Programmers' Niche: The Y of R

Vince Carey

Introduction

Recursion is a general method of function definition involving self-reference. In R, it is easy to create recursive functions. The following R function computes the sum of numbers from 1 to n inclusive:

```
>   s <- function(n) {
+     if (n == 1) return(1)
+     return(s(n-1)+n)
+   }
```

Illustration:

```
>   s(5)

[1] 15
```

We can avoid the fragility of reliance on the function name defined as a global variable, by invoking the function through `Recall` instead of using self-reference.

It is easy to accept the recursion/`Recall` facility as a given feature of computer programming languages. In this article we sketch some of the ideas that allow us to implement recursive computation without named self-reference. The primary inspiration for these notes is the article "The Why of Y" by Richard P. Gabriel (Gabriel, 1988).

Notation. Throughout we interpret $=$ and `=` as the symbol for mathematical identity, and use `<-` to denote programmatic assignment, with the exception of notation for indices in algebraic summations (e.g., $\sum_{i=1}^{n}$), to which we give the usual interpretation.

The punch line

We can define a function known as the applicative-order fixed point operator for functionals as

```
> Y <- function(f) {
+     g <- function(h) function(x) f(h(h))(x)
+     g(g)
+ }
```

The following function can be used to compute cumulative sums in conjunction with `Y`:

```
>   csum <- function(f) function(n) {
+     if (n < 2) return(1);
+     return(n+f(n-1))
+   }
```

This call

```
>   Y(csum)(10)

[1] 55
```

computes $\sum_{k=1}^{10} k$ without iteration and without explicit self-reference.

Note that `csum` is not recursive. It is a function that accepts a function `f` and returns a function of one argument. Also note that *if* the argument passed to `csum` is the "true" cumulative sum function, which we'll denote K, then `csum(`K`)(n)` will be equal to K`(n)`. In fact, since we "know" that the function `s` defined above correctly implements cumulative summation, we can supply it as an argument to `csum`:

```
>    csum(s)(100)

[1] 5050
```

and we can see that

```
>    csum(s)(100) == s(100)

[1] TRUE
```

We say that a function f is a fixed point of a functional F if $F(f)(x) = f(x)$ for all relevant x. Thus, in the calculation above example, we exhibit an instance of the fact that `s` is a fixed point of `csum`. This is an illustration of the more general fact that K (the true cumulative summation function) is a fixed point of `csum` as defined in R.

Now we come to the crux of the matter. `Y` computes a fixed point of `csum`. For example:

```
> csum(Y(csum))(10) == Y(csum)(10)

[1] TRUE
```

We will show a little later that `Y` is the applicative-order fixed point operator for functionals `F`, meaning that `Y(F)(x)` $=$ `F(Y(F))(x)` for all suitable arguments `x`. This, in conjunction with a uniqueness theorem for "least defined fixed points for functionals", allows us to argue that K (a fixed point of `csum`) and `Y(csum)` perform equivalent computations. Since we can't really implement K (it is an abstract mathematical object that maps, in some unspecified way, the number n to $\sum_{k=1}^{n} k$) it is very useful to know that we can (and did) implement an equivalent function in R.

Peeking under the hood

One of the nice things about R is that we can interactively explore the software we are using, typically by mentioning the functions we use to the interpreter.

```
> s

function(n) {
    if (n == 1) return(1)
    return(s(n-1)+n)
  }
```

We have argued that `Y(csum)` is equivalent to K, so it seems reasonable to define

```
> K <- Y(csum)
> K(100)

[1] 5050
```

but when we mention this to the interpreter, we get

```
> K

function (x)
f(h(h))(x)
<environment: 0x1494d48>
```

which is not very illuminating. We can of course drill down:

```
> ls(environment(K))

[1] "h"

> H <- get("h", environment(K))
> H

function (h)
function(x) f(h(h))(x)
<environment: 0x1494e0c>

> get("f", environment(H))

function(f) function(n) {
    if (n < 2) return(1);
    return(n+f(n-1))
  }

> get("g", environment(H))

function (h)
function(x) f(h(h))(x)
<environment: 0x1494e0c>
```

The lexical scoping of R (Gentleman and Ihaka, 2000) has given us a number of closures (functions accompanied by variable bindings in environments) that are used to carry out the concrete computation specified by `K`. `csum` is bound to `f`, and the inner function of `Y` is bound to both `h` and `g`.

