Functional JavaScript
Why or Why Not?

JSDC 2014

I'm Greg Weng
from Mozilla Taiwan

Gaia Developer

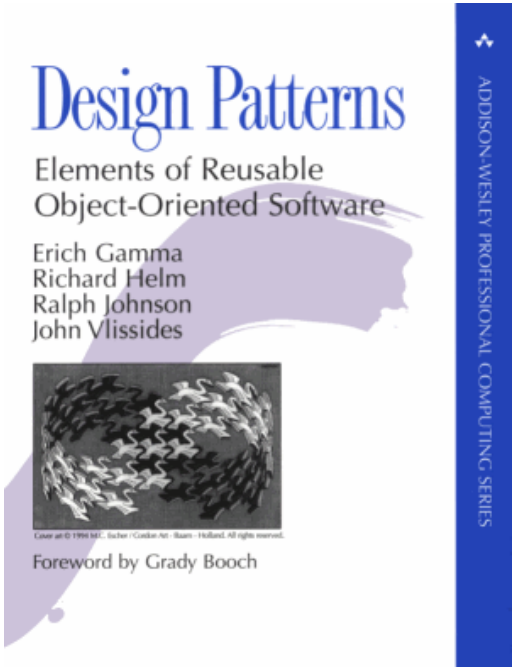
Co-organizer of Functional Thursday

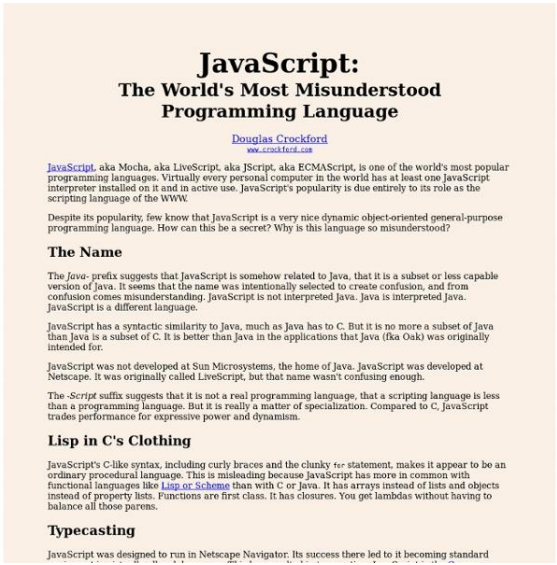
JavaScript Enthusiast

@GregWeng

about.me/snowmantw

@GregWeng
about.me/snowmantw





JavaScript:
The World's Most Misunderstood Programming Language

Douglas Crockford
www.crockford.com

JavaScript, aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use. JavaScript's popularity is due entirely to its role as the scripting language of the WWW.

Despite its popularity, few know that JavaScript is a very nice dynamic object-oriented general-purpose programming language. How can this be a secret? Why is this language so misunderstood?

The Name

The *Java-* prefix suggests that JavaScript is somehow related to Java, that it is a subset or less capable version of Java. It seems that the name was intentionally selected to create confusion, and from confusion comes misunderstanding. JavaScript is not interpreted Java. Java is interpreted Java. JavaScript is a different language.

JavaScript has a syntactic similarity to Java, much as Java has to C. But it is no more a subset of Java than Java is a subset of C. It is better than Java in the applications that Java (fka Oak) was originally intended for.

JavaScript was not developed at Sun Microsystems, the home of Java. JavaScript was developed at Netscape. It was originally called LiveScript, but that name wasn't confusing enough.

The *-Script* suffix suggests that it is not a real programming language, that a scripting language is less than a programming language. But it is really a matter of specialization. Compared to C, JavaScript trades performance for expressive power and dynamism.

Lisp in C's Clothing

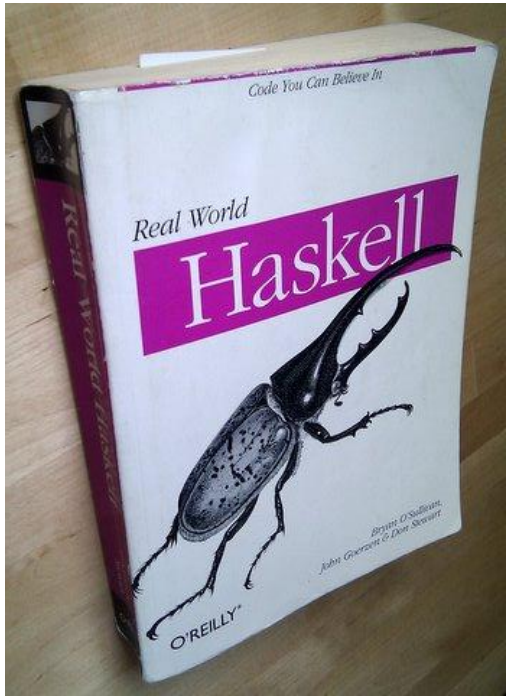
JavaScript's C-like syntax, including curly braces and the clunky `for` statement, makes it appear to be an ordinary procedural language. This is misleading because JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java. It has arrays instead of lists and objects instead of property lists. Functions are first class. It has closures. You get lambdas without having to balance all those parens.

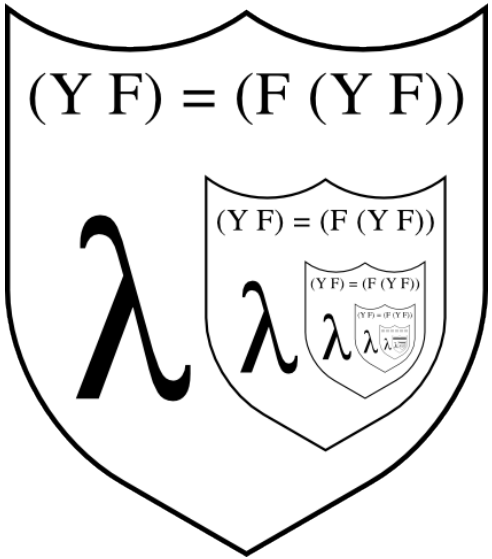
Typecasting

JavaScript was designed to run in Netscape Navigator. Its success there led to it becoming standard

@GregWeng

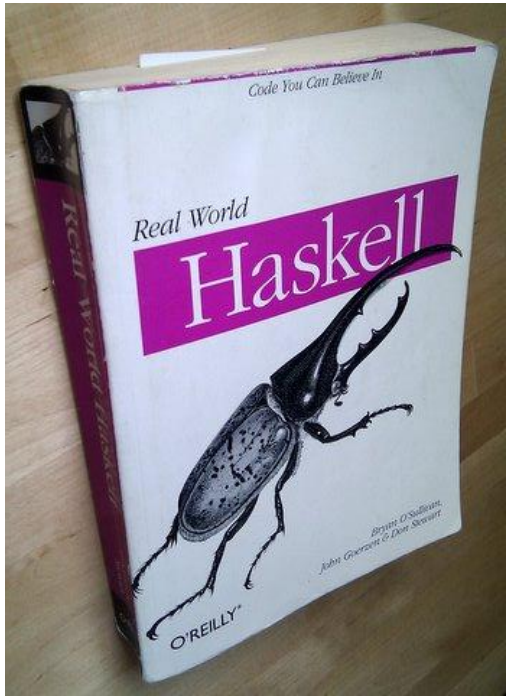
about.me/snowmantw

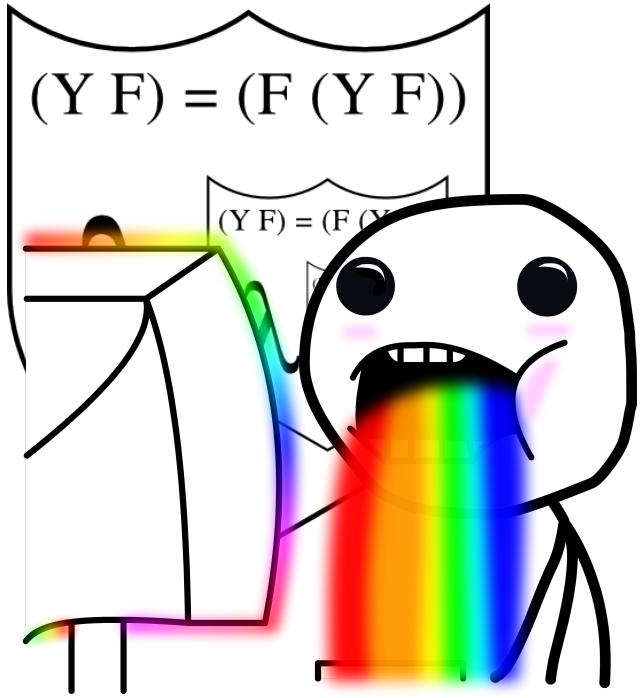




@GregWeng

about.me/snowmantw





@GregWeng
about.me/snowmantw

Outline

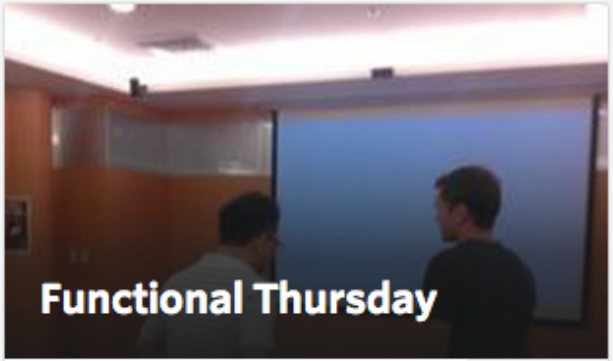
This talk is about...

- Why Functional Programming is useful
- What features we can use immediately
- How many libraries are ready now
- Trade-off: if feature X doesn't exist, is it worth to implement it?

Motivation


FIREFOX
FIREFOX OS
社群參與
訊息中心
關於我們
mozilla
Firefox
訊息中心 » 謀智台客 » 從 JAVASCRIPT 的 MAP/REDUCE 談起 FUNCTIONAL PROGRAMMING
4月
07
2014
snowmantw
從 JavaScript 的 Map/Reduce 談起 Functional Programming
自 ECMAScript 5.1 開始 JavaScript 加入了兩個關於陣列的函式：Array.prototype.map/reduce。這兩個函式可以針對陣列，讓開發者更清楚的描述接下來程式碼所要表達的運算性質。除此之外，也簡化了每次手寫迴圈進行尋訪陣列的繁瑣過程，特別是當尋訪實際上是要將原本的陣列映射(map)成另一個陣列，或是進行加總、檢驗等具有化簡(reduce)性質的操作。
function useLoop(arr) {
function doMap() {
var isOdd = [];
for (var i = 0; i != arr.length; i++)
isOdd[i] = (0 !== arr[i] % 2);
}
return isOdd;
}
function doReduce() {
var acc = 0;
for (var i = 0; i != arr.length; i++)
acc += arr[i];
function useMapReduce(arr) {
function doMap() {
var isOdd = arr.map(function(e, i, a)
return (0 !== e % 2);
});
return isOdd;
}
function doReduce() {
var acc = arr.reduce(function(p, c){
return p + c;
}, 0);
return acc;
關於作者
snowmantw
網站：http://snowmantw.tumblr.com/
台客編輯群
認識台客編輯群 »
文章搜尋

Motivation



77 Functors
Next Meetup: Oct 2

ORGANIZER

45 Coders

CO-ORGANIZER

294 Hackers

MEMBER

Premise

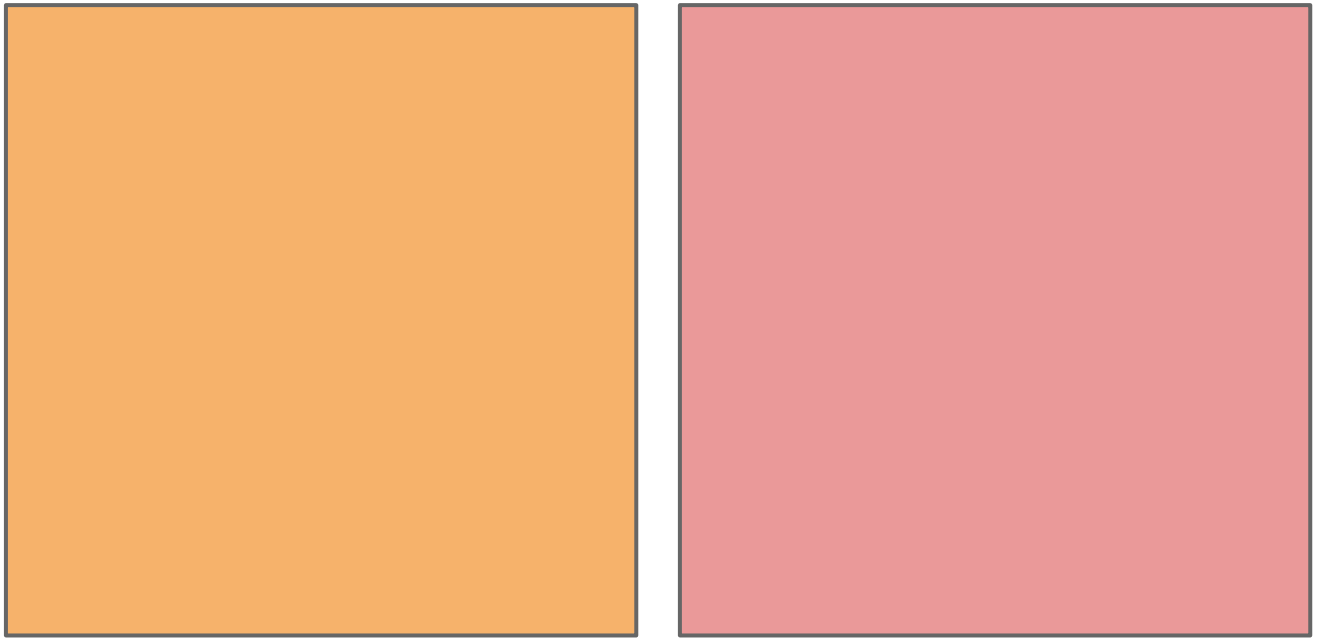
No Compiler

(No magic!)

Why Functional Programming is useful

Why Functional Programming is useful

...not really



Which one is better?

Which one is ~~better~~?

much useful?

Programming Functionally brings you

- A way to re-think about :programming:
- Other efficient patterns to complete your work
- Fun. The more you dig the more fun you'll get

Basic concepts of Functional Programming

First-class function | High-order functions | Function composition | Closure

Purity | Managed side-effects | Laziness

Recursion | Tail-recursion optimization | (Type)

Basic concepts of Functional Programming

JavaScript Ready

First-class function | High-order functions | Function composition | Closure

Purity | Managed side-effects | Laziness

Need some hard works

Recursion | Tail-recursion optimization | (Type)

Impossible if runtime doesn't support it (well)

Discuss it later...

function() {}

is everything

~~function() {}~~

is everything

or () => { } if you're a lucky bastard

Use Firefox to embrace () => 'the power of ES6!'
(Arrow Functions)

Part I

Computation = Transformation

Computation = Transformation

```
65535                              -- Number (*yes, it's a function)

65535 + 1                          -- Number → Number

[1]                                -- Number → Array Number

[1, 2, 3]                          -- Array Number → Array Number

[1, 2, 3].length                   -- Array Number → Number

[1, 2, 3].map((x) => `${ x }`)     -- Array Number →
                                      (Number → String) →
                                      Array String
```

Computation = Transformation

```
65535                                  -- Number (*yes, it's a function)

65535 + 1                              -- Number → Number

[1]                                    -- Number → Array Number

[1, 2, 3]                              -- Array Number → Array Number

[1, 2, 3].length                       -- Array Number → Number

[1, 2, 3].map((x) => `${ x }`)         -- Array Number →
                                       (Number → String) →
                                       Array String
```

Use FirefoxNightly to embrace `the ${power} of ES6!`
(Quasi-Literals)

About the signature

Array Number → (Number → String) → Array String

argument | function as argument | "return value"

[a] → (a → b) → [b]

a, b: type variables

Computation =

Transformation +

Composition

High-order Function

Composition: High-order function

High-order Function: receive functions as arguments

```
map:: [a] → (a → b) → [b]

reduce:: [a] → (a → b → b) → [a] → b

-- Note: these are *NOT* correct signatures in Haskell

-- but in JavaScript, we can treat [1,2,3].map as map::[a]...

-- which makes the code matches the type better
```

Composition: High-order function

High-order Function: receive functions as arguments

```
map:: [a] → (a → b) → [b]
```

```
reduce:: [a] → (a → b → b) → [a] → b
```

Not only useful for calculations

Composition: High-order function

Use map & reduce in different cases

Replace lots of old tricks of the plain loop

```
var result = {};
for (var i = 0; i < selectors.length; i++) {
  var selector = selectors[i];
  result.push(document.querySelector(selector));
}
```

```
var result =
selectors.map((selector) => document.querySelector(selector));
```

Use list methods usually make code clearer

Composition: High-order function

shared/js/nfc_utils.js — View full changes

```
((91 lines not shown))
+
+      // Now decode all other records and attach their data to poster.
+      for (var r = 0; r < records.length; r += 1) {
+        var record = records[r];
+        var typeStr = NfcUtils.toUTF8(record.type);
+
+        if (NfcUtils.equalArrays(record.type, NDEF.RTD_TEXT)) {
+          poster.text = poster.text || {};
+
+          var textData = NDEF.payload.decodeText(record.payload);
+
+          if (poster.text[textData.language]) {
+            // According to NFCForum-SmartPoster_RTD_1.0 3.3.2,
+            // there MUST NOT be two or more records with
+            // the same language identifier.
+            return null;
```

more...

