圏論とHaskell

宇佐見 公輔

第２回 関西日曜数学 友の会

自己紹介

- 宇佐見 公輔（うさみ こうすけ）
  - twitter: @usamik26 / connpass: usami-k
- 大学時代は数学専攻
  - でも圏論はほとんどやってない
- 本職はプログラマ（フェンリル株式会社）
  - でもHaskellは使ってない

圏論とHaskell

- 圏論は数学理論のひとつ
  - 写像に着目した抽象代数論
- Haskellはプログラミング言語のひとつ
  - 代表的なアプリとして Pandoc などがある
  - 言語設計として圏論を意識している

圏（Category）

圏の定義

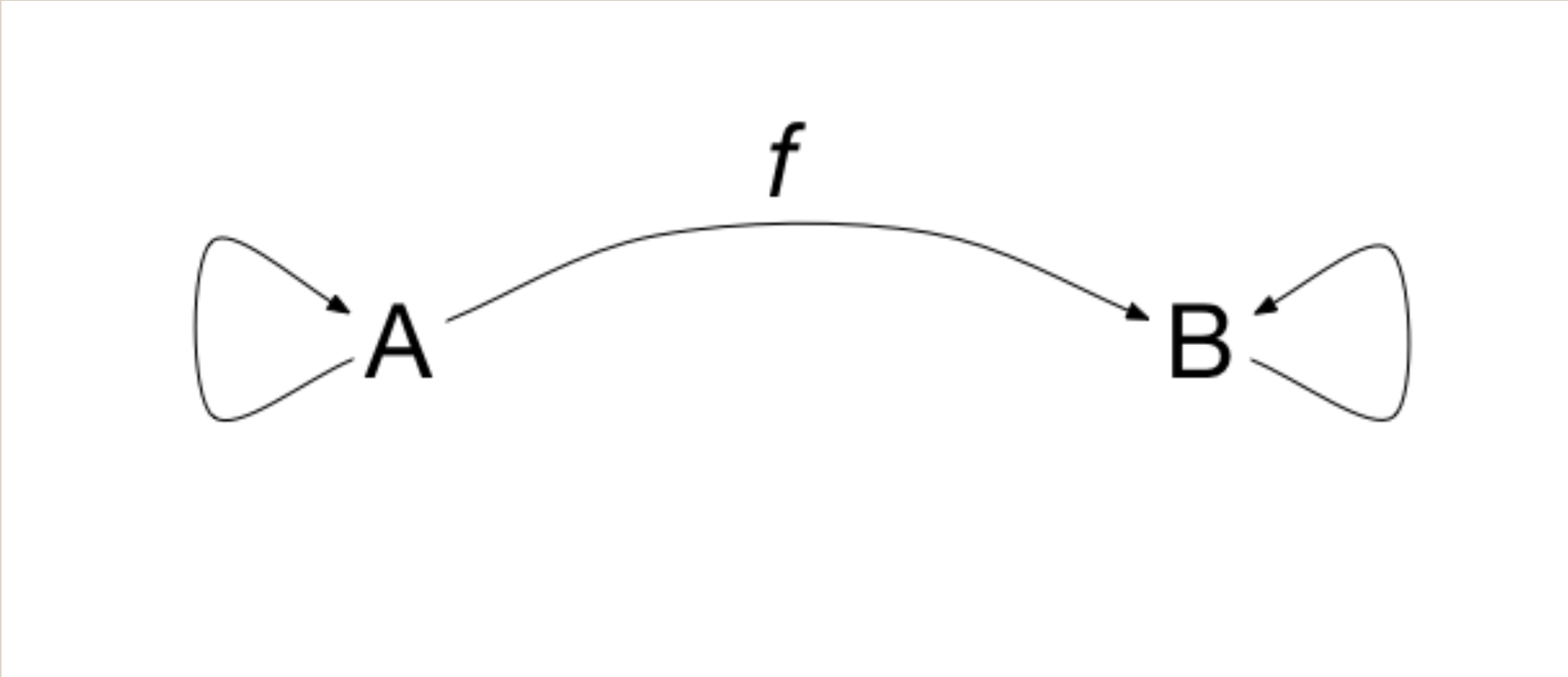
- 対象 A, B, C, $\ldots$
- 射 $f : A \to B$
- 射の合成 $g \circ f : A \to C$（ここで $f : A \to B$, $g : B \to C$）
  - 結合律 $h \circ (g \circ f) = (h \circ g) \circ f$
- 恒等射 $1_A : A \to A$（任意の対象 A について存在する）
  - 単位元 $f \circ 1_A = 1_B \circ f = f$

