Using Divergence and Rebase

Spencer Tipping

May 23, 2010

Contents

Introduction

Divergence is a JavaScript library centered around functions and function manipulation. It extends the prototypes of the core JavaScript data types to provide methods that make it easier to program in a functional style, and provides a coherent paradigm for generating functions from different types of values.

Rebase is a Divergence module that decompiles, transforms, and recompiles functions to extend the capabilities of JavaScript. This includes adding syntactic macros, operator overloading, and string interpolation to the language by default, and it can also be used as a basis for creating other extensions.

This guide is intended for anyone who read *JavaScript in Ten Minutes*[1] and would like to see some of the ideas put into practice. I think it covers everything, including syntactic macros, continuations, etc. If you haven't yet read *JavaScript in Ten Minutes*, I recommend it as pre-reading; this guide picks up about where it left off.

This guide is separated into two parts. The first part goes over the Divergence core library, and the second part introduces Rebase and the ways that it interacts with Divergence.

Getting Started

Divergence and Rebase are both hosted on Github. They can be retrieved from `http://github.com/spencertipping/divergence` and `http://github.com/spencertipping/divergence-rebase`, respectively. If you've checked out the user guide repository, then you're all set – you can open `shell.html` in a non-IE browser and run the examples from there.

If you don't have a local checkout or prefer a more hands-on approach, then you can download `divergence.js` and `divergence.rebase.js` from the Rebase repository, save them somewhere, and create an HTML file with the following contents:

```
<html>
  <head>
    <script src='divergence.js'></script>
    <script src='divergence.rebase.js'></script>
```

[1] Freely available at `http://github.com/spencertipping/js-in-ten-minutes`

```
    <script>
      // Examples can go here
    </script>
  </head>
  <body>
  </body>
</html>
```

Part I

Divergence Core

Chapter 1

Everything is a Function

The main goal of Divergence is to create a way to convert objects of any basic type into functions. For example, the string `'$0 + $1'` can be promoted into a function that adds its first two parameters using the `fn()` method. Similarly, arrays, numbers, booleans, and regular expressions all have defined promotions into functions. I refer to objects that have a `fn()` as *functionals*.

1.1 Booleans and Numbers

The promotion patterns for these are simple. If `n` is a number, then `n.fn()` returns a function which returns its $\mathtt{n}^{\text{th}}$ argument (where arguments' indices are zero-based).[1] So, for example:

```
var n = 1;
n.fn() (5, 6)        // => 6
```

Because of JavaScript's eager lexing, you can't say things like `5.fn()`; in this case, the dot is considered to be a part of the floating-point literal `5.`. The way to fix this is to put a space after `5`, so you have `5 .fn()`. Not quite as nice, but fortunately most function promotion is done implicitly in practice.

Booleans are promoted as uncurried Church booleans[2] – that is, `true` returns its first parameter and `false` returns its second.

1.2 Regular expressions

These get promoted into functions that attempt to match the regular expression against a string. For example:

[1] I haven't tried it yet, but I suspect bad things happen if you use floating-point or negative numbers.

[2] Don't worry if this sounds unfamiliar; it's just the theoretical background.

```
/foo (bar)/.fn() ('foo bar bif')          // => ['foo bar', 'bar']
/foo (bar)/.fn() ('bar bif baz')          // => null
```

Because failed matches return `null`, we have the very nice idiom that `strings.grep(/pattern/)` does exactly what you'd expect.[3]

Internally the `RegExp.exec()` method is used to achieve this behavior.

1.3 Arrays

`fn()` distributes across arrays. That is, `[f, g, h].fn()(x)` is equivalent to `[f.fn()(x), g.fn()(x), h.fn()(x)]`. One use of this is to reorder an argument list:

```
var g = f.flat_compose ([1, 0]);    // flat_compose implicitly calls fn()
g (x, y)                            // the same as f (y, x)
```

1.4 Functions

The `fn()` method exists for functions for the sake of uniformity. It just returns the function.

1.5 Strings

The `fn()` method for strings does a lot. In the simplest case, it just wraps the string inside `(function() {return <your string here>})` and runs it through `eval()`. For example, `'arguments[0]'.fn()` is the identity function.

