Better Living Through Functional Programming

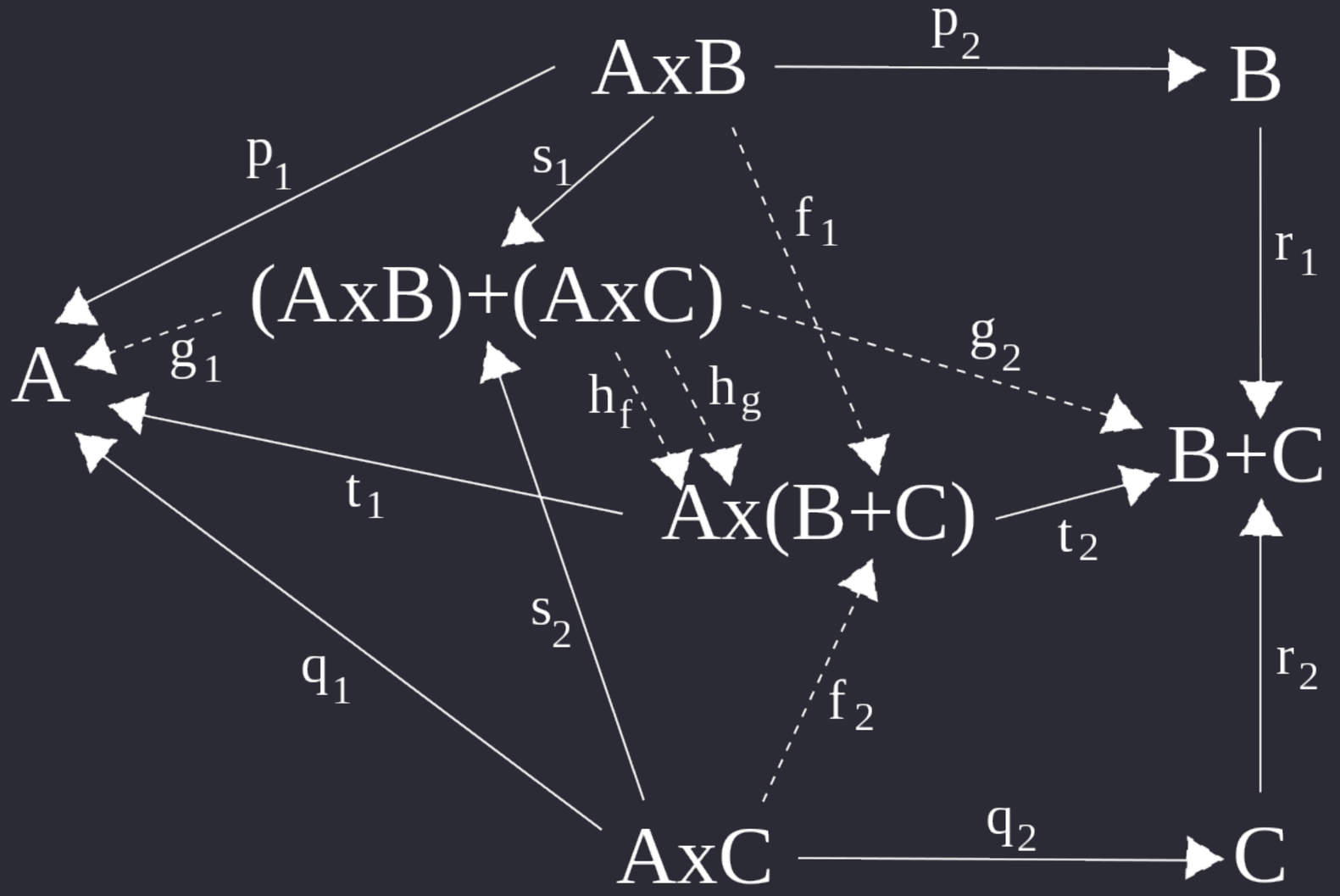
In a language you'll actually use

slalom

Goals

- Introduce various functional programing tools
- Explain why they can make your life better while programming
- Convince you to try some of these tools out

Not This


AxB
p2
B
p1
s1
f1
r1
(AxB)+(AxC)
A
g1
g2
hf
hg
B+C
t1
Ax(B+C)
t2
s2
q1
f2
r2
AxC
q2
C

Tools

Basic

- Functions
- Map, Filter, Reduce
- Immutability

Less Basic

- Currying & Partial Application
- Algebraic Data Types
- Monads

Pure Functions

“The competent programmer is fully aware of the strictly limited size of his own skull.”

