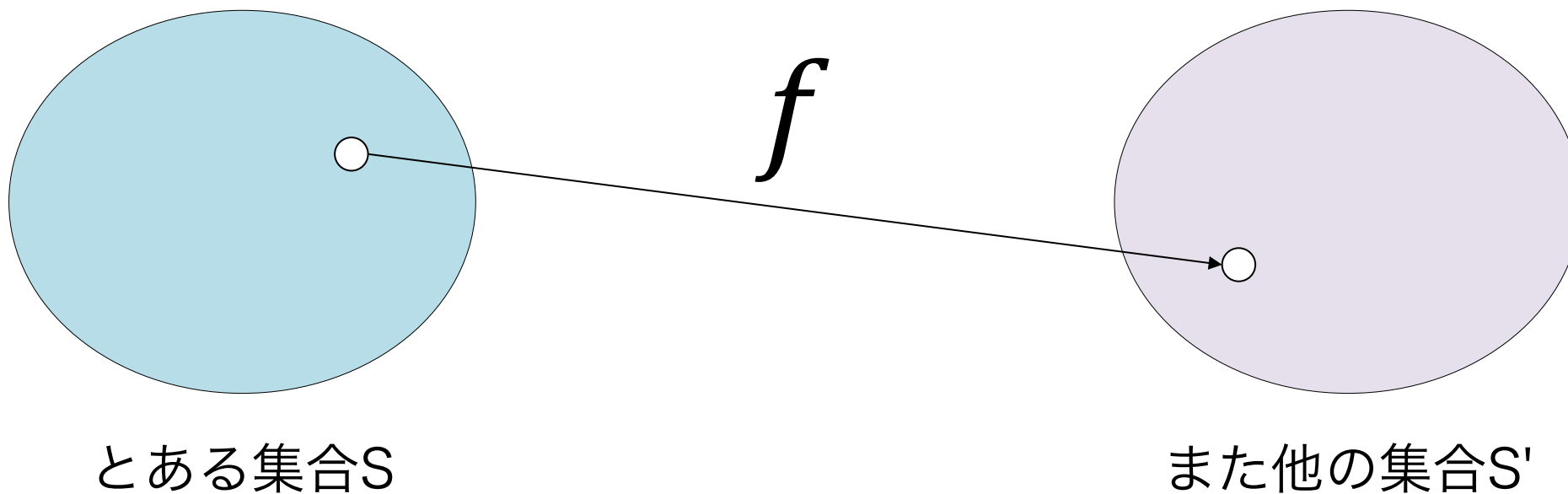


# 関数型プログラミングと型システムのメンタルモデル

株式会社 一休

伊藤 直也

ある集合に属する値を、関数  $f$  により、また他の集合に属する値に写す



これがどんな集合か記述するのが「型」

# なぜこの話？

- 昨今のプログラミング言語における「型」をどう捉えるか？
  - かくいう私も、00年代頃は「型付けめんどくさいなー」と思っていた
  - 最近は、当時とは全く異なるメンタルモデルを持っている
- プログラミング言語、フレームワークにおける関数型プログラミングからの影響
- Haskell を趣味で書くようになって、メンタルモデルが更新された
  - 静的型付け、型推論、関数型プログラミング、文脈計算…
  - なお今日の話は Haskell 特有の話ではありません。そのため主に TypeScript の例も交えて話します

# プログラミングの「メンタルモデル」

今日の話を面白いと思っていただけたら、ぜひ以下の文章にも目を通して見て下さい

- はじめに #Haskell – Qiita, <https://qiita.com/ruicc/items/8d631a19ba8cc36f243d>

> 現在、プログラミングという領域はまだ教育が整っていない為、多くのユーザが独力かつ長い時間をかけて、プログラミングを「使える」ようにして来ました。そのためそれぞれが持つメンタルモデルの多くは初めに触った言語や、その次に触った言語あたりに大きく依存していることでしょう。

> 僕らはプログラミングという知的活動に関わっていますが、そのメンタルモデルの構築や修正といった過程をずっと続ける必要があります。

# もとい、先の図は何を言っているか

- プログラムを「計算機への命令」ではなく「関数を適用して値を得る計算」の定義と捉える
- 関数  $f$  は、値を写すもの
- どんな集合からどんな集合に写されるか。その集合を型で表現する
- 集合の表現、つまり「型と型を組み合わせた構造化」がより良くできれば、プログラムが堅牢になる

# 注意

- 関数型プログラミングが他のあらゆるパラダイムよりも優れている、という話**ではありません**
- 特定のパラダイムはダメ、これはからは関数型、みたいな話**ではありません**

- 関数型プログラミングの見地からは関数や型をこんなふうに捉えることもできる
- プログラミングのさまざまな要素を、命令型とは異なる解釈ができて面白い。役に立つ (かも)

という話です

# React

- 関数型プログラミングのエッセンスを、フロントエンド開発に適用。広く普及した
- DOM に状態を埋め込んでいると状態管理が複雑になる。アプリケーションの状態を「書き換えるもの」ではなく「関数による状態遷移。不変なもの」と扱った

## 暗黙的な状態変化が減り、プレゼンテーションを宣言的に記述できるようになった

JavaScript

```
import { useState } from 'react';

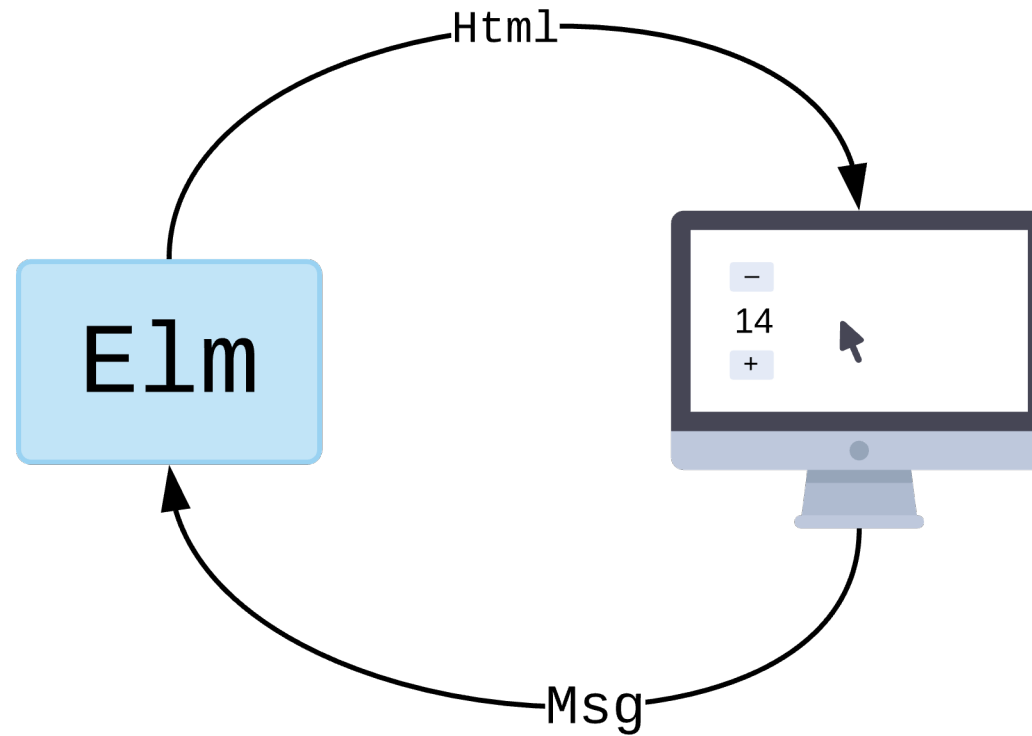
export default function MyApp() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}
```



