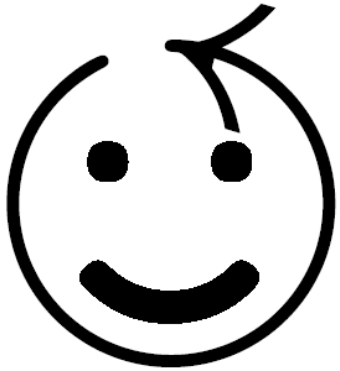
Introduction to Categorical Programming (revised)

HAMA.jp 2009
酒井 政裕 @ヒビルテ



今日話したいこと

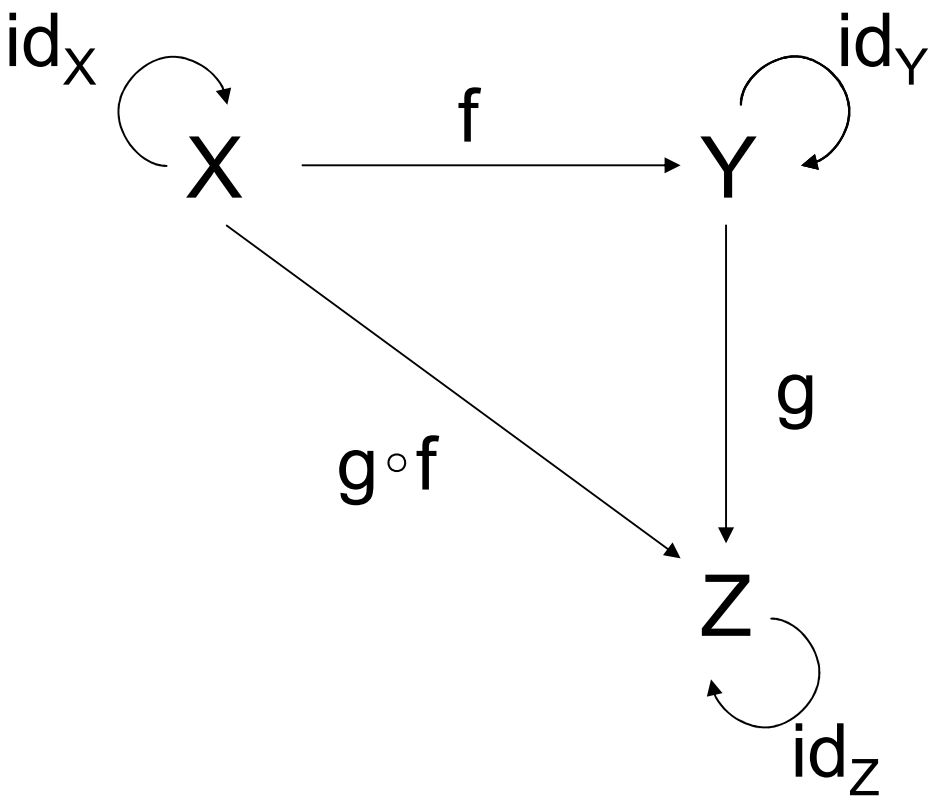
- なぜ圏論か?
- Haskell での圏論プログラミング
- The Evolution of a Haskell Programmer
- 圏論プログラミング言語 CPL

圏論とは?

- 圏論（けんろん、category theory）は、数学的構造とその間の関係を抽象的に扱う数学理論の１つである。考えている種類の「構造」を持った対象とその構造を反映するような対象間の射の集まりからなる圏が基本的な考察の対象になる。

Wikipedia「圏論」より

圏

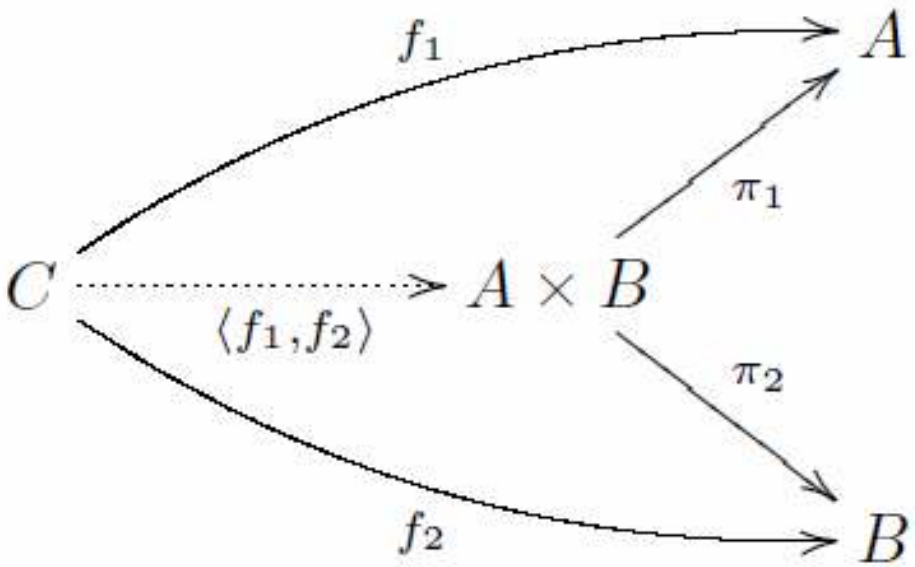


- 対象(Object)
  X, Y, Z, …
- 射(Morphism)
  f, g, id_X, …
- 射の合成: ∘

- 結合律
  $(h \circ g) \circ f = h \circ (g \circ f)$
- 単位元
  $f \circ id_X = f = id_Y \circ f$

圏論

- 対象と射(矢印)による抽象化
- 等式を図式で表現



具体例色々

対象	射
集合	関数
位相空間	連続関数
群	準同型
型	プログラム

Haskellのデータ型と関数は圏になる

- 対象=データ型
  - Int, Bool, [Int], Maybe Int, …
- 射=関数
  - not :: Bool→Bool, concat :: [[a]] → [a], …
- 恒等射:
  - id
- 射の合成=関数合成
  - f . g
- 単位元律
  - id . f = f = f . Id
- 結合律
  - (f . g) . h = f . (g . h)

圏論の考え方を適用できる、が…

抽象的無意味

- 圏論は

general abstract nonsense

- と呼ぶ人もいるくらい、一般的かつ抽象的
- よくある質問:
  - そんなのが、プログラミングに何の関係が?
  - いったい何の役に立つのか?

