Monads & Gonads

Douglas Crockford

Functional Programming

Programming with functions.

FORTRAN II (1958)

```
FUNCTION name(parameters)
DIMENSION ...
COMMON ...

name = expression
RETURN
END
```

First Class Functions

Higher Order Functions

Lexical Closure

Pure Functional Programming

More mathematical.

Functions as maps

Programming without side-effects.

Remove assignment, loops (use recursion instead), freeze all array literals and object literals.

Remove `Date` and `Math.random`.

Memoization

Caching

In the real world,
everything changes.

Immutability makes it hard to
interact with the world.

Monads

A loophole in the
function contract.

In order to understand monads, you need to first learn Haskell and Category Theory.

In order to understand monads, you need to first learn Haskell and Category Theory.

In order to understand burritos, you must first learn Spanish.

$$(M\ t) \to (t \to M\ u) \to (M\ u)$$

Not tonight, Josephine.

…you must first learn JavaScript

```
function unit(value)

function bind(monad, function (value))
```

All three functions return a monad.

Axioms

```
bind(unit(value), f) ==== f(value)

bind(monad, unit) ==== monad

bind(bind(monad, f), g)
                ====
bind(monad, function (value) {
    return bind(f(value), g);
})
```

```
bind(monad, func)
```

```
monad.bind(func)
```

The OO transform.

```
function MONAD() {
    return function unit(value) {
        var monad = Object.create(null);
        monad.bind = function (func) {
            return func(value);
        };
        return monad;
    };
}
```

Context Coloring

```
function MONAD() {
    return function unit(value) {
        var monad = Object.create(null);
        monad.bind = function (func) {
            return func(value);
        };
        return monad;
    };
}
```

Macroid

```
function MONAD() {
    return function unit(value) {
        var monad = Object.create(null);
        monad.bind = function (func) {
            return func(value);
        };
        return monad;
    };
}

var identity = MONAD();
var monad = identity("Hello world.");
monad.bind(alert);
```

Axioms

```
unit(value).bind(f) ==== f(value)

monad.bind(unit) ==== monad

monad.bind(f).bind(g)
                ====
monad.bind(function (value) {
    return f(value).bind(g);
})
```

Axioms

```
bind(bind(monad, f), g)

monad.bind(f).bind(g)
```

The Ajax Monad

```
monad.bind(f).bind(g)
```

Interstate (2001)

```
new Interform('text')
      .moveTo(100, 100)
      .setSize(400, 32)
      .moveInside()
      .setBgColor('pink')
      .select()
      .setZIndex(20000)
      .on('escapekey', 'erase');
```

ADsafe (2007)

```
var input = dom.q("input_text")
      .on('enterkey', function (e) {
          dom.q('#ROMAN_RESULT')
              .value(roman(input
                  .getValue()));
          input
              .select();
      })
      .focus();
```

monad.**bind**(func)

monad.method()

```
monad.bind(func)

    monad.bind(func, [
        a, b, c
    ])

monad.method()

    monad.method(a, b, c)
```

```
function MONAD() {
    var prototype = Object.create(null);
    function unit(value) {
        var monad = Object.create(prototype);
        monad.bind = function (func) {
            return func(value);
        };


        return monad;
    }




    return unit;
}
```

```
function MONAD() {
    var prototype = Object.create(null);
    function unit(value) {
        var monad = Object.create(prototype);
        monad.bind = function (func, args) {
            return func.apply(undefined,
                [value].concat(Array.prototype
                .slice.apply(args || [])));
        };
        return monad;
    }




    return unit;
}
```

```
function MONAD() {
    var prototype = Object.create(null);
    function unit(value) {
        var monad = Object.create(prototype);
        monad.bind = function (func, args) {
            return func(value, ...args));
        };
        return monad;
    }

    return unit;
}
```

```
function MONAD() {
    var prototype = Object.create(null);
    function unit(value) {
        var monad = Object.create(prototype);
        monad.bind = function (func, args) {
            return func(value, ...args));
        };
        return monad;
    }

    unit.method = function (name, func) {
        prototype[name] = func;
        return unit;
    };
    return unit;
}
```

```
function MONAD() {
    var prototype = Object.create(null);
    function unit(value) {
        var monad = Object.create(prototype);
        monad.bind = function (func, args) {
            return func(value, ...args));
        };
        return monad;
    }

    unit.lift = function (name, func) {
        prototype[name] = function (...args) {
            return unit(this.bind(func, args));
        };
        return unit;
    };
    return unit;
}
```

```
var ajax = MONAD()
    .lift('alert', alert);

var monad = ajax("Hello world.");

monad.bind(alert);

monad.alert();
```

