CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Dynamic test generation for R packages
Student:	Filippo Ghibellini
Supervisor:	Ing. Filip Křikava, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2018/19

Instructions

In addition to the source code itself, R packages come with an extensive set of examples of their usage in the form of code snippets. It is possible to take advantage of those to synthesize new tests.
1. Analyze the structure of an R package.
2. Analyze and propose ways to automatically generate more method-level unit tests from the code contained in the packages documentation and test suite.
3. Implement a prototype of the proposed solution with adequate documentation and test coverage.
4. Evaluate the advantages of this approach.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 8, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical informatics

Bachelor's thesis

Dynamic test generation for R packages

Filippo Ghibellini

Supervisor: Ing. Filip Křikava, Ph.D.

15th May 2017

Acknowledgements

I would like to thank prof Jan Vitek for making all this possible.

Declaration

In Prague on 15th May 2017 . . . .

Citation of this thesis

Ghibellini, Filippo. *Dynamic test generation for R packages.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Statistické programování nabývá na popularitě s tím jak se zároveň zvyšuje poptavka v příbuzných oborech jako Machine learning, Big data a podobné. R je hlavním hráčem v této oblasti avšak jeho jedinečný návrh znemožnil adopci pokroku z jiných standartních jazyků. V této práci představujeme nástroj umožnující nahrávání spuštění programu a následné generování unit testů kontrolující reproducibilitu sledovaného chování.

Klíčová slova unit testy, dynamické, tracování

Abstract

Statistical computing is gaining popularity with the increasing demand in related fields like Machine learning, Big data, and others. R is the main player in terms of programming languages but its unique design made it difficult to share advancements from standard languages. One of the artifacts is the deficiency of more advanced testing tools. In this thesis we present a tool that allows to record executions of an R program and generate unit tests asserting the reproducibility of the observed behaviour.

Keywords unit tests, dynamic, tracing

Contents

List of Figures

List of Tables

Introduction

Some of the main current trends of the IT industry are *Business intelligence*, *Machine learning*, *Big data* and related fields. It is a reaction of the companies, which try to maximize revenue by customizing the offers to the customer. All the above listed fields have roots in statistics, and so it doesn't surprise us that it creates demand for higher quality standards in statistical computing.

At the moment the most commonly used tool in this field is the programming language *R*. It is a freely available language developed in 1993. During its design process most emphasis was put on user friendliness, interactivity, and easy extension through library modules. While evaluating the above criteria, an average statistician was considered as the target user and not an average programmer as is the case with most other programming languages. The result is a very non-orthodox language having 3 main classes of users as described in [1].

- End users with a very basic understanding of the language
- Statisticians with a reasonable grasp of the language's semantics
- R core team members

It is now becomming increasingly common to run R programs in server batches or for them to be generate visualisations on web pages.

Such programs have to be mantained and extended for a long period of time and so the requirements on the effectiveness and reliability increase. One method to prevent the introduction of bugs in the process of extending a software project is **regression testing**.

During my project I will extend a package for the programming language R **Genthat**[2], that is capable of generating such tests from existing code. The package was originally developed to validate different VMs for the language R, although now it finds uses in different areas such as prototyping new type systems.

One of the main advantages of this approach over classical handwritten tests is the possibility to better localize the source of incompatibility thanks to more granular test coverage. Consider the following example

```
# package source
f1 <- function(x) x + 1
g1 <- function(y) f1(y * 10) - 100
# call in tests, manual page, vignette, ...
g1(55)
```

Genthat would generate 2 test cases for the above single call.

```
# ...
expect_equals(f1(550), 551)
# ...
expect_equals(g1(55), 451)
```

If the implementation of *f1* would suddenly be modified to return `x + 2` the generated tests would help the developer to quickly localize the source of the failure to the function f1.

Used concepts

Tracing

Tracing a program means observing the execution of a program either by source code modification or instrumenation of code. These observations are then stored in a structured manner resulting in what is called a trace.

Regression testing

A regression test is a testcase that can be run on a modified of the program to validate that some functionality is not broken by the modifications.

Formula

Expression that is not meant to be evaluated but used for its parse tree.

Promise

A promise is a special type of R value that allows lazy evaluation (i.e. evaluating expressions only when needed for another compuatation). It represents a value that is yet to be computed.