# Elm アーキテクチャ



<https://guide.elm-lang.jp/architecture/>

# Elm

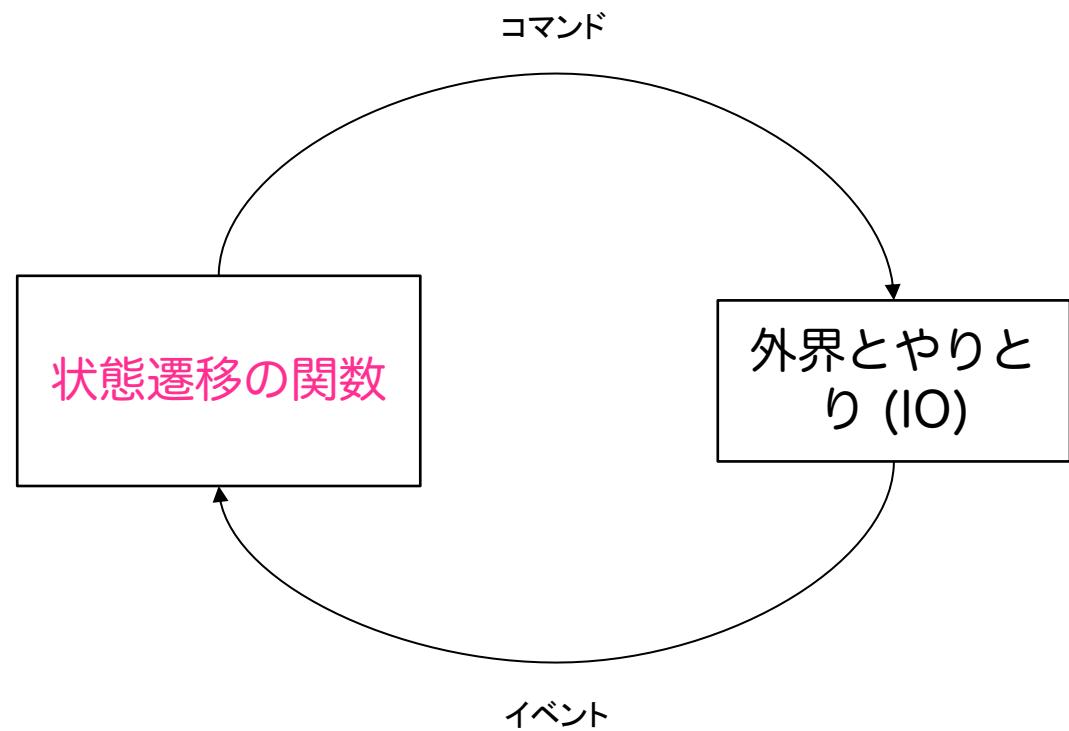
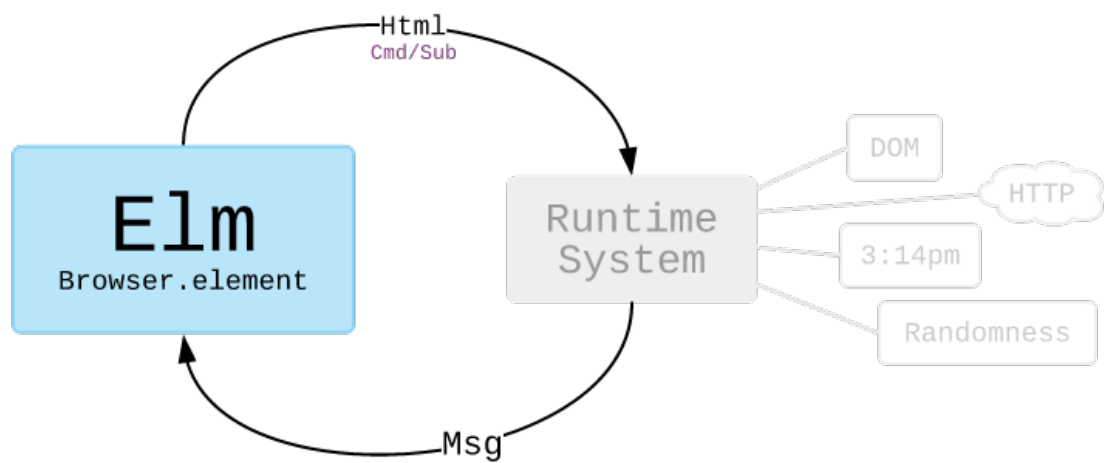
Elm

```
viewLikeButton : Photo -> Html Msg
viewLikeButton model =
    let buttonClass = if model.liked then ...
    div [ class "like-button" ]
        [ i [ class "fa fa-2x", class buttonClass, onClick ToggleLike ] [] ]
```

View は Model を描画。  
ユーザー操作に応じてイベントを送ると...

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        ToggleLike ->
            ( { model | photo = Maybe.map toggleLike model.photo }, Cmd.none )
        UpdateComment comment ->
            ( { model | photo = Maybe.map (updateComment comment) model.photo }, Cmd.none )
        SaveComment ->
            ( { model | photo = Maybe.map saveNewComment model.photo }, Cmd.none )
        LoadFeed (Ok photo) ->
            ( { model | photo = Just photo }, Cmd.none )
        LoadFeed (Err _) ->
            ( model, Cmd.none )
```

Elm ランタイムが update 関数を呼ぶ。関数にはイベントに応じたモデルの状態遷移を関数で記述する



 **状態の更新を、書き換えではなく、ある状態からある状態への遷移にすると良い...?**

- イミュータブルにすると良い...?
- 宣言的にすると良い...?
- 関数型プログラミングは良い...?

**「関数」をどう捉える？**

$$s = 1 + 2 + 3 + \dots + n$$

- どう書く? (最近見かけた)
  - for 文 (手続的、命令型)
  - 等差数列の公式
  - 再帰、畳み込み (宣言的、関数型)

# for 文で書く

```
int total = 0;

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

- 蓄えられている**値を書き換える**ことを繰り返して、その値から結果を得る
- 現在のコンピュータ・アーキテクチャの計算モデルに近い
  - メモリからレジスタにデータをロードし、演算を実行して、メモリに書き出す

ところで、我々はコンピュータを使って計算をしているが、プログラミング上の計算モデルまでそれに倣うことは必須なのか？

# 引数に関数を適用することを再帰（関数呼び出し）で繰り返し、値を得る

Haskell

```
sum [] = 0
sum (x : xs) = x + sum xs -- 関数 sum を再帰的に呼び (+) 関数を適用
```

TypeScript

```
const sum = (ns: number[]): number => {
  if (ns.length == 0) return 0

  const [x, ...xs] = ns
  return x + sum(xs)
}
```



# 値を書き換えるのではなく、関数の戻り値に関数を再帰的に適用して値を得る

```
f a b = a + b
```

Haskell

```
main = do
```

```
  print $ f 10 (f 9 (f 8 (f 7 (f 6 (f 5 (f 4 (f 3 (f 1 2))))))))
```

## 「戻り値に繰り返し関数を適用する」を関数にしたのが fold や reduce

Haskell

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

TypeScript

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

- 関数を適用することで値を得ることによって計算を成すのが、関数型プログラミングの考え方
- (根底にはラムダ計算 ... 今日はその話はしない)

# 文、式

- 文 (Statement)
  - for**文**、 if**文**、 print **文** ...
  - 計算機への何かしらの動作命令、手続き。**値を返さない**
- 式 (Expression)
  - 「計算を実行して結果を得るような処理を記述するための文法要素」 (Wikipedia)
  - **必ず値を返す**
  - 関数型プログラミングにおける関数は式

## (再掲) 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

## (再掲) 再帰的に関数を適用する

Haskell

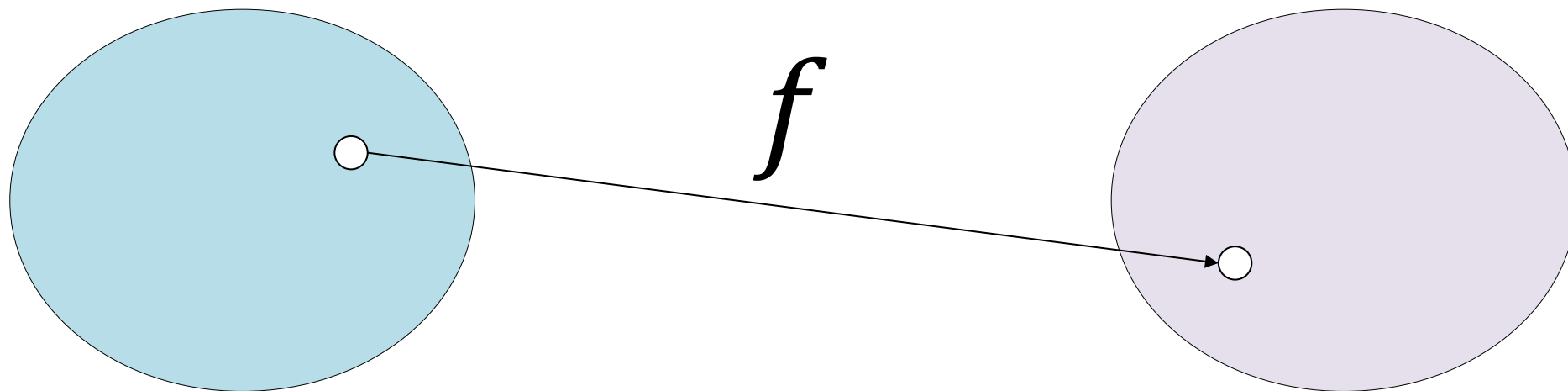
```
let s = foldl (+) 0 [1 .. n]
```

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
- 式は必ず値を返す。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

