Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@upb.ro
Departamentul de Calculatoare

2024

Cursul 9: Concluzie – Paradigma Funcțională

Caracteristici ale paradigmelor de programare

Paradigma de programare

Impact în scrierea unui program

- Paradigma de programare – un mod de a:
  - aborda rezolvarea unei probleme printr-un program;
  - structura un program;
  - reprezenta datele dintr-un program;
  - implementa diversele aspecte dintr-un program (cum prelucrăm datele);
- Un limbaj poate include caracteristici din una sau mai multe paradigme;
  - în general există o paradigmă dominantă;
- Atenție! Paradigma nu are legătură cu sintaxa limbajului!

Paradigma de programare

Legătura cu mașina de calcul

- paradigmele sunt legate teoretic de o mașină de calcul în care prelucrările caracteristice paradigmei se fac la nivelul mașinii;
- dar putem executa orice program, scris în orice paradigmă, pe orice mașină.

Paradigma de programare

Ce o definește

· În principal, paradigma este definită de

- elementele principale din sintaxa limbajului – e.g. existența și semnificația variabilelor, semnificația operatorilor asupra datelor, modul de construire a programului;
- modul de construire al tipurilor variabilelor;
- modul de definire și statutul operatorilor – elementele principale de prelucrare a datelor din program (e.g. obiecte, funcții, predicate);
- legarea variabilelor, efecte laterale, transparență referențială, modul de transfer al parametrilor pentru elementele de prelucrare a datelor.

Variabile și valori de prim rang

Variabile

Nume date unor valori

- în majoritatea limbajelor există variabile, ca NUME date unor valori – rezultatul anumitor procesări (calcule, inferențe, substituții);
- variabilele pot fi o referință pentru un spațiu de memorie sau pentru un rezultat abstract;
- elementele de procesare a datelor pot sau nu să fie valori de prim rang (să poată fi asociate cu variabile).

Funcții ca valori de prim rang

Definiție

+ | Valoare de prim rang – O valoare care poate fi:

- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcții
- întoarsă dintr-o funcție

Ex | Să se scrie funcția `compose`, ce primește ca parametri alte 2 funcții, `f` și `g`, și întoarce funcția obținută prin compunerea lor, `f ∘ g`.

Funcții ca valori de prim rang: Compose

C

```
1  int compose(int (*f)(int), int (*g)(int), int x) {
2      return (*f)((*g)(x));
3  }
```

- în C, funcțiile nu sunt valori de prim rang;
- pot scrie o funcție care compune două funcții pe o anumită valoare (ca mai sus)
- pot întoarce pointer la o funcție existentă
- dar nu pot crea o referință (pointer) la o funcție nouă, care să fie folosit apoi ca o funcție obișnuită

Funcții ca valori de prim rang:
Java


```
1  abstract class Func<U, V> {
2      public abstract V apply(U u);
3
4      public <T> Func<T, V> compose(final Func<T, U> f) {
5          final Func<U, V> outer = this;
6
7          return new Func<T, V>() {
8              public V apply(T t) {
9                  return outer.apply(f.apply(t));
10             }
11         };
12     }
13 }
```

- În Java, funcțiile nu sunt valori de prim rang – pot crea rezultatul dar este complicat, și rezultatul nu este o funcție obișnuită, ci un obiect.

Funcții ca valori de prim rang: `Compose`

Racket & Haskell

- Racket:

```
1 (define compose
2   (lambda (f g)
3     (lambda (x)
4       (f (g x)))))
```

- Haskell:

```
1 compose = (.)
```

- În Racket și Haskell, funcțiile sunt valori de prim rang.
- mai mult, ele pot fi aplicate parțial, și putem avea funcționale – funcții care iau alte funcții ca parametru.

Tipare a variabilelor

Modalități de tipare

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

\+ | **Tipare** – modul de gestionare a tipurilor.

Modalități de tipare

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ | Tipare – modul de gestionare a tipurilor.

- Clasificare după momentul verificării:
  - statică
  - dinamică
- Clasificare după rigiditatea regulilor:
  - tare
  - slabă

Tipare statică vs. dinamică

Exemplu

Ex Exemplu — Tipare dinamică

Javascript:

```
var x = 5;
if(condition) x = "here";
print(x);
```

→ ce tip are `x` aici?

Ex Exemplu — Tipare statică

Java:

```
int x = 5;
if(condition)
    x = "here";
print(x);
```

→ Eroare la *compilare*: `x` este `int`.

Tipare statică vs. dinamică

Caracteristici

Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă
- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Permite metaprogramare (v. `eval`)
- Python, Scheme/Racket, Prolog, JavaScript, PHP

Tipare tare vs. slabă

Exemple

- Clasificare după libertatea de a agrega valori de tipuri diferite.

Ex Tipare tare

Exemplu

`1 + "23"` → Eroare (Haskell, Python)

Ex Tipare slabă

Exemplu

`1 + "23" = 24` (Visual Basic)

`1 + "23" = "123"` (JavaScript)

Legarea variabilelor

Legarea variabilelor

Impactul asupra programului

- două posibilități esențiale:
  - un nume este întotdeauna legat (într-un anumit context) la aceeași valoare / la același calcul $\Rightarrow$ numele stă pentru un calcul;
    - legare statică.
  - un nume (aceeași variabilă) poate fi legat la mai multe valori pe parcursul execuției $\Rightarrow$ numele stă pentru un spațiu de stocare – fiecare element de stocare fiind identificat printr-un nume;
    - legare dinamică.

