Divergence Improved

Spencer Tipping

September 5, 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.

1.1 Conventions

I'll often make reference to "the `d` function". When I say this, I'm referring to either the global `divergence` function or a customized clone of it. (The important thing is that it is one of a class of functions that ultimately come from `divergence`; see section 6 if customization and cloning are unfamiliar.)

1.2 Disclaimer

Divergence is dangerous for developers in a couple of ways. First, it might not work; I'm designing this library as an academic as much as a developer, so there will be times when I make impractical choices for the sake of some form of purity. Second, it gives developers many ways to do things, and few of them are at all readable. For example:

```
d(0x71a)           // Adds one to the first argument
```

[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.

```
d(0x8ee5b)          // Returns first arg if truthy, false otherwise
d(0x8c8c)           // Returns first arg to the fourth power
d('xn5b')           // Returns first arg if truthy, null otherwise
d('x[1]')           // K function
d('x')              // Identity
d(0)                // Identity
d('x#p')            // Returns arguments[] as an array
d('x68t_R')         // Returns second argument if truthy, throws it otherwise
```

To be fair, each of these functions can be written more readably (in the same order as above):

```
d('|x| x + 1')
d('|x| x || !!x')
d('|x| x*x*x*x')
d('|x| x || null')
d('|x| (|y| x)')
d('|x| x')
d('|| arguments[0]')
d('|| Array.prototype.slice.call (arguments)')
function (x, y) {
  if (y) return y;
  else   throw y;
}
```

Divergence isn't there to make code readable in the small, however. Its purpose is to make easy things really easy, and to make some hard things easier. It does this by helping you condense your code semantically; things that are unrelated to your core logic can be packed into very few characters and peripherally mentioned, while core logic can occupy most of the code space. For example:

```
var variable = function (name) {
  this.name  = name;
  this.value = undefined;
};
variable.prototype.assign = function (v) {
  this.value = v;
  return this;
};
```

Using Divergence you could write this instead:

```
var variable = d('|name| @name = name, @value = undefined');
variable.prototype.assign = d('|value| @value = value, @');
```

Notice that the mundane code takes up less space, and the definitions relatively more. That's what Divergence was built to do.

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. As a special case, the command "77" deletes all but the top stack entry.[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(0x25f15f)     // => 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.[12]

2.1.2 Negative integers

Negative integers are treated just like positive ones, except that the entire stack is returned as an array. This can be useful for reordering or combining things, but is most useful when used to transform a function (see chapter 3). For example:

Listing 2.2 examples/negative-integer-functions.js

```
d(-0xa)         // => function (x, y, ...) {return [x + y, ...]}
d(-0x7188)      // => function (...) {return [1, 1, 1, ...]}
d(-0x10)        // => function (x, y, ...) {return [y, x, ...]}
d(-0x77)        // => function (x, ...) {return [x]}
d(-0x86)        // => function (x, ...) {return [x, ...]}
d(-0x7721)      // => function (x, y, z, ...) {return [x, z, y]}
```

2.1.3 Positive floating-point numbers

TBD

[12] And anything beyond seven digits significantly increases the chances that whoever's maintaining your code later will kill you.

2.1.4 Negative floating-point numbers

TBD

2.2 Strings

Strings delegate to domain-specific language parsers, and this delegation is managed entirely by the first non-whitespace 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 2.3 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
```

If passed multiple arguments, the function will return an array of results.

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 2.4 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'
```

If invoked on an array or a hash, the function will distribute across their values and return a transformed copy. If invoked on multiple arguments, each argument is transformed and the results are returned in an array.

2.2.3 Named-argument expressions

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

Listing 2.5 `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 2.6 `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)}
```

The `@` prefix preserves `this` in the context of a subfunction (that is, binds it to `this` outside of the subfunction body). For example:

Listing 2.7 `examples/named-argument-expression-this.js`

```
d('|| @ === (|| @)()').call(true)         // => false
d('|| @ === (@|| @)()').call(true)        // => true
```

Note that the notation is unambiguous in practice. In JavaScript `this` is always boxed, so expressions of the form `this || x` are trivially `this` and would never evaluate `x`. Therefore, there is never a case where `this || x` should be written, and thus the sequence `@|` always refers to the beginning of an inner function.[13]

2.2.4 Destructuring binds

Strings beginning with `[` or `{` are interpreted as pattern matches with bind variables. For example:

Listing 2.8 `examples/destructuring-binds.js`

```
d('[#x, #y, #z]')([1, 2, 3])
// => {x: 1, y: 2, z: 3}

d('{x: {y: {foo: #x}}}')({x: {y: {foo: 'bar'}}})
// => {x: 'bar'}
```

[13]Spaces also disambiguate these cases further, since generally there are spaces in expressions but not on decorators.

