Vignette: restorepoint

Sebastian Kranz*

February 3, 2013

Department of Mathematics and Economics, University of Ulm

Abstract

This package allows to debug R functions via restore points instead of break points. When called inside a function, a restore point stores all local variables. These can be restored for later debugging purposes by simply pasting the body of the function inside the R console. This vignette briefly illustrates the use of restore points and compares advantages and drawbacks compared to break points.

1 A simple example of debugging with restore points

Consider a function swap.in.vector that shall split a vector at a given position and then swap the left and right part of the vector. Here is an example of a call to a correct implementation:

```
swap.in.vector(1:5,3)

## [1] 3 4 5 1 2
```

Here is a faulty implementation that we want to debug:

```
library(restorepoint)
```

*sebastian.kranz@uni-ulm.de

```
library(restorepoint)
swap.in.vector = function(vec, swap.ind) {
    restore.point("swap.in.vector", to.global = FALSE)
    left = vec[1:(swap.ind - 1)]
    right = vec[swap.ind:nrow(vec)]
    c(right, left)
}
swap.in.vector(1:10, 4)

## Error:  Argument der Länge 0
```

The first line in the function specifies a restore point.

restore.point called inside a function

When restore.point(name) is called inside a function, it stores the current values of all local variables under the specified name. In the example, these local variable are vec and swap.ind and the name is "swap.in.vector".

restore.point is called directly in the R console

When restore.point("swap.vector", to.global=FALSE) is called directly in the R console the following happens:

1. The previously stored local variables are copied into a new environment that has the global environment as enclosing environment
2. The default R console is replaced by the *restore point console*. In this console R commands are evaluated in the environment created in the first step. To leave the restore point console and go back to the standard R console, one just has to press ESC.

In effect, we can now debug the function by simply copy & pasting the interior of the function (or parts of it) including the first line. (Using RStudio, we can just mark all lines and press Ctrl-Enter) We can inspect the variables and code by simply typing any desired command into the R console.

```
  restore.point("swap.in.vector", to.global=FALSE)
```

```
left  = vec[1:(swap.ind-1)]
right = vec[swap.ind:nrow(vec)]

## Error:  Argument der Länge 0

c(right,left)

## Error:  Objekt 'right' nicht gefunden
```

The error occurred in the third line. We can inspect the variables in more detail to narrow down the error.

```
swap.ind

## [1] 4

vec

##  [1]  1  2  3  4  5  6  7  8  9 10

swap.ind:nrow(vec)

## Error:  Argument der Länge 0

nrow(vec)

## NULL
```

There is the culprit. The command nrow returns NULL for a vector. We want to use length(vec) or NROW(vec) instead.

```
# Try an alternative formulation
length(vec)

## [1] 10
```

We can correct the code in our script and directly test it by pasting again the whole function body. There is no need to call the function again, since the parameter from the previous function call are still stored under the name "swap.in.vector".
Test the inside of the function by copy & paste it into the R console.

```
  restore.point("swap.in.vector", to.global=FALSE)
```

```
  left  = vec[1:(swap.ind-1)]
right = vec[swap.ind:nrow(vec)]

## Error:  Argument der Länge 0

c(right,left)

## Error:  Objekt 'right' nicht gefunden
```

The corrected function seems to work fine so far (indeed there is an error left that we remove in Section 3). Pressing ESC returns to the normal evaluation mode of the R Console.

2 Why I prefer restore points over break points

A standard tool to debug a function is to use a break point. In R this can be performed via a call to browser() inside the function (e.g. at the same position where we would call restore.point). When during execution of the function, browser() is called, the R console immediately changes into an interactive debugging mode that allows to step through the code and enter any R expressions. In contrast, when restore.point is called inside the function there are no direct visible effects: the debugging mode starts afterward, when we decide to paste the body of the function into the R console.

I personally prefer restore points over break points for the following reasons:

1. When debugging nested function calls, handling several break points can become very tedious, since the program flow is interrupted with every break point. Despite using traceback(), it is often not clear where exactly the error has occured. As a consequence, I tend to set too many break points and the program flow is interrupted too often.

