第2章　型クラス

関数などがいろいろな型の引数を許し、しかも _____ことを、特に _____（ad-hoc）ポリモルフィズムという。オブジェクト指向言語の _____はアドホック多相の一種である。オブジェクト指向言語では、単にポリモルフィズムという言葉で動的束縛を指すことがある。

また、動的束縛とよく似ている概念として _____（多重定義, overloading）がある。例えばCの「+」オペレータは int（整数）型にも double（倍精度浮動小数点数）型にも適用できる。しかし最終的には、適用される型によって全く異なる機械語に翻訳される。動的束縛と異なる点は、多重定義はコンパイル（型チェック）時に解決されてしまう、という点である。実行時にオペランドの型に応じて処理を振り分けるようなことは行なわない。多重定義もアドホック多相の実現方法の一つである。

従来、アドホック多相と ML などで採用されている型推論は相性が悪いものであった。Haskell は型推論の上に _____（type class）というアドホック多相を可能にするための仕組みを持っている。以下では Haskell の型クラスを、詳しく説明する。

2.1　Haskell の型クラス

型クラスの説明のために、例として、Eq という型クラスを取り上げる。

Haskell でも C と同じように「==」（等号）オペレータは Integer（整数）型にも Double（倍精度浮動小数点数）型にも適用できる。しかし、Haskell は多相を許す言語であるので、例えば、

```
member x []     = False;
member x (y:ys) = x==y || member x ys;

subset xs ys    = all (\ x -> member x ys) xs;
```

という関数 member や subset を定義すると、member 5 [1, 4, 7] のように [Integer]（整数のリスト）型の引数にも、member "Kagawa" ["Tokushima", "Ehime", "Kochi"] のように [String]（文字列のリスト）型の引数にも適用できる。しかし、member (\ x -> x-1) [\ x -> x+1, \ x -> x+2] のように関数のリストに適用することはできない。

それでは、member や subset はどのような型を持ち、どのように実装されているのであろうか? Scheme のように動的に型付けされる言語では、各データオブジェクトが型の情報をつねに保持しているので、実行時に型に応じて適切な関数を選択することができる。しかし、Haskell のようにコンパイル時に型チェックを行なう言語では、実行時にはデータは型の情報を保持していないのが普通である。

Haskell では、これらの関数の型は自動的に推論される（型推論 – ただし、その方法の詳細については本稿で取り扱う範囲を超える）。型推論の結果だけを示すと、member と subset は次のような型を持っている。

```
member :: _____________________________;
subset :: _____________________________;
```

ここで “Eq a =>” という部分は、 a という型変数が Eq という型の集まり（_____）に属していなければいけない、という型に関する制約（type constraint）を表す。Eq という型クラスは、==（等号）が定義されているような型の集まりのことである。型クラスは通常のオブジェクト指向言語でのクラス・インスタンスという言葉とは意味が異なるので注意する。通常のオブジェクト指向言語ではクラスは _____（つまり型）であるのに対し、Haskell の型クラスは _____である。（Java のインタフェースの概念に似ている。）

2.2 クラス宣言とインスタンス宣言

一般に型クラスとは、_____のことである。型クラスの定義には **class** というキーワードを用いる。例えば、Eq クラスの定義は Haskell では次のように書く。

```
class Eq a where {
  (==), (/=) :: a -> a -> Bool;
  a /= b = not (a == b)
};
```

これが、「型 a が型クラス Eq に属するためには、a-> a -> Bool という型を持つ 2 つの関数 (==), (/=) を持たなければいけない」という意味になる。一方、Integer 型が Eq クラスに属する（Integer が Eq の _____である）ことを宣言するためには、**instance** というキーワードを用いる。

```
instance Eq Integer where {
  (==)   = primEqInteger;
};
```

ここで primEqInteger はInteger -> Integer -> Bool という型を持つプリミティブ関数である。この宣言は、Integer 型に対する==オペレータの実際の実装は primEqInteger であることを示している。

同様に Double に対しても次のようにインスタンス宣言できる。

```
instance Eq Double where {
  (==)   = primEqDouble;
};
```

ほとんどの型（例えばリストや組）は Eq クラスに属するが、関数型については、一般に 2 つの関数が等価であるかどうかを判定することは原理的に不可能なので、Eq クラスに属さない。

Integer や Double は組込みのデータ型だが、ユーザ定義のデータ型も型クラスのインスタンスになることができる。例えば、Tree の場合、次のように宣言する。

```
instance Eq a => Eq (Tree a) where {
  Empty           == Empty           = True;
  Branch l1 n1 r1 == Branch l2 n2 r2 = l1 == l2 && n1 == n2 && r1 == r2;
  _               == _               = False
};
```

“Eq a =>” の部分は Tree a に等号を定義するためには、要素の型である a に等号が定義されていなければいけないという制約を表す。

2.3 Dictionary-Passing Style 変換

ここからは、Haskell が型クラスをどのように実装しているかを説明する。（ただし、このように実装しなければいけないと、Haskell の仕様に定められているわけではない。あくまでも良く使われる実装テクニックの一例である。）