[Records] → Poster

reducing ~= building

A real case in Gaia project (Bug 1039245)

Composition: High-order function

```
238 +      // Now decode all other records and attach their data to poster.
239 +      return records.reduce((poster, record) => {
240 +        var typeStr = NfcUtils.toUTF8(record.type);
241 +
242 +        if (NfcUtils.equalArrays(record.type, NDEF.RTD_TEXT)) {
243 +          poster.text = poster.text || {};
244 +
245 +          var textData = NDEF.payload.decodeText(record.payload);
246 +
247 +          if (poster.text[textData.language]) {
248 +            // According to NFCForum-SmartPoster_RTD_1.0 3.3.2,
249 +            // there MUST NOT be two or more records with
250 +            // the same language identifier.
251 +            return null;
252 +          }
253 +
254 +          poster.text[textData.language] = textData.text;
255 +        } else if ('act' === typeStr) {
256 +          poster.action = record.payload[0];
257 +        } else if (NDEF.TNF_MIME_MEDIA === record.tnf) {
258 +          poster.icons = poster.icons || [];
259 +          poster.icons.push({
260 +            type: NfcUtils.toUTF8(record.type),
261 +            bytes: record.payload
262 +          });
263 +        }
264 +        return poster;
265 +      }, uriPoster);
266 +    },
```

[Records] → Poster

reducing ~= building

It's not about SLOC; it's about semantics.

Composition: High-order function

Use map & reduce in different cases

Thinking in type brings us more possibilities

Composition: High-order function

Use map & reduce in different cases

Thinking in type brings us more possibilities

```
map:: [a] → (a → b) → [b], while a/URL, b/IO
```

Composition: High-order function

Use map & reduce in different cases

Thinking in type brings us more possibilities

`map:: [a] → (a → b) → [b], while a/URL, b/IO`

```
urls.map((url) => Http.get(url))                                    // map URL -> IO to [ URL ]
    .filter((response) => response.status !== 404 )
    .map((response) => Parser.comment(response))
    .map((comment) => UI.renderComment(comment))
    .execute()    // If we have lazy IO & async mixed Monad Transformer...will discuss it later
```

Composition: High-order function

Use map & reduce in different cases

Thinking in type brings us more possibilities

`map:: [a] → (a → b) → [b], while a/URL, b/IO`

```
urls.map((url) => Http.get(url))                                    // map URL -> IO to [ URL ]
    .filter((response) => response.status !== 404 )
    .map((response) => Parser.comment(response))
    .map((comment) => UI.renderComment(comment))
    .execute()    // If we have lazy IO & async mixed Monad Transformer...will discuss it later
```

In fact you can't do that because of async & eager evaluation

Composition: High-order function

Some advanced high-order functions of list

```
forEach:: [a] → (a → SideEffect; will discuss it later)

filter:: [a] → (a → Bool) → [a] * the type is similar with map

groupBy:: [a] → (a → a → Bool) → [[a]] * lo-dash has it

zipWith: [a] →[b] → (a → b → c) → [c] * worth to implement
```

Recommend use lo-dash library to obtain more functions of list

Composition: High-order function

High-order functions are not only useful for list

Although the list-transformation model is powerful

[URL] → [IO], [Datum] → DOM, [Event] → [Transition]

IMO it's definitely worth to use these functions.

Function Composition

Composition: Function composition

Function composition: compose tiny functions into larger one

```
compose:: (b → c) → (a → b) → a → c
```

```
(negate . sum . tail) [1,2,3]  -- Haskell
```

Composition: Function composition

Function composition: compose tiny functions into larger one

```
compose:: (b → c) → (a → b) → a → c
```

** In fact, using an operator is better than function call, since:*

```
(negate . sum . tail) [1,2,3]
```

Composition: Function composition

Function composition: compose tiny functions into larger one

```
compose:: (b → c) → (a → b) → a → c
```

** In fact, using an operator is better than function call, since:*

```
(negate . sum . tail) [1,2,3]
```

is MUCH better than:

```
compose(compose(negate, sum), tail) [1,2,3]
```

Composition: Function composition

Function composition: compose tiny functions into larger one

```
compose:: (b → c) → (a → b) → a → c
```

** In fact, using an operator is better than function call, since:*

```
(negate . sum . tail) [1,2,3]
```

So this is a feature IMO should not use massively in JS, unless we can have some better interfaces.

is MUCH better than:

Lo-dash has one _.compose, can try with that.

```
compose(compose(negate, sum), tail) [1,2,3]
```

Composition: Function composition

Function composition: compose tiny functions into larger one

Try to make the interface better...

```
functA . functB . functC ...     // our target

c(functA, functB, functC)        // *not* success...

funcA.c(functB).c(functC)       // need to hack Function.prototype
```

Composition: Function composition

Function composition: compose tiny functions into larger one

You can live without that, even in Functional language.

But when you have nice syntax and type system,

it would become more powerful.

Unfortunately in JS we don't have that

Partial Application

Composition: Partial application

Partial application: generate a new function with something bounded

Composition: Partial application

Partial application: generate a new function with something bounded

Examples:

```
map::                    [a] → (a → b) → [b]
```

Composition: Partial application

Partial application: generate a new function with something bounded

Examples:

```
map::                    [a] → (a → b) → [b]

map [1, 2, 3]::          (a → b) → [b]
```

Composition: Partial application

Partial application: generate a new function with something bounded

Examples:

```
map::                          [a] → (a → b) → [b]

map [1, 2, 3]::                (a → b) → [b]

map [1, 2, 3] toChar::         [b]
```

Composition: Partial application

Partial application: generate a new function with something bounded

Examples:

```
map::                          [a] → (a → b) → [b]

map [1, 2, 3]::                (a → b) → [b]

map [1, 2, 3] toChar::         [b]
```

Of course this not works in ordinary JavaScript functions

Why we need this?

Why we need Partial application

This feature makes program reusable & flexible

```
map [1,2,3] (add 2)
```

Why we need Partial application

This feature makes program reusable & flexible

```
map [1,2,3] (add 2)
```

is much better than:

Why we need Partial application

This feature makes program reusable & flexible

```
map [1,2,3] (add 2)
```

is much better than:

```
map [1,2,3] ((y)=> add 2+y)   -- need new anonymous fn OR
```

Why we need Partial application

This feature makes program reusable & flexible

```
map [1,2,3] (add 2)
```

is much better than:

```
map [1,2,3] ((y)=> add 2+y)  -- need new anonymous fn OR

let add2 = (y)=> add 2+y; map [1,2,3] add2

-- need to define a new and named function
```

Why we need Partial application

It's powerful also because you can complete the computation freely

```
fetchComment:: ArticleID → IO Comment
```

Why we need Partial application

It's powerful also because you can complete the computation freely

```
fetchComment:: ArticleID → IO Comment

let doFetch = (flip map) fetchComment  -- we don't have [a] now.
```

flip map: (a → b) → [a] → [b]

Why we need Partial application

It's powerful also because you can complete the computation freely

```
fetchComment:: ArticleID → IO Comment

let doFetch = (flip map) fetchComment  -- we don't have [a] now.

                flip map: (a → b) → [a] → [b]

-- we do `flip` here because our map use different signature from Haskell's
```

Why we need Partial application

It's powerful also because you can complete the computation freely

```
fetchComment:: ArticleID → IO Comment

let doFetch = (flip map) fetchComment  -- we don't have [a] now.

                  flip map: (a → b) → [a] → [b]
renderComments:: [IO Comment] → IO DOM
```

Why we need Partial application

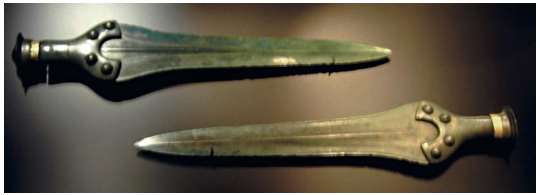
It's powerful also because you can complete the computation freely

```
fetchComment:: ArticleID → IO Comment

let doFetch = (flip map) fetchComment  -- we don't have [a] now.

                  flip map: (a → b) → [a] → [b]
renderComments:: [IO Comment] → IO DOM

(...after we get the article IDs): renderComments (doFetch IDs)

                                    flip map: (a → b) → [a] → [b]
```

Why we need Partial application

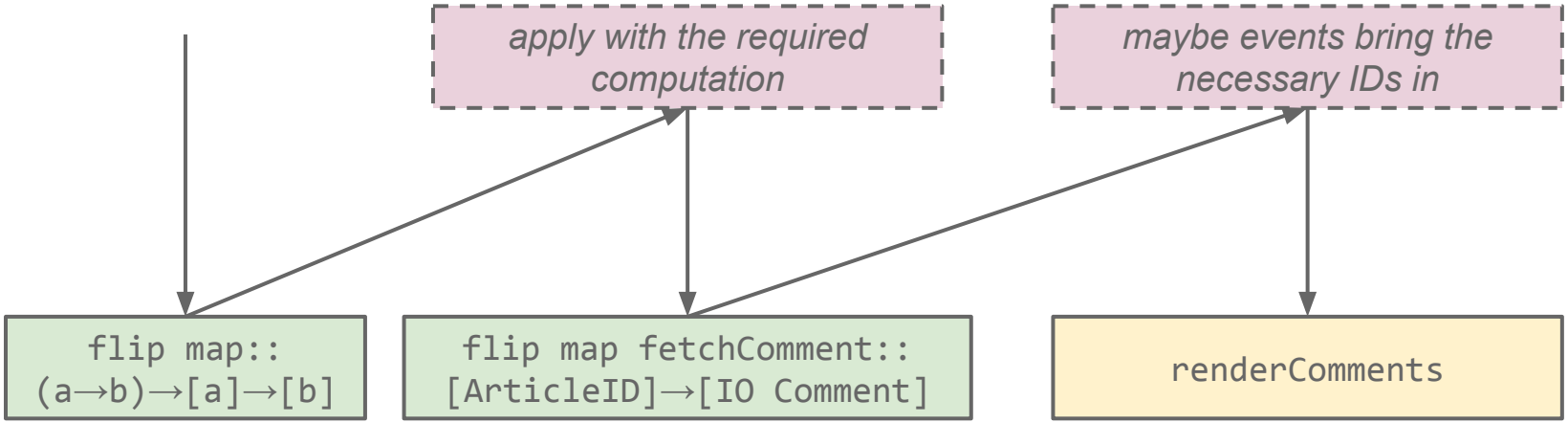
Just like a hero need to collect all things to clean the stage...


+

+

=

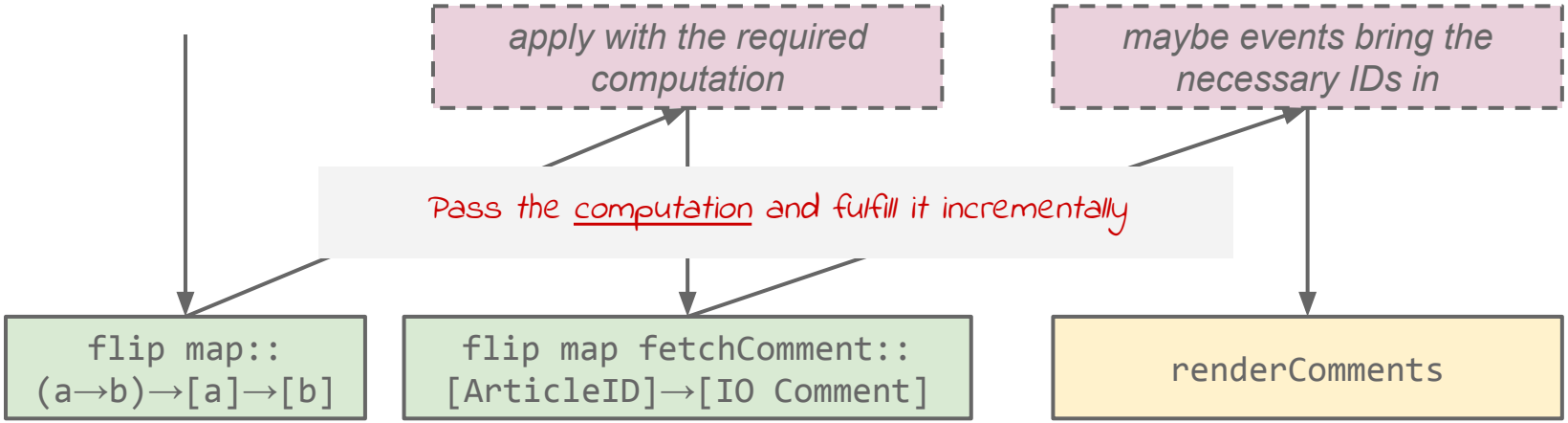

Why we need Partial application

A partially applicated function can be fulfilled later in a correct time



Why we need Partial application

A partially applicated function can be fulfilled later in a correct time



How to use it in JavaScript?

Partial Application in JavaScript

A simple "partial application" is ready if you play the `bind` trick

Partial Application in JavaScript

A simple "partial application" is ready if you play the `bind` trick

```
var map = (array, fn) => array.map(fn)      // make it pure (no 'this' is required)
var flip = (fn) => (a, b) => fn(b, a)       // yes this is valid with ES6 syntax
var fetchComment = (articleId) => { /*do IO and return IO comment*/ }
var renderComments = (comments) => { /*do rendering*/ }

var doFetch = flip(map).bind({}, fetchComment)  // partially apply the 'fetchComment'
var onArticleIDsCome = (IDs) => renderComments (doFetch IDs)
```

Partial Application in JavaScript

A simple "partial application" is ready if you play the `bind` trick

```
var map = (array, fn) => array.map(fn)      // make it pure (no 'this' is required)
var flip = (fn) => (a, b) => fn(b, a)       // yes this is valid with ES6 syntax
var fetchComment = (articleId) => { /*do IO and return IO comment*/ }
var renderComments = (comments) => { /*do rendering*/ }

var doFetch = flip(map).bind({}, fetchComment)  // partially apply the 'fetchComment'
var onArticleIDsCome = (IDs) => renderComments (doFetch IDs)
```

Note: bind is usually used to bind the 'this' in callbacks, but it can bind other arguments as well

Partial Application in JavaScript

Or, you can use lo-dash's `_.partial` method

```
// copy from lo-dash's API page
var greet = function(greeting, name) { return greeting + ' ' + name; };
var hi = _.partial(greet, 'hi');
hi('fred');
// → 'hi fred'
```

It depends whether you want to introduce a library in your project.

Currying

Composition: Currying

Curry: (a, b, c, d) → `(a)(b)(c)(d)`

Composition: Currying

Curry: (a, b, c, d) → `(a)(b)(c)(d)`

Or, if you like more formal introduction:

> *In mathematics and computer science, **currying** is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument (partial application). It was introduced by Moses Schönfinkel and later developed by Haskell Curry.*
>
> *-- Wikipedia(en): Currying*

Composition: Currying

Curry: (a, b, c, d) →**(a)(b)(c)(d)**

With curry, we can turn this

```
switchApp(appCurrent, appNext,
          switching, openAnimation, closeAnimation)
```

Steal from Gaia, System::AppwindowManager

Composition: Currying

Curry: (a, b, c, d) →`(a)(b)(c)(d)`

With curry, we can turn this

```
switchApp(appCurrent, appNext,
          switching, openAnimation, closeAnimation)
```

Steal from Gaia, System::AppwindowManager

Into this

```
curriedSwitchApp(appCurrent)(appNext)
         (switching)(openAnimation)(closeAnimation)
```

Composition: Currying

Curry: (a, b, c, d) →`(a)(b)(c)(d)`

With curry, we can turn this

```
switchApp(appCurrent, appNext,
          switching, openAnimation, closeAnimation)
```

Steal from Gaia, System::AppwindowManager

Into this

```
curriedSwitchApp(appCurrent)(appNext)
         (switching)(openAnimation)(closeAnimation)
```

In fact this is not a good example to show partial application & curry. But it's arity is so high, so...

Composition: Currying

Curry: (a, b, c, d) →**(a)(b)(c)(d)**

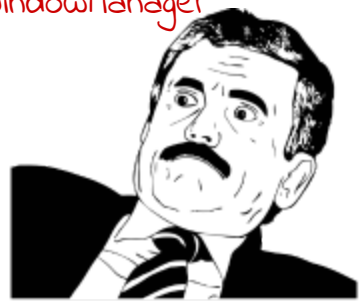
With curry, we can turn this

```
switchApp(appCurrent, appNext,
          switching, openAnimation, closeAnimation)
```

Steal from Gaia, System::AppwindowManager

Into this

```
curriedSwitchApp(appCurrent)(appNext)
         (switching)(openAnimation)(closeAnimation)
```



In fact this is not a good example to show partial application & curry. But it's arity is so high, so...

BUT...WHY ?!

