Monady w JavaScript

Marcin Najder - **Comarch**

@marcinnajder

Agenda

- Abstrakcyjne typy danych
- Funktory
- Monady
- Inna interpretacja funkcji map i bind
- Podsumowanie

Funkcyjne języki programowania

1958 – Lisp

1973 – ML
1984 - SML
1987 - Caml
1990 – Haskell
1996 - OCaml

2003 - Scala
2005 - F#
2007 - Clojure
2012 - TypeScript
2016 - ReasonML

ML = SML = OCaml = F# (= ReasonML)

Algebraiczne typy danych

Algebraiczne typy danych - iloczyn

```
type byte = 0..255;
type NumericUpDown = {hidden: boolean; value: byte;}
type Tuple = [boolean, byte];     // boolean * byte


/*
true        *       1       =           (true, 1)
false               2                   (true, 2)
                    3                   (true, 3)
                    ...                 ...
2 * 256
*/
```

Algebraiczne typy danych - suma

```
type BooleanOrByte = boolean | byte;

/*
true        +       1       =           true
false               2                   false
                    3                   1
                    ...                 ...
2 + 256
*/


type BooleanOrByte =
    | {type:"boolean"; value: boolean;}
    | {type:"byte"; value: byte;};

const b : BooleanOrByte = {type:"boolean", value: false};
```

Algebraiczne typy danych - suma

```
// tagged union types, discriminated unions
type PaymentMethod =
    | { type: "cash" }
    | { type: "check", checkNo: number }
    | { type: "card",  cardType: string, cardNo: string};

function format(p: PaymentMethod) {
    switch (p.type) {
        case "cash":    return "cash";
        case "check":   return `check ${p.checkNo}`;
        case "card":    return `card ${p.cardType} ${p.cardNo}`
    }
}

// https://fsharpforfunandprofit.com/ddd/
// https://pragprog.com/book/swdddf/domain-modeling-made-functional
```

Typ Option<T>

```
// Maybe = Just | Nothing
type Option<T> =
    | { type: "some", value: T }
    | { type: "none" };

function some<T>(value: T): Option<T> {
  return { type: "some", value };
}
function none<T>(): Option<T> {
  return { type: "none” };
}

function tryParseNumber(text: string): Option<number> {
  const n = Number(text);
  return Number.isNaN(n) ? none() : some(n);
}
```

Problem wartości “null”

```
interface Person {
    firstName: string;
    lastName: string;
    middleName : Option<string>;
}

formatPersonInfo({firstName: "Marcin", lastName: "Najder"});
// -> "MARCIN NAJDER"

function formatPersonInfo(p: Person){
    return [p.firstName,
            p.middleName.type === "none" ? "" : p.middleName.value,
            p.lastName]
        .map(name => name.toUpperCase())
        .join(" ");
}
```

Typ Result<T>

```
// Either = Left | Right
type Result<T, TError> =
    | { type: "ok", value: T }
    | { type: "error", error: TError };

function ok<T, TError = {}>(value: T): Result<T, TError> {
    return { type: "ok", value };
}
function error<TError, T = {}>(err: TError): Result<T, TError> {
    return { type: "error", error: err };
}

function tryParseIntNumber(text: string): Result<number, string> {
    const n0 = tryParseNumber(text);
    if(n0.type === "none") return error("text is not a number")
    return Number.isInteger(n0.value) ?
        ok(n0.value): error("text is not an integer");
}
```

Funktor

Array<T>

```
[1, 2, 3].map(x => x * 10).map(x => x.toString());
// -> ["10", "20", "30"]

// typ pomocniczy opisujacy funkcje T -> TR
type f2<T, TR> = (item: T) => TR;


type Array<T>{
    // ...
    map<TR>(f : f2<T, TR>) : Array<TR>
}

// map: Array<T> -> (T -> TR) -> Array<TR>
```

Promise<T>

```
http.get("www.google.com")
    .then(res => res.body).then(body => body.length);
// -> Promise<number>


type Promise<T>{
    // ...
    then<TR>(f : f2<T,TR> ) : Promise<TR>;
}

// then: Promise<T> -> (T -> TR) -> Promise<TR>
```

