Anonymous Recursive Functions

or, How to Square the Square Root of a Function

Brian Beckman

30 Oct 2022

Introduction

We want to do some calculations on a remote server. The server lets us send expressions to evaluate, one at a time, but doesn't let us define variables or functions because that would use up memory.

For example, we want to compute the factorial of a number, say 6, but the server doesn't have a built-in for factorial. We'd like to send the standard recursive definition

In[2]:=
```
fact[n_] := If[n < 1, 1, n fact[n - 1]]
```

then, call it like this:

In[3]:=
```
fact[6]
```

Out[3]=
```
720
```

But that's two shots, and we only get one shot. We can't define fact, using up memory in the server's symbol table, and then use it on the next shot.

Are we out of luck? No. In fact, The following does the trick, as this article explains:

In[4]:=
```
(d ↦ (g ↦ g@g)[sf ↦ d[m ↦ sf[sf]@m]])[
  f ↦ n ↦ If[n < 1, 1, n f[n - 1]]]@6
```

Out[4]=
```
720
```

We'll show how to convert any recursive function into an anonymous version of itself. Furthermore, to sweeten the deal, we'll show how to convert expensive anonymous recursive functions into cheap anonymous recursive functions. We'll do it in Mathematica and talk about doing it in Scheme and Python.

Anonymous Functions

We already know how to define functions that don't have names: lambda expressions. Mathematica has a concise notation. Here is one that computes its argument, `x`, times `(x+1)`, its argument plus one:

```
In[5]:= x ↦ x * (x + 1)
Out[5]= Function[x, x (x + 1)]
```

Notation

That is a lambda expression of one argument, namely `x`. Read it as “the function of x that produces x times $(x+1)$, or $x(x+1)$.” The star for multiplication is optional in Mathematica, so you can write `x*(x+1)` as `x(x+1)`. In Scheme or Python, you must write the star.

We know how to apply such a lambda expression to an actual argument, say, to 6: wrap the lambda expression in parentheses and follow it with an `@` sign:

```
In[6]:= (x ↦ x (x + 1))@6
Out[6]= 42
```

Or, write the function application with square brackets like this:

```
In[7]:= (x ↦ x (x + 1))[6]
Out[7]= 42
```

The meaning is exactly the same. `x@y` means the same as `x[y]`, no matter what `x` and `y` mean. We choose one or the other at will to satisfy subjective aesthetics.

In the following, Script letters like $\mathcal{D}$, $\mathcal{E}$, $\mathcal{F}$, $\mathcal{L}$, and $\mathcal{Y}$, are ***notional names***: non-denotable names, names we can’t write in our programming language, but names of denotable things we need to think about and don’t want to keep writing out verbatim over and over again. For example, we’ll see the following symbol-blizzard over and over again:

`d↦(g↦g@g)[sf↦d[m↦sf[sf]@m]]`

That's a literal, denotable expression that we'll send to our server as part of other expressions. But it's too much to look at while thinking, so we'll just call it $\mathcal{Y}$ for the sake of discussion. In fact, explaining that expression is the whole point of this article. It’s a gadget that makes anonymous recursive functions and passes them into domain code for application.

Recursion as Squaring the Square Root

It turns out we can evaluate anonymous recursive functions in one shot. We’ll do it by ***taking the square root, $\sqrt{\mathcal{F}}$, of the function $\mathcal{F}$ that we want***, in a notional space where function application is multiplication. For example, we want `fact`, but the server doesn’t let us define the name `fact`. But the server does let us define temporary names that go away in one shot. Those names are the *formal* *parameters* of lambda expressions. So if we can define $\sqrt{\text{fact}}$ and then apply it to itself -- square it -- we get the same effect as `fact`.

More generally, for any function $\mathcal{F}$, pass $\sqrt{\mathcal{F}}$ as an actual argument to $\sqrt{\mathcal{F}}$. When $\sqrt{\mathcal{F}}$ is invoked, the actual argument $\sqrt{\mathcal{F}}$ is bound to the parameter, `sf`, in the body of $\sqrt{\mathcal{F}}$. Also In the body of $\sqrt{\mathcal{F}}$, we refer to $\mathcal{F}$ by the expression `sf[sf]` $= \sqrt{\mathcal{F}}\left[\sqrt{\mathcal{F}}\right] = \left(\sqrt{\mathcal{F}}\right)^2 = \mathcal{F}$; we square the square root, $\sqrt{\mathcal{F}}$, to get the recursive function, $\mathcal{F}$, we want. What a great trick! Turns out we can easily compute $\sqrt{\mathcal{F}}$ for any function $\mathcal{F}$. We do `fact` as an example, first, then generalize.

