1 Introduction

Part of the movitation for this is to be able to deal with cached computations, often in the context of dynamically generated documents (Sweave and friends or XDynDocs), or for interactive dynamic documents which are rendered in R via an embedded browser widget.

An additional motivation is being able to summarize computations at a high-level to aid understanding algorithms, scripts and the computational process. We want to view code at a higher level, thinking more abstractly about the tasks rather than looking at the details of the code. This is, of course, where good software design comes into play. But when writing scripts, software design is not necessarily the focus of the author. So we are enabling the author to edit a script after it is created to annotate it at a higher level with labels for the blocks. We can also do this somewhat programmatically, albeit imperfectly.

We are also advocating the notion of programmatically manipulating scripts as objects that we can query and even modify and add code to.

2 Scripts

We are dealing with scripts, i.e. an ordered collection of R expressions. These might be either individual expressions or blocks of related code. We introduce two classes: Script and AnnotatedScript. Script is the general version; AnnotatedScript is a derived or sub-class. These are just lists of the expressions within the script. A Script object also has its location, i.e. the file/URL from which it was read.

Elements of a Script are of class ScriptNode.

There are several types of possible inputs for a script, i.e. formats of source files. We are considering simple R scripts, tangled code extracted from an Sweave document, code extracted from an XML document (i.e. Rdocbook from XDynDocs). We also introduce the notion of an "annotated R script" which consists of code blocks each within expression braces (i.e. ...) and labeled with elements of a vocabulary that identify the high-level nature of the task that this code is doing. For example, initialization, dataInput, simulation, EDA (exploratory data analysis), diagnostics, graphics, dataOutput. These are intended to help the reader understand the high-level nature of the code without focusing on the details of the actual expressions. The task identifiers/labels are added in the form

```
initialization({
      library(codetools)
      library(CodeDepends)
})
plot(model({
     fit = lm(y ~ ., data)
     plot(fit)
   }))
```

One can even provide names for the elements of the script via

```
mode.graphics = plot(model({
     fit = lm(y ~ ., data)
     plot(fit)
   }))
```

These can then be used to conveniently refer to a step in the script, e.g. to run just that node, to run the code up to that node, or to refer to a variable created within that step.

2.1 Reading Scripts

Scripts are best read with the function **readScript()**. We can specify the *type* of the script or the function will attempt to guess the type. The possible types are "xml", "R", "labeled" and "Stangled".

We can specify the location as a file name, a connection (e.g. a URL) or can specify the content of the script as a character vector via the *txt* parameter.

3 ScriptNode Meta-Data

getInputs is used to analyze a script or a node in a script to compute the meta-information about it, i.e the input and output variables, the functions used, the files and libraries referenced.

```
f = system.file("samples", "namedAnnotatedScript.R", package = "CodeDepends")
sc = readScript(f, "labeled")
names(sc)
getInputs(sc)
getInputs(sc[[1]])
```

Instead of using getInputs, we can use coercion as in

```
as(sc, "ScriptInfo")
```

We can find the output/generated variables associated with the entire script

```
getVariables(sc)
```

4 Visualizing Relationships between Variables

The meta-information allows us to determine which variables are inputs to defining others (in terms of inputs and outputs to blocks). We can also visualize these via a graph with nodes displaying the variables and edges illustrating which variables are inputs to others. The function **makeVariableGraph()** is used for this. We can then plot this using the *Rgraphviz* package.

```
f = system.file("samples", "results-multi.R", package = "CodeDepends")
g = makeVariableGraph(f)
library(Rgraphviz)
plot(g)
```

5 Visualizing Relationships between Tasks

In addition to looking at individual variables, we can visualize the relationships between tasks. We can see how the outputs of one task are fed into other tasks.

```
f = system.file("samples", "disjoint.R", package = "CodeDepends")
g = makeTaskGraph(f)
library(Rgraphviz)
plot(g)
```

When dealing with simple calls rather than code blocks containing multiple calls, the task graph and the variable graph will be very similar.

