Logică și structuri discrete

Recursivitate

Marius Minea
marius@cs.upt.ro

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

3 octombrie 2016

Recapitulare

Am revăzut: funcții (injective, surjective, bijective, inversabile)

Am definit funcții într-un limbaj de programare funcțional

Funcția e dată prin formulă, dar și prin domeniu și codomeniu:
tipuri în limbajele de programare
Tipurile ne spun pe ce fel de valori poate fi folosită o funcție

Recapitulare

Am revăzut: funcții (injective, surjective, bijective, inversabile)

Am definit funcții într-un limbaj de programare funcțional

Funcția e dată prin formulă, dar și prin domeniu și codomeniu:
tipuri în limbajele de programare
Tipurile ne spun pe ce fel de valori poate fi folosită o funcție

```
# let comp g f x = g (f x);;
val comp: ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

`f` are tipul `'c -> 'a` și `g` are tipul `'a -> 'b`
deci domeniul de valori al lui `f` e domeniu de definiție al lui `g`
compunerea are tipul `'c -> 'b` `'a`, `'b`, `'c` pot fi orice tip

Funcțiile pot avea ca argumente și/sau rezultat alte funcții

Prin compunerea funcțiilor rezolvăm probleme mai complexe:
`f` produce un rezultat, `g` îl prelucrează mai departe
calculul complet: $g \circ f$ (aplică `f`, apoi `g`)

Recursivitate

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

Recursivitatea e fundamentală în informatică:
reduce o problemă la un caz mai simplu al *aceleiași probleme*

$\Rightarrow$ o unealtă simplă și puternică în rezolvarea problemelor

Calculul expresilor aritmetice

O *expresie* (puțin mai) complicată:
(2 + 3) * (4 + 2 * 3) - 5 * 6 / (7 - 2) + (4 + 3 - 2) / (7 - 3)

Pentru a calcula, trebuie să înțelegem *structura* expresiei
(nu se vede tot timpul ușor într-un șir de caractere)

E *suma* a două subexpresii (+ e calculat ultimul):

(2 + 3) * (4 + 2 * 3) - 5 * 6 / (7 - 2)
\+ (4 + 3 - 2) / (7 - 3)

Apoi calculăm *expresiile mai simple*
(2 + 3) * (4 + 2 * 3) - 5 * 6 / (7 - 2) = 44
(4 + 3 - 2) / (7 - 3) = 1
44 + 1 = 45

Calculul celor două subexpresii: după *aceleași reguli*

O noțiune recursivă: expresia

Ce e o expresie numerică?

int + int 5 + 2
int – int 2 − 3
int * int -1 * 4
int / int 7 / 3

și mai simplu ? Da: int (5 e un caz particular de expresie)

și mai complicat? Da:
int * (int + int)
(int - int) / int
...

Putem scrie un număr finit de reguli ?

Expresia, definită recursiv

O *expresie*: $\begin{cases} \text{întreg} \\ \textit{expresie} + \textit{expresie} \\ \textit{expresie} - \textit{expresie} \\ \textit{expresie} * \textit{expresie} \\ \textit{expresie} \ / \ \textit{expresie} \end{cases}$

Am descris expresia printr-o *gramatică* – detalii în alt curs;
așa se descriu limbajele de programare

Ne interesează modul în care sunt structurate calculele, nu sintaxa concretă, deci nu tratăm paranteze, spații, etc.

O problemă nerezolvată: “problema $3 \cdot n + 1$”

Fie un număr pozitiv n:
dacă e par, îl împărțim la 2: $n/2$
dacă e impar, îl înmulțim cu 3 și adunăm 1: $3 \cdot n + 1$

$$f(n) = \begin{cases} n/2 & \text{dacă } n \equiv 0 \mod 2 \\ 3 \cdot n + 1 & \text{altfel (dacă } n \equiv 1 \mod 2) \end{cases}$$

Se ajunge la 1 pornind de la orice număr pozitiv ?

= Conjectura lui Collatz (1937), cunoscută sub multe alte nume

Exemple:
$3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$
$11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Câți pași până la oprire?

Vrem să definim funcția $p : \mathbb{N}^* \to N$ care exprimă numărul de pași până la oprire.

Nu avem o formulă cu care să definim $p(n)$ direct.
Dar dacă șirul n, $f(n)$, $f(f(n)), \ldots$ ajunge la 1, numărul de pași pornind de la n e cu unul mai mare decât de la $f(n)$:

$$p(n) = \begin{cases} 0 & \text{dacă } n = 1 \\ 1 + p(f(n)) & \text{altfel (dacă } n > 1) \end{cases}$$

Funcția p a fost definită *recursiv*: e folosită în propria definiție

Problema $3 \cdot n + 1$ în ML

