PARADIGME DE PROGRAMARE

Curs 1

Introducere. Tipuri de date abstracte. Inducție structurală.

INTRODUCERE

Administrative

https://ocw.cs.pub.ro/courses/pp

- Cursuri, teme, instrucțiuni de instalare
- Laborator (teorie, exerciții, soluții) – teoria înainte / exercițiile la laborator / soluțiile după
- Exemple de examene și teste
- Regulament – citiți-l la începutul semestrului

Structura fiecărui curs

- Quiz rapid – din cursul anterior
- Predare & exerciții – fiți activi, puneți întrebări!
- Rezumatul cursului curent – conceput pentru autotestare, nu doar pentru a fi citit

Obiectivul materiei – programatori mai buni

Alternativă la paradigmele imperativă și orientată obiect

- Paradigma funcțională

Cum sunt proiectate limbajele de programare

- Modele de calculabilitate
- Features: controlul complexității prin lizibilitate și eficiență
- Limbaje multiparadigmă pentru programatori multiparadigmă

Adaptarea rapidă la noi limbaje de programare

- Racket, Haskell

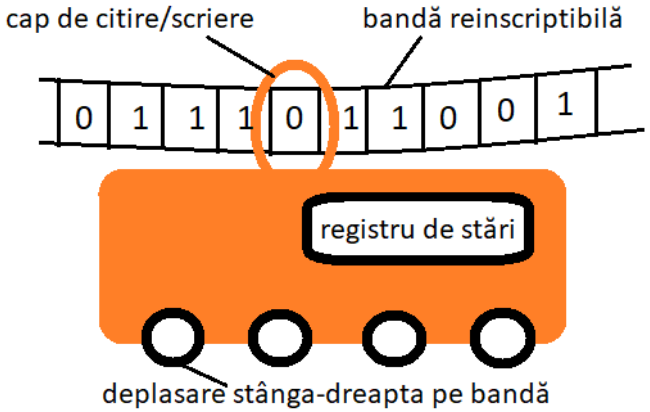
Provocări distractive

Modele de calculabilitate

Mașina Turing

bandă infinită de memorie care suportă operații de citire/scriere

funcționare: reguli care descriu mutarea asociată asociată unei anumite stări – „dacă ești în starea B și citești 0, scrie 1 și mută la dreapta”



Calcul Lambda

funcționare: funcții și aplicare de funcții „funcția de parametru x și corp C se aplică pe argumentul 2, rezultând C în care x este înlocuit cu 2” (C este la rândul său o funcție)

$(\lambda x.\lambda y.x+y\ 2) \rightarrow \lambda y.2+y$

fiecare funcție este ca o cutie neagră care își calculează rezultatul (nu interesează cum anume îl calculează)

Modele, paradigme, limbaje

Model de calculabilitate

- Oferă un model formal al efectuării calculului
- Diferă de alte modele prin CUM se calculează funcțiile, nu prin CE funcții se calculează

Paradigmă de programare

- Stil fundamental de a programa, bazat pe un anumit model de calculabilitate
- Mod de reprezentare a datelor (ex: variabile, funcții, obiecte, fapte, constrângeri)
- Mod de prelucrare a reprezentării (ex: atribuiri, evaluări, fire de execuție)

Limbaj de programare

- Limbaj formal capabil să exprime procesul de rezolvare a problemelor
- Sprijină una sau mai multe paradigme (ex: Scala, F# - POO și PF; Python – imperativ, POO și PF)

Comparație între paradigmele studiate

Paradigma	Reprezentarea datelor	Structura programului	Execuția programului	Rezultat	Limbaje
Imperativă	Variabile	Succesiune de comenzi	Execuție de comenzi	Stare finală a memoriei	**C, Pascal, Fortran**
Orientată Obiect	Obiecte	Colecție de clase și obiecte	Transmitere de mesaje între obiecte	Stare finală a obiectelor	**Java, C++**
Funcțională	Funcții	Colecție de funcții	Evaluare de funcții	Valoarea la care se evaluează funcția principală	**Racket, Haskell**

De ce?

