Oliokielten välikielten optimointi

Pietu Pohjalainen

Helsinki 6.9.2006

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Tiedekunta/Osasto — Fakultet/Sektion — Faculty
Matemaattis-luonnontieteellinen

Laitos — Institution — Department
Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Pietu Pohjalainen

Työn nimi — Arbetets titel — Title

Oliokielten välikielten optimointi

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	6.9.2006	67 + 4

Tiivistelmä — Referat — Abstract

Javan myötä ohjelmointikielten kääntämisprosessiin on uudelleen esitelty erityisen käsittelyn kohteeksi kelpaava välikieli, tavukoodi. Tavallisesti Java-ohjelmaa suoritettaessa erityinen virtuaalikone lataa tavukoodimuotoisen ohjelman esityksen, jota suoritetaan joko tulkkaamalla tai suoritusaikaisesti suoritusalustan ymmärtämälle kielelle kääntäen.

Tässä tutkielmassa tutkitaan välikielen tasolla tapahtuvia optimointimahdollisuuksia. Oliokielten dynaamisen luonteen vuoksi puhtaasti staattinen optimointi on vaikeaa ja siksi usein hedelmätöntä. Tutkielman yhteydessä kuitenkin tunnistettiin mobiiliohjelmointiin soveltuva suljetun maailman oletus, jonka puitteissa tavukoodin tasolla voidaan ohjelmaa parannella turvallisesti. Esimerkkinä tutkielmassa toteutetaan ylimääräisiä rajapintaluokkia poistava optimointi.

Koska optimointialgoritmit ovat usein monimutkaisia ja vaikeaselkoisia, tutkitaan työssä myös mahdollisuuksia niiden yksinkertaisempaan esittämiseen. Alunperin Javalla toteutetun luokkahierarkiaa uudelleenjärjestelevän algoritmin esiehtojen tarkastus onnistutaan kuvaamaan ensimmäisen kertaluokan logiikan kaavalla, jolloin esiehtojen tarkastus onnistuu tutkielman puitteissa toteutetulla logiikkakoneella. Logiikkakoneelle kuvataan logiikkakaavojen propositiot Javalla, mutta propositioiden yhdistely tapahtuu ja-konnektiiveja käyttävällä logiikkakielellä. Suorituskyvyltään logiikkakone on joissain tapauksissa Java-toteutusta nopeampi.

ACM Computing Classification System (CCS): D.1.0, D.2.3, D.3.4

Avainsanat — Nyckelord — Keywords
ohjelmointikielten kääntäjät, optimointi, oliokielet, välikielet

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Oliot, välikielet ja optimointi **3**

2.1 Olio-ohjelmointi 3

2.2 Kääntäjien käyttämät välikielet 9

2.3 Optimointi 18

3 Ohjelman analysointi ja parantelu **25**

3.1 Analysoijan maailmankuva 25

3.2 Kontrollivuokaavio ja peruslohkot 26

3.3 Kontrollivuoverkko ja kutsukaavio olio-ohjelmissa 30

3.4 Pakenemisanalyysi 33

3.5 Profilointi 34

3.6 Viestin vastaanottajien muistaminen 36

3.7 Luokkahierarkiaa uudelleenjärjestelevä optimointi 38

4 Optimoivan järjestelmän toteutus **41**

4.1 Optimointien määritteleminen formaalein menetelmin 41

4.2 Emoole 43

4.3 Uudelleenjärjestelevä optimointi Emoolella 49

4.4 Suorituskykymittaukset 50

5 Yhteenveto **57**

Lähteet **59**

A Liitteet **1**

A.1 Luokkahierarkian muodostava algoritmi Javalla 1

A.2 Poistettavat rajapintaluokat listaava Java-ohjelma 3

1 Johdanto

Ohjelmointikielen kääntäjä on ohjelma, joka muuntaa syötteenään saamansa lähdekielisen ohjelman semanttisesti samansisältöiseksi kohdekieliseksi ohjelmaksi. Koska suoraviivainen kääntäminen tuottaa helposti tehotonta koodia, tarvitaan *optimoijaa*, joka muokkaa kääntäjän tuottamaa koodia tehokkaampaan muotoon [ASU86, s. 585].

Optimointia käytetään pääasiassa kahden tavoitteen saavuttamiseen: joko parantamaan käännettävää ohjelmaa esimerkiksi suoritusympäristön asettaman rajan paremmalle puolelle tai yleisesti parantamaan käänettävän ohjelman suorituskykyä sopivan mittarin suhteen. Kääntäjään suunnitellaan optimointeja kustannustehokkuussyistä: monet parantelut ovat mekaanisia, mutta aikaa vieviä uudelleenjärjestelyitä, joita ei kannata antaa ihmisohjelmoijan tehtäväksi.

Tarkkaan ottaen on väärin kutsua ohjelmallisesti tapahtuvaa koodin parantelua optimoinniksi, sillä matemaattisessa mielessä tällä tarkoitettaisiin jollakin mittarilla mitattuna tehokkaimman mahdollisen muodon etsimistä. Käännösprosessin monimutkaisuuden hallitsemiseksi käytetään erilaisia väliesitysmuotoja, eli välikieliä, jotka prosessissa sijaitsevat lähde- ja kohdekielten välissä. Näin ajateltuna ohjelman kääntäminen on sarja muunnoksia korkeamman tason kielestä matalampaan siten, että kullakin askeleella joitakin korkeamman abstraktiotason rakenteita muunnetaan matalammalle tasolle.

Tässä tutkielmassa tarkastellaan oliosuuntautuneita välikieliä, eli kieliä, jotka sisältävät oliokielen kääntämistä helpottavia ominaisuuksia. Oliokielet sisältävät nykyisin dynaamisia ominaisuuksia, jotka tekevät suoritusta edeltävistä optimoinneista vaarallisia, sillä kunkin optimoivan muunnoksen tulee säilyttää käännettävän ohjelman semantiikka. Asian tekee ongelmalliseksi tämän semantiikan ymmärtäminen: kääntäjän tulee varautua myös sellaisten elementtien olemassaoloon, jotka eivät välttämättä ole olemassakaan vielä käännöksen suoritusaikana. Tutkielmassa keskitytään erityisesti olio-ominaisuuksien toteutus-, ja optimointimenetelmiin sekä niiden tuottamiin ongelmiin. Tämän vuoksi monet tärkeät olio-ohjelmien käytännön tehokkuuteen vaikuttavat tekijät, kuten roskienkeruualgoritmit ja rinnakkaisohjelmoinnin synkronointimekanismien toteutusmenetelmät jäävät työn aihealueen ulkopuolelle.

Empiirisenä osana toteutetaan olio-ohjelmiin upotettavissa oleva logiikkajärjestelmä, jolla voidaan joidenkin optimointien esiehtoja kuvata yksinkertaisia abstraktiokomponentteja yhdistelemällä. Logiikkajärjestelmässä käytettävää kyselyä verrataan vastaavaan Javakieliseen toteutukseen.

Tutkielma jakaantuu osiin seuraavasti. Luvussa 2 määritellään tutkielman kannalta olen-

naiset yleiset termit ja käsitteet. Luvussa 3 esitellään ohjelmallisia analyysejä ja niiden mahdollistamia optimoivia muunnoksia. Luvussa 4 esitellään tutkielman yhteydessä toteutettu järjestelmä, jolla voidaan deklaratiivisella kielellä määritellä erilaisia optimoitavia ohjelman rakenteita sekä järjestelmän käytäntöön soveltamisen tulokset. Lopuksi luvussa 5 tehdään yhteenveto tutkielmassa käsitellyistä aiheista. Liitteissä A.1 ja A.2 esitetään Java-kielinen toteutus empiirisessä osassa toteutetulle logiikkakyselylle.

2 Oliot, välikielet ja optimointi

Tässä luvussa määritellään tutkielmassa käytettävät yleiset käsitteet olio-ohjelmointi, välikielet ja optimointi. Koska nämä käsitteet ovat alan kirjallisuudessa monessa yhteydessä käytettyjä, rajataan käsittely kääntäjäkirjoittajan näkökulmasta tapahtuvaksi. Lisäksi luvussa tarkastellaan annettujen määritelmien suhdetta muihin läheisesti asiaan liittyviin käsitteisiin.

2.1 Olio-ohjelmointi

Perinteisempään rakenteiseen ohjelmointiin verrattaessa olio-ohjelmoinnissa (engl. *object-oriented programming*) ohjelman suunnittelun perusyksikkö on olio, joka kommunikoi toisten olioiden kanssa viestein. Erityisesti Smalltalk-kielen [GR83] kehittäjät korostavat olio-ohjelmointia luonnehdittaessa tätä viestinvälitykseen perustuvaa ajattelutapaa yksittäisten teknisten määritelmien sijaan. Olio-ohjelmoinnin voidaan kuitenkin katsoa alkaneen järjestelmien simulointiin suunnitellusta Simula 67-kielestä [ND78]. Se sisältää kaikki tässä tutkielmassa oliokielen piirteiksi katsottavat ominaisuudet:

- Tietokenttien ja niitä käsittelevien operaatioiden kapselointi (engl. *encapsulation*) yhteen.
- Simulaation ongelma-alueen jäsentäminen käsitehierarkian ja perinnän (engl. *inheritance*) avulla.
- Operaatioiden myöhäinen sidonta (engl. *dynamic binding*).

Tämä jako on mm. Scottin [Sco00, s. 530], Sebestan [Seb99, s. 437] sekä Preen [Pre95, s. 12–47] esittämien oliokielten määritelmien mukaisia. Myöhemmin tässä luvussa kaikki kolme käsitettä käsitellään tarkemmin.

On myös olemassa kieliä, joissa voidaan jollain tasolla käsitellä tilan ja operaatiot kapseloivia olioita, mutta joita ei voida kutsua oliosuuntautuneiksi kieliksi. Wegner esittää heikompaa oliokielen määritelmää *olioperustainen kieli* (engl. *object-based*) käytettäväksi niistä kielistä, joissa ei tunneta perinnän ja luokan käsitteitä. Näin itse päätermi voitaisiin määritellä seuraavasti [Weg87]:

```
object-oriented = objects + classes + inheritance
```

ja esitetty heikompi määritelmä jäisi käyttöön toisaalta niille proseduraalisten ohjelmointikielten sellaisille oliolaajennuksille, jotka eivät vielä täytä vahvempaa määritelmää mutta myös niille prototyyppikielille, jotka käyttävät luokan käsitteen sijaan toisia olioita uusien olioiden määrittelymenetelmänä. Eräät tutkijat taas esittävät, että juuri prototyyppipohjaisten kielten tulisi toimia esimerkillisinä oliokielten otoksina, sillä näin voidaan käsitteistöstä poistaa luokan käsite turhana [Tai93b, s. 28].

Tämän tutkielman aihepiirin kannalta ei kuitenkaan ole tarkoituksenmukaista erotella olioiden erilaisia luokka- ja prototyyppipohjaisia määrittelymekanismeja tämän tarkemmin. Tämän vuoksi tutkielmassa keskitytään yksittäisten piirteiden ominaisuuksien tarkasteluun ja käytetään termiä *oliokieli* laajassa merkityksessä.

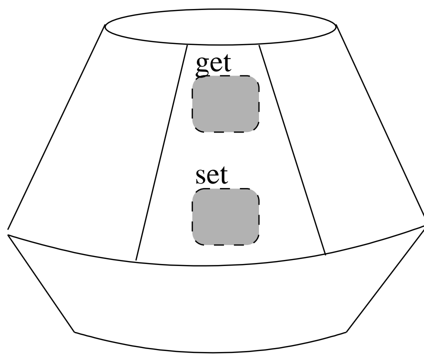
Vaikka kaikilla ohjelmointikielillä on mahdollista toteuttaa olio-ohjelmille ominaisia periaatteita noudattavia ohjelmia, eivät kaikki kielet silti välttämättä ole oliokieliä. Stroustrup huomauttaa, että ohjelmointikieli tukee olio-ohjelmointia (ja on siten oliokieli) vain, jos kieli tekee tällaisten piirteiden toteuttamisen helpoksi [Str88]. Toisin sanoen, mikäli piirteen toteuttaminen kielellä vaatii erityistä taitoa, ei voida puhua kielen tukevan tätä piirrettä. Oliokielet siis tekevät kapseloinnin, perinnän ja myöhäisen sidonnan käyttämisen helpoksi.

Kenttien ja operaatioiden kapselointi ja tiedon piilotus

Kapseloinnilla tarkoitetaan olion attribuuttien ja niitä käsittelevien operaatioiden modularisoimista sekä tietokenttien suojaamista suoralta ulkoapäin tapahtuvalta käsittelyltä. Tällä haetaan mahdollisuutta sijoittaa attribuuttien käyttämiseen tarkoitettu ohjelmalogiikka samaan paikkaan itse attribuuttien kanssa. Attribuuttien rakenteen tai operaatioiden toteutuksen muuttuessa voidaan joissain tapauksissa rajoittaa vaaditut muutokset kapseloidun yksikön alueelle sen sijaan, että koko ohjelma vaatisi muokkausta.

Kuvassa 2.1 esitetään kapseloitu olio ja sen Java-kielinen toteutus, jonka tehtävänä on säilyttää viite johonkin toiseen olioon. Kapseli tarjoaa julkisen rajapinnan, jonka kautta kapselin palveluita käyttävät tahot saavat pääsyn palveluihin. Koska esimerkissä kapselion tietokenttä *sisältö* on määritelty private-tyyppiseksi, ei se ole nähtävissä olion määrittelevän luokan ulkopuolelta. Tämän seurauksena kapselia käyttävät tahot joutuvat aina kulkemaan määriteltyjen get- ja set-palveluiden kautta, jolloin näiden kapselin sisäistä toteutusta voidaan vaihtaa vapaasti, kunhan toteutus vain täyttää kapselin ulkopuolelle tarjotun rajapinnan vaateet.

Kapseloinnin avulla toteutettava moduulin sisäistä toteutusta ja rakennetta koskevan tie-




```
public class Kapseli
{
      private   Object   sisältö = null;

      public   void    set(Object arg)   {  sisältö = arg;   }
      public   Object get()                 {   return sisältö; }
}
```

Kuva 2.1: Kapseli, jonka sisältöä voidaan käyttää get- ja set-metodeilla

don piilotus on apuväline ohjelman sisäisten abstraktioiden rakentamiseen. Palvelun käyttäjän kannalta epäolennaiset yksityiskohdat piilottamalla saadaan ohjelmoijan kognitiivista kuormaa pienennettyä. Toisaalta kapseloitujen olioiden ajatteleminen palveluita tarjoavina, vaihdettavissa olevina virtuaalikoneina mahdollistaa samalle käsitteelle useamman toteutuksen tuottamisen, joista sopivinta käytetään kulloisenkin tarpeen mukaan.

Tämä alun perin Parnasin esittämä [Par72] moduulin kapseloinnin malli, jossa tietokenttien ja koodin sijoittelupaikan lisäksi moduuli nähdään sopimuksena palvelun tuottajan ja tarjoajan välillä, kohtasi aluksi vahvaa vastustusta. Samalla vuosikymmenellä Brooks väitti moduulin yksityiskohtien peittämisen vaativan loppuun saakka suunniteltuja, täydellisesti ja täsmällisesti tehtyjä rajapintamäärittelyitä — ja argumentoi tämän olevan teoreettisesti kaunista, mutta käytännössä mahdotonta [Bro75, s. 78]. Jokaisen palvelukutsun kierrättämisen palvelurajapinnan kautta epäiltiin myös estävän riittävän tehokkuustason saavuttamisen [Lin76].

Myöhemmin, koneiden nopeuduttua Mooren lain [Moo65] mukaisesti kapseloinnilla saavutettu joustavuus on osoittautunut mikrotason tehokkuutta tärkeämmäksi ohjelmistotuotannon periaatteeksi. Lisäksi kääntäjäteknologian kehittymisen myötä ylimääräisiä abstraktioita osataan nykyisin purkaa, kuten esimerkiksi tämän tutkielman empiirisessä osassa näytetään. Tämän vuoksi myös Brooks tunnustaa nykyisin tiedon piilotuksen olevan ainoa tapa ohjelmistojen suunnittelun abstraktiotason nostamiseen [Bro95, s. 271-273].

Perintä

Kapseloinnin ohella perintä on olio-ohjelmoinnissa varsin keskeinen piirre. Siinä missä abstraktien tietotyyppien tarjoamien palveluiden mukaiseen moduuliajatteluun kannustavia kielten piirteitä löytyy myös ei-oliokielistä, on eräiden tutkijoiden mukaan perintä se

ominaisuus, joka erottaa olioajattelun muista ohjelmointiparadigmoista. Vaikka perinnän tärkeydestä olio-ohjelmoinnin määritelmässä ollaan yleisesti yhtä mieltä, esiintyy sen semantiikasta ja käyttötarkoituksista silti suuresti vaihtelevia tulkintoja, joista Taivalsaari esittää kattavan yhteenvedon [Tai96].

Yleisellä tasolla ajateltuna perintä tarkoittaa uusien käsitteiden määrittelemistä olemassa olevia käsitteitä käyttäen. Tämä ajatus voidaan formalisoida kaavalla [BC90]:

$$R = P \oplus \Delta R \tag{2.1}$$

jossa R kuvaa juuri määriteltyä uutta oliota, P olion kantaoliota, ΔR esittää ominaisuuksia, jotka erottavat R:än P:stä ja yhdistämisoperaatio $\oplus$ menetelmää, jolla eroavuudet yhdistetään lopputulokseksi.

Itse inkrementtioperaation $\oplus$ toteutuksista voidaan löytää kaksi toisistaan poikkeavaa mallia: katenoiva ja delegoiva. Katenoivassa mallissa kantaoliosta perittävät eroavuudet kopioidaan uuden olion ominaisuuksiksi, jolloin ominaisuutta käytettäessä sen määritelmä löytyy oliosta itsestään. Delegaatiomallissa taas oliossa pidetään muistissa jonkinlaista viittausta kantaolioon tai -luokkaan, ja käytettävän ominaisuuden puuttuessa viittauksia seurataan perintäketjuja pitkin siinä toivossa, että jokin ketjun lenkki määrittelisi halutun ominaisuuden [Tai93a].

Delegaatiomallissa olion vastaanottaessa viestin, johon se ei osaa vastata, ohjataan viesti kantaoliolle. Katenaatiomallissa kaikki perintäketjut latistetaan käännettäessä siten, että olion tunnistamat viestit sijaitsevat oliossa itsessään. Tämän seurauksena katenaatiomallissa ns. Fragile base class -ongelmaa [MS98] ei esiinny, sillä olion vanhemman muuttuessa muutokset eivät propagoidu olioon itseensä. Delegaatiomallia noudattavalla luokkapohjaisella kielellä kirjoitettu ohjelma voidaan muuntaa katenaatiomalliseksi käännöksen jälkeisellä muunnoksella, jossa kaikkien luokkaa perintähierarkiassa edeltävien luokkien toteutukset yhdistetään luokkaan itseensä.