Composition: Currying

With nice syntax, curry is natural

```
let result = foldr (\x y -> x + y) 0 [1..13]
```

Composition: Currying

With nice syntax, curry is natural

```
let result = foldr (\x y -> x + y) 0 [1..13]
```

Without such sugar, it becomes very clumsy

```
var result = foldr ((x, y) => x + y)) (0) (_.range(1, 13))
```

Composition: Currying

With nice syntax, curry is natural

```
curriedSwitchApp appCurrent appNext
                 switching openAnimation closeAnimation
```

Without such sugar, it becomes very clumsy

```
curriedSwitchApp(appCurrent)(appNext)
                (switching)(openAnimation)(closeAnimation)
```

Composition: Currying

With nice syntax, curry is natural

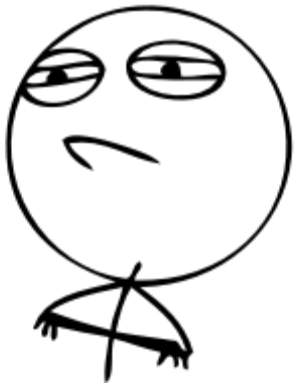
```
curriedSwitchApp appCurrent appNext
                 switching openAnimation closeAnimation
```

Without such sugar, it becomes very clumsy

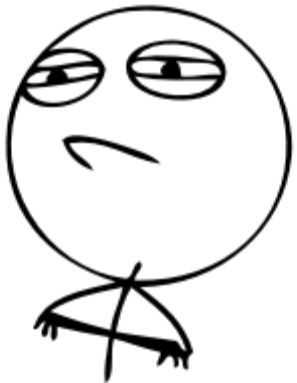
```
curriedSwitchApp(appCurrent)(appNext)
                (switching)(openAnimation)(closeAnimation)
```

We'll see more parentheses hell later, when we discuss the 'Monad'

But this can't stop us to implement this feature!



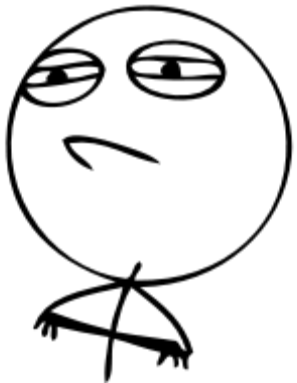
How to Curry JavaScript Functions



How to Curry JavaScript functions

Two ways lead to currying:

1. Define as curried function
2. Call helper to curry or uncurry it dynamically



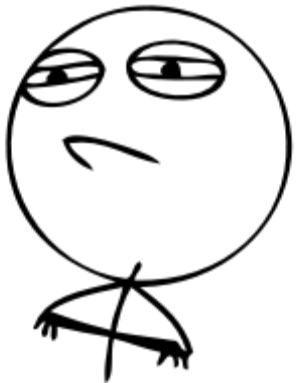
How to Curry JavaScript functions

Two ways lead to currying:

1. Define as curried function
2. Call helper to curry or uncurry it dynamically

without ES6, it could be a disaster.

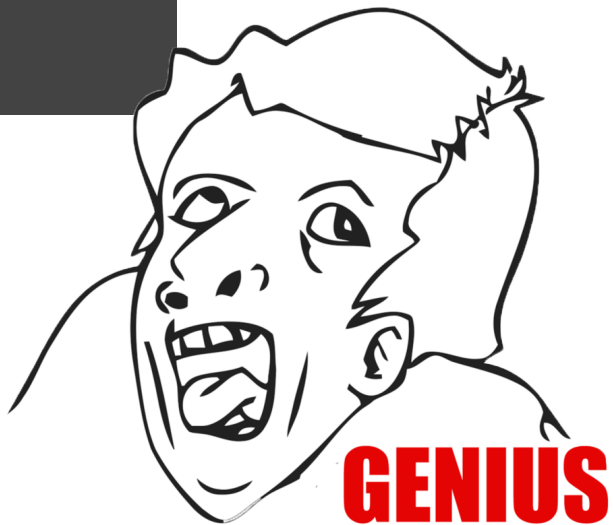
lo-dash has this: _.curry, but no _.uncurry



How to Curry JavaScript functions

Define a curried function without ES6 supporting...

```
var curriedSwitchApp =
function (appCurrent) {
return function (appNext) {
return function (switching) {
return function (openAnimation) {
return function (closeAnimation) {
    // do something...
}}}}}
```



How to Curry JavaScript functions

With ES6 (fat arrow function)

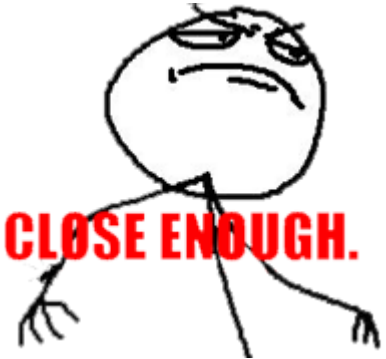
```
var curriedSwitchApp =
(appCurrent) => (appNext) =>  (switching) => (openAnimation) => (closeAnimation) => {
    // do something...
}
```

How to Curry JavaScript functions

Compare to Haskell...

```
var curriedSwitchApp =
(appCurrent) => (appNext) =>  (switching) => (openAnimation) => (closeAnimation) => {
    // do something...
}
```

```
let curriedSwitchApp appCurrent appNext switching openAnimation closeAnimation = -- do something
```



How to Curry JavaScript functions

Always use fat arrow to define pure function (binding no 'this')

```
var curriedSwitchApp =
(appCurrent) => (appNext) =>  (switching) => (openAnimation) => (closeAnimation) => {
   // do something...
}
```

```
var curriedSwitchApp =
function (appCurrent) {
return function (appNext) {
return function (switching) {
return function (openAnimation) {
return function (closeAnimation) {
    // do something...
}}}}}
```

...but, why we need Currying, after all?

...but, why we need Currying, after all?

Currying naturally make functions can be applied partially

And partial application makes program reusable & flexible

...but, why we need Currying, after all?

Currying naturally make functions can be applied partially

And partial application makes program reusable & flexible

IMO to use it is worth if we can adopt these in a basic library or framework

Part II

Computation =

Transformation +

Composition +

Context

Context

Let's think about the type of map

Context

Let's think about the type of map

```
[a] → (a → b) → [b]
```

```
m a → (a → b) → m b, while m/[ ]
```

Again, I don't follow Haskell's signatures strictly.

Context

Let's think about the type of map

```
[a] → (a → b) → [b]
```

```
m a → (a → b) → m b, while m/[ ]
```

So the List#map is only a special case of such 'map'...

Context

Let's think about the type of map

```
[a] → (a → b) → [b]

m a → (a → b) → m b, while m/[ ]

Maybe a → (a → b) → Maybe b

HTTP Request → (Request → Response) → HTTP Response
```

So the List#map is only a special case of such 'map'...

Context

Note their only difference is they're in different contexts

```
[a] → (a → b) → [b]

m a → (a → b) → m b, while m/[ ]

Maybe a → (a → b) → Maybe b

HTTP Request → (Request → Response) → HTTP Response
```

Context

Note their only difference is they're in different contexts

```
[a] → (a → b) → [b]
```

```
m a → (a → b) → m b, while m/[ ]
```

```
Maybe a → (a → b) → Maybe b
```

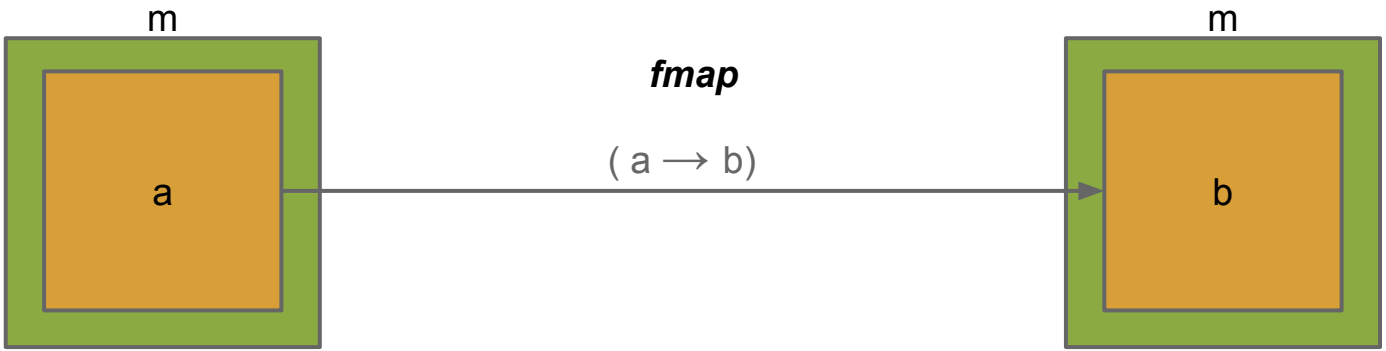
```
HTTP Request → (Request → Response) → HTTP Response
```

In fact, this kind of map called `fmap` in Haskell. Things with `fmap` called Functor

Functor

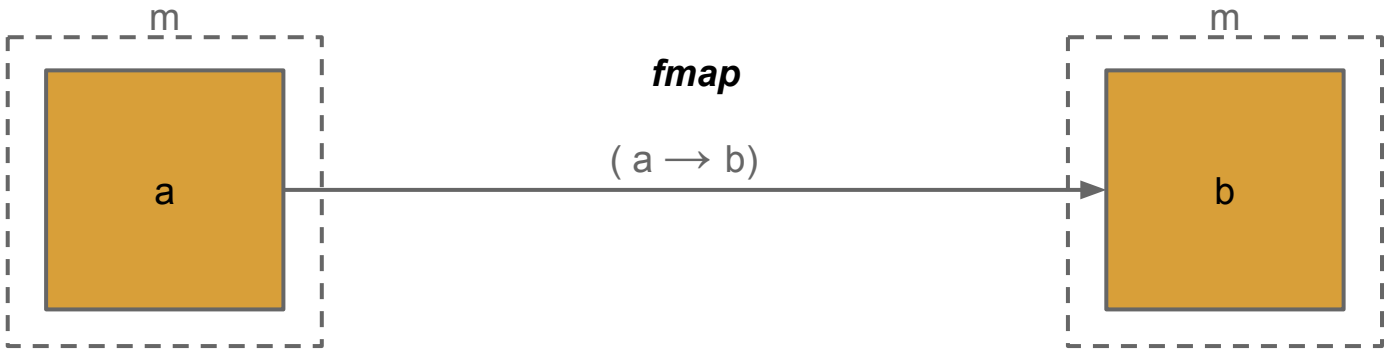
Context: Functor

Functor can **lift** a function into the specific context



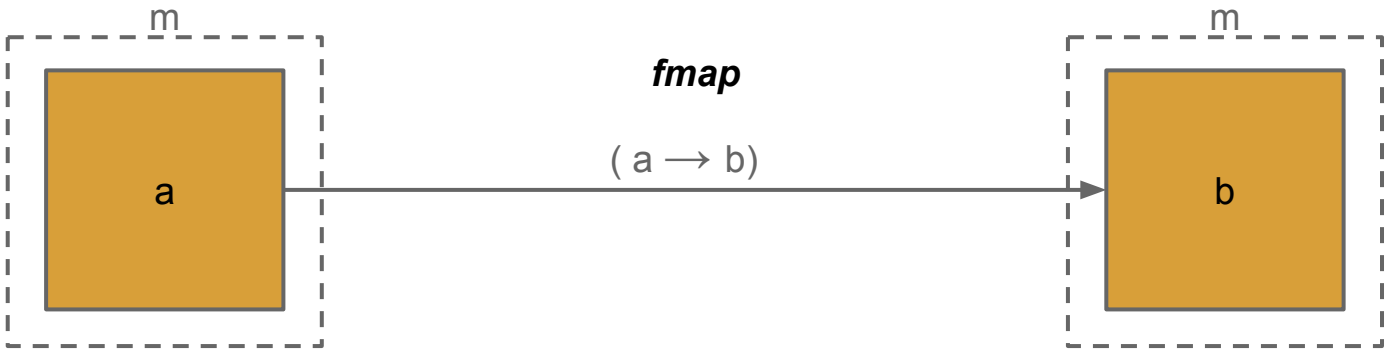
Context: Functor

The function, needn't to know anything about the context



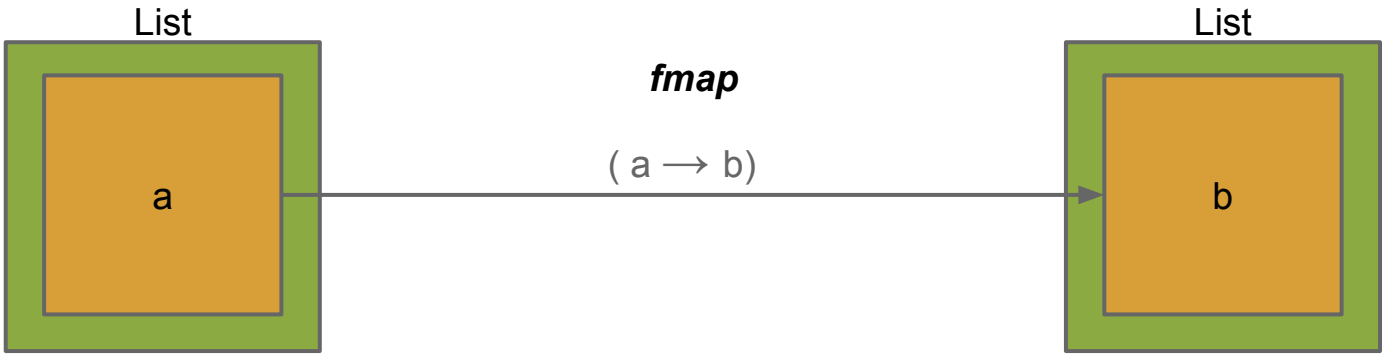
Context: Functor

It only need to care how turn the value from **a** to **b**



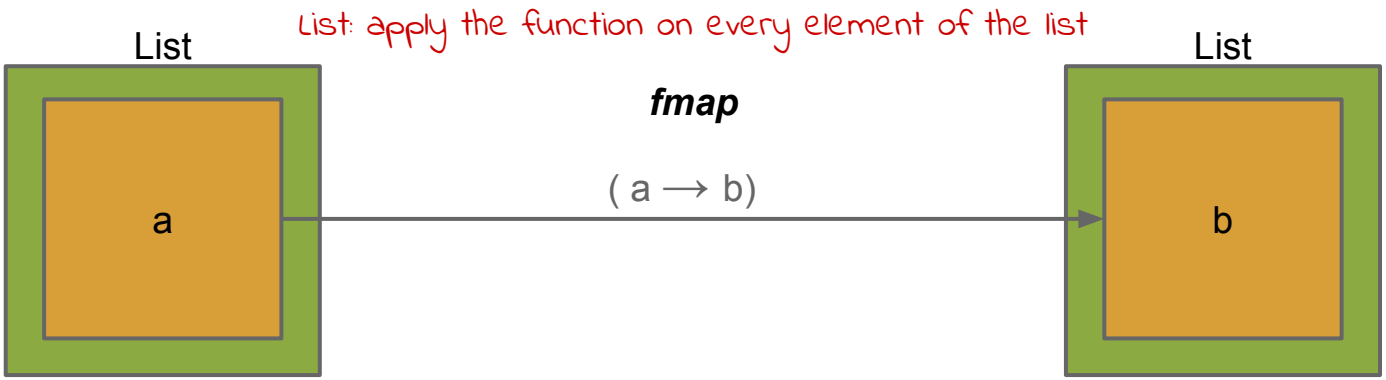
Context: Functor

How to apply context-relevant rules is encapsulated by fmap



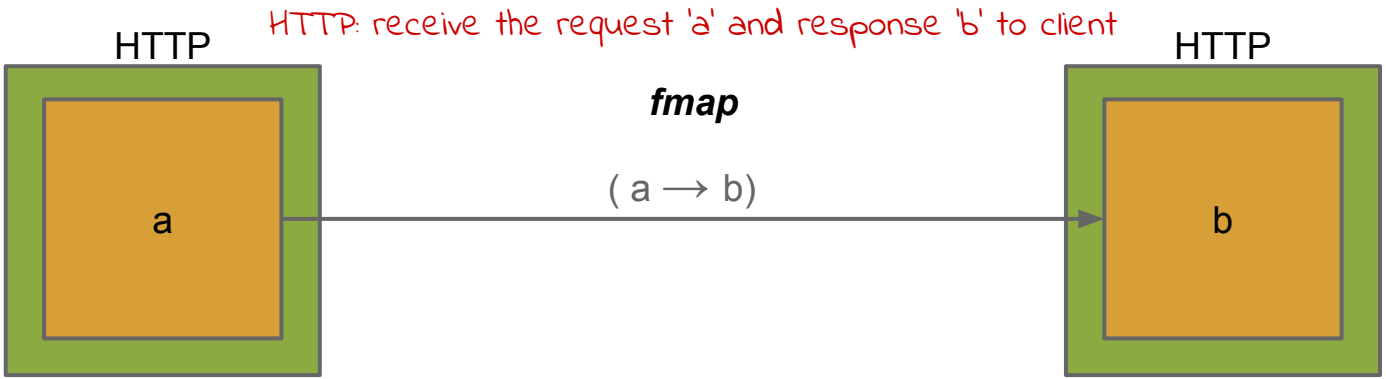
Context: Functor

How to apply context-relevant rules is encapsulated by fmap



Context: Functor

How to apply context-relevant rules is encapsulated by fmap



Context: Functor

Therefore the interface keep the same but implementations are various

```
List#fmap:: apply fn on every elements of the list
```

Context: Functor

Therefore the interface keep the same but implementations are various

```
List#fmap:: apply fn on every elements of the list

Maybe#fmap:: apply fn on the value OR not: if the value is Nothing
             then do nothing; otherwise, apply on and update it
```

