関数型言語におけるデータ構造への圏論の応用

2022 年 10 月 9 日

目次

1 Haskell と抽象構造

オブジェクト指向言語におけるデザインパターンとは、プログラムの一部を再利用、テストを容易にするための設計手法である。これらは継承やら実装やらのオブジェクト指向言語特有の操作によって抽象化されていて、さらにそれらの操作で閉じている。

関数型プログラミング、特に Haskell ではリストの操作や入出力などの副作用を伴う操作を、抽象的で汎用性があり極力関数型言語の操作で構成される設計によって与えている。これらは言うなればデザインパターンによって言語機能の拡張しているとみなすことができ、こういった設計もここではデザインパターンと呼ぶことにする。

1.1 関数型

Haskell にも一般のプログラミングと同じようにデータ型が存在し、整数値を持つ `Int` 型や `True` と `False` を持つ `Bool` 型などがある。また整数値を受け取って二倍にして返す、つまり $double(x) = 2x$ なる関数 `double` は `Int` から `Int` への関数であり、haskell ではこのように記述する。

```
double :: Int -> Int
double x = 2*x
-- >>> double 4
-- 8
```

`double` ある型 `a` からある型 `b` への関数は、関数の型 `a -> b` に含まれる。すなわち、関数そのものも値とみなすということである。それによって、関数型から関数型への関数なども定義できるようになる。その一例を示そう。

関数 `f :: Int -> Int` を、自身を合成した関数 `f.f :: Int -> Int` へと写す関数 `mul :: (Int -> Int) -> (Int -> Int)` は次のように記述する。

```
mul :: (Int -> Int) -> (Int -> Int)
mul f = f.f
-- >>> (mul double) (5)
-- 20
```

後に説明するが、`.` という記号は前後の関数を合成する操作であり、`f.f` は $f \circ f$ を意味する。すなわちこの計算は

$$(mul(double))(5) = (double \circ double)(5) = double(double(5)) = 20$$

のように行われる。このような関数を引数、戻り値にするような関数を高階関数と呼ぶ。

二つの整数を足し合わせる操作は $add(x, y) = x + y$ のように定義できる。この関数は引数を二つ持つ多変数関数であり、これは Haskell では

```
add :: Int -> Int -> Int
add x y = x + y
-- >>> add 5 6
```

```
    -- 11
```

のように型を与えて定義することができる。関数の型 `Int -> Int -> Int` は左から二つは引数の型で、一番右が返り値の型となっている。なぜこのような表記をするかというと、`Int -> Int -> Int` は `Int -> (Int -> Int)` の省略であり、$add(x, y)$ を $(add(x))(y)$ とみなしているためである。ゆえに `add(x) :: Int -> Int` は引数に `x` を足し合わせる関数となっている。

1.2 多相型と多相関数

関数型 `a -> b` は単一の型ではなく、任意の型 `a` , `b` に対して個別に定義できるのであった。このように既存の型から生成できる型を多相型という。また、多相型から多相型への型を考えることもできる。例えば高階関数と多変数関数を用いれば、関数を合成する操作に次のような型を与えることができる。

```
    apply :: (Int -> Int) -> Int -> Int
    apply f x = f(x)
    -- >>> apply double 10
    -- 20
```

しかし、引数の関数は `Int` か `Int` への関数に制限されているが、実際は関数がどのような型であっても、適用の操作は行えるはずである。そこで、 `apply` 関数の引数の型を `a` 、返り値の型を `b` に置き換える。

```
    apply :: (a -> b) -> a -> b
    apply f x = f(x)
    -- >>> apply double 10
    -- 20
    -- >>> apply not False
    -- True
```

例のために真理値を反転する関数 `not :: Bool -> Bool` を使用している。

このように関数の引数の型を任意の型 `a` , `b` に置き換えることで、様々な型の値に対応した関数を定義することができ、このような関数を多相関数と呼ぶ。

このような多相関数は基本的に既存の多相関数の組み合わせによって定義される。この例であれば、`apply` の定義に用いた `f(x)` はすでに haskell 側によって多相関数 `($) :: (a -> b) -> a -> b` としてすでに定義されているため、定義することができた。

値の適用が関数として与えられているように、関数の合成も関数で行うことができる。つまり関数 `double` の定義に用いた `.` が関数であり、

`(.) :: (b -> c) -> (a -> b) -> a -> c`

という型が与えられているということである。

このように関数型プログラミングでの処理の記述は変数の仲介を除けば、関数に関数を合成する操作や、関数に値を適用する操作によって行われる。

1.3 基本的なデータ型

任意の型 `a` , `b` に対して多相型である直積型 `(a,b)` は以下の関数で構成される。

```
(,) :: a -> b -> (a, b)
fst :: (a, b) -> a
snd :: (a, b) -> b

-- >>> (,) 4 5
-- (4,5)
-- >>> fst (4,5)
-- 4
-- >>> snd (4,5)
-- 5
```