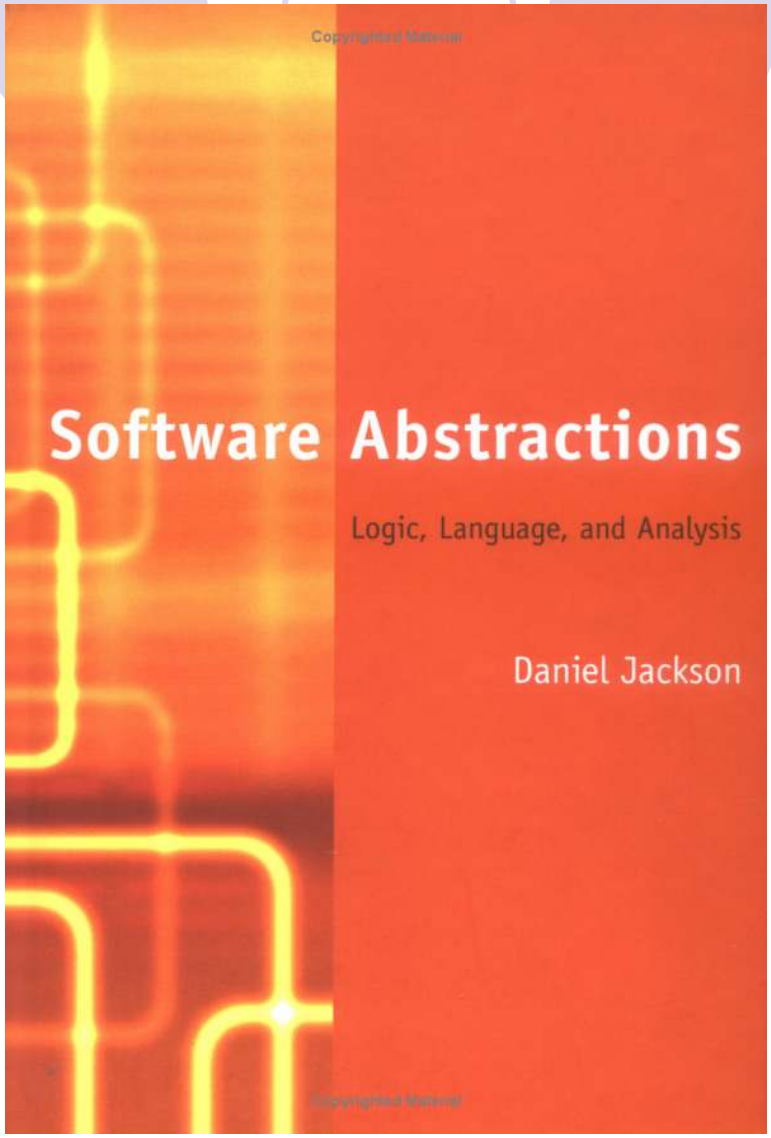
なぜ圏論なのか?

- ソフトウェアにとって「抽象化」は死活的に重要
  - 対象ドメインの概念を計算機上でどう表現する?
  - どうレイヤやモジュールを分け、どういう構造でプログラムを書くのか?
- 圏論は数学で育まれた抽象化の技法の宝庫
  - ソフトウェアやプログラミングに使える概念も沢山
  - 特に関数型言語は数学や圏論と相性が良い
    Haskellを使ってて良かったね (^^

ここで CM です

- Software Abstractions
  - Daniel Jackson
  - MIT Press / April 2006

- 抽象化とソフトウェアの設計に悩むあなたに。
  - 注: 圏論とは特に関係ありません。



今日話したいこと

- なぜ圏論か?
- Haskell での圏論プログラミング
- The Evolution of a Haskell Programmer
- 圏論プログラミング言語 CPL

Haskell での圏論プログラミング

圏論プログラミング

＝

圏論の概念を使って、
プログラムを構造化

Haskell での圏論プログラミング

- Haskellで圏論というとモナドとかArrowとか?
- 今回はそういう話ではなくて、
  再帰的なプログラムの書き方についての話
- まずは圏論のことは一度忘れて、普通の
  Haskellプログラムを……

リストの構造に関する再帰

- 例:
- sum :: [Int] → Int
  ```
  sum [] = 0
  sum (x:xs) =
    x + sum xs
  ```
- length :: [a] → Int
  ```
  length [] = 1
  length (x:xs) =
    1 + length xs
  ```
- パターン
  - 空リストの場合の値
  - consの場合の値を、headの値と、tailに対して関数を適用した結果から計算
- コードで書くと:
  ```
  f :: [X] → Y
  f [] = n
  f (x:xs) = c x (f xs)
  ```

高階関数foldrによるパターンの抽象化

- パターン:

```
f [] = n
f (x:xs) = c x (f xs)
```

- 高階関数として抽象化!

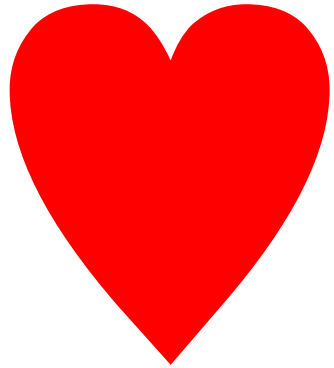
```
foldr :: (a→b→b) → b
       → [a] → b
foldr c n [] = n
foldr c n (x:xs) =
  c x (foldr c n xs)
```

foldrを使った定義例:

- sum = foldr (+) 0
- length = foldr
  (λ _ n → n+1) 0
- xs ++ ys =
  foldr (:) ys xs
- map f = foldr
  (λ x xs → f x : xs) []
- …

なぜfoldrのような形で関数を書くのか?

- 良い性質をもった再帰だから
  - 人間が読む上で、処理の流れが分かりやすい
  - c と n が停止する式で、xsが有限リストなら、foldr c n xs も停止する。
  - プログラムの性質が推論しやすい
    - 例) foldr c n . map f = foldr (λ x a → c (f x) a) n
  - 結果として、最適化しやすい
- それに対して、無制限の再帰は
  - 命令型言語での goto のようなもので、やっかい

foldr 

- 明示的な再帰を使って書く前に、foldrとかで書けないか、考えて見ましょう。

foldrと圏論の関係は?

