Inside-Out: Recursion vs. Function Application

(Draft 1)

Ted Szylowiec

Recursion is just function applications turned inside-out. More specifically, certain forms of multiple recursion, when imagined to be turned inside-out, become repeated applications of multiple return-value functions. What do we mean when we say that repeated function application is recursion "inside out"? We will examine this analogy through examples. The code in this article is Scheme R6RS and the implementation used is Guile 2.0.11.

First, we need a way to compose a multiple return-value function with itself repeatedly. Scheme provides the procedure `values` which acts like the multiple return-value version of the identity function. Folding a list of functions with `values` as the initial value for the accumulator does the job:

```
(define (many-compose . funcs)
  (fold compose values funcs))
```

To see how it works, let's use it on a function that takes a single argument and returns a single value: $r(x) = x^2$. Applying this function repeatedly four times gives

$$r(r(r(r(x)))) = x^{16}.$$

In code, we can do the same:

```
(define (sq u) (* u u))
(define sq4 (many-compose sq sq sq sq))
> (sq4 2)
65536
```

We will use `many-compose` in all of our examples.

1 Complex numbers

Rather than iterating r on a real number x, we can do it n times to a complex number $a_0 + ib_0$. As a function of n, the result would be

$$f(n) = (a_0 + ib_0)^{2^n}.$$

For example if $a_0 = 2$ and $b_0 = 3$, then iterating four fimes gives

$$f(4) = -815616479 - 13651680\,i.$$

Examining $(a_0 + ib_0)^{2^n}$ for the first few n, and with a little algebra, we can find recursive functions $a(n)$ and $b(n)$ such that

$$f(n) = a(n) + ib(n).$$

These are

$$\begin{aligned} a(n) &= a(n-1)^2 - b(n-1)^2 \\ b(n) &= 2\,a(n-1)\,b(n-1). \end{aligned} \tag{1}$$

with the initial conditions

$$\begin{aligned} a(0) &= a_0 \\ b(0) &= b_0. \end{aligned}$$

The relations (1) define mutually recursive functions a and b. They both depend on each other. This can be translated into mutually recursive procedures that calculate $f(n)$. We first introduce some symmetry between arguments and return values: `make-f` will take the two initial values a_0 and b_0 as arguments, and with that, create a function f that returns two values: $a(m)$ and $b(m)$.

```
(define (make-f a0 b0)
  (lambda (m)
    (letrec ((a (lambda (n)
                  (if (zero? n)
                      a0
                      (- (sq (a (- n 1)))
                         (sq (b (- n 1)))))))
             (b (lambda (n)
                  (if (zero? n)
                      b0
                      (* 2
                         (a (- n 1))
                         (b (- n 1)))))))
      (values (a m)
              (b m))))))
```

Now let's consider a function F that takes the current state, say $a(n), b(n)$ and returns the next, $a(n+1), b(n+1)$. The scheme for this function can be read off from (1):

```
(define (F a b)
  (values (- (sq a) (sq b))
          (* 2 a b)))
```

Notice the symmetry: two arguments and two return values. This makes it possible to compose `F` with itself repeatedly:

```
(define (make-Fn n)
  (apply many-compose (make-list n F)))
```

Now test all this, to make sure the recursive f does the same thing as repeated applications of the function F.

```
> (define myf (make-f 2 3))
> (myf 4)
-815616479
-13651680

> (define myFn (make-Fn 4))
> (myFn 2 3)
-815616479
-13651680
```

We have turned the mutual recursion inside-out and made it into repeated applications of a multiple return-value function.

But what if we want to raise $a_0 + ib_0$ to any power n, not merely powers of 2^n? Can we do it the same way, by mutual recursion? Yes. First examine complex multiplication of $z = a + ib$ with $z' = a' + ib'$:

$$zz' = (aa' - bb') + i(a'b + ab'). \tag{2}$$

We can define a closure that creates a function G which multiplies its aruments according to (2) and outputs two values corresponding to the real and imaginary parts of the result. This way, the function G is symmetrical in its arguments and its return values.

```
(define (make-G a1 b1)
  (lambda (a0 b0)
    (values (- (* a1 a0)
               (* b1 b0))
            (+ (* a1 b0)
               (* b1 a0)))))
```

With `many-compose`, we can create a function Gn that applies G exactly n times to some initial a_0 and b_0. In other words, Gn will multiply $a_0 + ib_0$ n times by some $a + ib$ which was made part of the closure G.

```
(define (make-Gn n G)
  (apply many-compose
         (make-list n G))))
```

For example, if we define G to be the function that multiplies something by $2 + 3i$, then applying G 15 times to $2 + 3i$ should give $(2 + 3i)^{16}$.

```
> (define G (make-G 2 3))
> (define G15 (make-Gn 15 G))
> (G15 2 3)
-815616479
-13651680
```

Which is correct.

