# Recursion, Tail Calls and Trampolines

November 17, 2015. By Dave Atchley. Follow @tuxz0r

Recursion is a control flow construct. It allows us to have a function call itself and repeat an operation until it arrives at a result. Many functional languages are built around recursion and lazy evaluation. While Javascript has many functional aspects as a language, it is currently not built to handle recursion in a safe and effective manner.

Does this mean we shouldn't use recursion in our programs? No, not necessarily. Some functions become much more readable, maintainable and succinct when written in recursive form. Some mathematical functions and algorithms fall into this category.

> *"Programs must be written for people to read, and only incidentally for machines to execute."* *Abelson & Sussman, Structure and Interpretation of Computer Programs*

But this isn't always the case; and it may be better to use iteration or some other mechanism to repeat an operation such as Promises or the browser's own event loop.

With all that in mind, however, let's take a look at recursion in Javascript in various forms to see how and why it works.

## Functions and the Stack

Before we can get into the topic at hand, we must have a basic understanding of Javascript's execution model - specifically, the execution or call stack.

Javascript is single threaded, which means that only one task can be run at any given time. When the Javascript interpreter on the page starts executing code it is running in an environment that is referred to as the Global Execution Context. Various language features will trigger the creation of a new execution context (*functions, eval, let blocks, closures, etc.*).

So what is available to our code in each of these execution contexts? The execution context for the currently running code is basically the accessible scope (*variables and functions*) for that code. I've covered scoping in Javascript previously; but here is an example:

```
var a = 4;

function foo(x) {
    var b = a * 4;

    function bar(y) {
        var c = y * b;
        return c;
    }

    return bar(b);
}

console.log(foo(a));
// 256
```

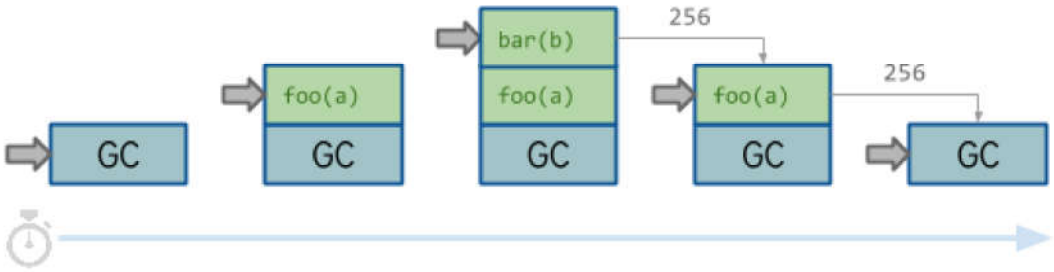
1 — global: a, foo

2 — foo: x, b, bar

3 — bar: y, c

Here, when the two functions `foo` and `bar` are encountered, the interpreter creates a new execution context. These contexts are created on a stack data structure, meaning that if a new context is created during the currently running code, this new execution context is created and stacked on top of the current context.

Here is a simplified execution stack for the above code. Keep in mind that because of the scope chain, execution contexts will be able to access variables and functions delared in any parent (*previous*) scope.




**Javascript Execution Stack - Example**

**GC** = Global Execution Context

= Current Execution Context

The important thing to note is that nested function calls add to the stack. When the code in a given context is finished executing, that stack entry is destroyed and control is returned to the executable code from the previous stack execution context.

Armed with this simple understanding of Javascript's execution contexts and stacks, let's move ahead with an example function to show how repeating a task works from simple iteration through various uses of recursion.