Efecte laterale (*side effects*)

Definiție

Ex | Exemplu În expresia `2 + (i = 3)`, subexpresia `(i = 3)`:

- produce valoarea `3`, conducând la rezultatul `5` al întregii expresii;
- are efectul lateral de inițializare a lui `i` cu `3`.

\+ | **Efect lateral** Pe lângă valoarea pe care o produce, o expresie sau o funcție poate modifica starea globală.

- Inerente în situațiile în care programul interacționează cu exteriorul → I/O!

Efecte laterale (*side effects*)

Consecințe

Ex În expresia `x-- + ++x`, cu `x = 0`:

- evaluarea stânga → dreapta produce `0 + 0 = 0`
- evaluarea dreapta → stânga produce `1 + 1 = 2`
- dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem

  `x + (x + 1) = 0 + 1 = 1`

- Importanța ordinii de evaluare!
- Dependențe implicite, puțin lizibile și posibile generatoare de bug-uri.

Efecte laterale (*side effects*)

Consecințe asupra programării lenești

- În prezența efectelor laterale, programarea lenișă devine foarte dificilă;
- Efectele laterale pot fi gestionate corect numai atunci când secvența evaluării este garantată $\rightarrow$ garanție inexistentă în programarea lenișă.
  - nu știm când anume va fi nevoie de valoarea unei expresii.

Transparență referențială

Pentru expresii

+ | Transparență referențială Confundarea unui obiect (“valoare”) cu referința la acesta.

+ | Expresie transparentă referențial: posedă o unică valoare, cu care poate fi substituită, păstrând semnificația programului.

Ex | Exemplu

- `x-- + ++x` → nu, valoarea depinde de ordinea de evaluare
- `x = x + 1` → nu, două evaluări consecutive vor produce rezultate diferite
- `x` → ar putea fi, în funcție de statutul lui `x` (globală, statică etc.)

Transparență referențială

Pentru funcții

+ | Funcție transparentă referențial: rezultatul întors depinde exclusiv de parametri.

Ex | Exemplu

```
int transparent(int x) {
    return x + 1;
}
```

```
int g = 0;

int opaque(int x) {
    return x + ++g;
}
```

- `opaque(3) - opaque(3) != 0`!
- Funcții transparente: `log`, `sin` etc.
- Funcții opace: `time`, `read` etc.

Transparență referențială

Avantaje

- Lizibilitatea codului;
- Demonstrarea formală a corectitudinii programului – mai ușoară datorită lipsei stării;
- Optimizare prin reordonarea instrucțiunilor de către compilator și prin caching;
- Paralelizare masivă, prin eliminarea modificărilor concurente.

Modul de evaluare

Evaluare

Mod de evaluare și execuția programelor

- modul de evaluare al expresiilor dictează modul în care este executat programul;
- este legat de funcționarea mașinii teoretice corespunzătoare paradigmei;
- ne interesează în special ordinea în care expresiile se evaluează;
- în final, întregul program se evaluează la o valoare;
- important în modul de evaluare este modul de evaluare / transfer a parametrilor.

Transferul parametrilor

- Evaluare aplicativă – parametrii sunt evaluați înainte de evaluarea corpului funcției.
  - *Call by value*
  - *Call by sharing*
  - *Call by reference*

- Evaluare normală – funcția este evaluată fără ca parametrii să fie evaluați înainte.
  - *Call by name*
  - *Call by need*

Call by value

În evaluarea aplicativă

Ex | Exemplu

```
1  // C sau Java
2  void f(int x) {
3       x = 3;
4  }
```

```
1  // C
2  void g(struct str s) {
3       s.member = 3;
4  }
```

· Efectul liniilor 3 este invizibil la apelant.

- Evaluarea parametrilor înaintea aplicației funcției și transferul unei copii a valorii acestuia
- Modificări locale invizibile la apelant
- C, C++, tipurile primitive Java

Call by sharing

În evaluarea aplicativă

- Variantă a *call by value*;
- Trimiterea unei referințe la obiect;
- Modificări locale asupra referinței invizibile la apelant;
- Modificări locale asupra obiectului referit vizibile la apelant;
- Racket, Java;

Call by reference

În evaluarea aplicativă

- Trimiterea unei referințe la obiect;
- Modificări locale asupra referinței și obiectului referit vizibile la apelant;
- Folosirea “&” în C++.

Call by name

În evaluarea normală

- Argumente neevaluate în momentul aplicării funcției → substituție directă (textuală) în corpul funcției;
- Evaluare parametrilor la cerere, de fiecare dată când este nevoie de valoarea acestora;
- în calculul λ.

Call by need

În evaluarea normală

- Variantă a *call by name*;
- Evaluarea unui parametru doar la prima utilizare a acestuia;
- Memorarea valorii unui parametru deja evaluat și returnarea acesteia în cazul utilizării repetate a aceluiași parametru (datorită transparenței referențiale, o aceeași expresie are întotdeauna aceeași valoare) – memoizare;
- în Haskell.

Sfârșitul cursului 9
Elemente esențiale

- caracteristicile unei paradigme;
- variabile, funcții ca valori de prim rang;
- legare, efecte laterale, transparență referențială;
- evaluare și moduri de transfer al paramet

+ Dați feedback la acest curs aici:
[https://docs.google.com/forms/d/e/1FAIpQLSeY7VuAt5n6hyHHnNUplLWfWt7UkJBGhkviewform]