プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 9b: オブジェクト指向の型システム

目次

オブジェクト指向言語とサブタイプ

オブジェクト指向言語の型システム

- Java: 静的型付けのオブジェクト指向言語、サブタイプ多相
- OCaml: Caml (ML 系言語)+Object、サブタイプと少し異なる多相

3 種類の多相型

- パラメータ多相
- アドホック多相
- サブタイプ多相

目次

Java 言語の型システムの基礎 (1/2)

考え方: クラス　型 (正確には、クラスのインタフェース　型)

```
interface PointInterface {
  double get_x (); double get_y ();
  void move (double dx, double dy);
}
class Point1 implements PointInterface {
  private double x; private double y;
  public double get_x () {...}
  public double get_y () {...}
  public void move (double a, double b) {...};
  public String toString () {...};
  Point1 (...) {...}
}
```

Point1 の実装が、PointInterface インタフェースを満たすとは:

- インタフェースにあるメソッドが、全て実装されている。
- インタフェースと実装における各メソッドの型は同じ。

Java 言語の型システムの基礎 (2/2)

Java の型システム (静的) が保証する実行時の性質の例:

- 「基本演算の型エラー (文字列を掛け算に渡す etc.)」は起きない。
- 「オブジェクトが持っていないメソッドを呼び出す」ことはない。

以下の性質は保証しない:

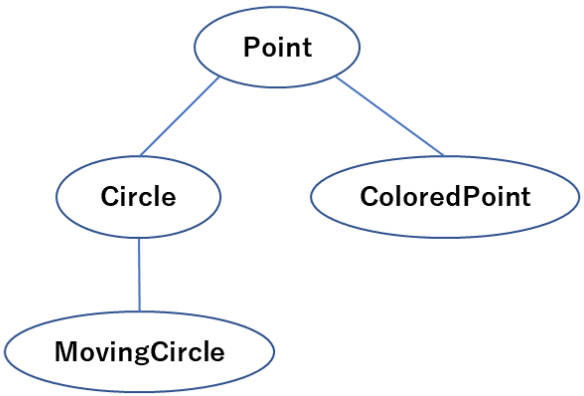
- 「0 による割り算」、「null ポインタ参照」、「配列の範囲外へのアクセス」が起きない。

サブタイプの基礎 (1)

サブタイプ関係 $A <: B$

- クラス A がクラス B を継承したら $A <: B$
- 推移的: $A <: B$ かつ $B <: C$ ならば $A <: C$

```
class Point {..get_x..}
class Circle extends Point {...}
class MovingCircle extends Circle {..tick..}
class ColoredPoint extends Point {...}
```



MovingCircle <: Circle は成立
MovingCircle <: Point は成立
MovingCircle <: ColoredPoint は不成立

サブタイプの基礎 (2)

サブタイプに対する型付け規則 (Liskov の置換規則):

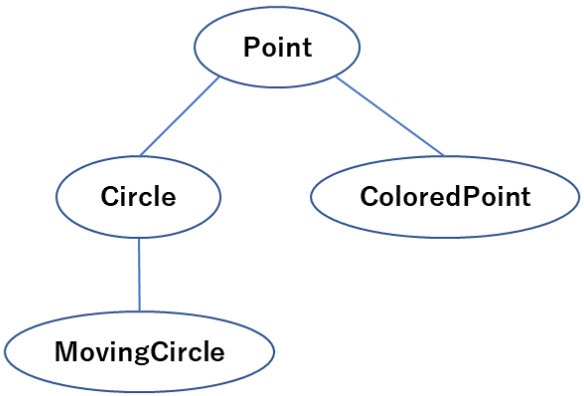
$$\frac{\Gamma \vdash M : T_2 \quad T_2 <: T_1}{\Gamma \vdash M : T_1}$$

子クラス (T_2) のオブジェクト (M) は、親クラス (T_1 のオブジェクトと思ってもよい。

M は複数の型を持つ ⇒サブタイプ多相 (subtyping polymorphism)

サブタイプの基礎 (3)

```
Circle c1 = new Point(...); // エラー
Circle c3 = new MovingCircle(...); // OK
Circle c4 = new ColoredPoint(...); // エラー
```

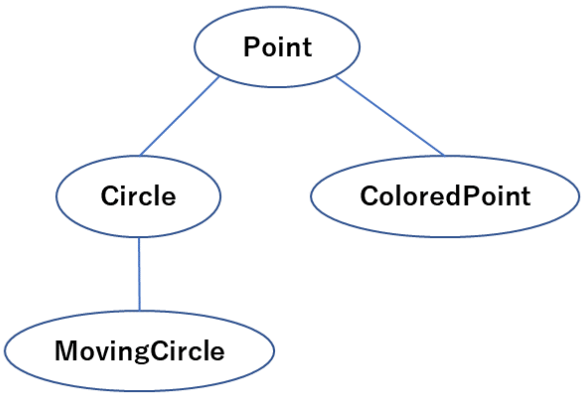


$$\dfrac{\dfrac{}{\vdash \texttt{new MC(...)} : \texttt{MC}} \quad \dfrac{}{\texttt{MC <: Circle}}}{\dfrac{\vdash \texttt{new MC(...)} : \texttt{Circle}}{\vdash \texttt{Circle c3 = new MC(...);} : \texttt{ok}}}$$

