can come up with a closed solution, you may avoid all (or most) computation.
- Consider changing the hardware. If there is and old cpu around with only a part of the computing power of an up-to-date cpu, but unused most of the time: running your computation here may save a lot,
- Consider local cooperation. You can share computing power. For example, XGrid is a solution that is readily available. `http://developer.apple.com/hardware/hpc/xgrid_intro.html`.
- Make use of regional resources. For example, BW Grid provides access to more than 1000 CPUs. `http://www.bw-grid.de/allgemeine-informationen/hardware/`.

Special Libraries, and Other Ways to Avoid the Problems

BLAS/LAPACK

For linear algebra, BLAS and LAPACK (or variants thereof) are standard libraries.

BLAS/LAPACK packages are used by default.

If hardware-optimised versions of BLAS/LAPACK are to be used, R must be recompiled.

Special Libraries, and Other Ways to Avoid the Problems

Sparse and/or large matrices

Libraries for sparse or large matrices are available using special R libraries:

```
library(bigmemory)      library(R.huge)
library(Matrix)         library(futile.matrix)
library(SparseM)
```


Special Libraries, and Other Ways to Avoid the Problems

References

CRAN task view

`http://cran.r-project.org/web/views/HighPerformanceComputing.html`

bigmemory: R-Forge Development Page
`https://r-forge.r-project.org/R/?group_id=556`

Large objects for R: R-Forge Development Page
`https://r-forge.r-project.org/R/?group_id=483`

See also
`http://developer.r-project.org/Sparse.html`

Special Libraries, and Other Ways to Avoid the Problems

R to C/C++ Interfaces

The Rcpp package provides a C++ library which facilitates the integration of R and C++

Rcpp: R-Forge Development Page
`https://r-forge.r-project.org/projects/rcpp/` `library(Rcpp)`

Optimisation

Classical Optimisation

Control-flow analysis

Basis: Control-flow graph

Basic block: must enter at beginning, exit only at end (no branches)

Local Optimisation

Classical Techniques

Well known, and widely available in compilers (often only as an option).

Not often used in interpreters.

Some of the following methods are not applicable on an interpreter level, but still may give helpful hints.

Local Optimisation- Classical Techniques

- Constant Folding: Evaluation of Constants at "Compile Time"
- Constant Propagation: Replace Variables by Constants if Value Does Not Change
- Algebraic Simplification And Re-association
- Operator Strength Reduction
- Copy Propagation
- Dead Code Elimination
- Common Subexpression Elimination
- Loop Uncoiling
- Global Optimisations and Data Flow Analysis
- Code Motion
- Machine Optimisation
- Register/Memory Allocation
- Instruction Scheduling
- Peephole Optimisation

skip examples

Local Optimisation

Classical Techniques: Constant folding

Main effect: constant evaluation made at compile time, not at run time.

```
1 + x + 2                    x + 3
```

Constants may be hidden, such as `n = nx *ny`.

back skip examples

Local Optimisation

Classical Techniques: Constant propagation

If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable.

back skip examples

Local Optimisation

Classical Techniques: Algebraic simplification and re-association

Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions. Re-association refers to using properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimisations such as constant-folding or loop-invariant code motion.

The most obvious of these are the optimisations that can remove useless instructions entirely via algebraic identities.

back skip examples

Local Optimisation

Operator Strength Reduction

Operator strength reduction replaces an operator by a "less expensive" one.

back skip examples

Local Optimisation

Copy Propagation

This optimisation is similar to constant propagation, but generalised to non-constant values. If we have an assignment $a = b$ in our instruction stream, we can replace later occurrences of a with b (assuming there are no changes to either variable in-between).

This may be a particularly valuable optimisation since it may be able to eliminate a large number of instructions that only serve to copy values from one variable to another.

back skip examples

Local Optimisation

Dead Code Elimination

If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream.