関数適用によって値を得る = 関数によって値を変換する・写す



式は**必ず**戻り値を返す。引数の値から戻り値に値を変換した、写した、と見ることもできる

# 命令型と関数型

- 命令型 ... 命令によって、計算機に、状態そのものを変化させる
- 関数型 ... 関数を適用することで値を得る・値を写す

関数型プログラミングはコンピュータではなく計算に焦点を当てる  
計算に焦点を当てるには、式によって計算を構成する  
式によって計算を「宣言」するので、宣言的になる

## 改めて、何を言ってるのか？

- プログラムを「計算機への命令」ではなく「関数を適用して値を得る計算」の定義と捉える
- 関数  $f$  は、値を写すもの
- どんな集合からどんな集合に写されるか。その集合を型で表現する
- 集合の表現、つまり「型と型を組み合わせた構造化」が良くできれば、プログラムが堅牢になる



# 関数による状態遷移の表現

## 関数による状態遷移の記述

$$S \xrightarrow{f} S'$$

ある状態  $s$  に関数  $f$  を適用して、別の状態  $s'$  を得る

// 顧客をアーカイブ状態にする

```
export const archiveCustomer = (customer: Customer): Customer => ({  
  ...customer,  
  archived: true,  
}))
```

TypeScript

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。状態変化が明示的

# Webアプリケーションと IO

- Webアプリケーション開発の多くの部分は、コンピュータ (外界) とのやりとり
  - データベースからデータを取得して ( IO )
  - Web API をコールして ( IO )
  - 端末に出力する ( IO )
  - IO はコンピュータへの命令なので、命令的に記述するのは自然

