Scheme-kesäkurssi luento 5

Timo Lilja

29. 7. 2009

Sisältö

Rekisterikonekielen simulaattori (SICP 5.2)

- toteuttaa luvussa 5.1 esitellyn rekisterikonekielen
- rajapinta:
  - `(make-machine ⟨regs⟩ ⟨ops⟩ ⟨controller⟩)` palauttaa uuden rekisterikoneobjektin
  - `(set-register-contents! ⟨machine⟩ ⟨reg⟩ ⟨val⟩)` asettaa rekisterin arvon
  - `(get-register-contents ⟨machine⟩ ⟨reg⟩)` palauttaa rekisterin arvon
  - `(start ⟨machine⟩)` käynnistää simulaation
- laskenta hitaampaa kuin normaalissa Scheme-tulkissa

gcd-machine ja käyttöesimerkki

```
(define gcd-machine
  (make-machine
   '(a b t)
   (list (list 'rem remainder) (list '= =))
   '(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)))

(set-register-contents! gcd-machine 'a 206)
done
(set-register-contents! gcd-machine 'b 40)
done
(start gcd-machine)
done
(get-register-contents gcd-machine 'a)
2
```


Rekisterikonesimulaattorin toteutus (SICP 5.2.1)

- simulaattori koostuu joukosta paikallisen tilan olioita, joita ohjataan viestinvälityksellä
  - `make-machine` luo koneeseen rekisterit, primtiivioperaatiot ja "assembloi" kontrollerin
  - `make-register` luo rekisteriobjektin, johon voi asettaa arvon ja lukea sen
  - `make-stack` luo pino-objektin
- `make-new-machine` luo paikallisen tilan olion, joka sisältää kaikille koneille yhteiset rakenteet
  - kontrollirekisterit `flag` ja `pc`
  - uusien rekisterien allokointi- ja käyttörutiinit
  - huolehtii simulaation suorittamisesta kutsumalla käskyjen suoritusproseduureja (*instruction execution procedures*)

Assembler (SICP 5.2.2 - 5.2.3)

- assembler muuntaa kontrollerikäskyt (lisp-lausekkeet) "konekielisiksi käskyiksi" eli suoritusproseduureiksi
- idea sama kuin nelosluvun analysoivassa tulkissa
  - proseduurit eivät ota argumentteja toisin kuin analysoivassa tulkissa, koska ympäristö ja laskennan muu konteksti on rekistereissä
- ennen simulaation suoritusta voidaan tehdä optimointeja
  - symboliset rekisteriviittaukset korvataan viitauksilla rekisteriobjekteihin
  - symboliset osoiteviittaukset (*label*) korvataan osoitinviittauksilla kontrollerisekvenssin vastaavaan kohtaan
- jokaiselle rekisterikonekielen käskylle ja alilausekkeelle (reg, label, op) omat suoritusproseduurit

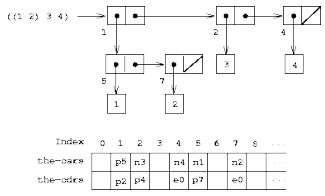
Sisältö

Muisti vektoreina (SICP 5.3.1)

- Kirjan tulkeissa ja rekisterikoneessa oletetaan listarakenteinen muisti primitiivinä
- tietokoneissa muisti on lineaarinen taulukko, jossa mv. osoitteessa olevaa dataa voidaan hakea ja muuttaa
- Schemessä voidaan mallintaa muistia vector-primitiivillä
  - `(make-vector ⟨k⟩ ⟨fill⟩)` luo uuden ⟨k⟩-alkioisen vektorin
  - `(vector-ref ⟨v⟩ ⟨n⟩)` palauttaa alkion ⟨n⟩
  - `(vector-set! ⟨v⟩ ⟨n⟩ ⟨val⟩)` asettaa vektorin ⟨v⟩ kohdan ⟨n⟩ arvoon ⟨val⟩

Lisp-datan esittäminen

- listarakenne koostuu vektoreista `the-cars` ja `the-cdrs`
- tyypitetyt pointterit sisältävät datan tyypin
- objektit ovat `eq?`, jos niiden osoitteet samat



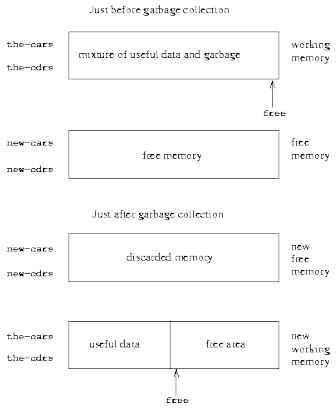
- symbolit tyyppipointteri `read`illä luetusta merkkijonosta
  - sama merkkijono = sama symboli: merkkijonot tallenetaan `obarray`hyn ja symboli on vain viittaus