Dead code frequently results from previous editing operations.

back skip examples

Local Optimisation

Loop Uncoiling

Loop uncoiling, or loop unrollment replaces two or more loop iterations by explicit statements.

back skip examples

Local Optimisation

Code Motion

Code motion unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation. The most common form of code motion is loop-invariant code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop.

back skip examples

Local Optimisation

Register and Memory Allocation

Registers are the fastest kind of memory available, but as a resource, they can be scarce. The problem is how to minimise traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches.

back skip examples

Local Optimisation

Common Subexpression Elimination

Two operations are common if they produce the same result. Note: they may appear in very different form!

back skip examples

Local Optimisation

Global Optimisations, Data-Flow Analysis

The additional analysis the optimiser must do to perform optimisations across basic blocks is called data-flow analysis.

back skip examples

Local Optimisation

Machine Optimisations

In this pass, specific machines features (specialised instructions, hardware pipeline abilities, register details) are taken into account to produce code optimised for this particular architecture.

back skip examples

Local Optimisation

Instruction Scheduling

Another extremely important optimisation of the final code generator is instruction scheduling. Because many machines have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions.

back skip examples

Local Optimisation

Peephole Optimisations

Peephole optimisation is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine dependent code improvements. For example, peephole optimisations might include elimination of multiplication by 1, elimination of load of a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect. back skip examples

Local Optimisation

Optimisation in R

Support tools for most standard techniques are available in `library(compiler)`.

Some are supported in the byte code compiler (under development).

R Built-in Optimisations

Function Arguments

Function Arguments in R

- As a rule, parameters are passed by value. (i.e. copied. Beware! This means memory allocation and copying.)
- The main exception are `environments`. These are not copied, but passed as a reference. You can make use of this and pass use information efficiently by just passing its name and the environment to look it up.
- R uses lazy evaluation: if a value needs to be calculated, the evaluated expression needed to calculate it is passed, together with the environment where the evaluation should take place. This is called a `promise`.
- R can pass a value by reference, if it can prove that it is unchanged during evaluation. This is used extensively for `.Internal` functions.

Function Arguments

Function Types

Functions come in three types

- Usual function closures: arguments are matched and a matched call is constructed.
- Specials: arguments are **not** evaluated before C code is called. Low Overhead.
- Builtins:.

`.Primitive` can be used to call internal C functions with minimal interface.

Other internal functions are interfaced using `.Internal`.

See list in *R Internals Ch 2* .

Function Arguments

Internal Functions

If an `.Internal` function can be used, prefer it over other solutions. It may be cheaper.

Do not spend time optimising something `.Internal`. (And do not spend time optimising it. This is something already in the optimised libraries.)

If only information from a variable descriptor is used by a function, assume the function is implemented as `.Internal`.
For example, `length()` is `.Internal` (and cheap to call).

Function Arguments

Parameter Passing Exceptions

By default, function arguments are passed by value.

There are few exceptions defined by in the language:

- environments
- promises
- ...

Function Arguments

Parameter Passing Exceptions: Environments

Each function definition, and each function call defines its environment.

Environments can be generated explicitly using
`new.environment()`.

Objects can be assigned to an environment using the `environment` in function `assign()`.

For an in-depth discussion of the possibilities, see (Gentleman and Ihaka, 2000).

Function Arguments

Example: Cache for Fibonacci

Creating a persistent local environment for a function to implement a persistent memory:

```
fibonacci <- local({
  memo <- c(1, 1, rep(NA, 100))
  f <- function(x) {
    if(x == 0) return(0)
    if(x < 0) return(NA)
    if(x > length(memo))  stop("'x' too big for implementation")
   if(!is.na(memo[x])) return(memo[x])
   ans <- f(x-2) + f(x-1)
   memo[x] <<- ans
   ans}
})
```

From P. Burns, R Inferno, Circle 6.1

Compilation

Byte code

- Installing an intermediate step between interpreted code and compilation is an active area in the R development. (Keyword "byte code")
- Compilation needs a controlled environment. Current approach; name spaces as used for packages.
- Good news: pre-compilation comes for free for packages (if implemented on your implementation).

Compilation

Byte Code

The byte code compiler needs hints about the function interface used for external access.

These hints are part of the declaration of a name space that must be included with the package. (See *Writing R extensions*)

Compilation

Byte Code

If a package is already in the repository in byte-compiled form, you get it for free. As of R 2.14, the basic package are byte-compiled.

If it is not byte-compiled, to install a package using the byte-compiler, use the command option

```
R CMD INSTALL ...--byte-compile ...
```

or use

```
install.packages(file, repos=NULL, type="source",
          INSTALL_opts="--byte-compile")
```

If you are building a package, use

```
ByteCompile:  true
```

in the `DESCRIPTION` file to get it byte-compiled by default.