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Dijkstra, How do we tell truths that might hurt

I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

Abraham Maslow, The law of instrument

The illiterate of the 21st century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn.

Alvin Toffler

Exemplu

Să se determine factorialul unui număr natural n folosind paradigmele:

- Imperativă
- Funcțională (Racket)
- Funcțională (Haskell)

Rezolvare imperativă

```
1.    int i,  factorial = 1;
2.    for (i = 2; i <= n; i++)
3.        factorial *= i;
```

← datele sunt reținute în variabilele i, factorial

← rezolvarea este o execuție succesivă de înmulțiri

← rezultatul se regăsește în starea finală a memoriei (în zona rezervată variabilei factorial)

De reținut

- Programele imperative au stare
- Starea diferă de la un moment al execuției la altul
- Elemente fundamentale: atribuirea, ciclarea
- Soluție tip „rețetă” (programul descrie CUM se construiește, pas cu pas, rezultatul)

Rezolvare funcțională – Racket

```
1.  (define (factorial n)
2.    (if (zero? n)
3.        1
4.        (* n (factorial (- n 1)))))
```

← datele sunt reținute în funcții (totul este o funcție)

← caz de bază în recursivitate

← apelul recursiv cu o nouă valoare a parametrului n

De reținut

- Programele funcționale nu au stare
- Ciclarea este înlocuită prin recursivitate
- Atribuirea este înlocuită printr-un apel recursiv cu noi valori ale parametrilor funcției
- Succesiunea de comenzi este înlocuită prin compunere de funcții
- Soluție declarativă (programul descrie CE este, din punct de vedere matematic, rezultatul)

Rezolvare funcțională – Haskell

```
1.  factorial 0 = 1
2.  factorial n = n * factorial (n - 1)
```

← totul este o funcție, deci nu este necesar un cuvânt cheie care să spună că urmează o funcție

De observat

- Haskell permite pattern matching pe parametrii formali ai funcției (feature existent în Haskell dar nu și în Racket)
- Dacă pattern matching-ul de pe linia 1 reușește, se întoarce rezultatul 1, altfel se încearcă potrivirea cu linia următoare (care, pe codul de mai sus, reușește întotdeauna)
- Aceeași paradigmă (ciclare → recursivitate, starea pasată ca parametru, succesiune → compunere), însă feature-ul suplimentar (pattern matching) duce la un cod mai succint și mai apropiat de reprezentarea matematică

Rezolvări funcționale „avansate”

Racket

```
1.  (define (factorial n)
2.    (apply * (range 2 (+ n 1))))
```

Haskell

```
1.  factorial n = product [1 .. n]
```

TIPURI DE DATE ABSTRACTE

Tipuri de date abstracte

- Tip de date abstract (TDA) = tip de date definit matematic, fără detalii de implementare
- Un TDA specifică:
  - Metode abstracte de a obține toate valorile tipului
  - Un set de operații posibile cu aceste valori
    - Ex: push, pop, empty? – pentru TDA Stack
      enqueue, dequeue, empty? – pentru TDA Queue
      insert, extract_min, decrease_key, empty? – pentru TDA PriorityQueue … etc.
  - Comportamentul operațiilor – contract care trebuie respectat de orice implementare ulterioară
    - Ex: oricum va fi implementată stiva (vector / listă înlănțuită / … etc.), empty? trebuie să întoarcă:
      - True pe stiva vidă
      - False pe orice altceva

Exemplu: Natural

- Metode abstracte de a obține toate valorile tipului
  - Ce este un număr natural?
  - Cum se obține?
- Operații posibile cu aceste valori
  - Ce funcționalități ar trebui să aibă numerele naturale?
- Comportamentul operațiilor
  - Comportamentul trebuie specificat pe toate numerele naturale: cum garantăm asta?

