プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 7: 多相型 (1)

目次

OCaml での型推論

OCaml 処理系は、式の実行前に型推論を行い、成功したら (型が付く式であれば)、実行結果と型を表示する。

```
# let g x y z = (x z) (y z) ;;
val g : ('a → 'b → 'c) → ('a → 'b) → 'a → 'c = <fun>
```

上記の結果をよく見ると:

- 計算結果: 関数の中身は示さず、`<fun>` と表示
- 型 `T1 -> T2 -> T3` は `T1 -> (T2 -> T3)` であるので、上記の型は、$(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$ を表す。

疑問: この型は何だろうか？

単純型付きラムダ計算の限界

先週の体系 (単純型付きラムダ計算) では、

- すべての型 α に対して $\vdash \lambda x.\ x : \alpha \to \alpha$ を導出できる。たとえば、
  - $\vdash (\lambda x.\ x)\ 3 : \texttt{int}$
  - $\vdash (\lambda x.\ x)\ \texttt{true}) : \texttt{bool}$
- しかし、1 つの式の中で複数の型で使用できない。
  - $\vdash (\lambda f.\ (f\ 3,\ f\ \texttt{true}))(\lambda x.\ x)$ は型がつかない。

一方、OCaml では以下の式に型が付く。

```
let f x = x in
  (f 3, f true, f 3.14, f (fun y → y*2)) ;;
```

⇒ OCaml は、単純型付きラムダ計算より強力な型システムを採用

多相型 (polymorphic type)

多相型＝1つの式が複数の型を持つことができる仕組み

以下の3種類に大別

- パラメータ多相: 最も基本的な多相
- サブタイピング多相: オブジェクト指向言語など
- アドホック多相

目次

パラメータ多相 (parametric polymorphism)

パラメータ多相 = 型パラメータ (型を表す引数) を持つ

- 型変数 $(\alpha, \beta, \cdots)$ と $\forall$ (「任意の型に対して」) を導入

$$T ::= \texttt{int} \mid T \to T \mid \alpha \mid \forall\alpha.\ T$$

例:

- $\vdash \lambda x.\ x \ : \ \forall\alpha.\ (\alpha \to \alpha)$
- $\vdash \lambda f.\ (\lambda x.\ (f\ x)) \ : \ \forall\alpha.\ \forall\beta.\ ((\alpha \to \beta) \to (\alpha \to \beta))$

パラメータ多相の例 (1)

関数 `id` の定義:

```
let id x = x ;;
```

定義時の `id` の型: $\forall\alpha.(\alpha \rightarrow \alpha)$

関数 `id` の使用:

```
id 35 ;;             (* α = int *)
id "tsukuba" ;;      (* α = string *)
id [10;20;30] ;;     (* α = int list *)
id (fun x→x+1) ;;    (* α = int → int *)
(id id) 5 ;;         (* 1st id: α = int → int *)
                     (* 2nd id: α = int *)
```

パラメータ多相の例 (2)

関数 `double`:

```
let double f x = f (f x) ;;
let add1 x = x + 1 ;;
let addstr x = x ^ "cd" ;;
(double add1) 35 ;;
(double addstr) "ab" ;;
(double id) add1  ;;
```

`double` の型:

$$\forall\alpha.((\alpha \to \alpha) \to (\alpha \to \alpha))$$

上記の 3 例では、α をそれぞれ `int`, `string`, `int` $\to$ `int` にしている。

パラメータ多相の例 (3)

関数 **append** (2 つのリストの結合):

```
open List ;;
append [1; 2; 3] [4; 5] ;;
append ["ab"; "cde"] ["f"; "gh"; "ijk"] ;;
append [add1; add1; id] [id; add1] ;;
```

append の型:

$$\forall\alpha.(\alpha\ \texttt{list} \to (\alpha\ \texttt{list} \to \alpha\ \texttt{list}))$$