Katenaatiomallissa puolestaan voidaan toteuttaa delegaatio mahdollistamalla jollakin tavalla vanhempaan kohdistuvien muutosten propagoiminen perintähierarkiaa alaspäin. Esimerkiksi Kevossa [Tai92] voidaan ADD-, REMOVE- ja RENAME -operaatioilla lisätä ominaisuuksia yksittäisiin olioihin. Lisäksi kielessä on ADD*-, REMOVE*-, ja RENAME* -operaatiot, joilla vaikutetaan oliosta perittyihin toisiin olioihin. Jälkimmäisiä operaatioita käytetään silloin, kun nimenomaisesti halutaan muokata myös oliosta perittyjä jälkeläisiä.

Määritelmä 2.1 on siinä mielessä yleiskäyttöinen, että sillä saadaan mallinnettua sekä luokka- että prototyyppipohjaiset perintämekanismit, sillä yhdistämisoperaatiota $\oplus$ tai

olioiden ominaisuuksia kuvaavien joukkojen P ja ΔR ominaisuuksia ei määritelmässä kiinnitetä millään tavalla. Näistä syystä määritelmä kattaa myös niin moniperinnän kuin yksiperinnän sekä delegaatiopohjaisen perinnän ja katenointipohjaisen perinnän.

Vaikkakin yllä esitetyllä määritelmällä voidaan mallintaa perinnän toimintaa, esiintyy perinnän käyttötarkoituksissa eroja. Simula-kielen alkuperäinen idea oli mallintaa sovelluksen kohdealuetta olioina ja niiden välisinä suhteina. Tätä perinnän käyttötapaa kutsutaan Simulan kehityspaikan perusteella perinnän skandinaaviseksi koulukunnaksi. Reaalimaailman mallintamisen sijaan perinnän käyttämistä ohjelmointiteknisenä koodin uudelleenkäytön välineenä kutsutaan puolestaan amerikkalaiseksi koulukunnaksi [Sch94].

Myöhäinen sidonta

Viestinlähetyksen myöhäinen sidonta tarkoittaa olio-ohjelmoinnissa perintää täydentävää ajattelutapaa, jonka seurauksena viestin vastaanottaja tiedetään vasta suorituksen aikana. Näin ollen metodikutsuissa voidaan tehdä kutsuvan ja kutsuttavan ohjelmakoodin välinen sidonta vasta kun kutsuttava tunnetaan tarkasti.

Myöhäisestä sidonnasta on kahdenlaista mallia: yleisempi, valtavirran oliokielissä käytettävä viestin vastaanottajan luokan perusteella toteutettava myöhäinen sidonta (engl. *single dispatch*) sekä harvinaisempi viestin kaikkien argumenttien luokkien perusteella toteutettava myöhäinen sidonta (engl. *multi dispatch, multi methods*), jota käytetään Common LISP:in oliolaajennus CLOS:issa [Ste84] sekä muutamissa tutkimuskielissä, kuten Cecilissä [Cha92].

Viestin vastaanottavan metodin valintaa eri viestinvälitysmekanismeilla voidaan selventää seuraavalla Javan kaltaisella kielellä kirjoitetulla esimerkillä:

```
public void foobar(String str, Object obj) {
    print('foobar: str: obj,');
}

public void foobar(String str, String obj) {
    print('foobar: str: str.');
}

public void main() {
    String str = new String();
    Object obj = str;
```

```
    foobar(str, obj);
}
```

Esimerkissä kaikki metodit ovat myöhään sidottuja. Kun käytetään Javan tapaan pelkästään vastaanottajan luokan mukaan tapahtuvaa sidontaa, eli ensimmäinen argumentti on viestin vastaanottava olio, ohjelma tulostaa

```
foobar: str: obj,
```

sillä kutsuvassa pääohjelmassa viimeisen argumentin käännösaikaisesti sidottu tyyppi on Object. Mikäli käytetään kaikkien argumenttien luokkien mukaan tapahtuvaa sidontaa ohjelma kuitenkin tulostaa

```
foobar: str: str.
```

Jälkimmäisen esimerkin mukaisen viestinvälityksen voi saada aikaan siirtämällä esimerkin multimetodeja käyttävälle ohjelmointikielelle tai vaikkapa suorittamalla ohjelman multimetodit toteuttavassa Javan virtuaalikoneessa [DLS$^+$01]. Optimoinnin kannalta näiden kahden menetelmän välinen ero on esimerkki siitä, että yhdelle viestinsidontamenetelmälle viriteltyä optimointia ei voida soveltaa semanttisesti toisenlaisen menetelmän optimoimiseen. Tämän tutkielman kannalta multimetodit antavat puolestaan luontevan toteutustavan luvussa 4 esitettävän logiikkakoneen primitiivien toteutukselle.

Suunnittelumallit

Suunnittelumallit (engl. *design patterns*) ovat tunnettuja malleja usein esiintyvien suunnitteluongelmien ratkaisemiseksi. Standardoidulla ratkaisuilla parannetaan ohjelmien ylläpidettävyyttä ja ymmärrettävyyttä, sillä kunkin suunnittelumallin käytön motivaatiot ja seuraukset ovat dokumentoituja. Alun perin rakennusarkkitehtuurin puolella [AIS$^+$77, Ale87] syntynyt idea on sittemmin adoptoitu ohjelmoinnin puolelle.

Olio-ohjelmien optimoinnin kannalta erityisesti Gamman ja kumppanien aluksi konferenssissa esittämät [GHJV93], ja sittemmin katalogiksi kokoamat [GHJV95] suunnittelumallit ovat tärkeitä, sillä ne keskittyvät juuri olio-ohjelmoinnissa esiintyvien suunnitteluongelmien ratkaisemiseen: miten kapselointia, perintää ja myöhäistä sidontaa käytetään ylläpidettävien ohjelmien koostamiseen. Gamman ja kumppanien suunnittelumallit ovat tärkeitä olio- ohjelmien optimointia ajatellen, sillä ne esittävät rakenteita, joita olio- ohjelmissa oletettavasti tavataan.

Yleisemmin ajateltuna ei olio-ohjelmoinnin ja suunnittelumallien välillä kuitenkaan ole sen erityisempää yhteyttä sen enempää kuin rakennusarkkitehtuurin ja ohjelmistoarkkitehtuurinkaan välillä. Suunnittelumalli on yleiskäsite, jolla viitataan yleiskäyttöisen ratkaisun ideaan. Yksittäisistä suunnittelumallien ilmentymistä, kuten Ainokaisesta (engl. Singleton) [GHJV95, s. 127] tai Juoksukaisesta (engl. Runabout) [Gro03], voidaan puolestaan puhua suhteessa olio-ohjelmointiin. Toisaalta käytössä oleva ympäristö vaikuttaa siihen, minkä voidaan sanoa olevan suunnittelumallin mukaista. Esimerkiksi rakenteisen ohjelmoinnin yhteydessä voitaisiin puhua kapseloinnin, perinnän ja myöhäisen sidonnan olevan hyviä suunnittelumalleja [GHJV95, s. 4]. Vastaavasti vaikkapa Iteraattori-mallin (engl. Iterator) [GHJV95, s. 257] ohjelmointikielen tasolla toteuttavalla kielellä, kuten *Icon*illa [GG83], ohjelmoitaessa ei erityisesti ajatella käytettävän kyseistä suunnittelumallia.

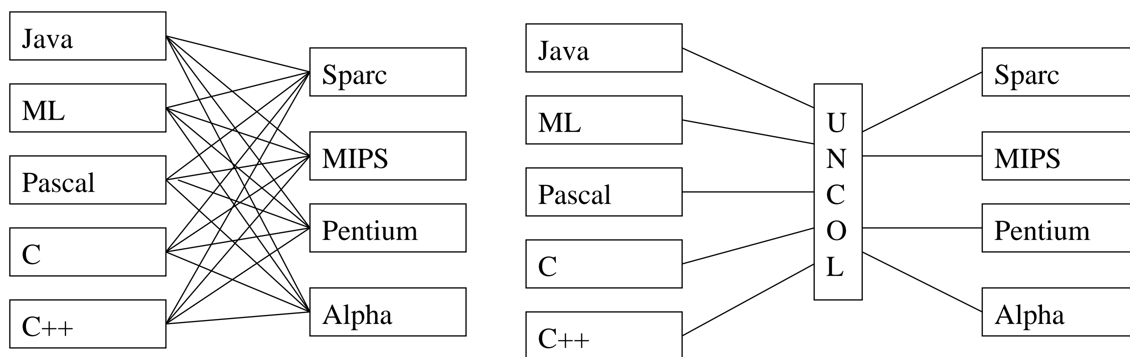
2.2 Kääntäjien käyttämät välikielet

Ohjelmointikielen kääntämiseen liittyvän monimutkaisuuden hallitsemiseksi kääntäjille on kehitetty prosessin eri vaiheisiin jakava referenssiarkkitehtuuri [ASU86, s. 10]. Kussakin vaiheessa saadaan rajattua kyseisen vaiheen tehtävien laajuutta, jolloin itse toteutustyö yksinkertaistuu. Käännösprosessin keskipaikkeilla sijaitsee välikoodin generoimisen vaihe, jonka tarkoituksena on erottaa lähdekielen ominaiset piirteet kohdekielen piirteistä.

Erityisen välikielen käyttäminen juontuu kääntäjien kirjoittamisen alkutaipaleelle. Jo varhain huomattiin, että niin korkean tason kieliä kuin erilaisia konearkkitehtuureitakin tulee syntymään suuret määrät. Mikäli haluttaisiin kirjoittaa jokaiselle korkean tason kielelle kääntäjä, joka generoi kohdekoodia kaikille olemassa oleville konearkkitehtuureille, tulisi toteutustyön määrä olemaan noin $m * n$, jossa m on korkean tason kielten määrä ja n kohdearkkitehtuurien lukumäärä.

Tämän vuoksi kehiteltiin ideaa yleiskäyttöisestä välikielestä, UNCOL:ista (engl. *Universal Computer Oriented Language*). Jokainen korkean tason kielen kääntäjä tuottaisi koodia UNCOL:ille, ja jokaiselle konearkkitehtuurille olisi UNCOL:ia lähdekielenään käytettävä kääntäjä [SWT+58]. Tällä järjestelyllä toteutustyön määrä olisi vain $m + n$ kuvan 2.2 näyttämällä tavalla.

Yhtä kaikkien konearkkitehtuurien ja korkean tason lähdekielien välille asettuvaa välikieltä on kuitenkin varsin vaikea, ellei mahdotonta määritellä, sillä niin lähdekielet kuin kohdearkkitehtuurit vaihtelevat suuresti. Suosimalla välikielen määrittelyssä jotain tiettyä lähde- tai kohdekielen ominaisuutta samalla vaikeutetaan jonkin toisen ominaisuuden



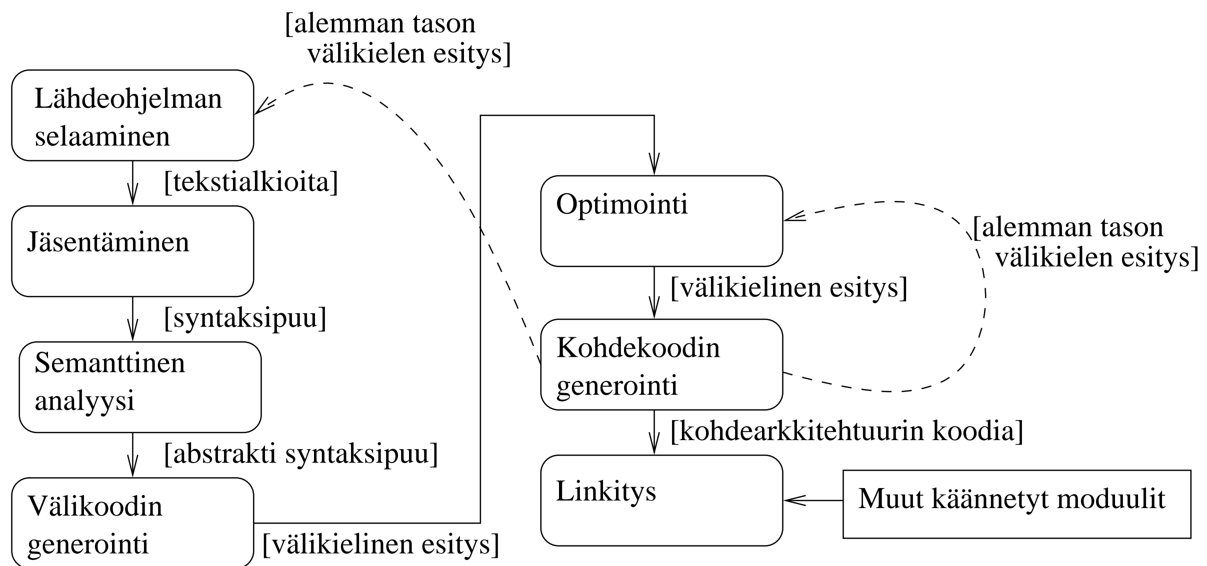
Kuva 2.2: Välikielen käyttämisen vaikutus toteutustyön määrään [App98, s. 156]

toteuttamista. Esimerkiksi määrittelemällä välikieleen kaikki tunnetut muistiviittaustavat tulee siitä samalla vaikeampi toteuttaa sellaisille kohdearkkitehtuureille, jotka eivät luonnostaan tue määriteltyjä viittaustapoja. Toisaalta, pitämällä välikielen määrittely mahdollisimman yksinkertaisena ja siten helposti toteutettavana siirtyy erilaisten lähdekielen rakenteiden kääntämisen monimutkaisuus välikieltä tuottavaan vaiheeseen.

Nykyisin monet kääntäjät käyttävät useampia välikieliä, joista jotkin voivat olla olemassa olevia ohjelmointikieliä. Esimerkiksi ensimmäinen C++ -kääntäjä *CFront* tuottaa välikielenään C:tä, jota puolestaan voidaan olemassa olevien C-kääntäjien ja niiden välikielten kautta kääntää kohdearkkitehtuureille [Str94, s. 66-68]. Näin uudelle, mutta läheisesti olemassa olevaa kieltä muistuttavalle kielelle kääntäjää toteutettaessa tarvitsee toteuttaa vain kielen uudet ominaisuudet.

Välikielten käyttämisellä huomattiin myös pystyttävän vähentämään kääntämisprosessin resurssivaatimuksia. Määrittelemällä välikieli siten, että sille käännetyt ohjelmat voidaan tallentaa välivarastoon, toisin sanoen välikielen määrittely ei ole vain joukko operaatioiden rajapintoja, vaan sillä on myös tekstuaalinen esitys, on koko käännösprosessi mahdollista suorittaa kahdella tai useammalla eri kääntäjällä, joiden kokonaismuistivaatimus yksitellen suoritettuna jää pienemmäksi kuin koko käännösprosessin suorittaminen yhdellä isommalla ohjelmalla [Con58]. Esimerkiksti CFrontin tapauksessa ensimmäinen käännösvaihe sisältää muunnoksen C++:sta C:ksi, joka oletettavasti on myös resurssivaatimuksiltaan pienempi muunnos kuin C++:sta jokaiselle kohdearkkitehtuurille koodia tuottava käännös.

Tämän idean seurauksena voidaan ajatella käännösprosessin muodostuvan joukosta välikieliä, joista kukin ottaa kantaa johonkin lähdekielen ominaisuuteen. Tällöin kääntäjän referenssiarkkitehtuuri muodostuu kuvan 2.3 kaltaiseksi, jossa kunkin optimointivaiheen



Kuva 2.3: Kääntäjän loogiset vaiheet

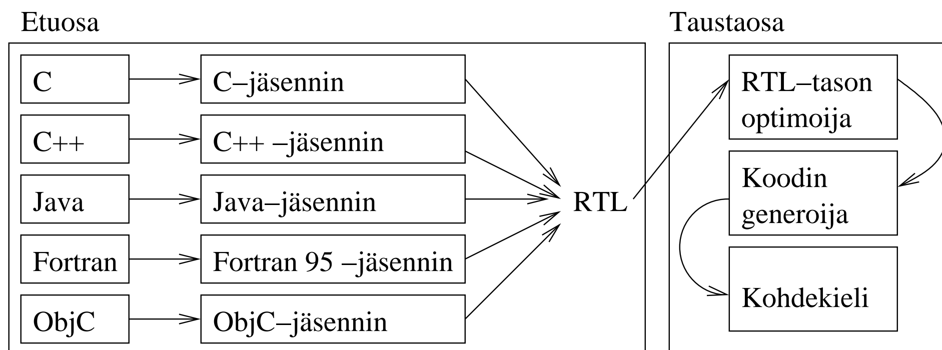
tuloksena tuotetaan saman tai alemman tason välikieltä, jota voidaan operoida joko saman kääntäjän myöhäisemmissä vaiheissa tai kokonaan erillisellä kääntäjällä. Tällöin kuitenkin kääntäjän etuosan tehtävät suoritetaan uudestaan.

Tällainen referenssiarkkitehtuuri tarjoaa kauniin yksinkertaistuksen kääntäjän toiminnalle. Kuitenkin vain harva kääntäjä noudattaa täysin tätä rakennetta optimointien sijainnin suhteen. Esimerkiksi GCC:n [Sta99, Sta04] Java-kääntäjässä vakioihin kohdistuva laskenta suoritetaan jo selausvaiheessa.

Yhden välikielen sijaan useamman välikielen käyttäminen helpottaa ohjelmointityyliltään erilaisten lähdekielien toteuttamista saman kääntäjän osina ja selventää optimointivaiheiden sijaintia käännösprosessin osina, sillä optimoinnin kohdistuminen tiettyyn välikieleen antaa myös osviittaa suoritettavan optimoinnin laajuudesta. GCC[1] käyttää kahta välikieltä: kullekin lähdekielelle ominaista abstraktin syntaksipuun sisältävää sisäistä esitysmuotoa ja tekstuaalisen esitysmuodon omaavaa Register Transfer Language, RTL:ää [Sco00, s. 499-501].

Kuvassa 2.4 näytetään GCC:n sisäiset välikielet. Kunkin lähdekielen jäsentäjä rakentaa kielelle ominaisen syntaksipuun, josta generoidaan kaikille kielille yhteistä välikieltä RTL:ää. Ongelmaksi kuitenkin muodostuu välikielen koneenläheisyys. Kunkin korkean tason lähdekielen kääntäminen koneenläheiselle välikielelle on työmäärältään paljon suurempi kuin mikäli käytettävissä olisi jokin korkeamman tason välikieli. Esimerkiksi Ja-

[1] Tarkemmin sanottuna GCC:n versiota 4.0 edeltävät versiot



Kuva 2.4: Register Transfer Language kaikkien lähdekielten välikielenä [Nov03]

vaa käännettäessä täytyy syntaksipuuta RTL:ksi muunnettaessa ottaa kantaa niin muuttujien sijoitteluun, metodikutsujen toteutukseen kuin synkronointiprimitiivien toteutukseen. Tällöin välikielen tasolla optimoitaessa on ensimmäisenä vaiheena tunnistaa esitysmuodosta lähdekielen rakenteita, joita optimoidaan.