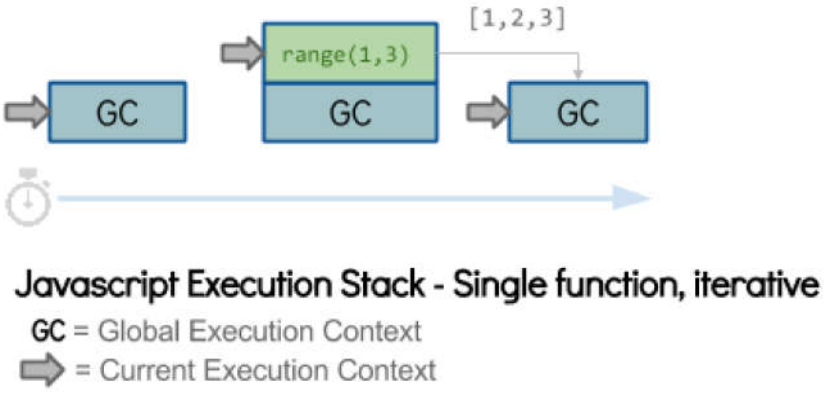
## Iterating

We'll try to be original in this post and not use the tried and true fibonacci sequence as our recursion example; instead, opting to create a simple function that will generate an array of integers in a range (*inclusively*) if given a starting number and an ending number.

Our function, `range`, simply iterates using a `while` loop to produce the given range and returns the array.

```
// Generate an array of integers in the range s -> e.
// Iterative implementation
function range(s, e) {
    var res = [];

    while (s != e) {
        res.push(s);
        s < e ? s++ : s--;
    }
    res.push(s);
    return res;
}

range(1,4);  // [1,2,3,4]
range(-5,1); // [-5,-4,-3,-2,-1,0,1]
```

The function is simple and performs our operation effectively. How does this look on the execution stack?




The `range` function is called within the Global Execution Context and triggers the creation of a new execution context on the stack. The range function performs its operations and returns the resulting array. This destroys the function's execution context and returns the result and control back to the code in the global context.

This isn't a function you would want to convert into a recursive style, as the performance and safety would seriously degrade. However, for the purposes of this blog post and our discussion, it will suffice, as its implementation and use are clear and simple.

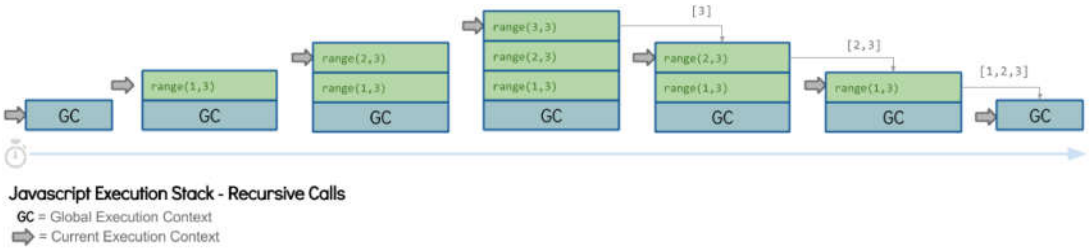
## Recursion

Moving our `range` function from iterative to recursive, we can keep our call API the same, taking a starting and ending integer for the range. Since the range we're building is inclusive, we push the starting number onto our `res` array and will call ourself recursively until we hit the ending integer. The new starting number passed to our self recursive call is incremented or decremented depending on the direction in which we're building the range.

As we get results back from our self recursive calls, we flatten them into our own results before returning using `Array#concat`.

```
// Generate an array of numbers in a given range.
// Recursive implementation
function range(s, e) {
  var res = [];

  res.push(s);
  return s == e ? res : res.concat(range(s<e ? ++s : --s, e));
}

range(1,4); // [1,2,3,4]
```

How does this look on the execution stack? Each recursive call adds an execution context to the stack until we reach the last, ending integer. Then, the stack begins to unwind as each call returns its results.




We simply push a number onto our results array and call ourself again for the next number. The terminal condition that will stop the repeating of our operation is when our start value is equal to the end value of our range. At that point, we're done, and each function on the stack can unwind, concatenating and flattening the results array as they go.