Roskienkeruu (SICP 5.3.2)

- roskienkeruun avulla voidaan automaattisesti vapauttaa muisti johon ei ole enää viittauksia.
- käyttökelpoinen muisti: saavutettavissa (rekisteri)koneen rekistereistä joko suoraan tai toistuvalla `car`/`cdr`-operaatioiden suorittamisella
- muuten muisti on roskaa (*garbage*) eli se ei voi vaikuttaa laskennan etenemiseen ja voidaan käyttää uudestaan
- roskienkeruuta on monenlaisia riippuen tilanteesta, kirjassa esitellään algoritmi stop-and-copy

Stop-and-copy-roskienkeruu (SICP 5.3.2)

- muisti jaetaan työmuistiin ja vapaaseen muistiin
- `cons` allokoi pareja työmuistista
- kun työmuisti on täynnä, käynnistetään roskienkeruu:
  - etsitään kaikki käyttökelpoiset parit työmuistista (root-lista) ja kopioidaan ne peräkkäisiin paikkoihin vapaasssa muistissa
  - prosessin päätyttyä vaihdetaan työmuisti ja vapaa muisti keskenään
- koska roskaa ei kopioida, on uudessa työmuistissa enemmän tilaa kuin vanhassa
- jaettujen rakenteiden kopioinnissa täytyy olla tarkkana
- rekisterikonekielellä tehty toteutus SICPin luvussa 5.3.2



Roskienkeruualgoritmien hyötyjä ja haittoja

+ estää osoitinviittaukset muistiin joka on jo vapautettu
+ estää kaksinkertaisen vapauttamisen
+ estää muistivuodot sellaiseen muistiosuuteen johon ei ole enää viitettä
− aiheuttaa jonkin verran ylimääräistä kuormitusta
− voi aiheuttaa ennakoimattomia viiveitä
− roskienkeruukielen ja eksplisiittisen muistihallinnan omaavan kielen koodin linkittäminen yhteen voi olla hankalaa

Sisältö

Rekisterikonekielinen Scheme-tulkki (SICP 5.4)

- metasirkulaarisen scheme-tulkin toteutus rekisterikonekielellä
- ympäristöjen käsittely, syntaksin lukeminen, primitiivioperaatiot simulaation ukopuolella kuten nelosluvussa
- rekisterikone, jossa seitsemän rekisteriä:
  - `exp` sisältää evaluoitavan lausekkeen, `env` ympäristön, `val` evaluoinnin tuloksen
  - `continue`-rekisteri toteuttaa rekursion
  - `proc`, `argl` ja `unev` käytetään evaluoimaan yhdistettyjä lausekkeita

Rekisterikonetulkin ydin

- vastaa metasirkulaarisen tulkin `eval`-proseduuria

eval-dispatch

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  ⟨...⟩
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

Proseduuriapplikaatioiden evaluointi

- proseduurikutsua evaluoitaessa evaluoidaan ensiksi operandi ja sen jälkeen argumentit
  - evaluointi tallentaa proseduurin rekisteriin `proc` ja argumentit listana rekisteriin `argl`
- tämän jälkeen proseduuriarvoista operandia sovelletaan (*apply*) operandeihin
- toiminta on analoginen metasirkulaarisen tulkin `eval` ja `apply`-proseduurien kanssa

Sekvenssien evaluointi ja häntärekursio (5.4.2)

- sekvenssit evaluoidaan asettamalla sekvenssin `unev`-rekisteriin, asettamalla `continue`-arvo oikein ja hyppäämällä kohtaan `ev-sequence`
- sekvenssin lauseketta evaluoitaessa:
  - tallennetaan rekisterit
  - asetetaan continue
  - hypätään `eval-dispatch`iin
- viimeistä lauseketta evaluoitaessa ei tallenneta rekistereitä, koska muuten häntärekursio rikkoontuisi:
  - tallentamalla rekisteriarvot jokainen häntäkontekstissa oleva lauseke jäisi kutsupinoon sisäkkäisissä kutsuissa
  - rekursion päätyttyä palataan, mutta koska ne ovat häntäkotekstissa, ei niiden paluuarvoja käytetä

Sisältö

Kääntäjä Schemestä rekisterikonekielelle (SICP 5.5)

- luvun kääntäjä kääntää Schemeä rekisterikonekielelle
- perustuu metasirkulaariseen tulkkiin, mutta suorittamisen sijasta emittoi koodia