Context: Functor

Therefore the interface keep the same but implementations are various

```
List#fmap:: apply fn on every elements of the list

Maybe#fmap:: apply fn on the value OR not: if the value is Nothing
             then do nothing; otherwise, apply on and update it

HTTP#fmap:: apply fn on the request to get the response, and do
            some underlying IO to send to the client
```

Context: Functor

Therefore the interface keep the same but implementations are various

```
List#fmap:: fmap [1,2] (+1) -- [2, 3]

Maybe#fmap:: fmap (Just 1) Nothing -- Maybe Just Nothing

             fmap (Nothing) 99     -- Maybe Nothing

HTTP#fmap:: fmap someRequest response404 -- (client get 404 page)
```

Context: Functor

We still need some constructor to lift pure value into a Functor

```
List#(constructor):: []                    -- [1,2,3] gives a List
```

Context: Functor

We still need some constructor to lift pure value into a Functor

```
List#(constructor):: []                        -- [1,2,3] gives a List

Maybe#(constructor):: Just a | Nothing -- Just 3 gives a Maybe;
                                       -- Nothing gives a Maybe,too
```

Context: Functor

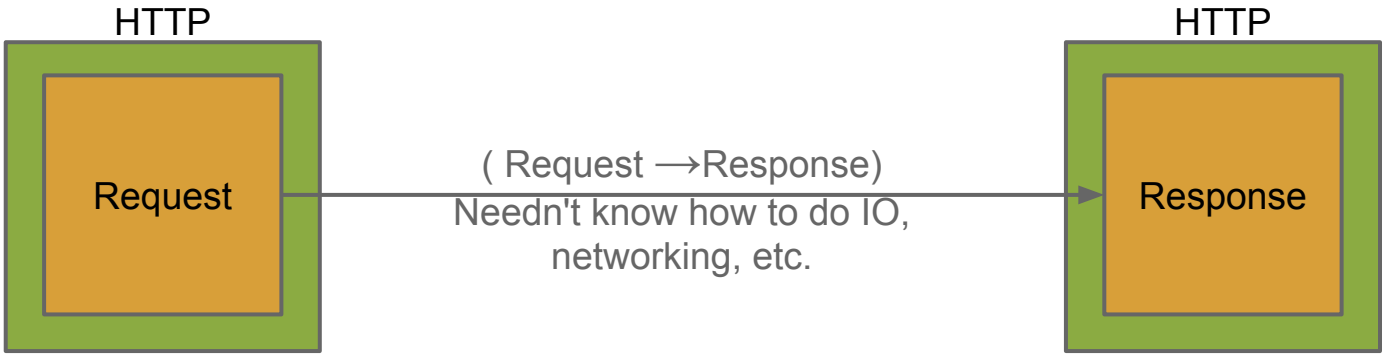
We still need some constructor to lift pure value into a Functor

```
List#(constructor):: []                      -- [1,2,3] gives a List

Maybe#(constructor):: Just a | Nothing -- Just 3 gives a Maybe;
                                       -- Nothing gives a Maybe,too

HTTP#(constructor):: startHTTPServer + client request, maybe
```

Context: Functor

This concept is useful because the function can keep simple

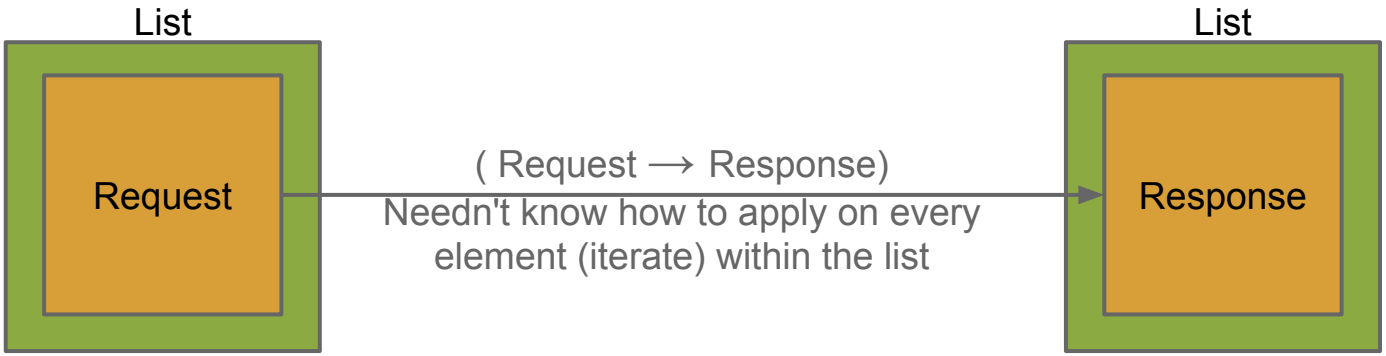
And apply them into different contexts (to do different things)



Context: Functor

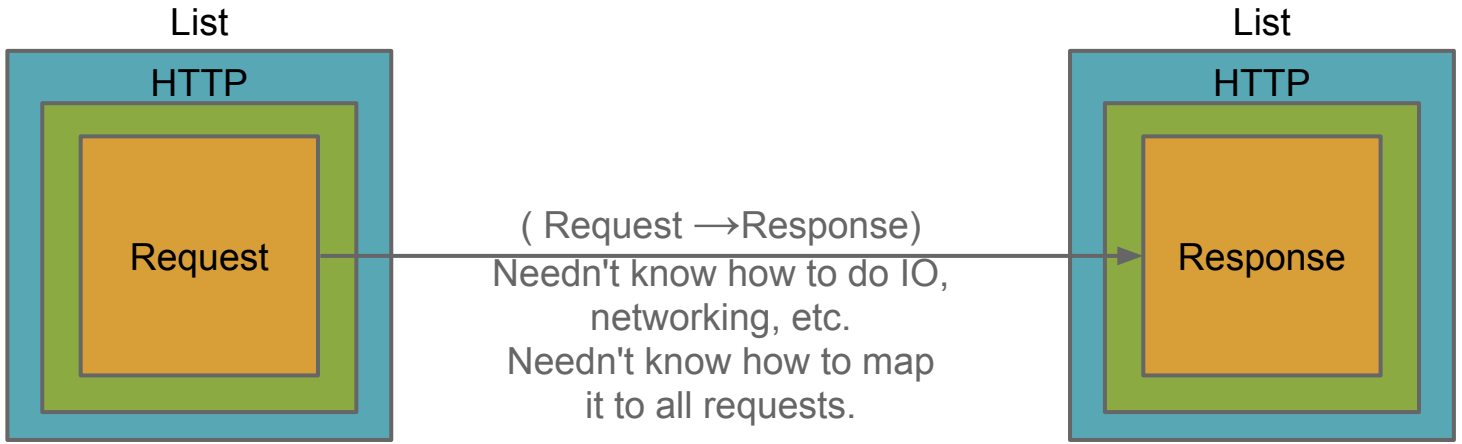
This concept is useful because the function can keep simple

And apply them into different contexts (to do different things)



Context: Functor

Contexts also [can be stockpiled] to do complex computations

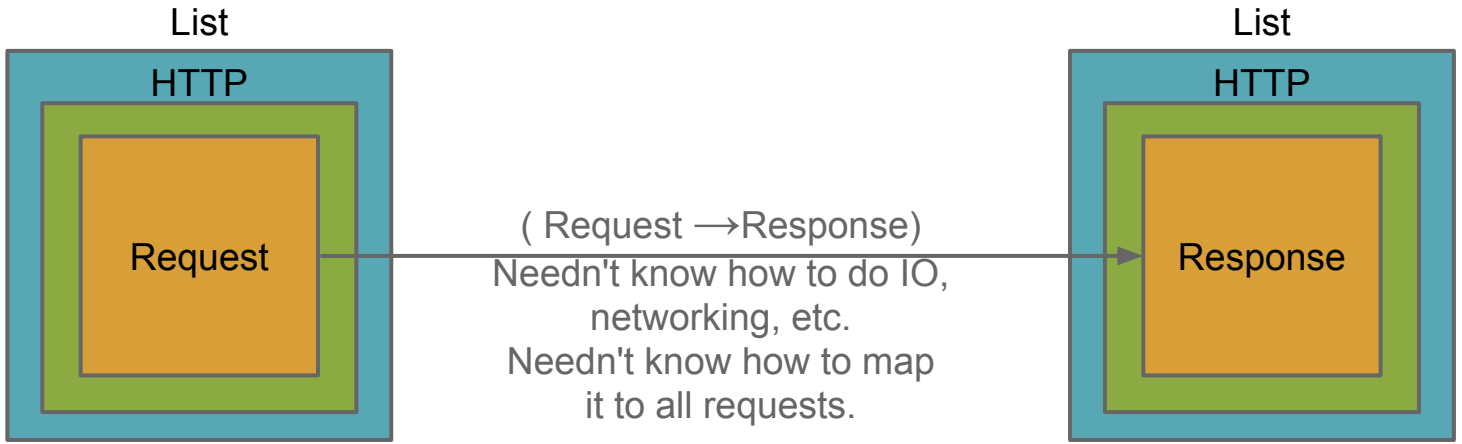


Context: Functor

Contexts also can be stockpiled to do complex computations

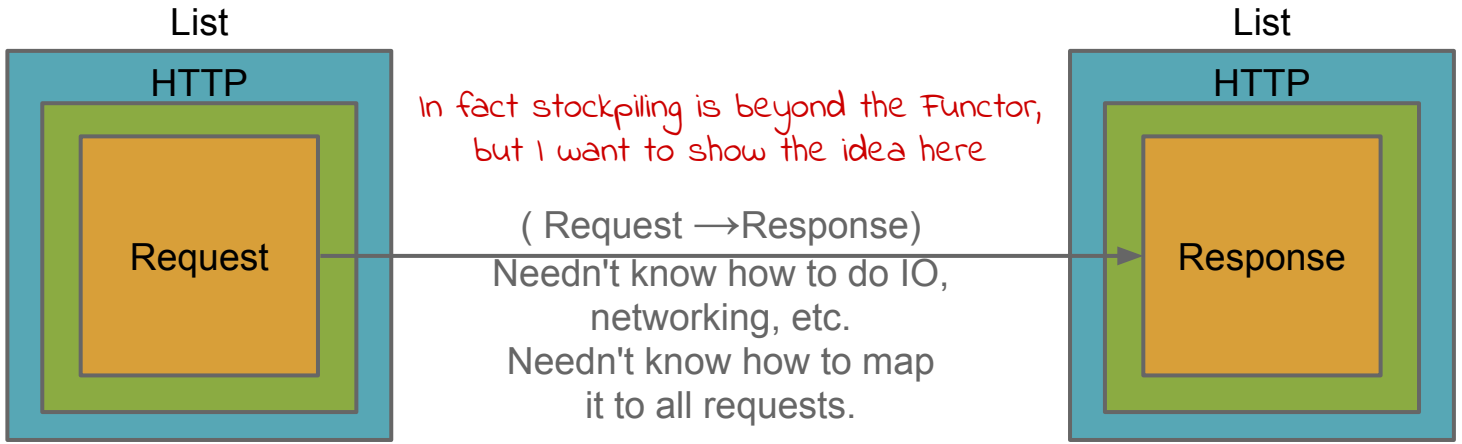
Context: Functor

Contexts also [can be stockpiled] to do complex computations



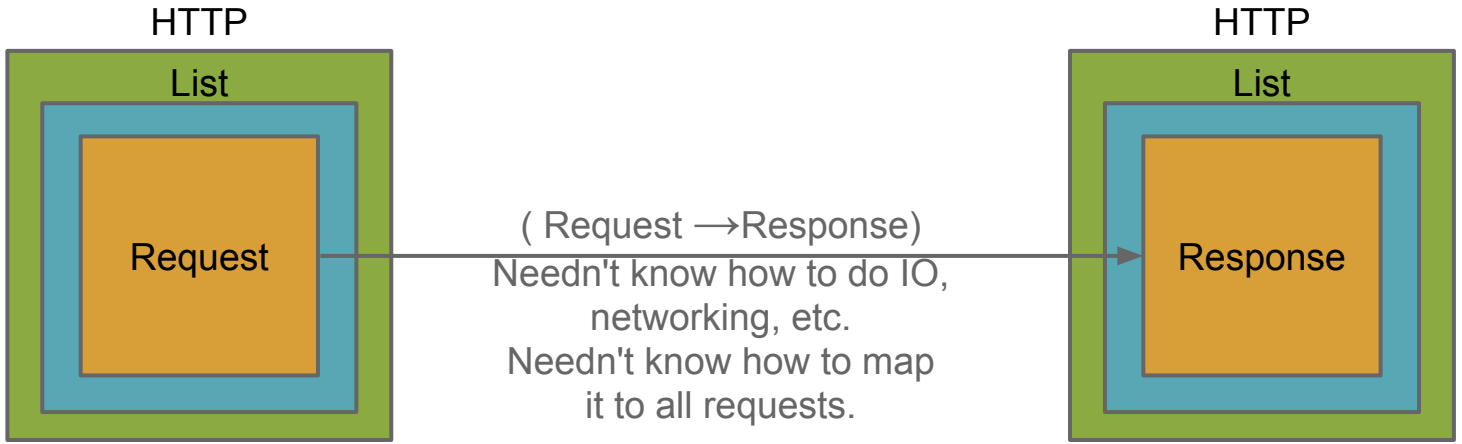
Context: Functor

Contexts also [can be stockpiled] to do complex computations



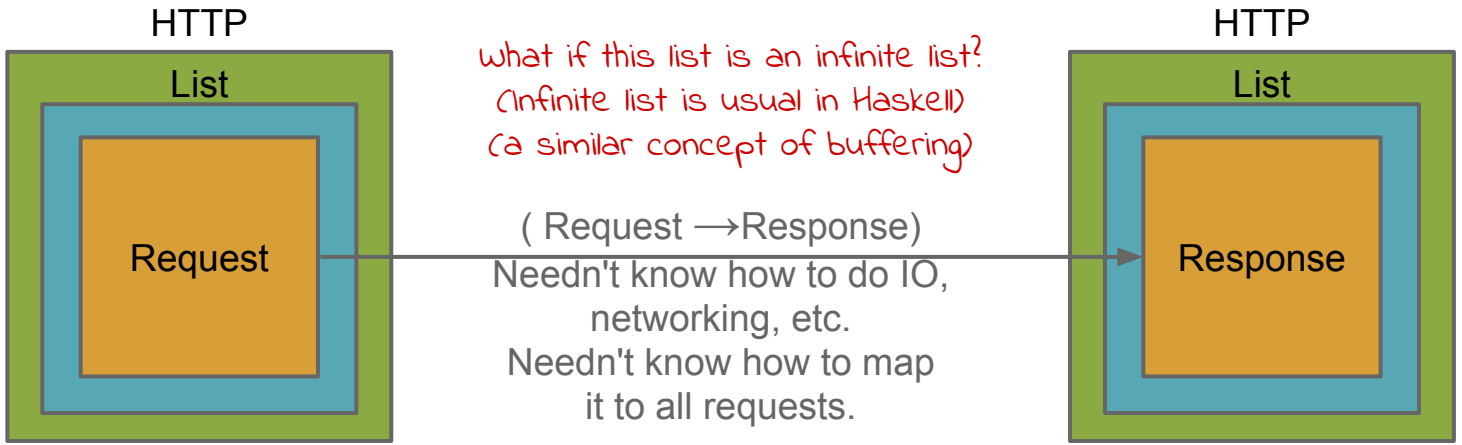
Context: Functor

Tip: the stockpiling order matters...



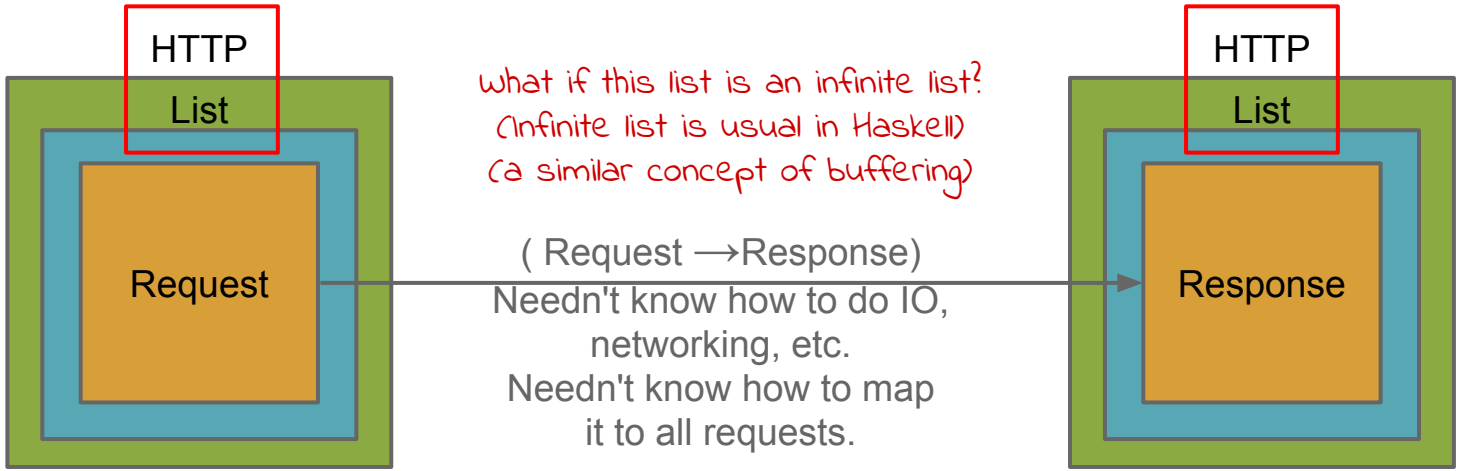
Context: Functor

Tip: the stockpiling order matters...