```
let f n = if n mod 2 = 0 then n / 2 else 3 * n + 1
```

În ML, `if c then e1 else e2` e o *expresie* (condițională)
dacă `c` e adevărată, are valoarea lui `e1`, altfel valoarea lui `e2`

```
let rec p n = if n = 1 then 0 else 1 + p (f n)
```

Cuvintele cheie `let rec` introduc o *definiție recursivă*:
funcția p e folosită (apelată) în propria definiție

Potrivirea de tipare

Putem scrie aceeași funcție și așa:

```
let rec p = function
  | 1 -> 0
  | n -> 1 + p (f n)
```

Citim această definiție astfel:

Definim p ca funcție, pe următoarele cazuri:
– dacă argumentul e 1, valoarea funcției e 0
– dacă argumentul are orice altă valoare (o notăm `n`), valoarea funcției e `1 + p (f n)`

Cuvintele cheie `fun` și `function` se folosesc diferit!
cu `fun x1 x2 ... ->` *expresie* putem scrie *orice expresie* pentru valoarea funcției (și putem avea oricâți parametri x1, x2 ...)
cu `function` definim o funcție prin *potrivire de tipare*, cu *un* parametru *implicit* (NU am scris ~~let p n = ...~~ ci denumim parametrul în stânga lui -> pe ramura unde avem nevoie de el

Fiecare ramură indică rezultatul (în dreapta) returnat dacă argumentul se potrivește cu tiparul din stânga.

Potrivirea de tipare (cont.)

Argumentul *implicit* care e potrivit cu tiparul poate fi:

- o constantă (aici, 1)
- o valoare structurată (pereche, listă cu cap/coadă, etc.)
- un identificator (nume) care indică tot argumentul (oricare ar fi)

Potrivirile se încearcă în ordinea indicată, până la prima reușită.
Tipizarea puternică permite avertismente dacă uităm un caz.

Ex.: o funcție care ia triplete de întregi și dă suma componentelor până la primul zero. Identificatorul special _ se potrivește cu orice:

```
let sumto0 = function
  | (0, _, _) -> 0
  | (x, 0, _) -> x
  | (x, y, z) -> x + y + z
```

dacă prima componentă e 0, rezultatul e 0, indiferent de celelalte
altfel, dacă a doua componentă e 0, adunăm doar prima (nu și a treia)
altfel, primele două sunt nenule, și le sumăm pe toate trei

```
let sumto0 (x, y, z) = (* echivalent *)
      if x = 0 then 0 else if y = 0 then x else x + y + z
```

Mecanismul apelului recursiv

În calculul recursiv

Fiecare apel face *"în cascadă"* *un nou apel*, până la cazul de bază

Fiecare apel execută *același cod*, dar cu *alte date*
(valori proprii pentru parametri)

Ajunși la cazul de bază, toate apelurile făcute sunt încă *neterminate*
(fiecare mai are de făcut adunarea cu rezultatul apelului efectuat)

Revenirea se face *în ordine inversă* apelării
(apelul cu indice 0 revine primul, apoi cel cu indice 1, etc.)

În interpretor, vizualizați apelurile și revenirea cu directiva
`#trace` *numefuncție*
reveniți la normal cu `#untrace` *numefuncție*

Șiruri recurente

progresie aritmetică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0\text{)} \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

Exemplu: $1, 4, 7, 10, 13, \ldots$ ($b = 1$, $r = 3$)

progresie geometrică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0\text{)} \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$$

Exemplu: $3, 6, 12, 24, 48, \ldots$ ($b = 3$, $r = 2$)

Definițiile de mai sus nu calculează x_n *direct* (deși se poate)
ci *din aproape în aproape*, folosind x_{n-1}.

șirul x_n e *folosit în propria definiție* $\Rightarrow$ recursivitate / recurență

Programăm: progresia aritmetică

întâi o progresie aritmetică cu baza și rația fixate:

$$x_0 = 3,\ x_n = x_{n-1} + 2 \text{ (pentru } n > 0)$$

Noțiunea recursivă (șirul) devine o *funcție*

Valorile de care depinde (indicele) devin *argumentele* funcției

```
let rec aritpr_3_2 = function
  | 0 -> 3
  | n -> 2 + aritpr_3_2 (n-1)
```

Cum parametrizăm funcția cu bază și rație ?

Definiții locale în ML

Până acum: definiții *globale*: `let` *noțiune* $=$ *expresie* adică

`let identificator =` *expresie* sau

`let functie param1 ... paramN =` *expresie*

Uneori, sunt utile definiții auxiliare. De exemplu, aria triunghiului de laturi a, b, c e $\sqrt{p(p-a)(p-b)(p-c)}$ unde $p = \frac{a+b+c}{2}$.
În ML, putem scrie