Exemplu: Natural

- Metode abstracte de a obține toate valorile tipului
  - Ce este un număr natural? – un întreg pozitiv care se folosește pentru a număra și a enumera
  - Cum se obține? – prin creșterea cu o unitate a altui număr natural
    (excepție: numărul 0 – valoarea de start, singura care nu are predecesor)
- Operații posibile cu aceste valori
  - Ce funcționalități ar trebui să aibă numerele naturale? – egalitate cu 0, +, –, *, /, etc.
- Comportamentul operațiilor
  - Comportamentul trebuie specificat pe toate numerele naturale: cum garantăm asta?
    – avem metode de a obține toate valorile tipului, așadar specificăm comportamentul operațiilor pe rezultatul fiecărei metode în parte

Exemplu: Natural

- Metode abstracte de a obține toate valorile tipului

```
zero : → Natural
succ : Natural → Natural
```

- Operații posibile cu aceste valori (o selecție)

```
zero? : Natural → Bool
add: Natural × Natural → Natural
```

- Comportamentul operațiilor

```
zero?(zero) = True          add(zero, n) = n
zero?(succ(m)) = False      add(succ(m), n) = succ(add(m, n))
```

Exemplu: List

- Metode abstracte de a obține toate valorile tipului
  - Ce este o listă?
  - Cum se obține?
- Operații posibile cu aceste valori
  - Ce funcționalități ar trebui să aibă listele?
- Comportamentul operațiilor
  - Comportamentul trebuie specificat pe toate listele

Exemplu: List

- Metode abstracte de a obține toate valorile tipului
  - Ce este o listă? – o colecție de valori, într-o anumită ordine
  - Cum se obține? – prin adăugarea unui element la începutul altei liste
    (excepție: lista vidă – valoarea de start)
- Operații posibile cu aceste valori
  - Ce funcționalități ar trebui să aibă listele? – test de listă vidă, concatenare, etc.
- Comportamentul operațiilor
  - Comportamentul trebuie specificat pe toate listele

Exemplu: List

- Metode abstracte de a obține toate valorile tipului

```
null : → List
cons : T × List → List
```

(T = tipul elementelor din listă) 0

- Operații posibile cu aceste valori (o selecție)

```
null? : List → Bool
++ : List × List → List
```

(++ = operator infixat cu rol de append)

- Comportamentul operațiilor

```
null?(null) = True
null?(cons(x, l)) = False

null ++ l = l
cons(x, l1) ++ l2 = cons(x, l1 ++ l2)
```

Exerciții

- Specificați comportamentul următoarelor operații ale tipurilor Natural și List:

```
* : Natural × Natural → Natural        (* = operator infixat pentru înmulțire)
pred : Natural → Natural               (predecesor)
first : List → T                       (primul element din listă)
rest : List → List                     (lista fără primul element)
length : List → Int                    (lungime)
reverse : List → List                  (lista inversată)
```

Constructori

- Constructori pentru TDA-ul t = toate operațiile care produc o valoare de tip t
  - Ex: `zero : → Natural`
    `succ : Natural → Natural`
    `zero? : Natural → Bool` (zero? nu este constructor al tipului Natural)
    `add : Natural × Natural → Natural`
- Constructori de bază = „metodele abstracte de a obține toate valorile tipului” = constructori necesari și suficienți pentru a genera toate valorile tipului
  - Ex: `zero` și `succ`
    - orice număr natural se poate obține folosind doar `zero` și `succ`
    - doi = `succ(succ(zero))`

Clasificare constructori

- Constructorii unui TDA t se clasifică în funcție de tipul parametrilor:
  - Nular – 0 parametri
    - construiește o valoare a lui t „din nimic”
    - `zero : → Natural, null : → List`
  - Extern – doar parametri care nu sunt de tip t
    - construiește o valoare a lui t doar din valori externe
    - `singleton : T → List` (potențial constructor pentru o listă cu un singur element de tip T)
  - Intern – cel puțin un parametru de tip t
    - construiește o valoare a lui t pe baza uneia sau mai multor valori „interne tipului”
    - `succ : Natural → Natural, cons : T × List → List`
- Notăm cu Σ_0, Σ_e, Σ_i mulțimea constructorilor nulari, externi, interni ai TDA-ului.