関数 **length** (リストの長さ):

```
length [1; 2; 3] ;;
length (append ["ab"; "cde"] ["f"; "gh"; "ijk"]);;
```

length の型:

$$\forall\alpha.(\alpha\ \texttt{list} \to \texttt{int})$$

目次

ML 系言語における多相型

制限されたパラメータ多相 (let 多相) を採用

- 多相型が導入されるのは let 式のみ

```
let f x = x in f (5 = f 3) ;;  (* OK *)
(fun f → f (5 = f 3)) (fun x → x) ;;  (* エラー *)
```

- ∀は「型の一番外側」のみに出現

```
let foo f = (f [2]) + (f [3.14]) ;;  (* エラー *)
```

型 $(\forall\alpha.(\alpha\ \texttt{list} \to \texttt{int})) \to \texttt{int}$ には、∀が内側に出現

- `let x = e1 in e2` は、`e1` が値 (関数や定数など) ならば多相型、値でなければ単相型

```
let x = foo 10 in ... ;;    (* x は単相型 *)
let f x = foo 10 in ... ;;  (* f は多相型 *)
```

let 多相の型システム

- 型の構文は 2 層から構成される。
  - 単相型: $T ::= \texttt{int} \mid T \to T \mid \alpha$　(ただし、α は型変数)
  - 多相型: $P ::= T \mid \forall\alpha.P$
- 型判断:　$(x_1 : P_1), \cdots, (x_n : P_n) \vdash e : T$
- 型付け規則 (単純型付きラムダ計算と異なる規則のみ):

$$\frac{\Gamma \vdash v : T_1 \quad \Gamma,\ (x : \forall\alpha_1.\cdots\forall\alpha_n.T_1) \vdash e : T_2}{\Gamma \vdash \texttt{let}\ x = v\ \texttt{in}\ e : T_2}$$

e は (制限のない) 式、v は値に限定 (関数・定数等)

$$\frac{(\texttt{typeof}_x(\Gamma) = \forall\alpha_1.\cdots\forall\alpha_n.T)}{\Gamma \vdash x : T\{\alpha_1 := T_1, \cdots, \alpha_n := T_n\}}$$

let 多相の型付け例 (1)

$\Gamma_1 = f : \forall\alpha.(\alpha \to \alpha)$ とする。

$$\dfrac{\dfrac{\overline{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.\ x : \alpha \to \alpha} \quad \dfrac{\color{red}{\overline{\Gamma_1 \vdash f : \mathtt{int} \to \mathtt{int}}} \quad \overline{\Gamma_1 \vdash 3 : \mathtt{int}}}{\Gamma_1 \vdash f\ 3 : \mathtt{int}}}{\vdash \mathtt{let}\ f = \lambda x.\ x\ \mathtt{in}\ f\ 3 : \mathtt{int}}$$

$$\dfrac{\dfrac{\overline{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.\ x : \alpha \to \alpha} \quad \dfrac{\color{red}{\overline{\Gamma_1 \vdash f : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})}} \quad \dfrac{\overline{\Gamma_1,\ y : \mathtt{int} \vdash y : \mathtt{int}}}{\Gamma_1 \vdash \lambda y.\ y : \mathtt{int} \to \mathtt{int}}}{\Gamma_1 \vdash f\ (\lambda y.\ y) : \mathtt{int} \to \mathtt{int}}}{\vdash \mathtt{let}\ f = \lambda x.\ x\ \mathtt{in}\ f\ (\lambda y.\ y) : \mathtt{int} \to \mathtt{int}}$$

赤字の 2 か所で、$\alpha := \mathtt{int}$ および $\alpha := \mathtt{int} \to \mathtt{int}$ と具体化

let 多相の型付け例 (2)

$\Gamma_2 = f : \forall\alpha.(\alpha \to \alpha)$ および、$II = \texttt{int} \to \texttt{int}$ とする。

$$
\dfrac{\dfrac{}{x:\alpha \vdash x:\alpha}}{\vdash \lambda x.\ x : \alpha \to \alpha}
\qquad
\dfrac{\dfrac{\dfrac{}{\color{red}{\Gamma_2 \vdash f : II \to II}} \quad \dfrac{}{\color{red}{\Gamma_2 \vdash f : II}}}{\Gamma_2 \vdash f\ f : II} \quad \dfrac{}{\Gamma_2 \vdash 3 : \texttt{int}}}{\Gamma_2 \vdash (f\ f)\ 3 : \texttt{int}}
$$
$$
\overline{\vdash \texttt{let}\ f = \lambda x.\ x\ \texttt{in}\ (f\ f)\ 3 : \texttt{int}}
$$

