Scheme-kesäkurssi luento 6

Timo Lilja

3. 8. 2009

Sisältö

Lausekkeiden kääntäminen (SICP 5.5.2)

- jokaiselle lauseketyypille oma koodigeneraattori, joka päättyy linkityskäskyihin:

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          `((goto (label ,linkage)))))))
```

Sijoitusten kääntäminen

- generoidaan rekursiivisesti koodi, joka laskee sijoituslausekkeen arvon ja liitetään se osaksi koodia, joka tekee varsinaisen sijoituksen:

```
(define (compile-assignment exp target linkage)
  (let ((var (assignment-variable exp))
        (get-value-code
         (compile (assignment-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence
        '(env val)
        (list target)
       `((perform (op set-variable-value!)
                  (const ,var)
                  (reg val)
                  (reg env))
         (assign ,target (const ok))))))))
```

Yhdistelmien kääntäminen (SICP 5.5.3)

- proseduurikutsut käännetään kaavalla:

```
<compilation of operator, target proc, linkage next>
<evaluate operands and construct argument list in argl>
<compilation of proc call with given target, linkage>
```

- rekisterit `env`, `proc` ja `argl` täytyy tallettaa operaattorien ja operandien evaluoinnin aikana
- toiminta analogista metasirkulaarisen tulkin eval-apply-syklin kanssa

Kääntäjäesimerkki ja optimointeja (SICP 5.5.5–5.5.7)

- luvussa 5.5.5 esimerkki factorial-funktion kääntämisestä
- luvussa 5.5.6 optimointi muuttujien arvojen etsimisestä
  - ympäristö koostuu kehyksistä joissa muuttujat ovat järjestyksessä.
  - määritellään leksikaalinen osoite joka kertoo suoraan, missä kehyksessä ja paikassa kyseinen muuttuja on
  - säästetään ympäristön turha läpikäynti
- luvussa 5.5.7 lisätään eksplisiittisen kontrollin tulkkiin tuki käännetyille proseduureille
  - muutos suoraviivainen: apply-dispatchiin tuki käännetylle proseduriityypille ja muutama ”liimakoodi”, jotta käännetty koodi saadaan käyttöön
  - monet lisp-tulkit tukevat käännettyjä proseduureja vastaavasti

Sisältö

Common Lisp

- kielistandardi, ANSI INCITS 226-1994 (R2004)
- muuttujat ja funktiot eri nimiavaruuksissa
- tukee sekä dynaamista että leksikaalista skooppia
- Schemessä totuusarvot `#t` ja `#f`,
  Common Lispissä `T` ja `NIL`.
- häntärekursio-optimoinnit ei pakollisia, mutta useat implementaatiot tukevat
- `defmacro`-tyyppinen makrojärjestelmä
- laajat standardikirjastot, iso spesifikaatio
- saatavilla useita implementaatioita
  - implementaatioissa usein hyvät debuggerit

Common Lisp Object System

- määrittely aloitettiin vuonna 1984 ja valmistui ANSI Common Lisp -standardissa vuonna 1990.
- koostuu geneerisistä funktioista perinteisemmän viestinvälityksen sijaan
- tukee moniperintää
- kaikki primääriset tyypit ovat ensimmäisen luokan objekteja
- CLOSin yleisimmin käytetyt määrittelyt ovat `defclass`, `defgeneric` ja `defmethod`. Instanssit luodaan kutsumalla funktiota `make-instance`

defclass

- Makro jolla luokkia määritellään on nimeltään `defclass`:

```
(defclass point ()
  (x
   y
   z))
```

- Määrittely ei tuota konstruktoreita, predikaattifunktioita tai aksessorifunktiota automaattisesti vaan ne pitää eksplisiittisesti pyytää
  - Common Lispin `defstruct`-rakenne generoi em. automaattisesti

instantiointi

- instantiointi tapahtuu kutsumalla funktiota `make-instance`:

```
(defclass point () (x y z))
(setf my-point (make-instance 'point))
(defun set-point-values (point x y z)
                (setf (slot-value point 'x) x
                      (slot-value point 'y) y
                      (slot-value point 'z) z))
(defun distance-from-origin (point)
                (with-slots (x y z)
                    point
                  (sqrt (+ (* x x) (* y y) (* z z)))))
```

Slots

- defclass-rakenteessa voi määritellä muuttujille ns. slot-määritteitä (slot-specifier):
  - :accessor määrittelee aksessointifunktiot
  - :reader määrittelee metodin jolla voidaan lukea muuttuja
  - :initarg määrittelee avainsanan joka on kelvollinen alustusarvo
  - :initform määrittelee oletusarvon jos alustusarvoa ei ole annettu
  - :allocation määrittelee, onko kyseessä instanssi- vai luokkamuuttuja

Esimerkki

```
(defclass daft-point ()
          ((x :accessor daft-x :initarg :x)
           (y :accessor daft-y :initform 3.14159)
           (z :reader daft-z :allocation :class)))
(setf (slot-value (make-instance 'daft-point) 'z) 42)
(setf my-daft-point (make-instance 'daft-point :x 19))
(list (daft-x my-daft-point)
      (daft-y my-daft-point)
      (daft-z my-daft-point))
=> (19 3.14159 42)
```

Aliluokat ja perintä

- CLOSissa määritellään aliluokat seuraavasti:

```
(defclass animal ()
          ((legs :reader leg-count :initarg :legs)
           (comes-from :reader comes-from
            :initarg :comes-from)))
(defclass mammal (animal)
          ((diet :initform 'antelopes :initarg :diet)))
(defclass aardvark (mammal)
          ((cute-p :accessor cute-p :initform nil)))
```

- Funktiolla `class-direct-superclass` saadaan lista käyttäjän määrittämistä yläluokista ja `class-precedence-list` näyttää kaikki yläluokat
- CLOSissa kaikki objetkit perivät luokat `standard-object` ja `t`, joka on CL:n perustyyppi

Moniperintä

- CLOS tukee moniperintää

```
(defclass figurine ()
          ((potter :accessor made-by :initarg :made-by)
           (comes-from :initarg :made-in)))
(defclass figurine-aardvark (aardvark figurine)
           ((name :reader aardvark-name
            :initarg :aardvark-name)
            (diet :initform nil)))
```

- mikäli moniperityissä luokissa on samannimisiä muuttujia, muodostetaan instantioituun objektiin yksi muuttuja, joka on näiden muuttujien yhdistelmä:
  - `:accessor` ja `:reader` unioni kaikista perityistä aksessoreista ja lukijoista
  - `:initarg` unioni kaikista muuttujalle määritellyistä avainsanoita
  - `:initform` spesifisin initialisointiarvo

Metodit

- metodeja määritellään defmethod-avainsanalla ja määrittely sisältää hahmon soveltamisen (pattern matching), jonka perusteella oikeaa metodia kutsutaan:

```
(defmethod my-describe (thing)
  (format t "~s could be anything"))
(defmethod my-describe ((animal animal))
  (format t "~s is an animal" animal))
(my-describe Eric)
=> #<ANTELOPE 2112B44C> is an animal
(my-describe (make-instance 'figurine))
=> could be anything
```

- kutsusyntaksi ei eroa tavallisesta funktiokutsusta
- hahmonsovitus onnistuu jos kutsussa on luokka tai sen aliluokka

Geneeriset funktiot ja multimetodit

- CLOSissa metodit ovat ”irrallaan” luokista
- perinteisimmässä oliokielissä `self`-argumentti on erityinen
  - CLOSissa voi spesialisoida minkä tahansa argumentin perusteella:

```
(defmethod collide-with ((x asteroid)
                         (y asteroid))
  ;; deal with asteroid hitting asteroid
  )
(defmethod collide-with ((x asteroid)
                         (y spaceship))
  ;; deal with asteroid hitting spaceship
  )
```

  - Useissa kielissä mahdollista ”simuloida” multimetodeja, Pythonissa dekoraattorit

Lopuksi

- CLOSissa paljon muutakin toiminallisuutta
- ei-konventionaalinen oliorajapinta, jonka tunteminen vähintäänkin yleissivistävää
- CLOSmaisia ominaisuuksia tullut/tulossa yleisemmin käytössä oleviin kieliin

Lähteitä

- en.wikipedia.org/wiki/Common_Lisp_Object_System
- en.wikipedia.org/wiki/Multiple_dispatch
- www.ravenbrook.com/doc/2003/07/15/clos-fundamentals/
- "Common Lisp the Language, 2nd edition"; Guy L. Steele Jr www-2.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Sisältö

Funktionaalinen ohjelmointi ja teollisuus

Miksi funktionaaliset kielet eivät menesty teollisuudessa?

- tutkimuskielen asema
- implementaatioiden laatu
  - työkalujen puute, standardien puute, standardikirjastot vailinaisia
- rekrytointipoolin/käyttäjäkunnan pienuus
  - onko projektissa enää parin vuoden päästä yhtään alkuperäistä kehittäjää?
- synergia: onko firman parempi käyttää vain muutamaa tunnettua tekniikkaa vai isoa joukkoa erilaisia työkaluja
- asenteet: elitistinen/akateeminen ”lelu”

Miksi funktionaalisten kielten pitäisi menestyä?

- joissain tilanteissa oman ohjelmointikielen kehittäminen voi kannattaa
  - Erlang: omien sanojensa mukaan paras tekniikka puhelinverkkojen hallintaan
- rekrytointi: osaavaan mainstream-kielenkin ohjelmoijan rekrytointi vaikeata
- funktionaalisten kielten rakenteet mahdollistavat parempien abstraktioiden tekemisen
  - koodi tehokkaampaa, toimivampaa ja tuottavampaa – vai onko?

Funktionaalisten kielten käyttäjiä

- SSH Communications Security `http://www.ssh.com/` (Scheme, SML)
- Mallintarkastusta:
  - Conformiq `http://www.conformiq.com/` (Scheme, Lisp)
  - Galois (Haskell) `http://www.galois.com/blog/2009/04/27/engineering-large-projects-in-haskell-a-decade-of-`
- Ericsson (Erlang) `http://www.erlang.se/workshop/2004/ulfwiger.pdf`
- Sijoitus- ja pankkitoimintaa:
  - Credit Suisse (Haskell)
  - Jane Street Capital (OCaml) `http://www.janestreet.com/technology/ocaml.php`
- Linspire (Haskell) `http://cufp.galois.com/2006/slides/CliffordBeshers.pdf`

Rinnakkaisohjelmointi

- moniydinarkkitehtuurit yleistymässä
  - rinnakkaisohjelmointi entistä yleisempää
- funktionaalisissa kielissä koodin sivuvaikutuksellisuutta pyritään minimoimaan
  - välttää perinteistä jaetun tilan säikeistämisen yleiset ongelmat
  - lukitukset ym. helpompia
- puhtaasti funktionaalisten kielten automaattinen rinnakkaistaminen helpompaa
- GPGPU/vektorikoneet yleistymässä
- paljon tutkittavaa vielä, implementaatioiden taso vaihteleva

Tulevaisuus

- mainstream-kielet ovat omaksuneet FP-kielten ominaisuuksia
- hybridikielet joissa voi kirjoittaa puhtaasti funktionaalisia osia
- yleissivistävää tuntea eri ohjelmointiparadigmojen kieliä
- tuleeko 2010-luvusta funktionaalisten kielten vuosikymmen?

Lähteitä

- Commercial Users of Functional Programming `http://cufp.galois.com/`
- Journal of Functional Programming `http://journals.cambridge.org/action/displayJournal?jid=JFP`
- `www.haskell.org/haskellwiki/Haskell_in_industry`
- Some uses of Caml in industry `http://cufp.galois.com/2007/slides/XavierLeroy.pdf`