プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 6: 型システムその 1 (基本)

型 (Type) とは?

「型」は「データの集合」の一種ではあるが、データの集合がすべて型になるとは限らない。

- コンピュータ (ハードウェア) で扱うことのできるデータの種類 のこと。
- 同じ演算が適用できるデータの集まり。

型システム： どのようなプログラムにどのような型がつくか、定めるための体系。プログラム言語の設計における最重要要素の 1 つ。
静的な型システム: 型がつくかどうかを、静的に (プログラム実行前に) チェックして、型がつくプログラムのみをコンパイル・実行する。
動的な型システム: 型がつくかどうかを、動的に (プログラム実行時に) チェックして、型がつかないことがわかったら実行時エラーとする。

microML と OCaml の違い

例 (OCaml の構文で記述する)：

```
1 + true
10 (fun x → x + 1)
fun x → x x
```

これらは、microML では書けるが、1-2 個目は実行時エラー、3 個目は後で使おうとするとエラーになる。

OCaml ではそもそも書けない (コンパイルエラー)。

→ OCaml の方がうれしい。(cf. ソフトウェア工学の大原則; 不具合は、なるべく上流で発見したい)

Type System

B. Pierce, “Types and Programming Languages”, MIT Press, 2002. (型理論の著名な教科書)

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

- プログラムの (良くない) 振舞いが「ない」ことを証明する。
- 扱い可能な (現実的な) 構文的な方法
- (プログラムの) フレーズを分類する
- (フレーズたちが、それぞれ) 計算する値の種類

具体的な型

基本型 (atomic type)

- 例: `int`, `bool`, `string`, ...

複合的な型: 既にある型と型構成子 (type constructor) を使って構成。

- 例
  - C 言語: 構造体 (strucut)、共用 (union)、ポインタ、関数など。
  - Java 言語: クラス (オブジェクトの型) など。
  - ML 言語: 直積、レコード (record)、バリアント (variant), 参照、関数、リスト、再帰的な型など。
  - Lisp/Scheme 言語: S 式 (S expression)

C の型の例

C 言語でも、型は結構複雑：

```
void foo (int *p, int *q) {
   while (*p) {
     *(q++) = *(p++);
   }
}
```

- void, int などの基本型
- * という型構成子 (便宜上 Ptr(·) と書くことにする)
- 引数の `int *p` は、「p が Ptr(int) 型である」ことを意味する。
- p++の型は Ptr(int) で、`*(p++)` の型は int である。
- 関数 foo の型は、 `(Ptr(int) * Ptr(int)) -> void`

C 言語は、コンパイル時に型検査を行なう。

- たとえば `q++ = *p++;` はエラー

OCaml の型 (1)

1. 基本型

```
10    : int
true  : bool
"abc" : string
let x = 10 in x * 2 + 3 : int
(fun x → x * 3 = 10) (5 + 1) : bool
```

2. 直積型 (組 tuple が持つ型)、リスト型

```
(10, 20, "abc") : int * int * string
[10; 20; 30] : int list
[[10; 20]; [30; 40; 50]] : (int list) list
```


OCaml の型 (2)

3. 関数型

```
fun x → x + 1 : int → int
fun x → (List.hd x) + 1 : (int list) → int
fun x → ((List.hd x) + 1) :: (List.tl x) : (int list) →
    (int list)
fun x → (x+1, x−1) : int → (int * int)
```

高階関数

```
fun f → (f 10) * 5  : (int → int) → int
fun x → (fun y → x + y) : int → (int → int)
fun f → (fun x → f ((f x) + 3)) : (int → int) → (int
   → int)
```

OCaml の型 (3)

4. バリアント型、代数データ型

```
type weekday = Monday | Tuesday | Wednesday | ...
Tuesday : weekday

type expr = CstI of int | Prim of string * expr * expr
CstI(3) : expr
Prim("+", CstI(3),CstI (4)) : expr
Prim("+", Prim("*",CstI(3),CstI(4)),CstI(5)) : expr
```

代数データ型は、バリアント型と再帰型を含む。
発展課題: GADT (一般化された代数データ型) とは何か調べよ。(OCaml あるいは Haskell の文脈で)

型の例

Sort 関数 (第一版; 特定の型の要素からなるリストを整列する):

```
Sort_int   :  int list  ->  int list
Sort_float :  float list  ->  float list
```

Sort 関数 (第二版; 任意の型のリストを整列する):

```
Sort :  ∀α.  (α list  ->  α list)
```

Sort 関数 (第三版; 要素間の比較関数もパラメータにとる):

```
Sort :  ∀α ((α * α -> bool) -> (α list  ->  α list))
Sort (<) [10; 30; 20] = [10; 20; 30]
Sort (>) [10; 30; 20] = [30; 20; 10]
```

型システムと型推論の詳細は、「計算論理学」の授業の資料を参照。

実行時 vs コンパイル時

式 (1+"abc") が「いつ」エラーになるか。

- MiniC や MiniML では、実行時に (動的に) エラーになる。
- C や OCaml では、コンパイル時に (静的に) エラーになる。

エラーは静的に見つかる方がよい。

- 早い段階でエラーが見つかる。
- (実行に時間がかかる場合)、速く見つかる。

静的な型システム

式 (1+"abc") の整合性に関するエラーを、静的に発見したい。

- 式に「型」(type) を付け、その型を追うことによって整合性を検査する。
- 式の種類ごとに、どのような型が付くかを決めたものを「型システム」という。
- 型の整合性を検査するだけで、多くのバグを発見できる。
- 静的に検査ができていたら，実行時には検査は不要⇒実行時の効率が良くなる.

