Logică și structuri discrete

Liste

Lucrul cu iteratori

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

7 octombrie 2013

Liste

Listele sunt un mod de bază de a lucra cu *colecții* de elemente.

O listă e o secvență *finită*, *ordonată*, de valori de *același tip*.

listele sunt *finite*, dar pot avea lungime oricât de mare
  spre deosebire de *tuple* (câte un tip pentru fiecare lungime):
    perechi, triplete, n-tuple
  deobicei, un tip separat (engl. *stream*) pentru secvențe infinite

ordinea elementelor contează: $[1; 3; 2] \neq [3; 1; 2]$
  spre deosebire de *mulțimi*

elementele unei liste sunt de *același tip*

accesul la elementele listei e *secvențial* (acces direct doar la primul)
  spre deosebire de *tablou*, cu *acces direct* la toate elementele

Listele ca tip recursiv

Listele pot fi *definite recursiv*:
o listă este o listă vidă, sau un *element* urmat de o listă.

Mai precis, pentru că listele sunt de un anumit *tip*:
– lista vidă este o listă de elemente de tip a (pentru orice tip)
– combinația dintre un element de tip a (ca prim element) și o listă de elemente de tip a (restul = coada listei) e deasemenea o listă de elemente de tip a.

Definiția de mai sus se numește și *inductivă*, deoarece definește toate listele (de un anumit tip):
– pornind de la cea mai simplă (*cazul de bază*),
– și prin modul de a construi o listă mai mare dintr-una mai mică (*pasul inductiv*).

Tipuri de date algebrice

Un *tip de date algebric* e definit ca o combinație de alte tipuri.
Variante comune sunt:

– tipul *produs* (produsul cartezian $A \times B$ a două tipuri A și B)

```
type physunit = float * string
```

O valoare a acestui tip e o pereche (în general: tuplu) de elemente din tipurile componente.

– tipul *sumă* (tip uniune, sau tip cu *variante*).
În ML, fiecare variantă are un *constructor* de tip, notat printr-o etichetă (tag), pentru a deosebi variantele între ele

```
type mix = Int of int | Real of float | Str of string
```

Limbajele funcționale au deobicei teoria tipurilor bine fundamentată matematic; ele permit definirea de tipuri de date algebrice.

Listele ca tip de date algebric

Pentru a da definiția recursivă a tipului listă, ca mai sus, ne trebuie:
– o valoare pentru *lista vidă*
– un *constructor* care formează o nouă listă dintr-un element (capul listei) și altă listă (coada noii liste)

Am putea defini *tipul recursiv* listă de elemente de tip `'a` ca fiind:
`type 'a list = Nil | Cons of 'a * 'a list`
(un tip sumă dintre un tip cu doar o valoare, și un tip produs)

`Cons` e *constructorul* care produce o listă dintr-un element și o listă iar `Nil` e un constructor de tip (fără argumente) pentru lista vidă

În ML, tipul listă e predefinit (nu e nevoie de definiția de mai sus, dar ea arată cum s-ar putea construi).

Lista vidă e `[]`, iar *constructorul* pentru liste e `::` (operator infix).

Lista ca tip de date în ML

ML definește tipul de date `'a list` pentru orice tip de element `'a`

Listele se scriu între paranteze drepte `[ ]` cu `;` între elemente:

```
# [1;3;2] (* o lista de intregi *);;
- : int list = [1; 3; 2]
# ["asta";"e";"o";"lista"] (* o lista de siruri *) ;;
- : string list = ["asta"; "e"; "o"; "lista"]
```

Operatorul `::` creează o listă dintr-un element (care devine capul noii liste) și o listă (care devine coada noii liste). *Lista vidă* e `[]` .
`::` e asociativ la dreapta (se poate folosi de mai multe ori).

```
# 2 :: [3;1];;
- : int list = [2; 3; 1]
# 1 :: 2 :: [];;
- : int list = [1; 2]
# [2;1] @ [3];;
- : int list = [2; 1; 3]
```

`@` concatenează două liste (intern, parcurge prima până la capăt).

Prelucrarea prin potrivire de tipare (pattern matching)

În general, tipurile de date algebrice se prelucrează prin tipare (cu un tipar pentru fiecare variantă).