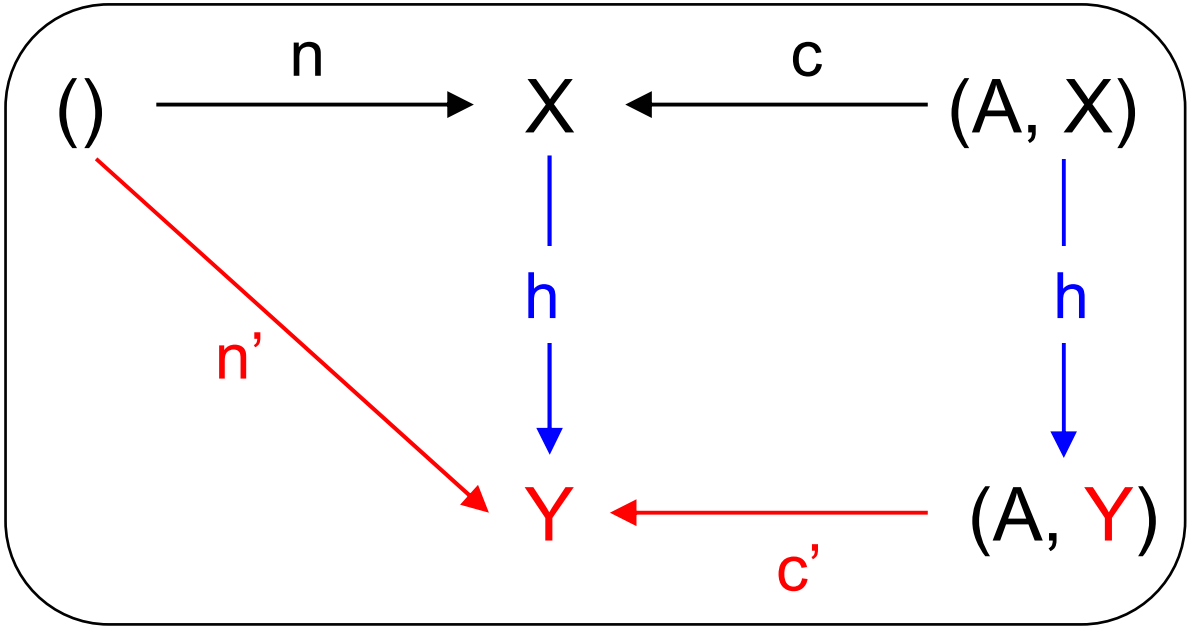
- リスト型とは

  nil, cons, foldr

  が定義された抽象データ型
- foldrは単なる便利な高階関数ではなく、実はリスト型にとって根源的な関数
- ⇒ 圏論的なリスト型の定義に由来

圏論でのリスト型の定義

- 型Xと関数 n :: ()→X, c :: (A, X)→X の組で、各 Y, n' :: ()→Y, c :: (A,Y)→Y に対して h . n = n' と c' . h = h . c を満たす h :: X→Y が唯一つ存在するもの



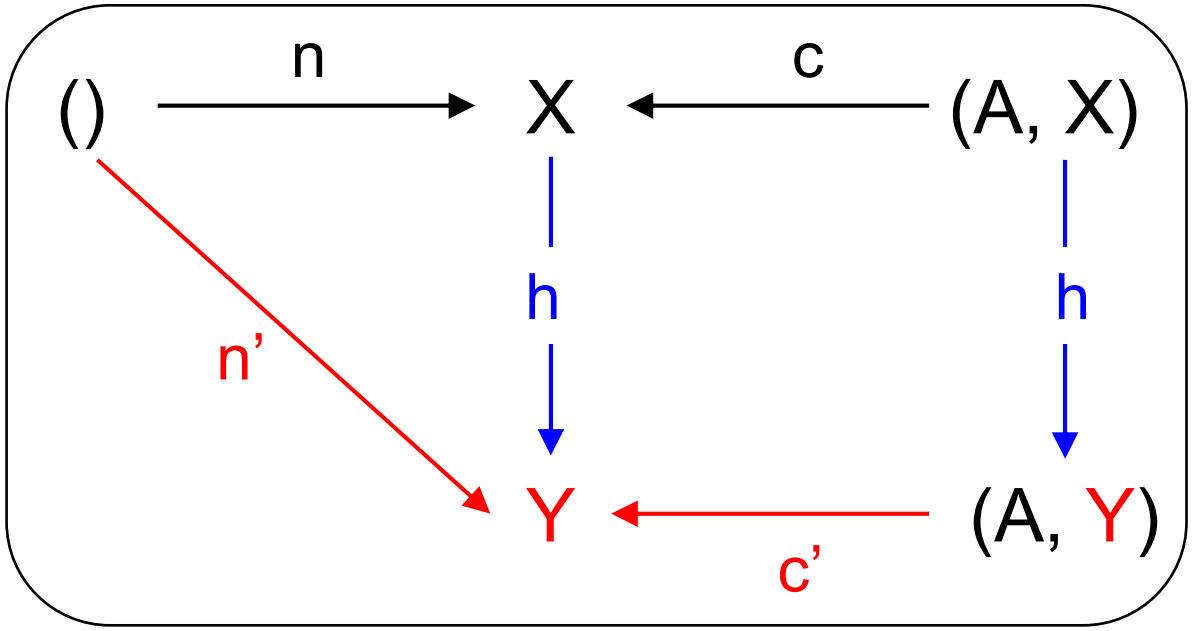
圏論では等式を図式で表現:

図式が可換
⇔
二点間の任意の経路にそって合成した結果が等しい

Haskellのリスト型との対応

- X = [A]
- n ≒ []
- c ≒ (:)
- h ≒ foldr n' c'

- h . n = n' ⇔
  foldr n' c' [] = n'
- h . c = c' . h ⇔
  foldr n' c' (x:xs) = c' x (foldr n' c' xs)



面倒なので、
以降では暗黙に
(非)カリー化し、
また X と ()→X
を同一視する

普遍性

- 「〇〇で、各〇〇に対して、△△を満たす関数がただ一つ存在する」
- この性質は「普遍性」と呼ばれる
- 圏論では普遍性でデータ型を特徴付ける

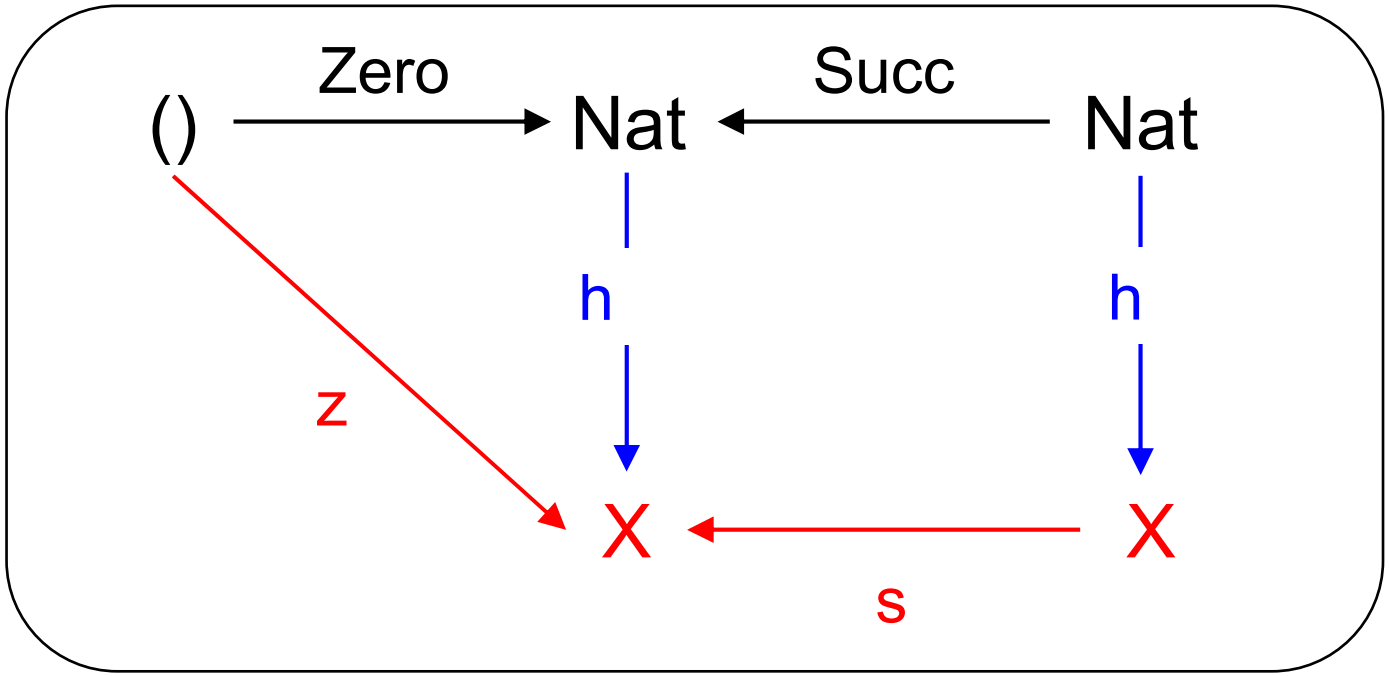
- 普遍性を満たせば、実装は何でも良い
  - たとえば チャーチエンコーディングとか http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt

他の帰納的データ型の場合の例 (自然数)

- 自然数のデータ型
  - **data** Nat = Zero
    | Succ Nat
- 畳み込み用の関数
  - iter :: a→(a→a)
    → (Nat→a)
    iter z s Zero = z
    iter z s (Succ n) =
    s (iter z s n)
  - (iter z s n は z に s を n回適用する)
- iterを使った定義例:
  - plus :: Nat→Nat→Nat
    plus n = iter n Succ
  - mult :: Nat→Nat→Nat
    mult n =
    iter Zero (plus n)

自然数の特徴付け

任意の型X と z :: X, s :: X→X に対して、
下図を可換にする関数 h : Nat→X が
唯一つ存在。(これが iter z s)



この先の話

- このような話はリストや自然数に限らず、他の帰納的データ型でも実は同様
- 有限リストを消費するfoldrの双対、無限リストを生成するunfoldr
- より複雑な関数を表現する方法
  - Hylomorphism, Paramorphism, Histomorphism, Comonadic Iteration, …
- ⇒ とても全部は無理だけど、一部を The Evolution of a Haskell Programmer をネタに紹介

目次

- なぜに圏論?
- Haskell での圏論プログラミング
- The Evolution of a Haskell Programmer
- 圏論プログラミング言語 CPL

The Evolution of a Haskell Programmer

- http://www.willamette.edu/~fruehr/haskell/evolution.html
- 階乗を色々な書き方で書いてみる話
- Categorical Programming に関係するもの
  - Beginning graduate Haskell programmer
  - Origamist Haskell programmer
  - Cartesianally-inclined Haskell programmer
  - Ph.D. Haskell programmer
  - Post-doc Haskell programmer
- 以降の説明は分からなくても気にしないで

Beginning graduate Haskell programmer

(graduate education tends to liberate one from petty concerns about, e.g., the efficiency of hardware-based integers)

```
-- Nat, plus, mult の定義
…

-- primitive recursion
primrec :: a → (Nat → a
    → a) → Nat → a
primrec z s  Zero    = z
primrec z s (Succ n) =
    s n (primrec z s n)
```

iterの強化版:
sの引数にnが追加

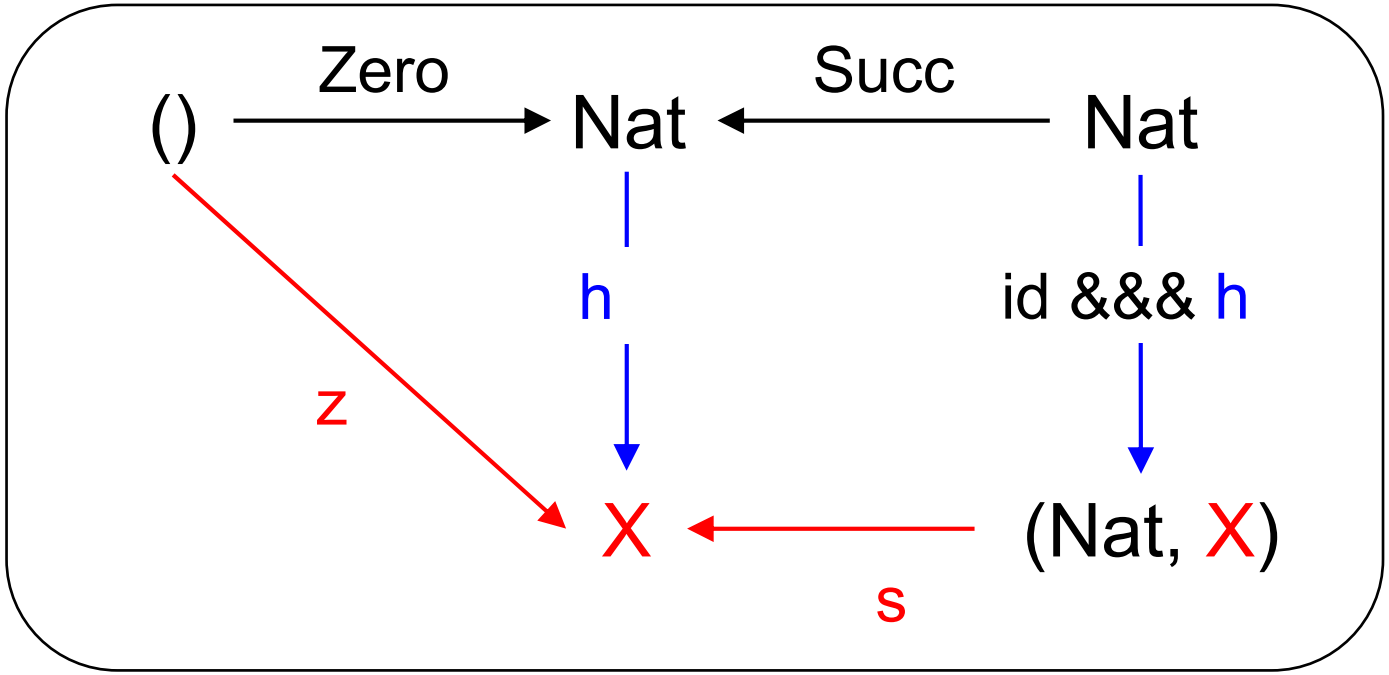
```
-- two versions of
    factorial
…
fac' :: Nat → Nat
fac' = primrec one (mult .
    Succ)
…
(zero : one : two :
    three : four : five : _) =
    iterate Succ Zero
```

原始帰納法

- 定義
  - primrec z s Zero   = z
  - primrec z s (Succ n) = s n (primrec z s n)
  - fac' = primrec one (mult . Succ)
- これって本当に階乗になってる?
  - fac’ Zero = primrec one (mult . Succ) Zero = one
  - fac’ (Succ n)
    = primrec one (mult . Succ) (Succ n)
    = (mult . Succ) n (primrec one (mult . Succ) n)
    = (mult . Succ) n (fac’ n) = mult (Succ n) (fac’ n)

原始帰納法 (cont'd)

- 自然数の普遍性の別の形:
  任意の型 X と z :: X と s :: Nat→X→X に対して、下図を可換にする h :: Nat→X がただ一つ存在。(それが primrec z s)



原始帰納法 (cont'd)

- primrec
  - primrec :: a → (Nat → a → a) → Nat → a
  - primrec z s Zero = z
  - primrec z s (Succ n) = s n (primrec z s n)
- 実はiterで表現できる
  - primrec' z s = snd .
    iter (Zero, z) (λ (a,b) → (Succ a, s a b))
  - 練習問題: 等しさを証明してみよう