でもアプリケーションが大きくなってくると…

IO -> 計算 (業務ロジック、状態遷移) -> IO

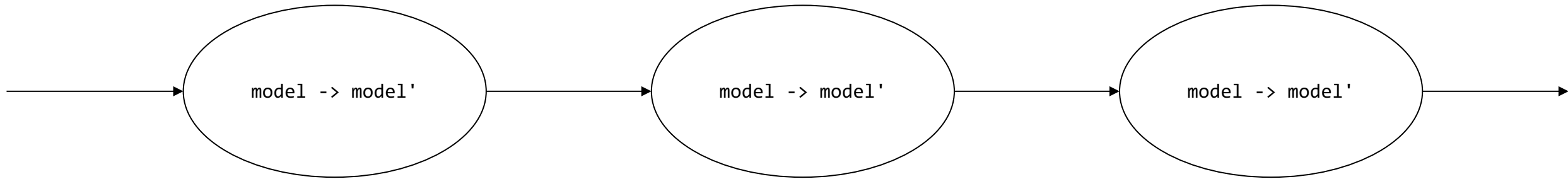
の真ん中の部分が大きく、複雑になってきませんか？

# (再掲) Elm

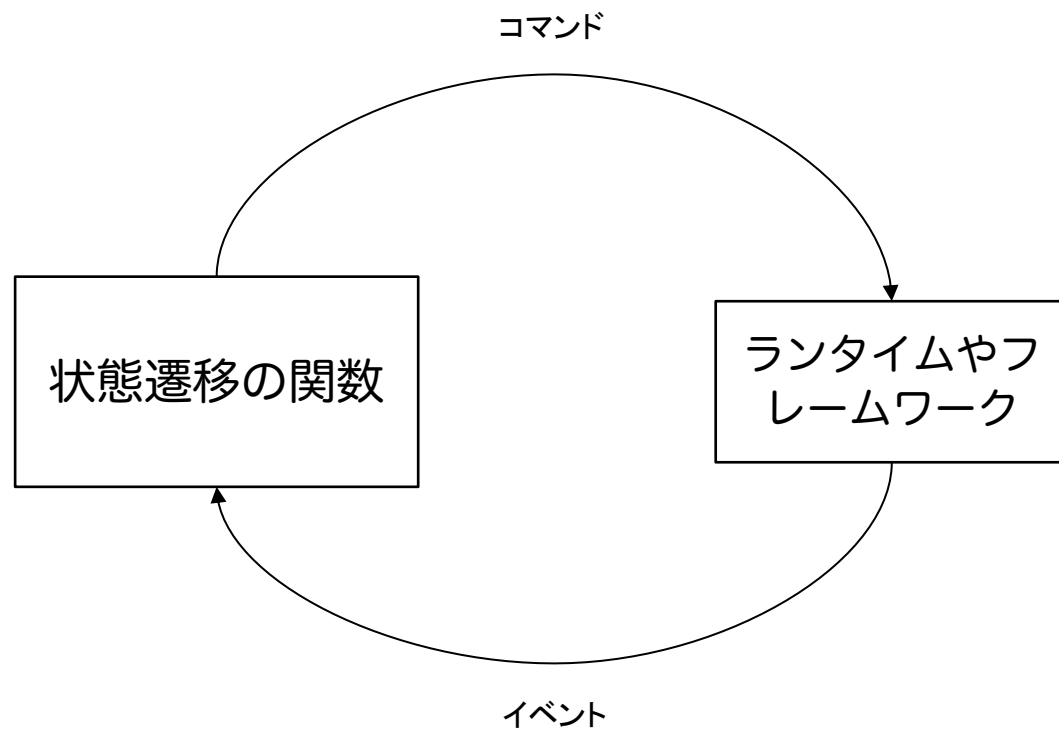
Elm

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    ToggleLike ->
      ( { model | photo = Maybe.map toggleLike model.photo }, Cmd.none )
    UpdateComment comment ->
      ( { model | photo = Maybe.map (updateComment comment) model.photo }, Cmd.none )
    SaveComment ->
      ( { model | photo = Maybe.map saveNewComment model.photo }, Cmd.none )
    LoadFeed (Ok photo) ->
      ( { model | photo = Just photo }, Cmd.none )
    LoadFeed (Err _) ->
      ( model, Cmd.none )
```

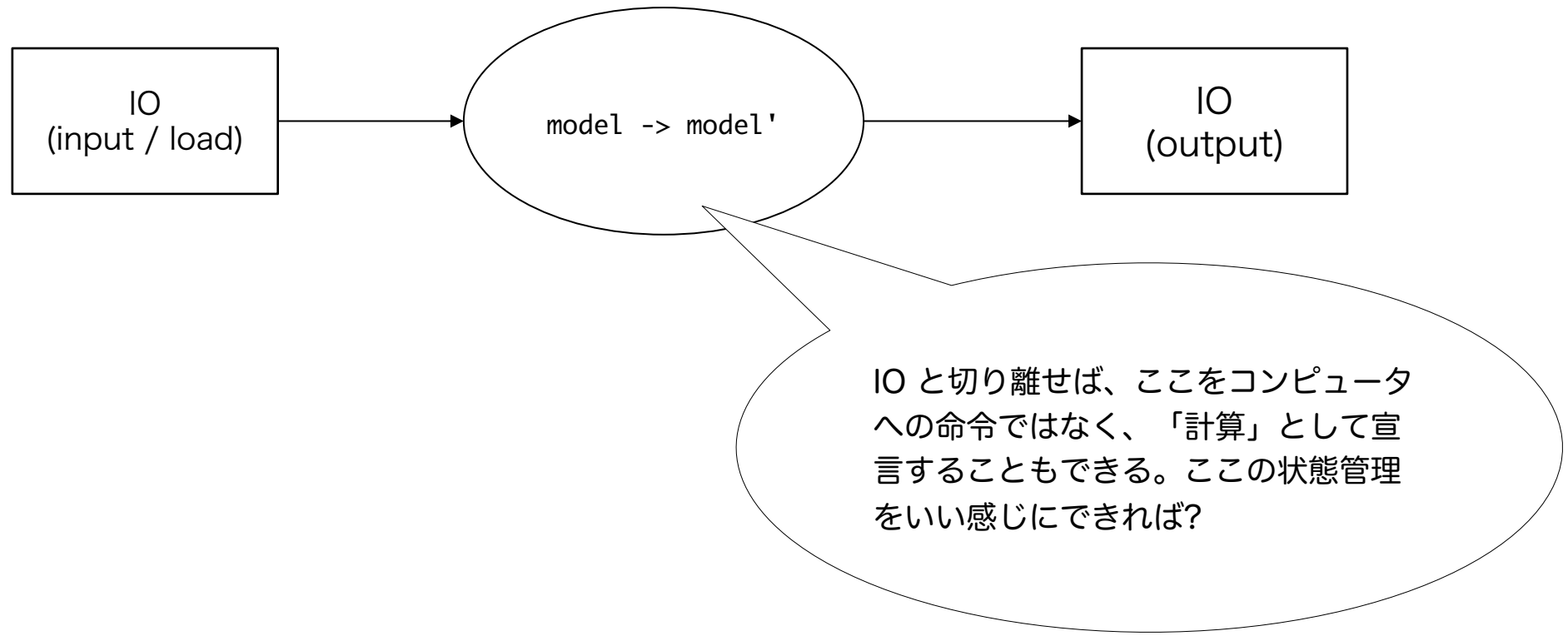
# イベントを契機に状態が遷移する・・・時系列に基づいた明示的な状態遷移



# イベントに伴い関数で状態を遷移させて、あとはフレームワークやランタイムに任せる

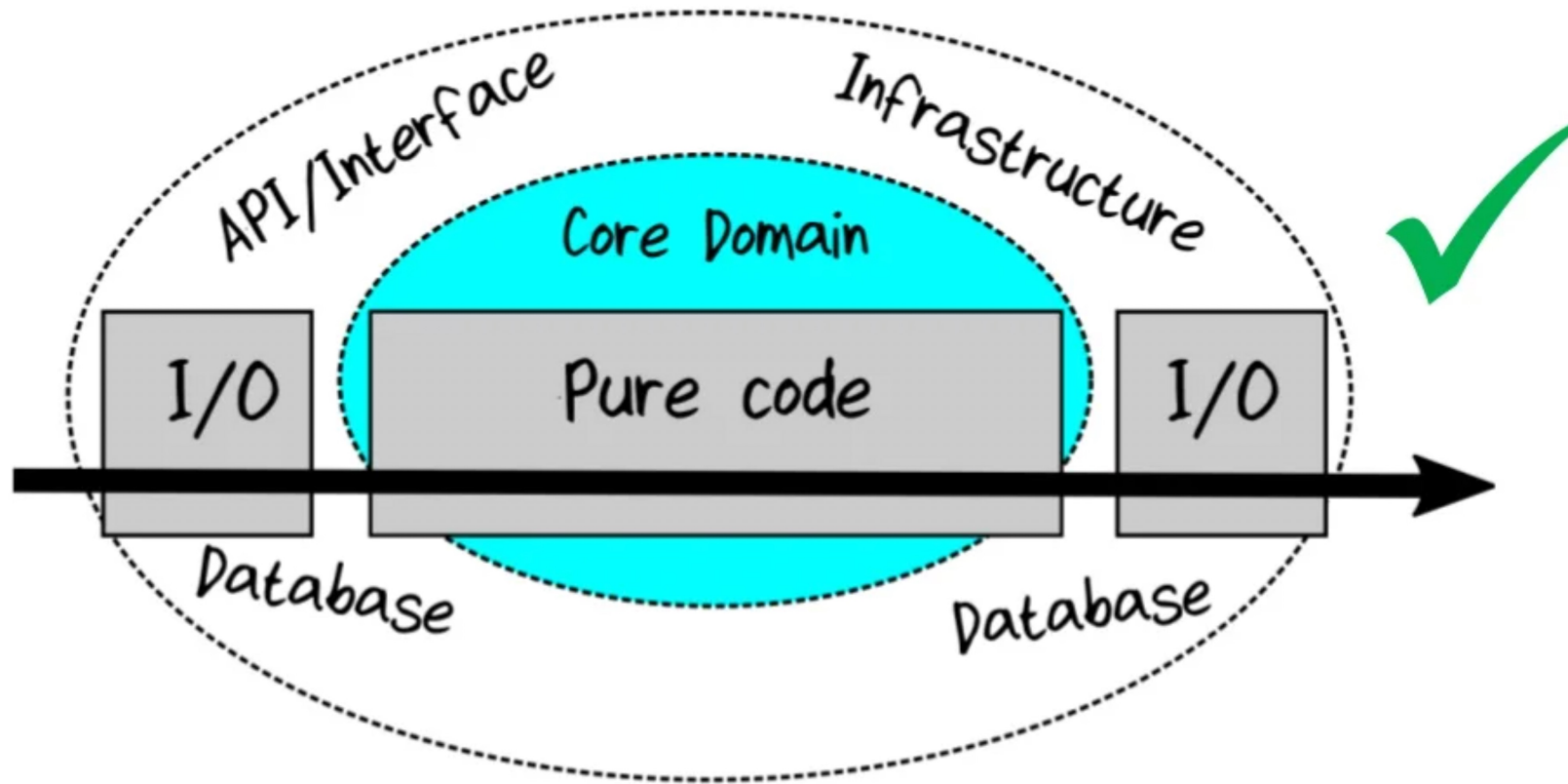


# IO -> 状態遷移 -> IO





## The "onion" architecture



Core domain is pure, and all I/O is at the edges  
See "Functional Core/Imperative Shell"

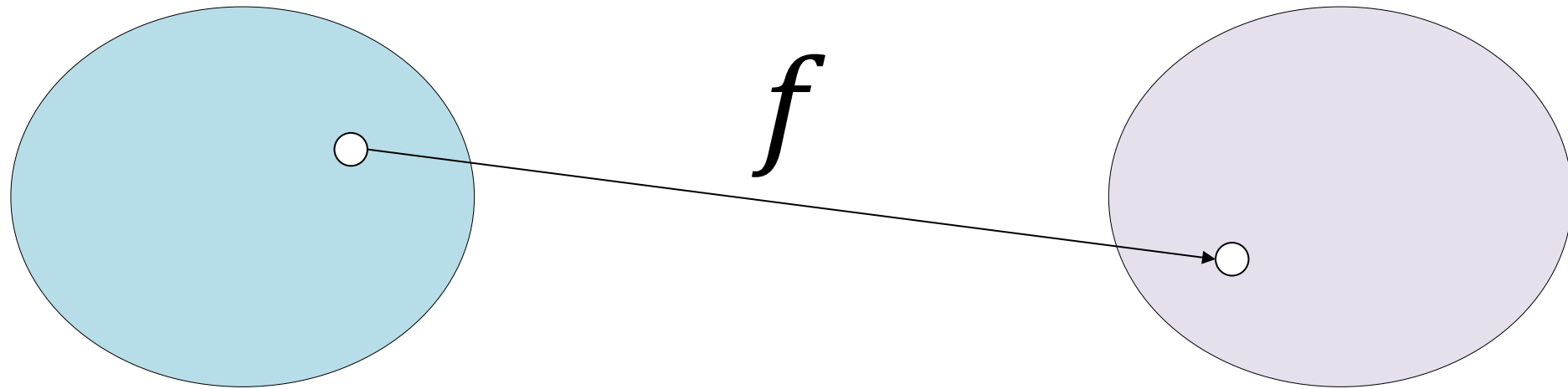
## オニオンアーキテクチャ (など) で得たいこと

- テstabiリティが向上する？ 部品を入れ替え可能になる？
  - それはそう。でも、副次的なものであって主目的ではない
- 業務ロジックを IO から切り離し、計算機への命令ではなく純粋なロジック (計算) として考えられるようにしたい
  - **色々なパラダイムを適用できる可能性が生まれる**
    - オブジェクト指向でも良いし、手続き型でも良いし、関数型プログラミングでも良い
    - うち、宣言的に状態管理するのは、(現時点では) 良いプラクティスではないか？ ← 今ここ

# React、Elm etc … 現代のフロントエンド開発

- 宣言的プログラミング
  - 「内部状態を書き換える」のではなく「状態から状態へ遷移させる。次の状態へ写す」
  - フロントエンドの状態管理を DOM から分離し「状態を書き換える」という命令ではなく「状態が次の状態に遷移する」という式だと捉え直した
  - 状態遷移を宣言的に記述することにより、**暗黙的な状態を意識せずにアプリケーションを記述できるようになった**

だんだん、メンタルモデルがアップデートされてきたでしょうか？



## (脱線) 文を使うとそこから芋づる式的に命令型になる

- Haskell の `forM_` 式 ... `for` 文みたいなもの (※実際は文ではなくユニット型を返す式)
  - ミュータブルな配列を書き換えたり、計算機と入出力するときに使う
  - 計算機への命令的な式なので戻り値がない
  - なお Haskell だからといって命令的に書けないわけではないし、**命令的に記述する方が良い場合もよくある**

```
main = do
  arr <- newListArray @IOArray (1, 10) [1 .. 10]

  forM_ [1 .. 10] $ \i -> do
    x <- readArray arr i
    writeArray arr i (x * 2)

  forM_ [1 .. 10] $ \i -> do
    readArray arr i >>= print
```

Haskell

# 同じ目的のデータ構造のイミュータブル版、ミュータブル版を使う比較

```
-- 状態の表現にイミュータブルなデータ構造
data UnionFind = UnionFind
{ parent :: IM.IntMap Int,
  size   :: IM.IntMap Int
}
deriving (Show)
```

Haskell

```
-- 状態の表現にミュータブルな配列
data UnionFind a v
= UnionFind
  (a v v)
  (IOUArray v Int)
```

Haskell

```
main = do
  [n, m] <- getInts
  uvs <- replicateM m getTuple

  let uf0 = newUF (0, n - 1)

  -- 関数 f により uf を次の状態に遷移させる
  let (_, xs) = mapAccumL f uf0 uvs
      where
        f uf (u, v) =
          let same = isSame uf u v
          in if same then (uf, same) else (unite uf u v, same)

