Master's thesis

Université de Lille

Static debloating of R applications: a case study

Sophie Kaleba

Supervisors: Francesco Zappa Nardelli, Giuseppe Lipari
August 2019

Acknowledgements

First, I would like to thank Francesco Zappa Nardelli, my advisor during this project, for all his advice and wise words. Likewise, I would like to thank Jan Vitek for his advice and just-as-much wise words throughout the whole project. Thanks also to Giuseppe Lipari for his remote supervision from Lille.

The tracer would not trace as well if it was not for Aviral Goel, who relentlessly pair-programmed with me and shared precious knowledge about R in general.

I do like just-in-time compilation, and was lucky to have a look under the hood of Ř, a JIT compiler for R: I thank Guido Chari and Olivier Flückiger for their explanations and their patience.

Contents

Introduction

The last fifteen years have seen significant advances in our ability to provide guarantees about the behavior of code written in systems programming languages such as C and C++. Ongoing research on debloating software leverages binary code analysis to reduce the attack surface of that code; this is achieved by removing unused, or rarely used, subsystems and getting rid of layers of abstraction.

Building a call graph is a common way to address this issue: it identifies functions that are reachable within an application, therefore identifying the core functions that are to be part of a reduced, yet executable, application. The *executable* constraint requires a certain level of soundness, easily provided by static analysis: a static call graph can provide an over-approximation representative of every run. However, the *reduced size* constraint requires that the over-approximation should be limited. In this project, our aim is to reduce applications which have mainly been left out of the scope of static analysis for now, namely applications written in dynamic languages. Indeed, statically reducing dynamic applications is challenging: how to correctly approximate runtime values ahead of time? We will use R applications as a starting point on this question: R is a dynamic language widely used in data analysis. Moreover, it presents some of the most perplexing semantics, mixing at the same time laziness, reflectivity, and functional and object-oriented principles.

The main contributions of this report are:

- The identification of R dynamic features that could compromise the building of a debloated static call graph; two categories of features have been identified: the ones affecting the lookup and the ones deferring evaluation.
- A study of the representation of these dynamic features in real R applications; to do so, we have first adapted an existing tracer and then traced a thousand of R packages. The results show that some of these behaviors, notably through the use of `assign`, are to be found in R packages.
- The identification of hints to adapt the call graph algorithm to address

the presence these dynamic features

Chapter 1 deals with the different ways of reducing applications: it notably presents different call graph algorithms that have been designed for dynamic applications. Chapter 2 focuses on R: we present the language and its dynamic aspects that make static call graph building more complex. We then check in Chapter 3 the real usage of these dynamic features by tracing their occurrences in 1000 R applications. Hints for adapting the call graph algorithm and for the next steps of this project are described in Chapter 4.

Chapter 1

Debloating dynamic applications

It is well known that, as time passes, applications get larger and larger, affecting their overall performance and maintenance cost. Over the years legacy code gets buried under new layers of abstractions, and it gets harder and harder to identify what remains relevant and what is not. Most importantly, code, used or even unused, may reference external libraries, which must be linked in the final binary, enlarging the attack surface [25].

How to *debloat* binaries, e.g. identifying and removing dead code and useless library dependencies, is thus an active area of research. In this report I will investigate the new problems that arise when debloating techniques are applied to applications written in dynamic languages. These feature reflection and ad-hoc environment management, which, as we shall see, make the problem harder than in their static languages counterpart.

1.1 What is bloat?

The term bloat covers all unused and unecessary components part of an application and its dependencies. Xu et al. refer to it as a "general situation where redundancy exists toward finishing a task, which could have been achieved more efficiently" [38].

Bloat can appear dynamically: Jiang et al., in the RedDroid project [15], distinguish between *compile-time bloat* and *install-time bloat*. The first category refers to bloat causing extra time spent to compile unused dependencies of an executable and the second refers to bloat resulting in redundant configuration files that are needed by an application to be platform-independent, although only one will be used at install-time. Xu et al. divide bloat into two categories; *memory bloat* refering to bloat that causes space inefficiencies, and *runtime bloat* refering to bloat that causes the execution of unecessary operations [38].

It is also possible to classify bloat syntactically, as follows:

Dead code. It corresponds to the operations that have been computed but whose result is not being used in the application (e.g. the result of an addition that is stored but not used).

Unreachable code. It corresponds to the statements or instructions that are not reachable from the main method or entry points of the application (e.g. a defined function not being called).

Repeated code. It corresponds to statements or instructions that are being repeated throughout the application.

Length of code. It refers to the overall number of characters being used in the whole application (e.g. symbol names and spacing of code).

Number of generated assembly instructions. It corresponds to the total number of assembly instructions representing the application.

The term *debloating* therefore means to reduce the bloat that is part of an application, including its dependencies.

1.2 How to debloat?

A lot of debloating tools have already been developed; they either address one specific kind of bloat or attempt at reducing several kinds of bloat at once.

Debloating one kind of bloat. Unreachable code represents a significant part of the bloat, therefore several debloating tools focus on pruning away the unreachable code. Quach et al. [25] have developed a dedicated compiler and loader on top of the LLVM framework to debloat C and C++ applications. During the compilation phase, a call graph is built using points-to analysis (see CFA in Section 1.4.1); it identifies the reachable functions of the application. This graph is stored in a dedicated section of the executable file and the loader only loads in memory the functions that are part of this call graph. Anon [2][1] produce an reduced heap snapshot of a Java application. First, they similarly identify the reachable classes, methods and fields through points-to analysis, and build a graph of all reachable objects (the heap snapshot) out of it. Then, they initialize the objects and propagate the newly obtained type information in the graph previously built. They iterate over this process until a fixpoint is reached. The RedDroid tool has been operating in the same fashion, building a call graph to debloat unreachable code. However, it automatically takes reflection into account via string analysis. It also handles call-backs and reduces install-time bloat by relying on user-provided configuration [15].

Some tools provide different approaches to tackle the other kinds of bloat. The elimination of repeated code has been targeted by Fraser et al. in their earlier works about code compression [9]. They identify via a suffix tree which parts of the assembly code are being repeated over the whole application. These repeated code instructions are then abstracted away into subroutines if they are of significant lenght. A similar approach named

[1]The article is to be published

outlining has been introduced as an optimization pass in LLVM in 2016[2]. Regarding minification bloat, Crockford developed JSMin, a minification tool for Javascript applications [6]. It removes comments and white-spaces to reduce the size of the application. The YUI compressor shrinks Javascript application further by reducing variables names [17].

Debloating multiple bloats. Other tools combine several debloating approaches to tackle different kinds of bloat at once for a single application.

Tip et al. [35] propose Jax, an application extractor for Java. Like the tools described before, it identifies and remove redundant components of the application by building a call graph. However, it also performs other debloating passes, such as merging class hierarchy and renaming symbols. It takes a class file as input and outputs a class archive containing the reduced application. In the same fashion, tree-shaking has been used for Lisp [3], Java [11] and Javascript applications. It aims at reducing the size of an application by bundling a new version of the application, stripped off its unused components, may it be classes and class members for Java in the R8 optimizing compiler or methods in Lisp. A similar approach has been adopted in Javascript in the module bundlers Webpack [16] and Rollup [1].

Heo et al. [13] propose an application specializer called Chisel. It uses reinforcement learning applied to delta debugging to debloat C and C++. First, it divides the application into different partitions and test them against properties given as input of the specializer. The total number of tested partitions is reduced by using reinforcement learning. The smallest partition passing the property test is kept as output. Similarly, Trimmer is an application specializer proposed by Sharif et al. [30]. It takes LLVM bitcode as input and produces a debloated executable which has been specialized for a user-provided configuration, like in RedDroid. The debloating is performed in three passes: first, input specialization, where user-defined configuration is applied in the program, which specialize the required user inputs. The second and third steps are loop unrolling and constant propagation. Once these three passes have been performed, the program relies on the LLVM compiler to be further optmized and reduced.

1.3 Debloating with a call graph

As we have seen in Section 1.2, unreachable code is usually identified by building a call graph. A **call graph** is a directed graph where each node represents a function of the application and each edge represents a call from function F_1 to function F_2. The root(s) of the graph consist of the entry-point(s) of the application. A call graph combines several control flow graphs to form an interprocedural control flow graph.