Chapter 1

Introduction to R

The design process of the R language was very different from the one of more common languages. As the target user was considered an average statistician. This decision shifted focus from adhering to commonly expected behaviour of programming languages (like all C derivates) or application of advancements in PL theory (like ML derivates, erlang, ...) to creating a language that is good at performing computations on vectors of data while being very user friendly, compatible with the way statisticians think about their problems and all in all allows to get things done. The result is a language that attracted a wide group of end users, but scared most PL researchers and tooling manufacturers away. One of the most popular works on the matter is appropriately called "The R inferno".

Here I will introduce the reader to the basic concepts of R that were necessary to understand in order to be able to work on this project. It is not supposed to be an exhaustive description of R and unlike traditional learning materials I will focus on the aspects of the language that are typically learned with usage, but are essential to this thesis as they fundamentally shaped the solution.

1.1 Values and types

1.1.1 Attributes

Almost any value in R's type system can have so called attributes set. They can be thought of as the value's metadata. Attributes can be accessed and modified through designated functions like *attributes* and *attr*. Alternatively the more common attributes have special functions to manipulate them like *names* or *dim*.

1.1.2 Atomic vectors

The need to perform fast computations on vectors of data has influenced the type system of R. It has no concept of a single number or single string. The most basic types for values are vectors. i.e. `5` is a numeric vector of length 1. Longer vectors are created through the concatenation function `c`, although beware that *c* is not a special operator, just a regular function like any other making it possible to redefine it (although it is not common to do so). Also vectors are homogeneous i.e. all the elements must be of the same type.

As if it wasn't enough there is the unfortunate naming of character vectors. Character vectors are vectors containing strings. i.e. `"foo"` is a character vector of length 1. `c("foo", "bar")` is a character vector of length 2. In C such a vector has type STRSXP and the elements have type CHARSXP but in R we call the whole vector a character vector thus you use `is.char()` to assert its type.

Vectors can be of 6 types: integer, numeric (doubles), logical, character, complex and raw.

Vectors can have length 0. In such cases they are represented respectively by: `integer(0)`, `numeric(0)`, `logical(0)`, `character(0)`, `complex(0)`, `raw(0)`. Calling c with no arguments will return `NULL` (obviously).

Concatenating any value of type other than one of the 6 above listed will result in a list.

1.1.3 Lists

As noted above concatenating non-atomic values will result in lists. As will concatenating heterogeneous values or concatenating values through the *list* function. A list is a collection of potentially heterogeneous values identified by their position and optionally a label. e.g.

```
l1 <- list(3,4,5)
l2 <- list(3,"foo")
l3 <- list(foo = 3, bar = 42)

l2[[1]] == 3
l2[[2]] == "foo"
l3[[2]] == l3$bar
```

1.1.4 S3 objects

Lists are also what fuels one of the many object systems R has. It is called S3 and the whole idea is to just label a list with a character vector and then dispatch function calls on this list based on the elements of that vector. The elements of the vector are called class names and they are stored in the *names* attribute of the list. Such lists are called S3 objects.

```
obj1 <- list(left = 3, right = 7)
class(obj1) <- c("addition", "bin-operation")

performOp <- function(x) UseMethod("performOp")
performOp.addition <- function(x) { x$left + x$right }
performOp.subtraction <- function(x) { x$left - x$right }
performOp.default <- function(x) stop("operation not
↪  implemented!")

10 == performOp(obj1)
```

1.1.5 Symbols

Symbols are just interned strings that allow quicker lookups and comparison for equality. In expressions they are used to encode identifiers, keywords and special symbols.

1.1.6 Language

The result of parsing R source code is a tree like structure we regard to as expressions. Values of this kind don't all have the same data type. Inner nodes typically will have a type of *language* while leaves can be either *atomic vectors* or *symbols*. Literal values are encoded as scalar vectors of the appropriate type - there is no wrapping element signaling that it is a lexical element.

Language values are internally encoded as lists where the first element identifies the the called function and the following elements are the arguments. It is possible to convert between lists and call with the functions `as.call` and `as.list`, thus a simple call like `3+x` can be encoded as `as.call(list(as.symbol("+"), 3, as.symbol("x")))`.

