第4章　型クラス

ポリモルフィズム（多相）とは関数（メソッド）などが＿＿＿＿＿＿＿＿＿＿＿＿を許すことをいう。さらに、関数などがいろいろな型の引数を許し、しかも＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿ことを、アドホック（ad-hoc – その場限りの、という意味）多相という。オブジェクト指向言語の動的束縛（dynamic binding）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の意味まで含むことがある。

動的束縛と似ている概念としてオーバーローディング（多重定義, overloading）がある。多重定義は一つの名前が複数の意味を持つことである。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（つまり静的な束縛である）。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。ただし、適用範囲はコンパイル時に型がわかる場合に限定されてしまう。

Haskell のように型推論と多相型[1]を許す言語では、コンパイル時に得られる型情報では演算子の実装を決定することはできない。例えば、次のような関数:

```
twice x = x + x;
```

を定義した時、`x` の型が整数か実数か決定できないので、+ の実装も決定できない。

そこで、Haskell ではアドホック多相を可能にするために、＿＿＿＿＿＿という仕組みが導入されている。

4.1　オブジェクト指向との関係

型クラスの説明に入る前に、オブジェクト指向言語で使用される概念との関連について触れる。

オブジェクト指向を特徴づけるキーワードとして動的束縛・継承（inheritance）・カプセル化（encapsulation）の3つがよく挙げられる。

Q 4.1.1 左の語句の意味と対応する右の説明を線で結べ。

[1]アドホックな多相に対して、型によって処理が変わらない多相のことをパラメトリック（parametric）な多相と言う。

動的束縛　・　・クラスの実装の詳細を他のクラスから隠すことができること
継承　・　・呼び出されるメソッドがオブジェクトの実行時の型で決まること
カプセル化　・　・上位クラスのメソッドの実装を下位クラスで再利用できること

カプセル化と継承は、ポリモルフィズムと動的束縛があってこそ意味がある概念である。ポリモルフィズムがあるから、実装の詳細を入れ替えても再コンパイルせずにコードを実行することができる。また、動的束縛があるから、継承したメソッドの意味がクラスに応じて自動的に変わる。そのため、プログラミング言語の実装という観点から見れば、ポリモルフィズム、特に動的束縛こそがオブジェクト指向の本質であると言っても良い。

4.2 オブジェクト指向と代数的データ型

Haskell や ML といった関数型言語では、複数の構成子 (constructor) からなる＿＿＿＿＿＿＿＿ (algebraic data type) を定義できる。代数的データ型を構成する各構成子を、オブジェクト指向型言語のクラスに相当すると見なせば、関数型言語もアドホックポリモルフィズムや動的束縛に相当する機構を持っている。関数は、さまざまな構成子の引数を受け取り、また呼び出されるコードがパターンマッチングにより、オブジェクトの実行時の構成子で定まる。

オブジェクト指向言語のクラスと関数型言語の代数的データ型の違いは、拡張性の方向である。代数的データ型は、既存の構成子に新しい関数を追加していくのは可能だが、既存の関数に新しい構成子を追加することはできない。（例えば組み込みの代数的データ型であるリスト型に新しい構成子を後から追加することはできない。）逆にクラスは既存の関数（メソッド）を新しいデータ型（クラス）に定義することは可能だが、既存のデータ型（クラス）に新しい関数（メソッド）を追加することはできない。（例えば、Java の組み込みのクラスである `String` クラスに、新しいメソッドを後から追加することはできない。）

型クラスは、関数型言語にオブジェクト指向言語と同じ方向の拡張性（既存の関数に新しい型を引数として追加する）を付与する仕組み、あるいはオブジェクト指向言語の動的束縛を関数型言語で解釈する仕組みと考えることもできる。

4.3 Haskell の型クラス

以下では Haskell の型クラスを説明する。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。一方、関数型は比較できない。例えば `(\ x -> x+x) == (\ x -> 2*x)` はエラーである。Haskell は多相を許す言語であるので、例えば、Prelude に定義されている

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

```
3
4   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+x) [\ x -> x+1, \ x -> 2*x] 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。一般に型クラスとは、_____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オブジェクトの集まり（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]。）

4.4 クラス宣言とインスタンス宣言

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。（Prelude に定義済み。）