The above analysis using repeated function applications should give us enough clues for constructing a mutually recursive procedure that does the same thing. From the fact that G takes two arguments and returns two values, we can guess that the mutual recursion will involve two functions, and the schema for these functions will look a lot like the insides of function G.

```
(define (make-g m a1 b1)
  (lambda (a0 b0)
    (letrec
      ((a (lambda (n)
            (if (zero? n)
                a0
                (- (* a1 (a (- n 1)))
                   (* b1 (b (- n 1)))))))
       (b (lambda (n)
            (if (zero? n)
                b0
                (+ (* a1 (b (- n 1)))
                   (* b1 (a (- n 1)))))))
      (values (a m)
              (b m))))))
```

This creates a closure that multiplies something n times by $a_1 + ib_1$. Let's create a g that multiplies something fifteen times by $2 + 3i$, and apply it to $2 + 3i$. We should get the same as above.

```
> (define g (make-g 15 2 3))
> (g 2 3)
-815616479
-13651680
```

And there we have it: repeated applications of G have been turned inside-out into a mutually recursive procedure g which does the same thing.

2 Pauli matrices

Any 2x2 matrix q with real or complex elements can be represented as a linear combination

$$q = wI + x\sigma_1 + y\sigma_2 + z\sigma_3$$

where I is the identity matrix and σ_1, σ_2, σ_3 are the Pauli spin matrices:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$
$$\sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

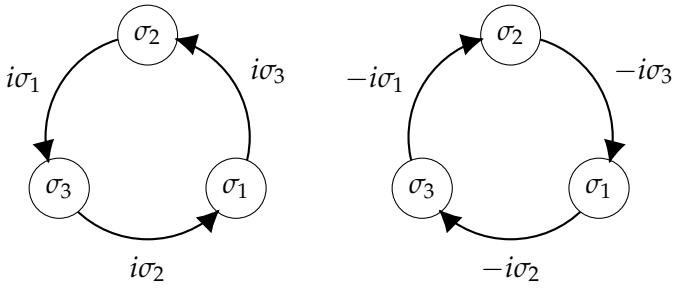
The Pauli matrices obey the following multiplication rules:

$$\sigma_1\sigma_2 = i\sigma_3 \qquad \sigma_2\sigma_1 = -i\sigma_3$$
$$\sigma_2\sigma_3 = i\sigma_1 \qquad \sigma_3\sigma_2 = -i\sigma_1$$
$$\sigma_3\sigma_1 = i\sigma_2 \qquad \sigma_1\sigma_3 = -i\sigma_2$$

and

$$\sigma_1^2 = \sigma_2^2 = \sigma_3^2 = I^2 = I.$$

These multiplication rules are summarized by the following easy-to-remember counter-clockwise and clockwise diagrams:



Suppose q and q' are two 2x2 matrices expressed as Pauli combinations:

$$q = wI + x\sigma_1 + y\sigma_2 + z\sigma_3$$
$$q' = w'I + x'\sigma_1 + y'\sigma_2 + z'\sigma_3.$$

Multiplication of q and q' goes like this:

$$qq' = AI + B\sigma_1 + C\sigma_2 + D\sigma_3$$

where

$$\begin{aligned} A &= ww' + xx' + yy' + zz' \\ B &= w'x + wx' + i\,(y'z - yz') \\ C &= w'y + wy' + i\,(z'x - zx') \\ D &= w'z + wz' + i\,(x'y - xy') \end{aligned} \tag{3}$$

Multiplication is, in general, non-commutative, so qq' is usually not the same as $q'q$.

