Functional Programming 101

Ethan Kent

2021-05-28

What is FP?

According to the internet, it has something to do with these things:

- Function composition
- Mapping values to values rather than updating state
- First-class functions
- Pure functions
- Avoidance of side effects
- Use of recursion
- Referential transparency

What is FP? (continued)

- Use of higher-order functions
- Immutability
- Avoidance of shared state
- Based on mathematical functions
- Lazy evaluation
- Use of expressions rather than statements
- Declarative rather than imperative

This is confusing What is FP?

I mean, what is OOP? What is imperative programming? Sort of a loose definition. These things are all relevant.

For the purposes of this presentation, FP is the paradigm we're discussing during this presentation. (This is our first example of recursion.)

A half-baked dichotomy

A highly dubious series of contentions and oversimplifications: FP has two main "flavors"—

ML languages

- Really into types.
- The *M* word.

A half-baked dichotomy (continued)

LISP

- Usually dynamic.
- Code is data, code is ASTs, etc.

A half-baked dichotomy (continued)

We're going to be using `fp-ts` (and `io-ts`) for today's demo. That is Haskell influenced. Haskell is an ML language.

Annoyance-Driven Development. Let's define Functional Programming by trying stuff and getting frustrated.

Function composition / piping

Consider *nix:

```
$> cal | grep Su | sed 's/ //g'
# => SuMoTuWeThFrSa
```

Same idea, TypeScript version:

```typescript
interface toStringable {
  toString: () => string;
}

const stringify = (anything: toStringable): string => anything.toString();

const toUpperCase = (myString: string): string => myString.toUpperCase();

const evenArrayElements = <T>(myArray: T[]): T[] =>
  myArray.filter((_, i) => i % 2 === 0);

const arrayFromString = (myString: string): string[] => [...myString];

const stringFromArray = (myArray: toStringable[]): string =>
  myArray.map(stringify).join("");
```

Same idea, TypeScript version (continued)

```
// Annoying
const everyOtherLetter = (myString: string): string =>
  stringFromArray(evenArrayElements(arrayFromString(myString)));

// Annoying
const output = everyOtherLetter(
  toUpperCase(stringify("Is a monad a burrito? We may never know."))
);

// => "I  OA  URT?W A EE NW"
```

Pipe

```typescript
export function pipe<A>(a: A): A;
export function pipe<A, B>(a: A, ab: (a: A) => B): B;
export function pipe<A, B, C>(a: A, ab: (a: A) => B, bc: (b: B) => C): C;
export function pipe<A, B, C, D>(
  a: A,
  ab: (a: A) => B,
  bc: (b: B) => C,
  cd: (c: C) => D
): D;

export function pipe(
  a: unknown,
  ab?: Function,
  bc?: Function,
  cd?: Function
): unknown {
  switch (arguments.length) {
    case 1:
      return a;
    case 2:
      return ab!(a);
    case 3:
      return bc!(ab!(a));
    case 4:
      return cd!(bc!(ab!(a)));
  }
  return;
}
```

Pipe, continued

```
const everyOtherLetter = (myString: string) =>
  pipe(myString, arrayFromString, evenArrayElements, stringFromArray);

// A little annoying
const capitalizedEveryOtherLetter = (input: toStringable) =>
  pipe(input, stringify, toUpperCase, everyOtherLetter);
```

Flow

```
function flow<A extends ReadonlyArray<unknown>, B>(
  ab: (...a: A) => B
): (...a: A) => B;
function flow<A extends ReadonlyArray<unknown>, B, C>(
  ab: (...a: A) => B,
  bc: (b: B) => C
): (...a: A) => C;
function flow<A extends ReadonlyArray<unknown>, B, C, D>(
  ab: (...a: A) => B,
  bc: (b: B) => C,
  cd: (c: C) => D
): (...a: A) => D;

function flow(ab: Function, bc?: Function, cd?: Function): unknown {
  switch (arguments.length) {
    case 1:
      return ab;
    case 2:
      return function (this: unknown) {
        return bc!(ab.apply(this, arguments));
      };
    case 3:
      return function (this: unknown) {
        return cd!(bc!(ab.apply(this, arguments)));
      };
  }
  return;
}
```