2. A related point. When I want to turn off invocation of the browser, I comment out #browser() manually and source again the function body again. That can become quite tedious. When using restore points, I typically just keep the calls to restore.point in the code even if they seem not necessary at the moment. Calls to restore.point are simply not very obtrusive. They just make silently a copy of the data. While there is some memory overhead and execution may slow down a bit, but usually I find that negligible.

3. The interactive browser used by browser() has a own set of command, e.q. pressing "Q" quits the browser or pressing "n" debugs the next function. For that reason, one cannot always simply copy & paste R code into the browser. In contrast, the only special key in the debug mode of restore point is Escape, which brings you back to the standard R console. The restore point browser makes debugging via copy & paste from your R script (or in RStudio, select code and press CTRL+Enter) much easier.

4. I often would like to restart from the break point after I changed something in the function, to test whether the new code works. But with nested function calls, e.g. inside an optimization procedure, for which an error only occurred under certain parameter constellations, it can sometimes be quite time consuming until the break point at which the error has occurred is reached again after a a restart. This problem does not arise for restore points, I can always restart at the restore point and test my modified function body.

3 The restore point console vs restoring into global environment

The example above used the call to restore.point with the option to.global=FALSE, which has the effect that future commands are evaluated in the restore point console. The main difference to the standard R console is that expressions are not evaluated in the global environment, but in an environment that emulates the local envirnment of the function that we want to debug. However, by default we have to.global = TRUE and debuggin takes a much simpler but quite dirty form. All stored objects are just copied into the global environment and the usual R console stays in place. You can test the example (make sure you have left the restore point console by pressing Esc).

```
  # If to.global=TRUE or not set, objects are restored into the
  # the global environment
  restore.point("swap.in.vector")

## Restored:  swap.ind,vec

left  = vec[1:(swap.ind-1)]
  #...
```

While this approach is quite dirty, I often prefer it for beeing slightly more convenient. Here are some disadvantages of the restore point console that make me often prefer the global environment approach.

- One has to press Esc to leave the restore point console
- One cannot press the "Up-Arrow" key to get the previous command

On the other hand, there are several advantages of using the restore point console instead of simply copying the variables into the global environment.

- Variables in the global environment are not overwritten. This may seem a very important point. Interestingly though, in my experience, most times I debug an R program, it doesn't really matter if I overwrite global variables when restoring objects.
- In my view more importantly, the restore point console allows to run function calls with the ellipsies, like f(...). From the standard R console running a call of the form f(...) is not possible. (At least, I found no way to assign a value to ... in the global environment. If somebody knows a way, please let me know.)
- If an error is caused in the restore point console, by default a stack trace as in traceback() is shown. I find that convenient.

Even though the points in favor of just using the global environment seem small, I nevertheless typically prefer that dirty approach and therefore made it the default.

Here are some more points that seem noteworthy.

- If you type as a single expression the function restore.point in the restore point console, the corresponding objects are restored and the restore point console changes to the corresponding environment. This does not happen when restore.point is called as part of more complex expressions inside { }, e.g. inside a function, a loop, or an if clause. Then the local objects are stored under the specified name.
- I programmed the restore point console such that if the command source is called, as a single command, then the restore point console automatically quits and returns to the standard console. The reason is that I typically source a file again, when I am finished with debugging, but I want then automatically return to the standard R console without having to press ESC before. (In later version, this behavior shall become optional).

4 Some advice and examples on using restore.points

4.1 When to set restore points

When writing a new function, I tend to always add a restore point in the first line, with name equal to the function name.

```
my.fun = function(par1, par2 = 0) {
    restore.point("my.fun")
    # ... code here ...
}
```

Unlike break points (see discussion below), restore points don't interrupt program execution. Even though most errors are found quickly, there are also often errors that remain hidden for a while. Therefore having restore points in all functions can be quite convenient, in particular in complex code.

One does not have to set restore points at the beginning of a function, but can put them also somewhere else in a function.

