# 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`
- function currying

# Representing possibly missing value

Javascript has two important special values we use for representing a *missing value* - `undefined` and `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. From this moment, 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 have the type `Some<string | undefined> | None` that correctly distinguishes whether we found a value of type `string | undefined` or found nothing. 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, or contains at least a single item. 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' });
```

We can describe `some` function as something that can take value of type `A` and put it inside of the `Option` context. We are using the term *context* because we can think of the `Option` being some kind of a box for values.

Let's use the `some` and `none` constructors to implement the `head` function.

```
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. A second element of an array is *head* of the array's *tail*. Let's try implementing that.

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

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

We might try to introduce a function that can unwrap the value of `Option<A>` and produce `A`, let's call it `unwrap`. Such a function would have a type `<A>(fa: Option<A>) => A` and the implementation of the `second` function would change by unwrapping the result of calling `tail`.

```
pipe(xs, tail, unwrap, head)
```

The question is whether such a function exists for the `Option` type. Unfortunately, there is no safe implementation of the `unwrap` function because there is nothing to return in case the value is `None`.

Another attempt is to introduce a function that will take the function `(a: A) => Option<B>` and transform it into a function of type `(a: Option<A>) => Option<B>`. Let's call that function `flatMap`. Body of the `second` function would change like this.

```
pipe(xs, tail, flatMap(head))
```

Can we implement the `flatMap` function? Let's check whether we can safely cover both the cases of optional value in a sensical way.

- If the input optional value is `None`, we can simply passthrough that value.
- If the input value is `Some<A>`, we can safely unwrap the value, call the original function and return this result.

That looks reasonable. We just need to implement a function that will tell us whether the input value of type `Option<A>` is `Some<A>` or `None`. We can do that by checking the value of `_tag` property.

```ts
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.

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

Alright! With the `flatMap` function, we can implement the `second` function as follows.

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

In this exercise, `flatMap` is the important function. By introducing `flatMap`, we embodied ourself with a very important ability to compose functions doing computations with possibly missing result. The `flatMap` is one of two functions we need in other to have a capabilities of the cursed *Monad* word. The other one is usually called *pure*, *return* or *of* and actually, we already implemented it. It is the `some` function which gives us an ability to wrap a value into the `Option`.

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

We'll discuss some more criteria and details on what's the Monad thingie actually about. For now, let's just say we need the two following 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>`.

Applying `flatMap` with a function is sometimes being referred to as *lifting* the function to a context. Let's recall again that `flatMap` is transforming a function of type `(a: A) => F<B>` to a function of type `(a: F<A>) => F<B>`. Thus it turns a function working on `A` into a function working on `F<A>`.

## 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.

```ts
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.

## Putting stuff into the `Option` context

We've already seen how to put values into the `Option` context using the `some` constructor. In `head` or `tail` functions, we evaluated a predicate and based on its value we decided whether to create some value or none. Value we put into the context was computed before it is put inside of the context.

This pattern is so common there is a general function `fromPredicate` that accepts a value we want to wrap into the option and predicate function that can operate upon that value. The `fromPredicate` works as a guard that guarantees we let the value in the context only if the predicate evaluates to true. If it evaluates to false, the context will contain none value.

```
export const fromPredicate = (p: (a: A) => boolean) => (a: A): Option<A> =>
  p(a) ? some(a) : none;
```

The `fromPredicate` is another example of constructor. It puts things into the context but in comparison to `some` function it can decide what trail to choose based on the predicate. We can think of `some` being a special case of the `fromPredicate` function. `some` might be implemented in terms of the `fromPredicate` by passing in a predicate that always returns `true` independently of the input value `A`.

```
const some = fromPredicate(() => true);
```

What would be the strategy to implement e.g. the `head` function using `fromPredicate`? If we use it to make sure we are in the successful trail only if the array is non-empty, then we can employ the `map` and do the 0th index access in the context where we are guaranteed the array contains at least a single item. And, that's basically it.

```ts
const head = <A>(xs: readonly A[]) => pipe(
  xs,
  O.fromPredicate((xs) => xs.length > 0),
  O.map((xs) => xs[0])
);
```

There is one more strategy we didn't try. Getting back to our analogy between optional value and an array containing at most 1 element, let's think about the `Array.prototype.filter` function. `Array`'s filter leaves in elements that satisfy the provided predicate function. Can we have an analogy of the `filter` function also for `Option` type? Sure thing we can!

The `filter` will accept a predicate and it will leave the `Option` value in the `Some` trail only if the predicate evaluates to true. If the input `Option` value is `None` we don't need to do anything because nevertheless it is already in the trail without a value thus we don't have a value to run the predicate upon. For the implementation we might try to reuse the `fromPredicate` function. The signature of `fromPredicate` applied with a predicate is `(a: A): Option<A>` and we would like to make it work on `Option<A>` instead of `A`. Such a mapping is exactly what the `flatMap` does! Using that knowledge we can formulate the `filter` function as follows.

```ts
export const filter = (p: (a: A) => boolean) => (fa: Option<A>): Option<A> =>
  pipe(fa, O.flatMap(O.fromPredicate(p)));
```

## Let me out!

In the following scenario, we're going to prepare a curried function `getFromRecord` that can accept an object of type `Record<string, string>`, a `string` and it will return `Option<string>`. Specifically:

- `Some<string>` containing the value under the input key if the key is present in the object, or
- `None` if the key doesn't exist in the object.

