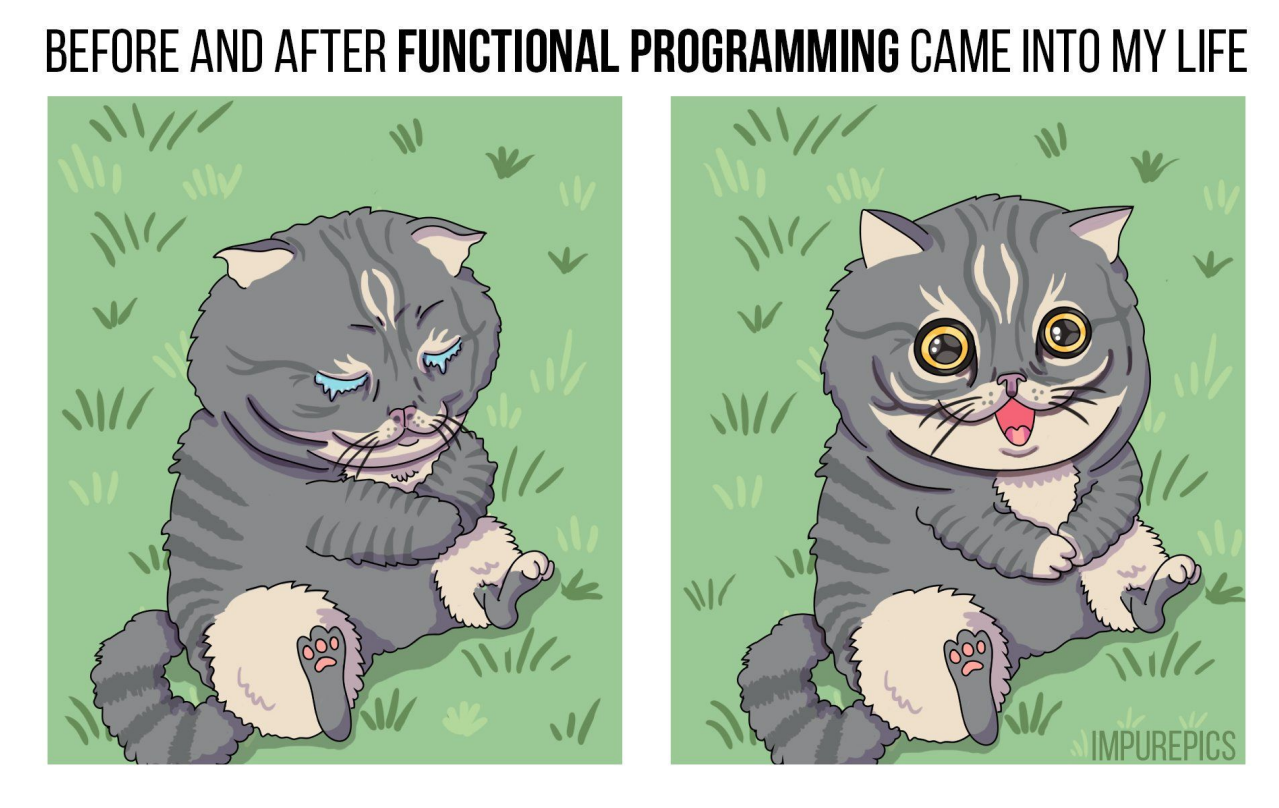
Функциональная обработка ошибок (и не только) на TypeScript

Или моноиды в категории эндофункторов
без страха и ужаса

Чего не будет в этом докладе?

- Жесткой теории категорий или супер-сложных подходов функционального программирования
- Голой теории(только Business Value)
- Требований наличия PhD у слушателей
- Чего то, что не может быть сделано на PHP, Python, JavaScript, TypeScript и почти любом другом языке
- Ссылок на конкретные реализации монад

Начнем же?


BEFORE AND AFTER FUNCTIONAL PROGRAMMING CAME INTO MY LIFE
IMPUREPICS

Начнем же!

