R Expression Dependency Graphs

Clark Fitzgerald

March 20, 2017

Abstract

1 Introduction

The idea behind code analysis is to treat the code itself as a data structure as "Programming on the language" opens up rich possibilities. This is an old idea, both the R and Julia language references cite Lisp as the inspiration [17] [3].

Compilers have used all manners of static analysis and intermediate optimizations to create more efficient code. Interpreted languages are much more limited in this respect. This project explores the use of an alternative evaluation model to improve performance while preserving language semantics.

The evaluation model for interpreted languages is simple. Each expression of code is evaluated in the order that it appears. Informally each expression is a line of code. This can be viewed as a set of constraints on the evaluation order of the expressions:

1. expression 1 executes before expression 2
2. expression 2 executes before expression 3
3. ...

What if these constraints are relaxed? Suppose expression 1 defines the variable `x`, which is not used until expression 17. Then one has the constraint:

1. expression 1 executes before expression 17

This can be generalized into a directed graph by considering expressions as nodes and constraints as edges. The edges are implicit based on the order of the statements in the code. Add an edge from $i \to k$ if expression k depends on the execution of expression i. It's safe to assume $i < k$, because expressions appearing later in a program can't affect expressions which have already run. Hence the expression graph is acyclic, ie. a DAG.

Scheduling execution based on the expression graph allows some expressions to execute in parallel. For example, the following adjacent lines are independent, so they can be computed simultaneously:

```
sx = sum(x)
sy = sum(y)
```

Mathematically, the standard evaluation model is a total ordering on the set of expressions in the code. The dependency graph is a partial ordering.

On a broader note, this is about embedding more intelligence into the system. Many languages have mechanisms or third party tools for explicitly requesting asynchronous evaluation. These typically require changing the code. This introduces complexity, makes maintenance more difficult, and makes the code less portable. It's more convenient to have one version of the code which can be passed to a system which will just "do the right thing", adjusting to different platforms, work loads, and data sizes on the fly.

2 Literature Review

The expression graph proposed above is similar to use-definition and definition-use chains. A definition-use chain consists of all expressions using a variable following the definition of that variable. This amounts to a subset of the edges in the expression graph, since it's possible that expressions depend on each other without variables. For example, consider the following R code to save a plot to a pdf:

```
pdf("xy.pdf")
plot(x, y)
title("x and y")
dev.off()
```

These expressions depend on each other, but only `plot(x, y)` uses variables.

The Use-definition chain has been around since at least 1978 when it was used to remove dead (unused) code [10]. Code usefulness is defined recursively; a computation is useful if the result is used later by another computation. This is combined with the "base case" of usefulness, a set of operations considered intrinsically useful. This author considers calls to subroutines and branch test instructions as intrinsically useful. We might take this idea and tweak it a little- define expressions in a data analysis script as intrinsically useful if they have a side effect, for example saving data to disk.

More general than the use-definition chain is the program dependence graph (PDG)[7]:

> A PDG node represents an arbitrary sequential computation (e.g., a basic block, a statement, or an operation). An edge in a PDG represents a control dependence or a data dependence. PDGs do not contain any artificial sequencing constraints from the program text; they reveal the ideal parallelism in a program. [15]

[15] goes on to make the practical distinction between ideal parallelism and useful parallisism. Overhead implies that the two often differ. The expression graph differs from the PDG since it allows the permutation of operations in a basic block.

The hierarchical task graph (HTG) was introduced in [8] to detect task parallelism in source code for use in compilers. As in the others, it incorporates the control flow for a fine grained parallelism. They allow 'compound nodes' containing nested HTG's. [6] describe constructing a task graph based on annotating the source code the program. [1] presents a model for predicting the run time of programs based on a task graph.

The literature cited in this section focuses primarily on compiled languages along with careful analysis of control flow. Most examples and applications presented along with these papers are for well-defined algorithmic problems. These algorithmic problems are often quite different than a high level data analysis script which may call down into several different algorithms.