Dimensiunea valorilor tipului

- Dimensiunea valorii = numărul de constructori de bază din formula care produce valoarea
  - `succ(succ(zero))` – dimensiunea 3
    `cons(1, cons(7, cons(2, null)))` – dimensiunea 4

- Constructorii:
  - nulari – dimensiunea 1
  - externi – dimensiunea 1

  creează valorile elementare ale tipului

  - interni – dimensiunea >1

  creează valori care se pot descompune în valori de dimensiune mai mică

Operatori și axiome

- Operatori ai TDA-ului t = operațiile posibile cu valorile TDA-ului t
  - `zero?, add, *, pred` – operatori pentru Natural
  - `null?, ++, first, rest, length, reverse` – operatori pentru List
- Axiome = reguli care definesc comportamentul operatorilor pe toate valorile tipului
  - `null ++ l = l` – axiome pentru ++
    `cons(x, l1) ++ l2 = cons(x, l1 ++ l2)`
- O axiomă pentru fiecare constructor de bază asigură acoperirea tuturor valorilor lui t.
- Scrierea axiomelor ~ scrierea unui program recursiv
  - Caz de bază: constructorii nulari și externi – rezultatul se calculează ușor pe valori elementare
  - Apel recursiv: pentru constructorii interni, rezultatul operației pe o valoare de dimensiune n se calculează în funcție de rezultatul pe valorile subiacente de dimensiune mai mică

Memento – factorial Racket / Haskell

```
1.  (define (factorial n)
2.    (if (zero? n)
3.        1
4.        (* n (factorial (- n 1)))))
```

← factorial este un operator al tipului Natural

← caz de bază: constructorul nular zero

← apel recursiv: n este succ(n - 1), se determină fact(n) în funcție de rezultatul fact(n - 1) pe valoarea subiacentă

```
1.  factorial 0 = 1
2.  factorial n = n * factorial (n - 1)
```

← același lucru, într-o sintaxă și mai asemănătoare cu sintaxa axiomelor

TDA-ul Natural – cu terminologie actualizată

- ~~Metode abstracte de a obține toate valorile tipului~~ Constructori de bază

```
zero : → Natural
succ : Natural → Natural
```

- ~~Operații posibile cu aceste valori (o selecție)~~ Operatori

```
zero? : Natural → Bool
add: Natural × Natural → Natural
```

- ~~Comportamentul operațiilor~~ Axiome

```
zero?(zero) = True          add(zero, n) = n
zero?(succ(m)) = False      add(succ(m), n) = succ(add(m, n))
```

De ce TDA-uri?

- Separă implementarea tipului de utilizarea lui
  - Utilizatorul folosește tipuri complexe ca și cum ar fi tipuri primitive simple
    - Prin intermediul operatorilor
    - Nu știe cum sunt implementați operatorii, știe doar că ei respectă „contractul” definit în axiome
    - Ex: inserția în hash table devine o operație simplă, utilizatorul nu trebuie să cunoască detalii despre coliziuni, rehashing, etc. – aceste detalii sunt ascunse în implementarea operației insert
  - Proiectantul poate modifica implementarea operatorilor fără a afecta utilizarea
    - Orice implementare care respectă „contractul” funcționează
      Contează CE fac operatorii, nu CUM.
    - Interfața nu se modifică → programele care folosesc interfața nu se modifică

De ce TDA-uri?

