4 Mulțimi în ML

O *mulțime* e o colecție *neordonată* de elemente de *același tip*.

Pentru liste de orice tip am folosit *aceleași* funcții din modulul `List` (spunem că sunt polimorfe, funcționează la fel indiferent de tip; mai precis, avem *polimorfism parametric*, tipul `'a list` fiind parametrizat cu tipul (arbitrar) `'a` al elementelor). Pentru mulțimi, trebuie să specificăm întâi tipul elementelor cu care dorim să lucrăm, și să creăm un modul. De exemplu, pentru mulțimi de șiruri:

```
module S = Set.Make(String)
```

după care putem folosi funcțiile prefixate cu numele (ales de noi, aici `S`) al modulului, de exemplu `S.add`, `S.isempty`, `S.union`.

Atenție! Spre deosebire de liste, nu putem folosi numele funcțiilor precedate de `Set` (~~`Set.add`~~, etc.)

În definiția mai sus, `String` (cu literă mare) e tot un *modul*, conținând o colecție de funcții care lucrează cu tipul `string` (cu literă mică). `Set.Make` e o funcție care ia ca parametru un modul (`String`) și produce un alt modul (pe care l-am numit `S`). O astfel de funcție pe module se numește *functor* – nu detaliem aici sistemul de module din OCaml.

Putem defini mulțimi cu orice tip de elemente, dacă întâi avem un modul cu anumite caracteristici: un tip `t` și o funcție totală de *ordine* peste elementele tipului. De exemplu, putem defini modulul

```
module Int = struct
  type t = int
  let compare = compare      (* echivalent: let compare = Pervasives.compare *)
end
```

și apoi modulul `module IS = Set.Make(Int)`. Modulul `Char` pentru tipul `char` e predefinit, `Int` nu.

Ca funcție de comparare am ales funcția standard `compare` din modulul de bază `Pervasives`, care e deschis automat (putem deci numi funcțiile simplu, fără prefix). Ea are tipul `'a -> 'a -> int`, deci poate compara elemente de tip arbitrar, returnând `0` dacă sunt egale, negativ dacă primul e considerat mai mic, și pozitiv dacă primul e mai mare (ca și diferența dintre numere). Pentru tipuri structurate, ea e definită să compare întâi prima componentă din cele două valori, iar dacă sunt egale a doua, etc. Cerințele pentru un modul care poate fi folosit ca parametru pentru `Set.Make` (un tip `t` și o funcție `compare: t -> t -> int`) formează o *semnătură* de tip (*signature*) sau *interfață*.

Dacă declarăm *în interpretor* un modul mulțime: `module CS = Set.Make(Char)` vor fi listate toate tipurile și funcțiile din modul, și care formează semnătura (interfața) modulului, de exemplu:

```
module CS: sig
    type elt = Char.t
    type t = Set.Make(Char).t
    val empty : t
    val is_empty : t -> bool
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val singleton : elt -> t
    ...
end
```

Sunt definite numele `elt` pentru tipul elementelor, și `t` pentru tipul mulțime (de elemente de tip `elt`). Vedem că funcția `add` ia un element x și o mulțime A și returnează o mulțime, $A \cup \{x\}$. Subliniem că (la fel ca toate operațiile în stil funcțional), funcția `add` *nu își modifică argumentul* mulțime, ci returnează o valoare obținută din acea mulțime prin adăugarea unui nou element.

Valori de tip mulțime Spre deosebire de liste, ML nu are o notație specială pentru valori de tip mulțime. Putem construi o mulțime pornind de la constanta `empty` (mulțimea vidă cu tipul dat de modul), și funcția `add`, de exemplu `CS.add 'a' (CS.add 'b' (CS.add 'c' CS.empty))`, sau `CS.add 'a' (CS.add 'c' (CS.singleton 'b'))`, unde `singleton` creează o mulțime cu un element. Mulțimile nefiind ordonate, nu contează ordinea în care adăugăm elementele în scrierile de mai sus.

Începând cu OCaml 4.01 putem scrie mai scurt cu operatorul `|>` (aplicare de la stânga la dreapta): `x |> f |> g` înseamnă `g (f x)` : `CS.(empty |> add 'a' |> add 'b' |> add 'c')`

4.1 Construirea unei mulțimi dintr-o listă

Exercițiul 1. Scrieți o funcție cu argument listă de întregi care returnează mulțimea întregilor din listă.