Context: Functor

How to define the transformer is also important



Context: Functor

Functor is useful, but it's not enough

Context: Functor

Functor is useful, but it's not enough

Sometimes we want to control the way to lift & process a value

Context: Functor

Functor is useful, but it's not enough

Sometimes we want to control the way to lift & process a value

Context: Functor

Functor is useful, but it's not enough

Sometimes we want to control the way to lift & process a value

```
fileStatus:: FilePath → IO FileStatus
```

```
range:: (Number,Number) → [] Number
```

Context: Functor

Functor is useful, but it's not enough

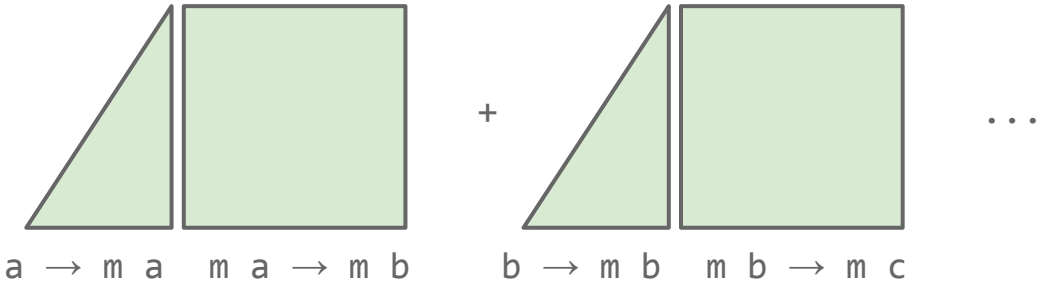
Sometimes we want to control the way to lift & process a value

```
fileStatus:: FilePath → IO FileStatus

range:: (Number,Number) → [] Number
```

That's why I don't show Functor in JavaScript here. And yes, I omit the Applicative here...

Context: Functor

So we need an advanced structure to describe the computations under the context, with some reasonable ways to compose them together



Monad

Context: Monad

Monad give us the power to control lifting & processing in bind

Context: Monad

Monad give us the power to control lifting & processing in bind

```
fmap:: m a → (a → b) → m b     -- Functor

bind:: m a → (a → m b) → m b   -- Monad
```

Context: Monad

Monad give us the power to control lifting & processing in bind

```
fmap:: m a → (a → b) → m b    -- Functor

bind:: m a → (a → m b) → m b  -- Monad

-- bind do unwrap 'm b' to 'b' implicitly, and pass it to next step
-- Things with bind and 'return' become Monad. And in theory,
-- every monad is an applicative functor (as well as a functor)
```

Context: Monad

Monad give us the power to control lifting & processing in bind

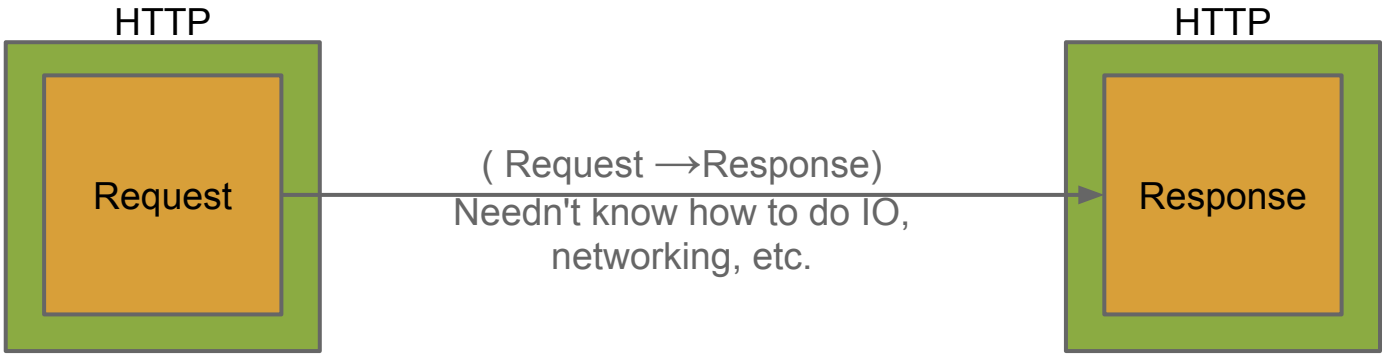
```
fmap:: m a → (a → b) → m b    -- Functor

bind:: m a → (a → m b) → m b  -- Monad

-- bind do unwrap 'm b' to 'b' implicitly, and pass it to next step
-- Things with bind and 'return' become Monad. And in theory,
-- every monad is an applicative functor (as well as a functor)
```

Yeah, I know this is not exactly express what Monad is, since there are so many tutorials and math explanations.

Context: Monad

For example, in our HTTP Functor:

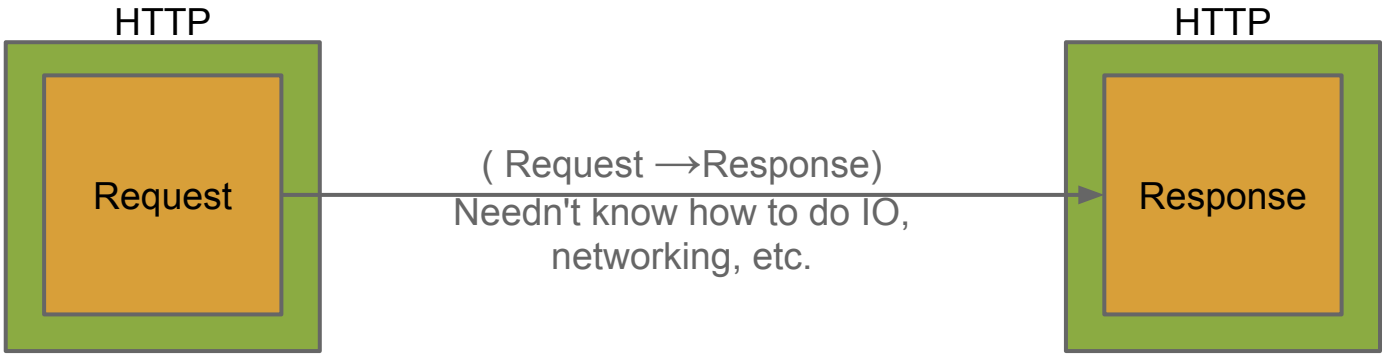
```
fmap:: m a → (a → b) → m b
```



Context: Monad

For example, in our HTTP Functor:

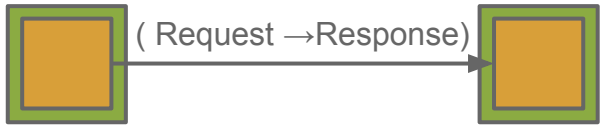
```
fmap:: m a → (a → b) → m b
```

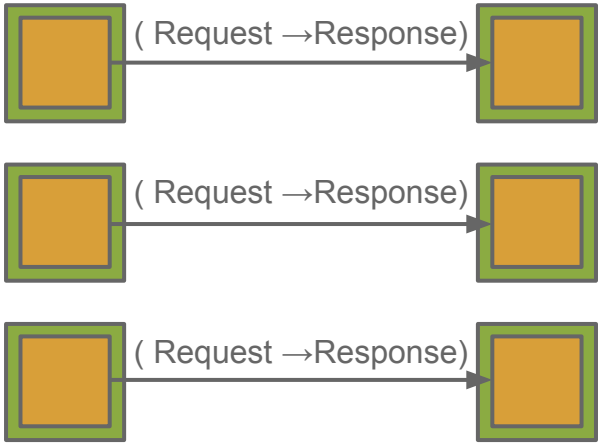


But we can't concat these handlers to push things sequentially to client (pushlet), or to change the behavior according to the previous result

Context: Monad

The need to sequentially manipulate values in the context





Context: Monad

The need to sequentially manipulate values in the context

```
fmap clientRequest responseHello
```

```
clientRequest
    >>= (\req -> return loginPage)
    >>= (\authReq -> case (doAuth authReq) of
          True -> return contentPage
          False -> return loginPage))
```

In Haskell, the '>>=' is the function 'bind', infix.

Context: Monad

```
clientRequest
    >>= (\req -> return loginPage)
    >>= (\authReq -> case (doAuth authReq) of
          True -> return contentPage
          False -> return loginPage))
```

Context: Monad

```
clientRequest
    >>= (\req -> return loginPage)
    >>= (\authReq -> case (doAuth authReq) of
          True -> return contentPage
          False -> return loginPage))
```

the 'bind' function, infix

NOT that 'return'!

Context: Monad

The 'return' means the default wrapping (lifting) function

Context: Monad

The 'return' means the default wrapping (lifting) function

```
(\req -> return loginPage):: Request → HTTP Response
```

```
m      #return a = m a
List   #return a =  [a]
Maybe#return a = Just a
HTTP   #return a = HTTP a
```

Context: Monad

The 'return' means the default wrapping (lifting) function

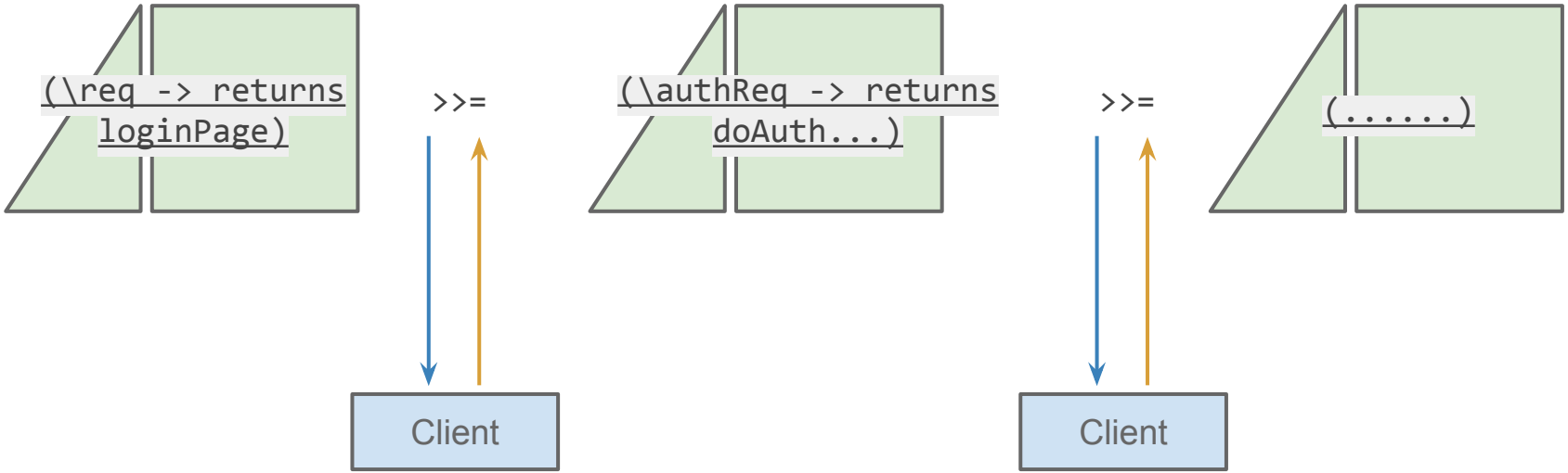
```
(\req -> return loginPage):: Request → HTTP Response
```

```
m    #return a = m a
List #return a =  [a]
Maybe#return a = Just a
HTTP #return a = HTTP a
```

I know the name is confused...if you're used to other language's 'return'

Context: Monad

How bind works (example)



Context: Monad

Monad allow us compose computations with more possibilities

Context: Monad

Monad allow us compose computations with more possibilities

1. Implement statements, state machine, etc.

Context: Monad

Monad allow us compose computations with more possibilities

1. Implement statements, state machine, etc.

2. Encapsulate side-effects and keep program pure*

Yes, I know this is controversial, but people use Monad to do that

Context: Monad

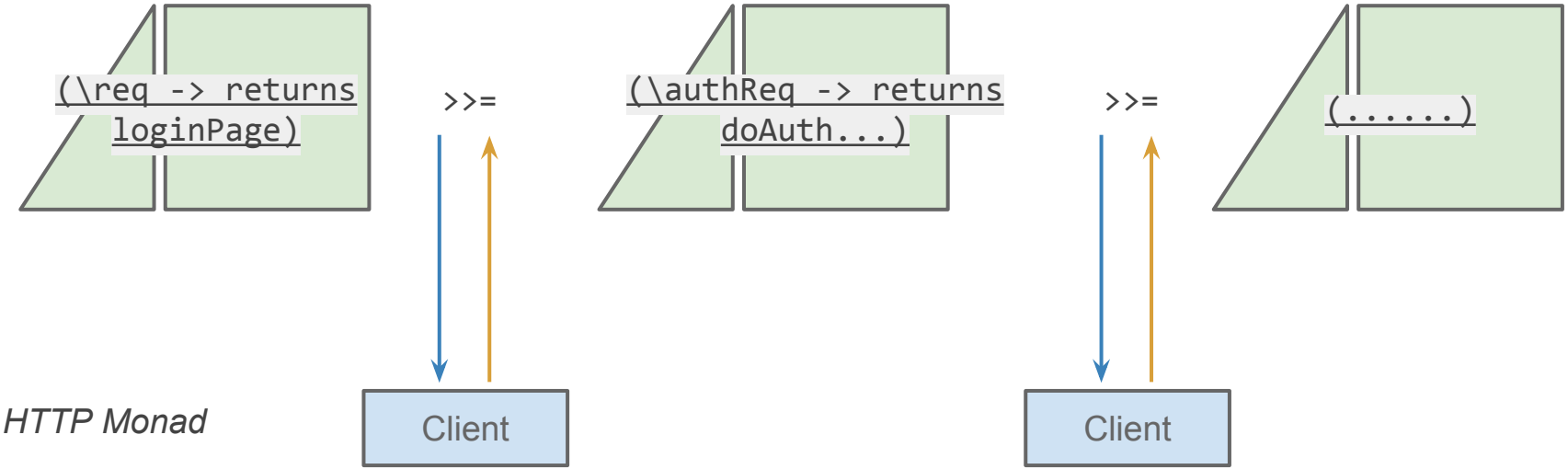
Monad allow us compose computations with more possibilities

1. Implement statements, state machine, etc.
2. Encapsulate side-effects and keep program pure*
3. Stockpile different Monads to gain more abilities

Yes, I know this is controversial, but people use Monad to do that

Context: Monad

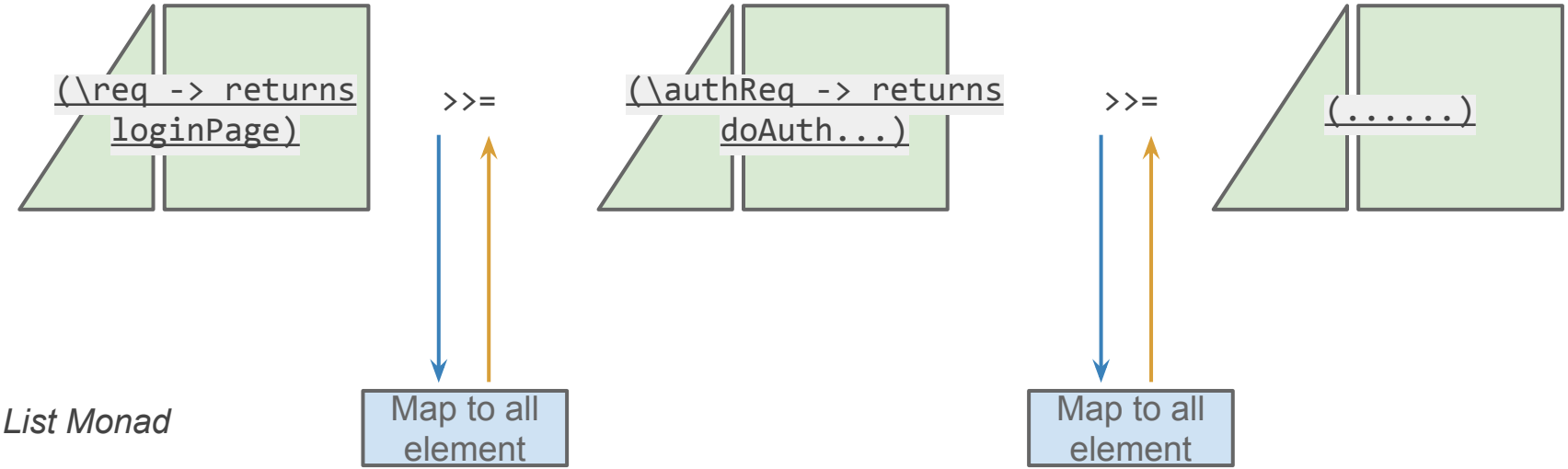
Different Monad has the same bind with different implementations