Call graphs represent the major starting point for more advanced analyses: they are used by solvers for data flow problems, flow-sensitive points-to

[2] http://lists.llvm.org/pipermail/llvm-dev/2016-August/104170.html

algorithms, security-related analyses or interprocedural constant propagation [29], but also for replacing dynamically dispatched method calls with direct method calls or inlining [27], [36], [26].

```
int dog_height() {return 55;}

void print_dog_height() {
    int h = dog_height();
    print(h);}

int main() {
    print_dog_height();
}
```

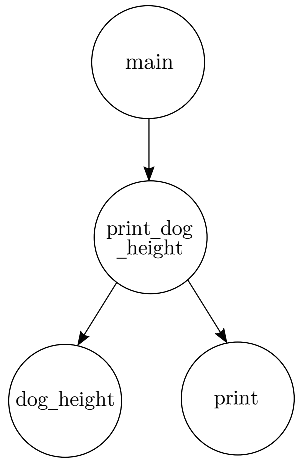

Figure 1.1: A simple call graph

Figure 1.1 shows an application and its related call graph; the entry point of the application is the `main` function. It calls the functions `print_dog_height`, that itself calls the `dog_height` and `print` functions. In this example, the four functions are reachable.

Call graphs are widely used to identify unreachable methods. To illustrate this relation, we can push a bit further the simple call graph example from Section 1.1 by adding a new function `cat_height` in the application. The call graph built upon this change is depicted in Figure 1.2. The node corresponding to the `cat_height` function has been added, but is not connected to any other nodes because it is not called in the application. If we traverse the graph starting from its entry point `main`, `cat_height` will never be reached. Consequently, we can consider `cat_height` as unreachable and prune it away from our application.

1.4 Building a call graph is not trivial

The reachable functions could be easily identified in the examples shown in Figures 1.1 and 1.2. Let's make the previous snippet of code more modular and introduce virtual calls; a virtual call is a call whose target will be resolved at run time. This kind of calls usually occurs in the presence of polymorphism: the call target depends on the run time type of the receiver of the call and this type is resolved via dynamic lookup in absence of optimizations.

Consider the modified snippet of code in Figure 1.3: a class structure has been added and the previous `<animal>_height` functions have been abstracted away as calls to a polymorphic `height()` function that applies to any type of mammals.

```
int cat_height() {return 30;}

int dog_height() {return 55;}

void print_dog_height() {
    int h = dog_height();
    print(h);}

int main() {
    print_dog_height();
}
```

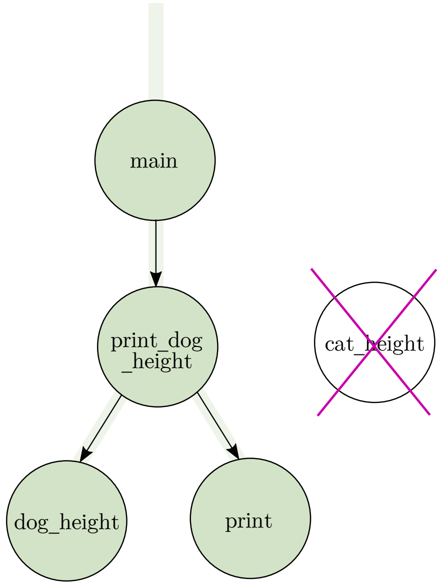

Figure 1.2: A call graph used to debloat: `cat_height` can be pruned away

```
void main() {
    Mammal m = new Dog();
    m = createAMammal();
    m.height();
}
```

Figure 1.3: How to handle virtual calls?

Without any other information about the application, it is difficult to determine ahead of time which `height()` functions are actually being called at run time; it could be the one referring to the `Mammal` class, the `Dog` class or another one. This example illustrates one of the challenges to handle when building a call graph: **the resolution of the virtual calls targets ahead of time**.

1.4.1 Techniques for resolving virtual call targets

Tip and Palsberg [36] have compared the existing techniques aiming at resolving virtual calls ahead of time. They have ranked them according to their cost and accuracy; the overall result of this comparison is available as a diagram in Figure 1.4. The four main algorithms, RA, CHA [7], RTA [4] and 0-CFA [31] will be detailed below and applied to the dynamic snippet from Figure 1.3.

Reachability Analysis (RA). This analysis focuses only on the name of the virtual function being called. It means that when a virtual call target has to be resolved, it is assumed that the target could be any of the functions of the application bearing the same name, disregarding its signature.

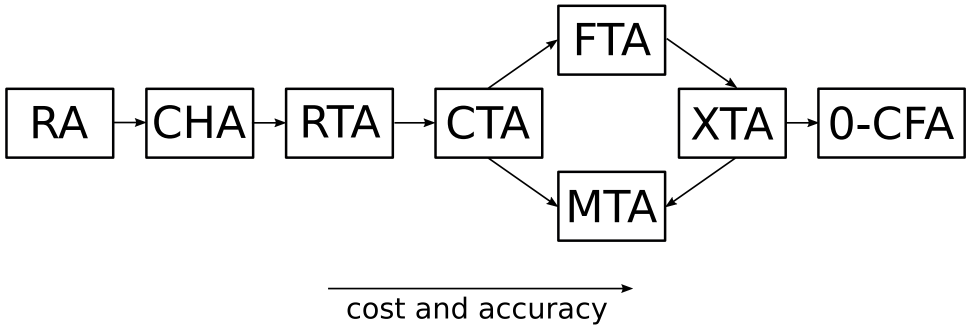

Figure 1.4: Overview of different call graph algorithms, ordered according to cost and accuracy

Class Hierarchy Analysis (CHA). This analysis relies on the class hierarchy of the application to resolve the virtual call targets. It extends the RA algorithm in that when identifying a call target, the type (class) of receiver is taken into account. For instance, in the following virtual call `r.m()`, the static declared type of `r` is considered, as well as all its subtypes as structured in the class hierarchy. Only the functions named `m()` that are declared in these classes are collected and considered reachable.

Rapid Type Analysis (RTA). This analysis is built upon CHA. It further restricts the potential virtual call targets by adding a constraint: the class of the receiver must have been instantiated in a reachable function. Typically, it means that, to be considered reachable, a function must be part of the class or subclass(es) of the receiver, like in CHA, but also that these classes must have been instantiated in a reachable method. If this is not the case, the function is not collected in the graph. Data flow information is not taken into account in this algorithm, like in RA and CHA, so the order in which the application statements are analyzed is not significant.

Several other analyses have extended RTA, mostly by adding more constraints to consider a target as reachable. XTA [36] takes data-flow into account: it saves the return type of reachable functions, as well as their parameter types, to be propagated in the analysis. Other adaptations exist, like CTA, FTA or MTA, that add constraints about data flow in the analysis.

Control Flow Analysis (0-CFA and k-CFA). 0-CFA is a context-insensitive analysis: if two function call bear the same name, only one of the two call sites will be analyzed. k-CFA are context-sensitive analyses, where k represents the number of iterations for the analysis. The higher, the more precise the resulst are, as it relies on the results from the previous iterations. CFA relies on pointer analysis to resolve virtual calls: given a variable, the analysis determines the set of objects the variable may point to. This set is then used to determine which function could be reachable.

1.4.2 One call graph per analysis

Call graphs are usually built on top of an intermediate representation of an application; the entry point of the application (the `main` function, for instance) constitutes the starting point of the analysis. Specific treatments apply according to the nature of the program point being traversed: if a `call` is met, for instance, its target is added to the list of reachable functions. Some treatments may differ according to the chosen analysis as illustrated by the previous Subsection.

As an example, we can build the call graph of the application described in Figure 1.3 using the four algorithms described above. The different call graphs obtained are available in Figure 1.6. **The RA call graph** (Figure 1.6a) contains eight reachable functions; it notably refers to all the four `height()` functions existing in the application, displayed in the class hierarchy diagram in Figure 1.5. **The CHA call graph** (Figure 1.6b) has seven reachable functions -one less than in RA. A quick look at the class hierarchy explains this output: the static type of `m` is `Mammal`. Each `height()` function contained in the `Mammal` class and subclasses will be collected. The functions not part of this sub-hierarchy are ignored, like `Table::height()`. **The RTA call graph** (Figure 1.6c) contains six reachable functions: the `Mammal::height()` function has been pruned away as well because there is no explicit instanciation of `Mammal` (e.g. `new Mammal()`) in the application. Finally, **The CFA call graph** (Figure 1.6d) contains six reachable functions, just like RTA: as CFA is flow-insensitive, it cannot determine whether `m` points to `Cat` or `Dog` because of the static `createAMammal()` function (which is defined as `Cat createAMammal{ return new Cat();}`).

