ソフトウェア技法: No.5 (型システム)

亀山幸義 (筑波大学情報科学類)

1 型システム

現代の多くのプログラム言語にとって、型 (type) は極めて重要である。プログラムの型の整合性を検査することにより、プログラムの多くのバグが取れるし、プログラムに型が書いてあると、変数や関数の意味がわかりやすくなるし、(クラスやモジュールの) インタフェースの記述においても、型は有用である。ここで、型の整合性とは、たとえば、a+b という式では、(1) a と b がともに整数型であること、(2) a+b という式全体は整数型として (まわりの文脈から) 利用されていること等[*1]を意味する。

OCaml, C, Java などのプログラム言語は、静的な型付けを行なう。これは、プログラムを実行する前に (コンパイル時などに) プログラムの型の整合性をチェックし、エラーがあれば、実行しないというものである。たとえば、if true then 3 else 2+"abc" という式は、潜在的なエラー原因となり得る 2+"abc" を含むので、型エラーと見なして、コンパイル時に検出する。このように「部分式は、実行されるかどうかにかかわらず、すべてチェックする」のが静的な型付けである。

静的な型付けと対照的なのは、動的型付けであり、Ruby, JavaScript, Python, Perl, Scheme などに見られる。これは、実行前の型検査は行わず、プログラム実行中に型の整合性を検査するものである。たとえば a+b という式を実行する直前に、a と b (を計算した結果) が数であることをチェックする。したがって、if true then 3 else 2 + "abc" という式は、else 節の実行をしないので、動的型付けの言語ではエラーとはならない。

静的な型付けは、実行時の型エラーがないことを保証する[*2]ため、動的な型付けにくらべて、信頼性、安全性などが高いとされている。一方で、プログラマは、プログラム作成段階では、多くの型エラーと格闘することになる。また、プログラムの一部が未完成のままテストをする、ということも許されない。そこでプログラミングにあたっては、型システムの基本の理解が必須である。

この章では、OCaml の型システムの基本事項をながめてみよう。

1.1 型検査と型推論

具体的な話にはいる前に、型検査と型推論という 2 つの言葉をおさえておこう。

型検査は、C や Java のように、すべての変数の型がプログラムに書かれており、型が整合しているかどうかのチェックだけをするものである。

一方、型推論は、OCaml のように、変数などの型はプログラムに書かれておらず (書かれていてもよいが、書かれていなくてもよい)、処理系としては、わからない型を推論しながら整合性のチェックもする必要がある。ここで、処理系の気持になって考えると、`fun x -> x + 5` というプログラムに対する型推論プログラムは、変数 `x` になんらかの型を割り当てて、プログラム全体の型が整合するようにしたい、という問題を解く。この場合、`x` を `int` 型とすると、プログラム全体は、`int->int` 型を整合的にもつことがわかる。一方、 `fun x -> x + 5.0` というプログラムは、整数型の演算と実数型の値がまざってしまっているので、`x` にどんな型を割り当てても、プログラム全体の型が整合することはない。

このように「いくつかの変数 (や部分式) の型がわからない部分に、それらにうまい型を割り当てて、プログラム全体の型が整合的になるようにできるか？」という問題が、型推論問題である。OCaml のコンパイラに含まれている型推論器 (型推論をおこなうプログラム) は、この型推論問題を解いてくれる。

1.2 式の型

さて、具体的な式の型を見てみよう。まずは、今まで何度も見てきた式の型である。

[*1] この言いまわしで、最後の「等」が気になった人もいるかもしれない。実際には、型の整合性は、本文の (1),(2) だけでなく、(3) a と b に含まれる変数や外部の関数が一貫した型をもっていること (さらに、他の部分でそれらの変数や関数が使われているとしたら、すべて一貫した型をもっていること) も必要である。

[*2] 正確には、このことはいつでも成立するのではなく、「型保存定理」という定理を証明できた言語に対してのみ成立する性質である。

```
(* foo1 : int *)
let foo1 =
    1 * 3 + 4 ;;

(* foo2 : float *)
let foo2 =
    let x = 10 in
    let y = 2.68 in
      if y > 1.3 then float (x * 3 + 4)
      else y *. 2.1 ;;

(* foo3 : float *)
let foo3 =
   let x = 384 in
   let y = 10.20 in
   if x > 0 then
     let z = true in
       float x
   else
     let w = 30 in
       y *. 3.14  ;;
```