Toisaalta välikielen koneenläheisyys vaikeuttaa myös kielikohtaisia optimointeja. Kun kaikki kääntäjän tekemät optimoinnit toteutetaan matalan tason välikielen tasolla, vaikeutuu lähdekielen rakenteiden tunnistaminen. Esimerkiksi tieto lähdekielen taulukkoviitteistä, tietotyypeistä ja kontrollirakenteista on RTL:än tasolla jo kadonnut. Tämän vuoksi uusissa GCC:n versioissa esitellään korkeamman tason välikieliä helpottamaan lähdekielen kääntämistä [Nov03].

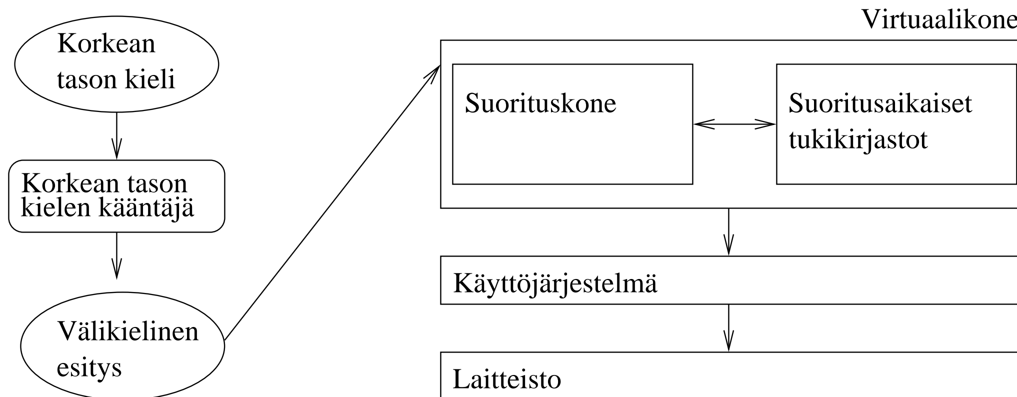
Välikielet siirrettävyyden välineenä

Käytetystä käskykannasta ja konearkkitehtuurista riippumatonta välikieltä voidaan käyttää myös siirrettävyyden välineenä. Vaikka välikieltä ei sellaisenaan pystyisikään suorittamaan (tai kääntämään) halutulla alustalla, on yksinkertaisemmalle kielelle helpompi toteuttaa tulkki (engl. *interpreter*) kuin monimutkaisemmalle kielelle. Eräs Pascal-kielen suosion syistä piilee tavassa, jolla järjestelmää levitettiin. Levityspaketti sisälsi Pascal-kääntäjän, joka tuotti P-koodi- (engl. *P-code*) nimistä välikieltä [Wir71], sekä Pascalilla toteutetun tulkin tälle välikielelle. Lisäksi itse kääntäjä oli sekä välikielisessä että Pascalilla itsellään kirjoitetussa muodossa [Sco00, s. 12-13].

Näin Pascal-kehitysympäristön asentaminen ympäristöön, josta sitä ennestään ei löydy, onnistuu kirjoittamalla olemassa olevilla työkaluilla tulkki välikielelle, P-koodille. Tämä onnistuu vaivattomasti, sillä toisaalta saatavilla on itse tulkin Pascal-kielinen toteutus ja toisaalta itse välikieli on suhteellisen pieni [Sco00, s.13]. Vastaavaa lähestymistapaa käy-

tettiin myös C-kielen edeltäjän BCPL:än välikielessä OCODE:ssa siirrettävyyden parantamiseksi [Ric71].

Käsky kerrallaan tulkkaava suoritus on esimerkki laajemmasta koneiden virtuaalisoinnin ilmiöstä. Suorituksessa oleva ohjelma ei ole tietoinen siitä, miten suorittava kone on toteutettu: onko se puhtaasti laitteistolla toteutettu, vaiko edistynyt suoritusaikaisesti suoritettavaa ohjelmaa optimoiva virtuaalikone. Kuvassa 2.5 esitetään esimerkiksi Javan suoritusympäristön tarjoava virtuaalikone, jota käyttämällä välikielisessä muodossa olevia ohjelmia voidaan suorittaa.



Kuva 2.5: Virtuaalikone suoritusalustana välikielisessä esitysmuodossa olevalle ohjelmalle

Nykyisin aktiivisessa käytössä olevista järjestelmistä Java [LY99] ja Microsoftin aloitteesta ISO-standardina määritelty Common Language Infrastructure [ISO03] käyttävät siirrettävyyttä edistävää välikieltä. Nämä järjestelmät kuitenkin eroavat alkuperäisen yleiskäyttöisen välikielen ideasta siinä, että kumpikin yrittää yleistää vain toisen puolen kääntäjän etu- ja takaosista. Javan välikielen tarkoituksena on taata laitteistojen välinen siirrettävyys, eli varsinainen tavoite on muunnettavuus välikielestä konearkkitehtuuriin päin [Gos95]. Tämä näkyy siinä, että välikielen tasolla esitellyt operaatiot ja tietotyypit on määritelty varsin tarkasti siten, että toteutuskohtaiselle tulkinnanvaraisuudelle ei ole jätetty paljoakaan sijaa.

Common Language Infrastructure (CLI) puolestaan tarjoaa yhteisen kohteen kaikille lähdekielille. Se määrittelee kaikille lähdekielille yhteisen välikielen, Common Intermediate Languagen (CIL), yhteisen suoritusympäristön Common Language Runtimen (CLR) ja kaikille kielille yhteisen tyyppijärjestelmän, Common Type Systemin (CTS) [Ham03]. CLI:ssä näkyy kuitenkin yhteisen välikielen kehittämisen vaikeus. Esimerkiksi määritelty tyyppijärjestelmä tukee vain toteutuksen yksiperintää mutta rajapintojen moniperintää.

Tällöin toteutuksen moniperiminen C++:an tai Eiffelin [Mey92] tyyliin ei onnistu välikielen tarjoamien ominaisuuksien puitteissa, vaan näiden kielten kääntäjää toteutettaessa joudutaan toteutuksen moniperintä toteuttamaan alustan tarjoamien ominaisuuksien ohitse [Gou02, s. 349-352].

Oliosuuntautuneet välikielet ja Javan tavukoodi

Stroustrup [Str88] ajattelee oliokielen olevan oliosuuntautunut, jos kyseisellä kielellä ohjelmoidessa oliomaisten ohjelmien tekeminen on helppoa. Tästä analogiana voi ajatella kääntäjän välikielen olevan oliosuuntautunut, jos vielä tämän kyseisen välikielen tasolla tuetaan oliokielten ominaisuuksia. Toisin sanoen, mikäli välikielen tasolla ei enää ole nähtävissä tukea olio-ominaisuuksille, on kyseessä jo alemman tason välikieli. Näin esimerkiksi C-kieli ei ole oliokielen välikieli, vaikkakin sitä oliokielenäkin toimivan C++:n välikielenä on käytetty.

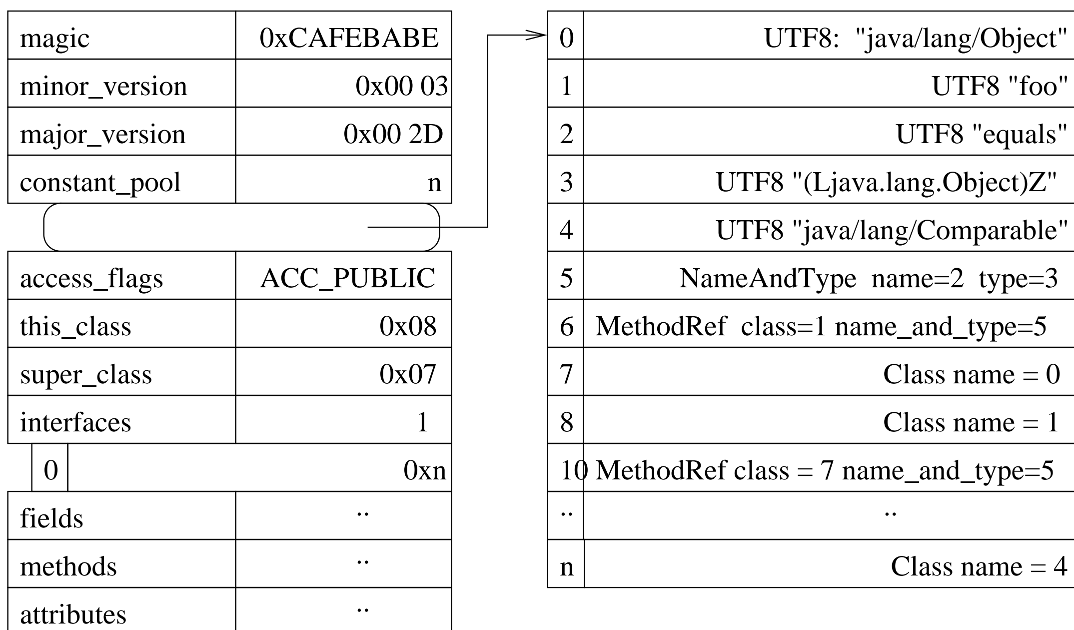
Tällä perusteella esimerkiksi Javan tavukoodi, CIL ja Smalltalkin tavukoodi [GR83, s. 594-610] ovat oliokielten välikieliä, sillä jokainen niistä tukee kapselointia, perintää ja myöhäistä sidontaa. Seuraavassa esitellään Javan tavukoodin tapa toteuttaa nämä olio-ominaisuudet. Koska CLR on olio-ominaisuuksiltaan varsin lähellä Javan virtuaalikonetta [Sin03], pätee seuraavassa esitettävä suurelta osin myös sen suhteen.

Javan välikieli on oliosuuntautunut välikieli, koska olio-ominaisuudet kapselointi, perintä ja myöhäinen sidonta ovat käsitteinä vielä tämän välikielen tasolla näkyvissä.

Javan tavukoodissa ohjelma jaetaan luokkiin. Luokkatiedoston rakenne on määritelty tavukoodin suorituksesta vastaavan virtuaalikoneen yhteydessä [LY99, §4.1]. Kuvassa 2.6 näytetään luokkatiedon rakenne julkiselle luokalle *foo*, joka perii luokan *java.lang.Object* ja toteuttaa yhden rajapinnan, *java.lang.Comparable*. Selvyyden vuoksi kuvasta jätetään luokan kentät, metodit ja attribuutit näyttämättä.

Luokkamääritys alkaa tunnustiedolla, heksadesimaalina esitettynä *0xCAFEBABE*. Tätä seuraa käytettyä Javan versiota esittävät versionumerot. Kentässä *constant_pool* ilmoitetaan luokan sisäisen symbolitaulun kenttien lukumäärä, esimerkissä n. Kenttä *access_flags* kertoo luokan olevan julkinen.

Kentät *this_class* ja *super_class* kertovat määriteltävän luokan ja sen kantaluokan. Arvot ovat viitteitä luokan symbolitauluun. Esimerkissä siis määritellään luokka, jonka nimi sijaitsee symbolitaulun indeksissä nolla (*foo*) ja se perii luokan, jonka nimi sijaitsee symbolitaulun indeksissä yksi (*java.lang.Object*). Luokkien symbolitaulut ovat siten tyypitettyjä: kentissä, joiden sisältönä on luokka, tulee osoittaa symbolitaulussa luokka-arvon



Kuva 2.6: Luokan *foo* rakenne Javan luokkatiedostona

sisältävään indeksiin.

Kenttä *interfaces* kertoo määriteltävän luokan toteuttamien rajapintojen lukumäärän. Sitä seuraa lista viitteitä kunkin toteutetun rajapinnan indeksiin symbolitaulussa. Lopuksi symbolitaulun sisällöstä voidaan päätellä, että määriteltävä luokka *foo* syrjäyttää kantaluokasta perityn metodin *boolean equals(java.lang.Object)*.

Operaatiokutsujen myöhäinen sidonta toteutetaan välikielen *invokevirtual* -komennolla. Välikielen tasolla kohdeluokan metodi annetaan viitteenä luokan symbolitauluun. Näin esimerkiksi välikielen symbolisen konekielen komento

```
invokevirtual java/lang/Object/equals(Ljava.lang.Object)Z
```

lukee suorituspinon huipulta *java.lang.Object*-luokan tai sen aliluokan ilmentymän ja kutsuu sen metodia *equals(java.lang.Object)*. Metodikutsun boolean-tyyppinen paluuarvo kirjoitetaan pinon huipulle. Suoritusympäristön tehtävänä on selvittää komennon ensimmäisenä argumenttina olleen olion konkreettinen tyyppi ja sen mukainen *equals*-metodin toteutus, jota kutsutaan. Kuvan 2.6 tapauksessa ylläoleva kutsu esitetään tavuina lyhyesti *0xB6 0x00 0x0A*, jossa ensimmäinen tavu (*0xB6*) vastaa suoritettavaa käskyä *invokevirtual*, ja kaksi seuraavaa tavua (*0x00 0x0A*) antavat symbolitaulun indeksin.

Oliosuuntautuneen välikielen optimoinnin kannalta voidaan erotella kaksi erillistä tapausta: ko. välikieltä lähde- ja kohdekielenään käyttävät uudelleenjärjestelevät optimoinnit se-

```
int *ip;

void interpreter() {
    while(ip) {
        ( (void (*)()) *ip++ )();
    }
}
```

Kuva 2.7: Funktio-osottimia käyttävä tulkkiajuri

kä alemman tason kieleen kääntämisen yhteydessä tapahtuvat optimoinnit. Ensimmäiseen ryhmään kuuluu tämän tutkielman luvussa 3.7 esiteltävä luokkarakennetta ja sitä kautta metodikutsukäskyjä uudelleenjärjestelevä optimointi. Jälkimmäisen ryhmän optimoinnit taas tulevat kyseeseen Javan kaltaisen dynaamisen kielen suoritusympäristön dynaamista kääntäjää tarkastellessa.

Dynaaminen kääntäminen

Jotta välikielellä esitettyä ohjelmaa voitaisiin suorittaa, tarvitaan sille kääntäjä, joka muuntaa ohjelman kulloisenkin suoritusalustan ymmärtämään muotoon. Vaihtoehtoinen tapa on käyttää tulkkia, joka on suoritusalustalle ominaisessa muodossa. Tällöin kunkin välikielen käskyn merkitys on kirjoitettu tulkkiin. Ohjelmaa suoritetaan käsky kerrallaan, jolloin yksinkertaisin tulkin toteutus toimii siten, että kukin välikielen käskyn toteutus sijaitsee funktio-osoittimen päässä ja kunkin kutsun jälkeen kasvatetaan käskyosoitinta yhdellä. Tällöin tulkkiajuri voidaan kirjoittaa vaikkapa C:llä kuvan 2.7 osoittamalla tavalla.

Tällä tulkin toteutuksella kunkin välikielen käskyn suorittaminen aiheuttaa funktiokutsun ja -paluun. Lisäksi tulkin tilan tallentamiseen (käskyosoitin yms.) joudutaan käyttämään toteutuskielen globaaleja muuttujia, jotka useimmat kääntäjät sijoittavat keskusmuistiin eivätkä näin osaa hyödyntää suorituskoneen rekistereitä kovinkaan hyvin.

Tulkkauksen tuottaman lisärasitteen vähentämiseksi on kehitelty erilaisia menetelmiä. Edellisen version funktiokutsujen tuottama lisärasite voidaan kiertää vaihtamalla funktiokutsut suoriksi hyppäyksiksi (engl. *threaded code*) [Bel73]. Tällöin kunkin välikielen käskyn toteutuksessa ei enää olla funktiokutsun tarjoamassa hiekkalaatikossa, vaan esimerkiksi rekisterienvarauskäytännön täytyy olla eri käskyjen toteutuksissa yhtenäinen. Tällaisen tulkkiajurin siirrettävä toteuttaminen on myös ongelmallista, eikä onnistu esi-

merkiksi ANSI-C:llä [Ert95].

Tulkkiajurin tuottaman tehohäviön vähentämiseksi suunniteltujen käskynvalitsemismekanismien lisäksi ongelmaa voi ratkoa muillakin tavoin. C- kieltä tulkattaessa hyväksi lähestymistavaksi on osoittautunut yksittäisten, granulariteetiltaan pienten välikielten käskysekvenssien yhdistäminen ns. superkäskyiksi [Pro95]. Kielen käskykanta kasvaa, mutta kukin yksittäinen käsky tekee 'enemmän' asioita, jolloin seuraavan käskyn valitsemiseen käytetyn koodin suorituksen frekvenssi laskee. Kokonaan tulkkauksen lisärasitteesta ei kuitenkaan tälläkään menetelmällä päästä eroon. Vasta dynaaminen kääntäminen mahdollistaa tulkkaussyklin aiheuttaman kuorman poistamisen.

Dynaamisella kääntämisellä tarkoitetaan välikielen suoritusaikaista kääntämistä suoritusalustan natiiville kielelle. Tekniikka tuli oliokielten yhteydessä tunnetuksi Smalltalk-80 -kielen nopeutuksena [DS84], mutta periaatteena se oli tunnettu jo aikaisemmin. Idean taustalla on huomio, jonka mukaan tulkatulla välikielellä esitetyn ohjelman muistinkulutus on usein pienempi kuin vastaavan ohjelman konearkkitehtuurin luonnollisella (natiivilla) kielellä esitetyn [Rau78]. Tämä johtuu siitä, että alhaisemman abstraktiotason kielellä saman toiminnallisuuden esittämiseen käytetään useampia käskyjä verrattuna yhden korkeamman tason kielen käskyyn.

Tulkkaussyklin tuottaman lisäkuorman takia tilankulutukseltaan pienemmän käskyn suorittaminen vie kuitenkin enemmän aikaa kuin vastaavan toiminnan suorittaminen natiivin käskykannan koodina. Rajoitetun muistitilan koneissa dynaamisella kääntämisellä voidaan hakea kompromissi suoritusnopeuden ja tilankulutuksen välillä. Esimerkiksi sisimmät silmukat voitaisiin suoritusaikaisesti kääntää natiiville käskykannalle, kun taas harvemmin suoritettavassa koodissa kärsitään suorituskykytappio, mutta saavutetaan pienempi tilankulutus.

Dynaamisen kääntämisen hyödyt eivät kuitenkaan rajoitu korkeamman abstraktiotason kielen kääntämiseen matalammalle tasolle. Sen lisäksi tekniikkaa käytetään myös tehokkaiden saman abstraktiotason kielten simulaattorien toteuttamiseen. Esimerkiksi *alpha*-konearkkitehtuurilla suoritettavan Windows NT:n kyky suorittaa *x86*-arkkitehtuurille käännettyjä binäärejä perustuu dynaamiseen kääntämiseen [HH97]. Dynaamisesta kääntämisestä voi olla hyötyä myös samaa lähde- ja kohdekieltä käytettäessä. Konearkkitehtuurien sisäiset ominaisuudet muuttuvat kunkin prosessorisukupolven myötä, sillä seuraavassa sukupolvessa poistetaan usein edellisen sukupolven toteutuksissa ilmenneitä pullonkauloja. Toisaalta, suoritusaikaisesti kääntäjä voi hyödyntää suoritusprofiilia ja järjestellä koodipolkuja välimuistien kannalta tehokkaammiksi kuin puhtaasti staattinen kääntäjä pystyy [BDB00].