- Demonstrații de corectitudine generice
  - În absența unui TDA, validăm separat două programe A și B care implementează același algoritm:
    - A cu stive implementate ca vectori
    - B cu stive implementate ca liste înlănțuite
  - Utilizând TDA-uri, aceeași demonstrație validează programe care folosesc implementări diferite
    - Proprietățile operatorilor se demonstrează exclusiv pe baza axiomelor
    - Demonstrațiile sunt facilitate de un set neredundant de axiome

Ex: Scriem

```
null ++ l = l
cons(x, l1) ++ l2 = cons(x, l1 ++ l2)
```

și nu

```
null ++ null = null
null ++ cons(x, l) = cons(x, l)
cons(x, l) ++ null = cons(x, l)
cons(x, l1) ++ cons(x, l2) = cons(x, l1 ++ cons(x, l2))
```

INDUCȚIE STRUCTURALĂ

Inducție structurală

- Inducția structurală = particularizare a inducției bine formate
  - X = mulțimea valorilor unui TDA t
  - $x \prec y \Leftrightarrow$ dimensiunea valorii x < dimensiunea valorii y
- Utilizare: demonstrează că toate valorile TDA-ului satisfac o proprietate P
  - Caz de bază (valorile de dimensiune 1 – constructorii nulari și externi)
    - Toți constructorii nulari respectă proprietatea P: $P(\sigma), \forall \sigma \in \Sigma_0$
    - Toți constructorii externi respectă proprietatea P: $P(\sigma(\ldots)), \forall \sigma \in \Sigma_e$
  - Pas inductiv: (valorile de dimensiune >1 – constructorii interni)
    - Toți constructorii interni respectă proprietatea P: $P(x) \Rightarrow P(\sigma(\ldots x \ldots)), \forall \sigma \in \Sigma_i$

      ipoteza inductivă
  - Observație: Inducția structurală este un fel de inducție completă după dimensiunea valorilor
    - Valorile de dimensiune n respectă P oricând toate valorile de dimensiune < n respectă P

Exemplu – asociativitatea adunării

- Pentru tipul Natural, cu constructorii

```
zero : → Natural
succ : Natural → Natural
```

și operatorul

```
add: Natural × Natural → Natural
```

caracterizat de axiomele

```
add(zero, n) = n                          (ADD1)
add(succ(m), n) = succ(add(m, n))         (ADD2)
```

demonstrăm prin inducție structurală proprietatea următoare (asociativitatea adunării):

```
add(a, add(b, c)) = add(add(a, b), c),   ∀a, b, c ∈ Natural
```

Exemplu – asociativitatea adunării

- Demonstrație prin inducție structurală după variabila a.
- Caz de bază – a = `zero` (constructor nular)
  - `add(zero, add(b, c)) = add(add(zero, b), c)` ⇔ (conform ADD1)
    `add(b, c) = add(b, c)` □
  - Cazul de bază este încheiat, întrucât Natural are un singur constructor nular și niciunul extern.
- Pas inductiv – a = `succ(x)` (constructor intern)
  - Ipoteză inductivă: `add(x, add(b, c)) = add(add(x, b), c)` $=_{\text{notație}}$ α
  - `add(succ(x), add(b, c)) = add(add(succ(x), b), c)` ⇔ (conform ADD2)
    `succ(add(x, add(b, c))) = add(succ(add(x, b)), c)` ⇔ (conform ADD2)
    `succ(add(x, add(b, c))) = succ(add(add(x, b), c))` ⇔ (conform II)
    `succ(α) = succ(α)` □
  - Demonstrația este încheiată, întrucât Natural are un singur constructor intern.

Inducție structurală versus inducție completă

- Inducția structurală demonstrează P pentru toate valorile posibile ale unui TDA
- Inducția completă demonstrează P pentru toate dimensiunile posibile ale datelor

- De ce inducție structurală când putem folosi o inducție completă?
  - Axiomele sunt cod, și demonstrația folosește direct codul
    (la inducție completă, este necesar un pas intermediar de raționament asupra codului, pentru a argumenta dimensiunea datelor)
  - Gestionează bine date complexe, ca structuri imbricate sau mutual recursive
    - Ex: un tip Folder – definit ca listă de Item, respectiv un tip Item – definit ca File sau Folder
  - Bună practică: dacă tipul se poate defini recursiv, demonstrează proprietățile sale prin inducție structurală