We are interested in finding a recursive procedure for successive squaring of a Pauli combination q, analogous to (1) which we found for complex numbers. If we set $q' = q$ in (3) and do some algebra, we find:

$$\begin{aligned} A &= w^2 + x^2 + y^2 + z^2 \\ B &= 2wx \\ C &= 2wy \\ D &= 2wz. \end{aligned} \tag{4}$$

This gives a way to calculate

$$(wI + x\sigma_1 + y\sigma_2 + z\sigma_3)^{2^n}$$

by repeated applications of a function H which we will now define. While the complex version of this function took two arguments and returned two values, the Pauli combination version H will take four arguments and return four values. The definition of H follows from the scheme in (4).

```
(define (G w x y z)
  (values (+ (sq w) (sq x) (sq y) (sq z))
          (* 2 w x)
          (* 2 w y)
          (* 2 w z)))
```

The higher-order function `make-Hn` creates a procedure that applies H n times to its arguments. We can test the idea on $q = I + 2\sigma_1 + 3\sigma_2 + 4\sigma_3$ by iterating q^2 four times, which should give q^{16}.

```
(define (make-Hn n)
  (apply many-compose (make-list n H)))
```

```
> (define H4 (make-Hn 4))
> (H4 1 2 3 4)
3826362843136
1414131548160
2121197322240
2828263096320
```

These are the correct values for the coefficients A, B, C and D if the squaring scheme (4) is applied four times to the Pauli combination q.

As usual, the mutually recursive version of this, h, can be deduced from H. It will be a four-way mutual recursion, and we will return four values just to keep the symmetry.

```
(define (make-h w0 x0 y0 z0)
  (lambda (m)
    (letrec
      ((w (lambda (n)
            (if (zero? n)
                w0
```

```
            (+ (sq (w (- n 1)))
               (sq (x (- n 1)))
               (sq (y (- n 1)))
               (sq (z (- n 1)))))))
      (x (lambda (n)
           (if (zero? n)
               x0
               (* 2
                  (w (- n 1))
                  (x (- n 1))))))
      (y (lambda (n)
           (if (zero? n)
               y0
               (* 2
                  (w (- n 1))
                  (y (- n 1))))))
      (z (lambda (n)
           (if (zero? n)
               z0
               (* 2
                  (w (- n 1))
                  (z (- n 1)))))))
    (values (w m) (x m) (y m) (z m)))))
```

Let's try it out.

```
> (define g (make-g 1 2 3 4))
> (g 4)
3826362843136
1414131548160
2121197322240
2828263096320
```

As expected. Here are two suggestions for exercises.

Using the relations (3), develop recursive and function-application procedures to raise q to any power n, analogous to what was done for complex numbers in (2).

Develop recursive and function-application procedures for successive squaring and powering of quaternions. Quaternions are very similar to Pauli combinations, so this shouldn't be too difficult.

3 Fibonacci numbers

The current Fibonacci number is the sum of the two previous ones. In a Fibonacci sequence, the two most current elements are used to find the next two most current elements. If a is the most current element and b is the previous one, the process looks like this:

$$a, b \to a + b, a \tag{5}$$

The current becomes the previous, and the new current is the sum of the current and previous. Confusing? Not if we rely on the schematic formula above, from which we can immediately build a function that, when iterated, returns Fibonacci number pairs. This function will take two arguments and return two values.

```
(define (Fib a b)
  (values (+ a b) a))
```

With the help of `many-compose`, `Fib` can be iterated to produce the nth fibonacci number (and also the one before it. The iterations begin with initial values $a = 1$ and $b = 1$. Two Fibonacci numbers are accounted for, therefore we need to apply `Fib` exactly $n - 2$ times to get the nth Fibonacci number and it's predecessor.

```
(define (Fibn n)
  ((apply many-compose
          (make-list (- n 2) Fib)) 1 1))
```

If all you want is, say, the 100th Fibonacci number, then this is just about the easiest way to get it. No memoization is necessary.

```
> (Fibn 100)
354224848179261915075
218922995834555169026
```

The first value returned is the 100th Fibonacci number. The second value returned is the 99th Fibonacci number.