  ...
```

関数の戻り値が次の状態のデータ構造なので、戻り値を引き回す関数型的な実装になる

```
main = do
  [n, m] <- getInts
  uvs <- replicateM m getTuple

  uf <- newUF @IOUArray (0, n - 1) (-1)

  forM_ uvs $ \ (u, v) -> do
    same <- isSame uf u v

    -- データ構造 uf に作用を起こして内部状態を変える
    unless same $ do
      unite uf u
```

ミュータブルなデータ構造を使うと値を戻さないなので、制御構造も含め自然と命令型の実装になる  
(ちなみに私は Union-Find はこちらを常用してます)

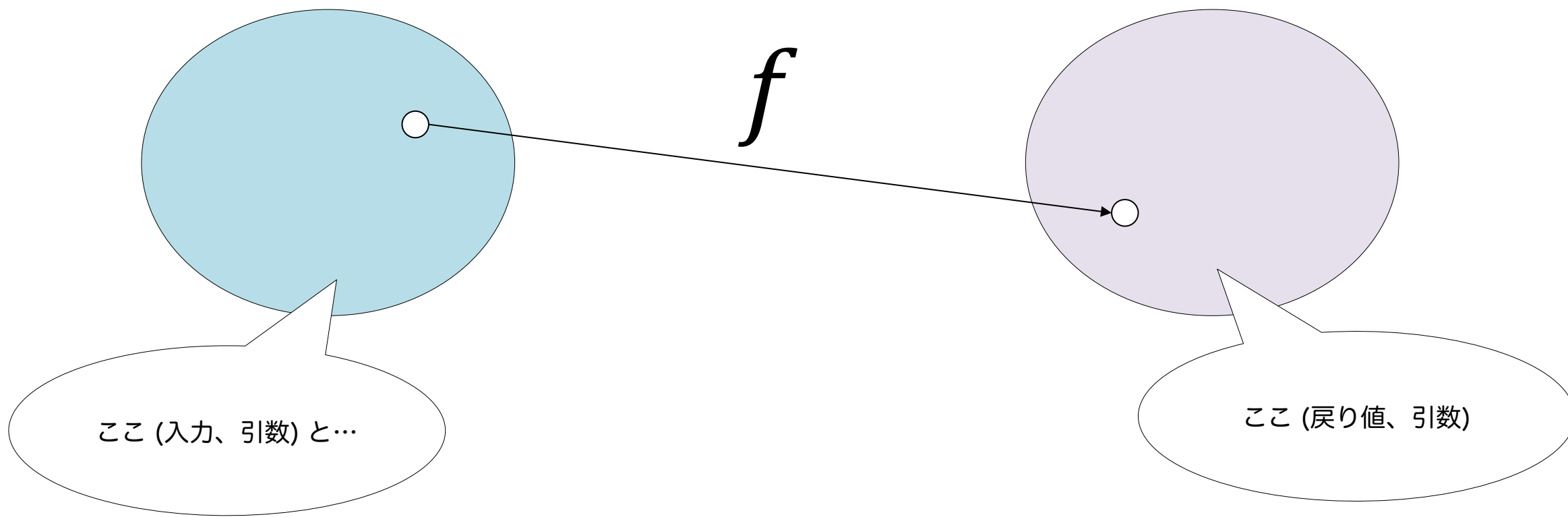
## (再掲) 先の図は何を言ってるのか？

- プログラムを「計算機への命令」ではなく「関数を適用して値を得る計算」の定義と捉える
- 関数  $f$  は、値を写すもの
- どんな集合からどんな集合に写されるか。その集合を型で表現する
- 集合の表現、つまり「型と型を組み合わせた構造化」が良くできれば、プログラムが堅牢になる

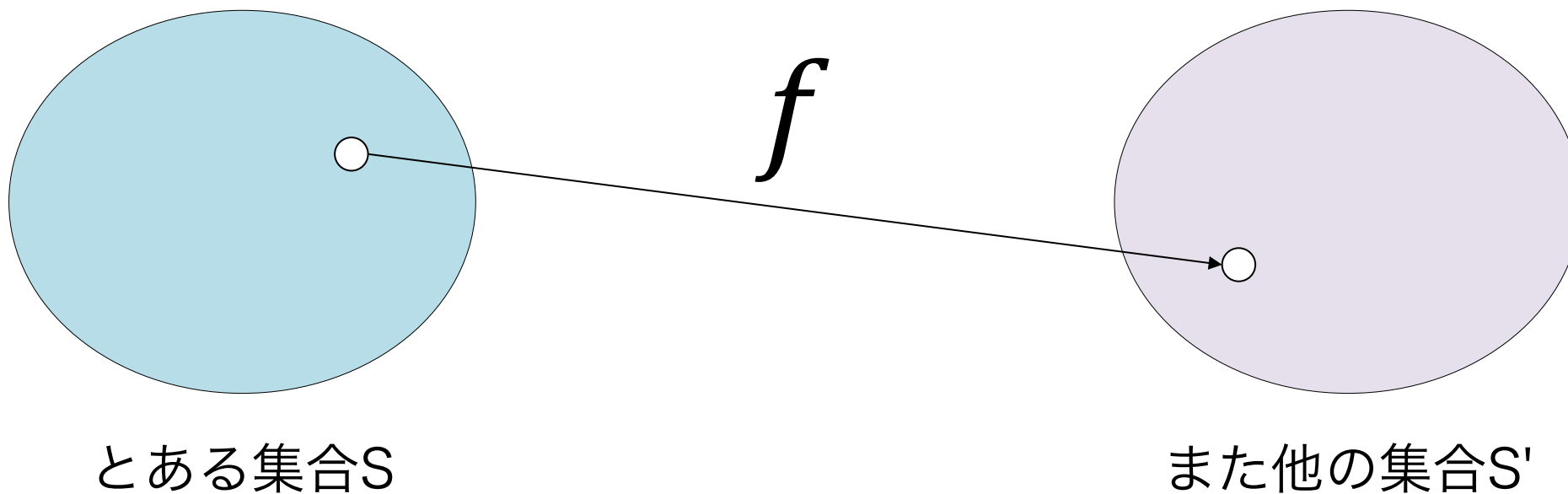
**型を集合をとらえる**



関数は値を変換する・写すものという見方ができることがわかった  
それを前提に、次は値が属する領域について考えてみます



ある集合に属する値を、関数  $f$  により、また他の集合に属する値に写す



これがどんな集合か記述するのが「型」

## 難しいことはなく、普段書いている型をどうみる？ という話


