Standard nonstandard evaluation rules

Thomas Lumley

December 3, 2002

This document is designed to clarify the various evaluation rules for function arguments in R and to make some suggestions for new code. The descriptions are based on R 1.5.1.

1 Standard evaluation model

R passes arguments by value: the arguments are evaluated in the calling environment and their values are passed to the function. If arguments are not specified then defaults are used and these are evaluated in the environment inside the function, so that local variables are found first, and then variables visible in the environment where the function was defined.

The evaluation of defaults in the environment inside the function is important, but can be abused. In my opinion we should discourage

```
function (formula, data = parent.frame(), ..., subset, ylab = varnames[response],
    ask = TRUE)
```

where the expression for `ylab` refers entirely to variables internal to the function.

[In fact, arguments are passed as promises to compute values rather than the values themselves. The only relevance of this point is in the detailed implementation of nonstandard evaluation rules.]

2 Nonstandard models

Many modelling and graphical functions have a `formula` argument and a `data` argument. If variables in the formula were required to be in the `data` argument life would be a lot simpler, but this requirement was not made when formulas were introduced. Authors of modelling and graphics functions are thus required to implement a limited form of dynamic scope, which they have not done in an entirely consistent way.

The two most common cases are handled in a uniform way across all the R and S-PLUS functions I am aware of

- All the variables in the formula are present in the data object, and there are no vector arguments other than the formula and data object
- All the variables in the formula are present in the data object or in the global environment, the function is called from the global environment, and the formula is specified explicitly (rather than as a variable), and there are no vector arguments other than the formula and data object.

When other vector arguments are given (eg `weights`, `pch`), or the function is not called from the global environment, or the formula was specified as a variable there may be differences between functions and between S dialects. Some of these differences are clearly deliberate, some result from insufficient paranoia on the part of the authors (myself included).

2.1 Most modelling functions

In the call

```
lm(y~x, data=df, weights=w)
```

the variables `x`, `y`, and `w` are looked up in `df` (which can be a list, data frame, or environment) and then in the environment of the formula `y~x`. The environment of the formula is by default the environment it was created in. Most commonly this will be the environment where `lm` was called and in this case R and S-PLUS are compatible. Functions that work this way include `lm`, `aov`, `glm`, the survival functions, `loglm`(MASS), and `gam`(mgcv). [though `gam` is incorrectly documented to use `parent.frame`].

The nonstandard evaluation is usually accomplished by some variation on the following standard idiom

```
mf <- match.call()
mf[[1]] <- as.name("model.frame")
mf$singular.ok <- mf$method <- mf$some.other.arg <- NULL
mf <- eval(mf,parent.frame())
```

The first line gets a copy of the current call. The second replaces the name of the function to be called with `model.frame`. The third line removes arguments that should not be passed to `model.frame` and so have the standard evaluation rules. Finally the constructed call to `model.frame` is evaluated in the calling environment.

One point of variation is whether a specified list of arguments is removed from the call (as above) or whether all but a specified list are removed. In the case above a `...` argument would be passed to `model.frame`, but in many functions the `...` argument is actually passed to a control function (eg `glm.control`, `coxph.control`).

A further infelicity is that the default `na.action` argument specified is a modelling function is not actually used. A side effect of the `match.call()/eval()` procedure is that the default specified by `model.frame` overrides the default specified by, say, `glm()`. One possible fix is to add the following line before the `eval` step above

```
mf$na.action <- substitute(na.action)
```

2.2 Mixed models

The `lme()` function puts all its variables in a call to `model.frame` whose `data` argument is either a specified data frame or the calling environment. However, the formula argument to `model.frame` is constructed inside various nlme utility functions and does not have a useful environment attached to it.

The effect is that the `data` argument is specified, variable lookup is done in that data frame and then in the environment inside `asOneFormula` (for most purposes equivalent to the global environment).

I'm not sure exactly what happens in `nlme`, but the same principle seems to hold as for `lme`: either all variables should be in the supplied data frame or all variables should be in the calling frame and no `data` argument should be used. The documentation can be read to say this, but I don't think it's clear if you don't already know.

2.3 Base graphics

Formula methods for graphics use a similar but not identical scheme; in the call

```
plot(y~x, data=df, col=z)
```

`x` and `y` are looked up in `df` and then the environment of the formula, but the point colours argument `col=z` is looked up first in `df` and then in the *calling* environment.

In this case only the formula is passed to `model.frame`. The additional graphical arguments are evaluated in the `data=df` argument enclosed in the calling environment `parent.frame`. The reason for this more complicated scheme is that `model.frame` requires all the variables to be vectors of the same length and graphical parameters may be scalars or vectors of varying lengths. The inconsistency in enclosing environment is still undesirable.