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ⟨...⟩
        ((begin? exp)
         (compile-sequence (begin-actions exp)
                           target
                           linkage))
        ((cond? exp) (compile (cond->if exp) ...))
        ((application? exp)
         (compile-application exp target linkage))
        (else
         (error "Unknown expression type -- COMPILE" exp))))
```


Linkkaus ja kohteet (SICP 5.5.1)

- kääntäjällä käänettävän lausekkeen lisäksi argumentit:
  - `target` määrittää rekisterin jossa lausekkeen arvo palautetaan
  - `linkage` määrittää jatketaanko seuraavaan käskyyn (`next`), palataanko proseduurista (`return` vai hypätäänkö johonkin tiettyyn osoitteeseen (label)
- Esimerkiksi kääntämällä lausekkeen 5 `linkage`-arvolla `next` saadaan:
  ```
  (assign val (const 5))
  ```
  kun taas `linkage`-arvolla `return`:
  ```
  (assign val (const 5))
  (goto (reg continue))
  ```

Käskysekvenssit (SICP 5.5.1)

- jokainen koodigeneraattori palauttaa käskysekvenssin rekisterikonekielen käskyjä jotka vastaavat lähdekielen lauseketta
- yhdistetyn lausekkeen koodi käännetään kääntämällä yksittäisten lausekkeiden koodi ja yhdistämällä se proseduurilla:
  ```
  (append-instruction-sequences <seq1> <seq2>)
  ```
  joka tuottaa sekvenssin:
  ⟨*seq1*⟩
  ⟨*seq2*⟩

Pinon käyttö (SICP 5.5.1)

- joissain tilanteissa rekisterien arvot on tallennettava ennen kuin lähdetään suorittamaan seuraavaa käskysekvenssiä
- kääntäjässä on `preserving`-funktio, joka yhdistää kaksi käskysekvenssiä niin, että tarvittavat rekisterit tallennetaan:
  ```
  (preserving (list <reg1> <reg2>) <seq1> <seq2>)
  ```
- preserving tuottaa seuraavat pino-operaatiot riippuen siitä, miten sekvenssit ⟨*seq1*⟩ ja ⟨*seq2*⟩ käyttävät rekistereitä ⟨*reg1*⟩ ja ⟨*reg2*⟩

⟨seq_1⟩ ⟨seq_2⟩	⟨save ⟨reg_1⟩⟩ ⟨seq_1⟩ ⟨restore ⟨reg_1⟩⟩ ⟨seq_2⟩	⟨save ⟨reg_2⟩⟩ ⟨seq_1⟩ ⟨restore ⟨reg_2⟩⟩ ⟨seq_2⟩	⟨save ⟨reg_2⟩⟩ ⟨save ⟨reg_1⟩⟩ ⟨seq_1⟩ ⟨restore ⟨reg_1⟩⟩ ⟨restore ⟨reg_2⟩⟩ ⟨seq_2⟩

Käskysekvenssien esittäminen (SICP 5.5.1)

- jokainen käskysekvenssi sisältää:
  - rekisterit, jotka täytyy alustaa ennen käskysekvenssiä
  - rekisterit, joiden arvoa käskysekvenssi muuttaa
  - varsinaiset käskyt
- em. tietojen avulla on helpompi toteuttaa `preserving`-funktio kuin jos sekvenssistä säilytetään vain varsinaiset käskyt

käskysekvenssitietotyyppi ja esimerkkejä

```
(define (make-instruction-sequence needs modifies stms)
  (list needs modifies stms))
(make-instruction-sequence '(env continue) '(val)
  '((assign val
      (op lookup-variable-value) (const x) (reg env))
    (goto (reg continue))))
(define (empty-instruction-sequence)
   (make-instruction-sequence '() '() '()))
```

Seuraavalla luennolla

Kääntäjän loppuosa:

- lausekkeiden kääntäminen (SICP 5.5.2)
- proseduuriapplikaatioiden kääntäminen (SICP 5.5.3)
- käskysekvenssien yhdistäminen (SICP 5.5.4)
- esimerkki ja optimointeja (SICP 5.5.5-5.5.6)
- käännetyn koodin linkittäminen tulkkiin (SICP 5.5.7)

Mahdollisia aiheita:

- Foreign Function Interfaces (FFI), lisp-koodin linkittäminen C-koodin kansssa
- Common Lisp Object System (CLOS)
- Funktionaalisten kielten käyttö:
  - teollisuudessa: kokemuksia, mielipiteitä ja ideoita
  - miksi funktionaaliset kielet eivät ole mainstreamiä
  - moniydinprosessorit ja funktionaaliset kielet