Flow, continued

```
const capitalizedEveryOtherLetter = flow(
  stringify,
  toUpperCase,
  everyOtherLetter
);
```

Purity

Wikipedia:

> In computer programming, a pure function is a function that has the following properties:
>
> 1. The function return values are identical for identical arguments (no variation with local static variables, non-local variables, mutable reference arguments or input streams).
> 2. The function application has no side effects (no mutation of local static variables, non-local variables, mutable reference arguments or input/output streams).

Purity: Why is it good?

- Easy to test
- Easier to reason about
- Easier to debug

Obstacles to purity

Roughly: the fallen state of the world.

Purity in an impure world

```
// -----------------------------------------------------------------------------┐
// I am pure and need to be protected from the ugly truth of the world          |
const myPristineFunction = (input: string): string => input.toLowerCase(); //   |
// -----------------------------------------------------------------------------┘

// ------------------------------------------------------------------------┐
// I am the jaded, world-weary stuff that knows the hard truths of reality |
interface Success<A> { //                                                  |
  success: A; //                                                           |
  tag: "_success"; // necessary for runtime checks                         |
} //                                                                       |
//                                                                         |
interface Failure<F> { //                                                  |
  failure: F; //                                                           |
  tag: "_failure";  // necessary for runtime checks                        |
} //                                                                       |
//                                                                         |
type TheUglyWorld<A, F> = Success<A> | Failure<F>; //                      |
// ------------------------------------------------------------------------┘
```

Purity in an impure world (continued)

We want `myPristineFunction` to be delivered to the correct spot where it can do its job, never having to think of how or why it got there. It will be "lifted" into the nasty fallible world without having to get infected by it. How?

Purity in an impure world (continued)

First a little housekeeping. For historical reasons, what we're toying with here is called an `Either` type in `fp-ts` (inspired, as it is, by Haskell).

- A `left` is an error (think `left = sinister`).
- A `right` is a success (think `right = correct`).

Also note the `readonly`s: functional programming likes immutability.

```
interface Left<E> {
  readonly _tag: "Left";
  readonly left: E;
}

interface Right<A> {
  readonly _tag: "Right";
  readonly right: A;
}

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

Purity in an impure world (continued)

```
// Constructors
const right = <E, A>(right: A): Either<E, A> =>
  ({ _tag: "Right", right });

const left = <E, A>(left: E): Either<E, A> =>
  ({ _tag: "Left", left });

// Type predicates
const isLeft = <E, A>(input: Either<E, A>): input is Left<E> => input._tag === "Left";
const isRight = <E, A>(input: Either<E, A>): input is Right<A> => input._tag === "Right";
```

Purity in an impure world (continued)

```
const whoKnows: Either<string, number> =
  Math.random() > 0.5 ? right(42) : left("It's gone terribly wrong!");

// Farhenheit to celsius: first thing you do, subtract 32, five-ninths.

const naiveSubtractThirtyTwo = (input: number): number => input - 32;
const naiveFiveNinths = (input: number): number => (input * 5) / 9.0;

// Annoyance level: rising
const result = isRight(whoKnows)
  ? right(naiveFiveNinths(naiveSubtractThirtyTwo(whoKnows.right)))
  : whoKnows;
```

Purity in an impure world (continued)

Let's do a little helper. We want to abstract this "take my function, 'lift' it up into that weird `Either` thingy, then apply it."

It's kind of like `map`ing over an array, (except it's just one thing), so let's call that helper function `map`.