Compilation

Byte Code

To generate a byte-compiled version of an individual function, use `library(compiler)` and function `cmpfun()`.

Compilation

Example

As a case study, we use an algorithm to calculate the shorth functional, a non-parametric functional for the analysis of one-dimensional data. The complexity of the algorithm keeps time consumption annoying, yet still in a range allowing for a scalable variation of sample sizes.

Shorth length $S_\alpha(x)$ at x covering α: minimum length of an interval containing x and covering a proportion α of the data.

$$S_\alpha(x) = \min\{|I| : x \in I; P(I) \geq \alpha\}$$

Shorth plot: Plot $x \mapsto -S_\alpha(x)$ for a collection of coverages α.

For the implementation, see `<http://lshorth.r-forge.r-project.org/>`.

Compilation

Case Study: Byte Code

No byte-code

```
> n <- 100
> system.time(lshorth(rnorm(n)))
   user  system elapsed
  0.022   0.002   0.027
> n <- 1000
> system.time(lshorth(rnorm(n)))
  0.124   0.008   0.134
> n <- 10000
> system.time(lshorth(rnorm(n)))
  2.213   0.028   2.237
```

Byte-code

```
> n <- 100
> system.time(lshorth(rnorm(n)))
   user  system elapsed
  0.017   0.002   0.021
> n <- 1000
> system.time(lshorth(rnorm(n)))
  0.114   0.015   0.131
> n <- 10000
> system.time(lshorth(rnorm(n)))
  1.135   0.030   1.167
```

Moderate until upto 50% gain in total job turnaround time.
Note: base libraries have already been byte-compiled.

Profiling

A Case Study: Melbourne Data

Melbourne Temperature Data

This example will be used for the remainder of this section.

```
library(lshorth)
melbourne3 <- data.frame(read.csv
("/data/melbourne/temp v pressure 3 hourly intervals.csv"))
dt<-as.POSIXlt(melbourne3[,1])#lenght 9??

# 15h data
melbourne15h <- melbourne3[dt$hour==15,]
melbourne15h$TomorrowT <- c(melbourne15h[-1,2], NA)

thigh <- c(32,99); tmed <- c(25.6,32); tlow <- c(21.7,25.6)
```


A Case Study: Melbourne Data

Melbourne Temperature Data

```
plotcond <- function(tlim,plim,main=NULL,...){
incond <-melbourne15h[
(melbourne15h[,2]>= tlim[1])  &
(melbourne15h[,2]<= tlim[2])  &
(melbourne15h[,3]>= plim[1]) &
(melbourne15h[,3]<= plim[2]),]
diffT <- incond[,4]-incond[,2]
diffT <- diffT[is.finite(diffT)]
ls <- lshorth(diffT, probs=c( 0.125,0.25,0.5,0.75, 0.875),plot=FALSE)
if (is.null(main)){main=paste("T ",tlim,"p ",plim, sep=" ")}
plot(ls, frame.plot=FALSE, main=main, cex=1.5, ...)
}
```


A Case Study: Melbourne Data

```
qtemp <- quantile(melbourne15h[,2], probs=seq(0,1,0.125)) # 1..9
qpress <- quantile(melbourne15h[,3], probs=seq(0,1,0.125))

plotcond3 <- function(tlim,main=NULL,...){
incond <-melbourne15h[ (melbourne15h[,2]>= tlim[1])
     & (melbourne15h[,2]<= tlim[2]),]
qpress<-quantile(incond[,3], probs=seq(0,1,1/6),na.rm=TRUE)

plotcond(tlim, plim=c(qpress[6],9999),
  main=paste("T: ", tlim[1], "...  ","C   p:",
  qpress[6],"...  ", "hpa", sep=""),  legend=NULL)
plotcond(tlim, plim=c(qpress[3],qpress[4]),
  main=paste("T: ", tlim[1], "...",tlim[2],"C   p:",
  qpress[3],"...",qpress[4], "hpa", sep=""), legend=NULL)
plotcond(tlim, plim=c(0,qpress[2]),
  main=paste("T: ", tlim[1], "...",tlim[2],"C   p:",
  "...",qpress[2], "hpa", sep=""),  legend=NULL)
}
```


