Divergence Improved

Spencer Tipping

August 31, 2010

Contents

Part I

Using Divergence

Chapter 1

Introduction

The original Divergence[1] has some shortcomings. For one thing, it puts a bunch of methods into the global prototype namespace (probably its biggest problem). This used to break jQuery, and now breaks jQuery UI's tab and accordion components.[2] Another problem is that macro definitions are permanent, unstructured, and collision-prone.

This rewrite of Divergence solves both problems. The only object placed into the global namespace is called `divergence`, and all customization is done to it or a copy of it. Macro definitions are scoped to a particular instance of Divergence; they are not global by default.

[1] http://github.com/spencertipping/divergence

[2] Not that is has to in theory; someone made the assumption that all array methods would be `dontEnum`, which isn't the case if you add stuff. The workaround is to use `hasOwnProperty`, or more importantly, not to use `for..in` on arrays.

Chapter 2

Building Functions

Just like the original Divergence, this version is all about building functions. Also just like the old version, it specifies conversions to promote any built-in data type into a function, and lets you use your own data types if they provide `.fn()` methods.

2.1 Numbers

Numbers are now much more expressive. Just like before, 0, 1, 2, 3, and 4 map to the first five positional parameters. However, there are some new cases:

2.1.1 Large positive integers

Integers larger than 4 are converted into hexadecimal and interpreted, where each digit is a command in a stack-based language. The stack's initial contents are the positional parameters, where `arguments[0]` is at the top and `arguments[arguments.length - 1]` is at the bottom. Digits are interpreted from left-to-right; so, for example, the number `0xab` is interpreted as the command `a` followed by the command `b`. The following commands are understood (along with mnemonics in the footnotes):

`5` Swap the top two stack entries.[1]

`6` Drop the top stack entry.[2]

`7` If the next digit is a 0, 1, 2, 3, or 4, then push that digit as a number onto the stack. Otherwise, push the stack depth onto the stack and process the next digit normally.[3]

[1] "5" looks kind of like "S", which stands for Swap.
[2] "6" is a backwards "d", which stands for Drop.
[3] "7", when rotated 180°, looks like the letter "L", which stands for Literal or Length.

`8` Duplicate the top stack entry.[4]

`9` Drop the second entry.[5]

`a` Add the two arguments on the top of the stack, and push the result. This also works on strings. If the top of the stack is an array, then push a new array consisting of the stack top concatenated with the second stack element; that is, `stack[0].concat([stack[1]])`.[6]

`b` Subtract `stack[1]` from `stack[0]`, pop both, and push the result. If either argument is non-numeric, then this operator applies `||` to the top two stack entries instead; that is, `stack[0] || stack[1]`.[7]

`c` Pop twice, multiply, and push. If either argument is non-numeric, then this operator applies `&&` to the top two stack entries instead; that is, `stack[0] && stack[1]`.[8]

`d` Pop twice, divide, and push. Operands are ordered the same way as they are for subtraction. If either argument is non-numeric, then this operator dereferences the stack top by the stack second instead of performing division; that is, `stack[0][stack[1]]`. If the stack top is undefined or null, then the second argument is dropped silently instead of being used for dereferencing.[9]

`e` Negate the top stack entry if it's a number. If it's not a number, then apply logical negation.[10]

`f` Invoke the top stack entry on the next one, and return the result. If the top of the stack isn't a function, then the current `d()` (that is, the one being used to convert this number to a function in the first place) is used to convert the stack top to a function first.[11]

The digits 0-4 push those positional parameters onto the top of the stack. For example, 0 pushes `arguments[0]`, 1 pushes `arguments[1]`, etc.

Here are some examples:

Listing 2.1 `examples/large-integer-functions.js`

```
d(0xa)    // => function (x, y) {return x + y}          (if numeric)
d(0xa)    // => function (x, y) {return x.concat([y])}  (if x is an array)
d(0xb)    // => function (x, y) {return x - y}          (if numeric)
```

[4] "8" looks like two "0"s.

[5] "9" is "6" upside-down, and `6` drops the top entry.

[6] "a" stands for Add or Append.

[7] "b" stands for suBtract.

[8] "c" stands for Combine, which in regular algebra is generally multiply, and multiplication translates to and in Boolean algebra.

[9] "d" stands for Divide or Dereference.

[10] "e" stands for nEgate.

[11] "f" stands for Function, obviously.

```
d(0xb)    // => function (x, y) {return x || y}         (if non-numeric)
d(0xc)    // => function (x, y) {return x * y}          (if numeric)
d(0xc)    // => function (x, y) {return x && y}         (if non-numeric)
d(0xd)    // => function (x, y) {return x / y}          (if numeric)
d(0xd)    // => function (x, y) {return x[y]}           (if x is non-numeric)

d(0x8a)   // => function (x)    {return x + x}          (if numeric)
d(0x8aa)  // => function (x, y) {return x + x + y}      (if numeric)

d(0x65b)  // => function (x, y, z) {return z - y}       (if numeric)
d(0x95b)  // => function (x, y, z) {return z - x}       (if numeric)
d(0xdd)   // => function (x, y, z) {return x[y][z]}     (if non-numeric)
d(0xdd)   // => function (x, y, z) {return (x / y) / z} (if numeric)

d(0x88cc) // => function (x) {return x * x * x}         (if numeric)
d(0xee)   // => function (x) {return !!x}               (if non-numeric)
d(0x7a)   // => function (x) {return x + 1}             (if numeric)
d(0x74a)  // => function (x) {return x + 4}             (if numeric)
d(0x748cc) // => function (x) {return x * 16}           (if numeric)
d(0x2f1f) // => function (f, x, y) {return f(y)(x)}
d(0x7)    // => function () {return arguments.length}
```

To be portable, you should use at most seven hex digits. Some browsers have integer math that can change the sign if the 32-bit is set.

