TypeScript と関数型プログラミング

2024.9.7

株式会社 一休
伊藤 直也

先に総論

- 関数型プログラミングのエッセンスを取り入れることでより良いプログラミングができる言語
  - React などが良い例
  - 高機能な型システム、関数リテラル、高階関数、クロージャ、map & reduce、分割代入などのシンタックスサポート
  - 代数的データ型もエミュレートはできる
- Haskell や Scala のような関数型プログラミング言語に比較すると機能は不足している
  - それが今後、改良されていくロードマップは考えにくい?
- 例外やエラーを含む副作用をどう扱うか、永続データをどう扱うかは言語が指示してくれない
  - Haskell や Scala、F# などと同様のアプローチをとろうにも、言語のサポート機構が不十分
  - サードパーティライブラリで「補う」程度が限度か

そもそも「関数型プログラミング」って?

- 命令型 / 関数型 ?
  - 文 (戻り値を伴わない) で計算を構成する … 命令型
  - 式 (戻り値を伴う) で計算を宣言する … 関数型
- 関数型プログラミングっぽいイディオム?
  - map / reduce
  - 部分適用、カリー化
- 型システム?
  - 代数的データ型とパターンマッチ
  - 文脈付き計算、モナド
- イミュータブル / ミュータブル?
  - 永続データ

命令型: for 文で書く

```
int total = 0;

for (int i = 1; i <= n; i++) {
    total += i;
}
```

- for文と代入文で、繰り返し演算結果を書き込む (代入する) 命令を行っている
- 計算機への命令 … 命令型
- 文で計算を構成すると、命令的になる

関数型: 再帰的に関数を適用する

Haskell

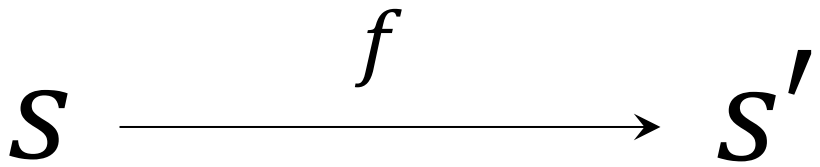
```haskell
let s = foldl (+) 0 [1 .. n]
```

TypeScript

```typescript
let s = [1, 2, 3, 4, 5].reduce((acc, i) => acc + i, 0)
```

- 式により計算を宣言する
- 式は必ず値を戻す。その値に再び関数を適用する
- 式で計算を構成すると、宣言的になる

関数によるオブジェクトの状態遷移



ある状態 s に関数 f を適用して、別の状態 s' を得る

TypeScript

```
// 顧客をアーカイブ状態にする
export const archiveCustomer = (customer: Customer): Customer => ({
  ...customer,
  archived: true,
})
```

```typescript
customer.archive()
```

```typescript
const archived = archiveCustomer(customer)
```

- 命令的に書く
  - オブジェクトの内部状態を書き換える命令を行うことで、状態を変化させる
  - 状態の変化が暗黙的

- 関数的 (宣言的) に書く
  - 引数のオブジェクトに関数を適用して、状態遷移後のオブジェクトを得る
  - 遷移前の状態は必ず引数に現れ、遷移後の状態は戻り値に現れる

宣言的に関数を記述するのはやりやすい言語

- 第一級関数、高階関数、クロージャ
- 関数リテラル
- **分割代入**
- map & reduce
- 静的型付け

「式で計算を宣言的に定義する」という点においては十分な機能を有する
特に分割代入のシンタックスは、イミュータブルな関数を記述するのに役に立つ

カリー化は言語の機能としてはないが、自分で記述すれば良い

- クロージャを返す関数として定義すればよい
- 部分適用による Dependency Injection などにも重要

```
export type getTagSortOrder = ({ groupId }: { groupId: RestaurantGroupId })
  => ResultAsync<number, PrismaClientError>

export const getTagSortOrder =
  ({ prisma }: applicationContext): getTagSortOrder =>
  ({ groupId }) =>
    ResultAsync.fromPromise(
      prisma.tag
        .aggregate({
          _max: {
            sortOrder: true,
          },
          where: { groupId },
        })
        .then((x) => (x._max.sortOrder ? x._max.sortOrder + 1 : 1)),
      PrismaClientError
    )
```