Obviously, nobody wants to type `arguments` that many times, so Divergence provides a simple regular-expression-based macro processor that does some expansions for you.[4] The ones that are enabled by default are:

1. Expressions of the form `$n`, where `n` is an integer, are expanded to `arguments[n]`. For example, `'$0 + $1'.fn()` adds its first two arguments.

2. The expression `$_` is expanded to `this`.

3. The expression `@_` is expanded to `Array.prototype.slice.call(arguments)`.[5]

4. Expressions of the form `@foo`, where `foo` is an identifier, are expanded to `this.foo`.

[3] `grep` is defined for arrays; see section **??**

[4] Note that because it just uses regular expressions on the input, there's no pretense of protecting things inside strings, etc. Rebase implements a much more sophisticated macro processor that avoids these problems.

[5] This promotes `arguments` into a proper array.

5. Expressions of the form `{|x, y, z| x + y |}` are expanded to expressions of the form `(function (x, y, z) {return x + y})`.

6. Expressions of the form `{< expression >}` are expanded to expressions of the form `(function () {return expression})`.

After performing those substitutions, the result is put inside `(function () {return <result>})` and run through `eval()`.

Here are some examples:

```
'$0[$1]'.fn() ({foo: 'bar'}, 'foo')                    // => 'bar'
'$_ + $0'.fn().call (5, 6)                             // => 11
'$0.map({< $0 + 1 >})'.fn() ([1, 2])                   // => [2, 3]
'$0.fold({|x, y| x * y |}).fn() ([1, 2, 3, 4, 5])      // => 120
'@foo + @bar'.fn().call ({foo: 10, bar: 5})            // => 15
'@_.length'.fn() (10, 15, 20)                          // => 3

// Important:
'@_.sort(), @_'.fn() (50, 40, 30)                      // => [50, 40, 30]
```

The last example illustrates an important point here. The substitutions are purely textual, so any expressions that get generated will be run multiple times and will probably generate unaliased values.

Chapter 2

Operations on Functionals

Functionals, as defined in the previous section, are objects with `fn()` methods. Because they present this interface, we can define operations that work on any functional and add them to all of the basic prototypes. All of the following functions are present on every functional.

2.1 `compose`

Composes two functions. Specifically, `f.compose(g)` is equivalent to the function:

```
function () {
  return f.fn() (g.fn().apply (this, arguments);
}
```

Notice that both `f` and `g` are automatically promoted into functions. This is true of almost all of the higher-order functions provided by Divergence.

2.2 `flat_compose`

Composes two functions, but expands the array returned by `g` and supplies the values as arguments to `f`. For example:

```
var f = '[$0, $1, $1]';           // A string, not a function
var g = '$0 + $1 * $1';
f.flat_compose(g) (3, 4)          // => 19
```

Specifically, `f.flat_compose(g)` is equivalent to:

```
function () {
  return f.fn().apply (this, g.fn().apply (this, arguments));
}
```

2.3 curry

Takes an integer `n` and returns a function that will evaluate the original when called `n` times. For example:

```
'$0 + $1 + $2'.curry(1) (1, 2, 3)          // => 6
'$0 + $1 + $2'.curry(2) (1, 2) (3)         // => 6
'$0 + $1 + $2'.curry(3) (1) (2) (3)        // => 6
'$0 + $1 + $2'.curry(4) (1) (2) (3) ()     // => 6
```

Arguments don't have to occur in any particular pattern; they're just stuck onto a queue as they're collected. If you want a more traditional implementation of `curry` that chops off extras, for example, then `flat_compose` and arrays will probably work:

```
var f = '$0 + $1 + $2'.curry(2).flat_compose([0, 1]);
f (1, 2, 3) (4, 5)                         // => 7
```

In this example, the first invocation of `f` took only `1` and `2`. `3` was lost because the array `[0, 1]` doesn't return it anywhere. So at this point the queue contains `1` and `2`. On the next invocation, `4` and `5` are passed in, so the queue is now `1, 2, 4, 5`. The function adds the first three arguments, totaling 7.

2.4 proxy

This function serves two purposes. One is to get a new function that is extensionally equivalent to, but referentially distinct from, the original function,[1] and the other is to completely intercept the invocation of the function.

