Paradigme de Programare

Ș.l. dr. ing. Andrei Olaru

Departamentul Calculatoare

2015 – 2016, semestrul 2

Cursul 2: Programare funcţională în Racket

tipuri

```
(define f (lambda (x) (or (list? x) (f (car x)))))
```

legare dinamică / statică — recusivitate

Cursul 2: Programare funcţională în Racket

Introducere

Racket vs. Scheme

Cum se numește limbajul despre care discutăm?

- Racket este dialect de Lisp/Scheme (așa cum Scheme este dialect de Lisp);
- la nivelul studiat, Racket este identic cu Scheme;
- Racket este derivat din Scheme, oferind instrumente mai puternice;
- Racket (fost PLT Scheme) este interpretat de mediul DrRacket (fost DrScheme);

[http://en.wikipedia.org/wiki/Racket_(programming_language)]

[http://racket-lang.org/new-name.html]

Discuție despre tipare

Tipuri în Racket

În Racket avem:

- numere: `1`, `2`, `1.5`
- simboli (literali): `'abcd`, `'andrei`
- valori booleene: `#t`, `#f`
- șiruri de caractere: `"șir de caractere"`

- perechi: `(cons 1 2)` $\rightarrow$ `'(1 . 2)`
- liste: `(cons 1 (cons 2 '()))` $\rightarrow$ `'(1 2)`

- funcții: `(λ (e f) (cons e f))` $\rightarrow$ `#<procedure>`

· Cum sunt gestionate tipurilor valorilor (variabilelor) la compilare (verificare) și la execuție?

Modalităţi 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 Tipare dinamică

Exemplu

Javascript:

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

→ ce tip are `x` aici?

Ex Tipare statică

Exemplu

Java:

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

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

Tipare statică vs. dinamică

Caracteristici

λPP

Tipare statică

- La compilare
- Valori şi variabile
- Rulare mai rapidă
- Rigidă: sancţionează orice construcţie
- Debugging mai facil
- Declaraţii explicite sau inferenţe de tip
- Pascal, C, C++, Java, Haskell

Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancţionează 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)

Ex Tipare slabă

Exemplu

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

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

Tiparea în Racket

- este dinamică

```
1  (if #t 'something (+ 1 #t)) → 'something
2  (if #f 'something (+ 1 #t)) → Eroare
```

- este tare

```
1  (+ "1" 2) → Eroare
```

- dar, permite liste cu elemente de tipuri diferite.

Legarea variabilelor

Variabile (Nume)
Proprietăți generale ale variabilelor

λPP

- Proprietăţi
  - identificator
  - valoarea legată (la un anumit moment)
  - domeniul de vizibilitate (*scope*) + durata de viaţă
  - tip
- Stări
  - declarată: cunoaştem identificatorul
  - definită: cunoaştem şi valoarea $\rightarrow$ variabila a fost *legată*

· în Racket, variabilele (numele) sunt legate *static* prin construcțiile `lambda`, `let`, `let*`, `letrec`, și sunt vizibile în domeniul construcției unde au fost definite (excepție face `define`).

Legarea variabilelor
Definiții (1)

+ | Legarea variabilelor – modalitatea de asociere a apariţiei unei variabile cu definiţia acesteia (deci cu valoarea).

+ | Domeniul de vizibilitate – *scope* – mulţimea punctelor din program unde o definiţie (legare) este vizibilă.

- Este determinat de modalitatea de legare a variabilelor.

Legarea variabilelor
Definiții (2)

+ | Legare statică – Variabilele din corpul unei expresii sunt extrase din contextul în care aceasta a fost definită.

- Domeniu de vizibilitate determinat prin construcţiile limbajului, putând fi desprins la compilare.

+ | Legare dinamică – Valorile variabilelor depind de momentul în care o expresie este evaluată.

- Domeniu de vizibilitate (al unei legări) determinat la execuţie.

Legarea variabilelor în Racket

- Variabile declarate (! și definite) în construcții interioare → legate static
  - `lambda`
  - `let`
  - `let*`
  - `letrec`
- Variabile *top-level* → legate dinamic
  - `define`

Construcţia `lambda`
Definiţie & Exemplu

- Leagă static parametrii formali ai unei funcţii
- Sintaxă:

```
1 (lambda (p1 ... pk ... pn) expr)
```

- Domeniul de vizibilitate al parametrului `pk`: mulţimea punctelor din `expr` (care este corpul funcţiei), puncte în care apariţia lui `pk` este liberă.

Construcţia `lambda`
Semantică

- Aplicaţie:

```
1  ((lambda (p1 ... pn) expr)
2     a1 ... an)
```

1. Evaluare aplicativă: se evaluează argumentele `ak`, în ordine aleatoare (nu se garantează o anumită ordine).
2. Se evaluează corpul funcţiei, `expr`, ţinând cont de legările `pk` $\leftarrow$ *valoare*(`ak`).
3. Valoarea aplicaţiei este valoarea lui `expr`.

Construcţia `let`
Definiţie, Exemplu, Semantică

- Leagă static variabile locale
- Sintaxă:

```
1  (let ( (v1 e1) ... (vk ek) ... (vn en) )
2    expr)
```

- Domeniul de vizibilitate a variabilei `vk` (cu valoarea `ek`): mulţimea punctelor din `expr` (corp let), în care apariţiile lui `vk` sunt libere.

