Functional Programming In TypeScript

Writing type-safe(r) code

Bertrand Caron
BFPG MeetUp - October the 8th, 2024

About me

- Principal engineer @ cascade.app (strategy software for planning and execution)
- PhD. in Computational Biochemistry
- 10+ years of experience as a software engineer
- Have worked in:
  - GIS (Geographical Information System)
  - FinTech (founder)
  - SaaS
- 6+ year of experience in TypeScript
- FP enthusiast
  - Going to BFPG for ~8 years

Context: Increasing popularity of JS and TS

[Source: StackOverflow 2024 Developer Survey, 65k respondents)

- 62.3% of developers have used JavaScript in the past year
- 38.5% of developers have used TypeScript in the past year
  - 5th most popular after JS (1), HTML/CSS (2), PY (3) and SQL (4)

A crash course in TS for FP enthusiasts

A primer on TypeScript: Primitive types

- **Primitive values**

```
const A: string = “Hello, BFPG”
const B: number = 42
const C: boolean = true
```

- **Special types + values**

```
const a: null = null
const b: undefined = undefined
const c: any = 42
const d: unknown = 42
const e: never = true // ← Compiler error, never does not have any members

const fail = (): never => {
     throw new Error(‘ERROR!’)
}

const exhaustiveUnion = (x: ‘a’) => {
     if (x === ‘a’) { return ‘HELLO’}
     else { console.log(`${x satisfies never}`)} // ← Compiler will throw an error x: ‘a’ |
‘b’
}
```

A primer on TypeScript: Composite types

- **Objects**

```
type animal = {
    name: string
    age: number
    vaccinated: boolean
}
```

- **Arrays / Tuples**

```
type X = string[]
type Y = [string, boolean]
```

- **Functions**

```
type F = (a: number, b: boolean) => [string, string]
```

FP concepts in TS: Algebraic data types

- **Sum types**

```
const maybeA: null | string = “hello”
```

  - Discriminated type unions

```
type Animal = {type: ‘dog’, name: ‘rex’ | ‘red’} | {type: ‘cat’,
age: number}
```

- **Product types**

```
type A = { a: string}
type B = { b: boolean}
const d: A & B = {a: ‘Hello”, b: 42}
```

- **Generics**

```
type Maybe<T> = T | undefined
```

FP concepts in TS: Generics, type mappers, ternaries

- **Generics**

```
type Maybe<T> = T | undefined
```

- **Type mappers (“type functions”)**

```
type MaybeObject<T extends object> = {
    [key in keyof T]: T[key] | undefined
}
```

- **Ternary expressions**

```
type MaybeBooleanObject<T extends objects> = {
    [key in keyof T]: T[key] extends boolean ? boolean | undefined : T[key]
}
```

FP concepts in TS: Recursive types

- Recursive types* (* with some caveats, limited recursion depth)

```
type Node = {
     value: number;
     children: Node[];
};
```

https://www.richard-towers.com/2023/03/11/typescripting-the-technical-interview.html

```
type SolveNextRow<row, placedQueens> =
    Solve<Next<S<row>>, placedQueens>, S<row>, placedQueens>

type Solve<candidates, row, placedQueens> = Equals<row, N> extends True
    ? candidates extends Cons<infer x, any>
        ? Cons<x, placedQueens>
        : Nil
    : candidates extends Cons<infer x, infer xs>
        ? SolveNextRow<row, Cons<x, placedQueens>> extends Nil
            ? Solve<xs, row, placedQueens>
            : SolveNextRow<row, Cons<x, placedQueens>>
        : Nil
```

Some extra oddities: Template literals, enums

- **Template literals**

```
type EmailType = ‘welcome’ | ‘unsubscribe’
type EmailRegion = ‘AU’ | ‘US’
type EmailId = `${EmailType}:${EmailRegion}`
// ‘welcome:AU’ | ‘unsubscribe:AU’ | ‘welcome:US’ | ‘unsubscribe:US’
```

