Funktionaalisten kielten ydinpiirteiden toteutus oliokielten virtuaalikonealustoilla

Laura Leppänen

Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 30. lokakuuta 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Laura Leppänen

Työn nimi — Arbetets titel — Title

Funktionaalisten kielten ydinpiirteiden toteutus oliokielten virtuaalikonealustoilla

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	30. lokakuuta 2016	106 sivua + 25 sivua liitteissä

Tiivistelmä — Referat — Abstract

Tutkielma käsittelee funktionaalisille kielille tyypillisten piirteiden, ensimmäisen luokan funktioarvojen ja häntäkutsujen toteutusta oliokielille suunnatuilla JVM- ja .NET-virtuaalikonealustoilla. Oliokielille suunnattujen virtuaalikonealustojen tarjoamien tavukoodirajapintojen rajoitteet verrattuna matalamman tason assembly-kieliin ovat pitkään aiheuttaneet valtavirran oliokielistä poikkeavien kielten toteuttajille päänvaivaa. Tarkasteltavista alustoista .NET-alustan tavoitteena on alusta asti ollut monenlaisten kielten tukeminen. JVM-alustalla erilaisten kielten toteuttajien tarpeisiin on havahduttu vasta viimeisten vuosien aikana.

Tutkielma tarkastelee, millaisia mahdollisuuksia alustat nykyisellään tarjoavat ensimmäisen luokan funktioarvojen ja häntäkutsujen toteuttajille ja miten alustoilla käytettävät toteutustekniikat poikkeavat perinteisistä konekieltä tuottavista kääntäjistä. Lisäksi esitetään arvio alustojen tarjoaman tuen soveltuvuudesta funktionaalisten kielten toteuttajien käyttöön ja verrataan alustojen tarjoamia tukitoimintoja. Arvioinnin tueksi esitellään oma prototyyppitoteutus Scheme-kielen osajoukolle, Cottontail Scheme, sekä selvitetään, millaisia toteutustekniikoita olemassa olevat funktionaaliset kielet tai funktionaalisia piirteitä tukevat moniparadigmakielet tällä hetkellä käyttävät virtuaalikonealustoilla.

Tutkielmassa tehdyn vertailun perusteella havaittiin, että molemmilla alustoilla on omat vahvuutensa: JVM-alustan sulkeumien tuki on parantunut huomattavasti erityisesti JVM-alustan Java SE 8 -version myötä, mutta .NET-alusta on edelleen ainoa alusta, joka tarjoaa sisäänrakennettua tukea häntäkutsuille. Olemassa olevien kielten vertailussa huomattiin, että harva kielistä hyödyntää ominaisuuksien toteutuksessa alustan tukea, mikä saattaa esimerkiksi JVM-alustan sulkeumatuen tapauksessa johtua yksinomaan toiminnallisuuden tuoreudesta. Kummankaan alustoista ei havaittu tällä hetkellä tarjoavan selvästi toista parempaa tukea esimerkiksi Schemen kaltaisten funktionaalisten kielten toteuttajille.

ACM Computing Classification System (CCS):

Software and its engineering ~ Functional languages
Software and its engineering ~ Procedures, functions and subroutines
Software and its engineering ~ Source code generation
Software and its engineering ~ Runtime environments
Software and its engineering ~ Virtual machines

Avainsanat — Nyckelord — Keywords
ohjelmointikielten kääntäjät, Scheme, funktioarvot, häntäkutsut, JVM, .NET

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Funktionaalisilla ohjelmointikielillä on pitkät perinteet akateemisen tietojenkäsittelytieteen maailmassa. Tyypillisesti ensimmäisenä funktionaalisena ohjelmointikielenä pidetään John McCarthyn suunnittelemaa Lisp-kieltä [McC60], jonka ensimmäinen toteutus julkaistiin vuonna 1962 [McC78]. Funktionaalisen paradigman akateemisesta suosiosta huolimatta paradigman suosio kaupallisessa ohjelmistokehityksessä oli pitkään vähäistä.

Noin viimeisen vuosikymmenen aikana funktionaaliset piirteet ovat kasvattaneet suosiotaan huomattavasti myös valtavirran ohjelmoinnissa. Erityisesti sulkeumia, anonyymejä funktioita ja korkeamman asteen funktioita pidetään jo lähes välttämättöminä ominaisuuksina moderneissa moniparadigmakielissä. Funktioiden käyttöä ensimmäisen luokan arvoina tukevat erilaisissa muodoissa esimerkiksi moniparadigmakielet Ruby, C# ja Python sekä Java- ja C++-kielten uusimmat versiot Javan versiosta 8 ja C++:n versiosta 11 alkaen.

Monille edellä mainituista kielistä on olemassa toteutus ainakin toisella suosituimmista oliokielten virtuaalikonealustoista, Java-virtuaalikoneella eli JVM:llä tai .NET-alustan CLR-virtuaalikoneella. Java ja C# ovat kumpikin omien alustojensa lippulaivakieliä, Java JVM-alustalla ja C# .NET-alustalla. Rubyn ja Pythonin toteutuksia ovat JRuby ja Jython JVM-alustalla sekä IronRuby ja IronPython .NET-alustalla. Toteutusalustan lisäksi toinen edellä mainittuja kieliä yhdistävä piirre on se, että ne painottavat olio-ohjelmointia funktionaalisen tyylin kustannuksella.

Samalla kun funktionaalisten piirteiden suosio valtavirran ohjelmointikielissä on kasvanut, oliokieliä varten suunnitellut virtuaalikonealustat ovat herättäneet kiinnostusta toteutusalustoina myös uusille funktionaalisille kielille tai vahvemmin funktionaalista tyyliä painottaville moniparadigmakielille. Esimerkkejä tällaisista kielistä ovat Scala ja Kotlin JVM-alustalla, F# .NET-alustalla sekä Clojure, jolle on olemassa toteutus molemmilla alustoilla[1].

Virtuaalikoneet alustoina houkuttelevat uusien kielten toteuttajia, koska ne helpottavat kielten siirtämistä alustalta toiselle ja mahdollistavat kielten yhteiskäytön. Esimerkiksi Scala- tai Clojure-ohjelmoija voi helposti käyttää hyödykseen olemassa olevia Java-kirjastoja, ja toisaalta C#-ohjelmoija voi kirjoittaa osan ohjelmiston toimintalogiikasta F#-kielellä ja hyödyntää siten funktionaalisen kielen tarjoamia abstraktiomahdollisuuksia. Lisäksi virtuaalikone tarjoaa esimerkiksi roskienkerääjän sekä muita kielen toteuttajan työtä helpottavia tukitoimintoja.

Olioparadigmasta poikkeavien kielten kannalta näissä alustoissa on kuitenkin perinteisesti ollut ongelmana se, että niiden tarjoamat tukitoiminnot on räätälöity pää-

[1] Myös Scalalla oli aiemmin .NET-takaosa, mutta sen tuki ja jatkokehitys lopetettiin virallisesti vuonna 2012.

sääntöisesti oliokielille. Tästä syystä funktionaalisten kielten piirteet voivat aiheuttaa kielten toteuttajille päänvaivaa. Oliokielten virtuaalikonealustojen rajoitteiden vuoksi on jopa suunniteltu erityisiä vaihtoehtoisia virtuaalikonealustojen toteutustekniikoita, jotka mahdollistaisivat monien erilaisten kielten tukemisen [WWW+13]. Toistaiseksi varteenotettavia vaihtoehtoja suurimmille oliokielten virtuaalikonealustoille ei kuitenkaan ole nähty tuotantokäytössä.

Tutkielman tavoitteena on selvittää, miten funktionaalisten kielten ydinpiirteet voidaan toteuttaa JVM- ja .NET-alustoilla ja miten näillä alustoilla käytetyt toteutustekniikat poikkeavat toteutuksista perinteisillä alustoilla. Tutkielma keskittyy tarkastelemaan perinteisesti funktionaalisiin kieliin yhdistettyjen piirteiden, funktioarvojen sekä häntäkutsujen toteutukseen liittyviä haasteita. Häntärekursio eli rekursiivisten häntäkutsujen käyttö on useimmissa funktionaalisissa kielissä silmukankaltaisten rakenteiden toteutukseen käytettävä tekniikka. Muita kuin suoraan rekursiivisia häntäkutsuja voidaan käyttää esimerkiksi tehokkaan keskinäisen rekursion aikaansaamiseksi. Käsitteet määritellään tarkemmin luvussa 2.

Nykyaikaisiin funktionaalisiin kieliin liitetään myös monia muita piirteitä kuten laiskuus ja hahmonsovitus. Kaikkien funktionaalisiin kieliin yhdistettyjen piirteiden toteuttamiseen paneutuminen tämän laajuisessa tutkielmassa olisi käytännössä mahdotonta. Näitä piirteitä ja niiden toteuttamiseen liittyvää tutkimusta sivutaan lyhyesti käsitteiden määrittelyn yhteydessä.

Toteutustekniikoiden esittelyn lisäksi tutkielma vertailee, onko tarkasteltavissa virtuaalikonealustoissa funktionaalisia piirteitä tukevan kielen toteuttajan kannalta merkityksellisiä eroja. Alustojen suoritusmallit ovat pääpiirteittäin hyvin samankaltaiset, joten toteuttajan kannalta kiinnostavia eroja ovat erityisesti juuri funktionaalisten kielten piirteiden toteutusta tukevat toiminnallisuudet ja niiden käyttökelpoisuus käytännön toteutuksissa.

Tutkielmassa esitellään oma funktionaalisen kielen kääntäjän prototyyppitoteutus molemmilla alustoilla. Oman toteutuksen tavoitteena on saada ensi käden tietoa alustojen tarjoamien ominaisuuksien käytettävyydestä käytännön toteutuksissa. Toteutettavaksi on valittu osajoukko Lisp-perheeseen kuuluvan Scheme-kielen R7RS-standardista [Sus13]. Scheme soveltuu hyvin funktionaalisten piirteiden tarkasteluun, koska kieli on yksinkertainen ja mahdollistaa tarkastelun rajaamisen funktionaalisten kielten toteutuksen kannalta kiinnostavimpiin piirteisiin.

Scheme on toteutettavan kääntäjän lähtökielenä kiinnostava myös siitä syystä, että useita Scheme-kääntäjätoteutuksia on jo olemassa molemmilla tarkasteltavilla alustoilla: GNU-projektin Kawa JVM-alustalla [Bot98], .NET-alustan IronScheme [Pri12] sekä Bigloo, joka tuottaa tavukoodia molemmille alustoille sekä lisäksi C-kieltä [SS02, BSS04]. Tämä antaa mahdollisuuden vertailla olemassa olevien kielten tavukooditason toteutuksia

paitsi tunnetuimpien JVM- ja .NET-alustojen funktionaalisten kielten ja moniparadigmakielten toteutusten osalta, myös Scheme-toteutusten osalta. Vaikka vertailtavien kielten edustamien paradigmojen kirjo on laaja — mukana on oliokieliä ja funktionaalisia kieliä sekä staattisesti ja dynaamisesti tyypitettyjä kieliä — havaitaan esimerkiksi funktioarvojen toteutuksissa olevan paljon samankaltaisuuksia.

Tutkielman luku 2 esittelee keskeisimmät funktionaalisten kielten piirteet ja niihin liittyvät käsitteet sekä Scheme-kielen perusteet. Luvussa 3 esitellään tarkasteltavien kohdealustojen suoritusmalli. Molemmat alustat määrittelevät omanlaisensa abstraktin koneen, joka suorittaa tavukoodiohjelmia. Luku 3 esittelee myös molempien alustojen tavukoodien perusteet. Luku 4 esittelee funktionaalisten kielten piirteiden toteutustekniikoita pohjautuen pääosin yleiseen kääntäjiä ja funktionaalisten kielten kääntäjiä käsittelevään kirjallisuuteen sekä osin artikkeleihin, jotka käsittelevät juuri toteutuksia virtuaalikonealustoilla. Luvun lopussa esitetään tavukoodin tarkasteluun ja yksittäisiä kääntäjiä koskeviin artikkeleihin perustuva vertailu muutamien tunnettujen funktionaalisten kielten ja moniparadigmakielten valitsemista toteutustekniikoista virtuaalikonealustoilla.

Luvussa 5 esitellään tutkielmaa varten tehdyn esimerkkitoteutuksen, Cottontail Schemen, käyttämät toteutustekniikat molemmilla kohdealustoilla. Luvussa perustellaan tehdyt toteutusvalinnat ja esitellään toteutuksessa kohdattuja ongelmakohtia. Luvussa 6 esitetään analyysi erilaisten toteutustekniikoiden valintaperusteista eri tyyppisissä kielissä ja vertaillaan kohdealustoja niiden tarjoaman funktionaalisten piirteiden toteutustuen osalta.

2 Funktionaalisten kielten piirteet

Funktionaalisten ja imperatiivisten kielten välillä ei aina ole selkeää rajaa. Useat funktionaalisia piirteitä tukevat moniparadigmakielet kuten Python [Mer01] ja C# [Stu11] mahdollistavat ohjelmien kirjoittamisen pääosin funktionaalisella tyylillä. Monet funktionaaliset kielet puolestaan tukevat myös imperatiivisista kielistä tuttuja piirteitä kuten muuttujiin sijoittamista ja silmukoita. Kaikki funktionaalisiksi kutsutut kielet eivät välttämättä jaa keskenään mitään yksikäsitteisesti määriteltyä joukkoa piirteitä [App98, s. 309–310]. Tässä luvussa määritellään, mitä työn yhteydessä tarkoitetaan funktionaalisten kielten piirteillä.

Appel [App98, s. 309–310] määrittelee funktionaalisten ohjelmointikielten keskeisimmiksi piirteiksi matemaattisten yhtälöiden kaltaisen päättelyn (engl. *equational reasoning*) ja korkeamman asteen funktiot. Korkeamman asteen funktioihin liittyy kiinteästi funktioiden käsittely ensimmäisen luokan arvoina [Sco09, s. 507]. Funktionaalisiin kieliin liitetään lisäksi usein häntäkutsujen ja erityisesti häntärekursion käyttö ongelmanratkaisussa, minkä vuoksi funktionaalisten kielten kääntäjätoteutuksilta tyypillisesti vaaditaan näiden rakenteiden optimointia [Sco09, s. 505, 508]. Tässä tutkielmassa tarkastelu keskittyy erityisesti ensimmäisen luokan funktioarvojen ja häntäkutsujen optimoinnin toteutukseen.

Tämä luku määrittelee keskeisimmät aihepiiriin liittyvät käsitteet. Lisäksi esitellään tutkielmassa tarkastelun kohteena oleva funktionaalinen kieli, Scheme. Koska Schemekieltä käytetään tutkielmassa myös pääasiallisena koodiesimerkkien kielenä, luku alkaa kielen esittelyllä. Viimeisessä aliluvussa luodaan lyhyt katsaus muutamiin usein erityisesti uudempiin funktionaalisiin kieliin liitettyihin piirteisiin, jotka rajautuvat tutkielman tarkastelun ulkopuolelle.

2.1 Scheme-kielen esittely

Scheme on alun perin 1970-luvulla luotu ja myöhemmin IEEE-standardoitu Lisp-perheen kieli [SJ75, IEE08]. Sen ensisijainen määritelmä on *Revised^n Report on the Algorithmic Language Scheme* -niminen aika ajoin uudistettava standardidokumentti, lyhyesti *RnRS*. Standardin tuorein versio on nimeltään R7RS vuodelta 2013 [Sus13].

Scheme on dynaamisesti tyypitetty kieli, jolla on sekä funktionaalisten että imperatiivisten kielten piirteitä. Hudak [Hud89] kuvailee Schemeä "imperatiiviseksi kieleksi, jolla on käyttökelpoinen puhtaasti funktionaalinen osajoukko". Toisin kuin esimerkiksi Haskell, Scheme ei aseta rajoitteita sivuvaikutuksille. Scheme sallii myös minkä tahansa muuttujan arvon uudelleenasetuksen. Idiomaattinen Scheme kuitenkin painottaa funktionaalista tyyliä, jossa tarpeettomia sivuvaikutuksia pyritään välttämään.

Kuten muissakin Lisp-perheen kielissä, Schemessä listat ovat keskeisessä asemassa. Sen lisäksi että listat ovat mahdollisesti Schemen käytetyimpiä tietorakenteita vektorien ohella,

myös Scheme-ohjelman rakenne voidaan esittää listana. Tätä mahdollisuutta käsitellä koodia datana kutsutaan *homoikonisuudeksi* [Sco09, s. 517], ja se tekee esimerkiksi Scheme-tulkin toteutuksesta kielellä itsellään suoraviivaista.

Kuten useimmilla dynaamisesti tyypitetyillä kielillä, Schemen tavallisin toteutusmuoto on tulkki, joka tarjoaa interaktiivisen REPL-käyttöliittymän (*read-eval-print loop*). Schemelle on kuitenkin olemassa myös useita ylläpidettyjä kääntäjätoteutuksia kuten Chez Scheme, Chicken Scheme, Gambit sekä tässäkin tutkielmassa tarkasteltavat Kawa ja Bigloo[2].

Scheme-ohjelman rakenne ja syntaksi

Scheme-ohjelma koostuu sulutetuista eli listamuotoisista lausekkeista ja *atomeista*. Näitä lausekkeita ja atomeita yhdessä kutsutaan *S-lausekkeiksi* (*S-expression*), joka on lyhenne sanoista *symbolinen lauseke* [McC60]. Atomilla tai *atomisella symbolilla* tarkoitetaan Scheme-kielessä mitä tahansa arvoa, joka ei ole lista [FF96, McC60]. Esimerkkejä Schemen atomeista ovat numeeriset arvot (esim. `42`, `3.14159`), kirjainmerkit (esim. `\#a`, `\#X`, `\#?`), merkkijonot (esim. `"Hello, World!"`), totuusarvot (esim. `#t`, `#f`) ja symbolit eli esimerkiksi funktioiden nimet (`fib`, `+`).

Listamuodossa esitetyt lausekkeet tulkitaan proseduurikutsuiksi tai erikoisrakenteiden kuten sisäänrakennettujen toimintojen tai makrojen käytöksi. Esimerkiksi lauseke `(fib 5)` voisi olla `fib`-nimisen proseduurin kutsu arvolla `5`.

Proseduurikutsun epätavallisen näköinen muoto kertoo toisesta keskeisestä piirteestä Schemen syntaksissa: proseduurikutsut ja erikoisrakenteiden kuten `if` käyttö esitetään prefiksimuodossa. Myös matemaattisissa operaatioissa kuten `+` ja `-` käytetään useimmista kielistä poiketen prefiksiesitystä, eli esimerkiksi matemaattinen lauseke $1 + 2$ esitetään Schemessä muodossa `(+ 1 2)`. Tämä yksinkertaistaa kielen syntaksia merkittävästi, koska useimmista muista kielistä poiketen ohjelmoijan tai kielen toteuttajan ei tarvitse välittää esimerkiksi operaattorien sidontajärjestyksestä, sillä sidontajärjestys ilmenee aina suoraan koodin rakenteesta.

Schemessä kaikki käyttäjän määrittelemät proseduurit luodaan lambda-lausekkeilla, eli useista moniparadigmakielistä poiketen ohjelmakoodin muoto ei tee eroa funktioarvojen ja muiden proseduurien välille. Schemen lambda-lausekkeet ottavat ensimmäisenä parametrinaan proseduurin parametrilistan. Loput parametrit ovat lausekkeita, jotka muodostavat funktion rungon. Lambda-lauseke `(lambda (x y) (+ x y 1))` määrittelee kaksiparametrisen nimettömän funktion, joka palauttaa parametriensa summan lisättynä

[2]Vaikka REPL-käyttöliittymät yhdistetään usein juuri tulkkeihin ja tulkattaviin kieliin, ne eivät toki ole sidottuja varsinaiseen tulkkitoteutukseen. Myös kaikki mainitut kääntäjätoteutukset tarjoavat REPL-käyttöliittymän, joka ainakin osassa tapauksista on toteutettu kääntäjän päälle siten, että käyttäjän syöttämää koodia käännetään "lennosta" ennen suoritusta (esim. [Bot16]).

yhdellä. Lambda-lausekkeessa voi olla sulutetun parametrilistan sijaan myös pelkkä muuttujanimi — tällöin muodostetaan funktio, joka ottaa vaihtelevan määrän argumentteja ja muodostaa niistä listan.

Uusien nimien määrittelyä varten Scheme tarjoaa `define`-rakenteen, jota voi käyttää sekä globaalien että paikallisten muuttujien määrittelyyn. Koska monesta muusta Lisp-murteesta poiketen Schemessä ei ole erillisiä nimiavaruuksia proseduureille ja muille arvoille, `define`-rakenteella voi sitoa nimiin sekä proseduureja että muita ensimmäisen luokan arvoja. Schemessä siis esimerkiksi `(define PI 3.14159)` sitoo arvon `3.14159` nimeen `PI`. Paikallisia muuttujia voi Schemessä sitoa myös `let`-, `let*`-, `letrec`- ja `letrec*`-rakenteilla, mutta yksinkertaisuuden vuoksi tässä tutkielmassa käytetään pääosin `define`-rakennetta.

Listarakenteet

Kuten edellä mainittiin, Schemessä lauseke muotoa `(fib 5)` on listarakenne, jonka ensimmäinen alkio on symboli `fib`. Scheme tulkitsee tämän proseduurikutsuksi, jolloin lauseke evaluoituu `fib`-proseduurikutsun argumentille `5` palauttamaksi arvoksi. Varsinainen listatietorakenne on Schemessä mahdollista luoda useammalla tavalla: proseduureilla `cons` ja `list` tai käyttämällä Schemen tarjoamaa `quote`-syntaksia, joka mahdollistaa literaaliarvojen määrittelyn.

Puhuttaessa listoista Lisp-murteiden yhteydessä tarkoitetaan tyypillisesti yhteen suuntaan linkitettyä listaa. Muiden Lisp-murteiden tavoin Scheme esittää linkitetyn listan pareina, jotka sisältävät yhden listan alkioista ja lisäksi viitteen listan loppuosaan. Listan viimeisenä alkiona tulee olla pari, jonka jälkimmäinen elementti on tyhjä lista eli Scheme-termistössä `null`.

Scheme ei tee eroa listaa esittävien parien ja muiden parien välille: kaikki parit luodaan `cons`-nimisellä proseduurilla[3]. Proseduurin tuottamia pareja nimitetään toisinaan proseduurin nimen mukaan myös *cons-soluiksi* (engl. *cons cell*).

Tyhjää listaa Scheme-koodissa esitetään `quote`-literaaliarvolla `'()` tai `(quote ())`. Proseduuri `null?` tarkistaa, onko sille parametrina annettu arvo tyhjä lista. `Quote`-syntaksilla voidaan luoda myös kokonaan literaaleista koostuvia listoja: esimerkiksi `'(1 2 3)` luo kolmialkioisen listan. `Quote`-syntaksilla on Schemessä myös muita käyttötarkoituksia, mutta tämän tutkielman yhteydessä niitä ei tarvita.

Listan `(1 2 3)` voi luoda yhtäpitävästi sekä lausekkeella `(list 1 2 3)` että lausekkeella `(cons 1 (cons 2 (cons 3 '())))`. Parin ja listan arvoja voidaan aksessoida proseduureilla `car` ja `cdr`: lauseke `(car p)` palauttaa parin `p` ensimmäisen arvon ja `(cdr p)` parin `p` jälkimmäisen arvon, eli listan tapauksessa listan loppuosan. Schemen listojen

[3]Proseduurin `cons` nimi on lyhenne sanasta *construct*.

toiminnasta ja listankäsittelyproseduurien nimien taustasta on lisätietoja liitteessä A.1.

Esimerkkejä Scheme-koodista

Kuvassa 2.1 on esimerkki kokonaisluvun kertoman laskennasta Schemellä. Kuvassa on vertailun vuoksi lisäksi esimerkki vastaavasta koodista Javalla kirjoitettuna. Esimerkkikoodi määrittelee `fact`-nimisen muuttujan, johon sidotaan lambda-lausekkeella luotava funktioarvo. Tämä funktio ottaa parametrinaan yhden kokonaislukuarvon. Funktion rungossa käytetty `if`-erikoisrakenne ottaa parametreinaan kolme S-lauseketta: evaluoitavan ehdon ja kaksi vaihtoehtoista haaraa. Jos ehto on tosi, Scheme evaluoi ensimmäisen haaran ja palauttaa sen paluuarvon. Jos ehto taas on epätosi, evaluoidaan ja palautetaan toisen haaran paluuarvo.

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n
           (fact (- n 1))))))
```

```
public static int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}
```

Kuva 2.1: Kokonaisluvun kertoman laskenta Schemellä (vasemmalla) ja Javalla (oikealla)

Lopussa funktio kutsuu itseään rekursiivisesti. Kuten on tyypillistä funktionaalisille kielille, Schemessä käytetään imperatiivisten silmukoiden sijasta rekursiota. Erityisesti on tavallista pyrkiä käyttämään häntärekursiivisia kutsuja, jotka on mahdollista kääntää tehokkaammaksi koodiksi. Häntärekursiivinen kutsu on rekursiivinen kutsu, joka suoritetaan funktion viimeisenä operaationa. Sekä häntärekursiiviset kutsut että ei-rekursiiviset häntäkutsut esitellään tarkemmin luvussa 2.3 sivulla 12. Tämän esimerkin koodi on kirjoitettu käyttäen tavanomaista rekursiota ilman häntäkutsuja, koska esimerkki on tällöin hiukan helppolukuisempi.

Schemen standardi määrittelee myös kontrollirakenteita, joiden käyttö muistuttaa imperatiivisten kielten silmukkarakenteita. Usein nämä rakenteet on kuitenkin toteutettu makroina, jotka piilottavat taakseen häntärekursiivisen proseduurin [Sus13, luvut 4.2.4 ja 7.3]. Schemen makrojärjestelmää kuvaillaan lyhyesti liitteessä A.2. Tyypillisesti Schemen silmukkarakenteita käytetään sellaisissa yhteyksissä, joissa hyödynnetään myös muita imperatiivisia ominaisuuksia kuten muuttujiin sijoitusta, syötteen lukua tai tulostusta [Sco09, s. 516].

Lisäksi koodiesimerkissä huomionarvoista on se, että kuten joissakin ALGOL-perheen kielissä mutta useimmista muista imperatiivisista kielistä kuten Javasta poiketen, Schemessä `if` on lauseke. Lisäksi arvon palauttaminen funktiosta Schemessä ei vaadi erillistä

`return`-lausetta, vaan funktio palauttaa oletusarvoisesti viimeisenä evaluoidun lausekkeen arvon.

Scheme määrittelee `define`-rakenteesta myös proseduureille tarkoitetun muodon, jossa eksplisiittisen lambda-lausekkeen voi jättää pois. Esimerkin proseduuri voitaisiin siis määritellä myös muodossa `(define (fact n)` *<body>*`)`. Jälleen käytämme kuitenkin tässä tutkielmassa syntaksin yksinkertaisena pitämisen vuoksi eksplisiittisiä lambda-lausekkeita.

Esimerkissä käytetyn `if`-rakenteen lisäksi muita keskeisiä kontrollirakenteita ovat `and`, `or` ja `begin`. Rakenteet `and` ja `or` käyttäytyvät periaatteeltaan melko samalla tavalla kuin esimerkiksi C:n sukuisissa kielissä. Poikkeamana esimerkiksi juuri C:n vastaavista operaatioista Schemen `and` ja `or` eivät kuitenkaan palauta totuusarvoa, vaan viimeisen evaluoidun lausekkeen arvon. Esimerkiksi lauseke `(or #f '(1 2 3) #t)` palauttaa listan `(1 2 3)`. Schemessä kaikki arvot paitsi `#f` tulkitaan totuusarvovertailussa todeksi.

`begin`-rakennetta käytetään tyypillisesti sivuvaikutusten ketjutukseen esimerkiksi `if`-lausekkeen haarana kuten kuvan 2.2 esimerkissä. Esimerkissä tulostetaan parametrina annetun listan alkiot välilyönneillä erotettuna ja päätetään tulostus rivinvaihtoon. `begin`-rakenteen avulla muodostetaan lohko, joka mahdollistaa tulostuslausekkeiden suorittamisen ennen rekursiivista kutsua `print-list`-proseduuriin. Käytännössä `begin`-rakenteen voi ajatella muistuttavan kaarisuljeparin muodostaman lohkon käyttöä esimerkiksi Javassa.

```
(define print-list
  (lambda (l)
    (if (null? l)
        (newline)
        (begin
          (display (car l))
          (display " ")
          (print-list (cdr l))))))

(print-list '(1 2 3 4 5)) ; tulostaa "1 2 3 4 5 \n"
```

Kuva 2.2: Sivuvaikutusten ketjutus `begin`-rakenteella.

2.2 Funktiot ensimmäisen luokan arvoina

Funktioiden käsittely ensimmäisen luokan arvoina ja korkeamman asteen funktiot ovat yhteisiä piirteitä useimmille funktionaalisille kielille [Hud89] [Sco09, s. 507–508]. Ensimmäisen luokan arvolla tarkoitetaan ohjelmointikielissä arvoa, jota voidaan käyttää funktion tai aliohjelman paluuarvona tai parametrina sekä sijoittaa muuttujaan [Str00].

Korkeamman asteen funktiolla tarkoitetaan funktiota, joka ottaa parametrinaan toisen funktion [App98, s. 309] tai palauttaa toisen funktion paluuarvonaan [App98, s. 125].

Usein ensimmäisen luokan arvoilta vaaditaan lisäksi mahdollisuutta luoda arvoja suoritusaikaisesti [Sco09, s. 154–155, 508]. Tyypillisesti funktionaaliset kielet ja nykyisin myös monet moniparadigmakielet sallivat anonyymien funktioiden suoritusaikaisen luonnin *lambda-lausekkeiksi* kutsutulla syntaksilla. Tällöin uusia funktioita on mahdollista määritellä toisten funktioiden sisällä.

Useimmissa kielissä esimerkiksi numeerisia tyyppejä edustavat arvot ovat ensimmäisen luokan arvoja, mutta tyypillisesti imperatiivisissa kielissä ja monissa oliokielissä funktioille sallitut operaatiot ovat rajatumpia. Esimerkiksi C:ssä funktioita on mahdollista käsitellä arvoina funktio-osoittimien avulla, mutta kieli ei salli sisäkkäisiä funktiomäärittelyjä, eikä funktioita ole mahdollista luoda suoritusaikana. Scott kuvaileekin C:n funktioita "ensimmäisen luokan arvoiksi tietyn rajoituksin" [Sco09, s. 155].

Sisäkkäiset funktiot voivat viitata myös funktion rungon ulkopuolella esiteltyihin, mutta funktion määrittelypaikalla näkyviin muuttujiin. Yleistä on esimerkiksi viitata ympäröivän funktion rungossa määriteltyihin muuttujiin. Yhdessä mutatoitavien muuttujien kanssa tämä mahdollistaa tilan kuljettamisen funktioarvojen mukana. Funktion määrittelypaikalla näkyvissä olevien muuttujien joukkoa kutsutaan määrittelypaikan *viittausympäristöksi* (engl. *referencing environment*) [Sco09, s. 122]. Määrittelypaikalla näkyvien muuttujien joukon määräytyminen riippuu siitä, onko kielessä *leksikaalinen* vai *dynaaminen* näkyvyys.

Leksikaalinen näkyvyys ja lohkorakenne

Kuten useimmissa laajalti käytetyissä nykyaikaisissa kielissä, myös useimmissa nykyaikaisissa funktionaalisissa kielissä on leksikaalinen näkyvyys, jota kutsutaan myös *staattiseksi* näkyvyydeksi. Leksikaalisella näkyvyydellä tarkoitetaan sitä, että jo käännösaikana on mahdollista selvittää, mihin muuttujan määritelmään tietyssä kohdassa esiintyvä muuttujanimi viittaa, sillä koodin rakenne määrittää muuttujien näkyvyysalueet [Sco09, s. 123]. Tätä muuttujanimen tai muun symbolin yhdistämistä sen määritelmään kutsutaan *sidonnaksi* (engl. *binding*) [Sco09, s. 112].

Kielet, joissa on leksikaalinen näkyvyys, ovat tyypillisesti *lohkorakenteisia*, eli niissä on tietty syntaksi tai operaattori, joka muodostaa lohkon eli leksikaalisen näkyvyysalueen [Sco09, s. 122–123]. C-kielessä ja Javassa tämä ominaisuus on kaarisulkeilla, Pythonissa sisennystaso määrittää näkyvyyslohkon ja joissakin kielissä kuten Rubyssa ja ALGOL-perheen kielissä käytetään avainsanapareja kuten `begin` ja `end`. Tyypillisesti esimerkiksi moduulit ja luokat määrittelevät myös oman näkyvyysalueensa [Sco09, s. 122]. Schemessä tietyt *sidontarakenteet* (engl. *binding constructs*) kuten `lambda` ja `let` muodostavat lohkon

[Sus13, luku 3.1].

Historiallisesti monissa funktionaalisissa kielissä ja erityisesti Lisp-perheen kielissä on ollut dynaaminen näkyvyys [Sco09, s. 505]. Tällöin ohjelman suoritusjärjestys vaikuttaa siihen, mitkä muuttujat ovat näkyvissä tietyssä ohjelman suoritusvaiheessa [Sco09, s. 122]. Muuttujanimien viittausten sidonta on siis tehtävä suoritusaikana. Nykyaikaisissa kielissä on kuitenkin harvinaista, että näkyvyys olisi oletuksena dynaaminen, vaikka osa kielistä mahdollistaakin dynaamisen näkyvyyden käytön aktivoinnin tiettyjä rakenteita käyttämällä. Yksi esimerkki kokonaan dynaamista näkyvyyttä käyttävästä kielestä on Emacs-tekstinkäsittelyohjelmassa käytettävä Emacs Lisp [Sta81].

Tässä tutkielmassa tarkasteltava Scheme on useimpien muiden nykyaikaisten kielten tavoin lohkorakenteinen ja siinä on leksikaalinen näkyvyys [Sus13, s. 5]. Myös Schemelle on kuitenkin olemassa standardin ulkopuolinen `fluid-let`-niminen laajennos, joka mahdollistaa dynaamisen näkyvyyden käytön yksittäisen muuttujan kanssa. Kuvan 2.3 esimerkki esittelee dynaamisen ja leksikaalisen näkyvyyden eroa tätä rakennetta käyttäen.

```
(define n 1)

(define add-n
  (lambda (x)
    (+ n x)))

(define add-dynamic
  (lambda (x)
    (fluid-let ((n 2))
      (add-n x))))

(define add-static
  (lambda (x)
    (let ((n 2))
      (add-n x))))

(add-dynamic 1)      ⟹ 3
(add-static 1)       ⟹ 2
```

Kuva 2.3: Scheme-kielinen ohjelma, joka esittelee dynaamisen ja leksikaalisen näkyvyyden välistä eroa.

Esimerkissä funktio `add-dynamic` käyttää `fluid-let`-rakennetta, jonka määrittelemien muuttujien näkyvyys on dynaaminen: rakenteen muodostaman lohkon sisällä oleva `add-n`-kutsu näkee muuttujan `n` arvon olevan `2`. Vertailun vuoksi funktiossa `add-static` käytetty tavallinen `let`-rakenne luo uuden `n`-muuttujan, joka näkyy vain `let`-rakenteen muodostamassa leksikaalisessa näkyvyysalueessa. Tällöin lohkon sisällä tehty kutsu funktioon `add-n` näkee vain ylätasolla tehdyn `n`-muuttujan sidonnan. Esimerkkiohjelma on

mahdollista kääntää esimerkiksi Kawalla, joka tukee `fluid-let`-laajennosta.

Sulkeumat

Kun sisäkkäinen funktio viittaa viittausympäristössään näkyviin muuttujiin, sen sanotaan luovan viittausympäristönsä ympärille *sulkeuman* (engl. *closure*) [Sco09, s. 153]. Sulkeuma *kaappaa* viittaukset funktion määrittelypaikan viittausympäristössä oleviin muuttujiin. Kaikissa tässä tutkielmassa tarkasteltavissa kielissä on leksikaalinen näkyvyys, joten viittausympäristö määrittyy suoraviivaisesti lohkorakenteen mukaan.

Sulkeuman luoneen funktion runko voi viitata ja monissa kielissä jopa muuttaa sen määritelmän viittausympäristössä esiteltyjen muuttujien arvoja vielä määrittelyympäristöstä poistumisen jälkeenkin, eli esimerkiksi sen jälkeen, kun funktioarvo on palautettu ympäröivän funktion paluuarvona. Tällöin sulkeumatoteutuksen täytyy varmistaa, että kaapattujen muuttujien elinaika on sidottu sulkeuman elinaikaan. Muuten sulkeuman olisi mahdollista viitata muuttujiin, joita ei enää ole olemassa sulkeuman kutsuhetkellä.

Kuvassa 2.4 on esimerkki sulkeumasta, joka kaappaa viittauksen ympäröivän funktion parametrilistassa määriteltyyn muuttujaan `x`. Esimerkissä aiemmin luotuun sulkeumaan tehtävä kutsu `(add10 4)` edellyttää, että muuttuja `x` on edelleen olemassa, ja sen arvo voidaan hakea kutsun aikana.

```
(define add
  (lambda (x)    ; lohko 1: funktion add leksikaalinen näkyvyysalue
    (lambda (y)  ; lohko 2: palautettava sulkeuma
      (+ x y))))

(define add10 (add 10))

(add10 4)     ⟹ 14
```

Kuva 2.4: Scheme-kielinen esimerkki sulkeumasta, joka viittaa viittausympäristössään määriteltyyn muuttujaan `x`.

Sulkeuman kaappaamien muuttujien joukkoa kutsutaan sulkeuman *ympäristöksi* (engl. *environment*) [App98, s. 312], ja näitä kaapattuja muuttujia kutsutaan funktion *vapaiksi muuttujiksi* (engl. *free variable*) [App98, s. 326]. Sulkeuman määrittelypaikkaa ympäröivän funktion näkökulmasta näitä sisempien funktioiden kaappaamia muuttujia taas kutsutaan *pakeneviksi muuttujiksi* (engl. *escaping variable*) [App98, s. 313]. Esimerkissä 2.4 muuttuja `x` on vapaa muuttuja sisemmän, anonyymin funktion rungossa ja pakeneva muuttuja `add`-funktion rungossa.

2.3 Häntäkutsut ja rekursio

Useimmille funktionaalisille kielille on korkeamman asteen funktioiden lisäksi yhteistä se, että tyypillinen ongelmanratkaisutyyli esimerkiksi erilaisten tietorakenteiden läpikäyntiä vaativissa ongelmissa on rekursio [Sco09, s. 505, 508]. Imperatiivisissa kielissä suosittujen iteraatiorakenteiden kuten `for`-silmukoiden ja ympäröivän tilan muuttamisen sijaan funktionaalisissa kielissä käytetään rekursiivisia funktioita, jotka kuljettavat tilaa eksplisiittisesti mukanaan kutsusta toiseen [Hud89].

Tehokkuuden vuoksi useissa funktionaalisissa kielissä on tapana, mikäli mahdollista, kirjoittaa rekursiiviset funktiot häntärekursiiviseen muotoon. Funktio on häntärekursiivinen, jos se kutsuu itseään vain *häntäkutsuna*. Häntäkutsuksi sanotaan mitä tahansa funktiokutsua, joka esiintyy funktion rungossa *häntäpositiossa*. Häntäpositiossa esiintyvän funktiokutsun tunnistaa siitä, että se on viimeinen ympäröivän funktion suorittama operaatio ennen paluuta funktiokutsusta [App98, s. 329].

Häntärekursio voidaan monissa tavanomaisissa tapauksissa kääntää silmukan kaltaiseksi koodiksi [Sco09, s. 272–273]. Tätä käännösaikana tehtävää koodin tehostamista kutsutaan *häntäkutsujen optimoinniksi* (engl. *tail call optimization*) tai *häntäkutsujen poistamiseksi* (engl. *tail call elimination*). Vaikka koodin optimoinnista puhuminen on hiukan epätarkkaa kielenkäyttöä, koska kyseessä ei varsinaisesti ole optimin määrittäminen, puhutaan tämän tutkielman yhteydessä häntäkutsujen optimoinnista, koska termi on vakiintunut. Kun kyse on juuri häntärekursiosta, voidaan puhua myös *häntärekursion optimoinnista*.

Kuvassa 2.5 on esimerkki kokonaisluvun kertoman laskennasta häntärekursiivisen apufunktion avulla. Funktio `fact` määrittelee paikallisen apufunktion `fact-acc`, joka kutsuu itseään häntäpositiossa. Ympäröivän funktion `fact` tarkoitus on alustaa apufunktion tarvitsemat laskennan tukena käytettävät parametrit sopivilla lähtöarvoilla.

Esimerkin tapauksessa `fact-acc`-funktion tarvitsema `acc`-kerääjäparametri (engl. *accumulator*) alustetaan arvolla 1. Apufunktio käyttää kerääjäparametria lopullisen tu-

```
(define fact
  (lambda (n)
    (define fact-acc
      (lambda (n acc)
        (if (zero? n)
            acc
            (fact-acc (- n 1)
                      (* n acc)))))
    (fact-acc n 1)))
```

Kuva 2.5: Häntärekursiivinen kokonaisluvun kertoman laskenta

loksen "keräämiseen" kierros kierrokselta. Tämän kaltainen kerääjäparametrin lisäys rekursiiviseen funktioon on yleinen tapa muuntaa funktioita häntärekursiiviseen muotoon funktionaalisissa kielissä. Osa kirjoittajista kutsuu kuvan 2.5 kaltaista paikallisesti käytettävää häntärekursiivista apufunktiota *funktionaaliseksi silmukaksi* [SS02], koska rakennetta käytetään funktionaalisissa kielissä silmukan kaltaisesti.

Yleistetyt häntäkutsut

Vaikka häntäkutsuista puhutaan usein juuri häntärekursion yhteydessä, kaikki häntäkutsut eivät ole rekursiivisia. Häntäkutsu on mikä tahansa funktiokutsu, joka esiintyy funktion viimeisenä lausekkeena. Kuvassa 2.6 on esimerkkejä erilaisista häntäpositiossa olevista funktiokutsuista. Esimerkiksi funktiossa, jonka runko koostuu `if`-lausekkeesta, `if`-lausekkeen molemmat haarat ovat rungossa häntäpositiossa.