Putem accesa direct capul listei și coada listei (după definiția recursivă a tipului de date listă).

```
match lista with
    []  ->  expr1
  | cap :: coada ->  expr2
```

Această construcție e o *expresie*, cu rezultat *expr1* dacă *lista* e vidă; altfel, identificatorii *cap* și *coada* sunt *legați* la cele două părți ale listei, și pot fi folosiți în *expr2*, care dă rezultatul întregii expresii

Mai concis, cu `function` definim o *funcție de argument listă* cu rezultat dat de valoarea expresiei–tipar:

```
function
  | []  ->  expr1
  | cap :: coada ->  expr2
```

Bara | la prima variantă e opțională dar poate face codul mai ușor de citit

Funcții predefinite cu liste

Modulul `List` oferă o serie funcții pentru lucrul cu liste.
Le putem folosi cu numele `List.`*numefunctie* .
Sau: se deschide inițial modulul: `open List`
apoi se poate folosi numele simplu, dar prima variantă e mai lizibilă
(e mai clar când apar liste; există funcții similare și în module)

Funcțiile cele mai simple: `List.hd` (*head*) și `List.tl` (*tail*)
returnează capul, și respectiv coada listei.

```
# List.hd [1;4;3];;
- : int = 1
# List.tl [1;4;3];;
- : int list = [4; 3]
```

Potrivire de tipare sau `hd` / `tl` ?

Funcțiile hd și tl nu sunt definite pentru lista vidă (sunt funcții parțiale pe tipul listă). La apel cu `[]` ele *generează o excepție*.

```
# List.hd [];;
Exception: Failure "hd".
# List.tl [];;
Exception: Failure "tl".
```

Vom discuta în alt curs despre tratarea excepțiilor.

Într-o funcție, trebuie să tratăm *toate cazurile*.

Folosind potrivirea de tipare (cu `match ... with` sau `function`) *compilatorul verifică* și ne asigură că nu am uitat nicio variantă.
Folosind `hd` și `tl` trebuie să ne asigurăm noi că nu avem lista vidă.

Preferăm de aceea accesul la cap/coadă prin *potrivirea de tipare*.

Lista ca tip de date abstract (TDA)

Un *tip de date abstract* e un tip de date definit prin operațiile care se pot efectua asupra lui (si constrângerile între acestea), fără a expune modul în care acestea sunt implementate.

TDA *izolează* (abstractizează) *definiția/interfața* de *implementare*.

Ne permite implementări modificabile și interschimbabile, fără a afecta programul care le folosește doar prin interfață.

TDA listă L cu tip de element E e deobicei definit prin:

$nil : () \to L$	(constructorul de listă vidă)
$cons : E \times L \to L$	(constructorul pentru liste)
$hd : L \to E$	(capul listei)
$tl : L \to L$	(coada listei)

și axiomele $hd(cons(e, l)) = e$ și $tl(cons(e, l)) = l$

Această definiție *abstractă* e suficientă pentru a defini lista ca obiect matematic și a raționa despre proprietățile listelor.

Funcții simple: lungimea unei liste

```
let rec len = function
  | [] -> 0
  | _ :: t -> 1 + len t
```

Tiparul _ se potrivește cu orice. Folosim pentru a ignora valoarea.

Variantă: în parcurgere, acumulăm rezultatul parțial
$\Rightarrow$ încă un parametru: elementele numărate până acum

```
let rec len2 r = function
  | [] -> r
  | _ :: t -> len2 (r+1) t
let len = len2 0 (* urmatorul argument e lista *)
```

(inițial, nu am numărat niciun element, de aici argumentul 0)

Modulul `List` definește funcția `List.length` .

Definiții locale

```
let rec len2 r = function
  | [] -> r
  | _ :: t -> len2 (r+1) t
let len = len2 0
```

se rescrie mai frumos:

```
let len =
  let rec len2 r = function
    | [] -> r
    | _ :: t -> len2 (r+1) t
  in len2 0
```

Definiția funcției ajutătoare `len2` e folosită `in` expresia `len2 0` care definește funcția dorită, `len` .
Numele `len2` e vizibil *doar* în expresia urmând lui `in` .