例：対象がふたつの圏

(1) 対象間の射がない（恒等射だけ）

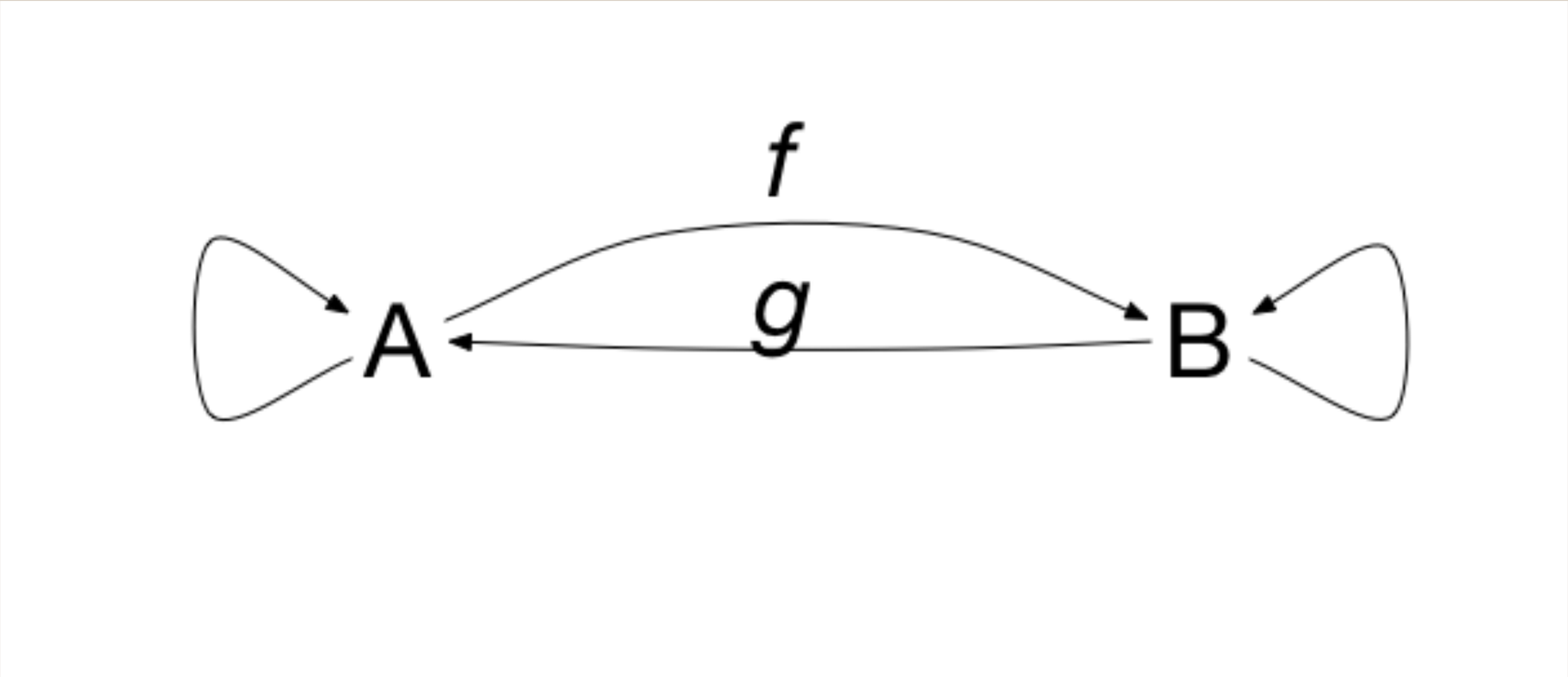
- これは圏になっている（合成可能な射はない）

(2) 対象間の射がひとつ



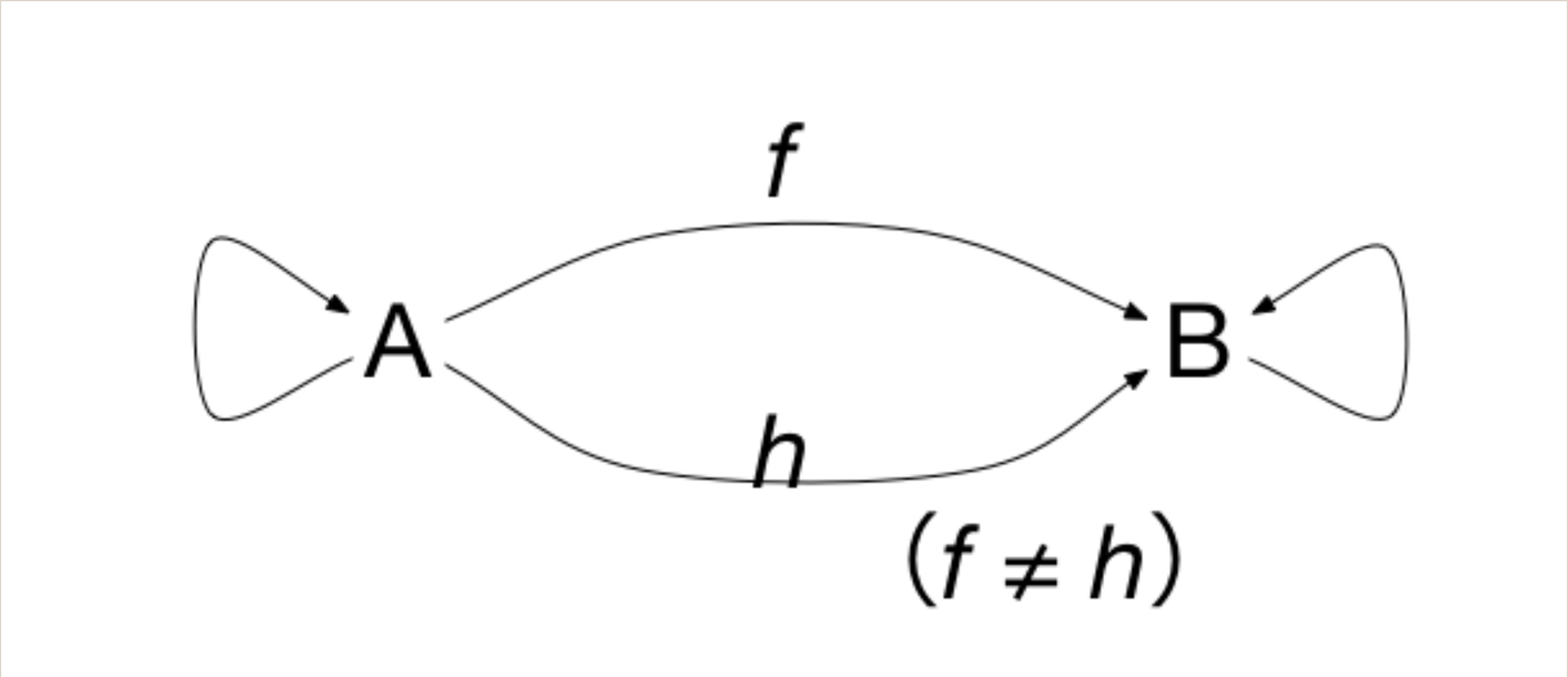
- これは圏になっている

(3) 対象間の射がふたつ：その1



