John Azariah's Blog

Ruminations of a partially-applied functional programmer...

Follow me on GitHub

Bouncing around with Recursion

Dec 7, 2020

Dec 7, 2020

Bouncing around with Recursion

This is my contribution to FsAdvent 2020

33 minute read

functional-programming recursion trampolines C# F#

Dedicated to Dr Shriram Krishnamurthi , who continues to inspire me after lo, these many years, to keep learning; and to Dr Mads Torgersen, who actively encouraged me to write this post!

Introduction

I recently had to venture into the world of preparing for technical interviews. I was astonished to see how it has devolved into a frankly grotesque side-show of books, videos and "competitive" coding websites, all purporting to land you a top coding job at some FAANG enterprise. While I could devote a whole blog post, or indeed, series, to dissecting the inanity of this method of evaluating someone's suitability to be an effective software engineer, I thought I'd write this post to dispel some of the egregiously faulty dogma bandied around about the use of recursion.

Most of these "elite" coders would start by warning you away from the "dangers" of recursion, and I want to use the foundations of functional programming, some advanced programming techniques, and some good, old-fashioned, reasoning to address the perceived deficiencies of recursion - even in more primitive languages.

Recursion is simply elegant

Consider a canonical example of recursion - finding the last element in a list. Let's use Python as an example language:

```
class Node(object):
    def __init__(self, d, t = None):
        self.data = d
        self.next = t
```

```
def last_element_of_list(head):
    def last_element_of_nonnull_list(xs):
        if (xs.next == None):
            return xs.data
        else:
            return last_element_of_nonnull_list(xs.next)

    if (head == None):
        return None
    else:
        return last_element_of_nonnull_list(head)
```

Such a program is very easy to comprehend, and it is trivial to establish correctness:

A list is either empty or not.
If the list is empty, return `None`
Otherwise, compute its last element as follows:

- If the tail of the list is empty, this is the last node, so return its `data`.
- Otherwise, the last element of this list is the last element of its tail, so compute and return that.

That's it - there aren't any other other normal boundary conditions, and the flow of the program cannot be made significantly easier to read and comprehend.

The catch comes when you try to run this code with a list of a thousand elements. Python, as many other stack-based languages, will push a frame on the stack for each of the thousand invocations of `last_element_of_nonnull_list`, leading to a Stack Overflow exception. You'll find a cursory discussion about this in the tutorial, concluding with solemn abjurations against using recursion in "real" programming.

But is this really a limitation of recursion itself? Or is this because we have implemented function invocation rather naïvely? Are there things we can do to fix this apparent runtime problem so we can retain the elegance of the code?

Tail recursion is necessary but not sufficient

You'll notice that the function above is tail-recursive. This means that the recursive call just returns the value it computed without any subsequent processing.

If you squint a little, you'll notice that the stack frame of the second call looks exactly like that of the first call, with the exception of the argument. So if we somehow cleverly re-used the same stack frame for all the calls by just adjusting that argument parameter, then the stack-overflow problem doesn't arise at all.

In fact, you don't need any fancy runtime support to do this. A lot of languages - including F# - have such "tail-call optimizations" built in, where a tail-recursive function can be emitted to use a single frame for its complete execution.

The following code in F#, therefore, will not suffer from Stack Overflow exceptions, even though it is identical - in layout and structure - with the Python version.

```
let last_element xs =
    let rec last_element_nn =
```

```
        function
        | x :: [] -> Some x
        | _ :: rest -> last_element_nn rest

    match xs with
    | [] -> None
    | _  -> last_element_nn xs
```

Conversely, several languages, including Python and C#, do not natively include tail-call-optimization in the language, so we are stuck with a growing stack frame in these languages. We'll look at ways to address this later in the post.

So what do we do now? Do we just crow about the superiority of F# and consider the problem solved?

Alas, we cannot, for a couple of reasons:

We want to find a reasonably general, language-agnostic, solution to the runtime performance problems of recursion
We will encounter problems where the recursion isn't immediately tail-recursive.

Let's address the second problem first with a couple of examples:

Problems where we can derive a tail-recursive expression

Let's consider the problem of computing a factorial.

Mathematically, we can express the factorial function recursively $\forall x \geq 0$ as:

$$0! = 1$$
$$x! = x * (x-1)!$$

In F#, this looks like:

```
let fact n =
    let rec fact_positive n' =
        if n' = 0 then
            1
        else
            n' * fact_positive (n' - 1)

    if (n >= 0) then
        Some <| fact_positive n
    else
        None
```

We can see that `fact_positive` is now no longer tail-recursive! We can no longer just cleverly reuse the same stack frame, because we have to complete the multiplication operation after jumping out of each frame. The fact that the language supports tail-call optimization seems to be irrelevant now, as the problem itself is not apparently tail-recursive.