Recursivitatea finală (prin revenire; tail recursion)

În varianta

```
let rec len = function
  | [] -> 0
  | _ :: t -> 1 + len t
```

adunarea se face doar la revenirea din apel: pentru o listă de lungime n vom avea n apeluri active când ajungem la sfârșitul ei.
$\Rightarrow$ poate depăși stiva la liste foarte lungi

Preferăm varianta 2:

```
let len =
  let rec len2 r = function
    | [] -> r
    | _ :: t -> len2 (r+1) t
  in len2 0
```

în care calculul (+) se face la apel, iar la sfârșit rezultatul poate fi returnat *direct* pe toată stiva de apeluri
$\Rightarrow$ compilatorul poate optimiza apelul în *iterație* (ciclu)

Recursivitatea finală e practic la fel de eficientă ca iterația

Funcții simple: testul de membru

Apare valoarea `x` în listă ?

```
let rec mem x = function
  | [] -> false
  | h :: t -> x = h || mem x t
```

```
val mem : 'a -> 'a list -> bool = <fun>
```

`x` e membru în listă dacă:

- `x` e capul listei `h`, SAU
- `x` e membru în coada listei `t`

Operatorul `||`: SAU logic: cel puțin un operand adevărat (`true`) dacă primul e `true`, rezultatul e `true` $\Rightarrow$ nu mai evaluează al doilea

`List.mem` e deasemenea o funcție predefinită .

Iteratori pentru prelucrări pe liste

a itera = a repeta

E natural să *prelucrăm toate elementele* unei liste.

Distingem trei cazuri principale

1. *Facem* ceva pentru fiecare element (ex. tipărim)
   (fără a produce vreo valoare ca rezultat) `List.iter`
2. *Transformăm* fiecare element al listei cu aceeași funcție
   obținem o nouă listă `List.map`
3. *Combinăm* toate valorile din listă `List.fold_left`
   (le acumulăm succesiv într-un rezultat) `List.fold_right`

Scriem doar funcția pentru *un pas* de prelucrare (un element)
Lista e *parcursă automat* de iteratori

Iterarea peste toate elementele listei

`List.iter : ('a -> unit) -> 'a list -> unit`

`List.iter f [e1; e2; ... en]` apelează `f e1; f e2; ... f en`

Funcția `f`: folosită pentru *efectul* (ex. tipărire), nu valoarea ei
ML are tipul `unit`, cu singura valoare notată `()` (C are `void`)

```
let rec iter f = function
    | [] -> () (* orice functie are valoare, aici () *)
    | h :: t -> f h; iter f t (* a; b are valoarea b *)
```

sau, cu o funcție auxiliară care evită retransmiterea parametrului `f`:

```
let iter f =
  let rec iterf = function
    | [] -> ()
    | h :: t -> f h; iterf t
  in iterf
```

```
List.iter print_int [1;2;3]    (* tipareste 123 fara spatii *)
List.iter (Printf.printf "%d ") [1;2;3]
1 2 3 - : unit = ()            (* tipareste 1 2 3 cu spatii *)
```

Transformarea tuturor elementelor dintr-o listă

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

`List.map [e1; e2; ... en]` e lista `[f e1; f e2; ... f en]`

Rezultatul lui `f` poate avea alt tip decât parametrul
putem obține o listă cu alt tip de elemente

```
let rec map f = function
    | [] -> []
    | h :: t -> f h :: map f t
```

sau, cu o funcție auxiliară

```
let map f =
  let rec mapf = function
      | [] -> []
      | h :: t -> f h :: mapf t
  in mapf
```

```
List.map ((+) 2) [3; 7; 4]                    (* lista [5; 9; 6] *)
List.map String.length ["acesta";"e";"un";"test"]
- : int list = [6; 1; 2; 4] (* lista lungimilor sirurilor *)
```

Combinarea elementelor dintr-o listă

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_left f a [e1; e2; ... en]
    = f (...f (f a e1) e2 ...)  en
```

`List.fold_left` calculează succesiv:

`a1 = f a e1`, `a2 = f a1 e2`, ..., `an = f` a_{n-1} `en` (rezultatul)

```
let fold_left f =
  let rec foldf a = function
    | [] -> a
    | h :: t -> foldf (f a h) t
  in foldf
```

`List.fold_left` prelucrează elementele de la cap, e *tail-recursive*.