Parsing

```
fn1(c(32,42), join("foo", x))
```

will result in an expression copying the following structure:

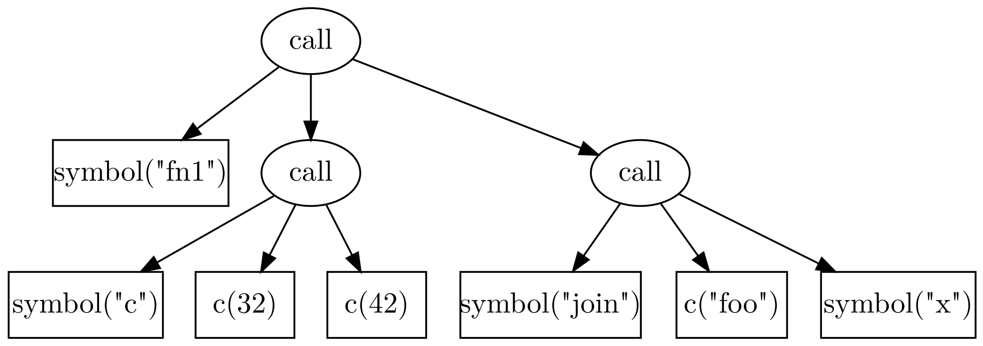

Figure 1.1: Structure of a call value

1.1.7 Enviroments

Enviroments are data structures that allow you to bind values to string keys. The reason they are not called maps or dictionaries like in other languages is that in R they represent one of the principal building blocks of the language. The whole scoping mechanism is built around environments. Let's see an example

```
a <- 5
f <- function() {
    b <- 3
    function(c) {
        a + b + c
    }
}
g <- f()
g(50)
```

The above code will unsurprisingly result in the value 58.

The interesting bit however is how the function *g* returned by the call to *f* keeps a reference to the variables it uses. Such functions are generally called closures or lambdas. In a language like C++, that has manual memory management, an object is returned instead of a function. This object copies the necessary bindings or addresses to those bindings and overloads the function application operator effectively behaving like a function. e.g.

```
int a = 5;

function<int(int)> f() {
    int b = 3;
    return [&a,b](int c) -> int {
        return a + b + c;
    };
}

int main()
{
    auto g = f();
    cout << g(50) << endl;
    return 0;
}
```

which is tranformed by the compiler into something like

```
int a = 5;

class Lambda1 {
    int &a;
    int b;
public:
    Lambda1(int &a, int b) : a(a), b(b) {};
    int operator()(int c) { return a + b + c; };
};

Lambda1 f() {
    int b = 3;
    return Lambda1(a,b);
}

int main()
{
    auto g = f();
    cout << g(50) << endl;
    return 0;
}
```

In languages like JavaScript or Python functions have always a scope bound to them and they are always evaluated in that scope. The bindings and values can be destroyed only when no function bound to them is accessible by live code.

R is very similar to JavaScript with one fundamental difference. In the above languages scopes are abstract concepts, they represent a namespace for bindings whereas in R scopes are just on-the-fly generated environments. For instance in R you can ask for the environment value and iterate over all its bindings.

When a function is called an evaluation environment is created. It is ephemeral by default [3] which means that it is destroyed as soon as the function returns, but in cases as the above call of *f* the environment is needed even after the call thus the evaluation environment ceases to be ephemeral.

```
r <- 88
f <- function(a) {
    b <- 42
    e <- environment()
    keys <- ls(e, all.names = TRUE)
    print(keys)
    lapply(keys, function(key) {
        print(paste0(key, " is bound to ", deparse(e[[key]])))
    })
}
g <- f(32)
```

the output of the above snippet is:

```
[1] "a" "b" "e"
[1] "a is bound to 32"
[1] "b is bound to 42"
[1] "e is bound to <environment>"
```

We can see that r is not seen by our code. Here we touch on another property of environments - inheritance.

1.1.7.1 Environment inheritance

To achieve scope nesting each environment is given the option to have a parent environment. Most operations working with the bindings of an environment will try to find a matching binding in the environment itself and on failure proceed to its parent and repeat this process until an orphan environment is reached (this case typically manifests in an error).

1.1.8 Functions

Functions in R are made up of a body, a list of formal arguments with optional default values and an environment.

1.2 (Im)mutability

One very specific aspect of R is how it handles mutation. It borrows the immutable behaviour from other functional languages making atomic vectors, lists, functions and others immutable. e.g.

```
a <- list("a", "b", "c")
b <- a
b[[2]] <- "x"
print(toString(a))
print(toString(b))
```

will output

```
[1] "a, b, c"
[1] "a, x, c"
```

Similiarly with functions:

```
a <- function(x, y) { x + y + z }
b <- a
formals(b) <- alist(x = ,y = ,z = 3)
print(a)
print(b)
```

outputs:

```
function(x, y) { x + y + z }
function (x, y, z = 3)
{
    x + y + z
}
```

As the above example requires *b* to be reassigned to a different function and variables are just bindings in environments it follows naturally that environments are the exception and they are indeed mutable. e.g.

```
e1 <- as.environment(list(a = 2))
e2 <- e1
e2$a <- 7
```

Here *e1* and *e2* point at the same environment for the whole execution of the program. On line 3 the environment is just modified in-place to bind 7 to the string "a".