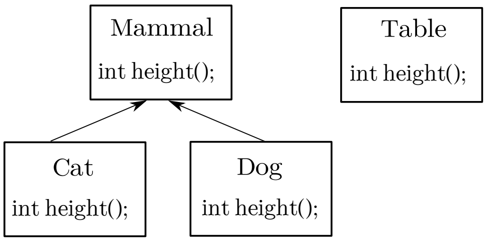

Figure 1.5: Class hierarchy

The call graph produced can sometimes differ according to the algorithm in use. That is why Tip and Palsberg have compared these algorithms in terms of cost and accuracy; RA is the most naive of the four algorithms, but also the least costly: the call graph obtained is usually over-approximated but is quickly built. The k-CFA family, on the other hand, is supposed to give more precise call graphs. However, it relies on a large set of data strutures to approximate precisely the state of the program throughout the control flow graph, which prevents this technique to scale. RTA remains a widely used algorithm to resolve virtual call targets fastly and quite precisely [36].

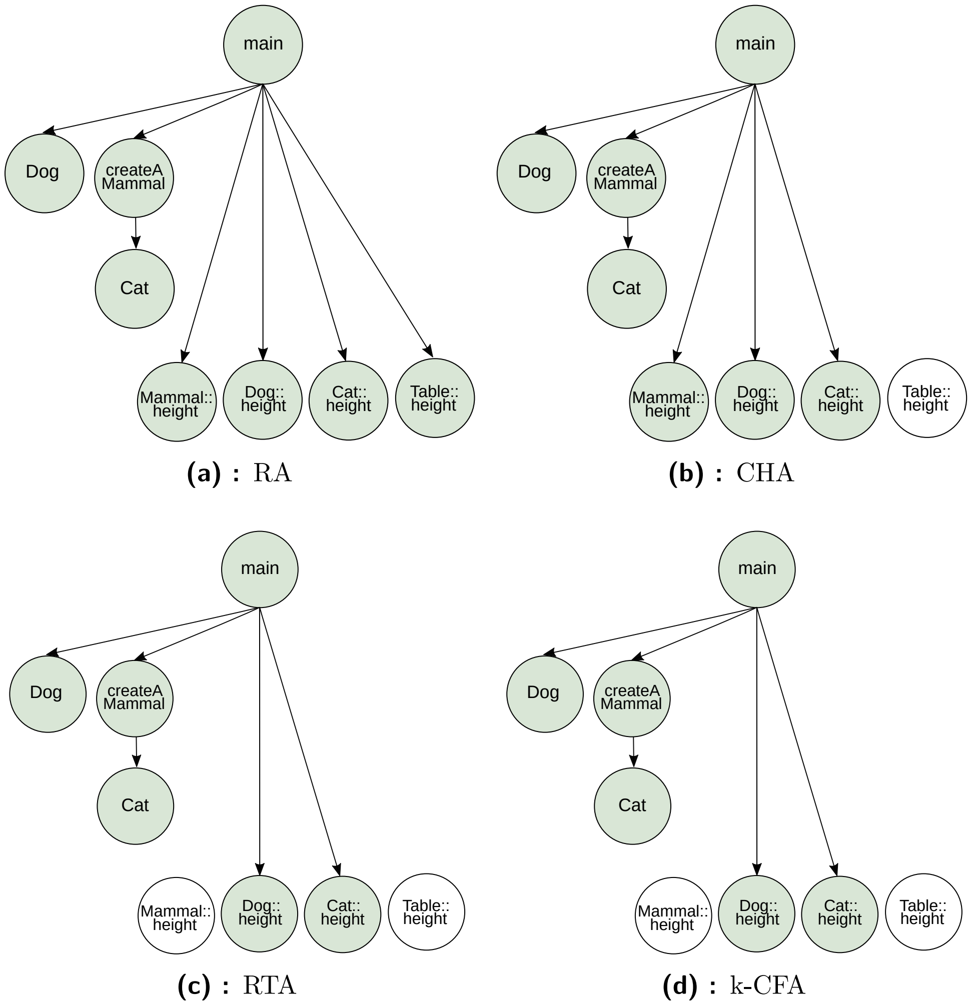

Figure 1.6: Different call graphs according to the algorithm used (reachable if green and has one edge at least)

1.4.3 Are we done then?

In Subsection 1.4.1, we have described several techniques that deal with more or less accuracy with virtual calls resolution. Does that mean that we can safely rely on these algorithms to build a call graph to then debloat our application? It depends to which extent we want the debloated application to be minimal *and* self-contained.

Is it possible to get a minimal call graph statically? Actually, not quite, because of the halting problem [24]. Indeed, as it is not possible to determine statically whether an application will terminate, it is therefore impossible to determine exactly which methods will be called in every case. While the call graph obtained could coincidentally be minimal, it is not possible to have an algorithm that will produce a minimal call graph every time. As a consequence, the call graph is a more or less accurate approximation of the reachable functions of an application: this is what the algorithms we described in Subsection 1.4.1 do when they attempt to approximate runtime values. Besides virtual calls, other dynamic features need to be approximated statically; for instance, reflective calls, as depicted in the snippet of code in Figure 1.7.

```
Class c = Class.forName("Raccoon");
Constructor constr = c.getConstructor();
Object o = constr.newInstance();
o.height();
```

Figure 1.7: A reflective call

In this case, it is difficult to statically determine for sure which object `o` is created, therefore making the virtual call resolution complex. One solution is to perform string analysis on the reflective functions [12]. Points-to analysis was also proposed as a solution [18]. However, in most of the debloating literature, reflection is handled by relying on user-provided information stating which type could be called via reflection [2], [11]. Apart from reflection, dynamic features such as dynamic library loading or indirect pointers affect the static analysis, for both static and dynamic applications [25].

Why aren't we building a call graph dynamically then? Building a call graph this way would address the problem induced by dynamic behaviors as we would not need to approximate runtime values. However, this might prevent the debloated application built of this graph to run, because the dynamic call graph may potentially not be representative of every run. The question that could arise then would be how tied debloating and call graph accuracy are. As we shall discuss in Chapter 4, it might not be that significant to get a very accurate call graph for debloating, as long as it allows the debloated application to run in the general case.

Lots of debloating techniques rely on identifying which methods to keep via the building of a call graph. However, building static call graphs over dynamic application is complex. It would be nice to identify which approach would suit debloating better.

Chapter 2

Debloating R applications

Building a call graph statically is not trivial, notably for dynamic languages. To push the question further, we chose to apply this problem to R, a dynamic language mainly used for data science. In this section, we will explain why we chose R among other languages. We will then describe the main features of R and focus on the ones that are hard to analyse statically.

2.1 Why R?

R is a multi-paradigm, open-source, dynamic language, initially developed in 1993 by Ross Ihaka and Robert Gentleman. It was built on top of its predecessor S and influenced by Scheme [33]. It is typically used in the research and statistical data analysis fields. Its community of developers has been pretty active and 14704 libraries (called "packages" in R) are available on the CRAN package repository in August 2019[1].

R is an interpreted language which relies on an AST interpreter, although the use of bytecode has been introduced in 2011 [34], leading to the use of both an extra bytecode interpreter and a bytecode compiler. In R, "everything that exists is an object, and everything that happens is a function call" [5]. R is an **object-oriented** programming language: GNU R features two main implementations of the oriented-object model, called S3 and S4, that allow the developer to code in an object-oriented fashion. S3 is the most simple system of the two: an attribute "class" is added to an existing R object. When a method call has to be resolved, a generic function determines which function needs to be called according to the type of the receiver. The S4 system is stricter: it relies on "slots" to get a unified way to create objects, classes and methods. A slot represents a property of an object, it is roughly equivalent to an "attribute" in Java. R is also a **functional** language: functions in R are first-class citizens, in the sense that they are considered as regular objects. As such, they can be passed as parameters, assigned to variables... just like other objects. In addition, R is also **lazy** in several aspects. First, its function arguments are lazily *evaluated*, which means they are evaluated only if they are accessed. To support this behavior, function arguments are boxed into

[1]https://cran.uni-muenster.de/web/packages/index.html

"promises", i.e. structures containing the expression, its environment, and the value of this expression once it has been evaluated in this envrionment. This last part makes promises different from "closures": a closure does not memoize the evaluated expression. Then, R packages are lazily *loaded*: a R package is represented as a promise in memory as long as none of its components (i.e. a function, a variable) has been evaluated. It will be fully loaded in memory once it has been evaluated [32]. Finally, R is also **reflective**: it is possible to manipulate, introspect and modify structures representing the running process itself and its behaviour [20].