```
function parseHex(str: string): number {  
    return parseInt(str, 16);  
}
```

TypeScript

- 低レイヤー視点では「ビット列のデータをソフトウェアにどう解釈させるか」意味付けするのが型
- 別のメンタルモデル ... 関数の視点で見ると「**型は集合**」とも捉えられる
  - 「string の集合に属する値を number の集合に属する値に写す」
  - 「この関数の定義域は string で、値域が number」

TypeScript: Documentation - 1 x +

← → ↺ 🏠 🔒 typescriptlang.org/docs/handbook/typescript-in-5-minutes-oop.html#types-as-sets 🔍 ☆ 🔌 🖨️ シークレット ⋮

Get Started 

TS for the New Programmer

TypeScript for JS Programmers

TS for Java/C# Programmers

TS for Functional Programmers

TypeScript Tooling in 5 minutes

Handbook >

Reference >

Modules Reference >

Tutorials >

What's New >

Declaration Files >

JavaScript >

Project Configuration >

## Types as Sets

In C# or Java, it's meaningful to think of a one-to-one correspondence between runtime types and their compile-time declarations.

In TypeScript, it's better to think of a type as a *set of values* that share something in common. Because types are just sets, a particular value can belong to *many* sets at the same time.

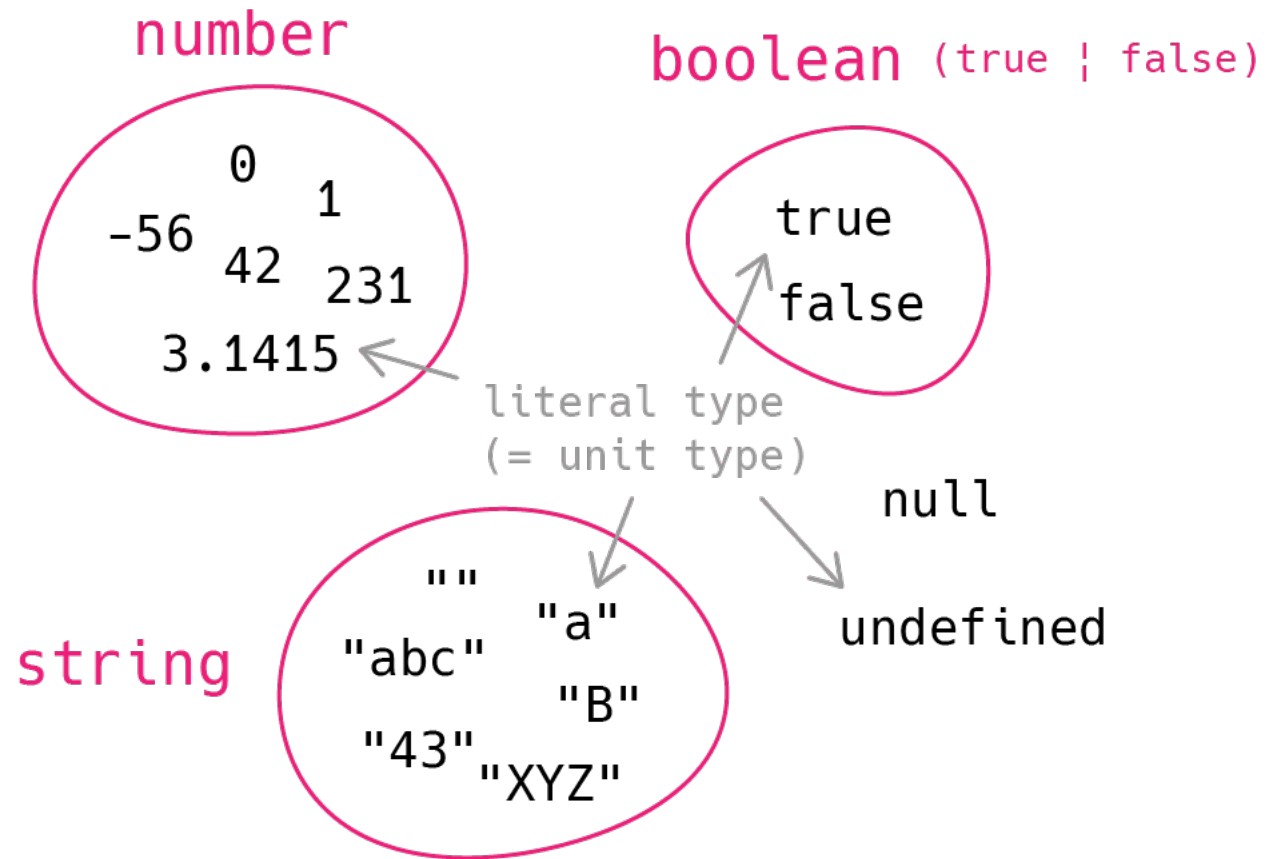
Once you start thinking of types as sets, certain operations become very natural. For example, in C#, it's awkward to pass around a value that is *either* a `string` or `int`, because there isn't a single type that represents this sort of value.

In TypeScript, this becomes very natural once you realize that every type is just a set. How do you describe a value that either belongs in the `string` set or the `number` set? It simply belongs to the *union* of those sets: `string | number`.

TypeScript provides a number of mechanisms to work with types in a set-theoretic way, and you'll find them more intuitive if you think of types as sets.

## Erased Structural Types

In TypeScript, objects are *not* of a single exact type. For example, if we construct an object that satisfies an interface, we can use that



「TypeScript における型の集合性と階層性」より引用  
<https://zenn.dev/estra/articles/typescript-type-set-hierarchy>

- 「型は、互いに関連する**値の集合**です」(プログラミング Haskell)
- 「型 (type) あるいはデータ型 (data type) とは、データが**どのような性質の集合に属するか**を示すものです」(関数型プログラミング実践入門)
- 「**(可能な操作や演算などの体系などが) 共通している値の集合が型**であり、それぞれの値は複数の集合 (型) に属することができる」(TypeScript における型の集合性と階層性)
- 「**型は値や処理の満たすべき性質を強く意味付けするために使うことができ**、「プログラムの正しさ」を保証するための重要なファクターです」

## 型はコンパイラなどのソフトウェア (ツール) が理解できる

- 値の取りうる範囲をより厳密に宣言できればコンパイル段階で値がその範囲に収まることが保証できる
- つまりアプリケーションを実際に動作させなくても、ありえない状態が作られないことを保証できる
- 型情報を色々なツールチェーンが読み取って、開発者体験を改善している



TypeScript Origins: The Documentary  
<https://www.youtube.com/watch?v=U6s2pdxebSo>

# 型をどう考えるか？

- ソフトウェア (コンパイラやツールチェーン) にプログラム構造の情報を与えるもの
- 関数  $f$  の定義域と値域の取りうる範囲を決めるもの

「後者で考えて型を書いた結果、前者によってプログラムが堅牢になり開発者体験もよくなるもの」というのが私のビュー



# 型は構造化することができる

TypeScript

```
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'))  
}
```

**型と型を組み合わせる ... 和 (OR) を使う**

# 代数的データ型

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

Haskell

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

Haskell

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

Haskell

- Haskell などの言語では新しい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論理積 (AND) だけでなく **論理和 (OR)** が使える

## 型の組み合わせに「和 (OR)」が使える

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

Haskell

- 新しいデータ構造の形を宣言する手段は各言語様々であるが、かつて多くの言語では積 (AND) で組み合わせるのが主な方法だった
  - struct や class などレコード型によるプロパティの組み合わせは AND
- AND だけで型と型を組み合わせると、不自然な階層や不必要な値の組み合わせが発生しやすい
  - 直積の階層構造 … 例えば class による階層構造

```
export interface ReservationHolder {  
  kind: 'ReservationHolder'  
  name: string  
  nameKana: string | null  
  phoneNumber: PhoneNumber  
  email: EmailAddress | null  
}
```

```
export interface Visitor {  
  kind: 'Visitor'  
  name: string  
  nameKana: string | null  
  phoneNumber: PhoneNumber  
}
```

```
interface HolderAndVisitor {  
  kind: 'HolderAndVisitor'  
  holder: ReservationHolder  
  visitor: Visitor  
}
```

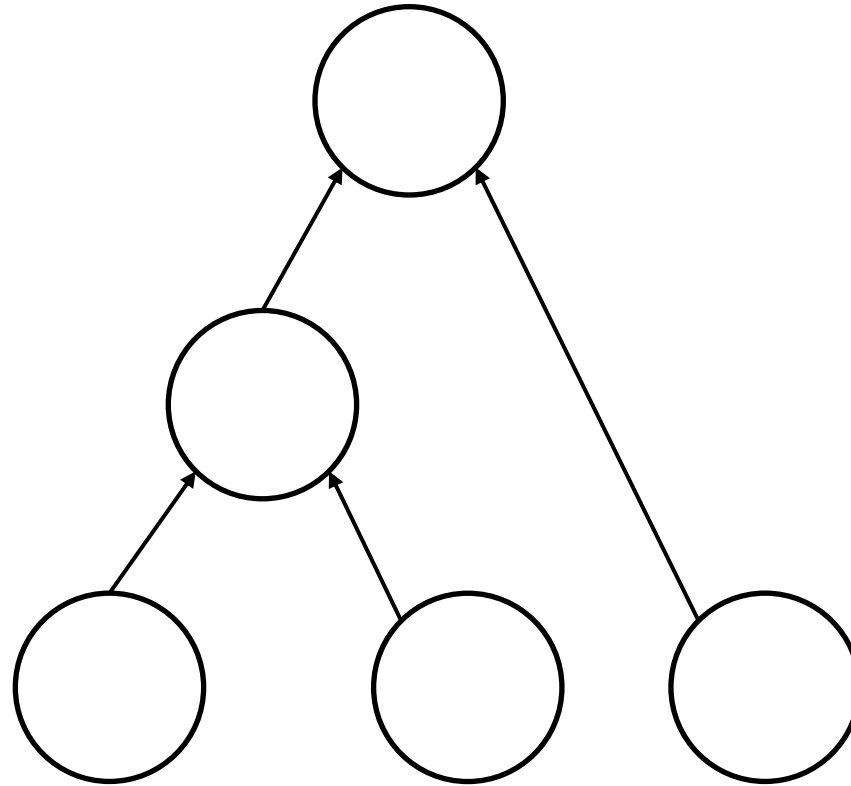
```
interface HolderOnly {  
  kind: 'HolderOnly'  
  holder: ReservationHolder  
}
```

```
interface VisitorOnly {  
  kind: 'VisitorOnly'  
  visitor: Visitor  
}
```

```
export type ReservationGuest = HolderAndVisitor | HolderOnly | VisitorOnly
```

構造が異なるものを「または  
(和 / OR)」で組み合わせる

## 積 (AND) だけで型と型を組み合わせていくと階層が必要になる



- 直積だけでは「似ているが少し構造が違うもの」をダブリなくまとめるのに階層が必要になる
- 似ている構造を抽出して親にする ... 安易にやると「継承による差分プログラミング」になってしまう
- 和があることで「階層にするか、並べるか」柔軟な組み合わせが可能になる

# “Making illegal states unrepresentable”