3 Languages Requirements

We can consider building the expression graphs described above for languages that are

- open source
- used for data analysis
- high level
- interpreted
- enable metaprogramming

Metaprogramming warrants more explanation. This refers to programmatically inspecting and potentially modifying code from within the language. It is needed to determine when variables are created and used, among other things. The current popular languages satisfying these requirements are Python, Julia, and R.

The basic unit to analyze is a single code expression. Consider how an expression uses symbols, aka variables or names. An expression may do any combination of the following things:

1. Define new symbols: `x = 10`
2. Use existing symbols: `sin(x)`
3. Redefine existing symbols: `x = 20`

Consider sorting a numeric vector as in Table 1. The Julia and Python methods modify their arguments in place. From a computational standpoint this is great, since it allows the implementations to use more space efficent sorting techniques. However, from a code analysis standpoint this behavior is undesirable, since it means that we need to assume generally that

Table 1: Sorting `x` in place

language	code
Python	`x.sort()`
Julia	`sort!(x)`
R	`x = sort(x)`

every method call in these languages both uses and updates the object. Since data analysis scripts mainly consist of function and method calls this will excessively contrain the problem.

It may be possible to recursively examine all functions and methods which are used, but this is not ideal for a couple reasons. First, it would require analyzing the underlying library code, which is orders of magnitude more code than what the user has written. Second, eventually we'll get to compiled code which requires totally different methods. For example, C code parsed with LLVM can be used to programmatically generate use-definition chains [12].

Hence functional programming and pass by value semantics make constructing graphs much more feasible. The R language is ideal in this respect.

4 Related Work

Bengston's `future` package provides a mechanism for asynchronous assignment and evaluation of R expressions [2]. Once the expression graph is created it might be possible to use similar mechanisms for evaluation.

Hester's covr package [9] checks unit test coverage of R code. It is a practical example of computing on the language, programmatically modifying the code by recording calls as they are made. He pointed out a nice relevant idea distinguishing between AST's, parse trees, and "lossless syntax trees": `https://news.ycombinator.com/item?id=13628412`. To inject code into a script one needs to be very careful to preserve structure lost after parsing, ie. comments and formatting. This is a non-trivial task.

A similar strategy of recording calls should work to collect timings and resource usage for functions called with various data sizes. This could be implemented with something like `System.time()`. This would allow us to time each expression. Then we can potentially use this to change the execution: ie. if $n > 1000$ then run a multicore version. This resembles something like Profile Guided Optimization (PGO).

Maybe a simpler way to do this is to just use the built in profiler. Use the results to determine whether parallelization is worth it, and maybe set some bounds for expected performance changes if one uses various forms of parallelism.

There may be potential to use statistical methods for this. Run it many times, collecting profiling results for various values and use this as the training data to produce a rule such as: if $n > 10^6$ then run it as multicore.

The vignette in Tierney's `proftools` package has some nice examples of visualizing profiling data [18]. The call graphs and related visualizations are conceptually similar to what might be done with the expression graph. `profvis` integrates with the IDE to indicate the actual line of source code along with the related timing info [5].

Xie's `knitr` facilitates reproducible computations for chunks of code in Rmarkdown documents [20]. One feature it enables is caching, ie. it doesn't need to run a chunk of code if nothing has changed. One can manually specify the chunk dependencies by relative or absolute indices, ie. -2 for the chunk 2 blocks in front of the current chunk, or 1 for the first chunk. This seems unreliable because it requires the user to accurately infer the dependency information, and it doesn't automatically adjust if one inserts new chunks in the document.

Knitr also has an `autodep` option to infer this dependency information. This works by comparing the global variables existing before and after running the code in each chunk. It stores this information in special files. So it doesn't use any static analysis of the code.

But the structure of knitr blocks here is actually very appealing- this is a great use case for the parallelism. Why not evaluate the chunks in parallel if possible? The case for Jupyter notebooks is similar, but would require an equivalent code dependency graph for Python.

5 Graph Construction

To start off we make two assumptions on the program. First it's assumed to be correct, meaning that it will run sequentially without errors. Second every expression should be strictly necessary. This can be achieved through a preprocessing step removing dead code.