Inducție structurală – utilizări (1)

- Proprietăți ale TDA-urilor, precum asociativitatea adunării
  - Consecință: rezultatul este corect, indiferent cum spargem inputul
- Reguli de rescriere pentru compilator (demonstrate în prealabil și introduse de proiectant în logica de optimizare a codului)
  - Ex: x + 0 devine x, lower(lower(str)) devine str
  - Ex (optimizare prin fuziune):
    - codul parcurge inputul pentru a filtra datele, apoi parcurge lista filtrată și îi calculează suma
    - compilatorul fuzionează cele două parcurgeri într-una singură, care efectuează și filtrare și adunare
  - Ex (de capcană): x / x nu devine 1, pentru că echivalența eșuează pe cazul de bază (x = 0)
  - Consecință: calculele și datele temporare inutile sunt eliminate în siguranță

Inducție structurală – utilizări (2)

- Demonstratoare automate de teoreme
  - Ex (proprietate demonstrată automat asupra unui cod dat):
    Pentru orice acces v[i] din cod, i nu este „out of bounds”.
  - Ex (strategie de optimizare):
    Se generează mii de alternative mai eficiente pentru o bucată costisitoare de cod, și se înlocuiește codul cu o alternativă pentru care s-a putut demonstra automat echivalența.
  - Ex (compilator sigur):
    CompCert este un compilator de C care, la compilare, demonstrează automat corectitudinea optimizărilor pe care le efectuează.
  - Consecință: se poate garanta siguranța unor sisteme critice (finanțe, transport, dispozitive medicale, etc.)

Rezumat

Modele de calculabilitate

Paradigme

Tipuri de date abstracte

Elemente TDA

Tipuri de constructori

Dimensiunea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme

Tipuri de date abstracte

Elemente TDA

Tipuri de constructori

Dimensiunea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte

Elemente TDA

Tipuri de constructori

Dimensiunea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte: tipuri definite matematic, fără detalii de implementare

Elemente TDA

Tipuri de constructori

Dimensiunea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte: tipuri definite matematic, fără detalii de implementare

Elemente TDA: constructori de bază, operatori, axiome

Tipuri de constructori

Dimensiunea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte: tipuri definite matematic, fără detalii de implementare

Elemente TDA: constructori de bază, operatori, axiome

Tipuri de constructori: nulari (0 parametri), externi (0 … de tip t), interni (are … de tip t)

Dimensiunea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte: tipuri definite matematic, fără detalii de implementare

Elemente TDA: constructori de bază, operatori, axiome

Tipuri de constructori: nulari (0 parametri), externi (0 … de tip t), interni (are … de tip t)

Dimensiunea valorii: numărul de constructori de bază din reprezentarea valorii

Inducție structurală

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte: tipuri definite matematic, fără detalii de implementare

Elemente TDA: constructori de bază, operatori, axiome

Tipuri de constructori: nulari (0 parametri), externi (0 … de tip t), interni (are … de tip t)

Dimensiunea valorii: numărul de constructori de bază din reprezentarea valorii

Inducție structurală: demonstrează că toate valorile TDA-ului satisfac o proprietate P

Etape inducție structurală

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda

Paradigme: imperativă, orientată obiect, funcțională

Tipuri de date abstracte: tipuri definite matematic, fără detalii de implementare

Elemente TDA: constructori de bază, operatori, axiome

Tipuri de constructori: nulari (0 parametri), externi (0 … de tip t), interni (are … de tip t)

Dimensiunea valorii: numărul de constructori de bază din reprezentarea valorii

Inducție structurală: demonstrează că toate valorile TDA-ului satisfac o proprietate P

Etape inducție structurală: caz de bază: dimensiune 0 – constructori nulari și externi
pas inductiv: dimensiune >0 – constructori interni