## Tail Calls & Recursion

In our previous example, the call we made to ourselves was part of an expression, specifically:

```
res.concat(range(s<e ? ++s : --s, e))
```

From an interpreter standpoint this means that we need to retain the current execution context, because we're dependent on the results from our self-recursive call in order to modify our local `res` array. If we were to instead have

our function return just the results of making the self call, rather than require the result of that call to be evaluated in an expression, we open up the possibility of simplifying our execution environment because we no longer have to retain the current execution context of the calling function.

Essentially, if we return the results of a function call (*recursive or not*) at the end of our function and none of our function's local environment is needed as part of that evaluation, then that function call is said to be in **tail position**.

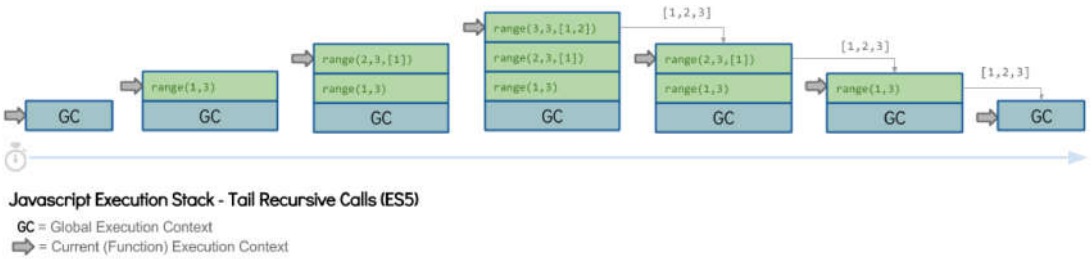
The reason this is important is because if we write calls in tail position, this can allow for the interpeter to optimize the call by reusing the existing stack frame. Let's re-write our function to put the recursive call in tail position instead.

```
// Generate an array of numbers in a given range.
// Tail Recursive implementation
function range(s, e, res) {
  res = res || [];
  res.push(s);
  return s == e ? res : range(s<e ? ++s : --s, e, res);
}

range(1,4); // [1,2,3,4]
```

In order to remove any dependencies on our local environment, in this case the `res` array, we pass our current results into the next call to ourself. This way, we're passing all the state our function needs to continue processing as it recurses to find a result. Notice the difference: in the tail recursive version we return either a result value or the result of calling ourself without requiring any local state.

```
// Non tail-call version
return s == e ? res : res.concat(range(s<e ? ++s : --s, e));


// Tail call version
return s == e ? res : range(s<e ? ++s : --s, e, res);
```

Let's see how this looks on the execution stack when run.




So, what's going on here? As each new execution context is added to the stack, the full results array is passed along as well. Our stack is still growing with each nested call but the interpreter isn't reusing the existing stack frame for the execution of the new function. What gives? The reason is that, currently, Javascript isn't making any optimizations for calls in tail position.

The stack space for a running Javascript program is finite, and so the more nested calls (*self-recursive or not*) that are made, the larger the stack grows until the process runs out of memory - specifically stack space. We can see this at play if we try generating very large ranges with our recursive implementation.

```
range(1,100);
// => [1,2,3,4,...,98,99,100]
ranger(1,32768);
// Uncaught RangeError: Maximum call stack size exceeded
```

In the next version of Javascript, ES6, tail call optimization is slated to be supported; but for now, calls in tail position act just like any other nested calls from a stack execution context perspective.

Are there ways around this? Sure, let's look at a couple of them next.

## Tail Recursion with Trampoline

One way to get around the lack of tail call optimization in languages like Javascript is to use a pattern called *trampolining*. A trampoline is a loop that iteratively invokes *thunk*-returning functions. Trampolined functions can be used to implement tail recursive function calls in stack based languages.

So, what is a *thunk*? A thunk is essentially a function that wraps a call to another function, along with any parameters it needs, for later execution. A way to emulate lazy evaluation.

