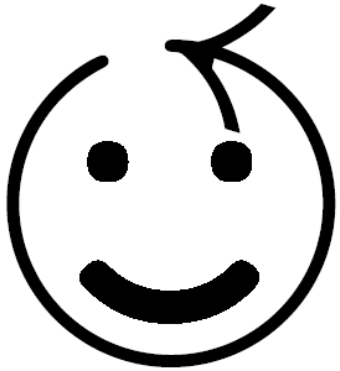
Introduction to Categorical Programming

Haskell Annual Meeting Autumn.jp 2009

酒井 政裕 @ヒビルテ



今日話したいこと

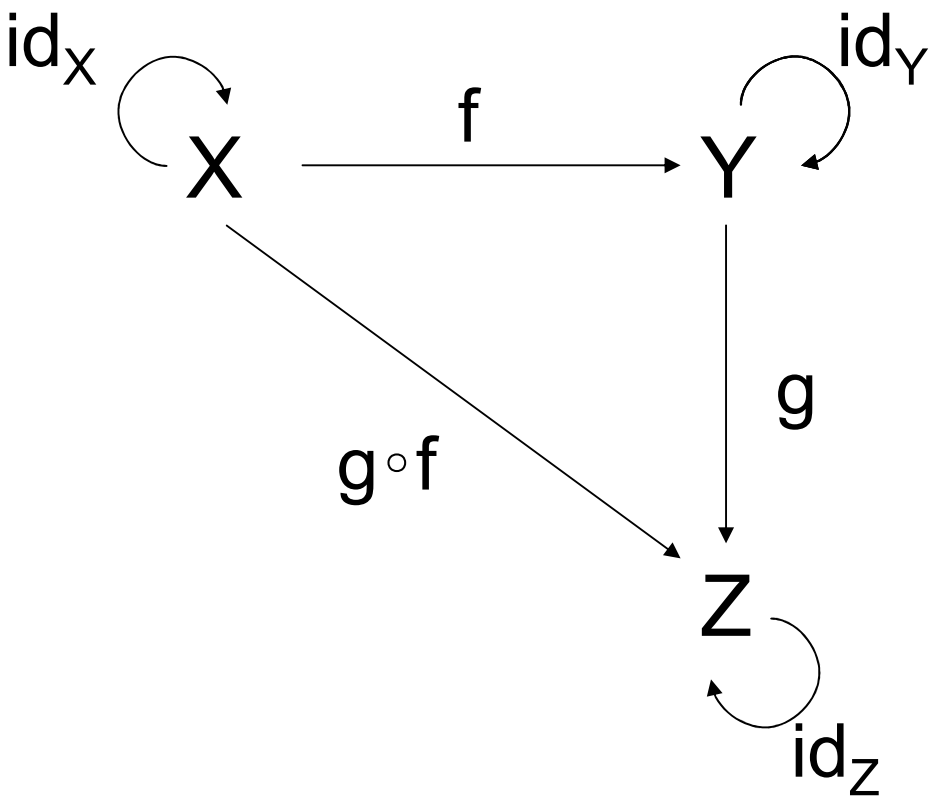
- なぜに圏論?
- Haskell での圏論プログラミング
- The Evolution of a Haskell Programmer
- 圏論プログラミング言語 CPL

