Observații despre scrierea în ML

Putem folosi valori, expresii, parametri care sunt funcții

În limbajele funcționale, funcțiile sunt *valori* care pot fi folosite la fel ca valori de orice alt fel (întregi, reali, șiruri, liste, etc.). Astfel, `fun x -> x + 2` e o *valoare* de tip funcție. Ea corespunde notației matematice $f(x) = x + 2$, dar fără să aibă un nume: e funcția anonimă care returnează cu 2 mai mult decât argumentul întreg primit.

Ca orice valoare, o funcție poate fi folosită mai departe în *expresii*. Putem *aplica* funcția la un argument, `(fun x -> x + 2) 3`, obținând valoarea 5.

Funcțiile pot fi *parametri* pentru alte funcții, sau pot fi returnate ca *rezultat* al acestora. Am scris `let comp f g x = f (g x)` pentru compunerea $f \circ g$. Putem scrie atunci
`comp (fun x -> x + 2) (fun x -> 3*x)` care are ca rezultat tot o funcție. Interpretorul nu ne dă o formula explicită, dar putem verifica că obținem aceleași valori ca și pentru rezultatul așteptat `fun x -> 3*x + 2` : fie direct, evaluând de exemplu `comp (fun x -> x + 2) (fun x -> 3*x) 2`, fie definind `let h = comp (fun x -> x + 2) (fun x -> 3*x)` și apoi apelând `h 1`, `h 2`, etc.

Parametrii sunt importanți pentru că ne permit să scriem prelucrari flexibile, fără să știm dinainte valorile prelucrate. La fel de simplu cum funcția `let sqr x = x * x` ne permite să ridicăm la pătrat orice *număr*, funcția `comp` ne permite să compunem *funcții* arbitrare.

Parametri de tip funcție apar în prelucrările pe liste. Pentru a înțelege ca exemplu funcția `List.filter` (care returnează lista acelor elemente care satisfac o condiție) și a putea scrie funcții similare, trebuie să înțelegem că
• o condiție asupra unei valori poate fi scrisă ca o *funcție* cu rezultat boolean
• funcția `filter` e scrisă pentru a funcționa cu *orice* condiție impusă elementelor, fără ca aceasta să fie cunoscută sau definită dinainte

Pentru primul aspect, evidențiem diferența dintre *expresie* și *funcție*:

Astfel, `x mod 3 = 0` e o *expresie* de tipul `bool`, corectă dacă `x` e definit în acel loc, și adevărată dacă `x` se împarte la 3. `fun x -> x mod 3 = 0` e o *funcție* de tipul `int -> bool`. Dacă e aplicată unui întreg, ea returnează adevărat dacă întregul e divizibil cu 3. Putem evalua deci `(fun x -> x mod 3) 6` care este `true`, sau putem da funcției un nume: `let div3 = fun x -> x mod 3 = 0` , echivalent cu `let div3 x = x mod 3 = 0` și apela `div3 6` .

Explicăm acum implementarea funcției `filter`:

```
let rec filter f = function
  | [] -> []                          (* din lista vida nu avem ce selecta *)
  | h :: t -> let ftail = filter f t in  (* selecteaza din coada listei *)
              if f h then h :: ftail else ftail (* adauga h daca e bun *)
```

Ea are ca parametri funcția `f` și lista prelucrată prin potrivire de tipare cu `function ...` . Dacă lista e nevidă, `filter` e apelată recursiv cu aceeași funcție `f` și coada listei, obținând o listă `ftail`. Apoi, testăm dacă `h` (capul listei) satisface funcția parametru `f`, adică `f h` e `true`. Dacă da, la rezultatul parțial `ftail` obținut din coada listei se adaugă `h` ca prim element, altfel returnăm doar lista `ftail`.

E important de înțeles că am scris funcția `filter` *fără a avea definită înainte vreo funcție* `f`. Funcția `f` e parametru, deci se specifică la momentul apelului: `filter (fun x -> x mod 3 = 0) [1;3;5;6]` sau `filter (fun x -> x < 5) [4;6;7;4;8;1]`

E incorect să scriem în `filter`: ~~`if h mod 2 = 0 then h :: ... else ...`~~ pentru că atunci `filter` va selecta doar elementele pare, și nu o vom putea folosi ca funcție generală, indiferent de elementele pe care am dori să le selectăm.

