Prelucrări pe tipuri recursive

Expresiile de diverse feluri (aritmetice, boolene) sunt printre cele mai familiare exemple de recursivitate. O expresie e fie o *expresie atomică* (care nu mai poate fi descompusă: număr, propoziție logică), fie obținută aplicând un *operator* unor sub*expresii* (operanzi).

Evaluarea unei expresii, chiar cu structură complicată: (3 + 9) / ((5 - 3) * (7 - 4)), se face după o regulă simplă: identificăm *ultimul* operator de aplicat (aici împărțirea /), evaluăm cei doi operanzi (expresii *mai simple*) și aplicăm operatorul. Cele scrise (*evaluăm* expresia *evaluând* întâi subexpresiile) ne arată că evaluarea e un procedeu inerent recursiv, urmărind *structura* expresiei. Cazul de bază e o expresie simplă care nu conține operatori (în exemplu, un număr).

Lucrăm în continuare cu un tip recursiv pentru formule propoziționale:

```
type bform = V of string
           | Neg of bform
           | And of bform * bform
```

Folosim un singur operator binar, fiind suficient oricum pentru a exprima orice formulă, și pentru că prelucrările date ca exemplu depind doar de *structura* formulei, nu și de înțelesul (semantica) ei.

Întrucât tipul formulă e definit recursiv, orice prelucrare (netrivială) a unei formule e în mod necesar recursivă, pentru că trebuie să descompună formula conform definiției. Spunem că astfel de prelucrări lucrează prin *descompunere structurală*.

Prelucrare fără rezultat, independentă între subformule

Acestea sunt printre funcțiile cele mai simple – de exemplu tipărirea. Prelucrarea fiecărei subformule se face independent; apelurile pentru mai multe subformule se fac unul după celălalt.

```
let rec print_form = function
  | V s -> print_string s
  | Neg e -> print_string "(~"; print_form e; print_char ')'
  | And (e1, e2) -> print_char '('; print_form e1; print_char '+';
                    print_form e2; print_char ')'
```

Prelucrare cu rezultat, independentă între subformule

Valoarea funcției depinde doar de formulă, fără alți parametri. Rezultatul se obține combinând valorile pentru subformule, fără ca ele să se influențeze reciproc. De exemplu, numărul de operatori din formulă:

```
let rec count_ops = function
  | V s -> 0
  | Neg e -> 1 + count_ops e
  | And (e1, e2) -> 1 + count_ops e1 + count_ops e2
```

Similar, *adâncimea* unei formule, adică numărul maxim de operatori "sub" care se află o propoziție (care se aplică acesteia). O propoziție are adâncime 0 (nu are operatori). Fiecare operator adaugă 1 la adâncime; pentru un operator binar, luăm în considerare adâncimea maximă dintre cei doi operanzi.

```
let rec maxd = function
  | V s -> 0
  | Neg e -> 1 + maxd e
  | And (e1, e2) -> 1 + max (maxd e1) (maxd e2)
```

Prelucrare cu parametru suplimentar, independentă între subformule

Variind problema de mai sus, tipărim adâncimea fiecărei propoziții. Transmitem un parametru care numără adâncimea curentă (numărul de operarori parcurși deja), și e incrementat la fiecare apel:

```
let printd =
  let rec pr1 d = function
  | V s -> print_int d; print_char ' '; print_endline s (* adauga \n *)
  | Neg e -> pr1 (d+1) e
  | And (e1, e2) -> pr1 (d+1) e1; pr1 (d+1) e2
  in pr1 0
```

Prelucrare cu parametru și rezultat, independentă între subformule

În acest caz, transmitem informație atât în jos, printr-un parametru, cât și în sus, ca rezultat. De exemplu, dorim să calculăm *suma* adâncimilor fiecărei propoziții din formulă.

```
let sumdepth =
  let rec sumd d = function
    | V s -> d
    | Neg e -> sumd (d+1) e
    | And (e1, e2) -> sumd (d+1) e1 + sumd (d+1) e2
  in sumd 0
```

Desigur, ca pentru orice funcție, rezultatul poate avea două componente. Putem astfel calcula adâncimea medie, returnând și numărul de propoziții. Împărțirea e întreagă (cu rest), iar cum orice formulă are propoziții, nu riscăm să împărțim la zero.

```
let avgdepth f =
  let rec avd d = function
    | V s -> (1, d)
    | Neg e -> avd (d+1) e
    | And (e1, e2) -> let (n1, s1) = avd (d+1) e1 and (n2, s2) = avd (d+1) e2
                      in (n1 + n2, s1 + s2)
  in let (n, s) = avd 0 f in s / n
```

Prelucrare cu dependențe între subformule

În acest caz, prelucrarea subformulelor nu mai e independentă. Aceasta se poate întâmpla când dorim prelucrarea într-o anumită ordine. Rezultatul pentru o subformulă poate deveni parametru de calcul pentru altă subformulă. De exemplu, pentru a numerota fiecare propoziție, transmitem ca parametru ultimul număr folosit *înainte* de a prelucra formula, și returnăm ultimul număr folosit *după* prelucrarea ei. Funcția dată tipărește numărul dat fiecărei propoziții și returnează ultimul, deci numără propozițiile. Pentru un operator binar, rezultatul pentru subformula din stânga `e1` devine parametru la calculul pentru formula din dreapta `e2`.

```
let countprop =
  let rec cntp n = function
    | V s -> let n1 = n+1 in Printf.printf "%d: %s\n" n1 s; n1
    | Neg e -> cntp n e
    | And (e1, e2) -> cntp (cntp n e1) e2
  in cntp 0
```

Invers, putem număra fiecare operator. Pentru o propoziție, numărul dat ca parametru e și cel returnat. Operatorul curent îl putem număra înainte sau după ce numărăm operatorii din subformule:

```
let cntopfirst =
  let rec cntop n = function
    | V s -> n
    | Neg e -> cntop (n+1) e
    | And (e1, e2) -> cntop (cntop (n+1) e1) e2
  in cntop 0

let cntoplast =
  let rec cntop n = function
    | V s -> n
    | Neg e -> 1 + cntop n e
    | And (e1, e2) -> 1 + cntop (cntop n e1) e2
  in cntop 0
```

În oricare din cazuri, am putea folosi în cadrul unei expresii compuse (`Neg, And`) valorile returnate pentru subexpresii, dacă problema o cere.