Cursul 3 - Definirea functiilor, Liste

- Lambda calcul - scurta introducere
- Lambda expresii in Haskell
- Definiţii recursive
- Definiţii locale
- Operaţii cu liste
- O altă reprezentare a listelor
  - Generatori
  - Gărzi

Lambda calcul – scurta introducere

Lambda calculul se bazează pe două operații de bază:

- Aplicația:
  - F.A care se scrie de obicei FA
    - F privit ca algoritm, A considerat input
    - In particular FF (recursie)

- Abstracția:
  - Daca M = M[x] este o expresie, atunci functia x →M[x] se noteaza λx.M[x]

Lambda calcul

- Variabile
  - libere: in λx.yx variabila y este libera
  - legate: in λx.yx variabila x este legata
- Substitutie: M[x:=N] aparitiile libere ale lui x se inlocuesc cu N
  - yx(λx.x)[x:=N] = yN(λx.x)
- (λx.M[x])N = M[x:=N]
  - λx.2*x+1)3 = 2*3 + 1

Lambda calcul - formal

- Multimea λ-termilor, notata Λ se construeste din multimea infinita de constante $C = \{c, c', \ldots\}$ si multimea infinita de variabile $V = \{v, v', \ldots\}$ astfel:
  - $c \in C \Rightarrow c \in \Lambda, x \in V \Rightarrow x \in \Lambda$
  - $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$
  - $x \in V, M \in \Lambda \Rightarrow (\lambda x M) \in \Lambda$
- Notatii:
  - $FM_1M_2\ldots M_n$ noteaza $(\ldots((FM_1)M_2)\ldots M_n)$
  - $\lambda x_1x_2\ldots x_n.M$ noteaza $\lambda x_1(\lambda x_2(\ldots\ (\lambda x_n(M))\ldots))$

Lambda calcul - formal

- Multimea variabilelor libere ale lui M, notata FV(M) este:
  - FV(x) = {x}
  - FV(MN) = FV(M) $\cup$ FV(N)
  - FV(λx.M) = FV(M) -{x}
- Substitutie: M[x:=N] este M in care orice aparitie libera a lui x se inlocueste cu N
- M este term inchis, sau combinator, daca FV(M) = Φ. Multimea combinatorilor se noteaza Λ^0

- Combinatorii standard:
  - I = λx.x K = λxy.x S = λxyz.xz(yz)
  - IM = M KMN = M SMNL = ML(NL)

Teoria ecuatiilor in Λ

- Axiome

(β) $(\lambda x.M)N = M[x:=N]$ pentru orice $M,N \in \Lambda$

$M = M$

$M = N \Rightarrow N = M$; $M = N, N = L \Rightarrow M = L$

$M = M' \Rightarrow MZ = M'Z$, $ZM = ZM'$

(ξ) $M = M' \Rightarrow \lambda x.M = \lambda x.M'$

(α) $\lambda x.M = \lambda y.M[x:= y]$

- Daca se poate demonstra ca M = N, spunem ca M si N sunt β-convertibili
- Notam $M \equiv N$ daca M si N sunt aceeasi modulo redenumirea variabilelor legate: $(\lambda x.x)z \equiv (\lambda y.y)z$

Reprezentarea recursiei in lambda calcul

Teorema de punct fix

(1) $\forall F \in \Lambda\ \exists X \in \Lambda$ astfel incat $FX = X$

(2) Exista combinatorul de punct fix

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

astfel incat $\forall F \in \Lambda\ F(YF) = YF$

Demonstratie

(1) Fie $W \equiv \lambda x.F(xx)$ si $X \equiv WW$ Atunci

$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv F(X)$

(2) La fel

Reprezentarea numerelor si operatiilor

Definitie. $F^0(M) = M$, $F^{n+1}(M) = F(F^n(M))$

Numerale Church: $c_0, c_1, \ldots \quad c_n \equiv \lambda fx.\ f^n x$

Operatii: $A_+ \equiv \lambda xypq.xp(ypq)$

$A_* \equiv \lambda xyz.x(yz)$

$A_{exp} \equiv \lambda xy.yx$

Teorema Pentru orice numar natural n, m:

(1) $A_+\ c_n c_m = c_{n+m}$

(2) $A_* c_n c_m = c_{n*n}$

(3) $A_{exp}\ c_n c_m = c_{n^m}$ (m diferit de zero)

Reprezentarea functiilor

Definitie.