Olio-ohjelmoinnissa dynaamista kääntämistä hyödynnetään kielten tarjoamien dynaamisen ominaisuuksien tehokkaaseen toteutukseen. Perusesimerkkinä on luokka, jolla ei ole ladattuna aliluokkia. Kun ohjelmakoodia suoritettaessa tulee vastaan kutsu kyseisen luokan olioon, voitaisiin kutsun kohdemetodi sitoa jo staattisesti. Tavallisesti joudutaan kuitenkin varautumaan tilanteeseen, jossa kyseiselle luokalle ilmestyy suoritusaikaisen luokkien lataamisen kautta aliluokka, joka syrjäyttää kantaluokkansa kohdemetodin. Tämän vuoksi ei staattinen, käännösaikainen sidonta ole yleensä mahdollista. Dynaamisella kääntämisellä voidaan saavuttaa staattisen sidonnan tuottama tehokkuushyöty, sillä mikäli optimoinnin validiutta rikkova aliluokka jossain suorituksen vaiheessa ilmaantuu, voidaan käännetty kutsukoodi hylätä ja palata optimoimattomaan versioon. Tarkemmin dynaamisen kääntämisen ja uudelleenkääntämisen ongelmakenttää käsitellään erityisesti Hölzlen väitöskirjassa [Höl94] sekä muissa Self-projektiin [US87] liittyvissä julkaisuissa [HCU91, HCU92, HU94].

2.3 Optimointi

Matemaattisessa mielessä optimointi tarkoittaa parhaimman mahdollisen muodon löytämistä. Täsmällisemmin matemaattisen optimointiongelman voidaan määritellä olevan pari (F, c), missä F on kaikkien mahdollisten muotojen joukko, c on kustannusfunktio ja kuvaus

$$c : F \rightarrow \mathcal{R} \tag{2.2}$$

Ongelmana on löytää $f \in F$, jolle $c(f) \leq c(y)$ kaikille $y \in F$. Tällaista muotoa f kutsutaan optimaaliseksi ratkaisuksi ongelmaan [PS82, s.4].

Kustannusfunktio tarkoittaa kuvausta, jolla arvotetaan kunkin mahdollisen muodon hyvyyttä. Taulukossa 2.1 esitetään muutamia mielekkäitä mittareita ohjelmien optimoinnin kohteiksi. Näistä ohjelman staattisen koon selvittäminen on varsin suoraviivainen toimenpide, mutta muut kustannusfunktiot ovat dynaamisia siinä mielessä, että niille saadaan arvo vasta, kun ohjelman suoritus annetulla syötteellä on päättynyt.

Kustannusfunktio	Selitys
cSize	Ohjelman staattinen koko
cSpeed	Ohjelman suoritusnopeus
cMemConsumption	Ohjelman muistinkulutus
cEnergy	Ohjelman energiankulutus

Taulukko 2.1: Mahdollisia kustannusfunktioita

Ricen esittämän kääntäjäkirjoittajan työllistymislauseen (engl. *full employment theorem for compiler writers*) mukaan ohjelman optimaalisen muodon f aikaansaaminen on yleisessä tapauksessa mahdotonta, sillä ongelma on laskennallisesti Turingin koneen pysähtymisongelmaa vastaava [Ric53].

Appel esittää seuraavalla tavalla koon suhteen täysin optimoivan kääntäjän mahdottomaksi. Oletetaan, että meillä olisi täysin optimoiva kääntäjä, jolle annetaan syötteenä laskentaprosessin kuvaus (syöteohjelma). Optimoiva kääntäjä tuottaa pienimmän tavan toteuttaa sama laskenta, eli syöteohjelma P käännetään muotoon Opt(P) siten, että samalla syötteellä P:n ja Opt(P):n tulos on aina sama.

Oletetaan sitten ohjelma Q, joka ei tuota mitään tulostetta, mutta ei myöskään pysähdy. Opt(Q) olisi nyt pseudokoodilla kirjoitettuna kuvan 2.8 mukainen.

```
1: Goto 1.
```

Kuva 2.8: Koon suhteen optimoiduin ohjelma

Täysin optimoiva kääntäjä osaisi siis kääntää kaikki ohjelmat, jotka eivät pysähdy mutta eivät myöskään tulosta mitään, yllä mainittuun muotoon. Toisin sanoen, täysin optimoivaa kääntäjää voitaisiin käyttää Turingin koneiden pysähtymisongelman ratkaisemiseen [App98, s. 387-388].

Nopeuden suhteen sama asia voidaan esittää ajattelemalla, että jokainen optimoitavan ohjelman käsky vie yhden aikayksikön. Kuvan 2.9 ohjelman suorittaminen siis kestää yhden, kaksi tai n aikayksikköä riippuen siitä, montako laskenta-askelta aliohjelmakutsun suorittaminen vaatii. Jos yleisessä tapauksessa voitaisiin näyttää, että aliohjelma ei tulosta mitään ja pysähtyy äärellisen laskenta-askelten määrän jälkeen, selvittäisiin ohjelman suorituksesta yhdellä, kuvan rivin 2 print-lauseella. Pysähtymisongelmaan vedoten kuitenkin tiedetään, että aliohjelman tuottamaa tulostetta ei voida yleisessä tapauksessa ennakoida.

```
1: Call foo()
2: Print "hello"
```

Kuva 2.9: Nopeuden suhteen optimoitava ohjelma

Koska tiedetään, että Turingin koneiden pysähtymisongelmaa ei saada ratkaistuksi laskennallisella algoritmilla, ei myöskään täysin optimoiva kääntäjä ole toteutettavissa. Sen sijaan optimoivasta kääntäjästä puhuttaessa tarkoitetaan kääntäjää, johon on toteutettu sään-

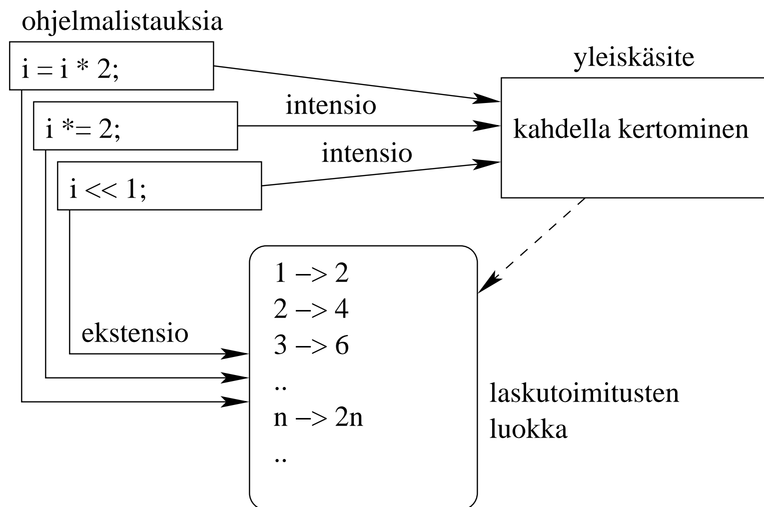
töjä ja kikkoja, joiden toivotaan vaikuttavan tuotettavan kohdeohjelman suorituskykyyn (tai muuhun resurssien käyttöön) positiivisesti.

Optimointien oikeellisuus

Yleisesti ajatellaan, että suoritettavien optimointien tulisi säilyttää ohjelman toiminta ja merkitys ennallaan. Toisin sanoen syötettäessä tietyllä tavalla toimiva ohjelma optimoijalle tulisi tuloksenkin olla samalla tavoin toimiva. Tämä herättää kuitenkin kysymyksen: minkä suhteen tulisi ohjelman toimia samoin?

Edellisessä kappaleessa ajatellaan optimoinnin olevan oikeellinen, mikäli laskenta tuottaa samalla syötteellä saman tuloksen. Näin ei kuitenkaan kaikissa tapauksissa ole — jos vaikkapa ajatellaan tosiaikajärjestelmää, joissa laskentaan kulutettu aika on yhtä lailla oleellinen tekijä, ei voida rajoittua syötteen ja tulosteen väliseen vastaavuuteen, vaan suoritusajan suhteen optimoitaessa tulisi jotenkin pystyä takaamaan vaaditut aikarajoitteet.

Ohjelman erään ilmaisumuodon, listauksen ja sen tarkoituksen välistä yhteyttä voidaan tarkastella ohjelman ekstension ja intension kautta. Niiniluoto esittää termin, tässä yhteydessä siis ohjelmalistauksen *ekstension* olevan se abstrakti olio, jonka esitys kyseinen ohjelma on; *intensio* taas on se ominaisuus tai yleiskäsite, jonka kautta ekstensio määritellään [Nii80, s. 119-122]. Tätä ajatusta selvennetään kuvassa 2.10, jossa esitetään muutamia syntaksiltaan erilaisia ohjelmarakenteita, joilla kaikilla kuitenkin on sopivat lähtöolettamukset hyväksyen sama intensio. Siten myös niiden ekstensiot ovat samat.



Kuva 2.10: Ohjelmalistauksen intensio ja ekstensio

Intensioiden ja ekstensioiden välistä suhdetta ohjelman optimointiin sovellettaessa juuri

vallitsevat lähtöolettamukset ovat keskeisessä asemassa. Lauseet *i *= 2* ja *i « 1* ovat ekstensioiltaan samat, mikäli lähtöolettamuksiin kuuluvat esimerkiksi äärettömän tarkkuuden luvut tai yhteneväinen ylivuodon hallinta. Kuitenkin, mikäli kertolasku on määritelty äärettömän tarkkuuden luvuilla toimivaksi ja bittikierto vakiokokoisilla bittikentillä toimivaksi, ovat operaatioiden ekstensiot samat vain niiltä osin kuin laskutoimituksessa ei tapahdu ylivuotoa.

Intensio on siis jotain, mikä määrää ohjelman ekstension erilaisissa yhteyksissä, joita voidaan esittää erityisen *indeksin* avulla. Indeksi sisältää kyseisen käyttöyhteyden kuvauksen. Näin intensio olisi siis kuvaus indeksien joukolta ekstensioiden joukolle [Nii80, s. 121]. Kappaleen alun tosiaikajärjestelmää kuvattaessa indeksiin siis kuuluisi tosiaikaisuuden asettamat vaatimukset suoritettaville optimoinneille — vain ne muunnokset, jotka näin tuottavat saman ekstension, ovat siten oikeellisia.

Optimoinnin kohteet ja tehokkuuden mittaaminen

Pääsääntöisesti tärkein optimoitava suure on ohjelman suorituksen kuluttama aika. Ohjelmia voidaan kuitenkin optimoida myös muiden rajallisten resurssien suhteen: ohjelman tarvitsema muisti tai ohjelman staattinen koko voivat joissakin tilanteissa olla suoritusnopeutta oleellisempia tekijöitä. Esimerkiksi jos ohjelma ennen suoritusta siirretään hitaan verkon yli kohdekoneeseen, voi ohjelman staattinen koko olla suoritusnopeutta tärkeämpi tekijä; jos taas kohdekone toimii akkujen varassa, voi energiankulutus olla suoritusnopeutta tärkeämpi tekijä.

Monet optimoinnit parantavat ohjelman suorituskykyä valitun resurssin suhteen jonkin toisen resurssin kustannuksella. Esimerkiksi muuttujan *size* arvon ollessa parillinen voidaan kuvassa 2.11 esitettävän silmukan läpikäyntiin vaadittujen hyppyjen määrä puolittaa avaamalla silmukka avauskertoimella kaksi.

```
for(int i=0; i<size; i++) {
    System.out.println(i);
}
```

Kuva 2.11: Avaamaton silmukka

Näin syntyvässä silmukassa kullakin kierroksella kasvatetaan indeksimuuttujaa avauskertoimella ja kopioidaan silmukan sisältö kunkin avauskierroksen osalta silmukan sisälle. Näin avattu silmukka kirjoitettaisiin kuvassa 2.12 esitettävällä tavalla.

```
for(int i=0; i<size; i+=2) {
   System.out.println(i);
   System.out.println(i+1);
}
```

Kuva 2.12: Kerran avattu silmukka

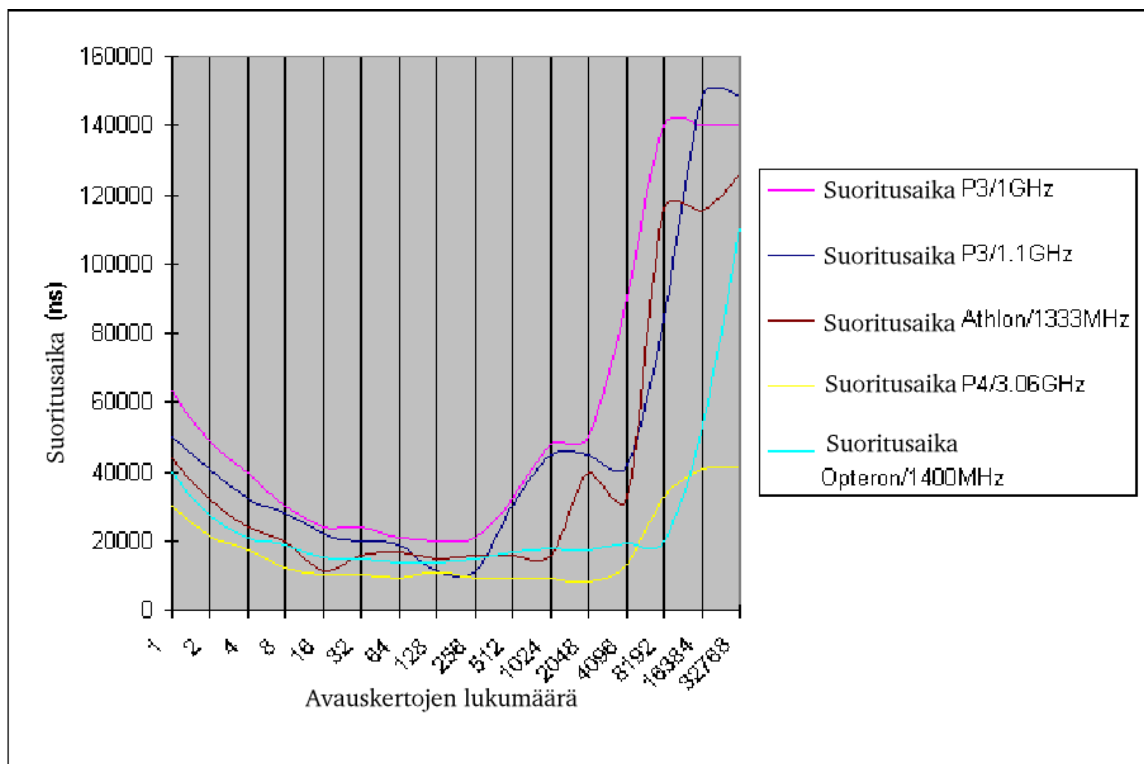
Esimerkissä koodin suoritusnopeutta yritetään parantaa staattisen koon ja sitä kautta myös muistinkulutuksen kustannuksella. Käännetyn ohjelmakoodin koolla ei vuosikymmeniin massamedioiden koon kasvun myötä ole ollut merkittävää vaikutusta ohjelmistojen suoritustehokkuuteen, joten silmukoiden avaaminen (engl. *loop unrolling*) on tapa vähentää niiden tuottamaa yleisrasitetta. Oheisvaikutuksena avaaminen saattaa parantaa muiden optimointien tehokkuutta [Muc97, s. 559– 562].

Kuitenkin, vaikka esimerkin muuttujan *size* arvo tunnettaisiin käännösaikaisesti, ei silmukkaa kannata välttämättä kokonaisuudessaan avata. Välimuistien ansiosta ohjelman suoritusajan ja staattisen koon välillä on yhteys, sillä nykyisissä konearkkitehtuureissa välimuistihudit (engl. *cache miss*) ovat suoritusajan suhteen varsin kalliita; siinä missä kaksikymmentä vuotta sitten tiedon hakeminen keskusmuistista prosessorin rekisteriin vei neljä prosessorisykliä, saattaa samaan asiaan nykyisissä arkkitehtuureissa kulua kymmenkertainen määrä [AK02, s. 21].

Asiaa voi kokeilla yksinkertaisella kokeella: tuotetaan yksinkertainen silmukka, jota on mahdollista avata esimerkiksi kahden potensseissa. Suoritetaan silmukan eri versioita ja mitataan suoritusajat. Oletettavasti aluksi suoritusajat laskevat, sillä suorituksen aikana sekä silmukan alkuun tapahtuvia hyppäyksiä että silmukan jatkoehdon testauskertoja suoritetaan vähemmän. Silmukan staattisen koon kasvaessa kuitenkin tarpeeksi suureksi suoritusajoissa tapahtunee kertaluokan hyppäys ylöspäin välimuistihutien astuessa mukaan kuvioon.

Koetta varten tuotettiin silmukka, jonka sisin osa konekielen tasolla on 87 tavua pitkä, sisältäen aritmeettisia lausekkeita. Silmukkaa avattiin 2^n kertaa, $n \in 0..15$, ja tulossilmukat suoritettiin erilaisilla $x86$-arkkitehtuurin prosessoreilla.

Kuvassa 2.13 esitetään, kuinka silmukan avaamiskierrosten lukumäärä vaikuttaa kokonaissuoritusaikaan. Nykyisissä *x86*-pohjaisissa prosessoreissa on useimmiten kahden tasoista välimuistia: L1- ja L2-välimuistit. Silmukan avauskerrointa kasvatettaessa suoritus nopeutuu niin pitkään kuin avattu versio mahtuu välimuistiin. Kuvan suoritusaikakäyristä voidaan melko helposti nähdä, että esimerkiksi *1,0GHz Intel Pentium III* -prosessorin L1-



Kuva 2.13: Silmukan avauksen vaikutus suoritusaikaan eri prosessoreilla

tason välimuisti täyttyi 256 avauskierroksen kohdalla ja L2-tason välimuisti puolestaan 2048 avauskierroksen kohdalla.

Prosessorivalmistajan määrittelyistä tarkistamalla nähdään, että käytetyssä prosessorissa on 32 kilotavua L1-välimuistia ja 256 kilotavua L2-välimuistia. Vastaavasti 256 avauskierroksella silmukan staattinen koko on $87 * 256 = 21$ kilotavua ja 2048 avauskierroksella $87 * 2048 = 174$ kilotavua. Useimmilla prosessoreilla välimuistin täyttyessä silmukan avaamisen seurauksena tapahtuu suoritusajassa raju hyppäys ylöspäin. Uudemmilla prosessoreilla, kuten kuvan P4:llä, hyppäys ei ole kuitenkaan aivan yhtä voimakas, johtuen mm. parantuneesta hyppykohteen ennustusmenetelmistä (engl. *branch prediction*) ja spekuloivasta suoritustavasta (engl. *out-of-order execution*).

Kokeesta nähdään, että yksinkertaisenkin optimoinnin tuloksiin vaikuttaa varsin suuri joukko eri tekijöitä: suoritusalustan ominaisuudet, optimointialgoritmille annettavat parametrit, optimointialgoritmin soveltuvuus optimoitavaan sovellukseen ja optimointialgoritmien keskinäinen vaikutus.