Specifically, `f.proxy()` is equivalent to

```
function () {return f.fn().apply (this, arguments)}
```

and `f.proxy(g)` is equivalent to (get ready, this is confusing):

```
function () {
  return f.fn().apply.apply (f, g.fn().apply (this, arguments));
}
```

Did you catch that? `apply` is itself a function, so we can `apply` it to things. In this case, we apply it to the result of `g`, which is expected to return an array of the form `[t, [x1, x2, ...]]` – `t` refers to the `this` value that `f` should receive, and `x1, x2, ...` are the arguments passed to `f`. This is the elephant-gun of composition; most of the time using `compose` or `flat_compose` will do the job.

[1] Things that are *extensionally equivalent* have the same observable behavior, and things that are *referentially distinct* refer to different objects. This condition is useful when you want to change the state of one object without affecting the other. One particular use case might be assigning a function as a method for multiple classes. Perhaps classes tag the functions, e.g. `method.belongsTo = theClass`. In this case you want the methods to behave the same way but have different attributes.

2.5 bind

The canonical implementation of `bind`, though this one doesn't preload arguments. It also marks the output function with a reference to the original and to the binding, so:

```
var o = {foo: 0, bar: 1};
var f = '@foo += @bar + $0';
var g = f.bind(o);
g(5)                          // => 6
g(3)                          // => 10
g.original                    // => f
g.binding                     // => o
```

The purpose of `bind` is covered in *JavaScript in Ten Minutes* and numerous other sources, but the idea is to fix `this` inside a function so that:

```
'$_'.fn() ()                  // => [object global]
'$_'.bind(5)()                // => 5
```

This is not much of an introduction to the concept of function binding; if the purpose or practical use of `bind` is at all unclear, you should definitely read something that goes over what it does.

2.6 ctor

The `ctor` function provides a one-step way to initialize the prototype of a function. For example, a quick definition of a 2D vector:

```
var vector2 = '@x = $0, @y = $1'.ctor (
              {plus: 'new @constructor(@x + $0.x, @y + $0.y)'.fn(),
                dot: '@x * $0.x + @y * $0.y'.fn()});
var v1 = new vector2 (3, 4);
v1.dot (v1)                           // => 25
```

`ctor` takes any number of hashes; they will all be merged together into the prototype of the function.

2.7 type

Prototypes have their advantages and disadvantages. The advantage of performance is paired with the disadvantage of irregular and non-first-class syntax. To illustrate this, consider a proxy function for a class constructor; its job is to pass whatever arguments you give it into the constructor for a class:

```
var my_class = function () {...};
my_class.prototype.foo = function () {...};
var my_proxy = function () {
  return new my_class ();
};
```

This doesn't quite work, since any arguments passed to `my_proxy` are lost. It would be nice to write it this way:

```
return new my_class.apply (this, arguments);
```

but that isn't what JavaScript's authors had in mind. As far as I know there is no way around this problem other than using a blank constructor and a first-class initializer:

```
var my_class = function () {};                    // This function is empty
my_class.prototype.foo = function () {...};
my_class.initialize = function () {...};          // The real constructor
var my_proxy = function () {
  return my_class.initialize.apply (new my_class (), arguments);
};
```

This pattern is abstracted by `type`, which behaves exactly like `ctor` does, except that it uses an empty constructor, uses the given function as the initializer, and returns that initializer. An important consequence of this is that the prototype becomes inaccessible by the usual route; instead, you have to create an instance and run `x.constructor.prototype` to alter it.[2]

2.8 Preloading

A feature of the `fn()` method I haven't mentioned is that you can use it to preload arguments to a function. For example:

```
var f = '$0 + $1'.fn (1);
f (2)                     // => 3
f (3, 4, 5)               // => 4

var g = '@_.length'.fn (3, 4, 5);
g (1, 2)                  // => 5
g (1)                     // => 4
g ()                      // => 3
```

This feature is present for all of the `fn()` methods, not just on the one for strings:

```
var f = [0, 1, 2].fn ('foo', 'bar');
f ('bif')                 // => ['foo', 'bar', 'bif']
```

[2] In the future I may introduce an option to make the prototype visible.