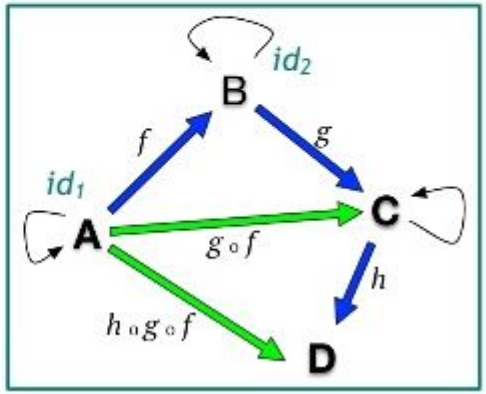
Category Laws

Associative Law

$(h \circ g) \circ f = h \circ (g \circ f)$

Identity Law

$f \circ id_1 = id_1 \circ f = f$



Учитель может летать...

Представим небольшое приложение

- Поиск информации о мастере
- Входит ли он в какую нибудь группу мастеров
- Знаем ли может он летать или нет

Хранение наших данных

```ts
type Master = {
  id: number;
  name: string;
  groupId?: number;
};

type MasterGroup = {
  id: number;
  name: string;
};

type MastersFlyInfo = {
  masterId: number;
  canFly: boolean;
};
```

```ts
export class MastersRepository {
  private masters: Master[] = [
    { id: 1, name: "First", groupId: 1 },
    { id: 2, name: "Second", groupId: 2 },
    { id: 3, name: "Some other" }
  ];

  findByName(name: string) {
    const result = this.masters.filter(m => m.name === name);

    return result.length ? result.pop() : null;
  }
}
```

Получение информации об учителе

```
const performSearch = (name?: string) => {
  var result = "Данные по учителю не найдены";

  if (name) {
    const master = mastersRepo.findByName(name);

    if (master) {
      const canFlyInfo = mastersFlyInfoRepo.findBymasterId(master.id);
      var masterGroup = null;

      if (master.groupId) {
        masterGroup = mastersGroupRepo.findById(master.groupId);
      }

      result = `Учитель ${master.name}`;
      if (masterGroup) {
        result += ` состоит в группе ${masterGroup.name}`;
      }

      if (canFlyInfo) {
        result += canFlyInfo.canFly ? ". Может летать" : ". Может не летать";
      } else {
        result += `. Не известно, может летать или нет.`;
      }
    }
  }

  return result;
};
```

Где же здесь можно стрельнуть себе по коленям?

- Пользователь может нажать на кнопку ничего не вводя
- Информации об указанном учителе у нас может и не быть
- Учитель может не состоять в группе
- Мы можем знать летает учитель или нет
- Ну а можем и не знать(может не быть информации в репозитории)

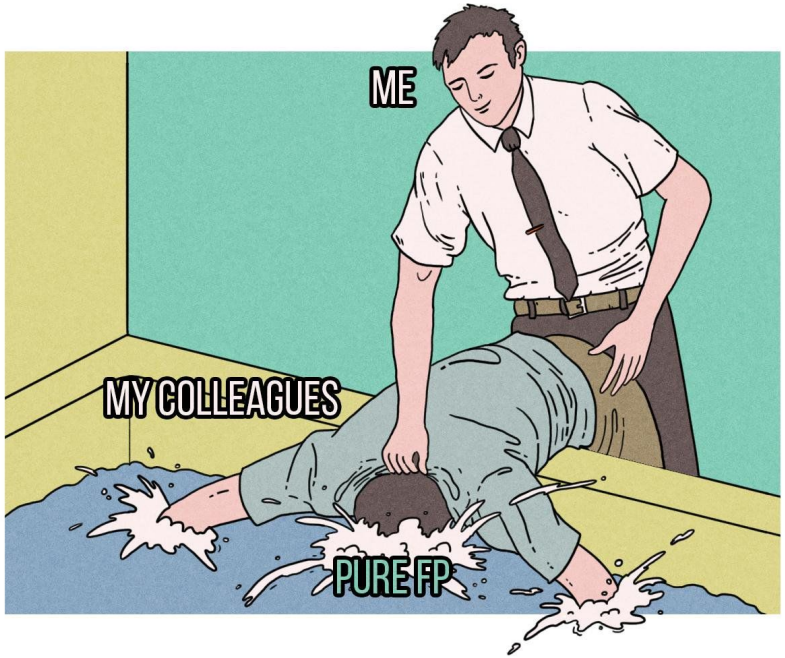
И таким образом

- На каждом этапе возможно отсутствие информации (вернется Null вместо модели)
- Нужно учесть все краевые случаи и написать множество конструкций if

Выход есть?