Kokonaisuudessaan ohjelmien optimointimenetelmät muodostavat niin monimutkaisen kokonaisuuden, että puhtaasti analyyttisella ajatuskehyksellä on varsin vaikea tuottaa hy-

viä tuloksia. Sen sijaan optimointitutkimuksessa tutkitaan erilaisia menetelmiä ja kikkoja, joilla saadaan käsiteltävää ohjelmakoodia paranneltua sekä näiden temppujen ja menetelmien yhteisvaikutusta erilaisissa konteksteissa.

3 Ohjelman analysointi ja parantelu

Ohjelmia optimoitaessa kääntäjä ylläpitää käännettävän ohjelman rakennetta erilaisissa tietorakenteissa. Kirjallisuudessa esitetään erilaisia matemaattisia konstruktioita ja ohjelmallisia rakenteita, joihin perustuen optimointeja voidaan suorittaa. Tässä luvussa esitetään muutamia perusrakenteita, joita käsittelemällä ohjelmia voidaan optimoida. Lisäksi esitetään yleisellä tasolla muutamia optimointimenetelmiä.

3.1 Analysoijan maailmankuva

Olio-ohjelmia analysoitaessa ja paranneltaessa törmätään ns. maailmankuvan ongelmaan. Oliokielten tarjoamat mahdollisuudet ohjelman inkrementaaliseen kehitykseen lisäävät aika-akselin paranteluiden oikeellisuuden tarkasteluun. Aiemmin luvussa 2.1 mainittu perinnän inkrementtioperaattori $\oplus$ saattaa kielestä riippuen mahdollistaa uusien rakenteiden esittelemisen jopa suoritusaikaisesti - ja analyysimenetelmien tulee ottaa tämä huomioon.

Ajatellaan käytännön esimerkkinä luokkahierarkian analysointia. Analyysissä muodostetaan esitys ohjelman sisältävistä luokista ja perintähierarkiasta. Kun jokin ohjelman osista lataa uuden luokan vaikkapa verkon yli sijainnista, joka ei ollut alun perin analyysin saatavilla, vanhentuu kyseisen analyysin tulos, ja samoin kaikki siihen perustuneet muunnokset ovat tämän jälkeen mahdollisesti virheellisiä, sillä ne perustuvat vanhentuneeseen tietoon.

Palsberg esittää, että ohjelmaa käännettäessä on tunnettava implisiittisesti kehitysvaiheen maailmankuva (engl. *world assumption*). Suljetussa maailmankuvassa (engl. *closed-world assumption*) kaikki analyysille tarpeellinen tieto on saatavilla ja muuttumatonta, kun taas avoimessa maailmankuvassa (engl. *open-world assumption*) analysoitava tieto saattaa täydentyä ja muuttua [PS94, s. 13]. Joissakin, erityisesti uudelleenkäytettäviksi tarkoitetuissa ohjelmistokehyksissä avoin maailmankuva on ainoa mahdollinen, sillä läheskään koko ohjelmaa ei vielä toimitusvaiheessakaan tunneta.

Olio-ohjelmien yhteydessä puhutaan tuntemattoman aliluokan ongelmasta [KV92]: laajennettaessa ohjelmaa periyttämällä uusia luokkia kantaluokista tarvitaan mekanismi periytettyjen luokkien liittämiseksi järjestelmään. Ongelmaksi muodostuu se, että olemassa oleviin luokkiin ei välttämättä voida tehdä mitään muutoksia, jolloin tuntemattomien luokkien liittäminen ei ole täysin suoraviivaista. Ongelma ratkeaa erityisellä liittymiskoodilla (engl. *entrance code*), jota käyttämällä uudet luokat liitetään järjestelmään.

Eräiden tutkijoiden mukaan suljetun maailman oletus voidaan tehdä vain kielissä, joissa

dynaaminen luokkien lataaminen on kokonaan kielletty [IKY+00]. Tämä ei kuitenkaan aivan pidä paikkaansa. Vastaesimerkkinä toimii edellä mainittu dynaaminen luokkien lataaminen siten, että ladattaessa ei ole mahdollista rikkoa analyysin tuottamia faktoja ohjelman rakenteesta. Toisaalta, koska sopivalla liittymiskoodilla voidaan ohjelmaa muokata ja laajentaa myös ilman dynaamista linkittämistä, ei luokkien suoritusaikaisen lataamisen kieltäminen kuitenkaan ole tae suljetusta maailmasta. Tämän vuoksi kääntäjätoteutukset sisältävät käännösaikaisia optioita, joilla voidaan kontrolloida suoritettavia analyyseja ja niihin perustuvia optimointeja.

Jos tiedetään, että ohjelma ei käytä analyysimenetelmän ulottumattomissa olevaa ohjelmakoodia, voidaan muunnoksia tehdä suljetun maailmankuvan mukaisesti. Esimerkki tilanteesta on kännykkäsovelluksiin tarkoitettu Javan miniversio Connected, Limited Device Configuration (CLDC), jossa tietoturvaperustein on ohjelmoijan määrittelemien luokkalataajien käyttäminen kielletty [RTV01, s. 51]. Tämän seurauksena kaikki tälle Javan versiolle suunnatun ohjelman luokat löytyvät samasta levityspaketista, jolloin oletus suljetusta maailmasta voidaan turvallisesti tehdä. Joissakin kääntäjissä, kuten Manta- nimisessä Java-kääntäjässä, suljetun maailmankuvan valitsemiseen on komentoriviltä toimiva vipu [VKB01].

Mikäli suljetun maailman oletusta ei voida turvallisesti tehdä, joudutaan turvautumaan muihin keinoihin. Dynaamista kääntämistä hyödyntävässä järjestelmässä voidaan käyttää ns. adaptiivista kääntämistä: kullekin suoritetulle optimoinnille talletetaan jollain tavalla esiehdot, joiden rikkoutuessa joudutaan palaamaan optimoimattomaan tilanteeseen. Alun perin Self-kielen toteutusryhmässä [HCU91] syntynyt idea on sittemmin otettu käyttöön myös uudempiin dynaamista kääntämistä hyödyntäviin järjestelmiin.

3.2 Kontrollivuokaavio ja peruslohkot

Ohjelman kontrollivuokaaviossa (engl. *control flow graph*) esitetään ohjelman kontrollin mahdolliset kulkureitit ohjelmakoodissa. Kontrollivuokaavio voidaan muodostaa sijoittamalla vuota esittävään verkkoon kukin ohjelmakoodin käsky omaksi solmukseen, ja yhdistämällä käskyä mahdollisesti seuraavien käskyjen solmut suunnatuilla särmillä.

Allen [All70] määrittelee kontrollivuokaavion suunnatuksi verkoksi

$$G = (B, E) \tag{3.1}$$

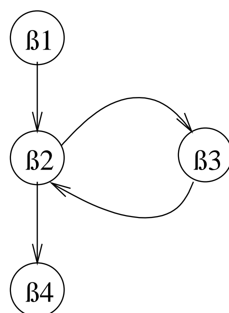
jossa B on peruslohkojen (engl. *basic block*) joukko $(\beta_1, \beta_2, ..., \beta_n)$ ja E puolestaan joukko suunnattuja särmiä $\Big((\beta_i, \beta_j), (\beta_k, \beta_l), ...\Big)$. Myöhemmin on huomattu, että kontrollin

tarkasteluissa on lisäksi hyödyllistä tuntea suoritusjärjestyksen aloittava solmu β_0, kuten esimerkiksi Kennedyn määritelmässä [Ken81]. Peruslohko puolestaan on järjestetty jono käskyjä, jotka suoritetaan aina peräkkäin. Toisin sanoen, kontrolli siirtyy aina peruslohkon alkuun ja hyppy- tai ohjelman lopetuskäskyt ovat aina peruslohkon viimeisinä käskyinä [ASU86, s. 528]. Kaaviossa kontrollin mahdollista kulkua kuvaavassa särmässä (β_i, β_j) ensimmäinen jäsen (β_i) tarkoittaa siten sitä peruslohkoa, jonka loppuun saavuttaessa kontrolli mahdollisesti siirtyy seuraavaan peruslohkoon (β_j).

Numero	Komento	Peruslohko	Seuraavat komennot
1.	print 'f'	$\beta 1$	(2)
2.	readUnsign i		(3)
3.	jmpIfZero i, 7	$\beta 2$	(4, 7)
4.	dec i	$\beta 3$	(5)
5.	print 'o'		(6)
6.	goto 3		(3)
7.	exit	$\beta 4$	$\emptyset$

Taulukko 3.1: Ohjelman $Fooo^n$ peruslohkot

Ajatellaan yksinkertaista taulukossa 3.1 esitettävää pseudokonekielen ohjelmaa $Fooo^n$, joka lukee *read*-komennolla toistojen määrän etumerkittömään kokonaislukuun ja tulostaa fo^n, jossa n on toistojen määrä.



G = { B, E }
B = { ß1, ß2, ß3, ß4 }
E = { (ß1, ß2); (ß2, ß3); (ß2, ß4); (ß3, ß2) }

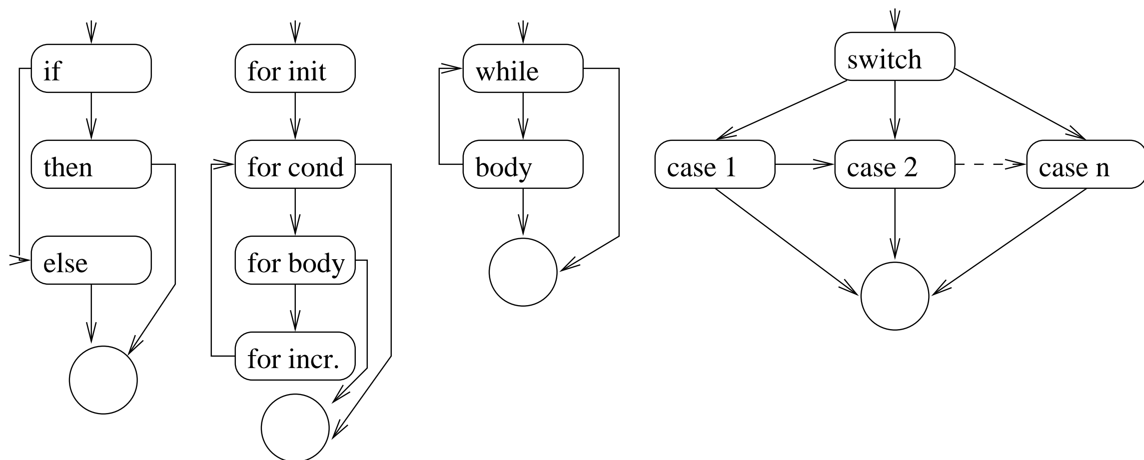
Kuva 3.1: Ohjelman $Fooo^n$ kontrollivuokaavio

Kontrollivuokaavio voidaan esittää normaaleja verkkojen esitystapoja käyttämällä, esimerkiksi palloja ja nuolia käyttämällä tai kaavion komponentit luettelemalla. Esimerkkinä kuvassa 3.1 näytetään $Fooo^n$-ohjelman peruslohkot ja mahdolliset kontrollisiirtymät näillä esitystavoilla.

Tämän kaltaisen kontrollivuokaavion automaattinen tuottaminen sopivasta esitysmuodosta on hyvin suoraviivaista. Aho ja kumppanit esittävät yksinkertaisen algoritmin kolmiosoite-lausekkeista (engl. *three-address statement*) koostuvien ohjelmien jaoittelemiseen peruslohkoihin: ohjelman ensimmäinen lauseke aloittaa uuden peruslohkon, jokainen hyppykomennon kohde aloittaa uuden peruslohkon, ja jokaisen hyppykomennon jälkeen alkaa uusi peruslohko [ASU86, s. 529].

Peruslohkon tasolla tapahtuva optimointi on yksinkertaisimmillaan ns. kurkistusaukko-optimointia (engl. *peephole optimization*), joissa ohjelman peruslohkon sisällä tarkistetaan, voitaisiinko kohdekoodin käskysarja korvata paremmalla [McK65]. Oliokielten useimmiten pinopohjaisissa välikielissä peruslohko esiintyy mahdollisen pinokäsittelyn uudelleenjärjestelyn yksikkönä.

Muokattua kontrollivuokaaviota voidaan käyttää myös kielen lauserakenteiden esittämiseen. Kuvassa 3.2 esitetään C-perustaisten kielten lauserakenteiden *if*, *for*, *while* ja *switch* esitys sellaisessa kontrollivuokaaviossa, jossa peruslohkojen sijaan sallitaan kielen kieliopin mukaiset rakenteet kussakin solmussa. Kuvan laatikot esittävät rakenteen komponentteja, nuolet kuvaavat mahdollista kontrollin siirtymistä ja pallo esittää lauseen vaikutusalueen loppumista. Katkoviivalla esitetään samankaltaisen komponentin toistamista.



Kuva 3.2: C-pohjaisten kielten lauserakenteiden esittäminen kontrollivuoverkossa

If-lauseessa suoritetaan aluksi haarautumista kontrolloiva ehto. Ehdon ollessa tosi siirrytään *then*-haaraan, muutoin *else*-haaraan. Valitun haaran jälkeen lauseesta poistutaan. *For*- ja *while*-lauseissa suoritetaan silmukan runkoa, kunnes silmukan kontrolloiva ehto evaluoituu epätodeksi tai silmukan rungossa kohdataan *break*-käsky. *Switch*-lauseessa valitaan lausekkeen arvon perusteella jokin haaroista. Mikäli haara loppuu *break*-käskyyn,

poistutaan lauseesta, muutoin suoritetaan seuraava haara.

Pinopohjaiselle kielelle kontrollirakenteiden kääntäminen on varsin suoraviivaista, jos suorituksen edetessä kunkin kontrollivuonverkon solmun rajan yli pidetään pinomuuttujien suhteen tilanne samana kuin kyseiseen laatikkoon tultaessa. Mallikoodi (engl. *template code*) esimerkiksi if-rakenteen kääntämiseen esitetään kuvassa 3.3. Koodia tuottavassa metodissa on käytettävissä kaksi operaatiota, *emit* ja *backpatch*, joista ensimmäisellä generoidaan rekursiivisesti käskyjä ja jälkimmäisellä paikataan avoimeksi jääneitä hyppykohteita.

```
Expression if;
Statement  then, else;

compileIF() {
    emit(if);
    emit(then);
    backpatch(if); // to point to else-branch
    emit(else);
    backpatch(then); // to jump out of structure
}
```

Kuva 3.3: If-lauseen käännösmalli

Näin tuotetun pinokoodin lukeminen on myöskin suoraviivaista, sillä käännettävien rakenteiden rajat näkyvät tuotetussa koodissa hyvin. Esimerkiksi lause $x = x + 1$ kääntyy suoraviivaisesti kuvan 3.4 pinokäskyiksi, joissa aluksi ladataan muuttujan x arvo sekä vakio 1 pinon huipulle. *Add*-käskyllä lasketaan yhteen kahden pinon päällimmäisen arvoa ja poistetaan ne samalla pinosta. Tulos puolestaan taas kirjoitetaan pinon huipulle. Lopuksi *store*-käskyllä tulos kirjoitetaan muuttujan x arvoksi.

```
load x
load 1
add
store x
```

Kuva 3.4: Lauseen $x = x + 1$ suoraviivainen pinokoodi

Jos samaa sijoituslausetta sijoitettaisiin n kertaa lähdekoodiin peräkkäin, olisi suoraviivaisen käännöksen tuloksena n kopiota edellä esitetystä pinokoodista. Pinokäskyjen jär-

jestelyllä voidaan kuitenkin ylimääräisiä paikallisten muuttujien latauksia ja tallennuksia järkeistää [Koo94]. Kuvassa 3.5 esitetään, kuinka samaa muuttujaa käyttävien lauseiden kääntämisessä voidaan hyödyntää pinoa.

```
load  x    // x = x + 1 alkaa;
load  1                                 load x
add                                     load 1
store x                             =>  load 2
load  x    // x = x + 2; alkaa          add
load  2                                 add
add                                     store x
store x
```

Kuva 3.5: Lauseiden x=x+1; x=x+2 käskyjen uudelleenjärjestely

Pinokäskyjen järjestelyllä on suurin vaikutus puhtaasti tulkkaavaa suoritusta käytettäessä, sillä tällöin suoritettavien käskyjen määrä useimmiten pienenee. Kun pinoa uudelleenjärjestelevien käskyjen (esimerkiksi *swap*, jolla vaihdetaan kahden päällimmäisen pinoalkion paikkoja, ja *dup*, jolla toisinnetaan päällimmäinen pinoalkio) suoritusnopeus on paikallisten muuttujien käsittelemistä nopeampaa, saadaan myös siinä suhteessa hyötyä [ME98]. Rajoitetun tilan ohjelmissa optimointia voi myös pienentää suoritettavan koodin kokoa.

Pinokäskyjen järjestely perustuu ns. pinon kuvan (engl. *stack picture*) analysointiin. Pinon kuva on kuhunkin käskyyn liitetty esitys pinon ominaisuuksista ennen käskyn suoritusta ja kullekin käskylle on määritelty käskyn pinoefekti. Nämä tiedot yhdistämällä voidaan päätellä joitakin pinokoodin ominaisuuksia. Esimerkiksi Javan yhteydessä pinon kuvaan liitetty tyyppijärjestelmätieto mahdollistaa tyyppiturvallisuusominaisuuksien tarkistamisen luokkia ladattaessa [Gos95].

3.3 Kontrollivuoverkko ja kutsukaavio olio-ohjelmissa

Javan tavukoodi tarjoaa houkuttelevan kohteen ohjelman kontrollivuoverkon muodostamiseen. Esimerkiksi Systä [Sys00, s.133-136] ja Lance et al. [LUW99] esittävät kaksi erilaista algoritmia, joilla Javan tavukoodiesityksestä muodostetaan kontrollivuokaavio lukemalla kukin ohjelman käsky kaavion solmuksi ja yhdistämällä peräkkäiset käskyt ja hyppykäskyjen kohteet särmillä.

Näin yksinkertainen lähestymistapa toimii vain, jos analysoitavassa kielessä ei ole rakenteita dynaamisille hypyille, joiden kohde määräytyy suoritusaikaisesti. Tällaisia hyppyjä esiintyy tyypillisesti virtuaalikutsujen ja poikkeusten yhteydessä, jolloin oliokielen kontrollivuokaavion muodostaminen on hieman monimutkaisempaa kuin yksinkertaisemman kielen kontrollikaavion.