```
(define selector
  (lambda (predicate? value)
    (if (predicate? value)
        (then-function value)
        (else-function value))))

(define sum
  (lambda (l)
    (if (null? l)
        0
        (+ (car l)
           (sum (cdr l))))))

(define prime?
  (lambda (n)
    (and (> 2 n)
         (not-divisible? n))))

(define print-and-double-when
  (lambda (n pred?)
    (if (pred? n)
        (begin
          (display n)
          (* 2 n)))))
```

```
(define f
  (lambda (continue-fun)
    (define value (get-value))
    (display value)
    (continue-fun value)))

(define g
  (lambda (args)
    ((get-closure)
     (transform-args args))))

(define outside-bounds?
  (lambda (n lower-bound upper-bound)
    (or (< n lower-bound)
        (> n upper-bound))))
```

Kuva 2.6: Esimerkkejä häntäkutsuista. Häntäpositiossa olevat funktiokutsut on alleviivattu. Muut funktiokutsut on merkitty vaaleammalla tekstillä. Esimerkkifunktiossa `g` vain viimeinen funktiokutsu `get-closure`-funktion palauttamaan sulkeumaan on häntäpositiossa, joten vain sulkeet on alleviivattu.

Häntäkutsu voi myös olla epäsuorasti rekursiivinen. Epäsuora häntärekursio voi syntyä

esimerkiksi, kun kaksi keskenään rekursiivista funktiota kutsuu toisiaan häntäkutsuilla. Esimerkki tällaisesta tapauksesta on kuvan 2.7 rekursiivinen määritelmä parittomille ja parillisille luvuille.

```
(define odd?
  (lambda (x)
    (if (zero? x)
        #f
        (even? (- x 1))))) ; häntäkutsu funktioon even?

(define even?
  (lambda (x)
    (if (zero? x)
        #t
        (odd? (- x 1))))) ; häntäkutsu funktioon odd?
```

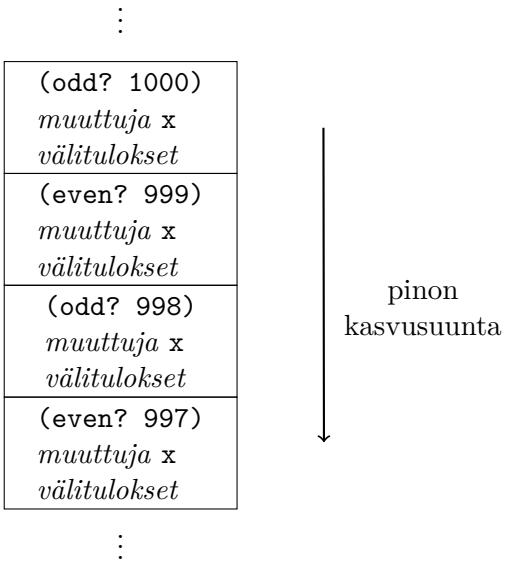
Kuva 2.7: Esimerkki keskinäisestä häntärekursiosta Scheme-kielellä. Literaalit `#f` (false) ja `#t` (true) ovat Schemen käyttämät totuusarvovakiot. Esimerkin häntäkutsut ovat epäsuorasti rekursiivisia.

Useiden funktionaalisten kielten määritelmät vaativat, että toteutus optimoi häntäkutsut. Esimerkiksi Schemen stardardi edellyttää, että häntärekursiiviset funktiot suoritetaan vakiotilassa, eivätkä muutkaan häntäkutsut kasvata aktivaatiotietuepinoa rajatta [Sus13, luku 3.5]. Häntärekursion optimointi on käytännössä välttämätöntä kielelle, jossa häntärekursio on ainoa tapa määritellä iteraatiorakenteita muiden silmukkarakenteiden puuttuessa. Keskinäinen rekursio vaatii kuitenkin jo tavallista funktionaalista silmukkaa monimutkaisempia häntäkutsujen optimointitekniikoita.

Optimoimattomat häntäkutsut tai kutsut, jotka eivät ole häntäpositiossa, johtavat suurilla syötteillä pinon ylivuotoon, koska jokainen proseduurikutsu varaa pinosta tilaa paikallisille muuttujilleen ja laskennan välituloksille (kuva 2.8). Kaikki häntäkutsujen optimointitekniikat siis edellyttävät, että pinosta siivotaan aika ajoin pois varattuja alueita, joita ei enää tarvita. Optimitilanteessa jokainen aktivaatiotietue poistettaisiin pinosta ennen seuraavaa häntäkutsua.

Continuation passing style

Yleinen häntäkutsujen optimointi mahdollistaa keskinäisen rekursion tehostamisen lisäksi *continuation passing style* -nimisen ohjelmointityylin [SJ75, SJ78] eli CPS-muodon käytön. Funktionaalisten kielten kääntäjät muuntavat usein käännettävän koodin CPS-muotoon tiettyjen optimointien helpottamiseksi, eli CPS-muoto toimii kääntäjässä välikielenä [Kel95]. CPS-muunnos voidaan tehdä esimerkiksi abstraktille syntaksipuulle ennen koodinluontia. Imperatiivisissa kielissä käytännössä vastaavana välikielenä toimii usein *static*



Kuva 2.8: Pinon tila funktiokutsun `(odd? 1000)` evaluoinnin aikana ilman häntäkutsujen optimointia.

single assignment form eli SSA-muoto [Kel95]. Steelen toteuttama ensimmäinen Scheme-kääntäjä Rabbit on varhainen esimerkki CPS-tyylin käytöstä kääntäjissä [SJ78]. Mikä tahansa koodi on mahdollista muuntaa CPS-muotoon ohjelmallisesti [App92].

CPS-muodossa yksikään funktio ei palaa kutsusta, vaan kontrolli etenee häntäkutsusta toiseen siten, että jokaiselle proseduurikutsulle annetaan ylimääräisenä parametrina kutsuttava *kontinuaatio* [SJ78, s. 67–68], joka on tyypillisesti funktioarvo. Arvon palauttamisen sijasta kutsuttu proseduuri kutsuu parametrina annettua kontinuaatiota antaen "paluuarvonsa" sille parametrina. CPS-muodossa kaikki funktiokutsut ovat häntäkutsuja.

Kuvassa 2.9 aiempi esimerkki (kuva 2.1, sivu 7) kokonaisluvun kertoman laskennasta Schemellä on muunnettu CPS-muotoon. Kuva on Scheme-koodin muodossa oleva esitys abstraktille syntaksipuulle tehtävästä muunnoksesta — ei siis varsinaista Scheme-koodia, jota tuotettaisiin missään vaiheessa käännösprosessia.

Esimerkkikoodissa `*`-merkkiin-päättyvät proseduurien nimet viittaavat proseduureihin, jotka ottavat viimeisenä parametrinaan kontinuaation. CPS-muotoisen koodin toiminta edellyttää, että kaikista käytettävistä kirjastofunktioista on olemassa kontinuaatioparametrin ottavat versiot. Esimerkkikoodissa viitatut funktiot kuten `zero?*` ja `display*` oletetaan sisäänrakennetuiksi funktioiksi, jotka ovat CPS-muotoa tukevia versioita R7RS-standardissa määritellyistä samannimisistä kirjastofunktioista.

Esimerkissä proseduuri `fact*` rakentaa jokaista proseduurikutsua varten kontinuaatioksi anonyymin proseduurin, joka ottaa parametrinaan kutsutun proseduurin tuloksen ja antaa sen parametrina seuraavan askeleen suorittavalle proseduurille. Viimeisenä

kutsuttava kontinuaatio on `EXIT`, joka voi olla esimerkiksi suoritusaikaisen ympäristön tarjoama funktio, joka päättää ohjelman suorituksen palaamatta funktiokutsusta.

```
(define (fact* n cont)
  (zero?* n
          (lambda (isZero)
            (if isZero
              (cont 1)
                (-* n
                    1
                    (lambda (nsub)
                      (fact* nsub ; epäsuorasti häntärekursiivinen kutsu
                             (lambda (f)
                               (** n f cont)))))))))

(fact* 10
  (lambda (n)
    (display* n EXIT)))
```

Kuva 2.9: Esimerkki kokonaisluvun kertoman laskennasta ja laskennan lopputuloksen tulostuksesta CPS-tyylillä. Kontinuaatioparametri ja siihen tehtävät viittaukset on korostettu `fact*`-proseduurin rungossa.

2.4 Muita funktionaalisiin kieliin yhdistettyjä piirteitä

Koska funktionaalisten kielten piirteiden kirjo on laaja, tutkielmassa keskitytään edellä kuvattuihin useimmille funktionaalisille kielille yhteisiin piirteisiin. Erityisesti tutkielman ulkopuolelle on rajattu niin sanottuihin *puhtaasti funktionaalisiin* (engl. *purely functional*) kieliin liittyvät ongelmat.

Puhtaasti funktionaalisiksi kieliksi kutsutaan kieliä, jotka suosivat vahvasti sivuvaikutuksettomia proseduureja ja rajoittavat sivuvaikutusten käyttöä [Sab98, Hud89]. Proseduuri on sivuvaikutukseton, jos sen tulos riippuu ainoastaan sille annetuista parametreista eikä proseduuri tee muutoksia ohjelman tilaan. Toisin sanoen sivuvaikutukseton proseduuri palauttaa samalla syötteellä kutsuttuna aina saman arvon.

Sivuvaikutuksettomia proseduureja kutsutaan joskus *puhtaiksi funktioiksi* [Sab98, Hud89]. Toisinaan puhtaista funktioista käytetään myös yksinkertaisesti termiä *funktio* ja sivuvaikutuksellisista proseduureista termiä *aliohjelma* (engl. *routine*, *subroutine*) [Str00]. Scheme-sanastossa eri tyyppisten proseduurien välille ei tehdä eroa, vaan molempia nimitetään yksinkertaisesti proseduureiksi [Que03, s. xviii]. Tässä tutkielmassa termejä proseduuri ja funktio käytetään synonyymeina.

Puhtaasti funktionaaliset kielet mahdollistavat yhtälömäisen päättelyn, eli niillä kirjoitetun ohjelmakoodin käyttäytymisestä on mahdollista tehdä samankaltaisia päätelmiä

kuin matemaattisista yhtälöistä [App98, s. 309]. Tämä on seurausta juuri sivuvaikutuksettomien proseduurien suosimisesta. Tyyppiesimerkki tällaisesta kielestä on Haskell.

Funktionaalisiin kieliin liitetään usein myös monia muita piirteitä, kuten erityisesti puhtaasti funktionaalisiin kieliin yhdistettävä laiska evaluaatio [Sco09, s. 523–524][Hud89] ja hahmonsovitus [Hud89]. Lisäksi funktionaalisille kielille on tyypillistä käyttää tietynlaisia funktionaalisia linkitettyjä listoja [Sco09, s. 508]. Erityistä funktionaalisissa kielissä käytetyille linkitetylle listoille on se, että ne ovat usein *pysyväistietorakenteita* (engl. *persistent data structure*), eli ne ovat muuttumattomia ja mahdollistavat usean rinnakkaisversion ylläpitämisen tietorakenteesta tehokkaasti [Oka98, s. 2]. Tällaisten listarakenteiden toimintaa on kuvailtu tarkemmin liitteessä A.1 erityisesti Lispin sukuisten kielten osalta. Useissa nykyaikaisissa funktionaalisissa kielissä kuten Scalassa ja Clojuressa myös muut muuttumattomat tietorakenteet kuin linkitetyt listat ovat pysyväistietorakenteita, mikä tekee niiden käytöstä tehokasta.

Erityisesti puhtaasti funktionaalisten kielten ja muidenkin staattisesti tyypitettyjen funktionaalisten kielten piirteiden kuten tyyppijärjestelmien ja laiskan evaluaation toteutukseen liittyy aivan oma ongelmajoukkonsa. Tämän vuoksi näitä piirteitä ei tarkastella tässä tutkielmassa. Tämän tyyppisiä ongelmia ovat käsitelleet esimerkiksi Haskellin kehittäjänä tunnettu Simon Peyton Jones [PJ87] sekä Scalan osalta Michel Schinz väitöskirjassaan [Sch05]. Jälkimmäinen keskittyy erityisesti Scalan tyyppijärjestelmän sovittamiseen yhteen JVM-alustan kanssa.

Schemen kaltaiset kielet, jotka tukevat funktionaalisen tyylin lisäksi myös joitakin imperatiivisia ominaisuuksia, tuovat esimerkiksi sulkeumien toteutukseen mielenkiintoista problematiikkaa sekä voivat soveltua helpommin vertailtaviksi esimerkiksi funktionaalisia piirteitä tukevien moniparadigmakielten kanssa. Tämä mahdollistaa vertailun monien erilaisten virtuaalikonealustoilla toteutettujen kielten välillä.

3 Kohdealustat

Tutkielmassa tarkastelun kohteina ovat kaksi suurinta ohjelmointikielten virtuaalikonealustaa, Oraclen Java Virtual Machine eli JVM [LYBB15] sekä Microsoftin hieman uudempi ISO- ja ECMA-standardoitu Common Language Infrastructure eli CLI [ECM12]. JVM:n tunnetuin toteutus on Oraclen jakelema Java Runtime Environment ja sen sisältämä Java HotSpot -virtuaalikone [Ora16d]. CLI:n toteutuksia ovat Windows-alustalla Microsoftin oma .NET Framework [Mic16a] ja sen virtuaalikonekomponentti Common Language Runtime (CLR) sekä avoimen lähdekoodin Mono Windows-, Linux- ja OS X -alustoilla [Mon16a].

Virtuaalikoneiden suorituskyvyn paraneminen on herättänyt monien dynaamisten ja funktionaalisten kielten toteuttajien kiinnostuksen kielten virtuaalikonetoteutuksiin. Houkuttelevia tekijöitä ovat erityisesti siirrettävyys, mahdollisuus käyttää olemassa olevia muilla virtuaalikonekielillä toteutettuja kirjastoja sekä valmis suoritusaikainen ympäristö roskienkerääjineen ja suoritusaikaisen profiloinnin perusteella optimoivine JIT-kääntäjineen [MG01]. Lisäksi virtuaalikonealustojen käyttämät korkean tason tavukoodikielet tekevät esimerkiksi tyyppien suoritusaikaisen yhteensopivuuden ja muistin käsittelyn oikeellisuuden varmistamisesta helpompaa verrattuna matalamman tason kohdekieliin kuten assembly-kieliin tai C-kieleen [MG01]. Mahdolliset virheet voidaan havaita jo suoritusta edeltävässä tavukoodin tarkastuksessa.

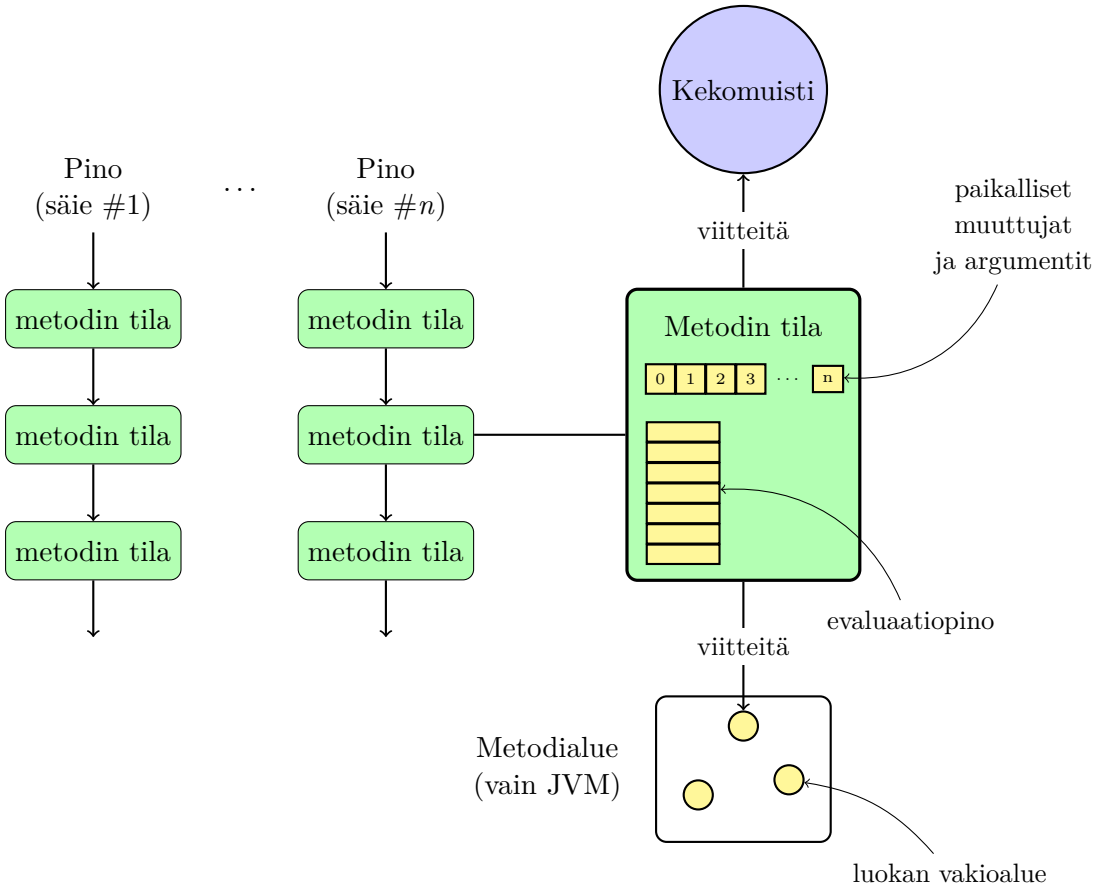
Molempien virtuaalikoneiden tarjoamat palvelut kielten toteuttajille ovat perinteisesti suosineet erityisesti oliokielten ja imperatiivisten kielten vaatimuksia muun tyyppisten kielten kustannuksella [WWW+13]. Erityisesti JVM oli nimensä mukaisesti alun perin juuri Javaa varten suunniteltu suoritusympäristö, mikä on jo pitkään aiheuttanut päänvaivaa muiden kielten toteuttajille JVM-alustalla [Ros09].

Meijerin ja Goughin [MG01] mukaan CLI puolestaan suunniteltiin alusta alkaen monenlaisten kielten toteutusalustaksi, vaikka tuki alun perin olikin painottunutta oliokieliin ja imperatiivisiin kieliin. CLI onkin tarjonnut jo ECMA-standardin ensimmäisestä versiosta lähtien esimerkiksi sisäänrakennetun häntäkutsukäskyn.

Funktionaalisten kielten suosion kasvu on tuonut funktionaalisia piirteitä sekä JVM-alustan Javaan että CLI-alustan C#-kieleen. Monet uudemmat funktionaaliset kielet kuten Scala ja Clojure sekä funktionaalista tyyliä tukeva oliokieli Kotlin puolestaan ovat valinneet JVM:n ensisijaiseksi toteutusalustakseen. Tämä on saanut myös Oraclen laajentamaan Java-virtuaalikoneen tarjoamia palveluita esimerkiksi dynaamisten kielten piirteiden ja sulkeumien toteuttamista tukevilla ominaisuuksilla. Tässä luvussa esitellään JVM:n ja CLI:n suoritusmallin ja tavukoodin perusteet sekä pohjustetaan funktionaalisten piirteiden toteuttamiseen vaikuttavien virtuaalikoneiden erityispiirteiden käsittelyä luvussa 4.

3.1 Suoritusmalli

Sekä JVM että CLI määrittelevät omanlaisensa abstraktin koneen ja käskyjoukon, jota alustalle käännetyt ohjelmat käyttävät. Molempien alustojen abstrakti kone perustuu pinomaiseen tietorakenteeseen, joka muistuttaa käyttäytymiseltään ja tarkoitukseltaan esimerkiksi C:n kaltaisten kielten pinoa [LYBB15, s. 12] [ECM12, s. 81-82]. Pino ei välttämättä ole yhtenäinen muistialue [LYBB15, s. 12], vaan se voidaan esittää esimerkiksi linkitettynä listana. Kuvassa 3.1 on yksinkertaistettu kuvaus molempien alustojen suoritusmallien yhteisistä piirteistä.



Kuva 3.1: Yksinkertaistettu esitys JVM:n ja CLI:n suoritusmalleista. Luokkien vakioalueet ovat JVM-alustan ominaisuus.

Molemmilla alustoilla suoritusympäristö on monisäikeinen, ja jokaisella säikeellä on oma pinonsa. Kun tehdään metodikutsu, säie luo metodin tilaa esittävän tietueen ja työntää sen pinoon. Tätä metodin tilan esitystä voidaan kutsua aktivaatiotietueeksi. Kun metodin suoritus päättyy joko poikkeukseen tai tavalliseen metodista paluuseen, metodin aktivaatiotietue poistetaan pinosta. Pinon koolla on yleensä tietty ennalta määritelty yläraja, jonka ylittymisen jälkeen ohjelman suoritus päättyy pinon ylivuotovirheeseen. Pi-

non sisältö on täysin suoritusaikaisen ympäristön hallinnassa, eli ohjelmoija tai kääntäjän koodinluontiosan toteuttaja ei voi itse manipuloida pinoa.

Tärkeimmät aktivaatiotietueen sisältämät osat ohjelmoijan kannalta ovat paikalliset muuttujat, metodin argumentit sekä evaluaatiopino. JVM-alustalla evaluaatiopinoa nimitetään operandipinoksi [LYBB15, s. 17]. Metodin argumentit ja paikalliset muuttujat talletetaan taulukkoon. JVM taltioi sekä argumentit että paikalliset muuttujat samaan taulukkoon [LYBB15, s. 16–17], kun taas CLI varaa molemmille erillisen taulukon [ECM12, s. 84]. Muuttujien ja argumenttien lataukset tehdään viittaamalla suoraan taulukoiden indekseihin.

Evaluaatiopinoon talletetaan välitulokset metodin varsinaisen laskennan aikana. Tämän pinon sisältöä manipuloidaan tavukoodikäskyjen avulla.

Yksinkertaiset primitiiviarvot kuten numerot ja totuusarvot talletetaan sellaisinaan paikalliseen muuttujataulukkoon tai evaluaatiopinoon. JVM-alustalla tietyt suurempikokoiset primitiiviarvot kuten `double`- ja `long`-tyyppiset arvot vievät kaksi paikkaa sekä evaluaatiopinosta että paikallisten muuttujien taulukosta [LYBB15, s. 17]. Muita arvoja eli *viittaustyyppisiä* arvoja käsitellään niiden muistiosoitteiden avulla. Tällöin muistiosoitteet osoittavat kekomuistista varattuun alueeseen.

JVM-alustalla aktivaatiotietue sisältää lisäksi viittauksen luokan *vakioalueeseen* (engl. *constant pool*) [LYBB15, s. 18]. Vakioalueeseen talletetaan esimerkiksi koodissa käytetyt numeeriset ja merkkijonoliteraalit sekä tiedot, jotka tarvitaan kutsuttavien metodien löytämiseksi suoritusaikana. Jokaisella luokalla on oma vakioalue, joka sijaitsee JVM-ympäristön metodialueessa eli paikassa, johon muun muassa käännettyjen metodien koodi talletetaan suoritusajaksi [LYBB15, s. 14].

3.2 Tavukoodi

Kun ohjelmakoodi käännetään suoritettavaksi JVM- tai CLI-alustalla, muodostetaan tiedosto tai joukko tiedostoja, jotka sisältävät luokkien, metodien ja luokkatason muuttujien määrittelyt sekä metodien koodin. Molempien virtuaalikoneiden hyväksymä matalan tason kieli on siis lähtökohtaisesti luokkapohjainen oliokieli. JVM-alustalla määritelmät kirjoitetaan joukkoon `class`-tiedostoja, joista jokainen sisältää yhden luokan määrittelyn sekä luokan vakioalueeseen talletettavat arvot. CLI-alustalla luodaan `exe`- tai `dll`-tiedosto, joka sisältää kaikkien ohjelman luokkien määrittelyt.

Luokkien määrittelyt muistuttavat rakenteeltaan pääpiirteittäin esimerkiksi Javan tai C#:n luokkia. Javaa tunteva tunnistaa esimerkiksi kuvan 3.2 JVM-koodista äkkiä HelloWorld-nimisen luokan määrittelyn, luokan oletuskonstruktorin sekä main-metodin, vaikka syntaksi poikkeaakin monilta osin Javasta. Oleellisin ero onkin metodien rungoissa, jotka koostuvat tutun Java-koodin sijaan tavukoodikäskyistä. Kuvassa 3.3 on vastaava

esimerkki CLI-alustan tavukoodista.

Java-tavukoodin (kuva 3.2) `main`-metodin numeroidulla rivillä 0 ladataan pinoon `System`-luokassa staattisena muuttujana `out` määritelty `PrintStream`-tyyppinen olio. Seuraavana rivillä 3 ladataan pinoon tulostettava merkkijono, joka lopulta tulostetaan `PrintStream`-olion `println`-metodia kutsumalla rivillä 5. CLI-tavukoodin (kuva 3.3) `main`-metodissa tulostus tehdään kutsumalla `Console`-luokassa määriteltyä staattista metodia `WriteLine`. Koska kutsuttava metodi on staattinen, ei `main`-metodissa tarvitse ladata kutsun kohteena toimivaa oliota kuten kuvan 3.2 Java-tavukoodin tapauksessa.

Virtuaalikoneiden suorittamaa koodia on perinteisesti kutsuttu tavukoodiksi, koska tavukoodikäskyt ovat yleensä yhden tavun mittaisia tunnisteita, jotka määrittävät käytettävän käskyn. Näin on esimerkiksi JVM:n tapauksessa [LYBB15, s. 25], mutta CLI-alustalla tavukoodikäskyt voivat koostua useammastakin tavusta [ECM12, s. 295]. Kuvien 3.2 ja 3.3 koodissa näkyvät ihmisille luettavassa muodossa olevat käskyjen nimet. Käytännössä siis esimerkiksi jokaista JVM-esimerkkikoodissa käytettyä käskyä siis kuitenkin vastaa jokin yhteen tavuun mahtuva kokonaislukuarvo. CLI-alustan tavukoodi on nimeltään Common Intermediate Language eli CIL. JVM-alustan tavukoodia kutsutaan usein Java-tavukoodiksi.

Monet käskyistä voivat lisäksi ottaa yhden tai useampia parametreja. Mahdollisten parametrien määrä riippuu käskystä. Kuvassa 3.2 osa käskyistä ottaa parametrinaan numeron, jota edeltää `#`-merkki. JVM-alustalla nämä ovat viitteitä luokan vakioalueeseen. `Javap`-ohjelman tuottamassa tulosteessa viitatut arvot luokan vakioalueesta on merkitty kommentteina rivien viereen koodin lukemisen helpottamiseksi. Esimerkiksi main-metodin rivi `ldc #3` lataa luokan vakioalueeseen paikkaan 3 talletetun merkkijonovakion `"Hello, world!"`. CLI-alustalla ei ole vastaavaa vakioalueen käsitettä, joten parametrit ovat suoraan tavukoodissa käskyjen yhteydessä. Esimerkiksi paluukäskyt `return` ja `ret` puolestaan eivät ota lainkaan parametreja.

Molemmat alustat tarjoavat koodinluontiin apukirjastoja, joten kääntäjän toteuttajan ei tarvitse itse huolehtia kaikista yksityiskohdista, kuten evaluaatiopinon kokovaatimuksista tai JVM:n luokkien vakioalueiden rakentamisesta. CLI-alustalla tarvittavat työkalut löytyvät suoraan .NET-kirjastoista, `System.Reflection`-nimiavaruudesta, minkä lisäksi Mono-alustalla on oma kirjastonsa, Mono.Cecil [Mon16b]. JVM-alustalla todennäköisesti suosituin kirjasto on ObjectWeb ASM, jota käytetään esimerkiksi Clojure-, JRuby- ja Jython-kääntäjissä [asm16].

CLI-alustalla on lisäksi kirjastokokoelma nimeltä DLR (*Dynamic Language Runtime*) [CT09], joka mahdollistaa koodin kääntämisen tavukoodin sijasta ennalta määritellyn muotoiseksi abstraktiksi syntaksipuuksi. DLR on suunnattu erityisesti dynaamisten kielten toteuttajille, ja se tarjoaa joitakin juuri dynaamisille kielille tarkoitettuja suoritusaikaisia lisäpalveluita.

HelloWorld.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

```
Classfile HelloWorld.class
  Last modified May 20, 2016; size 427 bytes
  Compiled from "HelloWorld.java"
public class HelloWorld
  flags: ACC_PUBLIC, ACC_SUPER[a]
{
  public HelloWorld();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1 // Method java/lang/Object."<init>":()V
         4: return

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2 // Field java/lang/System.out:
                             //       Ljava/io/PrintStream;
         3: ldc           #3 // String Hello, world!
         5: invokevirtual #4 // Method java/io/PrintStream.println:
                             //        (Ljava/lang/String;)V
         8: return
}
```

[a] Lippu `ACC_SUPER` liittyy taaksepäinyhteensopivuuteen vanhempien Java-kääntäjien tuottaman tavukoodin kanssa [LYBB15, s. 72], eikä ole tärkeä tavukoodin ymmärtämisen kannalta.

Kuva 3.2: Javalla kirjoitettu “Hello, world!” -ohjelma `HelloWorld.java` ja siitä tuotetun `class`-tiedoston sisältö JDK-työkalupaketissa mukana tulevalla `javap`-ohjelmalla purettuna. Koodista on karsittu luettavuuden vuoksi pois luokan vakioalueen määrittely ja muita yksityiskohtia.

HelloWorld.cs

```
using System;

public class HelloWorld
{
   public static void Main()
   {
      Console.WriteLine("Hello, World!");
   }
}
```

```
.class public auto ansi beforefieldinit HelloWorld
  extends [mscorlib]System.Object
{

  // method line 1
  .method public hidebysig specialname rtspecialname
         instance default void '.ctor' ()  cil managed
  {
      // Method begins at RVA 0x2050
  // Code size 7 (0x7)
  .maxstack 8
  IL_0000:  ldarg.0
  IL_0001:  call instance void object::'.ctor'()
  IL_0006:  ret
  } // end of method HelloWorld::.ctor

  // method line 2
  .method public static hidebysig
         default void Main ()  cil managed
  {
      // Method begins at RVA 0x2058
  .entrypoint
  // Code size 11 (0xb)
  .maxstack 8
  IL_0000:  ldstr "Hello, world!"
  IL_0005:  call void class [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
  } // end of method HelloWorld::Main

} // end of class HelloWorld
```

Kuva 3.3: C#-kielellä kirjoitettu “Hello, world!”-ohjelma ja siitä Mono-alustalla tuotetun `exe`-tiedoston sisältö `monodis`-ohjelmalla purettuna. Vastaavasti kuin kuvan 3.2 JVM-tavukoodissa, esimerkistä on karsittu esimerkiksi CLI:n assembly-yksikön määrittelyyn liittyviä yksityiskohtia.

Toiminto	JVM-käskyt	CLI-käskyt
paikallisen muuttujan lataus	aload, iload, lload, ...	ldloc, ldarg
talletus paikalliseen muuttujaan	astore, istore, lstore, ...	stloc, starg
staattinen metodikutsu	invokestatic	call
oliometodikutsu	invokevirtual	call, callvirt
rajapintametodikutsu	invokeinterface	callvirt
olion luonti ja alustus konstruktorilla	new + invokespecial	newobj
ehdollinen hyppy	ifeq, ifne, ...	brfalse, brtrue, ...
ehdoton hyppy	goto	br
paluu metodista	return, areturn, ireturn, ...	ret

Kuva 3.4: Esimerkkejä tavukoodikäskyistä

3.3 Tärkeimpiä tavukoodikäskyjä

Molemmat alustat määrittelevät suuren määrän käskyjä esimerkiksi laskentaa ja erilaisten primitiivityyppien käsittelyä varten. Tutkielman aiheen kannalta kiinnostavimpia käskytyyppejä ovat erilaiset metodikutsukäskyt sekä hyppykäskyt. Lisäksi kaikki esimerkkikoodi sisältää ainakin lataus- ja talletuskäskyjä. Tämä luku esittelee muutamia keskeisimpiä käskyjä. Kuvan 3.4 taulukossa on esimerkkejä käskyistä.

Latauskäskyillä tarkoitetaan käskyjä, jotka lataavat eli työntävät evaluaatiopinoon joko vakioarvoja tai muuttujista ladattuja arvoja. Tällaisia käskyjä ovat esimerkiksi CLI-alustan `ldarg` (*load argument*) ja `ldloc` (*load local*), jotka työntävät pinoon paikallisten muuttujien arvoja tai metodin parametrien arvoja. JVM-alustalla vastaavat käskyt ovat tyypitettyjä, mutta eivät tee eroa metodin parametrien ja paikallisten muuttujien välille: esimerkiksi käsky `aload` (*address load*) lataa paikallisen muuttujan tai metodin parametrin arvon, jota käsitellään viittaustyyppisen arvon osoitteena.

Talletuskäskyt puolestaan poistavat evaluaatiopinosta päällimmäisen arvon ja talletttavat sen käskyn osoittamaan muuttujaan. Jokaisella edellä mainituista latauskäskyistä on vastine talletuskäskyjen joukossa. Nämä käskyt näkyvät kuvassa 3.4. Sekä latausettä talletuskäskyt ottavat parametrinaan numeerisen indeksin paikalliset muuttujat tai argumentit sisältävään taulukkoon.

Paluu metodista tehdään JVM-alustalla tyypitetyillä käskyillä, kun taas CLI-alustalla paluuseen käytetään aina samaa `ret`-käskyä. JVM-alustalla `return`-käskyä käytetään `void`-paluutyyppisissä metodeissa. Muut käskyt kuten viittaustyyppisen arvon palauttava käsky `areturn` poistavat evaluaatiopinosta päällimmäisen arvon ja palauttavat sen kutsuvalle metodille työntämällä arvon kutsuvan metodin evaluaatiopinoon.

Hyppykäskyt ovat joko ehdollisia tai ehdottomia ja ottavat parametrinaan yhden kohdeosoitteen koodissa. Ehdolliset käskyt poistavat yhden tai useampia arvoja pinosta ja päättävät käskystä riippuvan vertailun perusteella, tehdäänkö hyppy. Ehdottomat käs-

kyt hyppäävät ilman vertailua. Esimerkiksi JVM-alustan `ifeq` ja CLI-alustan `brfalse` suorittavat hypyn, jos pinon päällimmäinen arvo voidaan tulkita epätodeksi totuusarvoksi. Todellisuudessa JVM käsittelee totuusarvoja kokonaislukuina. Käsky `ifeq` siis vertaa pinossa olevaa arvoa nollaan. Vastaavasti .NET-alustalla `null` ja `0` tulkitaan totuusarvovertailua tehtäessä epätodeksi.

Sekä JVM:llä [LYBB15, s. 322] että CLI:llä [ECM12, s. 308] hyppykäskyn kohteeksi annettavan osoitteen on oltava saman metodin sisällä kuin itse hyppykäsky. Hyppykäskyn avulla ei siis ole mahdollista siirtää suoritusta esimerkiksi toisen metodin sisään.

Molemmat alustat tarjoavat osin vastaavat joukot metodikutsukäskyjä. Tärkeimmät käskyt näkyvät kuvassa 3.4. JVM-alustalla staattisille metodeille, oliometodeille ja rajapintametodeille on omat kutsukäskynsä. CLI-alustalla periaate on hiukan erilainen, ja osa käskyistä kelpaa käytettäväksi useammissa eri tapauksissa. Molemmilla alustoilla olio- ja rajapintametodeja kutsuttaessa annetaan ylimääräisenä parametrina olio, jonka metodia halutaan kutsua. Tämä olio annetaan aina ensimmäisenä parametrina, joten kutsutun metodin rungossa se on aina argumenttitaulukon indeksissä 0.

JVM-tavukoodista poiketen CLI antaa mahdollisuuden kutsua esimerkiksi oliometodia kahdella eli tavalla. Toinen tapa on käyttää `callvirt`-käskyä, jolloin kutsuttava metodi etsitään suoritusaikana parametrina annetun olion todellisen tyypin perusteella. Toinen tapa on käyttää staattisten metodienkin yhteydessä käytettävää tavallista `call`-käskyä, jolloin kutsuttavan metodin sisältävä luokka ilmoitetaan suoraan koodissa ja on siis tiedossa jo käännösaikana.

Uudempana ominaisuutena JVM-alustalla on Java SE 7 -versiossa esitelty kutsukäsky `invokedynamic`, jonka tarkoituksena on erityisesti tukea dynaamisten kielten metodikutsujen toteutusta [Ros09]. Jokaiselle `invokedynamic`-käskyn käyttöpaikalle määritetään käännösaikana *bootstrap-metodi*, jota kutsutaan suoritusaikana ennen kyseisen `invokedynamic`-käskyn ensimmäistä suorituskertaa [LYBB15, s. 35].

Bootstrap-metodi palauttaa kutsupaikkaolion (engl. *call site object*), `CallSite`-luokan ilmentymän, joka sisältää viitteen metodiin, jota `invokedynamic`-käskyn halutaan todellisuudessa kutsuvan suoritusaikana. Kutsupaikkaolio sidotaan `invokedynamic`-käskyn käyttöpaikkaan, ja myöhemmin käskyä suoritettaessa kutsu tehdään kutsupaikkaolion osoittamaan metodiin. Tämä mahdollistaa kutsuttavien metodien sidonnan siirtämisen suoritusaikaan ja antaa kääntäjän toteuttajalle mahdollisuuden vaikuttaa metodin sidontaan bootstrap-metodin valinnalla ja toteutuksella. CLI-alustalla DLR-kirjastokokoelma tarjoaa käytännössä vastaavan toiminnallisuuden, jota esimerkiksi C# hyödyntää sellaisissa metodikutsuissa, joissa `dynamic`-avainsanalla esitelty muuttuja on metodikutsun argumenttina tai vastaanottajana [CT09, s. 17–19].

4 Funktioarvojen ja häntäkutsujen toteutus

Tämä luku esittelee yleisimmät kirjallisuudesta tunnetut sulkeumien ja häntäkutsujen optimoinnin toteutustekniikat. Lisäksi tarkastellaan virtuaalikoneympäristöjen aiheuttamia rajoituksia toteutustekniikoiden valinnalle sekä esitellään alustojen tarjoama sisäänrakennettu tuki häntäkutsuille ja sulkeumille.

Ensimmäinen aliluku esittelee perinteisillä alustoilla käytettävät sulkeumien toteutustekniikat. Toisessa aliluvussa perehdytään siihen, mitä vastaavia tekniikoita on mahdollista käyttää virtuaalikonealustoilla ja minkälaista sisäänrakennettua tukea ympäristöt tarjoavat sulkeumien toteuttamiseen. Kolmas aliluku keskittyy häntäkutsujen optimointitekniikoihin. Viimeisessä aliluvussa tarkastellaan millaisia toteutustekniikoita on valittu muutamissa JVM- ja CLI-alustojen suosituimpien kielten toteutuksissa sekä kahdessa Scheme-toteutuksessa, Kawassa ja Bigloossa.

4.1 Sulkeumien toteutukset perinteisillä alustoilla

Imperatiivisissa kielissä kuten C:ssä proseduurien suoritusaikainen esitys on suoraviivaista. Proseduuriin viittaamiseksi tarvitaan vain sen suoritusaikaisen koodin muistiosoite, jota voidaan helposti käyttää myös argumenttina kutsussa ja tallettaa muuttujaan. Verrattuna funktionaalisten kielten funktioarvoihin C:n proseduurit ovat kuitenkin rajoittuneita. C-ohjelmoija ei voi määritellä sisäkkäisiä funktioita, eivätkä kutsuparametreina kuljetettavat funktio-osoittimet voi kuljettaa mukanaan funktion rungon ulkopuolella määriteltyjä arvoja tai tilaa.

Kun käännetään sisäkkäiset proseduurit sallivaa lähtökieltä imperatiiviselle, suoria osoittimia muistiin tukevalle kohdekielelle, yksinkertaisinta on toteuttaa tapaus, jossa sisempänä määriteltyä funktiota käytetään ainoastaan emofunktion sisältä käsin, kuten funktiota `g` kuvan 4.1 esimerkkikoodissa. Koska tällaisen funktion ei tarvitse elää emofunktiotaan kauemmin, on mahdollista käyttää sulkeuman ympäristön toteutustapana *staattista osoitinta* (engl. *static link*) emofunktion aktivaatiotietueeseen [App98, s. 312] [Sco09, s. 126–127]. Koska emofunktion aktivaatiotietue on olemassa aina, kun kuvan 4.1 funktiota `g` kutsutaan, emofunktion aktivaatiotietueeseen voidaan siis viitata suoraan.

Funktionaalisissa kielissä kuten Schemessä funktiot ovat ensimmäisen luokan arvoja, joten funktio voi esimerkiksi palauttaa sen sisällä määritellyn funktion paluuarvonaan. Tällöin sulkeuman kaappaamien muuttujien elinikä on pitempi kuin niitä ympäröivän funktion. Tässä tilanteessa staattinen osoitin aktivaatiotietueeseen ei enää ole riittävä toteutus. Funktionaaliset kielet tyypillisesti takaavat, että sulkeumien kaappaamien paikallisten muuttujien elinaika on sidottu sulkeuman elinaikaan [Sco09, s. 156]. Tällöin staattisen osoittimen ylläpitäminen kasvattaisi turhaan aktivaatiotietuepinoa, koska aktivaatiotietueita ei voitaisi poistaa pinosta, niin kauan kuin niihin viittaavia sulkeumia

```
(define h
  (lambda (f)
    (f 9)))

(define f
  (lambda (x)
    (define g (lambda (y) (+ x y)))
    (h g)))
```

Kuva 4.1: Paikallisesti käytettävä apufunktio `g`, jonka elinaika on sidottu sen emofunktioon

on olemassa.

Sulkeuman ympäristön esitys

Kuvassa 4.2 on esimerkki funktiosta, jonka sisällä määritellyt funktiot muuttavat sulkeumansa kaappaamaa arvoa. Esimerkkifunktio `counter` määrittelee paikallisen muuttujan `c` sekä funktiot `get-counter`, joka palauttaa muuttujan `c` arvon, ja `increment-counter`, joka kasvattaa muuttujan `c` arvoa yhdellä. Proseduuri `increment-counter` käyttää `set!`-lauseketta, joka muuttaa aiemmin `define`-rakenteella määritellyn muuttujan arvoa. R7RS-standardin mukaan `set!`-lausekkeen paluuarvo on määrittelemätön [Sus13, luku 4.1.6], eli lausekkeella on oltava jokin paluuarvo, mutta varsinainen arvo on toteutuksen valittavissa [Sus13, luku 1.3.2].

Tässä tilanteessa tarvitaan sulkeuman ympäristölle esitys, johon on mahdollista kaapata viite, joka säilyy käyttökelpoisena molempien proseduurien koko elinajan. Lisäksi molempien proseduurien on nähtävä sama `c`-muuttujan arvo kaikkina ajanhetkinä, eli mikäli muuttuja `c` on esitetty esimerkiksi muuttumattomana primitiivityyppisenä arvona, molemmat funktiot eivät voi yksinkertaisesti luontihetkellään kaapata muuttujan `c` kokonaislukuarvoa, koska tällöin kumpikin funktio saisi siitä oman kopionsa. Staattisia osoittimia yleiskäyttöisempi ratkaisu on käyttää sulkeuman ympäristön toteuttamiseen dynaamisesta muistista eli keosta varattua muistialuetta.

Yksi mahdollisuus on sijoittaa aktivaatiotietueet kokonaisuudessaan keosta varattuun muistiin, jolloin niitä voidaan käyttää hyvin samankaltaisesti kuin pinossa olevia aktivaatiotietueita [App98, s. 312]. Tällainen ratkaisu ei kuitenkaan ole käyttökelpoinen kääntäjätoteutuksissa virtuaalikonealustoilla, koska kääntäjän toteuttaja ei voi vaikuttaa alustan sisäänrakennetun aktivaatiotietuepinon toimintaan tai toteutukseen. Tulkkitoteutuksissa asia on toisin, koska tulkki voi tarvittaessa ylläpitää omaa versiotaan aktivaatiotietuepinosta. Koska dynaamisen muistin käyttö on aktivaatiotietuepinoa hitaampaa, näyttää lisäksi ilmeiseltä, että aktivaatiotietueiden varaus kokonaan dynaa-