```
let arie a b c =
  let p = (a +. b +. c) /. 2. in
  sqrt (p *. (p -. a) *. (p -. b) *. (p -. c))
```

Definiția e tot de forma `let` *funcție arg1 ... argN* $=$ *expresie*
unde *expresie* are o nouă formă:

`let` *noțiune* $=$ *expr_def* `in` *expr_val*

Aceasta e o *expresie* cu valoarea dată de *expr_val*, în care numele definit în stânga lui = *(noțiune)* ia sensul dat în dreapta *(expr_def)*.

Definiții locale (exemplu cu funcții)

Putem scrie o funcție mai generală cu baza și rația ca parametri:

```
let rec aritpr base step = function (* ia un indice si da: *)
 | 0 -> base
 | n -> step + aritpr base step (n-1)
```

În apelul recursiv, `base` și `step` sunt aceleași, și vedem de două ori expresia `aritpr base step`, o funcție de 1 argument (indicele).

Rescriem cu o definiție locală numind funcția de 1 argument:

```
let aritpr base step =
  let rec ap1 = function
    | 0 -> base
    | n -> step + ap1 (n-1)
  in ap1
```

E la fel ca `let aritpr base step = ap1`, cu `ap1` definit doar local

Funcția `ap1` (de întreg) vede parametrii `base` și `step` ai lui `aritpr`

Putem defini apoi funcții care corespund unor progresii individuale:

```
let aritpr_3_2 = aritpr 3 2 (* progr. baza 3, ratia 2 *)
# aritpr_3_2 4
- : int = 11                (* termenul 4 al progresiei *)
```

Recursivitate: exemple

Recursivitatea e fundamentală în informatică:
reduce o problemă la un caz mai simplu al *aceleiași* probleme

obiecte: un *șir* e $\begin{cases} \text{un singur element} & \bigcirc \\ \text{un element urmat de un } \textit{șir} & \bigcirc\overbrace{\bigcirc\bigcirc\bigcirc}^{\text{șir}} \end{cases}$

ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

acțiuni: un *drum* e

$\begin{cases} \text{un pas} & \longrightarrow \\ \text{un } \textit{drum} \text{ urmat de un pas} & \overbrace{\longrightarrow\longrightarrow\longrightarrow}^{\text{drum}}\longrightarrow \end{cases}$

ex. parcurgerea unei căi într-un graf

Tipuri recursive

Definim un *tip recursiv* care să reprezinte structura unei expresii (împreună cu eventualul operator pentru calcul)

```
type expr = I of int
          | A of expr * expr | S of expr * expr
          | M of expr * expr | D of expr * expr
```

Am definit un tip cu mai multe *variante*.
Fiecare din ele trebuie scrisă cu un *constructor de tip* (etichetă), ales de noi: `I`,`A`, etc. (orice identificator cu literă mare)

Notația `expr * expr` reprezintă *produsul cartezian*, deci o pereche de două valori de tipul `expr`

Tipul `expr` e *recursiv* (o valoare de tip expresie poate conține la rândul ei componente de tip expresie)

Expresia $(2 + 3)$ * 5 se reprezintă ca `M (A(I 2, I 3), I 5)`

Evaluarea recursivă a unei expresii

Lucrul cu o valoare de tip recursiv se face prin *potrivire de tipare* (engl. pattern matching), pentru fiecare variantă din tip