`null`

Null Pointer Exception

Maybe

`NaN`

```
function MONAD(modifier) {
    var prototype = Object.create(null);
    function unit(value) {
        var monad = Object.create(prototype);
        monad.bind = function (func, args) {
            return func(value, ...args));
        };
        if (typeof modifier === 'function') {
            modifier(monad, value);
        }
        return monad;
    }
    return unit;
}
```

```
var maybe = MONAD(function (monad, value) {
    if (value === null || value === undefined)
{
        monad.is_null = true;
        monad.bind = function () {
            return monad;
        };
    }
});

var monad = maybe(null);
monad.bind(alert);
```

Concurrency

Threads are evil.

Turn Based Processing

- Single-threaded. Race free. Deadlock free.
- The Law of Turns:
  Never wait. Never block. Finish fast.
- Events. Message passing. ~~Threads. Mutexs.~~
- Web browsers.
- Most UI frameworks.
- Servers: Elko, Twisted, Nodejs.
- Asynchronicity can be hard to manage.

Promises

- Promises are an excellent mechanism for managing asynchronicity.
- A promise is an object that represents a possible future value.
- Every promise has a corresponding resolver that is used to ultimately assign a value to the promise.
- A promise can have one of three states: `'kept'`, `'broken'`, or `'pending'`.

Promises

- A promise is an event generator. It fires its event when the value of the promise is ultimately known.
- At any time after the making the promise, event handling functions can be registered with the promise, which will be called in order with the promise's value when it is known.

Promises

- A promise can accept functions that will be called with the value once the promise has been kept or broken.
- promise`.when(`success`,` failure`)` returns another promise for the result of your success function.

Make a vow

```
var my_vow = VOW.make();
    .keep(value)
    .break(reason)
    .promise
        .when(kept, broken)
```

Filesystem API

```
read_file(name)
     .when(function success(string)
{
                    ...
     }, failure);
```

Exceptions

- Exceptions modify the flow of control by unwinding the state.
- In a turn based system, the stack is empty at the end of every turn.
- Exceptions cannot be delivered across turns.
- Broken promises can.

Breakage flows to the end

```
my_promise
      .when(success_a)
      .when(success_b)
      .when(success_c, failure);
```

success_a gets the value of **my_promise**
success_b gets the value of success_a
success_c gets the value of success_b
unless any promise breaks:
    failure gets the reason

Composition

```
f()
    .when(function (f_value) {
        return g(f_value);
    })
    .when(function (g_value) {
        ...
    })
```

====

```
f()
    .when(function (f_value) {
        return g(f_value)
            .when(function (g_value) {
                ...
            });
    })
```

A promise is a monad

- The value is not known when the monad is made.
- Each promise is linked to two resolver functions, **`keep`** and **`break`**, that determine the promise's success and value.
- **`when`** can take two functions, **`bind`** only one.

```
var VOW = (function () {

    // function enqueue here...
    // function enlighten...

    return {
        make: function make() {
            ...
        }
    };
})();
```

```
var VOW = (function () {

    // function enqueue here...
    // function enlighten...

    return {
        make: function make() {
            ...
        }
    };
})();
```

