Gettin' Freaky Functional w/Curried JavaScript

Posted on 14th January 2015 by Erin Swenson-Healey in Web

Partial application refers to the practice of filling in a functions parameters with arguments, deferring others to be provided at a later time. JavaScript libraries like Underscore facilitate partial function application – but the API isn't for everyone. I, for one, feel icky sprinkling calls to `_.partial` and `_.bind` throughout my application.

Curried functions (found in Haskell and supported by Scala, among others) can be partially-applied with little ceremony; no special call format is required. In this blog post, I'll demonstrate an approach to currying JavaScript functions at the time of their definition in a way that enables partial function application without introducing lots of annoying parens and nested function expressions.

Currying in JavaScript: Usually a Huge Pain

A function can be said to have been "curried" if implemented as a series of evaluations of nested unary functions (representing each "parameter") instead of as a single, multiple-arity function (currying a unary function isn't all that interesting).

This binary function:

```
//
// (Number, Number) → Number
//
function sum(x, y) {
  return x + y;
}
```

...could be rewritten as a curried function like this:

```
//
// Number → Number → Number
//
function sum(x) {
  return function(y) {
    return x + y;
  };
}

var addTen = sum(10); // type: Number → Number

addTen(20); // 30
addTen(55); // 65
```

This approach starts to become unwieldy as the number of values to be provided by the caller grows:

```
//
//  type: Number → Number → Number → Number → Number
//
function sumFour(w) {
  return function (x) {
    return function (y) {
      return function (z) {
        return w + x + y + z;
      }
    }
  }
}

sumFour(1)(2)(10)(20); // 33

var addTen = sumFour(3)(3)(4);
addTen(20); // 30
```

Yuck. We get partial application – but with a very low signal-to-noise ratio. Four sets of parens are required to call `sumFour` – and its implementation requires three nested function expressions.

Higher-Order Currying for the Lazy

As luck may have it (thanks, Brendan Eich), we can write a higher-order currying function that can be applied to a fixed-arity function, returning a curried version of the original. Making things even sweeter, our curried function can be called like an

idiomatic JavaScript function (one set of parens and comma-separated arguments) instead of like the `sumFour` mess above – all without losing the ability to perform low-ceremony partial application.

We'll jump straight into the implementation:

```
function curry(fx) {
  var arity = fx.length;

  return function f1() {
    var args = Array.prototype.slice.call(arguments, 0);
    if (args.length >= arity) {
      return fx.apply(null, args);
    }
    else {
      return function f2() {
        var args2 = Array.prototype.slice.call(arguments, 0);
        return f1.apply(null, args.concat(args2));
      }
    }
  };
}
```

Calling `curry` with a multiple arity function produces a new function which, when called, either returns a partially applied version of the original or the result of full application. Leveraging `Function.prototype.length`, we can compare the number of provided arguments (which we cache in `args` after each function call) with the arity of the original. If we've received enough arguments to fully apply, the original function is called. If not, we recurse into our helper (`f1`)... returning a new, partially applied function.

Let's see what it looks like to call this function – and how to interact with the resulting function.

```
var sumFour = curry(function(w, x, y, z) {
  return w + x + y + z;
});
```

The name `sumFour` is now bound to a curried function returned from our call to `curry` in which we passed an anonymous, four-parameter function expression. Calling the resulting function will either result in a value (the sum of all four numbers) or a new function with some of its parameters filled in with values provided by the caller.

```
var
  f1 = sumFour(10),         // returns a function awaiting three arguments
  f2 = sumFour(1)(2, 3),    // returns a function awaiting one argument
  f3 = sumFour(1, 2, 7),    // returns a function awaiting one argument
  x  = sumFour(1, 2, 3, 4); // sumFour has been fully applied; x is equal to 1+2+3+4=10
  x2 = sumFour(1)(2)(3)(4); // fully applied; x2 equals 10
```

There are a few things of importance in this last example:

1. Our curried function can be applied to multiple arguments without the need for multiple sets of parameters
2. Our curried function can be partially applied with library support (beyond what was required to create the function bound to `sumFour`
3. The result of partially applying our curried function to arguments results in a function that is also curried

Cool, right?

In hopes of illustrating how this crazy function works, let's walk through the evaluation of the expression `(sumFour(1)(2, 3, 4))`:

1. Evaluate the subexpression `sumFour(1)`, which (you guessed it) applies `sumFour` to `1`
2. Call `f1` with a single argument `1`
3. Check to see if `f1` has been called with a number of arguments greater or equal to the number of parameters expressed in the anonymous function passed to `curry` (four: w, x, y, and z). The predicate returns false and `f2` is returned.
4. `f2` is called with three arguments `2` and `3` and `4`
5. Concatenate the first argument `1` with new arguments and apply `f1` to all four
6. Check number of arguments (now 4) against arity of original function (4) which satisfies predicate. The original function is then (fully) applied to all four arguments.

In Summary

JavaScript implementations don't make things easy for programmers that want to work with partially-applied and curried functions (lots of parens, nested function expressions). In this blog post, you've seen a demonstration of how one might roll a higher-order currying function that allows for partial function application without a ton of hassle.

Related Reads

1. Composing Asynchronous with Synchronous Functions in JavaScript
2. Tidying Up a JavaScript Application with Higher-Order Functions