HTTP Monad

Context: Monad

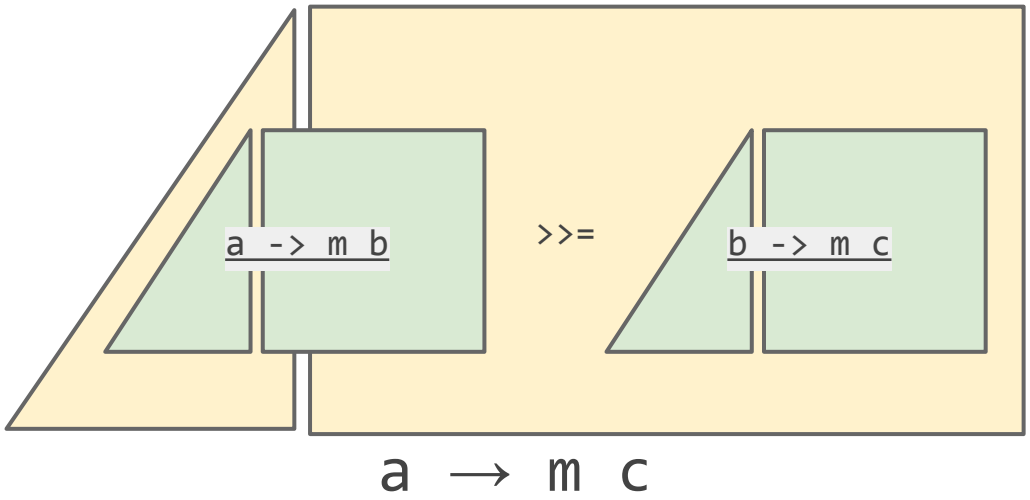
Different Monad has the same bind with different implementations



List Monad

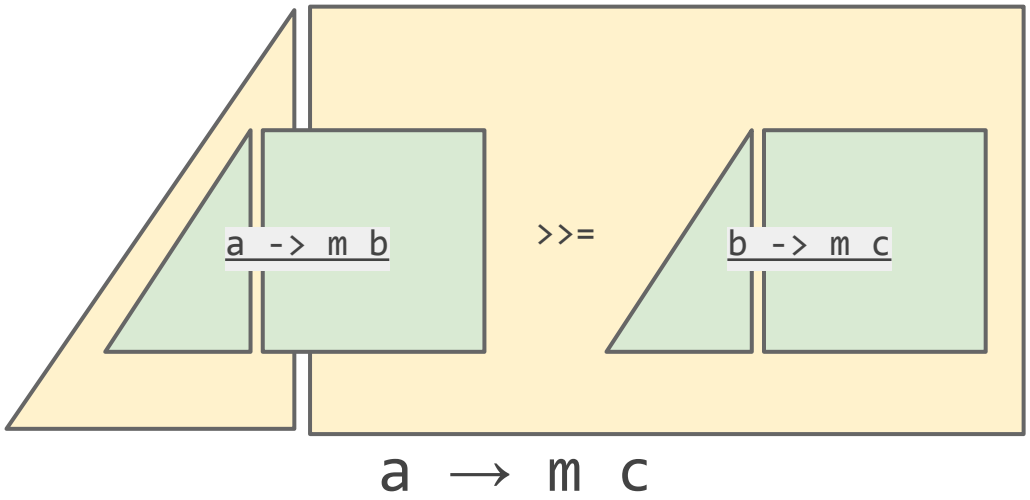
Context: Monad

Monadic actions can be chained with other actions while remaining the same type



Context: Monad

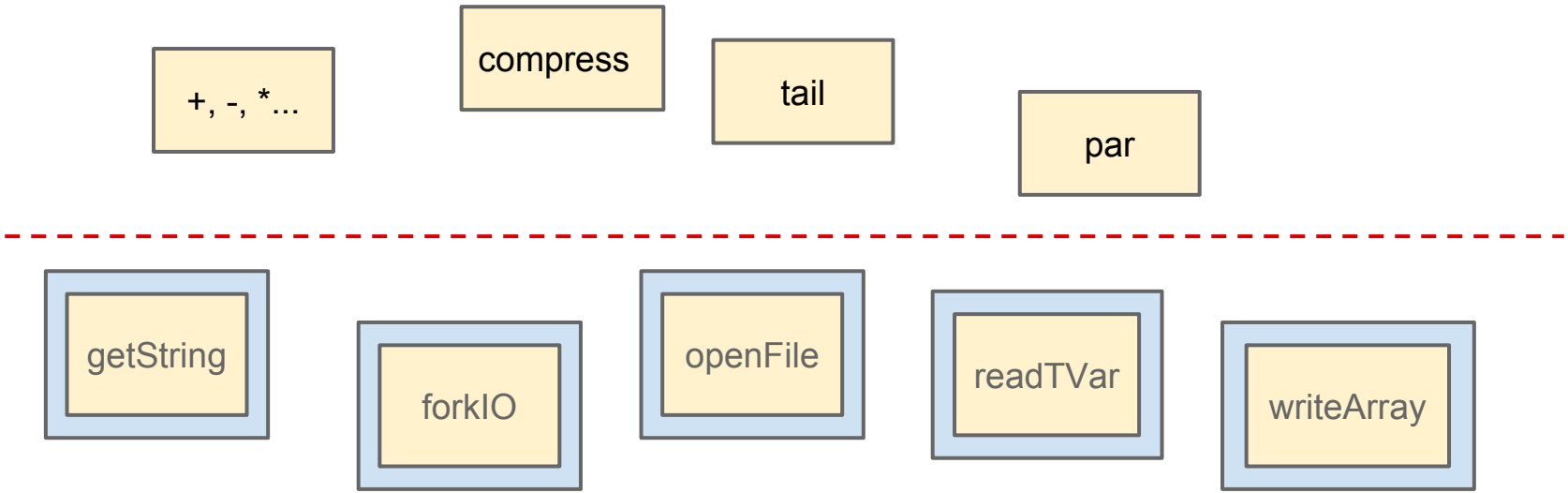
Monadic actions can be chained with other actions while remaining the same type




```
actionFoo = actionA >>= actionB
actionBar = actionFoo >>= actionC

(and so on...)
```

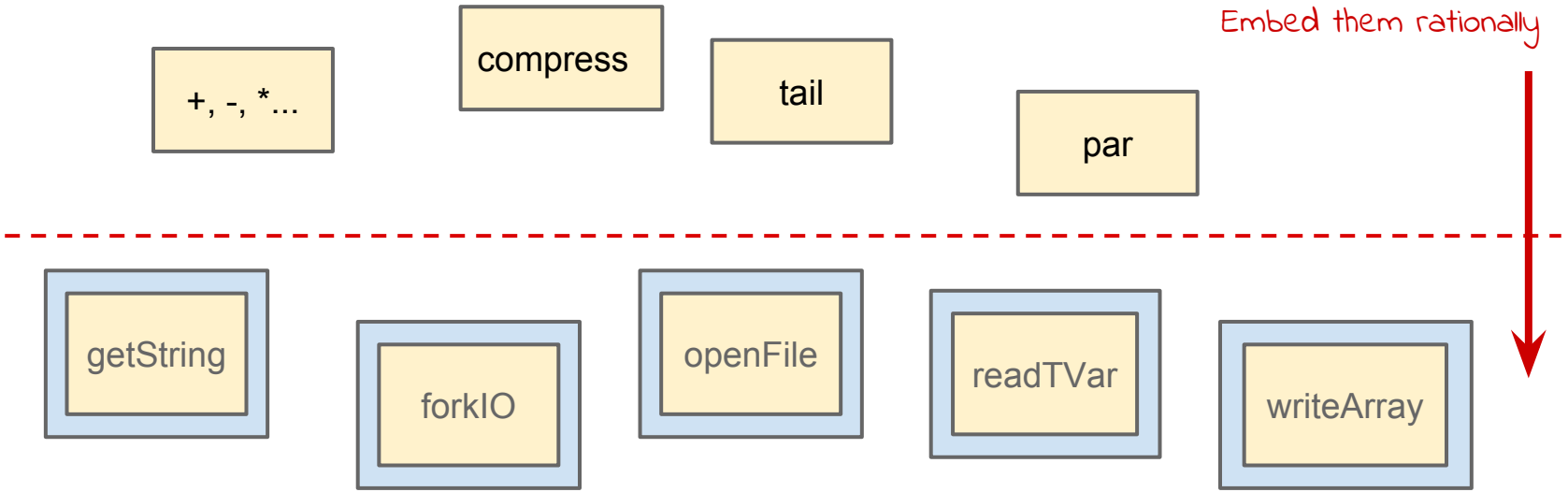
Context: Monad

Monadic actions and pure functions can be distinguished to prevent unexpected side-effects, if all methods with side-effects are wrapped



Context: Monad

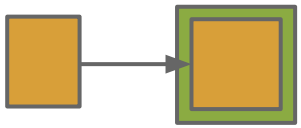
Monadic actions and pure functions can be distinguished to prevent unexpected side-effects, if all methods with side-effects are wrapped



Context: Monad

Another reason that Monad could encapsulate side-effects is:

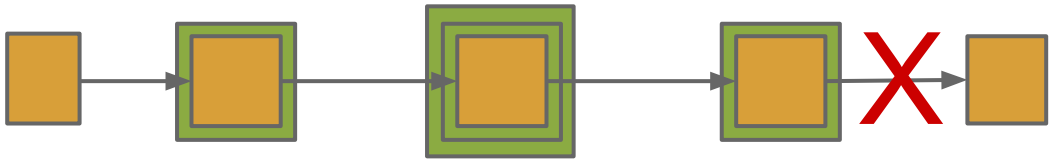
```
m#return:: a → m a
m#bind:: m a → (a → m b) → m b
```



Context: Monad

Another reason that Monad could encapsulate side-effects is:

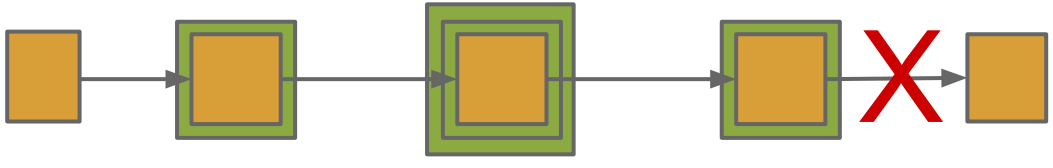
```
m#return:: a → m a
m#bind:: m a → (a → m b) → m b
```



Context: Monad

Another reason that Monad could encapsulate side-effects is:

```
m#return:: a → m a
m#bind:: m a → (a → m b) → m b
```



Context: Monad

Another reason that Monad could encapsulate side-effects is:

```
m#return:: a → m a
m#bind:: m a → (a → m b) → m b
```

There is no way to allow a monadic value escape:

```
m#unwrap:: m a → a
```

Context: Monad

Another reason that Monad could encapsulate side-effects is:

```
m#return:: a → m a
m#bind:: m a → (a → m b) → m b
```

There is no way to allow a monadic value escape:

```
m#unwrap:: m a → a
```

(Yes, I know Comonad or unsafe- can do that, but...)*

Context: Monad

So once your value get tainted via IO Monad (get from IO), you can never extract it to feed other outside computations

Context: Monad

So once your value get tainted via IO Monad (get from IO), you can never extract it to feed other outside computations

This is because IO operations come with side-effects

Context: Monad

The similar case in JavaScript is the Promised actions:

Context: Monad

The similar case in JavaScript is the Promised actions:

```
var promised =
Promise(() => {...})
  .then((a) => {...})
  .then((b) => {...})
  .then((c) => {...})
```

Context: Monad

The similar case in JavaScript is the Promised actions:

```
var promised =
Promise(() => {...})
  .then((a) => {...})
  .then((b) => {...})
  .then((c) => {...})
```

There is no way to get the correct value from outside. You must embed your function into the promise

Context: Monad

In fact, if we only care what Monad could bring to us, not type and other additional rules, we can find lots of similar usages in JS

```
Promise(() => {...})
  .then((a) => {...})
  .then((b) => {...})
  .then((c) => {...})
```

Context: Monad

In fact, if we only care what Monad could bring to us, not type and other additional rules, we can find lots of similar usages in JS

```
Promise(() => {...})
  .then((a) => {...})
  .then((b) => {...})
  .then((c) => {...})
```

Context: Ensure the following step only be executed after the previous one get done.

Context: Monad

In fact, if we only care what Monad could bring to us, not type and other additional rules, we can find lots of similar usages in JS

```
$('some-selector')
  .each(...)
  .animate(...)
  .append(...)
```

```
Context: Select, manipulate and check
the DOM element(s)
```

Context: Monad

In fact, if we only care what Monad could bring to us, not type and other additional rules, we can find lots of similar usages in JS

```
_.chain(someValue)
  .filter(...)
  .groupBy(...)
  .map(...)
  .reduce(...)
```

Context: Guarantee the value would be transformed by lo-dash functions

Context: Monad

In fact, if we only care what Monad could bring to us, not type and other additional rules, we can find lots of similar usages in JS

```
ensure()
  .frame()
  .element(...)
  .actions()
    .pulls(0, 400)
    .perform()
  .must(...)
```

```
Context: Ensure the integration test
only do what user can do, rather than
do something magically
```

(from Gaia project)

Context: Monad

These computations focus on the specific transforming of the context,
just like what Monads do in Haskell

```
Promise(() => {...})
  .then((a) => {...})
  .then((b) => {...})
  .then((c) => {...})
```

Async

```
$('some-selector')
  .each(...)
  .animate(...)
  .append(...)
```

DOM

```
_.chain(someValue)
 .filter(...)
 .groupBy(...)
 .map(...)
 .reduce(...)
```

_

Context: Monad

These computations focus on the specific transforming of the context,
just like what Monads do in Haskell

```
threeCoins = do
   a <- randomSt
   b <- randomSt
   c <- randomSt
   return (a,b,c)
```

State

```
main = do
   a <- ask "Name?"
   b <- ask "Age?"
   return ()
```

IO

```
add mx my = do
  x <- mx
  y <- my
  return (x + y)
```

Maybe

Context: Monad

So the question is not "*why we need Monad in JavaScript*", but "*is it worth to implement the fluent interface more Monadic*"?

How to make JavaScript More Monadic?

How to make JavaScript more Monadic?

Some requirements to get closer with real Monad

1. Eager vs. Lazy
2. Flow control mixed
3. Not enough type supporting
4. Need to follow how much Monad laws

How to make JavaScript more Monadic?

In fact it's easy to make our 'Monad' lazy with some type supporting

```
var action = (new Maybe()).Just(3)
  .then((v) => {
    return (new Maybe()).Just(v+99); })
  .then((v) => {
    return (new Maybe()).Just(v-12); })
  .then((v) => {
    return (new Maybe()).Nothing();  })
  .then((v) => {
    return (new Maybe()).Just(v+12); })

// Execute it with `action.done()`.
```

```
action = (Just 3)
        >>= \v -> return (v + 99)
        >>= \v -> return (v - 12)
        >>= \v -> Nothing
        >>= \v -> return (v + 12)
```

https://github.com/snowmantw/warmfuzzything.js/blob/master/maybe.js

How to make JavaScript more Monadic?

But things become crazy when the 'Monad' need to mix with Promise
(to support async steps natively)

```
var action = (new PromiseMaybe()).Just(3)
  .then((mSelf, v) => {
    mSelf.returns((new PromiseMaybe).Just(v+99)); })
  .then((mSelf, v) => {
    setTimeout(function() {  // Only for test. Meaningless.
      mSelf.returns((new PromiseMaybe).Just(v-12));
    }, 3000); })
  .then((mSelf, v) => {
    mSelf.returns((new PromiseMaybe).Nothing()); })
  .then((mSelf, v) => {
    mSelf.returns((new PromiseMaybe).Just(v+12)); });
```

https://github.com/snowmantw/warmfuzzything.js/blob/master/promise_maybe.js

How to make JavaScript more Monadic?

But things become crazy when the 'Monad' need to mix with Promise
(to support async steps natively)

```
var action = (new PromiseMaybe()).Just(3)
  .then((mSelf, v) => {
    mSelf.returns((new PromiseMaybe).Just(v+99)); })
  .then((mSelf, v) => {
    setTimeout(function() {  // Only for test. Meaningless.
      mSelf.returns((new PromiseMaybe).Just(v-12));
    }, 3000); })
  .then((mSelf, v) => {
    mSelf.returns((new PromiseMaybe).Nothing()); })
  .then((mSelf, v) => {
    mSelf.returns((new PromiseMaybe).Just(v+12)); });
```

https://github.com/snowmantw/warmfuzzything.js/blob/master/promise_maybe.js

It can be better, but the real problem is it's implementation is very tricky

How to make JavaScript more Monadic?

And currently it doesn't follow *Monad laws*...

How to make JavaScript more Monadic?

As a conclusion: to try to play with fluent interface and Monad may benefit us, but the more we gain the more we must pay

or someone must pay for us

How to make JavaScript more Monadic?

As a conclusion: to try to play with fluent interface and Monad may benefit us, but the more we gain the more we must pay

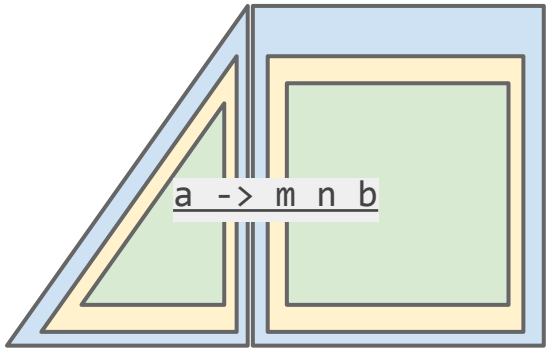
or someone must pay for us

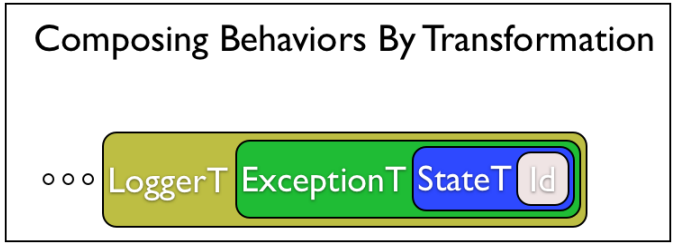
And we still have an unresolved issue here...

