Ben-Gurion University of the Negev

SOLUTION By Avi Ferdman

Principles of Programming Languages 202
Assignment 1

Responsible TA: Dor Litvak

Submission Date: 19/4/2020

Part 0: Preliminaries

Structure of a TypeScript Project

Every TypeScript assignment will come with two very important files:

- `package.json` - lists the dependencies of the project.
- `tsconfig.json` - specifies the TypeScript compiler options.

Before starting to work on your assignment, open a command prompt in your assignment folder and run `npm install` to install the dependencies.

What happens when you run `npm install` and the file `package.json` is present in the folder is the following:

1. `npm` will download all required modules and their dependencies from the internet into the folder `node_modules`
2. A file `package-lock.json` is created which lists the exact version of all the packages that have been installed.

What `tsconfig.json` controls is the way the TypeScript compiler (`tsc`) analyzes and typechecks the code in this project. We will use for all the assignments the strongest form of type-checking, which is called the "strict" mode of the `tsc` compiler.

Do not delete or change these files (*e.g.*, install new packages or change compiler options), as we will run your code against our own copy of those files, exactly the way we provide them.

If you change these files, your code may run on your machine but not when we test it, which may lead to a situation where you believe your code is correct, but you would fail to pass compilation when we grade the assignment (which means a grade of zero).

Testing Your Code

Every TypeScript assignment will have Mocha and Chai as dependencies for testing purposes. In order to run the tests, save your tests in the `test` directory in a file ending with `.test.ts` and run `npm test` from a command prompt. This will activate the execution of the tests you have specified in the test file and report the results of the tests in a very nice format.

An example test file `assignmentX.test.ts` might look like this:

```
import { expect } from "chai";
import { sum } from "./assignmentX";

describe("Assignment X", () => {
  it("sums two numbers", () => {
    expect(sum(1, 2)).to.equal(3);
  });
});
```

Every function you want to test must be `export`ed, for example, in `assignmentX.ts`, so that it can be `import`ed in the `.test.ts` file (and by our automatic test script when we grade the assignment).

```
export const sum = (a: number, b: number) => a + b;
```

What to Submit

You should submit a zip file called `<id1>_<id2>.zip` which has the following structure:

```
Part1.pdf
src
|- part2
   |- part2.ts
|- part3
   |- optional.ts
   |- result.ts
```

Make sure that when you extract the zip (using `unzip` on Linux), the result is flat, *i.e.*, not inside a folder. This structure is crucial for us to be able to import your code to our tests.

Part 1: Theoretical Questions

Submit the solution to this part as `Part1.pdf`.

1. List and explain 3 dimensions of variability across programming paradigms.
2. What are the types of the following functions:
   (a) `(x, y) => x + y`
   (b) `x => x[0]`
   (c) `(x, y) => x ? y : -y`

   Specify the most specific type in TypeScript you can provide for each function.
3. What are "shortcut semantics"? Explain and give an example.

Part 2: Fun with TypeScript

Complete the following functions in TypeScript in the file `src/part2/part2.ts`. Make sure to write your code using type annotations, and adhering to the Functional Paradigm.

Question 1

Write a **generic** function called `partition` that takes two parameters: a predicate and an array, and returns an array of arrays: the first array consists of all elements that satisfy the predicate, and the second array of the rest. For example:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
console.log(partition(x => x % 2 === 0, numbers)); // => [[2, 4, 6, 8], [1, 3, 5, 7, 9]];
```

Note: the order of the elements matters. They will appear in the same order in the resulting arrays as they were in the original array.

Question 2

Write a **generic** function called `mapMat` that works like `map` but on matrices. For example:

```
const mat = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

console.log(mapMat(x => x * x, mat)); // => [[ 1, 4, 9 ], [ 16, 25, 36 ], [ 49, 64, 81 ]]
```

Question 3

Write a **generic** function called `composeMany` that takes an array of $n \geq 0$ functions $f_i : T \rightarrow T$ and returns a function that is their composition. For example:

```
const squareAndHalf = composeMany([(x: number) => x / 2, (x: number) => x * x]);
console.log(squareAndHalf(5)); // => 12.5

const add3 = composeMany([(x: number) => x + 1, (x: number) => x + 1, (x: number) => x + 1]);
console.log(add3(5)); // => 8
```

Question 4

Given a Pokédex (a database of Pokémon of type `Pokemon[]`), write the following functions:

1. `maxSpeed` which returns an array of the Pokémon (as in a value of the `Pokemon` type) with the maximum "Speed" stat.
2. `grassTypes` which returns an array of the *English names* of all Grass type Pokémon sorted alphabetically.
3. `uniqueTypes` which returns an array of all the different Pokémon types (Grass, Fire, etc) sorted alphabetically.

Note: use the primitive `.sort` method on arrays to sort strings.

Part 3: A Fistful of Monads

"In functional programming, a **monad** is a design pattern that allows structuring programs generically while automating away boilerplate code needed by the program logic. Monads achieve this by providing their own data type (a particular type for each type of monad), which represents a specific form of computation, along with one procedure to wrap values of any basic type within the monad (yielding a **monadic value**) and another to compose functions that output monadic values (called **monadic functions**)." – Wikipedia

Let us look at an example of a specific monad called the Writer monad. The Writer monad encapsulates a value and a growing log of messages that accompany each operation done on the value.

First, as the definition above tells us, a monad provides a **data type**. So let us define this data type:

```
interface Writer<T> {
  tag: "Writer";
  value: T;
  log: string[];
}
```

Of course, as with any data type, we want a constructor and a type predicate:

```
const isWriter = <T>(x: any): x is Writer<T> => x.tag === "Writer";
const makeWriter = <T>(value: T): Writer<T> => ({ tag: "Writer", value: value, log: [] });
```

Now we will define two operations that accept a number and return a new Writer:

```
const square = (x: number): Writer<number> =>
    ({ tag: "Writer", value: x * x, log: [`${x} was squared`] });

const half = (x: number): Writer<number> =>
    ({ tag: "Writer", value: x / 2, log: [`${x} was halved`] });
```

What if we want to **compose** these two functions?

This is not easy to implement because `half` does not accept as a parameter a `Writer<number>` but a number - so that the output of `square` cannot be passed into `half`.

To enable composition of functions working with monads, and for every monad, there is a special function called `bind` (sometimes called `flatMap`).

Given a monad `M<T>`, `bind`'s signature is: `<T, U>(m: M<T>, f: (x: T) => M<U>) => M<U>`.

We can see that `bind` takes a monad of type `T`, extracts the value from the monad, acts on it, and returns a new monad. Let us implement `bind` for our Writer:

```
const bind = <T, U>(writer: Writer<T>, f: (x: T) => Writer<U>): Writer<U> => {
    const newWriter = f(writer.value);
    return {
        tag: "Writer",
        value: newWriter.value,
        log: writer.log.concat(newWriter.log)
    };
}
```

Let's go over using `bind` step-by-step:

First, we make an instance of our Writer:

```
const w1 = makeWriter(5);
console.log(w1); // => { tag: 'Writer', value: 5, log: [] }
```

Then, we use `bind` to extract the value from the writer, and pass it to the `square` function, giving us a new Writer:

```
const w2 = bind(w1, square);
console.log(w2); // => { tag: 'Writer', value: 25, log: [ '5 was squared.' ] }
```

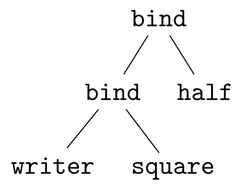
Now, we do the same with `half`:

```
const w3 = bind(w2, half);
console.log(w3); // => { tag: 'Writer', value: 12.5, log: [ '5 was squared.', '25 was halved.' ] }
```

If we don't use intermediate Writers, it would look something like this:

```
const writer = bind(bind(makeWriter(5), square), half);
```

This chain should look familiar:



This tree is exactly what `reduce` does! So we can actually use `reduce` to conveniently compose these functions:

```
import { reduce } from "ramda";
const writer = makeWriter(5);
const squareAndHalved = reduce(bind, writer, [square, half]);
console.log(squareAndHalved); // => { tag: 'Writer',
                              //      value: 12.5,
                              //      log: [ '5 was squared.', '25 was halved.' ] }
```

Your mission, now that you understand monads like a pro, is to implement two very well known monads: The `Optional<T>` and `Result<T>` monads.

The `Optional<T>` monad's purpose is to prevent us from encountering null pointer exceptions, or the equivalent risk in TypeScript, which consists of asking the value of a field within an `undefined` value.

When we write a function that could return either `undefined` or an actual value, we can return instead an `Optional` container. `Optional` is a disjoint union type of `Some` or `None`. For example:

```
const safeDiv = (x: number, y: number): Optional<number> =>
    y === 0 ? makeNone() : makeSome(x / y);

console.log(safeDiv(5, 2)); // => { tag: 'Some', value: 2.5 }
console.log(safeDiv(5, 0)); // => { tag: 'None' }
```

Now, if we design our functions to accept `Optional`s, we must check the two cases: `Some` or `None`.

Write the code to the following two questions in `src/part3/optional.ts`:

1. Implement the `Optional<T>` disjoint union type, including constructors and type predicates.
2. Implement `bind` for `Optional<T>`.

The `Optional` monad is very useful and appears in many languages such as Java, C++, Scala, Haskell, F# and OCaml. The "problem" with `Optional` is that it only tells us if a value exists, or not. What if we want to know why the computation failed? For that additional purpose, we introduce the `Result<T>` monad.

The `Result` monad is also a disjoint union between two types: `Ok` and `Failure`. The `Ok` type holds a value, and the `Failure` type holds a message. For example:

```
interface User {
  name: string;
  email: string;
  handle: string;
}

const validateName = (user: User): Result<User> =>
    user.name.length === 0 ? makeFailure("Name cannot be blank.") :
    user.name === "Bananas" ? makeFailure("Bananas is not a name.") :
    makeOk(user);

const user1 = { name: "Ben", email: "bene@post.bgu.ac.il", handle: "bene" };
const user2 = { name: "Bananas", email: "me@bananas.com", handle: "bene" };
console.log(validateName(user1)); // => { tag: 'Ok',
                                  //      value: { name: 'Ben',
                                  //               email: 'bene@post.bgu.ac.il',
                                  //               handle: 'bene' } }
console.log(validateName(user2)); // => { tag: 'Failure',
                                  //      message: 'Bananas is not a name' }
```

We can see that if our computation failed, we now have a message we can convey to our users, or have a result (in the specific case of the example, the success computation is of the same type as the input).

Write the code to the following four questions in `src/part3/result.ts`:

3. Implement the `Result<T>` disjoint union type, including constructors and type predicates.
4. Implement `bind` for `Result<T>`.

In the file `src/part3/result.ts` you are given a `User` interface and three validation functions. Your task is to write a user validation function which combines these three small validation functions in two ways:

5. Write a function `naiveValidateUser` which takes a `User` and returns a `Result<User>` **without** using `bind`.
6. Write a function `monadicValidateUser` which takes a `User` and returns a `Result<User>` using `bind`.

Good Luck and Have Fun!