```
d('[1, 2, {bif: #x}, @#xs]')([1, 2, {bif: 3}, 4, 5])
// => {x: 3, xs: [4, 5]}
```

Basically, arrays and objects are both allowed as containers, and # is used to mark a variable. The prefix `@#` is used to mark a splice variable, which can occur either in array or object context; this picks up the remaining (unmatched) entries. The results are returned in a flat hash mapping variable names to matched values. Any variable called `_` will match anything and be ignored; you can use it multiple times, in single or splice context. It's useful for things like indicating that an array has a bunch of useless stuff on the end.

You can also put guards onto destructuring binds. For example:

Listing 2.9 `examples/destructuring-bind-guards.js`

```
d('[#x, #y] | x < y')([1, 2])   // => {x: 1, y: 2}
d('[#x, #y] | x < y')([2, 1])   // => false
```

If the match fails either due to structural or constraint problems, `false` is returned. This has some convenient and inconvenient properties. The nice thing is that you won't get any errors if you ask for bound variables; they'll all just be `undefined`.[14] However, `false` will still fail any boolean condition, so you can fall out gracefully. The only bad part is that you can't use the === `undefined` or === `null` check.

Alternatives can be specified using `;`. For example:

Listing 2.10 `examples/destructuring-bind-alternatives.js`

```
d('[#x, #y] | x < y ; [#x, #y] | x > y')([1, 2])        // => {x: 1, y: 2}
d('[#x, #y] | x < y ; [#y, #x] | x < y')([2, 1])        // => {x: 1, y: 2}
d('[#x, #y] | x < y ; [#y, #x] | x < y')([1, 1])        // => false
```

If multiple arguments are provided, the function returns the results from the first match that succeeds:

Listing 2.11 `examples/destructuring-bind-multiple.js`

```
d('[#x, 3, #y] | x == y')([1, 3, 5], [2, 3, 2]) // => {x: 2, y: 2}
```

2.2.5 Terse-form stack functions

Large positive and negative integer values (see sections 2.1.1 and 2.1.2) implement a small stack-based command language. Within strings, however, many more characters are available. Strings beginning with `x` are interpreted in this language, which contains as a subset the language used for hexadecimal integers. Whitespace within these strings is ignored. For example:

Listing 2.12 `examples/terse-form-stack-functions.js`

[14] Unless you've named one of them `constructor` or something, but don't do that.