The Square Root of *Factorial*

To get the square root of factorial, just assume it exists and has a name, `sf`, as the parameter of a lambda expression, notionally called $\sqrt{\mathcal{F}}$. In the body of $\sqrt{\mathcal{F}}$, apply `sf[sf]`, the square of `sf`, wherever you want factorial. What is the value of `sf`? Just $\sqrt{\mathcal{F}}$ itself!. So $\sqrt{\mathcal{F}}\left[\sqrt{\mathcal{F}}\right]$ must be factorial, and we can apply it to numerical arguments:

In[8]:=
```
((sf ↦ n ↦ If[n < 1, 1, n sf[sf][n - 1]])@
   (sf ↦ n ↦ If[n < 1, 1, n sf[sf][n - 1]]))@6
```
Out[8]=
```
720
```

Before the final actual argument, `6`, and its application symbol, `@`, there is a lambda expression $\sqrt{\mathcal{F}}$ of one parameter `sf` applied to a cut-and-paste copy of its whole self via another application symbol `@`. That self-application squares the function $\sqrt{\mathcal{F}}$. Inside the recursive body -- inside the `If[...]` part -- there is a similar self-application, `sf[sf]`, applied to a numerical argument, `[n-1]`. So we see that the external squaring, `((sf↦...)@(sf↦...))` produces the same result as the internal squaring `sf[sf]`.

This is enough. Stop here if all you care about is a programming pattern for anonymous, remotable, recursive functions: just replace the body of the function, namely the `If[...]` part, in both places where it occurs, with the body of your desired recursive function, and call your function recursively via the self-application syntax `sf[sf]`. We'll show another example, `fib`, later.

However, there are worthwhile improvements. We can automate the programming pattern. We can write a general function $\mathcal{Y}$ that squares the square root of *any* function $\mathcal{F}$.

Four Improvements: Two Abstractions, One Model, and Packaging

To refresh the main idea: we have a square root $\sqrt{\mathcal{F}}$, that, when squared, produces the recursive ***domain function*** $\mathcal{F}$ of the ***domain parameters***, which does the real work we want.

Let's make a ***combinator*** (a function of a function) that can convert *any* function into a new function that receives its self application, the recursive function `f`, the square of the square root `sf`, as its first argument. This is a twist on the prior development. We want `sf[sf]` as the value of the first parameter `f`. We want to write `((...)[f↦n↦...])@6` in our example, with `f` as the self-application, namely the square, `sf[sf]`, and `n` as the domain parameter. We must solve for `(...)`.

Solve this in two steps: first, start with the prior development, in which `sf` is the parameter in a cut-and-paste self-application (squaring) of $\sqrt{\mathcal{F}}$. In the domain code, inside $\sqrt{\mathcal{F}}$, replace the square, the self-application `sf[sf]`, by a parameter `f` of a new anonymous function of `f`. Apply the new abstraction -- the new function of `f` -- to the actual argument `sf[sf]`. That's what ***abstraction*** means: replacing an expression $\mathcal{E}$ with a parameter `e` of a new function and then applying that new function to $\mathcal{E}$ as an actual argument.

In the second step, abstract the domain code into a parameter `d` of the final, general combinator $\mathcal{Y}$ (a notional name only) so that we write the domain code only once.

Step 1: Abstract the Internal Self-Application

Looking at the body of $\sqrt{\mathcal{F}}$, namely the `If[...]` part of

`sf ↦ n ↦ If[n < 1, 1, n sf[sf][n - 1]]`,

our first task is to abstract the internal self-application, `sf[sf]`, into a parameter `f`, the square of `sf`, then apply the new abstracted function of `f` to `sf[sf]` (or to `sf@sf`, same thing). This is exactly as we had before, only with `f` standing in for `sf[sf]`.

The fragment highlighted in yellow, below, is the new function of `f`, the new abstraction. However, this new abstraction fails to terminate even before applied to a numerical argument:

In[9]:=
```
(sf ↦ ((f ↦ n ↦ If[n < 1, 1, n f[n - 1]])[sf@sf])(* sf is still in scope! *))@
  (sf ↦ ((f ↦ n ↦ If[n < 1, 1, n f[n - 1]])[sf@sf])
   (* sf is still in scope! *));
```

$RecursionLimit : Recursion depth of 1024 exceeded during evaluation of
Function [f, Function [n, If [n < 1, 1, n f [n − 1]]]].