We can use the SVGAnnotation to make this an interactive, dynamic SVG plot.

6 Variable Lifetimes and Garbage Collection

When we call functions, the intermediate or local variables used by the function's body are garbage collected when the function call is completed. In scripts, however, we often leave top-level intermediate variables in the global environment even when they are no longer of use in the remainder of the script. With a little code analysis, for a given variable, we can find the last expression in the script and add code to remove it and so allow the value to be garbage collected, potentially freeing significant amounts of memory.

The function **findWhenUnneeded()** determines the index of the code block at which a variable is no longer used. We can then add an expression to the end of this code block of the form

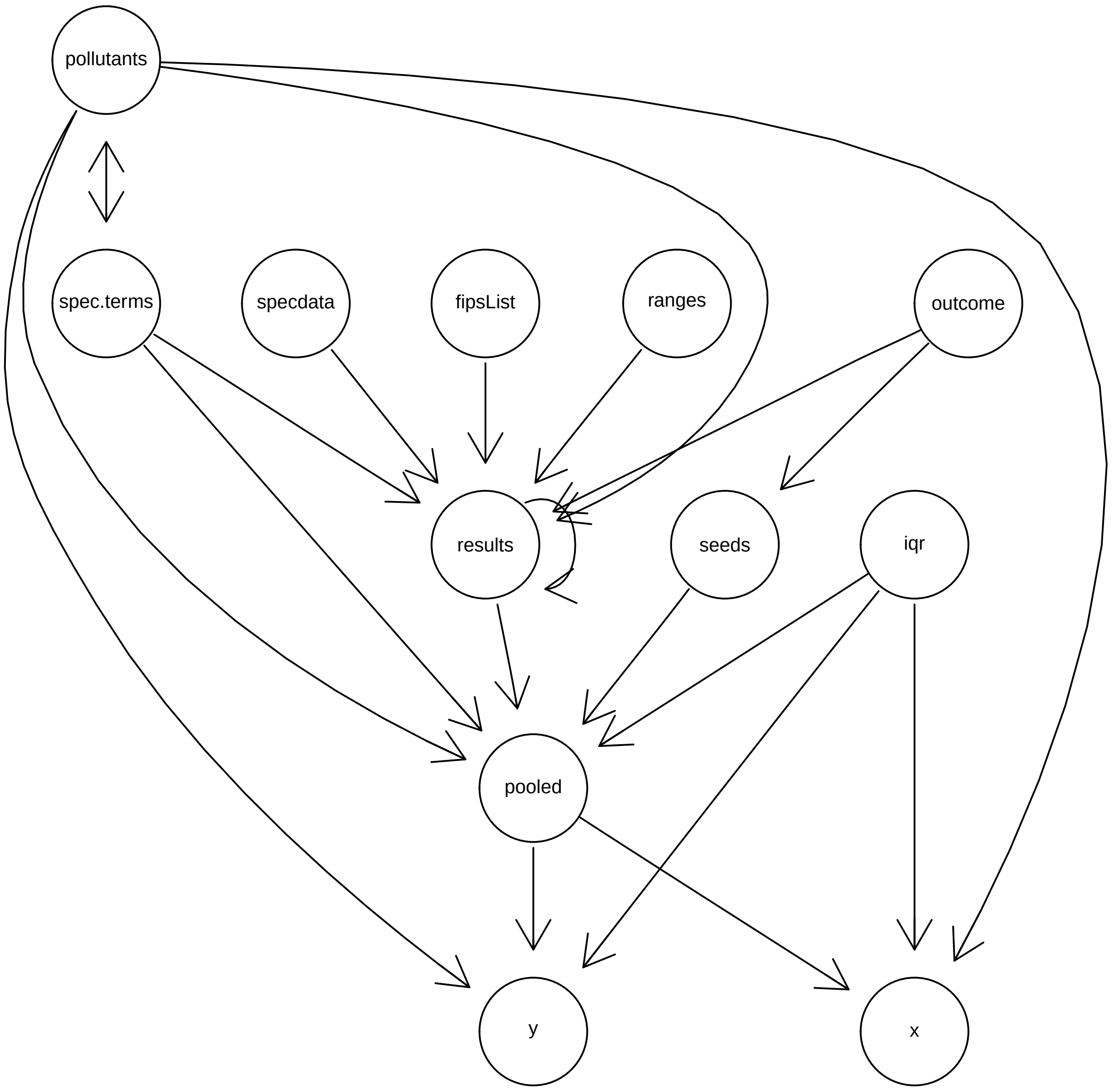

Figure 1: Relationship between variables in `results-multi.R`

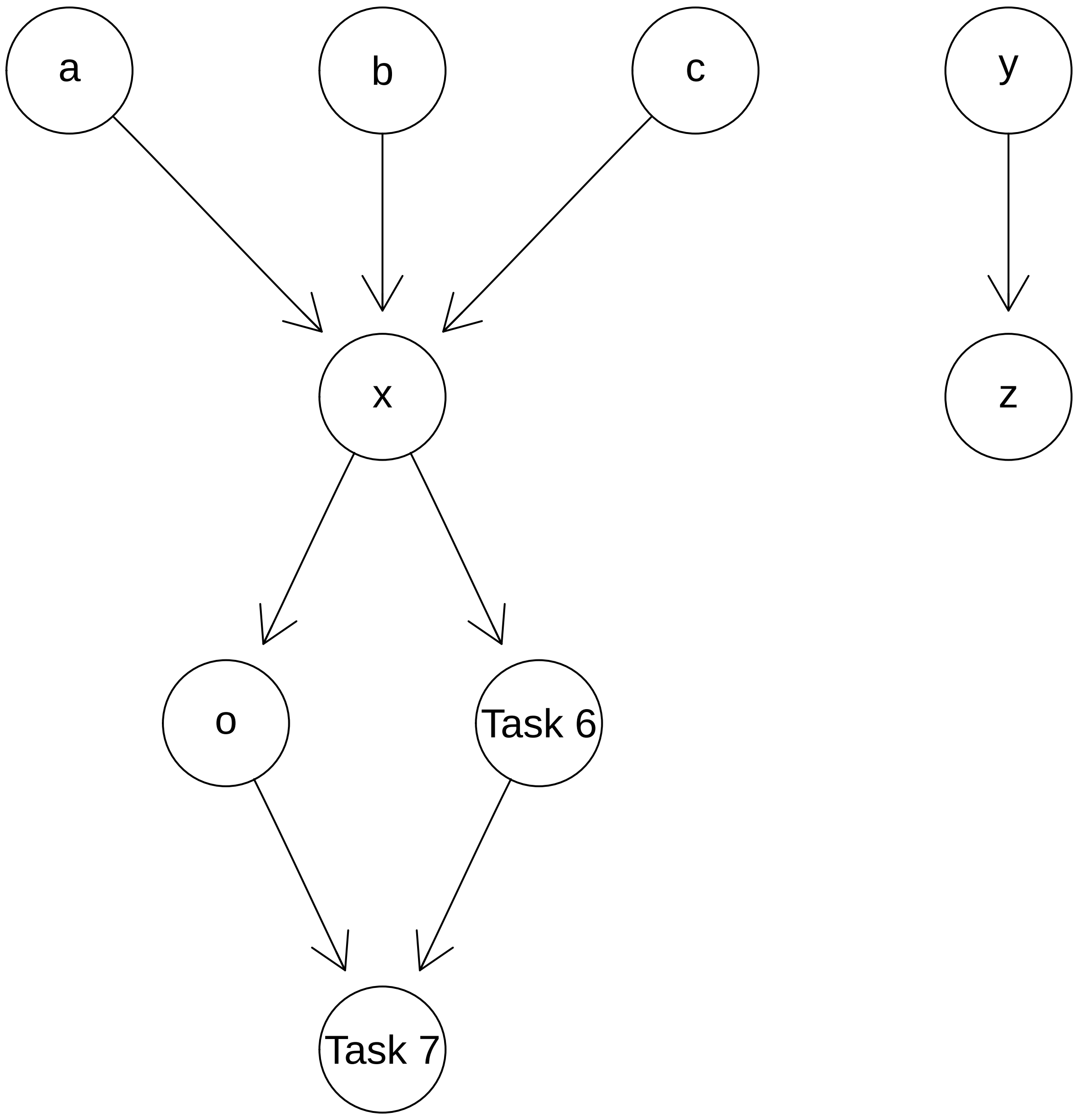