実行例を見て分かるように、`(a,b)` の値はある型 `a` と `b` の値をそれぞれ持つ。また、型 `a` , `b` の値 `x` , `y` に対して、

$$\texttt{fst (x, y)} = \texttt{x}, \quad \texttt{snd (x, y)} = \texttt{y}$$

が成り立つ。

直積型は二つの型の値を持つが、ゼロ個の型の値を持つ型も存在する。この型を Unit 型 `()` といい、ただ一つの値 `()` を持つ。

任意の型 `a` , `b` に対して多相型である直和型 `Either a b` は以下の関数で構成される。

```
Right :: b -> Either a b
Left :: a -> Either a b
either :: (a -> c) -> (b -> c) -> Either a b -> c

isZero :: Int -> Bool
isZero x = x==0

-- >>> either isZero not (Left 0)
-- True
-- >>> either isZero not (Right True)
-- False
```

`isZero` 関数の `x==0` は両辺が等しいか真理値を返す関数 `(==) :: Int -> Int -> Bool` であり、定義から分かるように `Either a b` は `a` か `b` の値のどちらか一方を持つ。

`either` 関数についてこの例では、`Int -> Bool` と `Bool -> Bool` を受け取っているため、次の引数の型は `Either Int Bool` を受け取る。もし `Either Int Bool` の値が `Int` であれば、`Int -> Bool` 型の `isZero` 関数に適用し、`Bool` であれば `Bool -> Bool` 型の `not` 関数に適用する、という関数である。

また任意の型 `x` とその値 `y` 、

```
f :: a -> x
g :: b -> x
```

に対して、

$$\texttt{either f g (Left x)} = \texttt{f x}$$
$$\texttt{either f g (Right x)} = \texttt{g x}$$

が成り立つ。

一般に直和を用いて新しい型を定義する場合、`Either`型を用いて定義するのではなく、haskellにおける多相型を定義する構文によって行われることが多い。また`either`関数もパターンマッチと呼ばれる構文によって行われる。

```
data EitherIB = Value Int | Judge Bool
calc :: EitherIB -> Bool
calc (Value a) = isZero a
calc (Judge a) = not a
-- >>> :t Value
-- Value :: Int -> EitherIB
-- >>> :t Judge
-- Judge :: Bool -> EitherIB
```

この例では、`EitherIB`が`Either Int Bool`に該当し、`Value`が`Left`、`Judge`が`Right`に対応する。

1.4 デザインパターンの例

最後にこれらを応用したMaybe型を見る。

```
data Maybe a  =  Nothing | Just a
```

`Maybe a`は`Either`型を用いると、`Either () a`と書ける。ここでの`Nothing`は`Unit`型の値を与える関数とみなせるが、値はただ一つであるため単に定数と見て良い。すなわち`Maybe a`はただ一つの値`Nothing`と任意の型`a`の値のどちらかの値を持つ型である。

この`Maybe`の想定している用途として、以下の例を扱う。

```
div10 :: Float -> Float
div10 = (10 /)
-- >>> div10 4
-- 2.5
-- >>> div10 0
-- Infinity
```

`div10`は10を与えられた数で割る関数であるが、結果のように零除算については Infinity という値が与えられる。これを素直に計算の失敗と見なし、型によって計算結果が正しく出力されないことを示したい。そこで、

```
div10' :: Float -> Maybe Float
div10' x = if x == 0 then Nothing else Just $ div10 x

-- >>> div10' 4
-- Just 2.5
-- >>> div10' 0
-- Nothing
```

という関数を定義する。これは零除算が発生する場合 `Nothing` を返し、そうでない場合は `Just` によって計算結果を `Maybe` に包んで返している。

これによって、 `Maybe Float` の値を用いて計算を行うときは、必ず `Nothing` と `Just` の場合分けが必要になり、他の言語でいう Null チェックが必要なことを型レベルで示していることになる。

またここではこれ以上語らないが、 `Maybe` 型の挙動はモナドと呼ばれるデザインパターンによって一般化される。これによって特に `a -> Maybe b` 型の関数と `b -> Maybe c` 型の関数の自然な合成を与えることができる。