Pentru început, definim modulele necesare pentru o listă de întregi:

```
module Int = struct
  type t = int
  let compare = compare
end
module IS = Set.Make(Int)
```

Începând cu OCaml 4.02, modulul `Set` are funcția `of_list` care ia o listă și returnează mulțimea elementelor ei: `IS.of_list [1;3;2]` . Discutăm cum se poate implementa această funcție.

Putem scrie funcția direct recursiv, folosind obișnuita potrivire de tipare pentru liste:

```
let rec set_of_intlist = function
  | [] -> IS.empty
  | h :: t -> IS.add h (set_of_intlist t)
```

Funcția e simplă dar are problema tipică scrierii în care rezultatul apelului recursiv (mulțimea elementelor din coada listei, `set_of_intlist t`) e folosită mai departe într-o operație (se adaugă capul listei la mulțime): nu e final recursivă, și pentru liste lungi va eșua prin depășirea stivei.

Pentru transformarea în funcție final recursivă (*tail recursive*) folosim obișnuita tehnică de a acumula rezultatul parțial într-un parametru suplimentar, actualizat *înaintea* apelului recursiv, și pentru funcția dorită, o apelăm cu mulțimea vidă ca valoare inițială pentru acumulator:

```
let set_of_intlist lst =
  let rec set_of_il2 res = function
    | [] -> res
    | h :: t -> set_of_il2 (IS.add h res) t
in set_of_il2 IS.empty lst
```

Această prelucrare cu acumulator o face și funcția `List.fold_left`. Mai sus, un pas de prelucrare pornește de la rezultatul parțial `res` și un element (aici `h`, capul listei) și produce noul rezultat parțial `IS.add h res`. Pentru a face același lucru cu `List.fold_left` avem nevoie de o funcție care exprimă acest pas de prelucrare: `fun res e -> IS.add e res` . Putem scrie atunci, echivalent:

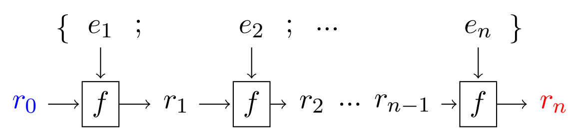
```
let set_of_intlist lst = List.fold_left (fun res e -> IS.add e res) IS.empty lst
```

Apelând în interpretor `let s = set_of_intlist [1;3;2]` nu se va vizualiza mulțimea `s`, deoarece nu e definită o modalitate standard de afișare pentru obiecte compuse arbitrare. Putem însă verifica evaluând `IS.elements s`, care ne arată `[1; 2; 3]`, funcția `elements` returnând lista elementelor mulțimii, crescător în ordinea definită de funcția `compare` specificată pentru tipul elementelor.

4.2 Parcurgerea mulțimilor

Nefiind ordonată, o mulțime nu are un element special cum e capul unei liste. Majoritatea parcurgerilor pe mulțimi se scriu uniform folosind funcția `fold`, asemănătoare ca funcție celor întâlnite la liste. Tipul ei este `(elt -> 'a -> 'a) -> t -> 'a -> 'a`. Ea ia deci trei parametri. Primul e o funcție cu doi parametri: un element al mulțimii, și rezultatul acumulat (de tip arbitrar), returnând un rezultat de același tip. Al parametru al lui `fold` e mulțimea pe care se aplică, și al treilea e valoarea inițială a acumulatorului (de același tip ca și rezultatul final). Parametrii corespund ca ordine și tip celor din `List.fold_right`. Documentația precizează că elementele sunt parcurse în ordine crescătoare, dar aceasta e o particularitate de implementare; adesea, rezultatul nu depinde de ordine. Același principiu de parcurgere se întâlnește pentru tipurile *colecție* (listă, mulțime, tablou) în multe alte limbaje.

Prelucrarea se face ca în figură, reamintind că primul parametru al lui f vine de sus (elementul), și al doilea e acumulatorul:



Parcurgerile s-ar putea scrie și explicit recursiv, pornind de la un element dat de funcțiile `min_elt`, `max_elt`, sau `choose` (oarecare), și continuând cu mulțimea obținută prin eliminarea elementului (`remove`). Deobicei, scrierea (și citirea) codului e mai greoaie, deci preferăm parcurgerea cu `fold`.