Option<T>

```
const text = console.readLine();

tryParseNumber(text).map(n => n * 10).map(n => n.toString() );
// -> Option<string>



function map<T,TR>(m: Option<T>, f: f2<T,TR>) : Option<T2> {
  return m.type === "none" ? none<TR>() : some(f(m.value));
}

// map: Option<T> -> (T -> TR) -> Option<TR>
```

Czym jest Funktor ?

```
interface F<T> { }
interface FunctorOperations {
  map<T,TR>(m: F<T>, f: f2<T,TR>): F<TR>;
}

const arrayFunctorOps: FunctorOperations = {
  map<T,TR>(m: Array<T>, f: f2<T, TR>): Array<TR> { return m.map(f); }
};
const promiseFunctorOps: FunctorOperations = {
    map<T,TR>(m: Promise<T1>, f: f2<T,TR>) { return m.then(f}; }
};
const optionFunctorOps: FunctorOperations = ... m.map(f} ... ;
const resultFunctorOps: FunctorOperations = ... m.map(f} ... ;
```

Monad

Array<T>

```
[1, 2, 3].map( x => x * 10);       // [10, 20, 30]
[1, 2, 3].map( x => [x, x * 10]); // [ [1, 10], [2, 20], [3, 30] ]
// ES2019
[1, 2, 3].flatMap( x => [x, x * 10]); // [ 1, 10, 2, 20, 3, 30 ]
```

```
type Array<T>{
    map<TR>(f : (item:T) => TR): Array<TR>;
    flatMap<TR>(f : (item:T) => Array<TR>): Array<TR>;
}

// flatMap: Array<T> -> (T -> Array<TR>) -> Array<TR>
```

Promise<T>

```
http.get("/api/items")
   .then( res => http.get(`/api/items/${res.body[0].id}`) );
// -> Promise<{...}>


type Promise<T>{
    then<TR>(f : f2<T, TR>) : Promise<TR>;
    then<TR>(f : f2<T, Promise<TR> >) : Promise<TR>;
}

// then: Promise<T> -> (T -> Promise<TR>) -> Promise<TR>
```

Option<T>

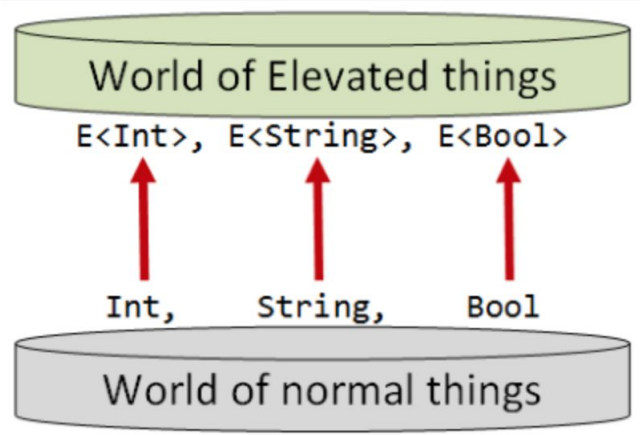
```
const tryReadNumber = () => tryParseNumber(console.readLine());


tryReadNumber()
  .bind(n => tryReadNumber().map(m => `${n}+${m}=${n+m}`) );
// -> Option<string>

function bind<T,TR>(m: Option<T1>, f: f2<T, Option<TR> >): Option<T2> {
  return m.type === "none" ? none<TR>() : f(m.value);
}

// bind: Option<T> -> (T -> Option<TR>) -> Option<TR>
```

