Paradigme de programare

2010-2011, semestrul 2

Curs 4

Programare functionala in Scheme - Cuprins

- Tipare
- Gestiunea memoriei
- Efecte laterale si transparenta referentiala
- Functii ca valori de ordinul 1
- Domeniu de vizibilitate a variabilelor
- Context computational si inchideri functionale

Tipare intr-un limbaj de programare

- Se atribuie o semnificatie datelor

- Siguranta (corectitudinea codului)
- Optimizare
- Documentare
- Abstractizare

Tipare intr-un limbaj de programare

- Tare (strong)
- Slaba (weak)

- Statica (static)
- Dinamica (dynamic)

Tipare tare/slaba

- Tare: nu se permit operatii pe argumente care nu au tipul corect (se converteste tipul numai in cazul in care nu se pierde informatie la conversie)
- Ex: 1 + “23” => Eroare

- Intr-un limbaj slab tipat nu se verifica corectitudinea tipurilor, se face cast dupa reguli specifice limbajului
- Ex:

  1+ “23” = 24 (Visual Basic)

  1+ “23” = “123” (JavaScript)

Tipare statica/dinamica

- Statica: verificarea de tip se face la compilare (Pascal, C, C++, Java, Haskell, ML, Scala etc) – atat variabilele cat si valorile au un tip asociat
- Dinamica: verificarea de tip se face la rulare (Python, Lisp, Scheme, Prolog, Javascript, PHP etc) – numai valorile au un tip asociat

Tipare statica vs tipare dinamica

- Viteza la rulare
- Conservatoare: Elimina din start toate erorile de tip (cate din buguri sunt erori de tip?)
- Debugging mai usor
- Necesita declaratii sau inferenta de tip

- Viteza a ciclului (edit-compile-test-debug)
- Flexibila: permite codului sa se poarte bine la rulare
- Debugging mai dificil (efectul poate sa apara mult dupa cauza)
- Permite constructe suplimentare (ex:eval)
- Faciliteaza metaprogramarea

Tiparea in Scheme

- Tare
- Dinamica

Gestiunea memoriei in Scheme

- Garbage collection – algoritm inventat in 1958 de John McCarthy ca parte a implementarii de Lisp
- Se elibereaza spatiul ocupat de obiecte care nu mai sunt accesibile in program
- Spatiul disponibil este compactat
- Probleme: raman in memorie obiecte moarte dar care inca sunt teoretic accesibile (nu exista un algoritm care sa determine care obiect a murit in program – v. halting problem)
- Avantaje: cod lizibil, siguranta

Gestiunea memoriei in Scheme

```
(let ((x (list ...)))
    (let loop ((y x))
        (if (null? y)
        '()
        (begin
            (...(car y) ...) ;; proceseaza (car y)
            (loop (cdr y))))))
```

- Primul element nu poate fi colectat pentru ca inca e accesibil din variabila x

Efecte laterale

- O functie /expresie produce efecte laterale daca, pe langa valoarea pe care o returneaza, mai are si alte efecte asupra starii programului sau a “lumii de afara”

- Ex: modificarea unor variabile/argumente, citirea/scrierea (d)in fisier, apelarea altor functii care produc efecte laterale

Efecte laterale

- Consecinte:
  - Conteaza ordinea de evaluare
  - Scade nivelul de abstractizare

- Programarea imperativa: uz necontrolat al efectelor laterale
- Programarea pur functionala: le elimina

- Consecinta: nu exista atribuiri in PF

Efecte laterale - exemple

```
int x=0
int f() {
  x<-15
  return x
}
```

f()+x => 30

x+f() => 15

Se pierde comutativitatea adunarii!

Efecte laterale - exemple

```
int a=4, b=6
int aduna() {
  while(a>0) {
    a--
    b++
  }
  return b
}
```

La final b=10.

```
int a=4, b=6
int aduna() {
  int s=b
  while (a>0) {
    a--
    s++
  }
  return s
}
```

La final b=6.

Efecte laterale in Scheme

Exista…

… dar noi ne facem ca nu exista.

Transparenta referentiala

Exista transparenta referentiala atunci cand:

- Nu exista efecte laterale
- Functiile/expresiile sunt pure (produc acelasi output de fiecare data cand sunt aplicate pe acelasi input)

- In principiu, o expresie poate fi substituita cu valoarea ei
- Daca expresia producea efecte laterale, in urma substitutiei acestea s-ar fi pierdut

- Consecinte: usurinta de a paraleliza, caching

Transparenta referentiala in Scheme

Nu exista.