Part II

Rebase

Chapter 3

Operator Overloading

One of the ways Rebase extends JavaScript is with operator overloading. In practice, this entails translating operator invocations into method calls.[1] For example, this function:

```
function (x, y) {return x + y}
```

would be rebased into:

```
function (x, y) {return x['+'](y)}
```

3.1 Standard operators

Because this transformation also affects regular values such as numbers and strings, Rebase installs handler functions for these objects. If you look at `String.prototype`, for instance, after Rebase is included, you'll probably see a bunch of functions whose names are operators; these are compatibility functions that just delegate to those operators to provide normal operation after the operators have been converted.

Rebase also provides some default operators in places where JavaScript's defaults aren't very helpful:

```
[1, 2, 3] * (function (x) {return x + 1})     // => [2, 3, 4]
[1, 2, 3] % (function (x) {return x % 2})     // => [1, 3]
[1, 2, 3] + [4, 5, 6]                         // => [1, 2, 3, 4, 5, 6]
[1, 2, 3] / (function (x, y) {return x + y})  // => 6
[1, 2, 3] >>$- f                              // => [1, 2, 3].flat_map (f)
```

[1] Some operators are not translated because they cause behavior to change. These include ==, ===, !=, !==, =, ++, --, &&, ||, ?:, function calls, dot-lookups, hash-lookups, commas, and !.

These operators are aliased to regular methods; * is aliased to `map`, / to `fold`, % to `grep`, + to `concat`, and `>>$-` to `flat_map`.[2] The `>>$-` notation comes from monadic binding and was chosen because it looks vaguely like Haskell's >>=.[3]

At this point you might reasonably ask about `>>$-`; this is certainly not a normal-looking operator! Quite right – Rebase allows you to combine binary operators to form new compound ones around certain identifiers. `$` is one such identifier (these are called "sandwiches"), and the rule is that if the compiled expression tree includes two adjacent binary operators around a sandwich identifier, then they get merged to form something like `>>$-`. Section **??** goes over this in more detail.

3.2 Defining new operators

It's actually very simple to define new operators for your classes, or to redefine the ones that Rebase defines for standard classes. Here is how you might go about defining a 2D vector, for instance:

```
var vector2 = function (x, y) {this.x = x; this.y = y};
vector2.prototype.toString =
  function () {return '<' + this.x + ', ' + this.y + '>'};
vector2.prototype['+'] =
  function (rhs) {return new vector2(this.x + rhs.x, this.y + rhs.y)};
vector2.prototype['-'] =
  function (rhs) {return new vector2(this.x - rhs.x, this.y - rhs.y)};
vector2.prototype['*'] =
  function (rhs) {return new vector2(this.x * rhs.x, this.y * rhs.y)};
vector2.prototype['/'] =
  function (rhs) {return new vector2(this.x / rhs.x, this.y / rhs.y)};
```

An equivalent and more compact way to do this using Divergence and `eval` inside a rebased function:

```
var vector2 = '@x = $0, @y = $1'.ctor (
  ([{toString: _ >$> '<#{this.x}, #{this.y}>'}] +
   '+ - * /'.split(' ') *
     (op >$> op.maps_to (
       'new vector2 (@.x #{op} $0.x, @.y #{op} $0.y)'.fn()))) / d.init);
```

Now you can use these operators:

```
d.rebase (function () {
```

[2] `map`, `grep`, `flat_map`, and `fold` are provided by the Divergence core library. `concat` is a standard array method.

[3] Because ≫= is an assignment operator in JavaScript, the left-hand side must be a proper lvalue. This is enforced within the JavaScript grammar, so had I used this notation all monadic binding would have do be done against unadorned variables.

```
  var v1 = new vector2 (3, 4);
  var v2 = new vector2 (1, 6);
  alert (v1 + v2);          // Alerts '<4, 10>'
}) ();
```

You can define compound operators the same way:[4]

```
vector2.prototype['-$*'] = function (rhs) {
  return this.x * rhs.x + this.y * rhs.y;
};
```

(Or, more idiomatically, `vector2.prototype['-$*'] = '@x * $0.x + @y * $0.y'.fn();`)

```
d.rebase (function () {
  var v1 = new vector2 (3, 4);
  alert (v1 -$* v1);        // Alerts '25'
}) ();
```

[4]Some compound operators are macros and will never be produced. See section **??** for more details.