```
const map = <E, A, B>(input: Either<E, A>, func: (a: A) => B): Either<E, B> =>
  isRight(input) ? right(func(input.right)) : input;

const subtracted = map(whoKnows, naiveSubtractThirtyTwo);
const finalAnswer = map(subtracted, naiveFiveNinths);

console.log(finalAnswer);
```

That `pipe` thing would be nice, though

```
const naiveSubtractThirtyTwo = (input: number): number => input - 32;
const naiveFiveNinths = (input: number): number => (input * 5) / 9.0;

const map = <E, A, B>(input: Either<E, A>, func: (a: A) => B): Either<E, B> =>
  isRight(input) ? right(func(input.right)) : input;

// Argument of type 'Either<string, number>' is not assignable to parameter of type 'number'.
//   Type 'Left<string>' is not assignable to type 'number'.(2345)
pipe(whoKnows, naiveSubtractThirtyTwo, naiveFiveNinths);
```

Piping with `Either`, continued

It seems like we might need to use that `map` thing, but the type isn't right.

```
const naiveSubtractThirtyTwo = (input: number): number => input - 32;
const naiveFiveNinths = (input: number): number => (input * 5) / 9.0;

const map = <E, A, B>(input: Either<E, A>, func: (a: A) => B): Either<E, B> =>
  isRight(input) ? right(func(input.right)) : input;

// Argument of type 'Either<unknown, unknown>' is not assignable to parameter of type '(a: Either<string, number>) => unknown'.
//   Type 'Left<unknown>' is not assignable to type '(a: Either<string, number>) => unknown'.
//     Type 'Left<unknown>' provides no match for the signature '(a: Either<string, number>): unknown'.(2345)
pipe(whoKnows, map(naiveSubtractThirtyTwo), map(naiveFiveNinths));
```

Piping with `Either`, continued

The problem is that in order to pass the result from one computation to the next, we need to switch to curried functions.

Curried functions, you say?

Yes.

```
const addThreeNumbers = (
  first: number,
  second: number,
  third: number
): number => first + second + third;

const curriedAddThreeNumbers =
  (first: number) => (second: number) => (third: number): number =>
    first + second + third;

console.log(curriedAddThreeNumbers(7)(10)(22)); // => 39
```

Okay, but why?

A curried function returns a "partially applied" function.

```
const addTwoNumbersToSeven = curriedAddThreeNumbers(7);
const addTo44 = addTwoNumbersToSeven(37);

console.log(addTo44(18)); // => 62
```

Okay, but *why*?

Let's redefine `map` to do the following:

1. Take in a pure function (like before).
2. Curry it into a function that takes an `Either`.

So like before, it takes a pure function and an `Either` and eventually applies the pure function to the `right` side of the `Either`.

The difference is that you give it the pure function and you get back a new function that takes an `Either`:

```
const map = <E, A, B>(func: (param: A) => B) => (
  input: Either<E, A>
): Either<E, B> => (isRight(input) ? right(func(input.right)) : input);
```

OKAY, BUT WHY?

Here's one way to apply it:

```
const naiveSubtractThirtyTwo = (input: number): number => input - 32;

const whoKnows: Either<string, number> =
  Math.random() > 0.5 ? right(42) : left("It's gone terribly wrong!");

const subtracted = map(naiveSubtractThirtyTwo)(whoKnows);
```

But that also means we can do this:

```
const subtracted = pipe(whoKnows, map(naiveSubtractThirtyTwo));
```

Piping with `Either`, continued

```
pipe(whoKnows, map(naiveSubtractThirtyTwo), map(naiveFiveNinths));
```

Piping with `Either`, continued

Or we could compose `naiveSubtractThirtyTwo` and `naiveFiveNinths` with `flow`, then use that in our `map`.

```
pipe(
  whoKnows,
  map(
    flow(naiveSubtractThirtyTwo, naiveFiveNinths),
  ),
);
```

`Option`

Where an `Either` represents a fallible operation, an `Option` represents a nullable value.