(1) `true` ≡ K ≡ λxy.x, `false` ≡ K* ≡ λxy.y

`if` B `then` P `else` Q poate fi reprezentat prin BPQ

(2) Perechea ordonata: [M, N] ≡ λz.MN

(3) Numerale: $\underline{0}$ ≡ I ≡ λx.x, $\underline{n+1}$ ≡ [false, $\underline{n}$]

Lema Exista combinatorii S^+, P^-, zero astfel ca pentru orice numar natural n au loc: $S^+\underline{n} = \underline{n+1}$, $P^- \underline{n+1} = \underline{n}$, Zero $\underline{0}$ = true, Zero $\underline{n+1}$ = false

Demonstratie: $S^+ \equiv \lambda x.[false, x]$, $P^- \equiv \lambda x.xfalse$, Zero ≡ λx.xtrue

Reprezentarea functiilor

Definitie. O functie numerica $f : N^p \rightarrow N$ este λ-definibila daca exista un combinator F astfel ca

$F\ \underline{n_1}\ \underline{n_2} \ldots\ \underline{n_p} = \underline{f(n_1, n_2, \ldots, n_p)}$

Teorema Functiile recursive sunt λ-definibile:

(1) Functiile initiale(zero, succesor, proiectia) sunt λ-definibile

(2) Clasa functiilor λ-definibile este inchisa la compozitie(compunere)($f(n) = g(h_1(n),\ldots,h_m(n))$)

(3) Clasa functiilor λ-definibile este inchisa la recursie primitiva ($f(0,n) = g(n)$, $f(k+1, n) = h(f(k,n), k, n)$)

(4) Clasa functiilor λ-definibile este inchisa la minimizare ($f(n) = \mu\, m[g(n, m) = 0]$)

Definirea funcţiilor

- Lambda expresii
  - Sintaxa:

```
\x -> exp
```

  - Expresia exp conţine x; poate fi o nouă lambda expresie
  - Exemple de funcţii definite ca lambda expresii:

```
λx -> x + x
λx -> (λy -> x + y)
λx -> (λy -> x*x + y*y)
```

Exemple

```
Main> (\x->x+x)8
16
Main> (\x->(\y->x*x+y*y)) 3 4
25
Main> (\x y->x*x+y*y) 3 4
25

Main> (\x-> sum[1..x]) 4
10

const :: a -> ( b -> a)
const x = \_ -> x

Main> const 1 2
1
Main> const [1,2,3,4] 2
[1,2,3,4]
Main> const 3 (2,0)
3
Main> const False 4
False
```

Exemple

- Folosim funcţia **map** pentru a obţine primele n numere impare:

```
map :: (a -> b) -> [a] -> [b]

odds :: Int -> [Int]
odds n = map f [0..n-1]
   where f x = x*2 +1 - definitie locala
```

Alternativă (lambda expresie):

```
odds n = map(\x -> x*2 + 1)[0..n-1]
```

Definirea funcţiilor

- Secţiuni
  - Funcţiile cu două argumente pot fi utilizate infix(ca operatori): `12 `div` 3`
  - Operatorii (+, *, etc.) pot fi utilizaţi ca funcţii:

    `(+) 3 4  (3+) 4  (+4) 3`
  - Dacă @ este un operator atunci expresiile de forma `(@)`, `(x@)`, `(@y)` se numesc secţiuni şi sunt funcţii:

```
(@) = \x ->(\y -> x @ y)
(x +) = \y -> x @ y
(+ y) = \x -> x @ y
```