All these aspects greatly influence the dynamism of R, make it an interesting candidate for our use case. Some of these features will be explained more in depth in Section 2.2, illustrated by concrete examples.

2.2 The dynamic features that break the building of the call graph - contribution 1

Upon closer investigation, it is possible to identify concrete functions or code patterns that, if found in use in R applications, complicate static analysis, and especially static call graph building. To fully understand how they work, we first need to explain what are environments in R and how lookup is performed.

Environments and namespaces. In R, an environment is a hashtable that associates symbols with their value (either a pointer or a R structure). All environments have a parent environment, expect for the empty environment that is the last ancestor of all environments. Every package imported in R comes with his own environment that contains his own bindings for functions and variables; these environments are called *namespaces*. In addition to these environments, functions also come with their environments: when a function is defined, it saves the *enclosing environment* in which the definition took place, because functions in R are closures. When a function is executed, an *execution environment* is created on the fly: it will hold the variables created by the function. Finally, every function call is associated with a *calling environment*, i.e. the environment from within the call was performed. It is also possible to create environments manually: we will refer to them as *user-defined environments*.

Name resolution in R. R dynamism can be easily illustrated by the way lookup is performed in the language. R is **lexically scoped**, which means that an object's value solely depends on its *lexical scope*, i.e. the environment in which the object was defined. Figure 2.1 shows an example: on the left, the call to `f1` results in an error, because `x` is considered unknown. This is because `x` is not defined in the scope of `f`. In the snippet on the right, `x` is defined in the same scope as `f` and the call resolves without error.

This said, sometimes the binding for an object does not exist in its local environment and a chain of environments has to be traversed to find the

```r
f <- function() {
 print(x)
 }

f1 <- function() {
 x <- 11
 f()
 }

f1()
```

ERROR: x unknown

```r
f <- function() {
 print(x)
 }

f1 <- function() {
 x <- 11
 f()
 }
x <- 22
f1()
```

SUCCESS: prints 22

Figure 2.1: Example of lexical scoping

symbol. Usually [19], this lookup chain starts at the current environment, then goes on with the enclosing environments if any. Afterwards, in R, the *search path* is traversed to look for the binding. This path consists of a list of **namespaces**. This path represents the hierarchy of namespaces at a given time and is structured as follows: it starts at the *workspace*, i.e. the current global namespace and ends at the base namespace [32], [37]. The other namespaces lie in between (as well as user-defined environments sometimes, which is not a common occurrence).

Let's illustrate lookup with the search path with an example. In Figure 2.2, both the packages `lobstr` and `pryr` contain a function `ast`, which prints the ast of the expression given as parameter. The `install.packages` and `library` calls first download the package to the disk and then attach the namespace to the namespaces that have already been loaded. In this example, the `pryr` package is loaded, and then the `lobstr` one. Their respective namespaces are thus linked in that order: first the `pryr` one and then the `lobstr` one.

```r
install.packages("pryr")
install.packages("lobstr")
library(pryr)
library(lobstr)

ast(3+2)
```

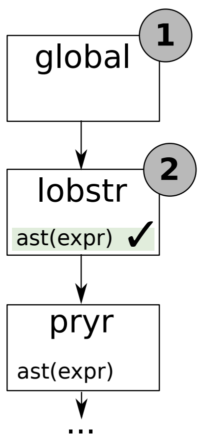

Figure 2.2: Traversing the search path

2.2.1 Affecting the dynamic lookup

The current environment is represented as a `global` namespace; it is the entry point of the search path. As a consequence, when it is time to resolve the `ast(3+2)` call, the lookup is performed as follows: first, the global namespace is searched to see if it contains a symbol named `ast`. Here, it is not the case, so we move to the next namespace in the list, the `lobstr` one. A symbol named `ast` is found and is bound to a function, the call can be resolved and `pryr`'s `ast` will not be called.

Dynamic behaviors like dynamic lookup make the static analysis trickier: the analysis has to approximate values to get a proper picture of the state of the program. In the example presented in Figure 2.2, it is not possible to simply infer whether the `ast` call refers to the `lobstr:ast` version or the `pryr:ast` one: one would need to know how scoping works and the state of the search path. We can straightforwardly tweak the virtual call resolution algorithms presented in Section 1.2 to make them deal with these R dynamic calls; we would need to replace the class hierarchy part by the search path. To do so, the state of the search path needs to be approximated, i.e. the hierarchy of namespaces and the bindings they contain.

R, however, provides functions that dynamically modify these values, on which the analysis relies on. First, it is possible to **modify the bindings of any environment**[2]. The Figure 2.3, for instance, shows that it is possible to rebind the `mean` function available in base R. The call to `modify_mean_binding` turns the `mean` function into a `sum` computation, all through the use of `assign`. A similar behavior is reproducible by using the super-assign operator (`<<-`).

```
mean(c(10,3)) ··········► RETURN: 6.5

modify_mean_binding <- function() {
 f <- function(x) return(sum(x))
 assign("mean", f, baseenv())
 }

modify_mean_binding()
mean(c(10,3)) ··········► RETURN: 13
```

```
mean(c(10,3)) ··········► RETURN: 6.5

modify_mean_binding <- function() {
 f <- function(x) return(sum(x))
 "mean" <<- f
 }

modify_mean_binding()
mean(c(10,3)) ··········► RETURN: 13
```

Figure 2.3: `assign` and `«-` modify the bindings of the function mean at run time

Note that `assign` and `«-` do not behave exactly the same way. `assign` will take any valid environment as a parameter and modify the binding in this environment accordingly (The binding is created if it does not exist in the given environment). The `«-` operator modifies the binding of the object given as left parameter. The environment in which this re-binding will occur depends on the current state of the search path; an existing definition of

[2]Under specific circumstances: bindings in packages are locked by default which means they cannot be modified at run time. However, the package `rlang` provides a very handy function called `env_binding_unlock` that allows to unlock the bindings for any namespace given as parameter.

the object is searched in parent environments. If it is found, the re-binding occurs. If it is not, the binding occurs in the global environment.

R also allows the developer to **modify the search path itself**, as shown in Figure 2.4. This example has the same search path as in Figure 2.2: the `lobstr` package has been loaded last. A new environment `e` is created, in which the `ast` symbol is bound to the `print` function. This environment is then attached to the current search path under the name `rtsbol`. Before the `attach` call, `ast(3+2)` was calling the `ast` from the `lobstr` package. Now that our custom environment has been attached last, an `ast(3+2)` call will refer to the `ast` from this custom environment, as it is the first on the search path to possess this binding. The reverse operation is also possible: a call to `detach` will detach an environment from the search path.

```
e <- new.env()
e$ast <- function(expr) print(expr)
attach(e, "rtsbol")
ast(3+2)
```

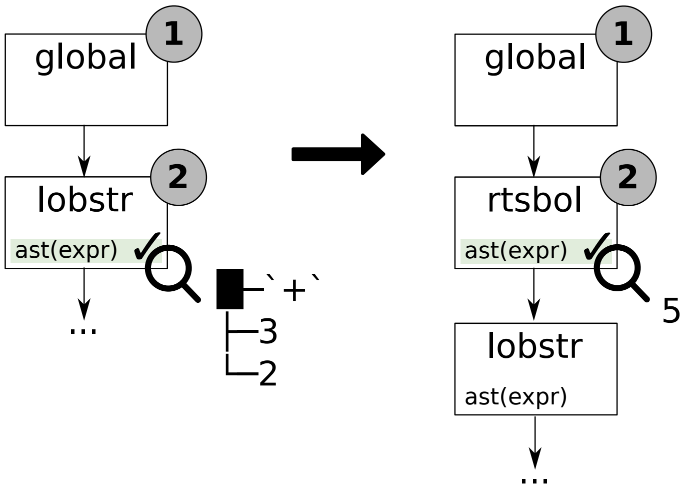

Figure 2.4: `attach` modifies the search path

A function called `library` takes advantage of both these constructs: it is used to load a package into the workspace, i.e it populates the namespace with the package components (functions, variables), and it then attaches the namespace to the search path.

These features modify permanently the bindings and search path: they need to be considered to build an accurate call graph, because we want to approximate the state of the search path and the current bindings to get a precise symbol resolution statically. Their dynamic modification implies that their approximate static states must be accordingly updated and that this updated information must flow correctly in the analysis.

