Haskeller のための圏論

Keichi

2011 年 7 月 2 日

1 はじめに

この文書は Haskell を勉強している筆者が、寄り道して勉強した圏論について、学習した事項をまとめたものです。主な内容は、HaskellWiki の Category theory のページ [1] の翻訳です。この文書は、Wiki の内容に Haskell で書いたサンプルコードや、図、用語の解説などを多少書き加えたものです。

Haskell は、圏論をプログラミング言語に応用してつくられた言語です。ただし圏論を知らなくても、Haskell の概念を理解することはできますし、コード自体は書けます。ただしファンクタやモナドなどの概念を考えた側は、圏論から着想を得ているわけで、圏論を学ぶことでより深く Haskell を理解できると私は思っています。

2 圏

2.1 定義

ある圏 (Category)C は、2 つの集合の組からなる:

- $Ob(C)$、C の「対象」(Object) の集合
- $Ar(C)$、C の「射」(Morphism) の集合

$Ar(C)$ に含まれる射 f は、$Ob(C)$ の要素であるドメイン (Domain) (もしくはソース (Source))$domf$ と、コドメイン (Codomain)(もしくはターゲット (Target)) $codf$ を持つ。$f : A \to B$ と書けば、f のドメインが A であり、コドメインが B であることを意味する。

$$dom(f) \xrightarrow{f} cod(f)$$

図 1 ドメインとコドメイン

また合成という操作が存在し、これは $\circ$ と表記される。例えば、射 $g \circ f$ が定義されるとき、f のコドメインは g のドメインと等しく、$g \circ f$ のドメインとコドメインはそれぞれ f のドメインと g のコドメインとなる。つまり、射 f と g が存在し、$f : A \to B$ かつ $g : B \to C$ ならば、$g \circ f = A \to C$ ということである。また、ある対象 A について、$id_A : A \to A$ という射が定義される。これは恒等射と呼ばれ、id とだけ表記されるこ

とも多い。

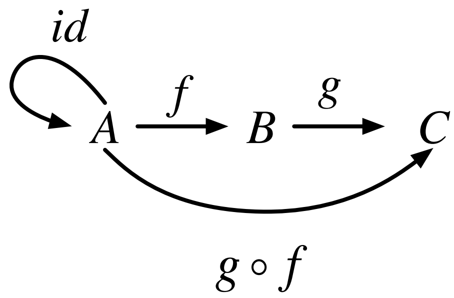

図 2　恒等射と合成

また、圏 C において Homset 集合を $hom_C(A,B) = \{f \mid f \in Ar(C), f : A \to B\}$ と定義する。これはつまり、圏 C において対象 A から B への射の集合である。

2.2　公理

また、C が圏であるためには、以下の条件が満たされている必要がある:

- $f : A \to B$ なら $f \circ id_A = id_b \circ f = f$ (恒等射は左単位元、右単位元である)
- $f : A \to B$ かつ $g : B \to C$ か $h : B \to D$ なら、$h \circ (g \circ f) = (h \circ g) \circ f$(射は結合律を満たす)

2.3　圏の例

- Set, 集合を対象とし、集合間の写像を射とする
- Mon, モノイドを対象とし、モノイドの射を射とする (モノイドは対象が 1 つだけの圏ともとらえれる)
- Grp, 群を対象とし、群の準同型写像を射とする
- Hask, Haskell の型を対象とし、関数を射とする圏
- Cat, 圏を対象とし、関手を射とする圏の圏
- 関数を対象とし、データの流れを射とする圏（データフローダイアグラム）

2.4　Haskell での例

Haskell の型と関数からなる圏 Hask を考えると、圏 Hask の対象 $Ob(Hask)$ は全ての Haskell の型の集合、

$$\{Int, Bool, Float, String, \ldots\}$$

である。また、$Ar(Hask)$ は全ての Haskell の関数の集合、

$$\{(+), length, even, \ldots\}$$

となる。圏 Hask において射の合成 $\circ$ は、関数合成演算子

```
(.)::(b−>c)−>(a−>b)−>(a−>c)
```

であり、恒等射は恒等関数

```
id::a->a
```

となる。単位元律および結合律が成り立つのは明らかである。(seq などの例外はあるが)

