Frames, Environments, and Scope in R and S-PLUS[*]

Appendix to *An R and S-PLUS Companion to Applied Regression*

John Fox

March 2002

1 Introduction

Section 2.2.1 of the text describes in some detail how objects are located along the search path in R and S-PLUS. I believe that the material presented there suffices for the everyday use of S in data analysis. Elsewhere in the text — for example, in describing local functions in Section 8.3.4 — I occasionally make reference to the different 'scoping' rules in R and S-PLUS.

The object of this Appendix is to provide a slightly deeper discussion of this issue, in particular with respect to the manner in which the values of variables are determined when functions are executed. This material, while difficult, is occasionally important in writing S programs. More complete treatments are available for S-PLUS in Becker, Chambers, and Wilks (1988: Sec. 5.4), and for both R and S-PLUS in Venables and Ripley (2000: Sec. 3.4).[1]

2 Basic Definitions: Frames, Environments, and Scope

Several general concepts are useful for understanding how S assigns values to variables. These definitions are adapted from Abelson, Sussman, and Sussman (1985), where they are developed in much greater detail. A good briefer discussion of most of these concepts may be found in Tierney (1990: Sec. 3.5).

- A variable that is assigned a value is said to be *bound* to that value. Variables are normally bound to values by assignments (e.g., `x <- 5`), or by passing values to function arguments [e.g., `f(x=2)`]. In the latter instance, the binding is local to the function call.
- A *frame* is a set of bindings. A variable may have at most one binding in a particular frame, but the same variable may be bound to different values in different frames. (In S, the same variable may be bound to a function definition and to an object of another mode, such as a numeric vector or list. The S interpreter is able to distinguish between the two values by context.)
- If a variable is *unbound* in a particular frame it is said to be a *free variable* in that frame. For example, when the function `f <- function (x) x + a` is called as `f(2)`, `x` is bound to the value `2` in the local frame of the function call, but `a` is a free variable in that frame. Likewise, the assignment `x <- 5` made at the command prompt binds the value `5` to `x` in the global frame. In R, the global frame is called the *global environment* or the *workspace*, and is kept in memory. This usage conflicts with the definition of the term 'environment' given below. In S-PLUS, global bindings are made in the *working directory* on disk.

[*]I am grateful to Robert Stine of the University of Pennsylvania for helpful comments on this appendix.

[1]As in the text, I use 'S-PLUS' as a shorthand for versions 3 and 4 of S, which correspond, for example, respectively to S-PLUS 2000 and 6.0 for Windows. I use 'S' more generally to denote both R and S-PLUS.

- *Scoping rules* determine where the interpreter looks for values of free variables. The scoping rules in R and S-PLUS are different, and are explained below. In the previous example, however (assuming that the function `f` was defined in the global frame), both the R and S-PLUS interpreters would look for the free variable `a` in the global frame and subsequently on the rest of the search path.[2]

- An *environment* is a sequence of frames. A value bound to a variable in a frame earlier in the sequence will take precedence over a value bound to the same variable in a frame later in the sequence. The first value is said to *shadow* or *mask* the second. This idea is familiar from the discussion in Section 2.2.1 of the search path in S. Indeed, the frames on the search path, starting with the global frame, are at the end of the frame-sequence of *every* environment. Therefore, variables that are bound to values in frames on the search path are globally visible, unless shadowed by bindings earlier in the sequence. I will call this sequence of frames the *global environment*. As noted, this usage conflicts with standard R terminology, in which the single frame of the workspace is called the 'global environment.'

- The *scope* of a variable binding is the set of environments in which it is visible.

In the literature describing R and S-PLUS (such as the references in Section 1), the terms *frame* and *environment* are used somewhat differently from the definitions given here, but my (more general) terminology serves our current purpose, and, in particular, facilitates comparisons between R and S-PLUS. For example, in the literature on S-PLUS, there is a distinction between *frames*, which exist in memory and hence are transient, and *databases* on the search path, which are libraries or lists (most commonly, data frames), but both are frames in the sense that they associate variables with values.