```
(define counter
  (lambda ()
    (define c 0)
    (define get-counter
      (lambda () c))
    (define increment-counter
      (lambda ()
        (set! c (+ c 1))))
    (cons get-counter increment-counter)))

(define f (counter))
(define get-counter (car f))
(define increment-counter (cdr f))

(get-counter)       ⟹ 0
(increment-counter) ⟹ <unspecified>
(get-counter)       ⟹ 1
```

Kuva 4.2: Yksinkertainen kapseloitu laskuri ja esimerkki sen käytöstä

misesta muistista haittaisi ohjelmien suorituskykyä.

Toinen, tyylikkäämpi ratkaisu on luoda erillinen tietue, joka sisältää vain ne muuttujat, joihin sisemmät funktiot viittaavat [App98, s. 313]. Esimerkiksi kuvan 4.2 funktion tapauksessa voitaisiin muodostaa tietue, joka sisältäisi laskurimuuttujan `c`.

Muuttujan jakamiseen useamman sulkeuman välillä on muutamia erilaisia tekniikoita. Yksi vaihtoehto on luoda tietue, joka sisältää kaikki näkyvyysalueen pakenevat muuttujat ja jakaa sama tietue kaikkien samassa näkyvyysalueessa luotavien sulkeumien välillä. Tällaista tietuetta kutsutaan *pakomuuttujatietueeksi* (engl. *escaping variable record*) [App98, s. 313].

Toinen mahdollisuus on kääriä jokainen kaapattava muuttuja pieneen tietueeseen tai olioon, joka tarjoaa rajapinnan talletetun arvon lukemiselle ja asetukselle (muun muassa [Que03, s. 362]) ja antaa kaikkien sulkeumien kaapata viite samaan olioon. Tällaista rakennetta kutsutaan useissa ohjelmointikielissä *viitesoluksi* (engl. *reference cell*) — esimerkiksi F#:ssa [Del16b].

Useimmissa virtuaalikonealustojen kääntäjätoteutuksissa käytetään jotakin muunnelmaa viimeiseksi mainituista tekniikoista. Eri toteutusten välillä vaihtelua on kuitenkin sekä sulkeuman ympäristön esitystavoissa että siinä, mihin sulkeuman rungon koodi sijoitetaan.

Sulkeumamuunnos ja lambda-nosto

Jotta sulkeuman ympäristö pystytään kutsupaikalla palauttamaan kaapatut muuttujat sisältävästä tietueesta, tehdään koodille usein *sulkeumamuunnos* (engl. *closure conversion*) [App98, s. 326–328]. Varsinaisen sulkeumamuunnoksen tekeminen ei kuitenkaan ole kaikkien toteutustekniikoiden yhteydessä välttämätön vaihe, kuten nähdään luvussa 4.2.

Sulkeumamuunnoksessa funktiot muutetaan muotoon, jossa ne viittaavat ainoastaan rungossaan määriteltyihin paikallisiin muuttujiin ja muodollisiin parametreihin. Toisin sanoen funktion vapaat muuttujat siirretään funktion parametreiksi. Funktio voi tällöin ottaa ylimääräisinä parametreina joko kaikki kaapatut muuttujat erillisinä parametreina tai viitteen koko ympäristötietueeseen. Jälkimmäisessä tapauksessa funktio muistuttaa ”ympäristötietueoliolle” määriteltyä oliometodia.

Sulkeumamuunnoksen jälkeen kääntäjän on huolehdittava siitä, että sulkeuman kutsupaikkoja muutetaan niin, että ne osaavat antaa oikeat ympäristöparametrit kutsuttavalle sulkeumalle. Sulkeumamuunnos voidaan tehdä esimerkiksi käännöksen aikana rakennetulle abstraktille syntaksipuulle.

Koska monissa kohdekielissä sisäkkäiset funktiot eivät ole mahdollisia, saatetaan sulkeumamuunnoksen lisäksi käyttää tekniikkaa, jossa sisemmät funktiomääritelmät nostetaan ylätasolle [Que03, s. 363]. Tällöin koko ohjelma muuntuu yksinkertaisesti joukoksi globaaleja funktioita, jotka parametrisoidaan sulkeuman ympäristöllä. Tämä menetelmä tunnetaan nimellä *lambda-nosto* (engl. *lambda lifting*) ja sitä voidaan hyödyntää esimerkiksi toteutuksissa, joissa kohdekielenä on C [Que03, s. 363].

Termiä lambda-nosto voidaan käyttää myös silloin, kun sisäkkäisiä funktiomääritelmiä nostetaan oliopohjaiselle kohdekielelle käännettäessä esimerkiksi ympäröivän luokan tasolle (esim. [Sch05, Goe12]). Kuten luvuissa 4.2 ja 4.4 nähdään, tekniikkaa käytetään myös funktionaalisten piirteiden JVM- ja CLI-toteutuksissa.

Esimerkki sulkeumien kääntämisestä C-kielelle

Yksinkertaisimmillaan C-kieltä kohdekielenä käyttävän toteutuksen sulkeumaesitys voisi olla esimerkiksi `struct`-tietue, joka sisältäisi osoittimen sulkeuman rungon sisältävään globaaliin proseduuriin sekä ympäristön kaapatut arvot taulukkona, erillisinä kenttinä tietueessa tai omana tietueenaan. Kuvassa 4.3 on esimerkkinä Bigloo-kääntäjän C-takaosan sulkeumaesitys, joka on käytännössä vain hieman tätä monimutkaisempi sisältäen lisäksi lähinnä funktion parametrien määrään ja varargs-funktioiden kutsumiseen liittyviä lisätietoja [BSS04]. Bigloo-toteutus luo jokaiselle sulkeumalle oman `struct`-tyypin [BSS04].

Bigloo-esimerkissä `env`-etuliitteellä alkavat muuttujat ovat sulkeuman kaappaamia muuttujia. Bigloon sulkeumatietueessa on kaksi funktio-osoittimille varattua kenttää. Käytettävä kenttä valitaan sen perusteella, onko kyseessä kiinteän vai vaihtelevan ar-

```
struct procedure {
  int             arity;
  bigloo_object (*entry)();
  bigloo_object (*va_entry)();
  bigloo_object   env0;
  bigloo_object   env1;
  ...
};
```

Kuva 4.3: Bigloon C-takaosan käyttämä sulkeumaesitys [BSS04].

gumenttimäärän funktio. Kenttä `entry` on kiinteän määrän argumentteja hyväksyvä funktio, jonka muodollisten parametrien määrä on talletettu `arity`-muuttujaan. Kenttää `va_entry` puolestaan käytetään silloin, kun sulkeuma ottaa vaihtelevan määrän argumentteja.

Kuvassa 4.4 on esimerkki sulkeumamuunnoksesta kuvan 4.2 sulkeumille hypoteettisessa C-kieltä kohdekielenä käyttävässä toteutuksessa. Muunnettavat sulkeumafunktiot on esitetty kuvan yläosassa alkuperäisessä muodossaan. Muunnoksen jälkeen jokainen funktio saa ylimääräisenä parametrina `env`-nimisen osoittimen pakomuuttujatietueeseen, jota esimerkkikoodissa esitetään `struct`-tietueella `counter_env_t`. Käännetyt funktiot viittaavat kaapattuun muuttujaan `c` parametrina saatavan pakomuuttujatietueen kautta. Esimerkissä oletetaan, että rakenne `scheme_object_t` on tietue, joka voi esittää mitä tahansa Scheme-arvoa.

Tällaisessa toteutuksessa sulkeumat `get-counter` ja `increment-counter` luova funktio `counter` luo tarvittavan `counter_env_t`-tyyppisen tietueen ja tallettaa muuttujan `c` lähtöarvon tietueeseen. Tämän jälkeen funktio-osoittimet talletetaan Bigloo-esimerkkikuvan 4.3 kaltaiseen rakenteeseen. Kuva 4.5 esittää mahdollisen käännöksen `counter`-proseduurille. Muistin varaukseen ja paluuarvona käytettävän paritietorakenteen rakentamiseen liittyvät yksityiskohdat on jätetty esimerkistä pois. Esimerkissä molempia sulkeumia esitetään tietueella `counter_closure_t`, joka sisältää osoittimet sulkeumien kesken jaettuun pakomuuttujatietueeseen ja sulkeuman rungon toteuttavaan funktioon. Tällä tavoin esitettäviä sulkeumia kutsutaan antamalla funktio-osoittimen osoittamalle funktiolle parametrina sen kanssa samaan tietueeseen talletettu osoitin pakomuuttujatietueeseen.

```
(define get-counter
  (lambda () c))

(define increment-counter
  (lambda ()
    (set! c (+ c 1))))
```

```
typedef struct counter_env {
    scheme_object_t c;
} counter_env_t;

scheme_object_t get_counter(counter_env_t *env) {
    return env->c;
}

scheme_object_t increment_counter(counter_env_t *env) {
    env->c = scheme_add(env->c, CONSTANT_NUMBER_ONE);
    return CONSTANT_UNDEFINED;
}
```

Kuva 4.4: Mahdollinen C-kielelle käännettäessä käytettävä sulkeumamuunnos kuvan 4.2 (sivu 28) esimerkin sulkeumille. Metodi `increment_counter` suorittaa viimeisenä sivuvaikutuksellisen operaation, joten sen paluuarvona on vakio `undefined`.

4.2 Sulkeumien tuki virtuaalikonealustoilla

Sulkeumien ympäristöjen toteuttaminen JVM- ja CLI-alustoilla ei perustasolla poikkea oleellisesti toteutuksesta, joka tuottaa esimerkiksi C-koodia. Molemmat alustat tuovat kuitenkin mukanaan omanlaisensa rajoitteet ja mahdollisuudet.

Oliokielissä metodeita tai funktioita ei tyypillisesti voi käsitellä ensimmäisen luokan arvoina. Yksi oliokielissä jo pitkään käytetty menetelmä rajoitteen kiertämiseksi on niin sanottujen kutsuttavien olioiden käyttö. Kutsuttavat oliot ovat olioita, jotka määrittelevät yleensä yhden kutsumetodin. Koska molempien alustojen tavukoodit ovat oliokieliä, kutsuttavat oliot ovat yksi mahdollinen toteutustekniikka sulkeumille.

Modernit korkean tason oliokielet eivät myöskään useimmiten salli C:n tai C++:n kaltaisia suoria osoittimia metodeihin tyyppi- tai muistiturvallisuuden vuoksi, mikä rajoittaa mahdollisuuksia käyttää joitakin edellisessä aliluvussa esiteltyjä sulkeumien toteutustekniikoita. Tarkasteltavista alustoista vain CLI tukee raakoja metodiosoittimia tavukooditasolla.

Molemmilla tarkasteltavilla alustoilla on kuitenkin kehitetty myös turvallisempia vaihtoehtoja perinteisille funktio-osoittimille. CLI-alustalla funktio-osoittimia voidaan esittää delegaattiluokkien avulla [AA12, s. 119–120]. JVM-alustalla puolestaan `MethodHandle`-rakenne ajaa saman asian. Molempia ominaisuuksia on mahdollista käyttää myös sulkeu-

```
(define counter
  (lambda ()
    (define c 0)
    (define get-counter ...)
    (define increment-counter ...)
    (cons get-counter increment-counter)))
```

```
typedef struct counter_closure {
    counter_env_t   *env; // osoitin pakomuuttujatietueeseen
    scheme_object_t (*fun)(counter_env_t*); // osoitin funktioon
} counter_closure_t;

scheme_object_t counter() {
    counter_closure_t *get_counter_c;
    counter_closure_t *increment_counter_c;
    counter_env_t *env;

    ... // muistin varaus rakenteille

    // kaapatun muuttujan alustus, (define c 0)
    env->c = CONSTANT_NUMBER_ZERO;

    // get-counter-sulkeuman rakennus
    get_counter_c->env = env;
    get_counter_c->fun = &get_counter;

    // increment-counter-sulkeuman rakennus
    increment_counter_c->env = env;
    increment_counter_c->fun = &increment_counter;

    ... // Sulkeumien käärintä scheme_object_t-rakenteisiin
    ... // ja paluuarvona käytettävän parin rakennus.
}
```

Kuva 4.5: Sulkeumien rakentaminen C-kielellä kuvan 4.2 (sivu 28) esimerkin tapauksessa.

mien toteuttamiseen.

Tässä aliluvussa esitellään kutsuttavien olioiden käyttöä erityisesti sulkeumien toteutustekniikkana. Lisäksi esitellään tarkemmin molempien alustojen tarjoamat vaihtoehdot funktio-osoittimille. Viimeisenä luvussa esitellään uusi tulokas toteutustekniikoiden joukossa, JVM-alustan versiossa Java SE 8 esitellyn suoritusaikaisen ympäristön tarjoaman sulkeumatuen käyttö.

Oliosulkeumat

Oliokielissä funktioiden käsittelyn rajoitteita on ollut tapana kiertää käyttämällä niin sanottuja kutsuttavia olioita. Kutsuttavat oliot määritellään tyypillisesti luokkina, jotka toteuttavat tietyn, yleensä yhden metodin määrittelevän rajapinnan. Muun muassa strategia- ja komento-suunnittelumalleja [GHJV95] voi pitää esimerkkeinä kutsuttavien olioiden käytöstä oliokielissä.

Jos kutsuttava olio kaappaa lisäksi muuttujia ympäristöstään, sitä voidaan kutsua *oliosulkeumaksi* [Sco09, s. 157]. Esimerkiksi Javassa funktioita tai metodeja itsessään ei voi käsitellä arvoina, joten erityisesti ennen lambda-lausekkeiden esittelyä Java 8:ssa funktion käyttäminen argumenttina metodikutsussa edellytti kutsuttavan olion käyttöä. Usein tällaiset kutsuttavat oliot on toteutettu Javassa käyttäen anonyymeja luokkia.

Kuvassa 4.6 on esimerkki anonyymin luokan käytöstä oliosulkeumana Javassa. Javassa anonyymien luokkien ja muiden sisäluokkien kaappaamat paikalliset muuttujat on merkittävä `final`-avainsanalla, eli anonyymin luokan metodit eivät voi muuttaa kaapatun muuttujan arvoa [GJS+15, s. 201]. Alkaen Java-spesifikaation Java SE 8 -versiosta `final`-avainsana on mahdollista jättää pois, mutta tällöin Java-kääntäjä edellyttää, että muuttuja on Java-spesifikaation käyttämän termistön mukaan *tosiasiallisesti lopullinen* (engl. *effectively final*) [GJS+15, s. 86]. Tällä tarkoitetaan, että `final`-avainsanan lisääminen muuttujan esittelyyn ei saisi aiheuttaa käännösvirhettä, eli kaapattavaa muuttujaa ei muuteta esittelyn jälkeen. Tämä rajoittaa anonyymien luokkien käyttökelpoisuutta sellaisenaan esimerkiksi Scheme-toteutuksessa, jossa kaapattuihin muuttujiin on pystyttävä tekemään muutoksia.

Yksi mahdollinen ratkaisu ongelmaan on kääriä kaapattavat muuttujat viitesoluihin, mikä mahdollistaa myös muuttujan jakamisen useamman saman näkyvyysalueen sisällä luodun sulkeuman kesken. Mikäli `final`-avainsanalla merkityn muuttujan arvo on olio, on sen kenttien arvoja mahdollista muuttaa `final`-avainsanasta huolimatta. Viitesolu on yksinkertainen rakenne, joka sisältää kaapatun muuttujan kenttänä ja joka voidaan esittää Javassa esimerkiksi kuvan 4.7 `ReferenceCell`-luokan kaltaisena rakenteena.

Nimettömät luokat ovat käytännöllinen piirre Java-ohjelmoijalle, joka ei tahdo määritellä uutta nimettyä luokkaa jokaiselle oliosulkeumalle. Jos kohdekielenä käytetään

suoraan Javaa, on nimettömiä luokkia mahdollista käyttää myös kääntäjässä sulkeumatoteutuksena.

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
interface IntToIntFunction {
    int apply(int i);
}

// Ohjelman pääluokan sisällä:

public static IntToIntFunction add(final int x) {
    return new IntToIntFunction() {
        public int apply(int y) { return x + y; }
    };
}

public static void main(String[] args) {
    IntToIntFunction add10 = add(10);
    System.out.println(add10.apply(5)); // tulostaa "15"
}
```

Kuva 4.6: Anonyymin luokan käyttö sulkeumana Javassa. Yllä on Java-esimerkkikoodin `add`-metodia vastaava Scheme-funktio.

```
public class ReferenceCell<T> {
    public T value;

    public ReferenceCell(T value) {
        this.value = value;
    }
}

public static IntToIntFunction add(int x) {
    final ReferenceCell<Integer> xCapt =
        new ReferenceCell<Integer>(x);
    return new IntToIntFunction(int y) {
        public int apply(int y) { return xCapt.value + y; }
    };
}
```

Kuva 4.7: Kaapattavan muuttujan käärintä viitesoluun

Nimettömät luokat ovat Javan piirre, joka kääntyy tavukooditasolla yksinkertaisesti kääntäjän luomaksi sisäluokaksi. Tavukoodia tuottavassa kääntäjässä voidaan siis yksinkertaisesti luoda nimetty luokka jokaiselle sulkeumalle. Tämä toteutustapa on mahdollinen molemmilla virtuaalikonealustoilla. Kuvassa 4.8 on esimerkki nimetyn luokan käytöstä oliosulkeumana Javassa kuvan 4.6 tapauksessa. Sulkeuman kutsuminen tapahtuu samalla tavoin kuin kuvassa 4.6.

```
public class AddX implements IntToIntFunction {
    private int x;

    AddX(int x) { this.x = x; }

    @Override
    public int apply(int y) { return x + y; }
}

public static IntToIntFunction add(int x) {
    return new AddX(x);
}
```

Kuva 4.8: Nimetty luokka oliosulkeumana Javassa

Oliosulkeuman tapauksessa sulkeuman rungon toteuttava metodi on oliometodina samassa luokassa kuin kaapatut muuttujat. Tällöin metodille ei tarvitse tehdä sulkeumamuunnosta, koska se voi viitata sulkeuman rungon vapaisiin muuttujiin suoraan `this`-viitteensä kautta.

Funktio-osoittimet CLI-alustalla

Kuten aiemmin mainittiin, CLI tukee matalan tason osoittimia metodien natiivikoodiin [ECM12, s. 171]. Matalan tason osoittimien käyttömahdollisuudet korkean tason kielistä kuten C#:sta käsin ovat rajoitettuja, mutta niitä on mahdollista käyttää CIL-koodista käsin: osoitin metodiin voidaan ladata `ldftn`-käskyllä ja sen osoittamaa metodia voidaan kutsua `calli`-käskyllä (*call indirect*) [ECM12, s. 171]. Metodiosoittimien sijaan C#:ssa käytetään niin kutsuttuja delegaatteja, joiden avulla myös C#:n sulkeumat on toteutettu.

Delegaattien nimi viittaa siihen, että ne ovat olioita, joilla on tieto siitä, miten jotakin metodia kutsutaan [AA12, s. 119–120]. Ohjelmoija siis "delegoi" metodin kutsumisen delegaattioliolle: kun ohjelmoija tekee kutsun delegaattiolioon, delegaattiolio kutsuu metodia, joka toteuttaa varsinaisen halutun toiminnallisuuden. Käyttäjän on mahdollista määritellä omat delegaattiluokkansa, mutta CLI tarjoaa valmiina geneeriset delegaattiluokat `Func<A1,...,An,R>` ja `Action<A1,...,An>`, joista ensimmäinen esittää tyyppiä `R` olevan arvon palauttavaa funktiota ja jälkimmäinen sivuvaikutuksellista funktiota,

joka ei palauta arvoa [AA12, s. 124]. Näiden tyyppien avulla on mahdollista esittää käytännössä lähes mitä tahansa funktiota.

Kuvassa 4.9 on esimerkki geneerisen delegaattiluokan käytöstä C#:ssa. C#-koodin alla ovat tärkeimmät osat kääntäjän tuottamasta tavukoodista. `Add`-metodin tavukoodin riveiltä IL_0000–IL_0008 huomataan, että kääntäjä luo pakomuuttujatietuetta varten sisäluokan `'<Add>c__AnonStorey0'`, jonka ilmentymän `x`-nimiseen kenttään kaapattavan muuttujan `x` arvo talletetaan. Varsinainen luokkamäärittely on kuvassa 4.10. Se sisältää kaapattavalle muuttujalle varatun kentän lisäksi sulkeumafunktion rungon toteuttavan metodin, jonka nimi on `'<>m__0'`[4].

Kuten tavukoodista nähdään, kulissien takana delegaattiluokat käyttävät hyväkseen matalan tason osoittimia metodeihin. CLI:n geneerisiä delegaattiluokkia käytetään sulkeumien toteuttamiseen käytännössä siten, että delegaattiluokan ilmentymä toimii kääreenä metodiosoittimelle ja sulkeuman ympäristötietueen virkaa toimittavan luokan ilmentymälle. Kuvassa 4.9 `Add`-metodin tavukoodissa raa'an funktio-osoittimen lataus `ldftn`-käskyllä tapahtuu rivillä IL_000e ja varsinaisen delegaattiolion luonti geneerisen delegaattityypin konstruktoria kutsumalla rivillä IL_0014. Ajatus on siis samankaltainen kuin luvussa 4.1 esitellyssä monien C:tä kohdekielenä käyttävien kääntäjien käyttämässä sulkeumaesityksessä.

Funktio-osoittimet JVM-alustalla

JVM-alustan `MethodHandle` on CLI:n delegaattiluokkien tavoin raakoja osoittimia turvallisempi esitys metodiosoittimelle. `MethodHandle` kuitenkin painottaa delegaattiluokkia vähemmän tyyppiturvallisuutta: `MethodHandle`-tyypillä ei ole aliluokkia, vaan kaikki metodiosoittimet ovat luokan `MethodHandle` ilmentymiä. Tällaisessa muodossa esitettävää funktioarvoa on siis mahdollista kutsua virheellisellä määrällä argumentteja tai virheellisen tyyppisillä argumenteilla. Tällöin kutsuyritys aiheuttaa poikkeuksen suoritusaikana. Tällainen löysempi tyypitys voi olla esimerkiksi joidenkin dynaamisten kielten toteuttajille toivottu ominaisuus.

`MethodHandle` tukee lisäksi `bindTo`-metodia, jonka avulla osa `MethodHandle`-olion osoittaman metodin parametreista voidaan sitoa argumenttiarvoihin ennen varsinaista kutsua. Tätä argumenttien sidontaa kutsutaan funktion *osittaiseksi soveltamiseksi* (engl. *partial application*) [Sco09, s. 531].

Kun sulkeuman runkoa esittävä metodi lambda-nostetaan luokan ylätasolle ja sille tehdään sulkeumamuunnos siten, että sen parametrilistan alkuun lisätään kaapattuja muuttujia vastaavat parametrit, kaapatut muuttujat voidaan sitoa `MethodHandle`-olioon

[4]Kääntäjän luoma oletuskonstruktori on poistettu esimerkistä koodin tärkeämpien kohtien korostamiseksi.

```
public class SimpleClosure {
    private static Func<int, int> Add(int x) {
        return y => x + y;
    }

    public static void Main() {
        Func<int, int> add10 = Add(10);
        Console.WriteLine(add10(5));
    }
}
```

```
.method private static hidebysig
       default class [mscorlib]System.Func`2<int32, int32> Add (int32 x)
       cil managed
{
    .maxstack 2
    .locals init (
            class SimpleClosure/'<Add>c__AnonStorey0' V_0)
    IL_0000:  newobj instance void class
              SimpleClosure/'<Add>c__AnonStorey0'::'.ctor'()
    IL_0005:  stloc.0
    IL_0006:  ldloc.0
    IL_0007:  ldarg.0
    IL_0008:  stfld int32 SimpleClosure/'<Add>c__AnonStorey0'::x
    IL_000d:  ldloc.0
    IL_000e:  ldftn instance int32 class
              SimpleClosure/'<Add>c__AnonStorey0'::'<>m__0'(int32)
    IL_0014:  newobj instance void class [mscorlib]
              System.Func`2<int32, int32>::'.ctor'(object, native int)
    IL_0019:  ret
}

.method public static hidebysig
       default void Main ()  cil managed
{
    // ...
    IL_0002:  call class [mscorlib]System.Func`2<int32,int32>
               class SimpleClosure::Add(int32)
    // ...
    IL_000a:  callvirt instance !1 class [mscorlib]
               System.Func`2<int32, int32>::Invoke(!0)
    // ...
}
```

Kuva 4.9: Delegaattiluokan käyttö sulkeumana C#:ssa: kääntäjän tuottama tavukoodi metodille `Add` ja osalle `Main`-metodista.

```
.class nested private auto ansi sealed beforefieldinit '<Add>c__AnonStorey0'
      extends [mscorlib]System.Object
{
  // Kenttä kaapattavalle muuttujalle
  .field  assembly  int32 x

  // Sulkeuman rungon toteuttava metodi.
  .method assembly hidebysig
         instance default int32 '<>m__0' (int32 y)  cil managed
  {
      .maxstack 8
      IL_0000:  ldarg.0
      IL_0001:  ldfld int32 SimpleClosure/'<Add>c__AnonStorey0'::x
      IL_0006:  ldarg.1
      IL_0007:  add
      IL_0008:  ret
  }
}
```

Kuva 4.10: C#-kääntäjän luoma esitys pakomuuttujatietueelle ja sulkeuman rungon toteuttavalle metodille kuvan 4.9 tapauksessa.

`bindTo`-metodilla. Tällöin `MethodHandle` voi suoraan toimia esityksenä sulkeumalle.

Kuvassa 4.11 on esitetty sulkeuman luonti `MethodHandle`-oliona Javassa. `MethodHandle`, lienee suunnattu enemmän kielten toteuttajien kuin Javaa käyttävien sovellusohjelmoijien tarpeisiin, joten sen käyttäminen Javasta käsin ei ole aivan yhtä yksinkertaista kuin delegaattiluokkien käyttö C#:ssa. Metodiosoitin luodaan `MethodHandles.lookup()`-kutsun palauttaman tehdasolion avulla. Kun luodaan esimerkin tapaan metodikahva, joka osoittaa staattiseen metodiin, tarvitaan parametreina metodin sisältävä luokka, metodin nimi sekä metodin tyyppi `MethodType`-oliona.

JVM-alustan suoritusaikaisen ympäristön sulkeumatuki

Javassa lambda-lausekkeet olisi ollut täysin mahdollista toteuttaa yksinkertaisesti syntaksisokerina jo pitkään anonyymien funktioiden korvikkeena käytetyille anonyymeille luokille. Odotuksista poiketen Java 8:n lambda-toteutus on kuitenkin hieman monimutkaisempi. Toteutus hyödyntää JVM-alustan Java SE 7 ja 8 -versioissa esiteltyjä uusia ominaisuuksia — erityisesti dynaamisesti tyypitettyjen kielten tarpeisiin tarkoitettua `invokedynamic`-käskyä sekä funktioarvojen luontiin kehitettyjä `LambdaMetafactory`-luokan metodeita [Goe12]. Tällä hetkellä ominaisuus ei tiettävästi ole vielä käytössä muissa kielitoteutuksissa, joten tarkastelemme ominaisuuden toimintaa ja toteutusvalinnan perusteluja Javan näkökulmasta.

MethodHandleExample.java

```
import java.lang.invoke.*;

public class MethodHandleExample {
    // Sulkeuman rungon toteuttava metodi
    public static Integer add(Integer x, Integer y) {
        return x + y;
    }

    public static MethodHandle addX(int x)
        throws NoSuchMethodException, IllegalAccessException {
        MethodHandles.Lookup lookup = MethodHandles.lookup();

        // Etsittävän metodin tyypin määrittely.
        MethodType mt = MethodType.methodType(
            Integer.class, Integer.class, Integer.class);

        // Metodiosoittimen luonti: etsii "add"-nimisen
        // staattisen metodin ja luo siihen osoittavan
        // MethodHandle-olion.
        MethodHandle mh = lookup.findStatic(
            MethodHandleExample.class, "add", mt);

        // Kaapattavien muuttujien sidonta eli
        // funktion osittainen soveltaminen
        mh = mh.bindTo(x);
        return mh;
    }

    public static void main(String[] args) throws Throwable {
        MethodHandle add10 = addX(10);
        System.out.println(add10.invoke(5));
    }
}
```

Kuva 4.11: `MethodHandle`-olion käyttö sulkeumana Javassa

`Invokedynamic`-käsky esiteltiin lyhyesti luvussa 3.3. `Invokedynamic`-käskyn käyttöpaikalle käännösaikana määritetty bootstrap-metodi luo ennen `invokedynamic`-käskyn ensimmäistä kutsukertaa kutsupaikkaolion, joka sidotaan käskyn yhteyteen. Tämä kutsupaikkaolio määrittää metodin, jota saman `invokedynamic`-käskyn suorituskerrat jatkossa kutsuvat. Käskyn pääasiallinen tarkoitus on tukea dynaamisten kielten toteutusta, mutta Javan sulkeumissa se on valjastettu hiukan toisenlaiseen käyttöön.

JVM määrittelee *funktionaalisen rajapinnan* (engl. *functional interface*) käsitteen, jolla tarkoitetaan rajapintaa, joka määrittelee yhden metodin [Ora16a]. Ajatus on siis samankaltainen kuin oliosulkeumien yhteydessä käytettävillä rajapinnoilla. Käyttäjä voi määritellä omia funktionaalisia rajapintojaan käyttämällä `@FunctionalInterface`-annotaatiota. CLI:n tavoin JVM määrittelee myös useita yleisiä käyttötapauksia kattavia sisäänrakennettuja funktionaalisia rajapintoja pakkauksessa `java.util.function` [Ora16e]. Esimerkiksi funktionaalinen rajapinta `Function<T,R>` esittää yhden `T`-tyyppisen parametrin ottavaa funktiota, jonka paluuarvo on tyyppiä `R`.

Java 8:ssa suoritusaikainen ympäristö luo automaattisesti lambda-lausekkeelle esityksen, joka mukautuu haluttuun funktionaaliseen rajapintaan. Esimerkiksi kuvan 4.12 tapauksessa suoritusaikainen ympäristö luo `Function<Integer,Integer>`-rajapintaan mukautuvan esityksen `add`-metodin palauttamalle sulkeumalle.

Sulkeumia käännettäessä Java-kääntäjä suorittaa lambda-noston, jossa lambda-lausekkeen runko nostetaan tapauksesta riippuen joko staattiseksi metodiksi tai oliometodiksi sen ympäröivän luokan sisälle [Goe12]. Lisäksi mahdolliset kaapattavat muuttujat lisätään ylimääräisiksi parametreiksi metodin parametrilistan alkuun, eli sulkeumafunktiolle tehdään sulkeumamuunnos. Kuvan 4.12 esimerkissä sulkeuman runko kääntyy yksityiseksi staattiseksi metodiksi `lambda$add$0`, joka ottaa `int`- ja `Integer`-tyyppiset parametrit, joista ensimmäinen on kaapattu muuttuja `x`. Paikkaan, jossa alkuperäinen käännettävä koodi luo sulkeuman, kääntäjä luo `invokedynamic`-kutsun. Kuvan 4.12 tapauksessa `invokedynamic`-kutsu luodaan siis `add`-metodin runkoon.

Kuten edellä mainittiin, `invokedynamic`-käsky vaatii bootstrap-metodin, jota kutsutaan ennen kuin käsky suoritetaan ensimmäisen kerran. Sulkeumaa luotaessa `invokedynamic`-käskyn bootstrap-metodina käytetään jotakin standardin määrittelemässä `LambdaMetafactory`-luokassa määritellyistä *lambda-metatehdasmetodeista* [Goe12]. Metatehdasmetodi luo kutsupaikkaolion, joka sidotaan suoritusaikana tunnettuun metodiin, joka osaa luoda tarvittavan sulkeumaolion. Sulkeumia luovaa `invokedynamic`-kutsupaikkaa kutsutaan *lambdatehtaaksi* (*lambda factory*) [Goe12], joten oletettavasti lambda-metatehdasmetodien nimi viittaa siihen, että ne ovat metodeja, jotka luovat lambdatehtaita.

Metatehdasmetodien toteutus on suoritusaikaisen ympäristön vastuulla, joten suoritusaikainen ympäristö voi valita lambdatehtaaksi minkä tahansa itse toteuttamansa

```
static Function<Integer, Integer> add(int x) {
    return (y) -> { return x + y; };
}
```

```
static java.util.function.Function<java.lang.Integer, java.lang.Integer>
    add(int);
descriptor: (I)Ljava/util/function/Function;
flags: ACC_STATIC
Code:
  stack=1, locals=1, args_size=1
     0: iload_0
     1: invokedynamic #2,  0 // InvokeDynamic #0:apply:
                             //     (I)Ljava/util/function/Function;
     6: areturn

// Sulkeuman rungon toteuttava metodi
private static java.lang.Integer lambda$add$0(int, java.lang.Integer);
  descriptor: (ILjava/lang/Integer;)Ljava/lang/Integer;
  flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
  Code:
    stack=2, locals=2, args_size=2
       0: iload_0
       1: aload_1
       2: invokevirtual #8 // Method java/lang/Integer.intValue:()I
       5: iadd
       6: invokestatic  #5 // Method java/lang/Integer.valueOf:
                           //     (I)Ljava/lang/Integer;
       9: areturn
```

Kuva 4.12: Sulkeuman luonti lambda-lausekkeella Javassa (yllä) ja Java-tavukoodissa (alla)

metodin, joka tuottaa tuloksena oikeanlaisen olion. Käytännössä tämä tarkoittaa sitä, että standardoitua metatehdasmetodia käytettäessä sulkeuman suoritusaikaisen esitysmuoto on täysin suoritusaikaisen ympäristön valittavissa.

Metatehdasmetodi saa parametreina muun muassa `MethodHandle`-tyyppisen osoittimen sulkeuman rungon toteuttavaan metodiin sekä funktionaalisen rajapinnan tyypin, johon palautetun arvon tulee mukautua. Toisin kuin `MethodHandle`-olioita muutoin käytettäessä, lambda-metatehtaita käytettäessä tarvittava `MethodHandle`-olio luodaan automaattisesti suoritusaikana. `Invokedynamic`-käskyn toteutus huolehtii tarvittavien parametrien antamisesta metatehdasmetodille.

Kuvassa 4.13 on esimerkki `invokedynamic`-käskyn bootstrap-metodilleen antamista parametreista kuvan 4.12 tapauksessa. Nämä parametrit ovat tavukoodissa erillisessä BootstrapMethods-osiossa ja sisältävät esimerkiksi sulkeuman rungon toteuttavan

```
BootstrapMethods:
  0: #27 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
         (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;
          Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;
          Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)
          Ljava/lang/invoke/CallSite;
    Method arguments:
      #28 (Ljava/lang/Object;)Ljava/lang/Object;
      #29 invokestatic SimpleClosure.lambda$add$0:
             (ILjava/lang/Integer;)Ljava/lang/Integer;
      #30 (Ljava/lang/Integer;)Ljava/lang/Integer;
```

Kuva 4.13: Lambda-metatehtaan parametrit

metodin nimen ja tyypin sekä toteutettavan sulkeumafunktion tyypin — tässä tapauksessa `(Ljava/lang/Integer;)Ljava/lang/Integer;`[5]. Varsinaiselle funktio-olion luovalle metodille puolestaan annetaan parametreina muun muassa sulkeuman ympäristöstä kaapatut muuttujat. Tämä on nähtävissä kuvassa 4.12 `add`-metodin tavukoodin rivillä 0, jolla funktio lataa ensimmäisen parametrinsa pinoon ennen `invokedynamic`-kutsua.

Oraclen Java-arkkitehdin Brian Goetzin mukaan Javan käyttämän toteutustavan valinnan taustalla olivat suorituskykyyn ja myöhempien optimointien mahdollistamiseen liittyvät syyt [Goe12]. Koska sulkeumaolion esitys sidotaan vasta suoritusaikana, Javan suoritusaikaisten ympäristöjen toteuttajat voivat vapaasti vaikuttaa sulkeumaesityksen yksityiskohtiin ja sulkeumaesitykseen voidaan tehdä muutoksia suoritusaikaisen ympäristön päivitysten yhteydessä.

Javan lambda-metatehtailla luotuja sulkeumaesityksiä kuitenkin koskevat samat `final`-avainsanan käyttöä koskevat rajoitteet kuin Javan sisäluokkiakin, eli kaapattuihin muuttujiin ei saa tehdä muutoksia. Muuttuvan tilan kaappaaminen edellyttää siis myös lambda-metatehtaita käytettäessä esimerkiksi viitesolujen käyttöä.

4.3 Häntäkutsujen optimointi

Koska useimmissa imperatiivisissa kielissä ja oliokielissä on idiomaattista käyttää häntärekursion sijasta silmukoita, häntäkutsuja ei tyypillisesti optimoida, ja häntärekursion käyttö johtaa usein tilan loppumiseen aktivaatiotietuepinosta. Koska häntärekursion käyttö on yleistä funktionaalisissa kielissä, puhutaan funktionaalisten kielten yhteydessä usein juuri häntärekursion optimoinnista, jolla tarkoitetaan pinotilan uudelleenkäyttöä häntärekursiivisten kutsujen yhteydessä.

[5] Java-tavukoodissa metodien tyypit ilmaistaan siten, että metodin parametrityypit ovat sulkeissa ja paluutyyppi sulkeiden jälkeen. Tyyppikuvaaja (engl. *descriptor*) [LYBB15, s. 75] muotoa `(A)B` siis merkitsee metodia, joka ottaa parametrinaan `A`-tyyppisen arvon ja palauttaa `B`-tyyppisen arvon.

Yksinkertaiset häntärekursiotapaukset eli “funktionaaliset silmukat” ovat kääntäjän toteuttajan kannalta helppoja: paikallinen funktio voidaan melko suoraviivaisesti muuntaa imperatiiviseksi silmukaksi. Tällöin häntärekursiivinen kutsu muuntuu yksinkertaisesti hyppykäskyksi proseduurin alkuun tai muuhun paikkaan, jossa häntärekursiivisen funktion loppuehdon tarkistus sijaitsee.

Kuvassa 4.14 on esimerkki Scala-kääntäjän käyttämästä Java-tavukoodikäännöksestä kuvan 2.5 (sivu 12) esimerkkiä vastaavalle häntärekursiiviselle apufunktiolle Scalassa. Käännetty Scala-funktio on esitetty kuvan yläpuoliskolla tavukoodin lukemisen helpottamiseksi. Esimerkiksi Clojure- ja Kotlin-kääntäjät tuottavat hyvin samankaltaista koodia, mutta Scala-kääntäjän tuottama tavukoodi muistuttaa järjestykseltään eniten alkuperäistä koodia ja on siten helpompi lukea.

Tavukoodissa rekursion loppuehdon tarkistus on sijoitettu metodin alkuun. Rivit 5–6 toteuttavat alkuperäisen proseduurin then-haaran, ja rivit 7–15 toteuttavat else-haaran. Metodin lopuksi else-haarassa lasketut uuden kierroksen parametrit talletetaan takaisin metodin parametripaikkoihin 1 ja 2 ja hypätään takaisin alkuun `goto`-käskyllä.

Koska häntärekursion optimointi on suoraviivaista, yksinkertaisia paikallisia silmukoita mielenkiintoisempia ovatkin esimerkiksi keskenään rekursiiviset funktiot ja yleistetyt häntäkutsut, joita esiintyy runsaasti esimerkiksi CPS-muotoisessa koodissa. Tässä aliluvussa keskitytään yleistettyjen häntäkutsujen optimointitekniikoihin.

Aliluku esittelee ensin perinteisillä alustoilla käytettyjä optimointitekniikoita. Erityisesti kiinnitetään huomiota trampoliinitekniikkaan, joka on käyttökelpoinen myös virtuaalikoneympäristössä. Lopuksi esitellään alustojen tarjoama tuki häntäkutsujen optimoinnille.

Yleistettyjen häntäkutsujen optimointitekniikat

Perinteisesti käännettäessä funktionaalisia kieliä alustoille, jotka sallivat kutsupinon manipuloinnin ja funktio-osoittimet, häntäkutsujen toteutus on suhteellisen yksinkertaista. Jos funktio `f` kutsuu funktiota `g` häntäpositiossa, voidaan ennen häntäkutsun tekemistä poistaa pinosta funktion `f` aktivaatiotietue ja antaa funktion `f` paluuosoite suoraan paluuosoitteeksi funktiolle `g` [App98, s. 329].