TypeScript の型システム

- 構造的部分型 (structural subype) で高機能な型システム
  - ジェネリクス
  - 合併型 (Union) / 交差型 (Intersection)
  - 動的な型定義 … Conditioinal Types、Utility Types、Mapped Types、infer ....
- 以下は、エミュレートできる
  - 代数的データ型とパターンマッチ
- ではあるが、以下はない
  - Optional や Result などを抽象的に扱う仕組み … モナドや型クラス

代数的データ型 … 静的型付けの関数型言語でデータ構造を定義する手段

```haskell
data Bool = True | False
```

Haskell

```haskell
data Maybe a = Nothing | Just a
```

Haskell

```haskell
data UnionFind = UnionFind { parent :: IM.IntMap Int, size :: IM.IntMap Int }
```

Haskell

- Haskell などの言語ではデータ構造を定義するのに代数的データ型で表現する
- 型を組み合わせるのに積 (AND) だけでなく 和 (OR) が使える

和で組み合わせて構築したものは、パターンマッチで分解

Haskell

```haskell
data Maybe a = Nothing | Just a
```

Haskell

```haskell
main = do
  let someValue :: Maybe String
      someValue = ...

  case someValue of
    Just s -> putStrLn (s ++ ", naoya")
    Nothing -> putStrLn "Farewell"
```

Just String か Nothing のどちらか

Maybe 型をパターンマッチで分解する

TypeScript は組み込みでは代数的データ型をサポートしてないが、エミュレートできる

Haskell

```haskell
data List a = Empty | Cons a (List a)
```

TypeScript

```typescript
interface Empty {
  kind: "Empty"
}

interface Cons<T> {
  kind: "Cons"
  head: T
  tail: List<T>
}

export type List<T> = Empty | Cons<T>
```

リテラル型でタグをつけておく

ユニオンで組み合わせた型

TypeScript

```ts
// List<T> への map 関数を実装
type map = <T, U>(f: (a: T) => U, xs: List<T>) => List<U>

export const map: map = (f, xs) => {
  switch (xs.kind) {
    case "Empty":
      return Empty()
    case "Cons":
      return Cons(f(xs.head), map(f, xs.tail))
    default:
      assertNever(xs)
  }
}

export function assertNever(_: never): never {
  throw new Error()
}

console.log(map(i => i * 2, myList))
```

タグに応じて分解 (リテラル型なのでちゃんと型が効く)

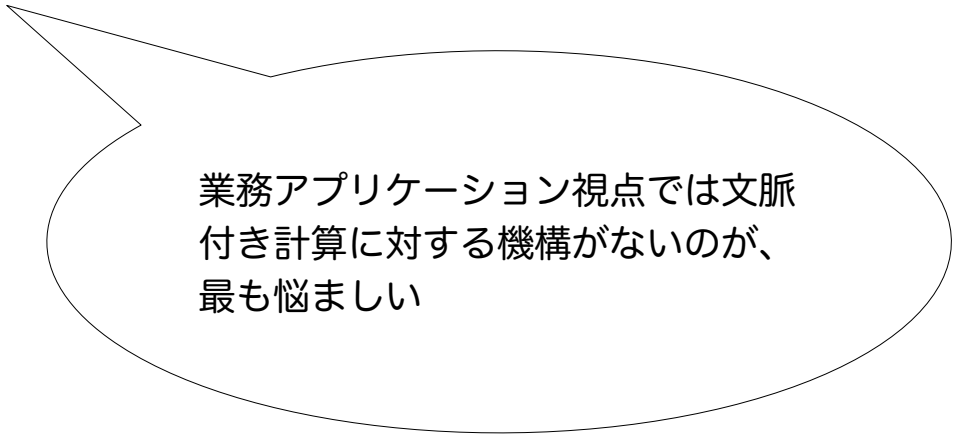
これを入れることで網羅性チェックが効く

ここまで見ると関数型スタイルでやっていくのに十分に見える?