1.3 Promises & laziness

R has one special type of value that wasn't mentioned in section 1.1 - promises. A promise is a triple of a value, an expression and an environment. They are used for delayed evaluation of expressions. The value is computed by evaluating the expression in the environment and then it is stored in the promise for successive uses (it is only computed once). It is done so only when the value is requested. The type of the resulting value is not known until the promise is forced.

This is the mechanism that fuels R's laziness. Although R is propagated as a lazy language, promises are by default used solely for function arguments. e.g.

```r
f <- function(a) {
    a + 3
}
x <- 7
f(x + 2)
```

When *f* is called, *a* is bound to a promise. And only when line 2 is executed `x + 2` is evaluated. The ability of promises to store environments makes it so that `x + 2` gets evaluated in the exact environment where x is bound.

To see this in action we can run:

```r
f <- function(x, y, z) {
    y + x + y
}
f(
    { print("eval x"); 2 },
    { print("eval y"); 3 },
    { print("eval z"); 9 }
);
```

resulting in

```
[1] "eval y"
[1] "eval x"
[1] 8
```

Notice that the second expression was evaluated first as it is the first element of the sum and that it was evaluated only once even though it appears twice in the sum. Also the third expression was not evaluated at all as it wasn't used in the function.

Although basically all operations in R are technically function calls, laziness isn't that common. Let's take the assignment operator as example. An assignment will always force the evaluation of the right side.

```r
f <- function(x) {
    y <- x
}
f({ print("eval x"); 2 });
```

Here the assignment forces the evaluation of the expression even though the value is never really used.

1.3.1 Substitution

One of the most used features of promises is the ability to request the unevaluated expression. This is commonly used to evaluate the argument in a different environment than prescripted by R or to generate labels for plots by converting the expression to a string. The expression can be retrieved by calling *substitute* on the promise. e.g.

```
f <- function(a) {
    print(substitute(a))
}
f(x + 2)
```

will output `x + 2`.

```
f <- function(expr1, env1) {
    eval(substitute(expr1), env1)
}
a <- 4
f(a + 2, as.environment(list(a=7,`+`=`+`)))
```

will return 9.

1.3.2 Functions accepting expressions

In R a function can consume an argument either by forcing its value or by requesting the expression that generated it. The second option is possible because arguments are wrapped in promises and you can call `substitute` on them to retrieve the expression bound to them (see 1.3.1).

Here I show example calls of such functions and their respective implementations. Both functions just print the expression passed in the first argument. But one expects the expression as a value whereas the other requests the expression used to generate the argument.

```
dump_forced <- function(expr) {
  print(expr)
}
```

```
dump_forced(quote(a + 2))
```

```
dump_subst <- function(form) {
  print(substitute(form))
}

dump_subst(a + 2)
```

Now let's consider we want to extract the argument into a variable. In the first case it will work just fine, whereas in the second case, assigning the expression to a variable would force the evaluation which would end with an error. Even if the expression was valid in the scope, substituting the argument within the function's body would still yield the variable name instead of the expression.

```
# ok
x <- quote(a + 2)
dump_forced(x)
# bad (a is not available in this scope)
y <- a + 2
dump_subst(y)
# bad (would output "z")
z <- quote(a + 2)
dump_subst(z)
```

Arguments passed to functions of the second type must always be expressed in the parenthesis of the call, making them hard to read if long. Also computed arguments can be passed to such functions only through reflection

```
e1 <- quote(a + 2) # this is a computed expression
eval(call("dump_subst", e1)) # passing a computed expression to
↪  a function that calls substitute on its argument
```

but not always as we can read in `do.call`'s documentation:

" The behavior of some functions, such as 'substitute', will not be the same for functions evaluated using 'do.call' as if they were evaluated from the interpreter. The precise semantics are currently undefined and subject to change. "

1.3.3 Formulas

As if the previous section wasn't confusing enough there is a special infix operator in R called ~. Its only function is to return an expression reflecting how it was called. i.e. `a ~ b` gets evaluated to `call("~", as.symbol("a"), as.symbol("b"))` we call such values formulas. Since ~ doesn't force its arguments it effectively behaves as if the whole expression was quoted (neither *a* nor *b* have to be in scope). Indeed, calling substitute or forcing an argument using this operator yields the same value. [1] i.e.

```
force_arg <- function(a) a
subst_arg <- function(a) substitute(a)
force_arg(x ~ y) == subst_arg(x ~ y)
```

You might be asking yourself now what this could be useful for. Why not just use a function that substitutes the argument just like `dump_subst`?