– Edsger W. Dijkstra

Impure

```
function canLegallyDrink(person){
 if(person.age >= 21) {
  person.permissions.push('canLegallyDrink');
 }
}
```

Pure

```
function canLegallyDrink(age){
  return age > 21
}

...

person.canLegallyDrink = canLegallyDrink(person.age);
```

Testing

```
function canLegallyDrink(person){
  if(person.age >= 21) {
    person.permissions.push('canLegallyDrink');
  }
}

it('should correctly calculate drinking age', () => {
  const person = {age: 21};
  canLegallyDrink(person);
  expect(person.permissions).toBe(['canLegallyDrink']);
});
```

Testing

```
function canLegallyDrink(age){
  return age > 21
}

it('should correctly calculate drinking age', () => {
  expect(canLegallyDrink(21)).toBe(true);
});
```

First Class Functions

First Class Functions

```
function handleResult(result){
  values.push(result);
}

function handleError(error){
  console.log(error);
}

response
  .then(handleResult)
  .catch(handleError);
```

Higher Order Functions

Higher Order Functions

```
function errorHandlerBuilder(logLevel){
 if (logLevel === 'debug'){
  return (error) => {
   console.log(error.message);
   console.log(error.stacktrace);
  }
 }
 return (error) => console.log(error.message);
}

response.then(handleResult)
 .catch(errorHandlerBuilder('debug'));
```

Java 8 Functional Interfaces

Java 8 Functional Interfaces

```
class FunctionalInterfaces{
  public static void main(String[] args){
    Function<T, R> function = (t) -> r;
    BiFunction<A, B, R> bifunction = (a, b) -> r;
    Supplier<T> supplier = () -> t;
    Consumer<T> consumer = (t) -> void;
    Predicate<T> predicate = (t) -> bool;
  }
}
```

Map, Filter, and Reduce

Map

```
const badNames = [
  'tEsT-FILE.js',
  'OTHER file.js',
  'I-hope_YOU NEVER-do.this.js'
  ]

const betterNames = badNames.map(_.camelCase);

/* [
  'testFile.js',
  'otherFile.js',
  'iHopeYouNeverDoThis.js']
*/
```

Map

```
var new_array = arr.map(function
callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

FlatMap

```
const values = [
  [1],
  [3, 2, 4],
  [6, 9, 3],
]

values.map(val => val + 1)
//  [2],[4, 3, 5],[7, 10, 4]]

_.flatMap(values, val => val + 1)
//  [2, 4, 3, 5, 7, 10, 4]
```

Filter

```
const statusCodes = [200, 201, 204, 304,
 400, 403, 500, 502]
const failureStatusCodes =
 statusCodes.filter(code => code >= 400)

// [400, 403, 500, 502]
```

Filter

```
const predicate (val) => boolean
const newArray = arr.filter(predicate)
```

Reduce

```
const books = [
  {title: 'The Joy of Clojure', length: 520},
  {title: 'Elixir in Action', length: 376},
  {title: 'Programming Rust', length: 633},
  ]

const totalPages = books.reduce((acc, val) => {
  return acc + val.length
}, 0)
```

Reduce

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => {
  return accumulator + currentValue;
}

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10
```

Map with Reduce

```
function map(arr, fn) {
  arr.reduce((acc, val) => {
    acc.push(fn(val));
  }, []);
}
```

Filter with Reduce

```
function filter(arr, predicate) {
 arr.reduce((acc, val) => {
  if(predicate(val)) {
   acc.push(val);
  }
 }, [])
}
```

List to Map

```
const contributors = [
  {name: 'Rick', lang: 'Clojure', stars: '821'},
  {name: 'Evan', lang: 'Elm', stars: '549'},
  {name: 'Guido', lang: 'Python', stars: '1830'},
  ...
]

const contributorsMap = contributors
  .reduce((acc, val) => {
    acc[val.name] = val;
    return acc;
  }, {})
```

Benefits

Data Flow vs Control Flow

```
const contributors = [
  {name: 'Rick', lang: 'Clojure', stars: '821'},
  {name: 'Evan', lang: 'Elm', stars: '549'},
  ...
]

const functionalLanguages = [
  'Clojure', 'Haskell', 'OCaml', 'Elm', 'Elixir', ...
]
```

Data Flow vs Control Flow

```
let functionalStars = 0;
for(let i = 0; i < contributors.length; i++){
  const contributor = contributors[i];
  if (functionalLanguages.includes(contributors.lang)){
    functionalStars += contributor.stars;
  }
}
```

Data Flow vs Control Flow

```
const functionalStars = contributors
  .filter(c => functionalLanguages.includes(c.lang))
  .map(c => c.stars)
  .reduce((total, current) => total + current, 0);
```

Refactoring

```
function isAFunctionalLanguage(lang){
  return functionalLanguages.includes(lang);
}

Array.prototype.sum = function() {
  return this.reduce((acc, val) => acc + val, 0);
}
```