We can turn these function applications inside-out to create a mutually recursive Fibonacci procedure. Following the symmetry of the problem, the mutual recursion should involve two functions $a(n)$ and $b(n)$.

```
(define (fib m)
  (letrec ((a (lambda (n)
                (if (= n 1)
                    1
                    (+ (a (- n 1))
                       (b (- n 1))))))
           (b (lambda (n)
                (if (= n 1)
                    1
                    (a (- n 1))))))
    (values (a (- m 1))
            (b (- m 1)))))
```

Fibonacci numbers by mutual recursion. This is a bit unusual. But it works, watch:

```
> (fib 10)
55
34
```

As with the usual non-memoized recursive procedure for Fibonacci numbers, we can't do `(fib 100)` because of the combinatorial explosion of function calls.

4 A combinatorics problem

Consider the following combinatorics problem. How many heads or tails sequences of length n are there such that they do not contain three consecutive heads? This well known problem can be solved by the following higher-order Fibonacci-like recursion

$$j(n) = j(n-3) + j(n-2) + j(n-1)$$

and initial conditions

$$j(1) = 2$$
$$j(2) = 4$$
$$j(3) = 7.$$

By analogy with the Fibonacci scheme in (5), we have the following recipe to generate current and previous solutions:

$$a, b, c \to a + b + c, a, b.$$

Let's make a function out of this that takes three arguments and returns three values:

```
(define (J a b c) (values (+ a b c) a b))
```

Since we already have the first three elements of the sequence, the nth element can be found by applying `J` exactly $n-3$ times.

```
(define (Jn n)
  ((apply many-compose
          (make-list (- n 3) H)) 7 4 2))
```

Let's test it. The next few elements of the sequence are 13, 24, 44, 81, 149 and 274. The 9th element should be 274.

```
> (Jn 9)
274
149
81
```

We also get the two previous elements.

Now then, it's time to turn `Jn` inside-out into a mutually recursive procedure. This time the recursion will involve three functions: $a(n)$, $b(n)$ and $c(n)$.

```
(define (j m)
  (letrec ((a (lambda (n)
                (if (= n 1)
                    7
                    (+ (a (- n 1))
                       (b (- n 1))
                       (c (- n 1))))))
           (b (lambda (n)
                (if (= n 1)
                    4
                    (a (- n 1)))))
           (c (lambda (n)
                (if (= n 1)
                    2
                    (b (- n 1))))))
    (values (a (- m 2))
            (b (- m 2))
            (c (- m 2)))))
```

Entering `(j 9)` at the REPL will give 274, 149 and 81.

5 The classic example

Any textbook that discusses mutual recursion will have this classic example. We want to create a predicate `my-even?` that determines if a number is even, by mutual recursion. n is even if $n-1$ is odd, and $n-1$ is odd if $n-3$ and so on. One function calls the other as the recursion proceeds down to the base case.

```
(define (my-even? m)
  (letrec ((e? (lambda (n)
                 (if (zero? n)
                     #t
                     (o? (- n 1)))))
           (o? (lambda (n)
                 (if (zero? n)
                     #f
                     (e? (- n 1))))))
    (e? m)))
```

Can this be turned inside out into repeated function applications? Yes it can. But first we should modify this to introduce some symmetry. Since there are two functions involved in the mutual recursion, there should be two return values. For example, the first return value can correspond to the evenness of m, while the second return value to oddness of m. It's redundant, but symmetrical.

```
(define (k m)
  (letrec ((a (lambda (n)
                (if (zero? n)
                    #t
                    (b (- n 1)))))
           (b (lambda (n)
```

```
                (if (zero? n)
                    #f
                    (a (- n 1))))))
    (values (a m)
            (b m))))
```

For example:

```
> (k 10)
#t
#f
```

The return values tell us that it is true that 10 is even, and it is false that 10 is odd. We can turn this inside out, into a function taking two arguments and returning two values, according to the scheme

$$a, b \to b - 1, a - 1.$$

We keep applying this until either a or b becomes zero. If a becomes zero then a was originally even and b was odd. If b becomes zero then a was originally odd and b was even. We apply this to a, b where b is just $a - 1$.

```
(define (K a b)
  (cond ((zero? a) (values #t #f))
        ((zero? b) (values #f #t))
        (else
          (values (- b 1) (- a 1)))))
```

We have to apply K exactly n times to $n, n - 1$.

```
(define (Kn n)
  ((apply many-compose
          (make-list n K)) n (- n 1)))
```

Let's test this.

```
> (Kn 10)
#t
#f
```

The classic case of mutual recursion has been turned inside-out into repeated function applications!