Ca exemplu, rescriem câteva funcții standard din modulul `Set`. Nu are sens să facem aceasta într-un program – ele există și le folosim – dar putem ilustra câteva parcugeri cu rezultate simple. Presupunem declarat modulul `module S = Set.Make(String)`

```
let union s1 s2 = S.fold (fun e s -> S.add e s) s1 s2
```

Funcția `union` parcurge mulțimea `s1` (al doilea argument al lui `S.fold`) pornind cu valoarea inițială `s2` (al doilea argument) și la fiecare pas de parcurgere (funcția dată ca prim argument) adaugă la acumulator (inițial `s2`) un element din mulțimea parcursă (`s1`).

Știind că `f = g` dacă și numai dacă `f x = g x` pentru orice argument, observăm că paranteza de mai sus înseamnă de fapt `S.add`, și la fel, nu mai sunt necesare argumentele `s1` și `s2` în ambele părți. Obținem astfel forma extrem de concisă

```
let union = S.fold S.add
```

ceea ce e ne indică faptul că `fold` e o funcție extrem de puternică și utilă. Asemănător, putem scrie:

```
let diff s1 s2 = S.fold (fun e s -> S.remove e s) s2 s1
```

Pentru intersecție, pasul de parcurgere e diferit: pornind de la mulțimea vidă, parcurgem o mulțime și adăugăm doar elementele care se află în cealaltă:

```
let inter s1 s2 = S.fold (fun e r -> if S.mem e s2 then S.add e r else r) s1 S.empty
```

Și pentru funcții cu rezultat boolean putem scrie o implemetare cu `fold`: acumulatorul e răspunsul (adevărat sau fals) obținut până în momentul curent. Pornim cu răspunsul `true`, care poate deveni fals în orice pas dacă elementul curent (din `s1`) nu e membru în `s2`.

```
let subset s1 s2 = S.fold (fun e r -> r && S.mem e s2) s1 true
```

Din logica funcției e evident că odată fals, rezultatul parțial nu mai poate deveni adevărat. Vom discuta în alt curs cum prelucrarea poate fi întreruptă aici folosind *excepții*.

4.3 Produsul cartezian

Pentru a calcula produsul cartezian $A \times B$, va trebui să scriem din nou module și funcții particularizate pentru tipurile A și B. De exemplu, pentru perechi de întregi și caractere:

```
module Int = struct  type t = int  let compare = compare  end
module S1 = Set.Make(Int)
module S2 = Set.Make(Char)
module ICPair = struct
  type t = int * char
  let compare = compare
end
module PS = Set.Make(ICPair)
```

Noțiunea de produs cartezian e folosită și în limbajul ML pentru definirea de tipuri compuse, cum e tipul `int * char` de perechi de întregi și caractere.

Putem privi produsul cartezian ca o matrice, în care pe linii variază primul element iar pe coloane al doilea. Mai întâi parcurgem prima mulțime, păstrând un element fix b din a doua, generând (și adăugând la rezultatul cu valoare inițială `r`) toate perechile variind elementul a_i din prima.

```
let pairwith s1 b r = S1.fold (fun e1 r -> PS.add (e1, b) r) s1 r
```

sau, contractând prin eliminarea lui `r` de ambele părți:

```
let pairwith s1 b = S1.fold (fun e1 -> PS.add (e1, b)) s1
```

Apoi, parcurgem `s2`, aplicând pentru fiecare element funcția `pairwith`, pornind de la mulțimea vidă:

```
let cartprod s1 s2 = S2.fold (fun e r -> pairwith s1 e r) s2 PS.empty
```

sau simplificat:

```
let cartprod s1 s2 = S2.fold (pairwith s1) s2 PS.empty
```

Vizualizăm din nou ca listă, folsind funcția `elements`:

```
cartprod S1.(singleton 1 |> add 2 |> add 3) S2.(singleton 'a' |> add 'b') |> PS.elements
  [(1, 'a'); (1, 'b'); (2, 'a'); (2, 'b'); (3, 'a'); (3, 'b')]
```