Refactoring

```
const functionalStars = contributors
  .filter(isAFunctionalLanguage)
  .map(c => c.stars)
  .sum();
```

Exposing Refactoring

```
let functionalStars = 0;
for(let i = 0; i < contributors.length; i++){
  const contributor = contributors[i];
  const detailedContributor = expensiveCall(contributor);
  if (functionalLanguages.includes(contributors.lang)){
    functionalStars += contributor.stars;
  }
}
```

Exposing Refactoring

```
let functionalStars = 0;
for(let i = 0; i < contributors.length; i++){
  const contributor = contributors[i];
  const detailedContributor = expensiveCall(contributor);
  if (functionalLanguages.includes(contributors.lang)){
    functionalStars += contributor.stars;
  }
}
```

Exposing Refactoring

```
const functionalStars = contributors
  .map(expensiveCall)
  .filter(isAFunctionalLanguage)
  .map(c => c.stars)
  .sum();
```

Exposing Refactoring

```
const functionalStars = contributors
  .map(expensiveCall)
  .filter(isAFunctionalLanguage)
  .map(c => c.stars)
  .sum();
```

Exposing Refactoring

```
const functionalStars = contributors
  .filter(isAFunctionalLanguage)
  .map(expensiveCall)
  .map(c => c.stars)
  .sum();
```

Java

```
List<Contributor> contributor = Arrays.asList(...);

contributor.stream()
  .filter(this::isAFunctionalLanguage)
  .map(this::expensiveCall)
  .map(Contributor::getStars)
  .sum();
```

Immutability

The Value of Values

- Referential Transparency
- Memoization
- Lower Cognitive Load
- No need for methods
- Easily Shared

Currying

Currying

```
const search = (country, state, store, item) => {
  ...
}

search('US', 'GA', '0121', 'hammer');

const search = (country) => (state) => (store) => (item) => {
  ...
}

search('US')('GA')('0121')('hammer');
```

Currying

```
var abc = function(a, b, c) {
  return [a, b, c];
};

var curried = _.curry(abc);

curried(1)(2)(3);
// => [1, 2, 3]

curried(1, 2)(3);
// => [1, 2, 3]

curried(1, 2, 3);
// => [1, 2, 3]
```

Partial Application

Partial Application

```
const search = (country, state, store, item) => {...}
const curriedSearch = _.curry(search);
const searchGeorgia = search('US', 'GA');

searchGeorgia('0121', 'hammer');

const searchAtlanta = search('US', 'GA', '0121')
const items = ['hammer', 'saw', 'awl'];

items.map(searchAtlanta);
```

Partial Application in Java

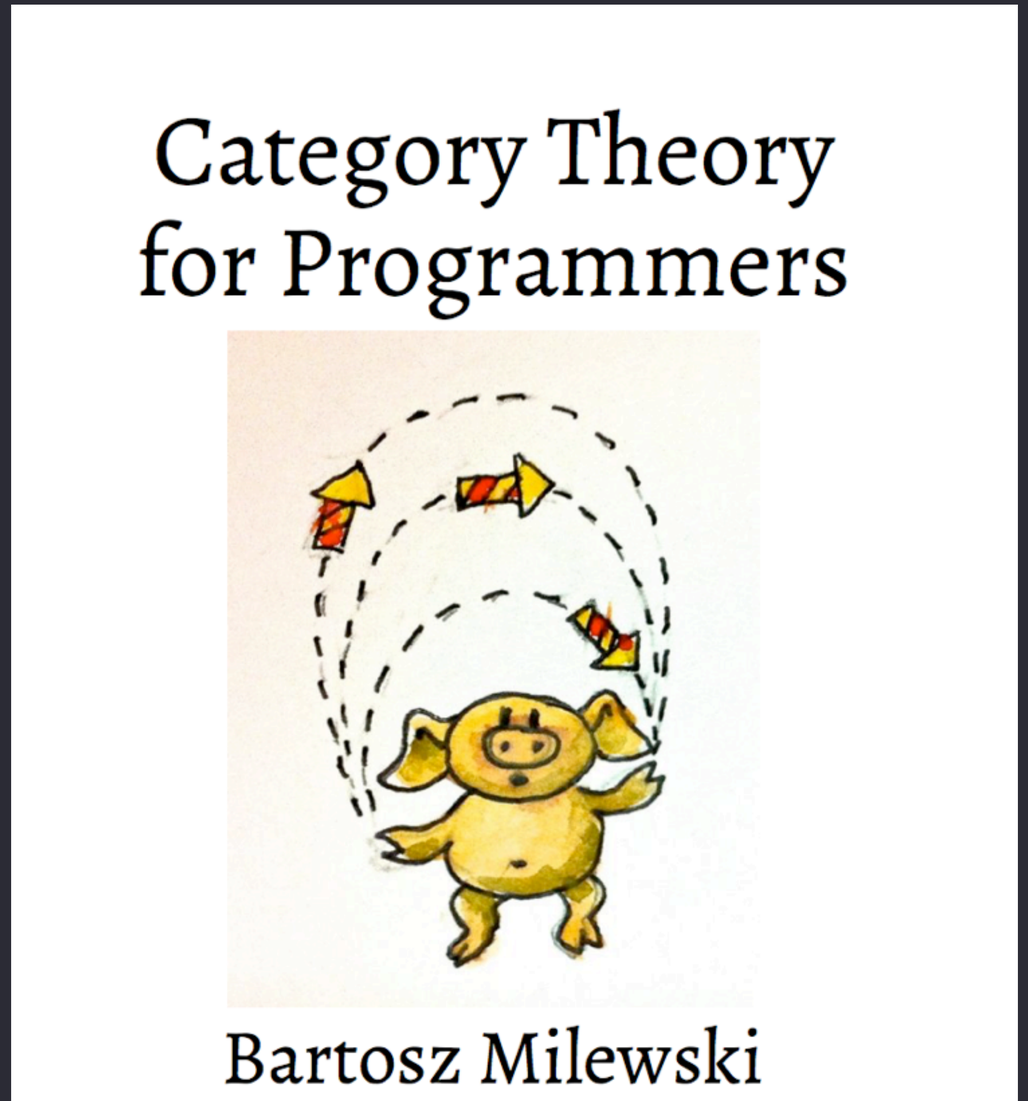
```
public class Example {
    public static int add(int x, int y) {
        return x + y;
    }

    public static  Function partial(BiFunction f, T x) {
        return (y) -> f.apply(x, y);
    }

    public static void main(String[] args) {
        Function adder = partial(Example::add, 5);
        System.out.println(adder.apply(2)); // 7
    }
}
```