Ex | Exemplu

```
1  (let ((x 1) (y 2))  (+ x 2))
```

· Atenție! Construcția `(let ((v1 e1) ...(vn en)) expr)` – echivalentă cu `((lambda (v1 ...vn) expr) e1 ...en)`

Construcţia `let*`
Definiţie & Exemplu

- Leagă static variabile locale
- Sintaxă:

```
1  (let* ((v1 e1) ... (vk ek) ... (vn en))
2     expr)
```

- Scope pentru variabila `vk` = mulţimea punctelor din
  - restul legărilor (legări ulterioare) şi
  - corp – `expr`

  în care apariţiile lui `vk` sunt libere

Ex | Exemplu

```
1  (let* ((x 1) (y x))
2     (+ x 2))
```

Construcţia `let*`
Semantică

```
(let* ((v1 e1) ... (vn en))
  expr)
```

echivalent cu

```
(let ((v1 e1))
  ...
    (let ((vn en))
      expr) ... )
```

- Evaluarea expresiilor `ei` se face în ordine!

Construcţia `letrec`
Definiţie

- Leagă static variabile locale
- Sintaxă:

```
1 (letrec ((v1 e1) ... (vk ek) ... (vn en))
2       expr)
```

- Domeniul de vizibilitate a variabilei `vk` = mulţimea punctelor din întreaga construcţie, în care apariţiile lui `vk` sunt libere

Construcţia `letrec`
Exemplu

Ex Exemplu

```
1 (letrec ((factorial
2             (lambda (n)
3               (if (zero? n) 1
4                   (* n (factorial (- n 1)))))))
5   (factorial 5))
```

Construcţia `define`
Definiție & Exemplu

- Leagă dinamic variabile *top-level*.
- Avantaje:
  - definirea variabilelor *top-level* în orice ordine
  - definirea de funcţii mutual recursive

Ex Definiții echivalente:

```
1  (define f1
2       (lambda (x)
3            (add1 x)
4  ))
5
6  (define (f2 x)
7       (add1 x)
8  ))
```

Construcția `define`
Exemplu de legare dinamică

- în Scheme (e.g. limbajul Pretty Big), `define` leagă dinamic și permite definiri multiple (în Racket nu mai este acceptat acest comportament):

Exemplu

```
1 (define x 0)
2 (define f (lambda () x))
3 (f)
4 (define x 1)
5 (f)
```

Output:

- Dezavantaj: codul devine neintuitiv și se corupe transparenţa referenţiale

Construcția `define`
Exemplu de legare dinamică

- în Scheme (e.g. limbajul Pretty Big), `define` leagă dinamic și permite definiri multiple (în Racket nu mai este acceptat acest comportament):

Exemplu

```
1 (define x 0)
2 (define f (lambda () x))
3 (f)
4 (define x 1)
5 (f)
```

Output: `0 1`

- Dezavantaj: codul devine neintuitiv și se corupe transparența referențiale

Evaluare

Evaluarea în Racket

- Evaluare aplicativă: evaluarea parametrilor înaintea aplicării funcţiei asupra acestora (în ordine aleatoare).

- Funcţii stricte (i.e.cu evaluare aplicativă)
  - Excepţii: `if`, `cond`, `and`, `or`, `quote`.

Controlul evaluării

- `quote` sau ’
  - funcţie nestrictă
  - întoarce parametrul neevaluat
- `eval`
  - funcţie strictă
  - forţează evaluarea parametrului şi întoarce valoarea acestuia

Ex Exemplu

```
1 (define sum '(2 + 3))
2 sum ; (2 + 3)
3 (eval (list (cadr sum) (car sum) (caddr sum))) ; 5
```

Construcția programelor prin recursivitate

Recursivitate

- Recursivitatea – element fundamental al paradigmei funcționale
  - Numai prin recursivitate (sau iterare) se pot realiza prelucrări pe date de dimensiuni nedefinite.

- Dar, este eficient să folosim recursivitatea?
  - recursivitatea (pe stivă) poate încărca stiva.

Recursivitate
Tipuri

- pe stivă: $factorial(n) = n * factorial(n-1)$
  - timp: liniar
  - spațiu: liniar (ocupat pe stivă)
  - dar, în procedural putem implementa factorialul în spațiu constant.

Recursivitate
Tipuri

- pe stivă: $factorial(n) = n * factorial(n-1)$
  - timp: liniar
  - spațiu: liniar (ocupat pe stivă)
  - dar, în procedural putem implementa factorialul în spațiu constant.

- pe coadă:
  $factorial(n) = fH(n,1)$
  $fH(n,p) = fH(n-1, p*n)\ ,\ n > 1\ ;\ p$ altfel
  - timp: liniar
  - spațiu: constant

- beneficiu *tail call optimization*

Sfârșitul cursului 2
Elemente esențiale

- Tipare: dinamică vs. statică, tare vs. slabă;
- Legare: dinamică vs statică;
- Racket: tipare dinamică, tare; domeniu al variabilelor;
- construcții care leagă nume în Racket: lambda, let, let*, letrec, define;
- evaluare aplicativă;
- construcția funcțiilor prin recursivitate.

+ Dați feedback la acest curs aici:
[http://goo.gl/forms/SjDsW06v5J]