Debugging Without (Too Many) Tears

New packages: debug and mvbutils

by Mark Bravington

"Man is born unto trouble, as surely as the sparks fly upward"[1]; and writing functions leads surely to bugs, even in R. Although R has a useful debugging facility in `trace` and `browser`, it lacks some of the sophisticated features of debuggers for languages like C. The new package `debug` offers:

- a visible code window with line-numbered code and highlighted execution point;
- the ability to set (conditional) breakpoints in advance, at any line number;
- the opportunity to keep going after errors;
- multiple debugging windows open at once (when one debuggee calls another, or itself);
- full debugging of `on.exit` code;
- the ability to move the execution point around without executing intervening statements;
- direct interpretation of typed-in statements, as if they were in the function itself.

I often use the debugger on other people's code as well as my own: sometimes to see exactly what went wrong after typing something, and sometimes merely to see how another function actually works. And sometimes— just sometimes— because there's a bug!

Although this article is mostly about `debug`, it makes sense to briefly mention another new package, `mvbutils`, which is required by `debug`. Besides its many miscellaneous utilities, `mvbutils` supports:

- hiearchical, searchable project organization, with workspaces switchable inside a single R session;
- function editing (interface to text editors), with multiple simultaneous edits, an "unfrozen" R prompt, and automatic backup;
- function code and plain-text documentation stored in the same R object, and editable in the same file;
- informal plain-text documentation via `help`, and conversion to Rd format;
- nested `sourceing`, and interspersal of R code and data in the same file;
- macro-like functions, executing inside their caller's environment;
- untangling and display of "what calls what" within groups of functions.

Once the packages are installed and loaded, detailed documentation can be obtained via `README.debug()` and `README.mvbutils()`. The code of `debug`— which is written entirely in R itself— reveals quite a bit about the workings of R, and will form the subject of a future Programmer's Niche article. The rest of this article is a basic introduction to usage.

Using the debugger

Consider this function, which is laboriously trying to calculate a factorial:

```
f <- function(i) {
  if (i<0)
    i <- NA
  else
    for(j in (i-1):2)
      i <- i * j
  i
}
```

This doesn't quite work: `f(3)` is rightly 6, but `f(2)` is 4 not 2. To prepare for debugging, type `mtrace(f)`, then `f(2)` to actually start. A window containing the line-numbered code of `f` will appear at the bottom of the screen, looking like a wider version of Figure 1. You may need to resize your R window to see it. In the R command window, you'll see a prompt[2] `D(1)>`. The debugger is awaiting your input.

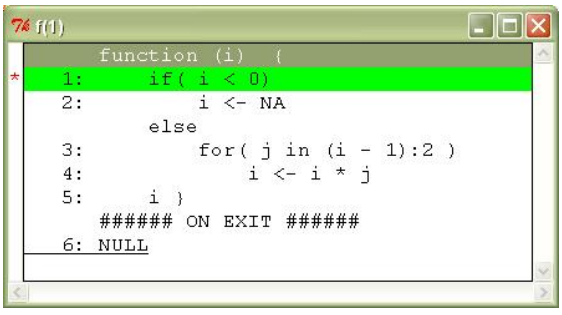

Figure 1: The code window at the start of debugging function `f`

In the R window, press `<ENTER>` to execute the bright green highlighted statement and print the result. The `FALSE` value comes from the conditional test in the `if` statement. The highlight will move

[1] Book of Job, 5:7

[2] The 1 is the current frame number, which matches the number in the title bar of the code window; see "The code window".

into the `else` block, to the `for` loop at line 3. Press <ENTER> again, and the values that the index will loop through are printed. The intention is obviously not to do the loop at all for `i<=2`, but instead you'll see `[1] 1 2` because the ":" is going the "wrong" way[3].

If you were really debugging this function, you would probably exit with `qqq()` at this point, and then edit `f`. For now, though, type `go(5)`. The debugger will begin executing statements in `f` continuously, without pausing for input. When line 5 is reached, it will be highlighted in the code window, and the `D(1)>` prompt will return in the R command window. At the prompt, type `i` to see its value. You can type any R statement (including assignments) at the prompt; it will be evaluated in the function's frame, and the result will be printed in the R command window. The pending statement in your function will stay pending, i.e. it won't be executed yet. To see all this, try hand-tweaking the result by typing `i<- 2`. Then type `go()` (without a line number this time) to run through the rest of the function and return the correct value at the normal R prompt.

If you type something at the `D(...)>` prompt that leads to an error, the error message will be printed but the `D(...)>` prompt will return, with the line that led to the error highlighted in the code window. You can carry on as if the error didn't happen; type some commands to patch the problem, or `mtrace` another function inside which the error occurred, or `skip` to another statement (see below), or type `qqq()` to finish debugging.