Origamist Haskell programmer

(always starts out with the “basic Bird fold”)

```
-- (curried, list) fold and an application
fold c n []     = n
fold c n (x:xs) = c x (fold c n xs)

prod = fold (*) 1
```

= foldr

```
-- (curried, boolean-based, list) unfold and
an application
unfold p f g x =
  if p x
     then []
     else f x : unfold p f g (g x)

downfrom = unfold (==0) id pred
```

≒unfoldr

```
-- hylomorphisms, as-is or "unfolded"
(ouch! sorry ...)

refold  c n p f g   =
  fold c n . unfold p f g

refold' c n p f g x =
  if p x
     then n
     else c (f x) (refold' c n p f g (g x))


-- several versions of factorial, all
(extensionally) equivalent
fac   = prod . downfrom
fac'  = refold  (*) 1 (==0) id pred
fac'' = refold' (*) 1 (==0) id pred
```

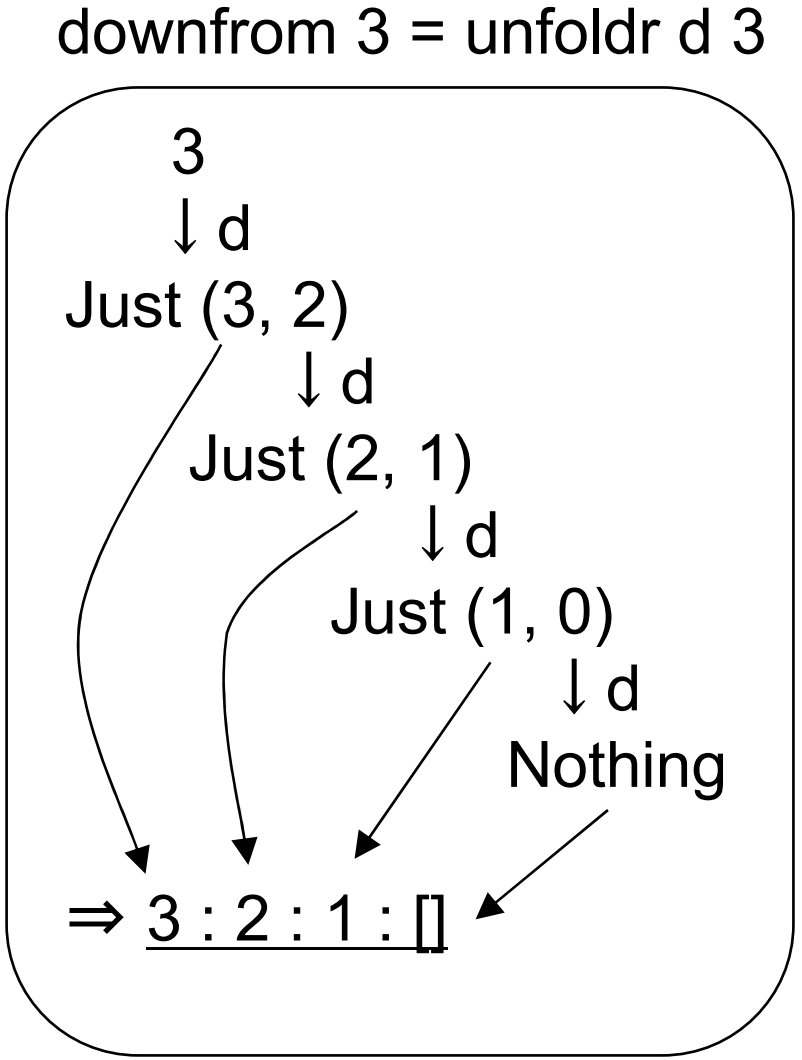
標準関数で書き直すと

```
prod :: [Int] → Int
prod = foldr (*) 1

downfrom :: Int → [Int]
downfrom = unfoldr d

d :: Int → Maybe (Int, Int)
d 0 = Nothing
d x = Just (x,x-1)

-- hylomorphisms (as-is)
refold :: (a → b → b) → b
          → (c → Maybe (a, c))
          → (c → b)
refold  c n f = foldr c n . unfoldr f
```

```
-- hylomorphism (unfolded)
refold' c n f x =
  case f x of
    Nothing → n
    Just (a,x) → c a (refold' c n f x)

-- several versions of factorial, all
(extensionally) equivalent
fac :: Int → Int
fac   = prod . downfrom
fac'  = refold  (*) 1 d
fac'' = refold' (*) 1 d
```

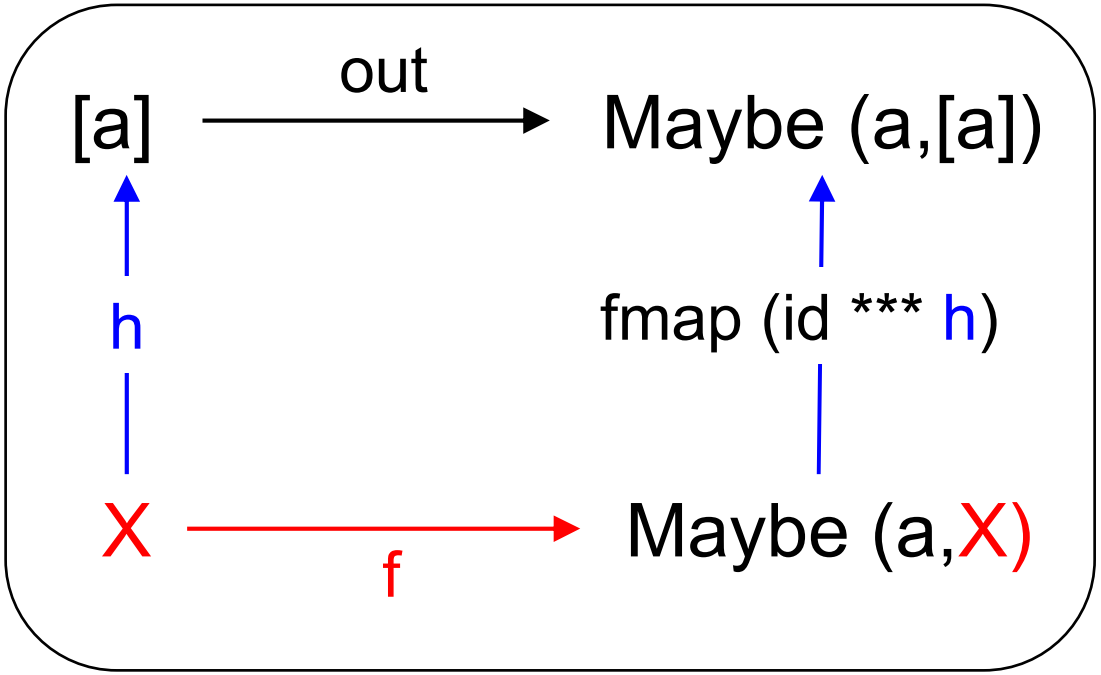
無限リストを生成する関数 unfoldr

```
unfoldr
  :: (b → Maybe (a, b))
 → (b → [a])
unfoldr f x =
  case f x of
     Nothing → []
     Just (a, y) →
         a : unfoldr f y
```