- Хочется, чтобы компилятор(если мы говорим про статически типизированные языки) нам не дал скомпилировать некорректный код
- Да и хочется чуть более простой записи кода

Посмотрим в сторону ФП



Встречайте тип данных “Maybe”

- Встречается во многих современных языках в стандартной поставке (Haskell, Scala, F#, ML, Java)
- Либо в виде библиотек(C++, TypeScript, C#, Java, PHP, Clojure, Python...)
- Легок в обращении
- Вызывает привыкание
- Желание рассказывать о монадах другим

В самом базовом виде его представим так:

```
export class Maybe<T> {
  private constructor(private value: T | null) {}

  static Just<T>(value: T) {
    if (!value) {
      throw Error("Value should be defined");
    }

    return new Maybe(value);
  }

  static Nothing<T>() {
    return new Maybe<T>(null);
  }
```

В самом базовом виде его представим так:

```
static Apply<T>(value: T) {
  return value ? Maybe.Just(value) : Maybe.Nothing<T>();
}
```

В самом базовом виде его представим так:

```
getOrElse(fallbackValue: T) {
  return this.value === null ? fallbackValue : this.value;
}
```

Наши репозитории немного изменятся

```ts
export type Master = {
  id: number;
  name: string;
  groupId: Maybe<number>;
};
```

```ts
findByName(name: string): Maybe<Master> {
  const result = this.masters.filter(m => m.name === name);

  return Maybe.Apply(result.length ? result.pop() : null);
}
```

Но может стоит сделать использование Maybe чуть более удобным?

- Все таки Maybe - это монада, а монада - это моноид, а моноид - это штука подобная множеству.
- А у массивов(*тоже подобная множеству штука*) есть очень удобный метод **map**, чтобы работать с его элементами.
- Закроем глаза и представим, что наш тип **Maybe** - это массив с одним значением*(или пустой)* и сделаем наш метод **map**, чтобы он работал соответственно

И у класса Maybe появляется метод

```
map<R>(f: (val: T) => R): Maybe<R> {
  if (this.value === null) {
    return Maybe.Nothing<R>();
  }

  return Maybe.Apply(f(this.value));
}
```

И уже сейчас мы можем использовать это

```
const getMaybeMasterId = (name?: string) =>
  mastersRepo.findByName(name).map(m => m.id);

console.log(
  getMaybeMasterId("Yoda")
    .map(x => x.toString())
    .getOrElse("Not found")
);
```

Но если нужно пойти глубже

- Просто пишем

```
const masterName = 'First'

const group = mastersRepo
  .findByName(masterName)
  .map(master => master.groupId.map(f => mastersGroupRepo.findById(f)));
```

Но если нужно пойти глубже

- И получаем тип

```
const group: Maybe<Maybe<Maybe<MasterGroup>>>
```

- Как то так себе
- Но можно сделать метод **flatMap**(он и у коллекций есть) - который применяет функцию и “уплощает” тип, а в нашем случае просто не обернет его

И наш flatMap

```
flatMap<R>(f: (wrapped: T) => Maybe<R>): Maybe<R> {
  if (this.value === null) {
    return Maybe.Nothing<R>();
  } else {
    return f(this.value);
  }
}
```

Посмотрим, что же вышло с Maybe?

```
const performSearch = (name?: string) =>
  Maybe.Apply(name)
    .flatMap(prepareMasterInfo)
    .getOrElse("Нет информации об учителе");
```

Посмотрим, что же вышло с Maybe?

```
const prepareMasterInfo = (name: string) =>
  mastersRepo.findByName(name).map(m => {
    const groups = getMastersGroupInfo(m).getOrElse(" не состоит в группах.");
    const fly = getMastersFlyInfo(m).getOrElse(
      " Не знаем о способностях летать"
    );

    return `Учитель ${name} ${groups} ${fly}`;
  });
```