4.2 Nested function calls

Restore points are particularly useful when debugging nested function calls and in situations in which errors arise only under specific parameter constellations (possibly randomly drawn ones). Here is an example of a faulty function that shall draw 10 random swap.point for a given vector and print the swapped version.

```
# Randomly choose 10 swap points
f = function(v) {
    restore.point("f")
    for (i in 1:10) {
        rand.swap.point = sample(1:length(vec), 1)
        sw = swap.in.vector(v, rand.swap.point)
        print(sw)
    }
}

set.seed(12345)
f(v = 1:5)

##  [1] NA NA NA  5  1  2  3  4  5 NA NA
##  [1] NA NA NA NA  5  1  2  3  4  5 NA NA NA
##  [1] NA NA NA  5  1  2  3  4  5 NA NA
##  [1] NA NA NA NA  5  1  2  3  4  5 NA NA NA
## [1] 5 1 2 3 4
## [1] 2 3 4 5 1
## [1] 4 5 1 2 3
## [1] NA  5  1  2  3  4  5
##  [1] NA NA NA  5  1  2  3  4  5 NA NA
##  [1] NA NA NA NA NA  5  1  2  3  4  5 NA NA NA NA
```

The result looks strange. There is a mistake either in function f or in swap.in.vector or in both. It is convenient to stop the execution whenever an obviously wrong result is encountered. For this purpose, we modify f by stopping execution if the length of the result is different than the length of the original vector. We also add a restore.point with name "f.in.loop" inside the loop.

```
# Randomly choose 10 swap points
f = function(v) {
    restore.point("f")
    for (i in 1:10) {
        rand.swap.point = sample(1:length(v), 1)
        sw = swap.in.vector(v, rand.swap.point)
        print(sw)
        restore.point("f.in.loop")
        stopifnot(length(sw) == length(v))
    }
}

set.seed(12345)
f(v = 1:5)

## [1] 4 5 1 2 3
```

```
## [1] 5 1 2 3 4
## [1] 4 5 1 2 3
## [1] 5 1 2 3 4
## [1] 3 4 5 1 2
## [1] 1 2 3 4 5 1

## Error:  length(sw) == length(v) is not TRUE
```

The error may have occurred in swap.in.vector or in f or in both. By restoring the restore point in swap.in.vector, we first have a look at the parameters of the last function call before execution has been stopped.

```
#swap.in.vector = function(vec,swap.ind) {
restore.point("swap.in.vector")

## Restored:  swap.ind,vec

swap.ind

## [1] 1

vec

## [1] 1 2 3 4 5

# vec has different values than the parameter v=1:5
# with which we have called f
```

We seem to call swap.in.vector, with a swap.point that is larger than the vector!. This suggests that there is an error in the function f. We restore our restore point "f.in.loop" and examine the local variables.

```
restore.point("f.in.loop")

## Restored:  i,rand.swap.point,sw,v

v

## [1] 1 2 3 4 5

rand.swap.point

## [1] 1

# There must be a mistake when rand.swap.point is drawn
rand.swap.point = sample(1:length(vec), 1)
# Indeed, we use the wrong variable: vec instead of v Corrected:
rand.swap.point = sample(1:length(vec), 1)
```

It can be helpful to include a restore point within the for loop in order to analyze the values of the local variables before the error has been thrown.

5 Known Caveats and Issues

5.1 Failure to make deep copies

One pitfall is that restore.point does not (yet) guarantee that all local variables are stored as a deep copy. This can only be a problem if parameters contain R variables that are copied by reference, like environments. So far I have only included a special treatment for local variables that are environments. Yet, e.g. it is not yet checked whether a local variable that is a list contains fields that are environments. Here is an example what can go wrong then:

```
x = 10
f = function(env, li) {
    restore.point("f")
    env$A = "X"
    identical(env, li$env)
}
env = new.env()
env$A = "A"
li = list(env = env)
f(env, li)

## [1] TRUE


restore.point("f")

## Restored:  env,li

# Correctly retrieve the original value for the parameter env
env$A

## [1] "A"

# Did not make a deep copy of the environment in the list
li$env$A

## [1] "A"

identical(env, li$env)

## [1] TRUE
```