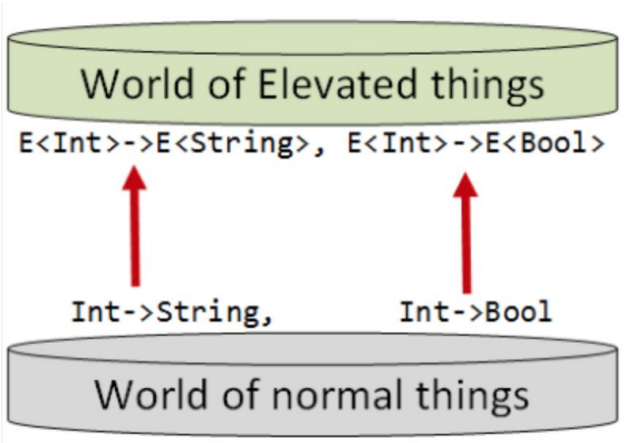
Czym jest Monad ?

```
interface M<T> { }
interface MonadOperations {
    bind<T, TR>(m: M<T>, f: f2<T, M<TR> >):  M<TR>;
    return_<T>(value: T): M<T>;
}

const arrayMonadOps: MonadOperations = {
    bind<T,TR>(m: Array<T>, f: f2<T, Array<TR> >): Array<TR> {
        return m.flatMap(f);
    },
    return_<T>(value: T): Array<T> {
        return [value];
    }
};
const promiseMonadOps: MonadOperations = ... Promise.resolve(value) ...
const optionMonadOps: MonadOperations = ... some(value) ...
const resultMonadOps: MonadOperations = ... ok(value) ...
```

Czym jest Monad ??

- Array, Promise, Observable, Iterable, Option, Result, … są Monadami
- Monad<T> to wartość “typu T” plus “kontekst”
  - **Option<T>** - wartość T … , ale także może jej nie być
  - **Promise<T>** - wartość T … , ale pojawi się za jakiś czas w przyszłości
  - **Array<T>** - wartość T … , no dobra wiele wartości T lub “jedno z wartości”
- https://fsharpforfunandprofit.com/posts/elevated-world/





Super ..., ale co to daje w praktyce ?

- Funktory i Monady to wzorce projektowe takie jak np. **Iterator** który daje nam:
  - API, pewna abstrakcja procesu iterowania
  - wsparcie w języku programowania (pętla "foreach")
  - funkcje działające na iteratorach np. filter, map, reduce, ...

```
interface Iterable<T> {
    iterator(): Iterator<T>;
}
interface Iterator<T>{
    next(): { done: boolean; value: T };
}

class Array<T> implements Iterable<T> { ... }
class Map<T> implements Iterable<T> { ... }
class Set<T> implements Iterable<T> { ... }
```

Pętla for..of

```
const collection = [1, 2, 3]; // new Map(), new Set()
for(const el of collection){
    console.log(el);
}


// petla for..of tlumaczona jest na:
const iterator = collection.iterator();
let ir;
while( !(ir = iterator.next()).done ) {
    console.log(ir.value);
}
```

Haskell - notacja “do”

```
// result:: Option<string>
result = do
    n <- tryReadNumber()
    m <- tryReadNumber()
    return `${n}+${m}=${n+m}`


// notacja "do" tlumaczona jest na:
tryReadNumber()
  .bind(n => tryReadNumber().bind(m => some(`${n}+${m}=${n+m}`) ));
```

C# - LINQ

```
Option<string> result =
    from m in tryReadNumber()
    from n in tryReadNumber()
    select `${n}+${m}=${n+m}`


// LINQ tlumaczone jest na:
tryReadNumber()
  .SelectMany(n => tryReadNumber()
    .SelectMany(m => new [] {`${n}+${m}=${n+m}`]} );


// swoje odpowiedniki maja: F#, Scala, Kotlin, ...
```

Option<T>

```
function f(): Option<string> {
    const n = tryReadNumber();
    if(n.type === "none") return none();

    const  m = tryReadNumber();
    if(m.type === "none") return none();

    return `${n.value}+${m.value}=${n.value+m.value}`) );
}


// alternatywny zapis korzystajac z bind
tryReadNumber()
    .bind(n => tryReadNumber().bind(m => some(`${n}+${m}=${n+m}`)));
```