We can already implement it just by using `Option` constructors because we know object property access returns `undefined` if the key doesn't exists and we can be sure there is key with value of `undefined` because of the specified type of the record.

```ts
const getFromRecord =
  (record: Record<string, string>) => (key: string): Option<string> => {
    const valueOrUndefined = record[key];
    return valueOrUndefined === undefined ? none : Some(valueOrUndefined);
  }
```

Such a solution is pretty okay. The expression in the return statement feels somehow general enough we could extract it to it's own function. What happens there is a conversion from `string | undefined` to `Option<string>`. Such a transformation can be easily generalised to conversion from `A | undefined` to `Option<A>`. Let's create a new function out of it.

```ts
export const fromUndefinedable = <A>(a: A | undefined): Option<A> =>
  a === undefined ? none : some(a);
```

Beautiful! Using the `fromUndefinedable` function, implementation of `getFromRecord` becomes trivial.

```
const getFromRecord =
  (record: Record<string, string>) => (key: string): Option<string> =>
    fromUndefinedable(record[key]);
```

The `getFromRecord` function can be used, for example, to pass around a closure with applied object representing some kind of configuration where we know some fields are missing or they are dynamically calculated during the runtime of the application.

```
const entrypoint = pipe(
  getFromRecord({ userName: "Patrik" }), // statically provided configuration
  doSomethingWithConfig
);


const doSomethingWithConfig =
  (getConfig: (key: string) => Option<string>) => pipe(
    getConfig("userName"),
    O.map((userName) => `Hello, ${userName}`)
  );
```

In the example above, we provide the example configuration and by calling the `getFromRecord` we create a closure with signature `(key: string) => Option<string>`, therefore we can call this function with a string value to potentially obtain the value.

What if we wanted to provide a default for the key? There are multi way to do that. We can try to go general again a create a function that will attempt to unwrap the value in the `Option`. We already discussed the `unwrap` function and we found out it is not possible to create its safe implementation. We have nothing to return in case the optional value is missing. What we can do is to let the caller provide us with the default we shall use in case of none value. The provided value will play role of the default configuration value.

```
export const getOrElse = <A>(onNone: () => A) => (fa: Option<A>): A =>
  isSome(fa) ? fa.value : onNone();
```

When we apply `getOrElse("some default")` with the option containing a string, it actually gets us out of the `Option` context and we no longer have to `map` over the value. Instead, we can directly apply the greeting function as a pure transformation.

```
const doSomethingWithConfig =
  (getConfig: (key: string) => Option<string>) => pipe(
    getConfig("userName"),
    O.getOrElse("unknown person"),
    (userName) => `Hello, ${userName}`)
  );
```

As we said, `getOrElse` can transform `Option<A>` to `A`. Therefore, it's destroying the optional context a getting us back to the original type `A`.

Providing a default value is not the only way how to unwrap the `Option<A>`. For lots of types, there are values we consider defaults in appropriate situations. For strings, it could be an empty string, for numbers it might be `0` or maybe `1`, or for lists an empty list, etc. Practically speaking, it is again a kind of default value but this time it's a canonical one. We choose such canonical default values because they combine nicely with other values in the type. For instance, we'd choose an empty string because when we concat an empty string with any other string `s` we are guaranteed to get the string `s`. For numbers we mentioned `0` and `1` because these are neutral values in respect to an addition or multiplication.

Ability to combine values and having the default or neutral one can be encapsulated into an object that will form a structure called *monoid*. A destruction of an `Option` type using such a monoid structure will form basis for very useful operation called `foldMap`. But about that later!

## 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 implicitly 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 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 value into the `Either` context and to filter a value in the `Either`.

To remain consistent with the `Option` data type, we should implement the `of` function that puts a value into the context. There is a little problem with type of the `of` function. We need to decide on the type of the `E`. It should be such a type that correctly represents the fact that there can't be any error. If there can't be any error we should seek such a type that doesn't have any value. Fortunately, in Typescript there is such a type and it's the `never`. The `never` type is also called a *bottom* type. The opposite of `never` type is the `unknown` type because it contains all the possible values. Following the analogy, the `unknown` type is also called a *top* 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 additional

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 convertibility 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>`. For the `Option<A>` type, we would reuse them thanks to the fact we can freely convert between `Either<null, A>` and `Option<A>`.

Let's get back to the general `Either<E, A>` for a moment and compare it to the `Option<A>`. We

can see the `Either` type can store more information then the `Option` type. 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