赤字の 2 か所で、$\alpha := II$ および $\alpha := \texttt{int}$ と具体化。

let 多相と OCaml

OCaml の型推論では、∀ を省略する。

```
# let g x y z = (x z) (y z) ;;
val g : ('a → 'b → 'c) → ('a → 'b) → 'a → 'c = <fun>
```

これは、g を使用するときは、以下の型付けであることをを表す。
(α, β, γ の順番は気にしない。)

$$g : \forall\alpha.\forall\beta.\forall\gamma.((\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)))$$

なぜ、制限するのか？

疑問: 一般的なパラメータ多相の体系は、すっきりとした体系なのに、ML 系言語は、なぜ、わざわざ、面倒な制限のはいった体系を採用したのか？

なぜ、制限するのか？

疑問: 一般的なパラメータ多相の体系は、すっきりとした体系なのに、ML 系言語は、なぜ、わざわざ、面倒な制限のはいった体系を採用したのか？
答: 型推論を可能とするため。

- 制限のない多相型 (System F)
- Rank-N 多相... $N \geq 3$ ならば型推論問題が決定不能
- Rank-1 多相 (let 多相)... 型推論問題が決定可能

Hindley-Milner 型システム

Hindley[1969] と Milner[1978] による型システム:

- let 多相をもつ型システム (Hindley,Milner)
- 効率良い型推論アルゴリズム W を提案 (Damas,Milner)
- 最初の ML 言語 (Standard ML) を設計 (Milner)

性質:

- Principal Type (主たる型); Γ と M に対して、$\Gamma \vdash M : T$ となる型 T の中で最も一般的な型
- 型推論問題; Γ と M に対して、$\Gamma \vdash M : T$ となる型 T の中で最も一般的な型を求める問題
- Hindley-Milner 型システムに対する型推論問題は決定可能 (アルゴリズム W)

Hindley-Milner 多相の利点

let 多相は、プログラムを書きやすくし、コード量削減にも役立つ。

- 多相関数の実装 (コード) は 1 つだけ
- 多相関数を使う時は、様々な型のデータに適用できる

目次

C++テンプレート

C++言語の template は、パラメータ多相による多相型を表す。

```
T add (T x, T y) {
  return x + y;
}
void foo () {
  float f  = add<float>(3.14, 2.78);
  string s = add<string>("abc", "def");
}
```

C++は、真のパラメータ多相でないとの見方もある。

- 型パラメータ T を具体化するたびに、コードが複製される。
- 真のパラメータ多相=1 つのコードが異なる型に対して使われる。

オブジェクト指向言語の Generics

オブジェクト指向言語では、「サブタイピング多相」を「多相」と呼び、「パラメータ多相」を「Generics」と呼ぶことが多い。

Java 言語の例 [Igarashi 2004]

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

$\forall X.$ `List<X>`という多相型を表す。(`X` が型変数 (型パラメータ))。

Hindley-Milner を超えて: Rank-2 多相

compare: 与えられたリストの「サイズ」(長さなど) を計算する関数 f と、2 つのリストをもらって、それらのサイズを比較する関数

```
let compare f lst1 lst2 =
   (f lst1) = (f lst2) ;;
compare List.length ["a"; "b"] [3; 5];; (* エラー *)
```

compare に付いてほしい型:

$$(\forall\alpha.(\alpha \to \texttt{int})) \to \beta\ \texttt{list} \to \gamma\ \texttt{list} \to \texttt{bool}$$

OCaml で comp に付く型:

```
('a → 'b) → 'a → 'a → bool
```

これは、$\forall\alpha.\forall\beta.((\alpha \to \beta) \to \alpha \to \alpha \to \texttt{bool})$ という型が付くことを意味する。