Self-reference via self-application

Before we work through the definition of `Y`, we briefly restate the punch line: `Y` transforms a functional having K as fixed point into a function that implements a recursive function that is equivalent to K. We want to understand how this occurs.

Define

```
>  ss <- function(f)function(n) {
+    if (n==1) return(1)
+    return(n+f(f)(n-1))
+ }
>  s(100)  # s() defined above

[1] 5050

>  ss(ss)(100)

[1] 5050
```

`s` is intuitively a recursive function with behavior equivalent to K, but relies on the interpretation of `s` in the global namespace at time of execution.

`ss(ss)` computes the same function, but avoids use of global variables. We have obtained a form of self-reference through self-application. This could serve as a reasonable implementation of K, but it lacks the attractive property possessed by `csum` that `csum(K)(x)` $=$ `K(x)`.

We want to be able to establish functional self-reference like that possessed by `ss`, but without requiring the seemingly artificial self-application that is the essence of `ss`.

Note that `csum` can be self-applied, but only under certain conditions:

```
> csum(csum(csum(csum(88))))(4)

[1] 10
```

If the outer argument exceeded the number of self-applications, such a call would fail. Curiously, the argument to the innermost call is ignored.

Y

We can start to understand the function of `Y` by expanding the definition with argument `csum`. The first line of the body of `Y` becomes

```
g <- function(h) function(x) csum(h(h))(x)
```

The second line can be rewritten:

```
g(g) = function(x) csum(g(g))(x)
```

because by evaluating `g` on argument `g` we get to remove the first `function` instance and substitute `g` for `h`.

If we view the last "equation" syntactically, and pretend that `g(g)` is a name, we see that `Y(csum)` has created something that looks a lot like an instance of recursion via named self-reference. Let us continue with this impression with some R:

```
> cow <- function(x) csum(cow)(x)
> cow(100)

[1] 5050
```

`Y` has arranged bindings for constituents of `g` in its definition so that the desired recursion occurs without reference to global variables.

We can now make good on our promise to show that `Y` satisfies the fixed point condition `Y(F)(x) = F(Y(F))(x)`. Here we use an R-like notation to express formal relationships between constructs described above. The functional `F` is of the form

```
F = function(f) function(x) m(f,x)
```

where `m` is any R function of two parameters, of which the first may be a function. For any R function `r`,

```
F(r) = function(x) m(r,x)
```

Now the expression `Y(F)` engenders

```
g = function(h) function(x)F(h(h))(x)
```

and returns the value `g(g)`. This particular expression binds `g` to `h`, so that the value of `Y(F)` is in fact

```
function(x) F(g(g))(x)
```

Because `g(g)` is the value of `Y(F)` we can substitute in the above and find

```
Y(F) = function(x)Y(F)(x)
     = function(x) F(Y(F))(x)
```

as claimed.

Exercises

1. There is nothing special about recursions having the simple form of `csum` discussed above. Interpret the following mystery function and show that it can be modified to work recursively with `Y`.

```
> myst = function(x) {
+     if (length(x) == 0)
+         return(x)
+     if (x[1] %in% x[-1])
+         return(myst(x[-1]))
+     return(c(x[1], myst(x[-1])))
+ }
```

2. Extend `Y` to handle mutual recursions such as Hofstader's male-female recursion:

$$F(0) = 1, M(0) = 0,$$

$$F(n) = n - M(F(n-1)),$$

$$M(n) = n - F(M(n-1)).$$

3. Improve R's debugging facility so that debugging `K` does not lead to an immediate error.

Bibliography

R. Gabriel. The why of Y. *ACM SIGPLAN Lisp Pointers*, 2:15–25, 1988.

R. Gentleman and R. Ihaka. Lexical scope and statistical computing. *JCGS*, 9:491–508, 2000.