Special debugging functions

The "big five" are: `mtrace` to prepare or unprepare debugging, `bp` to manage breakpoints, `go` to execute statements continuously, `skip` to move the execution point without actually executing any code, and `qqq` to halt the debugging session and return you to the usual R prompt (with a "non-error" message). The last three can only be called inside the debugger, the first two from inside or outside. `go` and `qqq` have already featured, and there isn't much more to say about them. As for the others:

- `mtrace(f)` does two things. First, it stores debugging information about `f` in `tracees$f`. The list `tracees` has one element for each function currently `mtraced`; it lives in the `mvb.session.info` environment, shown by `search()`. This environment is created when `mvbutils` is loaded, and lasts until the end of the R session. It is never written to disk, and is used to store session information, like "frame 0" in Splus.

  Second, `mtrace` overwrites `f` with a modified version that calls the debugger. This does not change `f`'s `source` attribute, so you will see no apparent change if you type `f` (but try `body(f)`). However, if you `save` an `mtraced` function and then `load` it into a new R session, you'll see an error when you invoke the function. Before saving, type `mtrace(f,F)` to untrace `f`, or `mtrace.off()` to untrace all `tracees`— or consult `?Save` for a shortcut.

  `mtrace`ing a function should not affect editing operations. However, after the function is read back into R, it will need to be `mtraced` again (unless you use `fixr` in `mvbutils`). All previous breakpoint will be cleared, and a new one will be set at line 1.

- To illustrate `skip`, type `f(-5)` to start debugging again, and press <ENTER>. The `if` test will be `TRUE`, so execution will move to line 2. At this point, you might decide that you want to run the loop anyway. Type `skip(3)` to move the execution point to the start of the `for` loop, and `i` to confirm that the value has not been altered. `skip(1)` will move you *back* to the start of the function, ready to start over. However, note that skipping backwards doesn't undo any changes.

- `bp(5)` will set a breakpoint at line 5 (if the debugger is already running), as shown by the red asterisk that appears to the left of line 5. Type `go()`; the highlight should re-appear on line 5. Now clear the line 1 breakpoints by `bp(1,F)`— the upper asterisk will disappear. Press <ENTER> to execute line 5 and finish the function, then `f(2)` again to restart the debugger. This time, when the code window appears, the highlight will be on line 5. Because the line 1 breakpoint was cleared, execution has proceeded continuously until the breakpoint on line 5 triggers. If all breakpoints are cleared, the entire function will run without the code window ever appearing, unless an error occurs (try `f('oops')`). Clearing all breakpoints can be useful when a function will be invoked repeatedly.

  Conditional breakpoints can be set by e.g. `bp(4,i<=j)`. The second argument is an (unquoted) expression which will be evaluated in the function frame whenever line 4 is reached; if the expression evaluates to anything except `FALSE`, the breakpoint will trigger. Such breakpoints are mainly useful inside loops, or when a function is to be called repeatedly. Because everything in a breakpoint expression

[3] One solution to this common loop bug, is to replace the ":" with the `%downto%` operator from `mvbutils`: `for( j in (i-1) %downto% 2)`. A far better way to calculate vectorized factorials, though, is based on `cumprod(1 %upto% max(x))[x]`, and there is always `round(exp(lgamma(i+1)))`.

gets evaluated in the function frame, including assignments, conditional breakpoints can be used for "invisible mending" of code. Put the "patch" statement followed by `FALSE` inside braces, as in the following hack to get `f` working OK:

```
bp( 4, { if(i<3) break; FALSE})
```

The `FALSE` at the end prevents the breakpoint from triggering, so that execution continues without pause. `f(2)` and `f(1)` will now work OK, giving possibly the world's worst factorial function.

Breakpoints are usually set inside the debugger, after the code window has appeared. However, they can also be set in advance. Unless you are clearing/setting at line 1, you will need to check the line list, which can be seen in `tracees$f$line.list`.

Debugging `on.exit` code

Code set in `on.exit` statements can be debugged just like normal body code. At the end of the main body code, the code window displays a section marked `##### ON EXIT #####`. Until an `on.exit` statement gets executed, this section just contains a `NULL` statement. When `on.exit` is invoked (either in the function, or from the prompt), the code window will change to show the new exit code. You can set breakpoints in the exit code and step through it just as if it were body code. The exit code will normally be reached when there are no more statements to execute in the body code, or following a `return` statement. Without the debugger, the exit code will also be run after an error; however, with the debugger, errors switch the debugger into step mode, but do not move the execution point.

If you quit the debugger via `qqq()`, the exit code will *not* be executed. To avoid this, `skip` to the start of the exit code, `go` to the line number of the final `NULL`, and then type `qqq()`.

Every statement that is executed or typed in while in the body code, will update the return value (except debug-specific commands such as `go()`). Once the debugger has moved into the exit code, the return value does not change. You can find out the return value by typing `get.retval()`. To forcibly change it, first `skip` back to any line in the body code, then call `return` to set the value and return to the exit code block.

Note that if your function completes (i.e. the debugger leaves it without you calling `qqq()`), the return value will be passed on OK and execution will carry on as normal, even if there were error messages during the execution of your function.