In our previous `range` function, our call to ourself was in tail position; and we returned the result of calling ourself directly. If, instead, we returned a function that wraps that call (*our thunk*), we can use a trampoline to continually execute ourself.

Let's take a look at the trampoline function.

```
function trampoline(fn){
    return function(){
        var res = fn.apply(this, arguments);
        while(res instanceof Function){
            res = res();
        }
        return res;
    }
}
```

Trampoline is essentially a higher-order function that takes a function as an argument and returns a new function - a *combinator*! This new function, when called, will repeatedly invoke the original function passed to trampoline as long as that function returns another function to execute (*the thunk*). Once it gets back a result that is not a function, it stops executing and just returns the result of the last call.

So, let's modify our previous range function to return a thunk instead of the result of calling ourself directly so that trampoline can repeatedly invoke it for us until we get a result.

```
// Generate an array of numbers in a given range.
// Tail Recursive implementation
function range(s, e, res) {
  res = res || [];
  res.push(s);
  // return a result or a thunk if we need to do more
  return s == e ? res : function() { return range(s<e ? ++s : --s, e, res); };
}

trampoline(range)(1,4); // [1,2,3,4]
```

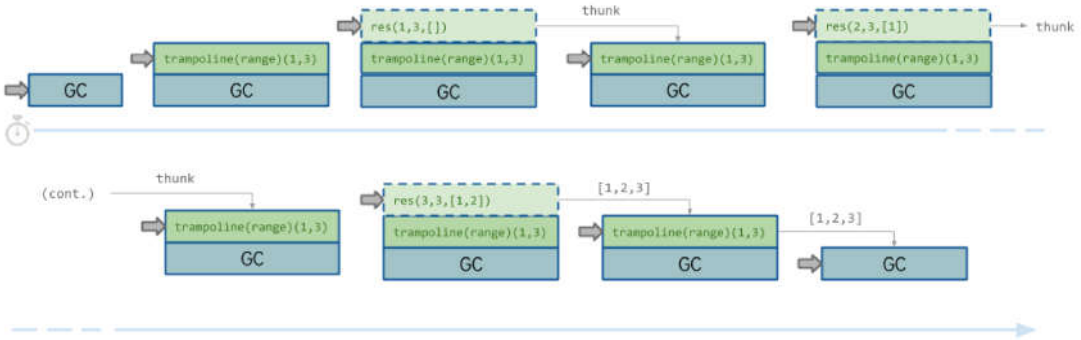
This style requires a slight modification when writing a recursive function; but, if we need to be recursive and need to work around a lack of tail-call optimization, such as in Javascript ES5, this allows us to proceed.

However, we are trading off performance for stack safety, as trampolined functions run much slower than straight iterative code or real, optimized tail-calls.

But, we can now make our call to our range function without worrying about a stack overflow as before.

```
range(1,100);
// => [1,2,3,4,...,98,99,100]
ranger(1,32768);
// => [1,2,3,4,...,32766,32767,32768]  WORKS!!
```

Here's how the call stack looks when executing our function using trampoline, the stack size is a constant, with a new stack only being created on each iterative execution of the recursive function.




Javascript Execution Stack - Tail Recursive Calls w/ Trampoline
GC = Global Execution Context
⇨ = Current (Function) Execution Context

This is much different than our version without trampolining, in which the stack size grows linearly based on the number of recursive calls we have to make.


## Considerations when using recursion?

Using recursion incorrectly or when not needed has the potential to cause problems:

- If you don't properly specify your terminal condition you could hit an infinite loop issue and lock up the browser.
- If your data or operation is sufficiently lengthy you run into the issue of stack overflow.
- Also, because Javascript uses *eager evaluation* and isn't really built to properly handle recursive style programming or tail-call optimization, writing in recursive form will typically slow down performance.

Regardless, recursion has very practical reasons for usage as well. Writing functions that need to repeat for a set period of time or until some condition is met is an ideal reason.