Oletetaan funktioiden kutsutiedon sisällyttävä kontrollivuokaavion muodostamisalgoritmi, jolle syötetään kuvassa 3.6 oleva ohjelmanpätkä. Ohjelmointikielessä, jossa ei tueta perintää ja myöhäistä sidontaa, voidaan nyt tästä pätkästä lisätä kontrollivuokaavioon kontrollin kulkua kuvaaviin särmiin rivin 2 kohdalta yksi uusi särmä, jossa kontrollin kohteena on abstraktin tietotyypin määritelty operaatio. Särmiä lisätään täsmälleen yksi, sillä kutsun kohde voidaan määritellä staattisesti.

```
1.     AbstractDataType adt = new AbstractDataType;
2.     adt.invokeOperation();
```

Kuva 3.6: Abstraktista tietotyypistä kutsutaan operaatiota

Siirryttäessä oliokieleen voidaan tämä sama algoritmi muokata tukemaan olio-ohjelmoinnin perintää ja myöhäistä sidontaa tulkitsemalla konservatiivisesti metodikutsun kohteiksi *kaikki* kutsun vastaanottajan staattisen tyypin aliluokat, sillä kontrollivuokaaviossa peruslohkojen väliset särmät kuvaavat mahdollista kontrollin siirtymistä. Tällä tiedolla saatetaan pystyä parantamaan operaatiokutsujen toteuttamista: joissain tapauksissa, esimerkiksi mikäli metodin toteuttavan luokan aliluokista mikään ei syrjäytä kyseistä metodia, saatetaan viestin kohde pystyä sitomaan jo staattisesti [DGC95, AH96, Fer95].

Krall ja Horspool esittävät algoritmin 3.1, jolla ohjelman kutsukaavio voidaan muodostaa [KH02]. Algoritmi aloittaa läpikäynnin analysoitavan ohjelman *main*-metodista ja lisää kaikki viitatut luokat ja niiden kantaluokat ja rajapinnat luokkahierarkiaan.

Ohjelmaa kokonaisuutena optimoitaessa avautuu mahdollisuus metodikutsujen optimointiin, jos esimerkiksi luokkahierarkian perusteella nähdään, että kutsulla on vain yksi mahdollinen kohde. Tällöin operaatiokutsun kohde voidaan selvittää jo käännösaikaisesti. Dynaamisissa kielissä ongelmaksi muodostuu kuitenkin analyysin suoritusaika, sillä luokkahierarkian analyysin jälkeen saatetaan luokkahierarkiaa muokata lataamalla uusia luokkia. Tällöin myös analyysin perusteella turvallisiksi todettu staattinen sidonta saattaa vaihtua epäturvalliseksi.

Ajatellaan vaikkapa kuvan 3.7 ohjelmaa, jossa kutsutaan luokan metodia. Nyt, mikäli luokkahierarkian analyysin perusteella luokalla *SpecificObject* ei ole aliluokkia tai mikään

Data : main on ohjelman käynnistävä metodi
Data : x() on luokkametodin x (staattinen) kutsu
Data : type(x) on lausekkeen x tyyppi
Data : x.y() on lausekkeen x tuottamasta viitteestä virtuaalisen metodin y kutsu
Data : subtype(x) on joukko, joka sisältää x:n ja kaikki sen alityypit
Data : method(x, y) on luokassa x määritelty metodi y
Data : m-stat on metodin suorituksessa esiintyvä staattisesti sidottava kutsu
Data : m-virtual on metodin suorituksessa esiintyvä dynaamisesti sidottava kutsu

```
callgraph ← main;
hierarchy ← ∅;
foreach m ∈ callgraph do
    foreach m − stat metodissa m do
        if m − stat ∉ callgraph then
            callgraph + m_stat;
    foreach e.m − virtual metodissa m do
        foreach c ∈ subtype(type(e)) do
            m − def ← method(c, m_virtual);
            if m − def ∉ callgraph then
                callgraph + m − def;
                hierarchy + (c, e);
```

Algoritmi 3.1: Luokkahierarkian ja kutsukaavion muodostava algoritmi [KH02]

aliluokista ei syrjäytä metodia *methodCall*, voidaan metodikutsun kohteen päätellä olevan nimenomaisesti luokan *SpecificObject* määrittelemä *methodCall* ja optimointina voidaan se laventaa kutsukohtaan.

Dynaamisessa kielessä olioviitteen tuottava metodi *getAnySpecificObject* voi kuitenkin esimerkiksi kysyä käyttäjältä verkko-osoitteen, josta seuraavalla kierroksella palautettavan olion luokka ladataan. Kun ladattu luokka on jokin *SpecificObject*in aliluokka tai jonkin aliluokan metodimäärittely muutetaan syrjäyttämään kantaluokan antama määrittely, muuttuu optimoinnin turvallisuuteen vaikuttavien esiehtojen totuusarvo eikä enää voidakaan päätellä metodikutsun kohdetta [**?**].

Ongelma voidaan estää dynaamisella takaisinkääntämisellä: esiehtojen muuttuessa pätemättömiksi palataan optimoimattomaan versioon, josta myöhemmillä dynaamisilla käännöksillä voidaan taas yrittää optimointeja. Palautusprosessissa joudutaan yllä esitetyssä

```
while(true) {
    SpecificObject obj = getAnySpecificObject();
    obj.methodCall();
}
```

Kuva 3.7: Operaation kutsuminen abstraktista tietotyypistä

tapauksessa päivittämään myös kutsupinoa, sillä kutsuvan kohdan muokattu versio löytyy pinosta [HCU92]. Tämän takia kirjallisuudessa menetelmää kutsutaan nimellä pinonsisäinen korvaus (engl. *in-stack replacement*).

3.4 Pakenemisanalyysi

Eräs tärkeä ero rakenteisen ja olio-ohjelmoinnin välillä voidaan nähdä ohjelmointikielen entiteettien elinajassa. Rakenteisessa ohjelmoinnissa on tavallista, että käsiteltävän kohteen elinaika on sidottu käsittelevän ohjelmakoodin elinaikaan. Olio-ohjelmoinnissa taas tavallisesti luotujen olioiden elinaika ei ole sidottu olion luovan metodin elinaikaan.

Olioita käsiteltäessä on luontevaa noudattaa viitesemantiikkaa, jolloin uutta oliota alustettaessa suoraviivainen toteutusmekanismi on aina varata tilaa keosta ja antaa roskienkerääjän huolehtia tilan vapauttamisesta siinä vaiheessa, kun olioon ei enää voida viitata. Automaattinen muistinhallinta vähentää mahdollisuuksia vaikeasti löydettävien virheiden tekemiseen, mutta keosta tilan varaaminen on hitaampaa kuin pinosta tapahtuva paikallisille muuttujille tarkoitettu tilan varaaminen.

Roskienkeruu ei kuitenkaan ole ainoa automaattisen muistinhallinnan menetelmä. Joissain tapauksissa olioiden elinaika on sidottu luovan ohjelmakoodin elinaikaan. Tällaisten olioiden tunnistamiseen käytetään pakenemisanalyysiä (engl. *escape analysis*) – kysymys on siitä, voiko viite olioon päästä olion instantioivan koodin näkyvyysalueen ulkopuolelle. Luonnollinen tilanvaraukseen liittyvä optimointi on varata analyysin osoittamille olioille tilaa keon sijaan pinosta [Bla99]. Tällöin kyseisten olioiden tilanvaraus ja -vapautus onnistuu automaattisesti suorituspinon ohjaamana [GS00].

Tämä optimointi toimii jo pienellä analyysivaivalla käytettäessä olioita kontrollirakenteina, kuten esimerkiksi Iteraattori-mallissa tehdään. Ajatellaanpa Javassa varsin tavallista kuvan 3.8 mukaista rakennetta, jossa kyseistä mallia käytetään. Pienelläkin analyysillä nähdään, että *iter*-oliota käytetään vain silmukan sisällä, jolloin tila sille voidaan varata paikallisten muuttujien alueelta, eli pinosta. Tällöin tilanvaraus on nopeampaa kuin keosta

varattaessa ja toisaalta roskienkerääjälle ei synny ylimääräistä työtä, sillä tila vapautetaan samassa yhteydessä kuin muidenkin paikallisten muuttujien varaama tila.

```
for (Iterator iter = coll.iterator(); iter.hasNext(); ) {
    Object ob = iter.next();

    // do something with the object
}
```

Kuva 3.8: foreach-rakenne Iteraattori-mallia käyttäen

Staattisella analyysillä varmuuden saaminen siitä, että esimerkissä viite *iter*-muuttujan viittaamaan olioon ei pääse pakenemaan ja on siten aidosti paikallinen olio, on kuitenkin vaikeaa. Yleisessä tapauksessa tulee käsitellä kaikkia mahdollisia rajapinnan Iterator-toteutuksia. Jos kuitenkin esimerkin *coll*-muuttujan konkreettinen tyyppi tunnetaan, voidaan sen tuottaman iteraattorin konkreettinen tyyppi analysoida ja tätä kautta tutkia iteraattorin *hasNext*- ja *next*-metodien toteutuksia. Mikäli kumpikaan näistä ei vuoda viitettä olioon itseensä, on esimerkin *iter*- olio paikallinen.

Koska muistinvarauksessa on nähtävissä pienelläkin vaivalla hyviä suorituskykyparannuksia, tullaan Javan seuraavassa versiossa 6 toteuttamaan myös pakenemisanalyysiin perustuvia muistinvarausoptimointeja.

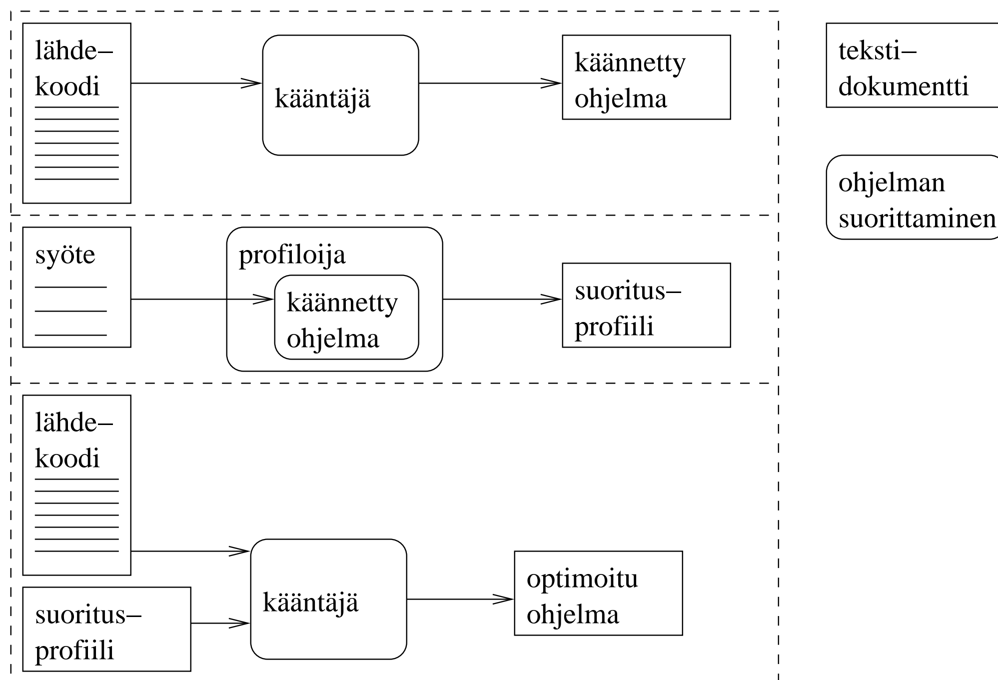
3.5 Profilointi

Yllä kuvatut analyysimenetelmät ovat sovellettavissa puhtaasti staattisessa ympäristössä käytettäviksi. Mikäli kuitenkin rajoitutaan vain ohjelman toiminnasta lähdekoodia tutkimalla saatuun tietoon, jää analyyseiltä suorituskyvyn kannalta oleellista tietoa piiloon.

Tilanteen parantamiseksi analyyseille voidaan syöttää profiilitietoa (engl. *profile information*), jota on kerätty seuraamalla ohjelman todellista suoritusta jollain syötteillä [Knu71]. Tällöin analyysirakenteita voidaan täydentää todellisella suoritusmittaritiedolla. Esimerkiksi monihaaraisesta *switch*-lausekkeesta voidaan kontrollivuokaavioon merkitä kunkin haaran valintatodennäköisyys myöhempiä optimointeja, kuten hyppykäskyjen järjestelemistä, varten.

Menetelmiä suoritusprofiilin keräämiseen on kahta laatua: ajastettuun näytteenottoon perustuva sekä kohdeohjelman muokkaamiseen perustuva. Ajastettu näytteenotto toimii käyttöjärjestelmä- tai muun ohjelmiston tukemana siten, että säännöllisin väliajoin suoritetta-

va ohjelma keskeytetään ja sen suoritustila merkitään talteen. Kohdeohjelman muokkaamiseen perustuva profilointi taas instrumentoi ohjelmaa lisäämällä suoritustilan laskureita sopiviin paikkoihin, jotka päätyvät sitten suoritukseen muun ohjelman ohessa [Knu71].



Kuva 3.9: Profiloijan käyttäminen käännöstuloksen parantamiseen

Jos käytettävissä on näytteenottoon perustuva profiloija, voidaan sen tuottamaa tietoa käyttää käännöksessä hyväksi kuvan 3.9 esittämällä tavalla. Kuvassa suorakulmaiset laatikot esittävät staattisia dokumentteja ja pyöristetyillä kulmilla olevat laatikot puolestaan ohjelman suoritusta jollain syötteellä. Prosessi aloitetaan kääntämällä optimoitava ohjelma virheenjäljitystiedot (engl. *debug information*) päälle käännettyinä, jolloin saadaan ensimmäinen, profiilin perusteella optimoimaton versio ohjelmasta (kuvan ylin laatikko).

Seuraavaksi tämä optimoimaton ohjelma suoritetaan profiloijan 'sisällä' mahdollisimman tavanomaista syötettä käyttäen. Profiloija tarkkailee ohjelman suoritusta ja muodostaa näytteiden perusteella ohjelman suoritusprofiilin (keskimmäinen laatikko). Tässä vaiheessa voi ohjelmoija tarkastella ohjelman suoritusprofiilia mahdollisten pullonkaulojen varalta. Ensimmäistä kertaa suoritusprofiilia tarkasteltaessa tulokset voivat olla järisyttäviä: FORTRAN-kokeessa monessa tarkastellussa ohjelmassa yli puolet suoritusajasta kului neljässä prosentissa ohjelman riveistä [Knu71].

Profiloinnin hyödyt eivät kuitenkaan rajoitu ohjelmakoodin ongelmakohtien etsimiseen, vaan suoritusprofiilia voidaan hyödyntää myös kääntäjän suorittamien optimointien syöt-

teenä kääntämällä alkuperäinen ohjelma uudestaan (kuvan 3.9 viimeinen laatikko). Tällöin optimoinneissa voidaan hyödyntää todellisen suorituksen pohjalta saatuja tietoja, joiden perusteella voidaan optimointeja sovittaa juuri tämän optimoitavan ohjelman erikoistapauksiin, vaikkapa switch-lauseen haarojen sijoittelun päätöksiin [CMH91].

Olio-ohjelmissa ohjelman suoritusprofiilia voidaan hyödyntää lähinnä virtuaalisten metodien sidonnassa. Suoritusprofiilia tutkimalla voidaan saada selville virtuaalisen kutsun tavanomaisimmat vastaanottajat. Vaikka periaatteessa virtuaalisen metodin lopullinen toteutus voidaan tietää vasta suoritusaikana, voidaan vastaanottajan tyyppien testausjärjestys järjestää profiilin perusteella siten, että useimmiten toteutunut konkreettinen luokka testataan ensimmäisenä [HU94].

Profiilin mahdollisuuksia tutkittaessa on huomattu, että olio-ominaisuuksia hyödyntävissä ohjelmissa noin 70% virtuaalisista kutsuista päätyy yleisimmin esiintyvälle vastaanottajalle ja jopa puolella myöhään sidotuista kutsuista on vain yksi mahdollinen vastaanottaja [GDGC95]. Suoritusprofiilia hyödyntämällä voidaan siis huomattavan suuressa osassa myöhään sidottuja kutsuja löytää kutsun konkreettinen vastaanottaja heti ensimmäisellä arvauksella. Toisaalta suoritusprofiili ei mahdollista dynaamisen sidonnan poistamista, sillä muutoin jäljelle jäävät 30% dynaamisista kutsuista ohjattaisiin väärälle vastaanottajalle.

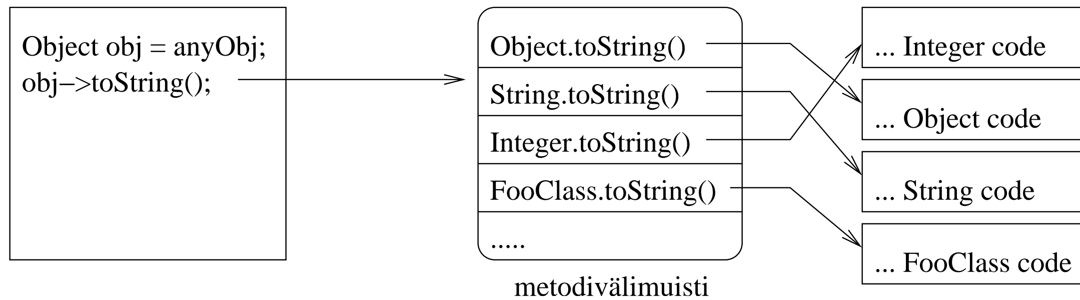
3.6 Viestin vastaanottajien muistaminen

Smalltalk-80-järjestelmässä metodikutsut ajatellaan viesteinä, joilla on vastaanottaja sekä metodin nimi, jonka perusteella kutsuttava funktio valitaan. Koska viestinvälityksen tehokkuus on tällaisen järjestelmän suorituskyvyn kannalta oleellinen tekijä, käyttivät järjestelmän toteuttajat huomattavastikin aikaa mekanismin tehostamiseen.

Tavallisesti olion vastaanottaessa viestin tutkitaan, löytyykö viestin signatuuria vastaavaa metodia olion luokasta. Ellei, tutkitaan tämän luokan kantaluokasta ja jatketaan niin pitkään kun etsinnän kohteena olevalla luokalla on kantaluokkia. Mikäli tällöinkään signatuuria vastaavaa metodia ei löydy, saa vastaanottaja *doesNotUnderstand*-viestin [GR83, s. 61].

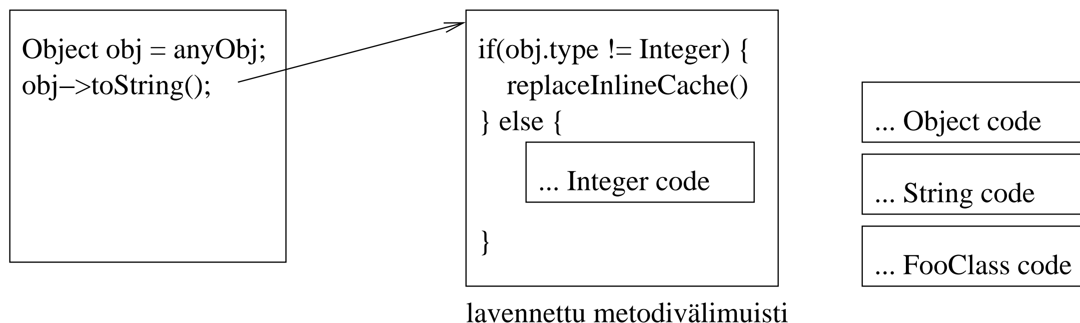
Standardiparannus metodinvalintaan on käyttää metodivälimuistina hajautustaulua, jossa avaimena on vastaanottava luokka sekä metodin signatuuri ja arvona kutsuttavan metodin osoite. Viestinvälityksen yhteydessä tutkitaan, löytyykö viestiä vastaava kirjaus hajautustaulusta. Kun löytyy, kutsutaan löydettyä funktiota; kun ei löydy, haetaan kutsuttava funktio tavallisella metodiselvitysmenetelmällä ja talletetaan se hajautustauluun [CPL83].