型システムの健全性 (Type Soundness):

- コンパイル時 (静的) に，型が整合したら，実行時の型の不整合 (実行時のエラー) は決して起きない。

型検査と型推論

型検査: 全ての変数 (や関数) の型が宣言されている言語で，型の整合性を検査すること．

- C 言語や Java 言語.
- 型検査は、変数や定数などのアトミックな式からはじめて、より大きな式の型が整合しているか検査する、という形式で行われる。

型推論: 変数 (や関数) の型が必ずしも宣言されていない言語で，その型を推論しつつ，型の整合性を検査すること．

- ML 言語や Haskell 言語.
- ML 言語では、「与えられた式に対して、最も一般的な型を推論する」という型推論アルゴリズムあり.

動的な型付け

- Lisp, Scheme, Ruby など。
- 実行時に型検査を行う。(`((lambda (x) (+ "abc" 100))` はエラーでない)
- 実行効率と，プログラムの理解のしやすさの観点からは、静的型システムに比べて不利。
- 静的な型システムで記述できないような、柔軟なプログラミングができる可能性がある。

プログラム言語の型システム

	C/C++	Lisp	ML,Haskell	Java	Ruby,JavaScript
静的/動的	静的	動的	静的	静的	動的
検査/推論	型検査	—	型推論	型検査	—

- 静的型システムと動的型システム
  - 静的:実行前に型の整合性を検査/推論。
  - 動的:実行時に行う。
- 型検査/型推論
  - 型検査: 変数の型は宣言済み⇒プログラムの型の整合性を検査。
  - 型推論: 変数の型が未知⇒推論しつつプログラムの型の整合性を検査。

多相型 (Polymorphism)

```
void swap (int *p, int *q) {
  int r;
  r = *p; *p = *q; *q = r;
}
```

swap 関数は (int *) 型だけでなく、どんな型でも使える。

```
void swap (T *p, T *q) {
  T r;
  r = *p; *p = *q; *q = r;
}
for any T.

# let swap (x,y) = (y,x) ;;
  - : 'a * 'b -> 'b * 'a = <fun>
```

多相型＝「任意の型」を含む型。

ML 言語の多相型 (polymorphic type)

map の型:

$(\alpha \to \beta) \to (\alpha$ list $\to \beta$ list$)$

```
let inc x = x + 1;;
map inc [1; 2; 3];;
=> [2; 3; 4]
```

(map の型は、(int -> int) -> (int list -> int list))

```
let add1 x = x ^ "1";;
map add1 ["kameyama"; "yukiyoshi"];;
=> ["kameyama1"; "yukiyoshi1"]
```

(map の型は、
(string -> string) -> (string list -> string list))

ML 言語の多相型 (polymorphic type)

ユーザ定義関数における多相型; let で導入される。

```
let f (x,y) = (y,x);;
==> f : α * β → β * α
let g x = x in ((g 10), (g "abc"))
==> g : α → α
```

ちなみに、以下の式は ML では、多相型と見なされない。

```
(fun h →  ((h 10), (h "abc"))) (fun x → x)
==> type error
```


ML 言語の多相型 (polymorphic type)

ユーザ定義関数における多相型; let で導入される。

```
let g f = f f;;
==> type error

let f x = x in
   (f f) 10 ;;
==> OK
```

f の型

- let で定義した f ... 'a -> 'a
- 1 つ目の f ... (int -> int) -> (int -> int)
- 2 つ目の f ... int -> int

多相型の利点

もし、map 関数を C 言語で書くとしたら‥。

- 方法 1. int 型に対する map, string 型に対する map などを別々に定義する。
- 方法 2. 「void 型に対する map」を定義して、使うときに各型に cast する。
- 方法 3. C++ の template を使う。「T 型に対する map」を定義して、この関数を使うときに T を具体化する。

方法 1 は、コード量が多くなる、同じコードを何度も書くため保守性が悪い、等のデメリットがある。
方法 2 は、型の検査を素通りするため、型に関する間違いのチェックができなくなる等のデメリットがある。
方法 3 は、多相型と基本的に同じ効用がある。ただし、C/C++言語自体に組み込まれた機能ではないので、型エラーが起きたときに原因となるコードを発見しにくい等のデメリットがある。

多相型の利点

- 前のスライドで挙げた問題点がない。

今回学んだ多相型は、parametric polymorphism と呼ばれるもの。ML 言語のほか、Haskell などの関数型言語で利用可能。
cf. オブジェクト指向言語の subtyping polymorphism、Haskell を含む一部の言語の ad hoc polymorphism.

まとめ

型が整合していたからといって、プログラムが「正しい」わけではない。しかし、型が整合しているかどうかの検査をやるだけでも、(人間がよくやる) 多くの間違いを早期に発見できることが多い。[経験的事実]

- 型システム
- 静的型付け vs 動的型付け
- 多相型

積み残し: オブジェクト指向言語 (特に Java) の型システム (Generics, subtyping polymorphism, ...) →次回か次々回

議論

動的型付けの方が静的型付け有利であるという主張がある。どういう点で動的型付けの方が有利なことがあり得るか、想像して書いてください。

解答の一例：より柔軟なプログラミングを行なう場合に有利なことがある。

- 型が「邪魔」になる場合、たとえば、
  - 「異なる型の要素を並べたリスト」を処理したい。
  - 「外部からやってくる (型のわからない) データ」を処理したい。
- 拡張性・再利用性
  - 将来、(設計時には思ってもいなかった) 拡張を行なう際の容易さ。
  - ある型のプログラムの一部を、別の型で使いたい。
- メタ・プログラミング (プログラムを生成するプログラム作り)

動的型のメリットについての参考図書: 「コードの世界」、まつもとゆきひろ (Ruby の設計者)、日経 BP 社, 2009 年。