Ennen virtuaalikoneiden nousua suosioon C oli suosittu kohdekieli funktionaalisille kielille vastaavista syistä kuin virtuaalikoneet tänään: valmiiksi tehokkaat C-kääntäjät tekivät alustalta toiselle siirtämisen helpoksi ja yhteiskäyttörajapintojen ansiosta C-kirjastoja saattoi käyttää funktionaalisista kielistä käsin (esim. [Que03, luku 10] [TLA92, Ser15, FMRW97]). Koska C itsessään ei tue häntäkutsuja eikä anna ohjelmoijan hallita kutsupinon käyttöä, on C:tä silmällä pitäen kehitetty useita mahdollisia optimointitekniikoita häntäkutsuille. Monet näistä ovat pienin muokkauksin käyttökelpoisia myös

```
def fact(n: Int): Int = {
  def fact_helper(n: Int, acc: Int): Int = {
    if (n == 0) acc
    else fact_helper(n - 1, acc * n)
  }
  fact_helper(n, 1)
}
```

```
private final int fact_helper$1(int, int);
  descriptor: (II)I
  flags: ACC_PRIVATE, ACC_FINAL
  Code:
    stack=3, locals=3, args_size=3
       // 'n == 0' -tarkistus
       0: iload_1
       1: iconst_0
       2: if_icmpne     7

       // Then-haara: acc-muuttujan lataus ja palautus
       5: iload_2
       6: ireturn

       // Else-haara: seuraavan kierroksen parametrien laskenta
       7: iload_1
       8: iconst_1
       9: isub
      10: iload_2
      11: iload_1
      12: imul

      // Muuttujien talletus ja hyppy takaisin 'n == 0' -tarkistukseen
      13: istore_2
      14: istore_1
      15: goto          0
```

Kuva 4.14: Häntärekursion optimointi Scala-kääntäjän tuottamana Java-tavukoodina

virtuaalikonealustoilla.

Yksi mahdollinen tekniikka on koko ohjelman koodin sijoittaminen yhteen funktioon ja funktiokutsujen korvaaminen kokonaan hyppykäskyillä tai suoritettavan funktion koodin valinta `switch`-lauseella [SO01]. Koska JVM ja CLI sallivat hypyt metodien sisällä, tätä tekniikkaa olisi periaatteessa mahdollista käyttää näilläkin alustoilla.

Käytännössä ainakin JVM kuitenkin tekee tekniikan käyttämisestä isompien ohjelmien kääntämiseen hankalaa, koska metodien koko on rajoitettu 64 kilotavuun [SO01], mikä rajoittaa tällä tyylillä käännettävän ohjelman kokoa. Lisäksi osa JIT-kääntäjistä, mukaan lukien Oraclen oma HotSpot, ei käännä metodeita, joiden rungot ylittävät tietyn kokorajan [SS02]. Tämä voi heikentää merkittävästi suorituskykyä.

Toinen C-kieltä kohdekielenä käyttävistä kääntäjistä tunnettu toteutustekniikka tunnetaan trampoliinimenetelmänä [Bak95]. Tässä tekniikassa häntäkutsun tekevä funktio kääritään ulkoiseen funktioon, joka toimii "trampoliinina". Kun sisäinen funktio tahtoo kutsua toista funktiota häntäpositiossa, se palauttaa trampoliinifunktiolle viestin, joka sisältää kutsuparametrien arvot sekä tiedon kutsuttavasta funktiosta: esimerkiksi suoraan kutsuttavissa olevan sulkeuman tai muun tunnisteen, jonka perusteella kutsuttava funktio voidaan löytää. Vastuu häntäkutsun suorittamisesta siirtyy siis trampoliinifunktiolle. Koska sisempi funktio palaa ennen "häntäkutsun" suorittamista, sen aktivaatiotietue poistuu automaattisesti pinosta, eikä pino pääse kasvamaan.

Tunnisteena voi periaatteessa toimia mikä tahansa arvo, jonka trampoliinikoodi pystyy yhdistämään kutsuttavaan funktioon. Yksinkertaisimmillaan kutsuttava funktio voitaisiin valita vaikkapa `switch`-lauseella, jolloin tunnisteena voisi toimia `switch`-lauseessa esiintyvä valintavakio. Tätä seuraavan kutsuttavan funktion tunnisteena toimivaa paluuarvoa kutsutaan joissakin lähteissä *kontinuaatioksi* (engl. *continuation*) [SO01].

Trampoliinien toiminta

Trampoliinifunktion runko on käytännössä silmukka, joka kutsuu jokaisella iteraatiolla uutta kontinuaatiota, joka palauttaa sille seuraavan kontinuaation. Silmukan suoritus jatkuu, kunnes trampoliinifunktio saa joltakin kontinuaatiolta paluuarvona lopullisen arvon, joka ei enää ole kontinuaatio. Mikäli kontinuaation esityksenä käytetään siis esimerkiksi sulkeumaa, paluuarvo täytyy jotenkin merkitä kontinuaatioksi, koska sulkeuma voi olla myös validi lopullinen paluuarvo. Merkintä voidaan tehdä esimerkiksi käärimällä paluuarvo tietueeseen ja lisäämällä tietueeseen totuusarvokenttä, joka kertoo onko kyseessä kontinuaatio vai lopullinen paluuarvo.

Kuvassa 4.15 on esimerkki mahdollisesta trampoliinifunktiosta Javaa käyttävässä Scheme-toteutuksessa. Esimerkkitapauksessa trampoliinifunktion kutsuman funktion paluuarvoa esitetään `ReturnValue`-luokan ilmentymällä. `ReturnValue`-luokka sisältää

edellä kuvatun totuusarvokentän `isContinuation` sekä `SchemeValue`-tyyppisen paluuarvon. Esimerkki olettaa, että tyyppi `SchemeValue` määrittelee funktion `apply`, joka suorittaa kutsun sulkeumafunktioon, mikäli `SchemeValue`-olio esittää sulkeumaa, ja heittää muussa tapauksessa poikkeuksen. Lisäksi oletetaan, että kontinuaatiofunktion vaatimat parametrit on sidottu ennalta osittain soveltamalla, jolloin trampoliinifunktio voi suorittaa `apply`-kutsun ilman parametreja.

```
class ReturnValue {
    public final boolean isContinuation;
    public final SchemeValue value;
}

public static SchemeValue trampoline(ReturnValue f) {
    while (f.isContinuation) {
        f = f.value.apply();
    }

    return f.value;
}
```

Kuva 4.15: Mahdollinen trampoliinifunktion toteutus

Tyyppiesimerkki keskinäisestä rekursiosta on kuvan 4.16 aiemmin luvussa 2.3 esitelty Scheme-koodi parillisten ja parittomien lukujen tunnistamiseen. Esimerkissä funktiot `odd?` ja `even?` kutsuvat toisiaan epäsuorasti rekursiivisilla häntäkutsuilla.

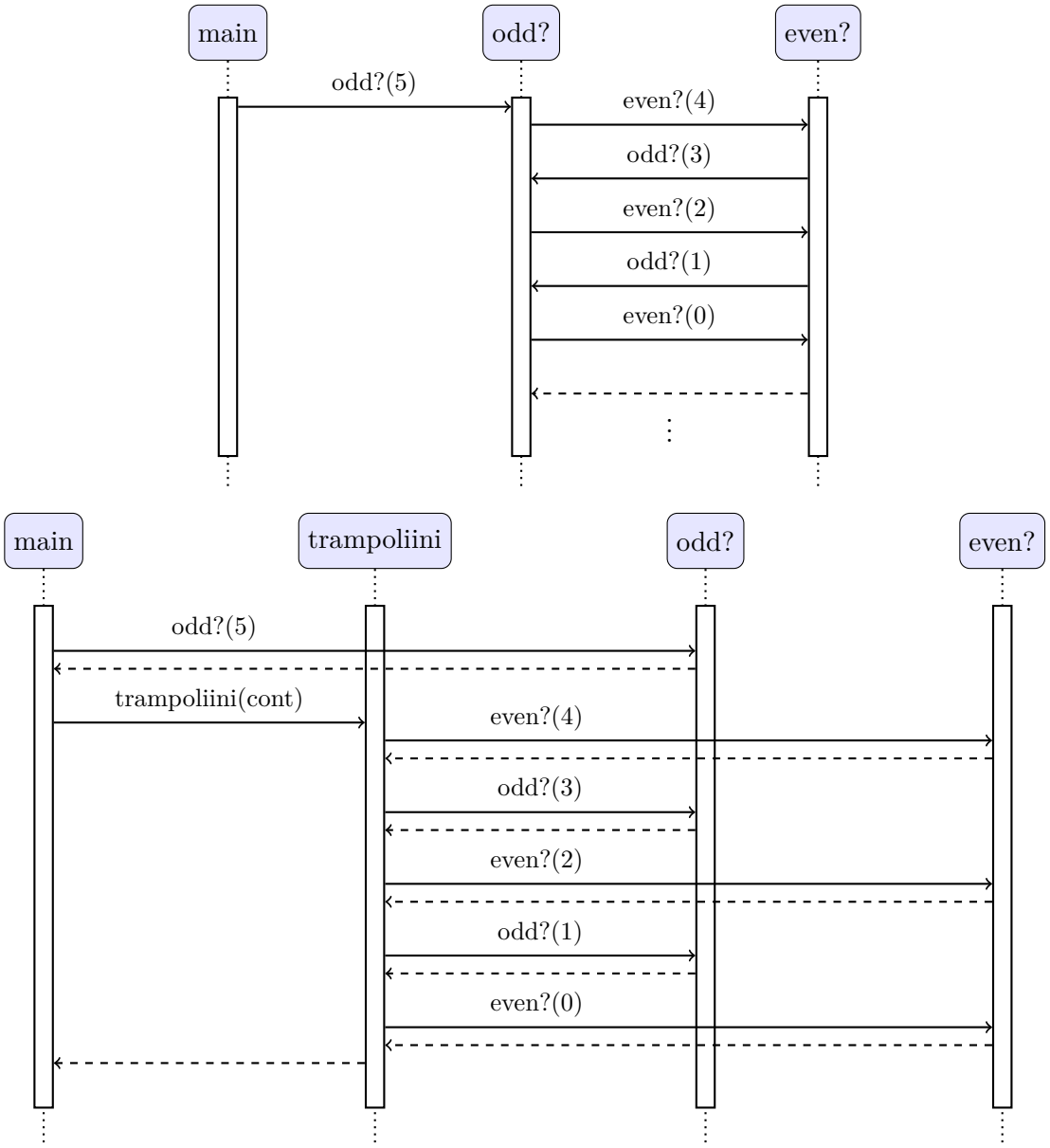
```
(define odd?
  (lambda (x)
    (if (zero? x)
        #f
        (even? (- x 1)))))

(define even?
  (lambda (x)
    (if (zero? x)
        #t
        (odd? (- x 1)))))
```

Kuva 4.16: Parittomien ja parillisten lukujen tunnistus keskinäisellä häntärekursiolla Schemessä

Kuva 4.17 havainnollistaa kontrollin kulkua ja pinon käyttäytymistä `odd?`- ja `even?`-funktioiden käytössä ilman häntäkutsuoptimointia ja trampoliinitekniikan kanssa. Kaaviossa funktiokutsut on merkitty yhtenäisillä nuolilla ja paluut funktiokutsuista katkovii-

vanuolilla. Lisäksi pinon toiminnan hahmottamiseksi voi ajatella, että yhtenäinen nuoli kasvattaa pinon kokoa yhdellä aktivaatiotietueella ja katkoviivanuoli poistaa pinosta yhden aktivaatiotietueen.



Kuva 4.17: Kontrollin kulku keskinäisessä häntärekursiossa trampoliinitekniikan kanssa (alempi kuva) ja ilman häntäkutsuoptimointia (ylempi kuva)

Ilman häntäkutsuoptimointia pino vain jatkaa kasvuaan siihen asti, että kaikki tarvittavat rekursiiviset kutsut on suoritettu, minkä jälkeen aktivaatiotietueet poistuvat pinosta yksitellen. Alemmassa kuvassa trampoliinitekniikka on ajateltu toteutetuksi

kuvan 4.15 kaltaisen apufunktion avulla. Tällöin `odd?`-funktiota kutsutaan ensin kerran, minkä jälkeen saatu kontinuaatio annetaan trampoliinille, joka kutsuu kontinuaatioita silmukassa, kunnes se saa lopullisen paluuarvon. Tällaista trampoliinia käytettäessä jokainen kutsu palaa välittömästi, jolloin pino pysyy vakiokokoisena.

Trampoliinitekniikan parannukset

Trampoliinimenetelmä soveltuu yleistettyjen häntäkutsujen toteuttamiseen käytännössä millä tahansa korkean tason kielellä. Monet funktionaalisten kielten toteuttajat virtuaalikonealustoilla kuitenkin välttelevät menetelmää johtuen sen oletetuista kustannuksista suorituskyvylle [SS02, SO01]. Bakerin [Bak95] mukaan trampoliinin kautta tehty kutsu C-kielellä vie 2–3 kertaa enemmän aikaa kuin tavallinen C-proseduurikutsu. Trampoliinitekniikan kustannuksista virtuaalikonealustoilla ei kuitenkaan näytä olevan viimeaikaisia mittauksia.

Baker [Bak95] kuvailee CPS-muodon käyttöön perustuvan tekniikan, jonka hän mainitsee olevan Appelin julkaisematon ehdotus. Schinz ja Odersky [SO01] kutsuvat menetelmää Bakerin artikkelin otsikon mukaan leikkisästi nimellä "Cheney on the M.T.A."[6].

Bakerin kuvailemassa menetelmässä CPS-muotoon muunnettu koodi käännetään tavalliseen tapaan ilman häntäkutsujen optimointia, mutta jokaisen uuden kutsun yhteydessä tarkistetaan pinon koko. Kun pino täyttyy, käynnistetään roskienkerääjä, joka siivoaa pinosta turhat aktivaatiotietueet. Tämän jälkeen hypätään takaisin trampoliiniin samaan tapaan kuin normaalissa trampoliinimenetelmässä. Tällöin ohjelma voidaan suorittaa rajoitetussa pinotilassa, mutta suorituskyky kärsii trampoliinin käytön kustannuksista huomattavasti harvemmin kuin tavallisessa trampoliinimenetelmässä.

Koska JVM ja CLI eivät salli pääsyä pinon käsittelyyn, Bakerin kuvailema menetelmä ei ole sellaisenaan käyttökelpoinen virtuaalikonealustoilla. Schinz ja Odersky [SO01] ovat kehittäneet Funnel-kääntäjäänsä varten menetelmästä muunnelman, jossa funktiot kuljettavat mukanaan pinossa olevien häntäkutsujen määrää, ja häntäkutsujen yhteyteen lisättävä trampoliinikoodi tarkistaa pinon koon sijasta tämän parametrin arvoa ennalta määriteltyä rajaa vasten. Kun raja ylittyy, funktio palauttaa kontinuaation trampoliinille `return`-ketjun tai poikkeuksen avulla.

Bakerin menetelmästä poiketen Funnel-kääntäjä ei kuitenkaan käytä CPS-muotoa, koska Schinzin ja Oderskyn mittausten mukaan CPS-muodon käyttö hidasti benchmark-

[6]Nimi viittaa muun muassa Kingston Trio -yhtyeen äänittämään poliittiseen protestilauluun *Charlie on the M.T.A*, joka protestoi Bostonin metrojärjestelmän ulospääsymaksuja. Kertosäkeessä lauletaan, että rahattomana Bostonin metrojärjestelmään jatkuvalle kiertoajelulle jäänyt Charlie on "mies, joka ei koskaan palaa", aivan kuin häntäkutsut CPS-muodossa. Nimi Cheney viittaa erääseen roskienkeruualgoritmiin [Che70].

tarkoituksessa käytetyn Fibonacci-funktion suoritusta jopa 20–25 kertaa verrattuna naiiviin toteutukseen [SO01].

Alustojen tarjoama häntäkutsutuki

JVM-spesifikaatio ei ainakaan toistaiseksi tarjoa lainkaan räätälöityä tukea häntäkutsujen optimointiin — itse asiassa koko spesifikaatio ei edes mainitse häntäkutsun käsitettä [LYBB15]. JVM-alustalla ainoa mahdollisuus tällä hetkellä näyttää siis olevan tukeutuminen edellä kuvattuihin C-kohdekielisistä kääntäjistä periytyviin tekniikoihin.

CLI-spesifikaatio määrittelee kutsukäskyille `call`, `calli` ja `callvirt` etuliitteen `tail.`, joka opastaa suoritusaikaista ympäristöä poistamaan kutsuvan metodin aktivaatiotietueen pinosta ennen kutsua [ECM12, s. 320]. Ohjelman suoritusta edeltävä tavukoodin automaattinen tarkastus edellyttää, että `tail.`-etuliitteellistä kutsukäskyä seuraa välittömästi paluukäsky `ret`, mikä varmistaa, että häntäkutsun suorittamisen jälkeen ei enää tarvita kutsuvan metodin aktivaatiotietuetta.

Kuvassa 4.18 on esimerkki `tail.`-etuliitteen käytöstä keskinäisessä häntärekursiossa. Esimerkkikoodi on tuotettu F#-kääntäjällä ja alkuperäinen F#-koodi on esitetty tavukoodin yläpuolella. F#-koodi on suoraviivainen käännös kuvan 4.16 (sivu 46) Scheme-esimerkkikoodista. Häntäkutsun toteuttavat käskyt `tail.`, `call` ja `ret` on korostettu molempien funktioiden tavukoodissa.

Ilmeisesti joissakin tilanteissa on myös mahdollista, että CLI-toteutus osaa itse ennen JIT-käännöstä natiivikoodiksi lisätä tavukoodiin `tail.`-etuliitteen sellaisiin paikkoihin, joissa sen käyttö on mahdollista [ECM12, s. 320]. Kielen toteuttaja ei kuitenkaan voi luottaa automaattiseen käsittelyyn, jota standardi ei takaa.

CLI-spesifikaatio kuitenkin sallii toteutusten käyttää tietyissä tilanteissa tavallista metodikutsua häntäkutsun sijaan huolimatta `tail.`-etuliitteestä. Erityisesti CLI-toteutuksen ei ole koskaan pakko noudattaa `tail. calli` tai `tail. callvirt` -käskyä, eli häntäkutsukäskyjä, joiden kohde ei ole tiedossa käännösaikana [ECM12, s. 320]. Lisäksi standardi edellyttää, että kutsuva metodi ja kutsuttava metodi ovat samassa CLI:n assembly-yksikössä [ECM12, s. 320], eli käytännössä samassa `dll`- tai `exe`-tiedostossa. Toteutuksen on kuitenkin sallittua optimoida häntäkutsut myös `calli`- ja `callvirt`-käskyjen yhteydessä tai assembly-yksiköiden välillä, ja ainakin Microsoftin .NET-toteutus näyttääkin tekevän näin useimmissa tapauksissa.

JVM-alustalle on monesti ehdotettu vastaavaa metodikutsun häntäkutsuksi merkitsevää etuliitettä [Ros07, Sch09]. Virallisesti ominaisuuden julkaisusta ei ole vielä tehty lupauksia, mutta OpenJDK:n alaisen Da Vinci Machine -projektin wiki-sivuilta löytyy viitteitä ominaisuuden prototyyppitoteutuksen testauksesta [Ros13]. Da Vinci Machine -projektin tarkoituksena ilmoitetaan olevan erityisesti dynaamisten kielten ja muiden

```
let rec isOdd (n: int): bool =
    if n = 0 then
        false
    else
        isEven(n - 1)
and isEven (n: int): bool =
    if n = 0 then
        true
    else
        isOdd(n - 1)
```

```
.method public static default bool isOdd (int32 n)
    cil managed
{
    .maxstack 8
    IL_0000:  ldarg.0
    IL_0001:  brtrue.s IL_0005
    IL_0003:  br.s IL_0007
    IL_0005:  br.s IL_0009
    IL_0007:  ldc.i4.0
    IL_0008:  ret
    IL_0009:  ldarg.0
    IL_000a:  ldc.i4.1
    IL_000b:  sub
    IL_000c:  tail.
    IL_000e:  call bool class MutualRecursion::isEven(int32)
    IL_0013:  ret
}

.method public static default bool isEven (int32 n)
    cil managed
{
    .maxstack 8
    IL_0000:  ldarg.0
    IL_0001:  brtrue.s IL_0005
    IL_0003:  br.s IL_0007
    IL_0005:  br.s IL_0009
    IL_0007:  ldc.i4.1
    IL_0008:  ret
    IL_0009:  ldarg.0
    IL_000a:  ldc.i4.1
    IL_000b:  sub
    IL_000c:  tail.
    IL_000e:  call bool class MutualRecursion::isOdd(int32)
    IL_0013:  ret
}
```

Kuva 4.18: Esimerkki CLI-alustan `tail.`-etuliitteen käytöstä F#-ohjelman häntäkutsuissa.

Javasta eri tavoin poikkeavien kielten tukeminen JVM-alustalla, ja kehityksen kerrotaan keskittyvän erityisesti esimerkiksi häntäkutsujen ja Scheme-tyylisten kontinuaatioiden tuen lisäämiseen alustalle [Ora16c]. Näyttää siis mahdolliselta, että JVM on lähivuosina ottamassa CLI-alustan kiinni häntäkutsutuessa.

4.4 Olemassa olevien kielitoteutusten vertailu

Tässä aliluvussa tarkastellaan funktionaalisten kielten ja moniparadigmakielten toteutuksia JVM- ja CLI-alustoilla pohjautuen kääntäjien tuottaman tavukoodin tarkasteluun ja yksittäisiä kääntäjiä koskeviin julkaisuihin. Tarkastelun tavoitteena on selvittää, mitä eri tekniikoita erilaiset kääntäjät käyttävät näillä alustoilla häntäkutsujen ja sulkeumien toteutuksessa. Osa vertailtavista kääntäjistä ei välttämättä toteuta esimerkiksi häntäkutsujen optimointia lainkaan. Kaikki tarkasteltavat kääntäjät kuitenkin tukevat ensimmäisen luokan funktioita jossakin muodossa ja toteuttavat lambda-lausekesyntaksin anonyymeille funktioille.

Vertailtavia toteutuksia ovat Java, Scala, Clojure, Kotlin, C#, F# sekä Scheme-toteutukset Kawa ja Bigloo. Kuvassa 4.19 on yhteenveto luvussa tarkasteltavista kielistä ja toteutuksista. Alustoilla tarjolla olevista Scheme-toteutuksista IronScheme jää tarkastelun ulkopuolelle, koska se on toteutettu kääntäjää varteen erikseen muokattua DLR-kirjastoa apuna käyttäen [Pri14]. Tämä tekee sen tarkastelun vaikeaksi muun muassa siksi, että toteutus ei salli kääntämistä `exe`-tiedostoksi, jolloin DLR-kirjaston syntaksipuuesityksestä tuotettua tavukoodia ei ole mahdollista nähdä.

Kieli/toteutus	**JVM**	**CLI**	**Tyyppi**
Java	✓		olio / moniparadigma
Kotlin	✓		olio / moniparadigma
Scala	✓		funktionaalinen / moniparadigma
Clojure	✓	✓	funktionaalinen (Lisp-perhe)
C#		✓	olio / moniparadigma
F#		✓	funktionaalinen / moniparadigma
Kawa (Scheme)	✓		funktionaalinen (Lisp-perhe)
Bigloo (Scheme)	✓	✓	funktionaalinen (Lisp-perhe)

Kuva 4.19: Luvussa tarkasteltavat kielet ja niiden toteutukset

Vertailua varten tarkasteltavilla kielillä toteutettiin alla listatut testiohjelmat. Osa testiohjelmista pohjautuu tutkielmassa aiemmin esiteltyihin esimerkkeihin. Muut testiohjelmat esitellään tässä aliluvussa. Testiohjelmat on koottu yhteen liitteessä C. Lihavoidulla tekstillä on merkitty nimi, jolla testitapaukseen voidaan viitata tekstissä.

- **add1**: kuvan 2.4 (sivu 11) kaltainen yksinkertainen sulkeuma

- **counter**: kuvan 4.2 (sivu 28) tapaus, jossa useampi sulkeuma jakaa saman ympäristön
- **factorial**: kuvan 2.5 (sivu 12) häntärekursio tai "funktionaalinen silmukka"
- **odd-even**: kuvan 2.7 (sivu 14) keskinäinen häntärekursio
- **add2**: useampitasoinen sisäkkäinen sulkeuma (kuva 4.25 sivulla 57)
- **double**: tapaus, jossa ylätason funktiota käytetään ensimmäisen luokan arvona (kuva 4.22 sivulla 54)

Osa testiohjelmista toteutettiin tietyille kielille vain siinä tapauksessa, että käyttäytyminen näissä tilanteissa ei ollut pääteltävissä aiempien koodiesimerkkien käyttäytymisen perusteella. CPS-tyylillä kirjoitettua ohjelmaa ei testattu erikseen, koska keskinäisen häntärekursion tapauksen tarkastelu näytti kertovan riittävällä tarkkuudella yleistettyjen häntäkutsujen käsittelystä.

Oliosulkeumatoteutukset

Sulkeumien toteutusten tarkastelu eri kielissä ei tuottanut suuria yllätyksiä. Suurin osa tarkasteltavista toteutuksista hyödyntää jotakin luvussa 4.2 esitellyistä tekniikoista. Kielet, joilla on toteutus molemmilla alustoilla, näyttävät käyttävän samoja tekniikoita molemmilla alustoilla luultavasti ylläpidon helpottamiseksi. Esimerkiksi Clojuren ClojureCLR-toteutus on hyvin suoraviivainen JVM-toteutuksen siirto CLI-alustalle, ja projektin tavoitteeksi mainitaankin pysyminen mahdollisimman lähellä JVM-toteutusta [clo16].

Tarkasteltavista kielistä oliosulkeumia käyttävät Scala, Clojure, F# ja Kotlin. Scalassa, F#:ssa ja Kotlinissa ylätason funktiot käännetään metodeiksi pääluokkaan, mutta jokaista varsinaista sulkeumaa kohti luodaan oma luokka, joka sisältää kentät kaapatuille muuttujille sekä yhden kutsuttavan metodin. Clojure puolestaan ei tee tällaista optimointia edes ylätason funktioiden kohdalla, vaan jokainen funktio saa oman `clojure.lang.AFunction`-luokasta perivän oliosulkeumaluokkansa. Clojure-kääntäjän luomien luokkien määrä onkin huomattavasti muita kääntäjiä suurempi.

Kukin oliosulkeumia käyttävä kieli määrittelee omat rajapintansa tai kantaluokkansa oliosulkeumaluokille. Esimerkiksi Kotlinissa oliosulkeuman luokka toteuttaa geneerisen rajapinnan `kotlin.jvm.functions.Function`*N*, joka määrittelee metodin `invoke`. Esimerkiksi **add1**-esimerkin (kuva 4.20) tapauksessa oliosulkeumaluokka toteuttaa rajapinnan `Function1<java.lang.Integer, java.lang.Integer>`.

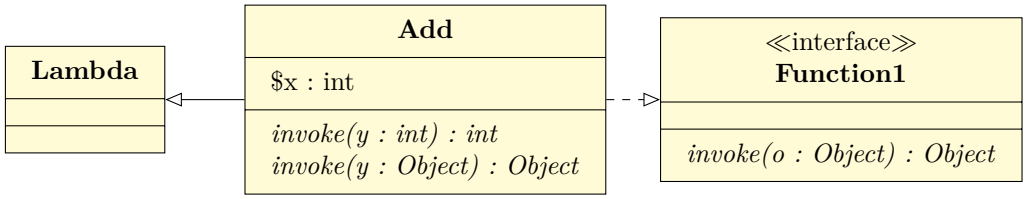
Kuvassa 4.21 on Kotlinin oliosulkeumien esittämisessä käyttämä luokkahierarkia.

add1

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

Kuva 4.20: Esimerkki `add1`: yksinkertainen muuttujan kaappaava sulkeuma

`Add`-luokka on **add1**-esimerkin oliosulkeuma[7]. Kotlinin oliosulkeumassa on kaksi `invoke`-metodia. Tämän esimerkin tapauksessa toinen metodeista ottaa parametrinsa primitiivityyppisinä arvoina ja toteuttaa sulkeuman varsinaisen rungon. Toinen puolestaan toteuttaa `Function1`-rajapinnassa määritellyn `invoke`-metodin ja purkamalla `java.lang.Number`- tai `java.lang.Integer`-olioon käärityn kokonaislukuarvon kääreestään ja delegoimalla sen jälkeen varsinaisen rungon toteuttavalle metodille. Sulkeuman rungon varsinainen toteutus on mahdollisesti tehty erillisenä metodina yksinkertaisesti vaadittujen tyyppitarkastusten ja -muunnosten erottamiseksi varsinaisesta toteutuskoodista. Kuvassa näkyvää `Lambda`-luokkaa ei ole dokumentoitu, mutta se näyttää Kotlinin toteutuksen koodin perusteella sisältävän tukitoimintoja, joita saatetaan käyttää esimerkiksi virheilmoitusten laadinnassa tai debugger-toteutuksissa.



Kuva 4.21: Kotlinin oliosulkeumatoteutus

Koska useat kielitoteutuksista kääntävät ylätason funktiot yksinkertaisesti metodeiksi ohjelman pääluokkaan, on kiinnostavaa tarkastella myös tilannetta, jossa ylätason funktiota käytetään ensimmäisen luokan arvona. Esimerkki tällaisesta tapauksesta on kuvassa 4.22, jossa ylätason proseduuri annetaan parametrina kirjastoproseduurille `map`, joka on tyypillinen esimerkki funktionaalisissa kielissä käytetyistä korkeamman asteen funktioista.

Oliosulkeumia käyttävät kielet toteuttavat esimerkin 4.22 kaltaisen tapauksen luomalla erillisen oliosulkeumaluokan, joka kutsuu metodia `double`. Useissa tapauksissa ylätason funktiot on toteutettu staattisina metodeina, jolloin sulkeumaolio ei tarvitse

[7]Kääntäjän luomien luokkien nimiä on yksinkertaistettu kaavioissa kuvien selkeyden ja tarpeettomien yksityiskohtien karsimisen vuoksi. Esimerkiksi Kotlinin oliosulkeumaluokka **add1**-esimerkin tapauksessa on todellisuudessa nimeltään `SimpleClosureKt$add$1`.

viitettä mihinkään muuhun olioon. Poikkeuksena tästä on Clojure, jossa ylätason proseduurit on toteutettu yksinkertaisesti oliosulkeumina, jolloin niiden käyttö ensimmäisen luokan arvoina ei vaadi erityiskäsittelyä.

double

```
(define double
  (lambda (x)
    (+ x x)))

(map double '(1 2 3))
```

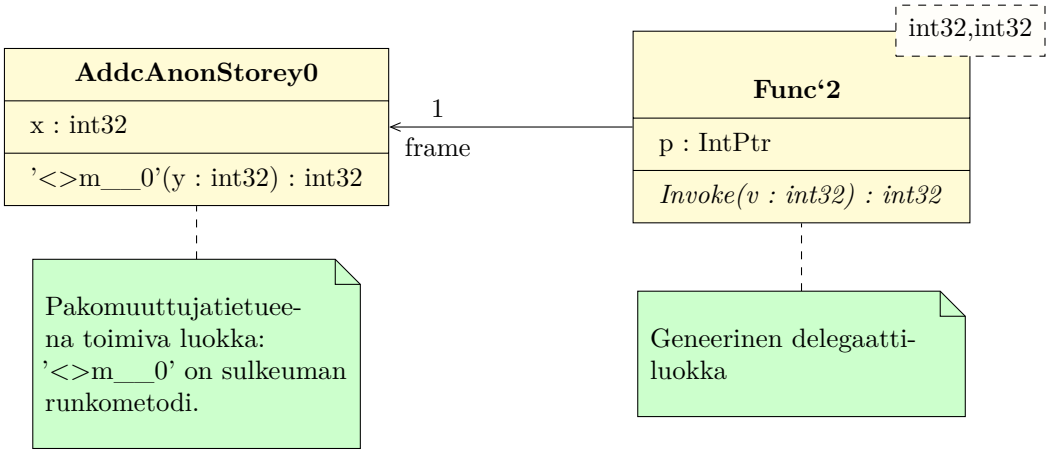
Kuva 4.22: Esimerkki `double`: ylätason proseduurin käyttö ensimmäisen luokan arvona

Alustan tukea hyödyntävät sulkeumatoteutukset

Tarkasteltujen kielten joukossa ainoa CLI-alustan delegaattiluokkia hyödyntävä toteutus on C#. C#-toteutuksessa sulkeumat käännetään siten, että jokaiselle sulkeumia luovalle näkyvyysalueelle luodaan oma sisäluokka, joka toimii sekä pakomuuttujatietueena että paikkana sulkeumien runkojen toteutuksille. Sulkeumien rungot toteutetaan tässä luokassa oliometodeina. Luokasta luodaan yksi ilmentymä, johon talletetaan kaapattavat muuttujat.

Varsinainen sulkeuma luodaan antamalla delegaattiluokalle parametreina raaka metodiosoitin sulkeuman toteuttamaan metodiin sekä viite pakomuuttujatietueesta luotavaan olioon. Raaka metodiosoitin taltioidaan `IntPtr`-tyyppiseen kenttään. `IntPtr` on CLI-alustan sisäänrakennettu alustakohtaisesti vaihteleva tyyppi, jota käytetään raakojen osoittimien esittämiseen [Mic16b]. Tässä toteutuksessa pakomuuttujatietueena toimivan luokan ei tarvitse toteuttaa mitään ennalta määriteltyä rajapintaa. C#-toteutuksen luokkarakenne on kuvattu kuvassa 4.23. Kuvan tapauksen tavukoodi esiteltiin aiemmin kuvissa 4.9 (sivu 37) ja 4.10 (sivu 38).

JVM-alustalla lambda-metatehtaan ja `MethodHandle`-rakenteiden käyttö on ilmeisesti vielä siinä määrin tuore mahdollisuus, että laajalti käytettyjen kielitoteutusten joukossa Java on vielä ainoa, joka hyödyntää sitä. Lambda-metatehtaan käytössä etuna verrattuna muihin tekniikoihin on ainakin teoriassa se, että suoritusaikaiseen ympäristöön tehtävät suorituskykyparannukset voivat välittömästi tuottaa hyötyä tätä tekniikkaa käyttävälle kielitoteutukselle. Toisaalta alustan tuen käyttö myös vapauttaa kääntäjän toteuttajan miettimästä sulkeuman esitystä.



Kuva 4.23: C#:n delegaattitoteutus **add1**-esimerkin (kuva 4.20) tapauksessa.

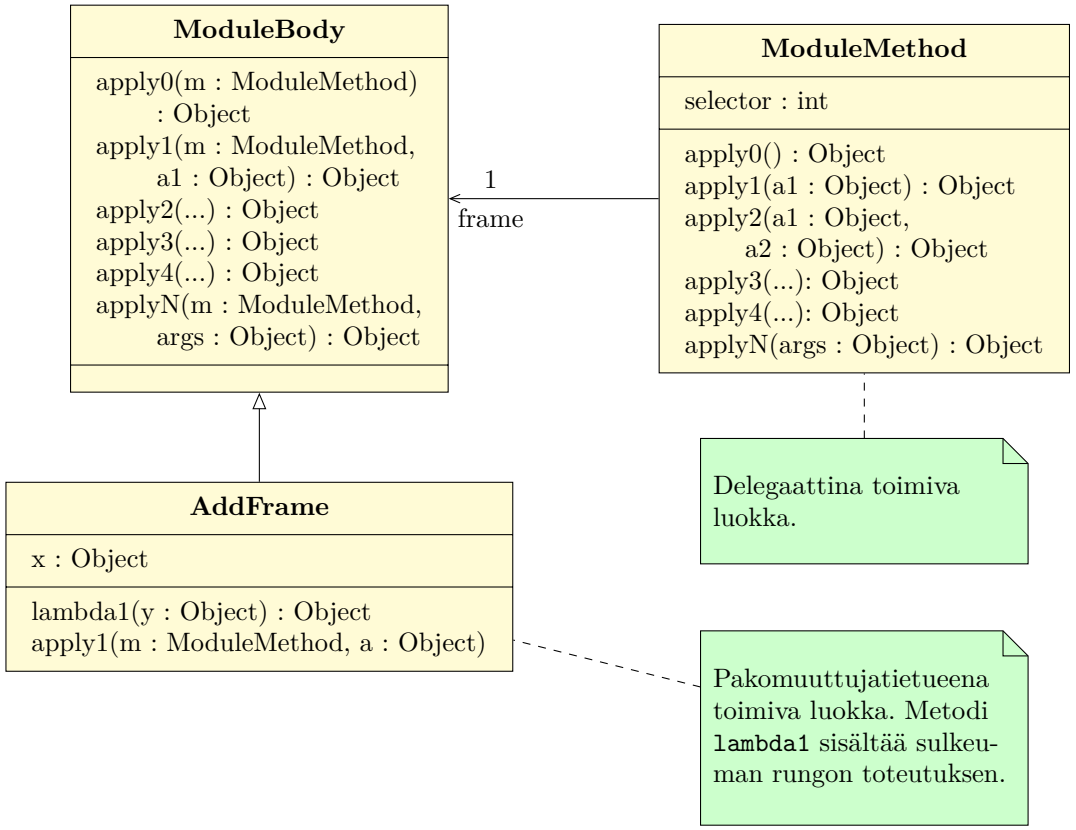
Delegaattityyliset tekniikat

C#:n ohella hieman delegaatteja muistuttavaa tekniikkaa käyttävät molemmat tarkastellut Scheme-toteutukset, Bigloo ja Kawa. Kawan käyttämä tekniikka muistuttaa C#:n käyttämää toteutustapaa siinä, että jokaiselle näkyvyysalueelle luodaan oma pakomuuttujatietueensa, joka sisältää myös sulkeumien rungot toteuttavat metodit. Ero on pääosin siinä, että CLI-delegaattien tai metodiosointinten sijaan kutsuttava metodi valitaan apumetodiin sijoitettavalla `switch`-lauseella erilliseen delegaattiolioon talletetun numeerisen indeksin perusteella.

Kuvassa 4.24 on luokkakaavio Kawan käyttämästä luokkahierarkiasta **add1**-esimerkin yksinkertaisen sulkeuman tapauksessa. Kawan toteutuksessa `add`-funktion näkyvyysalueelle luodaan pakomuuttujatietueena toimiva luokka, joka perii luokan `ModuleBody`. `ModuleBody` määrittelee metodit `apply0`–`4` ja `applyN`, jotka ottavat parametreinaan `ModuleMethod`-olion sekä sulkeumafunktiolle annettavat parametrit. Metodi `applyN` ottaa parametrinsa listamuodossa, ja sitä käytetään silloin, kun sulkeumafunktio ottaa enemmän kuin neljä parametria. Pakomuuttujatietueena toimiva aliluokka korvaa tarvitsemansa `apply`-metodit.

Delegaattiolioina käytetään kutsuttavia `ModuleMethod`-olioita, jotka sisältävät kutsuttavan metodin identifioivan lukuarvon sekä viitteen pakomuuttujatietueeseen. `ModuleMethod`-olio delegoi kutsut `ModuleBody`-oliolle eli pakomuuttujatietueena toimivalle oliolle.

Esimerkin **add1** tapauksessa näkyvyysalue esittelee vain yhden sulkeumafunktion. Funktion runko on toteutettu pakomuuttujatietueluokassa metodina `lambda1`. Lisäksi pakomuuttujatietueluokka korvaa yksiparametristen funktioiden kutsumiseen käytettävän `apply1`-metodin, jonka runko on `switch`-lause. Tämä `switch`-lause valitsee parametri-



Kuva 4.24: Kawan delegaattitoteutus **add1**-esimerkin (kuva 4.20) tapauksessa.

na saatuun `ModuleMethod`-olioon talletetun `selector`-kentän perusteella kutsuttavan metodin.

Bigloon toteutus poikkeaa Kawasta hieman, sillä se ei luo uutta luokkaa jokaiselle näkyvyysalueelle. Bigloossa sulkeumat lambda-nostetaan staattisiksi metodeiksi pääluokan ylätasolle, ja ne saavat parametrina delegaattioliona toimivan `bigloo.procedure`-olion, joka on ympäröivän luokan esiintymä [BSS04]. Tämä olio sisältää kutsuttavan metodin indeksin lisäksi taulukon, johon ympäristöstä kaapatut muuttujat tallennetaan. Tämä vähentää tarvittavien luokkien määrää entisestään, mutta muuttujaviittaukset sulkeumien rungossa on tehtävä indeksoimalla `bigloo.procedure`-olion sisältämää taulukkoa tavallisten parametrien lataus- ja talletuskäskyjen sijaan.

Delegaattityylisissä toteutuksissa kuvan 4.22 **double**-esimerkin kaltaisten ylätason funktioiden käyttö ensimmäisen luokan arvoina ei vaadi erityiskäsittelyä, sillä kutsuttavan metodin valinta voidaan tehdä samalla delegaattitekniikalla kuin muidenkin funktioiden. Kuvan 4.25 **add2**-esimerkin tyyppiset sisäkkäiset sulkeumat toteutetaan siten, että pakomuuttujatietue sisältää viitteen ympäröivän näkyvyysalueen pakomuuttujatietueeseen. Viittaukset ympäröivistä näkyvyysalueista kaapattuihin muuttujiin tehdään tällöin

tämän viitteen kautta. Tämä pätee myös C#:n delegaattitoteutukseen.

add2

```
(define add
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (+ x y z)))))
```

Kuva 4.25: Esimerkki `add2`: sisäkkäinen sulkeuma

Tilan jakaminen sulkeumien välillä

Kuvassa 4.26 on yhteenveto sulkeumien toteutuksessa käytetyistä tekniikoista. Kuten todettiin jo aiemmin, selvästi yleisin käytetty tekniikka erityisesti vertailun tunnetuimpien kielten joukossa ovat oliosulkeumat. Toisaalta Scheme-toteutukset huomioiden melko usein käytettyjä ovat myös tekniikat, joita voisi kuvailla delegaattityylisiksi.

Taulukon viimeiseen sarakkeeseen on merkitty kussakin toteutuksessa käytetyt tekniikat mutaabelin tilan jakamiseen sulkeumien välillä. Esimerkki mutaabelin tilan jakamisesta sulkeumien välillä on `counter`-tapaus (kuva 4.2 sivulla 28).

Taulukkoa silmäilemällä voi äkkiä havaita, että oliosulkeumia hyödyntävät toteutukset käyttävät pääosin viitesoluihin perustuvaa tilan jakoa ja delegaattityyliset toteutukset tyypillisesti jakavat koko pakomuuttujatietueen. Useissa viitesoluja käyttävissä kielissä muuttuja on erikseen merkittävä viitesoluksi, jos sen arvoa halutaan pystyä ylipäätään muuttamaan. Näin on F#:ssa (avainsana `ref`), Scalassa (muuttujatyyppi `var`) ja Cloju-ressa, jossa on useita erilaisia viitesolutyyppejä, joista ehkäpä yleisimmin käytetty on säieturvallinen `atom`.

Kieli	**OS**	**CLI-D**	**MD**	**LMT**	**Tilan jakaminen**
Java				✓	-
Scala	✓				VS
Clojure	✓				VS
C#		✓			jaettu PMT
F#	✓				VS
Kotlin	✓				VS
Kawa (Scheme)			✓		jaettu PMT
Bigloo (Scheme)			✓		VS

Kuva 4.26: Eri toteutusten käyttämät sulkeumien toteutustekniikat. Taulukossa listatut tekniikat ovat oliosulkeuma (**OS**), CLI-delegaattiluokat (**CLI-D**), muu delegaattityylinen toteutus (**MD**) ja lambda-metatehdas (**LMT**). Muut käytetyt lyhenteet ovat PMT eli pakomuuttujatietue ja VS eli viitesolu.