3 Scoping Rules in R and S-PLUS

In R, the environment of a function (i.e., the environment created by a function call) comprises the local frame of the function call followed by the environment in which the function was defined (the *enclosing environment*); this rule is called *lexical* or *static scoping*. A function together with its environment is termed a *closure*. In contrast, in S-PLUS, the environment of a function consists of the local frame of the function call followed directly by the global environment.[3] The following examples (some of them adapted from Tierney, 1990: Sec. 3.5) elucidate the consequences of this distinction.

I begin with a simple illustration, introduced in the preceding section:

```
> f <- function (x) x + a
> a <- 10
> x <- 5
> f(2)
[1] 12
```

When `f` is called, the local binding of $x \equiv 2$, shadows the global binding $x \equiv 5$. The variable `a` is a free variable in the frame of the function call, and so the global binding $a \equiv 10$ applies.[4] This example, diagrammed in Figure 1, produces an identical result in R and S-PLUS.

Now consider an example in which one function calls another:

```
> f<- function (x) {
+     a <- 5
+     g(x)
```

[2] This description is slightly simplified for S-PLUS. There are special frames, called the *top-level frame* and the *session frame*, that are interrogated before the global frame (i.e., the working directory in S-PLUS). Assignments made at the command prompt are evaluated in the top-level frame, and only committed to the global frame if an expression executes without error. Certain variables, such as options, are held in the session frame. Both the top-level frame and the session frame reside in memory. For most purposes, we may think of these special frames as part of the global frame.

[3] Another common rule is *dynamic scoping*, according to which the environment of a function comprises the local frame of the function followed by the environment from which the function was *called* (not *defined*, as in lexical scoping). Neither R nor S-PLUS employs dynamic scoping. Dynamic scoping is less powerful than lexical scoping for some purposes, but it is arguably more intuitive.

[4] To avoid confusion, I use $\equiv$ to represent a binding.

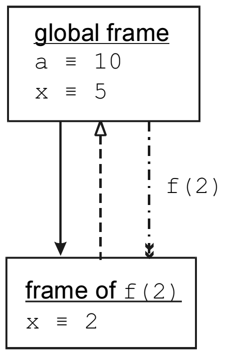

Figure 1: Environment for the function call `f(2)`. Each box represents a frame, with variables bound in the frame shown within the box. The solid arrow represents a function definition: The function `f` is defined in the global frame. The broken arrow represents the sequence of frames comprising the environment of the function call: The variable `a`, unbound in the frame of the call, is located in the global frame, while the local binding of `x` shadows the global binding. The dashed-dotted arrow represents the call; `f` is called from the global frame.

```
+     }

> g <- function(y) y + a

> f(2)
[1] 12
```

(I have used different names for the arguments of `f` and `g` — respectively, `x` and `y` — to emphasize the fact that argument names are arbitrary.) Again, R and S-PLUS produce the same result, but for subtly different reasons:

- In R, the global binding `a` ≡ `10` is used when `f` calls `g`, because `a` is a free variable in `g`, and `g` is defined at the command prompt in the global frame.
- In S-PLUS, the global binding `a` ≡ `10` is used simply because `a` is a free variable in `g`, and the S-PLUS interpreter *always* looks to the global frame after the local frame of the function call.

Note that in both cases the binding `a` ≡ `5` in the local frame of `f` is ignored (see Figure 2).[5]

The next illustration, employing a locally defined function, reveals the difference in the scoping rules for R and S-PLUS. Beginning with R:

```
> f <- function (x) {
+     a <- 5
+     g <- function (y) y + a
+     g(x)
+     }

> f(2)
[1] 7
```

The local function `g` is defined within the function `f`, and so the environment of `g` comprises the local frame of `g` followed by the environment of `f`. Because `a` is a free variable in `g`, the interpreter next looks for a value for `a` in the local frame of `f`; it finds the value `a` ≡ `5`, which shadows the global binding `a` ≡ `10` (see Figure 3).

In contrast, in S-PLUS:

[5] If R or S-PLUS were dynamically scoped (as described in note 3), then when `g` is called from `f` the interpreter would look first for a free variable in the frame of `f`.