Rank-N 多相

Rank-N 型の N: ∀ の出現位置が、どのくらい「型の内側」に来るか。

- $N = 1$ なら、∀ が内側に来ない。
- $N \geq 3$ の場合、型推論ができない。
- $N \geq 3$ を許す言語では、式の中に型を明示する。

OCaml における Rank-2 多相 (レコード型の拡張を使う):

```
type packed = {cont: 'a. 'a list → int};;
let compare f lst1 lst2 =
  (f.cont lst1) = (f.cont lst2);;
compare {cont=List.length} ["a"; "b"] [3; 5];; (* 成功 *)
```

Haskell における Rank-N 型:

```
comp :: (forall a. ([a] -> Int)) -> [b] -> [c] -> Bool
comp f lst1 lst2 = (f lst1) == (f lst2)
```

まとめ

多相型＝1つの式が複数の型を持つことができる型

パラメータ多相について:

- 型変数 (型パラメータ) を持つことにより複数の型を表現
- 型システムの表現力を高めたり、コード量削減に有益
- ML, Haskell, Java など多くの言語で利用
- Hindley-Milner 型システム: let 多相に制限 $\Rightarrow$ 型推論が可能
- let 多相より強力な多相は、プログラム中に型を明示する必要

目次

演習問題 (1)

OCaml 言語で以下の多相関数を実装し、実行例 (異なる型への具体化 5 個以上) を示しなさい。

1-1. 「リストのリスト」をもらって、それを「つぶしたリスト」を返す。

```
f1 : ('a list) list → 'a list
f1 [[10;20]; [30;40;50]];; => [10;20;30;40;50]
f1 [["a";"bcd"]; ["ef"]];; => ["a";"bcd";"ef"]
```

1-2. 「関数」と「非負の整数 n」と「初期値」をもらい、その関数を初期値に n 回繰返し適用した結果を返す。

```
f3 : ('a → 'a) → (int → ('a → 'a))
f3 (fun x → x * 2) 5 3;; => 96
f3 (fun x → x +. 2.0) 5 3.0;; => 13.0
```

演習問題 (2)

2-1. 第 2 回演習で実装した整列 (ソート) 関数 `sort` は、

```
sort : int list → int list
```

という型を持つ。これを一般化して、「大小比較をする関数」と「リスト」をもらって、「その順序で整列したリスト」を返しなさい。

```
sort2 : (int → int → bool) → (int list → int list)
sort2 (fun x y → x < y) [10;3;6];; => [3;6;10]
sort2 (fun x y → x > y) [10;3;6];; => [10;6;3]
```

演習問題 (2) つづき

2-2. 前問をさらに一般化して、以下の関数を作成しなさい。

```
sort3 : ('a → 'a → bool) → ('a list → 'a list)
sort3 (fun x y → x < y) [10;3;−6];; ==> [−6;3;10]
let strcomp s1 s2 = (String.length s1) < (String.length
    s2);;
sort3 strcomp ["a3";"x";"39a"] ;; => ["x";"a3";"39a"]
```

演習問題 (3: 発展課題)

3. 以下の型判断に対する型付け図を書きなさい。ただし、T は適当な型で置き換えなさい。

$$
\begin{aligned}
\vdash & \texttt{let } s = \lambda x.\ \lambda y.\ \lambda z.\ (x\ z)(y\ z) \texttt{ in} \\
& \texttt{let } k = \lambda x.\ \lambda y.\ x \texttt{ in} \\
& \texttt{let } f = \lambda x.\ ((s\ k)\ k)\ x \texttt{ in} \\
& \quad (f\ f)\ 7 : T
\end{aligned}
$$

付録: OCaml の List モジュール

以下の通り、多相関数群が提供されている。

```
length : 'a list → int   (* リストの長さ *)
rev    : 'a list → 'a list   (* リストの反転 *)
append : 'a list → 'a list → 'a list (* リストの連結 *)
map    : ('a → 'b) → 'a list → 'b list
fold_left  : ('a → 'b → 'a) → 'a → 'b list → 'a
fold_right : ('a → 'b → 'b) → 'a list → 'b → 'b
```

利用する際には、**List.length** とするか、**open List ;;** とやってから **length** を呼ぶ。
詳細は「OCaml List module」で検索。