JavaScript - notacja “do” na bazie generatorów ES6

```
do_(function* () {
    const n = yield tryReadNumber();
    const m = yield tryReadNumber();
    return `${n}+${m}=${n+m}`;
});  // -> Option<string>



tryReadNumber()
    .bind(n => tryReadNumber().bind(m => some(`${n}+${m}=${n+m}`)));



// "notacja do" z Haskell wykorzystujac generatory ES6
https://github.com/marcinnajder/misc/blob/master/2019_03_13_monad_do_notation_using_js_generators
```

JavaScript - notacja “do” na bazie generatorów ES6

```
do_(function* () {
    const res = yield http.get("/api/items");
    const obj = yield http.get(`/api/items/${res.body[0].id}`);
    return obj;
}); // -> Promise<{...}>
```

```
http.get("/api/items")
  .then( res => http.get(`/api/items/${res.body[0].id}`) );
```

```
// async/await ??? :)
```

JavaScript - notacja “do” na bazie generatorów ES6

```
do__(function* () {
    const i = yield [1, 2, 3];
    const j = yield ["a", "b"];
    return [i + j];
}); // -> Array<string>


// [ "1a", "1b", "2a", "2b", "3a", "3b" ]


[1, 2, 3].flatMap(i => ["a", "b"].flatMap(j => [i + j]));
```

Czy do_(...) może być użyteczne ?

```
do_(function* () {
    const n = yield tryReadNumber();
    let sum = 0;

    for(let i=0; i<n; ++i){
        const value = yield tryReadNumber();
        sum += value;
    }

    return sum;
});  // -> Option<number>

// jak to zapisac za pomoca bind ??
```

Funkcje działające “na iteratorach”

```
const collection = [1, 2, 3, 4];  // new Map(), new Set()
filter(collection, x => x % 2 === 0); // -> 2, 4


function* filter<T>(source: Iterable<T>, predicate: f2<T, boolean>)
    : Iterable<T> {
    for (var item of source)
        if (predicate(item))
            yield item;
}

// https://github.com/marcinnajder/powerseq
const items = pipe(
    [1, 2, 3, 4, 5, 6]
    filter(x => x % 2 === 0),
    map(x => x * 100));
```

Funkcje działające “na monadach”

```
// mapM: Array<T> -> (T -> M<TR>) -> M<Array<TR>>

["1", "2.6", "a" ].map(v => tryParseNumber(v) );
//-> Array<Option<number>>

mapM(["1", "2.6", "a" ], v => tryParseNumber(v) );
//-> Option<Array<number>>


["comarch.com","google.com"].map(url => http.get(url) );
// -> Array<Promise<string>>

mapM(["comarch.com","google.com"], url => http.get(url) );
// -> Promise<Array<string>>
```

Funkcje działające “na monadach”

```
// "monadic functions"
function mapM<T, TR>(items: T[], f: f2<T, M<TR> > ): M<TR[]> { ... }
function filterM<T>(items: T[], f: f2<T, M<boolean>> ): M<T[]> { .. }
function reduceM<T, TA>(items: T[], f: (prev: TA, item: T) => M<TA>, seed: TA):
M<TA> { ... }
...


// obecnie funkcje mapM, filterM, reduceM, ... sa wielokrotnie
// re-implementowane dla kazdego z typow Array, Promise, Observable,
// Iterable, ... w bibliotekach lodash, rambda, async.js, rxjs, ixjs, ...
```

Inna interpretacja map i bind

Currying

```
// number -> number -> number -> number
function add(a: number, b: number, c: number): number {
  return a + b + c;
}

// number -> (number -> (number -> number))
function addC(a: number) {
  return function (b: number) {
    return function (c:number){
      return a + b + c;
    };
  };
}             // let addC = a => b => c => a + b + c;

add(1,2,3) === addC(1)(2)(3)        // -> true

addC(1)(2)  (3), addC(1)  (2)(3)   // partial function application
```