```
d([0x71a, 'x71a'])(5)           // => [6, 6]
d([0xee,  'xee'])(1)            // => [1, 1]
```

Using a string allows the entry of additional commands, described in the following sections:

Control-flow commands

`g`, `G` Goto a label. The next character is the label name. Used with `l`. For example, the string `'xlaga'` produces an infinite-loop function. The alias `G` always refers to the `_` label; for example, the string `'xLG'` is a quicker way to write an infinite-loop function.

`l`, `L` Creates a named label. The syntax is `lx`, where `x` is some non-whitespace character. Labels can be gone to using `g` or conditional branching. Label definitions are mutable (though I do not advise using them this way), so you can redefine them within a function as the function is running. The alias `L` refers to an autogenerated label ID. The first one is `_`, and subsequent ones are the digits 0 to 9, the lowercase letters, and then the uppercase letters.[15]

`t`, `T` Jumps to a label if the stack top is truthy. (Doesn't modify the stack.) Its complement, `T`, jumps to a label if the stack top is falsy. For example, the function `Tnrlnn` returns its first argument if it's truthy and returns `null` otherwise.[16]

`r`, `R` Returns immediately. The return value is whatever is on the top of the stack. The idiom `ur` can be used to return nothing (analogous to `return;`). The `R` variant throws the stack top instead of returning it.

Array commands

`Pn` Here `n` is a single digit. This command pops `n` items off of the stack and into an array, where the array's first element is the previous stack top. The array is then pushed onto the stack.

`p` Pops a number n off of the top of the stack, and then pops n more items off of the stack and pushes them onto an empty array. The array is then pushed. Analogous to `P`, but the number is determined at runtime.

`U` Unpacks the array on the top of the stack into stack entries. The first element in the array becomes the top of the stack.

`i` A zero-based generalized iota, similar to the iota in APL. Replaces the stack top n with an array of consecutive integers from 0 to $n-1$. Distributes

[15]Seriously, I hope this sentence is never that useful. I mean, who's going to use anything like 63 labels in a one-liner anyway?

[16]A shorter and more readable version of this function is `n5b`.

across arrays and object values. If the top is a string, then it is replaced with an array of right-handed substrings (e.g. `'foo'` has right-handed substrings `'foo'`, `'oo'`, and `'o'`). Booleans, regular expressions, `null`, and `undefined` are wrapped in single-element arrays.

`#` Pushes the size of the stack top onto the stack. The size of arrays and strings is obtained from the `length` property, and the size of an object is the number of keys for which `hasOwnProperty` is true. Numbers, booleans, regular expressions, `null`, and `undefined` have size 0.

Typeclass commands

The commands `*`, `+`, `/`, `%`, and `^` are reserved for typeclasses that provide operator overloading.[17] When you define a typeclass, you can define shorthands for various operations; these are alternative names that can be used in an overloaded-operator context.[18]

Value commands

`@` Pushes `this` onto the stack.

`n` Pushes `null` onto the stack.

`u` Pushes `undefined` onto the stack.

`z` Pushes an empty array onto the stack. Equivalent to `70i`, but faster. For example, `'xz5LV85a5t_'` takes an integer n and returns an array $[n, n-1, n-2, \cdots, 1]$.

`Z` Pushes an empty object onto the stack.

`V` Decrement the item on the top of the stack. Works only for numbers.

Quotation

`[n, ]` Quotes a closure and pushes it onto the stack. Closures work by being preloaded with the first n things from the enclosing stack. So, for example, the function `'|x| (|y| x + y)'` can be written as `'x[1a]'`.

Combinatory logic

`K` Replace the stack top with a constant function returning the stack top. That is, $x \Rightarrow (\lambda y.x)$.

`I` Push the identity function onto the stack.

[17] See chapter 5 for more information about typeclasses.

[18] Previously this was handled by a Divergence module called Rebase. Now operator overloading functionality is built into d.

Invocation

`v` `stack[0].apply(null, stack[1])`. Pops both entries and pushes the result. The `v` stands for "variadic invocation".

`A` `stack[0].apply(stack[1], stack[2])`. Pops all three entries and pushes the result.

`B` Binds the stack top (as a function; if it is not a function, then it is run through `d` first) to the next entry, pops both, and pushes the result. For example, `'x72[0@]Bn5f'` returns 2. The inner function, `[0@]`, is bound by the `B` to the next entry down (the literal 2 pushed by `72`). Then we push `null`, swap the function and the value, and invoke the function. It returns `this`, which is 2.

Unbound

`k`

`j`

`h`

`m`

`o`

`q`

`s`

`w`

`x`

`y`

`C`

`D`

2.3 Regular Expressions

These are really simple; they just attempt to match against the input and return an array of matches if successful, `null` otherwise. (This is identical to the behavior of `exec()`.) There is a twist, though. The returned function is automatically homomorphic across arrays and object values, and if invoked on multiple arguments it concatenates their results. For example:

Listing 2.13 examples/regular-expressions-homomorphic.js

```
d(/foo/)('foo')                  // => ['foo']
d(/foo/)(['foo', 'bar'])         // => [['foo'], null]
d(/foo/)('foo', 'food')          // => ['foo', 'foo']

d(/f(o)o/)('foo')                // => ['foo', 'o']
d(/f(o)o/)(['foo', 'bar'])       // => [['foo', 'o'], null]
d(/fo(.)/)('foo', 'foad')        // => ['foo', 'o', 'foa', 'a']

d(/foo/)({bar: 'foo'})           // => {bar: ['foo']}
d(/foo/)({bar: 'bar'})           // => {bar: null}
```

2.4 Booleans

These are also very simple. `true` and `false` become functional decisionals. The exception is when one or more arguments are unspecified. In this case, they default to `true` and `undefined` respectively:

Listing 2.14 examples/booleans.js

```
d(true)(4, 5)             // => 4
d(false)(4, 5)            // => 5
d(true)()                 // => true
d(false)()                // => undefined
```

2.5 Arrays

Arrays, just like in the original Divergence library, are homomorphic across `d`.[19] For example:

Listing 2.15 examples/arrays.js