A Case Study: Melbourne Data

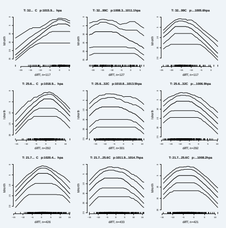
Melbourne Temperature Data

```
bigplot <- function() {
oldpar <- par(mfrow=c(3,3),
   cex.lab =1.5, cex.main =1.5)
plotcond3(tlim=thigh)

plotcond3(tlim= tmed)

plotcond3(tlim= tlow)

par(oldpar)
}
```


A Case Study: Melbourne Data

Melbourne Temperature Data

```
> system.time(bigplot())
   user  system elapsed
  0.377   0.021   0.405
```



Profiling Time: Rprof()

Rprof *Enable Profiling of R's Execution*

Description

Enable or disable profiling of the execution of R expressions.

Usage

```
Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02,
       memory.profiling=FALSE)
```

skip help

Profiling Time: Rprof()

Usage

```
Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02,
       memory.profiling=FALSE)
```

Arguments

filename	The file to be used for recording the profiling results. Set to NULL or "" to disable profiling.
append	logical: should the file be over-written or appended to?
interval	real: time interval between samples.
memory.profiling	logical: write memory use information to the file?

skip help

Profiling Time: Rprof()

Details

Enabling profiling automatically disables any existing profiling to another or the same file.
Profiling works by writing out the call stack every `interval` seconds, to the file specified. Either the `summaryRprof` function or the wrapper script `R CMD Rprof` can be used to process the output file to produce a summary of the usage; use `R CMD Rprof --help` for usage information.
How time is measured varies by platform: on a Unix-alike it is the CPU time of the R process, so for example excludes time when R is waiting for input or for processes run by `system` to return.
Note that the timing interval cannot usefully be too small: once the timer goes off, the information is not recorded until the next timing click (probably in the range 1–10msecs).
Functions will only be recorded in the profile log if they put a context on the call stack (see `sys.callssys.calls`). Some primitive functions do not do so: specifically those which are of type `"special"` (see the 'R Internals' manual for more details).

Profiling Time: SummaryRprof()

summaryRprof	*Summarise Output of R Sampling Profiler*

Description

Summarise the output of the `Rprof` function to show the amount of time used by different R functions.

Usage

```
summaryRprof(filename = "Rprof.out", chunksize = 5000,
             memory=c("none","both","tseries","stats"),
             index=2, diff=TRUE, exclude=NULL)
```

skip help

Profiling Time: SummaryRprof()

Usage

```
summaryRprof(filename = "Rprof.out", chunksize = 5000,
             memory=c("none","both","tseries","stats"),
             index=2, diff=TRUE, exclude=NULL)
```

Arguments

`filename`	Name of a file produced by `Rprof()`
`chunksize`	Number of lines to read at a time
`memory`	Summaries for memory information. See 'Details' below
`index`	How to summarize the stack trace for memory information. See 'Details' below.
`diff`	If `TRUE` memory summaries use change in memory rather than current memory
`exclude`	Functions to exclude when summarizing the stack trace for memory summaries

skip help

Profiling Time: SummaryRprof()

Details

This function provides the analysis code for `Rprof` files used by `R CMD Rprof`.
As the profiling output file could be larger than available memory, it is read in blocks of `chunksize` lines. Increasing `chunksize` will make the function run faster if sufficient memory is available.
When called with `memory.profiling = TRUE`, the profiler writes information on three aspects of memory use: vector memory in small blocks on the R heap, vector memory in large blocks (from `malloc`), memory in nodes on the R heap. It also records the number of calls to the internal function `duplicate` in the time interval. `duplicate` is called by C code when arguments need to be copied. Note that the profiler does not track which function actually allocated the memory.
With `memory = "both"` the change in total memory (truncated at zero) is reported in addition to timing data.

skip help

Profiling Time: SummaryRprof()

Details

With `memory = "tseries"` or `memory = "stats"` the `index` argument specifies how to summarize the stack trace. A positive number specifies that many calls from the bottom of the stack; a negative number specifies the number of calls from the top of the stack. With `memory = "tseries"` the index is used to construct labels and may be a vector to give multiple sets of labels. With `memory = "stats"` the index must be a single number and specifies how to aggregate the data to the maximum and average of the memory statistics. With both `memory = "tseries"` and `memory = "stats"` the argument `diff = TRUE` asks for summaries of the increase in memory use over the sampling interval and `diff = FALSE` asks for the memory use at the end of the interval.

