Code Analysis Project Proposal

Clark Fitzgerald

February 24, 2017

Abstract

1 Introduction

The idea behind code analysis is to treat the code itself as a data structure. This is an old idea, the R language reference cites the Lisp language as the inspiration.

The interpreter works in a simple way. The `Rscript` command will go through a file line by line and parses then evaluates each statement. R's `source()` function on the other hand first parses the whole thing, and then evaluates it all. We propose to first parse and then examine whole scripts at a time, from just a couple lines to a few hundred lines of code. The whole script is parsed into a directed graph data structure. Nodes contain the expressions and the edges contain the dependence information. TODO: but what about control flow? loops?

Big point- Compilers have used all manners of static analysis and intermediate optimizations to create more efficient code. Interpreted languages are much more limited. But why not build refactoring tools that can improve the performance?

Why this matters. 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.

2 Dependencies

We need to start out with some definitions and basic concepts to make all of this precise. The following material is from the R Language Reference [10].

"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(lm(y ~ x,
        data = xydata))
```

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. Complex assignment

"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
}
```

TODO: Not sure how to handle blocks. One option is to go inside them and look at all the individual statements. The other option, which seems easier at first, is to do what CodeDepends does and group them into one single logical unit. Presumably if a programmer writes a block like this they intend the whole thing to execute together. One typically sees blocks along with `if` statements.

But in the end I think I'll have to recurse inside the expressions at the script level, but not down into the supporting library R code. For example, to get a minimum set of code to perform a final operation one would need to go into a block.

So how do we detect dependencies in code?

Idea talking with math students- count the incoming and outgoing edges. Use this info.

3 Graph Construction

The graphs can be built sequentially, as the script is parsed.

The nodes in the graph are code expressions. An expression may do any combination of the following things:

1. Define new symbols
2. Use existing symbols
3. Redefine existing symbols

Consider the following simple script:

```
x <- 1:10   # expression 1
plot(x)     # expression 2
x <- 1:5    # expression 3
plot(x)     # expression 4
```

The first expression defines the new symbol `x`. The third expression redefines `x` without making use of the existing definition.

The second expression uses `x`, so we define an edge from node 1 to 2. More generally we define an edge from the most recent expression that updated `x`. Therefore the fourth expression creates an edge from 3 to 4.

The complete graph is then:

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

But how will we know if the symbol exists? After parsing we could potentially see the timelines recording which symbols live, and if they are ever removed. This may depend on conditional statements, which we can't know. The safe thing then seems to just assume that they will be used.

TODO: Is this just for the global namespace? How will it be different for others? I think it's ok to just focus on the globals first. Environments in general may require more thought.

"R adheres to a set of rules that are called lexical scope. This means the variable bindings in effect at the time the expression was created are used to provide values for any unbound symbols in the expression." [10] A primary goal is to respect R's lexical scoping rules.

Let k be the number of cores on a machine. To accelerate code by running in parallel 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. Additionaly, each expression should take long enough that it's worth the overhead of using an additional R process to evaluate it. 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()
```

Side note- where exactly is the overhead of using multiple processes? Is there latency? There's certainly a limit to how much data we move.

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 multiprocessing can't help.

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

4 Code graph and parse tree

The code graph and related data structures are related to the parsed script, which we'll refer to as the **parse tree**. Since the parser doesn't care about spaces, tabs, and comments, different scripts can produce the same parse tree. R's `getParseData()` preserves all of the source information, and this can be very helpful for debugging or IDE's. Later this might be useful for injecting code. For our immediate purposes scripts differing only in comments and white space can be considered the same, which gives a 1 to 1 relationship between scripts and parse trees.

Many parse trees can give rise to the same code graph. For example, the code 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
```

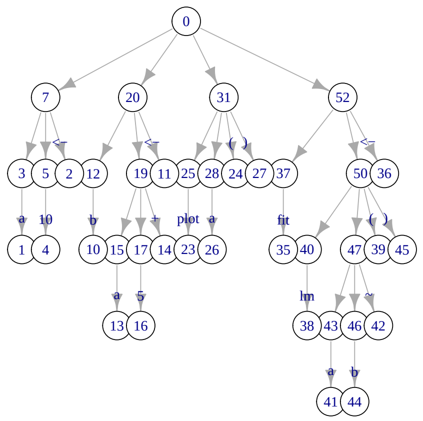
Therefore we lose information by converting from a parse tree to this code graph.

