ソフトウェア技法: No.6 (直積型と代数的データ型)

亀山幸義 (筑波大学情報科学類)

[2017/07/28; 課題の addplus で例題の書き方が不適切だったので修正しました]

1 はじめに

Pascal や Modula-2 言語の創始者ニクラウス・ヴィルト (Niklaus Wirth) の有名なフレーズ「データ構造＋アルゴリズム＝プログラム」は、プログラミングにおけるデータ構造の重要さを端的に表している。現代の多くのプログラム言語は、様々なデータ構造を型として用意するとともに、問題ごとに適切なデータ構造を、「型」として定義する仕組みを用意している。ユーザ (プログラマ) が、新しいデータ型を定義して使うことができれば、プログラムを書く対象 (対象領域、解くべき問題) に合ったデータ構造を用意することにより、適切な抽象度のプログラムを作成することができる。C 言語では、様々なデータ構造を struct (構造体) として表現するのが一般的である。オブジェクト指向言語のクラスは、様々なデータとそれに対する操作をまとめて、ひとかたまりのオブジェクトを作るための仕組みである。

OCaml を始めとする関数型言語は、豊富なデータ型を用意している。ここまで見てきたものは、基本型 (int, float, char, string, ...) と型を構成する仕組みとしてリスト型、関数型であるが、これ以外にも、直積型、レコード型、バリアント型、配列型など、様々なデータ型を用意している。

また、OCaml でユーザが新しい型を定義する仕組みとして、代数的データ型 (algebraic datatype) がある。これは、多様なデータ型を 1 つの仕組みで表現できる便利な手段であり OCaml プログラミングには欠かせない存在となっているので、ここで紹介する[*1]。

2 直積型

OCaml に備わっている型の 1 つとして、直積型 (product) を学習しよう。これは、集合における「直積集合」($A \times B$ など) の概念に対応する型であり、直積型の要素は、(a, b) の形の「組 (tuple)」である。

OCaml では直積型は asterisk (*) を使って $A * B$ と記述する。

```
(* 文字列と浮動小数点数の組 *)
let ex1 = ("pi", 3.14) ;;

let ex2 = ("e",  2.68) ;;

(* 組の要素を取り出す関数は fst と snd である。*)
let _ = fst ex1 ;;

let _ = string_of_float (snd ex2) ;;
```

[*1] なお、OCaml における、もう 1 つの重要なデータ構造として、モジュール (module) がある。モジュールは、オブジェクト指向言語のクラスに類似した「データ抽象」を提供するのであるが、モジュールを自分で定義するのは、この授業の範囲を越えるので、参考書等で自習してほしい。

```
(* 組に対するパターンマッチ *)
let show (p : string * float) : string =
   match p with
   | (s, f) → s ^ (string_of_float f) ;;

(* 参考; match 式で、ケースが1つしかない場合は、let でも良い *)
let show (p : string * float) : string =
   let (s,f) = p in
      s ^ (string_of_float f) ;;

(* 組を作るのは、単に (a,b) と書くだけ。実は、括弧なしで a,b でもよいのだが、
   わかりくくなるので、この授業では必ず (a,b) と書くことにしておこう。*)
let add_pair (p : string * float) (d : float) : string * float =
   match p with
   | (s, f) → (s, f +. d) ;;
```

Fibonacci 関数や「2 番目に大きな関数」を求める関数は、組を使うと書きやすい。

```
(* 組を使わない、素朴な実装 *)
let rec fib_slow n =
  if n <= 2 then 1
  else fib_slow (n−2) + fib_slow (n−1) ;;

(* 組を使った、高速な実装 *)
let rec fib_walk n =
  if n <= 1 then (1,1)
  else
    match fib_walk (n−1) with
    | (v1,v2) → (v2, v1+v2) ;;

let fib_fast n =
  fst (fib_walk n) ;;
```

```
(* リストの大きい方から上位２つの数を返す *)
let rec take_top2 (lst : int list) : int * int =
  match lst with
   | [] → failwith "too short list"
   | [v1] → failwith "too short list"
   | [v1; v2] → if v1 > v2 then (v1,v2)
                else (v2,v1)
   | h::t →
      begin
        match take_top2 t with
        | (v1,v2) →
           if h > v1 then (h,v1)
           else if h > v2 then (v1,h)
           else (v1,v2)
      end ;;
```

3 代数的データ型

3.1 単純な場合 (バリアント型)