Profiling Time

Profiling Time

```
Rprof("lshorth_prof.txt", interval=0.002)
bigplot()
Rprof(NULL)
summaryRprof("lshorth_prof.txt")
```


Profiling Time: SummaryRprof() I

```
> summaryRprof("lshorth_prof.txt")
$by.self
               self.time self.pct total.time total.pct
"axis"             0.088    34.65      0.088     34.65
"[.data.frame"     0.042    16.54      0.066     25.98
"sort.int"         0.030    11.81      0.030     11.81
"lshorth"          0.020     7.87      0.074     29.13
"[.factor"         0.020     7.87      0.020      7.87
"which.min"        0.018     7.09      0.022      8.66
"title"            0.018     7.09      0.018      7.09
":"                0.004     1.57      0.004      1.57
"&"                0.004     1.57      0.004      1.57
"plot.new"         0.004     1.57      0.004      1.57
"plotcond"         0.002     0.79      0.234     92.13

...
```


Profiling Time: SummaryRprof() I

```
$by.total
                total.time total.pct self.time self.pct
"bigplot"            0.254    100.00     0.000     0.00
"plotcond3"          0.254    100.00     0.000     0.00
"plotcond"           0.234     92.13     0.002     0.79
"plot.lshorth"       0.112     44.09     0.000     0.00
"plot"               0.112     44.09     0.000     0.00
"axis"               0.088     34.65     0.088    34.65
"lshorth"            0.074     29.13     0.020     7.87
"[.data.frame"       0.066     25.98     0.042    16.54
"["                  0.066     25.98     0.000     0.00
"sort.int"           0.030     11.81     0.030    11.81
"sort.default"       0.030     11.81     0.000     0.00
"sort"               0.030     11.81     0.000     0.00
"which.min"          0.022      8.66     0.018     7.09
```


Profiling Time: SummaryRprof() II

```
"[.factor"        0.020     7.87     0.020     7.87
"title"           0.018     7.09     0.018     7.09
"Axis.default"    0.018     7.09     0.000     0.00
"Axis"            0.018     7.09     0.000     0.00
"rug"             0.018     7.09     0.000     0.00
":"               0.004     1.57     0.004     1.57
"&"               0.004     1.57     0.004     1.57
"plot.new"        0.004     1.57     0.004     1.57
">"               0.002     0.79     0.002     0.79
"par"             0.002     0.79     0.002     0.79
"lines.default"   0.002     0.79     0.000     0.00
"lines"           0.002     0.79     0.000     0.00
"plot.xy"         0.002     0.79     0.000     0.00
$sample.interval
[1] 0.002
$sampling.time
[1] 0.254
```

Profiling Time

Using Rprof

The profile is easily scattered with garbage, and there is no semantic filter.

Use `Rprof()` selectively for zones of interest, do several runs using the append option.

Profiling Time

Profiling Time and Memeory

```
Rprof("lshorth_profm.txt",
   interval=0.002,
   memory.profiling=TRUE)
bigplot()
Rprof(NULL)
summaryRprof("lshorth_profm.txt", memory="both")
```


Profiling Time I

```
> summaryRprof("lshorth_profm.txt", memory="both")
$by.self
               self.time self.pct total.time total.pct mem.total
"axis"             0.092    39.32      0.092     39.32       7.7
"lshorth"          0.028    11.97      0.044     18.80      24.1
"[.data.frame"     0.024    10.26      0.056     23.93       7.6
"lapply"           0.022     9.40      0.022      9.40       0.0
"[.factor"         0.018     7.69      0.018      7.69       2.5
"title"            0.018     7.69      0.018      7.69       0.1
"which.min"        0.012     5.13      0.012      5.13       2.6
"<="               0.008     3.42      0.008      3.42       0.0
"&"                0.004     1.71      0.004      1.71       1.0
"<"                0.004     1.71      0.004      1.71       6.1
"plot.new"         0.002     0.85      0.002      0.85       1.4
"sys.call"         0.002     0.85      0.002      0.85       0.3

$by.total
```