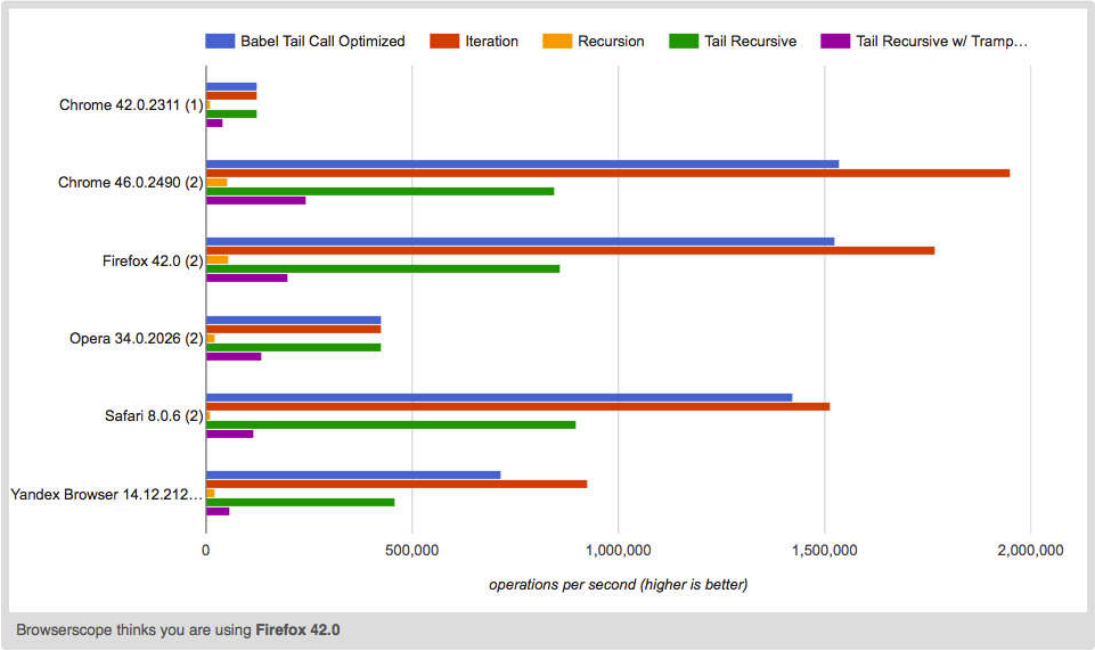
```
// Call ourselves repeatedly until some other async process
// loads our app's local storage we need.
(function waitForLocalStorage(key) {
   if (localStorage.getItem(key)) {
      // do something with loaded data
   }
   else {
      setTimeout(waitForLocalStorage.bind(null, key), 100);
  }
})('mydata');
```

Many searching and sorting algorithms rely on recursion to perform their operations; and, if re-written using iterative loops would be much more complex, unweildy or harder to read/maintain. Typically, imperative implementations using explicit loops will create and need to manage more state. Think about binary searching algorithms like tree-traversal.

Also, since we know that ES6 will support tail-call optimization via direct recursion and mutual recursion at some point, we can use methods like *trampoline* or even ES6 transpilers like Babel to allow us to write recursive functions that are performant and easily take advantage of new language/browser features as they become available.

### Performance comparison

As a simple test, I put together a jsperf test looking at the performance of each of the different versions of our `range` function above. Here are the results:




Interstingly, we can see that the straight, iterative approach is the hands-down winner. If you need performance at all costs, this is clearly your best bet with the current implementaiton of Javascript.

The trampolined version isn't as slow as recursion not in tail position; but, unlike the recursive and tail recursive versions, it will not throw a RangeError if you have more nested calls than the browser has stack space. It's a clear winner for writing safe recursive style code if performance is not the primary focus.

Babel transpiles our code into a new function that processes using a while loop; and because of this, gets closer to the native iterative performance. We'll briefly cover Babel's transpilation at the end of this post.

