Scheme-kesäkurssi luento 4

Riku Saikkonen

8. 7. 2009

Sisältö

Laiska evaluointi (SICP 4.2.1)

- laiska evaluointi (lazy, normal-order, non-strict): proseduurikutsun argumentit evaluoidaan vasta kun niitä tarvitaan esim. tulostamiseen
- laiskasti evaluoidulla kielellä `(define (unless p t e) (if p e t))` toimii
- laiskat listat ≈ virrat (ei tarvitse erikseen virtoja)
- käytössä muutamissa oikeissa kielissä, mm. Haskell
- koodista vaikea nähdä milloin sitä suoritetaan
  - sivuvaikutuksellista koodia hyvin vaikea ymmärtää
  - koodin suoritusaikaa ja varsinkin muistinkäyttöä usein vaikea arvioida
- tavallisemmassa Schemessä on manuaalisempi tapa laskea laiskasti: `delay` ja `force`

Memoisaatio ja thunkit (SICP 4.2.2)

Miten tulkissa toteutetaan vielä laskematta oleva arvo?

- uusi tyyppi: "thunk", joka sisältää evaluoimattoman lausekkeen ja sen ympäristön (ei näy ohjelmoijalle)
- jos samaa arvoa tarvitaan monta kertaa:
  - joko se lasketaan aina uudelleen (harvinaista)
  - tai korvataan thunk lasketulla arvolla (ns. memoisaatio)
- kirjan tulkin esitysmuoto: `(thunk lauseke ympäristö)` ja `(evaluated-thunk arvo)`
- toteutus tulkissa apuproseduureilla `(delay-it exp env)` ja `(force-it obj)`

Laiskan tulkin toteutus (SICP 4.2.2)

Muutokset kohdan 4.1 tulkkiin

```
(define (actual-value exp env) (force-it (eval exp env)))
(define (eval exp env) ...
  ((application? exp) (apply (actual-value (operator exp) env)
                             (operands exp) env)) ...)
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
          (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown procedure type" procedure))))
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Sisältö

Epädeterminististä ohjelmointia (SICP 4.3.1)

- epädeterministisen laskennan idea: lausekkeella voi olla useita mahdollisia arvoja, joita kokeillaan järjestyksessä
- kirjassa lisätään Scheme-tulkkiin uusi primitiivi `amb`:
  - `(amb 1 2 3)` palauttaa jonkin arvoista `1`, `2` ja `3`
  - `(amb)` palaa taaksepäin ja kokeilee edellisen `amb`in seuraavaa vaihtoehtoa
- voi käyttää erityisesti hakuongelmissa, joissa luodaan paljon mahdollisia ratkaisuja ja karsitaan niistä sopivat eri ehdoilla
  - vrt. kirjan virtaesimerkit
  - `amb`illa ei tarvitse koko ajan käsitellä usean ratkaisun mahdollisuutta

Esimerkki ambin käytöstä: ongelma (SICP 4.3.2)

Pulma kirjasta

Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

Esimerkki ambin käytöstä: ratkaisu (SICP 4.3.2)

Pulman ratkaisu