次に、if-then-else 式では then 以下の式と else 以下の式の型は一致しなければいけない。

```
(* goo1 : int *)
let goo1 =
   let x = 10 in
   let y = 20 in
   if x > 0 then x + 1 else y * 2 ;;

(* 以下の式は型エラー *)
let _ =
   let x = 10 in
   if x > 0 then x else "unexpected input" ;;

(* 以下の式も型エラー *)
let _ =
   let x = 10 in
   if x > 0 then x + 3  ;;  (* then-part が int 型で、else-part が unit 型*)

(* goo2 : unit *)
let goo2 =
   let x = 10 in
   if x > 0 then print_int (x + 3)    ;;
```

goo2 の型である unit は、要素として () だけを持つ型である。この () は、何の使い道もなさそうな値だが、print 式など、「返す値には興味がない (副作用のみに興味がある) 式の返り値である」ことを明示するために使われる。

```
(* match 式でも、if−then−else と同様に、すべての場合において、型が一致しなければいけない。*)
(* 以下は型エラー *)
let _ =
   let x = 10 in
   match x with
   | 1 → x + 3
   | 2 → 25.0
   | _ → "予期しない入力です!" ;;

(* goo11 : int *)
let goo11 =
   let x = 1 in
   match x with
   | 1 → x + 3
   | 2 → 25
   | _ → failwith "予期しない入力です!"   ;;
```

ここで、`failwith "..."` は、どんな型の式としても使うことができる (型が整合する) 不思議な式である。

1.3 関数の型

OCaml では関数もきちんと型がつかなければいけない。X 型の引数をもらい、Y 型の返り値を返す関数の型は、`X->Y` と表示する。また、複数の引数を取る関数の場合は、それぞれの引数の型を `X1`, `X2`, ... `Xn` とし、返り値の型を `Y` とすると、`X1 -> X2 -> ... -> Xn -> Y` という型になる。これは、括弧を補うと `X1 -> (X2 -> (... -> (Xn -> Y)...))` という型のことであるが、この段階では、単純に、「n 引数の関数の型を `X1 -> X2 -> ... -> Xn -> Y` と表示する」ということだけをおさえておけばよい。

```
(* goo20 : float → (int → (string → string)) *)
(* これは、goo20 : float → int → string → string とも書かれる *)
let goo20 x y z =
  if x > float y then
       z ^ (string_of_int y)
  else z ^ (string_of_float x) ;;

(* 再帰関数の場合は、その関数の型も一貫していないといけない *)
(* mc91 : int → int *)
let rec mc91 x =
  if x > 100 then x − 10
  else mc91 (mc91 (x + 11)) ;;

(* 以下は型エラー *)
let rec goo21_buggy x =
  if x > 100 then float (x − 10)
  else goo21_buggy (goo21_buggy (x + 11)) ;;
```

OCaml の処理系は、型推論をやってくれるが、プログラムを理解しやすくするため、自分で、変数や式の型を明示してもよい。その場合は必ず、かっこをつけて （式 ： 型） の形で書く。*)

```
(* goo31 : int → int *)
let rec goo31 (x :int) =
  if x > 100 then (x − 10 : int)
  else (goo31 (goo31 (x + 11)) : int) ;;

(* goo20 : float → int → string → string *)
let goo20 (x : float) (y : int) (z : string) =
  if x > float y then
       z ^ (string_of_int y)
  else z ^ (string_of_float x) ;;

(* ただし、関数の返り値（結果）の型を明示するときは、かっこは書かない *)
let rec goo32 (x :int) : int =
  if x > 100 then x − 10
  else goo32 (goo32 (x + 11)) ;;

(* 複数の引数を持つ関数であっても、型は 1 つずつ分けて書く。*)
let goo33 (x :int) (y: int) : int =
  x * 2 + y ;;

(* goo20 : float → int → string → string *)
let goo20 (x : float) (y : int) (z : string) : string =
  if x > float y then
       z ^ (string_of_int y)
  else z ^ (string_of_float x) ;;
```