Figure 2: Relationship between tasks in `disjoint.R`

```
rm(varName)
```

We can visualize the life cycles and the points at which the different variables are used and defined in the code using **getDetailedTimelines()**. This allows us to see if variables should deleted or the order of the computations changed to avoid holding variables in memory for long periods over which they are not used. An example is

```
f = system.file("samples", "results-multi.R", package = "CodeDepends")
sc = readScript(f)
dtm = getDetailedTimelines(sc)
plot(dtm, main = "Variable time-line for results-multi.R")
```

The result is

6.1 Functions

We can also use the tools here for functions. For example, let's find the "length" of each function in the search path where "length" means the number of top-level expressions in the body.

```
z = sapply(search(),
            function(x) {
             objs = objects(x)
             if(length(objs))
               structure(
                  sapply(objs, function(o) {
                               tmp = get(o, x)
                               if(is.function(tmp))
                                  length(body(tmp))
                               else
                                  0
                            }),
                  names = objs)
             else
                integer()
            })

names(z) = gsub("^package:", "", names(z))

sizes = data.frame(size = unlist(z),
                   package = rep(names(z), sapply(z,length)))
```

Let's visualize this data:

```
library(lattice)
densityplot(~size | package, sizes, plot.points = FALSE)
densityplot(~size, sizes, groups = package,
             auto.key = list(columns = 5), plot.points = FALSE)
```

We can then look at some of the big functions and summarize and visualize these.

```
names(which.max(z[["utils"]]))
```

yielding **read.DIF()**. Then we can look at the variables (including the function's parameters)

```
info = getInputs(read.DIF)
table(getVariables(info))
```