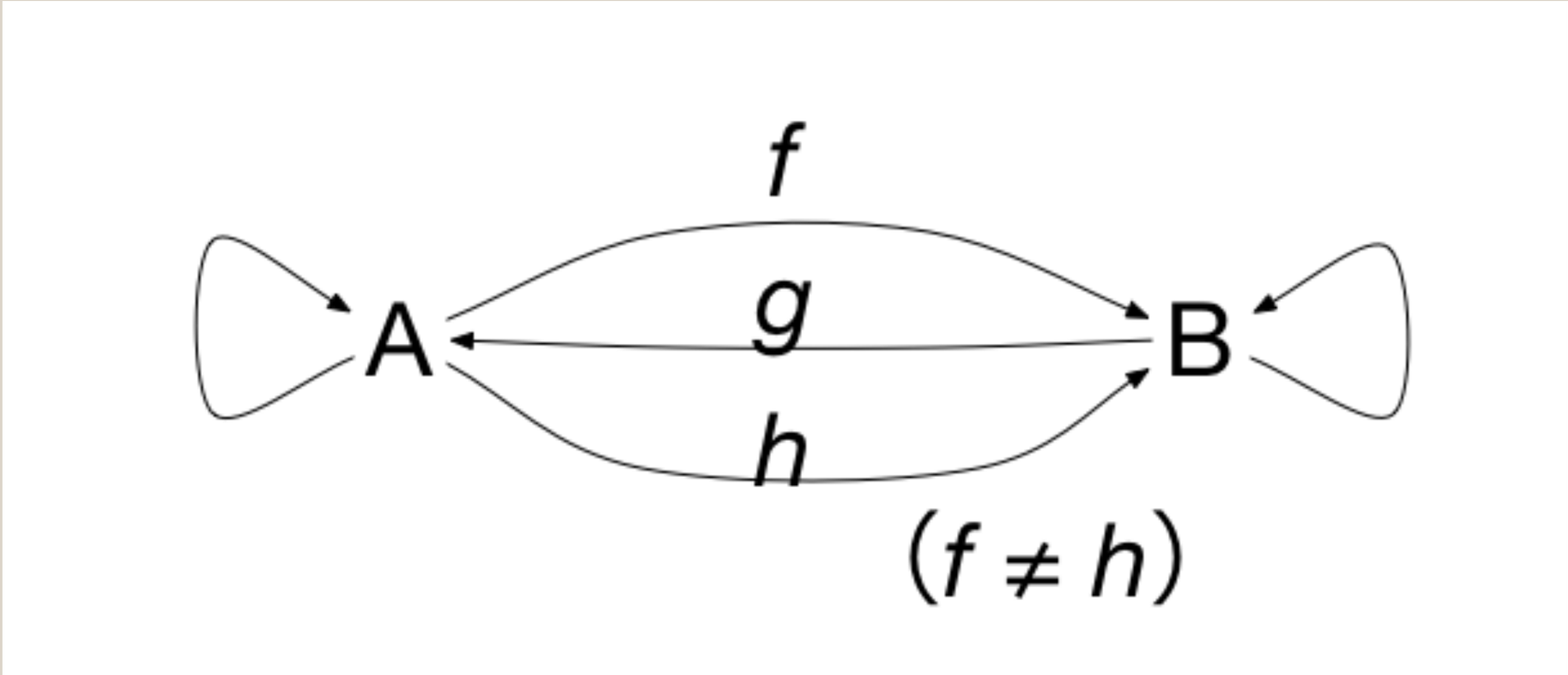
- これは圏になっている（$f \circ g = 1_A$）

(4) 対象間の射がふたつ：その2



- これは圏になっている（$A \to B$ の射が複数あっても問題ない）

(5) 対象間の射がみっつ？？



- こんな圏はない（それはなぜでしょうか？）

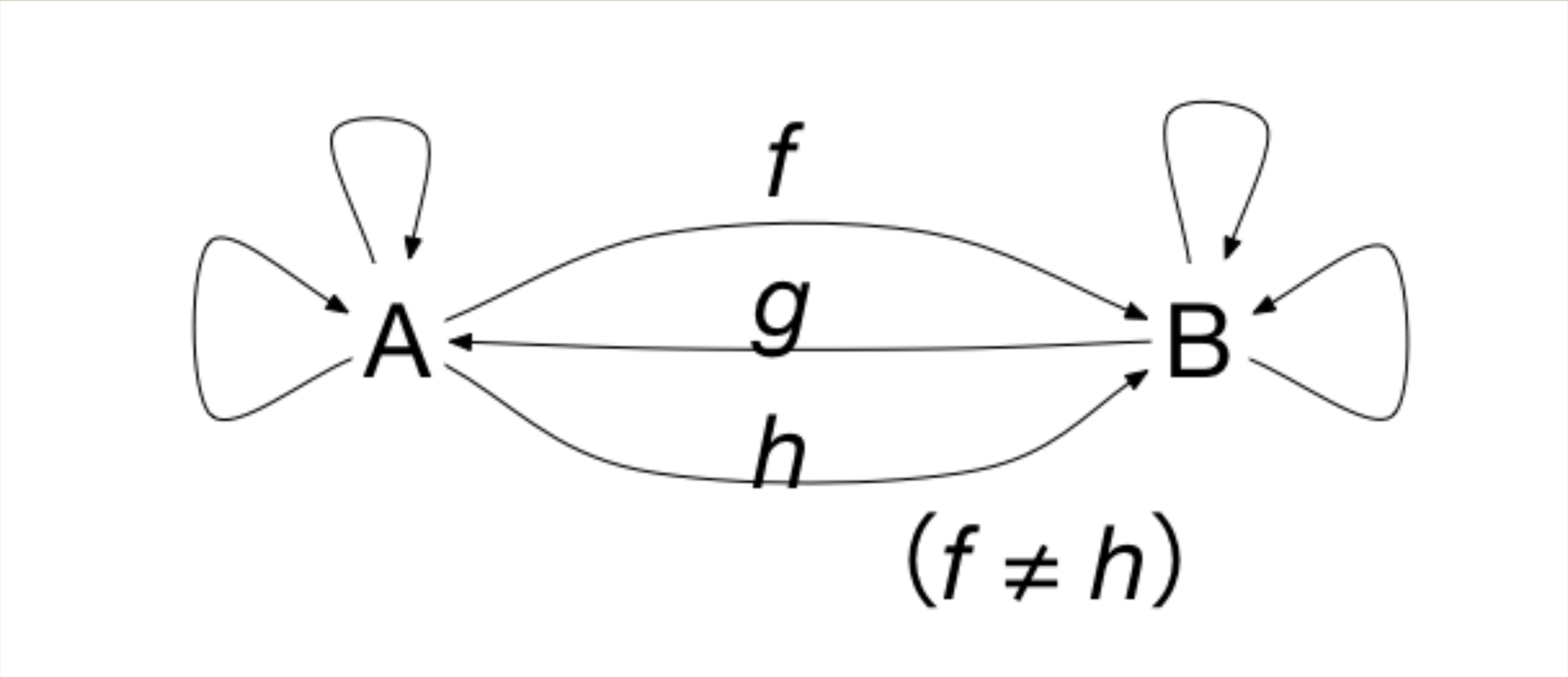
(5) 対象間の射がみっつ？？

- $g \circ f = 1_A$ と $h \circ g = 1_B$ に注意する：なぜなら、それに相当する射が他に存在しないから

そのことから、

- $h \circ (g \circ f) = h \circ 1_A = h$
- $(h \circ g) \circ f = 1_B \circ f = f$
- したがって $f = h$ となる（仮定に反する）

(6) 対象間の射がみっつ



- この場合は $g \circ f$ が 1_A でない射になりうる

Haskell における圏

対象は Haskell の型とする

- 基本データ型 : Int, Char, ...
- リスト : [Int], [Char], ...
- Maybe 型 : Maybe Int, Maybe Char, ...