1.4 リストの型

「リスト」という型そのものはなく、必ず「Xのリスト」という形の型になり、Xには要素の型がはいる。1 つのリストの中で、int 型の要素と float 型の要素が混在することはできない。OCaml では、「X 型を要素とするリストの型」を `list(X)` ではなく、`X list` と後置記法で表す。

```
(* lst1 : int list *)
let lst1 = 1 :: 2 :: 3 :: [] ;;

(* 要素型が違えば、違う型なので、以下の式は型エラー *)
let _ = 3.14  :: lst1 ;;

(* lst2 : string list *)
let lst2 = ["I"; "am"; "a"; "student."] ;;

(* lst3 : (int list) list, この括弧は省略して int list list と書いてよい。 *)
let lst3 = [[]; lst1; 1 :: lst1; 1 :: 2:: lst1] ;;

(* リストに対する match 式も、すべての場合において同じ型の値を返さなければいけない *)
(* count_positive  : float list → int *)
let rec count_positive (lst : float list) : int =
  match lst with
  | [] →  0
  | h :: t  →  if h > 0.0 then 1 + (count_positive t)
               else count_positive t ;;
```

1.5 多相型

リストの append 関数の型を見てみよう。

```
let rec append (lst1 : int list) (lst2 : int list) : int list =
  match lst1 with
  | [] → lst2
  | h::t → h::(append t lst2) ;;

(* 処理系の出力
   val append : int list → int list → int list = <fun>
 *)
```

出力からわかるように、この append 関数は `int list` 型の要素を 2 つもらって、`int list` 型の要素を返す。ところが、上記の append 関数で、型指定をしなければ、違う型が返ってくる。

```
let rec append lst1 lst2 =
  match lst1 with
  | [] → lst2
  | h::t → h::(append t lst2) ;;

(* 処理系の出力
   val append : 'a list → 'a list → 'a list = <fun>
 *)
```

1 つ前の出力と比べると `int` のところが、`'a` に変化していることがわかる。これは一体何であろうか？

それを知るために、まず、もう 1 つ入力してみよう。

```
let rec append_str (lst1 : string list) (lst2 : string list) : string list =
  match lst1 with
  | [] → lst2
  | h::t → h::(append_str t lst2) ;;

(* 処理系の出力
   val append_str : string list → string list → string list = <fun>
*)
```

これは、関数 append の定義の本体はまったく同一だが、引数を `int list` から `string list` に変化させたものである。これでも型エラーはなく実行できる。(ただし、このように定義した `append_str` を、`int list` 型の要素に適用するとエラーである。)

これらからわかるように、append 関数 (型指定をしないもの) は、複数の型の引数を取ることができる。もっと正確にいうと、

$$X \texttt{ list -> } X \texttt{ list -> } X \texttt{ list}$$

という形の、どの型でも取ることができる。ここで X の部分は型でありさえすればなんでもよい。`int` や `string` だけではなく、`int list` でも `string list list` でもよいし、後でててくる `int array` など、何でもよい。

上記のように「どの型でもよい」という部分を含む型のパターンのことを、型スキーム (type scheme) と呼び、「どの型でもよい」という部分を表す部分を型変数と呼ぶ。

現代のプログラム言語では、1 つの関数 (あるいは式) が、複数の型で使えるようにする仕組みを持つものが多い。この仕組みを **多相型**(polymorphic type) と呼ぶ。

OCaml では、型変数の名前は、`'a` や `'b1` など、single quotation (') の後に記号列を書いて表現する。

```
append : 'a list → 'a list → 'a list
```

ここで、型変数`'a` にはどんな型を入れて使ってもよい。つまり、以下の式はすべて型が整合している。(型エラーはない。)

```
let _ = append [1; 2; 3] [4; 5] ;;
let _ = append [1.0; 2.0; 3.0] [4.0; 5.0] ;;
let _ = append [["abc"; "def"]; []; ["ghi"]] [["jkl"; "mno"]] ;;
```

型変数`'a` にはどんな型をいれてもよいが、すべての`'a` に対して同じ型をいれなくてはいけない。

```
(* 以下の式は型エラー *)
let _ = append [1; 2; 3] [4.0; 5.0] ;;
```

append 関数以外にもいろいろな関数が多相型を持つ。これまで出てきた関数の中では、List.hd, List.tl, reverse (リストの要素を逆順にしたリストを返す)、length (リストの長さを返す) は以下の多相型を持つ。

```
List.hd : 'a list → 'a

List.tl : 'a list → 'a list

length  : 'a list → int

reverse : 'a list → 'a list
```

