How I Learned to Stop Worrying and Love the Functor

Runes

```
// Optional:
public func <^><T, U>(f: T -> U, a: T?) -> U?
public func <*><T, U>(f: (T -> U)?, a: T?) -> U?
public func >>-<T, U>(a: T?, f: T -> U?) -> U?
public func pure<T>(a: T) -> T?

// Array:
public func <^><T, U>(f: T -> U, a: [T]) -> [U]
public func <*><T, U>(fs: [T -> U], a: [T]) -> [U]
public func >>-<T, U>(a: [T], f: T -> [U]) -> [U]
public func pure<T>(a: T) -> [T]
```

Argo

```
extension User: Decodable {
  static func decode(j: JSON) -> Decoded<User> {
    return User.create
      <^> j <| "id"
      <*> j <| "first_name"
      <*> j <| "last_name"
      <*> j <|? "email"
  }
}
```


#newgirl

(Re-)Introducing: `map`

Array.map

In the Bad Old Days

```
NSArray *numbers = @[@1, @2, @3, @4];
NSMutableArray *mutable = [NSMutableArray array];

for (NSNumber *number in numbers) {
    NSNumber *newNumber = @([number integerValue] + 1);
    [mutable addObject: newNumber];
}

NSArray *numbersPlusOne = [mutable copy];
```

```
extension Array<T> {
  func map<U>(transform: T -> U) -> [U] {
    var array: [U] = []

    for x in self {
      let y = transform(x)
      array.append(y)
    }

    return array
  }
}
```

Usage

```
let numbers = [1, 2, 3, 4]
let numbersPlusOne = numbers.map { $0 + 1 } // [2, 3, 4, 5]
```

Bonus

```
func addOne(x: Int) -> Int {
  return x + 1
}

let numbers = [1, 2, 3, 4]
let moreNumbers = numbers.map(addOne) // [2, 3, 4, 5]
```

Real World Usage

```
struct User {
  let name: String
  let email: String
}

let mcNulty = User(name: "Jimmy McNulty", email: "mcnulty@bpd.gov")
let theBunk = User(name: "Bunk Moreland", email: "thebunk@bpd.gov")

let users = [mcNulty, theBunk]
let emails = users.map { $0.email }

sendEmails(emails)
```

Optional.map

Optional

```
enum Optional<T> {
   case .Some(T)
   case .None
}
```

Conditional Unwrapping

```
let myOptional: String? = "Hello World"

if let string = myOptional {
  println(string)
}
```

Optional.map

```
extension Optional<T> {
  func map<U>(transform: T -> U) -> U? {
    if let x = self {
      return .Some(transform(x))
    } else {
      return .None
    }
  }
}
```

Usage

```
let optional: String? = "foo"
let foobar = optional.map { $0 + "bar" } // .Some("foobar")

let otherOptional: String? = .None
let nope = optional.map { $0 + "bar" } // .None
```

Usage

```
func appendBar(x: String) -> String {
  return x + "bar"
}

let optional: String? = "foo"
let moreFoobar = optional.map(appendBar) // .Some("foobar")
```

Real World Usage

```
let name = json["name"] // String?
let user = name.map { User(name: $0) } // User?
let company = user.map { Company(employees: [$0]) } // Company?

let numberOfEmployees = company?.numberOfEmployees ?? 0 // Int
```

Spot the Difference

```
// Array
func map<T, U>(x: [T], f: T -> U) -> [U]

// Optional
func map<T, U>(x: T?, f: T -> U) -> U?
```

Spot the Difference

```
// Array
func map<T, U>(x: Array<T>, f: T -> U) -> Array<U>

// Optional
func map<T, U>(x: Optional<T>, f: T -> U) -> Optional<U>
```

Contextual Types

```
func map<T, U>(x:    Array<T>, f: T -> U) ->    Array<U>
func map<T, U>(x: Optional<T>, f: T -> U) -> Optional<U>
```

Contextual Types

```
func map<T, U>(x:    Array<T>, f: T -> U) ->    Array<U>
func map<T, U>(x: Optional<T>, f: T -> U) -> Optional<U>
func map<T, U>(x:   Result<T>, f: T -> U) ->   Result<U>
```

Contextual Types

```
func map<T, U>(x:    Array<T>, f: T -> U) ->    Array<U>
func map<T, U>(x: Optional<T>, f: T -> U) -> Optional<U>
func map<T, U>(x:   Result<T>, f: T -> U) ->   Result<U>
func map<T, U>(x:   Future<T>, f: T -> U) ->   Future<U>
```

Contextual Types

```
func map<T, U>(x:    Array<T>, f: T -> U) ->    Array<U>
func map<T, U>(x: Optional<T>, f: T -> U) -> Optional<U>
func map<T, U>(x:   Result<T>, f: T -> U) ->   Result<U>
func map<T, U>(x:   Future<T>, f: T -> U) ->   Future<U>
func map<T, U>(x:   Signal<T>, f: T -> U) ->   Signal<U>
```


COFFEE
MR. RADAR
Everybody got that?

What if...[1]

```
protocol Context<T> {
  func map<U>(transform: T -> U) -> Self<U>
}

func map<C: Context, T, U>(x: C<T>, transform: T -> U) -> C<U> {
  return x.map(transform)
}
```

[1] rdar://problem/18575907

Hello {{NAME}}

```
func sayHello(names: Array<String>) -> Array<String> {
  return names.map { "Hello \($0)!" }
}
```

Hello {{NAME}}

```
func sayHello(name: Optional<String>) -> Optional<String> {
  return name.map { "Hello \($0)!" }
}
```

Hello {{NAME}}

```
func sayHello(name: Result<String>) -> Result<String> {
  return name.map { "Hello \($0)!" }
}
```

Hello {{NAME}}

```
func sayHello(name: Future<String>) -> Future<String> {
  return name.map { "Hello \($0)!" }
}
```

Hello {{NAME}}

```
func sayHello(name: Signal<String>) -> Signal<String> {
  return name.map { "Hello \($0)!" }
}
```

Hello {{NAME}}

```
func sayHello<C: Context>(name: C<String>) -> C<String> {
  return name.map { "Hello \($0)!" }
}
```

What if...[1]

```
protocol Context<T> {
  func map<U>(transform: T -> U) -> Self<U>
}

func map<C: Context, T, U>(x: C<T>, transform: T -> U) -> C<U> {
  return x.map(transform)
}
```

[1] rdar://problem/18575907

Turns out...[1]

```
protocol Functor<T> {
  func map<U>(transform: T -> U) -> Self<U>
}

func map<F: Functor, T, U>(x: F<T>, transform: T -> U) -> F<U> {
  return x.map(transform)
}
```

[1] rdar://problem/18575907

Gordon Fontenot

@gfontenot
thoughtbot.com
buildphase.fm