- 宣言的イディオムは書きやすい
- 分割代入でイミュータブルにデータ更新するのもやりやすい
- 高機能な型システム
- カリー化や代数的データ型っぽいこともできる

足りないと思っている点 (主観)

- Optional、Result など「文脈付き計算」のための型がない。またそれを扱う支援機構がない
- 永続データ構造が基本にはなっていない



Result 型 … 成功 / 失敗の分岐を型で表現できる

TypeScript

```ts
export function toColor(value: string): Result<Color, ValidationError> {
  return /^#[0-9a-f]{3}([0-9a-f]{3})?$/i.test(value)
    ? ok(value as Color)
    : err(new ValidationError('色の値が不正です。#FFFFFF形式で指定してください'))
}
```

neverthrow、fp-ts、Effect などが同様の型をライブラリで提供する

Result型で、失敗の可能性のある計算を一本道に合成することができる

```ts
import { Result, ok, err } from 'neverthrow'

function itsUnder100(n: number): Result<number, Error> {
  return n <= 100 ? ok(n) : err(new Error('100より大きい数字です'))
}

function itsEven(n: number): Result<number, Error> {
  return n % 2 == 0 ? ok(n) : err(new Error('奇数です'))
}

function itsPositive(n: number): Result<number, Error> {
  return n > 0 ? ok(n) : err(new Error('負数です'))
}

const result = ok(96).andThen(itsUnder100).andThen(itsEven).andThen(itsPositive)

result.match(
  (n) => console.log(n),
  (error) => { throw error }
)
```

しかし、Result 型に入った値が入れ子になるとややこしい

```
export const Tag = (input: TagInput): Result<Tag, ValidationError> => {
  const tagId = TagId(input.id)
  const groupId = RestaurantGroupId(input.groupId)
  const label = TagLabel(input.label)
  const icon =
    input.icon && input.iconType ?
      TagIcon({
        symbol: input.icon,
        type: input.iconType,
        color: input.color,
      })
    : ok(NoIcon())
  const sortOrder = FractionalIndex(input.sortOrder)

  const values = Result.combine(tuple(tagId, groupId, label, icon, sortOrder))

  return values.map(([id, groupId, label, icon, sortOrder]) => ({
    ...input,
    id,
    groupId,
    label,
    icon,
    sortOrder,
  }))
}
```

値がだいたい Result に入っているため、複数 Result があると合成してフラットにする必要があり面倒な作業になってくる

「コンテナに入った文脈つきの値」を扱う、組み込みの機能がある言語なら…

Haskell

```haskell
makeTagId :: String -> Either ValidationError TagId
makeTagId tagId
  | null tagId = Left (ValidationError "tagId is empty")
  | otherwise = Right (TagId tagId)

makeTag :: String -> String -> Int -> Either ValidationError Tag
makeTag id gid order = do
  tagId <- makeTagId id
  groupId <- makeGroupId gid
  sortOrder <- makeSortOder order

  return (Tag tagId groupId sortOrder)
```