Todennäköisesti syynä tekniikoiden selkeään jakautumiseen on se, että delegaattityylistä toteutusta käytettäessä sulkeumien runkometodit sijoittuvat oliometodeiksi pakomuuttujatietuetta esittävään luokkaan, jolloin niillä on välitön pääsy kaikkiin esittely-ympäristönsä pakeneviin muuttujiin. Toisaalta viitesolut sopivat luontevasti yhteen oliosulkeumien kanssa. Oliosulkeumia käytettäessä erillisten jaettavien pakomuuttujatietueiden rakentaminen tuottaisi vain ylimääräistä työtä ja kasvattaisi tarvittavien luokkien määrää entisestään. Bigloon käyttämä delegaattityyli poikkeaa hiukan muista toteutuksista, mikä lienee syynä muiden toteutusten linjasta poikkeavan tilanjakotekniikan valintaan.

Java poikkeaa muista kielistä siinä, että se ei salli lainkaan muutoksia sulkeuman kaappaamiin arvoihin. Java 8:n lambda-lausekkeiden kaappaamia arvoja koskevat samat rajoitteet kuin Javan nimettömiä luokkiakin: kaapatun muuttujan on oltava merkitty `final`-avainsanalla tai "tosiasiallisesti lopullinen", kuten mainittiin luvussa 4.2. Ohjelmoija voi kiertää rajoitetta käärimällä itse kaapattavat muuttujat viitesolun kaltaiseen rakenteeseen. Tämä kuitenkin edellyttää ohjelmoijalta ylimääräistä työtä muihin tarkasteltuihin kieliin verrattuna. Goetzin [Goe13] mukaan syynä Javan poikkeavaan tulkintaan sulkeumista ovat rinnakkaisuuden aiheuttamat haasteet mutatoitavien akkumulaattorimuuttujien oikeaoppiselle käytölle. Alla on Goetzin [Goe13] esimerkki tällaisesta käyttötapauksesta.

```
int sum = 0;
list.forEach(e -> { sum += e.size(); });
```

Esimerkkikoodissa muuttujaa `sum` käytetään sulkeumassa mutatoitavana akkumulaattorimuuttujana. Esimerkin syntaksi muistuttaa Java 8:aa, mutta koodi ei käänny Java-kääntäjällä, sillä sulkeuman kaappaaman muuttujan arvon muuttaminen ei ole Javassa sallittua. Goetz mainitsee, että tämän kaltaiset tapaukset ovat haastavia toteuttaa monisäikeisessä ympäristössä oikein aiheuttamatta kilpatilanteita. Tästä syystä Javan tapauksessa on päätetty kieltää tällaiset rakenteet kokonaan ja tarjota niiden sijaan yleisten mutatoitavien akkumulaattorimuuttujien käyttötapausten toteuttamiseen valmiita rinnakkaisuuden kannalta turvallisia kirjastofunktioita. Tällaisia ovat esimerkiksi `java.util.stream`-pakkauksen `reduce` ja `sum` [Goe13].

Käytännössä Javan valitsema lähestymistapa ei kuitenkaan ole ainoa ratkaisu säieturvallisuusongelmaan, sillä esimerkiksi Clojure ratkaisee vastaavan ongelman säieturvallisesti toteutetulla `atom`-viitesolulla, jonka arvo on muutettavissa vain atomisella operaatiolla `swap!`. Javan tapaus on toki siinä mielessä erilainen, että koska Java tavallisesti sallii muiden kuin `final`-avainsanalla merkittyjen muuttujien mutaatiot, olisi todennäköisesti vaikea varmistaa, että ohjelmoija voi käyttää tämän kaltaisia sulkeumia vain oikeanlaisen viitesolurakenteen kanssa.

Häntäkutsut

Tiedot häntäkutsujen optimoinnista tarkasteltavissa kielissä on koottu kuvan 4.27 taulukkoon. Useimmat vertailun kielistä optimoivat häntärekursion silmukaksi, mutta harva kieli tukee yleistettyjen häntäkutsujen optimointia.

Kieli	Häntärekursio	Yleinen	Huomautuksia
Java			
Scala	✓		
Clojure	✓		
C#			
F#	✓	(✓)	vain keskinäinen rekursio
Kotlin	✓		
Kawa (Scheme)	✓	✓	käännösvipu `--full-tailcalls`
Bigloo (Scheme)	✓	(✓)	vain CLI, `-fdotnet-full-tailc`

Kuva 4.27: Häntäkutsujen optimointi tarkasteltavissa toteutuksissa.

Oliokielet Java ja C# eivät optimoi häntärekursiota tai muita häntäkutsuja lainkaan. Tämä on odotettu tulos, sillä oliokielissä silmukoiden käyttö ja tilan muuttaminen on rekursion käyttöä tyypillisempi ohjelmointityyli. Kaikki muut tarkastellut kielitoteutukset optimoivat häntärekursiivisen funktion silmukaksi, mutta osa toteutuksista vaatii erillisen avainsanan käyttöä häntärekursiivisessa koodissa. Kotlinissa häntärekursiiviset funktiot merkitään `tailrec`-avainsanalla ja Clojuressa häntäkutsu korvataan `recur`-avainsanalla. F#:ssa kaikki rekursiiviset funktiot on merkittävä `rec`-avainsanalla, mutta merkintä ei liity häntärekursioon, vaan yksinkertaisesti ohjeistaa kääntäjää tuomaan esiteltävän funktion nimen käytettäväksi funktion sisällä [Del16a]. `Rec`-avainsanan käyttö F#:ssa periytynee toisesta ML-perheeseen kuuluvasta kielestä, OCamlista.

Clojuressa paikallisten "funktionaalisten silmukoiden" toteutuksessa käytetään lisäksi erillistä `loop`-rakennetta yhdessä `recur`-avainsanan kanssa [Hic16b]. Tätä rakennetta olisi ehkäpä täsmällisempää kuvailla erilliseksi silmukkarakenteeksi kuin häntärekursion optimoinniksi, vaikka rajanveto tässä onkin haastavaa. Silmukkarakenne toimii siten, että jokaisella kierroksella silmukkamuuttujien arvoiksi päivitetään `recur`-avainsanalle parametreina annetut arvot, eli sen käyttö poikkeaa hieman imperatiivisten kielten silmukkarakenteista, joissa vastaava käyttö edellyttäisi yleensä koodissa näkyvää muuttujien mutatointia. Alla on esimerkki Clojuren silmukkarakenteen käytöstä **factorial**-tapauksessa.

```
(defn fact [n]
  (loop [m n acc 1] ; tässä esitellään silmukkamuuttujat m ja acc
    (if (zero? m)
      acc
      (recur (- m 1) (* acc m)))))
```

Tarkasteltavista toteutuksista ainoat, jotka tukevat yleistettyjä häntäkutsuja kaikissa tapauksissa näyttävät olevan Kawa ja Bigloon CLI-toteutus. Kawa vaatii erillisen `--full-tailcalls`-käännösvivun käyttöä häntäkutsuoptimoinnin päälle kytkemiseksi ja käyttää toteutustekniikkana trampoliineja.

Bigloo ei optimoi yleistettyjä häntäkutsuja lainkaan JVM-alustalla [SS02], mutta antaa mahdollisuuden kytkeä alustan tukea hyödyntävä optimointi päälle CLI-alustalla. Häntäkutsuoptimointi ei ole Bigloossa toiminnassa oletuksena, sillä Bres, Serpette ja Serrano havaitsivat varhaisissa suorituskykymittauksissaan CLI-alustan häntäkutsujen käytön hidastavan suoritusta joissakin tilanteissa merkittävästi — pahimmassa tapauksessa jopa viisi kertaa optimoimatonta versiota hitaammaksi [BSS04]. Mittaustulokset ovat kuitenkin vuodelta 2004, joten tilanne voi olla sittemmin muuttunut.

Clojure tarjoaa sisäänrakennettuna trampoliinifunktion, mutta ohjelmoijan on käytettävä sitä itse, eli toteutus ei itse optimoi häntäkutsuja. Lisäksi trampoliinifunktion käyttö edellyttää, että sen kanssa käytettävät funktiot muokataan häntäkutsun sijasta palauttamaan kontinuaationa toimiva sulkeuma, mikä saattaa tehdä trampoliineja käyttävästä koodista hiukan tavallisia häntärekursiivisia kutsuja käyttävää koodia vaikealukuisempaa. Trampoliinien kanssa käytettävät funktiot eivät myöskään pysty palauttamaan funktioarvoa paluuarvonaan muuten kuin kontinuaationa, vaan tällaisessa tilanteessa funktioarvo on käärittävä johonkin muuhun tietorakenteeseen [Hic16a]. Clojuren trampoliinifunktio on tarkoitettu lähinnä keskinäisen häntärekursion toteutukseen [Hic16a].

F# puolestaan optimoi häntäkutsut tavallisen häntärekursion lisäksi ainoastaan keskinäisessä häntärekursiossa. Keskinäiseen häntärekursioon voi kuitenkin osallistua enemmän kuin kaksi funktiota. Keskenään rekursiiviset funktiot on esiteltävä erillistä `let rec ... and` -rakennetta käyttämällä, sillä normaalisti F#:n funktiot voivat viitata vain esittelyjärjestyksessä niitä edeltäviin funktioihin ja arvoihin. F# optimoi keskinäisen häntärekursion käyttämällä CIL-alustan häntäkutsutukea, kuten nähtiin esimerkkikuvassa 4.18 (sivu 50).

Alustojen tarjoaman tuen käyttö funktionaalisten piirteiden toteutuksessa

Tässä luvussa esiteltiin ensin kolme eri tyyppistä lähestymistapaa ensimmäisen luokan funktioarvojen ja sulkeumien toteutukseen: oliosulkeumat, tyyppi- ja muistiturvalliset funktio-osoittimet kuten CLI-alustan delegaatit ja JVM-alustan `MethodHandle` sekä JVM-alustan lambda-metatehdas. Olemassa olevien kielten vertailussa havaittiin lisäksi Scheme-toteutusten käyttävän delegaattityylisiä toteutustekniikoita, joissa funktio-osoittimia simuloidaan hyppytaulumaisesti käytettävän `switch`-lauseen avulla. Lisäksi luvussa esiteltiin kaksi menetelmää häntäkutsujen optimointiin: CLI-alustan sisäänrakennettu häntäkutsutuki ja trampoliinitekniikka. Vertailluista kielistä varsinaista alustan tukea

hyödynsivät vain Java ja C# sulkeumien osalta sekä F# ja Scheme-toteutus Bigloo häntäkutsutuen osalta.

Seuraavassa luvussa esitellään oma toteutus Scheme-kielen osajoukolle molemmilla kohdealustoilla. Oman toteutuksen tavoitteena on arvioida alustojen tarjoaman sisäänrakennetun tuen soveltuvuutta Schemen kaltaisen yksinkertaisen funktionaalisen kielen toteutukseen. Omaan toteutukseen ja toteutuksen yhteydessä havaittuihin seikkoihin sekä tässä luvussa esitettyyn olemassa olevien kielten vertailuun perustuva analyysi alustojen tukitoiminnoista esitellään luvussa 6.

5 Cottontail Scheme

Tässä luvussa esitellään tutkielmaa varten tehty esimerkkitoteutus Scheme-kielen osajoukosta. Toteutuksessa tavoitteena oli käyttää mahdollisuuksien mukaan alustakohtaisesti tarjolla olevaa sulkeuma- ja häntäkutsutukea. Tapauksissa, joissa alustan tukea ei ollut tarjolla tai alustan tuki ei mahdollistanut kaikkien vaadittujen ominaisuuksien toteuttamista, on nojauduttu yleisempiin menetelmiin kuten trampoliineihin. Esiteltävä toteutus on nimeltään Cottontail Scheme[8].

Esiteltävä toteutus sisältää takaosan sekä CLI-alustalle että JVM-alustalle. Toteutus koostuu kahdesta ohjelmasta: CLI-kääntäjästä ja erillisestä JVM-takaosasta. Molemmat kääntäjät käyttävät hyväkseen CLI-kääntäjässä toteutettua etuosaa. Kääntäjän koodin lataukseen ja käyttöön liittyvistä yksityiskohdista on tarkempaa tietoa liitteessä B.

Luvun aluksi käydään läpi valitun Scheme-kielen osajoukon rajaus ja esitellään kääntäjän osat ja tukikirjastot. Tämän jälkeen esitellään valitut funktioarvojen ja häntäkutsujen tavukooditason toteutukset ensin JVM-alustalla ja sitten CLI-alustalla. Lopuksi käydään läpi harkittuja vaihtoehtoja valituille toteutustavoille sekä perustellaan tehdyt toteutusvalinnat.

5.1 Lähdekielen rajaus ja kääntäjän toteutus

Koska tutkielman tarkastelun kannalta keskeisimpiä ominaisuuksia Schemessä ovat lähinnä funktioarvot ja häntäkutsut, toteutettavaksi on rajattu pieni osajoukko kielestä. Erityisesti seuraavia ominaisuuksia on yksinkertaistettu tai rajattu toteutuksen ulkopuolelle.

- Todellisia matemaattisia lukutyyppejä kuten reaaalilukuja ja kompleksilukuja esittämään pystyvän numeeristen tyyppien tornin [Sus13, s. 32] sijaan kaikki lukuarvot esitetään kaksoistarkkuuden liukulukuina eli CIL-alustalla `System.Double` ja JVM-alustalla `double`-tyyppisinä arvoina.
- Makrojärjestelmä ei sisälly toteutukseen.
- Proseduurit `call-with-current-continuation` ja `dynamic-wind` [Sus13, s. 52–53] eivät sisälly toteutukseen.[9]
- Valtaosa R7RS-standardissa määritellyistä kirjastoproseduureista ei sisälly to-

[8] Nimi mukailee varhaisissa Scheme-kääntäjissä käytettyä nimentätapaa: ensimmäinen toteutettu Scheme-kääntäjä oli nimeltään Rabbit [SJ78], ja eräs myöhempi optimoiva kääntäjä puolestaan oli nimeltään Hare [Teo91], oletettavasti viittauksena alkuperäiseen kääntäjään.

[9] Nämä ominaisuudet liittyvät niin sanottuihin ensimmäisen luokan kontinuaatioihin, eivätkä suoranaisesti esimerkiksi CPS-tyyliin. Ensimmäisen luokan kontinuaatioiden toteutus ei kuulu tämän tutkielman piiriin.

teutukseen lukuun ottamatta listojen käsittelyyn, tulostukseen ja tavanomaiseen aritmetiikkaan liittyviä proseduureja. Kaikki liitteessä C listatut esimerkkiohjelmat ovat riittävän yksinkertaisia käännettäviksi Cottontail Schemen nykyisellä kirjastotototeutuksella.

- Toteutus tukee symboliarvoja, mutta käytännössä nämä esitetään tavallisina merkkijonoina ja vertailu tapahtuu merkkijonovertailuna. Tavallisesti symbolit toteutettaisiin tavalla, joka mahdollistaa nopeamman vertailun.

- Toteutus ei sisällä REPL-käyttöliittymää, eli vain kääntäminen JVM-alustalla `class`-tiedostoiksi ja CLI-alustalla `exe`-tiedostoksi on mahdollista.

Kirjastotototeutuksen rajaus ja kielioppi

Kuvassa 5.1 on toteutetun Schemen osajoukon EBNF-kielioppi. Kielioppi on karsittu versio R7RS-standardidokumentissa esitetystä kieliopista [Sus13, s. 62–65]. Esityksestä on tiivistämisen vuoksi karsittu pois tunnisteiden ja erilaisten literaaliarvojen muodostussäännöt. Kieliopissa *komento* merkitsee erityisesti lauseketta, joka suoritetaan vain sen sivuvaikutusten vuoksi.

Tunnisteiden ja numeeristen literaalien muodostussääntöjä on hiukan yksinkertaistettu. Esimerkiksi numeeriset literaalit eivät salli erilaisten kantalukujen käyttöä. Nyrkkisääntönä tunnisteiksi hyväksytään useimmat tavallisimmat Scheme-ohjelmissa käytetyt tunnisteet, jotka sisältävät numeroita, kirjainmerkkejä ja väliviivoja. Numeerisiksi literaaleiksi hyväksytään tavalliset kokonaisluku- ja desimaalilukuliteraalit.

Kuva 5.2 listaa toteutuksen tukemat kirjastoproseduurit ja kontrollirakenteet. Sisäänrakennettujen proseduurien ja kontrollirakenteiden nimien käyttö omien muuttujien niminä sekä näiden nimien uudelleenasetus `set!`-lausekkeella on estetty toteutuksen yksinkertaistamiseksi. Standardin mukaan ohjelman oman koodin ulkopuolelta tuodun nimen eli esimerkiksi jonkin standardissa määritellyn kirjastoproseduurin nimen uudelleenmäärittely `define`-rakenteella tai nimen osoittaman arvon asettaminen `set!`-rakenteella ei ole sallittua ja saa johtaa toteutuksessa virhetilanteeseen [Sus13, s. 25]. Tämä antaa kääntäjätoteutuksille optimointimahdollisuuksia, koska kääntäjä saa olettaa, että esimerkiksi nimi `+` viittaa aina standardissa määriteltyyn summausoperaatioon. Standardi kuitenkin kehottaa REPL-toteutuksia sallimaan nimien uudelleenmäärittelyn ja asetuksen [Sus13, s. 25].

data	→	*atominen data*
	→	*lista*
atominen data	→	*totuusarvo*
	→	*lukuarvo*
	→	*merkkijono*
	→	*symboli*
symboli	→	*tunniste*
lista	→	`(`*data*`*)`
ohjelma	→	(*komento* | *määritelmä*)$^+$
määritelmä	→	`(define` *tunniste lauseke*`)`
komento	→	*lauseke*
lauseke	→	*tunniste*
	→	*literaali*
	→	*proseduurikutsu*
	→	*lambda-lauseke*
	→	*ehtolauseke*
	→	*muu kontrollirakenne*
	→	*sijoitus muuttujaan*
literaali	→	*lainaus*
	→	*totuusarvo*
	→	*lukuarvo*
	→	*merkkijono*
lainaus	→	`'`*data*
	→	`(quote` *data*`)`
proseduurikutsu	→	`(`*operaattori operandi*`*)`
operaattori	→	*lauseke*
operandi	→	*lauseke*
lambda-lauseke	→	`(lambda` *parametrit runko*`)`
parametrit	→	`(`*tunniste*`*)`
	→	*tunniste*
runko	→	*määritelmä* komento* lauseke*
ehtolauseke	→	`(if` *lauseke lauseke lauseke*`)`
sijoitus muuttujaan	→	`(set!` *tunniste lauseke*`)`
muu kontrollirakenne	→	`(begin` *komento* lauseke*`)`
	→	`(and` *lauseke*$^+$`)`
	→	`(or` *lauseke*$^+$`)`

Kuva 5.1: Cottontail Schemen EBNF-kielioppi. Konekirjoitusfontilla merkityt merkit kuten `define` ovat kieliopin päätesymboleja. Kursiivilla merkityt nimet ovat välikesymboleja.

Tyyppi	Proseduurit tai kontrollirakenteet
tulostus	display, newline
muuttujien asetus	set!
aritmetiikka	+, -, /, *
listaproseduurit	cons, car, cdr, list, null?
korkeamman asteen funktiot	map
vertailu	<, >, eq?, zero?
totuusarvojen käsittely	not
sivuvaikutusten ketjutus	**begin**
muut kontrollirakenteet	**if**, **and**, **or**

Kuva 5.2: Toteutuksen sisältämät kirjastoproseduurit ja kontrollirakenteet. Kontrollirakenteet on korostettu.

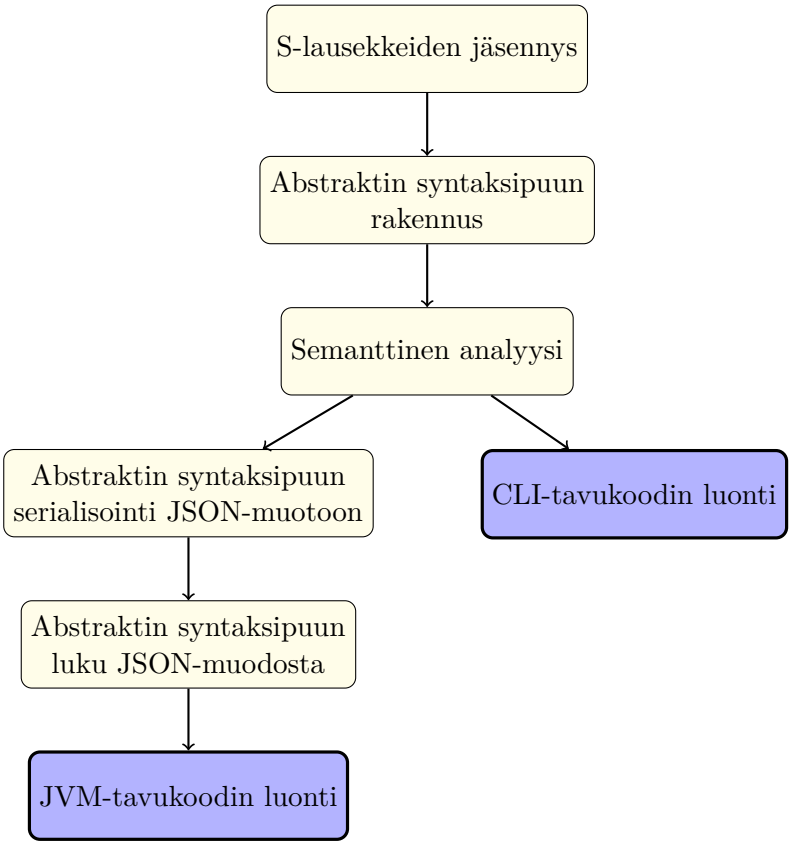
Käännöksen vaiheet

Toteutettu kääntäjä koostuu jaetusta etuosasta sekä CLI- ja JVM-takaosista. Toteutus jakautuu kahteen erilliseen itsenäisesti ajettavaan ohjelmaan, joista toinen toteuttaa etuosan ja CLI-takaosan, ja toinen toteuttaa JVM-takaosan. Etuosa jaetaan molempien takaosien välillä muuntamalla semanttisen analyysin tuottama abstrakti syntaksipuu JSON-muotoon, joka tulostetaan standardiulostuloon. JVM-alustalle käännettäessä siis CLI-kääntäjän etuosa ja JVM-takaosa yhdistetään putkittamalla CLI-etuosan tuottama tuloste JVM-takaosalle.

Käännösprosessin vaiheet on kuvattu kuvassa 5.3. Koska tarkastelun kohteena on JVM- ja CLI-tavukoodin luonti, kiinnostavia ovat pääosin koodinluontivaiheet, jotka on korostettu kuvassa. Kääntäjän CLI-takaosassa koodinluonti on toteutettu `System.Reflection`-rajapinnan avulla. JVM-takaosa puolestaan käyttää ObjectWeb ASM -kirjastoa.

Kääntäjän etuosa koostuu yksinkertaisesta S-lausekkeiden jäsennyksestä, varsinaisen abstraktin syntaksipuun rakentavasta vaiheesta sekä semanttisesta analyysista, joka merkitsee abstraktiin syntaksipuuhun useita takaosan kannalta hyödyllisiä tietoja. Tavanomaisten tarkistusten kuten muuttujaviittausten ja muiden kuin ensimmäisen luokan proseduurikutsujen argumenttitarkistusten lisäksi semanttinen analyysi merkitsee esimerkiksi sulkeumiin tiedon niiden kaappaamista muuttujista. Koska Scheme on dynaamisesti tyypitetty kieli, ensimmäisen luokan proseduurikutsujen argumentteja ei voida tarkastaa käännösaikana.

Lisäksi kaikkiin proseduureihin merkitään tieto siitä, käytetäänkö niitä ensimmäisen luokan arvoina tai asetetaanko proseduurin nimen osoittama arvo uudelleen jossakin vaiheessa ohjelmaa. Tämän perusteella on mahdollista tehdä esimerkiksi valintoja siitä, mitkä proseduurit on tarpeen esittää proseduuriolioina ja mitkä voidaan esittää yksinkertaisesti metodeina ohjelman pääluokan sisällä.



Kuva 5.3: Käännösprosessin vaiheet

Kirjastototeutus

Molempien alustojen toteutuksia varten on toteutettu oma apukirjasto, joka sisältää toteutetut kirjastoproseduurit, Schemen tyyppien ja proseduurien esityksen määrittelyn sekä mahdollisesti joitakin apumetodeita, joita kutsutaan luodusta koodista käsin. JVM-alustalla kirjasto on toteutettu Javalla, ja se toimitetaan osana JVM-kääntäjätoteutuksen sisältävää `jar`-pakkausta. CIL-alustalla kirjaston toteutuskieli on C#, ja se muodostaa oman `dll`-pakkauksensa.

Koska Scheme on dynaaminen kieli, kaikkien muuttujien on voitava sisältää minkä tyyppisiä arvoja tahansa. Koska tässä prototyyppitoteutuksessa tehokkuus ei ole tärkein tavoite, Cottontail Scheme ei edes pyri esimerkiksi käyttämään primitiivityyppejä missään yhteydessä, vaan jokaiselle Scheme-tyypille on määritelty oma luokkansa. Tällöin ei tarvita esimerkiksi `invokedynamic`-käskyjä tai DLR-kirjastoa tavallisten funktiokutsujen toteuttamiseen.

Molemmat kirjastot määrittelevät pääpiirteittäin saman tyyppihierarkian. Cottontail Schemen tyyppien kantaluokka molempien alustojen kirjastoissa on nimeltään `CTObject`.

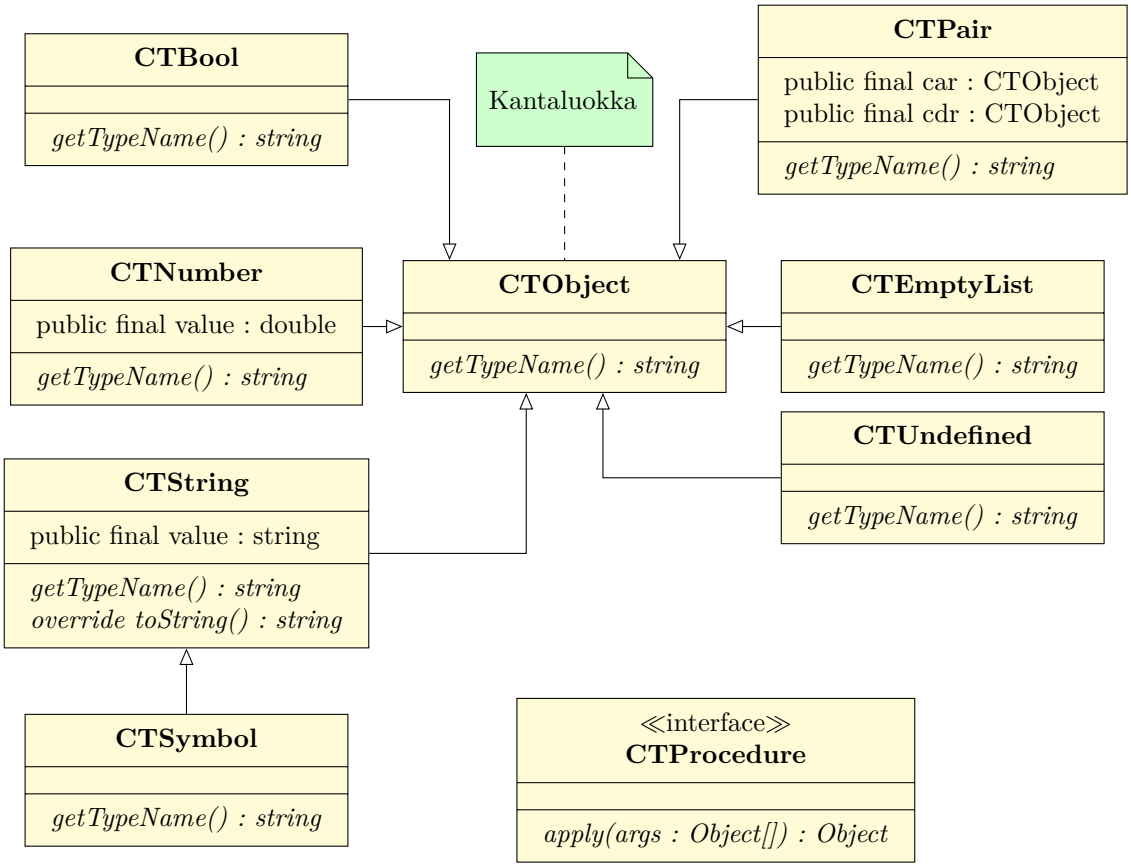
Tuettuja tyyppejä ovat kaksoistarkkuuden liukulukuina esitettävät numeroarvot, totuusarvot, merkkijonot, symbolit, proseduurit, tyhjät listat ja parit. Näitä tyyppejä vastaavat toteutuksessa luokat `CTNumber`, `CTBool`, `CTString`, `CTSymbol`, `CTProcedure`, `CTEmptyList` ja `CTPair`. Lisäksi toteutus määrittelee `CTUndefined`-tyypin, jonka avulla esitetään sivuvaikutuksellisten lausekkeiden kuten `set!`, `display` ja `newline` paluuarvoa. Tyypeistä `CTBool` ja `CTUndefined` luodaan ennalta vakioina käytettävät ilmentymät. Esimerkiksi `CTBool`-luokasta on siis ohjelman suorituksen aikana olemassa kaksi ilmentymää, joista toinen esittää arvoa `#t` ja toinen arvoa `#f`.

Kuvassa 5.4 on JVM-kirjastotototeutuksen mukainen luokkahierarkia tärkeimpine metodeineen. CLI-toteutus sisältää samannimiset luokat, mutta joissakin toteutusyksityiskohdissa on joitakin toteutuksen tarkastelun kannalta merkityksettömiä eroja. Luokkahierarkiassa näkyvä metodi `getTypeName()` palauttaa tyypin nimen tyyppivirheiden virheilmoituksia varten.

Merkittävämpiä eroja on `CTProcedure`-tyypin määrittelyssä — JVM-alustalla kyseessä on funktionaalinen rajapinta, CLI-alustalla taas `CTObject`-luokasta perivä luokka. `CTProcedure`-tyypin toteutukseen liittyviä yksityiskohtia tarkastellaan tarkemmin funktioarvojen toteutukseen tutustuttaessa.

Molempien alustojen kirjastotototeutukset sisältävät `BuiltIns`-nimisen luokan, joka sisältää kaikkien sisäänrakennettujen kirjastoproseduurien toteutukset. Kukin kirjastoproseduuri suorittaa itse tarvittavat tyyppitarkastukset parametreilleen: esimerkiksi `+`-proseduuri tarkastaa ennen operaation suorittamista, että kaikki sen saamat argumenttiarvot ovat tyyppiä `CTNumber`. Virheellisen tyyppiset argumentit aiheuttavat suoritusaikaisen `TypeError`-poikkeuksen. Käyttäjän määrittelemistä proseduureista tuotettuun tavukoodiin ei tällöin tarvitse lisätä tyyppitarkastuksia tai -muunnoksia, sillä ne käsittelevät kaikkia arvoja ainoastaan ennalta määriteltyjen `CTObject`- tai `Object`-tyyppisiä arvoja parametreinaan ottavien kirjastoproseduurien avulla. On mahdollista, että laajempi kirjastotototeutus ja tuki joillekin Cottontail Schemen toteutuksen ulkopuolelle rajatuille tyypeille vaatisi tyyppitarkastusten lisäyksiä myös käyttäjän koodista tuotettuun tavukoodiin, mutta nykyisen laajuisessa toteutuksessa tällaiset tarkastukset ovat tarpeettomia.

Kirjastossa määritellyt varargs-proseduurit kuten `+` ja `list` ottavat parametrinaan taulukon olioita. Tämä poikkeaa hieman käyttäjän määrittelemistä varargs-proseduureista, jotka ottavat parametrinsa Schemen listamuodossa. Tämä aiheuttaa joitakin eroavaisuuksia käyttäjän määrittelemien proseduurien ja sisäänrakennettujen proseduurien käytössä ensimmäisen luokan arvoina, sillä sisäänrakennettuja varargs-proseduureja voidaan kutsua suoraan antamalla parametrina taulukko. Käyttäjän määrittelemät varargs-proseduurit puolestaan vaativat, että niitä kutsuttaessa parametreista rakennetaan `CTPair`-tyyppinen listarakenne joko käyttäjän määrittelemän proseduurin rungossa ennen proseduurikutsua,



Kuva 5.4: Cottontail Schemen tyyppihierarkia JVM-toteutuksessa

varargs-proseduurin toteuttavan metodin rungossa tai jossakin apumetodissa. Toteutus on hiukan erilainen molemmilla alustoilla, joten palaamme asiaan koodinluonnin esittelyssä.

5.2 Koodinluonti JVM-alustalla

Tässä aliluvussa esitellään funktioarvojen ja häntäkutsujen tavukooditason toteutus Cottontail Scheme -kääntäjän JVM-takaosassa. JVM-alustalla Cottontail Scheme kääntää Scheme-ohjelman yhdeksi luokaksi, jossa globaalit muuttujat ja ylimmän tason proseduurit on määritelty staattisina muuttujina ja metodeina. Häntärekursiiviset kutsut käännetään silmukkamaiseksi koodiksi.

Kuvassa 5.5 on esitetty Cottontail Schemen tuottamaa yksinkertaisen Scheme-ohjelman käännöstä pääpiirteittäin vastaava Java-koodi. Esimerkkikoodin `main`-metodin viimeinen rivi liittyy yleistettyjen häntäkutsujen toteutukseen, jota tarkastellaan myöhemmin. Kuvan `main`-metodin koodissa käytetään muuttujia lukemisen helpottamiseksi. Oikea koodi pitää välitulokset operandipinossa. Häntärekursiivinen proseduuri `print-list`

on käännetty silmukaksi.

print-list.scm

```
(define print-list
  (lambda (l)
    (if (null? l)
      (newline)
      (begin
        (display (car l))
        (display " ")
        (print-list (cdr l))))))

(print-list '(1 2 3 4 5 6))
```

```
public class PrintList {
    private static Object printList(Object l) {
        while (!BuiltIns.isNull(l)) {
            BuiltIns.display(BuiltIns.car(l));
            BuiltIns.display(new CTString(" "));
            l = BuiltIns.cdr(l);
        }
        return BuiltIns.newline();
    }

    public static void main(String[] args) {
        Object[] arr = new Object[] {
            new CTNumber(1),
            // ...
            new CTNumber(6)
        };
        Object result = printList(BuiltIns.toList(arr));
        ProcedureHelpers.trampoline(result);
    }
}
```

Kuva 5.5: Cottontail Schemen JVM-takaosan tuottamaa käännöstä vastaava Java-koodi yksinkertaiselle Scheme-ohjelmalle, joka tulostaa listan häntärekursiivisen funktion avulla

Koska esimerkiksi ylätason proseduurit ovat pelkkiä staattisia metodeja, monia proseduureja voidaan kutsua suoraan staattisilla metodikutsuilla sen sijaan, että rakennettaisiin kutsuttava proseduuriolio. Esimerkki tästä on ylätason proseduuri `print-list` kuvassa 5.5. Poikkeuksen muodostavat ensimmäisen luokan arvoina käytettävät proseduurit sekä muuttujia kaappaavat sisäkkäiset proseduurit, jotka on esitettävä tavalla tai toisella kutsuttavina olioina.

JVM-alustalla funktioarvojen toteutus on mahdollista tehdä pääasiallisesti neljällä eri tyylillä: käyttämällä jotakin muunnelmaa oliosulkeumista, käyttämällä Bigloon

tai Kawan tyylistä delegaattimaista toteutusta, `MethodHandle`-rakenteiden avulla tai käyttäen `LambdaMetafactory`-toiminnallisuutta Java 8:n tavoin. Cottontail Scheme -prototyyppitoteutuksen lopullisessa versiossa on käytössä Java 8 -tyylinen sulkeumatoteutus. Schemen dynaaminen tyypitys tosin vaatii joitakin lisäyksiä Javan käyttämään tyyliin. Aiemmassa vaiheessa toteutuksessa kokeiltiin myös `MethodHandle`-pohjaista toteutusta, jota verrataan lambda-metatehtaan käyttöön myöhemmin luvussa 5.4.

Käytetyt funktionaaliset rajapinnat

JVM-toteutuksessa ensisijainen proseduuriolion esitys on funktionaalinen rajapinta `CTProcedure`. `CTProcedure` määrittelee yhden metodin, `apply`, joka ottaa parametrinsa taulukkona `Object`-tyyppisiä arvoja. Kaikki ensimmäisen luokan arvoina käytettävät proseduurit esitetään viime kädessä `CTProcedure`-rajapintaan mukautuvina arvoina. Kaikki Cottontail Schemen ensimmäisen luokan proseduurit siis ottavat parametrinsa taulukkoon pakattuina.

`CTProcedure`-rajapinnan lisäksi toteutus määrittelee funktionaaliset rajapinnat `CTProcedure0`, `CTProcedure1`, `CTProcedure2`, `CTProcedure3`, `CTProcedure4` ja `CTProcedure5`. Kuvassa 5.6 on esimerkkejä Cottontail Schemen funktionaalisista rajapinnoista. Jokaiselle parametrien lukumäärälle nollasta viiteen on siis oma funktionaalinen rajapintansa. Esimerkiksi `CTProcedure2` määrittelee `apply`-proseduurin, joka ottaa parametreinaan kaksi `Object`-tyyppistä arvoa.

Funktionaaliset rajapinnat

```
@FunctionalInterface
public interface CTProcedure {
    Object apply(Object[] args);
}

@FunctionalInterface
public interface CTProcedure0 {
    Object apply();
}

@FunctionalInterface
public interface CTProcedure2 {
    Object apply(Object a, Object b);
}
```

Kuva 5.6: Osa Cottontail Schemen JVM-toteutuksen funktionaalisista rajapinnoista

Proseduurit, jotka ottavat enemmän kuin viisi parametria, esitetään suoraan taulukon parametrinaan ottavalla funktionaalisella rajapinnalla `CTProcedure`. Tämä vaatii

proseduurien käännösvaiheessa hiukan ylimääräistä työtä, jota tarkastellaan myöhemmin. Varargs-proseduurit puolestaan mukautuvat `CTProcedure1`-rajapintaan, koska ne sallivat käytännössä yhden parametrin, jonka tulee olla lista.

Dynaamisen tyypityksen hallinta ensimmäisen luokan funktiokutsuissa

Koska Scheme on dynaamisesti tyypitetty kieli, mitä tahansa ensimmäisen luokan arvona käytettävää proseduuria, joka ei ota vaihtelevaa määrää argumentteja, on mahdollista kutsua suoritusaikana virheellisellä argumenttimäärällä. Lisäksi varargs-proseduurit vaativat listan rakentamisen niille annetuista parametreista. Koska varargs-proseduurit esitetään samalla funktionaalisella rajapinnalla kuin yksiparametriset proseduurit, ei ensimmäisen luokan funktioarvoa kutsuttaessa voida tietää, onko kyseessä varargs-proseduuri.

Varargs-proseduurien ongelma olisi mahdollista ratkaista erilaisilla tyypitysvalinnoilla — esimerkiksi esittämällä varargs-prosedurit omalla rajapinnallaan — ja ensimmäisen luokan proseduurikutsujen yhteyteen lisättävillä tyyppitarkastuksilla ja argumenttimäärän tarkastuksilla. Toisaalta ennen jokaista ensimmäisen luokan funktiokutsua tehtävät tyyppitarkastukset ja argumenttimäärän tarkastukset kasvattaisivat tuotettua tavukoodia. Näiden tapauksien käsittelyä varten Cottontail Scheme määrittelee kuvassa 5.7 listatun joukon apumetodeja.

```
public static CTProcedure match0(CTProcedure0 proc, String procedureName);
public static CTProcedure match1(CTProcedure1 proc, String procedureName);
public static CTProcedure match2(CTProcedure2 proc, String procedureName);
public static CTProcedure match3(CTProcedure3 proc, String procedureName);
public static CTProcedure match4(CTProcedure4 proc, String procedureName);
public static CTProcedure match5(CTProcedure5 proc, String procedureName);
public static CTProcedure matchN(CTProcedure proc, int arity,
    String procedureName);
public static CTProcedure matchVarargs(CTProcedure1 proc);
```

Kuva 5.7: Ensimmäisen luokan proseduurikutsujen argumenttimäärien tarkastukseen ja argumenttien purkamiseen käytettävät apumetodit.

Apumetodit ottavat metodista riippuen parametrina `CTProcedure0`–`5`- tai `CTProcedure`-tyyppisen proseduuriolion. Muiden kuin varargs-proseduurien kohdalla otetaan parametrina lisäksi proseduurin nimi virheilmoituksia varten. Metodit `match0`–`match5` ja `matchN` käärivät parametrina saamansa proseduuriolion uuteen Java 8 -sulkeumaan, joka mukautuu rajapintaan `CTProcedure`. Tämä sulkeuma tarkistaa `Object`-taulukossa saamiensa argumenttien määrän. Mikäli argumenttien määrä on oikea, puretaan argumentit taulukosta ja kutsutaan alkuperäistä käärittyä sulkeumaa. Metodi `matchN`, jota

käytetään yli viisiparametristen proseduurien kanssa poikkeaa muista `match`-metodeista siten, että se antaa purkamattoman taulukon argumenttina kääritylle proseduurille. Jos argumenttien määrä ei täsmää funktion odottamaan parametrimäärään, heitetään `ArityMismatchError`-poikkeus. Kuvassa 5.8 on esimerkkinä lievästi yksinkertaistettu esitys metodien `match1` ja `matchN` toteutuksista Cottontail Schemen JVM-alustan tukikirjastossa.

```
// Argumenttimäärän tarkastus yksiparametrisille sulkeumafunktioille
public static CTProcedure match1(CTProcedure1 proc,
    String procedureName) {
    return (Object[] args) -> {
        // Argumenttimäärän tarkastus
        checkArgs(procedureName, 1, args);

        // Kädityn sulkeuman (proc) kutsu taulukosta
        // puretuilla argumenteilla
        return proc.apply(args[0]);
    };
}

// Argumenttimäärän tarkistus yli viisiparametrisille
// sulkeumafunktioille
public static CTProcedure matchN(CTProcedure proc,
    int n, String procedureName) {
    return (Object[] args) -> {
        checkArgs(procedureName, n, args);
        return proc.apply(args);
    };
}

private static void checkArgs(String procedureName, int expectedArity,
    Object[] args) {
    if (args.length != expectedArity)
        throw new ArityMismatchError(procedureName, expectedArity,
            args.length);
}
```

Kuva 5.8: Argumenttimäärän tarkastuksen ja argumenttitaulukon purkamisen toteutus metodissa `match1` sekä yli viisiparametristen funktioiden yhteydessä käytettävän `matchN`-metodin toteutus.