```
interface User {  
  memberId: MemberId | undefined  
  guestId: GuestId | undefined  
}
```

TypeScript

レコードは「かつ (AND)」

取り得る値の種類数は各属性の積になる (直積)

$$2 \times 2 = 4$$

- ・ 両方 undefined
- ・ 両方の値が埋まる

という仕様上あり得ない状態が生まれる

```
interface Member {  
  userId: MemberId  
}  
  
interface Guest {  
  guestId: GuestId  
}
```

TypeScript

```
type User = Member | Guest
```

ユニオンは「または (OR)」

取り得る種類数は各属性の和 (直和)

$$1 + 1 = 2$$

仕様上あり得ない状態は表現しない  
→ そしてコンパイラがそれを理解する

```

type connection_state =
| Connecting
| Connected
| Disconnected

type connection_info = {
  state:           connection_state;
  server:          inet_addr;
  last_ping_time:  time option;
  last_ping_id:    int option;
  session_id:      string option;
  when_initiated:  time option;
  when_disconnected: time option;
}

```

OCaml

```

type connecting  = { when_initiated: time; }
type connected   = { last_ping   : (time * int) option;
                    session_id: string; }

type disconnected = { when_disconnected: time; }

type connection_state =
| Connecting of connecting
| Connected  of connected
| Disconnected of disconnected

type connection_info = {
  state: connection_state;
  server: inet_addr;
}

```

OCaml

「なぜ次に学ぶ言語は関数型であるべきか - YAMAGUCHI::weblog」より引用  
<https://ymotongpoo.hatenablog.com/entry/20111105/1320506449>



## こちらよりも...

TypeScript

```
export class Tag {  
  state: 'Unvalidated' | 'Validated' | 'Created',  
  id: TagId | undefined,  
  groupId: RestaurantGroupId,  
  label: string,  
  icon: TagIcon | undefined,  
  sortOrder: number | undefined,  
  builtin: boolean | undefined  
}
```

## こちらの方が、値の組み合わせパターンが少なく厳密

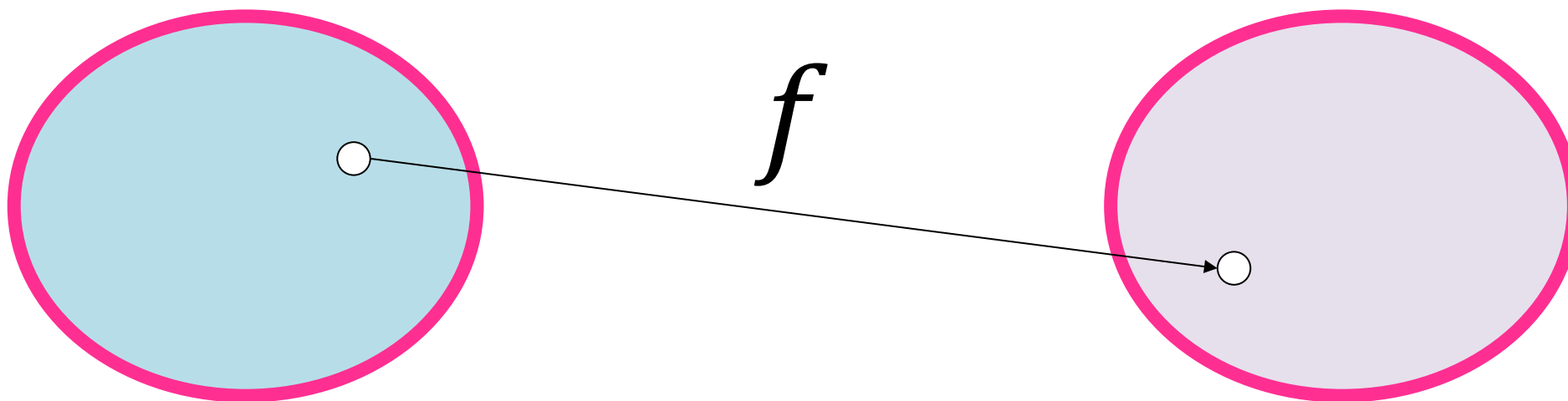
TypeScript

```
interface UnvalidatedTag {
  kind: 'Unvalidated'
  groupId: string
  label: string
  icon?: { symbol: string; type: TagIconType; color?: string | null | undefined } | null | undefined
}

interface ValidatedTag {
  kind: 'Validated'
  groupId: RestaurantGroupId
  label: string
  icon: TagIcon
}

export interface CreatedTag {
  kind: 'Created'
  id: TagId
  groupId: RestaurantGroupId
  label: TagLabel
  icon: TagIcon
  sortOrder: number
  builtin: boolean
}

export type Tag = UnvalidatedTag | ValidatedTag | CreatedTag
```



- ある状態からある状態へ遷移するとき、それぞれ状態が取りうる値の集合に、不要な状態が少なければ少ないほど関数  $f$  の正しさを (コンパイラレベルで) 保証することができる
- 直積と直和による組み合わせに限らず、型システムの表現力が高ければ、集合をより正しいものにすることができる

## 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

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

Haskell

```
main = do
```

```
  let someValue :: Maybe String
```

```
      someValue = ...
```

Just String か Nothing のどちらか

```
case someValue of
```

Maybe 型をパターンマッチで分解する

```
  Just s -> putStrLn (s ++ ", naoya")
```

```
  Nothing -> putStrLn "Farewell"
```

# TypeScript でもいけます

Haskell

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

TypeScript

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

リテラル型でタグをつけておく

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

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

ユニオンで組み合わせた型

```
// 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)  
  }  
}
```

タグに応じて分解 (リテラル型なのでちゃんと型が効く)

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

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

# GraphQL にも Union型

- GraphQL にも Union 型があり、上手く使うと堅牢性を高められる
- クライアント側で型に応じた分解が可能

```
query GetSearchResults {  
  search(contains: "Shakespeare") {  
    __typename  
    ... on Book {  
      title  
    }  
    ... on Author {  
      name  
    }  
  }  
}
```

GraphQL

実際のWebアプリケーション開発でどうしているかは、以下のスライドをご覧ください



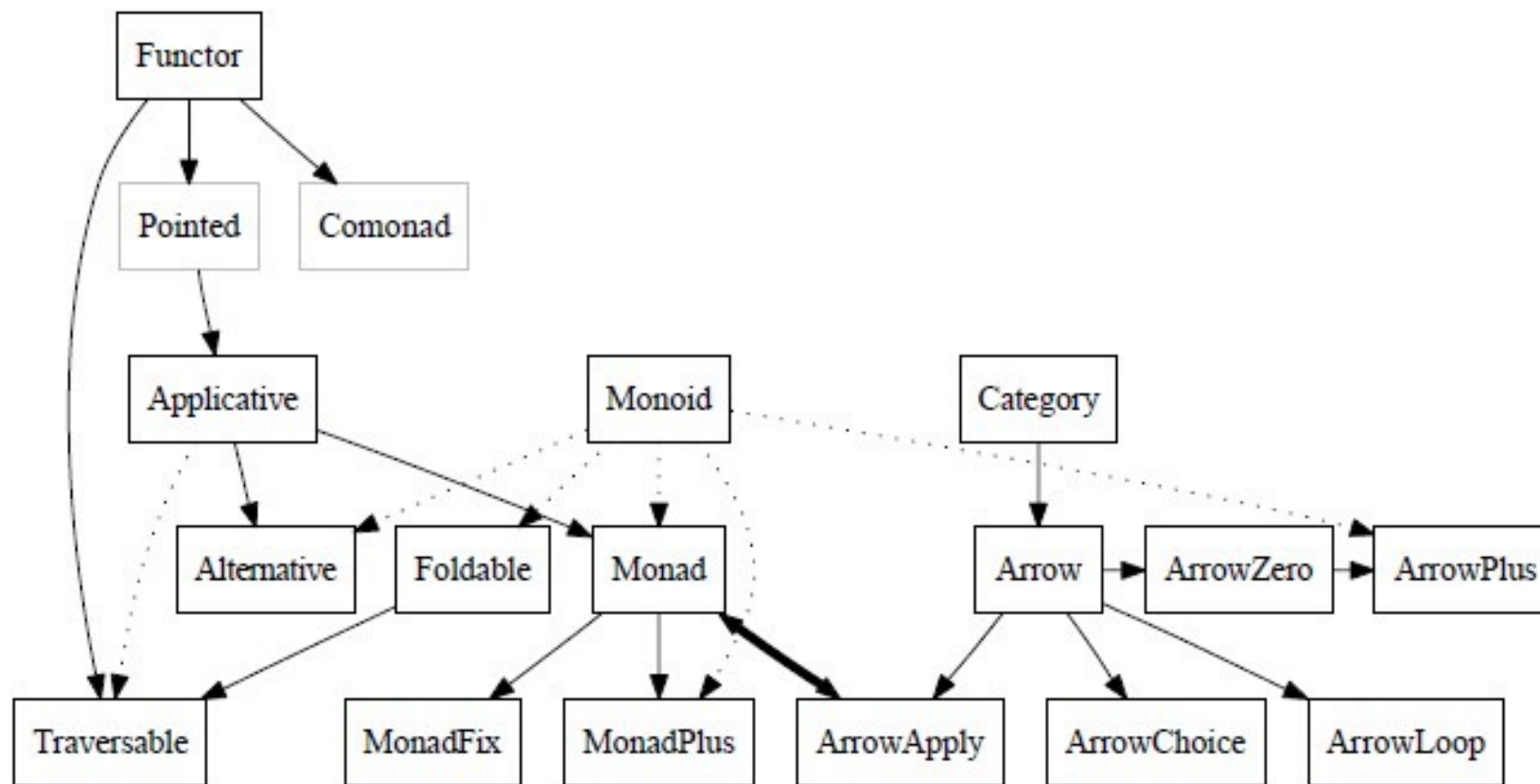
<https://speakerdeck.com/naoya/>



## 型によって文脈を表現する (詳細割愛)

- Maybe (Optional) ... 存在しないかもしれない値
- Either (Result) ... 失敗するかもしれない値。起こりうる失敗
- IO ... 副作用を持つ操作、または外界とのインタラクション
- List ... 非決定計算

## 型クラス ... 型の分類と階層 (詳細割愛)



## (おまけ) 関数の入出力を集合とみなすということは …

- 数学的には、集合に演算を入れると「代数的構造」が現れる
- 群論的な抽象 … モノイド、群、半環、環 …
- これを意識して実装をすると、数学方向への抽象化につながる。良いプラクティスのひとつ

## 例えば動的計画法 (ナップザック問題) の実装