- **Enums**

```
enum UserType {
    Admin = ‘Admin’,
    Viewer = ‘Viewer’,
}

// Both a const, and a type!
const mapping: Record<UserType, number> = {
    [UserType.Admin]: 1,
    [UserType.Viewer]: 2,
}
```

Example of cool TS type tricks

- Type function that reverse an arbitrary tuple

```
type Reverse<T extends unknown[]> = T extends [...infer U, infer P]
  ? [P, ...Reverse<U>]
  : []

type MyTuple = [string, boolean, number]
type ReversedTuple = Reverse<MyTuple> // ← [number, boolean, string]
```

https://github.com/bertrand-caron/ts-challenges
https://github.com/type-challenges/type-challenges

TypeScript: The not so good

- All TS code is valid JS code (i.e. all the “bad” JS ideas leak into TS)

```
typeof null === ‘object’
typeof []  === ‘object’
```

- Everything is mutable by default

```
const constArray: [] = []
constArray.push(1) // ← Not a compiler error!
```

- No type level concept of runtime exception

```
const throwingFunction = (): boolean => {
      if (Math.random() > 0.5) { return true}
      else {throw new Error(‘ERROR’)}
}
```

- No first-class level concept of IO, pure functions, side effects, etc.
- No pattern matching
- Type system can be very lenient, or bent
    - Type casting: `[] as unknown as number` (can be mitigated with linting)
    - Type guards: deferring type casting logic to developers

```
const isBoolean = (value: unknown): value is boolean => {return true}
```

Some cool FP libraries in TS

`fp-ts`: Typed functional programming in TypeScript

Typed functional programming in TypeScript

`fp-ts` provides developers with popular patterns and reliable abstractions from typed functional languages in TypeScript.

Disclaimer. Teaching functional programming is out of scope of this project, so the documentation assumes you already know what FP is.

Core Concepts

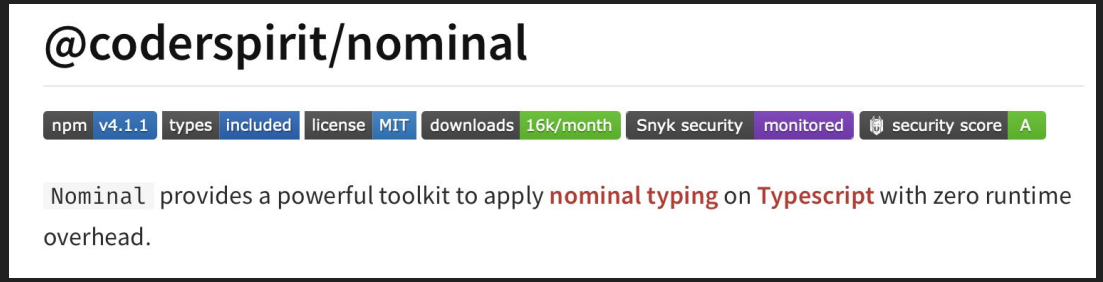
The goal of `fp-ts` is to empower developers to write pure FP apps and libraries built atop higher order abstractions. It includes the most popular data types, type classes, and abstractions from languages like Haskell, PureScript, and Scala.

Example: `Option` monad in `fp-ts`

```ts
import * as O from 'fp-ts/Option'
import { pipe } from 'fp-ts/function'

const double = (n: number): number => n * 2

export const imperative = (as: ReadonlyArray<number>):
string => {
  const head = (as: ReadonlyArray<number>): number => {
    if (as.length === 0) {
      throw new Error()
    }
    return as[0]
  }
  const inverse = (n: number): number => {
    if (n === 0) {
      throw new Error()
    }
    return 1 / n
  }
  try {
    return `Result is ${inverse(double(head(as)))}`
  } catch (e) {
    return 'no result'
  }
}
```

```ts
import * as O from 'fp-ts/Option'
import { pipe } from 'fp-ts/function'

const double = (n: number): number => n * 2

export const functional = (as: ReadonlyArray<number>):
string => {
  const head = <A>(as: ReadonlyArray<A>): O.Option<A> =>
(as.length === 0 ? O.none : O.some(as[0]))
  const inverse = (n: number): O.Option<number> => (n ===
0 ? O.none : O.some(1 / n))
  return pipe(
    as,
    head,
    O.map(double),
    O.flatMap(inverse),
    O.match(
      () => 'no result', // onNone handler
      (head) => `Result is ${head}` // onSome handler
    )
  )
}
```