3 関手

3.1 定義

C, D を圏とすると、関手 (Functor)$F : C \to D$ は関数 $F_{objects} : Ob(C) \to Ob(D)$ と関数 F_{arrows} : $Ar(C) \to Ar(D)$ の組である。前者は対象関数とよばれ、後者は射関数とよばれる。

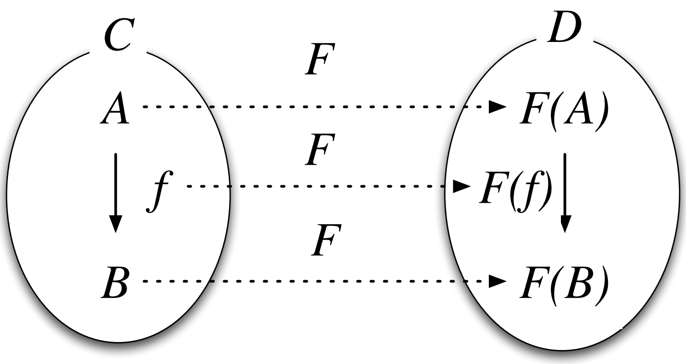

図 3　関手 F

3.2 公理

- 圏 C において $f : A \to B$ が存在すれば、圏 D において $F(f) : F(A) \to F(B)$ である
- 圏 C において $g : B \to C$、$f : A \to B$ が存在すれば、$F(f) \circ F(g) = F(f \circ g)$ が成り立つ
- 圏 C の全ての対象について、$id_{F(A)} = F(id_A)$

3.3 Haskell における関手

厳密には、Haskell 圏における関手は、Haskell の型と関数に対するあらゆる操作うち、上記の公理を満たすものである。しかし、Haskell はプログラミング言語であるので、実際 Haskell で扱うのは、対象関数と射関数の両方が Haskell で書ける関手だけである。具体的には、Haskell における関手の対象関数は、多くの場合データコンストラクタであり、射関数は多相関数 fmap となる。これは型クラス Functor で宣言されている:

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

Haskell において、List や Maybe、Tree などは全て Functor クラスのインスタンスであり、Functor クラスのインスタンスは以下の条件

```
fmap id = id
fmap f . fmap g = fmap (f . g)
```

を満たすべきであるとされている。これは上記の公理に相当する。

具体的な例として、Hask 圏から Maybe 圏への関手 Maybe を考える。関手 Maybe の対象関数は Just であり、射関数は fmap となる。

```haskell
--元の圏の上での関数f
f :: Int -> Int
f x = x * x

--fをMaybeの圏の射に写像する
g :: Maybe Int -> Maybe Int
g = fmap f

--元の圏の上での値
a = 3
--aをMaybeの圏の対象に写像する
b = Just a

main = do
    --元の圏の上で対象に射を適用
    print $ f a
    --Maybeの圏の上で対象に射を適用
    print $ g b
```

4 自然変換

4.1 定義

C, D を圏とし、$\Phi, \Psi : C \to D$ を関手、$X, Y \in Ob(C)$ とし、$f \in hom_C(X, Y)$ とする。

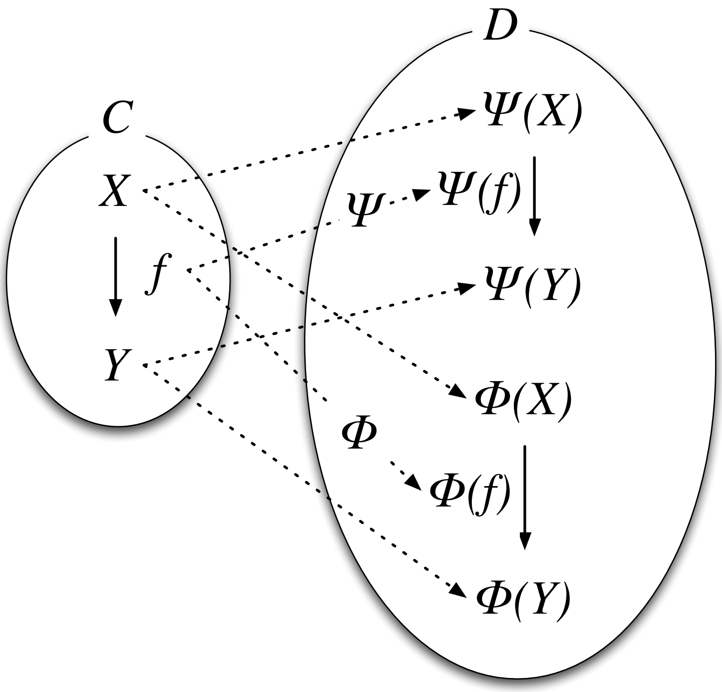

図 4　関手 Φ と Ψ

自然変換 (Natural Transoformation)$\eta : \Phi \to \Psi$ は、C の各対象を、以下の条件を満たす D の射と対応づける関数である。

