ソフトウェア技法: No.7 (高階関数とその型付け)

亀山幸義 (筑波大学情報科学類)

1 高階関数 (higher-order function)

関数型言語は、ラムダ計算 (Lambda Calculus) に基づいた言語であり、Church が始めたラムダ計算は、「関数」の概念 (だけ) を徹底的に極めた体系であり、(1) 変数, (2) 関数を作ること, (3) 関数を使うことの 3 つしか持たない極めてシンプルな体系である。

$t ::= x$		変数
	$\lambda x.t$	関数を作る (ラムダ抽象)
	$t\ t$	関数を引数に適用する (関数適用)

ラムダ計算は、非常に簡潔な体系でありながらとても強力であり、「ラムダ計算で表現できる関数たち」と「チューリング機械 (Turing Machine) で表現できる関数たち」とは一致することが知られている。つまり、プログラムとして表すことのできる関数は、常に、ラムダ計算の式として表現できるのである。この話の詳細は、ラムダ計算の専門書、あるいは、専門科目の「計算モデル論」を参照してほしい。

ラムダ計算を非常に強力な体系にしているのは、関数をデータとして扱うことができる機能、つまり、関数を関数で扱うことができる機能である。関数を引数に取る関数、および、返り値が関数である関数[*1]を総称して**高階関数 (higher-order function)** と言う。なお、数学では、高階関数のことを、汎関数 (functional) と言う。高階関でない関数、つまり、引数も返り値も関数でないもの (数や文字列などのデータであるもの) を、一階の関数 (first-order function) と言う。

関数型言語の重要な特徴は、高階関数が使えることである。逆に、高階関数が使えるプログラム言語は、関数型言語と呼んでも差しつかえない。この資料では、高階関数を OCaml で定義し利用する方法と、その型付けについて学ぶ。

1.1 単純な高階関数

まずは、簡単な例からはじめよう。

```
(* 準備としての定義; add3 や mul2 自体は高階関数ではない *)
let add3 x =  x + 3 ;;
let mul2 x =  x * 2 ;;

(* foo1 が、最初の高階関数である *)
let foo1 f =  (f 10) + 2 ;;

(* foo1 は引数として関数 add3 や mul2 を取る *)
let _ =  foo1 add3 ;;
let _ =  foo1 mul2 ;;
```

関数 `foo1` の引数 `f` は、その本体において、`f 10` として使われている。これは、関数適用の形であるので、`f` が関数であることがわかる。`foo1 f` は、`(f 10) + 2` を計算してその結果を返す。したがって、`foo1 add3` は `add3 10 + 2 = 10 + 3 + 2= 15` を返し、`foo1 mul2` は `mul2 10 + 2 = 10 * 3 + 2 = 22` を返す。

上記の例では、関数をあらわす引数として `f` という変数を使っているが、これは、普通の変数であり、たとえば `x` を使っても全く同じである。

[*1] もちろん、引数と返り値の両方が関数であるものも高階関数である。

```
(* foo1 とまったく同じ内容の高階関数 *)
let foo1a x =  (x 10) + 2 ;;
let _ =  foo1a add3 ;;
let _ =  foo1a mul2 ;;
```

関数を表す引数の変数名としては、f や g を使うことが多いが、そのあたりは好みの問題もあり、自分のプログラムの中で統一して使えばそれでよい。

もう少し複雑な例を見てみよう。

```
let foo2 f =  f (f 10) ;;
let foo3 f =  (f (f 10)) + (f 20) ;;

let _ =  foo2 add3 ;;
let _ =  foo2 mul2 ;;
let _ =  foo3 add3 ;;
```

さらに、関数以外のものと関数を受けとる高階関数も定義できる。

```
(* 関数をあらわす引数 f と 通常のデータをあらわす引数 x を両方とる高階関数 *)
let foo4 f x =  f (f (x + 10)) ;;

let _ =  foo4 add3 3 ;;
let _ =  foo4 add3 4 ;;
let _ =  foo4 mul2 5 ;;
let _ =  foo4 mul2 6 ;;
```

高階関数の引数となる関数は、add3 などのように名前を与えられている必要はなく、無名関数でもよい。無名関数は OCaml では fun x -> ... といった構文で利用することができる。

```
let _ =  foo1 (fun x → x * 2 + 10) ;;
let _ =  foo4 (fun x → x * x + x) 9 ;;
```

