Paradigme de programare

2010-2011, semestrul 2

Curs 3

Cuprins

- Tipuri de date abstracte – recursivitate – demonstratii de corectitudine
- Transformarea formalismului λ intr-un limbaj de programare (lambda-0)
- Practica in limbajul lambda-0

Tipuri de date abstracte (TDA)

- colectie de date + operatii asociate
- specificate precis (matematic), independent de implementare

- constructori de baza (ex: zero, succ)
- operatori (ex: add, sub, zero?)
- axiome

  (ex: add(succ(m),n) = succ(add(m,n)))

TDA – reguli in scrierea axiomelor

- Axiomele se scriu pe toti constructorii de baza
- Orice in plus e redundant
- Orice in minus face comportamentul functiei sa ramana neprecizat pe anumite valori

Ex (pentru functia factorial):

fact(zero) = 1

fact(succ(n)) = succ(n) * fact(n)

De la axiome la programare functionala (1)

Calculul factorialului

fact(zero) = 1

(caz de baza)

fact(succ(n))

= succ(n) * fact(n)

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (- n 1)))))
```

De la axiome la programare functionala (2)

Inserarea unui element intr-o lista sortata

```
ins(x,[ ]) = [x]
ins(x,y:ys)
    | x:y:ys,          pt x<=y
    | y:(ins x ys),    pt x>y
```