This comes handy when you want to overload the function using the S3 1.1.4 dispatch mechanism instead of analyzing the expression yourself to handle the different cases.

```
    f1 <- function(x) UseMethod("f1")
    f1.formula <- function(x) print("<passed formula>")
    f1.default <- function(x) print("<passed non-formula>")

    a <- c(3,4)
    f1(a ~ b) # calls f1.formula
    f1(a + 2) # calls f1.default
```

Listing 1: S3 based expression function

```
    f2 <- function(x) {
        e <- substitute(x)
        if (e[[1]] == `~`) {
            # formula handling
            # e holds the formula
        } else {
            # non-formula handling
            # x holds the value
        }
    }
```

Listing 2: Alternative implementation

[1] Applying the operator actually returns a call with the class attribute set to "formula" while substituting the argument returns a call extending "call". This is not reflected in the comparison for equality.

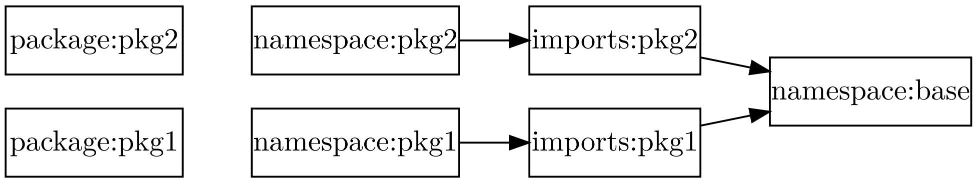

Figure 1.2: Simplified view of environment hierarchy

f2 in listing 2 has the same behaviour as f1 in listing 1 but it doesn't require ~ to have an implementation (just the parser to be able to parse   as an infix operator). The first function is easier to read, but it comes with the limitation of accepting only expressions whose top-level operator is ~.

I would like to finish this section with a quote from https://stat.ethz.ch/R-manual/R-devel/library/stats/html/formula.html.

" Variable names can be quoted by backticks 'like this' in formulae, although there is no guarantee that all code using formulae will accept such non-syntactic names. "

1.4 The module system

It is not uncommon for scripting languages to not fully abstract away the module system. In NodeJS (a JavaScript runtime) for instance every script has a special variable named *exports*. The contents of this variable are returned to others who include this script. Modules typically export an object containing all the functions it needs to export. R has a not very well documented module system that is best described by [4]. Each package's source code is evaluated in a special environment called the package **namespace environment**. Before the evaluation the namespace's parent is set to another special environment called the package's **imports environment** that contains the bindings with all the functions the package requires from other packages. The imports packages itself inherits from *namespace:base* which contains all the basic R functions and that in turn inherits from the global environment. Once evaluated, the packages exported function's bindings are copied into a new environment called the **package environment** (a better name would have been exports environment). All the bindings in the imports environments are copied from other packages' package environments. This is possible because the package environments of all packages have all the bindings locked (i.e. they cannot be modified).

namespace:base inherits from a special environment called the global environment (this is the one in which user code runs in). The global environment then inherits from a chain of package environments of packages that were included by the user. The chain finally ends with the empty environment.

Not all dependencies must be included by copying the exported functions into the imports package. Only dependencies specified in the "Imports" section. Dependencies specified in the "Depends" section are supplied by attaching them just like user-requested packages are. The later option is fragile as a function with the same name defined in the global environment or in a sooner attached package will be returned instead of the one the package expected (see [4] for details).

1.5 Structure of an R package

R packages have a predefined structure they have to follow. It is thoroughly described in [5] so we will not go into too much detail in here. The only parts that are interesting for this project were the *DESCRIPTION* and *NAMESPACE* files. The DESCRIPTION file contains the basic information about the package like the name, version number, dependencies, etc. The NAMESPACE file documents which names are required by the package and which names it exposes.