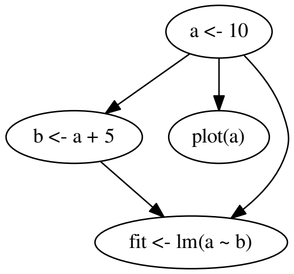
Listing 1: Simple script

```
a <- 10
b <- a + 5
plot(a)
fit <- lm(a ~ b)
```

For visualizing the control flow and dependencies it's important to have the entire expression available in the node as in Figure 1b. This lets users easily connect the graph with the source file. Another thought that would make this easier- put the source code line number along with the line of code.

Duncan was pointing out how I don't need the edge connecting the first line to the last line in Figure 1b, because it's implied by the path through the intermediate node. True. So I could go through and remove those edges. How to do that? Can a dominator tree help with this? It's nto a


(a) Parse tree



(b) Expression dependency graph

Figure 1: Different representations of the script in Listing 1

This would then help me see where the parallelism is, and put it in an appropriate data structure. Once I do that I can collapse simple threads consisting of chains of nodes with single inputs and outputs down into something like this:

Looking at some of Nicks stuff- what I'm really thinking about is breaking up the basic blocks into expressions and looking at relationships between them, with parallelism as one potential end goal. He's looking more carefully at the control flow implied by how the user wrote the code.

```
A may have many incoming edges, C may have many outgoing edges.

Then:
A -> B -> C   becomes a single node ABC
```

This will preserve the semantics and the ideal parallelism. Proof? It also leaves a data structure that's as simple as it can be, which is what I'd want to use when actually modifying the source code.

But the task graph is not the same as control flow. Control flow is things like for, while, return, etc. The task graph is a higher level construct. Nodes are R expressions that appear at the top level of a script, which means all control flow is captured *inside* a node. Create an edge from node A to B if B uses a variable that was most recently updated in A. There is some ambiguiity here, since variables may be created within conditional statements. In that case we can take the conservative approach of just creating all the edges. An example:

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

```
if(coinflip()) x <- 20
if(coinflip()) x <- 30
y <- x + 1
```

But is the task graph a DAG? Yes, I think it has to be, unless you're doing something totally weird. The reason is that R scripts execute sequentially, they don't have a GOTO statement. It's true that loops can repeat the same lines of code many times, but `for(){...}` and `while(){...}` are just single top level expressions- we're not expanding them out in the graph n times. This is an argument in favor of not recursing into these expressions- if we do then it opens up many problems.

One could make the argument that calling `source()` on some script a couple different times in a program can make for repeated code which causes trouble with the DAG. One way of handling this is to go read the contents of the file being sourced, then consider this statement just like any other- taking in variables and producing them.

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

Variable updates don't need to be DAGS- but then again it doesn't make perfect sense to talk about control flow in terms of variables.

5 Task Graph Literature Review

Most of these highly cited papers seem to focus on making systems faster and more efficient. What about the value of looking at a task graph for educational / explanation purposes?

Ferrante et. al. introduce the program dependence graph. [3]. The approach I'm looking at now should be higher level than this, because this explicitly handles control flow. I can get out of this by only looking at the top level statements in a script. The control flow there is super simple- all the statements execute in order!

TODO: see which graph definitions we can borrow from this and other papers- ie. "hammock".

[2] describe constructing a task graph based on annotating the source code the program. They go into every iteration of the loop, which differes from what I'm proposing.

[1] presents a model for predicting the run time of programs based on a task graph. Their definitions on page 98 are nice and clear, I should use them. Based on the graphs of their figures the programs seem very like very regular, well defined algorithmic programs. A general data analysis script in R is probably much more variable than this.

This is one way to distinguish and justify my work here as different from CS- general forms of scripting and data analysis are usually quite different than a pure focus on algorithms.

Nice explanation of program dependence graph (PDG):

> 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. [8]

We can borrow their idea of distinguishing ideal parallelism from useful parallelism. Overhead makes these two different.

The hierarchical task graph was introduced in [4] for the purpose of detecting 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.

