プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 6b (型システム)

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。

MiniML と OCaml の差

- MiniML/OCaml 言語では、int や bool の定数のほかに「関数」をあらわすデータがある。このような関数の型は、どうなるであろうか？
- ML 言語のプログラム中には、int や bool といった型名を書かないが、そんなことで大丈夫だろうか？
- OCaml では (100+"abc") などの式は，コンパイル時にエラーとなる．(MiniML では実行時にエラーになる) この仕組みは？

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。
  - 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。

型の例 (関数の型)

```
(fun x→ x+1) :  int → int
(fun f→(fun x→ f(f(x+1)))) :  (int→ int)→(int→ int)
(fun x→ x) :  'a → 'a
(fun f → (fun x→ f (f x))) :  ('a → 'a) → ('a → 'a)
```

型推論の詳細は、3 学期「計算論理」の授業を参照。

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: `('a -> 'b) -> ('a list -> 'b 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 : 'a * 'b -> 'b * 'a

let g x = x in ((g 10), (g "abc"))
==> g : 'a -> 'a
```

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
```


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。

まとめ

- 型システム
- 静的型付け vs 動的型付け
- 多相型

宿題: 以下の関数を型付けしてみよ。(まず，自分の頭で型付けしたあと，OCaml 処理系で型を教えてもらうのがよい．)

```
let f x y = x (x (y+10));;
let g x = (fst x) + (snd x) + 10;;
let i_comb x = x ;;
let k_comb x y = x ;;
let s_comb x y z = (x z) (y z) ;;
let b_comb x y z = x (y z) ;;
```