Source References

by Duncan Murdoch

Abstract Since version 2.10.0, R includes expanded support for source references in R code and '.Rd' files. This paper describes the origin and purposes of source references, and current and future support for them.

One of the strengths of R is that it allows "computation on the language", i.e. the parser returns an R object which can be manipulated, not just evaluated. This has applications in quality control checks, debugging, and elsewhere. For example, the **codetools** package (Tierney, 2009) examines the structure of parsed source code to look for common programming errors. Functions marked by `debug()` can be executed one statement at a time, and the `trace()` function can insert debugging statements into any function.

Computing on the language is often enhanced by being able to refer to the original source code, rather than just to a deparsed (reconstructed) version of it based on the parsed object. To support this, we added source references to R 2.5.0 in 2007. These are attributes attached to the result of `parse()` or (as of 2.10.0) `parse_Rd()` to indicate where a particular part of an object originated. In this article I will describe their structure and how they are used in R. The article is aimed at developers who want to create debuggers or other tools that use the source references, at users who are curious about R internals, and also at users who want to use the existing debugging facilities. The latter group may wish to skip over the gory details and go directly to the section "Using Source References".

The R parsers

We start with a quick introduction to the R parser. The `parse()` function returns an R object of type `"expression"`. This is a list of statements; the statements can be of various types. For example, consider the R source shown in Figure 1.

```
1: x <- 1:10          # Initialize x
2: for (i in x) {
3:   print(i)         # Print each entry
4: }
5: x
```

Figure 1: The contents of 'sample.R'.

If we parse this file, we obtain an expression of length 3:

```
> parsed <- parse("sample.R")
> length(parsed)
[1] 3

> typeof(parsed)

[1] "expression"
```

The first element is the assignment, the second element is the `for` loop, and the third is the single `x` at the end:

```
> parsed[[1]]

x <- 1:10

> parsed[[2]]

for (i in x) {
    print(i)
}

> parsed[[3]]

x
```

The first two elements are both of type `"language"`, and are made up of smaller components. The difference between an `"expression"` and a `"language"` object is mainly internal: the former is based on the generic vector type (i.e. type `"list"`), whereas the latter is based on the `"pairlist"` type. Pairlists are rarely encountered explicitly in any other context. From a user point of view, they act just like generic vectors.

The third element `x` is of type `"symbol"`. There are other possible types, such as `"NULL"`, `"double"`, etc.: essentially any simple R object could be an element.

The comments in the source code and the white space making up the indentation of the third line are not part of the parsed object.

The `parse_Rd()` function parses '.Rd' documentation files. It also returns a recursive structure containing objects of different types (Murdoch and Urbanek, 2009; Murdoch, 2010).

Source reference structure

As described above, the result of `parse()` is essentially a list (the `"expression"` object) of objects that may be lists (the `"language"` objects) themselves, and so on recursively. Each element of this structure from the top down corresponds to some part of the source file used to create it: in our example, `parse[[1]]` corresponds to the first line of 'sample.R', `parse[[2]]` is the second through fourth lines, and `parse[[3]]` is the fifth line.

The comments and indentation, though helpful to the human reader, are not part of the parsed object. However, by default the parsed object does contain a `"srcref"` attribute:

```
> attr(parsed, "srcref")
```

```
[[1]]
x <- 1:10

[[2]]
for (i in x) {
  print(i)          # Print each entry
}

[[3]]
x
```

Although it appears that the "srcref" attribute contains the source, in fact it only references it, and the print.srcref() method retrieves it for printing. If we remove the class from each element, we see the true structure:

```
> lapply(attr(parsed, "srcref"), unclass)

[[1]]
[1] 1 1 1 9 1 9
attr(,"srcfile")
sample.R

[[2]]
[1] 2 1 4 1 1 1
attr(,"srcfile")
sample.R

[[3]]
[1] 5 1 5 1 1 1
attr(,"srcfile")
sample.R
```

Each element is a vector of 6 integers: (first line, first byte, last line, last byte, first character, last character). The values refer to the position of the source for each element in the original source file; the details of the source file are contained in a "srcfile" attribute on each reference.