ここで、高階関数の型がどうなるかを見てみよう。型 A の要素を引数にとり、型 B の要素を返す関数の型は、A -> B であった。それを踏襲して、高階関数は引数が A -> B といった型を持ったり、返り値が A -> B といった型を持ったりする。

上記の高階関数の型は、以下の通りである。

```
foo1 : (int → int) → int
foo2 : (int → int) → int
foo3 : (int → int) → int
foo4 : (int → int) → int → int
```

(int -> int) -> int という型のように、引数部分に関数型が来るのが高階関数の特徴である。

これらの括弧は省略できないことに注意されたい。たとえば、foo1 の型 (int -> int) -> int で、括弧を省略すると int -> int -> int となるが、関数型をあらわす->は右結合であるので、これは、int -> (int -> int) という意味になってしまい違うものになる。関数 foo4 の型は、括弧を補って書くと (int -> int) -> (int -> int) となる。

1.2 様々な高階関数

高階関数の使用例をいくつか見てみよう。

```
(* compose は、つの関数 2f と g を合成する *)
let compose f g x =  g (f x) ;;

let _ = compose add3 mul2 10 ;;
let _ = compose mul2 add3 10 ;;
(* 引数の関数を2回適用する *)
let double_apply f x =  f (f x) ;;

let _ = double_apply add3 10 ;;

(* 引数の関数を3回適用する *)
let triple_apply f x =  f (f (f x)) ;;

let _ = triple_apply add3 10 ;;
```

高階関数を再帰的に定義することも、もちろん問題ない。

```
(* 引数の関数をn回適用する *)
let rec multiple_apply n f x =
   if n = 0 then x
   else multiple_apply (n − 1) f (f x) ;;

let _ = multiple_apply 5 add3 10 ;;
let _ = multiple_apply 6 add3 10 ;;
let _ = multiple_apply 7 add3 10 ;;
```

1.3 リスト上の高階関数 map

リスト上の (リストを対象にした) 高階関数をいくつか見てみよう。最初は、何はともあれ、map 関数である。これは Map/Reduce フレームワークで有名になったが、もとは、関数型言語の住人である。ちなみに map というのは「地図」という意味の名詞でもあるが、もう 1 つの意味は「写像」あるいは「関数」である[*2]。

map 関数は、OCaml の List モジュールに含まれているので、List.map という名前で使うことができる。

[*2] “map” は、動詞としても使うことができて、“f maps x to y.” という文は 「f(x)=y」を意味する。

```
(* リストの要素に全て add3 を適用する *)
let _ = List.map add3 [1; 5; 8; 2; 4] ;;

(* map する関数が返すものは int 型でなくてもよい。
   以下のものは x>3 を満たすかどうかの判定結果を返す *)
let _ = List.map (fun x → (x > 3)) [1; 5; 8; 2; 4] ;;

(* リストの要素を文字列に直す *)
let _ = List.map (fun x → string_of_int x) [1; 5; 8; 2; 4] ;;

(* fun x → string_of_int x は 単に string_of_int としても同じ *)
let _ = List.map string_of_int [1; 5; 8; 2; 4] ;;
```

map 関数の使い方がわかったところで、この動作の仕組みを理解するため、自分で定義してみよう。

```
(* map 関数と同じ意味を持つ関数を自作する *)
let rec my_map f lst =
  match lst with
  | []  → []
  | h::t → (f h) :: (my_map f t) ;;

let _ = my_map add3 [1; 5; 8; 2; 4] ;;
let _ = my_map mul2 [1; 5; 8; 2; 4] ;;
let _ = my_map string_of_int [1; 5; 8; 2; 4] ;;
```

関数 my_map が List.map と同じ動作をするか、いろいろな例で試してほしい。

関数 my_map (および List.map) の型は若干複雑である。

```
my_map : ('a → 'b) → 'a list → 'b list
       = ('a → 'b) → ('a list → 'b list)
```

ここで 'a と'b は「任意の型」をあらわす型変数であり、'a と'b は同じでも違っていてもよい。これを、my_map の「一般的な型」と呼ぶことにする。

この型を理解するため、以下の使用例を考えよう。

```
let _ = my_map add3 [1; 5; 8; 2; 4] ;;
```

ここでの my_map は、int -> int 型を持つ add3 と、int list 型を持つリストをもらって、int list 型のリストを返しているので、(int -> int) -> (int list -> int list) という型の関数である。上記の一般的な型において、'a=int および'b=int と置くと、この型が得られることがわかる。