@coderspirit/nominal: Nominal typing




```
import { WithBrand } from '@coderspirit/nominal'

type Email = WithBrand<string, 'Email'>
type Username = WithBrand<string, 'Username'>

const email: Email = 'admin@acme.com' as Email // Ok
const user: Username = 'admin' as Username // Ok
const text: string = email // OK
const anotherText: string = user // Ok

const eMail: Email = 'admin@acme.com' // Error, as we don't have a cast here
  const mail: Email = user // Error, as the brands don't match
```

gvergnaud/ts-pattern: Pattern matching for TypeScript

TS-Pattern

The exhaustive Pattern Matching library for TypeScript with smart type inference.


```ts
import { match, P } from 'ts-pattern';

type Data =
  | { type: 'text'; content: string }
  | { type: 'img'; src: string };

type Result =
  | { type: 'ok'; data: Data }
  | { type: 'error'; error: Error };

const result: Result = ...;

const html = match(result)
  .with({ type: 'error' }, () => <p>Oups! An error occured</p>)
  .with({ type: 'ok', data: { type: 'text' } }, (res) => <p>{res.data.content}</p>)
  .with({ type: 'ok', data: { type: 'img', src: P.select() } }, (src) => <img src={src} />)
  .exhaustive();
```

FP in TS: My personal take

Aspect 1: Tooling

1. Rely on existing tools to improve code quality and compiler strictness
    - tsconfig.json: {strict: “true, noUncheckedIndexedAccess: true}
    - `eslint` + `typescript-eslint` (with “strict” ruleset)
        - Disallow type casting, etc.
        - Avoid “JSisms”: Boolean casting, ==, etc.

2. Use external libraries that augment the capabilities of the language
    - `gvergnaud/ts-pattern` (pattern matching)
    - `@coderspirit/nominal` (nominal typing)
    - `fp-ts` (everything else)
    - [Insert your favourite validation library here]

Aspect 2: Code styling

1. Prefer immutable data structures over mutable ones
   - `Readonly<>` (types), `Object.freeze()` (values)
2. Use discriminated unions to exclude impossible types
3. Limit use of OOP constructs and patterns
   - Use immutable class instances (i.e. all properties are `readonly` and set once by the constructors)
4. Prefer expressions over statements
   - reduce() or map() instead of `for/while` loops
   - ternaries instead of `if/else`
   - Always have matching `else` clause for any `if` statement
5. Prefer “FP-style” functions
   - Write pure functions (when possible)
   - Segregate impure function, and try typing them (e.g. errors as values rather than exceptions)
   - Push side-effect to the boundary of the application
6. Validate your inputs and outputs!
   - The type system is only as good as its weakest link!
7. Use FP architectural patterns (e.g. event sourcing) to complement your FP code

TS === great FP language?!

PROS

- Huge community === huge impact!
- Powerful and flexible type system
  - Type system is turing-complete!
  - Getting better everyday (very tight release schedule!)
- Commercial language
  - Good “get s**t done” to correctness ratio (if done correctly)

CONS

- Based on a weird language
  - Inherits all of its idiosyncracies
- Non-trivial type model (set theory)
  - Weird edge cases
- TS codebase is only as good as its developers
  - Language allows some very good, but also very poor practices
  - Type guards and type casting can hide huge gaps in type safety

Thank you for your attention

Questions?

Functional paradigms > functional code?

- Correct code does not mean bug-free
- Functional paradigms help compensate for the limitation of the language
  - Event sourcing: Append-only ledger of events (instead of mutable databases)