代数的データ型の単純な場合は、バリアント型 (直和型) とも呼ばれる。集合論において、集合 $S_1, S_2, ...S_n$ の直和集合とは、これらの集合の要素を「重複がないようにタグをつけて」集めたものである。たとえば、集合 $\{1,2,3\}$ と集合 $\{1,3,5\}$ の和集合は、$\{1,2,3,5\}$ であるが、直和集合は、$\{(left,1),(left,2),(left,3),(right,1),(right,3),(right,5)\}$ といった集合となり、もとの集合の各要素にタグをつけたものを集めてきたものである。ここでは、もとの集合のどちらから来ているかに応じて、left あるいは right というタグを付けた。バリアント型はこの直和集合に非常によく似ており、タグをユーザが明示的に定めるものである。

バリアント型を 1 つ定義してみよう。次のものは、1 週間の 7 つの曜日を 1 つの型としたものである。

```
(* 型の定義は type というキーワードで始まる *)
type days =
   | Sunday | Monday | Tuesday | Wednesday | Thursday
   | Friday | Saturday ;;

let _ = Wednesday ;;
(* 処理系の出力
     - : days = Wednesday
*)
```

これで、7 つの曜日を表す単語が days 型の要素となった。縦棒は、区切りの記号である。なお、型の名前は、変数や関数と同様、小文字で始め、型の要素につけるタグ (構成子、コンストラクタ) は大文字で始める必要がある。

次に、days 型の要素を扱う関数を作ってみよう。days 型の要素を使うためには、パターンマッチを使う。

```
let is_weekday d =
  match d with
  | Sunday   → false
  | Saturday → false
  | _        → true ;;

let _ = is_weekday Friday ;;
let _ = is_weekday Saturday ;;

let nextday d =
  match d with
  | Sunday   → Monday    | Monday    → Tuesday
  | Tuesday  → Wednesday | Wednesday → Thursday
  | Thursday → Friday    | Friday    → Saturday
  | Saturday → Sunday     ;;

let _ = nextday Wednesday ;;
```

関数 nextday は days -> days という型をもつ。

次に、grade という型を定義しよう。UT 大学では、grade (成績) として、A+, A, B, C, D もしくは、点数 (整数) をつける。このようなデータを一括して扱いたい、たとえば、ある学生の成績一覧を、["A"; 80; "C"; "D"; 90] といったリストで表したい、とする。

これまでの知識では、種類の異なるデータを 1 つのリストにいれようとしても、型が異なるので、できなかった。しかし、バリアント型を用いると、このようなデータを 1 つの型にすることができる。

```
(* grade 型の定義 *)
type grade =
   | Word of string
   | Numeric of int  ;;

let g1 = Word("A+") ;;
let g2 = Word("C") ;;
let g3 = Numeric(90) ;;
let lst1 = [g1; g2; g3] ;;

(* 以下は型エラー *)
let _ = Word(70) ;;
let _ = Numeric("A") ;;
let _ = Numeric(3.14) ;;
```

これらの例でわかるように、grade 型の要素は、Word(s) か Numeric(n) の形をしている。ここで s は string 型, n は整数型でなければいけない。逆に、これらの形のデータは、すべて grade 型を持つ。

grade 型のデータを使うときも、パターンマッチを使う。

```
(* 合格の成績かどうかの判定関数 *)
let pass g =
   match g with
   | Word(s)    → (s = "A+") || (s = "A") || (s = "B") || (s = "C")
   | Numeric(n) → (n >= 60) ;;

let _ = pass g1 ;;
let _ = pass g3 ;;
```

バリアント型の 1 つ 1 つのケース[*2]の引数は、複数個あってもよいし、0 個でもよい。

```
type foo =
   | A                          (* 引数が 0 個のケース *)
   | B of int                   (* 引数が 1 個のケース *)
   | C of bool * string         (* 引数が 2 個のケース *)
   | D of string * int * float  (* 引数が 3 個のケース *)
   ;;

(* 以下の式は foo list 型を持つ *)
let _ = [A; B(10); C(true, "abc"); D("c", 25, 2.7)] ;;
```

ここで、データ型の構成子が複数の引数を持つ場合 (上記の C や D の場合) は、型の名前の間を * で区切っている。これは、直積型との類似性から * を使っているのであるが、ここは直積を表しているわけではないことに注意せよ。

OCaml における重要な注意点として、データ型の構成子 (上記の Word, Numeric,A,B,C,D など) は、異なる型のものを含めてすべて相異ならないといけない、ということがある。つまり、上記に加えて、以下のように定義すると，A は goo 型の構成子になってしまい，foo 型の A は使えなくなってしまう.

```
type goo = A | Word ;;

let _ = A ;;

(*  システムの出力
   - : goo = A
 *)
```

このように「型の構成子の名前は、(型を越えて、全体で) 唯一でなければならない」というのは、良い点もあるが (構成子名から型が一意的に決まるのでプログラム上のミスが減る)、悪い点もある (似たような構成子に違う名前をつけないといけないので、煩わしい)。OCaml のモジュールは名前空間を分けてこの問題を解決するために使える仕組みである。興味がある人は自分で勉強してほしい。