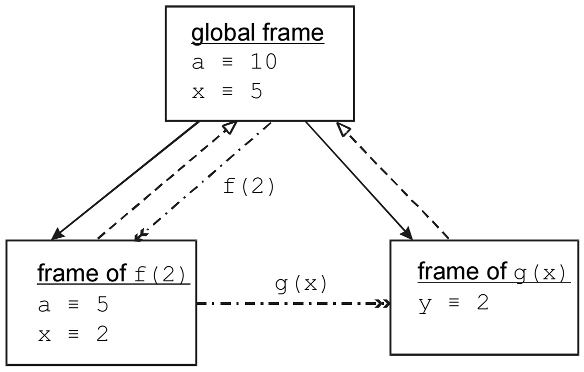

Figure 2: In this case, both `f` and `g` are defined in the global frame; `f` is called from the global frame, while `g` is called from `f`. The free variable `a` in `g` gets its value from the global frame in both R and S-PLUS.

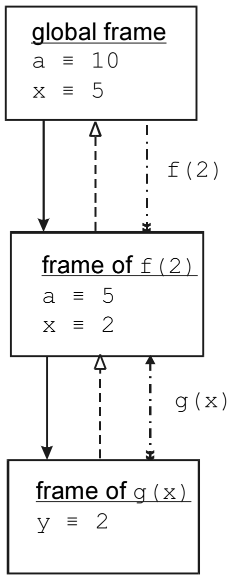

Figure 3: Lexical scoping in R: `g` is a local function defined in `f` and called from `f`. The free variable `a` in `g` is located in the frame of `f(2)`, and shadows a variable by the same name in the global frame.

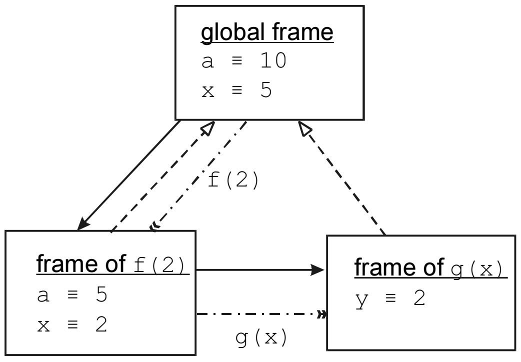

Figure 4: Scoping in S-PLUS: Even though `g` is a local function defined in `f` and called from `f`, the free variable `a` in `g` takes its value from `a` in the global frame.

```
> f <- function (x) {
+     a <- 5
+     g <- function (y) y + a
+     g(x)
+     }

> f(2)
[1] 12
```

When it encounters the free variable `a` in `g`, the S-PLUS interpreter ignores the binding `a` ≡ `5` in `f` and looks instead to the global environment, where it finds the binding `a` ≡ `10`. (Figure 4). Consequently, to pass a local variable to a local sub-function in S-PLUS, it is most straight-foward to incorporate the variable as an argument to the sub-function:

```
> f <- function (x) {
+     a <- 5
+     g <- function (y, b) y + b
+     g(x, a)
+     }

> f(2)
[1] 7
```

This version produces identical results in R and S-PLUS (as diagrammed in Figure 5).

Because locally defined variables are visible to local functions, the lexical scoping rule of R is somewhat more convenient than the scoping rule employed by S-PLUS. Lexical scoping is also more powerful in certain circumstances. Consider the following R function:

```
> make.power <- function(power){
+     function(x) x^power
+     }
>
```

The `make.power` function returns a closure as its result:

```
> square <- make.power(2)
> square
function(x) x^power
<environment: 01485604>
```

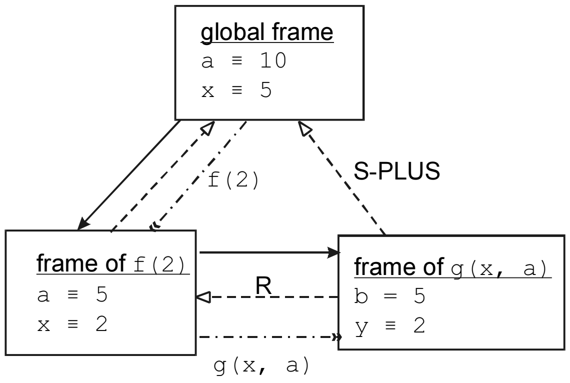

Figure 5: The function `g` is a local function defined in and called from `f`. Because there are no free variables in `g`, however, it does not matter that the environment of `g` is different in R and S-PLUS.

