

# fold for all

coord\_e@coins20LT 第 14 回



# Recursion is the goto of functional programming

- Erik Meijer (人) 曰く  
<https://twitter.com/headinthebox/status/384105824315928577>
- 明示的な再帰は goto みたいなもの
- 振る舞いについて論じにくい
  - 等式変形ができない、やりにくい
  - 最適化がしにくい

# ループもないくせに何いうてはるのです？

- 再帰を閉じ込めたコンビネータを組み合わせる

# パワー1

```
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (h:t) = f h : map f t
```

## パワー 2

foldr は構造の要素に次々に関数を適用していく畳み込み関数の一つ。foldr (☆) '王' "遊戯" とすると '遊' ☆ ('戯' ☆ '王') のように右結合の形になる。

<https://scrapbox.io/haskell-shoen/foldrトリック>

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc []      = acc
foldr f acc (h:t) = foldr f (f h acc) t
```

## map, foldr, あとたまに unfold があればだいたいなんでもできる

```
-- (+)   は \a b -> a + b
-- (*2)  は \x  -> x*2
-- の糖衣構文
doubleSum :: [Int] -> Int
doubleSum = foldr (+) 0 . map (*2)
```

## うれしさ

```
-- 停止性をいうのに帰納法がいる
doubleSumRec []      = []
doubleSumRec (h:t) = h*2 + doubleSumRec t
```

```
doubleSum = foldr (+) 0 . map (*2) -- 有限のリストに対しては止まる
          = foldr ((+) . (*2)) 0    -- 等式変形による最適化
```

```
incDoubleSum = doubleSum . map (+1)
              = (foldr ((+) . (*2)) 0) . map (+1)    -- 等式変形
              = foldr ((+) . (*2) . (+1)) 0          -- 等式変形による最適化
incDoubleSumRec = doubleSumRec . map (+1)            -- これ以上変形できない
```

**リストだけなんですか？**



## まあ型クラスとコード生成でいけるというのはそう

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

class Foldable f where
  foldr :: (a -> b -> b) -> b -> f a -> b

data MyList a
  = MyNil
  | MyCons a (MyList a)
  deriving (Functor, Foldable)
```

# ここから本題

fold を map からつくりたい！

## fold を map からつくる

わかりにくいので `Int` で特殊化します

```
data IntList
  = Nil
  | Cons Int IntList
```

## 再帰をとりだす

```
data IntListF a
  = NilF
  | ConsF Int a
  deriving Functor -- fmap :: (a -> b) -> IntListF a -> IntListF b

data Fix f = Fix (f (Fix f))
-- Fix :: f (Fix f) -> Fix f

unFix :: Fix f -> f (Fix f)
unFix (Fix f) = f
```

- ↑ として `Fix IntListF` は `IntList` と同型
- `data FixIntList = Fix (IntListF FixIntList)` と考えるとわかりやすい

## 再帰をとりだす

実際下の関数が定義できる

```
-- 定義は省略
fromListF :: Fix IntListF -> IntList
toListF   :: IntList -> Fix IntListF
```

# fold を書く

fmap から fold が導出できた！

```
foldListF :: (IntListF a -> a) -> Fix IntListF -> a
foldListF f = f . fmap (foldListF f) . unFix
```

-- 畳み込みの例

```
length :: Fix IntList -> Int
length = foldListF $ \case
  NilF      -> 0
  ConsF _ acc -> 1 + acc

sum :: Fix IntList -> Int
sum = foldListF $ \case
  NilF      -> 0
  ConsF x acc -> x + acc
```

# 型付け

```
foldListF f      :: Fix IntListF -> a
fmap             :: (a          -> b) -> f      a          -> f      b
fmap (foldListF f) :: IntListF (Fix IntListF) -> IntListF a
```

```
unFix           :: Fix f          -> f          (Fix f)
fmap (foldListF f) :: IntListF (Fix IntListF) -> IntListF a
fmap (foldListF f) . unFix :: Fix IntListF      -> IntListF a
```

```
fmap (foldListF f) . unFix :: Fix IntListF -> IntListF a
f                               :: IntListF a -> a
f . fmap (foldListF f) . unFix :: Fix IntListF      -> a
```

```
foldListF :: (IntListF a -> a) -> Fix IntListF -> a
foldListF f = f . fmap (foldListF f) . unFix
```

## これは fold です（大声）

- `IntListF (Fix IntListF)` の `Fix IntListF` の部分（再帰的な出現）に `fmap` で畳み込みの処理を流す
  - `NilF` には `Fix IntListF` がないため no-op になって止まるのがミソ
- 結果これまでの畳込みの結果が `a` に詰まった `IntListF a` ができたので最後に今回の `f` を投げて完成

```
foldListF :: (IntListF a -> a) -> Fix IntListF -> a
foldListF f = f . fmap (foldListF f) . unFix
--           ^      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ ^ 非本質
--           |      \ 再帰的な部分について畳み込みをやる
--           - 今までの畳込みが終わり、このステップをやる
```



## これは fold です (大声)

- `IntListF a -> a` が畳み込みの本体
  - `IntListF a` は再帰的な部分が `a` になっている
  - 今までの畳込みの結果が `a` に入っているとき、次の結果 `a` をつくる
- リストでは2ケースあった: `foldr :: (Int -> a -> a) -> a -> [Int] -> a`
  - 最初の結果: `NilF -> init`
  - `f(今の要素、今までの結果) = 次の結果`: `ConsF x acc -> f x acc`

```
-- foldr の2引数は IntListF a -> a に変換可能
folder :: (Int -> a -> a) -> a -> IntListF a -> a
folder f init NilF = init
folder f init (ConsF x acc) = f x acc
```

## これは fold です（大声）

（再帰せずに）いつもの `foldr` に変形可能

```
foldr :: (Int -> a -> a) -> a -> IntList -> a
foldr f i = foldListF g . toListF
  where
    g NilF = i
    g (ConsF x acc) = f x acc
```

## 要求が `fmap` だけということは

- リストに限らず任意の `Functor` で畳み込みが可能
  - `fmap` できる構造は `Fix` で再帰的にすることができ、
  - そうしたものは `cata` で畳み込みができる
- Catamorphism と呼ぶ（カッコいい）

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix
```

# 応用例

```
-- 構文木
data ExprF a
  = NumF Int
  | NegF a
  | AddF a a
  | MulF a a
  deriving Functor

-- 再帰しない eval
eval :: Fix ExprF -> Int
eval = cata $ \case
  NumF i   -> i
  NegF v   -> -v
  AddF v1 v2 -> v1 + v2
  MulF v1 v2 -> v1 * v2
```

## まとめ

- 明示的な再帰を嫌おうという政治的主張の紹介
- 再帰をとりだすとなんか fmap だけで畳み込みできてすごいね

## 実世界

- [hnix: Haskell implementation of the Nix language](#)
- [dhall - crates.io](#)

## 参考文献

- 矢澤にこ先輩と一緒に catamorphism !
- おじいさん、今日のご飯は Catamorphism ですよ
- Recursion Schemes - haskell-shoen