```
d([0x71, '.foo'])({foo: 'bar'})   // => [1, 'bar']
d([0xee, '.foo'])({foo: 'bar'})   // => [true, 'bar']
d([0xee, '.foo'])(null)           // => [false, null]
d([0xa, 0xb, 0xc])(2, 3)          // => [5, -1, 6]
d([])(1, 2, 3)                    // => []
```

2.6 Objects

Same thing here. These are homomorphic across `d` for values, so:

[19] The `d` in question, of course, is the same `d` that you called on the array. The same goes for objects (section 2.6). See chapter 6 for an explanation of why this is a relevant question.

Listing 2.16 examples/objects.js

```
d({foo: 0x71a})(5)                    // => {foo: 6}
d({two: 0x8ab, four: 0x74cb})(6, 7) // => {two: 5, four: 17}
d({foo: '||@'}).call(5)               // => 5
```

Chapter 3

Transforming Functions

As explained in chapter 2, you can use Divergence to build functions from values. However, that occupies only one parameter; `d` can accept more. Parameters after the first are used to modify the function being generated. So, for example, consider the following invocation:

```
d(0x748cc, 'amap', {suchThat: '|f| f([1])[0] === 16',
                      ensure: '|xs| xs.constructor === Array',
                     require: '|xs| xs.constructor === Array'});
```

`0x748cc` is the initial function (which multiplies its first argument by 16; see section 2.1.1 for details), `'amap'` is a nullary transformation that lifts the function to distribute componentwise across arrays, and the following object contains a list of transforms to be applied in arbitrary order. In this case, `suchThat` is an in-place unit test, `ensure` is a postcondition, and `require` is a precondition. The resulting function would look something like this:

```
function (xs) {
  if (! (xs.constructor === Array)) throw new Error (...);
  var result = [];
  for (var i = 0, l = xs.length; i < l; ++i)
    result.push (16 * xs[i]);
  if (! (result.constructor === Array)) throw new Error (...);
  return result;
}
```

In addition, this code would be run at function-definition time:

```
// Here, f is the above function
if (! (f([1])[0] === 16)) throw new Error (...);
```

Obviously a lot has happened to the original function and the intermediate structures. Divergence keeps functions in intermediate format until all of the transformations have been run, and compiles the function into native JavaScript just before returning it. This transformation step isn't primitive; it's a customization that is built into the default copy of Divergence and inherited.

3.1 Arrays

Array transformation arguments are treated as argument list manipulators (`flat_compose` in the original Divergence). They are homomorphic across `d` and let you do things like changing argument order, deleting arguments, and transforming them. For example:

Listing 3.1 `examples/array-transforms.js`

```
d(0xb)(5, 4)                              // => 1
d(0xb, [1, 0])(5, 4)                      // => -1
d(0xb, [0x8a, 1])(5, 4)                   // => 6
d(0xd, [1, 2])(0, true, 'constructor')    // => function Boolean () {...}
```

3.2 Numbers

Numbers are run through `d` and composed normally. It's often useful to use negative integers for quick argument swapping. For example:

Listing 3.2 `examples/number-transforms.js`

```
d(0xa)(1, 3)              // => 4
d(0xe, 0xa)(1, 3)         // => -4
d(0, 0xb)(4, 2)           // => 2
d(0xb, -0x10)(4, 2)       // => -2
```

3.3 Regular expressions

These are composed normally, and are most useful when you expect a string as input. For example:

Listing 3.3 `examples/regexp-transforms.js`

```
d(0x6a, /(.)foo(.)/)('afoob')             // => 'ab'
d(0xd, /foobar(length)/)('foobarlength')  // => 12
d(0xee, /foobar/)('foo')                  // => null
```

If the regular expression fails to match, the function is never executed and `null` will be returned. Normal regular-expression rules apply for array or multiple arguments.

3.4 Generalized transforms

Strings and objects are used to specify generalized transforms. These are arbitrary functions that can transform the function you're building. In the example above, `suchThat`, `require`, `ensure`, and `amap` are all such transformations. For example, these are all equivalent (modulo order in the second case):

```
d(0x71a, 'amap', 'vlift')
d(0x71a, {amap: null, vlift: null})
d(0x71a, 'amap', {vlift: null})
d(0x71a, {amap: null}, 'vlift')
```

Divergence comes with several generalized transforms, and you can write your own. The built-in transforms are described in chapter 4.

3.5 Strings

If a string doesn't match one of the defined generalized transforms, then it is interpreted as a string function (see section 2.2) and is expected to return an array to be used as the function's argument list. For example:

Listing 3.4 `examples/string-transforms.js`