Fortunately, this is actually not too difficult to fix.

Remember that we are allowed to vary the input parameters if we want to be tail recursive, so what if we passed in an `accumulator` carrying the result into the recursive function, and use that to obviate the post processing?

```
let fact n =
    let rec fact_positive (n', acc) =
        if n' = 0 then
            acc
        else
            fact_positive ((n' - 1), (n' * acc))

    if (n >= 0) then
        Some <| fact_positive (n, 1)
    else
        None
```

At this point, we are tail-recursive again, and we have found a general formula for converting this kind of problem into tail-recursive form.

If you're bold, you can use the `bigint` type for your accumulator, do some minor type surgery, and effortlessly compute enormous factorials without any fear of stack overflow.

The process to make these kinds of functions tail-recursive is pretty mechanical. See if you can follow the instructions to convert the following function into tail-recursive form:

Tuple the input argument to the recursive function with an accumulator
In the terminating case, return the accumulator
In the recurring case, compute the new accumulator with the post-processing operator and pass it in with the reduced argument
To start the recurrence, pass in the zero value for the accumulator

```
let sum_list xs =
    let rec sum_nonnull_list xs' =
        match xs' with
        | [] -> 0
        | x :: rest -> x + sum_nonnull_list rest

    match xs with
    | [] -> None
    | _  -> Some <| sum_nonnull_list xs
```

When you are done, you should get something like:

```
let sum_list xs =
    let rec sum_nonnull_list (xs', acc) =
        match xs' with
        | [] -> acc
```

```
        | x :: rest -> sum_nonnull_list (rest, x + acc)

    match xs with
    | [] -> None
    | _  -> Some <| sum_nonnull_list (xs, 0)
```

By now we're feeling pretty good about ourselves and our choice of F# as a programming language. But are we ready to call victory?

Unfortunately not, because we still haven't found a solution we can translate to other languages, and we are about to come unstuck when we encounter any function that doesn't isn't singly recursive!

Problems where we cannot derive a tail-recursive expression

There are many useful functions we can encounter that are not singly recursive. That is to say, the recursive portion of the function may invoke itself more than once. In this case, it's obvious that we simply cannot make the function tail recursive, as there are at least two invocations, both of which cannot be the only call, as is required for tail recursion.

Consider the famous `Fibonacci` function. Mathematically, we define it $\forall n \geq 0$ as:

$$\mathbb{F}(0) = 1$$

$$\mathbb{F}(1) = 1$$

$$\mathbb{F}(x) = \mathbb{F}(x-1) + \mathbb{F}(x-2)$$

So what do we do with a function like this?

Most people will encounter the exponential time complexity of the `Fibonacci` function, and reasonably conclude that we should never use recursion for such functions. Indeed, as the iterative solution is relatively simple to derive, it seems reasonable to give up the elegance and reasonability of the recursive expression and hand-derive the equivalent iterative solution.

But how about binary tree traversals?

These are doubly-recursive functions which recur down the left- and right- subtrees at each node. While they are beautifully elegant to write out in recursive form, it's difficult to defend the fact that such a solution will blow the stack in practice. However, this time the iterative solutions are not straight-forward - in fact, these are popular interview questions simply because they are so complex and have so many places for bugs to hide.

As mathematicians, computer scientists and engineers, we should realize that perhaps eschewing recursion to escape the runtime problems is perhaps barking up the wrong tree. Perhaps a much more satisfying - and rigorous - approach is to use the tools afforded to us by computer science to address these runtime issues, whilst preserving the elegance and easy reason-ability of the recursive form.

The rest of this post is a two-level deep dive into this approach:

Continuations…

From the early days of computer science, we have known about a computation construct called continuations. I will attempt to provide a rudimentary description and application on these here, noting that there is a wealth of serious literature available for the interested.

Consider the following expression:

```
a * (b + c)
```

The traditional way to compute this is to compute `(b + c)` first, call that `v`, and then compute the product `a * v`.

Let's write that out:

```
let add x y = x + y
let mul x y = x * y

// a * b + c
add b c |> mul a
```

In this case, it's clear that we have some post-processing being done to the result of `add b c`, so it's similar to our `Fibonacci` program in that regard. Now, let's pass functions around instead of scalar values:

```
let add_c x y f = (x + y) |> f
let mul_c x y f = (x * y) |> f

// a * b + c
add_c b c (fun v -> mul_c a v (fun x -> x))
```

You can think of the `f` functions as things that continue with the result of the computation in `add_c` and `mul_c`. This may look pointless and verbose until you realize, that all the function calls in the expression are tail calls. As in, the only thing that actually returns anything is the `id` function passed in to `mul_c`.