```
(define (ins x L)
    (cond
        ((null? L) (cons x L))
        ((<= x (car L)) (cons x L))
        (else
            (cons (car L)
                  (ins x (cdr L))))))
```

Cum sa gandim recursiv

- Traducand axiomele in cod
- Alegand subproblema la care, daca am sti raspunsul, ne-ar fi mai usor sa rezolvam problema
- Si mai mecanic: scadem dimensiunea parametrilor pe rand si vedem care din rezultate ne-ar ajuta (ex: take n L)
- O explicatie mai detaliata a recursivitatii gasiti la http://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html

De ce recursivitate?

- Ideea programarii functionale: sa apropie programarea de matematica (de specificatia formala)
- Recursivitatea se traduce din axiome
- Inductia structurala permite demonstratii ale proprietatilor TDA
- Aceleasi demonstratii se pot efectua usor pe codul recursiv (ajuta foarte mult faptul ca nu exista efecte laterale)
- Mai mult: intreg calculul lambda se preteaza foarte bine la demonstratii prin inductie, avand numai 3 constructori de baza

Transformarea formalismului λ intr-un limbaj de programare

- Calcul λ = cod masina
- Plan:
  - Valori booleene
  - Conditionala if
  - Perechi
  - Liste
  - Numere naturale
- Evaluare normala => functii nestricte
- Recursivitatea textuala nu e permisa (se poate evita folosind combinatori de punct fix)

TDA Bool

Constructori de baza

T : -> Bool

F : -> Bool

Operatori

not : Bool -> Bool

and : Bool * Bool -> Bool

or : Bool * Bool -> Bool

Axiome

not(T) =

not(F) =

…

TDA Bool - axiome

Axiome

(not T) = F

(not F) = T

(and T b) = b

(and F b) = F

(or T b) = T

(or F b) = b

Practic tipul Bool selecteaza intre 2 valori: true si false.

Cum il implementam in cod masina λ?

TDA Bool - implementare

$T \equiv_{def} \lambda x\ y.x$

$F \equiv_{def} \lambda x\ y.y$

$(T\ a\ b) \rightarrow (\lambda x\ y.x\ a\ b) \rightarrow^{*} a$

$(F\ a\ b) \rightarrow (\lambda x\ y.y\ a\ b) \rightarrow^{*} b$

TDA Bool - implementare

not $\equiv_{def}$?

and $\equiv_{def}$?

or $\equiv_{def}$?

TDA Bool - implementare

$$\text{not} \equiv_{\text{def}} \lambda x.(x\ F\ T)$$

$$(\text{not } T) \rightarrow (\lambda x.(x\ F\ T)\ T) \rightarrow (T\ F\ T) \rightarrow F$$

$$(\text{not } F) \rightarrow (\lambda x.(x\ F\ T)\ F) \rightarrow (F\ F\ T) \rightarrow T$$

TDA Bool - implementare

$$\text{and} \equiv_{\text{def}} \lambda x\ y.(x\ y\ F)$$

$$(\text{and}\ T\ b) \rightarrow (\lambda x\ y.(x\ y\ F)\ T\ b) \rightarrow^{*} (T\ b\ F) \rightarrow b$$

$$(\text{and}\ F\ b) \rightarrow (\lambda x\ y.(x\ y\ F)\ F\ b) \rightarrow^{*} (F\ b\ F) \rightarrow F$$

TDA Bool - implementare

$or \equiv_{def} \lambda x\ y.(x\ T\ y)$

$(or\ T\ b) \rightarrow (\lambda x\ y.(x\ T\ y)\ T\ b) \rightarrow^{*} (T\ T\ b) \rightarrow T$

$(or\ F\ b) \rightarrow (\lambda x\ y.(x\ T\ y)\ F\ b) \rightarrow^{*} (F\ T\ b) \rightarrow b$

Conditionala if

Axiome

(if T a b) = a

(if F a b) = b

If trebuie neaparat sa fie o functie nestricta, altfel recursivitatile nu s-ar termina (si o serie de alte nenorociri).

Conditionala if – implementare

$$\text{if} \equiv_{\text{def}} \lambda p\ x\ y.(p\ x\ y)$$

$$(\text{if T a b}) \rightarrow (\lambda p\ x\ y.(p\ x\ y)\ \text{T a b}) \rightarrow^{*} (\text{T a b}) \rightarrow \text{a}$$

$$(\text{if F a b}) \rightarrow (\lambda p\ x\ y.(p\ x\ y)\ \text{F a b}) \rightarrow^{*} (\text{F a b}) \rightarrow \text{b}$$

TDA Pair

Constructori de baza

pair : A * B -> Pair

Operatori

fst : Pair -> A

snd : Pair -> B

Axiome

…

TDA Pair - axiome

Axiome

(fst (pair a b)) = a

(snd (pair a b)) = b

Tipul pair produce o pereche in care ulterior putem pasa un “mesaj” care ne readuce primul/cel de-al doilea element.

Cum il implementam in cod masina λ?

TDA Pair - implementare

$pair \equiv_{def} \lambda x\ y\ z.(z\ x\ y)$

$(pair\ a\ b) \rightarrow (\lambda x\ y\ z.(z\ x\ y)\ a\ b) \rightarrow^{*} \lambda z.(z\ a\ b)$

$fst \equiv_{def} \lambda p.(p\ T)$

$snd \equiv_{def} \lambda p.(p\ F)$

$(fst\ (pair\ a\ b)) \rightarrow^{*} (\lambda p.(p\ T)\ \ \lambda z.(z\ a\ b)) \rightarrow (\lambda z.(z\ a\ b)\ T) \rightarrow^{*} a$

$(snd\ (pair\ a\ b)) \rightarrow^{*} (\lambda p.(p\ F)\ \ \lambda z.(z\ a\ b)) \rightarrow (\lambda z.(z\ a\ b)\ F) \rightarrow^{*} b$

TDA List

Constructori de baza

nil : -> List

cons : A * List -> Pair

Operatori

car : List -> A

cdr : List -> List

null? : List -> Bool

append : List * List -> List

Axiome

...

TDA List - axiome

Axiome

(car (cons e L)) = e

(cdr (cons e L)) = L

(null? nil) = T

(null? (cons e L)) = F

(append nil B) = B

(append (cons e A) B) = (cons e (append A B))

Cum implementam tipul List in cod masina λ?

TDA List - implementare

$nil \equiv_{def} \lambda x.T$

$cons \equiv_{def} pair$

$(cons\ e\ L) \rightarrow (\lambda x\ y\ z.(z\ x\ y)\ e\ L) \rightarrow^{*} \lambda z.(z\ e\ L)$

$car \equiv_{def} fst$

$cdr \equiv_{def} snd$

TDA List - implementare

$$\text{null?} \equiv_{\text{def}} \lambda L.(L \;\; \lambda x\, y.F)$$

(null? nil) $\rightarrow$ (λL.(L λx y.F) λx.T) $\rightarrow^{*}$ (λx.T …) $\rightarrow$ T

(null? (cons e L)) $\rightarrow$ (λL.(L λx y.F) λz.(z e L)) $\rightarrow$

(λz.(z e L) λx y.F) $\rightarrow$ (λx y.F e L) $\rightarrow$ F

TDA List - implementare

append $\equiv_{def}$

λA B.

(if (null? A)

B

(cons (car A) (append (cdr A) B)

Problema: in calculul λ nu pot avea recursivitate textuala.

Combinatori de punct fix

- (functionala) Fix este combinator de punct fix <=> pt orice functionala F, (Fix F) = (F (Fix F))
- cu ajutorul combinatorilor de punct fix putem defini functii recursive la modul textual nerecursiv

Ex:

fact = (Fix λf n.(if n=0 then 1 else n*f(n-1))) = (Fix F)

$\rightarrow$ (F (Fix F))

$\rightarrow$ λn.(if n=0 then 1 else n*((Fix F)(n-1))

Combinatori de punct fix clasici

- Pentru functii unare

$c1 \equiv_{def}$

$\lambda F.$

$(\lambda g\ x.(F\ (g\ g)\ x)$

$\lambda g\ x.(F\ (g\ g)\ x))$

- Pentru functii binare

$c2 \equiv_{def}$

$\lambda F.$

$(\lambda g\ x\ y.(F\ (g\ g)\ x\ y)$

$\lambda g\ x\ y.(F\ (g\ g)\ x\ y))$

Implementarea lui append folosind c2

append $\equiv_{def}$

```
  (c2
     λF.
        λA B.
           (if (null? A)
               B
               (cons (car A) (F (cdr A) B))))
```

TDA Nat

Constructori de baza

zero : -> Nat

succ : Nat -> Nat

Operatori

pred : Nat -> Nat

zero? : Nat -> Bool

add : Nat * Nat -> Nat

Axiome

…

TDA Nat - axiome

Axiome

(pred (succ n)) = n

(zero? zero) = T

(zero? (succ n)) = F

(add zero n) = n

(add (succ m) n) = (succ (add m n))

Cum implementam tipul Nat in cod masina λ?

TDA Nat - implementare

zero $\equiv_{def}$ nil

succ $\equiv_{def}$ λn.(cons nil n)

pred $\equiv_{def}$ cdr

zero? $\equiv_{def}$ null?

add $\equiv_{def}$ append