Exemple

```
Hugs.Base> :type (+)
(+) :: Num a => a -> a -> a
Hugs.Base> :type (1+)
(1 +) :: Num a => a -> a
Hugs.Base> :type (+2)
flip (+) 2 :: Num a => a -> a
Hugs.Base> :type (True&&)
(True &&) :: Bool -> Bool
Hugs.Base> :type (||False)
flip (||) False :: Bool -> Bool
Hugs.Base > :t flip
flip :: (a -> b -> c) -> b -> a -> c
```

Aplicaţii ale secţiunilor

- Construirea de funcţii simple

```
(1+) = \y -> 1+y
(1/) = \y -> 1/y
(*2) = \x -> x*2
(/2) = \x -> x/2
```

- Aflarea tipului operatorilor
- Utilizarea operatorilor ca argumente pentru alte funcţii

Definiţii recursive

- O funcţie care are, în expresia ce defineşte valoarea sa, numele său este o funcţie recursivă

```
fact  :: Integer -> Integer
fact n = if n == 0 then 1 else n*fact(n-1)


fact 1 =                                    (definitie)
if 1 == 0 then 1 else 1*fact(1-1)=
1 * fact(1-1) =
1*if (1-1) == 0 then 1 else (1-1)*fact((1-1)-1)=
1*if 0 == 0 then 1 else (1-1)*fact((1-1)-1)=
1*1 =
1
```

Definiţii recursive

```
fact(-1) = (definitie)
if -1 == 0 then 1 else (-1)*fact(-1-1)=
(-1) * fact(-1-1) =
(-1)*if (-1-1) == 0 then 1 else (-1-1)*fact((-1-1)-1)=
(-1)*((-2)*fact(-1-1-1))
```

…

- Redefinim funcţia:

```
fact      :: Integer -> Integer
fact n    |n<0 = error "argument negativ"
          |n == 0 = 1
          |n>0  = n*fact(n-1)

error :: String -> a
```

Definiţii recursive

```
Main> fact 33
8683317618811886495518194401280000000
Main> fact 0
1
Main> fact (-3)

Program error: argument negativ
```

Definiţii recursive

- O funcţie recursivă se defineşte astfel:
  - Se defineşte tipul:

```
produs :: [Int] -> Int
```

  - Se enumeră cazurile:

```
produs[]          =
produs(n:ns)      =
```

  - Se definesc cazurile de bază:

```
produs[]          = 1
produs(n:ns)      =
```

  - Se definesc celelalte cazuri

```
produs[]          = 1
produs(n:ns)      = n*produs ns
```

  - Generalizări, simplificări:

```
produs :: Num a =>[a] -> a
```

Liste

- Tipul listă

```
[1,2,3,4] :: Num a => [a]
[[1,2,3,4],[1,2]] :: Num a => [[a]]
[(+),(*),(-)] :: Num a => [a -> a -> a]
```

- Constructorul(cons) :

```
(:) :: a -> [a] -> [a]
[1,2,3] = 1:(2:(3:[])) = 1:2:3:[]
```

- Funcţia `null`

```
null       :: [a] -> Bool
null[]     = True
null x:xs  = False
```

```
Hugs.Base> null []
True
Hugs.Base> null [1,2]
False
```

Operaţii cu liste

- Concatenare: ++

```
(++)     :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)

[1,2] ++ [3,4,5] =                  {notatie}
(1:(2:[]))++(3:(4:(5:[]))) =        {def ++, ec2}
1:((2:[])++(3:(4:(5:[])))) =        {def ++, ec2}
1:(2:([]++(3:(4:(5:[]))))) =        {def ++, ec1}
1:(2:(3:(4:(5:[]))))=               {notatie}
[1,2,3,4,5]
```