Profiling Time II

```
                total.time total.pct mem.total self.time self.pct
"bigplot"            0.234    100.00      40.9     0.000     0.00
"plotcond3"          0.234    100.00      40.9     0.000     0.00
"plotcond"           0.218     93.16      39.7     0.000     0.00
"plot.lshorth"       0.134     57.26       9.2     0.000     0.00
"plot"               0.134     57.26       9.2     0.000     0.00
"axis"               0.092     39.32       7.7     0.092    39.32
"[.data.frame"       0.056     23.93       7.6     0.024    10.26
"["                  0.056     23.93       7.6     0.000     0.00
"lshorth"            0.044     18.80      24.1     0.028    11.97
"lapply"             0.022      9.40       0.0     0.022     9.40
"lines.default"      0.022      9.40       0.0     0.000     0.00
"lines"              0.022      9.40       0.0     0.000     0.00
"par"                0.022      9.40       0.0     0.000     0.00
"plot.xy"            0.022      9.40       0.0     0.000     0.00
"unlist"             0.022      9.40       0.0     0.000     0.00
"[.factor"           0.018      7.69       2.5     0.018     7.69
"title"              0.018      7.69       0.1     0.018     7.69
"Axis.default"       0.016      6.84       0.4     0.000     0.00
```


Profiling Time III

```
"Axis"          0.016     6.84      0.4     0.000     0.00
"rug"           0.016     6.84      0.4     0.000     0.00
"which.min"     0.012     5.13      2.6     0.012     5.13
"<="            0.008     3.42      0.0     0.008     3.42
"&"             0.004     1.71      1.0     0.004     1.71
"<"             0.004     1.71      6.1     0.004     1.71
"plot.new"      0.002     0.85      1.4     0.002     0.85
"sys.call"      0.002     0.85      0.3     0.002     0.85
"%in%"          0.002     0.85      0.3     0.000     0.00
"match"         0.002     0.85      0.3     0.000     0.00

$sample.interval
[1] 0.002

$sampling.time
[1] 0.234
```


Profiling Time

Using Rprof

Memory is attributed to the function active at the end of the sampling interval.

This may be misleading.

Profiling Time

Profiling Functions

```
Rprof(filename = "Rprof.out",
              append = FALSE,
              interval = 0.02,
              memory.profiling=FALSE)

summaryRprof(filename = "Rprof.out",
              chunksize = 5000,
              memory=c("none","both","tseries","stats"),
              index=2, diff=TRUE, exclude=NULL)
```

Summaries for time information.

Profiling Memory: `Rprofmem()` I

`Rprofmem` *Enable Profiling of R's Memory Use*

Description

Enable or disable reporting of memory allocation in R.

Usage

```
Rprofmem(filename = "Rprofmem.out",  append = FALSE, threshold = 0)
```

skip help

Profiling Memory: `Rprofmem()`

Usage

```
Rprofmem(filename = "Rprofmem.out",  append = FALSE, threshold = 0)
```

Arguments

`filename`	The file to be used for recording the memory allocations. Set to `NULL` or `""` to disable reporting.
`append`	logical: should the file be over-written or appended to?
`threshold`	numeric: allocations on R's "large vector" heap larger than this number of bytes will be reported.

skip help

Profiling Memory: `Rprofmem()`

Details

Enabling profiling automatically disables any existing profiling to another or the same file.
Profiling writes the call stack to the specified file every time `malloc` is called to allocate a large vector object or to allocate a page of memory for small objects. The size of a page of memory and the size above which `malloc` is used for vectors are compile-time constants, by default 2000 and 128 bytes respectively.
The profiler tracks allocations, some of which will be to previously used memory and will not increase the total memory use of R.

Profiling Memory: `Rprofmem()`

Profiling Memory

```
Rprofmem("lshorth_profmem.txt")
bigplot()
Rprofmem(NULL)
```

Note: at sampling time, only the size of memory requested is known. There is no variable name associated to it.

To identify the objects, given their size, you can use a code snippet as in

```
## find the 10 largest objects in the base package
z <- sapply(ls("package:base"), function(x)
            object.size(get(x, envir = baseenv())))
as.matrix(rev(sort(z))[1:10])s
```

A `summary()` method for `Rprofmem()` is still under construction.

