Pure functional programming in Typescript

Contents

Introduction

Motivation for the pure functional programming

- compare with procedural paradigm
- referential transparency
- railway-oriented programming

The power of function composition

- implement `flow` and `pipe`

Representing possibly missing value

Javascript has two important special values we use for representing a *missing value* - `undefined` nad `null`. They are often used as a result of computations such as

- searching for an element in array,
- fetch a row from a database database,
- retrieving a value from `Map`.

Let's take the *finding an element in a list* as an example. `Array.find` returns `undefined` in case no element is found.

```
const array = [1, undefined];
const maybeFoundElement = array.find((i) => i === undefined)
```

If we assumed arbitrary value for the `array` variable, does `maybeFoundElement === undefined` mean the element was not found or that we found an element with value `undefined`? Unfortunately, we can't say for sure. It would be super nice to have a special value that would be used only for representing a missing value. We, don't have such a value by default, but what we can do is to wrap the result of computations where the value might be missing in a *tagged* object a use the tag to distinguish actual value from the missing one.

`Option` type

Let's introduce two types we will use to represent *none* and *some* value.

```
type None = { readonly _tag: 'none' };
type Some<A> = { readonly _tag: 'some', readonly value: A };
```

The type representing a missing value is simply an object with `_tag` set to `none`. `Some` is little bit more interesting. For `Some` we want to have an ability to represent *some number*, *some string*, *some user credentials* or generally *some* ***A***. Therefore, we need to make the `Some` type **generic** in the `value` type.

So far we might have used types like `string | undefined` for representing either `string` or missing value. Now, we can use type `Some<string> | None`. Now, if we want to represent correctly a

result of `find` function on an array of type `(string | undefined)[]` we can because we can have type such as `Some<string | undefined> | None`. Let's give the `Some<A> | None` a name for convenience.

```
type Option<A> = Some<A> | None
```

Implementing some primitive array functions

Let's implement some primitive functions for working with arrays. The problem with arrays is that we can't be sure whether the array is empty, contains at least a single item, or contains at least `n` items. Therefore, if we want to write a function returning the first element of an array, it is actually a computation that might return a missing value because the array might be empty. Let's start by implementing `constructors` for the `Option<A>` type.

```
export const some = <A>(value: A): Some<A> =>
  ({ _tag: 'some', value });

export const none: None = ({ _tag: 'none' });
```

Let's implement the `head` function that returns a first element of an array if present.

```
import * as O from './Option';

export const head = <A>(xs: readonly A[]): Option<A> =>
  xs.length === 0 ? O.none : O.some(xs[0]);
```

Somewhat opposite of head is a tail of the array. The tail of an array contains all the items of an array but the first one. The tail of an array might be an empty array, but if the input array is empty in the first place, there is no tail because there is not even a first element. Therefore `tail` will be a computation the might return a missing value. We have another use-case for our `Option` type!

```
export const head = <A>(xs: readonly A[]): Option<A> =>
  xs.length === 0 ? O.none : O.some(x.slice(1));
```

We have a way to obtain the first element of an array. If we are asked to get the last one now it should be no surprise we'll use the `Option` type again. We need at least a single element to be able to get the last element.

```
export const head = <A>(xs: readonly A[]): Option<A> =>
  xs.length === 0 ? O.none : O.some(x[x.length - 1]);
```

The last function missing is `init` which returns everything but the last element. Again, we need at least one element in order to get the init, therefore for any input array the computation of the init might result in a missing value.

```
export const head = <A>(xs: readonly A[]): Option<A> =>
  xs.length === 0 ? O.none : O.some(x.slice(0, x.length - 1)]);
```

Computing a second element of an array

After we implemented `head`, `tail`, `last` and `init` functions it must a breeze to implement function returning the second element or nth element. We might do something like this.

```
export const second = <A>(xs: readonly A[]): Option<A> =>
  xs.length < 2 ? O.none : O.some(x[1]);
```

But a it feels like we could have reused existing functions to express this one. Second element is *head* of a *tail*.

```
const second = <A>(xs: readonly A[]) => pipe(xs, tail, head);
```

Are we done? Typescript's type checker doesn't think so. The `tail` returns `Option<A[]>` a head accepts `A[]`. Let's try to generalize this problem. We have a value of `Option<A>` for arbitrary `A` and a function of signature `(a: A) => Option<B>` and we need to way to run that function with `Option<A>`. Just by looking on the types we can spot two options.