So we took some arbitrary chain of function invocations, and quite mechanically, converted them to accept continuation functions, and tail-call to those continuation functions.

A theoretical note here is that this expression here is a `bounded` continuation - in that it actually returns something. There are systems that use `unbounded` continuations, where the chain of control is continuously passed around - and such constructs are supremely powerful. Complete computation systems - including threading, co-routines, and virtually any control flow structure can be synthesized using unbounded continuations.

Specifically, here is the recipe to convert a function to CPS:

Create a version of the function taking a "continuation" argument - a function to jump to at the end of the computation.
When the function returns an immediate value (not something computed by a function call), send that value to the continuation.
When a function call appears in tail-call position, make the call and pass the continuation (unchanged) to it.
When a function call appears in non-tail-call position, make a new continuation where you execute the function call and name the result, and carry on with the rest of the expression.

Let's work out the steps with an example:

```
// WARNING - this is horribly slow, and will blow the stack
let rec fib_naive n =
    match n with
    | 0 -> 1
    | 1 -> 1
    | _ -> fib_naive (n - 1) + fib_naive (n - 2)

// WARNING - this is still slow, but it is constant stack space
let rec fib_cps n c =                          // 1. Add `c` as argument
  match n with
  | 0 -> c 1                                   // 2. Send `1` to `c`
  | 1 -> c 1                                   // 2. Send `1` to `c`
  | _ ->
    fib_cps
      (n - 1)
      (fun f1 ->                               // 4. Apply to first invocation of `fib_cps`
        fib_cps
          (n - 2)
          (fun f2 ->                           // 4. Apply to second invocation of `fib_cps`
            c (f1 + f2)))
```

It's easy to see here that there are no functions that aren't in tail call position now, and we got here by systematically applying some rules. Indeed, there are some compilers that automatically do this for languages where unbounded continuations are fundamental building blocks.

The sharp-eyed amongst you will notice that we have not really sped up `Fibonacci` at all - and you are right. That would require `memoization`, which is yet another blog post in itself.

Trampolines

The skeptical amongst you will agree that we have indeed laboured heavily to get to this point, but with seemingly not much improvement to show for it! And you would be right. In order to take full advantage of these tail-call converted functions, we will need another structure called a trampoline.

Trampolines allow us to take full advantage of tail-called functions by providing a mechanism to keep a constant height stack whilst evaluating these functions, even for languages that do not have tail-call optimization! Now we're cooking with gas!

To use a trampoline, we do some surgery to the function just like we did for CPS conversion, so that the function doesn't eagerly evaluate the entire recurrence at one go. Rather, it computes each step of the recurrence and returns a suspended continuation to the caller. The trampoline calls these continuations one at a time, using constant stack space, until the computation is complete.

The key literature resource I used here was Stackless Scala With Free Monads by Rúnar Bjarnason.

Additionally, the following blog posts were instrumental in the evolution of my solution:

Tail recursion in C#
Jumping the trampoline in C# – Stack-Friendly Recursion
Tail Recursion And Trampolining In C#

So let's see what this Trampoline finally looks like:

```
[<AutoOpen>]
module Trampoline =
    type Trampoline<'a> =
        | Return of 'a
        | Suspend of (unit -> 'a Trampoline)
        | Flatten of {| m : 'a Trampoline; f : 'a -> 'a Trampoline |}
```

The `Trampoline` itself is very simple - there are really only two cases to work with:

We have the terminal case - in this case, we return a value of type `Return`
A tail call position function call - in this case we suspend the function and return a value of type `Suspend`
The `Flatten` type is used to compose computations, as well as when we try to execute the Trampoline value.

The `execute` function looks like:

```
    let execute (head : 'a Trampoline) : 'a =
        let rec execute' = function
            | Return v -> v
            | Suspend f ->
                f ()
                |> execute'
            | Flatten b ->
                match b.m with
                | Return v ->
                    b.f v
                    |> execute'
                | Suspend f ->
                    Flatten {| m = f (); f = b.f |}
                    |> execute'
                | Flatten f ->
                    let fm = f.m
                    let ff a = Flatten {| m = f.f a ; f= b.f |}
                    Flatten {| m = fm;  f = ff |}
                    |> execute'
        execute' head
```

As you'd expect, the `Return` and `Suspend` are handled in a straightforward way. The `Flatten` value is used to chain computed values to their continuations monadically - a full explanation of this may take a post in itself, so either read the Bjarneson paper or accept this as a piece of opaque machinery, as this whole class could just genuinely come from a library.