無限リストの型の圏論的な定義

- 任意の型 X, 関数 f :: X → Maybe (a, X) に対して、以下を可換にする関数 h : X→[a] が唯一つ存在。(それが unfoldr f)




```
-- リストの分解関数
out :: [a]→Maybe (a,[a])
out [] = Nothing
out (x:xs) = Just (x, xs)
```

Hylomorphism (refold)

- Haskellでは

  有限リストの型 = 無限リストの型

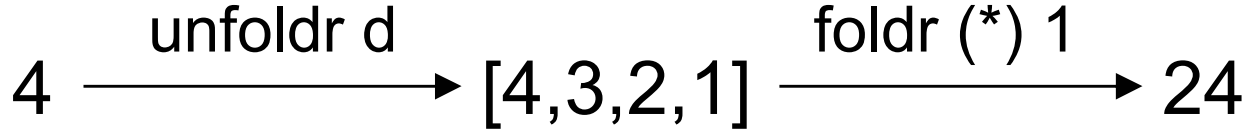
  なので、
  - unfoldr f :: x→ [a]
  - foldr c n :: [a]→y

  が結合できる!

- refold c n f = foldr c n . unfoldr f :: x → y

Hylomorphismを使った階乗の定義

- 階乗の定義
  - fac = refold (*) 1 d = foldr (*) 1 . unfoldr d
- 計算例



- 中間データが生成されて、効率が悪いが、refold’ を使えば中間データ無しに計算可能

Cartesianally-inclined Haskell programmer

(prefers Greek food, avoids the spicy Indian stuff;
inspired by Lex Augusteijn's "Sorting Morphisms" [3])

- Origamist と同じく Hylomorphism で定義
- ただし、関数名がギリシャ語風に
  - foldr → cata (Catamorphism)
  - unfoldr → ana (Anamorphism)
  - refold → hylo (Hylomorphism)
- 引数の与え方も少し変わっている

Ph.D. Haskell programmer

(ate so many bananas that his eyes bugged out, now he needs new lenses!)

- またもや Hylomorphism を使った定義だけど……
- 型レベルでの不動点演算子を使って型定義

  newtype Mu f = In (f (Mu f))

  data N x = Zero | Succ x

  type Nat = Mu N
- 再帰を行う関数も汎用的なものに
  - foldr や iter が
    cata :: Functor f ⇒ (f a → a) → (Mu f → a) に。
  - unfoldr が
    ana :: Functor f ⇒ (a → f a) → (a → Mu f) に。

Post-doc Haskell programmer

(from Uustalu, Vene and Pardo's "Recursion Schemes from Comonads" [4])

- Beginning graduate Haskell programmer と同じく、原始帰納法で階乗を定義
- ただし、Ph.D. Haskell programmer での汎用的な定義を利用
- また、原始帰納法を直接書くのではなくて、Comonadic Iteration という超一般的な再帰パターンの特別な場合として記述

目次

- なぜに圏論?
- Haskell での圏論プログラミング
- The Evolution of a Haskell Programmer
- 圏論プログラミング言語 CPL

圏論プログラミング言語 CPL

- CPL (Categorical Programming Language)
- 圏論プログラミングの源流
- 特徴
  - 圏論に基づいたデータ型定義
  - 型定義から導出される項書き換え規則による実行
  - ポイントフリー
  - 停止性の保証

プログラム例

```
-- 終対象 (ユニット型)
right object 1 with !
end object;

-- 直積 (タプル)
right object prod(a,b) with pair is
  pi1: prod → a
  pi2: prod → b
end object;

-- べき対象 (関数型)
right object exp(a,b) with curry is
  ev: prod(exp,a) → b
end object;
```

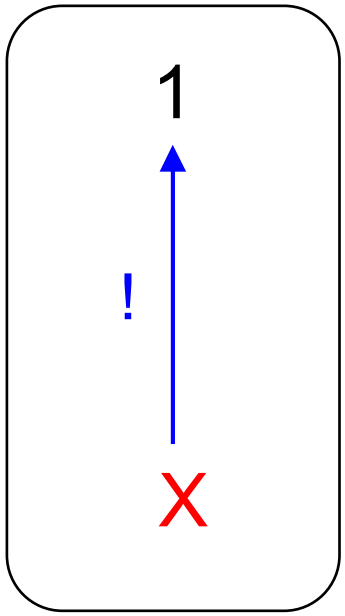
```
--  自然数型
left object nat with pr is
  zero: 1 → nat
  succ: nat → nat
end object;

-- 関数定義の例
let add = ev.pair(pr(curry(pi2),
    curry(s.ev)).pi1, pi2)
let mult = ev.prod(pr(curry(zero.!),
    curry(add.pair(ev, pi2))), id)
let fact = pi1.pr(pair(s.zero, zero),
    pair(mult.pair(s.pi2, pi1), s.pi2))
```

データ型定義

- Left Object / Right Object の二種類
- Left Object
  - 普通の有限のデータ型
  - 自然数, リスト, 論理値, 直和, etc.
  - 値の構造が重要
- Right Object
  - 無限かもしれないデータ型
  - ユニット型, 直積, 関数, 無限リスト, オートマトン, etc,
  - 値の振る舞いが重要
- Haskellと違って有限のリストと無限リストは別の型

終対象 (ユニット型)

right object 1 **with** !
end object;