2.1.2 Positive floating-point numbers

todo

2.1.3 Negative integers

todo

2.1.4 Negative floating-point numbers

todo

2.2 Strings

Strings delegate to domain-specific language parsers, and this delegation is managed entirely by the first character of the string. Divergence includes a few such languages built-in, and others can be defined later on. Here are the ones Divergence comes with:

2.2.1 Dereferencing

If a string begins with a dot, then it is treated as a monadic dereferencer. It will not fail if it hits a null reference; it simply stops dereferencing at that point. So, for example:

Listing 2.2 examples/dereferencing-functions.js

```
d('.foo.bar')({foo: {bar: 5}})          // => 5
d('.foo.bar')({foo: {bif: 5}})          // => undefined
d('.foo.bar')({bif: {baz: 5}})          // => undefined
```

2.2.2 String replacement

If a string begins with /, it is treated as a replacement command. All regexps are considered to have the modifier g implicitly; this can be changed by anchoring the regexp to the beginning or end of the string.

Listing 2.3 examples/string-replacement-functions.js

```
d('/foo/bar')('foobar')                 // => 'barbar'
d('/f(o)o/b$1r')('foobar')              // => 'borbar'

// Multiple replacements are also possible:
d('/foo/bar; /bif/baz')('foobif')       // => 'barbaz'

// And conditionals:
d('/foo/bar && /bif/baz')('foobif')     // => 'barbaz'
d('/foo/bar && /bif/baz')('forbif')     // => 'forbif'
d('/foo/bar || /bif/baz')('foobif')     // => 'barbif'
```

2.2.3 Named-argument expressions

If a string begins with |, then it is parsed as a named-argument function. For example:

Listing 2.4 examples/named-argument-expressions.js

```
d('|foo| foo + 1')      // => function (foo) {return foo + 1}
d('|x, y| x + y * 2')   // => function (x, y) {return x + y * 2}
d('|x| (|y| x + y)')    // => function (x) {return function (y) {return x + y}}
```

A couple of macros are available for brevity. One expands expressions of the form `@foo`, where `foo` is some identifier, into `this.foo`. (Using `@` without a following identifier expands into `this`.) The other provides shortcuts for `call` and `apply`: the operator `#c` expands to `.call`, and the operator `#a` expands to `.apply`. For example:

Listing 2.5 examples/named-argument-expression-macros.js

```
d('|foo| foo#c(@, @x)')         // => function (foo) {return foo.call(this, this.x)}
d('|x, y| @x = x, @y = y')      // => function (x, y) {return this.x = x, this.y = y}
d('|f, xs| f#a(f, xs)')         // => function (f, xs) {return f.apply (f, xs)}
```

Chapter 3

Instances

Unlike before, Divergence isn't just one function. You can create a new instance of Divergence with its own configuration, which can be useful for isolated regions of code that require a particularly common pattern, unit tests, etc. The most common way to do this is to use `new`:

Listing 3.1 `examples/instance-new.js`

```
new divergence (function (d) {
  // Code in here can access d, which is a copy of the global divergence.
  // To create a copy of d:
  new d (function (new_d) {
    // new_d is a copy of d, and will inherit any d-specific customizations
    // specified earlier.
  });

  // Another way to do it:
  d.clone (function (new_d) {
    // This is exactly the same as above, except that its return value is
    // intact.
  });

  // To grab the copy for later:
  var new_d = d.clone();
});
```

3.1 Return value of `new`

Because `new` always returns a hash, not a function, using the `new divergence(f)` constructor won't return either `f`'s return value, nor will it return the new Divergence. Instead, it returns an object containing both. So, for example:

Listing 3.2 examples/instance-new-return.js

```
var result = new divergence (function (d) {
  d.foo = 'bar';
  return 5;
});
result.result             // => 5
result.divergence.foo     // => 'bar'
```

If you care about the return value of your function, it's probably easier to use `divergence.clone`:

Listing 3.3 examples/instance-clone-return.js

```
var result = divergence.clone (function (d) {
  d.foo = 'bar';
  return 5;
});
result                    // => 5
```

In this case there is no way to access the scoped `d`, though you can return it explicitly if you want to hang on to it.

3.2 Roles

Sometimes you want to keep a set of customizations around for reuse. You can do this by creating a *role*, which is simply a function that modifies a Divergence instance. For example, this role adds an `assert` method to `d`:

Listing 3.4 examples/instance-role-assert.js

```
divergence.role.create ('assert', function (d) {
  d.assert = function (what, message) {
    if (! what) throw new Error ('Assertion failed: ' + message);
    return what;
  };
});

d.assert          // => undefined
```

Roles are attached to whichever Divergence instance they were created on. You can now use that role:

Listing 3.5 examples/instance-role-use.js

```
new divergence (function (d) {
  d.role.use ('assert');                  // Adds 'assert' to d in-place
  d.assert (3 === 3, 'basic math');       // => true
});

```

```
new divergence.using ('assert', function (d) {
  // d is a clone of divergence, but also with 'assert'
  d.assert (true, 'should pass');         // => true
});

divergence.role.use ('assert');           // Not a great idea; see next paragraph
divergence.assert (1, 'truthy 1');        // => 1
```

Roles can't be "un-used", so generally the best approach is to add a role to a copy of your divergence function.

Part II

Extending Divergence