例えばモナドには、入れ子になったコードを平坦化する効用がある

この手の機構があれば認知負荷低く実装できるが、残念ながら TypeScript にはない

永続データプログラミング、永続データ構造

筆者の関数型言語の定義

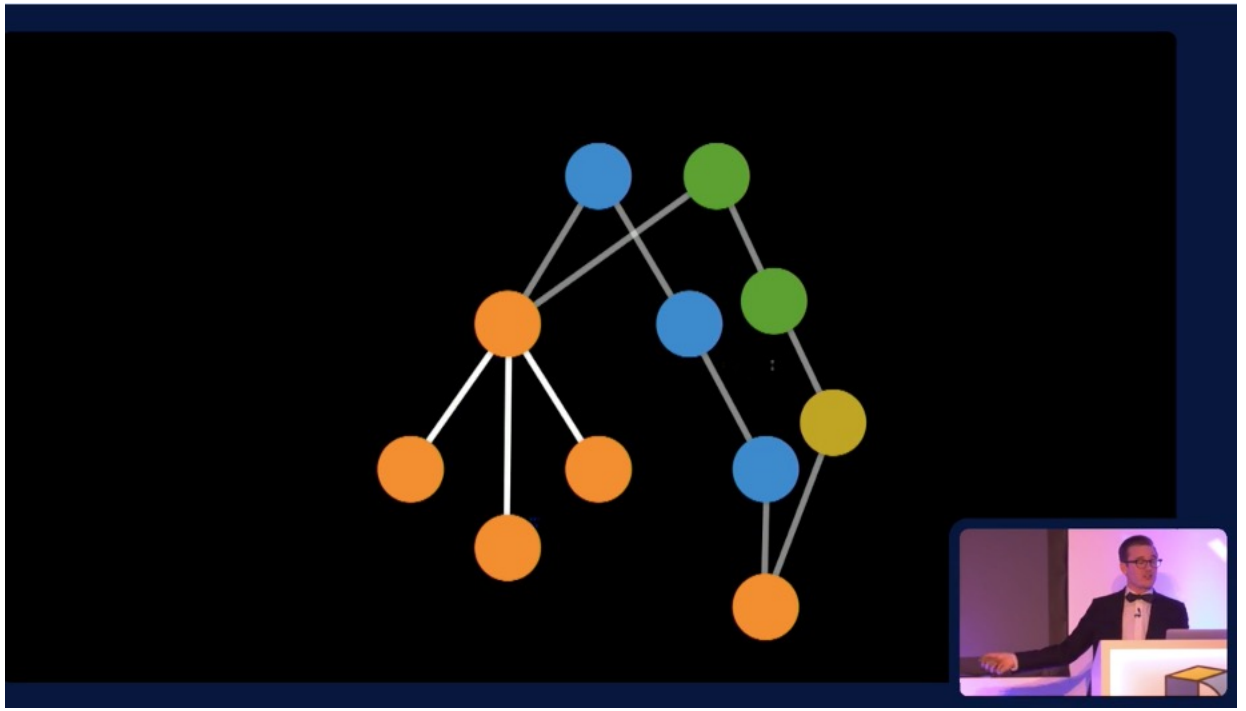
筆者の関数プログラミングの定義、すなわちこの特集での定義は、「永続データプログラミング」です。永続データとは、破壊できないデータ、つまり再代入できないデータのことです。そして、永続データを駆使して問題を解くのが永続データプログラミングです。

また関数型言語とは、永続データプログラミングを奨励し支援している言語のことです。関数型言語では、再代入の機能がないか、再代入の使用は限定されています。筆者の定義はかなり厳しいほうだと言えます。

［入門］関数プログラミング―質の高いコードをすばやく直感的に書ける！
https://gihyo.jp/dev/feature/01/functional-prog/0001

永続データ構造

- 変更される際に変更前のバージョンを常に保持するデータ構造
- Structural Sharing により、効率的にそれらのデータを保持する
- イミュータブルにデータ構造を変更するとき、全体をコピーしなくて済む



https://vimeo.com/166790294

Haskell の基本的なデータ構造は永続データ構造

Haskell

```haskell
main :: IO ()
main = do
  let xs = scanl' (flip Set.insert) Set.empty [1 .. 10 ^ 5]

  print $ Set.size (last xs)
```

10^5 個の Set が作られるがその都度全体が毎回コピーされるわけではない

永続データ構造ならイミュータブルにデータを扱いつつパフォーマンスと両立できる

TypeScript と永続データ構造

- 組み込みのデータ構造は永続データ構造ではなく、基本はミュータブル
- Immutable.js など永続データ構造を提供するライブラリがあるが、いまいち流行ってない
  - 業務アプリケーショでは永続データ構造を使う必要のあるほど、パフォーマンス要件が厳しい場面が少ないのかも

(再掲) 総論

- 関数型プログラミングのエッセンスを取り入れることでより良いプログラミングができる言語
  - React などが良い例
  - 高機能な型システム、関数リテラル、高階関数、クロージャ、map & reduce、分割代入などのシンタックスサポート
  - 代数的データ型もエミュレートはできる
- Haskell や Scala のような関数型プログラミング言語に比較すると機能は不足している
  - それが今後、改良されていくロードマップは考えにくい?
- 例外やエラーを含む副作用をどう扱うか、永続データをどう扱うかは言語が指示してくれない
  - Haskell や Scala、F# などと同様のアプローチをとろうにも、言語のサポート機構が不十分
  - サードパーティライブラリで「補う」程度が限度か