Algebraic Data Types

In TypeScript

Algebraic Data Types

Just Set Algebra for Types



Product Types

```
interface Loggable { log(): void; }

interface Serializeable { serialized(): string; }

interface Person {
  name: string;
  age: number;
}

type LoggablePerson = Person & Loggable & Serializeable;

function foo(person: Person, loggablePerson: LoggablePerson){
  person.log() // invalid

  loggablePerson.log(); // valid
  loggablePerson.serialized() // also valid
}
```

Union Types

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
}

padLeft("Hello world", 4); // returns "    Hello world"
```

Union Types

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();    // errors
```

Union Types

```
let s = "foo";
s = null; // error, 'null' is not assignable to 'string'
let sn: string | null = "bar";
sn = null; // ok

sn = undefined; // error, 'undefined' is not assignable
to 'string | null'
```

Union Types

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
      // ...
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not
allowed here
```

Discriminated Unions

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}
```

Discriminated Unions

```
type Shape = Square | Rectangle | Circle;


function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

Discriminated Unions

```
type Action =
   { type: 'INCREMENT_COUNTER', delta: number }
|  { type: 'RESET_COUNTER' }

type Counter = { value : number }

function counter (state: Counter = { value: 0 }, action: Action): Counter {
  switch (action.type) {

    case 'INCREMENT_COUNTER':
      const { delta } = action
      return { value: state.value + delta }

    case 'RESET_COUNTER':
      return { value: 0 }

    default:
      return state
  }
}
```

Monads

“All told, a monad in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor..”

Questions?

“The Monad interface specializes in (and I’m simplifying here) pulling values out of their container types, operating on them, and putting them back into the same container. ”

Useful Monads

Stream

```
class Example {
  Integer getTotalAge(List<Family> families){
    return people.stream()
      .flatMap(Family::getMembers)
      .map(Person::age)
      .sum();
  }
}
```

Optionals

```
class Example {
  String getPostalCode(Person person){
    if(person != null){
      Address address = person.getAddress();
      if(address != null){
        String postalCode = address.getPostalCode();
        if(postalCode != null){
          return postalCode;
        }
      }
    }

    return "";
  }
}
```

Optionals

```
class Example {
  String getPostalCode(Person person){
    return Optional.ofNullable(person)
      .map(Person::getAddress)
      .map(Address::getPostalCode)
      .orElse("")
  }
}
```

CompletableFuture

```
class Example {
  String getMessage(){
    CompletableFuture<String> completableFuture =
      supplyAsync(() -> "Hello")
          .thenCompose(s -> supplyAsync(() -> s + " Async"))
          .thenApply(s -> s + " World");
  }
}
```

Want to learn more?

Learn a pure functional language

Questions?

Thank You

slalom