2.2.2 Deferring evaluation using `eval` and `with`

R also enables the developer to modify temporarily and dynamically the way evaluation is performed. For instance, it features ways to **defer evaluation**. It notably proposes an `eval` function that operates in the same fashion as in Javascript: it evaluates its parameter, which can be an unevaluated AST or a string yet to be parsed. Figure 2.5 depicts a simple call of `ast` wrapped in an `eval`.

```
eval(parse(text="ast(3+2)"))
```

Figure 2.5: `eval` hides a call to ast

In this example, the parameter of `eval` hides a call to `ast`[3], but it could also hide a search path modification or a rebinding. Figure 2.6 shows how `eval` hides the call: on a regular call to `ast`, the function `ast` would be loaded and then evaluated, and both operations would appear explicitely as bytecode, just like the call for `parse`, for instance (`ldfun_ parse` and then `; parse(text=...)`). In the `eval` case, the `ast` call is pushed as a string and the combination of `parse` and `eval` handles the loading and evaluation of this string, implicitely. As a consequence, we must determine which action `eval` is performing if we want to build a precise call graph [28], [12], [14].

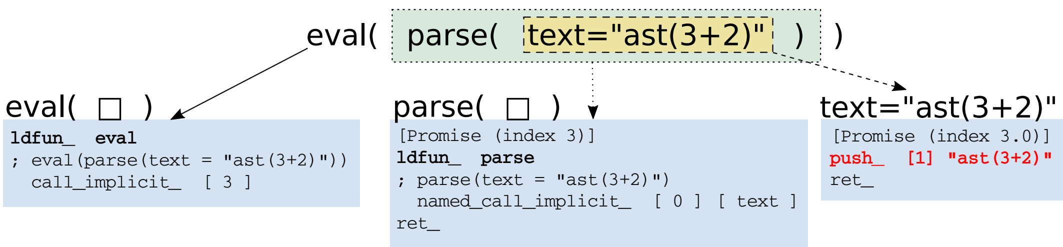

Figure 2.6: `eval` hides the call to `ast` (bytecode has been simplified)

In R, it is also possible to **chose the environment in which the evaluation takes place** by a call to the function `with`. This function is initially meant to interact with databases, but it can be easily twisted to a case similar as in Figure 2.7: in this example, the `ast` called in the one contained by the environment `e` that is *not* part of the search path. If `e` does not contain `ast`, then the search path is traversed. Knowing which `ast` is called implies here to also have an approximation for `e`, on top of the approximation of the search path structure and active bindings.

```
e <- new.env()
e$ast <- function(expr) print(expr)
with(e, ast())
```

Figure 2.7: Chosing the namespace in which the evaluation takes place

[3]note that *which* `ast` is called is a whole different problem (see 2.2.1)

Chapter 3

A study of R dynamism: tracing R applications for dynamic features

In the previous chapters, we have listed several R dynamic constructs that could make the static call graph building more complex. Before investigating alternative ways to build call graphs that would take these constructs into account, we want to confirm that these constructs are used in real R code. An existing R tracing infrastructure has been previsouly developed to get data about promises in R applications [10]: we have tuned it to get data about these dynamic behaviours. This part deals with the adapted tracer and the R constraints that had to be coped with. It also presents the results of the tracing of a thousand of R packages.

3.1 The existing R tracing infrastructure

The infrastructure that we have used in this study has been developed by Goel [10], and was built upon previous works on TraceR [23]. It comprises of three parts:

- **R-dyntrace**, an instrumented R interpreter where probes have been inserted to record the interpreter state on specific program execution events. Notably, there are recorded events for several types of function calls (including the S3 and S4 ones), interactions with variables (read, write,...);

- **The tracer**, loaded as a R package in R-dyntrace. It models the state of the interpreter according to the information gathered by the probes. This is the part that we have tuned to get information about the dynamic behaviour of R;

- **Dynalyzer**, combined with a tracing pipeline. It executes R-dyntrace in addition with the tracer on a corpus of R applications provided by the user. It automatically extracts the runnable code from the applications and produces memory-efficient output tables. It is meant to scale to be used on a very large set of R packages.

3.2 Adapting the tracer - contribution 2

The existing tracer needs to be modified to get data about the dynamic features described in Section 2.2. We have focused as a starting point on the use of the functions `assign` and `<<-` because they seem more likely to be used in R applications. To illustrate how the modified tracer works, we rely on an example and explain how and why we did adapt the tracer to get the produced results.

3.2.1 Motivating example

The snippet in Figure 3.1 illustrates different behaviors involving the use of `assign` and `<<-`. Four functions are defined:

- `foo` and `foobar`, which print a given statement;
- `bar`, which redefines `foo` in the global environment;
- `baz`, which defines `foobar`, giving it the same definition as `foo`.

```
foo <- function() {print("Original function")}

bar <- function() {
 assign("foo",
        function() {print("Modified once")},
        globalenv())
 }

baz <- function() { "foobar" <<- foo}

# first calls to foo and foobar
foo() #call_1
foobar() #call_2

bar()
baz()

foobar <<- function() {print("Modified twice")}
assign("foo", foobar, globalenv())

# following calls to foo and foobar
foo() #call_3
foobar() #call_4
```

Figure 3.1: `assign` and `<<-` in use

The output of this application is available in Figure 3.2: the first line corresponds to the first call to `foo`: it still points to the initial definition of `foo`. Second call triggers an error: `foobar` has not yet been defined, because `baz` has not been called yet. The last two lines of the output correspond respectively to the second call to `foo` and the the second call to `foobar`. Both functions have been redefined by above calls to `assign` and `<<-`. Note that the `<<-` acts as a simple assignment when it is used directly in the global environment. `"Modified function"` is never printed: the effects of the `bar` and `baz` calls have been shadowed by the last calls to `assign` and `<<-`.

```
> "Original function"
> Error in foobar() : could not find function "foobar"
> "Modified twice"
> "Modified twice"
```

Figure 3.2: Output from application in 3.1

We can use the tracer on this application. Simplified results are available in Table 3.1. Each row of the table correponds to a dynamic function call. The columns display :

- `function_name`, in the form **namespace__name::function__name**. This is the name of the dynamic function call that has been collected by the tracer;
- `function_type`, either a closure, special or builtin; This is the type of the dynamic function;
- `dyn_call_count`, the number of times this function has been called dynamically in the applicatino;
- `redefining_symbol`, `1` if this function is redefining an existing symbol, `0` otherwise;
- `symbol_name`, the name of the symbol being (re)defined;
- `environment_address`, the address of the environment in which the (re)definition is taking place;
- `to_fresh_env`, `1` if the (re)definition occurs in an environment that has been manipulated on the current execution stack, `0` otherwise;
- `parent_id`, the id of the function calling this specific function (the id is a hash of the function definition and package name).

This table shows that there is one dynamic call to `assign` in the application, `assign` being a closure part of the `base` environment. This `assign` call defines the binding of the `foo` symbol in the environment located at `0x63690fd0`. The second line indicates that there is also a dynamic call to `<<-` in the application, `<<-` being a special part of the `base` environment as well. This call defines the binding of the `foobar` symbol in the environment located at `0x63690fd0`.

function _name	function _type	dyn_ call_ count	redefining_ symbol	symbol _name	environment_ address	to_ fresh_ env	parent_id
(base::assign)	Closure	1	0	foo	0x63690fd0	0	x1gq#AL 6I+bfWuZE HqtDzA==
(base::«-)	Special	1	0	foobar	0x63690fd0	0	nDl4hy+O 37k2CojxU IYA7w==

Table 3.1: Tracer output related to Figure 3.1

3.2.2 Description of the tracing process

To get the results obtained in Table 3.1, the tracer has to go through several steps. Once the execution of the application has started:

1. If a call is being executed, the tracer enters one of the probes related to function calls (`closure_probe` or `special_probe`) and inspects the given call.

2. If this is a call to `assign` or `«-`, the arguments are processed to get the (re)defined symbol name and the caller environment address. These information are then pushed on a assignment stack part of the current custom tracer state.

   The execution goes on.

3. If an assignment is being performed, the tracer enters one of the probes related to assignments and inspects the given assignment.

4. If this assignment matches with the top of the assignment stack, the tracer retrieves the information needed for our experiment and stores them in the dynamic output table.

5. Once the assignments and calls are resolved, the objects created for the analysis (higher-level representations of calls, functions...) are destroyed.

3.2.3 Work done

The tracer is a small R package (less than 100 LOC) that calls a C++ library (more than 6000 LOC): most of the modifications required for this study were made in the C++ part.