Proseduuri `matchVarargs` puolestaan käärii saamansa proseduuriolion sulkeumaan, joka kutsuu `BuiltIns`-luokassa määriteltyä `list`-proseduuria. Tällä tavoin proseduurin parametrina saama taulukko muunnetaan `CTPair`-tyyppiseksi listaksi. Jokaista sulkeuman luontia seuraa siis Cottontail Schemen tuottamassa tavukoodissa kutsu yhteen näistä apumetodeista.

Kaikille sisäänrakennetuille proseduureille on määritelty valmiit proseduurioliot osana kirjastototeutusta. Nämä käyttäytyvät pääpiirteittäin samalla tavoin kuin käyttäjän määrittelemät proseduurioliot. Varargs-proseduurien kohdalla ei kuitenkaan tarvita erillistä käärintää muunnosproseduuriin `matchVarargs`, koska sisäänrakennetut proseduurit ottavat jo valmiiksi parametrina taulukon. Tällöin sisäänrakennettu varargs-proseduuri saadaan sellaisenaan mukautumaan `CTProcedure`-rajapintaan.

Toinen dynaamisen tyypityksen aiheuttama haaste on se, että yritys tehdä ensimmäisen luokan proseduurikutsu voi epäonnistua suoritusaikana siksi, että kutsuttava arvo ei olekaan proseduuri. Tämä ongelma on kuitenkin helppo ratkaista suoritusaikaisella tyyppitarkastuksella. Tämän vuoksi Cottontail Scheme suorittaa kutsut sisäänrakennetun `callProcedure`-apumetodin avulla, joka suorittaa tarvittavan tyyppitarkastuksen ennen kutsua ja tarvittaessa heittää poikkeuksen tyyppiä `NotAProcedureError`. Tuotantokäyttöön tehdyt kääntäjät kuten Kawa ja Bigloo tyypillisesti optimoivat koodia luomalla tyyppitarkastuksen suoraan osaksi kutsuvan metodin rungon koodia, jolloin vältetään ylimääräinen metodikutsu.

Muuttujia kaappaamattomien funktioarvojen toteutus

Kuvassa 5.9 on esimerkki ylätason proseduurista, jota käytetään ensimmäisen luokan arvona antamalla se parametrina toiselle proseduurille. Tällaisessa tapauksessa proseduuri `double` kääntyy samassa kuvassa esitettyä Java-koodia vastaavaksi JVM-tavukoodiksi.

```
(define double
  (lambda (x)
    (+ x x)))

(map double '(1 2 3))
```

```
private static Object Double$1(Object a) {
    return BuiltIns.plus(new Object[] {a, a});
}
```

Kuva 5.9: Ylätason proseduuri ensimmäisen luokan arvona

Jotta proseduuria voidaan käyttää ensimmäisen luokan arvona, on saatava sille esitys, joka mukautuu funktionaaliseen rajapintaan `CTProcedure1`. Tämä saadaan aikaan käyttämällä luvussa 3.3 esiteltyä `invokedynamic`-kutsua, jonka bootstrap-metodina toimii JVM-ympäristön sisäänrakennettu lambda-metatehdasmetodi. Lambda-metatehtaita esiteltiin Java 8:n osalta luvussa 4.2.

`Invokedynamic`-käskyn yhteydessä määritellään, mihin funktionaaliseen rajapintaan

```
// CTProcedure1-rajapintaan mukautuvan funktioarvon luonti
0: invokedynamic #106,  0 // InvokeDynamic #0:apply:
                          //   ()Llib/CTProcedure1;

// Kääriminen argumenttitarkistuksen tekevään sulkeumaan
5: ldc           #108 // String double
7: invokestatic  #112 // Method lib/ProcedureHelpers.match1:
                      //   (Llib/CTProcedure1;Ljava/lang/String;)
                      //   Llib/CTProcedure;
```

```
BootstrapMethods:
  0: #98 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
                      (Ljava/lang/invoke/MethodHandles$Lookup;
                      Ljava/lang/String;
                      Ljava/lang/invoke/MethodType
                      ;Ljava/lang/invoke/MethodType;
                      Ljava/lang/invoke/MethodHandle;
                      Ljava/lang/invoke/MethodType;)
                      Ljava/lang/invoke/CallSite;
Method arguments:
  #99 (Ljava/lang/Object;)Ljava/lang/Object;
  #102 invokestatic Double.Double$1:(Ljava/lang/Object;)Ljava/lang/Object;
  #99 (Ljava/lang/Object;)Ljava/lang/Object;
```

Kuva 5.10: Double-funktioarvon luonti (yllä) ja invokedynamic-käskyn bootstrap-parametrit (alla)

sulkeumaesityksen halutaan mukautuvan. Ensimmäisellä kerralla `invokedynamic`-käskyä suoritettaessa suoritusympäristö kutsuu lambda-metatehdasmetodia, joka palauttaa kutsupaikkaolion. Tämä kutsupaikkaolio ilmoittaa metodin, jota kutsumalla haluttu funktioarvo voidaan luoda. JVM-virtuaalikoneen toteutus siis määrittää käytetyn sulkeumaesityksen.

Kuvassa 5.10 on esitetty sekä funktio-olion luonti tavukoodina että bootstrap-metodina toimivalle lambda-metatehdasmetodille annettavat parametrit. Kuten edellä mainittiin, `invokedynamic`-käskyn avulla tehtävää sulkeuman luontia seuraa kutsu yhteen `match`-apumetodeista, joka käärii funktioarvon argumenttimäärän tarkastuksen tekevään sulkeumaan.

Bootstrap-metodin parametreista nähdään, että `invokedynamic`-kutsu luo funktioarvolle esityksen, jonka `apply`-metodi ottaa parametrinaan yhden `Object`-tyyppisen arvon ja palauttaa `Object`-tyyppisen arvon. Parametrit määrittävät myös, että funktion runko toteutetaan `Double`-nimisessä luokassa sijaitsevan `Double$1`-metodin avulla. Nämä argumenttirivit on korostettu kuvassa 5.10.

Paikallisten muuttujien kaappaus

Myös muuttujien kaappaus pystytään tekemään suoraan lambda-metatehdasta käyttäen. Tällöin sulkeuman rungon toteuttavalle metodille tehdään sulkeumamuunnos, jossa kaapattavat muuttujat lisätään sen parametreiksi. Lambda-metatehdasta kutsuttaessa ladataan lisäksi kaapattavien muuttujien arvot pinoon ennen `invokedynamic`-käskyä.

Koska lambda-metatehdas kaappaa muuttujat sellaisenaan, tarvitaan jokin lisäratkaisu, joka mahdollistaa saman muuttujan jakamisen useamman sulkeuman välillä. Cottontail Scheme käyttää toteutustekniikkana viitesoluja, jotka on toteutettu `CTReferenceCell`-nimisenä luokkana. `CTReferenceCell` määrittelee metodit `get` ja `set`. Käytännössä proseduurin rungon luonti etenee seuraavalla tavalla:

1. Proseduurin p rungosta etsitään kaikki proseduurimäärittelyt ja anonyymit proseduurit c_1, ..., c_n.

2. Löydettyjen proseduurien c_1, ..., c_n kaappauslistoista muodostetaan yhdiste C.

3. Proseduurin p rungon alkuun luodaan koodi, joka käärii jokaisen kaappauslistojen yhdisteeseen C kuuluvan muuttujan `CTReferenceCell`-tyyppiseen olioon. Jos muuttuja on vapaa proseduurin p rungossa, eli se on kaapattu jostakin ympäröivästä näkyvyysalueesta, se on jo saatu `CTReferenceCell`-tyyppisenä parametrina, eikä uusi käärintä ole tarpeellista.

4. Sisäkkäisten proseduurien c_1, ..., c_n rungot lambda-nostetaan staattisiksi metodeiksi luokan ylätasolle. Sulkeumamuunnoksessa jokaista kaapattua muuttujaa kohti lisätään metodin parametrilistan alkuun uusi `CTReferenceCell`-tyyppinen parametri.

Varsinaiset sulkeumat luodaan pääpiirteittäin samaan tapaan kuin muuttujia kaappaamattomat proseduurioliot. Kuvassa 5.11 on esimerkki yksinkertaiselle sulkeumalle tuotettua tavukoodia vastaavasta Java-koodista. Sulkeuman rungon toteutusta vastaavaan metodiin on siis lisätty uusi `CTReferenceCell`-tyyppinen parametri, joka vastaa kaapattua muuttujaa `x`.

Kuvan 5.11 sulkeuman luonti proseduurissa `add` kääntyy kuvassa 5.12 esitetyksi tavukoodiksi. Koodissa luodaan uusi `CTReferenceCell`-olio, johon kääritään ensimmäisenä parametrina saatu arvo. Viitesolun luonnin jälkeen se sijoitetaan takaisin parametrin paikalle, mikä ei tosin juuri tämän funktion tapauksessa olisi tarpeellista, koska arvo ladataan välittömästi uudelleen sulkeuman luontia varten. Mikäli ensimmäiseen parametriin olisi kuitenkin myös muita viittauksia funktion rungossa, ne muunnettaisiin käyttämään viitesolun `get`- ja `set`-metodeita.

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
private static Object Lambda$1(CTReferenceCell a1, Object a2) {
    return BuiltIns.plus(new Object[] {a1.get(), a2});
}
```

Kuva 5.11: Yksinkertainen sulkeuma ja sulkeuman rungosta luotua metodia vastaava Java-koodi

Koska kaikkien kaapattavien muuttujien arvoja ei muuteta, olisi mahdollista optimoida koodinluontia siten, että viitesoluja käytettäisiin vain silloin, kun kaapatun muuttujan tunniste esiintyy `set!`-lausekkeessa kohteena. Cottontail Schemessä viitesoluja kuitenkin käytetään kaikkien kaapattujen arvojen esittämiseen.

```
private static java.lang.Object Add$1(java.lang.Object);
  descriptor: (Ljava/lang/Object;)Ljava/lang/Object;
  flags: ACC_PRIVATE, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
        // Viitesolun luonti ja parametrina saadun arvon käärintä siihen
        0: new           #14  // class lib/CTReferenceCell
        3: dup
        4: aload_0
        5: invokespecial #29  // Method lib/CTReferenceCell."<init>":
                              //   (Ljava/lang/Object;)V

        // Viitesoluun käärityn arvon takaisinsijoitus
        8: astore_0

        // Sulkeuman luonti alkaa kaapattujen muuttujien
        // latauksella pinoon
        9: aload_0
       10: invokedynamic #44,  0  // InvokeDynamic #0:apply:
                                  //    (Llib/CTReferenceCell;)
                                  //    Llib/CTProcedure1;
        15: ldc           #45 // String Lambda$1
        17: invokestatic  #51 // Method lib/ProcedureHelpers.match1:
                              //    (Llib/CTProcedure1;Ljava/lang/String;)
                              //     Llib/CTProcedure;
        20: areturn
```

Kuva 5.12: Sulkeuman luonti proseduurista `add` luodussa tavukoodissa

Yli viisi parametria ottavien funktioiden käyttö ensimmäisen luokan arvoina

Kuten mainittiin aiemmin, yli viisi parametria ottavat proseduurit esitetään funktionaalisella rajapinnalla `CTProcedure`, jonka `apply`-metodi ottaa parametrinaan taulukon `Object`-tyyppisiä olioita. Mikäli proseduuri on nimetty, sitä voidaan sekä käyttää ensimmäisen luokan arvona että kutsua suoraan nimellä. Tällöin toteutuksella on kaksi vaihtoehtoa, jotka on esitetty kuvassa 5.13: tällainen proseduuri voidaan joko toteuttaa metodina, joka ottaa parametrinaan taulukon (kuvassa oikealla), tai voidaan luoda erillinen taulukon parametrinaan ottava apumetodi, joka purkaa parametrit taulukosta ja antaa ne varsinaiselle proseduurin toteuttavalle metodille (kuvassa vasemmalla).

```
public static Object func(
    Object a1, Object a2,
    Object a3, Object a4,
    Object a5, Object a6) {
    // ...
}

// Parametritaulukon purkava
// apumetodi
public static Object funcHelper(
    Object[] args) {
    return func(
        args[0], args[1],
        args[2], args[3],
        args[4], args[5]);
}
```

```
public static Object func(
    Object[] args) {
    // ...
}
```

Kuva 5.13: Yli viisiparametrisen proseduurin käännösmahdollisuudet: parametrit purkava apumetodi (vasemmalla) ja proseduurin toteutus kokonaan taulukkoparametrisena (oikealla).

Jos proseduuri toteutetaan metodina, joka ottaa parametrinaan taulukon, kaikki kutsupaikat joutuvat pakkaamaan argumentit taulukkoon ennen metodikutsua, vaikka kyse ei olisi ensimmäisen luokan proseduurikutsusta. Lisäksi funktion koodi on luotava siten, että sen tekemät viittaukset sen parametreihin viittaavat paikallisten muuttujien sijasta parametrina saadun taulukon indekseihin. Tämä vie enemmän tavukoodikäskyjä ja on todennäköisesti paikallisten muuttujien käyttöä hitaampaa.

Mikäli luodaan erillinen apumetodi, joka purkaa argumentit taulukosta, joudutaan luomaan hieman enemmän koodia, mutta varsinaisen päämetodin runko voi viitata tavalliseen tapaan paikallisiin muuttujiinsa, ja metodia suoraan nimellä kutsuvat kutsupaikat ovat yksinkertaisempia. Cottontail Schemen toteutuksessa päädyttiin jälkimmäiseen ratkaisuun. Koska JVM-alustalla ei ole mahdollista luoda yleiskäyttöistä apumetodia, joka

voisi antaa käännösaikana tuntemattoman määrän parametreja toiselle metodikutsulle, on jokaiselle tällaiselle proseduurille luotava oma kuvassa 5.13 esitetyn `funcHelper`-metodin kaltainen apumetodi. Tällöin luotavat funktioarvot käyttävätkin funktion rungon toteutuksena luotua apumetodia varsinaisen funktion toteuttavan metodin sijaan.

Häntäkutsujen toteutus

Koska JVM ei tarjoa erillistä tukea häntäkutsuille, ainoa tapa toteuttaa häntäkutsut on käyttää jotakin muunnelmaa trampoliinitekniikasta. Cottontail Scheme toteuttaa yksinkertaisen trampoliiniapumetodin, jonka avulla häntäkutsut optimoidaan.

Cottontail Schemen toteutuksessa häntäpositiossa olevan proseduurikutsun sisältävä proseduuri kääntyy muotoon, jossa häntäkutsun sijasta palautetaan `CTTailContinuation`-olio. `CTTailContinuation` on yksinkertainen olio, joka sisältää ainoastaan `value`-kentän, johon talletetaan kutsuttava proseduuri. Kenttä on tyyppiä `CTProcedure0`, koska sen kaikkien parametrien tulee olla ennalta sidottuja, jolloin sitä voidaan kutsua ilman parametreja. Käytännössä häntäkutsun sisältävä proseduuri siis osittainsoveltaa kutsumaansa proseduuria kutsuparametreihin.

Itse trampoliinimetodi on yksinkertainen silmukka, joka kutsuu `CTTailContinuation`-olion `value`-kentän arvona olevaa proseduuria, kunnes se saa paluuarvon, joka ei enää ole tyyppiä `CTTailContinuation`. Trampoliinimetodin koodi on kuvassa 5.14.

```
public static Object trampoline(Object v) {
    Object currentValue = v;
    while (currentValue instanceof CTTailContinuation) {
        currentValue = ((CTTailContinuation) currentValue).value.apply();
    }
    return currentValue;
}
```

Kuva 5.14: Cottontail Schemen käyttämä trampoliiniapumetodi

Myös häntäpositiossa oleva kutsu voidaan tehdä ensimmäisen luokan funktioon. Tämän vuoksi parametreja sidottaessa tarkistetaan, onko kutsuttava arvo proseduuri. Koska kaikki funktioarvot esitetään `CTProcedure`-tyyppisinä arvoina, tarkistuksen toteutus on yksinkertainen. Kuvassa 5.15 on kirjastofunktio, jonka avulla `CTprocedure`-proseduuriolion argumentit sidotaan. Sidonta tehdään käärimällä funktioarvon kutsu uuteen Java 8 -sulkeumaan.

Käytännössä siis jokainen häntäpositiossa oleva kutsu kääntyy `CTTailContinuation`-olion palautukseksi ja jokainen muussa kuin häntäpositiossa oleva kutsu tehdään trampoliinimetodin avulla sen varalta, että proseduuri johtaa häntäkutsuihin. Esimerkiksi

```
public static CTProcedure0 bindArgs(Object proc, Object[] args) {
    if (proc instanceof CTProcedure) {
        CTProcedure p = (CTProcedure) proc;
        return () -> p.apply(args);
    } else {
        throw new NotAProcedureError(proc);
    }
}
```

Kuva 5.15: Proseduurin osittainen soveltaminen häntäkutsun argumentteihin ja tyyppitarkistus `CTProcedure`-oliolle

tuttu esimerkki kokonaisluvun kertoman laskennasta kääntyisi yleistettyjä häntäkutsuja käytettäessä kuvan 5.16 Java-koodia muistuttavaan muotoon. Kuvan esimerkissä apufunktio `fact-acc` on nostettu ylätasolle `fact`-funktion rinnalle, koska se yksinkertaistaa tuotettua tavukoodia. Normaalisti toteutus kääntäisi kuvan häntärekursiivisen apufunktion silmukan kaltaiseksi koodiksi. Toteutuksessa on kuitenkin mahdollista kytkeä häntärekursion erityiskäsittely pois päältä testitarkoituksia varten.

Toteutusta olisi mahdollista optimoida käyttämällä esimerkiksi Schinzin ja Oderskyn Funnel-kääntäjää varten kehittämää tekniikkaa, jolloin koodi kävisi trampoliinissa harvemmin. Cottontail Schemessä toteutus on kuitenkin pidetty yksinkertaisena, koska tutkielman pääasiallisena tarkoituksena ei ole erityisesti suorituskyvyn optimointimenetelmien tarkastelu.

5.3 Koodinluonti CLI-alustalla

Koodinluonti yksinkertaisimmille Scheme-ohjelmille on CLI-alustalla hyvin samankaltaista kuin JVM-alustalla. Ohjelma kääntyy luokaksi, jossa ylätason proseduurit ovat staattisia metodeita. Häntärekursiiviset funktiot käännetään silmukan kaltaiseksi koodiksi samoin kuin JVM-alustalla.

CLI-alustalla yleistettyjen häntäkutsujen toteutus on selvästi yksinkertaisempi kuin JVM-alustalla, sillä häntäkutsujen yhteydessä kutsukäskyn eteen yksinkertaisesti lisätään `tail.`-etuliite. Häntäkutsuja käyttävälle ohjelmalle luotu koodi on siis hyvin samankaltaista kuin esimerkiksi kuvassa 4.18 (sivu 50) esitetty F#-kääntäjän luoma koodi keskinäisen häntärekursion tapauksessa.

Alustalla tarjolla olevat vaihtoehdot sulkeumien toteuttamiseen ovat käytännössä erilaiset oliosulkeumatoteutukset, Bigloon ja Kawan tyyliset delegaattimaiset toteutukset sekä CLI-alustan omat delegaatit. CLI-alustalla luontevimmalta sulkeumien toteutustavalta vaikuttaisivat alustan omat delegaattiluokat. Cottontail Schemen CLI-toteutuksessa päädyttiin kuitenkin Kawa-tyyliseen delegaattimaiseen toteutukseen, sillä yleistettyjen

```
(define fact
  (lambda (n)
    (fact-acc n 1)))

(define fact-acc
  (lambda (n acc)
    (if (zero? n)
        acc
        (fact-acc (- n 1)
                  (* n acc)))))
```

```
class Fact {
    private static Object fact(Object n) {
        // Proseduuriolion luonti ja argumenttien pakkaus taulukkoon
        CTProcedure factAccProc = ProcedureHelpers.match2(Fact::factAcc);
        Object[] args = new Object[] { n, new CTNumber(1) };

        // Argumenttien sidonta eli proseduurin osittainen soveltaminen
        CTProcedure0 proc = ProcedureHelpers.bindArgs(factAccProc, args);

        // Kontinuaatio-olion palautus
        return new CTTailContinuation(proc);
    }

    private static Object factAcc(Object n, Object acc) {
        if (BuiltIns.isZero(n)) {
            return acc;
        } else {
            CTProcedure factAccProc = ProcedureHelpers.match2(
                Fact::factAcc);

            // Rekursiivisen kutsun argumenttien laskenta
            // ja pakkaus taulukkoon
            Object[] args = new Object[] {
                BuiltIns.minus(new Object[] { n, new CTNumber(1) }),
                BuiltIns.mult(new Object[] { n, acc })
            };

            CTProcedure0 proc = ProcedureHelpers.bindArgs(
                factAccProc, args);
            return new CTTailContinuation(proc);
        }
    }

    // ...
    // Kutsu esimerkiksi main-metodissa.
    ProcedureHelpers.trampoline(fact(new CTNumber(42)));
    // ...
}
```

Kuva 5.16: Trampoliinin käyttö `fact`-proseduurin käännöksessä luotua tavukoodia vastaavana Java-koodina

häntäkutsujen toteutus delegaattiluokkien yhteydessä osoittautui mahdottomaksi, jos käytetään alustan omaa häntäkutsutukea. Sulkeumaesityksen valintaa tarkastellaan tarkemmin luvussa 5.4.

Sulkeumaesitys

Koska sulkeumaesitys on CLI-alustalla toteutettu kokonaan itse, se on selvästi JVM-alustalla käytettyä sulkeumatoteutusta monimutkaisempi. Sulkeumien esitykseen tarvittavien luokkien hierarkia on esitetty kuvassa 5.17.

CLI-toteutuksessa sulkeumaa esittävällä luokalla `CTProcedure` on erilliset `apply`-metodit parametrimäärille 0–5. Lisäksi proseduuriolioilla on metodi `applyN`, joka ottaa parametrinaan mielivaltaisen kokoisen taulukon `CTObject`-arvoja.

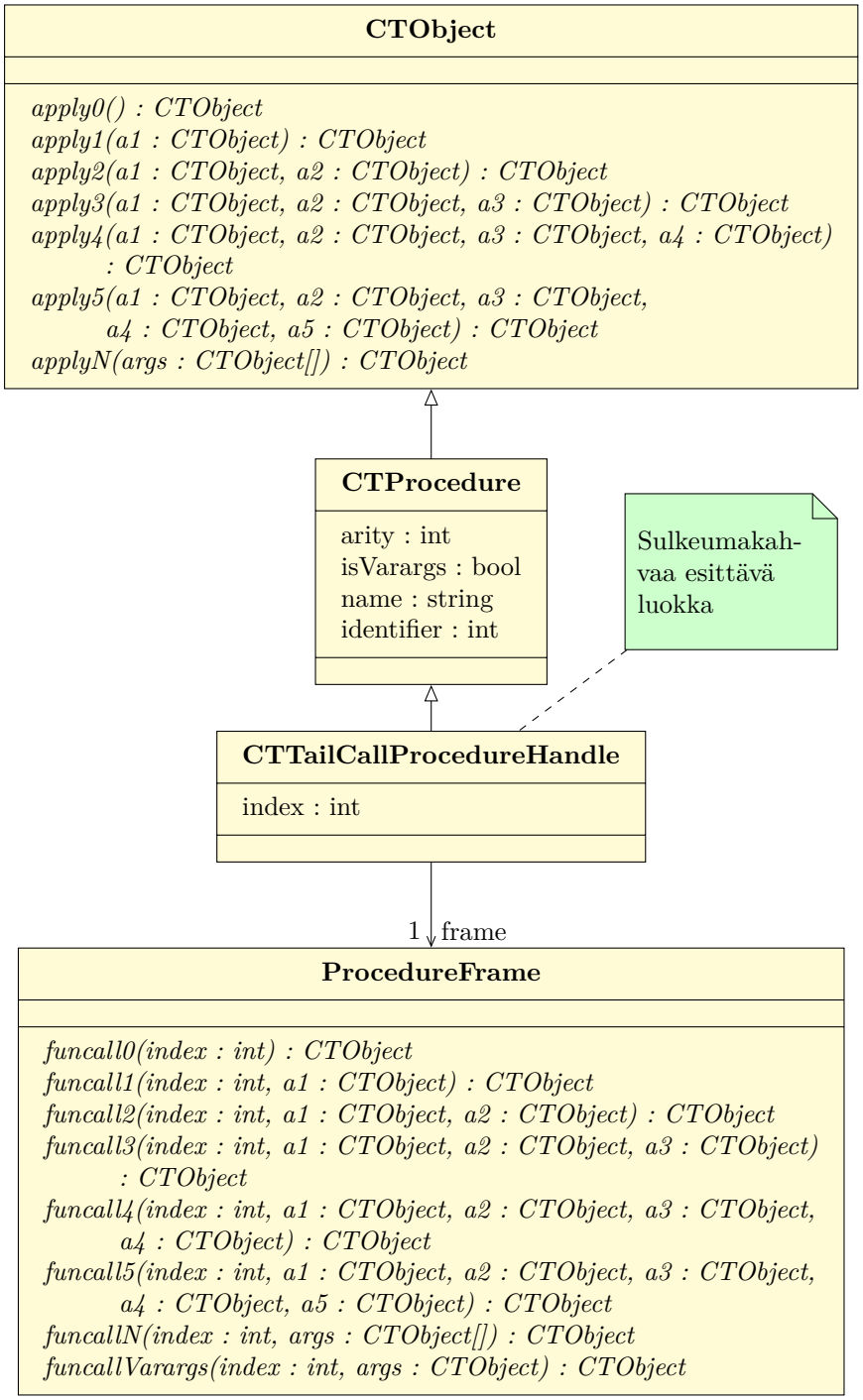
`Apply`-metodien toiminta on pitkälti samankaltaista kuin `match`-metodien JVM-toteutuksessa. Mikäli kyseessä ei ole varargs-proseduuri, ne vertaavat saamiensa parametrien määrää `CTProcedure`-olion `arity`-kenttään talletettuun parametrimäärään. Mikäli parametrimäärät poikkeavat toisistaan, heitetään poikkeus. Muussa tapauksessa `apply`-metodi kutsuu haluttua metodia. Metodikutsun toteutus riippuu proseduuriolion tyypistä.

Kaikille `apply`-metodeille on `NotAProcedureError`-poikkeuksen heittävä oletustoteutus `CTObject`-kantaluokassa. Tällä tavoin käsitellään helposti tapaukset, joissa yritetään kutsua arvoa, joka ei ole proseduuri.

Sisäänrakennettujen proseduurien tapauksessa sulkeuman sisäisenä toteutuksena käytetään `CTProcedure`-luokasta periviä `CTDelegateProcedure0`–`5` ja `CTDelegateProcedureVarargs`-luokkia, jotka käärivät geneerisen `Func`-tyyppisen delegaattiolion. Tällöin `apply`-metodi kutsuu käärimäänsä delegaattioliota.

Käyttäjän määrittelemien proseduurien käyttö ensimmäisen luokan arvoina on toteutettu Kawa-kääntäjän käyttämää tyyliä muistuttavalla delegaattityylillä. Koska käyttäjän on voitava tehdä häntäkutsuja myös ensimmäisen luokan arvoina käytettäviin funktioihin, tarvitaan erillinen `CTProcedure`-luokan aliluokka, joka tukee häntäkutsujen tekemistä. Koska yleisiä häntäkutsuja ei ole mahdollista tehdä suoraan C#- tai edes F#-koodista käsin, tämä luokka täytyy luoda tavukoodina. Häntäkutsutoiminnallisuuden toteuttava luokka on nimeltään `CTTailCallProcedureHandle`, ja Cottontail Schemen prototyyppitoteutuksessa se luodaan osaksi käännetyn ohjelman tavukoodia.

`CTTailCallProcedureHandle`-olio tarvitsee `CTProcedure`-luokkaan talletettavien tietojen lisäksi viitteen `ProcedureFrame`-tyyppiseen olioon sekä numeerisen indeksin, joka toimii kutsuttavan metodin tunnisteena. `CTTailCallProcedureHandle`-oliot muistuttavat käyttötarkoitukseltaan ja toteutukseltaan Kawan `ModuleMethod`-olioita, eli niiden tarkoitus on osoittaa funktioiden runkojen toteutukset sisältävästä pakomuuttujatie-

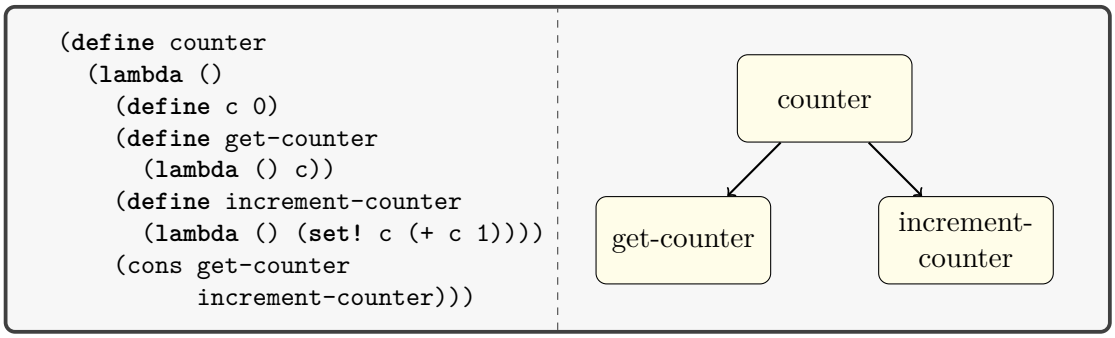


Kuva 5.17: Keskeisimmät proseduurien esitykseen liittyvät tyypit ja niiden tärkeimmät metodit ja kentät

tueluokasta kutsuttava metodi. Kawan delegaattitoteutus esitettiin aiemmin kuvassa 4.24 (sivu 56). Pakomuuttujatietueina Cottontail Schemen CLI-toteutuksessa toimivat `ProcedureFrame`-tyyppiset oliot.

Proseduurin käännösprosessi

Sisäkkäisten proseduurimäärittelyjen voi ajatella muodostavan puun, jossa ylätason proseduuri on juurisolmu ja sisimmillä tasoilla määritellyt sulkeumat ovat lehtisolmuja. Kuvassa 5.18 on esimerkki yksinkertaisesta sulkeumien muodostamasta puurakenteesta aiemminkin nähdyn `counter`-esimerkin tapauksessa. Esimerkin tapauksessa puussa on vain kaksi tasoa, mutta yleisessä tapauksessa sulkeumat voivat muodostaa mielivaltaisen syvän puun.



Kuva 5.18: Sulkeumien muodostama puu

Proseduurin käännös on rekursiivinen prosessi, joka suoritetaan syvyyssuuntaisena läpikäyntinä sisäkkäisten sulkeumien muodostamalle puulle. Sisimpien sulkeumien eli puun lehtisolmujen koodi luodaan ensimmäisenä, ja koodinluonti etenee kohti juurisolmuja. Yksittäisen proseduurin tai sulkeuman p rungon käännösprosessi etenee pääpiirteittäin seuraavasti:

1. Kuten JVM-alustan toteutuksessa, ensin etsitään proseduurin p rungosta anonyymit proseduurit ja proseduurimäärittelyt c_1, ..., c_n. Kuvan 5.18 esimerkkitapauksessa proseduurin `counter` rungosta löydetään proseduurimäärittelyt `get-counter` ja `increment-counter`.

2. Löydettyjen proseduurien c_1, ..., c_n kaappauslistoista muodostetaan yhdiste C, joka on proseduurin p pakenevien muuttujien joukko. Esimerkiksi kuvan 5.18 tapauksessa proseduurin `counter` pakenevien muuttujien joukkoon kuuluu ainoastaan muuttuja `c`.

3. Kääntäjä luo uuden `ProcedureFrame`-luokan aliluokan, joka toimii pakomuuttujatietueena.

- Luokkaan lisätään kenttä jokaiselle muuttujalle, joka kuuluu pakenevien muuttujien joukkoon C ja joka ei ole vapaa proseduurin p rungossa.
- Mikäli osa pakenevista muuttujista on vapaita proseduurin p rungossa, eli ne on määritelty jossakin ympäröivässä näkyvyysalueessa, lisätään luokkaan myös viite ympäröivän näkyvyysalueen `ProcedureFrame`-olioon. Tämän viitteen kautta sulkeuma voi viitata ympäröivään näkyvyysalueeseen. Viittaukset näihin vapaisiin muuttujiin tehdään ympäröivän näkyvyysalueen viitteen kautta.
- Jokaista löydettyä proseduuria c_1, ..., c_n kohti lisätään luokkaan metodi.
- Proseduurit c_1, ..., c_n numeroidaan.
- Kääntäjä korvaa `ProcedureFrame`-luokassa määritellyt `funcall`-metodit. Metodit ottavat yhtenä parametrinaan edellisessä vaiheessa luodun numeerisen indeksin, joka toimii metodin tunnisteena. Kutsuttava metodi valitaan `switch`-lauseella ja kutsu suoritetaan häntäkutsuna.

4. Proseduurin rungon koodi luodaan.

Lähes jokaista sisäkkäisiä proseduurimäärittelyjä sisältävää proseduuria kohti siis luodaan uusi luokka, joka on `ProcedureFrame`-luokan aliluokka. Varsinaiset sulkeumat muodostetaan luomalla `CTTailCallProcedureHandle`-olioita, joille annetaan parametrina proseduurin numeerinen indeksi ja viite `ProcedureFrame`-olioon. Kuvassa 5.19 on esimerkki kuvan 5.18 esimerkkikoodin tavukoodikäännöstä pääpiirteittäin vastaavasta C#-koodista.

Esimerkkitapauksen proseduuriolioita kutsutaan `CTObject`-luokassa määritellyillä `apply`-metodeilla. Metodien toteutus `CTTailCallProcedureHandle`-luokassa tarkistaa saadun parametrimäärän vertaamalla sitä `arity`-kenttään tallennettuun odotettuun parametrimäärään. Mikäli parametrien määrä on oikea, delegoidaan kutsu suoraan vastaavalle `funcall`-metodille `CounterFrame`-luokassa. Esimerkiksi `CTTailCallProcedureHandle`-luokan `apply0`-metodi siis tekee tällöin häntäkutsun `CounterFrame`-olion `funcall0`-metodiin, jolle annetaan parametrina `CTTailCallProcedureHandle`-delegaattiolion sisältämä numeerinen metoditunniste.

Yli viisiparametriset funktioarvot käyttävät `funcallN`-metodia. JVM-alustalla käytetyn kaltaista argumentit taulukosta purkavaa erillistä apumetodia ei tarvita, koska argumenttien purku voidaan tehdä korvatun `funcallN`-metodin rungossa. Kuvassa 5.20 on esimerkki argumenttien purusta `funcallN`-metodissa kuusiparametriselle proseduurille.

```
// Pakomuuttujatietue.
public class CounterFrame : ProcedureFrame
{
    // Kaapattu muuttuja
    public CTObject c;

    public CTObject GetCounter() { return c; }

    public CTObject IncrementCounter() {
        c = BuiltIns.plus(new CTObject[]{c, new CTNumber(1)});
        return Constants.Undefined;
    }

    public override CTObject funcall0(int index) {
        switch (index) {
        case 1:
            return GetCounter(); // tail. call -käsky
        case 2:
            return IncrementCounter(); // tail. call -käsky
        default:
            return base.funcall0(index);
        }
    }
}

// Counter-proseduurin toteutus
public static CTObject Counter() {
    CounterFrame f = new CounterFrame();
    f.c = new CTNumber(0); // Muuttujan c alustus

    // Proseduurikahvan luonti parametrisoidaan ProcedureFrame-oliolla,
    // kutsuttavan metodin numeerisella tunnisteella (1 ja 2) ja funktion
    // parametrien lukumäärällä (molempien funktioiden tapauksessa 0).
    CTTailCallProcedureHandle getCounter =
        new CTTailCallProcedureHandle(f, 1, 0);
    CTTailCallProcedureHandle incrementCounter =
        new CTTailCallProcedureHandle(f, 2, 0);

    return BuiltIns.Cons(getCounter, incrementCounter);
}
```

Kuva 5.19: Pakomuuttujatietueen esitys yksinkertaisen sulkeuman tapauksessa