- Avem efecte laterale
- Avem functii impure

Functii ca valori de ordinul 1

Date de ordinul 1 – pot fi:

- Valori ale unor variabile
- Argumente ale unor functii
- Valori returnate de functii
- Membri ai unei structuri compuse

Lucruri
Substantive
Date

Actiuni
Verbe
Functii

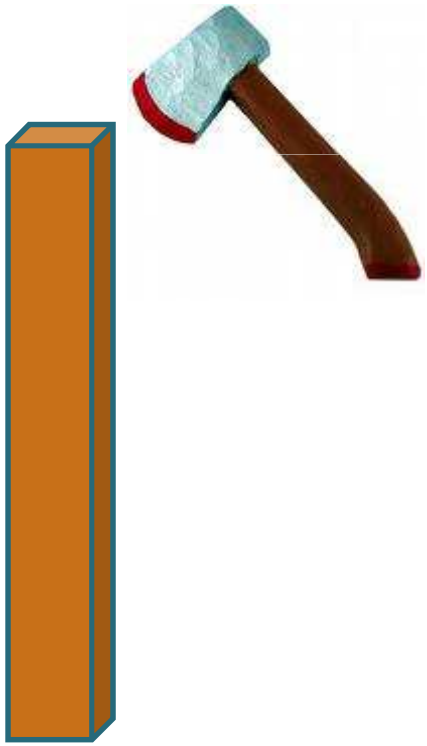
Functii ca valori de ordinul 1

In programarea functionala: ideea ca toate tipurile de date sunt valori de ordinul 1

Lucruri

Substantive

Date



Actiuni

Verbe

Functii

Consecinte ale functiilor ca valori de ordinul 1

- Functii anonime
- Functii curry
- Functionale (functii de nivel inalt)

- Functie de ordinul 1 = functie care nu ia ca argumente functii
- Functie de ordinul 2 = functie de functii de ordin 1
- etc

Exercitiu

Putem construi un exemplu cu de toate?

Domeniu de vizibilitate a variabilelor

Variabila = pereche identificator-valoare

- exista in program intr-o anumita zona in care e vizibila si valoarea poate fi accesata prin referirea identificatorului
- Are o durata de viata

Vizibilitatea variabilelor

```
(define a 1)
(define n 2)
(define (fact n)
    (if (< n 2)
        1
        (* n (fact (- n 1)))))
(fact n)
```

Vizibilitatea variabilelor

```
(define a 1)
(define n 2)
(define (fact n)
    (if (< n 2)
        1
        (* n (fact (- n 1)))))
(fact n)
```

Vizibilitatea variabilelor

```
(define a 1)
(define n 2)
(define (fact n)
    (if (< n 2)
        1
        (* n (fact (- n 1)))))
(fact n)
```

Legare pe lant static/dinamic a variabilelor

- Static: regiunea variabilelor este controlata textual, prin mecanisme specifice limbajului (lexical scoping)
- Dinamic: regiunea variabilelor este controlata dinamic, in functie de timp

Scheme:

- Dinamic pt variabilele top-level (definite cu define)
- Static pentru restul (let, let*, letrec, lambda…)

Exemplul anterior revizitat

cu elemente de legare pe lant dinamic…

Context computational al unui punct P din program

= toate variabilele care il au pe P in domeniul lor de vizibilitate (=> toate perechile identificator valoare)

Obs1: intereseaza variabilele libere

Obs2: intr-un limbaj cu legare pe lant static, contextul unui punct P este structural vizibil imediat ce am scris programul; Valoarea contextului depinde insa de executie (de timp).

Inchidere functionala

= o functie +

contextul in punctul de definire a functiei

- Concept inventat in anii ’60 si implementat in totalitate pentru prima data in Scheme
- Utilitate: intarzierea evaluarii (si altele)

Inchidere functionala - exemplu

```
(define o          ;; compunerea a 2 functii
  (λ (f g)
    (λ (x)
      (f (g x)))))
```

Ce se intampla cand evaluam

```
(o (λ (x) (* x x))
   (λ (x) (+ 2 x)))          ?
```

Inchidere functionala - exemplu

Se creeaza o inchidere functionala,

adica functia:

```
(λ (x)
  (f (g x)))
```

cu contextul:

```
f <- (λ (x) (* x x))
g<- (λ (x) (+ 2 x))
```

Cu alte cuvinte – o functie care stie cine erau variabilele ei libere in momentul definirii functiei.

Inchidere functionala – modificarea contextului

O inchidere functionala

- ramane mereu o pereche functie-context
- este o valoare de ordinul 1

Contextul:

- nu se mai modifica in cazul in care este compus din variabile legate pe lant static
- se poate modifica in cazul variabilelor legate pe lant dinamic