Functional Programming in JavaScript

Dragos Vlad Cacuci

Motivation

Motivation

Reusability

Motivation

Reusability

Testability

Functional Programming

Theory of computable functions

Two fundamental ideas

- Turing machine
- Lambda calculus

Imperative Programming Disadvantages

- Shared state
- Mutable data
- Side effects

Hard to: reason about, reuse, debug, test

Functional Programming

- No shared state
- Immutable data
- No side effects

Easier to: modularize, compose, debug, test

Programming paradigms

Common programming paradigms:

- Imperative
- Functional
- Object-Oriented
- Event-Driven
- Logic

JavaScript, a multi paradigm programming language

JavaScript = Freedom

Functional programming is the “*programming paradigm that is going to play a much bigger role going forward as modern programming languages add more and more features that reduce boilerplate, repetition and syntax noise*”

Eric Elliot - The two pillars of JavaScript

JavaScript, a multi paradigm programming language

JS support for FP:

first-class functions, closures and lambda syntax

FP programs:

small, reusable and predictable pure functions

First-class and higher order functions

Higher order functions

First class functions

In JS, functions are objects

Higher order functions

Partial application, Currying

Example: Successor function

```
const add = (a, b) => a + b;

const addCurried = (a) => (b) => a + b;

const successor = addCurried(1);

successor(1); // 2
successor(2); // 3
successor(3); // 4
```

Example: Changing values from a data grid

Higher order components

Pattern that emerges from React’s compositional nature

Cross-cutting concerns

Application Example 4: Feature flagging

Application Example 5: Width provider

Pure functions

No side effects

Increased modularity: easier to test, debug, reuse, parallelize

Referential transparency

```
const add = (a, b) => a + b;

const multiply = (a, b) => a * b;

add(multiply(add(1, 2), 4), add(1, 2)); // 15
```

Pure functions

No side effects

Increased modularity: easier to test, debug, reuse, parallelize

Referential transparency

```
const add = (a, b) => a + b;

const multiply = (a, b) => a * b;

add(multiply(add(1, 2), 4), add(1, 2)); // 15

add(multiply(3, 4), 3); // 15
```

Purity by convention

Pure function: mapper between input arguments and return values

Not enforced by JavaScript

Example: Sorting with impure functions

Example: Sorting with pure functions

Immutability by convention

Object state cannot be modified after creation

Expressions evaluate to new data structures

Persistent Data Structures

JS support for Immutability :

Object.assign, spread operator, map, filter, reduce

ImmutableJS

Data structures: List, Stack, Map, OrderedMap, Set, OrderedSet, Record

Structural sharing via hash maps, tries and vector tries

Lazy evaluation

https://facebook.github.io/immutable-js/

Purely functional state

How?

Maintain referential transparency by making the state updates explicit

Return the new state along with the new values, leaving the old state unmodified

Separate the new state computation from the new state communication

Example: Passing state to pure functions

ReduxJS

Predictable state container for JavaScript apps

- Single store
- Emit action to change state using dispatch
- Use pure reducers

Example: Library application using Redux

Recursion

Replace while or for loops

- Avoid side effects
- Maintain statelessness

Example: Factorial using for loop

Example: Factorial using for recursion

This might result in in an stack overflow error

Tail call optimization

Last action of another function

Reuse the current stack frame for the function call

No syntax for denoting tail call optimization specified by ES6

New stack frames will not be created: O(1) memory complexity

Example: TCO Factorial

Conclusions

Stateless

Modular

Easier to combine

Easier to test

Easier to debug

Further reading

Programming JavaScript Applications, Eric Elliot, 2013

Functional Programming in JavaScript, Luis Atencio, 2016

Functional Programming in Scala, Paul Chiusano, Runar Bjarnason, 2015

Learning React, Eve Porcello, Alex Banks, 2017

Thank you!