Putting it all together, we can write the following code:

```
[<AutoOpen>]
```

```
module Trampoline =
    type Trampoline<'a> =
        | Return of 'a
        | Suspend of (unit -> 'a Trampoline)
        | Flatten of {| m : 'a Trampoline; f : 'a -> 'a Trampoline |}
        with
            static member (<|>) ((this : 'a Trampoline), (f : 'a -> 'a)) =
                Flatten {| m = this; f = (f >> Return) |}

            static member (>>=) ((this : 'a Trampoline), (f : 'a -> 'a Trampoline)) =
                Flatten {| m = this; f = f |}

    let execute (head : 'a Trampoline) : 'a =
        let rec execute' = function
            | Return v -> v
            | Suspend f ->
                f ()
                |> execute'
            | Flatten b ->
                match b.m with
                | Return v ->
                    b.f v
                    |> execute'
                | Suspend f ->
                    Flatten {| m = f (); f = b.f |}
                    |> execute'
                | Flatten f ->
                    let fm = f.m
                    let ff a = Flatten {| m = f.f a ; f= b.f |}
                    Flatten {| m = fm;  f = ff |}
                    |> execute'
        execute' head

module Fact =
    open System.Numerics
    let fact n =
        let rec fact' n accum =
            if n = 0 then
                Return accum
            else
                Suspend (fun () -> fact' (n-1) (accum * (BigInteger n)))

        if (n < 0) then invalidArg "n" "should be > 0"

        fact' n BigInteger.One
        |> execute
```

It's immediately apparent that our tail-recursive `Factorial` version can be mechanically converted to a continuation-passing `Trampoline`. While it is true that with F#, we didn't need the trampoline for this, we can see that this same method can work for C# as well!

```
    public static class FunctionalExtensions
```

```
{
    public static Func<A, C> AndThen<A, B, C>(this Func<A, B> f, Func<B, C> g) =>
        a => g(f(a));
    public static Action<A> AndThen<A, B>(this Func<A, B> f, Action<B> g) =>
        a => g(f(a));

    public static A PipeTo<A>(this A value, Func<A, A> f) => f(value);

    public static void PipeTo<A>(this A value, Action<A> a) => a(value);
}

public abstract class Trampoline<A>
{
    public static Trampoline<A> Return(A a) =>
        new Members.Return(a);
    public static Trampoline<A> Suspend(Func<Trampoline<A>> f) =>
        new Members.Suspend(f);
    public static Trampoline<A> Flatten(Trampoline<A> m, Func<A, Trampoline<A>> f) =>
        new Members.Bind(m, f);

    public Trampoline<A> Bind(Func<A, Trampoline<A>> f) =>
        new Members.Bind(this, f);
    public Trampoline<A> Map(Func<A, A> f) =>
        new Members.Bind(this, f.AndThen(Return));

    public static A Execute(Trampoline<A> head)
    {
        var tr = head;

        while (true)
        {
            switch (tr)
            {
                case Members.Return a: return a.Value;
                case Members.Suspend r: tr = r.Resume(); break;
                case Members.Bind f:
                    switch (f.M)
                    {
                        case Members.Return fr: tr = f.F(fr.Value); break;
                        case Members.Suspend fs: tr = Flatten(fs.Resume(), f.F); break;
                        case Members.Bind ff: tr = Flatten(ff.M, a => Flatten(ff.F(a), f.F
                    }
                    break;
            }
        }
    }

    private static class Members
    {
        public sealed class Return : Trampoline<A>
        {
            public Return(A value) => Value = value;
            public A Value { get; }
        }
```

```
            public sealed class Suspend : Trampoline<A>
            {
                public Suspend(Func<Trampoline<A>> resume) => Resume = resume;
                public Func<Trampoline<A>> Resume { get; }
            }

            public sealed class Bind : Trampoline<A>
            {
                public Bind(Trampoline<A> m, Func<A, Trampoline<A>> f)
                {
                    M = m;
                    F = f;
                }

                public Trampoline<A> M { get; }
                public Func<A, Trampoline<A>> F { get; }
            }
        }
    }

    BigInteger Factorial(int n)
    {
        Trampoline<BigInteger> F(int n, BigInteger accum) =>
            (n == 0)
                ? Trampoline<BigInteger>.Return(accum)
                : Trampoline<BigInteger>
                    .Suspend(() => F(n - 1, accum * n));

        return Trampoline<BigInteger>.Execute(F(n, BigInteger.One));
    }
```

Tree Climbing With Trampolines

We're finally here. Let's consider the Tree Traversal algorithms.