一般に、多相型の関数 (型変数'a などを含む型をもつ関数) は、実際に使われるときは、型変数を具体的な型に置きかえて使うことになる。これを「型変数の具体化」と言う。上記の例では、'a=int および 'b=int という型変数の具体化を行ったことになる。

演習問題 1. 以下の 2 つの使用例で、my_map の型を考え、型変数がどのように具体化されたか考えなさい。

```
let _ = my_map (fun x → (x > 3)) [1; 5; 8; 2; 4] ;;
let _ = my_map string_of_int [1; 5; 8; 2; 4] ;;
```

補足。map 関数は、リストのすべての要素に同一の操作をすることができるので、非常に便利である。また、並列計算では、リストのの各要素への関数適用は完全に独立 (計算が互いに依存しないで独立にできる) であるため、map に着目した並列化は、高速化の効果が期待できるものである。

1.4 リスト上の高階関数 fold

次に、畳み込み (fold) とよばれる関数を見てみよう。これも map 関数と同じかそれ以上に、関数プログラミングの世界で非常に重要な関数である。なお、関数型言語での畳み込みは fold という英語であり、解析系の数学の授業で習う「畳み込み (convolution)」とは特に関係がない。

fold 関数には、左畳みこみと右畳みこみがあるが、左畳み込み (fold left) の方で考えてみよう。これは、List モジュールで fold_left という関数として実装されている。

```
(* 準備 *)
let hoo1 x y = x + y ;;
let hoo2 x y = x * y ;;

(* 使ってみよう *)
let _ = List.fold_left hoo1 0 [1; 5; 8; 2; 4] ;;
let _ = List.fold_left hoo2 1 [1; 5; 8; 2; 4] ;;
```

1 つ目は、0+1+5+8+2+4 = 20 を計算しているだけであり、2 つ目はかけ算で同様のことをやっているだけである。

map は、リストの要素を個別に操作していたが、fold は、リストの要素を全部「まとめて 1 つにする」という処理になっていることがわかるだろう。このように、まとめて 1 つにする操作を動詞で reduce(集約する、簡約する) と言う。

上記で、0 や 1 は計算の初期値を与えているものであり、これがないと、空リストの場合の処理に困ってしまう。

リストの値の最大値の計算も簡単にできる。

```
let hoo3 x y = if x > y then x else y ;;
let _ = List.fold_left hoo3 min_int [1; 5; 8; 2; 4] ;;
```

ここで min_int は最小の (マイナスで、絶対値が大きな) 整数である。

さて、ここまでは、左から計算しようと右から計算しようと同じ結果になる例ばかりであった。これなら、左の fold であろうと右の fold であろうと同じである。次の例は、この違いがでてくるものである。

```
(* より大きい要素で、一番左にあるものを返す；そういう要素がなければ何を返してもよい 4 *)
let hoo5 x y = if x > 4 then x else y ;;
let _ = List.fold_left hoo5 0 [1; 5; 8; 2; 4] ;;

(* リストを二進数とおもって、十進数に変換する。
   たとえば [1; 0; 1; 1] を 2^3 + 2 + 1 = 11 に変換する *)
let hoo6 x y = x * 2 + y ;;
let _ = List.fold_left hoo6 0 [1; 0; 1; 1] ;;
```

fold が返すものは整数や実数等とは限らない。

```
(* もらってきたものを文字列にして連結する *)
let hoo7 x y = x ^ (string_of_int y) ;;

(* fold で壊した分を、文字列でくっつける *)
let _ = List.fold_left hoo7 "" [1; 5; 8; 2; 4] ;;

(* もらってきたものを つの要素にして連結する 2 *)
let hoo8 x y = x ^ (string_of_int y) ^ (string_of_int y) ;;
let _ = List.fold_left hoo8 "" [1; 5; 8; 2; 4] ;;

(* しかし、以下は型エラーである *)
let hoo9 x y = x ^ y ;;
let _ = List.fold_left hoo9 "" [1; 5; 8; 2; 4] ;;
```

最後の例で少し混乱したかもしれない。そこで、`fold_left` の正式な意味を見てみよう。OCaml の List モジュールのマニュアル[*3]には、`List.fold_left` 関数について、