```
(define (require p) (if (not p) (amb)))
(define (distinct? items)
  (cond ((null? items) true) ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
(define (multiple-dwelling)
  (let ((b (amb 1 2 3 4 5)) (c (amb 1 2 3 4 5))
        (f (amb 1 2 3 4 5)) (m (amb 1 2 3 4 5))
        (s (amb 1 2 3 4 5)))
    (require (distinct? (list b c f m s)))
    (require (not (= b 5)))
    (require (not (= c 1)))
    (require (not (= f 5)))
    (require (not (= f 1)))
    (require (> m c))
    (require (not (= (abs (- s f)) 1)))
    (require (not (= (abs (- f c)) 1)))
    (list (list 'baker b) (list 'cooper c)
          (list 'fletcher f) (list 'miller m)
          (list 'smith s))))
```

Miten amb toteutetaan? (SICP 4.3.3)

- `amb`in laskenta pitää joskus peruuttaa edelliseen haarautumiskohtaan ja kokeilla seuraavaa vaihtoehtoa
- otetaan laskennan tila (kontinuaatio) talteen haarautumiskohdassa ja hypätään siihen takaisin
- kirjassa `amb`-tulkki on toteutettu CPS-tyyliin (ks. edellinen luento), mutta kontinuaatioargumentteja on kaksi:
  - `success`-kontinuaatio on tavallinen CPS-kontinuaatio, jolla laskenta etenee
  - `fail`-kontinuaatiossa pidetään tallessa edellistä haarautumiskohtaa (jossa on viittaus sitä edelliseen jne.)
- tulkki toimii muuten samoin kuin aiemmin, mutta
  - `(amb)` kutsuu `fail`-kontinuaatiota (ilman argumentteja)
  - `(amb 1 2 3)` kutsuu `success`-kontinuaatiota paluuarvolla `1` ja luo uuden `fail`-kontinuaation
  - `set!` luo `fail`-kontinuaation joka peruu sijoituksen palauttaen muuttujan edellisen arvon

ambin toteutus analysoivaan tulkkiin (SICP 4.3.3)

Osa tulkin koodista

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env (lambda (pred-value fail2)
                   (if (true? pred-value)
                       (cproc env succeed fail2)
                       (aproc env succeed fail2)))
              fail))))
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices) (fail)
            ((car choices) env succeed
             (lambda () (try-next (cdr choices))))))
      (try-next cprocs))))
```

ambin voi toteuttaa call/cc:llä

Yksinkertaisen ambin toteutus Scheme-proseduurina

```
(define (final-fail) (set! amb-thunks (list final-fail))
  (error "AMB: Out of choices"))
(define amb-thunks (list final-fail))
(define (amb-list choices)
  (call-with-current-continuation
   (lambda (cont)
     (define (fail)
       (let ((next (car amb-thunks)))
         (set! amb-thunks (cdr amb-thunks)) (next)))
     (define (add-choice! choice)
       (set! amb-thunks (cons (lambda () (cont choice))
                              amb-thunks)))
     (for-each add-choice! choices)
     (fail))))
(define (amb . choices) (amb-list choices))
```

- mutta tässä esim. set! ei peru muutoksiaan
- ja ambin argumentit evaluoidaan etukäteen (voisi estää makrolla) ja käsitellään lopusta alkuun

Sisältö

Logiikkaohjelmointi (SICP 4.4)

- logiikkaohjelmointi on oma ohjelmointiparadigmansa (yleisin kieli Prolog)
- logiikkaohjelmassa on:
  - joukko tosiasioita (fact) ja sääntöjä (rule)
  - joihin tehdään kyselyitä (query) esim. interaktiivisesti
- logiikkaohjelma on eräänlainen ”tietokanta”, josta voi etsiä kyselyyn sopivia tosiasioita tai sääntöjen sovelluksia
- ei täysin deklaratiivista: sääntöjen ohjelmoijan pitää miettiä mm. niiden sovittamisjärjestystä (ks. SICP 4.4.3)
- säännöt vastaavat proseduureja, mutta korkeamman tason abstraktioita kuten proseduuriargumentteja ei ole
- kirjassa on hyvin yksinkertainen logiikkaohjelmointikieli
  - esim. Prologissa on lisäksi I/O:ta, tosiasioita muuttava sijoituslause ja hakuja katkaiseva ”cut”-operaattori

Esimerkki: (append-to-form x y z) (SICP 4.4.1)

Määritelmä (säännöt)

```
(rule (append-to-form () ?y ?y))
(rule (append-to-form (?u . ?v) ?y (?u . ?z))
      (append-to-form ?v ?y ?z))
```

Kyselyesimerkkejä

```
(append-to-form (a b) (c d) ?z)
⇒ (append-to-form (a b) (c d) (a b c d))
(append-to-form (a b) ?y (a b c d))
⇒ (append-to-form (a b) (c d) (a b c d))
(append-to-form ?x ?y (a b c d))
⇒ (append-to-form () (a b c d) (a b c d))
   (append-to-form (a) (b c d) (a b c d))
   (append-to-form (a b) (c d) (a b c d))
   (append-to-form (a b c) (d) (a b c d))
   (append-to-form (a b c d) () (a b c d))
```