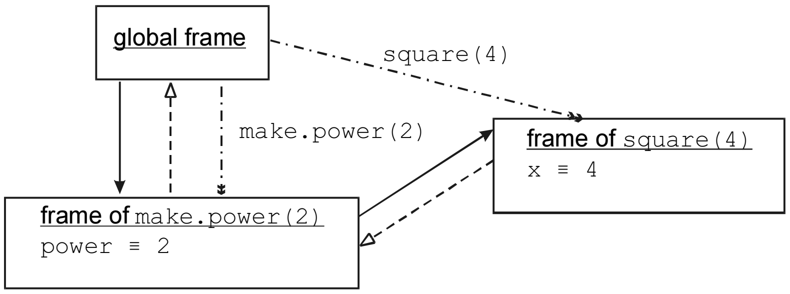

Figure 6: Lexical scoping in R: The function `square` is defined in the frame of `make.power(2)` and thus its environment includes this frame. The variable `power` is free in the frame of `square(4)`, but the binding `power` ≡ `2` is located in the frame of `make.power(2)`, even though `square` is called from the global frame.

```
> square(4)
[1] 16

> cuberoot <- make.power(1/3)
> cuberoot
function(x) x^power
<environment: 01486AE0>

> cuberoot(64)
[1] 4
```

Notice what happens here (Figure 6): When `make.power` is called with the argument `2` (or `1/3`), this value is bound to the local variable `power`. The function that is returned is defined in the local frame of the call to `make.power` and therefore inherits the environment of this call, including the binding of `power`.

Because S-PLUS is not lexically scoped, this procedure fails (cryptically):[6]

```
> make.power <- function(power){
+     function(x) x^power
+     }
>
```

[6] The output is from S3 (S-PLUS 2000), but a similar error is produced in S4 (S-PLUS 6.0).

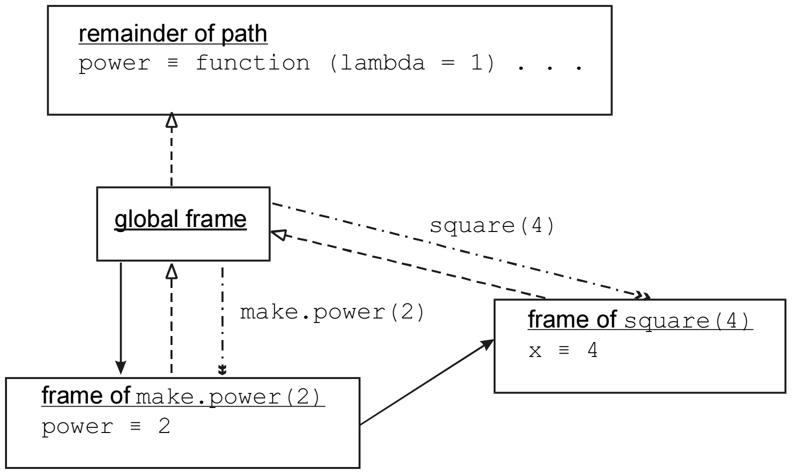

Figure 7: Scoping in S-PLUS: Although `square` is defined in the frame of `make.power(2)`, it tries to resolve the reference to the free variable `power` in the global frame, and failing that, along the remainder of the search path. Eventually, it finds a *function* bound to `power`, causing `square(4)` to fail.

```
> square <- make.power(2)
> square
function(x)
x^power

> square(4)
Error in x^power: Non-numeric second operand
```

In S-PLUS, `make.power` returns a *function* rather than a *closure* (i.e., a function together with an environment): In the function `square`, `power` is a free variable, not bound to the value `2`. Consequently, when `square` is called, the S-PLUS interpreter looks for a binding for `power` in the global environment; it finds a function called `power`, producing an error when it tries to use this function as if it were a numerical exponent (see Figure 7).

References

Abelson, H., G. J. Sussman & J. Sussman. 1985. *Structure and Interpretation of Computer Programs.* Cambridge MA: MIT Press.

Becker, R. A., J. M. Chambers & A. R. Wilks. 1988. *The New S Language: A Programming Environment for Data Analysis and Graphics.* Pacific Grove CA: Wadsworth.

Tierney, L. 1990. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* New York: Wiley.

Venables, W. N. & B. D. Ripley. 2000. *S Programming.* New York: Springer-Verlag.