```
1   class Eq a where
2     (==), (/=) :: a -> a -> Bool    -- /=ではないので注意
3     a /= b = not (a == b)           -- デフォルトの定義
```

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。

[2] というよりも、順序から言えば Java のインタフェースが Haskell の型クラスに似ている、というほうが適切である。

そして例えば以前紹介した Direction 型が Eq クラスに属する（Direction 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。

```
1  instance Eq Direction where
2    North == North = True
3    South == South = True
4    West  == West  = True
5    East  == East  = True
6    _     == _     = False
```

(/=) 演算子については、クラス宣言の中でデフォルトの定義が用意されているので、インスタンス宣言では定義を省略することができる。

また、Tree 型の場合、次のように宣言する。

```
1  instance Eq a => Eq (Tree a) where
2    Empty           == Empty           = True
3    Branch l1 n1 r1 == Branch l2 n2 r2
4                        = l1 == l2 && n1 == n2 && r1 == r2
5    _               == _               = 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。

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。

Q 4.4.1 組込みの Bool 型と等価なデータ型:

```
data MyBool = MyTrue | MyFalse
```

を Eq クラスのインスタンスとして宣言せよ。（Bool 型に対する Eq クラスのインスタンスは標準ライブラリ中で宣言されているので、宣言をプログラム中に書く必要はない。）

Q 4.4.2 第 3 章で定義したじゃんけん（RPS）型を Eq クラスのインスタンスとして宣言せよ。

4.5 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。（以下の説明用コードでは主要でないメソッドは省略している。）

```
1  class Eq a => Ord a where    -- Ord は order （順序）のこと
2     (<), (<=), (>=), (>) :: a -> a -> Bool
3     max, min             :: a -> a -> a
4      ...
5
```

```
6   class Show a where
7      show                  :: a -> String
8       ...
9
10  class (Eq a, Show a) => Num a where
11     (+), (-), (*)         :: a -> a -> a
12     fromInteger           :: Integer -> a
13      ...
14
15  class Num a => Fractional a where
16     (/)                   :: a -> a -> a
17      ...
```

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、Eq は Ord のスーパークラスである。これは、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、ということを意味する。

実は、これまで触れていなかったが、1 や 3.14 などの数値リテラルは、Haskell ではそれぞれ、

```
1    :: (Num t) => t
3.14 :: (Fractional t) => t
```

という型を持っている。だから、例えば 1 という数値リテラルは、Int としても、Double としても使用できる。

Show, Eq, Ord クラスなどに対するインスタンス宣言は、ほとんどのデータ型で必要になり、しかも同じような定義になるので、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。

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

これで、Tree に対して ==, >, show などのメソッドが定義される。

Q 4.5.1 組込みのリスト型と等価なデータ型

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

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。

（クラスの定義中にデフォルトの実装が定義されているので、Eq クラスの==メソッドと Ord クラスの<=メソッドだけを定義すれば、他のメソッドの定義は自動的に生成される。）

4.6 型クラスとオブジェクト指向言語の型システムの違い

型クラスは、Haskell にオブジェクト指向言語的な拡張性を取り入れているが、オブジェクト指向言語の型システムで可能であることをすべて模倣できるわけではない。

オブジェクト指向言語では、あるスーパークラスを継承するクラスに属するオブジェクトは、一つの変数に代入したり、一つのコンテナーにまとめたりすることができる。しかし、Haskell では、例えば Integer と Char と [Integer] はすべて Show クラスに属するが、それらを一つのリストにまとめることはできない。つまり、次のようなプログラムは型付けできない。

```
-- let impossible = [1,'a',[1,4,7]]
-- in  map show impossible
```

もちろん型システムの安全性を妥協すれば、このような拡張はいくらでも可能だが、Haskell では型システムの安全性（型チェックを通ったプログラムは、実行時に型エラーを起こさないということ）が優先されている。

4.7 型クラスの問題点

型推論とアドホック多相をエレガントに両立したと言える型クラスだが、いくつかの問題点も残っている。

曖昧さ (ambiguity)　例えば、次のようなクラス定義があるとする。

```
1  class Show a where
2    show :: a -> String
3  class Read a where
4    read :: String -> a
```