ところで、型変数の名前の a には特に意味がなく、'a -> 'a という型と'b -> 'b という型は、実質的に同じ型を意味する。型を表す変数は、a,b,c,... という文字を適当に使っているだけである。後で出てくる List.map 関数の型スキームは、以下のように2つの型変数 'a と 'b を使う。

```
List.map : ('a → 'b) → ('a list → 'b list)
```

2 演習問題

1. 以下のプログラムの型が整合するかどうか (プログラムに型がつくかどうか)、まず自分の頭で考え、そのあと、OCaml 処理系にかけて確かめなさい。また、型がつく例については、なぜその型がついたか、また、型がつかない例については、なぜ型がつかないか、理由を考えなさい。(なお、以下の例は、関数適用の結果が停止しないものが含まれているので、実行するときには注意せよ。)

```
let rec ex0 x =
  if x = 1 then 1
  else if x mod 2 = 0 then ex0 (x / 2)
  else ex0 (x * 3 + 1) ;;

let ex1 x = [x] ;;

let rec ex2 x = ex2 (x+1) ;;

let rec ex3 x = [ex3 (ex2 x)] ;;

let ex4 x = (x,x) ;;

let ex5 x = (x*2, x=2) ;;
```

```
let rec ex6 x y =
   match x with
   | [] → []
   | h1::t1 →
     begin
       match y with
       | [] → []
       | h2::t2 →
           (h1 *. h2) :: (ex6 t1 t2)
     end ;;

let rec ex7 x y z=
   if x <= y then y
   else
     ex7 (ex7 (x−1) y z) (ex7 (y−1) z x) (ex7 (z−1) x y) ;;

(* [2017/7/26] 課題 ex8 はコピペミスだったので削除しました *)

let rec ex9 = 0 :: ex9 ;;

let rec ex10 = List.tl (List.tl (List.tl ex9)) ;;
```

2. 多相的関数について以下の設問に答えなさい。

```
(* append が多相であることを確認する *)
let rec append lst1 lst2 =
  match lst1 with
  | [] → lst2
  | h::t → h::(append t lst2) ;;

let intlist1 = [1; 2; 3] ;;
let intlist2 = [4; 5] ;;
let _ = append intlist1 intlist2 ;;
let _ = append (append intlist1 intlist2) (append intlist2 intlist1) ;;

let floatlist1 = [3.1; 1.5; 2.6] ;;
let floatlist2 = [1.2; 3.4] ;;
let _ = append floatlist1 (append floatlist2 floatlist2) ;;

let intintlist1 = [intlist1; intlist2] ;;
let intintlist2 = [intlist2; intlist2; intlist1] ;;
let _ = append intintlist1 intintlist2 ;;
```

```
(* 実は、関数を要素とするリストも作れる *)
let foo x = x;;
let goo x = x + 1;;
let hoo x = x + 2;;
let ioo x = x + 3;;
let joo x = x + 4;;
let funlist1 = [foo; goo; hoo] ;;
let funlist2 = [ioo; joo] ;;

let _ = append funlist1 funlist2 ;;
```

関数 append は 'a list -> 'a list -> 'a list という多相型を持つが、上記のそれぞれの適用例において、型変数'a が、どのような具体的な型になっているか答えなさい。

また、length や reverse 関数も同様に多相型を持つ。これらを、いろいろな型のリストに適用可能なことを確認しなさい。(3 個以上の要素型に対して試しなさい。)

3. [発展課題] 以下の関数の型が何になるか、また、どうしてそうなるか説明しなさい。(OCaml で実行すれば型はわかるが、なぜ、そのような型になるか、自分の言葉で説明しなさい。)

```
(* 多相関数の不思議な使い方 *)
let f x = x in
   (f 10, f "abc", f f) ;;

(* 高階関数; 関数 f をもらい、それを 2 回適用する *)
let ex10 f x = f (f x) ;;

(* 高階関数; 関数 f と g の合成関数を作る *)
let ex11 f g x = g (f x) ;;

(* 高階関数; 関数 f を n 回適用する *)
let rec ex12 n f x =
   if n= 0 then x
   else f (ex12 (n−1) f x) ;;
```

高階関数については、資料 No.7 に詳しい説明があるので、この発展課題を解く人はそちらも参照されたい。