As mentioned earlier, these are beautifully succinct algorithms expressed recursively, but bug-ridden, inscrutable monsters if written iteratively. A good test of our approach will be how closely we are able to match the elegance of the recursive solution.

Additionally, since I promised that this approach will be truly general, we will show the code in C# as well, which does not have native tail-call optimization.

F#

```
    // The basic BinaryTree structure
    type BinaryTree<'node> =
        {
            Value : 'node
            Left  : BinaryTree<'node> option
            Right : BinaryTree<'node> option
        }
```

```
    with
        static member Apply(n, ?l, ?r) = { Value = n; Left = l; Right = r }
        member this.AddLeft(n)  = { this with Left  = (Some << BinaryTree<_>.Apply) n }
        member this.AddRight(n) = { this with Right = (Some << BinaryTree<_>.Apply) n }

    // In-Order Traversal
    let foldInOrder root seed consume =

        // The (stack-unsafe but elegant) recursive version
        let rec foldRecursive node accum =
            match node with
            | None -> accum
            | Some n ->
                foldRecursive n.Left accum
                |> (fun left -> consume left n.Value)
                |> (fun curr -> foldRecursive n.Right curr)

        // The (stack-safe and equally elegant) recursive version
        let rec foldTrampoline node accum =
            match node with
            | None -> Return accum
            | Some n ->
                Suspend (fun () -> foldTrampoline n.Left accum)
                >>= (fun left -> Return (consume left n.Value))
                >>= (fun curr -> Suspend (fun () -> foldTrampoline n.Right curr))

        foldTrampoline root seed
        |> execute
```

This is profoundly satisfying, because we have crafted a stack-safe version of the elegant version of the recursive solution without losing any of the elegance.

C#

Let's see if we are able to do the same thing with C#!

```
    public class Tree<TNode>
    {
        public TNode Value { get; }
        public Tree<TNode> Left { get; private set; }
        public Tree<TNode> Right { get; private set; }

        public Tree(
            TNode value,
            Tree<TNode> left = null,
            Tree<TNode> right = null)
        {
            Value = value;
            Left = left;
            Right = right;
        }
```

```
        public Tree<TNode> AddLeft(TNode value) =>
            Left = new Tree<TNode>(value);

        public Tree<TNode> AddRight(TNode value) =>
            Right = new Tree<TNode>(value);
    }

    public static class InOrderVisitors
    {
        public static R VisitInOrder_Recursive<T, R>(this Tree<T> root, R seed, Func<R, T, R>
            root is null
                ? seed
                : VisitInOrder_Recursive(root.Left, seed, consume)
                    .PipeTo(left => consume(left, root.Value))
                    .PipeTo(curr => VisitInOrder_Recursive(root.Right, curr, consume));

        public static R VisitInOrder_Trampoline<T, R>(this Tree<T> root, R seed, Func<R, T, R>
        {
            Trampoline<R> V(Tree<T> node, R accum) =>
                (node is null)
                    ? Trampoline<R>.Return(accum)
                    : Trampoline<R>.Suspend(() => V(node.Left, accum))
                        .Bind(left => Trampoline<R>.Return(consume(left, node.Value)))
                        .Bind(curr => Trampoline<R>.Suspend(() => V(node.Right, curr)));

            return Trampoline<R>.Execute(V(root, seed));
        }
    }
```

Again, we were able to build a virtually identical recursive solution with the Trampoline, and we are able to show that even with languages that do not support CPS natively, or even TCO, we can continue to use recursion safely and efficiently..

Conclusion

We started off talking about how recursion was unfairly reviled for its runtime characteristics. However, with proper methodology, we explored Tail Recursion, Continuations and Trampolines and finally have arrived at the point where we can verify that all our sleight of hand has paid off.

This has been a deep dive into functional programming, exploring functions, composition, recursion, tail-calls, continuations and trampolines. We approached the topics with rather more rigour than you might need when trying to crack a coding interview, and hopefully concluded that we can do better than wave our hands in the air with unsubstantiated nonsense about how recursion is not a safe and elegant way of coding.

Along the way, I hope you got an appreciation for the wealth of concepts and literature available to be used!

Happy holidays, and happy typing!

What do you think?

5 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments **John Azariah's Blog** **Disqus' Privacy Policy** **Login**

Recommend **Tweet** **Share** **Sort by Best**

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Be the first to comment.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**

This page was generated by GitHub Pages.