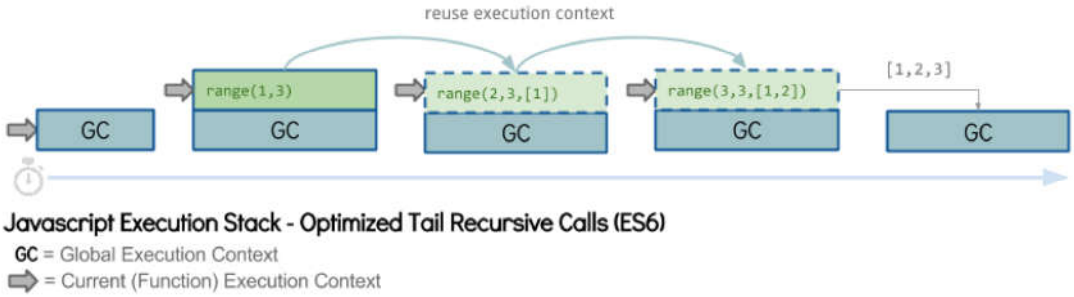
As a note, using IE Edge/Win 10 on Browserstack reported the browser as *Chrome 42.0.2311 (1)*, the first entry in the graph above.

## Tail Call Optimization in ES6

We'll end our post with a glimpse at how tail-call optimization would work in ES6 and how that changes the stack during execution. Here's our tail-recursive version of our range function, again:

```
function range(s, e, res) {
  res = res || [];
  res.push(s);
  return s == e ? res : range(s<e ? ++s : --s, e, res);
}
```

With proper tail-call optimization by the Javascript interpreter, our resulting call stack might look like the following:




Here, Javascript recognizes the tail-call and can then reuse the existing stack frame to make the recursive call, removing any previously local variables and state from the old call.

If you're using Babel to transpile, Babel handles tail-call optimization on direct, self-recursion. So our above range function would be re-written by Babel into the following:

```
// Generated using Babel 5.x
"use strict";

function range(_x, _x2, _x3) {
  var _again = true;

  _function: while (_again) {
    var s = _x,
        e = _x2,
        res = _x3;
    _again = false;

    res = res || [];
    res.push(s);
    if (s == e) {
      return res;
    } else {
      _x = s < e ? ++s : --s;
      _x2 = e;
      _x3 = res;
      _again = true;
      continue _function;
    }
  }
}
```

In this case, Babel inlines the function execution inside a while loop, using `continue` to implement a jump style control to repeat the loop until the given condition is true. This is much more efficient than using `trampoline` to wrap recrusive functions for a number of reasons:

- Our function, in this case, is completely re-written so there are no nested function calls, keeping our stack context at a constant.
- The task our function performs is inlined in a loop, which is much more performant

Using trampoline, we still have to invoke a function call, using the returned *thunks*, on each iteration, so we keep the overhead of creating and destroying

stack frames.

Trampolining is more generally applicable to a wide range of tail-recursive functions (*so long as they return thunks*) which is definitely more flexible and maintainable; but if performance is an issue, re-writing your code to use loops or taking advantage of transpilers like Babel is your best bet.

## Find out more

You can find a lot about functional programming and recursion online from blogs and other sources, but the following are a few that I found helpful.

- ES6 Browser Compatibility Matrix
- Trampolines in Javascript - Reginald Braithwaite
- ES6 Tail-call Optimization Explained - Kyle Owen
- What is the Javascript Execution Context & Stack - David Shariff

javascript recursion es6 tail call optimization

Share this:

View Comments...

**13 Comments** **datchley.name** **jiaheng wang**

**Recommend** 4 Tweet Share **Sort by Best**

Join the discussion…

**llacroix** • 3 years ago

One thing you can do to improve performances is to return an object instead of a function. As a recursive tail call would reuse the same "memory". As an example, return an object of type Param. Use the attributes of that objects in your method and return this object when you want to continue. If you want to return something, then return an object of a different type than Param.