The reason both bytes and characters are recorded in the source reference is historical. When they were introduced, they were mainly used for retrieving source code for display; for this, bytes are needed. Since R 2.9.0, they have also been used to aid in error messages. Since some characters take up more than one byte, users need to be informed about character positions, not byte positions, and the last two entries were added.

The "srcfile" attribute is also not as simple as it looks. For example,

```
> srcref <- attr(parsed, "srcref")[[1]]
> srcfile <- attr(srcref, "srcfile")
> typeof(srcfile)

[1] "environment"

> ls(srcfile)

[1] "Enc"       "encoding"  "filename"
[4] "timestamp" "wd"
```

The "srcfile" attribute is actually an environment containing an encoding, a filename, a timestamp, and a working directory. These give information about the file from which the parser was reading. The reason it is an environment is that environments are *reference* objects: even though all three source references contain this attribute, in actuality there is only one copy stored. This was done to save memory, since there are often hundreds of source references from each file.

Source references in objects returned by parse_Rd() use the same structure as those returned by parse(). The main difference is that in Rd objects source references are attached to *every* component, whereas parse() only constructs source references for complete statements, not for their component parts, and they are attached to the container of the statements. Thus for example a braced list of statements processed by parse() will receive a "srcref" attribute containing source references for each statement within, while the statements themselves will not hold their own source references, and subexpressions within each statement will not generate source references at all. In contrast the "srcref" attribute for a section in an '.Rd' file will be a source reference for the whole section, and each component part in the section will have its own source reference.

Relation to the "source" attribute

By default the R parser also creates an attribute named "source" when it parses a function definition. When available, this attribute is used by default in lieu of deparsing to display the function definition. It is unrelated to the "srcref" attribute, which is intended to *point to* the source, rather than to *duplicate* the source. An integrated development environment (IDE) would need to know the correspondence between R code in R and the true source, and "srcref" attributes are intended to provide this.

When are "srcref" attributes added?

As mentioned above, the parser adds a "srcref" attribute by default. For this, it is assumes that options("keep.source") is left at its default setting of TRUE, and that parse() is given a filename as argument file, or a character vector as argument text. In the latter case, there is no source file to reference, so parse() copies the lines of source into a "srcfilecopy" object, which is simply a "srcfile" object that contains a copy of the text.

Developers may wish to add source references in other situations. To do that, an object inheriting from class "srcfile" should be passed as the srcfile argument to parse().

The other situation in which source references are likely to be created in R code is when calling

`source()`. The `source()` function calls `parse()`, creating the source references, and then evaluates the resulting code. At this point newly created functions will have source references attached to the body of the function.

The section "Breakpoints" below discusses how to make sure that source references are created in package code.

Using source references

Error locations

For the most part, users need not be concerned with source references, but they interact with them frequently. For example, error messages make use of them to report on the location of syntax errors:

```
> source("error.R")

Error in source("error.R") : error.R:4:1: unexpected
'else'
3:   print( "less" )
4: else
   ^
```

A more recent addition is the use of source references in code being executed. When R evaluates a function, it evaluates each statement in turn, keeping track of any associated source references. As of R 2.10.0, these are reported by the debugging support functions `traceback()`, `browser()`, `recover()`, and `dump.frames()`, and are returned as an attribute on each element returned by `sys.calls()`. For example, consider the function shown in Figure 2.

```
1: # Compute the absolute value
2: badabs <- function(x) {
3:   if (x < 0)
4:     x <- -x
5:   x
6: }
```

Figure 2: The contents of 'badabs.R'.