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

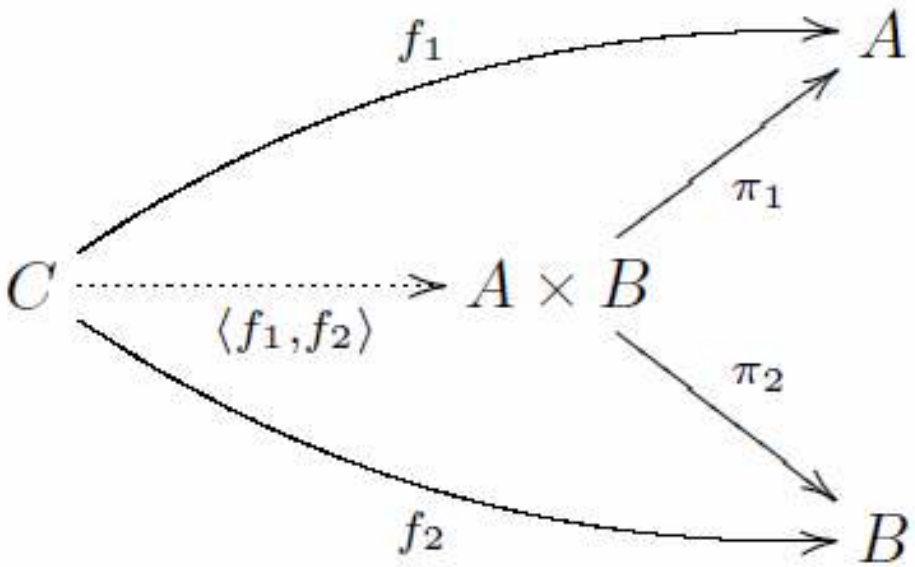


- 対象(Object)
  X, Y, Z, …
- 射(Morphism)
  f, g, id_X, …
- 射の合成: ∘

- 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



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

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?

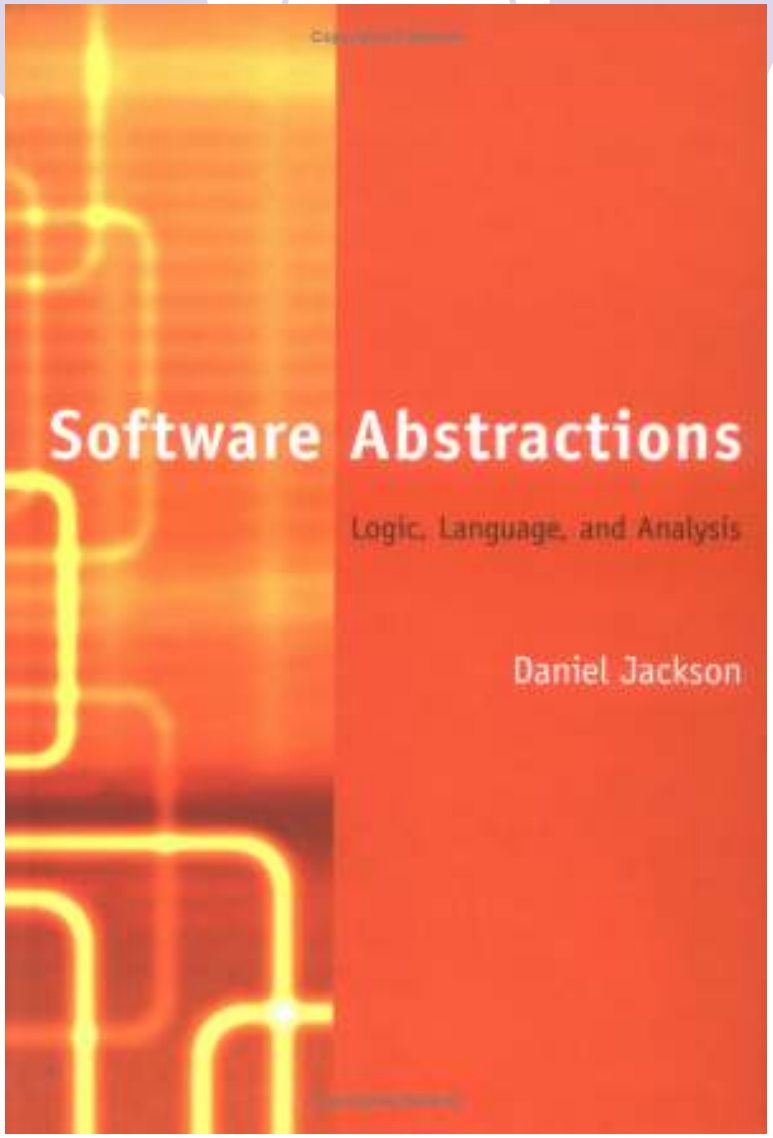
なぜ圏論なのか?

