第8章　オブジェクト指向

この章では、これまでの章で取り上げてきた “制御構造” とは直交するプログラミング言語の特徴である “オブジェクト指向” を取り上げる。

オブジェクト指向の導入の具体的なやりかたには、いくつも選択肢があり、どれがベストかを決めることは難しい。現実のプログラミング言語と同じようなやりかたでオブジェクト指向を導入しようとすると、かなり大がかりなモノになってしまう恐れがある。

そこでまずオブジェクト指向がどのようなモノであるかを整理し、できる限り最小限の拡張として対象言語 Util に導入することにする。

8.1　オブジェクト指向とは

オブジェクト指向を特徴づけるキーワードとして _____（dynamic binding）・_____（inheritance）・_____（encapsulation）の 3 つがよく挙げられる。

動的束縛　_____
継承　_____
カプセル化　_____

このうち動的束縛は、_____（polymorphism, 多相）という概念を前提としている。

ポリモルフィズム　_____

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だろう。

このうち、ポリモルフィズムについては、Util ファミリーはもともと強い型付けのない言語であるため、はじめから問題とはならない。UtilOOP では動的束縛（と継承）の実装に焦点を絞り、副作用を持たない単純な言語である UtilRec に動的束縛と継承を追加することにする。（状態やエラー処理・入出力などはオブジェクト指向とは直交する独立な概念であるので、UtilOOP への導入はチャレンジ問題としておく。）

また、カプセル化についても、隠すこと自体は実行時の問題ではなく、型検査（type check）と同じく静的解析（static analysis）の段階で取り扱う問題と考えられるので、Util ファミリーの想定範囲外として、ここでは取り扱わないことにする。

8.2 クラスと代数的データ型

Haskell や ML といった関数型言語で一般的な代数的データ型 (Algebraic Data Type) もある意味ではポリモルフィズムや動的束縛という特徴を持っている。関数は様々な構成子 (constructor) の引数を許すし、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。

クラスと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。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。

Util の場合には、強い型付けもないし、効率も二の次ので、両方の拡張性を与えることが可能である。この章の以下の議論は、Util への代数的データ型の導入と捉える事も可能である。

8.3 クラス・メソッド 定義の文法

はじめに UtilOOP での、クラス・メソッド の定義の文法とそれを読み込んだ時のデータ型を定めておく。ここでは、構文解析と解釈をできるだけ単純にすることを優先してデザインする。

まず、クラスやメソッドは大域的に（つまりプログラムの一番外側で）定義されるものとする。関数などの内部などでは定義されない。大域宣言の構文規則は次のようなものとする。

$$
\begin{array}{rcl}
\mathit{Decl} & \rightarrow & \mathit{Ident}\ \mathit{Ident}\ \ldots\ \mathit{Ident} = \ \mathit{Expr} \\
\mathit{CPat} & \rightarrow & \mathit{Ident} \ \mid \texttt{(}\ \mathit{Ident}\ \mathit{Ident}\ \ldots\ \mathit{Ident}\ \texttt{)} \\
\mathit{ExtPat} & \rightarrow & \mathit{CPat} \ \mid \ \mathit{Ident}\ \texttt{@}\ \mathit{CPat} \\
\mathit{MethodDecl} & \rightarrow & \mathit{Ident}\ \mathit{ExtPat}\ \ldots\ \mathit{ExtPat} = \mathit{Expr} \\
\mathit{GlobalDecl} & \rightarrow & \textbf{let}\ \mathit{Decl}\ \textbf{in}\ \mathit{GlobalDecl} \\
 & \mid & \textbf{letrec}\ \mathit{Decl}\ \textbf{in}\ \mathit{GlobalDecl} \\
 & \mid & \textbf{class}\ \mathit{Decl}\texttt{;}\ \ldots\ \texttt{;}\ \mathit{Decl}\ \textbf{in}\ \mathit{GlobalDecl} \\
 & \mid & \textbf{method}\ \mathit{MethodDecl}\texttt{;}\ \ldots\ \texttt{;}\ \mathit{MethodDecl}\ \textbf{in}\ \mathit{GlobalDecl} \\
 & \mid & \mathit{Expr}
\end{array}
$$