Monad Transformer

Context: Monad Transformer

How to stockpile two or more Monads to do various things?

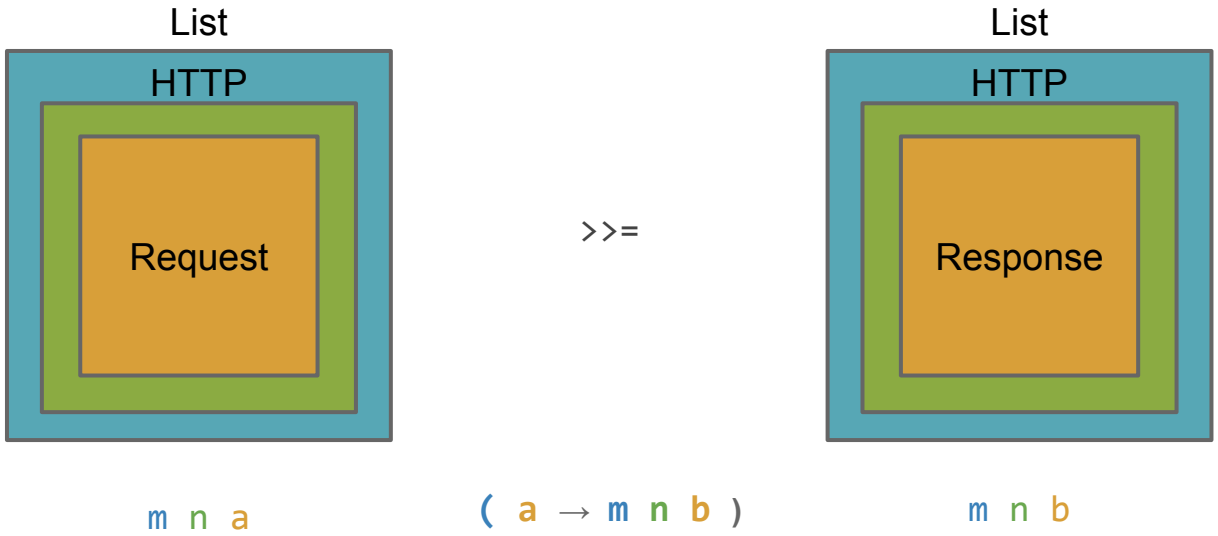




CSE230 Wi14 - Monad Transformers

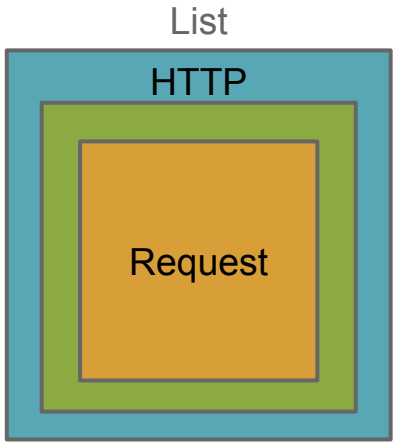
Context: Monad Transformer

How to stockpile two or more Monads to do various things?



Context: Monad Transformer

This require a Monad Transformer to 'bind' two kinds of Monads



HTTPT (Transformer)

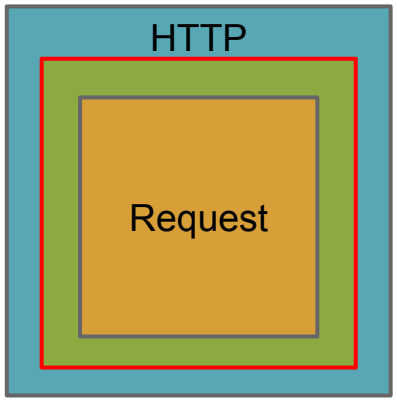
>>=

Context: Monad Transformer

This require a Monad Transformer to 'bind' two kinds of Monads

HTTPT ("T"ransformer) would not care what the outer monad is

~~List~~ m



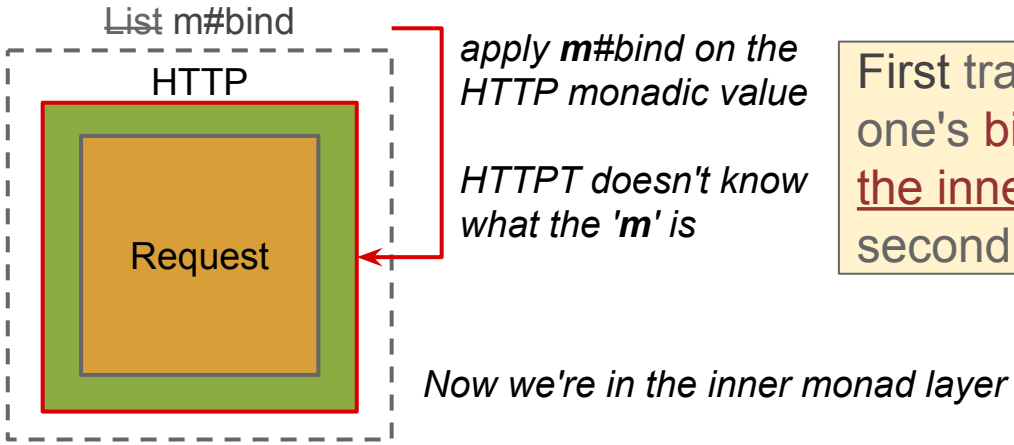
HTTPT (Transformer)

>>=

The transformer only know the inner one is the specific Monad it can handle with
ex: (MaybeT → Maybe, HTTPT → HTTP)

Context: Monad Transformer

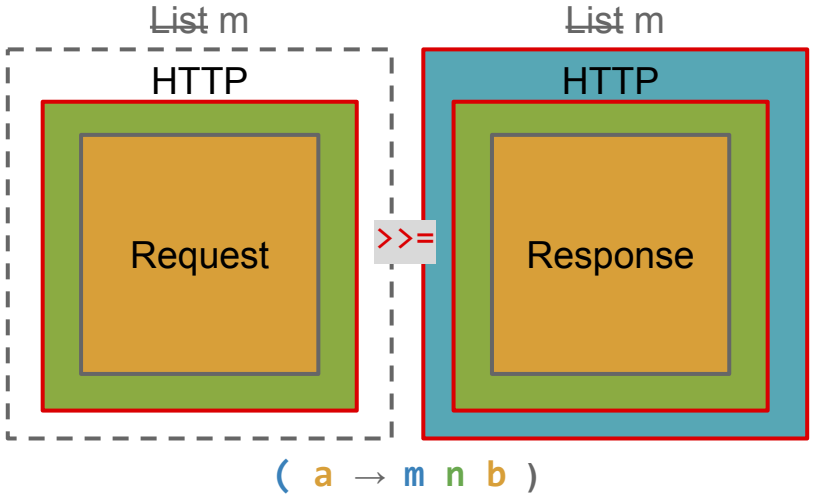
The bind function in a Monad Transformer would:



First transformer would call the outer one's bind function to apply the rule on the inner monadic value, and dig into the second layer (inner monad)

Context: Monad Transformer

The bind function in a Monad Transformer would:



Then transformer apply the specific Monad's binding rules on the inner monadic value, including to call the embedded function, just like what the ordinary Monad does, but now we get (m n b) rather than (m b)

http://en.wikibooks.org/wiki/Haskell/Monad_transformers#A_simple_monad_transformer:_MaybeT

Context: Monad Transformer

A not so successful try: PromiseMaybeT

Now it can stockpile arbitrary PromiseMonad on one PromiseMaybe monadic action with another

But since our basic 'Monad' is tricky, the transformer, is tricky, too

https://github.com/snowmantw/warmfuzzything.js/blob/master/maybet.js

After all...

Is it worth to make 'real' Monad in JavaScript?

It's worth if you...

- Already use libraries with fluent interface like jQuery, lo-dash or even the native Promise

- Care about how to restrict some effects within a specific domain only, meanwhile keep the code reusable and flexible

The reasons makes playing with Monad become too expensive:

- Rather than to see what effects Monad can bring to us, focusing on type and syntax issues too much, while this language indeed lacks some important features for Monad

- Need to implement several basic components in the Monad form

Part III

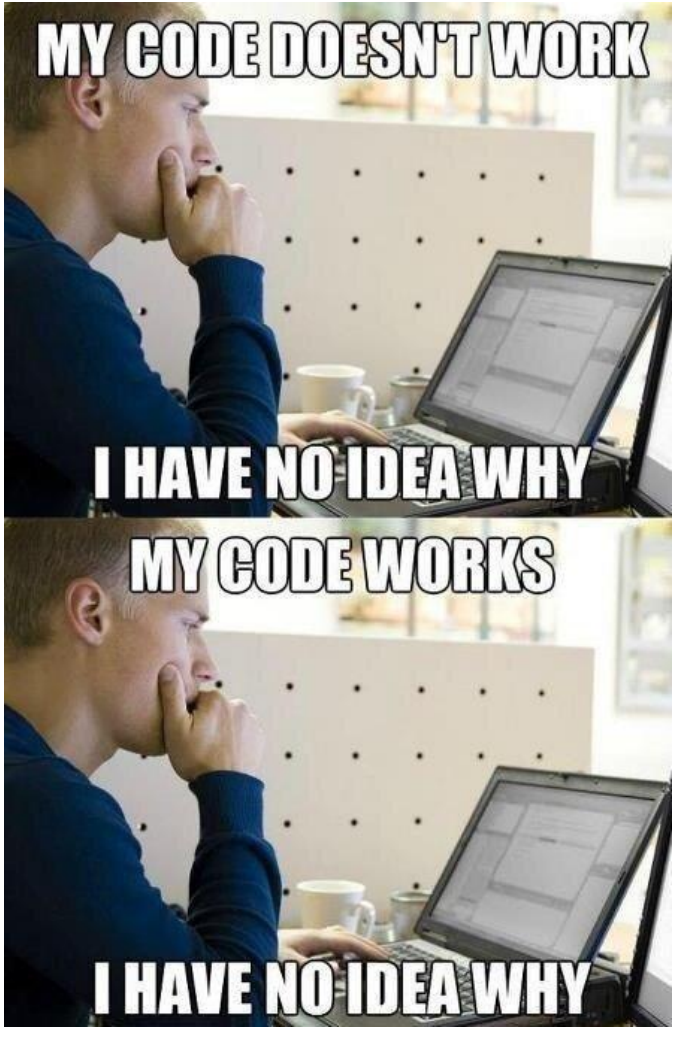
Purity

Purity

Nobody like surprises

Purity

Nobody like surprises



Purity

In theory, purity means no side-effects

Purity

In theory, purity means no side-effects

```
putStrLn:: String → IO ()
```

Purity

In theory, purity means no side-effects

```
putStrLn:: String → IO ()

-- the truth is...

putStrLn:: String → RealWorld → ((), RealWorld')
```

Purity

In JavaScript we needn't play such magic...

Purity

In JavaScript we needn't play such magic...

But to isolate different effects and compose them together rationally is still good for us

```
290  addNotification: function ns_addNotification(detail) {
291    // LockScreen window may not opened while this singleton got initialized.
292    this.lockScreenContainer = this.lockScreenContainer ||
293      document.getElementById('notifications-lockscreen-container');
294    var notificationNode = document.createElement('div');
295    notificationNode.classList.add('notification');
296    notificationNode.setAttribute('role', 'link');
297
298    notificationNode.dataset.notificationId = detail.id;
299    notificationNode.dataset.obsoleteAPI = 'false';
300    if (typeof detail.id === 'string' &&
301        detail.id.indexOf('app-notif-') === 0) {
302      notificationNode.dataset.obsoleteAPI = 'true';
303    }
304    var type = detail.type || 'desktop-notification';
305    notificationNode.dataset.type = type;
306    var manifestURL = detail.manifestURL || '';
307    notificationNode.dataset.manifestURL = manifestURL;
308
309    if (detail.icon) {
310      var icon = document.createElement('img');
311      icon.src = detail.icon;
312      icon.setAttribute('role', 'presentation');
313      notificationNode.appendChild(icon);
314    }
315
316    var dir = (detail.bidi === 'ltr' ||
317               detail.bidi === 'rtl') ?
318          detail.bidi : 'auto';
319
320    var titleContainer = document.createElement('div');
321    titleContainer.classList.add('title-container');
322    titleContainer.lang = detail.lang;
323    titleContainer.dir = dir;
324
325    var title = document.createElement('div');
326    title.classList.add('title');
327    title.textContent = detail.title;
328    title.lang = detail.lang;
329    title.dir = dir;
330    titleContainer.appendChild(title);
331
332    var time = document.createElement('span');
333    var timestamp = detail.timestamp ? new Date(detail.timestamp) : new Date();
```

from Gaia/System/notifications.js (v2.0)

line: 290 ~ 490 (200 lines)

```
290  addNotification: function ns_addNotification(detail) {
291    // LockScreen window may not opened while this singleton got initialized.
292    this.lockScreenContainer = this.lockScreenContainer ||
293      document.getElementById('notifications-lockscreen-container');
294    var notificationNode = document.createElement('div');
295    notificationNode.classList.add('notification');
296    notificationNode.setAttribute('role', 'link');
297
298    notificationNode.dataset.notificationId = detail.id;
299    notificationNode.dataset.obsoleteAPI = 'false';
300    if (typeof detail.id === 'string' &&
301        detail.id.indexOf('app-notif-') === 0) {
302      notificationNode.dataset.obsoleteAPI = 'true';
303    }
304    var type = detail.type || 'desktop-notification';
305    notificationNode.dataset.type = type;
306    var manifestURL = detail.manifestURL || '';
307    notificationNode.dataset.manifestURL = manifestURL;
308
309    if (detail.icon) {
310      var icon = document.createElement('img');
311      icon.src = detail.icon;
312      icon.setAttribute('role', 'presentation');
313      notificationNode.appendChild(icon);
314    }
315
316    var dir = (detail.bidi === 'ltr' ||
317               detail.bidi === 'rtl') ?
318          detail.bidi : 'auto';
319
320    var titleContainer = document.createElement('div');
321    titleContainer.classList.add('title-container');
322    titleContainer.lang = detail.lang;
323    titleContainer.dir = dir;
324
325    var title = document.createElement('div');
326    title.classList.add('title');
327    title.textContent = detail.title;
328    title.lang = detail.lang;
329    title.dir = dir;
330    titleContainer.appendChild(title);
331
332    var time = document.createElement('span');
333    var timestamp = detail.timestamp ? new Date(detail.timestamp) : new Date();
```

from Gaia/System/notifications.js (v2.0)

line: 290 ~ 490 (200 lines)

- Create notification
- Detecting gesture
- Append notification
- Play sound
- Color one container
- Scroll container to top

...

```
290  addNotification: function ns_addNotification(detail) {
291    // LockScreen window may not opened while this singleton got initialized.
292    this.lockScreenContainer = this.lockScreenContainer ||
293      document.getElementById('notifications-lockscreen-container');
294    var notificationNode = document.createElement('div');
295    notificationNode.classList.add('notification');
296    notificationNode.setAttribute('role', 'link');
297
298    notificationNode.dataset.notificationId = detail.id;
299    notificationNode.dataset.obsoleteAPI = 'false';
300    if (typeof detail.id === 'string' &&
301        detail.id.indexOf('app-notif-') === 0) {
302      notificationNode.dataset.obsoleteAPI = 'true';
303    }
304    var type = detail.type || 'desktop-notification';
305    notificationNode.dataset.type = type;
306    var manifestURL = detail.manifestURL || '';
307    notificationNode.dataset.manifestURL = manifestURL;
308
309    if (detail.icon) {
310      var icon = document.createElement('img');
311      icon.src = detail.icon;
312      icon.setAttribute('role', 'presentation');
313      notificationNode.appendChild(icon);
314    }
315
316    var dir = (detail.bidi === 'ltr' ||
317               detail.bidi === 'rtl') ?
318          detail.bidi : 'auto';
319
320    var titleContainer = document.createElement('div');
321    titleContainer.classList.add('title-container');
322    titleContainer.lang = detail.lang;
323    titleContainer.dir = dir;
324
325    var title = document.createElement('div');
326    title.classList.add('title');
327    title.textContent = detail.title;
328    title.lang = detail.lang;
329    title.dir = dir;
330    titleContainer.appendChild(title);
331
332    var time = document.createElement('span');
333    var timestamp = detail.timestamp ? new Date(detail.timestamp) : new Date();
```

from Gaia/System/notifications.js (v2.0)

line: 290 ~ 490 (200 lines)

There are so many

requests

from so many different

contexts

```
290  addNotification: function ns_addNotification(detail) {
291    // LockScreen window may not opened while this singleton got initialized.
292    this.lockScreenContainer = this.lockScreenContainer ||
293      document.getElementById('notifications-lockscreen-container');
294    var notificationNode = document.createElement('div');
295    notificationNode.classList.add('notification');
296    notificationNode.setAttribute('role', 'link');
297
298    notificationNode.dataset.notificationId = detail.id;
299    notificationNode.dataset.obsoleteAPI = 'false';
300    if (typeof detail.id === 'string' &&
301        detail.id.indexOf('app-notif-') === 0) {
302      notificationNode.dataset.obsoleteAPI = 'true';
303    }
304    var type = detail.type || 'desktop-notification';
305    notificationNode.dataset.type = type;
306    var manifestURL = detail.manifestURL || '';
307    notificationNode.dataset.manifestURL = manifestURL;
308
309    if (detail.icon) {
310      var icon = document.createElement('img');
311      icon.src = detail.icon;
312      icon.setAttribute('role', 'presentation');
313      notificationNode.appendChild(icon);
314    }
315
316    var dir = (detail.bidi === 'ltr' ||
317               detail.bidi === 'rtl') ?
318          detail.bidi : 'auto';
319
320    var titleContainer = document.createElement('div');
321    titleContainer.classList.add('title-container');
322    titleContainer.lang = detail.lang;
323    titleContainer.dir = dir;
324
325    var title = document.createElement('div');
326    title.classList.add('title');
327    title.textContent = detail.title;
328    title.lang = detail.lang;
329    title.dir = dir;
330    titleContainer.appendChild(title);
331
332    var time = document.createElement('span');
333    var timestamp = detail.timestamp ? new Date(detail.timestamp) : new Date();
```

from Gaia/System/notifications.js (v2.0)

line: 290 ~ 490 (200 lines)

Even without any FP ideas,

to response these requests

with individual logic units is

trivial and reasonable

DOM
Create notification
Change container's style

UI
Append notification
Scroll container to top

Gesture
Detect gesture on notification

Sound
Play sound

Asynchronous
Manage asynchronous operations

Conditional Statements
If...else to do or not to do things

I/O
Get/write data and control device

...