Deasemenea, nu e recomandabil să definim înainte o funcție cu același nume ca și funcția parametru:

~~`let pare x = x mod 2 = 0`~~
~~`let rec filter pare = function ...`~~

sau

~~`let f x = x > 4`~~
~~`let rec filter f = function ...`~~

pentru că poate da impresia greșită (la scrierea sau citirea programului) că `filter` folosește funcția definită înainte, când de fapt ea lucrează cu parametrul `f`, care poate fi orice. Dați un nume specific funcției parametru folosite, și faceți asta imediat înainte de apel, programul devine astfel mai clar:

```
let rec filter f = ... (* functia generala cu parametru functie f *)
let rec prim n = ...   (* determina daca n e prim sau nu *)
filter prim [2;3;4;5;6;7;8;9;10;11;12];; (* aici le folosim *)
```

Valori imutabile, secvențiere și transmiterea parametrilor

Scriem o funcție care returnează lista tuturor cifrelor pare dintr-un număr.

```
let rec even2 dl n =
  let dig = n mod 10
  in let newdl = if dig mod 2 = 0 then dig :: dl else dl
     in if n < 10 then newdl else even2 newdl (n / 10)
let evendig = even2 []
```

Urmărim funcția `even2` cu parametrul suplimentar lista `dl` în care acumulăm cifrele pare găsite. Obținem cu `n mod 10` ultima cifră a numărului `n` și îi dăm un nume `dig` pentru folosire ulterioară. Dacă cifra e pară, ea trebuie adăugată la listă, altfel, lista rămâne neschimbată. Dăm un nume `newdl` listei rezultate. Dacă numărul `n` la care am ajuns are o singură cifră, acesta e chiar rezultatul, altfel apelăm recursiv funcția cu noua valoare a listei și numărul rămas după ștergerea ultimei cifre (`n/10`). Funcția dorită `evendig` e obținută furnizând lista vidă ca valoare inițială pentru apelul la `even2`.

Intenția "dacă cifra e pară o adăugăm la listă; apelăm funcția cu noua listă" e tradusă adesea greșit în cod de forma: ~~`if dig mod 2 = 0 then dig::dl; even2 dl (n/10)`~~

Acest cod, deși pornind de la o intenție logic bună, are multiple probleme. În primul rând, în ML lucrăm cu valori imutabile: operatorii nu pot modifica valoarea operanzilor, ei doar produc noi rezultate. Scriind `dig::dl` *nu modificăm* valoarea lui `dl`, ci creăm *o nouă listă*, cu un element în plus. De aceea, nici secvențierea cu punct-virgulă nu e bună: apelul la `even2` nu va folosi (posibil) lista augmentată, pentru că `dl` are aceeași valoare ca înainte. Folosim o nouă valoare scriind expresia respectivă acolo unde e nevoie de ea (de regulă, ca parametru la o funcție). Dacă expresia e complicată, putem s-o evaluăm întâi, dând valorii un nume, și folosim apoi numele respectiv cu `let nume = expresie in` *expresie-folosind-nume*, ca în exemplul de mai sus.

În fine, codul va genera o eroare de tip: având un `if` fără `else`, valoarea de pe ramura vidă e implicit `()` (nimic) de tipul `unit`, în timp ce pe ramura `then` avem `int list`. Întotdeauna, pe ambele ramuri trebuie să avem aceeași valoare. De aceea, când calculăm o nouă valoare doar pe una din ramuri (`n+1`, `elem::lst`, etc.) trebuie să indicăm și valoarea în celălalt caz (adesea cea veche: `n`, `lst`, etc.).

Să scriem lucrurile mai simplu

Pentru liste, `x :: []` e echivalent cu `[x]`, lista cu un element.
Pentru orice expresie booleană `bexp`, `if bexp then true else false` e echivalent cu `bexp`. Aceasta reiese din definiția lui `if`: dacă condiția `bexp` e adevărată, rezultatul e expresia de pe ramura `then`, aici `true`, altfel (dacă `bexp` e `false`), rezultatul e expresia de pe ramura `else`, aici `false`. În ambele cazuri, rezultatul e chiar valoarea lui `bexp`, deci nu e nevoie de `if ... then ... else .`

Nu folosiți ~~`if List.length lst = 0`~~ ca să testați dacă `lst` e lista vidă. Nu e un simplu test, conceptual (și poate și în implementare) se parcurge toată lista (posibil de mii sau mai multe elemente) pentru a-i calcula lungimea. Comparați direct `if lst = []`. Și mai bine, folosiți potrivirea de tipare. Aveți astfel direct nume pentru părțile listei în cazul listei nevide fără a mai folosi `List.hd` și `List.tl` și compilatorul poate verifica mai bine decât la `if` că nu ați uitat vreun caz.