Why? Let's calculate. Let $s\mathcal{F}$, notionally, stand for this function of `sf` that binds `sf@sf` to the parameter `f`:

In[10]:=
```
sℱ = (sf ↦ ((f ↦ n ↦ If[n < 1, 1, n * f[n - 1]])[sf@sf]))
```

Out[10]=
```
Function[sf, Function[f, Function[n, If[n < 1, 1, n f[n - 1]]]][sf[sf]]]
```

Apply $s\mathcal{F}$ -- this function of `sf` -- to a copy of itself exactly as before:

In[11]:=
```
sℱ@sℱ;
```

$RecursionLimit : Recursion depth of 1024 exceeded during evaluation of
Function [f, Function [n, If [n < 1, 1, n f [n − 1]]]].

This can't work. `sf` becomes $s\mathcal{F}$, and the argument `sf@sf` is evaluated before binding to `f` in `(f↦ ...)`. That is called ***applicative-order evaluation*** or ***call-by-value*** -- evaluate arguments before applying the function. It's the norm in practical programming languages like Mathematica, Scheme, Lisp, and most things we're familiar with, and it's too early for this job.

- Aside: An alternative is called *normal-order evaluation* or *call-by-name*.

We can delay evaluation of by redefining $\mathcal{sF}$ to apply

`m ↦ sf[sf]@m`

to the numerical argument instead of `f`, as follows

```
In[12]:= (sf ↦ n ↦ If[n < 1, 1, n * (m ↦ sf[sf]@m)[n - 1]]);
```

We've temporarily lost the abstraction of `sf[sf]` into `f`, but gained a delayed evaluation of `sf[sf]`. We'll get `f` back in a minute.

`m↦sf[sf]@m` always has the same value as `sf[sf]` when applied to any argument. The two expressions just evaluate `sf@sf` at different times. In the first case, `sf@sf` is evaluated later when `m↦sf[sf]@m` is applied to the actual argument `n`, substituting `n`'s value for the parameter `m`.

- This is a general technique for delaying the application of any function: replace the application with a function of some parameter, the new function getting evaluated at the correct time.
- We note in passing that in lazy languages like Haskell, this step is automatic and implicit -- we don't write it -- because evaluation of all expressions is always delayed. That's similar to normal-order evaluation, maybe even equivalent.

Let's back off and write our very first original self-application with `m↦sf[sf][m]` manually in place of `f`.

```
In[13]:= (sf ↦ n ↦ If[n < 1, 1, n (m ↦ sf[sf]@m)[n - 1]])[
           sf ↦ n ↦ If[n < 1, 1, n (m ↦ sf[sf]@m)[n - 1]]]@6

Out[13]= 720
```

Now, as before, abstract `m↦sf[sf]@m` into a parameter `f` of a new lambda:

```
In[14]:= (sf ↦ (f ↦ n ↦ If[n < 1, 1, n f[n - 1]])[m ↦ sf[sf]@m])[
           sf ↦ (f ↦ n ↦ If[n < 1, 1, n f[n - 1]])[m ↦ sf[sf]@m]] @6

Out[14]= 720
```

the result does not spin because evaluation of `sf[sf]` is delayed until needed to apply to a numerical argument.

Step 2: Abstract the Domain Code

The abstraction on `f` now completely and minimally encloses the domain code $\mathcal{D}$ `=f↦n↦If[n<1,1,n f[n-1]]`. Abstract that:

`Write a new function of a parameter d` (1) with a body that replaces 𝒟, the old function of f, with an application of d. (2)

Apply that new function of d to the old function of f (3)

In[15]:=
```
(d ↦
    (sf ↦ d[m ↦ sf[sf]@m])[
     sf ↦ d[m ↦ sf[sf]@m]])[
  (* The following domain code 𝒟 is substituted for d. *)
  f ↦ n ↦ If[n < 1, 1, n f[n - 1]]]@6
```
Out[15]=
```
720
```

This has become a blizzard of symbols, but we can read it by the mnemonics that `sf` stands for $\sqrt{\mathcal{F}}$, the square root of the recursive function $\mathcal{F}$ or `f`, and that `d` stands for the domain code $\mathcal{D}$.