この定義の元で、次のような関数を定義する。

```
foo x = show (read x)
```

この関数の型は、foo :: (Show a, Read a) => String -> String となる。型変数 a の型は、=>の左辺にしか現れないので、型推論で決定できない。（すなわち曖昧である。）すると次のプログラムのように、

```
foo x = show (read x :: Integer)
```

a の具体的な型をユーザが明示しなければ意味が定まらなくなってしまう。（プログラムの意味が型宣言に依存することになり、気持ち悪い。）

単相性制限 (monomorphism restriction)　とは、(大雑把に言えば)「関数束縛でなくしかも明示的な型が与えられていない変数束縛では、いずれかの型クラスに属する型変数は一般化しない」という、いささか不自然な規則のことである。

例えば、次のような関数を考える。

```
genericLength :: Num a => [b] -> a
genericLength []     = 0
genericLength (_:xs) = 1 + genericLength xs
```

genericLength 関数はライブラリ関数の `length :: [b] -> Int` 関数と同じ定義を持つ関数だが、戻り値の型がより一般化されている。

この関数を利用して、次のような式を定義する。

```
let v = genericLength [1,2,3,4]
  in (v, v)
```

単相性制限がなければ、Dictionary-Passing Style 変換を行なうと、上の式は次のように変換される。

```
\ d1 d2 -> let v' d = genericLength' d [1,2,3,4]
              in (v' d1, v' d2)
```

すると genericLength (の DPS 変換後の関数) が 2 度呼び出され、同じリストの長さの計算を 2 度行なってしまうという変なことが起こる。単相性制限のもとでは、次のように DPS 変換される。

```
\ d -> let v = genericLength' d [1,2,3,4]
         in (v, v)
```

しかし、この場合、2 つの v の出現を別の型で使用することはできない。

わかりにくいエラーメッセージ　型クラスはエラーメッセージがわかりにくい、またはそもそもエラーにならない、という欠点がある。参考文献 [4] に多くの例が挙げられている。

例えば、

```
f0 xs = map (+1) xs    -- (+1) は \ x -> x+1 の意味
```

と書くべきところを間違って、

```
-- セクションの括弧忘れ
f0 xs = map +1 xs
```

と定義したとする。この定義は型エラーにならず。

```
f0 :: (Num (t -> (a -> b) -> [a] -> [b]),
       Num ((a -> b) -> [a] -> [b])) =>
      t -> (a -> b) -> [a] -> [b]
```

という型を持つと判定される。以下のような定義も型エラーとならず、変な型になってしまう。

```
-- (-1) をセクションとして使えると勘違いした。
-- f1 xs = map (\ x -> x-1) xs と書かなければならない。
f1 xs = map (-1) xs

-- : 演算子の型を勘違いした。
-- f2 n = n : [4,5,6] と書くべきである。
f2 n = [n] : [4,5,6]

-- Haskell では浮動少数点数リテラルを . から開始することはできない。
-- . 演算子と解釈されてしまう。
-- f .  g = \ x -> f (g x)
-- f3 r = r * sin 0.2 と書くべきである。
f3 r = r * sin .2
```

Q 4.7.1 上記の f1, f2, f3 の型を確かめよ。

4.8 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、このような実装方法が 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方法の一例である。）

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。

まず、クラス宣言は次のようなメソッドを要素に持つ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' = \ (e, _) -> e         -- (==) に対応する

ne' :: Eq' a -> (a -> a -> Bool)
ne' = \ (_, n) -> n         -- (/=) 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。

```
eqDirectionDic :: Eq' Direction
eqDirectionDic = (eqDirection, \ a b -> not (a `eqDirection` b))
  where North `eqDirection` North = True
        South `eqDirection` South = True
        West  `eqDirection` West  = True
        East  `eqDirection` East  = True
        _     `eqDirection` _     = False

eqTreeDic :: Eq' a -> Eq' (Tree a)
eqTreeDic (eqA, _) = (eqTree, \ a b -> not (eqTree a b))
  where Empty `eqTree` Empty  = True
        Branch l1 n1 r1 `eqTree` Branch l2 n2 r2
          = l1 `eqTree` l2 && n1 `eqA` n2 && r1 `eqTree` r2
        _     `eqTree` _      = False
```

そして型クラスを使っている（ ... => ... という型を持つ）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（Eq' a 型）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' :: _______ -> a -> [a] -> Bool
member' d x []     = False
member' d x (y:ys) = eq' d x y || member' d x ys

subset' :: _______ -> [a] -> [a] -> Bool
subset' d xs ys = all (\ x -> member' d x ys) xs
```

これらの関数の呼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 North [North, South, West]
  ↦ member' ____________________ North [North, South, West]
subset [North, South] [North, South, West]
  ↦ subset' _____________________ [North, South] [North, South, West]
```

