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 de formă complexă: $(3+9)/((5-3)*(7-4))$, se face după o regulă simplă: găsim *ultimul* operator de aplicat (aici împărțirea /), evaluăm cei doi operanzi (expresii *mai simple*) și aplicăm operatorul. Evaluarea e deci un procedeu recursiv, care urmărește *structura* expresiei (tot recursivă, prin definiție). Cazul de bază e o expresie simplă fără 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*.

1 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 prt_form = function
  | V s -> print_string s
  | Neg e -> print_char '~'; prt_form e
  | And (e1, e2) -> print_char '('; prt_form e1; print_char '&'; prt_form e2; print_char ')'
```

2 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)
```

3 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 print_depth =
  let rec print_d d = function
  | V s -> print_int d; print_char ' '; print_endline s (* adauga \n *)
  | Neg e -> print_d (d+1) e
  | And (e1, e2) -> let d1 = d+1 in print_d d1 e1; print_d d1 e2
  in pr1 0
```

4 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, determinăm dacă o anumită propoziție (dată printr-un șir de caractere) apare cu polaritate negativă (sub un număr impar de negații) într-o formulă. Pentru aceasta, transmitem de sus în jos un boolean cu această informație (subformula curentă e sub un număr impar de negații? – inițial, fals):

```
let isneg v =  (* v e fixat pentru functia interioara *)
  let rec isnv neg = function (* neg: numar impar de negatii? *)
    | V s -> neg && (s = v)  (* polaritate negativa si e propozitia cautata *)
    | Neg e -> isnv (not neg) e (* schimba polaritatea *)
    | And (e1, e2) -> isnv neg e1 || isnv neg e2   (* macar pe o ramura *)
  in isnv false (* initial, polaritatea nu e negativa *)
```

5 Simplificări: prelucrare la revenirea din recursivitate

Când parcurgem o structură recursivă e important să deosebim dacă facem prelucrarea la parcurgerea în jos sau la revenire. Când simplificăm o formulă, e important să o facem (și) la *revenirea* din apel (după prelucrarea subformulelor), pentru a folosi eventualele rezultate deja simplificate.

Să considerăm formule care au și variabile propoziționale și constante boolene:

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

Scriem o funcție care simplifică după regulile $A \wedge F = F$, $A \wedge T = A$, oricare ar fi formula A.

```
let rec simpand = function          (* incomplet, simplifica DOAR in jos ! *)
  | And (_, B false) | And (B false, _) -> B false (* nu mai parcurgem restul *)
  | And (e, B true) | And (B true, e) -> simpand e (* simplifica mai departe in jos *)
  | And (e1, e2) -> And (simpand e1, simpand e2) (* simplifica DOAR in jos! *)
  | Neg e -> Neg (simpand e)
  | p -> p     (* orice altceva: B _, V _ *)
```

Pe prima ramură e bine că detectarea tiparului se face la parcurgerea în jos, nu are sens să mai prelucrăm subformula care dispare prin conjuncție cu *fals*. Și pe a doua ramură ($e \wedge T = e$) e suficient apelul recursiv care va simplifica subformula `e`. Pe a treia ramură însă pierdem ocazii de simplificare. Apelată cu `And(V "p", And(B true, B true))`, pe nivelul doi de apel recursiv, funcția simplifică `And(B true, B true))` la `B true`. Dar fără a căuta tiparul și la revenire, nu se mai simplifică și rezultatul `And(V "p", B true)`.

Rescriem deci funcția ca să verifice tiparul căutat *după* ce subformulele au fost deja simplificate:

```
let rec simpand = function
  | And (_, B false) | And (B false, _) -> B false (* nu mai parcurgem e *)
  | And (e1, e2) -> (match (simpand e1, simpand e2) with
                   | (e, B true) | (B true, e) -> e
                   | (_, B false) | (B false, _) -> B false (* F poate veni de jos *)
                   | (e1, e2) -> And (e1, e2))     (* orice altceva *)
  | Neg e -> Neg (simpand e)
  | p -> p
```

Acum obținem simplificarea completă pâna la `V "p"`.

Am scris și funcții pentru *evaluarea* unor formule (aritmetice în cursul 2, apoi la cursul 6 pentru formule boolene dată fiind o atribuire/interpretare). Acestea sunt cazuri particulare de simplificare până la o singură valoare. Codul urmează aceeași secvență: evaluează întâi subformulele, și apoi combină rezultatele (aici, o simplă operație pe întregi sau booleni).

6 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.