`List.fold_left f a [b1; b2; ...; bn]` は `f (... (f (f a b1) b2) ...) bn` である

と記述されている。ここで明確になったとおり、`fold_left` は左右対称な操作ではなく、「最初の、種になる要素 `a` からはじめて、次々とリストの要素 `b1, b2, ...,bn` を関数 `f` で結合していく、という形をしている。`f` の第一引数の型は、「種になる要素」と同じ型 (従って、最終的に返す答えの型と同じ) であり、第二引数の型は、リストの要素と同じ型である。

練習のために、`fold_left` を自前で定義した上で、その型を見てみよう。

```
let rec my_fold_left f a lst =
   match lst with
   | [] → a
   | h::t → my_fold_left f (f a h) t ;;
(*
  val my_fold_left : ('a → 'b → 'a) → 'a → 'b list → 'a = <fun>
 *)
```

関数 `my_fold_left` の型をよく吟味して、どのような引数をとっているか考えてみほしい。

最後に、`fold_right` の使用例を見てみよう。

```
(* fold_right は fold_left と左右逆である *)
let _ = List.fold_right
         (fun x → fun y → sqrt (float x) +. y)
         [1; 5; 8; 2; 4]
         0.0 ;;
```

関数 `fold_right` を自前定義するとどうなるか、また、その型はどうなるか、自分で確認してほしい。

演習問題 2. `fold_left` のいろいろな使用例 (上記) において、一般的な型における型変数がどのように具体化されたか考えなさい。

[*3] 検索するとすぐ出てくるので、ここでは URL は記載しない。検索の際には OCaml List Module などのキーワードをいれるとよい。

1.5 応用

前に学習した二分木の走査 (traverse) について、もう 1 度見てみよう。

与えられた二分木を、左から右の順番に走査をして各ノードの値を文字列にして連結したものを返す関数 `string_of_tree` と、各ノードの値を整数のまま並べたリストを返す関数 `flatten` について考える。

```
type binary_tree =
   | Leaf
   | Node of int * binary_tree * binary_tree ;;

let rec string_of_tree (bt : binary_tree) : string =
   match bt with
   | Leaf → ""
   | Node(n,bt1,bt2) →
      let s1 = string_of_tree bt1 in
      let s2 = string_of_tree bt2 in
        (string_of_int n) ^ " " ^ (s1 ^ " " ^ s2) ;;

let rec flatten (bt : binary_tree) : int list =
   match bt with
   | Leaf → []
   | Node(n,bt1,bt2) →
      let s1 = flatten bt1 in
      let s2 = flatten bt2 in
        n :: (List.append s1 s2) ;;
```

2 つの関数は、非常に似た構造をしており、二分木を左から右に走査する部分と、各ノードに具体的な処理をする部分から構成されている。この前者は、2 つの関数で共通であるため、これをなんとか抽象化して、まとめたい。

二分木を左から右に走査する一般的な関数 `traverse` を次のように定義する。この関数は、「各ノードにおける具体的な処理」を関数 `f` という引数として受けとる高階関数である。

```
let rec traverse (f : int → 'a → 'a → 'a) (a : 'a) (bt : binary_tree) : 'a =
   match bt with
   | Leaf → a
   | Node(n,bt1,bt2) →
      let s1 = traverse f a bt1 in
      let s2 = traverse f a bt2 in
        f n s1 s2 ;;

let string_of_tree (bt : binary_tree) : string =
   traverse (fun n → fun s1 → fun s2 →
                (string_of_int n) ^ " " ^ (s1 ^ " " ^ s2))
            ""
            bt ;;

let flatten (bt : binary_tree) : int list =
   traverse (fun n → fun s1 → fun s2 →
                n :: (List.append s1 s2))
            []
            bt ;;
```

このように構成することにより、(1) 二分木の走査部分 (関数 traverse で実現) と、(2) ノードにおける具体的な処理 (traverse の引数となる関数で実現) を分離して記述することができ、プログラム全体が論理的に非常に美しく、維持や再利用がしやすい形になる。