First, to specifically track the `assign` and `«-` calls, the `closure` and `special`-related probes were modified to filter these calls. Then, two extra probes related to assignment in environments had to be lifted out of a previous study to get better details about the application state without compromising the study results. They were heavily modified to fetch information about the calls that is used afterwards in our study output, to know for instance whether the call is dynamic or which symbol is being redefined in which environment. In correlation, the tracer has high-level representations of the traced application components such as calls, functions or arguments; this study required amendments to some of these representations, especially the `Call` class to make them hold more information about its state. The existing layout of the table summarizing the traced calls has been modified to hold these data, it has been named `dynamic_call_summaries`.

In overall, 420 LOC were added to the C++ code of the tracer to be used in our study. This version has been shaped by trial and error and required multiple iterations to get relevant results. Part of the complexity can be explained by the complexity of the R language itself: for instance, gathering basic data about the calls, such as the value of their arguments, was a perplexing experience due to the laziness and the related side-effects. Besides, most operations required to have a clear understanding of R internals to be handled properly

3.2.4 Adapting to R

As previous Subsection 3.2.3 suggests, the existing tracer had to be adapted to be used in our study. We had to carefully identify which features of the tool were relevant regarding our goal. In addition, the API had to be modified to get enough data to conduct the analysis. In this subsection, we detail the choices we made in the custom tracer that may seem intriguing at first.

Not all assigns are considered dynamic. Only two dynamic calls are being gathered in Table 3.1, yet the application from Figure 3.1 contains two `assign` calls and two `<<-` calls. The reason is that we only focus on "very" dynamic calls, i.e. calls that potentially interfere with outer (non-local) environments. In the example from Figure 3.1, the calls to `assign` and `<<-` that are performed in `foo` and `bar` fit into the "very" dynamic category.

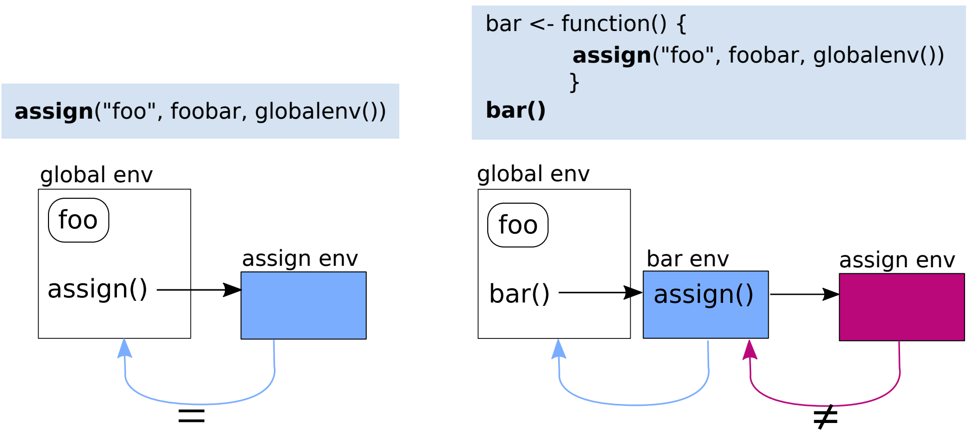

Figure 3.3: Left `assign` is not dynamic, right `assign` is

In Figure 3.3, the `assign` wrapped in the `bar` call is "very" dynamic: every function executes in a fresh environment and the calling environment of `assign` is the `bar` environment. This environment is different from the one given as third parameter of the `assign` call (`globalenv()`): the call is thus considered dynamic. When `assign` and `<<-` are only interfering with their local environments, they behave as simple assignments that will not make the call graph building more complex. Because we need to identify if a call is "very" dynamic, we need to have information about the current execution environment and the caller environment: if they differ, the call is considered as dynamic. This is the reason why we extract specific information about environments in the Step 2 of the tracing process described in Section 3.2.2.

Two probes for two kinds of functions. There are three types of functions in R: closures, specials and builtins. Closures are the standard functions in R, i.e. it is the kind of function created when the `function` keyword is used. A closure object holds its arguments, its body and its enclosing environment (the environment in which it was defined)[1]. Specials and builtins are both internal functions that point to primitives' index. Specials do not evaluate their arguments while builtins do [33], [32]. Each of these functions is executed differently internally (via `execClosure` for closures, `eval` and `bceval` for specials and builtins, `forceAndCall` for builtins), which is why we require three different probes to trace the program state at these points.

`assign` is a closure and `<<-` is a special, which means that we have to rely on two of these probes for our experiment. The `special` probe is reached after the special's arguments have been evaluated, while the arguments are still boxed into unevaluated promises when reaching the `closure` probe: that is the reason why we need to process the arguments differently in Step 2.

Two probes to get away with side-effects. We need the `closure` and `special` related probes to respectively trace the calls to `assign` and `<<-`. In these two probes, a first processing is performed to get information about symbols and environments (Step 2). However, most of the information present in the final output table is not gathered within these probes, but within the assignment probes that are potentially reached later in Step 4: the `environment_variable_define_probe` and `environment_variable_assign_probe`.

We rely on these two probes to avoid side-effects: we mentioned earlier that a very dynamic call must interfere with an outer environment. This is true but incomplete: to be considered as very dynamic in our experiment, the call must also bind the value of the symbol to a function. Let's consider this snippet: `assign("foo", function() print("plop"), globalenv())`. This call is potentially "very" dynamic: it binds `foo` to a function printing `"plop"` in a potential outer environment. In this case, identifying the second argument as a function is straightforward. It gets more complex in this situation: `assign("foo", f, globalenv())`. In this case, `f` could point to any kind of object: a lookup must be performed to figure out whether it points to a function. However, performing the lookup through the R internal API could trigger unwanted side-effects that could modify the state of the application

[1] The lexical scoping principle induces that we evaluate the expression against this enclosing environment.

and affect the tracing results. The `environment` probes are reached at a point in execution where the function arguments have been looked up: this is the reason why we rely on these for the tracing process.

Getting qualitative insights on dynamic function usage. Our tracer gathers data to be used for a qualitative analysis: it enables us to identify the reasons why such dynamic calls have been executed. Once a dynamic call has been spotted, the tracer saves the caller id (`parent_id`) of the dynamic function. All function definitions are saved by the tracer in a dedicated table and we can easily check the caller definition in this table by using the id collected during the tracing. We use this piece of information to manually check the dynamic call site in the source code. Relying on the caller id only allows the analysis to scale but it sometimes requires some heavy manual analysis to precisely identify the location of the call site. As a consequence, we have added an option in the tracer to get the filename and line number of the dynamic call. It takes advantage of the debug information stored by the R interpreter. This gives more easy-to-consult results but adds a significant overhead to the tracing process.

3.3 Study results: assessing dynamism quantitatively and qualitatively

We want to know if the constructs using `assign` and `<<-` depicted in Section 2.2 are encountered in real R applications. In this section, we present the results obtained after tracing these behaviors and provide hints to explain their usage.

3.3.1 Set-up

The tracing was performed on a Dell Precision with Ubuntu 18.04 LTS, a 3 Ghz processor Intel Xeon, 32 Gb 2400MHz DDR4 of RAM. We ran the tracing pipeline over 1000 R packages available on the CRAN repository. The R version used is 3.5.0 (2018-04-23).

The packages and their dependencies had first to be installed in the R-dyntrace environment; they were then traced using the tracing pipeline. The analyzer part of the pipeline extracts runnable code from the R packages, i.e. tests, vignettes (runnable documentation) and code examples; these scripts are then traced. Errors may arise during the installation and extraction phases: some packages may not be available for the R version we are using, some system libraries may be missing, preventing the package to be properly installed, or some dependencies may not be resolved. In our experiment, 6101 extracted scripts were valid and traced.

3.3.2 Results

Quantitative results

Calls to `assign` and `<<-` have been traced 2183311 times over the whole package corpus. Among these calls, 157434 dynamic calls to `assign` and `<<-` were identified. Figure 3.4 sums up the distribution of `assign` and `<<-` regarding the total number of dynamic occurrences: there is an over-representation of dynamic `assign` calls, which are more than a hundred time more called than their `<<-` counterpart. All the dynamic `<<-` calls from our corpus redefine the bindings of already existing symbols; almost 80% of the dynamic `assign` calls do likewise.