クラス宣言・インスタンス宣言や制約された型を持つ関数は、コンパイル時にそれらを用いない普通の関数やデータの定義に書き換えられる。

ず、クラス宣言は次のような型の宣言とアクセサの定義に翻訳される。

```
type Eq' a = ____________________________________________;

eq' :: Eq' a -> (a -> a -> Bool);
eq' = ___________________;       -- (==) に対応する

ne' :: Eq' a -> (a -> a -> Bool);
ne' = ___________________;       -- (/=) に対応する
```

この Eq' a 型のようなオブジェクトは一般に _____（method dictionary）と呼ばれる。

インスタンス宣言は具体的な型を持つ辞書オブジェクトの定義に翻訳される。

```
eqIntegerDic :: Eq' Integer;
eqIntegerDic = (primEqInteger, \ a b -> not (primEqInteger a b));

eqDoubleDic :: Eq' Double;
eqDoubleDic = (primEqDouble, \ a b -> not (primEqDouble a b));
```

そして型制約（ ... =>）をもつ関数の定義は、コンパイル時に次のように辞書オブジェクトを追加の引数とする関数の定義に書き換えられている。つまり高階関数になる。

```
member' :: ____________________________________;
member' d x []     = False;
member' d x (y:ys) = eq' d x y || member' d x ys;

subset' :: ____________________________________________;
subset' d xs ys = all (\ x -> member' d x ys) xs;
```

これらの関数の呼び出しは次のように型に応じて具体的な辞書オブジェクトを渡される形に書き換えられる。

```
member 4 [2, 5, 8]            ↦  member' ________________ 4 [2, 5,8]
subset [0.0, 1.0] [1.0, 2.0]  ↦  subset' _______________ [0.0, 1.0] [1.0, 2.0]
```

このような書き換え（Dictionary-Passing Style 変換）は Haskell ではコンパイル中の型推論時に自動的に行なわれる。つまり、アドホック多相の実行時のコストは、辞書オブジェクトの中から適切なオフセットを用いて関数を取り出し、それを起動するだけになる[1]。動的に（つまり実行時に）メソッドを探しているように見えて、実際には静的に（コンパイル時に）ほとんどの必要な処理が済んでいる。

問 2.3.1 次のように定義された関数 lookup*:*

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

lookup :: Eq a => a -> [(a, b)] -> Maybe b;
lookup x ((n,v):rest) = if n==x then Just v else lookup x rest;
lookup x []           = Nothing;
```

の *DPS* 変換後の形 lookupD*:*

```
lookupD :: Eq' a -> a -> [(a, b)] -> Maybe b
```

を定義せよ。

2.4 その他の型クラス

Eq の他に実用上重要な型クラスとして、Ord, Show, Num などがある。

```
class Eq a => Ord a where {
   (<), (<=), (>=), (>) :: a -> a -> Bool;
   max, min             :: a -> a -> a ;
   ...
};

class Show a where {
   show                 :: a -> String;
   ...
};

class (Eq a, Show a) => Num a where {
   (+), (-), (*)        :: a -> a -> a;
   fromInteger          :: Integer -> a;
   ...
};

class Num a => Fractional a where {
   (/)                  :: a -> a -> a;
   ...
};
```

[1]ただし高階関数になるので、最適化が難しくなってしまうことは充分ありえる。

主要でないメソッドは省略している。Ord は不等号、Show は文字列への変換、Num と Fractional は四則演算のメソッドを定義している型クラスである。

クラス宣言の =>の左側にあるクラスは、スーパークラスと呼ばれる。例えば、Ord クラスのインスタンスになる型は、必ず Eq クラスのインスタンスでなければならない。

Show, Eq, Ord クラスなどに対するインスタンス宣言は、ほとんどデータ型で必要になるので、データ型の宣言が deriving というキーワードを持っていれば、Haskell の処理系がこれらのインスタンス宣言を自動的に生成してくれることになっている。例えば次のように書く。

```
data Tree a = Branch (Tree a) a (Tree a) | Empty deriving (Eq, Ord, Show)
```

問 2.4.1 組込みのリスト型と等価なデータ型

```
data MyList a = MyCons a (MyList a) | MyNil
```

を、deriving を用いずに、Eq クラスと Ord クラスのインスタンスとして宣言せよ。Ord クラスのメソッドにはいわゆる辞書式の順序を用いよ。

（==と<=だけ定義すれば、他のメソッドの定義は自動的に生成される。）

この章の参考文献

[1] Philip Wadler and Stephen Blott 「How to make *ad-hoc* polymorphism less *ad-hoc*」1988 年 10 月
Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 60–73
型クラスのアイデアを最初に紹介した論文である。

[2] Cordelia Hall, Kevin Hammond, Simon Peyton Jones and Philip Wadler
「Type Classes in Haskell」 1996 年
ACM Transactions on Programming Languages and Systems 18 巻 2 号, pp. 109–138
現在の Haskell の型クラスを詳しく説明している。

[3] Mark P. Jones 「Typing Haskell in Haskell」 2000 年 11 月
http://www.cse.ogi.edu/~mpj/thih/
Haskell の（型クラスに関する部分を含む）型推論を Haskell のプログラムとして説明している論文である。