3.2 再帰的データ型

再帰的データ型というのは、データ型の定義が再帰的、つまり、型 abc の定義の中に型 abc があらわれるものである。(再帰的データ型と再帰的関数定義の間に、直接的な関係はない。ただし、再帰的データ型を操作する関数は、ほとんどの場合、再帰的関数である。)

[*2] 上記の days 型における Word of string のように、縦棒で区切られた 1 つの定義を「ケース」と言う。

再帰的データ型の例として、二分木 (binary tree) を定義しよう。二分木にもいろいろあるが、ここでは、ノード (節, node) は、必ず 2 つの子供のノードと、整数データを 1 つ持ち、葉 (leaf) にはデータがない、というタイプのものを考える。

```
(* 二分木のデータ型の定義 *)
type binary_tree =
   | Leaf
   | Node of int * binary_tree * binary_tree ;;

(* 二分木を作る *)
let bt1 = Leaf ;;
let bt2 = Node(2, Node(3,Leaf,Leaf), Leaf) ;;
let bt3 = Node(2, Node(3,Leaf,Leaf), Node(0,Leaf,Leaf)) ;;
```

上記の定義からわかるように binary_tree (binary tree の意味) という型は、Leaf か Node(n,bt1,bt2) の形の要素を持つ。ここで n は整数型、bt1,bt2 は再び binary_tree 型の要素である。このように、binary_tree 型の定義の中に binary_tree があらわれているので、再帰的な型定義となっている。

binary_tree 型の要素は、再帰を有限回繰返して使って得られる要素たちである。

さて、binary_tree が定義できたところで、そのデータを使ってみよう。これもパターンマッチを使えばよい。

```
(* 二分木の整数値の総和 *)
let rec sum_tree bt =
   match bt with
   | Leaf → 0
   | Node(n,bt1,bt2) → n + (sum_tree bt1) + (sum_tree bt2) ;;

let _ = sum_tree bt3 ;;

(* 二分木の高さ *)
let rec height bt =
   match bt with
   | Leaf → 0
   | Node(_,bt1,bt2) →
     let h1 = height bt1 in
     let h2 = height bt2 in
       if h1 > h2 then h1 + 1 else h2 + 1 ;;

let _ = height bt3 ;;
```

総和や高さの計算が、非常に簡単に定義できることに驚く人もいるかもしれない。このように、再帰データ型で表現されたデータに対する操作を非常に自然に (人間が考えるのと同じ形で) 表現できる点が、関数型言語の大きな利点の 1 つである。

今度は、二分木を出力する関数を定義しよう。

```
(* 二分木の左右反転 *)
let rec mirror bt =
   match bt with
   | Leaf → Leaf
   | Node(n,bt1,bt2) → Node(n,mirror bt2, mirror bt1) ;;

let _ = mirror bt3 ;;

(* 二分木から、0 という値を持つノードおよびその下の部分木を削除 *)
let rec remove_zero bt =
   match bt with
   | Leaf → Leaf
   | Node(n,bt1,bt2) →
     if n = 0 then Leaf
     else Node(n,remove_zero bt1, remove_zero bt2) ;;

let _ = mirror bt3 ;;
```

4 応用

代数的データ型を使って、整数と四則演算からなる「式」を表現してみよう。たとえば、(2+(3*4))-(5/2) といったものを式と呼ぶ。この式の構文 (抽象構文木) を、expr というデータ型で表現すると以下のようになる。

```
type expr =
   | Const of int
   | Add   of expr * expr    (* a + b を表す *)
   | Sub   of expr * expr    (* a − b を表す *)
   | Times of expr * expr    (* a * b を表す *)
   | Div   of expr * expr    (* a + b を表す *)
   ;;
let e1 = Const(3) ;;
let e2 = Const(5) ;;
let e3 = Add(e1, Times(e2,e1)) ;;
let e4 = Times(e1, Times(e2,e1)) ;;
let e5 = Times(Add(e1, Times(e2,e1)),Sub(e1,Div(e2,e1))) ;;
let e6 = Div(e4,e5) ;;
```

上記の式に対して、それを計算した結果を返す関数 eval を定義しよう。(eval という名前は「評価する」という意味の evaluate に由来する。)

```
let rec eval (e : expr) : int =
   match e with
   | Const(i)      → i
   | Add(e1,e2)    → (eval e1) + (eval e2)
   | Sub(e1,e2)    → (eval e1) − (eval e2)
   | Times(e1,e2) → (eval e1) * (eval e2)
   | Div(e1,e2)    → (eval e1) / (eval e2) ;;

let _ = eval e3 ;;
let _ = eval e4 ;;
```