```
interface Some<A> {
  _tag: "Some";
  some: A;
}

interface None {
  _tag: "None";
}

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

`Option`, continued

And the related functions:

```
// Type constructors
const some = <A>(some: A): Option<A> => ({ _tag: "Some", some });
const none: Option<never> = { _tag: "None" };

// Type predicates
const isSome = <A>(input: Option<A>): input is Some<A> => input._tag === "Some";
const isNone = (input: Option): input is None => input._tag === "None";

const map = <A, B>(func: (param: A) => B) => (input: Option<A>): Option<B> =>
  isSome(input) ? some(func(input.some)) : none;
```

An example with `Either` and `Option`

Imagine we make a database request that could error, and it's also possible the ID isn't found. Note at this point we're switching to the real `fp-ts` library.

```
// "Server"
const databaseLookup = (id: number): 404 | Record<string, string> = ({
  1: {name: "Peter Sagan", status: "Fading"},
  2: {name: "Tadej Pogačar", status: "Rising"},
  4: {nome: "Lance Armstrong", status: "Shameless"},
  7: {name: "Marianne Vos", status: "GOAT"},
}[id] || 404);

const networkEndpoint = (
  id: number
): 404 | Record<string, string> | undefined =>
  Math.random() < 0.67 ? databaseLookup(id) : undefined;
```

```
// Client-side networking
interface RiderData {
  name: string;
  status: string;
}

type RequestResult = E.Either<string, O.Option<RiderData>>;

const requestData: (id: number) => RequestResult = flow(
  networkEndpoint,
  E.fromNullable("Gremlins stuck in the machine"),
  E.map(O.fromPredicate((data) => data !== 404))
) as (id: number) => RequestResult;
```

```typescript
// Client-side rendering
const handle200 = (data: RiderData) => {
  console.log(`Found ${data.name}, status: ${data.status}`);
};

const handle404 = () => {
  console.log("Result not found");
};

const handle500 = (error: string) => {
  console.error(`Server error: ${error}`);
};

// WOW LOOK AT THIS SNAZZY CODE!
const render: (data: RequestResult) => void =
  E.match(handle500, O.match(handle404, handle200));

pipe(4, requestData, render);
```

Uh-oh...

```
$> ts-node server.ts
Found undefined, status: Shameless
```

We didn't validate

```
import * as t from "io-ts";

const RiderData = t.type({ name: t.string, status: t.string });
type RiderData = t.TypeOf<typeof RiderData>;

type RequestResult = E.Either<string | t.Errors, O.Option<RiderData>>;

const requestData: (id: number) => RequestResult =
  flow(
    networkEndpoint,
    E.fromNullable("Gremlins stuck in the machine"),
    E.chainW((response) =>
      response === 404
        ? E.of(O.none)
        : pipe(response, RiderData.decode, E.map(O.of))
    )
  );

const handle500 = (error: unknown) => {
  console.error(`Server error: ${JSON.stringify(error, null, 2)}`);
};
```

Something something category theory?

The stuff we've already looked at has this category theory stuff in it. Roughly speaking:

- Things that let you do that `map` bit are functors.
- Things that let you `fold` or `match` are semigroups.
  - Things that let you `fold` or `match` without having to pass in a starting value are monoids.

Umm, what about monads?

Remember where we mapped a validation function (that takes a value and returns an `Either`) and kind of `map`ed it onto that possible server error...

But that server error was already an `Either`...

And so we could have wound up with `Either<Either< ... >>`?

But because we used `chain` instead of `map`, it "smashed" the two `Either`s together? (Some languages call that `flatMap`, by the way.)

Yes I remember that. What are monads?

A monad is a thing like `Either` or `Option` that lets you:

- Do that flat map thing: lift a function up and smash the nested output.
- Stick a value into the `Either` or `Option` in the first place, i.e. `E.right(42)` for example.
- Plus follow some rules about how it's implemented.

Whoa, why those things?

So you can "lift" something up into a context like fallibility (with `Either`), nullability (with `Option`), async (with `Task`), side effects (with `IO`), fallible async (with `TaskEither`), dependency injection (with `Reader`), fallible async with dependency injection (with `ReaderTaskEither`), etc...

And then keep plugging in functions that don't know about that context...

And have the computations continue on the happy path but sensible error handling on the sad path...

And make all of that be enforced by the language and type system...