- 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

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!

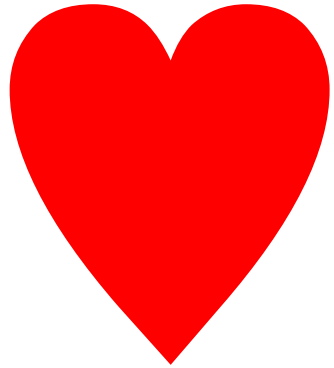
```
foldr :: (a→b→b) → b
       → [a] → b
foldr c n [] = n
foldr c n (x:xs) =
  c x (foldr c n xs)
```

foldrを使った定義例:

- sum = foldr (+) 0
- length =
  foldr (λ _ n → n) 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

リストの代数系

- 代数 = 型とその型の値を作る演算の組
- リストの代数
  - 型と二つの演算 (b, n :: b, c :: a→b→b)
  - 具体例:　([a], [], (:)),　(Int, 0,　λ _ x → x+1)
- 準同型
  - 代数 (b, n, c) から代数 (b', n', c') への準同型 h
  - h :: b→b' で以下を満たすもの
    - h n = n'
    - ∀x::a, y::b.　h (c x y) = c' x (h y)

foldrとリスト型の本質

- 代数 ([a], [], (:)) の特別な性質 (普遍性)
  - 任意の代数 (b,n,c) に対して、([a], [], (:)) から (b,n,c) への、準同型 h が唯一つ存在！
- foldr c n はこの唯一つの準同型
  - foldr :: (a→b→b) → b → ([a]→b)
    foldr c n [] = n
    foldr c n (x : xs) = c x (foldr c n xs)
- 普遍性を持つ (b, n, c) がリスト型
  これを満たせば実装は何でもいい。

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)
- Natの特徴づけ
  - (Nat, Zero, Succ) は
    (x, z::x, s::x→x) の中で
    普遍的なもの
  - 一意な準同型は 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、木など、他の帰納的データ型でも実は同様
  - F-始代数と catamorphism
- foldrの双対
  - 無限リストとunfoldr
  - その一般化: F-終余代数と anamorphism
- より複雑な関数を表現する方法
  - Paramorphism, Histomorphism, Comonadic Iteration
- ……というようなことを、ちゃんと話すつもりが、時間切れ。ここからが面白い話なのにごめんなさい

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
- 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

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)
```

```
-- two versions of
    factorial
…
fac' :: Nat → Nat
fac' = primrec one (mult .
    Succ)

(zero : one : two :
    three : four : five : _) =
    iterate Succ Zero
```

iterの強化版:
sの引数にnが引数に追加

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)

- 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ましょう

他のやつ

- 準備が間に合わなかったので省略 m(_ _)m
- でも簡単に概要だけ……

- Origamist Haskell programmer
- Cartesianally-inclined Haskell programmer
  - リストに関する Hylomorphism
- Ph.D. Haskell programmer
  - Polytypic な Hylomorphism
- Post-doc Haskell programmer
  - Polytypic な Paramosphirm, Zygomorphism

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)
- 圏論に基づいたデータ型定義を持つプログラミング言語
- 実装: http://www.tom.sfc.keio.ac.jp/~sakai/hiki/?CPL

終対象 (terminal object)

- 終対象 (terminal object) 1
  - right object 1 with !
    end object;
- いわゆるユニット型
- 任意の型からの関数がただ一つ存在する型
- Xからの1への唯一の関数を！: X→1 と表す

自然数

- left object nat with pr is
  zero: 1 -> nat
  succ: nat -> nat
  end object;

- 任意の型X と z : 1→X, s : X→X に対して、
  以下を満たす
  pr(z,s) : nat→X が唯一つ存在
  - pr(z, s) . zero = z
  - pr(z, s) . succ = s . pr(z, s)
- 左から右の書き換え規則とみなす

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

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

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

-- 直和 (Either)
left object coprod(a,b) with
    case is
  in1: a -> coprod
  in2: b -> coprod
end object;
```

おわりに

- これから結論を考えるよ