This function is syntactically correct, and works to calculate the absolute value of scalar values, but is not a valid way to calculate the absolute values of the elements of a vector, and when called it will generate an incorrect result and a warning:

```
> source("badabs.R")
> badabs( c(5, -10) )

[1]   5 -10

Warning message:
In if (x < 0) x <- -x :
  the condition has length > 1 and only the first
  element will be used
```

In this simple example it is easy to see where the problem occurred, but in a more complex function it might not be so simple. To find it, we can convert the warning to an error using

```
> options(warn=2)
```

and then re-run the code to generate an error. After generating the error, we can display a stack trace:

```
> traceback()

5: doWithOneRestart(return(expr), restart)
4: withOneRestart(expr, restarts[[1L]])
3: withRestarts({
       .Internal(.signalCondition(
           simpleWarning(msg, call), msg, call))
       .Internal(.dfltWarn(msg, call))
   }, muffleWarning = function() NULL) at badabs.R#2
2: .signalSimpleWarning("the condition has length
       > 1 and only the first element will be used",
       quote(if (x < 0) x <- -x)) at badabs.R#3
1: badabs(c(5, -10))
```

To read a traceback, start at the bottom. We see our call from the console as line "`1:`", and the warning being signalled in line "`2:`". At the end of line "`2:`" it says that the warning originated "`at badabs.R#3`", i.e. line 3 of the 'badabs.R' file.

Breakpoints

Users may also make use of source references when setting breakpoints. The `trace()` function lets us set breakpoints in particular R functions, but we need to specify which function and where to do the setting. The `setBreakpoint()` function is a more friendly front end that uses source references to construct a call to `trace()`. For example, if we wanted to set a breakpoint on 'badabs.R' line 3, we could use

```
> setBreakpoint("badabs.R#3")

D:\svn\papers\srcrefs\badabs.R#3:
 badabs step 2 in <environment: R_GlobalEnv>
```

This tells us that we have set a breakpoint in step 2 of the function `badabs` found in the global environment. When we run it, we will see

```
> badabs( c(5, -10) )

badabs.R#3
Called from: badabs(c(5, -10))

Browse[1]>
```

telling us that we have broken into the browser at the requested line, and it is waiting for input. We could then examine `x`, single step through the code, or do any other action of which the browser is capable.

By default, most packages are built without source reference information, because it adds quite substantially to the size of the code. However, setting

the environment variable `R_KEEP_PKG_SOURCE=yes` before installing a source package will tell R to keep the source references, and then breakpoints may be set in package source code. The `envir` argument to `setBreakpoints()` will need to be set in order to tell it to search outside the global environment when setting breakpoints.

The `#line` directive

In some cases, R source code is written by a program, not by a human being. For example, `Sweave()` extracts lines of code from Sweave documents before sending the lines to R for parsing and evaluation. To support such preprocessors, the R 2.10.0 parser recognizes a new directive of the form

```
#line nn "filename"
```

where `nn` is an integer. As with the same-named directive in the C language, this tells the parser to assume that the next line of source is line `nn` from the given filename for the purpose of constructing source references. The `Sweave()` function doesn't currently make use of this, but in the future, it (and other preprocessors) could output `#line` directives so that source references and syntax errors refer to the original source location rather than to an intermediate file.

The `#line` directive was a late addition to R 2.10.0. Support for this in `Sweave()` appeared in R 2.12.0.

The future

The source reference structure could be improved. First, it adds quite a lot of bulk to R objects in memory. Each source reference is an integer vector of length 6 with a class and `"srcfile"` attribute. It is hard to measure exactly how much space this takes because much is shared with other source references, but it is on the order of 100 bytes per reference. Clearly a more efficient design is possible, at the expense of moving support code to C from R. As part of this move, the use of environments for the `"srcfile"` attribute could be dropped: they were used as the only available R-level reference objects. For developers, this means that direct access to particular parts of a source reference should be localized as much as possible: They should write functions to extract particular information, and use those functions where needed, rather than extracting information directly. Then, if the implementation changes, only those extractor functions will need to be updated.

Finally, source level debugging could be implemented to make use of source references, to single step through the actual source files, rather than displaying a line at a time as the `browser()` does.

Bibliography

D. Murdoch. Parsing Rd files. 2010. URL `http://developer.r-project.org/parseRd.pdf`.

D. Murdoch and S. Urbanek. The new R help system. *The R Journal*, 1/2:60–65, 2009.

L. Tierney. *codetools: Code Analysis Tools for R*, 2009. R package version 0.2-2.

Duncan Murdoch
Dept. of Statistical and Actuarial Sciences
University of Western Ontario
London, Ontario, Canada
`murdoch@stats.uwo.ca`