2.4 Lattice graphics

Lattice uses a slightly different system again. The formula arguments are looked up in the specified `data` argument and then in the environment of `latticeParseFormula`. Other arguments (eg `groups` and `subset` are evaluated in the `data` argument and then in the calling environment. Presumably the use of the environment of `latticeParseFormula` is a minor bug. It causes problems only when a lattice function is called inside another function and one of the arguments is a local variable and a `data=` argument is provided. An artificial example is

```
data(trees)
h <- function(df){
         x <- 1:10
         y <- 1:10
         xyplot(y~x, data=df)
       }
h(trees)
```

3 Functions of models

Functions such as `summary`, `residuals` and so on generally operate as if the model object contained all the necessary data (which in many cases it does). Difficulties arise with functions that refit models.

The `update()` function refits the model by constructing a function call and evaluating it in the calling environment. In some cases this is clearly what users expect, as in this code snippet from MASS

```
ph.fun <- function(data, i) {
  d <- data
  d$calls <- d$fitted + d$res[i]
  coef(update(fit, data=d))
}
```

but in other cases users want to use the original data frame (local or not) and just update the formula (PR#1861).

The `step()` and `stepAIC` functions look up the data in the environment of the model formula, and so (typically) performs the model search using the data from the original fit.

The phrase `model<-update(model)` can be used to refit a model to data in the local environment, even changing the environment associated with the model formula.

4 Variable capture with `with()`

The `with()` function allows an expression to be evaluated with variable lookup in a specified data frame, and then the calling environment. The `plot` example above can be written

```
with(df, plot(y~x,col=z))
```

For interactive use at the command line this is very effective. For programming some care is needed to ensure that variables in the data frame do not accidentally override local variables.

5 Recycling, subsetting and `NA` removal

A further difficulty in handling the nonstandard evaluation mechanisms is the removal of missing values and the use of the `subset` argument. The modelling functions accept a `na.action` argument specifying how to handle missing values. If rows of the model frame containing missing values are removed (as is the default), it is not clear whether the same rows of other arguments should be removed. Similarly, the `subset` argument applied to the variables defined in the formula may or may not be applied to the other nonstandardly evaluated arguments. Differences of opinion exist within R-core on the correct behaviour, and each possibility makes some things hard. For interactive use it is possible to get around the difficulties using `with`, but this is harder when programming because of the possibilities of unintended variable capture.

The current implementation is that functions apply the `subset` argument to all these arguments. The `na.action` argument is not needed in base graphics functions (as NAs are not plotted); in the modelling functions, all rows with missing values in either the formula variables or other variables such as `weights` are removed. I think this is the wrong behaviour for graphics functions, but the right behaviour for modelling functions if the extra arguments are things like `weights` and `strata` that are conceptually part of the model frame. On the other hand, having different behaviour for the two classes of functions is difficult.

If subsetting and NA removal are done then a further decision is needed about the recycling rule; should it be applied before or after the final subset is taken? In the base graphics functions it happens afterwards, which I think is wrong:

```
x <- 1:10
y <- 1:10
z <- 1:2
plot(x~y, col=z)
plot(x~y, col=z, subset=2*(1:5))
df <- data.frame(x,y,z)
plot(x~y, col=z, data=df, subset=2*(1:5))
```

The first two plot both red and black points, the last plots only red points.

6 Proposals

The ambiguity in evaluation rules arises because some arguments need to be evaluated according to formula/data rules and some don't. One possible solution for new code is to pass formulas or quoted expressions when the standard variable lookup is not to be used.

That is, a new modelling function

```
xyzlm(y~x, data=df, foo=z)
```

would look up `z` in the calling environment. If the `xyzlm` function wants to look up `z` in `df` it should specify one of the following

```
xyzlm(y~x, data=df, foo=~z)
xyzlm(y~x, data=df, foo=quote(z))
xyzlm(y~x, data=df, foo=expression(z))
```

This allows more flexibility than the current system and is not ambiguous as the evaluation rules in the function call are standard. A possible refinement would be to say that a formula argument takes part in subsetting and NA removal but an expression argument does not.

People should be encouraged to

1. Thorougly document nonstandard evaluation if it can't be avoided
2. Use the environment of a suitable formula as the enclosing environment when evaluating in a data frame.
3. Use standard patterns (like the `model.frame/eval` one) where possible (to keep the insanity localised)
4. For new code, where possible, pass formulas or quoted expressions when the standard variable lookup is not desired. It would be useful to have a single function like `model.frame` that does the necessary evaluation.

At a minimum, lattice and base graphics should use the same evaluation rules, and it probably makes sense for them to use the environment of the formula as the enclosing environment for compatibility with `model.frame`.