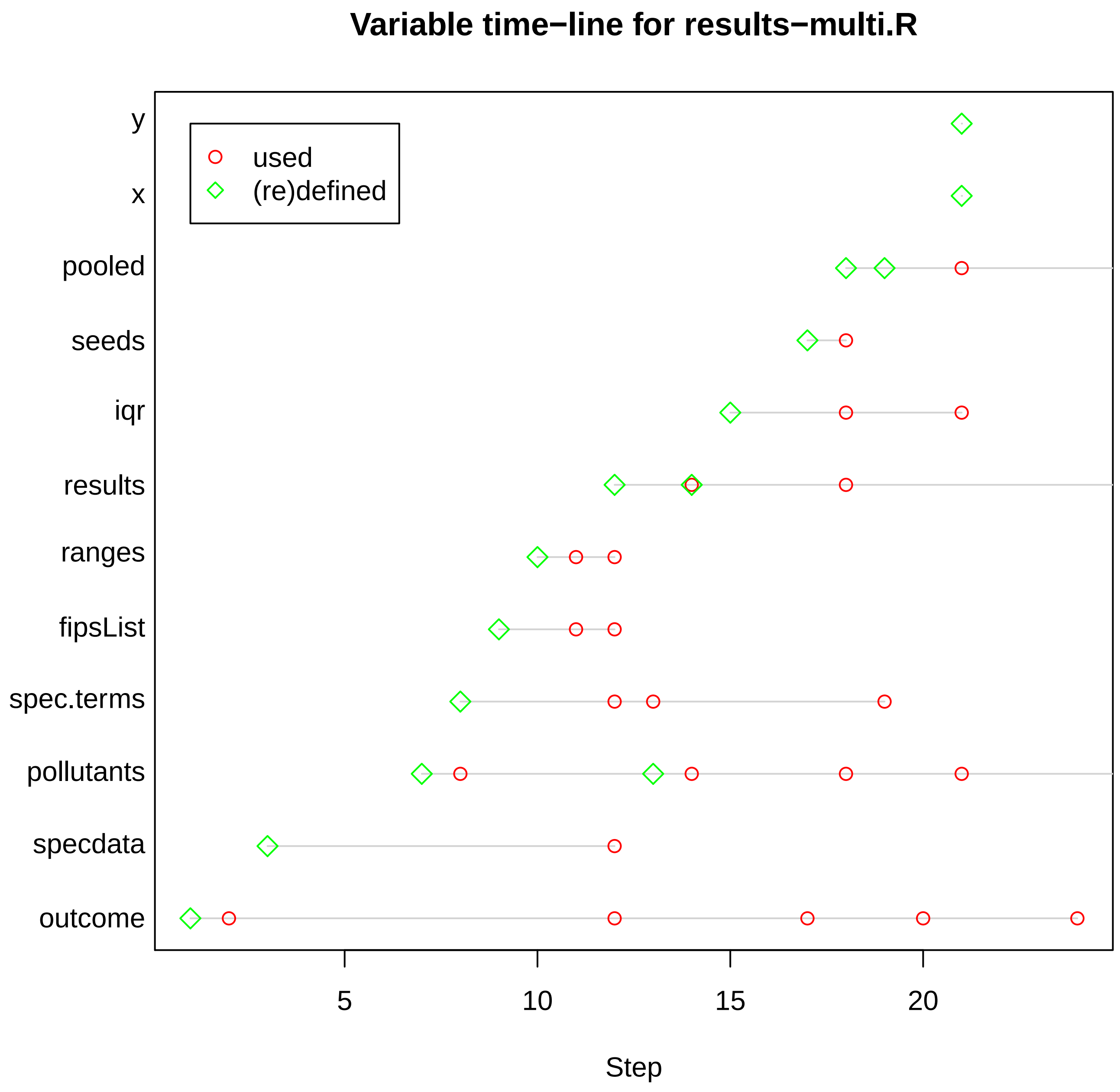

Figure 3: Time-line of variables in the `results-multi.R` script

and also create a graph

```
makeVariableGraph(info = info)
plot(g)
```

Since we cannot use the annotation scheme for functions, it is useful to try to guess the tasks using **guessTaskType()** and to try to be able to group expressions into blocks.

7 Function Call Graph

This package allows us to compute the call graph between a set of functions. We can specify a (named) list of functions or a package or a single function (by name). Methods then compute the call graph. We can the visualize this using *Rgraphviz*.

```
g = makeCallGraph("CodeDepends")
library(Rgraphviz)
plot(g, "circo")
```

8 Avoiding Redundant Computations

Some scripts have redundant computations and being able to identify these can save time and memory. Clearly, computations that create variables that are not used subsequently are typically not necessary. (There may be side effects such as creating plots, files, opening connections, etc.) But, as in compilation, we might also identify computations that are done several times that might be more efficiently done once and assigned to an intermediate variable.

9 Identifying Potential Parallelism

Many scripts follow a strictly sequential path where outputs from one block constitute the inputs to the next block. However, there are many which perform one task and then another separate task and then use the outputs from these two in the next task. The fact that these two initial tasks are “separate” means they can be run in parallel.