Chapter 2

Analysis and design

2.1 Possible ways to generate method level tests

2.1.1 Static analysis based generation

By static analysis we mean the act of performing some kind of analysis on the source code, trying to infer the possible inputs it might accept. From checks on the results of calls to the method it would be possible to infer the subset of expected return values. e.g.

```r
f <- function(x, y) {
    x + y
}

x <- f(a, b)

if (x == 1) {
    g()
} else if (x == 2) {
    h()
}
```

From line 2 it would be possible to infer that both x and y must be of some numeric type based on the application of the + operator, which is limited to those in R. On line 5 we call the function and check whether the returned value is one of $\{1, 2\}$. This might hint that the function should not return a different value. We can then generate random values for the inputs and assert that when we apply the function to them the result satisfies the expected properties.

As we can see this approach would force us to make a lot of assumptions about the intentions of the programmer.

Pros	Cons
• No need for code calling the method	• A lot of assumptions about the intentions of the programmer • Hard decisions about how the analysis should behave

2.1.2 Trace based generation

A more sane approach is to run some already available code that makes use of the method, capture a trace and then transform the trace into a test case.

For instance; let's say we are interested in the function f. Running the available code would leave us with a record of f's call even though it is a case of a indirect call to f.

```
f <- function(x, y) {
    x + y
}

g <- function(a) {
    f(a, a+1) + 3
}

g(3)
```

Tracing the above code would result in the trace:

```
call to f;
    args: 3, 4
    retv: 7
call to g:
    args: 3
    retv: 10
```

It is then possible to transform the trace into a testcase like the following one

```
test_that("generated 7. april 2017", {
    expected_retv <- 7
    expect_equal(f(3,4), expected_retv)
})
```

This aproach makes no assumptions about the programmers intentions. It simply records some execution and generates a test that checks that the function's behaviour is consistent with the observed one.

Pros	**Cons**
• No assumptions about programmers intentions • All tests are valid calls from the domain point of view	• Only generates testcases for existing calls • No input space is explored

Chapter 3

State of the art

3.1 Other projects

As was mentioned in tooling for R is scarce and so I was not surprised to not find any other test generation frameworks targeted at R.

3.2 Genthat

3.2.1 Introduction

Genthat extracts existing source code from a package's tests, manual pages and vignettes. It then traces the execution of that code and generates unit tests based off those traces. Such tests can be used for VM compatibility checking (original purpose), or as regression tests.

3.2.2 Limitations

Before the project started Genthat allowed to trace basic R code by forcing all the arguments. Running it on packages from CRAN yielded a success rate close to 0%. Big part of the package was written in C/C++ which makes its behaviour hard to understand by common R users.

3.2.2.1 Environments

Genthat had no support for tracing environments as it used the function *deparse* to serialize values. The result of calling deparse on an environment is the string `"<environment>"`.

3.2.2.2 Forced arguments

Forcing all the arguments while tracing prevented Genthat from handling properly any function substituting its arguments as they are typically not expected to be evaluated.

Consider the following code. The first argument is forced during tracing which will result in the value 10. The function *f1* though evaluates the expression of the first argument in the data frame passed in the second environment. Thus the generated test completely fails to reflect the behaviour of the original code.

```
x <- 5
f1(2 * x, df)
```

```
f1(10, df)
```

Chapter 4

Realisation

4.1 Serialization function

I decided to implement the serialization of values as a C++ function. This is in conflict with the original intention to minimize C/C++ code, but in this case the readability of the code is better in C++ as it results in a clear switch statement handling each SEXP type.

Serialization was originaly implemented by simply calling *deparse* on values. As this couldn't handle environments and some other values couldn't be unserialized, I decided to write a custom function. First in R, but this version was extremely slow and often broke because of unexpected value types. The C++ version just has to handle all the possible values of the TYPEOF enum. What before took hours to run now finishes in a matter of minutes.

```
string do_serialize_value(SEXP s)
{
    switch (TYPEOF(s)) {
    case NILSXP:
        return "NULL";
    case VECSXP: { /* lists */
        RObject protected_s(s);
        int n = XLENGTH(s);
        string ret = "list(";
        SEXP names = Rf_getAttrib(s, R_NamesSymbol);
        for (int i = 0 ; i < n ; i++)
        {
            SEXP val = VECTOR_ELT(s, i);
            string label =
                (names == R_NilValue ||
                 string(CHAR(STRING_ELT(names, i))) == "")
                 ? ""
```

```
                  : (escape_list_key(string(CHAR(
                  ↪  STRING_ELT(names, i)))) + "=");
            ret += string(i == 0 ? "" : ",") + label +
            ↪  do_serialize_value(val);
        }
        ret += ")";
        return wrap_in_attributes(s, ret, false);}
    // ...
    case INTSXP: {
        RObject protected_s(s);
        int n = XLENGTH(s);
        string elems = "";
        for (int i = 0; i < n; i++)
        {
            int val = INTEGER(s)[i];
            string str_val = val == NA_INTEGER ? "NA_integer_"
            ↪  : (to_string(val) + "L");
            elems += (i == 0 ? "" : ",") + str_val;
        }
        return wrap_in_attributes(s, n == 0 ? "integer(0)" : n
        ↪  == 1 ? elems : "c(" + elems + ")", true); }
```