As of 2009, most efforts to parallelize R have focused on the lower level programming mechanisms. These needed to be in place before any higher level automatic detection could be built and function. [9]

What I'm looking at is very close to the Use-define chain. `https://en.wikipedia.org/wiki/Use-define_chain`. But I'm staying just slightly higher level. The hope is that staying at a high level will make this more productive / easier to reason about and use than a low level thing that examines every bit of control flow.

The Use-definition chain has been around since at least 1978 when it was used to remove dead (unused) code[6]. This reference has a nice consideration of what makes a computation useful or not. It defines it recursively- something is useful if it's used later by another computation. This is combined with the "base case" of usefulness- there's a set of operations that are 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 R expressions as intrinsically useful if they have a side effect, ie. any plotting commands, saving data to disk, etc.

So I wonder how much this has to be tied to R? CodeDepends is of course purely for R. But other tools exist to manipulate parse trees independently of any language, for example `https://github.com/tree-sitter/tree-sitter`. Differences among languages... it would have to recognize that a Python methods often mutate their objects, for example:

```
x = [3, 1, 2]
x.sort()        # Sorts x in place, returning None
```

LLVM has some functionality for working with these use / def chains [7]. 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`. I 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.

6 How CodeDepends works

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 ??. Essentially it walks the tree of code, 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.

As a user I'd like a clean, well defined entry point and API for this stuff. The problem I had today was trying to pass arguments into `inputCollector`. I did this by hacking through the code of `makeTaskGraph`.

TODO: Ask Duncan where the separation between CodeDepends and CodeAnalysis should be. How about makeTaskGraph? Probably first I should just look more in CodeAnalysis to see what's there.

7 Related Work

Jim Hester's covr package [5] 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`

So if I'm going to programmatically modify the source code then this should really respect things like comments and whitespace. This could be more difficult than it appears on the surface.

TODO Gives me an idea... a similar strategy might work to collect timings and resource usage for functions called with various data sizes. This could be implemented with something like `System.time()`. We could attach timings to each expression. Then we can use this info to programmatically modify the code: ie. if $n > 1000$ then run a multicore version. This is something like Profile Guided Optimization (PGO). This technique is now being used for Python. Could R use it too?

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. After all, we have easy access to all the machine learning type things that we need. We can use it to automatically tune. But this implies that there is some way to tune it. Right now the only "trick" I have up my sleeve is to try to parallelize it.

The vignette in Tierney's `proftools` package has some nice examples of visualizing profiling data. The call graphs and related visualizations are conceptually similar to what I have in mind. Wickham's `profvis` integrates with the IDE to indicate the actual line of source code along with the related timing info.

Knitr (TODO cite) facilitates reproducible computations for chunks of code in Rmarkdown documents. 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 you can? I wonder how difficult it would be to hook into knitr's system... And while I'm at it, why not Jupyter Notebooks? If I can figure out how to build an equivalent dependency graph for Python that would be great.

8 Applications

What are all the useful things to do when rewriting code? Duncan's very helpful docs here http://www.omegahat.net/CodeDepends/design.pdf give a handful of reasons.

Other things that I can think of:

Identification and elimination of dead code.

- Given a big ugly script, pull out the minimal set of code to produce a final result. For example, everything needed to make ‘plot(finalresult)‘. - Parse Rhistory to find the minimal set of code to reproduce a result. - Separate single scripts into multiple scripts if they actually have two independent sequences of computation.

These ideas stay much closer to the syntax and language of R compared to what Nick is doing.

I’ve been considering this for the purpose of analyzing, accelerating and optimizing essentially single scripts. But how important is this generally? It only matters if someone’s script runs slow. Probably this mostly happens due to one large computation. If they’re doing two large and independent computations in one script shouldn’t that be two scripts? But these tools should be able to split them up at that level if I ask them to.

Remove all objects as soon as they’re finished being used in the script. If creating many large intermediate objects this could help with not using excessive memory.