1. We might try to introduce a function that can unwrap the value of `Option<A>` and produce `A`, let's call it `unwrap: <A>(fa: Option<A>) => A`. If such a function exists then we're done because we just need to compose `tail` with `unwrap`.
2. Another way is to try introduce a function that will take the function `(a: A) => Option<B>` and transform it onto a function `(a: Option<A>) => Option<B>`. We'll give that function name `flatMap`. If such a `flatMap` exists, we're done because everything we have to do to fix the implementation of `second` is to compose `flatMap` with `head`.

The first option is unfortunately not doable. `unwrap` doesn't make sense for `Option` type because we have nothing to return in case the value is `None`. The second option is more promising because if we have a function that can transform `A` to `Option<B>` we can make it accept `Option<A>` by triggering the original function only if `Option<A>` is `Some<A>` and otherwise just return `None`.

We need to implement the `isSome` function first.

```
export const isSome = <A>(fa: Option<A>): fa is Some<A> => fa._tag === 'some';
```

With `isSome` we have everything we need to implement the `flatMap` function. Note that it is important for the implementation of `flatMap` that `isSome` is a guard function instead of a one returning just a `boolean`. Check yourself why as an exercise.

```
export const flatMap = <A, B>(f: (a: A) => Option<B>) => (fa: Option<A>):
↪  Option<B> =>
  isSome(fa) ? some(f(fa.value)) : fa;
```

And with the `flatMap` function, we can implement `second` pretty easily.

```
const second = <A>(xs: readonly A[]) => pipe(xs, tail, flatMap(head));
```

From this examle, `flatMap` is the important function. By introducing `flatMap`, we enbodied ourself with a very important ability to compose functions doing computations with possibly missing result.

Data types supporting `flatMap` operations are very close to having capabilities needed to call themself **Monads**.

Option is a Monad

`flatMap` is one of two functions we need to implement in other to have a full *Monad instance* for the Option data type. The remaining one is usually called *pure*, *return* or *of* and it is very simple. Moreover, we actually already implemented it. It is the `some` function which gives us an ability to wrap a value into the *Option*.

```
export const of: (a: A) => Option<A> = some;
```

We'll discuss some more criteria and details on what does it mean for something to be a Monad. For now, let's agree we have a Monad if for a type `F<A>` we have two functions

- `of` of type `<A>(a: A) => F<A>` and
- `flatMap` of type `<A, B>(f: (a: A) => F<B>) => (fa: F<A>): F<B>`.

Mapping over an option

Let's imagine we want to compute a discount on a product. To do so, we implement a function `computePriceDiscount` which will accept the initial price without a discount, and the discount which is a number between 0 and 1. The discount might be missing if the costumer is not eligible to any discount, therefore we will model the discount using `Option<number>` type. We want to show the final discount to the user and we want to specifically report the situation when the discount is not available.

So our function will have a signature `(initialPrice: number, maybeDiscount: Option<number>) => Option<number>`. It will compute the final discount when `discount` is `Some` and return it as `Some<number>` and it will pass through the `None` if there is no discount. We could implement such a function using functions working on `Option` we already have. But this kind of *mapping* over an possible missing value is such a commonly used pattern it deserves its own combinator.

```
export const map = <A, B>(f: (a: A) => B) => (fa: Option<A>): Option<B> =>
  isRight(fa) ? some(f(fa.value)) : fa;
```

This new `map` function makes the implementation of `computePriceDiscount` a child game. We just need to *map* over the `maybeDiscount` value with a function that multiplies the discount by the price. And we're done!

```
const computePriceDiscount = (price, maybeDiscount: Option<number>):
↪  Option<number> =>
  pipe(
    maybeDiscount,
    O.map((discount) => discount * price)
  );
```

The name of the `map` function deserves a little bit of commentary. We already have `map` function on Javascript arrays. Array's `map` function creates a new array by applying the function to each element of the original array. If we forget about our definition of `Option` for a moment and pretend the `Option` is an array that can have at most a single element then our implementation of the `map`

function could simply use the Array's `map` because an empty array stays the same after mapping over it and every single item array would get transformed into another single item array.

The ability of mapping over a value in *context* is something we will see pretty often in the functional programming. By providing such an ability is the first step to define a **Functor** instance for a given data type.