つまり、クラス宣言（**class**）とメソッド宣言（**method**）、それに **let** と **letrec** による通常の関数などの定義のあとに、式が続く形になっている。

また、クラス名（構成子名）としてはアルファベットの大文字で始まる名前、メソッド名としてはアンダースコア（_）で始まる名前を使用することにして通常の関数名と区別する。

この文法によると UtilOOP では、クラスは例えば次のように定義される[1]。

```
class Point x y = Unit in
class ColorPoint x y c = Point x y in ...
```

クラス定義は単にクラスと同じ名前を持つ構成子（コンストラクタ）を定義する構文だと考える。“=” の左辺にクラス名（構成子名）とオブジェクトの構成要素（フィールド）を並べて書く。上の例では

[1]本当は `class ColorPoint x y c extends Point x y in ...` と書けば本物のオブジェクト指向言語っぽいのだが、UtilOOP では単に構文解析部を書き直す手間をケチってこのように書くことにしているのである。

Point は 2 つのフィールドを持つクラスであり、ColorPoint は 3 つのフィールドを持つクラスである。“=” の右辺はスーパークラスであり、いま定義されているクラスのフィールドがスーパークラスのフィールドにどのように対応しているかを示している。ここで、Unit はすべてのクラスのスーパークラスである（Java の Object クラスに相当する。）

また、メソッドの定義については、UtilOOP では、次のように書く。

```
method
  _setX (Point x y) = \ x1 -> Point x1 y;
  _setY (Point x y) = \ y1 -> Point x y1;
  _move (Point x y) = Point (x+1) (y+1)
in ...
```

これで Point クラスに対するメソッド _setX, _setY, _move を定義している。また、_move メソッドを、

```
_move p@(Point x y) = let p1 = _setX p (x+1) in _setY p1 (y+1)
```

のように定義し、オブジェクト自身にあたる変数（Java の _____ に相当）を “@” の前に書いて右辺で利用することもできる。この 2 つの_move メソッドの定義は、Point のサブクラスにこのメソッドを適用する時に意味が異なってくる。

UtilOOP は、以下の点でよくあるオブジェクト指向言語の文法と異なっているので注意する必要がある。

- カプセル化を考慮していないので、メソッドの定義はクラス定義の中ではなく、プログラムの一番外側ならばどこにでも書くことができる。
- オブジェクト指向言語では、

  オブジェクト (. などの演算子) メソッド 引数 ...

  という順番でメソッドの呼出しを書くものが多いが、UtilOOP では、普通の関数呼び出しと同じように、

  メソッド オブジェクト その他の引数 ...

  という形で書くことにする。

このクラス宣言・メソッド宣言を含む大域宣言に対する抽象構文木として、次のようなデータ型 Global を用意する。

```
type Lhs = (String {-関数名-}, [String] {-仮引数の並び-});
type Decl = (Lhs {-左辺-}, Expr {-右辺-});

type CPat = (String {-構成子名-}, [String] {-仮引数の並び-});
data Pat = ConsPat CPat | AsPat String {-@の左-} CPat {-@の右-}
         deriving Show;
type MethodLhs = (String {-メソッド名-}, [Pat] {-パターンの並び-});
type MethodDecl = (MethodLhs {-左辺-}, Expr {-右辺-});

data Global = LetG [Decl] Global | LetrecG [Decl] Global
            | ClassG [Decl] Global | MethodG [MethodDecl] Global
            | ExprG Expr deriving Show;
```

この章の最終目標はこの大域宣言を解釈する関数:

```
interpGlobal :: Global -> Env -> M Value;
```

を定義することである。

8.4 動的束縛の実現

動的束縛については、どのようにして実行時にオブジェクトの型に応じて適切なメソッドの実装を選択するかが問題となる。UtilOOP では簡単のため、環境と同じように連想リストを用いることにする。この連想リストは _____をキーとする。

```
type Dict = [((String, String), Value)];
```

この連想リストを環境と区別するために特に _____（method dictionary）、あるいは単に辞書と呼ぶことにする[2]。

UtilOOP の計算の型 M はこの辞書を隠れた引数として受け渡す型として表す。このモナドの基本関数の定義は次のようになる。

```
type M a = Dict -> a;

unitM :: a -> M a;
unitM a = \ d -> a;

bindM :: M a -> (a -> M b) -> M b;
m `bindM` k = \ d -> let { a = m d } in k a d;
```