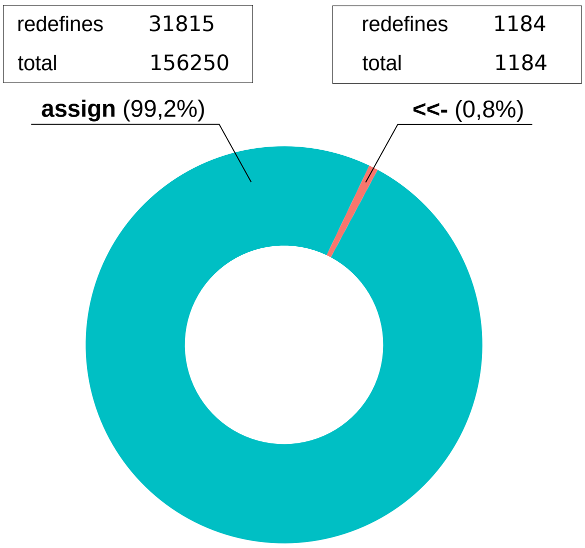

Figure 3.4: Proportion of `assign` and `<<-` over total number of dynamic calls

Qualitative results

Over our corpus, the `assign` calls (re)defines the bindings of twelve different symbols: their distribution is shown in Figure 3.5. In the figure, the symbols `.Methods` and `.Method` are separated from the rest of the group due to their higher number of occurrences: they indeed represent 98,9% of the total number of `assign` cases. A similar unbalanced distribution is observed for the dynamic `<<-` calls: 92% of the total number of dynamic `<<-` cases modify the bindings of only two symbols: `pathGrob` and `vars`. Such unbalanced results exist because these symbols are being modified in language functions called a lot of times.

Disregarding the distribution shape, we can focus on the reasons motivating the need for very dynamic behaviors; in most cases, it is possible to identify these reasons by looking at the source code of their call site.

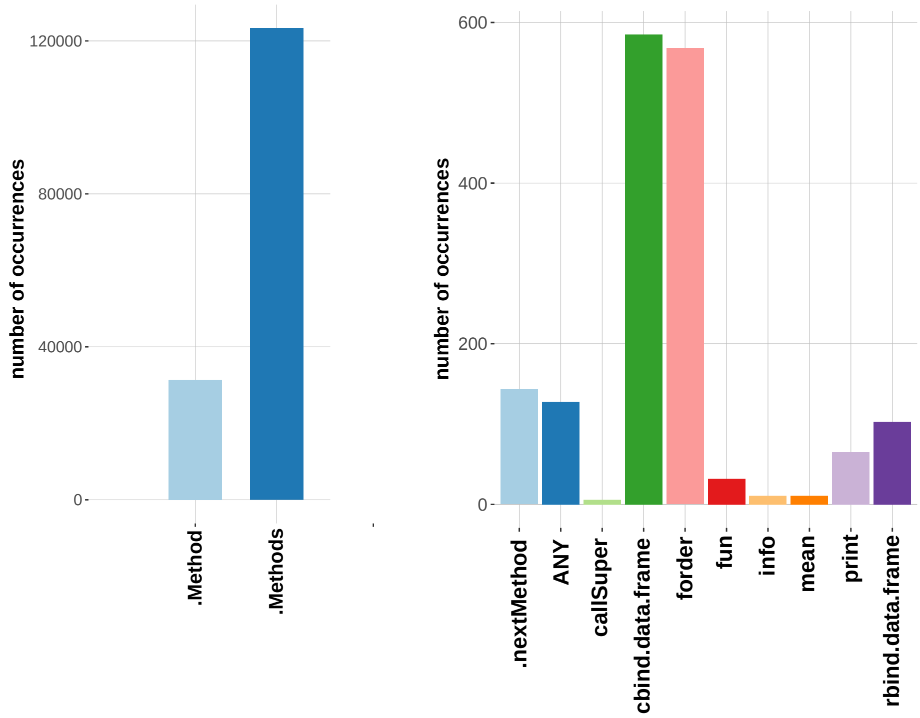

Figure 3.5: (Re)defined symbols by `assign`

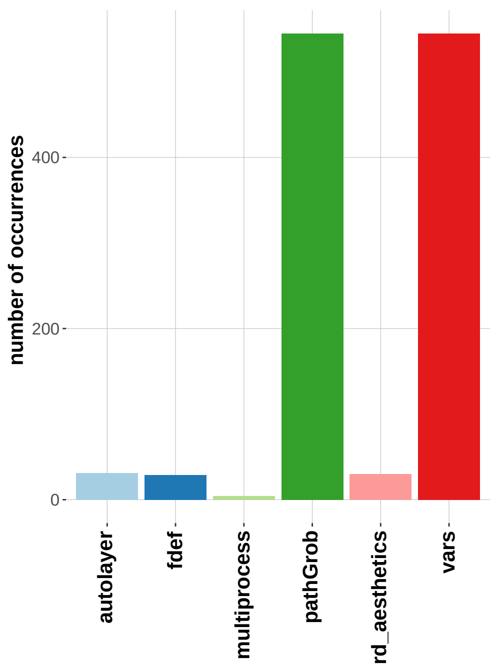

Figure 3.6: (Re)defined symbols by `<<-`

Populating namespaces. This is the first cause of dynamic uses of `assign`. The `.Method` and `.Methods` redefinitions are both triggered by calls to `library`, which is called to load a package into the workspace (i.e populating the namespace with the package bindings) and attach the namespace to the search path. `.Method` is used in the S3 object-oriented model. This model uses generic functions to dispatch functions, that is to say the generic function decides which function to dispatch to. For instance, `add` in Figure 3.7 is a generic function. Plain `add` corresponds to the generic function.

```
add <- function(x, y, ...) {
    UseMethod("add", x, y)
}

add.numeric <- function(x, y, ...) return(x+y)
add.character <- function(x, y, ...) return(paste(x,y))
```

Figure 3.7: `add` as a generic S3 function

The `.Method` symbol is used during the dispatch phase: it stores the function found by the dispatch, `.Method` lays in the environment in which the generic function is being evaluated. A `loadMethod` function is defined in `.InitMethodDefinitions` and contains the `assign(".Method", method, envir = envir)` that triggered the tracer. The action of this `loadMethod` function corresponds to the use of `.Method` described above.

The `.Methods` symbol is used in the S4 object-oriented model. It corresponds to a merged methods list, that is an object formerly used to store methods for dispatch. It is being redefined by a call to `.makeGeneric` that creates a generic function: this generic function definition is the one being assigned to `.Methods`.

Avoiding name clashes. Redefinition also occurs to avoid name clashes; we have seen in Section 2.2 that symbols are easily shadowed in R depending on the search path structure. The tracing shows several occurrences of dynamic calls that are meant to prevent name clashes. For instance, the redefinition of `vars` in the `<<-` case (Figure 3.6) is explicitely justified as such in the comments: `# To avoid namespace clash with dplyr`. This comment can be found in the `ggplot2` package which provides a `vars` function that "takes inputs to be evaluated in the context of dataset". The `dplyr` package does provide a `vars` function as well with a different behavior. When `ggplot2` is loaded in the workspace, it automatically assigns `dplyr::vars` to `vars` if the `dplyr` package is required.

Assuring compatibility. Compatibility with other packages is also a concern: for instance, the redefining of `print` from the `data.table` package in Figure 3.6; this package relies on the `print` function defined in the `ggplot2` package. However, this function has been updated in `ggplot2` and now return a non-void value: it is dynamically redefined to keep the same value type as before, to avoid patching the whole code of the `data.frame` package. `cbind.data.frame` and `rbind.data.frame` are being redefined for the same reasons: the `cbind(datatable, dataframe)` function does not behave as expected. This issues could be solved using less dynamic solutions but this leads to compatibility issues (`[it] introduced compatibility issues with package ‘IRanges‘, since ‘IRanges‘ also masks ‘cbind‘.`)

Improving performance. Dynamism is also introduced to boost performance: the `data.table` package binds the `forder` symbol to a fast order algorithm in a temporary environment. This optimized ordering is then used in the data table through an `eval` call.

Chapter 4
Future works

The tracing helped us identify the presence of reflective code constructs in real R applications. Moreover, such behaviors have been found in package code, some of which being widely used by R users. In this chapter, we discuss several solutions that could be considered to deal with this dynamism in the debloating process.

4.1 Plugging the call graph into the Ř compiler infrastructure

Ř is a just-in-time (JIT) compiler available for R [8]. It proposes several optimizations passes that are being performed in a SSA intermediate representation called PIR. It notably proposes a scope resolution pass: it relies on scope analysis to turn the loads from memory into PIR SSA variables to boost performance.

The scope analysis that is performed is a forward data-flow analysis: each program point computes information about the past behavior of the program [22]. When the CFG is traversed, if a `ldVar` or `ldFun` instruction is met, its potential bound value(s) are being approximated in regard of the current abstract state of the application, notably the current abstract environment hierarchy. For instance, if a `ldFun plop` is met, the function `plop` will be searched in the abstract environments that are part of this hierarchy.