Let me out!

TODO: implement ***`getOrElse`***

Filtering the value

TODO: implement the ***`filter`***

Pure computation that can fail

In the previous chapter we introduced an `Option` type. Using an `Option` is a very clean way to encode computations that can fail for only a single, usually an implicitely known reason. In case our computation might fail for multiple reasons and we care about those different reasons, `Option` is no longer good. An example of a functionality where we care about the error and there are multiple way to fail is a password validation during the registration. Let's say our policies for the user password are the following ones.

- it must contains at least a single alpha character
- it must contains at least a single number
- it must be longer then 5 characters

Functions describing these constrains might look as follows.

```
const containsAlphaCharacter = (password: string) =>
  /[a-zA-Z]/.test(password);

const containsNumberCharacter = (password: string) =>
  /\d/.test(password);

const isLongEnough = (password: string) =>
  password.length >== 5;
```

With the `O.filter` combinator, we can implement implement the validation as follows.

```
const validate = (password: string) => pipe(
  O.of(password),
  O.filter(containsAlphaCharacter),
  O.filter(containsNumberCharacter),
  O.filter(isLongEnough),
);
```

This will work and it's a very clean and compact way to encode the password validation. But there is a huge practical disadvantage. The user of our system would receive an error message indicating the password is invalid but we have no way to propagate what exactly is the problem. The `Either` type will solve the problem.

Either

Let's create a type that will be a sum type of two types wrapping *either* the right successful value `A` or the left value describing the failure `E`.

```
type Left<E> = { _tag: 'left', left: E };
type Right<A> = { _tag: 'right', right: A };

type Either<E, A> = Left<E> | Right<A>;
```

Let's create constructors for the `Either` type the same way we did for the `Option`.

```
export const left = <E>(e: E): Left<E> =
  ({ _tag: 'left', left: e });

export const right = <A>(a: A): Right<A> =
  ({ _tag: 'right', right: a });
```

And we'll definitely find a use for functions telling us whether `Either<E, A>` in hand is specifically `Left<E>` or `Right<A>`.

```
export const isLeft = <E>(fa: Either<E, A>): fa is Left<E> =
  fa._tag === 'left';

export const isRight = <A>(fa: Either<E, A>): fa is Right<A> =
  fa._tag === 'right';
```

Implementing the validation

To implement the validation, we need an ability to put a thing into the `Either` and to filter a value in the `Either`. We already have the `right` function so the `of` function is going to be an alias to `right`. *TODO discuss the never type*

```
export const of: <A>(a: A): Either<never, A> => right(a);
```

The `filter` function needs to be a little bit different from the one working on `Option`. When the predicate returns `false` we don't have a single known value in the failure rail but generally any value of the `E`. Because of that, the signature of `filter` for `Either` will differ by an aditional argument which will be the provided value of `E` to be used when the predicate evaluates to `false`. To make this difference clear, we'll call the function `filterOrElse` instead of just `filter`.

```
export const filterOrElse =
  <A, E>(predicate: (a: A) => boolean, onFalse: () => E)
```

```
    => (fa: Either<E, A>): Either<E, A> =>
  isLeft(fa) || predicate(fa.right) ? fa : onFalse();
```

Now, we can rewrite the `validate` function to report the specific problem it failed on.

```
const validate = (password: string): Either<string, string> => pipe(
  E.of(password),
  E.filterOrElse(containsAlphaCharacter, () => "must contain at least 1 letter"),
  E.filterOrElse(containsNumberCharacter, () => "must contain at least 1
  ↪  number"),
  E.filterOrElse(isLongEnough, () => "must have at least 5 characters"),
);
```

We should note here this is probably still not the best solution. If we call `validate('abcd')` the output will be an error stating the password must contain at least 1 number. The problem is the password is also not long enough because it contains only 4 characters. Ideally, the `validate` function reports all the problem at once so we can display all the problem to the user.