Посмотрим, что же вышло с Maybe?

```
const getMastersGroupInfo = (master: Master) =>
  master.groupId
    .flatMap(x => mastersGroupRepo.findById(x))
    .map(g => ` состоит в группе ${g.name}.`);

const getMastersFlyInfo = (master: Master) =>
  mastersFlyInfoRepo
    .findBymasterId(master.id)
    .map(i => i.canFly)
    .map<string>(canFly => (canFly ? " Может летать" : " Не может летать"));
```

В более иерархических структурах подобный подход окупается еще активнее

```
function getSupervisorName(enteredId: string) {
    if (enteredId) {
        const employee = repository.findById(parseInt(enteredId));
        if (employee && employee.supervisorId) {
            const supervisor = repository.findById(employee.supervisorId);
            if (supervisor) {
                return supervisor.name;
            }
        }
    }
}
```

В более иерархических структурах подобный подход окупается еще активнее

```
function getSupervisorName(maybeEnteredId: Maybe<string>): Maybe<string> {
    return maybeEnteredId
        .flatMap(employeeIdString => Maybe.fromValue(parseInt(employeeIdString))) // parseInt ca
        .flatMap(employeeId => repository.findById(employeeId))
        .flatMap(employee => employee.supervisorId)
        .flatMap(supervisorId => repository.findById(supervisorId))
        .map(supervisor => supervisor.name);
}
```

А если нам мало только наличия или отсутствия результата?

На помощь приходит другой тип данных - Either

- Все очень похоже на **Maybe**, но теперь вариант, который был **Nothing** стал хранить значение - мы не теряем информацию об ошибке
- В случае **map**/**flatMap** будет обработана только правая проекция(не ошибка)
- Очень удобно использовать для валидации(в случаях, когда достаточно сохранения только первого сообщения об ошибке)

Как же выглядит наш тип Either

```ts
export class Either<A, B> {
  private constructor(private left: A, private right: B) {}

  static Left<A, B>(error: A): Either<A, B> {
    return new Either(error, null);
  }

  static Right<A, B>(value: B): Either<A, B> {
    return new Either(null, value);
  }
```

Как же выглядит наш тип Either

```
map<R>(f: (arg: B) => R): Either<A, R> {
  if (this.isLeft()) {
    return new Either(this.left, null);
  }

  return new Either(null, f(this.right));
}
```

Как же выглядит наш тип Either

```ts
type EitherMatch<A, B, R, T> = {
  left: (error: A) => R;
  right: (value: B) => T;
};
```

```ts
match<R, T>(cases: EitherMatch<A, B, R, T>): R | T {
  if (this.isLeft()) {
    return cases.left(this.left);
  }

  return cases.right(this.right);
}
```

Для чего можно использовать?

```
const noHelloValidation = (text: string) =>
  text === "Hello"
    ? Either.Left<string, string>("No hello, plz")
    : Either.Right<string, string>(text);

const noByeValidation = (text: string) =>
  text === "Bye"
    ? Either.Left<string, string>("No goodbyes, plz")
    : Either.Right<string, string>(text);

const maxStrLenValidation = (size: number) => (text: string) =>
  text.length <= size
    ? Either.Right<string, string>(text)
    : Either.Left<string, string>("String too long");
```

Для чего можно использовать?

```
const textToVaildate: Either<string, string> = Either.Right("Test String");

const result = textToVaildate
  .flatMap(noHelloValidation)
  .flatMap(noByeValidation)
  .flatMap(maxStrLenValidation(5))
  .match({
    left: x => `Validation error: ${x}`,
    right: value => value
  });

console.log(result);
```

Для упрощения работы с кодом, где возможны исключениями

Тесты проще, больше не нужно обрабатывать исключения и делать какую либо дополнительную проверку, кроме значения

```
const dangerousFunction = () => {
  try {
    throw "Hello, haha";
    return Either.Right(5);
  } catch (x) {
    return Either.Left(x);
  }
};

dangerousFunction().match({
  left: x => console.log("There was error", x),
  right: x => console.log("All fine ", x)
});
```

Какие плюшки нам это дает?

- Мы можем не разрываться и описывать счастливый путь работы приложения
- Код выглядит линейнее, а наша программа работает как будто на рельсах
- Компилятор будет отслеживать корректность нашего приложения за счет типов

Примеры можно посмотреть здесь:

Maybe

Either

Input
with
Either

Mens et Manus

Контакты

anihirash@gmail.com
github.com/nihirash