ここで、MCircle を MC と略記した。

サブタイプの基礎 (4)

```
class Point {double get_x..}
class MovingCircle extends Circle {int tick..}
Circle c3 = new MovingCircle(...);
c3.get_x();   // OK
c3.tick();   // エラー
```



$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{c3} : \texttt{Circle}} \quad \dfrac{}{\texttt{Circle} <: \texttt{Point}}}{\dfrac{\Gamma \vdash \texttt{c3} : \texttt{Point}}{\Gamma \vdash \texttt{c3.get\_x()} : \texttt{double}}}$$

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{c3} : \texttt{Circle}} \quad \dfrac{}{X \ \texttt{Circle} <: \texttt{MC}}}{\dfrac{\Gamma \vdash \texttt{c3} : \texttt{MC}}{\Gamma \vdash \texttt{c3.tick()} : \texttt{int}}}$$

ただし、$\Gamma = \texttt{c3} : \texttt{Circle}$ である。

サブタイプのまとめ

サブタイプ関係:

- 型の間の順序関係 (正確には、前順序 preorder)
- (素朴には) クラス間の親子関係＋推移律
- (後述) 構造のある型に対するサブタイプは複雑

サブタイプ多相:

- Liskov の置換規則

実は、

- サブタイプがある型システムは、型推論が困難
- → Java は明示的型付け 型検査でよい)
- → 多くのオブジェクト指向言語は動的型付け

(進んだ話題) 構造のある型に対するサブタイプ関係

問題: 基本型以外に対するサブタイプ関係はどうなっているか？

(進んだ話題) 構造のある型に対するサブタイプ関係

問題: 基本型以外に対するサブタイプ関係はどうなっているか？

Java の型の構文:

$$
\begin{array}{rcll}
T & ::= & C & \text{クラス定義 } C \text{ に対応する型} \\
 & | & T[\,] & \text{配列型 } (T \text{ は要素の型}) \\
 & | & G\langle T \rangle & \text{Generics 型} \\
 & | & \cdots &
\end{array}
$$

以下の問題を考える:

- $A <: B$ ならば $A[\,] <: B[\,]$ か？
- $A <: B$ ならば $G\langle A \rangle <: G\langle B \rangle$ か？

この問題はやや難しいが、多相型システムの応用として考えてみよう。

Java の Generics(パラメータ多相型)

```
class MyList<X> {
  X head;
  MyList<X> tail;
  public MyList(X h, MyList<X> t){
    head = h; tail = t;
  }
  public int length () {
    if (tail == null) return 1;
    else return tail.length() + 1;
  }
}
```

上記の MyList を使うとき、X を String などに具体化する。

```
MyList<Integer> il1 = new MyList<>(7, null);
MyList<Integer> il2 = new MyList<>(15, il1);
MyList<String>  sl  = new MyList<>("plm",null);
```

Generics とサブタイプ多相 (1/3)

問題: A <: B ならば $G\langle A\rangle$ <: $G\langle B\rangle$ か？

- 初期の Java では、当然そうなると思われていた。
- しかし，$G\langle T\rangle$ への「書き込み」をすると、型の整合性が崩れる。

`Number <: Object` だが、
`ArrayList<Number> <: ArrayList<Object>`でない。

```
ArrayList<Number> x1 = new ArrayList<>();
ArrayList<Object> y1 = new ArrayList<>();
y1 = x1; // This is problematic!
y1.add("abc");  // y1 and x1 are ["abc"]
x1.add(7); // y1 and x1 are ["abc",7]
x1.get(0); // ==> "abc"
```

`ArrayList<Number>`型の第 0 要素が`"abc"`になってしまった.