Converting between Option and Either

We can see, the `Either` type is pretty similar to the `Option` type. The only difference is we can store something in the `Left` type. Actually, if we fix the `E` to a single value type we'll get a type that describes the same information as the `Option` type.

```
type RestrainedEither<A> = Either<null, A>;
```

The restrained type can be either `Left<null>` or `Right<A>`. And the only value that has a type `Left<null>` is the `left(null)`. Therefore without any loss of information we can convert it to and from an Option.

```
const toOption = <A>(fa: Either<null, A>): Option<A> =>
  isLeft(e) ? none : some(e.right);

const toEither = <A>(fa: Option<A>): Either<null, A> =>
  isSome(fa) ? right(fa.value) : left(null);
```

This is very interesting! We have a way to convert `Either<null, A>` to `Option<A>` and also the other way around. This kind of *convertability* has a mathematical name - **isomorphism**. Also, we can say `Either<null, A>` is isomorphic to `Option<A>`. The practical consequence of that is we could have defined `Option<A>` in terms of `Either<null, A>`. Therefore, all the handy functions would be implemented only once for `Either<E, A>` and for `Option<A>` we would resuse them thanks to the fact we can freely convert between `Either<null, A>` and `Option<A>`.

If we *unrestrain* the `Either` type back to the general one `Either<E, A>` and compare it to the `Option<A>` we can see the `Either` can store more information then the `Option`. That means we can't really convert them between each other without a loss or a creation of information. If we want to convert `Option<A>` to `Either<E, A>` we have to deal with the missing value properly. The most general solution is to provide a way to specify what to put in the `Left` value if the `Option` is `None`.

```
export const fromOption = <E>(onNone: () => E) => <A>(fa: Option<A>): Either<E,
↪  A> =>
  iSome(fa) ? right(fa.value) : left(onNone());
```

What about the other way around? If we have an `Either<E, A>` and want to convert it to the `Option<A>` we unfortunately loose information about the value stored in `Left<E>`. Compared to `fromOption`, we don't have the variability for a value representing the missing value because it can be only `none`.

```
export const toOption = <E, A>(fa: Either<E, A>): Option<A> =>
  isRight(fa) ? some(fa.right) : none;
```

Higher kinded types

Introduction to type-level programming

In typescript, it is possible to think about generic types as they were *type-level functions*. For example, the following generic type can convert `number` type to `string` type.

```
type ToInput<T> = T extends number ? string : never;
```

If the type function receives something else then `number` it will return `never`. Otherwise, it will return `string` if `T` is a subtype of `number`.

```
type A = ToInput<number>; // => string
type B = ToInput<string>; // => never
```

We can extend it the type-level function to convert `string` to `string` and `boolean` to `'false' | 'true'`.

```
type ToInput<T> =
  T extends number ? string
  : T extends string ? string
  : T extends boolean ? 'false' | 'true'
  : never;
```

What if we wanted to be able to extend the behaviour if this type-level function without touching its implementation. Something like this.

```
type UserDefinedTypeToInput<T> =
  T extends UserId ? string
  : T extends CommentId ? string
  : never;

type ToInputExtended<T> = ToInput<T, UserDefinedTypeToInput>;
```

This is the same as providing callback function in the term-level code. But in this case we would like to do it in the type-level code.

To be able to pass functions around or return them as results is usually called a support of higher-order functions. In other words, we can treat functions as values. Ability to treat type constructors as types is an equivalent problem. Unfortunately, traditional languages including Typescript don't support this. Anyway, why is it such a big deal and why specifically for the functional programming.

We've already seen `Option` and `Either` types and some patterns repeating for both of them. We mentioned the **Functor** for example. Functor instance is an object that implements a `map` function which must satisfy certain laws. In case of `Option`, the `map` function has the following signature.

```
type OptionMap = <A, B>(f: (a: A) => B) => (fa: Option<A>): Option<B>
```

And `Either` has this one.

```
type EitherMap = <A, B>(f: (a: A) => B) => <E>(fa: Either<E, A>): Either<E, B>
```

As we can see, they are very similar. The main difference is type constructor, `Option` or `Either` in this case. If we wanted to generalize the idea of this map function we would like to writing something as follows.

```
type Map<F> = <A, B>(f: (a: A) => B) => (fa: F<A>): F<B>
```

And we could compute types for our maps using this type constructor.

```
type OptionMap = Map<Option>;
type EitherMap = Map<Either>;
```

In case of `Map<Either>`, we have more problems because the `Either` type needs 2 types to evaluate to an actual type. Let's not deal with this problem for now and pretend the type is correctly partially applied.

Encoding using `this` trick

Encoding using type-level map

Functional blocking effects

Introducing `IO` type

Typeclasses

Functor instances

Monad instances

Validation