Hajautustauluun perustuva viestin vastaanottajan valitseminen esitellään kuvassa 3.10.



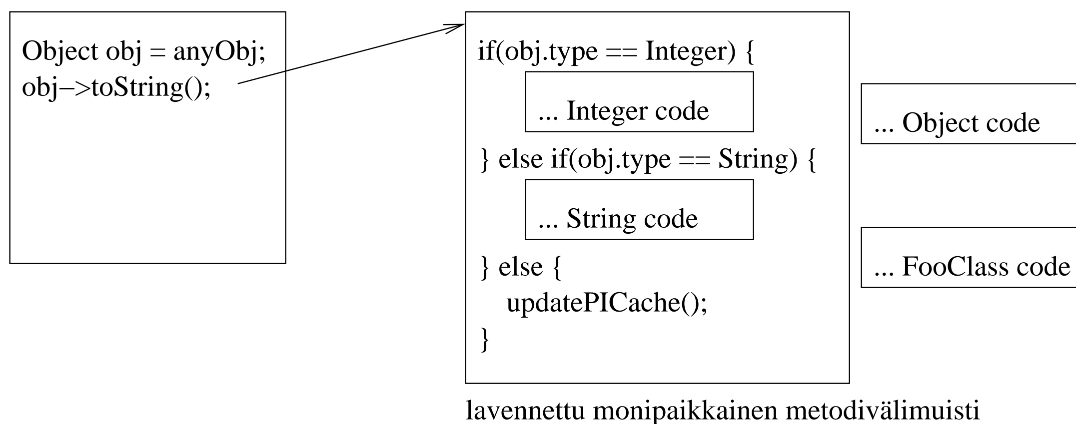
Kuva 3.10: Hajautustauluun perustuva viestin vastaanottajan valitseminen

Yhden, keskitetyn hajautustaulun sijaan voidaan viestin vastaanottaja sijoittaa myös kutsupaikassa sijaitsevaan välimuistiin. Ensimmäinen Smalltalk-80:ssä käytetty lavennettu metodivälimuisti (engl. *inline cache*) tallensi kullekin kutsupaikalle edellisen kutsukerran kohdemetodin osoitteen [DS84]. Koska myöhäisen sidonnan idea kuitenkin on mahdollistaa konkreettisen kohdemetodin vaihtuminen, tarvitaan kohdekoodiin myös tarkistus siitä, että kutsuttava luokka on edelleenkin sama. Mikäli on, voidaan metodi suorittaa; mikäli ei ole, haetaan oikea metodi normaalilla valitsemismenettelyllä ja sijoitetaan sen osoite kutsuvaan koodiin. Lavennetuilla metodivälimuisteilla kutsupaikan ja -kohteen sijoittelu esitetään kuvassa 3.11.



Kuva 3.11: Lavennettuun metodivälimuistiin perustuva viestin vastaanottajan valitseminen

Lavennettua metodivälimuistia käytettäessä tehokkuushyöty perustuu havaintoon, jonka mukaan myös myöhäistä sidontaa käytettäessä suurin osa tietystä kutsupaikasta tapahtuvista kutsuista kohdistuu yhden ja saman luokan metodiin. Tällöin globaalin metoditaulun läpikäyminen tuottaa vain ylimääräistä kuormaa, kun taas kutsukohtaan laventamalla ylimääräinen lisärasite tavallisessa, ei-polymorfisessa tapauksessa rajoittuu yhteen vertailulauseeseen.



Kuva 3.12: Useampipaikkaiseen lavennettuun metodivälimuistiin perustuva viestin vastaanottajan valitseminen

Olio-ohjelman virtuaalisten metodien tapausta analysoidessa huomataan, että yksipaikkaisen metodivälimuistin sijaan voidaan käyttää useampipaikkaista välimuistia, jolloin kullekin kutsukohdalle voidaan muodostaa mahdollisia kutsun kohteita sisältävä polymorfinen lavennusvälimuisti (engl. *polymorphic inline cache*), jossa tyyppitarkistusta tekevää käärekoodia muokataan siten, että tyyppitarkistuksen epäonnistuessa välimuistitauluun lisätään uuden luokan nimi ja selvitetty metodin osoite ja mikäli kutsun kohde on riittävän pienikokoinen, voidaan se myös laventaa välimuistialueelle [HCU91]. Kuvassa 3.12 näytetään tällaisen välimuistin rakenne, kun kutsupaikan *obj* on ollut sidottuna Integer- ja String-luokkien olioihin.

3.7 Luokkahierarkiaa uudelleenjärjestelevä optimointi

Tutkielman puitteissa kehitettiin ohjelman luokkarakennetta uudelleenjärjestelevä optimointimenetelmä. Optimointi on idealtaan yksinkertainen: kun oletetaan luvussa 3.1 esitetty suljettu maailmankuva, voidaan sellainen rajapinta poistaa, jonka toteuttavien luokkien lukumäärä on korkeintaan yksi. Tällaisia tilanteita syntyy esimerkiksi käytettäessä ohjelmistokehystä, joka määrittelee erikoistamispisteitä rajapintamekanismia käyttäen. Aikaisemmin aihetta on käsitelty lähteessä [Poh04].

Oliopohjaisiin ohjelmistokehyksiin (engl. *object-oriented application framework*) perustuvassa ohjelmistokehityksessä käytettävä ohjelmistokehys antaa sovelluksen rungon, johon sovelluskohtaiset palaset liitetään erityisten erikoistamispisteiden (engl. *hot spot*) kautta. Javaa käytettäessä on luonnollista toteuttaa nämä rajapintaluokkina: kehys mää-

rittelee rajapinnan, jonka sovelluspuolen luokka toteuttaa.

Algoritmin 3.1 idea on eräs tapa tuottaa optimoitavan ohjelman luokkahierarkian rakenne. Dynaamisia kieliä optimoitaessa kuitenkin reflektion kautta käytettävät metodikutsut jäävät huomioimatta: jos ohjelma esimerkiksi kysyy käyttäjältä seuraavaksi kutsuttavan metodin nimeä, ei algoritmi tätä tunnista. Toisaalta, koska kehitelty algoritmi perustuu puhtaasti luokkahierarkian rakenteeseen, voidaan käyttää yksinkertaista, kaikkien luokkien listauksen läpikäyvää algoritmia 3.2.

Data : classlist() on ohjelman kaikki luokat ja rajapinnat sisältävä joukko
Data : implements(x) on kaikki luokan x toteuttamat rajapinnat sisältävä joukko
hierarchy $\leftarrow \emptyset$;
foreach $c \in$ *classlist()* **do**
    foreach $i \in$ *implements(c)* **do**
        $hierarchy$ += (c, i);
    end
end

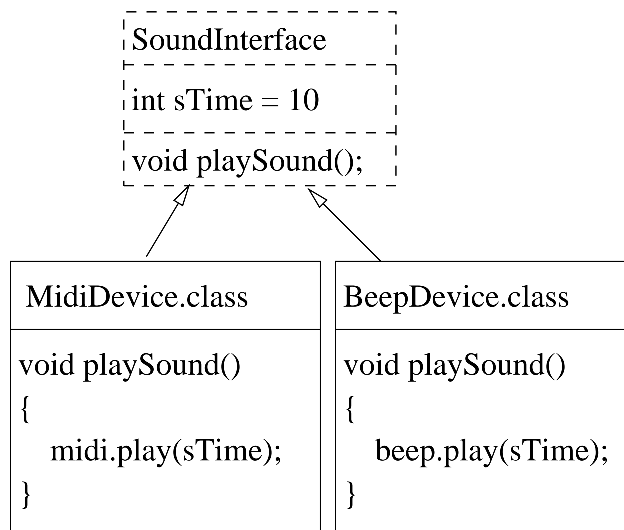
Algoritmi 3.2: Luokkahierarkian muodostava yksinkertainen algoritmi

Tämän yksinkertaisemman algoritmin hyvänä puolena on toimivuus myös reflektiota käyttävien ohjelmien kanssa. Toisaalta se on epätarkka, sillä se ei tunnista ohjelman luokkalistauksessa olevia käyttämättömiä luokkia.

Esimerkiksi kännyköihin pelejä ja muita ohjelmia kehitettäessä on luonnollista abstrahoida eri kännykkämallien väliset ominaisuudet rajapinnan taakse. Kullekin kännykkämallille tuotetaan toteutus, joka paketoidaan kyseiselle laitteelle sovitettuun versioon. Kuvassa 3.13 esitetään yleinen äänirajapinta sekä toteutukset kahdelle eritasoiselle laitteistolle.

Eroavuudet eri äänentoistolaitteistojen tai -toteutusten välillä on nyt kapseloitu rajapintansa taakse. Sovelluslogiikan sisältävä ohjelmistokehys sisältää vain tämän rajapinnan. Kehyksestä johdetut sovellukset puolestaan täydentävät kehystä antamalla toteutukset kustakin erikoistamiskohdasta. Tällöin esimerkkitapauksessamme kahdelle eri laitteelle sovitetut sovellusversiot sisältävät kumpikin erikoistamiskohdan esittävän rajapintaluokan sekä itse toteutuksen sisältävän luokan. Rajapintaluokka voidaan kuitenkin lopullisesta sovelluksesta poistaa, sillä rajapinnalla haetaan vain kehitysaikaista joustavuutta. Kuvassa 3.14 esitetään luokkarakenteen uudelleenjärjestämisen vaikutus esimerkkitapauksessa.

Esimerkki voidaan esittää myös formaalimmin. Javassa, jossa luokalla on korkeintaan yksi kantaluokka, riittää pelkkiä luokkia analysoitaessa talletusrakenteeksi puu, jossa kulla-



Kuva 3.13: Luokkahierarkia yksinkertaisille äänille ja MIDI-äänille

kin solmulla on tasan yksi välitön vanhempi[2].

Kun Javan yhteydessä käsitellään rajapintatietoa, ei puurakenne enää ole riittävä, sillä yhden kantaluokan lisäksi kukin luokka voi toteuttaa mielivaltaisen määrän rajapintoja. Tällöin tarvitaan yleisempi verkkorakenne. Määritellään Javassa luokkahierarkiakaavio suunnatuksi verkoksi

$$G = (C, I) \tag{3.2}$$

jossa C on kaikkien luokkien ja rajapintojen joukko $(\gamma_1, \gamma_2, ..., \gamma_n)$ ja I puolestaan joukko suunnattuja särmiä $\Big((\gamma_i, \gamma_j), (\gamma_k, \gamma_l), ...\Big)$. Verkossa rajapinnan toteuttamista kuvaavassa särmässä (γ_i, γ_j) ensimmäinen jäsen tarkoittaa luokkaa, joka toteuttaa toisena jäsenenä olevan rajapinnan.

Ensimmäisen kertaluvun predikaattilogiikalla optimoinnin esiehto rajapinnan γ_x suhteen voidaan nyt kirjoittaa muotoon

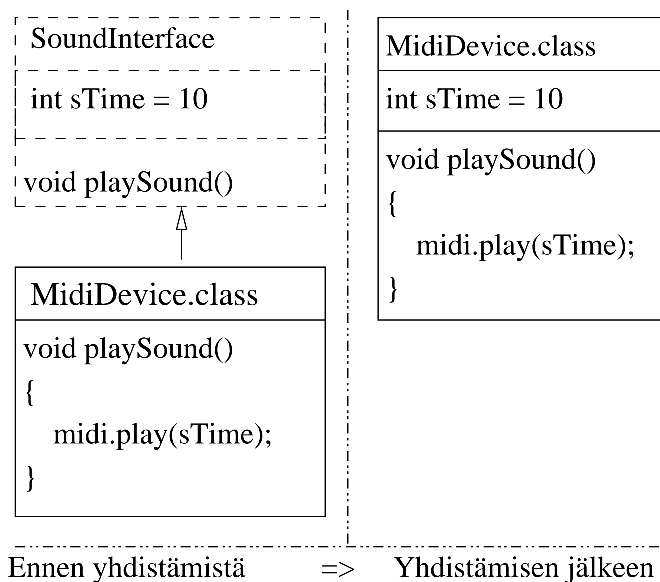
$$A(x) = (x, \gamma_x) \in I \tag{3.3}$$

$$\exists x A(x) \wedge \forall x \forall y (A(x) \wedge A(y) \rightarrow (x = y)) \tag{3.4}$$

eli on olemassa täsmälleen yksi luokka x, joka toteuttaa rajapinnan γ_x.

Suljetun maailman oletuksen pätiessä voidaan ohjelmasta poistaa kaikki ne rajapinnat, joiden osalta kaava 3.4 pätee. Tällaisten rajapintaluokkien listaamiseen löytyy erilaisia tapoja, joita käsitellään seuraavassa luvussa.

[2]Poikkeuksena tietysti *java.lang.Object*



Kuva 3.14: Rajapintaluokan yhdistäminen toteuttavaan luokkaan

4 Optimoivan järjestelmän toteutus

Edellisessä luvussa esitettiin muiden olio-optimointien lisäksi luokkahierarkiaa uudelleenjärjestelevä optimointi, jonka määritelmä onnistuttiin antamaan ensimmäisen kertaluokan predikaattilogiikkana. Tässä luvussa näytetään miten vastaavantasoisesta määritelmää voidaan käyttää suoraan optimoivan järjestelmän lähdekoodina.

4.1 Optimointien määritteleminen formaalein menetelmin

Kääntäjän suorittamat optimoinnit toteutetaan tavallisesti osana kääntäjän muuta lähdekoodia. Tällöin toteutuksesta tavallisesti muotoutuu käsinkirjoitettu monimutkainen systeemi. Kääntäjäteknologioihin liittyy vahvasti toteutustyön automatisointi. Kääntämisprosessin moniin osiin on onnistuneesti saatu sovellettua laskennallisia formalismeja: selaajia voidaan automaattisesti tuottaa säännöllisistä lausekkeista esimerkiksi *lex*illä [LS], ja jäsentimiä voidaan generoida puolestaan *yacc*illa [Joh79].

Kääntäjän optimoinneille vastaavia generaattoreita on varsin vähän, sillä tehtävään ei ole löytynyt sopivaa Turingin konetta heikompaa laskentaformalismia - ja sellaisen löytyminen on varsin epätodennäköistä, sillä erilaiset optimoinnit toimivat niin monella abstraktiotasolla, että niiden kaikkien kattaminen tavallista ohjelmointikieltä ilmaisukyvyttömämmällä, ja siten helpommin ymmärrettävällä formalismilla olisi yllättävä löydös. Eri

abstraktiotasoisia yksittäisiä formalismeja on kylläkin kehitelty: kääntäjän takaosan generointivaiheeseen on kehitelty dynaamiseen ohjelmointiin perustuvia työkaluja, esimerkiksi *twig* [AGT89]; lähdekielen tasolla toimiville optimoinneille on kehitelty esimerkiksi Gospel-niminen (General Optimization Specification Language) kieli [WS94, WS97].

```
DECLARATION
    Statement: Si;
PRECONDITION
    Code_Pattern
        any Si: type(Si.opr2) == const AND
                type(Si.opr3) == const AND
                Si.opcode     != assign;
ACTION
    modify(Si.opr2,   eval(Si.opr2, Si.opcode, Si.opr3));
    modify(Si.opcode, assign);
```

Kuva 4.1: Vakio-operandit käännösaikaisesti laskeva optimointi Gospel-kielellä [WS94]

Gospel-kielessä optimoitavat rakenteet esitetään predikaattilogiikan kaltaisilla kaavoilla, jotka käsittelevät optimoitavasta ohjelmasta rakennetun mallin tieto- ja kontrolliriippuvuuksia. Kun mallista löydetään optimoitava osuus, annetaan optimoinnin kuvauksessa mallia muokkaavat käskyt, jotka suoritetaan käännösaikaisesti ja jotka parantavat käännettävää ohjelmaa halutulla tavalla [WS97].

Optimointisäännön kirjoittamisen jälkeen järjestelmään liittyvä *Genesis*-niminen työkalu lukee säännön määrittelyn ja generoi siitä kyseisen optimoinnin toteuttavan kohdekoodin, joka puolestaan voidaan yhdistää kääntäjään samaan tapaan kuin selaaja- ja jäsentäjägeneraattorin tuottamat kääntäjän osat yhdistetään [WS94].

Esimerkiksi 'lausekkeen korvaaminen vakiolla'-optimointi esitetään Gospel-kielessä kuvan 4.1 esittämällä tavalla. Optimointi toimii siten, että löydettäessä kolmiosoitemuodossa olevasta ohjelman muodosta operaatio, jonka molemmat operandit ovat vakioita, suoritetaan kyseinen operaatio jo käännösaikaisesti ja korvataan alkuperäinen operaatio sijoituksella. Tällöin vaikkapa ohjelmakoodin lause $func(2+4)$ korvattaisiin lauseella $func(6)$, kuten kuvassa 4.2 esitetään.

Gospel-järjestelmään kuuluu optimointien määritelmistä lopullisen optimoijan generoiva työkalu, jolla määrätyssä muodossa oleviin ohjelmiin voidaan soveltaa määriteltyjä optimointeja. Järjestelmän yleiskäyttöisyyttä kuitenkin haittaa tämänhetkisen toteutuksen

```
alkuperäinen:           paranneltu
--------------          -----------------
tmp1 = 2 + 4            tmp1 = 6
func(tmp1)              func(tmp1)
```

Kuva 4.2: Vakio-operandin käännösaikainen evaluointi

käyttämä kolmiosoitekoodi, joka ei ole muiden kääntäjäjärjestelmien kanssa yhtenevä.

Tässä tutkielmassa ongelmaa asiaa lähestytään yksinkertaisemmin. Koodin generoinnin sijaan käytetään tulkkia, jolle Prolog-järjestelmän tapaan syötetään kysely. Kyselyn vastauksena saadaan analyysimenetelmien tuottamaan malliin sovitetut kyselyn vapaat muuttujat.

4.2 Emoole

Tutkielman ohessa toteutetun logiikkatulkin nimi on *Embeddable Object-Oriented Logic Engine*, Emoole. Tulkki on toteutettu Javalla, ja se on upotettavissa pienellä vaivalla muihin Java-ohjelmiin. Toisin kuin Prologissa, Emoolessa itse päättelysäännöt toteutetaan myös Javalla ja itse tulkille annetaan vain ohjeet siitä, minkälaiseen hahmoon päättelysääntöjen tuntema tieto pitäisi sovittaa. Järjestelmän yleisrakenne esitetään kuvassa 4.3.