2 圏論

関数型言語では関数という言葉が使われるように、型を属する値の集合、関数を値の集合から値の集合への写像とみなすことができる。集合を写像の視点から観察する場合、圏論によって一般化することが有効である場合が多い。そのため以降は型と関数の議論を圏論によってモデル化することを考える。

2.1 型と関数の圏

型を対象、関数を射とする圏 $\mathbb{H}$ を定義する。圏の定義は [CTI]2.0.1 を参照 Haskell には二つの関数が等しいことを判定する一般的な操作は無い。そのため関数の等号を改めて定義する。

定義 2.1.1（関数の外延性） 関数 `f :: a -> b` 、 `g :: a -> b` と型 `a` の任意の値 `x` において

$$\texttt{f x} = \texttt{g x} \iff \texttt{f} = \texttt{g}$$

定義 2.1.2（型と関数の圏）

対象 すべての型の集合を対象集合 $\mathrm{Obj}(\mathbb{H})$ とする。また各型を対象とみなす場合 $A, B, C \cdots$ と表記し、型とみなす場合は `a` , `b` , `c` $\cdots$ とする。また各型の値を圏論の文脈では $a, b, c \cdots$ と表記することにする。

射 対象 A から対象 B への射集合 $\mathbb{H}(A, B)$ を集合と見なした関数型 `a -> b` とする。また各射は $f, g, h \cdots$ と表記する。

射の合成 射の合成を行う写像 $\circ : \mathbb{H}(B, C) \times \mathbb{H}(A, B) \to \mathbb{H}(A, C)$ を任意の射 $f : A \to B,\ g : B \to C$ に対して `f.g` によって定義する。

恒等射の存在　任意の型 `a` に対して恒等関数 `id :: a -> a` は、 `id x = x` と定義できる。この恒等関数を恒等射とし、$id_A : A \to A$ と表記する。

結合律　関数の外延性より、集合の圏における結合律と同様の手法で証明できる。[CTI]2.0.3 を参照

単位元律　結合律と同様に [CTI]2.0.3 を参照

次に型の直積と直和の普遍性との関係を見る。

命題 2.1.3　圏 $\mathbb{H}$ は積を持つ

証明 2.1.4（$\mathbb{H}$ **の積**）　(`(a, b)` , `fst` , `snd`) は型 `a` , `b` における積であることを示せば良い。任意の型 `x` 、任意の関数

```
f :: x -> a
g :: x -> b
```

に対して、

```
h :: x -> (a, b)
h x = (f x, g x)
```

とする。すると明らかに

$$\texttt{fst . h \$ x} = \texttt{f x}$$
$$\texttt{snd . h \$ x} = \texttt{g x}$$

であり、ある関数 `i :: x -> (a, b)` によって与えられた値 `i x :: (a, b)` が

$$\texttt{fst (i x)} = \texttt{f x}$$
$$\texttt{snd (i x)} = \texttt{g x}$$

を満たす時、 `fst` , `snd` の性質から `i x = (f x, g x)` が成り立つ。よって 2.1.1 の関数の外延性より `i` = `h` である。これにより射の対 `h` の一意に存在することを示せた。 よって (`(a, b)` , `fst` , `snd`) は積であり、任意の型 `a` , `b` に対して存在するから $\mathbb{H}$ は積を持つ。

命題 2.1.5（$\mathbb{H}$ **の終対象**）　圏 $\mathbb{H}$ は終対象を持つ。

証明 2.1.6　ある型 `x` において

```
f :: x -> ()
f x = ()
```

なる写像を考える。またこのような関数は任意の `x` について考えることができる。

```
g :: x -> ()
g x = ()
```

なる写像を考えると、`f x` = `()` = `g x` が成り立ち、2.1.1 の関数の外延性より `f` = `g` が成り立つ。`f :: x -> ()` なる関数が一意に存在するから `()` は $\mathbb{H}$ における終対象であり、$\mathbb{H}$ は終対象を持つ。

定義 2.1.7（余積の定義） ある対象 A, B に対して $(A+B,\ \iota_A,\ \iota_B)$ が余積であるとは、$\iota_A : A \to A+B,\ \iota_B : B \to A+B$ であり、任意の対象 X と任意の射 $f : A \to X,\ g : B \to X$ に対して

$$[f, g] \circ \iota_A = f,\ [f, g] \circ \iota_B = g$$

であるような $[f, g] : A + B \to X$ が一意に存在するときである。

命題 2.1.8（$\mathbb{H}$ の余積）

証明 2.1.9 $(Eitherab,\ Left,\ Right)$ が終対象

2.2 関手と自然変換

2.3 関数型とべき随伴