```
public class LambdaFrame : ProcedureFrame
{
    // Kuusiparametrinen metodi, joka toteuttaa
    // sulkeuman rungon.
    public CTObject Lambda1(CTObject a1, CTObject a2, CTObject a3,
        CTObject a4, CTObject a5, CTObject a6) {
        // ...
    }

    public override CTObject funcallN(int index, CTObject[] args) {
        switch (index) {
        case 1:
            // Argumenttien purku taulukosta
            return Lambda1(args[0], args[1], args[2],
                args[3], args[4], args[5]);
        default:
            return base.funcallN(index, args);
        }
    }
}
```

Kuva 5.20: Yli viisiparametristen proseduurien käsittely

5.4 Sulkeumaesitysten valinta Cottontail Schemessä

Molemmilla alustoilla toteutusta suunniteltaessa harkittiin useita eri toteutustapoja erityisesti sulkeumille. JVM-alustalla pääasiallisena vaihtoehtona harkittiin alustan tukeman `MethodHandle`-rakenteen käyttöä. CLI-alustalla puolestaan suunnitelmana oli alun perin delegaattiluokkien tai matalan tason metodikahvojen käyttö sulkeumaesityksen toteutuksessa. Tämä aliluku käy läpi vaihtoehtoisten toteutustapojen vertailussa esille tulleita seikkoja, joiden perusteella lopullisiin toteutustapoihin päädyttiin.

Sulkeumaesityksen valinta JVM-alustalla

Lambda-metatehtaiden ohella toinen mahdollisuus sulkeumien toteuttamiseen JVM-alustan sisäänrakennetun tuen avulla on `MethodHandle`-rakenteen käyttö. Hiukan samaan tapaan kuin metodiosoittimet CLI-alustalla, `MethodHandle` näyttää olevan eräänlainen matalamman tason ominaisuus, jota lambda-metatehdaskin käyttää toteutuksessaan [Ora16b]. `MethodHandle`-rakenteet ovat olleet JVM-alustalla hiukan lambda-metatehtaita pitempään, jo Java SE 7 -versiosta alkaen.

Suoraan `MethodHandle`-olioiden päälle rakentuvaa toteutustapaa kokeiltiin varhaisessa vaiheessa Cottontail Schemen sulkeumatoteutuksessa. `MethodHandle` vaikuttaa olevan melko voimakas työkalu, joka sallii monia toiminnallisuuksia, joita pelkkien sulkeumien

toteuttaja ei välttämättä tarvitse. Tällaisia toiminnallisuuksia ovat esimerkiksi funktion osittaisen soveltamisen mahdollistava `bindTo`-metodi, jota sulkeumien toteuttaja käyttää lähinnä ympäristöstä kaapattujen muuttujien sitomiseen, sekä mahdollisuus liittää metodeihin ennalta määriteltyjä *adaptereita* kuten `asVarargsCollector`, joka muuntaa taulukkomuotoisen parametrin ottavan `MethodHandle`-olion varargs-tyylillä kutsuttavaksi.

Käytännössä toteutustekniikka `MethodHandle`-rakenteen kanssa olisi ollut pitkälti sama kuin lambda-metatehdasta käytettäessä: sulkeuman toteuttavalle metodille tehtäisiin sulkeumamuunnos, kaapatut parametrit käärittäisiin viitesoluolioon ja sidottaisiin `MethodHandle`-olioon yksitellen `bindTo`-metodilla, ja tuloksena saatu `MethodHandle`-olio käärittäisiin vielä toiseen sulkeumaan, joka suorittaa tarvittavat parametritarkistukset. Parametritarkistukset olisi mahdollista jättää myös `MethodHandle`-olion hallittavaksi, sillä se heittää poikkeuksen, jos sitä kutsutaan virheellisellä parametrimäärällä. Tämä kuitenkin olisi ilman jonkinlaista ylimääräistä käsittelyä — esimerkiksi poikkeukset kaappaavan ja käyttäjäystävällimmän poikkeusviestin luovan kääreolion käyttöä — johtanut Cottontail Schemen käyttäjän kannalta epäystävälliseen virheilmoitukseen.

Pääasiallinen syy, jonka vuoksi lambda-metatehtaan käyttöön päädyttiin, oli `MethodHandle`-pohjaisen tekniikan vaatima suhteellisen runsas tavukoodirivien määrä verrattuna lambda-metatehtaan käyttöön. Siinä missä lambda-metatehtaan käyttö vaatii yhden rivin jokaista kaapattua muuttujaa kohti sekä yhden `invokedynamic`-käskyrivin, `MethodHandle` vaatii vähintään noin seitsemän riviä pelkästään `MethodHandle`-olion luomiseksi, minkä lisäksi jokainen muuttujakaappaus vaatii kolme riviä — `MethodHandle`-olion latauksen tai duplikoinnin pinoon, muuttujan latauksen sekä kutsun `bindTo`-metodiin. Koska jokainen kaapattu muuttuja sidotaan erillisellä metodikutsulla, voi `MethodHandle`-olioiden käytöllä olettaa olevan myös vaikutusta suorituskykyyn, mutta tätä ei mitattu toteutustapaa valittaessa.

Lisäksi `MethodHandle`-olion luonnin vaatima rivimäärä riippuu kohteena olevan metodin tyypistä, sillä metodin tyyppi rakennetaan suoritusaikana `MethodHandles.Lookup`-luokan toteuttamaa metodin etsintää varten. `MethodHandle`-rakenteen luontitapa siis käytännössä paisuttaa ohjelmalle tuotettavaa tavukoodia merkittävästi erityisesti ohjelmassa, jossa sulkeumia käytetään runsaasti.

`MethodHandle`-rakenteiden tarjoamat mahdollisuudet voivat olla hyödyllisiä monille dynaamisten kielten toteuttajille, mikäli ei haluta esimerkiksi pakottaa kaikkia sulkeumaolioita ottamaan parametrejaan taulukkomuodossa, sillä `MethodHandle`-rakenteen käärimien metodien ei tarvitse mukautua mihinkään yksittäiseen rajapintaan. Lisäksi ennalta määritellyt adapterit voivat olla joillekin toteuttajille hyödyllisiä. Ilmeisesti joitakin tämän kaltaisia tekniikoita on ollut käytössä ainakin JRuby-kääntäjässä [Nut08], joskaan kyse ei JRubyn kohdalla ilmeisesti ole sulkeumatoteutuksesta.

Toisaalta myös lambda-metatehdas tarjoaa huomattavasti enemmän hienosäätövaraa kuin esimerkiksi Cottontail Schemen toteutuksessa on käytetty. Dokumentaation perusteella ei ole ainakaan välittömästi ilmeistä, voisiko samankaltaisia asioita siis saada aikaan myös lambda-metatehtaan avulla. Näyttää joka tapauksessa siltä, että mikäli tällaisia ominaisuuksia ei ehdottomasti tarvita, lambda-metatehtaan käyttö Java 8:n käyttämällä tyylillä on toteuttajan kannalta huomattavasti yksinkertaisempi ja tiiviimpään tavukoodiin johtava ratkaisu.

Sulkeumaesityksen valinta CLI-alustalla

Häntäkutsujen toteutus CLI-alustalla onnistui suoraviivaisesti kutsukäskyjen `tail.`-etuliitteen avulla, joten JVM-alustalla käytetylle trampoliinitoteutukselle ei ollut tarvetta. Hiukan odottamattomasti häntäkutsujen toteutus tällä menetelmällä kuitenkin aiheutti päänvaivaa sulkeumaesityksen valinnassa.

Alun perin toteutuksessa oli ajatuksena käyttää suoraan CLI-alustan delegaattiluokkia sulkeumien esittämiseen. Tällöin proseduurin runkoa luotaessa rakennettaisiin esimerkiksi pakomuuttujatietue, ja delegaattioliolle annettaisiin parametreina viite pakomuuttujatietueeseen ja metodiosoitin sulkeuman rungon toteuttavaan metodiin. Sama pakomuuttujatietue voitaisiin tällöin jakaa kaikkien saman proseduurin rungossa määriteltyjen sulkeumien kesken.

Huomionarvoista on, että CLI-alustalla ei ole mitään erityistä syytä olla käyttämättä tilan jakamiseen JVM-toteutuksessa käytettyjä viitesoluja, eikä toisaalta JVM-alustalla pakomuuttujatietueita. Pakomuuttujatietue tosin vaatii erillisen luokan määrittelyn jokaiselle näkyvyysalueelle, joten se aiheuttaa hieman viitesoluja enemmän työtä. JVM-alustalla päädyttiin viitesolujen käyttöön juuri niiden yksinkertaisuuden takia, erityisesti yhdessä lambda-metatehtaan toiminnan kanssa.

CLI-alustalla delegaattiluokkien käytössä kuitenkin nousi ongelmaksi se, että CLI-delegaatit eivät näytä tekevän häntäkutsua osoittamaansa metodiin. Huolimatta siitä, että delegaattia kutsutaan häntäkutsulla, esimerkiksi delegaatin avulla toteutettujen keskenään häntärekursiivisten ensimmäisen luokan proseduurien suoritus päättyy suurilla syötteillä pinon ylivuotoon. Näyttää siis siltä, että ensimmäisen luokan funktioihin tehtäviä häntäkutsuja ei voitaisi delegaatteja käytettäessä optimoida lainkaan lisäämättä toteutukseen jotakin trampoliinien kaltaista tekniikkaa. Lopullisessa toteutuksessa osoittimet sisäänrakennettuihin kirjastometodeihin on kuitenkin vielä toteutettu delegaatteina, koska toteutuksen sisältämät kirjastometodit eivät koskaan johda uusiin häntäkutsuihin, jolloin häntäkutsuoptimointia ei tarvita.

Koska standardi näyttää sallivan raakojen metodiosoittimien käytön ja kutsumisen `calli`-käskyllä, oli seuraava ajatus funktioarvojen toteuttamiseksi oman häntäkutsu-

ja käyttävän delegaattityylisen toteutuksen tekeminen. CLI-alustalla raa'at osoittimet voidaan tallettaa alustakohtaisesti määriteltyä `IntPtr`-osoitintyyppiä oleviin kenttiin. Tällainen oma delegaattitoteutus siis taltioisi `IntPtr`-tyyppiseen kenttään metodiosoittimen sekä lisäksi toiseen kenttään viitteen pakomuuttujatietueeseen ja tarjoaisi `apply`-metodin, joka suorittaisi häntäkutsun `IntPtr`-kentän osoittamaan metodiin.

Standardi näyttää kuitenkin johtavan tässä asiassa harhaan, sillä mikä tahansa yritys käyttää `ldftn`-käskyn paluuarvoa muussa yhteydessä kuin delegaattikonstruktorin parametrina johtaa siihen, että CLI-alustan automaattinen tavukooditarkastaja ei hyväksy tavukoodia. Tällaisia ohjelmia ei useimmissa tapauksissa pysty suorittamaan, vaan suoritusyritys päättyy tavukooditarkastajan heittämään poikkeukseen[10].

Oletettavasti .NET-alustan omalla delegaattitoteutuksella tai .NET-kirjastoilla ylipäätään on siis jonkinlainen standardissa dokumentoimaton erityisoikeus näiden käskyjen käyttöön. Bres, Serrano ja Serpette ovat myös harkinneet Bigloon .NET-toteutuksen varhaisessa vaiheessa sulkeumien toteutusta metodiosoittimien avulla, mutta he päätyivät vastaavasti siihen tulokseen, että tällä tavoin on mahdoton tuottaa .NET-alustan tavukooditarkastajan hyväksymää koodia [BSS04].

Koska toteutus tehtiin Microsoftin .NET-alustalla, on mahdollista että esimerkiksi Monon toteutus poikkeaa tässä kuvatusta ja sallii tekniikoita, jotka eivät .NET-alustalla ole mahdollisia. Tätä ei kuitenkaan tarkasteltu toteutuksen yhteydessä, eikä tällainen toteutus olisi CLI-toteutusten välillä siirrettävä.

Lopputulema vertailussa oli siis se, että CLI-alustalla on joko käytettävä jotakin oliosulkeumien kaltaista yleiskäyttöistä menetelmää sulkeumien toteuttamiseen tai toteutettava häntäkutsut ilman alustan tukea. Toteutuksessa päädyttiin siis lopulta käyttämään Kawa-tyylistä delegaattimaista lähestymistapaa, jonka etu tavallisiin oliosulkeumiin verrattuna on se, että luokkia tarvitsee luoda vähemmän.

Cottontail Schemessä valitussa toteutuksessa on kuitenkin huomattava, että sekä pakomuuttujatietueiden kantaluokkana toimivan `ProcedureFrame`-luokan `funcall`-metodit että `CTObject`-luokan `apply`-metodit ovat virtuaalisia metodeita, jolloin niitä on kutsuttava `callvirt`-käskyllä. Kaikki funktioarvoihin tehtävät häntäkutsut siis tehdään `tail. callvirt` -käskyillä.

Kuten luvussa 4.3 mainittiin, CLI-spesifikaatio ei vaadi toteutuksilta häntäkutsujen optimointia, mikäli `tail.`-etuliitettä käytetään muiden kutsukäskyjen kuin `call`-käskyn yhteydessä, esimerkiksi siis juuri `callvirt`-käskyn edellä. Käytännössä sekä Microsoftin .NET-toteutus että Mono näyttävät testien perusteella optimoivan nämä tapaukset, mutta spesifikaatio jättää toteuttajan epävarmaksi tällaisen toteutuksen luotettavuudesta.

[10] Jostakin syystä kaikki virheet eivät tee ohjelmasta ajokelvotonta. Esimerkiksi testaamalla havaittiin, että metodiosoittimen lataus `ldftn`-käskyllä ja sen välitön kutsuminen `calli`-käskyllä aiheuttaa virheitä `peverify`-tavukoodintarkistajan tulosteeseen, mutta ohjelma on kuitenkin mahdollista ajaa ainakin kaikissa tapauksissa, joita testattiin.

Samankaltainen ongelma tosin on myös Bigloon .NET-toteutuksessa, jossa vastaavat `funcall`-metodit ovat myös virtuaalisia. Virtuaalisten metodien käytöstä sulkeumien toteuttamisessa ei kuitenkaan ole ilmeistä tapaa päästä eroon kehittämättä esimerkiksi jonkinlaista alustan refleksiivisyysominaisuuksien käyttöön perustuvaa sulkeumaesitystä, joka mahdollistaisi kutsujen tekemisen metodeihin, joita ei tunneta käännösaikana.

6 Toteutustekniikoiden arviointi

Tässä luvussa arvioidaan JVM- ja CLI-alustojen tarjoamia mahdollisuuksia funktionaalisten piirteiden toteuttamiseen huomioiden sekä luvussa 5 kuvatussa Cottontail Scheme -toteutuksessa että luvussa 4.4 esitetyssä muiden funktionaalisia piirteitä tukevien ohjelmointikielten vertailussa esille tulleet seikat. Lisäksi tuodaan esille mahdollisia ongelmakohtia Cottontail Schemen kaltaisessa funktionaalisen kielen toteutuksessa ja ehdotetaan näihin liittyviä mahdollisia jatkotutkimuslinjoja.

Koska tutkielman painotus oli erilaisten toteutustekniikoiden esittelyssä ja laadullisessa vertailussa kääntäjän toteuttajan näkökulmasta, olisi toteutustekniikoiden tarkastelua kiinnostavaa jatkaa erityisesti tässä tutkielmassa vähäisemmälle huomiolle jääneen suorituskykyvertailun osalta. Tutkielmassa lähestymistapana oli tehdä yksi kielitoteutus molemmille alustoille käyttäen kullakin alustalla mahdollisimman paljon alustan tarjoamaa tukea, esimerkiksi juuri sulkeumien toteuttamiseen suunniteltua toiminnallisuutta. Suorituskykyvertailuun keskittyvä tarkastelu puolestaan vaatisi useampien tekniikoiden toteuttamisen rinnakkain samalle alustalle, jotta alustojen väliset erot suorituskyvyssä eivät vaikuttaisi vertailuun.

Luvussa käsitellään ensin sulkeumatoteutuksiin liittyviä kysymyksiä. Sen jälkeen tarkastellaan häntäkutsujen optimointitekniikoita ja niiden käyttöä moderneissa funktionaalisissa ja moniparadigmakielissä. Lopuksi esitetään lyhyt vertailu ja tehdään yhteenveto havainnoista.

6.1 Sulkeumatoteutuksen valinta

Kielitoteutusten vertailussa todettiin, että valtaosa vertailluista kielitoteutuksista nojautuu sulkeumien toteutustekniikoista ehkäpä toteuttajan kannalta yksinkertaisimpaan, eli oliosulkeumiin. Haittapuolena oliosulkeumien käytössä on se, että luokkia tarvitaan paljon: yksi jokaista sulkeumaa kohti.

Vaihtoehtoisten tekniikoiden kuten CLI-delegaattien sekä Kawan ja Bigloon kaltaisten delegaattitekniikoiden etuna on se, että luokkia tarvitaan vähemmän. JVM-alustan sisäänrakennettu tuki vähentää myös tarvittujen luokkien määrää, mutta lisäksi se siirtää sulkeumaesityksen ylläpito- ja optimointivastuun suoritusympäristön toteutukselle.

Delegaattityyliset tekniikat ja oliosulkeumat

Delegaattityylisten tekniikoiden etu oliosulkeumiin verrattuna on se, että luokkia tarvitaan vähemmän — yleensä yksi jokaista sulkeumia esittelevää näkyvyysaluetta kohti. Esimerkiksi Bigloo-toteutuksessa on valittu olla käyttämättä oliosulkeumia juuri siksi, että suorituskykymittausten perusteella Scheme-ohjelmien vaatima luokkien määrä olisi

aiheuttanut liian suurta kuormaa JVM-alustan luokkien lataajalle, ja lisäksi luokkien todettiin kasvattavan tuotetun binäärin kokoa merkittävästi .NET-alustalla [BSS04]. Esimerkkinä Scheme-ohjelman sisältämien sulkeumien määrästä Bres mainitsee Bigloo-kääntäjän, jonka kerrotaan sisältävän 4000 sulkeumaa [BSS04]. Myös Kawassa käytettiin alun perin yksinkertaisia oliosulkeumia, mutta toteutuksessa siirryttiin myöhemmin delegaattityyliseen toteutukseen ilmeisesti juuri suorituskykyyn liittyvistä syistä [Bot16].

Delegaattityylisiä toteutuksia tarkasteltaessa on kuitenkin syytä huomata, että esimerkiksi Kawan ja Cottontail Schemen CLI-toteutuksen käyttämä delegaattityyli ei välttämättä sovi luontevasti yhteen staattisesti tyypitettyjen kielten kanssa. Staattisesti tyypitetyissä kielissä kutsuparametrien ja paluuarvojen tyypit tekisivät metodiosoittimien simuloinnista `switch`-lauseen avulla hankalaa, koska samassa näkyvyysalueessa määritellyt sulkeumafunktiot voivat ottaa eri tyyppisiä parametreja ja palauttaa eri tyyppisiä arvoja. Tällöin `switch`-lauseen sisältävän `funcall`-metodin olisi joko otettava kutsuparametrit `Object`-tyyppisinä ja suoritettava tyyppimuunnoksia ennen varsinaisten sulkeumametodien kutsumista tai vaihtoehtoisesti `funcall`-metodeita tarvittaisiin pahimmassa tapauksessa yhtä monta kuin on eri tyyppisiä ensimmäisen luokan funktioita. Staattisesti tyypitetyssä kielessä oliosulkeumat ovat todennäköisesti myös tehokkaampi toteutustapa, koska ei tarvita ylimääräisiä tyyppimuunnoksia tai kutsuja `funcall`-metodien kaltaisten apumetodien kautta.

Bigloossa käytetty tekniikka ei ole käytännöllinen staattisesti tyypitetyissä kielissä myöskään siitä syystä, että tarvittujen luokkien määrän karsinta perustuu siihen, että kaikki kaapattujen muuttujien arvot voidaan pakata samaan taulukkoon. Staattisesti tyypitetyille kielille sopivampi tekniikka olisi esimerkiksi CLI-alustan delegaattiluokkien käyttö. Useimmat luvussa 4.4 esitetyn vertailun oliosulkeumia käyttäneistä kielistä olivat kuitenkin JVM-alustalla, jolloin CLI:n delegaattiluokkien kaltaista delegaattitoteutusta ei ole ollut saatavilla, mikä on tehnyt tämän kaltaisen tekniikan käytön hyvin epäkäytännölliseksi. Bigloon tai Kawan tyylinen tekniikka todennäköisesti sopisi kuitenkin mainiosti esimerkiksi Clojureen, joka on dynaamisesti tyypitetty kieli.

Bresin [BSS04] mukaan Bigloo-toteutusta tehtäessä arvioitiin myös CLI-alustan delegaattiluokkien soveltuvuutta sulkeumien esittämiseen, mutta ainakin tuolloin delegaattien kautta tehtävät metodikutsut olivat hitaampia kuin kutsuttavan metodin valinta `switch`-lauseella. Tavallisia oliosulkeumia käyttävän Clojuren osalta ei näytä olevan olemassa dokumentaatiota siitä, miksi on päädytty juuri oliosulkeumiin, vaikka suuren luokkien määrän voisi olettaa Clojuressakin johtavan mahdollisiin suorituskykyhaittoihin.

Oliosulkeumia käyttävistä kielistä ainakin Scalassa on panostettu sulkeumaesityksen sijaan muihin optimointitapoihin kuten sulkeumien täydelliseen eliminointiin esimerkiksi laventamalla sulkeumat kutsuvan metodin runkoon tilanteissa, joissa se on mahdollista [Dra10, s. 29–30]. Vertailun testiohjelmia käännettäessä havaittiin, että myös F# ja Kotlin

poistavat joissakin tilanteissa sulkeumaolion kokonaan vastaavalla lavennustekniikalla. Tällaisilla optimoinneilla voi olla merkittävä vaikutus tilanteissa, joissa käytetään lähinnä ylätason funktioita ensimmäisen luokan arvoina tai käytetään paljon apufunktioina toimivia sulkeumia, joita tarvitaan vain sulkeuman määritelleen funktion sisällä.

Delegaattityylisen tekniikan käyttö puolestaan vähentää tarvittavien sulkeumien määrää tilanteissa, joissa samassa näkyvyysalueessa määritellään useita sulkeumia, joita halutaan kutsua sulkeumat määritelleen funktion ulkopuolelta käsin. Ainakin dynaamisesti tyypitetyissä kielissä delegaattityylinen esitys olisi toisaalta myös mahdollista yhdistää lavennustekniikoiden kanssa, jolloin tarvittujen luokkien ja metodikutsujen määrä vähenisi entisestään. Esimerkiksi Bigloo tekeekin jo joitakin tämän kaltaisia optimointeja.

JVM-alustan tuki

Ylläpidettävyyden kannalta paras vaihtoehto sulkeumien esittämiseen lienee ainakin teoriassa JVM-alustan lambda-metatehdastuki, joka siirtää vastuun sulkeumaesityksestä kokonaan suoritusaikaiselle ympäristölle. Lienee kuitenkin epätodennäköistä, että jo vanhemmat kielet kuten Scala siirtyisivät kokonaan uuteen sulkeumaesitykseen. Uusille kielitoteutuksille tekniikka näyttää kuitenkin varteenotettavalta vaihtoehdolta. Toisaalta alustan tuessa on huonojakin puolia. Erityisesti ylläpidettävyys hankaloituu merkittävästi siinä tilanteessa, että kieli tahdotaan siirtää JVM-alustan lisäksi esimerkiksi C-kielen tai CLI-alustan päälle, jolloin tarvitaan kokonaan uusi sulkeumatoteutus. Oliosulkeumatyyli on helpommin siirrettävissä alustalta toiselle.

Dynaamisen kielen tapauksessa oliosulkeumat ja Bigloon ja Kawan kaltaiset delegaattityyliset esitykset voivat olla lambda-metatehtaan käyttöä käytännöllisempiä myös niiden joustavuuden vuoksi. Koska toteuttajalla on täysi kontrolli sulkeuman esitykseen, onnistuu esimerkiksi parametrien taulukosta purkamiseen ja parametritarkistuksiin liittyvien funktiokutsujen lisääminen suoraan oliosulkeumaa esittävään luokkaan. Tällöin oliosulkeuman `apply`-metodi voi huolehtia tarvittavista toimenpiteistä, eikä tarvita esimerkiksi Cottontail Schemen JVM-toteutuksen kaltaisia apufunktioita ja kääreenä käytettäviä parametrien tarkistussulkeumia.

Johtuen lambda-metatehtaiden uutuudesta ja vähäisestä käytöstä tuotantolaatuisissa kielitoteutuksissa herää myös epäilys tekniikan vaikutuksesta suorituskykyyn. Suoritusaikainen ympäristö voi hyvin käyttää sulkeuman esityksenä vaikkapa tavallisia oliosulkeumia [Goe12]. Mikäli suuri luokkien määrä siis todella haittaa luokkalataajan toimintaa, suorituskykyhaittoja luulisi ilmenevän myös tällaista lambda-metatehdastoteutusta käytettäessä. Hieman epäselvää on myös, onko tuki ylipäätään tarkoitettu Schemen ja muiden funktionaalisten kielten kaltaiseen laajamittaiseen käyttöön, ja aiheutuuko suuresta

`invokedynamic`-kutsupaikkojen määrästä itsessään haittaa ohjelmien suorituskyvylle.

JVM-alustan metodikahvojen hyödyt verrattuna lambda-metatehtaiden käyttöön jäävät toistaiseksi puuttuvien tosimaailman käyttöesimerkkien puutteessa hieman epäselviksi, sillä niiden käyttö on selvästi lambda-metatehtaita monimutkaisempaa. Metodikahvat tosin ovat jo lambda-metatehtaita vanhempi ominaisuus, joten on mahdollista, että niiden tarkoitus on toimia matalamman tason työkaluna. GitHub-sivustolla tehdyn haun perusteella MethodHandle-olioita käytetään tällä hetkellä olemassa olevista kielitoteutuksista ainakin JRubyssa [JRu16], mutta on mahdollista että JRuby käyttää niitä muihin tarkoituksiin kuin Rubyn funktionaalisten piirteiden toteutukseen. JRuby ja Jython jäivät luvun 4.4 vertailun ulkopuolelle, koska kummankaan toteutuksen uusimmilla versioilla ei kirjoitushetkellä ollut mahdollista tuottaa `class`-tiedostoja, jotka olisivat mahdollistaneet tavukoodin tarkastelun.

Lambda-metatehtaiden ja `invokedynamic`-käskyn vaikutuksia funktionaalisella kielellä toteutetun ohjelman suorituskykyyn voitaisiin vertailla toteuttamalla esimerkiksi Cottontail Schemen JVM-takaosaan rinnakkain kaksi tai useampia erillisiä sulkeumatoteutuksia. Tällöin olisi mahdollista vertailla erilaisten sulkeumatoteutusten käyttäytymistä.

6.2 Häntäkutsujen optimointi virtuaalikonealustoilla

Alustojen välillä erot häntäkutsutuessa olivat hyvin suoraviivaiset: CLI tukee häntäkutsuja sisäänrakennettuna, JVM ei. CLI-alustan häntäkutsutuen käyttö kääntäjässä oli lisäksi melko yksinkertaista. JVM-alustalla häntäkutsutuki on vielä tulevaisuudessa, mutta ilmeisesti toiminnallisuus on kuitenkin jo kehitteillä.

Modernien funktionaalisten kielten ja moniparadigmakielten vertailussa luvussa 4.4 huomattiin, että harva kielistä käytti häntäkutsuoptimointia. Vertailun laajemmin käytetyistä kielistä ainoa yleistettyjä häntäkutsuja optimoinut kieli oli F#, joka optimoi yleistetyistä häntäkutsuista ainoastaan ne, jotka ovat keskenään rekursiivisia. Molemmat tarkastellut Scheme-toteutukset puolestaan toteuttivat ainakin toisella alustoista häntäkutsuoptimoinnin.

Tässä aliluvussa arvioidaan CLI-alustan häntäkutsutuen käyttökelpoisuutta käytännön toteutuksissa. Toisaalta analysoidaan myös sitä, miksi häntäkutsutuen toteuttaminen näyttää olevan nykykielissä Scheme-toteutusten ulkopuolella harvinaista. Kielten vertailusta herää kysymys, pidetäänkö häntäkutsujen optimointia enää tarpeellisena funktionaalisissakaan kielissä, vai onko häntäkutsuista tullut historiallinen seikka funktionaalisten kielten kehityskaaressa kuten Lisp-perheen kielten dynaamisesta näkyvyydestä aikoinaan.

CLI-alustan häntäkutsutuen arviointi

CLI-alustan häntäkutsut ovat suhteellisen helppokäyttöinen ominaisuus toteuttajan kannalta, vaikka ominaisuus osoittautuikin epäyhteensopivaksi CLI-alustan delegaattiluokkien kanssa. Standardin CLI-toteutuksille antama päätäntävalta `tail.`-etuliitteen optimoinnissa kuitenkin herättää epäilyksen siitä, voiko alustan häntäkutsutukea hyödyntävä toteutus taata häntäkutsujen optimointia, jos kääntäjätoteutus siirretään toimimaan toisen CLI-toteutuksen päällä. Esimerkiksi Cottontail Schemen ja Bigloon CLI-toteutuksissa kutsut sulkeumiin ovat `tail. callvirt` -käskyjä, joiden optimointia standardi ei vaadi. Standardin vaatimusten löysyydestä huolimatta sekä Microsoftin .NET-toteutus että Mono näyttävät kuitenkin optimoivan sekä `tail. callvirt` että `tail. calli` -käskyt jopa assembly-yksiköiden välillä.

Toinen heräävä kysymys liittyy `tail.`-etuliitteellisten kutsujen suorituskykyyn. Bigloon toteuttajat raportoivat häntäkutsujen käytön aiheuttaneen paikoin suuriakin suorituskykykustannuksia, vaikka kustannukset keskimäärin pysyivätkin suhteellisen pieninä [BSS04]. Tästä syystä Bigloon toteutus ei edes oletuksena optimoi yleistettyjä häntäkutsuja. Koska Bresin, Serranon ja Serpetten tekemät mittaukset kuitenkin ovat vuodelta 2004, voisi olla syytä toteuttaa vastaava mittaus uudelleen.

Cottontail Schemen toteutuksen yhteydessä ei tehty varsinaisia suorituskykymittauksia, mutta JVM- ja CLI-alustan testiohjelmia rinnakkain ajettaessa havaittiin, että Microsoftin .NET-alustalla ajettuna keskinäistä häntärekursiota hyödyntävä koodiesimerkki oli huomattavasti JVM-versiota hitaampi erityisesti käytettäessä sulkeumaolioiden välistä häntärekursiota. Tämä ei kuitenkaan ole suora osoitus siitä, että trampoliinitekniikka olisi CLI-alustan häntäkutsujen käyttöä tehokkaampi. Mahdollisia selityksiä ohjelmien käyttäytymiselle voisivat olla esimerkiksi seuraavat:

- Häntäkutsukäskyt virtuaalisiin metodeihin eli `tail. callvirt` -käskyt voivat olla hitaampia kuin staattisiin metodeihin tehtävät `tail. call` -käskyt.
- JVM-alustalla käytetty sulkeumaesitys voi olla jostakin syystä selvästi nopeampi kutsua kuin CLI-alustalla käytetty esitys. Mahdolliset syyt tähän eivät ole ilmeisiä: CLI-alustan toteutus vaatii ylimääräisiä virtuaalisia metodikutsuja, mutta toisaalta JVM-alustalla sulkeumat kääritään esimerkiksi parametritarkistusten vuoksi toisiin sulkeumiin, minkä voisi myös odottaa hidastavan suoritusta.
- JVM on todennäköisesti ylipäätään optimoidumpi ja nopeampi kuin .NET tai muut CLI-toteutukset.

Esimerkiksi Bres, Serpette ja Serrano totesivat Bigloo-toteutuksen olevan noin 1,5–2 kertaa hitaampi CLI-alustalla kuin JVM-alustalla, mikä voisi viitata viimeisen mahdollisuuden suuntaan [BSS04]. Näiden muuttujien rajaaminen pois yhtälöstä edellyttäisi

vaihtoehtoisten sulkeumaesitysten sekä trampoliinitekniikan toteuttamista Cottontail Schemen CLI-toteutukseen ja huolellista benchmark-testausta, jossa rajattaisiin pois myös esimerkiksi virtuaalikoneen käynnistymisen viemä aika ja mahdollisten JIT-kääntäjän kesken suorituksen tekemien optimointien vaikutukset. Tällöin toteutukseen täytyisi lisäksi lisätä sisäänrakennettu tapa mitata metodikutsun viemää aikaa. Yksinkertainen `tail.`-kutsujen vertailu häntäkutsuoptimoimatonta versiota vasten olisi mahdollista tehdä myös suoraan Bigloo-toteutuksen avulla.

Häntäkutsujen optimointi moderneissa kielissä

Kielitoteutusten vertailussa havaittiin, että harva kielitoteutus JVM-alustalla tukee yleistettyjä häntäkutsuja tai edes kahden funktion välistä keskinäistä rekursiota erikoistapauksena. Perimmäisenä syynä on todennäköisesti alustan puutteellinen tuki ja haluttomuus käyttää trampoliinitekniikkaa johtuen oletetuista suorituskykykustannuksista.

Erityisesti mielenkiintoista oli se, ettei Scalakaan tue yleistettyjä häntäkutsuja huolimatta siitä, että Odersky on aiemmin ollut kehittämässä "Cheney on the M.T.A" -menetelmästä virtuaalikoneyhteensopivaa versiota Funnel-kääntäjän tarpeisiin [SO01]. Schinzin ja Oderskyn menetelmä kuitenkin aiheuttaa edelleen suorituskykykustannuksia ja lisäksi kasvattaa tuotettua tavukoodia johtuen jokaiseen funktioon lisättävistä pinon koon tarkistuksista. Oletettavasti kielen yleinen suorituskyky on siis katsottu Scalassa tärkeämmäksi ominaisuudeksi kuin häntäkutsujen tuki.

Koska niin harva kääntäjä ylipäätään näyttää tukevan yleistettyjä häntäkutsuja, voidaan ehkäpä myös tehdä oletus, että nykykielten toteuttajat eivät enää pidä häntäkutsuja välttämättömänä ominaisuutena, ja häntärekursion optimoinnin katsotaan kattavan yleisimmät häntäkutsujen käyttötapaukset. Jopa Scheme-toteutukset Kawa ja Bigloo vaativat erillisen käännösvivun käyttöä häntäkutsujen päälle kytkemiseksi huolimatta siitä, että R7RS-standardi edellyttää yleistä häntäkutsujen optimointia. Moniparadigmakielten tapauksessa ohjelmoijalla on myös mahdollisuus nojautua kielen imperatiivisiin ja olio-ohjelmointiominaisuuksiin tilanteissa, joissa suorituskyky on erityisen tärkeää, mikä saattaa vähentää tarvetta tai haluja optimoida tämän kaltaisia funktionaalisia piirteitä.

Toisaalta esimerkiksi CPS-muodon käyttö saattaa olla menettämässä suosiotaan virtuaalikonealustojen funktionaalisissa kielissä, mikä entisestään vähentää tarvetta yleiselle häntäkutsutuelle. Yksikään vertailluista toteutuksista ei hyödynnä CPS-muotoa ainakaan millään näkyvällä tavalla.

CPS-muoto voi olla virtuaalikonealustoilla vähemmän hyödyllinen kuin perinteisillä alustoilla esimerkiksi johtuen siitä, että virtuaalikonealustat eivät salli rajoittamattomia hyppyjä, vaan hyppyjen on tapahduttava saman metodin sisällä. Toinen syy on se, että toteuttajalla ei ole pääsyä pinon manipulointiin eikä toteuttaja voi esimerkiksi valita,

kuljetetaanko parametreja rekistereissä vai laitetaanko ne pinoon.

Esimerkiksi Schemen alkuperäisessä Rabbit-kääntäjässä tärkeä CPS-muotoon liittyvä ajatus oli se, että se mahdollisti funktionaalisen kielen kääntämisen imperatiiviseksi kohdekieleksi käsittelemällä häntäkutsuja `goto`-käskyinä, joiden on ainoastaan lisäksi pystyttävä kuljettamaan mukanaan kutsun parametreja [SJ78, s. 47]. CPS-muodon yhteydessä voidaan jopa erityisesti pyrkiä välttämään pinon käyttöä, sillä CPS-muodossa kaikki funktiot eivät välttämättä edes tarvitse aktivaatiotietuetta [App98, s. 329–332]. Tällöin kutsuparametrit pyritään pitämään rekistereissä ja tarvittaessa kekomuistissa. Tällaiset tekniikat eivät ole erityisen hyödyllisiä virtuaalikonealustoilla, koska pinon manipulointi ja rajoittamattomat hypyt eivät ole mahdollisia.

6.3 Alustojen keskinäinen vertailu ja johtopäätökset

Microsoft on rakentanut CLI-alustalle vahvan maineen monia eri tyyppisiä kieliä tukevana alustana [MG01, Hug07], ja monet ominaisuudet kuten häntäkutsutuki ja delegaatit ovat olleet alustalla tarjolla jo pitkään. Tämä loi tutkielman alkuvaiheessa ennakko-odotuksen siitä, että CLI-alustan tuki olisi JVM-alustaa kypsempi ja kattaisi tarvitut käyttötapaukset paremmin.

Odotusten mukaisesti häntäkutsujen toteutus CLI-alustalla olikin suoraviivaisempaa johtuen sisäänrakennetusta häntäkutsutuesta. Yllättävämpi seikka oli se, että häntäkutsuja ei ollut suunniteltu toimimaan yhdessä delegaattien kanssa. Tämä synnytti vaikutelman, että delegaattituki on mahdollisesti tarkoitettu lähinnä oliokielten kuten C#:n sekä kielten yhteiskäyttörajapintojen käytettäväksi. Havainto osoittaa, että jos virtuaalikonealustoille toteutetaan funktionaalisten kielten piirteitä tukevia ominaisuuksia, ominaisuudet on suunniteltava huolellisesti yhdessä toimiviksi. Esimerkiksi delegaattiluokat, jotka käyttäisivät `tail.`-kutsuja osoittamansa metodin kutsumiseen, tekisivät delegaateista käyttökelpoisempia yleistettyjä häntäkutsuja tukevan funktionaalisen kielen toteuttajalle.

Cottontail Scheme -toteutusta tehtäessä osoittautui, että JVM on ottanut CLI-alustan kiinni ainakin sulkeumien tuessa. JVM:n tarjoama mahdollisuus jättää sulkeumaesitys suoritusaikaisen ympäristön vastuulle on omaperäinen ajatus, joka vapauttaa kääntäjän toteuttajan käyttämästä aikaa sulkeumaesityksen optimointiin ja ylläpitoon. CLI-alustalla kääntäjän toteuttaja joutuu luopumaan joko delegaateista sulkeumien toteutusvälineenä tai häntäkutsutuesta ensimmäisen luokan funktioiden kanssa.

Osa kielten toteuttajista on valinnut olla käyttämättä CLI-alustan delegaattiluokkia myös delegaattikutsujen hitauden vuoksi [BSS04], joten on mahdollista, että CLI-alustan delegaattiluokat eivät ylipäätään ole optimaalinen sulkeumien toteutusmenetelmä. C#:ssa delegaattiluokat ovat olleet käytössä jo ennen lambda-lausekkeiden tuontia kieleen, jo-

ten delegaattiluokat on mahdollisesti valittu lambda-lausekkeiden toteutustekniikaksi muista syistä. C#:n toteutusvalinnan taustalla saattavat olla esimerkiksi kielten yhteiskäyttörajapintojen tarpeet tai yksinkertaisesti se, että delegaatit olivat jo ennestään käytössä kielessä, ja lambda-lausekkeiden tuen rakentaminen delegaattiluokkien varaan oli helppoa.

Vaikka tässä tutkielmassa tarkastelun kohteena eivät olleet erityisesti dynaamiset kielet, syntyi myös vaikutelma, että JVM-alustalla dynaamisten kielten tuki on alettu ottaa vakavasti, mikä tulee esille esimerkiksi `invokedynamic`-käskyn ja `MethodHandle`-tyypin suunnittelussa. Dynaamisten kielten tuki ei kuitenkaan ole JVM:n yksinoikeus, sillä CLI-alustan DLR-kirjastokokoelma tukee myös monia esimerkiksi juuri `invokedynamic`-käskyn kaltaisia ominaisuuksia [CT09, s. 8]. JVM:n tulevaisuus näyttää kuitenkin valoisalta: mikäli Da Vinci Machine -projektissa kaavaillut ominaisuudet kuten esimerkiksi häntäkutsutuen lisääminen JVM-alustalle toteutuvat, JVM voi olla tulevaisuudessa hyvin houkutteleva alusta sekä funktionaalisille että dynaamisille kielille.

Koska Scheme on sekä funktionaalinen että dynaaminen kieli, Cottontail Scheme -toteutus saattaisi pystyä jatkossa toimimaan myös pohjana CLI- ja JVM-alustojen dynaamisten kielten tuen vertailulle. Tällöin voitaisiin toteuttaa kielen tyyppijärjestelmä uudestaan alustojen tarjoaman dynaamisten kielten tuen päälle. Joka tapauksessa alustojen tarjoamaa tukea olisi mielenkiintoista vertailla myös erityisesti dynaamisten kielten toteutuksen näkökulmasta.

Vertailun perusteella kumpikaan alustoista ei näytä siltä, että sen tarjoamat ominaisuudet yksin riittäisivät houkuttelemaan funktionaalisten kielten toteuttajia toiselle alustalle toisen sijaan. Merkittävin ero tällä hetkellä on se, että häntäkutsuja erityisen tärkeinä pitävät kielten toteuttajat joutuvat valitsemaan CLI-alustan, mikäli ei haluta tehdä omaa häntäkutsutoteutusta tai trampoliineja ei haluta käyttää esimerkiksi suorituskykysyistä. Epäselväksi tosin jää, onko CLI-alustan tuki häntäkutsuille todellisuudessa trampoliinitekniikan käyttöä suorituskykyisempi vaihtoehto.

Muissa tapauksissa alustan valinta todennäköisesti tehdään muilla perusteilla — esimerkiksi alustan suorituskyvyn, suosittuuden, siirrettävyyden, tuttuuden tai valmiiksi kielten yhteiskäyttöä varten tarjolla olevan kirjastovalikoiman perusteella. Molempia alustoja kohteenaan käyttävien kääntäjien tapauksessa puolestaan yleiskäyttöiset tekniikat kuten oliosulkeumat ja trampoliinien käyttö vievät todennäköisesti ylläpitosyistä voiton alustan tarjoamasta tuesta, koska samoja toteutuksia voidaan käyttää molemmilla alustoilla.

7 Yhteenveto

Tutkielmassa tarkasteltiin mahdollisia ensimmäisen luokan funktioarvojen ja yleistettyjen häntäkutsujen optimoinnin tekniikoita JVM- ja CLI-alustoilla vertaillen niitä perinteisillä alustoilla käytettyihin toteutusmenetelmiin. Oliokielten virtuaalikonealustojen tärkeimmät rajoitteet verrattuna konekieltä kohdekielenä käyttäviin perinteisempiin toteutuksiin liittyvät hyppykäskyjen ja pinon käsittelyn rajoituksiin sekä raakojen funktio-osoittimien puuttumiseen tai käytön rajoittamiseen. Tämän vuoksi funktionaalisten kielten piirteiden toteutus ei onnistu samoilla tekniikoilla kuin perinteisissä konekieltä tuottavissa kääntäjissä. Toisaalta monet C-kieltä kohdekielenä käyttävistä kääntäjistä periytyvät tekniikat todettiin pienin muutoksin käyttökelpoisiksi myös virtuaalikonealustoilla.

Tutkielmassa tunnistettiin neljä eri tyyppistä lähestymistapaa ensimmäisen luokan funktioarvojen ja sulkeumien toteutukseen: oliosulkeumat, tyyppi- ja muistiturvalliset funktio-osoittimet kuten CLI-alustan delegaatit ja JVM-alustan `MethodHandle`, funktio-osoittimia `switch`-lauseen avulla simuloivat delegaattityyliset menetelmät sekä JVM-alustan lambda-metatehdas, joka siirtää vastuun sulkeumaesityksen rakentamisesta suoritusaikaiselle ympäristölle. Lambda-metatehdas mahdollistaa suoritusaikaisen ympäristön päivitysten yhteydessä tehtävät parannukset sulkeumien toteutukseen, minkä on mahdollista hyödyttää kaikkia tätä toteutustekniikkaa käyttäviä kielitoteutuksia.

Häntäkutsujen optimointiin tunnistettiin kaksi menetelmää: CLI-alustan sisäänrakennettu häntäkutsutuki ja C-kieltä kohdekielenä käyttävistä kääntäjistä periytyvä trampoliinitekniikka. JVM-alustalla trampoliinitekniikka on toistaiseksi ainoa mahdollinen toteutustapa häntäkutsuille.

Alustojen tarjoamien toteutustekniikoiden toimintaa käytännön kääntäjätoteutuksessa arvioitiin toteuttamalla oma Cottontail Scheme -niminen prototyyppitoteutus Scheme-kielen osajoukolle molemmilla tarkastelluilla alustoilla. Toteutuksessa hyödynnettiin JVM-alustalla lambda-metatehdasta ja trampoliinitekniikkaa. CLI-alustalla häntäkutsut toteutettiin alustan sisäänrakennetun tuen avulla ja sulkeumille luotiin oma delegaattityylinen esitys. CLI-alustan sisäänrakennetut delegaattiluokat osoittautuivat käyttökelvottomiksi sulkeumien toteuttamiseen tilanteessa, jossa on tarve tukea myös yleistettyjä häntäkutsuja funktioarvoihin, sillä CLI:n oma delegaattitoteutus ei käytä häntäkutsuja.

Lisäksi vertailtiin muutamien JVM- ja CLI-alustojen suosituimpien kielten sekä kahden näillä alustoilla toteutetun Scheme-toteutuksen käyttämiä toteutustekniikoita. Vertailtavina olivat Java, Clojure, Scala, Kotlin, Microsoftin F#- ja C#-kielet sekä Scheme-toteutukset Kawa ja Bigloo. Vertailussa yleisimmäksi käytetyksi ensimmäisen luokan funktioarvojen toteutustekniikaksi todettiin oliosulkeumat erityisesti JVM-alustan kielissä. Lisäksi havaittiin, että yksikään vertailun suosituimmista kielistä ei optimoi

yleistettyjä häntäkutsuja todennäköisesti johtuen erityisesti JVM-alustan puutteellisesta tuesta ja trampoliinitekniikan käytön suorituskykykustannuksista. JVM-alustan uusia tekniikoita kuten lambda-metatehtaita ei vielä käytetty muissa kielitoteutuksissa kuin Java 8:ssa ja tutkielmassa esitellyssä toteutuksessa, Cottontail Schemessä, mikä todennäköisesti johtuu toiminnallisuuden tuoreudesta ja siitä, että useimmat vertailluista kielitoteutuksista ovat valmistuneet ennen toiminnallisuuden esittelyä.

Alustojen vertailussa molemmilla alustoilla havaittiin olevan omat vahvuutensa. JVM-alustalla sulkeumatuki on parantunut vauhdilla viime vuosien aikana, mutta CLI lienee toistaiseksi ensisijainen vaihtoehto kielitoteutukselle, jonka on tuettava yleistettyjen häntäkutsujen optimointia. Alustojen kilpavarustelu näyttää viime vuosina vain kiihtyneen: molemmat alustat ovat esitelleet esimerkiksi uutta toiminnallisuutta dynaamisten kielten tueksi ja JVM-alustalla tutkimus häntäkutsujen ja muiden funktionaalisiakin kieliä hyödyttävien ominaisuuksien lisäämisestä alustalle jatkuu. Kummankaan alustan tällä hetkellä tarjoamat toteutuksen tukiominaisuudet tuskin kuitenkaan pystyvät yksin nostamaan alustaa selkeäksi voittajaksi funktionaalisten kielten käännösalustana.

Lähteet

[AA12] Albahari, J. ja Albahari, B., *C# 5.0 in a Nutshell: The Definitive Reference*. O'Reilly, viides painos, 2012.

[App92] Appel, A. W., *Compiling with Continuations*. Cambridge University Press, 1992.

[App98] Appel, A. W., *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[asm16] ASM - users, `http://asm.ow2.org/users.html`, 2016.

[Bak95] Baker, H. G., CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Notices*, 30,9(1995), sivut 17–20.

[Bot98] Bothner, P., Kawa: compiling dynamic languages to the Java VM. *Proceedings of the USENIX Technical Conference, FREENIX Track*. USENIX Association, 1998.

[Bot16] Bothner, P., Kawa internals: Compiling Scheme to Java, `https://www.gnu.org/software/kawa/internals/index.html`, 2016.

[BSS04] Bres, Y., Serrano, M. ja Serpette, B. P., Bigloo.NET: compiling Scheme to .NET CLR. *Journal of Object Technology*, 3,9(2004), sivut 71–94.

[Che70] Cheney, C. J., A nonrecursive list compacting algorithm. *Communications of the ACM*, 13,11(1970), sivut 677–678. URL `http://doi.acm.org/10.1145/362790.362798`.

[clo16] ClojureCLR, `https://github.com/clojure/clojure-clr`, 2016.

[CT09] Chiles, B. ja Turner, A., Dynamic Language Runtime, `http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation`, 2009.

[Del16a] Delimarsky, D., Recursive functions: The rec keyword (F#), `https://msdn.microsoft.com/visualfsharpdocs/conceptual/recursive-functions-the-rec-keyword-%5bfsharp%5d`, 2016.

[Del16b] Delimarsky, D., Reference cells (f#), `https://msdn.microsoft.com/visualfsharpdocs/conceptual/reference-cells-%5bfsharp%5d`, 2016.

[Dra10] Dragos, I., *Compiling Scala for Performance*. École Polytechnique Fédérale de Lausanne, 2010.

[ECM12] ECMA-335: Common Language Infrastructure (CLI), partitions I to VI, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf, 2012.

[FF96] Friedman, D. P. ja Felleisen, M., *The Little Schemer*. The MIT Press, neljäs painos, 1996.

[FMRW97] Feeley, M., Miller, J. S., Rozas, G. J. ja Wilson, J. A., Compiling higher-order languages into fully tail-recursive portable C. *Technical Report 1078, département d'informatique et r.o., Université de Montréal*.

[GHJV95] Gamma, E., Helm, R., Johnson, R. ja Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GJS+15] Gosling, J., Joy, B., Steele, G., Bracha, G. ja Buckley, A., The Java® Language Specification, Java SE 8 Edition, http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, 2015.

[Goe12] Goetz, B., Translation of lambda expressions, http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html, 2012.

[Goe13] Goetz, B., State of the lambda, http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html, 2013.

[Hic16a] Hickey, R., API for clojure.core - Clojure v1.8 (stable), http://clojure.github.io/clojure/clojure.core-api.html#clojure.core/trampoline, 2016.

[Hic16b] Hickey, R., Clojure reference: Special forms, http://clojure.org/reference/special_forms, 2016.

[Hud89] Hudak, P., Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21, sivut 359–411.

[Hug07] Hugunin, J., Bringing dynamic languages to .NET with the DLR. *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, New York, NY, USA, 2007, ACM, sivut 101–101, URL http://doi.acm.org.libproxy.helsinki.fi/10.1145/1297081.1297083.

[IEE08] 1178-1990 IEEE standard for the Scheme programming language, 2008.

[JRu16] JRuby Team, Github: JRuby, an implementation of Ruby on the JVM, https://github.com/jruby/jruby, 2016.

[Kel95] Kelsey, R. A., A correspondence between continuation passing style and static single assignment form. *SIGPLAN Notices*, 30,3(1995), sivut 13–22. URL http://doi.acm.org.libproxy.helsinki.fi/10.1145/202530.202532.

[LYBB15] Lindholm, T., Yellin, F., Bracha, G. ja Buckley, A., The Java® Virtual Machine Specification, Java SE 8 Edition, https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf, 2015.

[McC60] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3,4(1960), sivut 184–195. URL http://doi.acm.org/10.1145/367177.367199.

[McC78] McCarthy, J., History of LISP. *SIGPLAN Notices*, 13,8(1978), sivut 217–223.

[Mer01] Mertz, D., Charming Python: Functional programming in Python, part 1, http://www.ibm.com/developerworks/library/l-prog/index.html, 2001.

[MG01] Meijer, E. ja Gough, J., Technical overview of the Common Language Runtime (or why the JVM is not my favorite execution environment). Tekninen raportti, Microsoft Research, 2001.

[Mic16a] Microsoft, .NET Framework, https://www.microsoft.com/net, 2016.

[Mic16b] Microsoft, .NET Framework class library: IntPtr structure, https://msdn.microsoft.com/en-us/library/system.intptr(v=vs.110).aspx, 2016.

[Mon16a] Mono Project, Mono project home, http://www.mono-project.com, 2016.

[Mon16b] Mono Project, Mono.Cecil, http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/, 2016.

[Nut08] Nutter, C., A first taste of invokedynamic, http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html, 2008.

[Oka98] Okasaki, C., *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[Ora16a] Oracle, Annotation type FunctionalInterface, http://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html, 2016.

[Ora16b] Oracle, Class LambdaMetafactory, http://docs.oracle.com/javase/8/

docs/api/java/lang/invoke/LambdaMetafactory.html, 2016.

[Ora16c] Oracle, The Da Vinci Machine project — a multi-language renaissance for the Java™ Virtual Machine architecture, http://openjdk.java.net/projects/mlvm/, 2016.

[Ora16d] Oracle, Java HotSpot -ryhmän verkkosivut, http://openjdk.java.net/groups/hotspot/, 2016.

[Ora16e] Oracle, Package java.util.function, http://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html, 2016.

[PJ87] Peyton Jones, S., *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pri12] Pritchard, L., IronScheme, https://ironscheme.codeplex.com/, 2012.

[Pri14] Pritchard, L., Curious about IronScheme's implementation, http://ironscheme.codeplex.com/discussions/559894, 2014.

[Que03] Queinnec, C., *Lisp in Small Pieces*. Cambridge University press, 2003.

[Ros07] Rose, J., Tail calls in the VM, https://blogs.oracle.com/jrose/entry/tail_calls_in_the_vm, 2007.

[Ros09] Rose, J. R., Bytecodes meet combinators: Invokedynamic on the JVM. *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, New York, NY, USA, 2009, ACM, sivut 2:1–2:11, URL http://doi.acm.org.libproxy.helsinki.fi/10.1145/1711506.1711508.

[Ros13] Rose, J., Tail call support in Davinci, https://wiki.openjdk.java.net/display/mlvm/TailCalls, 2013.

[Sab98] Sabry, A., What is a purely functional language? *Journal of Functional Programming*, 8, sivut 1–22.

[Sch05] Schinz, M., *Compiling Scala for the Java Virtual Machine*. École Polytechnique Fédérale de Lausanne, 2005.

[Sch09] Schwaighofer, A., Tail call optimization in the Java HotSpot™ VM. Pro gradu, Johannes Kepler Universität Linz, 2009.

[Sco09] Scott, M. L., *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., 2009.

[Ser15] Serrano, M., Bigloo: a practical Scheme compiler, user manual for version 4.2a, https://www-sop.inria.fr/indes/fp/Bigloo/doc/bigloo.pdf, 2015.

[SJ75] Sussman, G. J. ja Jr., G. L. S., Scheme: An interpreter for extended lambda calculus. *MIT AI Lab Memo AIM-349*.

[SJ78] Steele Jr., G. L., RABBIT: A compiler for SCHEME. *MIT AI Lab Technical Report AITR-474*.

[SO01] Schinz, M. ja Odersky, M., Tail call elimination on the Java Virtual Machine. *Proceedings of ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability*. Elsevier, 2001, sivut 155–168.

[SS02] Serpette, B. P. ja Serrano, M., Compiling Scheme to JVM bytecode: a performance study. *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, New York, NY, USA, 2002, ACM, sivut 259–270, URL http://doi.acm.org/10.1145/581478.581503.

[Sta81] Stallman, R. M., EMACS the extensible, customizable self-documenting display editor. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, New York, NY, USA, 1981, ACM, sivut 147–156, URL http://doi.acm.org/10.1145/800209.806466.

[Str00] Strachey, C., Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13, sivut 11–49.

[Stu11] Sturm, O., *Functional programming in C#: Classic programming techniques for modern projects*. John Wiley & Sons, Ltd, 2011.

[Sus13] Sussman, G. J. e. a., Revised^7 report on the algorithmic language Scheme, http://r7rs.org, 2013.

[Teo91] Teodosiu, D., HARE: An optimizing portable compiler for Scheme. *SIGPLAN Notices*, 26,1(1991), sivut 109–120. URL http://doi.acm.org/10.1145/122203.122211.

[TLA92] Tarditi, D., Lee, P. ja Acharya, A., No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1,2(1992), sivut 161–177. URL http://doi.acm.org/10.1145/151333.151343.

[WWW+13] Würthinger, T., Wimmer, C., Wöß, A., Stadler, l., Dubosq, G., Humer, C., Richards, G., Simon, D. ja Wolczko, M., One VM to rule them all. *Onward! 2013 Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, sivut 187–204.

A Lispin sukuisten kielten erityispiirteitä

Tässä liitteessä esitellään kaksi Lispin sukuisiin kieliin kuten Schemeen liittyvää erityispiirrettä: funktionaalisten linkitettyjen listojen käyttö ja makrojärjestelmät. Liite esittelee myös Lispin sukuisten kielten listojen käsittelyssä käytettyjen funktioiden nimien historiaa. Linkitettyjen listojen käyttöä käsittelevä materiaali pätee suurelta osin myös muihin funktionaalisiin kieliin.

A.1 Funktionaaliset linkitetyt listat

Yhteen suuntaan linkitetyt listat ovat yleinen ohjelmoinnissa käytetty tietorakenne. Funktionaalisissa kielissä ja erityisesti Lispin sukuisissa kielissä ne ovat kuitenkin erityisasemassa. Useimmissa funktionaalisissa kielissä on tapana käyttää linkitettyä listaa sellaisissa tilanteissa, joissa imperatiivisessa kielessä käytettäisiin taulukkoa. Linkitetyt listat ovat erityisasemassa muun muassa siksi, että ne soveltuvat taulukoita paremmin esimerkiksi rekursiiviseen läpikäyntiin, ja uusien listojen rakentaminen rekursiivisesti lisäämällä uusia alkioita listan alkuun on luontevaa.

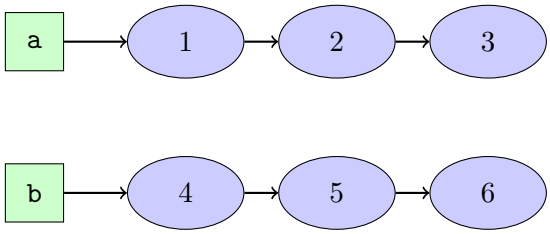
Usein linkitetyt listat toteutetaan funktionaalisissa kielissä pysyväistietorakenteina. Listat pysyvät tällöin muuttumattomina muistissa. Esimerkiksi jonkin listarakenteen esitys muistissa voi olla usean suuremman listarakenteen loppuosana. Tällöin operaatiot, jotka imperatiivisessa kielessä vaatisivat muutoksia listan sisältämiin arvoihin, voivat tehokkaasti tuottaa tuloksenaan näennäisesti uusia listarakenteita.

Kuva A.1 havainnollistaa listojen toimintaa Scheme-ohjelmassa. Esimerkiksi kahden listan katenointi toisiinsa ei siis tuota kokonaan uutta listaa, vaan voi jakaa listan loppuosan kahden listan kesken. Koska Scheme ei ole puhtaasti funktionaalinen kieli, listat eivät kuitenkaan ole täysin muuttumattomia. Täten esimerkiksi muutoksen tekeminen listan `b` viimeiseen alkioon näkyisi kaikissa kolmessa listassa, joiden osana lista `b` on: `b`, `ab` ja `c`.

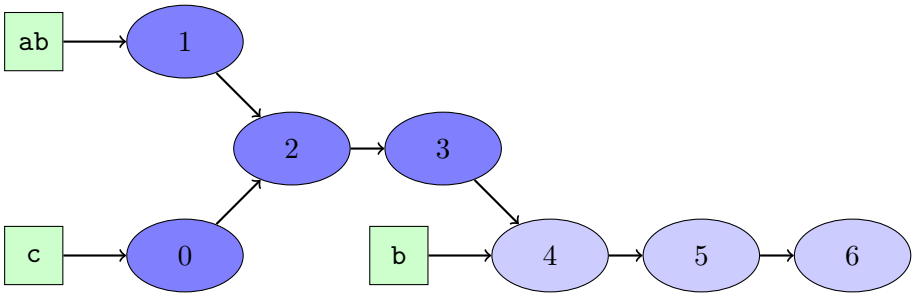
Aloittelevalle Lisp-ohjelmoijalle voivat aiheuttaa ihmetystä linkitettyjen listojen käsittelyssä käytettävien proseduurien erikoiset nimet. Proseduurien `car` ja `cdr` nimillä on kuitenkin historiallinen selitys.

Lispin alkuperäisen toteutusalustan, IBM 704:n 36-bittinen sana koostui kahdesta 15-bittisestä osasta nimiltään *address* ja *decrement* sekä *prefix*- ja *tag*-biteistä, ja sen assembly-kieli tuki käskyjä, jotka tekivät näiden osien erottamisesta alkuperäisestä sanasta helppoa. Alkuperäinen Lisp-toteutus sisälsi assembler-makrot `car` (***C**ontents of the **A**ddress part of **R**egister number*), `cdr` (***C**ontents of the **D**ecrement part of **R**egister number*), `cpr` (***C**ontents of the **P**refix part of **R**egister number*) ja `ctr` (***C**ontents of the **T**ag part of **R**egister number*), jotka ottivat parametrinaan muistiosoitteen ja palauttivat vastaavan osan sanasta [McC78].