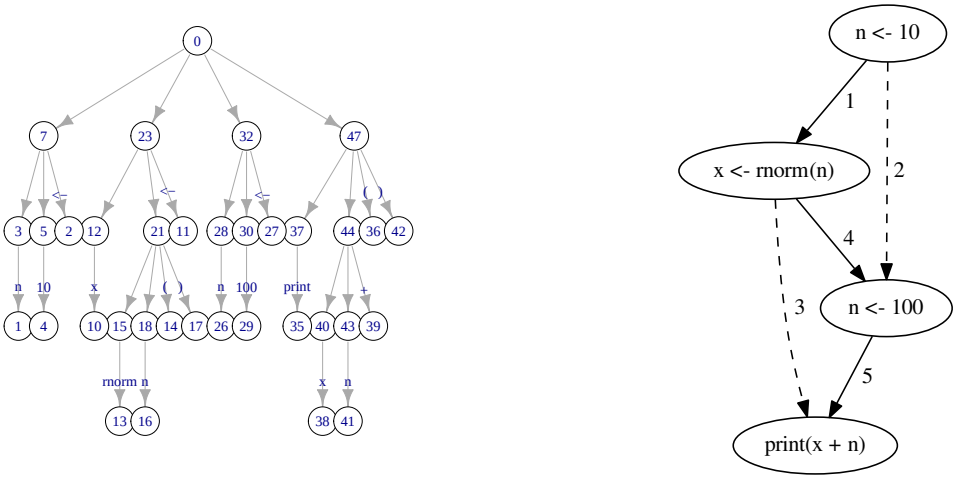
The expression graph and related data structures are related to the parsed script, referred to as the **parse tree**. The expression graph is a function of the parse tree. Since the parser doesn't care about non significant white space and comments, different scripts can produce the same parse tree. Many parse trees can give rise to the same code graph. For example, the expression graph shouldn't care if one uses `=` or `<-` for assignment. Nor will it care about the ordering of two adjacent lines binding a symbol to a literal constant:

```
a = 1
b = 2
```

Therefore we lose information by converting from a parse tree to an expression graph.

We can also introduce an artificial (graph style) source and sink representing the beginning and end of the program, respectively.

Figures 1a and 1b illustrate the ideas of a parse tree an expression dependency graph for the four lines of code in listing 1. Edges 1 and 3 represent the respective uses of the variable `n` and `x`. Edge 2 comes from the redefinition of `n`. Edge 5 propagates the most recent definition of `n`. The least obvious is edge 4, which is necessary to respect R's lexical scoping semantics since `x <- rnorm(n)` uses the first definition of `n`. The general rule here is that all statements using one version of `n` must execute before `n` can be redefined.



(a) Parse tree

(b) Expression dependency graph

Figure 1: Different representations of the script in Listing 1

The dashed edges in 1b are redundant for representing expression dependence given the other edges. Indeed, the code in listing 1 must run sequentially. One may wish to remove such redundant edges, especially for visual presentation of a larger program.

Listing 1: Simple script

```
n <- 10
x <- rnorm(n)
n <- 100
print(x + n)
```

The existence of some edges may depend on conditional statements which can't be known until run time. In this case the conservative and correct way to handle the situation is to add the edges in question. For example, in the following code one assumes that the expression `x <- 10` will run.

```
# coinflip() randomly returns TRUE or FALSE
if(coinflip()){
    x <- 10
}
```

6 Task Based Parallelism

It's reasonable to try to improve code performance if slow speed affects many users. For scripts, superficially it might appear that few people are affected. Indeed, if a researcher

writes one script that takes a couple minutes to run, and they run it a couple times then it doesn't matter much, and there's no point in attempting to accelerate it. However, these same sorts of "scripts" can be used in more serious ways. For example, scripts can be used as Extract Transform Load (ETL) tools that run as batch jobs every day or every hour. Then the script runs many times, so it's important to realize more performance. For ease of use and maintainability it's nice to have automatic tools to accelerate the performance. One can write natural, beginner level scripts and have them run much faster.