9 Detecting Parallelism

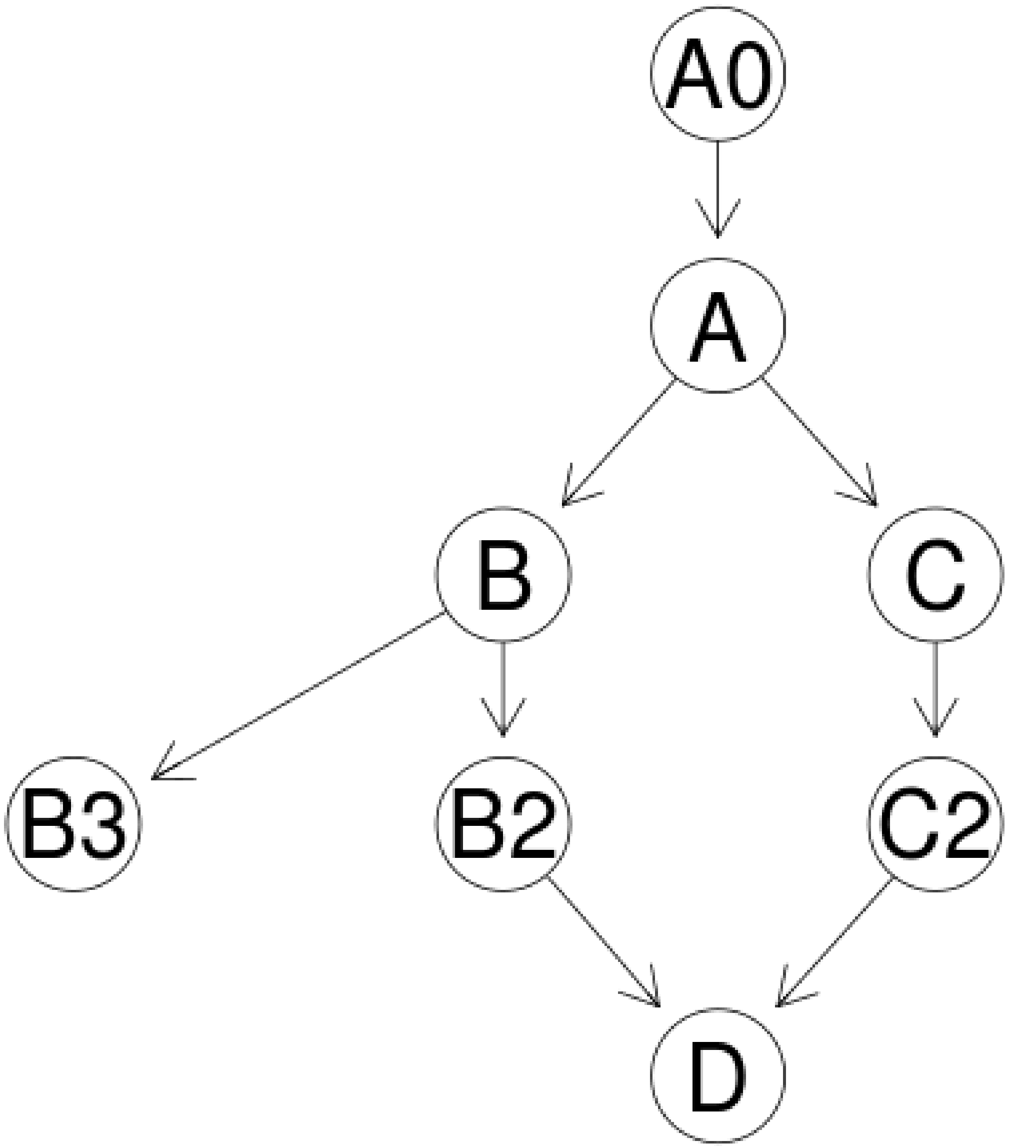
Not sure how to generally detect where the possibilities for parallel are. In a simple case of k obvious threads with one common ancestor and common child there is an obvious option:

Remove the common ancestor and child and select the k disjoint subgraphs between them.