```
List.fold_left (+) 0 [3; 7; 4]   (* suma pornind de la 0: 14 *)
```

```
List.fold_left (fun s e -> s + String.length e) 0 ["a";"si";"b"]
- : int = 4      (* suma lungimilor sirurilor din lista *)
```

`0 + length "a"= 1` $\rightarrow$ `1 + length "si"= 3` $\rightarrow$ `3 + length "b"= 4`

Cum funcționează și cum folosim `List.fold_left`

`List.fold_left`: un *ciclu*, odată pentru fiecare element al listei
calculează un *rezultat, actualizat la fiecare iterație*
(pentru fiecare element)

`List.fold_left` are nevoie de 3 parametri:

1. o funcție f cu 2 parametri de tip `'a -> 'b -> 'a`
   p1: rezultatul calculat până acum de tip `'a`
   p2: elementul curent din listă de tip `'b`
   rezultatul f(p1, p2) devine p1 în apelul cu următorul element
2. valoarea inițială de tip `'a`
   (rezultatul pentru lista vidă, și p1 pentru primul apel al lui f)
3. lista de prelucrat de tip `'b list`

Un limbaj imperativ ar folosi o *variabilă, atribuită* la fiecare iterație
În `List.fold_left`, rezultatul funcției f în fiecare iterație e folosit de f în următoarea iterație (ca prim parametru).

Exemple de funcționare pentru `List.fold_left`

Minimul unei liste

```
let list_min = function
    | [] -> invalid_arg "empty list" (* exceptie *)
    | h :: t -> List.fold_left min h t

list_min [3; 9; -2; 4]     -> List.fold_left min 3 [9; -2; 4]
(fun m e -> min m e) 3 9       -> min 3 9 = 3
(fun m e -> min m e) 3 (-2)    -> min 3 (-2) = -2
(fun m e -> min m e) (-2) 4    -> min (-2) 4 = -2
```

Inversarea unei liste:
capul listei rămase de inversat devine capul rezultatului acumulat

```
let rev = List.fold_left (fun t h -> h :: t) [] [3; 7; 5]
(fun t h -> h :: t) [] 3          -> 3 :: [] = [3]
(fun t h -> h :: t) [3] 7         -> 7 :: [3] = [7; 3]
(fun t h -> h :: t) [7; 3] 5      -> 5 :: [7; 3] = [5; 7; 3]
```

`List.rev` există ca funcție standard pentru liste

Combinarea elementelor dintr-o listă

```
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
fold_right f [a1; a2;...; an] b
    = f a1 (f a2 (...  (f an b) ...))
```

`fold_right` calculează, de la dreapta la stânga
rezultatul `b0 = f a1 b1` $\leftarrow$ `b1 = f a2 b2` $\leftarrow$ ... b_{n-1} `= f an b`

```
let fold_right f lst b =
  let rec foldf b = function
    | [] -> b
    | h :: t -> f h (foldf b t)
  in foldf b lst
```

`fold_right` prelucrează elementele de la coadă, nu e *tail-recursive*.

Exemplu: filtrarea unei liste

`List.filter : ('a -> bool) -> 'a list -> 'a list`

`List.filter f [a1; a2; ...; an]` :
lista elementelor ak pentru care `f ak` e adevărată

```
let filter f =
  let rec filtf = function
    | [] -> []
    | h :: t -> let ft = filtf t in
                if f h then h :: ft else ft
  in filtf
```

```
 List.filter (fun x -> x mod 3 = 0) [1;2;3;4;5;6];;
- : int list = [3; 6]
List.filter ((<) 3) [1;2;3;4;5;6];; (* fun x -> 3 < x *)
- : int list = [4; 5; 6]  (* (<) 3 = fun x -> (<) 3 x *)
```

Exercițiu: scrieți `filter` folosind `List.fold_right`

Exemplu: ciurul lui Eratostene

```
(* lista numerelor de la a la b *)
(* rescrieti in mod tail-recursive pornind de la b *)
let rec fromto a b =
  if a > b then [] else a :: fromto (a+1) b

let rec sieve = function
  | [] -> []
  | h :: t -> h :: sieve (List.filter
                          (fun x -> x mod h <> 0) t)

let primes = sieve (fromto 2 1000)
```

Importanța iteratorilor

Separă partea mecanică de cea funcțională
(parcurgerea standard a listei de prelucrarea specifică problemei)

Nu necesită scrierea (repetată) a codului de parcurgere.

Intenția prelucrării poate fi scrisă mai clar (mai direct).

Reduce probabilitatea erorilor la sfârșitul prelucrării (lista vidă)

În multe cazuri, această parcurgere standard poate fi *paralelizată*

Aceste idei au fost preluate *dincolo de limbajele funcționale*

Java 8 introduce interfața `Stream<T>`:
- are iteratori similari (`map`, `filter`, `reduce`) celor descriși
- permite paralelizarea lor
- permite prelucrări cu funcții anonime (lambda expressions)