たとえば、二分木のノードの最大値を求めたり、二分木のノードで最も左にある非負の値を求めたりするのも、上記の (2) のみ入れ換えれば簡単に実現することができる。

さらに、(1) だけを交換することにより、別の順序での走査も容易に実現できる。以下のものは、右から左に走査する高階関数である。

```
let rec traverse2 (f : int → 'a → 'a → 'a) (a : 'a) (bt : binary_tree) : 'a =
   match bt with
   | Leaf → a
   | Node(n,bt1,bt2) →
      let s2 = traverse2 f a bt2 in
      let s1 = traverse2 f a bt1 in
        f n s1 s2 ;;

let flatten2 (bt : binary_tree) : int list =
   traverse2 (fun n → fun s1 → fun s2 →
                n :: (List.append s1 s2))
             []
             bt ;;
```

特筆すべきことは、flatten2 の定義には影響を与えず走査する順番だけを変更すればよかったことである。

このように高階関数を適切に使えば、プログラム全体の構成が見やすくなり、わかりやすく、保守・再利用がしやすいプログラムとすることができる。

この章での traverse は、リストにおける fold と非常に似た動きをすることに気付いた人もいるだろう。これらは本質的に同種の操作であり、これらを更にまとめて、いろいろなデータ型に対する一般的な操作と考えることもできる。

2 演習

(1) List.fold_left または List.fold_right を使って、与えられた整数リストの最小値を求める関数 min を書きなさい。ただし、max_int が最大の整数であることを使ってよい。

例: `min [1;5;2;3;4] = 1`

例: `min [] = max_int`

(2) List.fold_left または List.fold_right を使って、与えられた文字列のリストに対して、辞書式順序で最後に来る文字列を求める関数 dic_last を書きなさい。ただし、空文字列 ("") が、辞書式順序で最初にくることを使ってよい。また、OCaml では文字列 s1,s2 に対して、 s1 < s2 とすると辞書式順序での比較ができることを使ってよい。

例: `dic_last ["aloha"; "mahalo"; "komo"] = "mahalo"`

例: `dic_last [] = ""`

(3) リストに対する高階関数 List.iter (iterator の意味) と同等の動きをする関数 my_iter を定義せよ。

例: `my_iter print_int [111; 222; 333]` = `()` (ただし、111222333 が印刷される。)

(4) (発展課題) リストに対する高階関数 List.fold_right と同等の動きをする関数 my_fold_right を定義せよ。

例： `my_fold_right (fun x -> fun y -> (float x) +. y) [1; 2; 3] 0.0` = `6.0`

(5) (発展課題) 関数 f : float -> float と、float 型の値 v を与えられたとき、x における f の微分 f'(v) の近似値を求める関数 deriv を定義せよ。

例: `deriv (fun x -> x *. x +. x +. 1.0) 3.5` = 8.0 程度の値

ヒント: $f'(x)$ は、$\frac{f(x+\epsilon)-f(x)}{\epsilon}$ という式で、ϵ を、プラスおよびマイナス方向から、0 に近付けたときの極限である。よって、微係数の近似値を計算するもっとも単純な方法は、(1) プラスの小さな ϵ に対してこの式を計算し、また、(2) マイナスの小さな ϵ に対してこの式を計算して、それらが近ければ、答えとする、というものである。(関数 f がこの点において微分可能であると仮定してよいなら、(1) と (2) は十分近いはずであり、片方だけ計算してもよい。)

さらに頑張るなら、ϵ をどんどん 0 に近付けて、結果が、誤差として許容できる範囲におさまったときのものを、答えとするのが良いだろう。

(6) (発展課題) OCaml のリストモジュールには、List.sort 関数が用意されており、これは、任意の順序で、ソート (整列) をおこなうものである。この関数について調べて、「整数リストを小さい順から並べる」「文字列リストを辞書式順に並べる」など、いろいろなソートが実現できることを示しなさい。

3 付録: ラムダ計算におけるコンビネータ

ここで、ラムダ計算などの分野で、コンビネータ (combinator) とよばれる高階関数を導入しよう。

これらは、それぞれ 1 文字の名前をもっており、I, K, S, B, C, W の 6 つは OCaml で次のように定義できる。

```
let combinator_i x     = x ;;
let combinator_k x y   = x ;;
let combinator_s x y z = (x z) (y z) ;;
let combinator_b x y z = x (y z) ;;
let combinator_c x y z = (x z) y ;;
let combinator_w x y   = (x y) y ;;
```

これらのコンビネータの型を調べてみなさい。コンビネータに対して、たとえば、等式 $SKKx = Ix$ (任意の式 x に対して) が成立する。

なお、ここでは代表的なコンビネータを紹介したが、一般に、コンビネータは、自由変数をもたないラムダ式のことである。