```
1:  (define a (list 1 2 3))
2:  (define b (list 4 5 6))
```




```
3:  (define ab (append a b))
4:  (define c (cons 0 (cdr ab))) ; luo uuden listan, jossa listan ab
5:                               ; ensimmäinen alkio on korvattu nollalla
```



Kuva A.1: Linkitettyjen listojen esitys muistissa Scheme-ohjelmassa. Tummemmalla merkityt solut ovat riveillä 3 ja 4 lisättyjä uusia soluja. Listojen a ja b muistiesitys ei muutu rivien 3 ja 4 suorituksen aikana. Kuvan soluesitys on yksinkertaistettu: listan solun esitys muistissa ei välttämättä sisällä listan alkiota suoraan, vaan se voi sisältää viitteen muualla muistissa sijaitsevaan arvoon. Listan alkio voi myös olla toinen lista, jolloin listarakennetta voi käyttää puurakenteen esityksenä.

Address-osan sisältämä osoitin osoitti cons-solun sisältämään arvoon ja decrement-osan sisältämä osoitin seuraavaan cons-soluun. Näin cons-solu pystyttiin esittämään yhdellä 36-bittisellä sanalla. Jälkimmäiset kaksi operaatiota, `cpr` ja `ctr`, eivät kuitenkaan ole jääneet elämään myöhemmissä Lisp-versioissa tai -murteissa.

A.2 Makrojärjestelmät

Makrojärjestelmät ovat olleet alusta asti tyypillisiä Lisp-perheen kielille. Makroista puhuttaessa on kuitenkin syytä tähdentää, että Lisp-makrot eivät ole C-tyylisiä, *leksikaalisia* esiprosessorimakroja, jotka toimivat yksinkertaisesti korvaamalla ohjelmakoodin tekstistä makronimen esiintymät sen määritelmällä ennen varsinaista käännösvaihetta. Lisp-tyylisten, *syntaktisten* makrojen lavennus tehdään käännöksen tai tulkkauksen aikana ennen koodin suoritusvaihetta, esimerkiksi syntaksipuun esitystä käsittelemällä. Esimerkiksi Schemen Kawa-toteutus JVM-alustalla suorittaa dokumentaation mukaan

makrojen lavennuksen semanttisen analyysin yhteydessä [Bot16]. Tällöin kääntäjä voi makrolaajennusta suorittaessaan hyödyntää tietoa koodin semantiikasta, mikä esiprosessorimakrojen kanssa ei ole mahdollista.

Esimerkiksi Schemessä makroina voidaan määritellä standardin mukaiset boolean-operaattorit `and` ja `or`, monihaaraisen `if`-rakenteen korvaava `cond`-rakenne sekä useita paikallisten muuttujien määrittelyä helpottavia rakenteita (`let`, `letrec`, `let*`). Standardi ei kuitenkaan edellytä mainittujen ominaisuuksien toteuttamista makrojen avulla, eli rakenteet voivat olla toteutettuina myös suoraan kääntäjässä esimerkiksi tehokkuussyistä. Tyypillisesti makroina määritellään sellaisia operaatioita, joiden parametreja ei haluta evaluoida ennen kutsua, sillä Schemen funktiokutsut ovat arvokutsuja (engl. *call-by-value*) [Sus13, luku 1.1].

Schemen makrojärjestelmä on lisäksi siinä mielessä erityinen, että Scheme oli ensimmäinen ohjelmointikieli, joka toteutti *hygieeniset* makrot [Sus13, s. 3]. Hygieenisyydellä tarkoitetaan muun muassa sitä, että makron määritelmässä esiintyvät muuttujanimet eivät voi vahingossa päätyä viittaamaan tai kätkemään makron käyttöpaikan ympäristössä esiteltyjä nimiä [Que03, s. 341–342]. Esimerkiksi Common Lispin makrot eivät ole hygieenisiä, vaan niiden turvallinen käyttö edellyttää makron rungossa käytettävien muuttujanimien luontia `gensym`-funktiolla, joka luo taatusti ainutlaatuisen muuttujanimen.

B Cottontail Scheme -kääntäjä

Cottontail Scheme -kääntäjä koostuu F#-kielellä toteutetusta CLI-kääntäjästä sekä Scalalla toteutetusta JVM-kääntäjästä. Molemmat hyödyntävät samaa, CLI-kääntäjän osana toteutettua etuosaa. Tästä syystä molempien ohjelmien on ainakin toistaiseksi oltava läsnä koneella, jolla tuotetaan koodia JVM-alustalle. CLI-kääntäjää voidaan käyttää myös itsenäisesti, mikäli tuotetaan koodia vain .NET- tai Mono-alustalle.

Toteutukseen sisältyy lisäksi suoritusaikainen tukikirjasto, joka on CLI-alustalla toteutettu C#-kielellä ja JVM-alustalla Javalla. CLI-kääntäjä kopioi kirjaston `dll`-tiedoston samaan hakemistoon käännetyn ohjelman kanssa, jolloin ohjelman ajaminen on yksinkertaista. JVM-alustalla vaaditaan kääntäjän `jar`-tiedoston lisäys classpath-muuttujaan.

Toteutuksen versionhallinta on osoitteessa `https://github.com/Lateks/ct-scheme`. Koodin ylätason hakemistot ovat seuraavat:

CottontailScheme	CLI-toteutus ja kääntäjän etuosa
CottontailSchemeJVM	JVM-takaosa ja JVM-alustalla käytettävät kirjastot
CottontailSchemeLib	CLI-alustalla käytettävän kirjaston toteutus
CottontailSchemeTests	kääntäjän etuosan testit
test_programs	Scheme-kielellä kirjoitettuja testi- ja esimerkkiohjelmia

Varsinaisen koodinluonnin toteutus löytyy CLI-toteutuksessa tiedostosta `CottontailScheme/CodeGenerator.fs` ja JVM-toteutuksessa tiedostosta `CottontailSchemeJVM/src/main/scala/backend/CodeGenerator.scala`. JVM-alustan kirjastototeutus on hakemistossa `CottontailSchemeJVM/src/main/java/lib/`.

Kääntäjäjulkaisu on ladattavissa `zip`-pakettina osoitteesta `https://github.com/Lateks/ct-scheme/releases`. Samasta osoitteesta on myös mahdollista ladata toteutuksen koodi `zip`- tai `tar.gz`-pakettina, mikäli halutaan tarkastella koodia käyttämättä versionhallintaohjelmia.

Häntäkutsujen optimointi ja käännösvivut

Oletuksena toteutus optimoi häntäkutsut ja kääntää häntärekursiiviset funktiot silmukoiksi molemmilla alustoilla. Testaustarkoituksia varten molemmat optimoinnit on mahdollista kytkeä pois päältä käännösvivuilla.

`--notc`	kytkee yleistettyjen häntäkutsujen optimoinnin pois päältä
`--notailrec`	kytkee häntärekursion optimoinnin silmukaksi pois päältä

Samat käännösvivut toimivat molemmilla kohdealustoilla. Käännösvivut ovat toisistaan riippumattomia. Yleistettyjen häntäkutsujen optimoinnin pois kytkentä sallii

edelleen häntärekursiivisten funktioiden optimoinnin. Mikäli taas kytketään pois päältä ainoastaan häntärekursion optimointi, häntärekursiiviset funktiot optimoidaan käyttäen yleistettyjen häntäkutsujen optimointitekniikoita.

Käyttö CLI-alustalla

Kääntäminen Windows-alustalla (.NET):

```
> CottontailScheme.exe hello_world.scm
```

Kääntäminen Monolla:

```
> mono CottontailScheme.exe hello_world.scm
```

Tuotetun ohjelman ajaminen Windows-alustalla (.NET):

```
> HelloWorld.exe
```

Tuotetun ohjelman ajaminen Monolla:

```
> mono HelloWorld.exe
```

Käyttö JVM-alustalla

Kääntäminen Windows-alustalla:

```
> java -jar CottontailSchemeJVM.jar hello_world.scm
```

Kääntäminen Monoa käyttäen:

```
> java -jar CottontailSchemeJVM.jar --mono hello_world.scm
```

Kääntäjä on mahdollista ajaa myös putkittamalla takaosan JSON-tuloste JVM-takaosalle, kuten kuvailtiin luvussa 5.1:

```
> CottontailScheme.exe --json hello_world.scm |
  java -jar CottontailSchemeJVM.jar
```

Tuotetun ohjelman ajaminen vaatii sekä kääntäjän `jar`-tiedoston että ohjelman `class`-tiedoston sisältävän hakemiston lisäämisen Javan classpathiin. Ajaminen Windows-alustalla:

```
> java -cp "CottontailSchemeJVM.jar;." HelloWorld
```

Ajaminen muilla alustoilla:

```
> java -cp "CottontailSchemeJVM.jar:." HelloWorld
```

C Testiohjelmat

Tähän liitteeseen on koottu luvussa 4.4 käytetyt testiohjelmat. Testiohjelmat ja niiden puretut tavukoodit ovat ladattavissa myös GitHubista seuraavasta osoitteesta: `https://github.com/Lateks/mt-examples`. Osoitteesta löytyvät myös Cottontail Scheme -toteutuksen testiohjelmille luomat tavukoodit.

Liitteessä listataan ensin kaikki testiohjelmat Scheme-toteutuksina. Tämän jälkeen esitetään listaukset muita kieliä varten tehdyistä toteutuksista, mikäli lukija haluaa esimerkiksi kokeilla ohjelmia itse. Tavukoodit on jätetty tästä liitteestä pois niiden pituuden ja vaikean luettavuuden vuoksi. Tavukooditiedostojen rivit voivat olla hyvin pitkiä, mikä joissakin tilanteissa voi aiheuttaa yksittäisen rivin jakautumisen jopa 5–10 rivin matkalle A4-paperilla. Tavukoodista kiinnostunut lukija voi siis ladata tavukoodit yllä mainitusta osoitteesta tai kääntää ohjelmat itse ja tuottaa tavukoodin alla annetuilla ohjeilla.

Käännettäessä Bigloo-ohjelmia .NET-alustalla käytettiin häntäkutsut aktivoivaa käännösvipua `-fdotnet-full-tailc`. Lisäksi Bigloo käyttää oletuksena C-takaosaa, joten kohdealusta ilmoitetaan vivuilla `-dotnet` ja `-jvm`. Kawa-ohjelmat käännettiin vivulla `-full-tailcalls`. Bigloo-ohjelmat ovat muodoltaan hiukan erilaisia kuin tavalliset Scheme-ohjelmat johtuen muun muassa Bigloon omasta moduulijärjestelmästä, joten Bigloo-ohjelmat on toteutettu erikseen Kawa-ohjelmista. Kuva C.1 listaa käytetyt kääntäjäversiot.

Kieli/Toteutus	**Versio**
Java	1.8.0_72
Scala	2.11.7
Clojure	1.8.0
C# (Mono)	4.2.4.0
C# (.NET)	12.0.40629.0
F# (Mono)	- [a]
F# (.NET)	14.0.23020.0
Kotlin	1.0.0-beta-4589
Kawa	2.1
Bigloo (JVM)	4.1a
Bigloo (CLI)	2.8c [b]

[a] Monon F#-kääntäjä OS X -alustalla ei ilmoita versiota. F#-kieliversio on 4.0. Käytetty Mono-versio on 4.2.4.

[b] 2.8c oli tuorein löydetty Windows-asennusohjelma, joka tuki .NET-alustalle kääntämistä. Versiot muille alustoille eivät sisällä CLI-takaosaa. Dokumentaatio kuitenkin viittaa edelleen .NET-takaosaan, joten oletettavasti sen tukea ei ole lopetettu.

Kuva C.1: Luvussa tarkasteltavat kielet ja niiden toteutukset

Vertailua varten käännetyt ohjelmat purettiin tavukoodiksi JVM-alustan `javap`-ohjelman, Monon `monodis`-ohjelman ja .NET-alustan `ildasm`-ohjelman avulla. `Monodis` ja `ildasm` eivät vaadi purettavan `exe`-tiedoston lisäksi muita parametreja. JVM-ohjelmien `class`-tiedostot purettiin yksitellen komennolla `javap -c -s -l -verbose -private`.

C.1 Kaikki testiohjelmat Scheme-toteutuksina

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))

(display ((add 5) 4))

(define add10 (add 10))

(display (add10 90))
(display (add10 999))
```

Kuva C.2: Testiohjelma **add1** (koodirepositoriossa `ex1_add1`)

```
(define add
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (+ x y z)))))

(display (((add 5) 4) 3))
```

Kuva C.3: Testiohjelma **add2** (koodirepositoriossa `ex2_add2`)

```
(define increment-counter #f)
(define get-counter #f)

(define init-counter
  (lambda ()
    (define c 0)

    (set! increment-counter
      (lambda ()
        (set! c (+ c 1))))

    (set! get-counter
      (lambda () c))))

(init-counter)
(display (get-counter))
(increment-counter)
(display (get-counter))
(increment-counter)
(increment-counter)
(display (get-counter))
```

Kuva C.4: Testiohjelma **counter** (koodirepositoriossa `ex3_counter`)

```
(define double
  (lambda (x)
    (* x 2)))

(display (map double '(1 2 3 4 5)))
```

Kuva C.5: Testiohjelma **double** (koodirepositoriossa `ex4_double`)

```
(define fact
  (lambda (n)
    (define fact-helper
      (lambda (n acc)
        (if (zero? n)
            acc
            (fact-helper (- n 1) (* acc n)))))
    (fact-helper n 1)))

(display (fact 10000))
```

Kuva C.6: Testiohjelma **factorial** (koodirepositoriossa `ex5_factorial`)

```
(define odd?
  (lambda (x)
    (if (zero? x)
        #f
        (even? (- x 1)))))

(define even?
  (lambda (x)
    (if (zero? x)
        #t
        (odd? (- x 1)))))

(display (odd? 10))
(display (even? 10))
(display (odd? 9))
(display (even? 9))
(display (odd? 2147483647))
```

Kuva C.7: Testiohjelma **odd-even** (koodirepositoriossa `ex6_odd_even`)

C.2 Testiohjelma add1

C#: `SimpleClosure.cs`

```
using System;

public class SimpleClosure
{
    private static Func<int, int> Add(int x)
    {
        return y => x + y;
    }

    public static void Main()
    {
        Console.WriteLine(Add(5)(4));

        Func<int, int> add10 = Add(10);

        Console.WriteLine(add10(90));
        Console.WriteLine(add10(999));
    }
}
```

F#: SimpleClosure.fs

```
let add (x: int) =
    // printfn estää kääntäjää optimoimasta sulkeumaa kokonaan pois,
    // jotta pystytään näkemään sulkeumaesitys
    printfn "Creating closure add(%d)" x
    fun (y: int) -> x + y

[<EntryPoint>]
let main argv =
    printfn "%d" ((add 5) 4)
    let add10 = add 10
    printfn "%d" (add10 90)
    printfn "%d" (add10 999)
    0
```

Java: SimpleClosure.java

```
public class SimpleClosure {
    static Function<Integer, Integer> add(int x) {
        return (y) -> {
            return x + y;
        };
    }

    public static void main(String[] args) {
        System.out.println(add(5).apply(4));

        Function<Integer, Integer> add10 = add(10);
        System.out.println(add10.apply(90));
        System.out.println(add10.apply(999));
    }
}
```

Kotlin: SimpleClosure.kt

```
fun add(x: Int): (Int) -> Int {
    return { y -> x + y }
}

fun main(args: Array<String>) {
    println(add(5)(4))

    val add10 = add(10)
```

```
    println(add10(90))
    println(add10(999))
}
```

Scala: `SimpleClosure.scala`

```
object SimpleClosure {
    def main(args: Array[String]): Unit = {
        println(add(5)(4))

        val add10 = add(10)
        println(add10(90))
        println(add10(999))
    }

    def add(x: Int): Int => Int = {
        (y: Int) => x + y
    }
}
```

Clojure: `simple_closure.clj`

```
(ns simple-closure
  (:gen-class))

(defn add [x]
  (fn [y]
    (+ x y)))

(defn -main []
  (println ((add 5) 4))
  (def add10 (add 10))
  (println (add10 90))
  (println (add10 999)))
```

Bigloo: `simple_closure_bigloo.scm`

```
(module simple_closure (main main))

(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
(define main
  (lambda (args)
    (print ((add 5) 4))

    (define add10 (add 10))

    (print (add10 90))
    (print (add10 999))))
```

C.3 Testiohjelma add2

Testiohjelma toteutettiin kielillä, jotka käyttävät CLI-delegaatteja tai muuta delegaattityylistä toteutusta. Testitapausten **add1** ja **counter** perusteella tiedetään muiden kielitoteutusten Javaa lukuun ottamatta käyttävän viitesoluja, jolloin tämä tapaus ei vaadi erikoiskäsittelyä.

C#: `NestedClosure.cs`

```
using System;

public class NestedClosure
{
    private static Func<int, Func<int, int>> Add(int x)
    {
        return (y) => (z) => x + y + z;
    }

    public static void Main()
    {
        Console.WriteLine(Add(5)(4)(3));
    }
}
```

Bigloo: `nested_closure_bigloo.scm`

```
(module nested_closure (main main))

(define add
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (+ x y z)))))
```

```
(define main
  (lambda (args)
    (print (((add 5) 4) 3))))
```

C.4 Testiohjelma double

Testiohjelma toteutettiin kielillä, jotka käyttävät oliosulkeumia. Clojure-testiohjelmaa ei tarvittu, koska testitapauksen **add1** perusteella tiedetään Clojuren kääntävän kaikki ylätason funktiot sulkeumiksi, jolloin tähän tapaukseen ei tarvita erityiskäsittelyä.

F#: `DoubleX.fs`

```
let doubleX (x: int) = 2 * x

[<EntryPoint>]
let main argv =
    printfn "%A" (List.map doubleX [1; 2; 3; 4; 5])
    0
```

Kotlin: `DoubleX.kt`

```
fun double(x: Int): Int {
    return 2 * x
}

// Estää kääntäjää optimoimasta sulkeumaa pois.
fun getDouble(): (Int) -> Int {
    return ::double
}

fun main(args: Array<String>) {
    println(listOf(1, 2, 3, 4, 5).map(getDouble()))
}
```

Scala: `DoubleX.scala`

```
object DoubleX {
    def main(args: Array[String]): Unit = {
        println(List(1, 2, 3, 4, 5).map(doubleX))
    }

    def doubleX(x: Int): Int = {
        2 * x
```

```
    }
}
```

C.5 Testiohjelma counter

C#: Counter.cs

```
using System;

public class Counter
{
    public static Func<int> getCounter;
    public static Action incrementCounter;

    public static void initCounter() {
        int c = 0;
        getCounter = () => c;
        incrementCounter = () => c = c + 1;
    }

    public static void Main()
    {
        initCounter();
        Console.WriteLine(getCounter());
        incrementCounter();
        Console.WriteLine(getCounter());
        incrementCounter();
        incrementCounter();
        Console.WriteLine(getCounter());
    }
}
```

F#: Counter.fs

```
let createCounter () =
    let c = ref 0
    let incrementer = fun () -> c := !c + 1
    let getter = fun () -> !c
    (incrementer, getter)

[<EntryPoint>]
let main argv =
    let (increment, get) = createCounter()
    printfn "%d" (get ())
```

```
increment ()
printfn "%d" (get ())
increment ()
increment ()
printfn "%d" (get ())
0
```

Java: `Counter.java`

Esimerkki oman viitesoluesityksen käytöstä Javassa. Muuten tapausta ei ole mahdollista toteuttaa Javan sulkeumilla.

```
public class Counter {
    static IntSupplier getCounter;
    static Runnable incrementCounter;

    static void initCounter() {
        IntRef c = new IntRef(0);
        getCounter = () -> c.get();
        incrementCounter = () -> c.set(c.get() + 1);
    }

    public static void main(String[] args) {
        initCounter();
        System.out.println(getCounter.getAsInt());
        incrementCounter.run();
        System.out.println(getCounter.getAsInt());
        incrementCounter.run();
        incrementCounter.run();
        System.out.println(getCounter.getAsInt());
    }

    private static class IntRef {
        private int val;

        public void set(int val) {
            this.val = val;
        }

        public int get() {
            return val;
        }
```

```
        public IntRef(int val) {
            this.val = val;
        }
    }
}
```

Kotlin: `Counter.kt`

```
var incrementCounter: (() -> Unit)? = null
var getCounter: (() -> Int)? = null

fun initCounter(): Unit {
    var c = 0
    incrementCounter = { -> c = c + 1 }
    getCounter = { -> c }
}

fun main(args: Array<String>) {
    initCounter()
    println(getCounter!!())
    incrementCounter!!()
    println(getCounter!!())
    incrementCounter!!()
    incrementCounter!!()
    println(getCounter!!())
}
```

Scala: `Counter.scala`

```
object Counter {
  var incrementCounter: () => Unit = null
  var getCounter: () => Int = null

  def main(args: Array[String]): Unit = {
      initCounter()
      println(getCounter())
      incrementCounter()
      println(getCounter())
      incrementCounter()
      incrementCounter()
      println(getCounter())
  }
```

```
  def initCounter(): Unit = {
      var c = 0
      incrementCounter = () => c = c + 1
      getCounter = () => c
  }
}
```

Clojure: `counter.clj`

```
(ns counter
  (:gen-class))

(defn counter []
  (let [c (atom 0)
        increment-counter (fn [] (swap! c #(+ % 1)))
        get-counter (fn [] @c)]
    { :increment-counter increment-counter
      :get-counter get-counter }))

(defn -main []
  (let [ctr (counter)]
    (println ((:get-counter ctr)))
    ((:increment-counter ctr))
    (println ((:get-counter ctr)))
    ((:increment-counter ctr))
    ((:increment-counter ctr))
    (println ((:get-counter ctr)))))
```

Bigloo: `counter_bigloo.scm`

```
(module counter (main main))

(define increment-counter #f)
(define get-counter #f)

(define init-counter
  (lambda ()
    (define c 0)

    (set! increment-counter
      (lambda ()
        (set! c (+ c 1))))
```

```
    (set! get-counter
      (lambda () c))))

(define main
  (lambda (args)
    (init-counter)
    (print (get-counter))
    (increment-counter)
    (print (get-counter))
    (increment-counter)
    (increment-counter)
    (print (get-counter))))
```

C.6 Testiohjelma factorial

C#: Factorial.cs

```
using System;
using System.Numerics;

public class Factorial
{
    private static BigInteger FactHelper(BigInteger n, BigInteger acc) {
        if (n.IsZero)
            return acc;
        else
            return FactHelper(n - BigInteger.One, acc * n);
    }

    public static BigInteger Fact(int n) {
        return FactHelper(new BigInteger(n), BigInteger.One);
    }

    public static void Main()
    {
      // Tämä ylivuotaa .NET-alustalla.
      // Mono-alustalla vaaditaan suurempi syöte ylivuodon aikaansaamiseksi.
      Console.WriteLine(Fact(10000));
    }
}
```

F#: Factorial.fs

```
open System
open System.Numerics

let factorial (n: int): BigInteger =
    let rec factorialHelper (n: BigInteger) (acc: BigInteger) =
        if n.IsZero then
            acc
        else
            factorialHelper (n - BigInteger.One) (acc * n)
    factorialHelper (BigInteger n) BigInteger.One

[<EntryPoint>]
let main argv =
    printfn "%A" (factorial 10000)
    0
```

Java: Factorial.java

```
import java.math.BigInteger;

class Factorial {
    public static BigInteger factHelper(BigInteger n, BigInteger acc) {
        if (n.equals(BigInteger.ZERO))
            return acc;
        else
            return factHelper(n.subtract(BigInteger.ONE), acc.multiply(n));
    }

    public static BigInteger fact(int n) {
        return factHelper(BigInteger.valueOf(n), BigInteger.ONE);
    }

    public static void main(String[] args) {
        System.out.println(fact(100000));
    }
}
```

Kotlin: factorial.kt

```
import java.math.BigInteger

fun factorial(n: Int): BigInteger {
```

```
    tailrec fun factorialHelper(n: BigInteger, acc: BigInteger): BigInteger {
        if (n == BigInteger.ZERO)
            return acc
        else
            return factorialHelper(n - BigInteger.ONE, acc * n)
    }

    return factorialHelper(BigInteger.valueOf(n.toLong()), BigInteger.ONE)
}

fun main(args: Array<String>) {
  print("Result = \${factorial(10000)}\n")
}
```

Scala: `Factorial.scala`

```
object Factorial {
    def main(args: Array[String]): Unit = {
      val res = fact(10000)
      println(res)
    }

    def fact(n: Int): BigInt = {
      def fact_helper(n: Int, acc: BigInt): BigInt = {
        if (n == 0) acc
        else fact_helper(n - 1, acc * n)
      }
      fact_helper(n, BigInt(1))
    }
}
```

Clojure: `factorial.clj`

```
(ns factorial
  (:gen-class))

(defn fact [n]
  (loop [n n acc 1]
    (if (zero? n)
      acc
      (recur (- n 1) (*' acc n)))))

(defn -main []
```

```
  (println (fact 10000)))
```

Bigloo: `factorial_bigloo.scm`

```
(module examples (main main))

(define fact
  (lambda (n)
    (define fact-helper
      (lambda (n acc)
        (if (zero? n)
            acc
            (fact-helper (- n 1) (* acc n)))))
    (fact-helper n 1)))

(define main
  (lambda (args)
    (display (fact 10000))))
```

C.7 Testiohjelma odd-even

Testiohjelmaa ei toteutettu Kotlinilla, koska dokumentaation perusteella tiedetään jo vain häntärekursion olevan optimoitavissa `tailrec`-avainsanalla. Vastaavasti C#:n ja Javan tapauksessa tapaus on selkeä jo ennalta, koska kumpikaan toteutus ei optimoi edes **factorial**-tapausta, mutta testiohjelmat esitetään kuitenkin tässä.

C#: `MutualRecursion.cs`

```
using System;

public class MutualRecursion
{
    private static bool IsOdd(int n) {
        if (n == 0)
            return false;
        return IsEven(n - 1);
    }

    private static bool IsEven(int n) {
        if (n == 0)
            return true;
        return IsOdd(n - 1);
    }
```

```
    public static void Main() {
        Console.WriteLine(IsOdd(10));
        Console.WriteLine(IsEven(10));
        Console.WriteLine(IsOdd(9));
        Console.WriteLine(IsEven(9));
        Console.WriteLine(IsOdd(2147483647));
    }
}
```

F#: MutualRecursion.fs

```
open System

let rec isOdd (n: int): bool =
        if n = 0 then
            false
        else
            isEven(n - 1)
    and isEven (n: int): bool =
        if n = 0 then
            true
        else
            isOdd(n - 1)

[<EntryPoint>]
let main argv =
    printfn "%b" (isOdd 10)
    printfn "%b" (isEven 10)
    printfn "%b" (isOdd 9)
    printfn "%b" (isEven 9)
    printfn "%b" (isOdd 2147483647)
    0
```

Java: MutualRecursion.java

```
class MutualRecursion {
    private static boolean isOdd(int n) {
        if (n == 0)
            return false;
        return isEven(n - 1);
    }
```

```
    private static boolean isEven(int n) {
        if (n == 0)
            return true;
        return isOdd(n - 1);
    }

    public static void main(String[] args) {
        System.out.println(isOdd(10));
        System.out.println(isEven(10));
        System.out.println(isOdd(9));
        System.out.println(isEven(9));
        System.out.println(isOdd(2147483647));
    }
}
```

Scala: `MutualRecursion.scala`

```
object MutualRecursion {
    def main(args: Array[String]): Unit = {
        println(isOdd(10))
        println(isEven(10))
        println(isOdd(9))
        println(isEven(9))
        println(isOdd(2147483647))
    }

    def isOdd(n: Int): Boolean = {
        if (n == 0)
            return false
        return isEven(n - 1)
    }

    def isEven(n: Int): Boolean = {
        if (n == 0)
            return true
        return isOdd(n - 1)
    }
}
```

Clojure: `mutual_recursion_trampolines.clj`

Esimerkki `trampoline`-funktion käytöstä Clojuressa. Dokumentaation perusteella tiedetään, että Clojure ei optimoi ei-rekursiivisia häntäkutsuja [Hic16b]. Koodirepositoriosta

löytyy kuitenkin myös versio ilman trampoliinien käyttöä.

```
(ns mutual-recursion-trampolines
  (:gen-class))

(declare is-even-fn)

; Palauttaa nyt sulkeuman
(defn is-odd-fn [n]
  (do
    (if (zero? n)
      false
      #(is-even-fn (- n 1)))))

(defn is-even-fn [n]
  (do
    (if (zero? n)
      true
      #(is-odd-fn (- n 1)))))

(defn is-odd [n]
  (trampoline is-odd-fn n))

(defn is-even [n]
  (trampoline is-even-fn n))

(defn -main []
  (println (is-odd 10))
  (println (is-even 10))
  (println (is-odd 9))
  (println (is-even 9))
  (println (is-odd 2147483647)))
```

Bigloo: `mutual_recursion_bigloo.scm`

```
(module odd-even (main main))

(define odd?
  (lambda (x)
    (if (zero? x)
        #f
        (even? (- x 1)))))
```

```
(define even?
  (lambda (x)
    (if (zero? x)
        #t
        (odd? (- x 1)))))

(define main
  (lambda (args)
    (display (odd? 10))
    (display (even? 10))
    (display (odd? 9))
    (display (even? 9))
    (display (odd? 2147483647))))
```