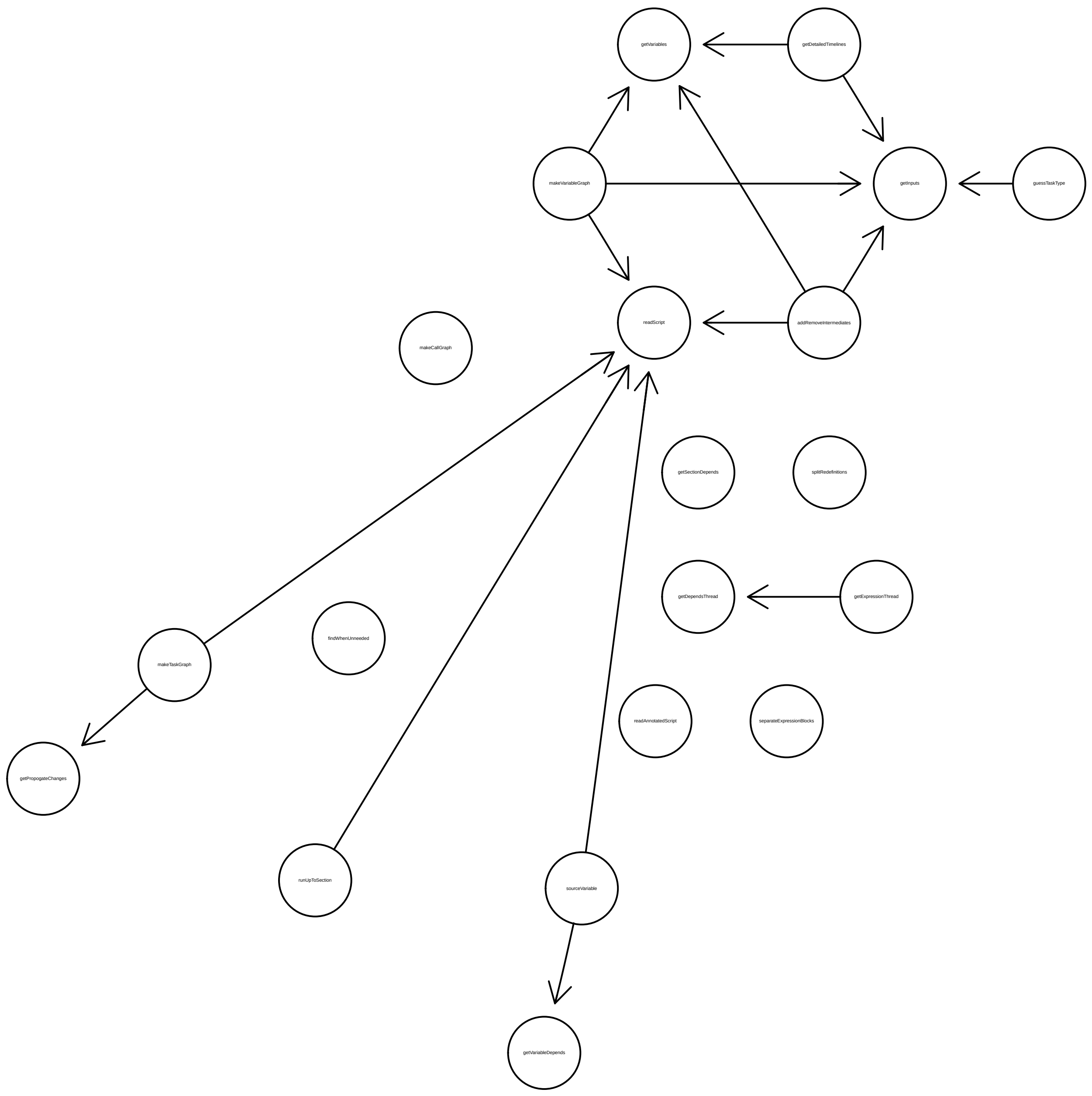

Figure 4: Function call graph for the CodeDepends package.