Esimerkki: "tietokanta" (SICP 4.4.1)

Tietokanta (tosiasiat)

```
(address (Fect Cy D) (Cambridge (Ames Street) 3))
(job (Fect Cy D) (computer programmer))
(salary (Fect Cy D) 35000)
(supervisor (Fect Cy D) (Bitdiddle Ben))
...
```

Kyselyesimerkkejä

```
(job ?x (computer programmer))
⇒ (job (Hacker Alyssa P) (computer programmer))
   (job (Fect Cy D) (computer programmer))
(and (job ?person (computer programmer))
     (address ?person ?where))
⇒ (and (job (Hacker Alyssa P) (computer programmer))
       (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)))
   (and (job (Fect Cy D) (computer programmer))
       (address (Fect Cy D) (Cambridge (Ames Street) 3)))
(and (salary ?person ?amount)
     (lisp-value > ?amount 30000)) ⇒ ...
```

Logiikkatulkin toteutus (SICP 4.4.2, 4.4.4)

- kokonaan uusi tulkki
- tulostaa kaikki kyselyn vastaukset (esim. Prolog tulostaa niitä yksi kerrallaan samaan tapaan kuin `amb`-tulkki)
- perustuu hahmonsovitukseen: käydään tosiasioita läpi järjestyksessä ja yritetään sovittaa niitä kyselyyn
  - kyselyssä voi olla muuttujia, tosiasioissa ei
  - samankaltaista hahmonsovitusta on myös monissa funktionaalisissa kielissä (esim. Haskell ja ML) ja Schemen `define-syntax`-makroissa
- ja unifikaatioon: etsii sääntöjä, joiden lopputulokset sopivat kyselyyn
  - molemmissa voi olla muuttujia
- käyttää virtoja peruuttavaan hakuun (`amb`iakin voisi periaatteessa käyttää)

Sisältö

Rekisterikoneen kieli (SICP 5.1)

- yksinkertainen "konekieli" (oikeammin assembler), tavoitteena ymmärtää tulkkien ja kääntäjän matalan tason toimintaa
  - vakiomäärä rekistereitä (saa valita itse), joihin voi tallettaa joko Scheme-arvon tai koodiosoitteen
  - pino-operaatiot käskyinä (osoitteita ei tarvitse laskea)
  - primitiivioperaatiot saa valita itse (esim. `+`, `cons`)
- muistia saa käyttöön joko `cons`-primitiivillä tai kohdassa 5.3 tehdyllä simuloidulla muistilla
- koodi ei ole samassa muistissa kuin data (ns. Harvard-arkkitehtuuri)

Rekisterikone-esimerkki (SICP 5.1.1)

Scheme

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Rekisterikone (jakojäännös rem-primitiivillä)

```
(controller
 test-b
   (test (op =) (reg b) (const 0))
   (branch (label gcd-done))
   (assign t (op rem) (reg a) (reg b))
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
 gcd-done)
```

Rekursioesimerkki (SICP 5.1.4)

Rekursiivinen kertoma

```
(controller
   (assign continue (label fact-done))
 fact-loop
   (test (op =) (reg n) (const 1))
   (branch (label base-case))
   (save continue)
   (save n)
   (assign n (op -) (reg n) (const 1))
   (assign continue (label after-fact))
   (goto (label fact-loop))
 after-fact
   (restore n)
   (restore continue)
   (assign val (op *) (reg n) (reg val))
   (goto (reg continue))
 base-case
   (assign val (const 1))
   (goto (reg continue))
 fact-done)
```

SICP luku 5

Kirjan luvussa 5 on:

- rekisterikonekielen määritelmä (5.1)
- Schemellä tehty rekisterikonekielen tulkki (5.2)
- muistinhallintaa ja roskankeruuta (5.3)
- rekisterikonekielellä tehty Scheme-tulkki (5.4)
- Schemellä tehty kääntäjä Schemestä rekisterikoneelle (5.5)
- interaktiivinen tulkki, josta voi kutsua kääntäjää ja käännettyjä proseduureja (5.5.7)