Operaţii cu liste

- Concatenarea (++) este asociativă iar [] este element neutru:

```
(xs ++ ys) ++ zs = xs ++ ( ys ++ zs)
xs ++ [] = [] ++ xs
```

- Funcţia `concat` concatenează o listă de liste:

```
concat :: [[a]] -> [a]
concat [] = []
concat(xs:xss) = xs ++ concat xss
```

```
Hugs.Base> concat [[1,2,3],[4,5],[6],[7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

Funcţia `reverse`

- Funcţia `reverse` inversează elementele unei liste:

```
reverse :: [a] -> [a]
reverse[] = []
reverse (x:xs) = reverse xs ++ [x]
```

- Funcţia `reverse` realizează inversarea unei liste de lungime n în n(n-1) paşi

```
Hugs.Base> reverse [1,2,3,4,5,6]
[6,5,4,3,2,1]
Hugs.Base> reverse (reverse [1,2,3,4,5,6])
[1,2,3,4,5,6]
Hugs.Base> reverse "epurasul usa rupe"
"epur asu lusarupe"
```

Funcţia `length`

```
length :: [a] -> Int
length[] = 0
length(x:xs) = 1 + length xs
```

- Se poate dovedi (de la definiţiile ++ şi `length`):

  `length(xs++ys) = length xs + length ys`

- Funcţia `length` calculează lungimea unei liste în n paşi indiferent de natura elementelor

```
Hugs.Base> length [1,2]
2
Hugs.Base> length [undefined, undefined]
2
```

Funcţiile `head, tail`

```
head :: [a] -> a
head[x:xs] = x
tail :: [a] -> [a]
tail(x:xs) = xs
```

- Sunt operaţii ce se execută în timp constant

- Compunere de funcţii (.):

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

```
(f.g)x = f(gx)
```

Funcţiile last, init

```
last ::[a] -> a
last = head.reverse
      Main> (head.reverse)[1,2,3]
      3
      Main> last[1,2,3]
      3
init :: [a] -> [a]
init = reverse.tail.reverse
      Main> init [1,2,3]
      [1,2]
      Main> (reverse.tail.reverse)[1,2,3]
      [1,2]
```

Funcţia **take**

- Funcţia **take**: primele n elemente din lista xs
- Definiţie recursivă:
  - Tipul funcţiei:

```
take      :: Int -> [a] -> [a]
```

  - Enumerarea cazurilor: sunt 2 valori pentru primul argument: 0 şi n+1, două pentru al doilea argument: [] şi x:xs

```
take 0 []                =
take 0 (x:xs)            =
take (n+1) []            =
take (n+1) (x:xs)        =
```

  - Definirea cazurilor de bază:

```
take 0 []                = []
take 0 xs                = []
take (n+1) []            = []
take (n+1) (x:xs)        =
```

Funcţia take

- Definirea celorlalte cazuri:

```
take 0 []          = []
take 0 xs          = []
take (n+1) []      = []
take (n+1) (x:xs)  = x:take n xs
```

- Generalizare, simplificare:

```
take:: Integral b => b -> [a] -> [a]

take 0 xs          = []
take (n+1) []      = []
take (n+1) (x:xs)  = x:take n xs
```

Funcţiile take, drop

```
take    :: Int -> [a] -> [a]
take 0 xs = []
take (n+1)[] = []
take (n+1)(x:xs) = x:take n xs

drop    :: Int -> [a] -> [a]
drop 0 xs = xs
drop(n+1)[] = []
drop(n+1)(x:xs) = drop n xs
```

Funcţiile take, drop

- Proprietăţi:

```
take n xs ++ drop n xs      = xs
take m . take n             = take(m `min` n)
drop m.drop n               = drop(m+n)
take m . drop n             = drop n. take(m+n)
```

```
Main> ((take 3).(take 5))[1] == take(3`min`5)[1]
True
Main> (drop 2. drop 4) [1] == drop(2+4)[1]
True
Main> (take 5.drop 3)[1] == (drop 3.take(5+3))[1]
True
```

Funcţia splitAt