High-order Function

Partial Application

Curry

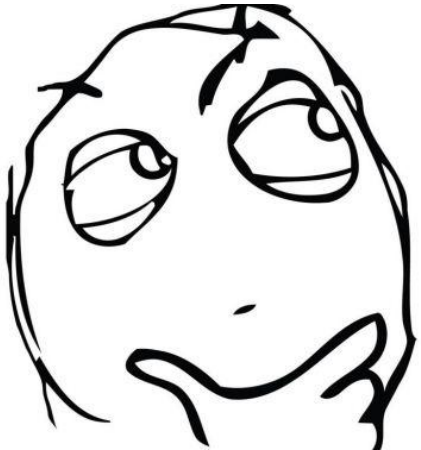
Functor

Monad

Monad Transformer

...

It looks like FP concerns data level only...?



"But 80% of my work is for UI changes"

That's why

React + Flux rocks

Let's think about what is an GUI program...

Behavior trigger event

Data changed

View redraw

Let's think about what is an GUI program...

Behavior trigger event

Data changed

View redraw

It's so simple, right?

Let's think about what is an GUI program...

Behavior trigger event

Data changed

View redraw

Let's think about what is an GUI program...

Behavior trigger event

Data changed

Can be done purely, while
IO is relatively simple than drawing

View redraw

Lots of side-effects
(The nature of DOM APIs)

problem?

With React we only care about data changes

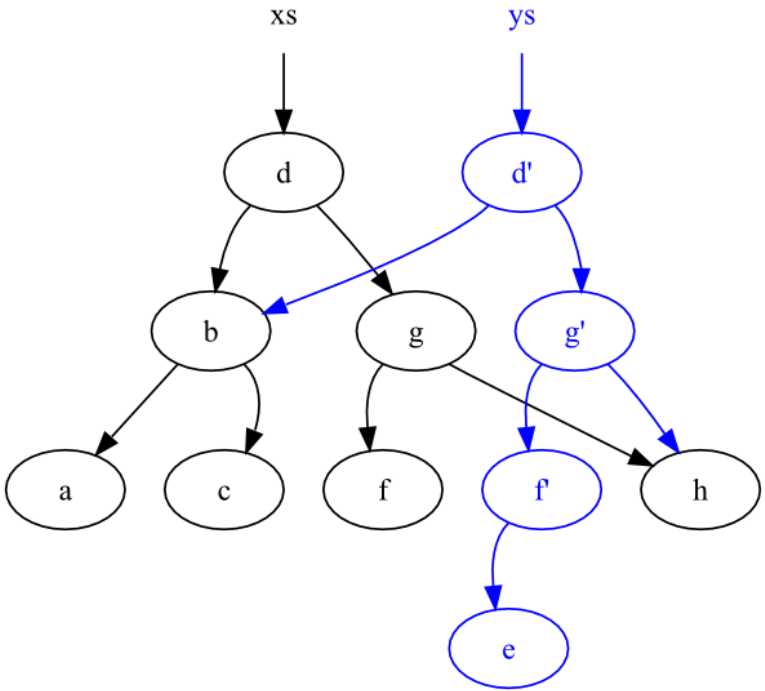
And 'create' a new view every time it get changed

And 'create' a new view every time it get changed

and efficiency is what React should care about

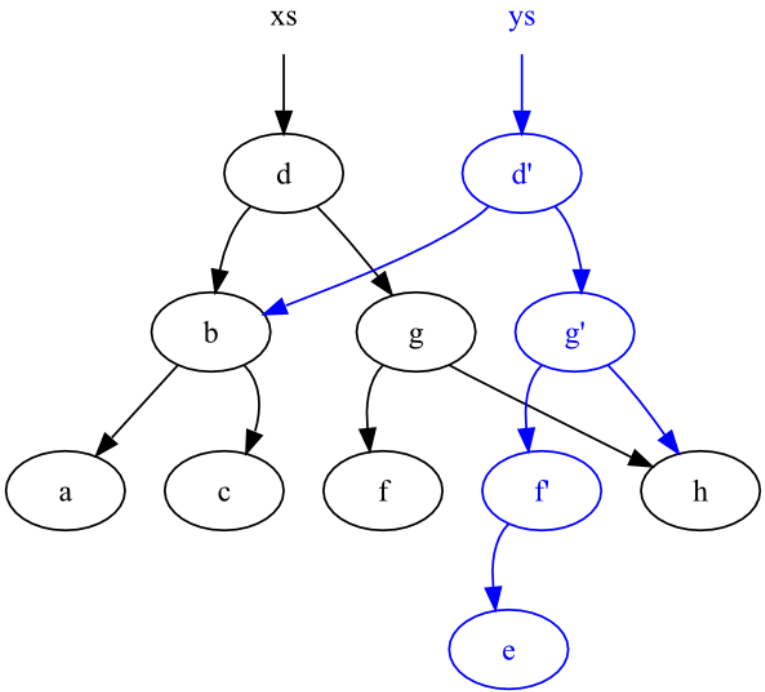
This is like what the immutable data structure become mutable in FP...

```
ys = insert ("e", xs)
```

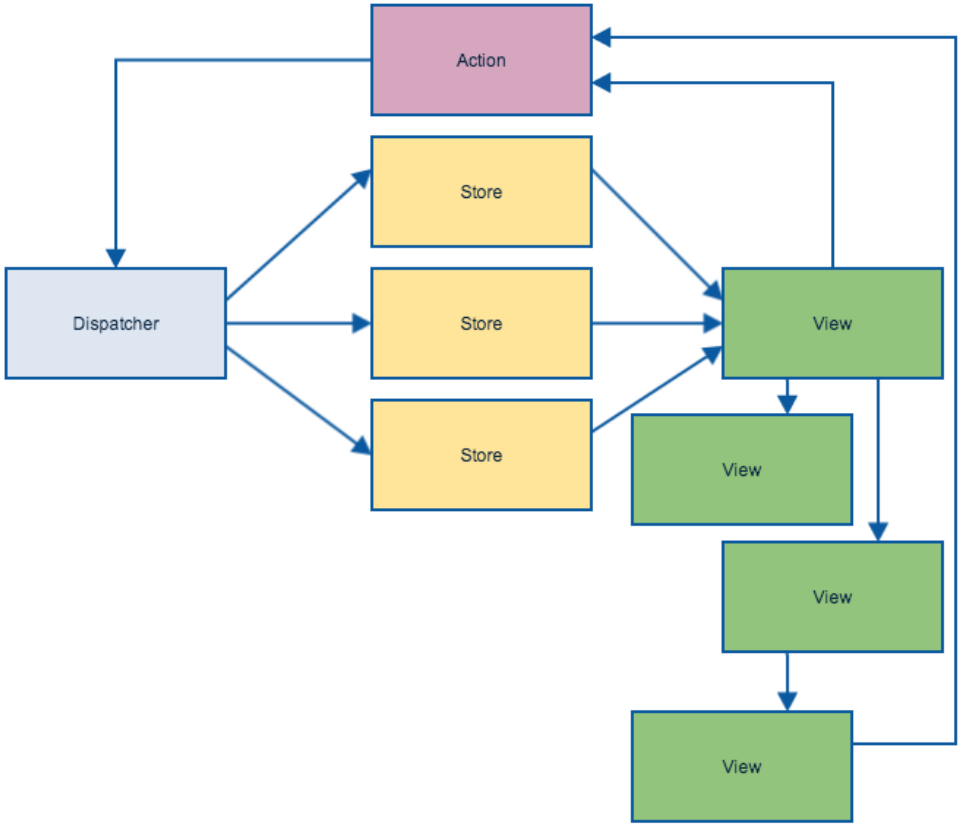


This is like what the immutable data structure become mutable in FP...

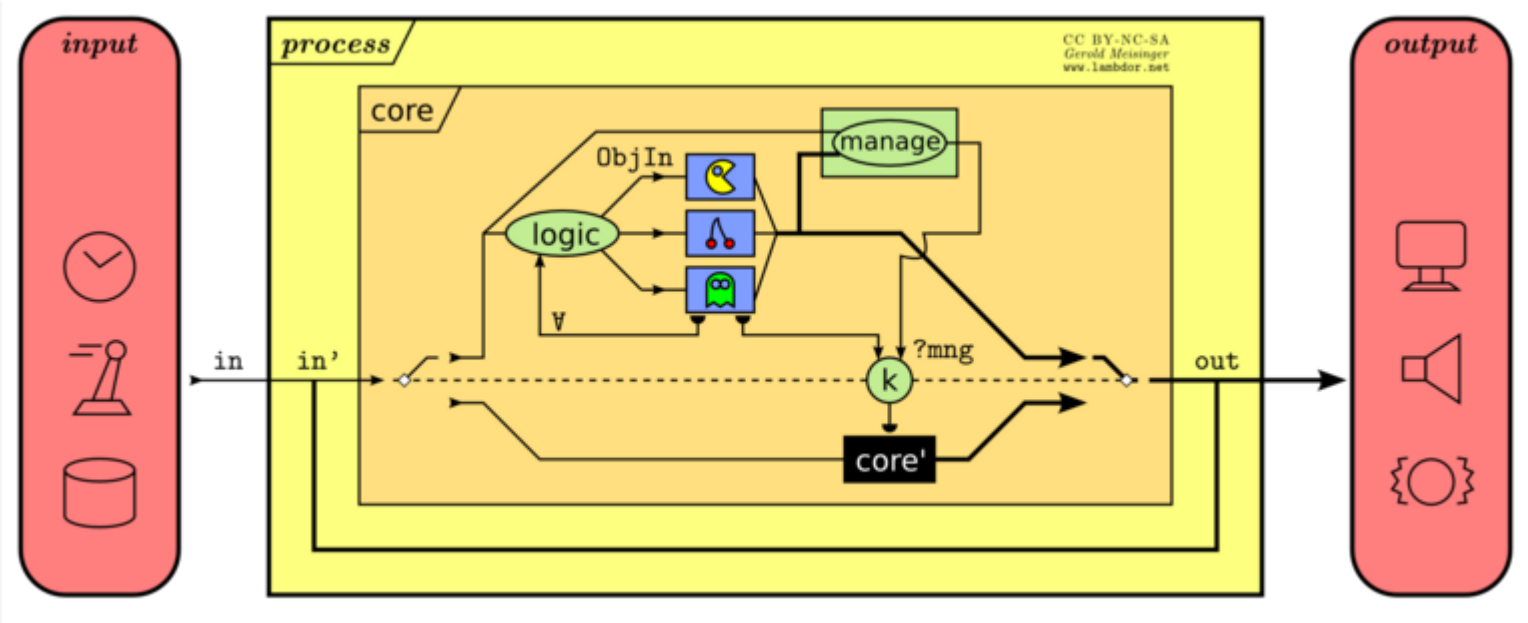
```
view' = render(..)
```



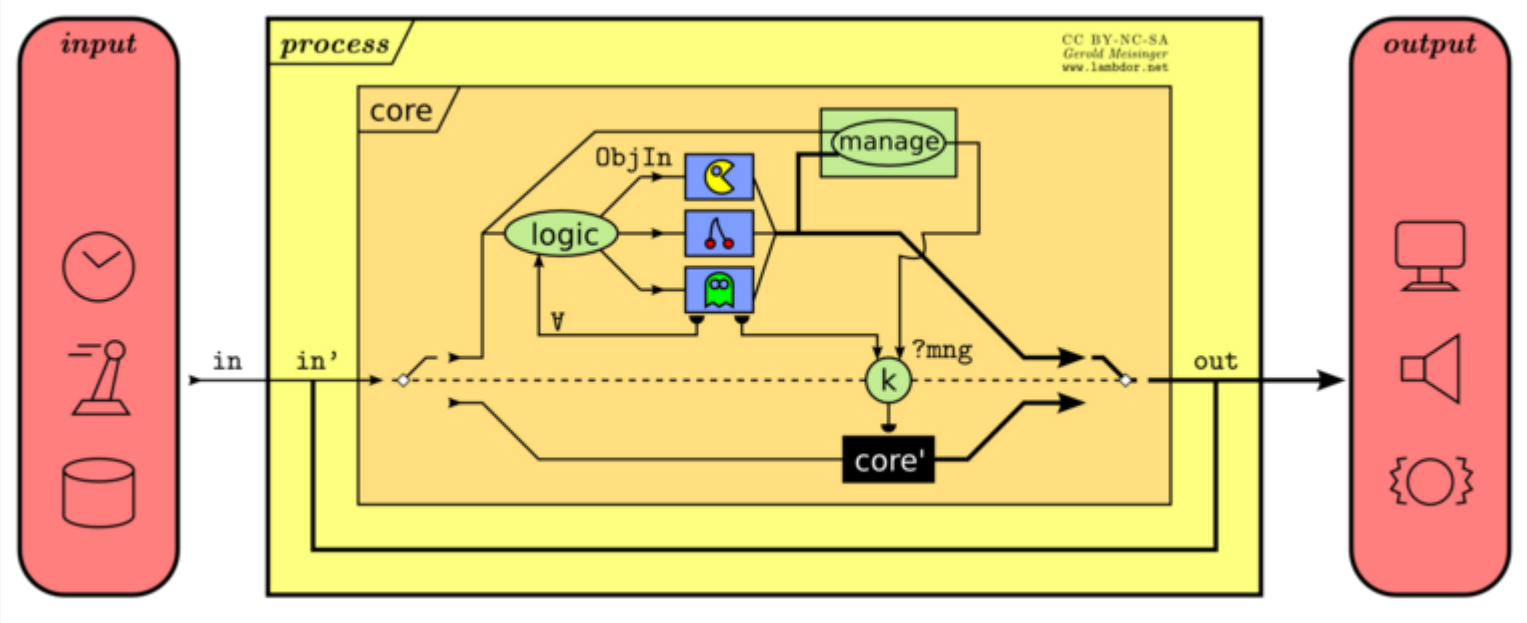
And this is what Flux looks like...



And this is what Flux looks like...



And this is what Flux looks like...



Yampa, *a Functional Reactive Programming framework in Haskell*

So React & Flux is really close to FP...

So React & Flux is really close to FP...

It's great because we can build a full Functional Programming stack on it

So React & Flux is really close to FP...

It's great because we can build a full Functional Programming stack on it

...with high-order function, partial application, functor, Monad, Monad Transformer, etc.

Conclusion

Conclusion

- Use High-order Functions as much as possible, including to define and customize it

Conclusion

- Use High-order Functions as much as possible, including to define and customize it
- Use fluent interface as much as possible: use it to enforce computation focusing on special domains, and benefits from isolation and managed side-effects

Conclusion

- Use High-order Functions as much as possible, including to define and customize it
- Use fluent interface as much as possible: use it to enforce computation focusing on special domains, to benefit from isolation and managed side-effects
- If libraries are ready, consider to use Partial Application or Currying

Conclusion

- Use High-order Functions as much as possible, including to define and customize it
- Use fluent interface as much as possible: use it to enforce computation focusing on special domains, to benefit from isolation and managed side-effects
- If libraries are ready, consider to use Partial Application or Currying
- If it's possible, consider making fluent interface more monadic, including laziness, purity, etc.

Conclusion: library & framework

- lo-dash is your friend
- transducer in JavaScript is a good way to understand reducing deeply
- immutable-js make your code purer
- React & Flux bring you a whole new FRP world

Conclusion: library & framework

- lo-dash is your friend
- transducer in JavaScript is a good way to understand reducing deeply
- immutable-js make your code purer
- React & Flux bring you a whole new FRP world

In fact there are lots of 'functional' related libraries...

Thanks and
Q & A