第4章　型クラス

ポリモルフィズム（多相）とは ____________________________ 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、__________（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 ________（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 ________________（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・____（inheritance）・__________（encapsulation）の 3 つがよく挙げられる。

動的束縛	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
継承	____________________________________
カプセル化	____________________________________

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。そのため、プログラミング言語の実装の観点から見れば、ポリモルフィズムと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。

[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。

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 `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。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]）

4.4 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。（Prelude に定義済み。）

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  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。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追加することができる。例えば、§ 3.9 で紹介した Tree 型の場合、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] というようりも、Java のインタフェースが型クラスに似ている。

4.5 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[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 4.5.1 次のように定義されている関数 *lookup:*

[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。

```
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
```

を示せ。（*lookup* は標準ライブラリに定義済みの関数である。）

4.6 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。）

```
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
   ...
```

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。実は、これまで触れていなかったが、1, 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。

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 4.6.1 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。

（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。）

4.7 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、一つの配列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、関数型言語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。

4.8 一般的なオブジェクト指向言語について

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 [4] 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 [5] の第 6 章に解説がある。

Common Lisp のオブジェクト指向拡張（CLOS）は ＿＿＿＿＿＿＿＿（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 4.8.1 実際のオブジェクト指向言語（*Smalltalk, CLOS, JavaScript, C++, Java* 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。

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、具体的に Haskell のプログラムを用いて説明している。

[4] 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
Squeak は今最もポピュラーな Smalltalk の実装の一つである。

[5] 久野 靖 「入門 JavaScript」 2001 年 8 月 ASCII
ホームページでの利用法ではなくて、JavaScript というプログラミング言語そのものを深く知りたい人向けの JavaScript の入門書である。

[6] 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
その名のとおり JVM の仕様に関する本である。

[7] 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
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述べている。