Generics とサブタイプ多相 (2/3)

Java の解決策:

- Generics 型は、サブタイプ関係を保存しないこととした。
  - 「$A <: B$ ならば $G\langle A\rangle <: G\langle B\rangle$」は成立しない。(invariant)
- このままでは柔軟性が低い
  - リスト・配列・スタックなどの「入れ物 (container)」データ型を定義するとき，Generics とサブタイプが使えないと，プログラミングが非常にやりにくい.
- wildcard を使った Generics 型を導入。
  - $G\langle ?\ \text{extends}\ B\rangle$ により、「$A <: B$ ならば $G\langle A\rangle <: G\langle B\rangle$」が成立。(covariant 宣言; 次のページで説明)
  - (参考) この逆の contravariant 宣言もある: $G\langle ?\ \text{super}\ B\rangle$
- 現実的な妥協; 互換性のため、配列型は上記の修正をしなかった⇒実行時の型検査が必要

Generics とサブタイプ多相 (3/3)

Generics クラスを使う側で以下の指定をする (covariant にしたい場合):

- 型パラメータを `<? extends T>` と記述.
- この Generics クラスに対して「読み出し」のみ可能。
- この Generics クラスに対して「書き込み」するとコンパイル・エラー。

例:

```
ArrayList<Number> x2 = new ArrayList<>();
ArrayList<? extends Object> y2 = new ArrayList<>();
x2.add(10);
y2 = x2;
y2.get(0);   // 読み出し操作は許される
// y2.add("abc"); // 書き込み操作は許されない
```

cf. OCaml では、Generics 相当のデータ型を「定義」する側で, invariant/covariant 等を判定する.

目次

OCaml 言語のオブジェクト指向の型システム

OCaml のオブジェクトシステムの型は独特:

- サブタイプ関係の定義
  - Java は、継承したらサブタイプになる。(nominal subtyping)
  - OCaml は、継承しなくてもインタフェースの中身によりサブタイプ関係になる。(structural subtyping)
- 多相
  - サブタイプ多相はなく，明示的に型変換する: `(p1 :> Point)`
  - Row Polymorphism がある。「情報ロス」が起きない点でサブタイプ多相より優れている。

最近の言語では上記の機構が取り入れられることもある。(例. TypeScript の structural subtyping)

OCaml の Structural Subtyping (構造的サブタイプ) (1/2)

```
class point (x0 : int) =
  object (self : 'a)
    val x = x0   method get_x = x
    method equal (p : 'a) = (p#get_x = x)
  end;;
class cpoint (x0 : int) (c0 : string) =
  object (self : 'a)
    inherit point
    val c = c0   method get_c = c
  end;;
```

サブタイプ関係は、継承に関係なくインタフェースの比較で決まる:

- `cpoint = <get_x:int, get_c:string, equal:cpoint->bool>`
- `point = <get_x:int, equal:point->bool>`
- 上記の定義で `equal` メソッドがなければ `cpoint <: point`
- `equal` メソッドがあれば `cpoint <: point` でない。

OCaml の Structural Subtyping (構造的サブタイプ) (2/2)

OCaml では、「型構成子とサブタイプ」の関係はシンプルに定まる。

$$
\begin{array}{rl}
\text{直積型} & (A_1 <: B_1) \wedge (A_2 <: B_2) \Rightarrow (A_1 * A_2) <: (B_1 * B_2) \\
\text{リスト型} & (A_1 <: B_1) \Rightarrow A_1\ \texttt{list} <: B_1\ \texttt{list} \\
\text{関数型} & (B_1 <: A_1) \wedge (A_2 <: B_2) \Rightarrow (A_1 \to A_2) <: (B_1 \to B_2)
\end{array}
$$

関数型の左引数のみ反転していることに注意.

書き込み操作ができるデータ型は invariant:
参照型 $(A_1 <: B_1) \Rightarrow (A_1\ \texttt{ref} <: B_1\ \texttt{ref})$ は不成立

Row Polymorphism の例

```
let p1 =
  object (self : 'a)
  method name = "Tsukuba Taro"
  method height = 180
  end ;;

(* no loss of information *)
let f x = (x, x#height) ;;
f p1 ;;
```

サブタイピング多相では、以下の型付け:

```
f : <height : int> → <height : int> * int
f p1 : <height : int> * int
```

Row 多相では、以下の型付け:

```
f : (< height : 'b; .. > as 'a) → 'a * 'b
f p1 : < height : int; name : string > * int
```

目次

オブジェクト指向の型システム

特徴

- 型の整合性=通常の「式」の型の整合性以外に、インタフェースと実装の整合性
- サブタイプ=代入可能
- サブタイプ多相
- サブタイプがあるために型システムは複雑になる。

オブジェクト指向は動的な要素を多く持つため動的型付けの言語が多いが、最近では、静的型付けの Java, Scala, TypeScript などの言語が多く使われるようになっている。
これらをきちんと理解するためには、型システムを理解する必要がある。

調査課題 (発展課題)

以下のいずれかの課題について、A4 サイズ 1 ページ程度でまとめなさい。

- Java 言語における covariance, contravariance, invariance について調査し、それぞれの例を 1 つずつあげた上で、どのような場合に使うべきか用途を説明しなさい。
- 構造的サブタイプ (structural subtyping) について調査し、Java 言語における「継承すればサブタイプになる」方式 (nominal subtyping) と比較して、利点と欠点について論じなさい。
- OCaml の Row Polymorphism について調査をして、Java などにおけるサブタイプ多相と比較して、利点と欠点について論じなさい。