```
splitAt ::Int -> [a] -> ([a], [a])

splitAt n xs = (take n xs, drop n xs)

splitAt 0 xs          = ([], xs)
splitAt n+1 []        = ([], [])
splitAt n+1 (x:xs)    = (x:ys, zs)
                        where (ys,zs)=splitAt n xs
```

Indexarea în liste

- Operatorul de indexare (!!):

```
(!!) :: [a] -> Int -> a

(x:xs)!!0              = x
(x:xs)!!(n+1)          = xs!!n
```

- Proprietăţi:

```
(xs ++ ys)!!k = if k<n then xs!!k else ys!!(k-n)
                       where n = length xs
```

```
Main> "alabalaportocala"!!12
'c'
Main> "aaaa" !!(-1)

Program error: Prelude.!!: negative index

Main> ("aaa"++"bbb")!!4
'b'
```

Funcţia map

- Funcţia map aplică o funcţie fiecărui element al unei liste

```
map :: (a -> b) -> [a] -> [b]

map f [] = []
map f(x:xs) = fx : map f xs
```

```
Main> map square [2,1,4]
[4,1,16]
Main> map (<5) [ 3,7,5,4,3,7,8]
[True,False,False,True,True,False,False]
Main> sum(map square[1..3])
14
Main> sum(map sqrt[1..8])
16.3060005260357
```

Funcţia map

- Proprietăţi:

```
map id           = id
map(f.g)         = map f.map g
f.head           = head . map f
map f.tail       = tail.map f
map f.reverse    = reverse.map f
map f.concat     = concat.map(map f)
map f (xs++ys) = map f xs ++ map f ys
```

Funcţia `filter`

- Argumente: o funcţie booleană p şi o listă xs
- Returnează sublista elementelor din xs ce satisfac p

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)= if p x then
                          x:filter p xs
                          else filter p xs
```

```
Hugs.Base> filter odd [1,2,3,4,5,6,7,8]
[1,3,5,7]
Hugs.Base> (sum.map (*4).filter even)[1..10]
120
```

Funcţia `filter`

- Proprietăţi:

```
filter p . filter q = filter(p si q)
unde (p si q)x = px && qx
filter p . concat = concat . map(filter p)
```

```
Hugs.Base> (filter (>3).filter(<7))[1,2,3,4,5,6,7,8,9]
[4,5,6]
Hugs.Base> (filter (<5).concat)[[1,3,6],[3,8]]
[1,3,3]
Hugs.Base> (concat.map(filter(<5)))[[1,3,6],[3,8]]
[1,3,3]
```

Liste: o altă notaţie

- Analogie cu definirea unei mulţimi:

  $\{1, 4, 9, 16, 25\} = \{x^2 \mid x \in \{1..5\}\}$

- List comprehension

```
Hugs.Base> [x*x|x<-[1..5]]
[1,4,9,16,25]
Hugs.Base> [x*x|x<-[1..5], odd x]
[1,9,25]
Hugs.Base> [x*x|x<-[1..10], even x]
[4,16,36,64,100]
```

Liste

- Sintaxa Haskel pentru definirea listelor:

  [e | Q] unde

  - e este o expresie
  - Q este un calificator
    - O secvenţă – posibil vidă – de forma

      gen1, gar1, gen2, gar2, ...
    - Generator: **`x <- xs`**
      - x este variabilă sau tuplă de variabile
      - xs este o expresie cu valori liste
    - Gardă: o expresie cu valori booleene (gărzile pot lipsi)

Liste

- Dacă în expresia [e | Q] calificatorul Q este vid atunci scriem doar [e]

- Regula generator:

  [e|x<- xs, Q] = concat(map f xs) where fx = [e|Q]

- Regula gardă:

  [e|p, Q] = if p then [e|Q] else []