Which statements can be traced?

The basic unit of execution in `debug` is the numbered line. Certain statements get individual lines, while other statements are grouped together in a single line, and therefore can't be stepped into. Grouped statements comprise:

- The test expressions in `if`, `while` and `switch` statements.
- The lists of items to be looped over in `for` statements.
- Calls to "normal" (non-flow-control) functions. You can usually `mtrace` the function in question if you want to see what's happening.
- Assignments. If you write code such as `x<- { for( i in 1:5) y<- i/y+y/i; y}`, you're on your own!

The code window

In the present version of `debug`, all you can do in the code window is look at your code. You can scroll up or down and select lines, but this has no effect on debugging. Several code windows can be open at once, not necessarily for different functions (try calling `f` from within `f`), and can be distinguished by the function name and frame number in the title. The window whose number matches the `D(...)>` prompt is currently "active". Some aspects of window position and appearance can be controlled via `options`; see the package documentation.

Strategies for debugging

There are no rules for how to go about debugging, but in most cases I begin by following one of two routes.

When faced with an actual error message, I first call `traceback()` to see where the error occurred; often, this is not in my own functions, even if one of them is ultimately responsible. I then call `mtrace` either on my function, or on the highest-numbered function from `traceback` for which the cause of error is mysterious. Next, I simply retype whatever gave the error, type `go()` when the code window appears, and wait for the crash. The debugger highlights the error line, and I can inspect variables. If it's still not obvious why the crash occurred, I `mtrace` whichever function is invoked by the error line— say `glm`.. Then I press `<ENTER>` to take me into `glm`, and either examine the arguments, or type `go()` again to take me to the error line in `glm`, etc. etc.

If there's no error but `f` is giving strange results, then I `mtrace` and invoke `f`, and step through the code with `<ENTER>`, watching intermediate results and printing any variables I'm curious about. If there is an unproblematic block of code at the start

of `f`, then it's faster to use `go(N)` (or `bp(N)` followed by `go()`) to get to line number `N` after the block; sometimes `N` will be the first line of the exit code. Once I understand roughly where the problem is—as shown by a variable taking on an unexpected value— I will either `qqq()` or patch things up, by setting variables directly from the keyboard and/or using `skip`, just to check that the next few statements *after* the bug are OK.

The fancier features, such as conditional breakpoints and `bp(1,F)`, are used less, but can be invaluable timesavers where loops are involved.

Speed issues

Functions do run slower when `mtraced`, though this is only likely to be a problem if a lengthy loop has to execute before reaching the interesting code. You can speed things up by wrapping innocuous code near the start in an `evalq({...})` call. For instance, if you `mtrace` and run this function

```
slowcrash <- function( x) {
  for( i in 1:(10^5)) x <- x+1
  crash <- x + "oops!"
}
```

then you'll be waiting a long time before `go` arrives at the accident scene. Instead, try `mtrace`ing and running this:

```
fastcrash <- function( x) {
  evalq({ for( i in 1:(10^5)) x <- x+1 })
  crash <- x + "oops!"
}
```

If an error occurs in a complex statement inside a loop, but only after a large number of loop iterations have completed, you can try a different trick. Put the loop contents into an `mlocal` "macro" function (see `mvbutils`) called e.g. `loopee`; replace the guts of the loop with a call to `loopee()`; make sure `loopee` is *untraced*; invoke `f`; and type `go()`. When the debugger stops at the offending iteration, on the `loopee()` line, call `mtrace(loopee)` and press `<ENTER>` to step into `loopee`.

Omissions and limitations

1. `try(...)` statements work fine, but can't yet be stepped into (it's on the to-do list). Fancy error handling via `tryCatch` etc. isn't explicitly handled, and I don't know how the debugger will respond.

2. The debugger endeavours to invisibly replace certain commands with "debug-friendly" equivalents, both in function code and in keyboard input. It is possible to defeat this with really strange code such as `eval(parse(text='next'))`. If you write that sort of code, you'll get what you deserve...

3. A few key `base` functions (and most functions in the debugger) can't be debugged directly. A workaround is to copy them to a new variable and `mtrace` that.

4. There is no "watch window" and no "global expression-triggered breakpoint" facility. Both could be added (the Splus version of `debug` did have a watch window at one point), but would take me some time.

5. Loss of breakpoints after editing is annoying; the Splus version tries harder to keep them, and I might implement the same for R.

6. The code window is very unsophisticated, reflecting my TCL/TK abilities; mark-and-copy would be nice, for example.

7. S4 methods aren't handled (though S3 methods are, even hidden ones); I have no plans to write my own S4 methods, and hope not to have to debug anyone else's!

Feedback on `debug` and `mvbutils` is welcome. I will fix bugs, and will try to add feasible enhancements as time permits.

Mark Bravington
CSIRO Mathematics and Information Sciences
Hobart, Tasmania, Australia
mark.bravington@csiro.au