Profiling Memory: `Rprofmem()`

```
> noquote(readLines("lshorth_profmem.txt", n=15))
 [1] 1104 :"bigplot"
 [2] 4872 :"bigplot"
 [3] 4872 :"bigplot"
 [4] 1064 :"bigplot"
 [5] 280 :"bigplot"
 [6] 816 :"bigplot"
 [7] 1584 :"bigplot"
 [8] 384 :"bigplot"
 [9] 256 :"bigplot"
[10] new page:"bigplot"
[11] 456 :"<Anonymous>" "par" "bigplot"
[12] 1728 :"<Anonymous>" "par" "bigplot"
[13] 1728 :"<Anonymous>" "par" "bigplot"
[14] 1064 :"<Anonymous>" "par" "bigplot"
[15] 328 :"<Anonymous>" "par" "bigplot"
```


Profiling Memory: `Rprofmem()` I

Profiling Memory

A memory threshold can be installed to avoid uninformative events.

```
Rprofmem("lshorth_profmem1024.txt", threshold=1024)
bigplot()
Rprofmem(NULL)
```


Profiling Memory: `Rprofmem()` I

```
> noquote(readLines("lshorth_profmem1024.txt", n=30))
 [1] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
 [2] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
 [3] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
 [4] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
 [5] 76512 :"NextMethod" "[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
 [6] 2824 :"NextMethod" "[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
 [7] 2824 :"NextMethod" "[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
 [8] 2824 :"[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
 [9] 1231360 :"[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
[10] 2824 :"[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
[11] 1231360 :"[.factor" "[" "[.data.frame" "[" "plotcond3" "bigplot"
[12] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[13] 1231360 :"[.data.frame" "[" "plotcond3" "bigplot"
[14] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
[15] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[16] 5608 :"[.data.frame" "[" "plotcond3" "bigplot"
```


Profiling Memory: `Rprofmem()` II

```
[17] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
[18] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[19] 5608 :"[.data.frame" "[" "plotcond3" "bigplot"
[20] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
[21] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[22] 5608 :"[.data.frame" "[" "plotcond3" "bigplot"
[23] 76512 :"[.data.frame" "[" "plotcond3" "bigplot"
[24] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[25] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[26] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[27] 8232 :"anyDuplicated.default" "anyDuplicated" "[.data.frame" "[" "plotcond3" "big
[28] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[29] 2824 :"[.data.frame" "[" "plotcond3" "bigplot"
[30] 1231360 :"[.data.frame" "[" "plotcond3" "bigplot"
```


Profiling Memory

Duplicates

The internal function `duplicate` is called when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

It is a major cause of hard-to-predict memory use in R.

`tracemem()` can be used to track the creation of duplicates.

Profiling Memory: tracemem()

tracemem	*Trace Copying of Objects*

Description

This function marks an object so that a message is printed whenever the internal function `duplicate` is called. This happens when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

Usage

```
tracemem(x)
untracemem(x)
retracemem(x, previous = NULL)
```

skip help

Profiling Memory: tracemem()

Usage

```
tracemem(x)
untracemem(x)
retracemem(x, previous = NULL)
```

Arguments

`x`	An R object, not a function or environment or `NULL`.
`previous`	A value as returned by `tracemem` or `retracemem`.

skip help

Profiling Memory: tracemem()

Details

This functionality is optional, determined at compilation, because it makes R run a little more slowly even when no objects are being traced. `tracemem` and `untracemem` give errors when R is not compiled with memory profiling; `retracemem` does not (so it can be left in code during development).
When an object is traced any copying of the object by the C function `duplicate` or by arithmetic or mathematical operations produces a message to standard output. The message consists of the string `tracemem`, the identifying strings for the object being copied and the new object being created, and a stack trace showing where the duplication occurred. `retracemem()` is used to indicate that a variable should be considered a copy of a previous variable (e.g. after subscripting).
The messages can be turned off with `tracingState`.
It is not possible to trace functions, as this would conflict with `trace` and it is not useful to trace `NULL`, environments, promises, weak references, or external pointer objects, as these are not duplicated.
These functions are primitive.

Profiling Memory: tracemem()

Tracing Memory

```
tracemem(thigh)
bigplot()
untracemem(thigh)
```


Profiling Memory: Rprofmem() I

Profiling Memory

```
> tracemem(thigh)
[1] "<0x1090a7188>"
```

ToDo 7: Example

Gentleman, Robert and Ross Ihaka. 2000. *Lexical scope and statistical computing*, Journal of Computational and Graphical Statistics **9**, 491–508.