As you're reusing the same object, you don't have to allocate memory when calling continuations. If you return a trampoline, you still have to create a closure on each call to the continuation. With my method, you return the state for the next call and reuse it later.

On my computer using node, the result is almost 50% faster than the trampoline returning a function continuation.

1 • Reply • Share ›

**datchley** Mod → llacroix • 3 years ago

If you have some code, I'd love to take a look at it. But, the trampoline function is only called once, which returns a

function. That function, when called would then continue to execut the results of the original function passed to trampoline as long as it returned a function (continuation).

If that original function instead, as you are suggesting, returned an object, you would still have to pass that object (or parts of it) to a call to your original function and you'd still be maintaining state, just within an object, and making function calls - unless you're suggesting you do the processing inline, iteratively, in which case it sounds just like a standard loop as in the iterative approach.

But, again, I'd love to see your implementation.



**llacroix** ➚ datchley • 3 years ago

http://paste2.org/7YsnXZ1K

Here, range2 is using an object as param, range3 is using the param as "this". I'm not sure if one has a speed improvement on the other but I felt it could make sense to check. Both return an object with a key "value". This could allow chaining the methods to build parameters for an other function using the same trampoline. Obviously creating the parameter with new Param is quite verbose but this kind of remind me of old plain "let" in scheme. You can see that the first value is {s: 1: e: 100, res: []} Which means, that we can remove "res || []".

Anyway, if you can check in jsperf how it runs against the other version. That would be cool



**datchley** Mod ➚ llacroix • 3 years ago

llacroix sorry I hadn't gotten back sooner; but I was swamped and didn't get a chance to play around with your code and do some perf testing.

Indeed, your implementation is definitely faster than using continuations (or thunks), since it removes the need for wrapped function calls, reducing the number of calls and stack frames by simply passing the params back.

A couple of things that sped up the performance on yours was removing the need to iterate of the passed in object in the constructor and removing the use of `instanceof` in the check in the trampoline. You

can see this in the second test of using Params, which I modified with the help of Vyacheslav Egorov, one of the v8 compiler devs.

You can see the results of the various tests here: http://jsperf.com/iterating...

Something else to notice, is that v8 is significantly better at optimizing code than other Javascript engines. None of these, even using thunks, comes close to writing the function or algoithm in an iterative style; but, with TCO around the corner in v8 and other engines, the ability to do tail-recursive calls directly; and not having to re-write the primary function to take into account returning thunks or special param type objects, will likely be much closer if not comparable to iterative performance.

Fantastic idea though, and I appreciate you sharing the code! Definitely an implementation to consider depending on the application and its needs.

Reply • Share ›

**llacroix** ➚ datchley • 3 years ago

I guess the optimized version of Param is cool, until value isn't defined then loop. This way, there is no need to use instanceof. Thanks for trying out that alternative. Anyway, until TCO will be supported, anything trying to emulate TCO is more or less just a hack.

Apparently ES6 should have TCO but it's not there yet, unfortunately.

Reply • Share ›

**Alexey Nesterov** • 2 months ago

Thank you, man! Great explanation.

Reply • Share ›

**Elliot Cameron** • 4 months ago

Checking instanceof Function prohibits the function from actually returning a function. :)

Reply • Share ›

**Yi Wang** • 5 months ago

so basically, "trampoline" function is turning recursion to a while loop I think

loop I think

Reply • Share ›

**Guillaume Lathoud** • 5 months ago

Here's a possibility to use tail calls - including mutual recursion - in today's JavaScript *and* obtain excellent performance: http://glat.info/fext

Reply • Share ›

**Vlad** • 10 months ago

Awesome, thank you for such a great blog!

Reply • Share ›

**Milton Inostroza** • 2 years ago

Hi Dave,

## Dave Atchley

Read more posts by this author.