- $\forall A \in Ob(C) \to \eta_A \in hom_D(\Phi(A), \Psi(A))$
- $\eta_Y \circ \Phi(f) = \Psi(f) \circ \eta_X$

ここで、η_A を A における η のコンポーネントとよぶ。よって、D **において**図 5 の関係が成り立つ。

$$
\begin{tikzcd}
\Phi(X) \arrow[r, "\Phi(f)"] \arrow[d, "\eta_X"] & \Phi(Y) \arrow[d, "\eta_Y"] \\
\Psi(X) \arrow[r, "\Psi(f)"] & \Phi(Y)
\end{tikzcd}
$$

図 5　自然変換 η

4.2　Haskell による例

これは、図 5 関係の具体的な例 Haskell で書いたものである。

```haskell
--この関数は Data.Maybe で定義されています。
maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]

main = do
    print $ fmap even $ maybeToList $ Just 5
    print $ maybeToList $ fmap even $ Just 5
```

この例と上の図 5 の対応関係を考えてみる。ファンクタ Ψ は List、ファンクタ Φ は Maybe と対応する。また、自然変換 η_X、η_Y は maybeToList に相当する。（ここで maybeToList は多相関数であるため、η_X と η_Y は 1 つの多相関数 maybeToList に集約される）また、f は even であり、これが fmap によってそれぞれ $\Psi(f)$ と $\Phi(f)$ にリフトされる。fmap は多相であるため、Φ、Ψ 両方のファンクタの射部分に対応する。

5　Kleisli トリプル

5.1　定義

Haskell で一般的に「モナド」と呼ばれているものは実は圏論のモナドとは異なっていって、直接的に対応するのは圏論で Kleisli トリプルとよばれている。

圏 C 上の Kleisli トリプル (Kleisli-Triple) とは

- 関数 $T : Ob(C) \to Ob(C)$
- 射 $\eta_A :\to T(A)$(各 $A \in Ob(C)$)
- $(-)^*$: 射 $f^* : T(A) \to T(B)$ を各 $f : A \to T(B) \in Ar(C)$ についてつくる演算子

からなる 3 つ組 (トリプル)$(T, \eta, (-)^*)$ である。

5.2　公理

また、Kleisli トリプルは以下の条件を満たす:

- $\eta_A^* = id_{T(A)}$
- $f : A \to T(B)$ なら、$f^* \circ \eta_A = f$
- $f : A \to T(B)$ なら、$g^* \circ f^* = (g^* \circ f)^*$

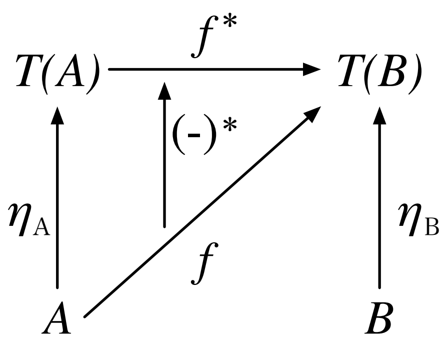

図 6　Kleisli トリプル

5.3　Haskell での例

Haskell のモナドは Kleisli トリプルと対応する。型クラス Monad の定義は

```
class Monad t where
    (>>=) :: t a -> (a -> t b) -> t b
    return :: a -> t a
```

であり、これはそれぞれ、

- Monad のインスタンスのデータコンストラクタ (Maybe なら Just) が関数 T

- return が η
- (=<<) が $(-)^*$

というように圏論の Kleisli トリプルと対応している。また、上記の Kleisli トリプルの満たすべき条件は、以下の Haskell のモナド則にそれぞれ対応する。

```haskell
(return x) >>= f == f x
m >>= return == m
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

参考文献

[1] http://www.haskell.org/haskellwiki/Category_theory