Let's take a moment to analyze what the serialization of an integer vector involves. First we wrap the value being serialized in the *RCpp* wrapper type *RObject*. We don't actually use the C++ API it provides but it helps us by protecting the value from R's garbage collector as long *protected_s* is not out of scope. We retrieve the length of the vector through the C interface's *XLENGTH* macro and iterate over all the elements. The optional names of the elements are stored in a special attribute. The elements themselves are serialized by a recursive call to the same function. The resulting vector literal is then passed to *wrap_in_attributes* that if necessary wraps everything in a call setting the attributes. *wrap_in_attributes* doesn't wrap the literal if the only attribute is *names* (unlike *deparse*)as this information is redundant with the literal itself.

The above code has to also be able to detect cycles in the serialization structure as environments allow them. In case of a detected loop the function currently simply throws an error. Another problem with an R implementation will be described in the following subsection.

4.2 R errors

When R throws an error the error message is stored in a single globally shared designated location. As the tracing code is executed in the *onExit* handler which is triggered also when the function exits because of an error, if our handler throws itself an error, it will overwrite the original error.

```r
mikes_function <- function() stop("mike messed up")
trace(mikes_function, exit = function() stop("joe messed up"))
tryCatch({
  mikes_function()
}, error = function(e) print(paste0("errormsg: ", e)))
```

The above code will complain that Joe messed up. Not knowing this you end up with error messages absolutely unrelated to the code you were debugging. Wrapping the handler in a tryCatch wouldn't help as it doesn't stop the propagation of the original error, but the original error message was already overwritten.

```r
mikes_function <- function() stop("mike messed up")

trace("mikes_function", exit = function() {
    tryCatch({
        stop("joe messed up")
    }, error = function(e) { NULL }) # ignore the error
})

tryCatch({
  mikes_function()
}, error = function(e) print(paste0("errormsg: ", e)))
```

As we can see from the output R doesn't help much as it explicitly suggests that the error occurred in *mikes_function*.

```
[1] "mikes_function"
Tracing mikes_function() on exit
[1] "errormsg: Error in mikes_function(): joe messed up\n"
```

There was a point in time during development when the C++ onExit handler called an R serialization function. When the traced function threw an error the serialization function would fail and overwrite the error message, it would then only unroll the stack to the point where it was invoked by C. In C it behaved as if the call succeeded. When the handler terminated the R stack continued unrolling because of the original error but with a new error message. Just like above. Oh, the joys of debugging R!

4.3 Expressions

As explained in 1.3.2 handling exceptions is not a trivial task. The solution found is expected to handle most of the common cases.

Instead of forcing the argument promises directly, we deconstruct the promise expression into its elementary formulas (symbols and literals). We then store in the trace the original expression as well as the value of all those symbols bound in the calling environment. Consider the following code

```
a <- 42
f(a + 2 + x, df)
```

When tracing the first argument we will decompose the argument expression into its elementary expressions yielding the set $\{a, `+`, 2, x\}$. Of those we will ignore '+' as we consider such symbols as environment-provided. Ignoring 2 which is a self-contained literal we are left with $\{a, x\}$. Both of these are possible references into the caller scope. We then try to look for them in the caller scope and possibly store their values (x will not be found as it is provided by the function). The result of this step of tracing is a list looking like `list(expr=quote(a + 2 + x), vals=list(a = 42))`. Such a trace allows us to handle most uses of formulas.

