3 suggestions to start with FP

in `Typescript` and `fp-ts`

Intro to FP

OO	FP
Dependency Injection	Function
Visitor Pattern	Function
Decorator	Function
Middleware	Function
Factory	Function

Reality

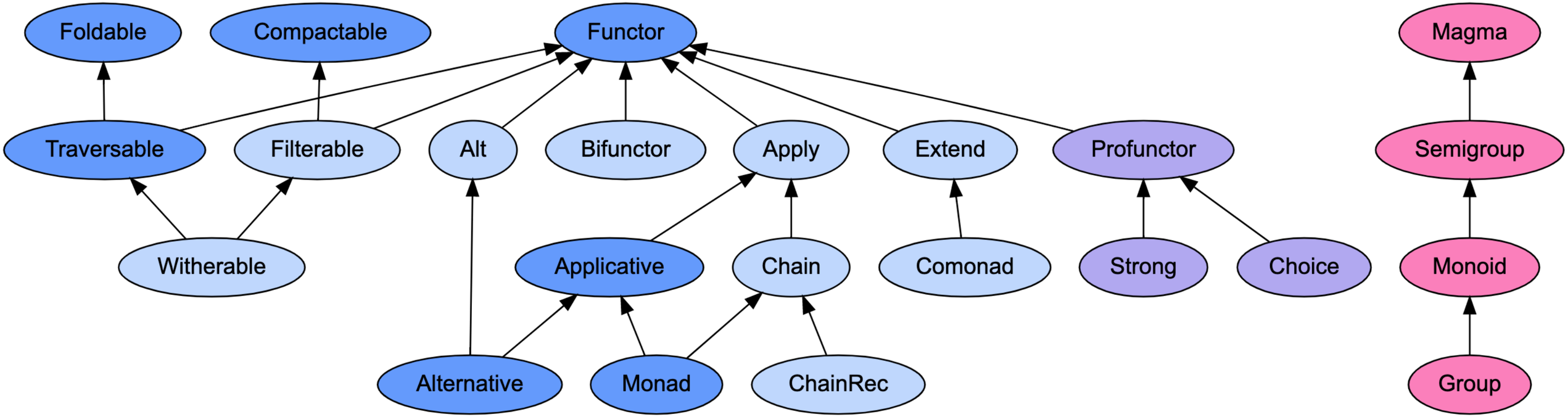
JS/TS	FP
Class	Function
Instance	Function
Dependency Injection	Function
Decorator	Function
Middleware	Function
node APIs	Function
DOM APIs	Function


HKT


ADT

Monoid

Functor


Foldable
Compactable
Functor
Magma
Traversable
Filterable
Alt
Bifunctor
Apply
Extend
Profunctor
Semigroup
Witherable
Applicative
Chain
Comonad
Strong
Choice
Monoid
Alternative
Monad
ChainRec
Group

Agenda

1. Typed functional programming
2. Designing with types
3. Functional error handling

Typed functional programming in TypeScript

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

Typed functional programming

1. forget about a lot of concepts: classes, inheritance, decorators...
2. focus on **data** and **functions** only
3. profit!

Data

Data

→ plain values, no behavior attached

Data

→ plain values, no behavior attached

→ generally immutable:

Data

→ plain values, no behavior attached

→ generally immutable:

→ different value $\iff$ different reference

Functions

Functions

→ in the mathematical sense: no "impure" or "side-effectful" functions

Functions

→ in the mathematical sense: no "impure" or "side-effectful" functions

→ relation associating each element of the *domain* to a single element of the *codomain*

Functions

→ in the mathematical sense: no "impure" or "side-effectful" functions

→ relation associating each element of the *domain* to a single element of the *codomain*

→ "total": defined for every value of the *domain*

Typed functional programming in TS - Rules

Typed functional programming in TS - Rules

→ `"strict": true` in *tsconfig.json*

Typed functional programming in TS - Rules

→ `"strict": true` in *tsconfig.json*

→ annotate function return types (`void` not allowed!)

Typed functional programming in TS - Rules

→ `"strict": true` in *tsconfig.json*

→ annotate function return types (`void` not allowed!)

→ define `total` functions

Typed functional programming in TS - Rules

→ `"strict": true` in *tsconfig.json*

→ annotate function return types (`void` not allowed!)

→ define `total` functions

→ when possible, prefer `polymorphic` functions

Typed functional programming in TS - Rules

→ `"strict": true` in *tsconfig.json*

→ annotate function return types (`void` not allowed!)

→ define `total` functions

→ when possible, prefer `polymorphic` functions

→ use "type driven development": `declare function`

Typed ~~functional~~ programming in TS - Rules

→ `"strict": true` in *tsconfig.json*

→ annotate function return types (`void` not allowed!)

→ define `total` functions

→ when possible, prefer `polymorphic` functions

→ use "type driven development": `declare function`

Are these signatures OK?

```
declare function sum(a: number, b: number): number

declare function length(a: string): number

declare function replicate<A>(a: A, n: number): Array<A>
```

Are these signatures OK?

```
declare function parseInt2(s: string): number

declare function head<A>(as: Array<A>): A
```

Suggestion #1

Good FP in TS rules are just good TS rules

To start with FP in TS,

let the signatures talk straight

Designing with types

`SubscriptionStatus`

Designing with types

`SubscriptionStatus`

→ for the user to access the service, email must be verified

Designing with types

`SubscriptionStatus`

→ for the user to access the service, email must be verified

→ the "Solo" plan is completely free

Designing with types

`SubscriptionStatus`

→ for the user to access the service, email must be verified

→ the "Solo" plan is completely free

→ the "Business" plan costs money, so any "Business" user should have selected a payment mode