関数 eval は、expr -> int という型を持ち、式を与えられると、それを計算した結果 (整数) を返す。
今度は、式を「印刷」してみよう。印刷といってもいきなり print するのではなく、文字列に変換することを考える。

```
let rec string_of_expr (e : expr) : string =
   match e with
   | Const(i)      → string_of_int i
   | Add(e1,e2)    → "(" ^ (string_of_expr e1) ^ " + " ^ (string_of_expr e2) ^ ")"
   | Sub(e1,e2)    → "(" ^ (string_of_expr e1) ^ " − " ^ (string_of_expr e2) ^ ")"
   | Times(e1,e2) → "(" ^ (string_of_expr e1) ^ " * " ^ (string_of_expr e2) ^ ")"
   | Div(e1,e2)    → "(" ^ (string_of_expr e1) ^ " / " ^ (string_of_expr e2) ^ ")"
   ;;

let s4 = string_of_expr e4 ;;
let _ = print_endline s4 ;;
```

このように、構文木をデータ型として表して、その評価器 (インタプリタ) を記述するのは、OCaml が最も得意とするところである。

5 演習

(1) 特別な値を持つデータ型に関する問題:
以下のデータ型 intplus は、整数型に「正の無限大」と「負の無限大」を追加したものである。

```
type intplus =
   | Fin of int
   | Inf of bool   (* Inf(true) は正の無限大、Inf(false)は負の無限大 *)
   ;;
```

- この型の要素に対する「足し算」をあらわす関数 addplus を定義せよ。ただし、正の無限大と負の無限大を足した結果は不定なので、failwith で例外を発生させること。[2017/07/28; 例題の書き方が不適切だったので修正しました]
  例: addplus (Fin(10)) (Fin(-5)) = Fin(5)
  例: addplus (Fin(10)) (Inf(true)) = Inf(true)
- 同様に、かけ算 timesplus を定義せよ。
- (発展課題) intplus に「不定」をあらわす構成子を追加して、足し算とかけ算を再定義せよ。

(2) 2 分木に対する問題:

- 2 分木 (binary_tree 型のデータ) におけるノードの個数 (Node というタグの個数) を計算する関数 `size` を定義せよ。
  例: `size(Node(1,Node(2,Node(3,Leaf,Leaf),Leaf),Node(4,Leaf,Leaf)))`$= 4$
  例: `size(Leaf)` $= 0$
- 2 分木 (binary_tree 型のデータ) を、inorder (中間順、通りがけ順) で走査して得られる整数値たちを、リストにして返す関数 `flatten` を定義せよ。
  例: `flatten(Node(1,Node(2,Node(3,Leaf,Leaf),Node(4,Leaf,Leaf)),Node(5,Leaf,Leaf)))` $=$`[3;2;4;1;5]`
- 2 分木 (binary_tree 型のデータ) が、バランス木であるかどうかを判定する関数 `is_balanced` を定義せよ。ただし、バランス木とは、すべてのノードにおいて、その左部分木の高さと、右部分木の高さの差の絶対値が 1 以下である木のことである。
  例: `is_balanced(Node(1,Node(2,Node(3,Leaf,Leaf),Leaf),Leaf))`=false
  例: `is_balanced(Node(1,Node(2,Node(3,Leaf,Leaf),Leaf),Node(4,Leaf,Leaf)))`=true
- (発展課題) 2 分木 (binary_tree 型のデータ) が、整列された木であるかどうかを判定する関数 `is_sorted` を定義せよ。ただし、整列された木とは、すべてのノードにおいて、そのノードの値が、左部分木に含まれるすべてのデータの値以上であり、右部分木に含まれるすべてのデータの値以下であることである。
  例: `is_sorted(Node(4,Node(2,Node(5,Leaf,Leaf),Node(3,Leaf,Leaf)),Node(5,Leaf,Leaf)))`=false
  例: `is_sorted(Node(4,Node(2,Node(1,Leaf,Leaf),Node(3,Leaf,Leaf)),Node(5,Leaf,Leaf)))`=true

(3) 式に対する問題

- 式 (expr 型のデータ) が、何個の乗算を含むかを返す関数 `count_times` を定義せよ。
  例: `count_times(Add(Const(10),Times(Const(20),Times(Const(30),Const(40)))))` $= 2$
- (発展課題) $e \cdot 0 = 0$ という法則に従って、式 (expr 型のデータ) を簡略化する変換をする関数 `opt` を定義せよ。
  例: `opt(Add(Const(10),Times(Const(20),Const(0))))` $=$ `Add(Const(10),Const(0))`
- (発展課題) 式の構文を拡張して、より広い範囲の式を表現して、その値を計算できるようにせよ。たとえば、C 言語における式を構成するオペレータたちを追加してみよ。