4.4 Function decoration

R has a called *trace* that can be used like so:

```
fn1 <- function() { print("traced fn") }
trace("fn1",
    tracer = quote(print("entered function")),
    exit = quote(print("leaving function")),
    print = FALSE
    )
```

```
> fn1()
[1] "entered function"
[1] "traced fn"
[1] "leaving function"
```

When we enter the function we need to record the arguments and store them so we can pair them with the return value when the function exits. Pairing the values gets complicated with trace as there's no unique identifier of a call. Registering an on.exit handler in the trace entry handler is not an option as the tracer is not evaluated directly in the traced function's body and so on.exit whill not work as wanted. Instead I created a custom tracing mechanism that prepends the function's body with an expression that generates a new id for the call, stores the arguments in a map and registers an on.exit handler. The exit handler can then use the id to retrieve the arguments from the map. Let's see a simplified example of a traced function:

```
id_counter <- 0

function(arg1, arg2, ...)  {
    {
        # generate id
        id <- id_counter
        id_counter <<- id_counter + 1
        # store arguments in a map
        storeArgs(id, sys.call())
        # register the on.exit handler
        on.Exit(gen_exit_handler(id))
    }
    # ...
    print("original body")
    # ...
}
```

Since functions are immutable this new function needs to be reassigned in all the necessary environments, including the package environment whose bindings are locked. R provides a function *unlockBinding* that allows you to ignore the lock.

4.5 Return value comparison

R has impure IO which makes it hard to deduce whether a function is pure or not. Generating regression tests for IO operations doesn't make much sense, thus it is desirable to ignore impure functions. As a heuristic Genthat now before generating a test reruns the function with the same arguments in a different environment and compares the outcome with the traced return value. If the values don't match it assumes the function performs some IO operation and it aborts the testcase generation.

4.6 Floating point number serialization

For the calls to be reproducible we need the serialization mechanism to preserve equality. Also the above explained impurity detection mechanism depends on equality comparison of the return values. This is a generally accepted big problem for floating point numbers. But here it cannot be ignored as simple number literals produce such values, and thus they are more common than integers in R. The simple process of parsing and deparsing a number literal will break equality. As the returned literal by deparse doesn't have to evaluate back into the original value.

To solve this, I had to modify how the serialization function handles floating point numbers so that it would return the binary representation of the value as it stored in memory. The result is a somehow intimidating representation of simple values.

```
serialize_r(5L)
[1] "5L"
serialize_r(5)
[1] "readBin(as.raw(c(0,0,0,0,0,0,0x14,0x40)), n=1,
↪  \"double\")"
```

Possible further improvements

Here I will describe ways in which Genthat could be further improved.

Globally available references

Serializing a value is unnecessary if the value is always available and its contents were not interesting from the test's perspective. For instance we are talking about values like the global environment, functions from base, predefined dataframes. Genthat could be modified to recognize such values and avoid the serialization by recreating them by calling their canonical constructors. e.g. `f(globalenv())` should generate an identical call in the generated test instead of trying to recreate a new environment reflecting the contents of the global environment at the time of the call.

Function serialization

Functions passed as arguments or returned from functions could be serialized. This would require to record also the function's environment with the whole inheritance chain.

Single operation serialization

By serializing all the arguments through one call, it would be possible to restore the environment graph even if it is not a tree.

R builtin serialization

We are currently discussing the option to expose a serialization function in the official C API with one of the members of core R team Tomas Kalibera. This could significantly reduce the line count of our serialization function.

Conclusion

In this thesis I presented R and its features that make method-level tests hard to generate. I analyzed the possible ways to generate regression tests and decided to extend an existing package *Genthat*. After introducing how Genthat works, I explained the improvements I designed and implemented.

Bibliography

[1] Morandat, F.; Hill, B.; et al. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, Berlin, Heidelberg: Springer-Verlag, 2012, ISBN 978-3-642-31056-0, pp. 104–131, doi:10.1007/978-3-642-31057-7_6. Available from: http://dx.doi.org/10.1007/978-3-642-31057-7_6

[2] Petr Maj, J. V., Tomas Kalibera. testR – R language test driven specification. [Accessed: 2017-05-13]. Available from: http://petamaj.github.io/other/testr.pdf

[3] Wickham, H. *Advanced R*. Chapman & Hall/CRC The R Series, Taylor & Francis, 2014, ISBN 9781466586963. Available from: http://adv-r.had.co.nz

[4] Gupta, S. How R Searches and Finds Stuff. Available from: http://blog.obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/

[5] Team, R. C. Writing R Extensions. Available from: https://cran.r-project.org/doc/manuals/r-release/R-exts.html

Appendix A

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
thesis ............................. the directory with thesis sources
thesis.pdf ................................... thesis in pdf format
```