任意の対象Xが与えられたときに、

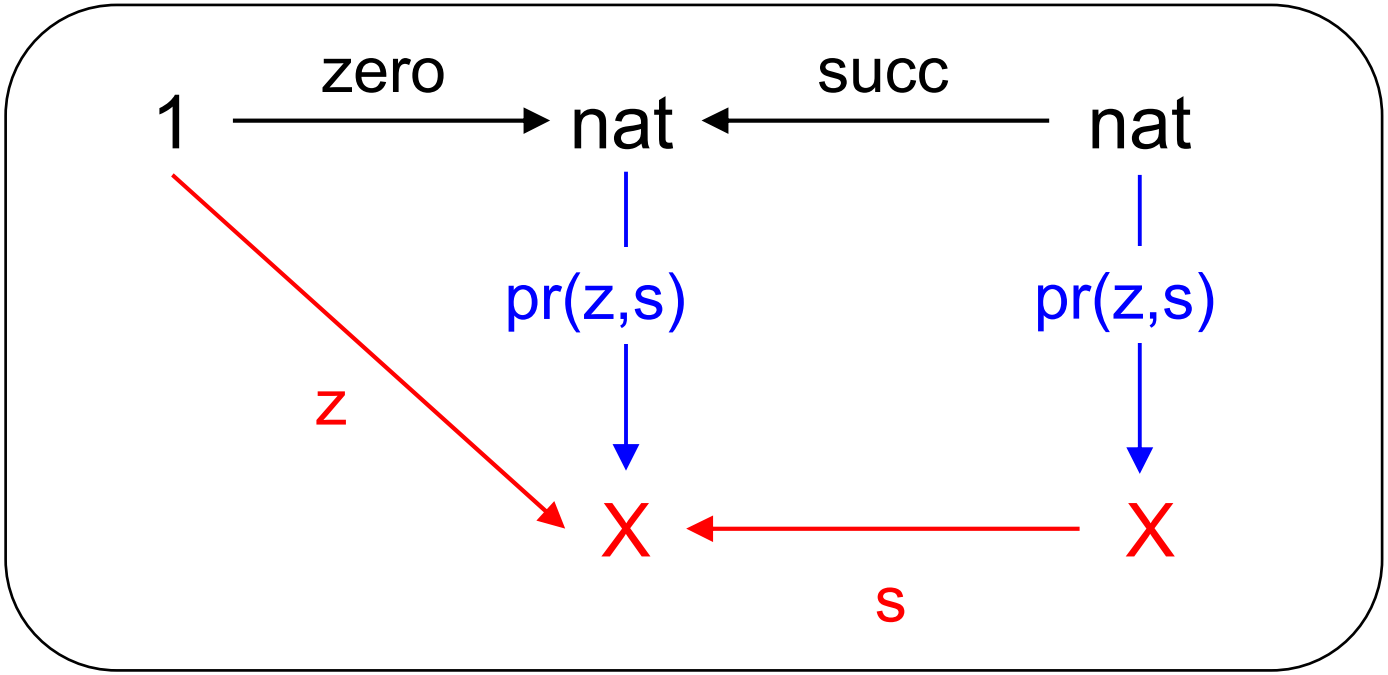
Xから1への関数が一意に存在

その関数を！と表記

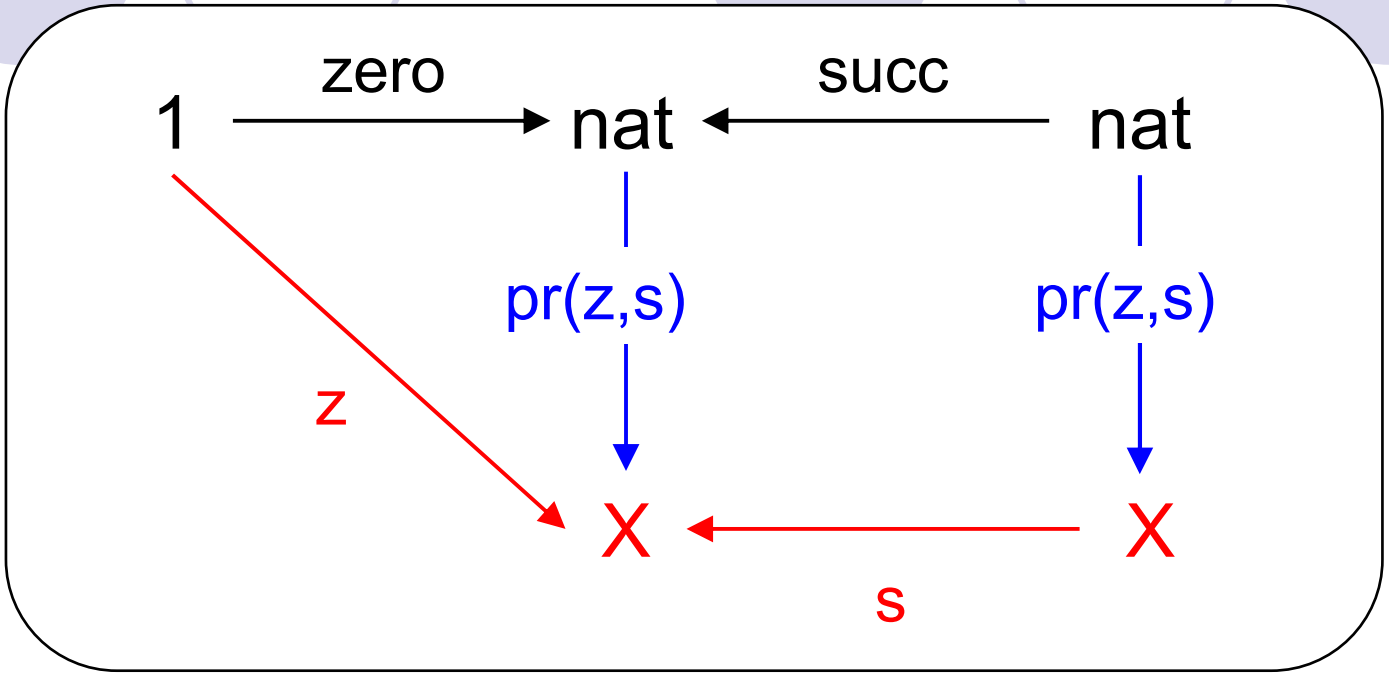
自然数型

left object nat **with** pr **is**
  zero: 1 → nat
  succ: nat → nat
end object;

X, z :1→X, s : X→X に対して、下図を可換にする関数 pr(z,s): nat→X が一意に存在



自然数型



- pr(z,s) . zero = z
- pr(z,s) . succ = s . pr(z,s)

が成り立つ。

これを左辺⇒右辺の書き換え規則とみなす。

Haskellで書いた
iterがprに対応

自然数を用いた計算例

- 2倍する関数
  - double = pr(zero, succ . succ) : nat→nat
- 計算例:
  - double . succ . succ . succ . zero
    - ⇒ pr(zero, succ.succ).succ.succ.zero
    - ⇒ succ.succ.pr(zero, succ.succ).succ.succ.zero
    - ⇒ succ.succ.succ.succ.pr(zero,succ.succ).succ.zero
    - ⇒ succ.succ.succ.succ.succ.succ.pr(zero,succ.succ).zero
    - ⇒ succ.succ.succ.succ.succ.succ.zero

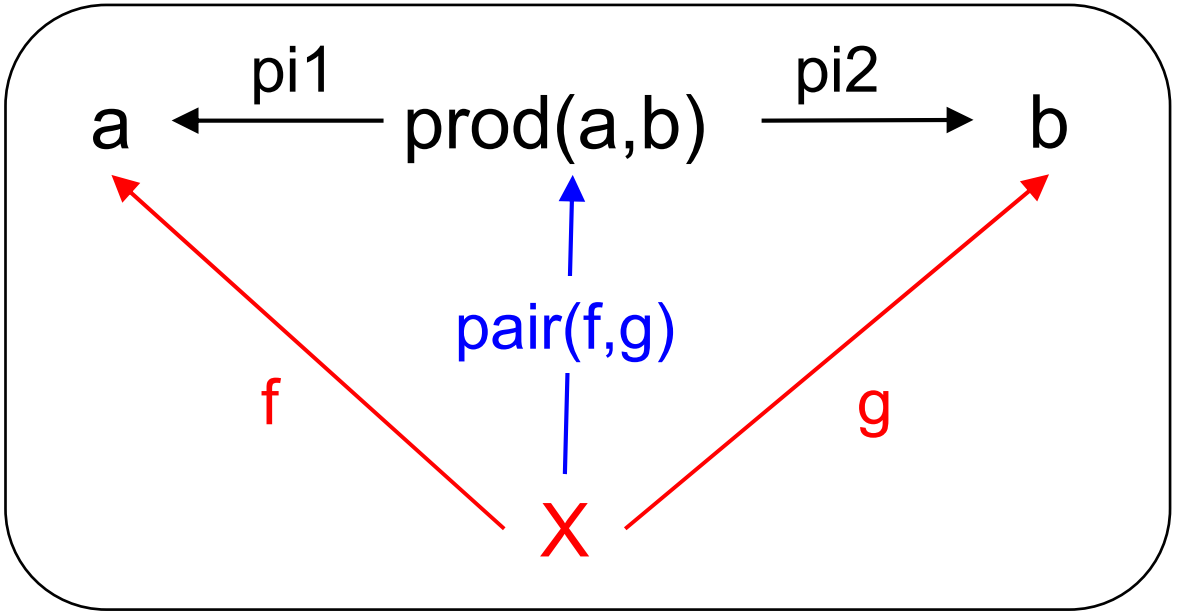
pr(f,g).succ ⇒ g.pr(f,g) を適用

pr(f,g).zero ⇒ f を適用

直積 (タプル型)