As of 2009, most efforts to parallelize R have focused on the lower level programming mechanisms [16]. These needed to be in place before any higher level automatic detection could be built and function.

Let k be the number of cores on a machine. To accelerate code using this single machine the best possible case is if we can keep all k cores busy at once. This will happen if there are nk expressions which can be independently scheduled for some positive integer n. On a two core machine that script might look something like:

```
# These could be run in parallel
a = long_running_func()
b = long_running_func()
```

The worst possible case is if the second long running computation depends on the first, and everything else depends on the second. Then it must run in serial so parallelism can't help.

```
a = long_running_func()
b = long_running_func2(a)  # depends on a
# Now perform many operations on b
```

6.1 Static Execution

If overhead and expression run time are approximately known then the question of optimal execution for the complete expression dependency graph can be framed as a scheduling optimization problem and solved statically. The objective function to minimize is the total wall clock time to complete execution. Constraints come from the depenedency graph and that at most k cores may be active at one time.

To consider an alternative evaluation model the overhead required to parallelize an expression should require less time than running the expression itself. Rounding to orders of magnitude, here are some rough times for reference executing on a modest machine. Simple R expressions take 10^{-7} seconds to evaluate. Using a system level parallel fork requires 10^{-3} seconds of overhead. Evaluation on an existing local socket cluster takes 10^{-4} seconds. Then either of these well established methods for parallelism in R won't become efficient until the code under evaluation takes on the order of 10^{-3} seconds. These timings include latency for interprocess communication on a single machine. Bandwidth is also an issue, since serializing large amounts of data between processes, ie. millions of floating point numbers, will impact performance. For example, listing 2 squares each element of a vector of one million floating point numbers. Memory transfer overhead causes a parallel evaluation to take more than an

order of magnitude more time than the regular version. Technical solutions such as threading and shared memory have the potential to reduce these sources of overhead.

Listing 2: Overhead caused by memory transfer

```
library(microbenchmark)
library(parallel)
n = 1e6
x = rnorm(n)

# Elementwise, return vector of length n
############################################################

# 1.4 ms
microbenchmark(y = x*x)

# 24 ms
microbenchmark({mcparallel(x*x); y = mccollect()[[1]]})

# Dot product, return scalar
############################################################

# 2.3 ms
microbenchmark(y = sum(x*x))

# 7.2 ms
microbenchmark({mcparallel(sum(x*x)); y = mccollect()[[1]]})
```

Issues of latency and bandwidth generally become more complex for different architectures such as distributed systems and GPU's [14].

6.2 Dynamic Execution

Alternatively a dynamic execution model can be used. This relies on a master / worker architecture. The reference version could be a multicore system which forks to evaluate expressions. At a high level, the master runs an event loop that pops expressions from the top of the expression dependency graph. This approach is appealing because it can do dynamic load balancing without needing to know.

Here's an algorithm: Insert the artifical node 0 representing the beginning of the script, so that nodes without parents now have node 0 as a parent. Mark these direct descendants of node 0 as ready. Let the workers begin evaluating these expressions.

1. Event loop checks to see if any are done.
2. Expression e_i finishes and the results are available again on master.

3. For each expression e_j which depends directly on e_i: check if e_j has no other existing parents then mark it as ready.
4. Remove e_i from the graph.
5. Free workers begin executing any of the nodes that are marked as ready.

There's some nuance here by trying to keep each of k workers as busy as possible while still respecting the constraints. Ie. there may be bottlenecks where only one worker can be active, but after that all the others can go.

This algorithm could also be refined into a priority queue by keeping the ready nodes in a heap, with the values determining the heap order as the number of expressions that depend on that expression, directly or indirectly. This is a little naive- it would be better to have timings of the code and do something more optimal in terms of reducing run time.

The process forking every time will be quite inefficient. The more intelligent thing to do is 'pipeline' the operations, and send whole related blocks of expressions to individual processes to evaluate.

7 Challenges

R's flexibility makes code analysis challenging in some cases.