As of now, the abstract environment hierarchy is updated when a `mkEnv` instruction is met: a new abstract environement is created and is linked to its parent environment. In any case, this hierarchy will only contain local environments that have been created in the same compilation unit; indeed, it is assumed that all other environments could change and they are thus not taken into account. If the symbol is not found in the hierarchy, then the value is flagged as "tainted" and the load will not be optimized. We could modify the way environments are taken into account; packages bindings are locked by default, so we could assume that the package environments are not going to change. As a consequence, we could take them into account when the lookup is performed in the analysis.

This same approach could be used in the call graph construction to resolve names and it may be worth considering debloating by mixing both static analysis and values obtained dynamically, in the same fashion as [21]. Indeed, the optimisations usually performed at runtime by the compiler could be used as the first debloating steps of an application by eliminating dead branches and dead stores, removing unecessary environments. Then further debloating passes could be implemented and applied.

4.2 Statically determining dynamic usages

The call graph built upon this tweak would not be accurate, as the environment hierarchy would be highly simplified. Call graph building approaches are usually conservative: when a dynamic call is hard to resolve statically, its target is preferably over-approximated, meaning the amount of potential targets exceeds the real amount of targets. This conservative approach is usually preferred when false-negatives have to be avoided, for instance when performing just-in-time optimizations [24]. From the debloating perspective, we could question whether soundness is a requirement for the call graph building part.

Building an unsound call graph implies that it may not be representative of every run because some potential call targets may not have been identified. Specifically in our case, this would mean some aspects of the dynamism could be ignored. As a consequence, the debloated application built on top of this call graph would not contain these calls. Would this compromise the debloated application execution?

We can consider the dynamism usages identified in Section 3.3.2. Some of these usages could be ignored in the building without compromising the execution, like for instance the performance-related ones: ignoring the rebinding of `forder` would result in a performance drop, but would not prevent the application from running. Similarly, sone cases of comptability-related usages could also be ignored: for instance, if the `IRanges` package is not part of the application, rebinding to keep compatibility with this package is unnecessary.

Therefore, soundness is not to be seek at all cost for debloating R applications. However, the dynamic calls would still need to be statically analyzed to determine their usage in the application, to decide whether their action can be ignored or not. Finding ways to statically classify dynamic calls regarding their usage would be an exciting, yet tricky, next step for this project. The study of R dynamism should also be pursued to get a better picture of the different usages that have to be considered.

4.3 Further adapting the call graph algorithms

Although some kinds of dynamic calls can be ignored, some others still need to be taken into account to build the call graph. To do so, specific abstract operations could be implemented: unoptimized, a call to `assign` is translated to a `LdFun` PIR instruction. A special treatment could be applied when a `LdFun` to `assign` is traversed: the corresponding arguments should be analyzed to determine if the assign call is dynamic; if so, the binding in the affected abstract environement should be updated. This would help achieve a more accurate lookup statically. Same would apply when a call to `<<-` is traversed, and potentially to other dynamic calls with some variations.

These specific abstract operations would need to be inserted in the chosen call graph building approach, which is yet to be determined. RTA and k-CFA seem to be strong contenders: RTA because it proposes a good cost-accurracy trade-off, k-CFA because it would help achieve a better accuracy level. Depending on the sought level of soudness, it would also be interesting to compare call graphs built statically and dynamically.

4.4 Less dynamism

One yet-to-explore solution could consist in finding ways to reduce the number of dynamic occurrences in R applications. A similar approach have previously been adopted for Javascript [14]: because `eval` calls obstruct static analysis, the authors transform their occurrences to other language constructs that go along with static analysis. For instance, when an `eval` is tracked with a constant argument like `eval("var x;")`, it is replaced by this constant argument. They also rely on constant propagation to turn the argument into a constant and pruning away the call to `eval`. In the same way, proposing a dynamism-aware linter could also help reducing the unwise use of dynamic features: equivalent less dynamic constructs could be proposed to the developer. The `lintr`[1] R package is available on CRAN: it performs static analysis for R and provides hints for syntax error, semantic issues and adherence to style. It could be possible to add some analysis passes to this linter that would identify problematic dynamic patterns and propose hints for turning them less dynamic.

[1] https://github.com/jimhester/lintr

Bibliography

[1] Rollup, a javascript modules bundler. https://github.com/rollup/rollup.

[2] Anon. Initialize Once, Start Fast: Application Initialization at Build Time. page 24, 2019.

[3] G. Attardi and C. Italia. The Embeddable Common Lisp. page 12, 1995.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.

[5] J. M. Chambers. Object-Oriented Programming, Functional Programming and R. *Statistical Science*, 29(2):167–180, May 2014. doi:10.1214/13-STS452.

[6] D. Crockford. Jsmin, a javascript minification tool, 2001. http://www.crockford.com/javascript/jsmin.html.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[8] O. Flückiger, G. Chari, J. Jecmen, M. Yee, J. Hain, and J. Vitek. R melts brains - an IR for first-class environments and lazy effectful arguments. *CoRR*, abs/1907.05118, 2019, 1907.05118. URL `http://arxiv.org/abs/1907.05118`.

[9] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 117–121, New York, NY, USA, 1984. ACM. doi:10.1145/502874.502886.

[10] A. Goel and J. Vitek. On the Design, Implementation and Use of Laziness in R. 1:27, 2019.

[11] Google. R8 source code repository, 2017. https://r8.googlesource.com/r8/.

[12] N. Grech, G. Kastrinis, and Y. Smaragdakis. Efficient Reflection

String Analysis via Graph Coloring. *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany*, 2018. doi:10.4230/lipics.ecoop.2018.26.

[13] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*, pages 380–394, Toronto, Canada, 2018. ACM Press. doi:10.1145/3243734.3243838.

[14] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 34, Minneapolis, MN, USA, 2012. ACM Press. doi:10.1145/2338965.2336758.

[15] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–199. IEEE, 2018.

[16] T. Koppers, J. Ewald, S. T. Larkin, and K. Kluskens. Webpack, a javascript modules bundler. https://github.com/webpack/webpack.

[17] J. Lecomte. Yui, a javascript compressor, 2007. http://yui.github.io/yuicompressor/.

[18] B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and K. Yi, editors, *Programming Languages and Systems*, volume 3780, pages 139–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/11575467_11.

[19] M. Lutz. *Learning python: Powerful object-oriented programming.* " O'Reilly Media, Inc.", 2013.

[20] P. Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM. doi:10.1145/38765.38821.

[21] E. Mera, P. López-García, G. Puebla, M. Carro, and M. V. Hermenegildo. Combining static analysis and profiling for estimating execution times. In M. Hanus, editor, *Practical Aspects of Declarative Languages*, pages 140–154, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] A. Møller and M. I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[23] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the r language. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 104–131, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[24] G. C. Murphy, D. Notkin, and E. S. . Lan. An empirical study of static call graph extractors. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 90–99, March 1996. doi:10.1109/ICSE.1996.493405.

[25] A. Quach, A. Prakash, and L. Yan. Debloating Software through Piece-Wise Compilation and Loading. page 19, 2018.

[26] D. Rayside and K. Kontogiannis. Extracting java library subsets for deployment on embedded systems. *Science of Computer Programming*, 45(2-3):245–270, 2002.

[27] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950312.

[28] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do. In M. Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813, pages 52–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-22655-7_4.

[29] B. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979. doi:10.1109/TSE.1979.234183.

[30] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 329–339, Montpellier, France, 2018. ACM Press. doi:10.1145/3238147.3238160.

[31] O. Shivers. Control flow analysis in scheme. In *ACM SIGPLAN Notices*, volume 23, pages 164–174. ACM, 1988.

[32] R. D. C. Team. R internals, 2019. URL `https://cran.r-project.org/doc/manuals/r-release/R-ints.html`.

[33] R. D. C. Team. R Language Definition, 2019. URL `https://cran.r-project.org/doc/manuals/r-release/R-lang.html`.

[34] L. Tierney. The r bytecode compiler and vm. R Implementation, Optimization and Tooling, 2019. URL `https://riotworkshop.github.io/`.

[35] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 292–305, New York, NY, USA, 1999. ACM. doi:10.1145/320384.320414.

[36] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM. doi:10.1145/353171.353190.

[37] H. Wickham. Advanced R, 2019. URL `http://adv-r.had.co.nz/`.

[38] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, page 421, Santa Fe, New Mexico, USA, 2010. ACM Press. doi:10.1145/1882362.1882448.