この辞書は環境と同じようなものに見えるかも知れないが、その受け渡され方は環境と大きく異なる。環境の場合は、関数が実行される時、その中の式は関数が呼び出される環境ではなく関数が定義された環境で評価される。それは下の関数抽象と関数適用に関する interp の定義からわかる。

```
interp (Lambda x m) e = unitM (Fun (\ v -> interp m ((x, v) : e)));
interp (App f x)    e = interp f e `bindM` \ v  ->
                        case v of {
                          Fun g -> interp x e `bindM` \ y ->
                                   g y;
                          _     -> failM "Function expected"
                        };
```

これは、C 言語・Java・Scheme・Haskell などほとんどのプログラミング言語で採用されている静的スコープ（static scope）のルールである。

しかし、辞書については _____場所ではなく、_____場所で有効な辞書が関数に渡される。（上の M の定義で実際にそうなることを確認せよ。）さもないと、あるクラスで定義されるメソッドの中から、後で定義されるサブクラスに対するメソッドの実装が見えず、動的束縛がうまく動作しない。

8.5 クラスとメソッドの表現

まず簡単のために継承のない場合を考える。

メソッド名・クラス名は単純に String 型で表現する。オブジェクトはクラス名と構成要素（フィールド）のリストの組として表す。UtilOOP で扱う “値” の型を次のように定義する。

```
data Value = ...  | MethodV String | ObjectV String [Value];
```

関数適用の関数の位置にあるものが通常の関数ではなくメソッドだった場合、オブジェクトの属する

[2] もちろん、より現実的なプログラミング言語では、メソッドの呼び出しは頻繁に起こることなので、辞書としてもっと効率の良いデータ構造を使用する必要がある。

クラスに応じて適切なメソッドの実装を選択し、起動する必要がある。そこで、関数適用（App）に対する interp は次のように変更される。

```
interp (App f x) e = interp f e `bindM` \ v  ->
                     case v of {
                       Fun g     -> interp x e `bindM` \ y ->
                                    g y;
                       MethodV m -> interp x e `bindM` \ y ->
                                    applyMethod m y y e;
                       _         -> failM "Function expected";
                     };
```

ここで、applyMethod はオブジェクトのクラスに対応するメソッドの実装を検索する関数である。interp は関数の位置にあるものがメソッドだった時、この applyMethod を呼び出す。applyMethod に同じオブジェクト（y）が 2 つ渡されている理由は、継承の説明のところで明らかにする。

```
applyMethod :: String -> Value -> Value -> Env -> M Value;
applyMethod m self (ObjectV c vs) e dic =
  case lookup (m, c) dic of {
    Just f  -> interp (App (App (Const f) (Const self))
                           (Const (ObjectV c vs))) e dic
  };
applyMethod m self _ e dic =
    error ("Method "++m++" is applied to non-object.");
```

また、lookup は Haskell の Prelude（標準ライブラリ）に次のように定義されている連想リストからキーに対応している値を検索する関数である[3]。

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

lookup :: Eq a => a -> [(a, b)] -> Maybe b;
lookup k []          = Nothing;
lookup k ((x,y):xys) = if k==x then Just y else lookup k xys;
```

なお、関数適用（App）以外に対する interp 関数の書き換えは必要ない。

8.6 クラス・メソッド定義の解釈

クラス・メソッド定義については辞書データを書き換えるために明示的に扱う必要があるので、次のように UtilOOP の計算の型 M ではなく、いわばその “1 つ上のレベル” で計算している。UtilOOP の場合、1 つ上のレベルの計算の型は、UtilRec の計算の型であるトリビアルな計算の型 Id になる。

```
type Id a = a;

unitId :: a -> Id a;
unitId a = a;

bindId :: Id a -> (a -> Id b) -> Id b;
m `bindId` k = k m;
```

状態・エラー処理など他の特徴を持つ言語の場合、1 つ上のレベルは、その特徴に対応した計算の型（ST や Err など）になる。

interpGlobal は、補助関数 interpClassDecls や interpMethodDecls を呼び出して、その結果として返される新しい環境と辞書で残りの部分を解釈する。

[3] lookup の型中の Eq a =>は a に Equality（==）が定義されているという、型変数に対する制約を表す。