```
for (int j = 0; j <= W; ++j) dp[0][j] = 0;

for (int i = 0; i < N; ++i) {
    for (int j = 0; j <= W; ++j) {
        if (j >= w[i]) dp[i+1][j] = max(dp[i][j-w[i]] + v[i], dp[i][j]);
        else dp[i+1][j] = dp[i][j];
    }
}

cout << dp[N][W] << endl;
```

C++

```
let f (w, v) (wi, vi)
    | v == minBound = [(w, v)]
    | otherwise = [(w, v), (w + wi, v + vi)]

let dp = accumArrayDP @UArray f max minBound (0, wx) [(0, 0)] wvs

print $ maximum (elems dp)
```

Haskell

集合に演算2つと単位元を入れれば DP になるよう抽象化している (DPは半環)

## 累積演算

```
let s = scanl' (+) 0 xs
```

## しゃくとり法

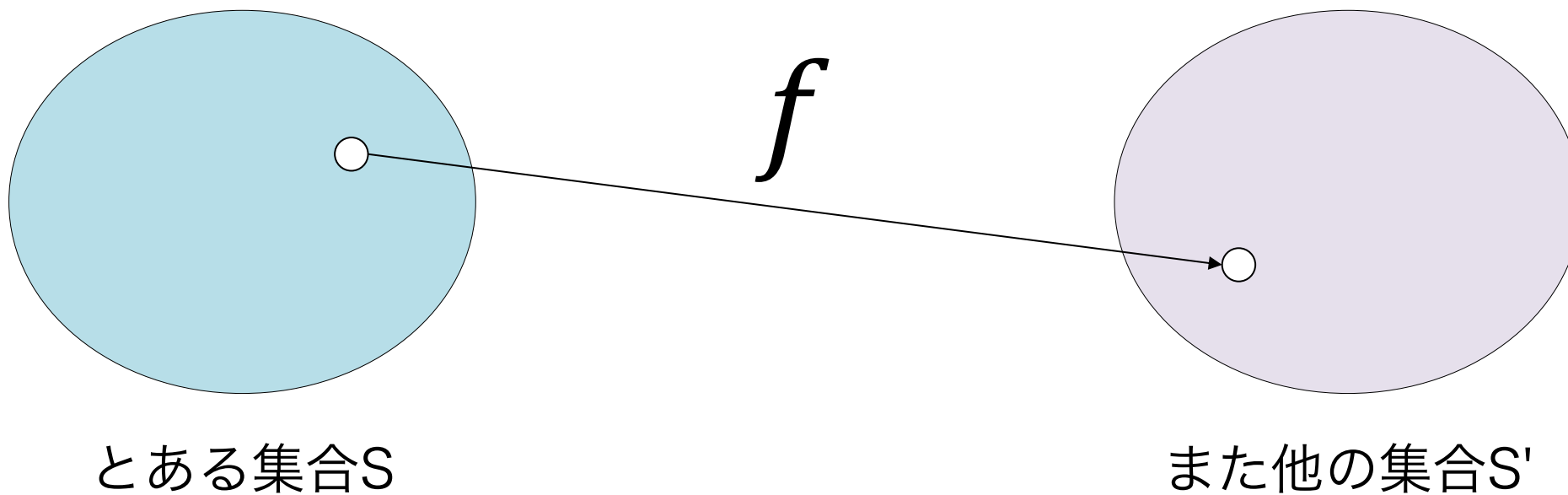
```
let sections = shakutori (1, n) (s !) (+) (-) 0 (\acc r -> acc + r <= k)
```

## ダブリング

```
let dp = doubling f (+) (10 ^ 5, 10 ^ 18) xs
```

- アルゴリズムを代数的構造を意識して抽象化する。「このアルゴリズム (計算構造) にどんな結合演算を入れて使うか」というインタフェースの関数が得られる
- どんな演算に入れられるか? ... 型が保証する (入れられる演算の集合を定義する)

ある集合に属する値を、関数  $f$  により、また他の集合に属する値に写す



これがどんな集合か記述するのが「型」

## (再掲) 図は何を言っているか

- プログラムを「計算機への命令」ではなく「関数を適用して値を得る計算」の定義と捉える
- 関数  $f$  は、値を写すもの
- どんな集合からどんな集合に写されるか。その集合を型で表現する
- 集合の表現、つまり「型と型を組み合わせた構造化」がより良くできれば、プログラムが堅牢になる

最近はこの考えを中心に置いて、プログラム設計をしています  
「集合である型で設計し、関数で状態を遷移」

# どんなプログラミング言語でも関数型でガンガンやっていくべき？

- 冒頭にも述べた通り、関数型プログラミング最強、という話はしていません
- よって、私見は「いいえ」 / 言語の特性に合わせましょう
  - Haskell、F#、OCaml、Scala などそうすることを前提に作られている言語は問題ないでしょう
  - TypeScript ... 関数型としての側面はありますが、シンタックス面では前段の言語に比較すると不足も感じます。一方 TS の型システムは、ご存知の通り動的な特徴もあり、非常に強力です。型は集合のメンタルモデルで上手に使い、関数型のエッセンスを取り入れつつも、命令的に書けばいい場面では無理をしない...ぐらいが良いバランスではないでしょうか?
    - 私は Result 型を使って関数型寄りですが、いい側面もありますが面倒なこともあります
  - Rust、Kotlin、Swift ... 型に「OR」相当の機能があり、型システムは現代的です。どんなスタイルがいいかは、詳しくないのでわかりません。各言語なりの「〇〇 Way」があると思います、識者にお聞きください



# まとめ

- 関数型プログラミングの見地から式、関数、型などをみると命令型で捉えていた時とは少し違う見方ができます
- 「計算機への命令」なのか「計算」なのか
- 「関数は値を写すもの、型は集合」と見ることもできます
- 昨今のプログラミング言語やフレームワークには関数型の影響が垣間見えます
- 自分が使っている言語、フレームワーク、ツールの特性に合わせて、無理のない範囲で、エッセンスを取り入れていくとより楽しく、より良い開発ができると思います

## 参考文献

- Will Kurt 著, 株式会社クイープ 訳, 「入門 Haskell プログラミング」, 翔泳社, 2019
- Graham Hutton 著, 山本和彦 訳, 「プログラミング Haskell 第2版」, ラムダノート, 2019
- Scott Wlaschin 著, 「Domain Modeling Made Functional」, 2021
- Jeremy Fairbank (著), ヤギのさくらちゃん (翻訳), 「プログラミングElm — 安全でメンテナンスしやすいフロントエンドアプリケーション開発入門」, マイナビ出版, 2021
- 大川徳之著, 「[増補改訂] 関数プログラミング実践入門」, 技術評論社, 2016
- Web のリソース
  - TypeScript における型の集合性と階層性, <https://zenn.dev/estra/articles/typescript-type-set-hierarchy>
  - 「なぜ次に学ぶ言語は関数型であるべきか - YAMAGUCHI::weblog」, <https://ymotongpoo.hatenablog.com/entry/20111105/1320506449>