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関数を取り出し、それを起動するだけになる。要するに、動的束縛は高階関数で解釈できる[3]。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 4.8.1 次のように定義されている関数 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
```

[3]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可能性はある。

（lookup は標準ライブラリに定義済みの関数である。）の Dictionary-Passing Style 変換後の形 lookupD:

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

を示せ。例えば、lookup 1 [(1,2),(4,7)] と lookupD eqIntDic 1 [(1,2),(4,7)] の値が、あるいは lookup 1.1 [(1.4,0.1),(4.2,6.9)] と lookupD eqDoubleDic 1.1 [(1.4,0.1),(4.2,6.9)] の値が同じ値になる。（eqIntDic, eqDoubleDic は各自で適宜定義する。）

4.9 一般的なオブジェクト指向言語の実装について

一般的なオブジェクト指向言語でも、たいていは “メソッド辞書” に対応するデータを扱うことで、動的束縛を実現している。しかし、関数の独立した引数としてではなく、オブジェクトに付随する形になっていることが多い。つまり、各オブジェクトがクラスに対応するデータ構造へのポインタを持っていて、クラスに対応するデータ構造がメソッドの辞書を含んでいるという場合が多い。

一方、代数的データ型の場合は、メソッド辞書に相当するものは各関数に付随しているかたちになる。

Smalltalk のメソッド呼出しの実装の方法は、例えば参考文献 [5] の第 6 章に説明されている。

JavaScript は、クラスではなく＿＿＿＿＿＿＿＿（prototype）という概念に基づくオブジェクト指向を採用している。（ちなみにプロトタイプ方式を最初に広めた言語は Self と言う言語である。）JavaScript のメソッド呼出しの仕組みは [6] の第 6 章に解説がある。ただし、最近の JavaScript はクラスも採り入れている。

Common Lisp のオブジェクト指向拡張（CLOS）は多重メソッド（multi-method）と言って、他の多くのオブジェクト指向言語と異なり、2 つ以上のパラメータの型（クラス）によって実際に呼出すメソッドの実装を決定する仕組みを持っている。

Java では、char, int や double などのプリミティブ型に対して、他のオブジェクトと異なる扱いをする。これは、実行時にこれらのプリミティブ型のデータに型（クラス）の情報を持たせることができないからである。

問 4.9.1 実際のオブジェクト指向言語（Smalltalk[5], CLOS, JavaScript[6], C++, Java[7], Python, Ruby など）で動的束縛や継承がどのように実装されているか調べよ。

4.10 まとめ

型クラスは、Haskell でアドホック多相を可能にするための仕組みである。既存の関数を新しいデータ型に定義するための仕組みでもある。コンパイル時に高階関数への変換が行なわれるので、実行時のコストはほとんどかからない。しかし、わかりにくいエラーメッセージなどの問題は残っている。

4.11　さらに詳しく知りたい人のために ...

文献 [1] は、型クラスのアイデアを最初に紹介した論文である。文献 [2] は、現在の Haskell の型クラスを詳しく説明している。文献 [3] は、Haskell の（型クラスに関する部分を含む）型推論を、具体的に Haskell のプログラムを用いて説明している。文献 [4] は、型クラスのエラーメッセージの問題点と改良法について詳しく述べている。

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

[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

[3] Mark P. Jones 「Typing Haskell in Haskell」 2000 年 11 月
`http://www.cse.ogi.edu/~mpj/thih/`

[4] Bastiaan Heeren and Jurriaan Hage 「Type Class Directives」 2005 年 1 月
Seventh International Symposium on Practical Aspects of Declarative Languages (LNCS 3350), pp.253–267

[5] Mark Guzdial and Kim Rose 編、 軋音組 訳
「Squeak 入門 過去から来た未来のプログラミング環境」 2003 年 3 月 星雲社

[6] 久野 靖 「入門 JavaScript」 2001 年 8 月 ASCII

[7] Tim Lindholm and Frank Yellin 著、村上 雅章 訳
「Java 仮想マシン仕様 第 2 版」 2001 年 5 月 ピアソン・エデュケーション