```
interpGlobal :: Global -> Env -> Dict -> Id Value
...  -- LetG, LetrecGに対する部分は省略
interpGlobal (ClassG decls n) e dic  =
           interpClassDecls  decls e dic `bindId` \ (e1, d1) ->
           interpGlobal n e1 d1;
interpGlobal (MethodG decls n) e dic =
           interpMethodDecls decls e dic `bindId` \ (e1, d1) ->
           interpGlobal n e1 d1;
interpGlobal (ExprG exp) e dic       = interp exp e dic;
```

この interpClassDecls と interpMethodDecls は実際にクラス定義・メソッド定義を処理する関数: interpClassDecl と interpMethodDecl を順番に呼び出すだけの関数である。

```
interpClassDecls :: [Decl] -> Env -> Dict -> Id (Env, Dict);
interpClassDecls (decl:decls) e0 d0 =
  interpClassDecl decl e0 d0   `bindId` \ (e1, d1) ->
  interpClassDecls decls e1 d1;
interpClassDecls []           e0 d0 = unitId (e0, d0);

interpMethodDecls :: [MethodDecl] -> Env -> Dict -> Id (Env, Dict);
interpMethodDecls (decl:decls) e0 d0 =
  interpMethodDecl decl e0 d0   `bindId` \ (e1, d1) ->
  interpMethodDecls decls e1 d1;
interpMethodDecls [] e0 d0           = unitId (e0, d0);
```

interpClassDecl はコンストラクタ関数を定義し、それを環境に登録する。

```
interpClassDecl :: Decl -> Env -> Dict -> Id (Env, Dict);
interpClassDecl ((id, args), rhs) e0 d0 =
  let {
    loop []     vs = ObjectV id (reverse vs);
    loop (x:xs) vs = Fun (\ v -> unitM (loop xs (v:vs)));
    constr = loop args []
  } in
  unitId ((id, constr) : e0, d0);
```

interpMethodDecl はメソッドの実装を生成し、メソッド辞書にその実装を格納する。

```
interpMethodDecl :: MethodDecl -> Env -> Dict -> Id (Env, Dict);
interpMethodDecl ((id, [pat]), rhs) e0 d0 =
  let {
    m = Fun (\ self ->
          unitM (Fun (\ (ObjectV _ args) ->
            let {
              alist = case pat of {
                  AsPat x (c, ps) -> (x, self) : zip ps args;
                  ConsPat (c, ps) -> zip ps args
              }
            } in
            interp rhs (alist ++ e0))));
    c = case pat of {
      AsPat _ (c, _) -> c;
      ConsPat (c, _) -> c
    };
  } in
  unitId (newMethodId id e0, ((id, c), m):d0);

newMethodId id e0 = case lookup id e0 of {
                      Some (MethodV _) -> e0;
                      _                -> (id, MethodV id) : e0
                    }
```

辞書に格納されるメソッドの実装 m の第 1 引数（self）はオブジェクト自身に相当する引数である。この引数の使い方は、継承を導入した時に説明する。また、この m は第 2 引数として渡されるオブジェクトの各要素（フィールド）の値（args）を束縛した新しい環境（alist ++ e0）を生成し、右辺（rhs）の式を評価する。

8.7 継承の実現

継承を実現するためには、辞書の中にそのクラスに対応するメソッドの実装が見つからなかったとき、代わりにスーパークラスのメソッドを検索できるようにすれば良い。

UtilOOPでは継承を次のように実現している。まず、クラスを定義する時に、__superという特別な名前のメソッドを、スーパークラスのオブジェクトを戻り値とするメソッドとして定義する。

```
interpClassDecl ((id, args), rhs) e0 d0 =
  let {
    loop []     vs = ObjectV id (reverse vs);
    loop (x:xs) vs = Fun (\ v -> unitM (loop xs (v:vs)));
    constr = loop args [];
    super = Fun (\ self ->
              unitM (Fun (\ (ObjectV _ ys) ->
                let {
                  alist = zip args ys
                } in
                interp rhs (alist++e0))))
  } in
  unitId ((id, constr) : e0, (("__super", id), super):d0);
```

そして、applyMethod関数の中で、メソッドが見つからなかった時、この__superを呼び出して、スーパークラスのオブジェクト（obj1）にキャスト（変換）し、そのオブジェクトに対して、applyMethodを再帰的に呼出す。