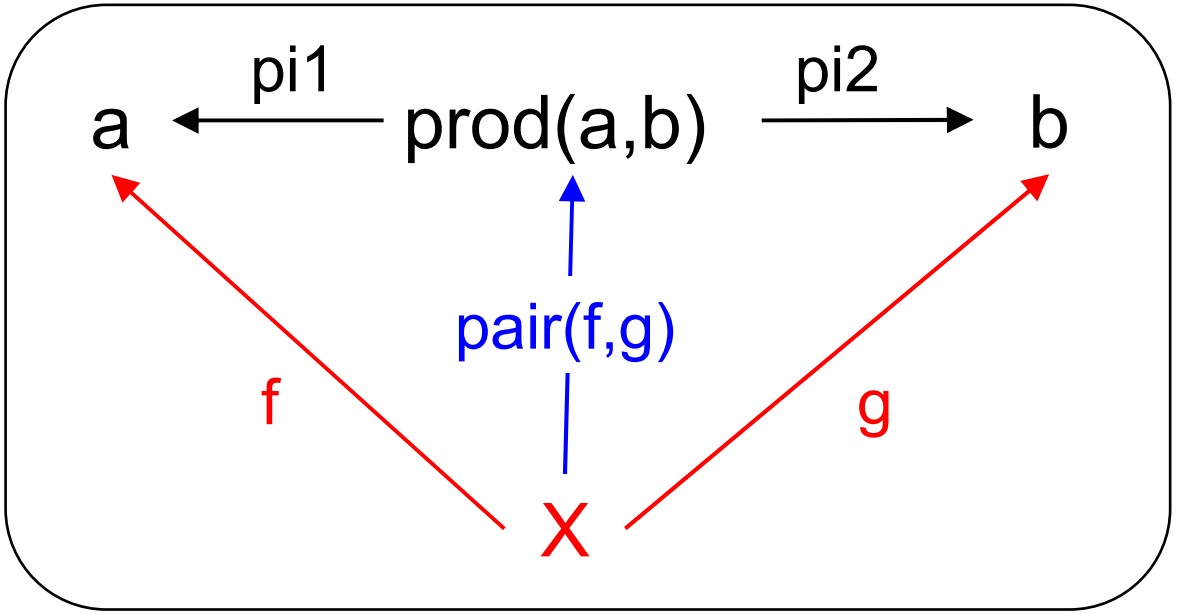
```
right object prod(a,b) with pair is
  pi1: prod → a
  pi2: prod → b
end object;
```

定義しようとしている
型の引数は
書かない



任意のX, f:X→a, g:X→b に対して、左図を可換にする pair(f,g) :
X→prod(a,b)
が一意に存在

直積 (タプル型)



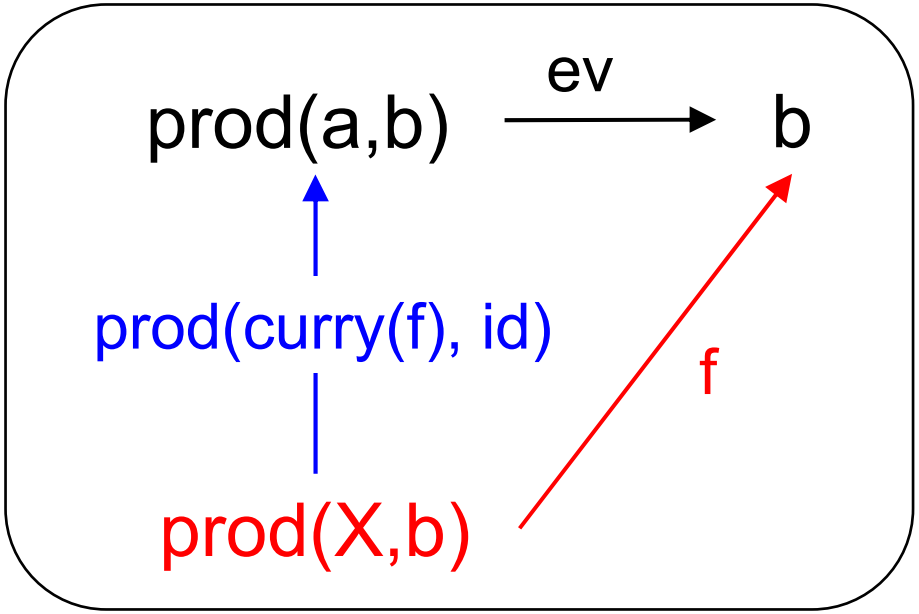
- pi1 . pair(f, g) ⇒ f
- pi2 . pair(f, g) ⇒ g
- prod(f,g) ⇒ pair(f.pi1, g.pi2)

べき対象 (関数型)

(関数型ですら組み込み型ではない)

```
right object exp(a,b) with curry is
  ev: prod(exp, a) → b
end object;
```

evはevalの略だけど、
Lispでのapplyに相当



- ev . prod(curry(f), id)
  ⇒ f
- exp(f, g)
  ⇒
  curry(g . ev . prod(id, f))

簡単な関数定義

- add = ev.prod(pr(curry(pi2), curry(s.ev)), id)
- mult = ev.prod(pr(curry(zero.!), curry(add.pair(ev, pi2))), id)
- fact = pi1.pr(pair(s.zero, zero), pair(mult.pair(s.pi2, pi1), s.pi2))

これは流石に、変態言語(Esoteric Language)っぽいね

CPLのリソース

- 実装:
  - http://www.tom.sfc.keio.ac.jp/~sakai/hiki/?CPL
- 論文:
  - Categorical Programming Language
    http://www.tom.sfc.keio.ac.jp/~hagino/thesis.pdf

おわりに

- 普段使っている foldr や unfoldr の背景には深遠な世界が
- 圏論でのデータ型の特徴づけを用いることで、性質の良い再帰を構造的に書ける

- 無理に圏論プログラミングする必要はないけど、変態プログラミングの一種としても楽しいかも

参考文献

- Categorical Programming Language
  http://www.tom.sfc.keio.ac.jp/~hagino/thesis.pdf
- Categorical programming with inductive and coinductive types
  http://www.cs.ut.ee/~varmo/papers/thesis.pdf
- The Evolution of a Haskell Programmer
  http://www.willamette.edu/~fruehr/haskell/evolution.html