などのものが対象

- Maybe 型について補足 : Maybe Int の値は Just 0, Just 1, ... または Nothing

射は Haskell の関数とする

```haskell
incr :: Int -> Int
incr x = x + 1
```

- `incr` は `Int` 型の値を受け取り `Int` 型の値を返す関数
- これを `Int` から `Int` への射とみなす
- この対象と射によって圏ができる `:` 以後 **Hask** と書く
  - なお、恒等射は恒等関数 `id`、合成は関数合成 `(.)`

引数を複数持つ関数は？

```
add :: Int -> Int -> Int
add x y = x + y
```

- `add` は二つの `Int` 型の値を受け取り `Int` 型の値を返す関数
- ・・・という解釈のままだと **Hask** の射ではない

カリー化

```
add :: Int -> (Int -> Int)
```

- `add` は `Int` 型の値を受け取り `Int -> Int` 型の値を返す

```
add 3 :: Int -> Int
(add 3) y = 3 + y -- `add 3` は「3 を足す関数」である
```

- `Int -> Int` は関数の型であり、**Hask** の対象
- `add` は対象 `Int` から `Int -> Int` への **Hask** の射

余談

- 多引数関数を1引数関数とみなす方法としてタプルも考えられる
  - `(Int, Int) -> Int`
- しかし、Haskell のカリー化のほうが、圏論に適合させるという意味ではエレガントだと思われる

Hask の部分圏

- 特定の型だけを集めて部分圏を考えることができる
- 例：リスト型だけを対象とした部分圏 **List**
- 例：`Maybe` 型だけを対象とした部分圏 **Maybe**

関手（Functor）

関手の定義

- 圏から圏への関手 $F : \mathbf{C_1} \to \mathbf{C_2}$
- A が $\mathbf{C_1}$ の対象 $\Rightarrow$ $F(A)$ は $\mathbf{C_2}$ の対象
- f が $\mathbf{C_1}$ の射 $\Rightarrow$ $F(f)$ は $\mathbf{C_2}$ の射
- $\mathbf{C_1}$ の恒等射 1_A について $F(1_A) = 1_{F(A)}$
- $\mathbf{C_1}$ の合成 $g \circ f$ について $F(g \circ f) = F(g) \circ F(f)$

Haskell における関手

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

- Functor という型クラスがある
- これが実際に圏論における関手の概念にあたるもの
- より明示的に言うと、Functor 型クラスのインスタンスが関手

関手の例 `Maybe`

- `Maybe` は `Functor` 型クラスのインスタンス
- これは、圏 **Hask** から圏 **Maybe** への関手である
- `Maybe` は任意の型 `T` から新しい型 `Maybe T` を作る
  - `Int` から `Maybe Int`, `Char` から `Maybe Char`, ...
- つまり、**Hask** の対象を **Maybe** の対象にうつす

関手の例 Maybe

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
fmap f :: Maybe a -> Maybe b
fmap f (Just x) = Just (f x)
fmap f Nothing = Nothing
```

- `f :: a -> b` を `fmap f :: Maybe a -> Maybe b` に
- つまり、**Hask** の射を **Maybe** の射にうつす
- したがって、`Maybe` は **Hask** から **Maybe** への関手

Functor で何が嬉しいのか

- Maybe Int 型の値をそのまま扱おうとすると場合分けが必要
  - 値を持つ Just 1 などの場合
  - 値を持たない Nothing の場合
- Functor の考えを使えば、Nothing のことは忘れていい
  - Int についてだけ関数定義すればいい

Haskell の Functor の注意

- `Functor` 型クラスのインスタンスを作っても、それが関手の定義を満たしていることは保証されない
- 対象と対象の対応、射と射と対応は、自然にできる
- 恒等射と合成については、定義を満たすように自分で気をつける

例 : 関手にならないもの

```
CMaybe a = CNothing | CJust Int a
fmap f CNothing = CNothing
fmap f (CJust counter x) = CJust (counter + 1) (f x)
```

以下のように恒等射を保たない、また合成も保たない

```
fmap id (CJust 0 "abc") -- CJust 1 "abc"
id (CJust 0 "abc") -- CJust 0 "abc"
```

さらなる話題

さらなる話題

- 圏論におけるモナド ： 自己関手で自然変換を持つもの
- Haskell で重視されるのはモナド（Monad）
  - Monad 同士をチェインさせることができる
  - 副作用を Monad に閉じ込めることができる
- それはまたそのうち・・・