```
applyMethod m self obj e dic =
  let { ObjectV c _ = obj } in
  case lookup (m, c) dic of {
     Nothing -> let { Some super = lookup ("__super", c) dic } in
                interp (App (App (Const super) (Const self)) (Const obj))
                    e dic `bindId` \ obj1 ->
                applyMethod m self obj1 e dic;
     Just f  -> interp (App (App (Const f) (Const self)) (Const obj)) e dic
  };
```

再帰呼出しの際、第1引数には、キャストされたオブジェクト（obj1）ではなく、常にキャストされる前のオブジェクト（self）が渡されていることに注意する。こうでないと、動的束縛はうまく動作しない。

最後にrunは空の辞書をプログラムに渡す。

```
initDict :: Dict;
initDict = [];

run :: String -> String;
run prog = showValue (interpGlobal (parse prog) initEnv initDict);
```

これで、UtilOOPの実装は完成である。

8.8 実行例

例として、次のようなUtilOOPのプログラムを考える。

```
class Point x y = Unit in
method
  _getX (Point x y) = x;
```

```
  _getY (Point x y) = y;
  _setX (Point x y) = \ x1 -> Point x1 y;
  _setY (Point x y) = \ y1 -> Point x y1;
  _move self@(Point x y) = let p1 = _setX self (x+1) in _setY p1 (y+1);
  _print (Point x y) = "Point (" ++ (toString x) ++ ", " ++ (toString y) ++ ")"
in
class ColorPoint x y c = Point x y in
method
  _getC (ColorPoint x y c) = c;
  _setC (ColorPoint x y c) = \ c1 -> ColorPoint x y c1;
  _setX (ColorPoint x y c) = \ x1 -> ColorPoint x1 y c;
  _setY (ColorPoint x y c) = \ y1 -> ColorPoint x y1 c;
  _print (ColorPoint x y c) =
    "ColorPoint (" ++ (toString x) ++ ", "  ++ (toString y)
    ++ ", " ++ (toString c) ++ ")"
in
  (_print (_move (Point 0 1))) ++ ", "
  ++ (_print (_move (ColorPoint 2 3 "red")))
```

これを実行すると、

```
_____________________________________________________________
```

のような結果が出力される。メソッド _move は ColorPoint クラスに対しては定義されていないが、継承により ColorPoint クラスに対しても _move メソッドが呼び出せること、そしてその場合、_move の中の _setX, _setY は ColorPoint に対して定義されているものが呼び出されていることがわかる。

問 8.8.1 メソッド呼出しを実装している関数 *applyMethod* にオブジェクト自身を表す引数（第 *1* 引数）とキャストされた引数（第 *2* 引数）の *2* つが必要な理由を考えよ。もしこの *2* つをまとめてしまうと、上のプログラムの実行例はどのような結果になるか？

問 8.8.2 （チャレンジ） *UtilOOP* に状態・入出力・エラー処理などの特徴を導入した *Util* のバリエーションを実装せよ。

8.9 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 “辞書” に対応するデータを隠れた引数とすることで、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。（連想リストやハッシュ表を引くのは、実用的な言語では非現実的である。）

一方、代数的データ型の場合は、辞書は各関数に付随している形になる。

具体的に辞書の中で目的のメソッドを探し出す方法の詳細は、各言語により固有の事情があり、さまざまである。

Smalltalk のメソッド呼出しの実装の方法は、例えば参考文献 [2] 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 [3] 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 8.9.1 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。

問 8.9.2 （チャレンジ） *UtilOOP* に *CLOS* のような多重メソッドを導入せよ。

この章の参考文献

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman 著、和田 英一 訳
「計算機プログラムの構造と実行 第 2 版」（原題: Structure and Interpretation of Computer Programs）
2000 年 2 月 ピアソン・エデュケーション
別名 purple book と呼ばれる名著である。型（クラス）と演算（メソッド）の組をキーとして実装を決定する方法は、この本の § 2.4 で**データ主導プログラミング**（data-directed programming）と呼ばれている。

[2] 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。

[3] 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。

[4] 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。