7.1 Reproducibility

Reproducing random streams allows one to perform the exact same random computation. This is useful for investigating ano The R documentation for `parallel::mcparallel` explains how this works in the case of parallel forking:

> The behaviour with 'mc.set.seed = TRUE' is different only if 'RNGkind("L'Ecuyer-CMRG")' has been selected. Then each time a child is forked it is given the next stream (see 'nextRNGStream'). So if you select that generator, set a seed and call 'mc.reset.stream' just before the first use of 'mcparallel' the results of simulations will be reproducible provided the same tasks are given to the first, second, ... forked process.

If a dynamic model is used as described in section 6.2, then reproducing the exact computation may not be possible, since the code may not execute in the same order. However, if this level of reproducibility is important then one can force it through manually seeding the functions that matter. For example:

```
critical_random_func()
```

becomes

```
{set.seed(123); critical_random_func()}
```

7.2 Dynamic Evaluation

Correct, legitimate code can be written that depends on the results of dynamic evaluation. Indeed, some symbols may not currently exist. This is more common inside package code that defines and uses many functions, and less common in scripts. Here's an example:

```
f = function() 0        # 1
g = function() f() + 1  # 2
f = function() 10       # 3
g()                     # 4
```

The last line returns 11, since it uses most recent version of `f()`. An expression dependency graph that does not correctly handle dynamic evaluation here will consist of these edges:

```
1 -> 2, 1 -> 3, 2 -> 4
```

So the statements could be written in the following order, which respects the partial order:

```
f = function() 0        # 1
g = function() f() + 1  # 2
g()                     # 4
f = function() 10       # 3
```

In this case the call to `g()` will incorrectly return 1 instead of 11. Hence there is a "hidden" dependency implicit here: `3 -> 4`. This comes back to lexical scoping rules, since we need to look up the correct `f()`.

One possible way to get around this is to first inline all user defined functions, and then perform the expression dependency analysis. One substitutes the bodies of `g(), f()` so the code presented above becomes simply `10 + 1`. This also has the effect of removing user defined functions from the expression graph.

7.3 Mutable Objects

Environments and reference class objects in R are mutable. They are a special case and must be handled carefully. One way to handle them is to treat any access of one of these objects as a redefinition. This resembles the conservative approach one would have to take for Python or Julia methods, and could result in a similar outcome of having too many constraints.

With environments one could refer to all variables specifically through (environment, symbol) pairs. Through these pairs all variables in environments essentially act like regular variables, and variables in the global environment are just a special case.

7.4 Non Standard Evaluation

The R code `lm(y  x, data = d)` is ambiguous because `x, y` may be global variables or they may be columns in the data frame `d`. CodeDepends doesn't detect the dependency

on variables `y, x` because of non standard evaluation (NSE). For interactive use NSE can be convenient, but it comes at the cost of referential transparency [19]. The programming model in this case is no longer functional, which makes code dependency analysis much more difficult.

As before, the conservative approach is to add edges and dependencies when in doubt. So in the code above one assumes that all of the variables `x, y, d` will be used.

7.5 Side Effects

As mentioned in section 2, it's difficult to recognize the dependence structure with plotting commands. One way to handle this is to use a preprocessing step to collapse all steps modifying the graphics device into one block of code. This can be implemented by scanning the script and adding braces around the lines of code that open and close graphics devices, for example:

```
{   # Added by preprocessor
pdf("plot.pdf")
... # plot(), title(), text(), etc.
dev.off()
}   # Added by preprocessor
```

This block will then be executed together as one top level expression.

8 Conclusion

This work presented an alternative execution model for R code based on dependency analysis.

To get some idea of the potential for speedup, if I have an unlimited number of cores, no overhead, and each expression takes roughly the same amount of time to execute then the script can run no faster than the length of the longest path in the graph. Hence speedup will be the total number of statements divided by the length of the longest path in the graph. By the way, all of those assumptions are unreasonable.

A Definitions

The following material is from the R Language Reference [17].

"A **statement** is a syntactically correct collection of tokens." Less formally one can think of a statement as a single line of code. The 'line' rule doesn't always hold in practice, since a semicolon can put two statements on one line, and a single statement may span multiple lines.

```
# Two statements on line
a = 1; b = 2

# One statement on multiple lines
plot(x,
     y)
```

symbol is a variable name such as **a, b** above. The words symbol, variable, and name are used interchangeably. For consistency I'll stick with symbol.