```
d(0xa, '|x, y| [x, 2*y]')(3, 4)          // => 11
d(0, '|x, y| [y]')(3, 4)                 // => 4
d(0, '|x, y| [x]')(3, 4)                 // => 3
d(1, '|x, y| [x, y]')(3, 4)              // => 4
```

3.6 Functions

A function is used on the argument list and is expected to return an array of new arguments. For example:

Listing 3.5 `examples/function-transforms.js`

```
d(0xa, d(-0xc))(4, 5, 6)                 // => 26
d(0xb, d('|x| [x, x]'))(5)               // => 0
d(0xc, d('|x| [x, x]'))(5)               // => 25
```

Chapter 4

Built-in Transforms

Divergence comes with a handful of transforms that are useful for working with JavaScript functions and data types. Some of them are standalone (e.g. `require`, `ensure`), while others (e.g. `amap`, `xlift`) are generated from a data structure. Divergence provides a framework for defining Haskell-style typeclasses and operations on their instances.

4.1 Defining a Typeclass

Here is the definition for the `foldable` typeclass:

Listing 4.1 `examples/foldable.js`

```
d.typeclass('foldable', 'fold', 'zero', 'merge', 'ret').define ({
  map:  '|f| @fold((@|x, y| @merge(x, @ret(f(y)))), @zero())',
  lift: '|f| @fold((@|x, y| @merge(x, f(y))), @zero())'});
```

Here's how this works. First, we tell Divergence that we're defining a `foldable` typeclass, and instances are assumed to provide implementations of `fold`, `zero`, `merge`, and `ret`. Then we proceed to define methods that we get for free, in this case `map` and `lift`.

For arrays, `map` can be defined in terms of `fold` like this:

```
var map = function (xs, f) {
  return fold ([], xs, function (ys, x) {
    return ys.concat ([f(x)]);
  });
};
```

However, this definition works only for arrays. To generalize it, we need to remove all array-specific references. Here's what the code looks like in general form, where the extra parameter `t` provides collection-specific methods:

```
var map = function (t, xs, f) {
  return t.fold (t.zero(), xs, function (ys, x) {
    return t.merge (ys, t.ret (f(x)));
  });
};
```

Now there is only one step left. We need to take a regular function and return a mapping function, but we don't have a collection yet. That is, instead of having a function `map` that does this:

```
var ys = map(xs, f);
```

we want `map` to do this:

```
var ys = map(f)(xs);
```

If `fold` is already a function-transform, then we're all set. Here's how that works:

```
var map = function (t, f) {
  return t.fold (t.zero(), function (ys, x) {
    return t.merge (ys, t.ret (f(x)));
  });
};
```

All we've done here is curried out the `xs` parameter at every level. Also notice that it isn't hard to get from the above definition of `map` to what we have implemented in the `foldable` typeclass.

4.1.1 Making `Array` foldable

The definition to make `Array` an instance of `foldable` isn't hard at all:

Listing 4.2 `examples/array-foldable.js`

```
d.instance(Array, 'foldable', {
  zero:  '||[]',
  merge: '|x, y| x.concat(y)',
  ret:   -0x77,
  fold:  function (f) {
    var args = arguments;
    return function (xs) {
      var start   = args.length === 2;
      var initial = start ? args[1] : xs[0];
      for (var i = start, l = xs.length; i < l; ++i)
        initial = f(initial, xs[i]);
      return initial;
    };
  }});
```

The following functions are provided for each instance of `foldable` (examples here are for arrays):

4.1.2 amap

Causes the function to be invoked componentwise across an array. This transform takes no parameters. For example:

Listing 4.3 examples/amap.js

```
d(0x71a, 'amap')([1, 2, 3, 4])          // => [2, 3, 4, 5]
d(0x8a, 'amap')([1, 2, 'foo'])          // => [2, 4, 'foofoo']
d(0xee, 'amap')(['foo', null])          // => [true, false]
d(0xee, 'amap')([])                     // => []
d(0x8c, 'amap')([1, 2, 3])              // => [1, 4, 9]
d(-0x8c, 'amap')([1, 2, 3])             // => [[1], [4], [9]]
```

4.1.3 alift

Lifts a function into the array monad. More specifically, causes the function to be invoked componentwise across the array entries and the results, which should be arrays, are then concatenated. `false`, `undefined`, `null`, and other falsy values are considered to be equivalent to empty arrays. For example:

Listing 4.4 examples/alift.js