Dog balls

```
var VOW = (function () {

    // function enqueue here...
    // function enlighten...

    return {
        make: function make() {
            ...
        }
    };
}());
```

```
make: function make() {
    var breakers = [], fate,
        keepers = [],  status = 'pending';

    // function herald here...

    return {
        break: function (reason) {
            herald('broken', reason, breakers);
        },
        keep: function (value) {
            herald('kept', value, keepers);
        },
        promise: {
            ...
        }
    };
}
```

```
function herald(state, value, queue) {
    if (status !== 'pending') {
        throw "overpromise";
    }
    fate = value;
    status = state;
    enlighten(queue, fate);
    keepers.length = 0;
    breakers.length = 0;
}
```

```
make: function make() {
    var breakers = [], fate,
        keepers = [], status = 'pending';

    // function herald here...

    return {
        break: function (value) {
            herald('broken', value, breakers);
        },
        keep: function keep(value) {
            herald('kept', value, keepers);
        },
        promise: {
            ...
        }
    };
}
```

```
promise: {
    is_promise: true,
    when: function (kept, broken) {
        var vow = make();
        switch (status) {
        case 'pending':
            enqueue(keepers, kept, vow.keep, vow.break);
            enqueue(breakers, broken, vow.break, vow.break);
            break;
        case 'kept':
            enqueue(keepers, kept, vow.keep, vow.break);
            enlighten(keepers, fate);
            break;
        case 'broken':
            enqueue(breakers, broken, vow.break, vow.break);
            enlighten(breakers, fate);
            break;
        }
        return vow.promise;
    }
}
```

```
function enqueue(queue, func, resolver, breaker) {
    queue[queue.length] = typeof func !== 'function'
        ? resolver
        : function (value) {
            try {
                var result = func(value);
                if (result && result
                        .is_promise === true) {
                    result.when(resolver, breaker);
                } else {
                    resolver(result);
                }
            } catch (e) { breaker(e); }
        };
}
```

```
function enlighten(queue, fate) {
    queue.forEach(function (func) {
        setImmediate(func, fate);
    });
}
```

https://github.com/douglascrockford/monad

```
var VOW = (function () {

    function enqueue(queue, func, resolver, breaker) {
        queue[queue.length] = typeof func !== 'function'
            ? resolver
            : function (value) {
                try {
                    var result = func(value);
                    if (result && result.is_promise === true) {
                        result.when(resolver, breaker);
                    } else {
                        resolver(result);
                    }
                } catch (e) { breaker(e); }
            };
    }

    function enlighten(queue, fate) {
        queue.forEach(function (func) {
            setImmediate(func, fate);
        });
    }

    return {
        make: function make() {
        var breakers = [], fate,  keepers = [],  status = 'pending';

        function herald(state, value, queue) {
            if (status !== 'pending') {
                throw "overpromise";
            }
            fate = value;
            status = state;
            enlighten(queue, fate);
            keepers.length = 0;
            breakers.length = 0;
        }

        return {
            break: function (reason) {
                herald('broken', reason, breakers);
            },
            keep: function (value) {
                herald('kept', value, keepers);
            },
            promise: {
                is_promise: true,
                when: function (kept, broken) {
                    var vow = make();
                    switch (status) {
                    case 'pending':
                        enqueue(keepers, kept, vow.keep, vow.break);
                        enqueue(breakers, broken, vow.break, vow.break);
                        break;
                    case 'kept':
                        enqueue(keepers, kept, vow.keep, vow.break);
                        enlighten(keepers, fate);
                        break;
                    case 'broken':
                        enqueue(breakers, broken, vow.break, vow.break);
                        enlighten(breakers, fate);
                        break;
                    }
                    return vow.promise;
                }
            }
        };
    };
}());
```

Our Friend the Monad

The Identity Monad
The Ajax Monad
The Maybe Monad
The Promise Monad

Further Viewing

Carl Hewitt. The Actor Model (everything you wanted to know, but were afraid to ask)

http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

Mark Miller. Secure Distributed Programming with Object-capabilities in JavaScript

http://www.youtube.com/watch?v=w9hHHvhZ_HY
http://www.youtube.com/watch?v=oBqeDYETXME