We get a big benefit: ***we only write the domain code once***, and that's a big deal, especially if it's complicated. *Don't Repeat Yoursel*f is a general principle in software engineering. It saves pitfalls now, during development, and later, during maintenance.

Step 3: Model the Self-Application

But we’re still writing the squaring of the square root twice. Let’s eliminate that final copy-paste code. Write a function `g↦g@g` that just self-applies any other function. Replace our self-application

```
(sf ↦ d[m ↦ sf[sf][m]])[
 sf ↦ d[m ↦ sf[sf][m]]]
```

with an application of `g↦g@g` to `f↦s[n↦f[f][n]]`:

In[16]:=
```
(d ↦
    (g ↦ g@g)[
     sf ↦ d[m ↦ sf[sf]@m]])[
  (* The following domain code 𝒟 is substituted for d. *)
  f ↦ n ↦ If[n < 1, 1, n f[n - 1]]]@6
```
Out[16]=
```
720
```

Demonstrate the Generality

We can now apply the outer combinator -- which always stays the same -- to a different, famous, recursive function -- which we only write once -- as domain code:

In[17]:=
```
(d ↦ (g ↦ g@g)[sf ↦ d[m ↦ sf[sf]@m]])[
  (* The following domain code is substituted for d. *)
  f ↦ n ↦ If[n < 2, 1, f[n - 2] + f[n - 1]]]@6
```
Out[17]=
```
13
```

Step 4: Packaging as a Combinator

Package the outer combinator as a notional function 𝒴 (our one-shot server doesn't permit this definition, but it's notionally convenient nonetheless. In denotable expressions !)

```
In[18]:= 𝒴 = d ↦ (g ↦ g@g) [sf ↦ d[m ↦ sf[sf]@m]];
```

and test on our two examples

```
In[19]:= 𝒴[f ↦ n ↦ If[n < 1, 1, n f[n - 1]]]@6
Out[19]= 720
```

```
In[20]:= 𝒴[f ↦ n ↦ If[n < 2, 1, f[n - 2] + f[n - 1]]]@6
Out[20]= 13
```

Step 5: More Arguments

Because we have a good grip on how 𝒴 works, we can write the two-argument version straightaway, without a hiccup! Functions of two arguments are invoked in Curried fashion, one argument at a time, as in `f[a][b]` rather than `f[a,b]`. Such Currying makes it trivial to extend 𝒴 to any number of arguments.

```
In[21]:= 𝒴2 = d ↦ (g ↦ g@g) [sf ↦ d[m ↦ n ↦ sf[sf][m][n]]];
```

Here is 𝒴2, operating on a made-up function of two arguments. I don't care to analyze this function, only to illustrate it in action. We'll have a useful function of two arguments in the next chapter on memoizing [sic].

```
In[22]:= 𝒴2[f ↦ m ↦ n ↦ If[m < 2, 1, n f[m - 1][If[n < 2, 1, m f[m][n - 1]]]]][3][4]
Out[22]= 648
```

My made-up function explodes rapidly. The following are the only results on positive-integer inputs that don't overflow Mathematica's recursion limits.

```
In[23]:= Table[𝒴2[f ↦ m ↦ n ↦ If[m < 2, 1, n f[m - 1][If[n < 2, 1, m f[m][n - 1]]]]][a][b],
           {a, 3}, {b, 4}] // TableForm
```

Out[23]//TableForm=

1	1	1	1
1	2	3	4
1	6	54	648

That' s a nice segue into ...

Memoizing [sic]

A downside of recursive functions is that they are often expensive. That “different famous recursive function” mentioned above is Fibonacci, a schoolbook case of exponential complexity. Calling our anonymous version of Fibonacci on `27` feels really slow (more than 2 seconds), and exponentially slower for bigger arguments: 10 seconds on `30` and a couple of minutes on `35`. Don’t try it on `40`.

In[24]:=
```
Timing[ 𝒴[f ↦ n ↦ If[n < 2, 1, f[n - 2] + f[n - 1]]]@27 ]
```
Out[24]=
```
{2.08192, 317811}
```

Our paranoid evaluation server has a time limit of a quarter of a second or so and would kill our jobs.

Are we out of luck for recursive functions? NO!

If our evaluator lets us define temporary symbols, and it always does so as function parameters, and if the built-ins give us hash tables, dictionaries, association lists, or some such, we can build tables of intermediate values and avoid recursive calls. This, also, is a general technique, the simplest instance of ***dynamic programming***, and it’s called ***memoization*** [sic, not memorization].