```
d('|x| [x + 1]', 'alift')([1, 2, 3])          // => [2, 3, 4]
d('|x| [x, x + 1]', 'alift')([1, 2, 3])       // => [1, 2, 2, 3, 3, 4]
d('|x| x&1 && [x]', 'alift')([1, 2, 3, 4])    // => [1, 3]
d([0], 'alift')([1, 2, 3])                    // => [1, 2, 3]
d([0x71a, 0x8c], 'alift')([1, 2, 3])          // => [2, 1, 3, 4, 4, 9]
d(-0x8c, 'alift')([1, 2, 3])                  // => [1, 4, 9]
d([-0x8c], 'alift')([1, 2, 3])                // => [[1], [4], [9]]
```

4.1.4 afold

Lifts a function into a left-fold over arrays. That is, calling your function on an array is equivalent to calling a traditional `foldLeft` implementation on both your function and the array. For example:

Listing 4.5 examples/afold.js

```
d(0xa, 'afold')([1, 2, 3])              // => 6
d(0xc, 'afold')([2, 3, 4])              // => 24
d(0xa, 'afold')([1])                    // => 1
d(0x18c0a, 'afold')([[], 1, 2, 3])      // => [1, 4, 9]
```

You can also specify the first fold argument as a transform parameter:

Listing 4.6 examples/afold.js (continued)

```
d(0xa, {afold: 4})([1, 2, 3])           // => 10
d(0x18c0a, {afold: []})([1, 2, 3])      // => [1, 4, 9]
```

Chapter 5

Typeclasses

Adding functionality to default constructs such as arrays and objects is difficult, especially in a dynamically-typed language with poor global containment. Divergence doesn't have a great solution to this, but it avoids global namespace pollution. The idea is that you define typeclasses in Haskell style, and then promote types into those typeclasses later on. When you want to use functionality provided by a typeclass, you do so by lifting a function into an operator over instances of that typeclass. In other words, rather than focusing on the values you instead focus on the functions that manipulate those values.

Here's a simple example of the difference:

Listing 5.1 examples/array-map-traditional.js

```
var mapped = my_array.map (d('|x| x + 1'));
```

The reason this example works is that JavaScript does dynamic dispatch on the `map` method based on the prototype of `my_array`. However, since `Array` is global, we can't modify it.

Listing 5.2 examples/array-map-typeclass.js

```
var mappable = d.typeclass(...);
var mapped   = d('|x| x + 1', mappable.map)(my_array);
```

Here, Divergence has to decide how to lift the function. It's a runtime-dispatch decision just as it was before, but this time the type-selection logic happens in library-space. This lets Divergence compile a custom version of the function for that data structure (if the `specialize_typeclass_functions` and `inline_typeclass_members` compiler options are set). Note that the type selection logic isn't blazingly fast; constructor functions are first hashed by their `toString()` output, and then a referential comparison is done on the identity. So, for example, if we want to lift a function using `map`, and the `mappable` typeclass contains `Array` and `Object`, then this is the process when that function is invoked:

1. Find the constructor of the first argument.

2. Invoke `toString()` on the constructor to obtain its source code.
3. Look up that string in the `mappable.instances` hashtable. This returns an array of pairs.
4. Find the constructor in that array and retrieve the second element of the pair (the instance method table).
5. Invoke the element's `map` method on the original function, and invoke that result on the first argument.
6. Repeat this process for each additional argument.

If there are few instances or each instance has a short and unique constructor, then this won't be a slow operation. The problem arises when there are a bunch of instances with identical constructor code, or code that is very long. In that case, using typeclasses will be computationally expensive.[1]

5.1 Defining a typeclass

Divergence doesn't keep a registry of typeclasses. A typeclass is just another value, and you can modify it using standard method calls. For example, here is a typeclass definition:

Listing 5.3 `examples/defining-the-mappable-typeclass.js`

```
var mappable = d.typeclass({
```

[1]And unfortunately there is no way to memoize the process, since JavaScript references are opaque.

Chapter 6

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 6.1 `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();
});
```

6.1 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 6.2 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 6.3 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.

6.2 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 6.4 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 6.5 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