pipe

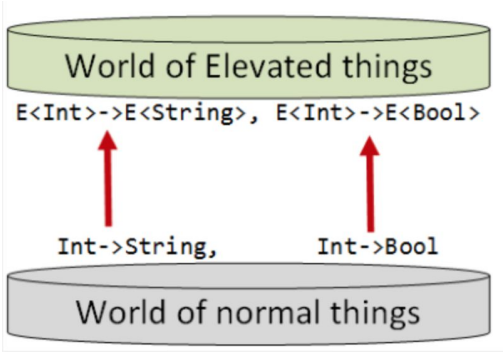
```
pipe(10,
    n => n * 10,              // 100
    n => n.toString(),        // "100"
    s => s.length );          // 3
```

```
function pipe(a: any, ...fs: Function[]) {
  return fs.reduce((prev, el) => el(prev), a);
}
```

|> - pipe w JavaScript

```
// https://github.com/tc39/proposal-pipeline-operator
// This proposal introduces a new operator |> similar to F#, OCaml,
Elixir, Elm, Julia, Hack, and LiveScript, as well as UNIX pipes. ...

let result = exclaim(capitalize(doubleSay("hello")));
result //=> "Hello, hello!"

let result = "hello"
  |> doubleSay
  |> capitalize
  |> exclaim;

result //=> "Hello, hello!"
```

Inna interpretacja funkcji map



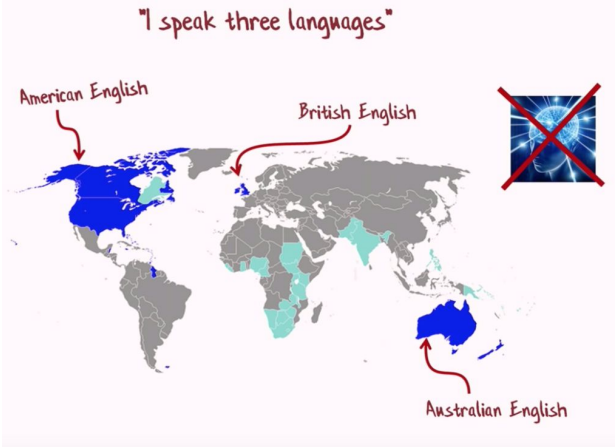

```
// Option<T> -> (T -> TR) -> Option<TR>

// (T -> TR) -> Option<T> -> Option<TR>

// (T -> TR) -> (Option<T> -> Option<TR>)


// funkcja "map" nazywana jest czesto "lift"
const stringLength = (text: string) => text.length;
// -> string -> number
const stringLength0 = map(stringLength);
// -> Option<string> -> Option<number>

pipe(tryParseNumber(...),
    map(n => "a".repeat(n)), ...  )

// Option<number> -> ( Option<number> -> Option<string>) -> ...
```

Inna interpretacja funkcji bind

```
// Option<T> -> (T -> Option<TR>) -> Option<TR>

// (T -> Option<TR>) -> Option<T> -> Option<TR>

// (T -> Option<TR>) -> (Option<T> -> Option<TR>)


const tryParseNumber = (text: string) => { ... };
// string -> Option<number>
const tryParseNumber0 = bind(tryParseNumber);
// Option<string> -> Option<number>


pipe(tryParseNumber(...),
    bind(n => n < 1 ? none() : some("a".repeat(n)) ), ...  )
// Option<number> -> ( Option<number> -> Option<string>) -> ...
```

Podsumowanie

- Monady nie są takie straszne ;)
- TypeScript jest niesamowity!
- Warto się uczyć języków …
  - https://youtu.be/0fpDlAEQio4?t=720 Four Languages from Forty Years Ago - Scott Wlaschin



Materiały

- F#
  - https://fsharpforfunandprofit.com
  - https://fsharpforfunandprofit.com/video/
  - https://pragprog.com/book/swdddf/domain-modeling-made-functional
- Haskell
  - https://channel9.msdn.com/Series/C9-Lectures-Erik-Meijer-Functional-Programming-Fundamentals
  - http://learnyouahaskell.com/
- SML/OCaml, LISP, Ruby
  - https://courses.cs.washington.edu/courses/cse341/13wi/
- Kod
  - https://github.com/marcinnajder/powerfp
  - https://github.com/marcinnajder/powerseq

Dziękuję za uwagę !