プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 9a: データ抽象

目次

型システムの目的 (再掲)

[スライド No.1 から]

型システムの目的 4. プログラムの「抽象化」の手段

- 抽象化 (ここでの意味) = インタフェースと実装の分離
- 抽象データ型 [Liskov; Turing Award (2008)]

「抽象化」とは何か？

データ抽象 (1/2)

データ抽象＝データとデータの操作に関する「インタフェース」と「実装」の分離

例: 「整数を要素とするスタック」のデータ構造の実現:

```
type stack =
    | Empty
    | Push of int * stack;;
let empty_stack () = Empty;;
let is_empty s = (s = Empty);;
let push x s = Push(x,s);;
let top s = match s with
   | Push(x,_) → x
   | Empty → failwith "Empty Stack";;
let pop s = ...
```

使用例:

```
let st = push 5 (push 3 (empty_stack ()));;
top (pop st);; (* ==> 3 *)
```

データ抽象 (2/2)

スタックのインタフェース:

```
type stack
empty_stack : unit → stack
is_empty    : stack → bool
push        : int → stack → stack
top         : stack → int
pop         : stack → stack
(* これらが「スタックらしく振舞う」ことも要求すべき.
たとえば, pop(push(x,s))=s などが成立すべき.  *)
```

スタックの実装はいろいろ:

- 前ページの実装
- `stack` を，`int list` として実現
- `stack` を，`int array` とその「長さ」の組で実現

抽象データ型

抽象データ型 (abstract data type) ＝インタフェース部分のみ

```
type stack
empty_stack : unit → stack
is_empty    : stack → bool
push        : int → stack → stack
top         : stack → int
pop         : stack → stack
(* これらの関数が満たすべき性質 *)
```

具体データ型 (具象データ型: concrete data type) ＝実装

抽象データ型の考え方: Barbara Liskov (Turing Award 2008)

- 実装の隠蔽 (Hiding)
  - そのデータ型を使う側は，インタフェースだけを参照する.
  - 実装を変更しても，使う側には影響がない.
- 大規模プログラムの開発で極めて有用

抽象化の手段

型を使えば，抽象データ型の考えに基づいたプログラミングは可能だが，情報の隠蔽は保証されない．(実装の詳細に依存するコードを書いてもエラーにならない．)

抽象化をサポートする言語機構:

- モジュール (ALGOL,Modula-2, ML 系言語)
- クラス (オブジェクト指向言語)
- 型クラス (Haskell)
- 最近では... 抽象クラス (Java,etc.),trait(Scala,Rust),protocol(Swift)

目次

OCaml モジュール (1/2)

スタックモジュールのインタフェース:

```
module type STACK = sig
  type stack
  val empty_stack : unit → stack
  val is_empty    : stack → bool
  val push        : int → stack → stack
  val top         : stack → int
  val pop         : stack → stack
end
```

スタックモジュールの実装その 1:

```
module Stack : STACK = struct
  type stack =
    | Empty
    | Push of int * stack
  let empty_stack () = Empty
  let is_empty s = (s = Empty)
  ...
end
```

OCaml モジュール (2/2)

スタックモジュールの実装その 2:

```
module Stack2 : STACK = struct
  type stack = int list
  let empty_stack () = ([] : int list)
  let is_empty s = (s = [])
  ...
end
```

モジュールシステム

モジュールシステムのポイント:

- データ抽象化 (実装の隠蔽) をある程度実現
  - インタフェースを使う側は、実装 (コード) がどうなっているかを知らない.
- インタフェースと実装の整合性を，型の整合性に帰着
  - インタフェース＝モジュールの持つべき型
  - 実装＝そのモジュール型を持つ要素
  - 型エラーの例: push が未定義、pop の型が違う，etc.
  - 型エラーでない例: 上記のほか dup 関数も持つ
- (欠点) 実装が満たすべき性質 (仕様) をインタフェースに書けない。

目次

オブジェクト指向とは？

オブジェクト指向 (Object Oriented) プログラム言語の特徴

- Abstraction (抽象化)
- Inheritance (継承)
- Subtyping (サブタイピング) ⇒ 後の授業資料を参照
- Dynamic lookup (動的ルックアップ)
- Open recursion (開いた再帰)

プログラム例 (Java 言語)

Point クラスの定義と、そのクラスのオブジェクトの生成

```
class Point {
  private double x;
  private double y;
  public double get_x () { return x; }
  public double get_y () { return y; }
  public void move (double dx, double dy) {
    x += dx; y += dy; }
  Point (double x_init, double y_init) {
    x = x_init;
    y = y_init;
  }
}
  ...
  Point p = new Point(3.14, 2.86);
```

get_x 等をメソッド (method) と呼ぶ.