```
let rec eval = function
  | I i -> i
  | A (e1, e2) -> eval e1 + eval e2
  | S (e1, e2) -> eval e1 - eval e2
  | M (e1, e2) -> eval e1 * eval e2
  | D (e1, e2) -> eval e1 / eval e2
```

Evaluăm o expresie de acest tip: `eval (M (A(I 2, I 3), I 5))` returnează 25.

Când un tip de date e definit recursiv
- funcțiile care îl prelucrează vor fi natural recursive
- deobicei cu câte un caz pentru fiecare variantă a tipului respectiv

Elementele unei definiții recursive

1. *Cazul de bază* (*NU* necesită apel recursiv)
= cel mai simplu caz pentru definiția (noțiunea) dată, definit direct
    termenul inițial dintr-un șir recurent: x_0
    un element, în definiția: șir = element   sau   șir + element

E o *EROARE* dacă lipsește cazul de bază (apel recursiv infinit!)

2. *Relația de recurență* propriu-zisă
– definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni

3. Demonstrație de *oprire a recursivității* după număr finit de pași
(ex. o mărime nenegativă care descrește când aplicăm definiția)
– la șiruri recurente: indicele (≥ 0 dar mai mic în corpul definiției)
– la obiecte: dimensiunea (definim obiectul prin alt obiect mai mic)

Sunt recursive, și corecte, următoarele definiții ?

? $x_{n+1} = 2 \cdot x_n$
? $x_n = x_{n+1} - 3$
? $a^n = a \cdot a \cdot \ldots \cdot a$ (de n ori)
? o frază e o înșiruire de cuvinte
? un șir e un șir mai mic urmat de un alt șir mai mic
? un șir e un caracter urmat de un șir

O definiție recursivă trebuie să fie *bine formată* (v. condițiile 1-3)

ceva nu se poate defini doar în funcție de sine însuși
se pot utiliza doar noțiuni deja definite
nu se poate genera un calcul infinit (trebuie să se oprească)

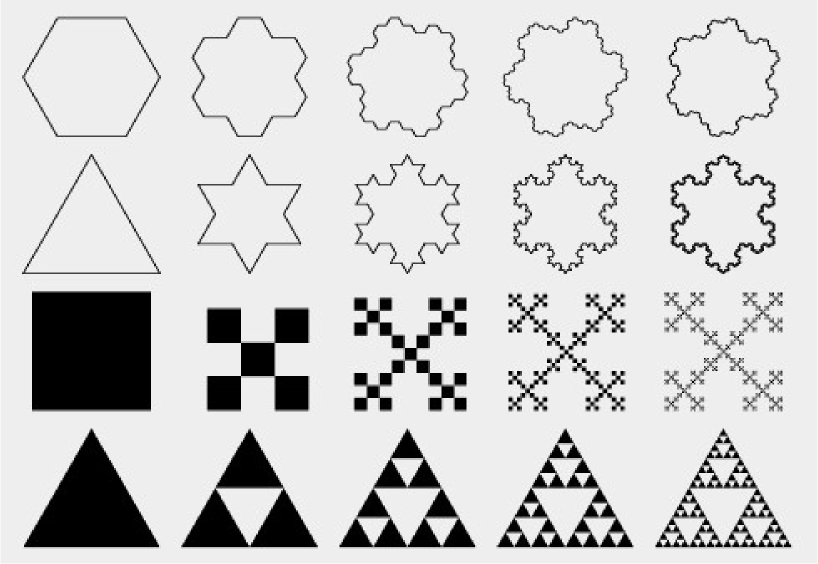
Fractali

Figuri geometrice în care o parte a figurii e similară întregului
acesta e aspectul *recursiv*

Apar în natură, sau pot simula artificial figuri din natură

Analiza lor are aplicații în diverse domenii: geografie/geologie, medicină, prelucrarea semnalelor, electrotehnică (microantene), etc.

Exemple simple de fractali



Imagine: `http://mathworld.wolfram.com/Fractal.html`

Generarea recursivă a unui fractal

Scriem o *funcție* pentru noțiunea recursivă (figura)

Caracteristicile figurii devin *parametrii* funcției
dimensiunea, poziția (coordonatele), orientarea, etc.

Apelul funcției va *desena* figura
(sau va produce comenzile de desenare)

Să desenăm simplu

SVG = Scalable Vector Graphics:
un format de imagine bazat pe XML

Comenzi simple:

m x y (moveto): mută punctul curent

l x y (lineto): desenează linie din punctul curent în cel indicat

versiuni cu coordonate absolute (M, L) și relative (m, l)

caz particular:
h x linie orizontală (de lungime x)
v y linie verticală (de lungime y)

Structura XML a unui fișier SVG are elemente standard la început/sfârșit

Scrierea/tipărirea în OCaml

Funcții dedicate pentru fiecare tip:
`print_int`, `print_float`, `print_string` etc.

```
print_int 5
print_float 3.4
```

Funcția de tipărire formatată `Printf.printf` (asemănător cu C)
Dacă o folosim des, *deschidem* modulul `Printf` și scriem simplu:

```
open Printf
printf "un intreg: %d\n" 5
printf "un real %f si inca unul: %f\n" 2.3 4.7
```

Putem defini atunci:
`let lineto x y = printf "l %.2f %.2f" x y`
(tipărește coordonatele cu două zecimale)
sau mai simplu: `let lineto = printf "l %.2f %.2f"`

De știut

Să recunoaștem și definim *noțiuni recursive*

Să recunoaștem dacă o definiție recursivă e *corectă*
(are caz de bază? se oprește recursivitatea?)

Să rezolvăm probleme scriind *funcții* recursive
cazul de bază + pasul de reducere la o problemă mai simplă

Să definim și folosim *tipuri recursive*