Mathematica automatically has a hash table, DownValues, for every symbol. So memoizing Fibonacci is nearly trivial, if we can name it:

In[25]:=
```
mfib[n_] := (mfib[n] = If[n < 2, 1, mfib[n - 1] + mfib[n - 2]]);
Timing[ mfib[27] ]
```
Out[26]=
```
{0.000085, 317811}
```

Orders of magnitude faster, linear instead of exponential, just by saving intermediate values onto the DownValues hash-table via `mfib[n] = ...`

The expressions `m[n-1]` and `m[n-2]` then become table lookups instead of recursive calls almost all the time. Wolfram foresaw this in 1980 when he designed Mathematica to use the same notation `[]` for table lookup as for function-call. Actually, it’s inherent in the term-rewriting method of Mathematica, but it’s brilliant, however it was conceived.

We can’t easily exploit Mathematica’s DownValues: function parameters don’t have them. Even so, it doesn’t port easily to non-symbolic programming languages like Python. Instead, we’ll use Mathematica’s Association, which is like Python’s dictionary.

The idea for linearizing the exponential process of Fibonacci by memoizing is as follows:

1. Factor our domain code `d` to do “lookup,” check-or-install a given key-value pair in an Association. We’ll use it twice, that’s why we want to name it rather than write it out verbatim twice.
2. Modify the recursive function, `f`, to take an Association and return a pair of an Association and a value, similarly to Haskell’s State Monad. That’s why we needed 𝒴2, the converter for two-parameter domain code.

3. Check the Association before recursively calling; this is the critical step for avoiding exponential run time.
4. Incrementally add new key-values pairs to the dictionary before returning it to recursive calls already on the stack.

We'll use Mathematica's `Module` to save a lot of keyboarding. `Module` allocates local variables. In an ordinary language like Scheme, we'd use `let` to allocate temporary variables. `Let` is 100% equivalent to Scheme lambda applied to arguments, where the symbolic parameter of the lambda is the temporary. In Mathematica, `Module` is equivalent to a lambda applied to arguments, but *only* when we do not mutate the parameters. That's because Mathematica is actually a term-rewriter instead of an applicative-order-evaluator. Mathematica feels free to throw away parameters, replacing them with values. Function parameters are not first-class symbols in Mathematica; pity. We want to mutate the values to avoid all kinds of grotesque argument threading. We trust we could do this entire thing with only lambdas and no Modules, but it would be longer and not instructive.

Here is memoized Fibonacci applied to 400. Without memoization, this computation would not complete in 15 billion years. Here, it takes 35 milliseconds.

In[27]:=
```
Timing[
 Module[{d = f ↦ a ↦ n ↦ (* has the shape of domain code! *)
        If[a[n] =!= Missing["KeyAbsent", n],
         {a, a[n]},
         f[a][n]]},
   𝒴2[f ↦ a ↦ n ↦
         Module[{v1, a1, v2, a2},
          {a1, v1} = d[f][a][n - 1];
          {a2, v2} = d[f][a1][n - 2];
          {Prepend[a1 ~ Join ~ a2, n → (v1 + v2)], v1 + v2}
         ]][
     <|1 → 1, 0 → 1|>][400]][[2]] ]
```

Out[27]=
```
{0.026715,
 284 812 298 108 489 611 757 988 937 681 460 995 615 380 088 782 304 890 986 477 195 645 969 ⋱
  271 404 032 323 901}
```

Finally, just to show that we could eliminate at least the outer Module, is an equivalent expression:

In[28]:=
```
Timing[
 (d ↦
    𝒴2[f ↦ a ↦ n ↦
          Module[{v1, a1, v2, a2},
           {a1, v1} = d[f][a][n - 1];
           {a2, v2} = d[f][a1][n - 2];
           {Prepend[a1 ~ Join ~ a2, n → (v1 + v2)], v1 + v2}
          ]][
      <|1 → 1, 0 → 1|>][400])[
  f ↦ a ↦ n ↦ (* has the shape of domain code! *)
     If[a[n] =!= Missing["KeyAbsent", n],
      {a, a[n]},
      f[a][n]]][[2]] ]
```

Out[28]=
```
{0.032036,
 284 812 298 108 489 611 757 988 937 681 460 995 615 380 088 782 304 890 986 477 195 645 969 ⋱
  271 404 032 323 901}
```