抽象化

OO の抽象化＝実装の隠蔽 (Liskov の抽象化)

- 実装: 前ページの Point クラスの定義
- インタフェース: public 指定されたメソッドたちとその型

Java では上記 2 つを別々に書くこともできる。

```
interface PointInterface {
  double get_x ();
  double get_y ();
  void move (double dx, double dy);
}
class Point implements PointInterface {
  private double x; private double y;
  public double get_x () { return x; }
  ...
}
```

継承

継承＝「実装」の再利用

```
class Point { ... }
class ColoredPoint extends Point {
  private String c;
  public String get_c () { return c; }
  public void move (double dx, double dy) {
    super.move(dx * 10, dy * 10);}
  ColoredPoint(...) {}
}
```

Point ＝親クラス (super class), ColoredPoint ＝子クラス (sub class)

- 子クラスは、親の実装を再利用可能。
- 子クラスは、メソッド追加・メソッド実装変更できるが、親と同じインタフェースである必要。

サブタイプ (subtype)

Java では，子クラスは親クラスのサブタイプ (subtype) となる．

```
class Point { ... }
class ColoredPoint extends Point {...}
  ...
  Point p1 = new ColoredPoint(2.7, −1.5, "red"); // OK
  ColoredPoint cp2 = new Point(1.3, 2.8); // エラー
  ...
```

親 (Point) クラスの式を書くべきところに、子 (ColoredPoint) クラスの式を書くのは OK。

逆はエラー (型の整合性のエラー)。

動的ルックアップ

ルックアップ＝メソッドを起動する時，どの実装が使われるか探す事

動的ルックアップ＝動的 (実行時) に探す事

```
class Point implements PointInterface {
  public void move (double dx, double dy) {
    x += dx; y += dy; }
    ... }
class ColoredPoint extends Point {
  public void move (double dx, double dy) {
    x += dx; y += dy; }
  ... }
  ...
  Point p1 = new ColoredPoint(1.3, 2.8, "red");
  p1.move(1.0, −5.0);
```

p1.move で起動されるのは、p1 の型 (Point) の move 実装でなく、p1 の中身 (ColoredPoint のオブジェクト) が持つ move 実装。

[参考] 開いた再帰

Java の this (OCaml の self) は，オブジェクト自身を指す.

```
class Parent {
  public void go (int i) { this.go(i+1); }
  Parent () {}
}
class Child extends Parent {
  public void go (int i) {
    if (i < 17) { super.go(i); }
    else { System.out.println("Finished"); } }
  public void callgo (int i) { super.go(i); }
  Child () { }
}

  .....
  Child p3 = new Child();
  p3.callgo(0);   // 無限ループにならない！
```

p3.callgo(0) は，Parent の go を呼び出すが，その go の中の this.go は，その時点の this の go (p3 が持つ Child の go) である。

OO 言語たち

- Simula [1960 年代, K. Nygaard]...SmallTalk の OO に大きな影響
- SmallTalk [1970 年代, Xerox PARC 研究所, Alan Kay]
- C++ [1984-, Stroustrup]
- Java [1990-, Gosling]
- Ruby [1993-, Matsumoto]
- JavaScript [2005-, Eich]
- Scala [2003-, Odersky]
- 最近では，非常に多い.

演習問題

- モジュールとオブジェクト (オブジェクト指向) は、「抽象化」を提供するという点で概念上は類似しているが、違いも多い。モジュールシステムを持つ言語と、オブジェクト指向の言語を 1 つずつ取り上げ、両者のメリット、デメリットを比較しつつ論じなさい。
- Liskov の抽象データ型の考えは、$pop(push(x, s)) = s$ などの仕様 (満たすべき性質) をインタフェースに含めたものであった。現実の大多数のプログラム言語では、仕様を記述するための論理式などを表現できないので、自然言語等で記述している。これを克復するため、論理的な性質を記述することができる依存型やリファインメント型について研究がなされている。「依存型を持つプログラム言語」について調査し、その概要を説明しなさい。