Product type

```
type SubscriptionStatus = {
  emailVerified: boolean
  subscriptionTier: "Solo" | "Business"
  paymentMode?: "Cash" | "Credit"
}
```

Product type

```
type SubscriptionStatus = {
  emailVerified: boolean // # = 2
  subscriptionTier: "Solo" | "Business" // # = 2
  paymentMode?: "Cash" | "Credit" // # = 3
}
```

Product type

```
type SubscriptionStatus = {
  emailVerified: boolean // # = 2
  subscriptionTier: "Solo" | "Business" // # = 2
  paymentMode?: "Cash" | "Credit" // # = 3
}
```

representable states: **2 * 2 * 3 = 12**

Impossible! ❌

```
const impossibleStatus: SubscriptionStatus = {
  emailVerified: false,
  subscriptionTier: "Business",
  paymentMode: undefined
}
```

Make impossible states irrepresentable

```
interface NotVerified {
  type: "NotVerified"
}

interface Solo {
  type: "Solo"
}

interface Business {
  type: "Business"
  paymentMode: "Cash" | "Credit"
}

type SubscriptionStatus = NotVerified | Solo | Business
```

Sum type

```
interface NotVerified // # = 1

interface Solo // # = 1

interface Business // # = 2

type SubscriptionStatus = NotVerified | Solo | Business
```

Sum type

```
interface NotVerified // # = 1

interface Solo // # = 1

interface Business // # = 2

type SubscriptionStatus = NotVerified | Solo | Business
```

representable states: **1 + 1 + 2 = 4** ✅

Exhaustiveness checking

```
function isCool(status: SubscriptionStatus): boolean {
  switch (status.type) {
    case "Business":
      return status.paymentMode === "Credit";
    case "Solo":
      return true;
    case "NotVerified":
      return false;
  }
}
```

fold/match function

```
declare function foldSubscriptionStatus<T>(
  status: SubscriptionStatus,
  onNotVerified: () => T,
  onSolo: () => T,
  onBusiness: (paymentMode: Business["paymentMode"]) => T
): T
```

Suggestion #2

Safe code requires safe design upfront

To start with FP in TS,

reason in terms of sum and product types

Function composition

```
function toUpperCase(s: string): string
function toChars(s: string): Array<string>

const result1 = toChars(toUpperCase("foo"))
```

Function composition with |>

```
function toUpperCase(s: string): string
function toChars(s: string): Array<string>

const result2 =
    "foo"
    |> toUpperCase
    |> toChars
```

Function composition in fp-ts

```
function toUpperCase(s: string): string
function toChars(s: string): Array<string>

const result3 = pipe(
    "foo",
    toUpperCase,
    toChars
)
```

Error handling - the FP way

fp-ts Option

```
import * as O from 'fp-ts/lib/Option'
```

Error handling - the FP way

fp-ts Option

```
function parseInt2(s: string): O.Option<number>
```

Error handling - the FP way

```
function parseInt2(s: string): O.Option<number> {
  const parsed = parseInt(s)
  return isNaN(parsed) ? O.none : O.some(parsed)
}
```

Error handling - the FP way

```ts
function head<A>(as: Array<A>): O.Option<A> {
  return as.length >= 1 ? O.some(as[0]) : O.none
}
```

```ts
(import { head } from 'fp-ts/lib/Array')
```

Option is a sum type

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

interface None {
  _tag: "None"
}

interface Some<A> {
  _tag: "Some"
  value: A
}
```

Error handling - the FP way

```ts
type User = { first: string; last: string }
declare const ranking: Array<User>

import * as O from "fp-ts/lib/Option"

const winner1 = pipe(
  ranking,
  head,
  O.fold(
    () => "Everybody wins!",
    user => `${user.first} ${user.last}`
  )
)
```

Interoperating with non-FP code

```
declare const possiblyNull: string | null

O.fromNullable(possiblyNull) // => Option<string>

const chars: Array<string> = pipe(
  possiblyNull,
  O.fromNullable,
  O.map(toUpperCase),
  O.map(toChars),
  O.fold(() => [], identity) // same as O.getOrElse(() => [])
)
```

Why is Option better?

compared to e.g. `A | null`

Why is Option better?

compared to e.g. `A | null`

→ "standard" in FP, e.g. Scala or Purescript: `data`

```
Maybe a = Nothing | Just a
```

Why is Option better?

compared to e.g. `A | null`

→ "standard" in FP, e.g. Scala or Purescript: `data Maybe a = Nothing | Just a`

→ shares the same interface as any other FP data type - learn once, use anywhere

Why is X from fp-ts better?

```
const result1: number = pipe(
  "not number",
  parseInt2,
  O.getOrElse(() => 0)
)

import * as E from "fp-ts/lib/Either"

type Fail = "Invalid" | "TooMuch"

function parseInt3(s: string): E.Either<Fail, number>
```

Why is X from fp-ts better?

```
  pipe(
    "not number",
-   parseInt2,
+   parseInt3,
-   O.map(n => n * 2),
+   E.map(n => n * 2),
-   O.getOrElse(() => 0)
+   E.getOrElse(() => 0)
  )
```

Suggestion #3

Write 100% FP code from day 1 is not possible

In fp-ts,

start from low hanging fruits like Option

Summary

To start with FP in TS, using fp-ts

Summary

To start with FP in TS, using fp-ts

1. Let the signatures talk straight

Summary

To start with FP in TS, using fp-ts

1. Let the signatures talk straight
2. Reason in terms of `sum` and `product` types

Summary

To start with FP in TS, using fp-ts

1. Let the signatures talk straight
2. Reason in terms of `sum` and `product` types
3. Start from the low hanging fruits like `Option`

buildo
software that fits

@giogonzo

any question?