Assignment is the binding of a symbol to an R object.

"An **expression** contains one or more statements." Expression objects in the langauge contain parsed but unevaluated statements.

Statements can be grouped together using braces to form a **block**. Since expressions can be nested, we can consider a block just a special type of expression.

```
{
a = 1
b = 2
}
```

B CodeDepends

CodeDepends is the underlying package which generates the expression dependency information [11].

The parsing uses an object oriented wrapper around R's builtin `parse()` which handles different file types ie. script or dynamic documents among other arguments. This doesn't directly expose code comments. At first glance it seems CodeDepends uses the tokens more indirectly, through functions like `is.name()`.

The workhorse functions are in `CodeDepends.R`. Overall, the approach resembles `codetools::walkCode` as discussed in [4]. Essentially it walks the parse tree, calling a function to collect usage information.

When analyzing a single expression, first a collector object is created with `inputCollector()`. This is a closure that maintains a list of everything in the expression that has been seen so far: files, variables, function calls, etc. It returns a list of functions to update the data in the closure.

`getInputs.language()` takes a collector object and recurses through expressions until it finds the leaf nodes which can be functions, calls, assignments, names, literals, or pairlists. Upon finding one of these leaf nodes it calls the collector object. Many special cases of functions are handled in functionHandlers.R, such as `$, rm, for` as well as non standard evaluation. Special attention seems to have been paid to dplyr operations.

C LLVM

LLVM has some functionality for working with these use / def chains [13]. For example: `http://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains`. Following their terminology, an object of class `Value` is used by potentially many objects of class `User`. The *def-use* chain is the list of all `Users` for a particular `Value`. Think of this as all the places in the code where this variable propagates. In contrast, the *use-def* chain is the list of all `Values` for a particular `User`. This is all the inputs to a newly created object. There's room for both of these chains to be expanded recursively.

In this example the def-use chain for `x` is only `[y]`, but the recursive one is `[y, z, z2]`. The use-def chain for `y` is `[z, z2]`.

```
x = 10
y = x + 5
z = y + 2
z2 = y + 100
```

We might be able to use this along with R code to determine if an R function calling C code is pure.

References

[1] Vikram S Adve and Mary K Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Transactions on Computer Systems (TOCS)*, 22(1):94–136, 2004.

[2] Henrik Bengtsson. *future: A Future API for R.* R package version 1.2.0-9000.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *arXiv preprint arXiv:1411.1607*, 2014.

[4] John M Chambers. *Extending R.* CRC Press, 2016.

[5] Winston Chang and Javier Luraschi. *profvis: Interactive Visualizations for Profiling R Code*, 2017. R package version 0.3.3.

[6] Michel Cosnard and Michel Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122. IEEE, 1995.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[8] Milind Girkar and Constantine D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, 1992.

[9] Jim Hester. *covr: Test Coverage for Packages*, 2017. R package version 2.2.2.

[10] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163–179, 1978.

[11] Duncan Temple Lang, Roger Peng, Deborah Nolan, and Gabriel Becker. *CodeDepends: Analysis of R code for reproducible research and code comprehension*. R package version 0.4-2.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[13] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[14] N. Matloff. *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*. Chapman & Hall/CRC The R Series. CRC Press, 2015.

[15] Vivek Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5.6):779–804, 1991.

[16] Markus Schmidberger, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, and Ulrich Mansmann. State-of-the-art in parallel computing with r. *Journal of Statistical Software*, 47(1), 2009.

[17] R Core Team. R language reference, 2016.

[18] Luke Tierney and Riad Jarjour. *proftools: Profile Output Processing Tools for R*, 2016. R package version 0.99-2.

[19] H. Wickham. *Advanced R*. Chapman & Hall/CRC The R Series. CRC Press, 2015.

[20] Yihui Xie. *knitr: A General-Purpose Package for Dynamic Report Generation in R*, 2016. R package version 1.15.1.