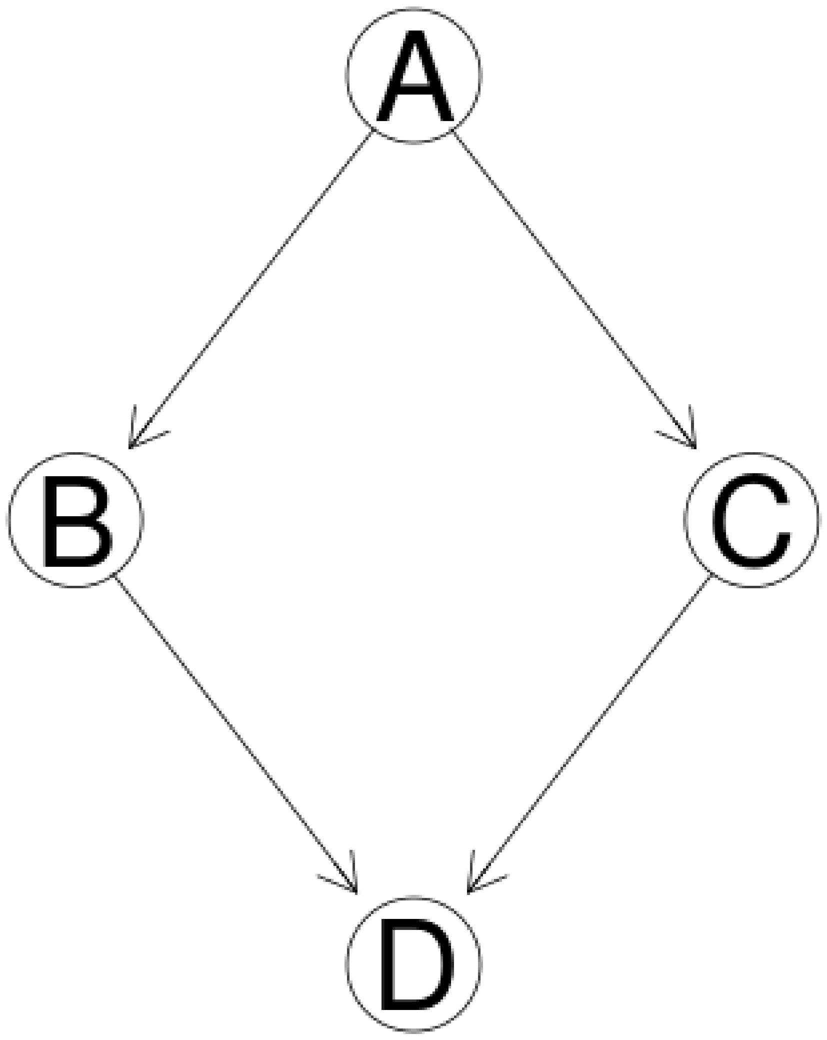
Alternatively:

1. Start at the top of the graph, and move down until one node has multiple children. Gather those child threads into 2 groups. 2. Same thing starting at the bottom.

The most straightforward thing to do right now is take a more complex graph, say something like this:



And attempt to collapse the structure into one where the parallel threads become obvious:



The essential part for the multiprocessing is to have many "adjacent" threads (is there a correct graph term for this?) Optionally the adjacent threads can share a common parent and or child nodes.

10 The right data structure

I want to work with a DAG constraining the order in which the statements must execute. This DAG can initially be built sequentially with the following algorithm:

```
for statement in allstatements
```

```
    for input variable x in statement
        create an edge from the most recently seen statement which outputs x
```

It's not sufficient to just draw a single edge for the most recently used variable. This will make a tree, but then it won't necessarily contain all the dependency info.

But why am I interested in this DAG? Maybe more interesting is the partial ordering that comes from it. And the threads. So maybe I need to think about how to go from the DAG to the partial ordering. Wikipedia article for partially ordered sets cites the set of vertices of a DAG ordered by reachability as an example.

TODO: Can this be a tree? Yes. Is it always a tree? I don't think so... Maybe once extraneous stuff is trimmed off?

If the intermediate variable ‘b‘ is not used anywhere we probably want to collapse the following into a single block of code, since it has to happen sequentially. Therefore adding ‘future‘ here can't help at all. Unless it should happen at the same time as another block... then two blocks can execute simulataneously.

```
# Begin block
a = 100
b = a + 5
c = sum(b)
# end block
```

11 Hard things

‘fit = lm(y x)‘ doesn't detect the dependency of ‘fit‘ on ‘x, y‘.

Following the control flow for ‘lm‘ we see the following calls: ‘model.matrix.lm‘, ‘model.frame.default‘, ‘as.formula‘. But eventually this ‘y x‘ itself must be evaluated in which case ‘.Primitive(" ")‘ is called, which certainly does something weird. But what?

Haven't yet checked things like: ‘¡¡-, assign‘

Recursion? Iterating updates?

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] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

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