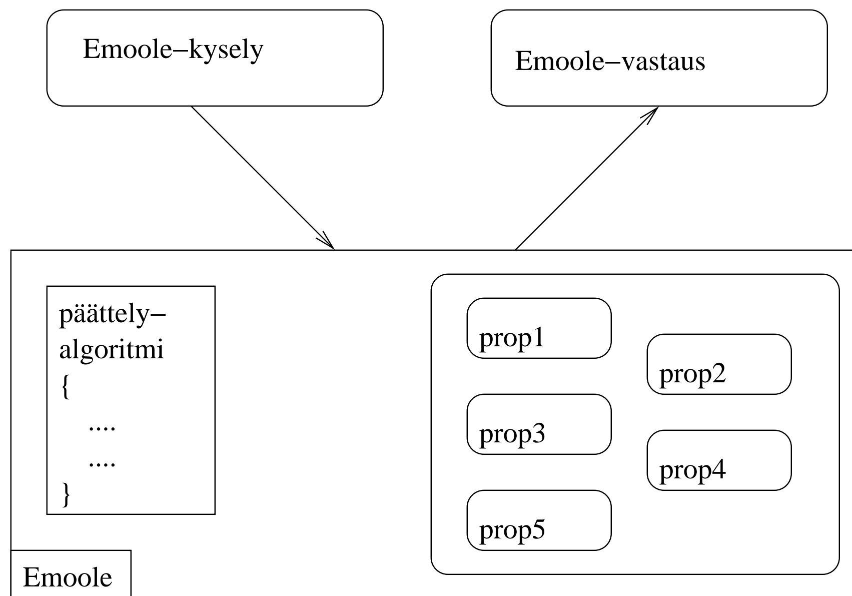
Emoole koostuu järjestelmään ohjelmoidusta päättelyalgoritmista sekä sovellusaluekohtaisista propositiosäännöistä, joita päättelyalgoritmi kutsuu. Systeemiä käytetään kirjoittamalla propositiosääntöjä käyttäviä yksinkertaisia kyselyitä, jotka voidaan kirjoittaa joko Emoolen omalla kielellä tai suoraan alla olevan Java-kielen oliorakenteina.

Emoolen kyselykieli mahdollistaa propositiosääntöjen ketjuttamisen AND- konnektiivilla, merkkijono-literaalit sekä mielivaltaisen määrän vapaita muuttujia. Monimutkaisemmat kyselyt kirjoitetaan suoraan Javana, jolloin voidaan hyödyntää sen tarjoamaa tyyppijärjestelmää. Kehitelty kyselykieli toimii lähinnä konseptuaalisena apuna; käytännön sovellutuksissa Emoole-kyselyt kirjoitettaneen Javalla.

Otetaan esimerkiksi perintää testaava propositio EXTENDS. Sen evaluointimetodi on kuormitettu seuraavasti[3]:

- evaluate(Variable var, String clazz) - muuttujan *var* arvoiksi sidotaan kaikki ne luo-

[3] Variable on Emoole:n luokka



Kuva 4.3: Emoolen korkean tason rakenne

kat, jotka perivät muuttujaan *clazz* sidottun luokan.

- evaluate(String clazz, Variable var) - muuttujan *var* arvoiksi sidotaan kaikki ne luokat, jotka muuttujaan *clazz* sidottu luokka perii.
- evaluate(String inheritee, String baseClass) - palauttaa arvon 'tosi', jos luokka *inheritee* on luokan *baseClass* jälkeläinen.

Uusien samaa ongelmakenttää koskevien propositioiden toteuttaminen on varsin suoraviivaista: helposti voidaan laajentaa esimerkkiä käsittelemään rajapintoja, luokkien jäsenmuuttujia ja -metodeja ja niin edelleen. Näin saadaan aikaan oliojärjestelmään upotettavissa oleva päättelykone, jonka päättelysäännöt voidaan kirjoittaa suoraan oliokielellä.

Käyttöesimerkki

Määreitellään propositiot CLASS(x), EXTENDS(x, y) ja EQUALS(x, y). Näistä keskimmäinen määritellään kuten edellisessä aliluvussa. Propositio CLASS määritellään seuraavasti:

- evaluate(Variable x) - muuttujan *x* arvoiksi sidotaan kaikki järjestelmään ladatut luokat, luokan *JavaClass* instansseina.

- evaluate(String clazz) - palauttaa arvon tosi, jos järjestelmästä löytyy luokka, jonka nimi on sama kuin merkkijonon *clazz* arvo. Tämä metodi tarvitaan, jotta Emoolen päättelyalgoritmi pysähtyy.

Propositio EQUALS puolestaan määritellään:

- evaluate(Object first, Object second) - välittää kutsun *first.equals(second)* paluuarvon Boolean-olioksi käärittynä.

Nyt voidaan Emoolelle syöttää kysely kaikkien luokan *java.lang.Object* aliluokkien listaamiseksi kuvan 4.4 mukaisesti. Vastaavan toiminnallisuuden kirjoittaminen suoraan Javalla vaati noin 100 riviä ohjelmakoodia. Suoraviivainen Java-toteutus hyödyntää vain yhtä suoritussäiettä siinä missä Emoole-kysely on luonteeltaan automaattisesti hajautettavissa useammalle säikeelle.

```
CLASS(x)                          AND
EQUALS(x, 'java.lang.Object')     AND
CLASS(y)                          AND
EXTENDS(y, x)
```

Kuva 4.4: Kysely kaikkien järjestelmässä olevien luokkien listaamiseksi

Päättelyprosessi

Kuvan 4.4 kyselyä suoritettaessa Emoolen jäsentäjä lukee kyselyn ja luo instanssit kyselyssä esiintyvistä elementeistä itse päättelykoneelle. Muuttujista x ja y syntyy Emoolen *Variable*-luokan olioita ja vakiosta 'java.lang.Object' syntyy Javan String-tyyppinen merkkijono, jolla on kyseinen arvo.

Itse päättelyprosessi on toteutettu suoraviivaisena silmukkana, joka käyttää pinoa välitulosten ylläpitämiseen. Tulkissa on neljä käskyä, joita soveltamalla hahmonsovitus tapahtuu:

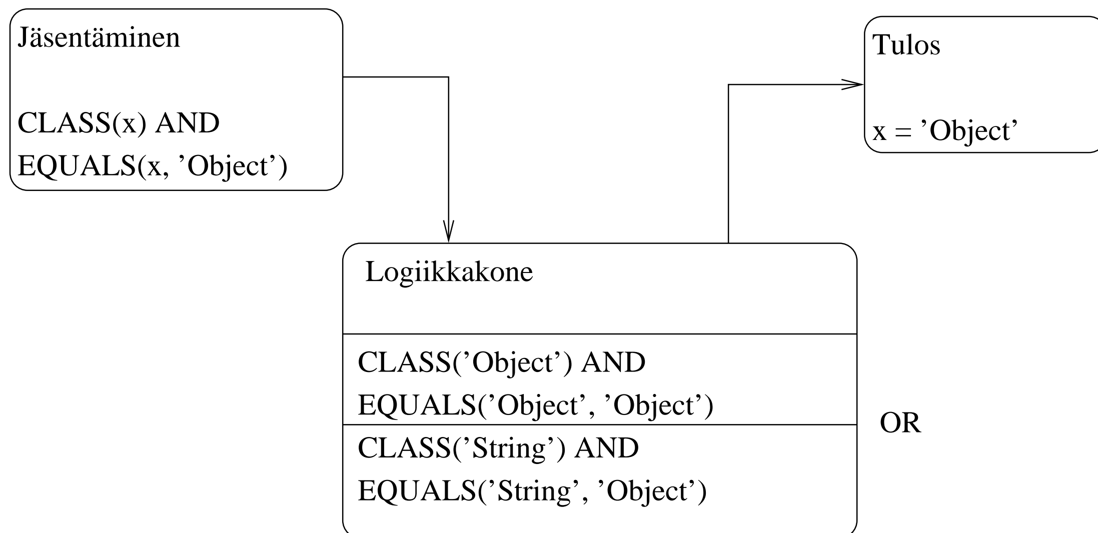
1. Lue arvoja pinon huipulta
2. Korvaa muuttuja arvolla
3. Testaa proposition totuusarvo

4. Kirjoita proposition tulos pinoon

Kukin propositio toteutetaan Javalla rajapinnan *Proposition* toteuttavaan luokkaan. Kullekin kyselyn propositiolle haetaan arvoja kutsumalla luokan *evaluate*-metodia, kunnes se palauttaa arvon *true* tai *false*. Muilla paluuarvoilla tulos kirjoitetaan takaisin pinoon ja silmukka aloitetaan alusta.

Proposition totuutta tarkasteltaessa käsittelevä metodi valitaan soveltamalla myöhäistä sidontaa kaikkiin argumentteihin, eli tavallisten Javan metodien sijaan kunkin proposition metodit käsitetään CLOS:in ja Cecilin tapaan multimetodeiksi. Näin samalla propositiolla voidaan sekä generoida uusia vastausvaihtoehtoja että tarkistaa hahmonsovittamisessa kullakin hetkellä kokeiltavana olevan hahmon osien sopivuutta nykyiseen dataan.

Kuvassa 4.5 esitetään päättelyn eteneminen kuvassa 4.4 annetun yksinkertaisen kyselyn yhteydessä.



Kuva 4.5: Päättelyprosessi yksinkertaisella kyselyllä

Yleisesti esitettynä päättely kulkee seuraavasti:

1. Jäsentäjä käsittelee annetun syötteen ja muodostaa listan, joka sisältää kaikki kyselyyn kuuluvat propositiot ja niiden argumentit. Tämä lista asetetaan työskentelypinon huipulle.

2. Logiikkakone lukee kyselyn pinon huipulta, vaihtaa muuttujia propositioiden luettelointisääntöä käyttäen literaaleiksi ja kirjoittaa pinoon kaikki ne kyselyt, jotka ovat vielä tässä vaiheessa tosia.

3. Tulosjoukkoon päätyvät ne kyselyt, joiden kaikki propositiot ovat muuttujien muuntamisen jälkeen tosia.

Suurin työ tehdään siis propositioiden toteutuksessa, eli proposition luettelointisäännössä ja totuussäännössä. Kuvan 4.5 suoritusta voidaan seurata myös tarkkailemalla tehtyjä metodikutsuja sekä pinon tilaa kussakin vaiheessa. Tällöin suoritusprosessi näyttää seuraavalta:

```
Pino (1): {CLASS(x)  AND  EQUALS(x, 'Object')}
```

Aluksi pinossa on syötteenä annettu kysely. Ensimmäinen käsiteltävä muuttuja on x, jolloin proposition CLASS luettelointisäännöltä kysytään kaikki saatavilla olevat luokat (joita ovat *Object* ja *String*).

Kullekin mahdolliselle vaihtoehdolle jokainen muuttujan x ilmentymä uudelleenkirjoitetaan ja tulokset kirjoitetaan pinoon. Kukin pinossa oleva kysely siis tarkoittaa TAI-konnektiivilla yhdistettyä tulosjoukon mahdollisuutta. Pinon tila:

```
Pino (2): {CLASS('Object') AND EQUALS('Object', 'Object')},
          {CLASS('String') AND EQUALS('String', 'Object')}
```

Nyt prosessi alkaa taas alusta. Logiikkakone lukee pinon päältä ensimmäisen kyselyn. Koska kyselyssä ei ole enää muuttujia, kutsutaan molempien kyselyssä esiintyvien propositioiden totuussääntöä. Nämä molemmat palauttavat toden, joten tämä x:n arvo päätyy tulosjoukkoon. Pinon tila nyt:

```
Pino (1): {CLASS('String') AND EQUALS('String', 'Object')}
```

Prosessi aloitetaan taas alusta. Ensimmäinen propositio suorittuu todeksi, mutta jälkimmäinen ei, joten tämä vaihtoehto (x = 'String') hylätään.

Päättelyprosessissa käytetään siis ajatusta, jonka mukaan proposition palauttaessa epätoden, kyseinen (osa-)kysely hylätään.

Vertailua

Emoolessa päättelysääntöjen sitomisessa suoritettavaan metodiin sovelletaan myöhäistä sidontaa kaikkien argumenttien suhteen, CLOS:in ja Cecilin tapaan. Tällä menettelyllä

saavutetaan mahdollisuus käyttää logiikkatulkkia myös sellaisten kohdealuemallien kanssa, joita ei ole alun perin suunniteltu Emoolen kanssa käytettäviksi, sillä mikäli kohdealuetta kuvaava oliomalli ei suoraan tue tulkin suoritussemantiikkaa, voidaan päättelyn vaatimat metodit lisätä oliomalliin ilman luokkahierarkiassa tapahtuvia muutoksia.

Erilaisten formaalien logiikkajärjestelmien toteutuksia on tehty lähes yhtä kauan kuin tietokoneita on ohjelmoitu. Eräs ensimmäisen kertaluokan logiikan toteutuksen kuvaus löytyy esimerkiksi Robinsonilta [Rob65]. Myös oliomaailman ja logiikkakielen välisiä siltaratkaisuja on esitetty aiemminkin. Esimerkiksi ALF-järjestelmä näyttää, miten Prologin ja Smalltalkin välinen silta voidaan muodostaa [Mel88].

ALF-järjestelmän perusteluissa annetaan kolme tavoitetta, jotka logiikka- ja olio-ohjelmoinnin yhdistävän sillan tulisi täyttää:

1. Logiikka- ja oliojärjestelmien tulisi käyttää samaa dataa siten, että mikä tahansa olio voisi olla propositio logiikkajärjestelmässä ja päinvastoin. Puolelta toiselle vaihdettaessa ei tulisi olla tarvetta minkäänlaiselle muunnokselle.

2. Jotta logiikkajärjestelmä integroituisi luontevasti oliojärjestelmään, olisi logiikkajärjestelmän toteutuksen oltava oliomainen.

3. Oliojärjestelmän luokkahierarkian tulisi toimia luontevasti logiikkajärjestelmän kanssa.

Emoole jakaa näistä tavoitteista kaksi ensimmäistä: jotta systeemi saisi minkäänlaista hyväksyntää olio-ohjelmoijien keskuudessa, tulee sen olla helposti upotettavissa olemassa oleviin olio-ohjelmiin. Erityisesti yksiperintää soveltavaa isäntäkieltä käytettäessä ALF:in kolmas tavoite on kuitenkin ristiriidassa tämän tavoitteen kanssa, sillä ALF:ssä propositiot (predikaatit) ovat luokan *Predicate* aliluokkia. Tämä estää käytännössä luontevan propositioiden toteuttamisen olemassa olevaan oliojärjestelmään, sillä muutos vaatisi joko nykyisen luokkahierarkian muokkaamisen logiikkajärjestelmän vaatimusten mukaiseksi tai omaa, erillistä siltaustehtäviin erikoistunutta lisäluokkarakennetta.

Tällainen kohdealuevaatimusten vuotaminen (engl. *leakage of concerns*) on ongelma, joka on vasta viime vuosina huomattu aspektiohjelmoinnin yhteydessä. Monesti tiettyä ohjelmiston toiminnan aspektia hoitavaa järjestelmää kehiteltäessä pääsee kyseisen yksityiskohdan hoitamiseen liittyvä yksityiskohta vuotamaan oliomallin puolelle. ALF:n tapauksessa tämä näkyy logiikkajärjestelmän vaatimana kantaluokkana Tämä ajattelumalli romahtaa viimeistään siinä vaiheessa, kun yritetään sovittaa kahta ristiriitaiset vaatimukset esittävää järjestelmää saman ohjelmiston yhteyteen.

Tätä ongelmaa ratkomaan on Javan keskuudessa noussut ns. POJO-ajattelutapa. Siinä sovellusaluemalli pidetään mahdollisimman puhtaana erilaisten lisäjärjestelmien luokkien rakenteelle asettamista vaatimuksista ja kirjoitetaan lisäjärjestelmät siten, että ne eivät aseta kohdealueelle ylimääräisiä vaatimuksia. Näin sovellusaluemalli voidaan kirjoittaa tavallisina Java- luokkina, *Plain Ordinary Java Objects* -tyyliin[4].

Emoolen lähestymistapa myötäilee POJO-ajattelua. Kohdealuetta ei yritetä saada täysin vapaaksi lisäjärjestelmän asettamista vaatimuksista, vaan Emoole sovittautuu käyttämään yleisesti tunnustetun pienimmän yhteisen tekijän, tavallisen Java-olion, rakennetta.

4.3 Uudelleenjärjestelevä optimointi Emoolella

Edellä esitetyn analyysikoneiston sovellettavuutta olio-ohjelmien optimointiin testattiin kokeellisesti toteuttamalla Java-ohjelman luokkarakennetta uudelleenjärjestelevä optimointi kahdella eri tavalla. Ensimmäisenä tapana kirjoitettiin analyysiohjelma perinteiseen Java-muotoon ja toisena tapana Emoolen logiikkakielellä.

Itse optimointina suoritetaan ylimääräiset rajapintaluokat poistava optimointi. Mikäli koko ohjelmasta löytyy vain yksi tietyn rajapinnan toteuttava luokka, voidaan kyseinen rajapintaluokka poistaa, ja kaikki sitä käyttävät kutsut ohjata osoittamaan suoraan tähän ainoaan rajapinnan toteuttavaan luokkaan [Poh04]. Tällaisia rakenteita esiintyy erityisesti ohjelmistokehyksiä käytettäessä, jolloin kehyksen erikoistamiskohdat, (engl. *hot spot*), on esitelty rajapintoina.

Metodi	Selitys
CLASS(JavaClass x)	Tosi, jos argumentti on ladattuna.
CLASS(Variable var)	Listaa kaikki ladatut luokat.
INTERFACE(JavaClass x)	Tosi, jos argumentti on rajapintaluokka.
IMPLEMENTS(JavaClass x, JavaClass y)	Tosi, jos luokka x toteuttaa rajapinnan y

Taulukko 4.1: Toteutettujen propositioiden kuormitetut suoritusmetodit

Käytettyjen kolmen proposition evaluointimetodit on kuormitettu taulukon 4.1 mukaisesti. Tämän luvun alussa olleeseen esimerkkiin erona on BCEL-kirjaston [Dah01] luokan *JavaClass* käyttäminen luokkien listaamiseen tavallisten merkkijonojen sijaan. Tämä on mahdollista, ja jopa suotavaa, sillä Emoolen päättelyalgoritmi ei sinällään aseta rajoituksia kyselyissä käytettävien luokkien suhteen.

[4] Myös muotoa Plain Old Java Objects käytetään.

Emoolella tarvittava analyysiohjelma kirjoitetaan näitä propositioita käyttäen kuvan 4.6 näyttämällä tavalla. Ohjelman suoritus tuottaa muuttujille x ja y arvoja, joissa luokka x toteuttaa rajapinnan y. Kutsuvassa koodissa tutkitaan, että löytyykö jollekin rajapinnalle y useampi kuin yksi toteuttava luokka, jolloin sitä ei voida poistaa optimoitavasta ohjelmasta. Javalla vastaavan toiminnallisuuden toteuttaminen vie liitteessä A.2 esitetyllä tavalla toteutettuna 98 riviä.

```
class    (x)      AND class (y) AND
interface(y)      AND
implements(x, y)
```

Kuva 4.6: Ohjelman rajapinnat ja toteuttavat luokat listaava logiikkaohjelma Emoolella

4.4 Suorituskykymittaukset

Optimointialgoritmin suorituskykyä voidaan arvioida kahdella mittarilla: toisaalta itse algoritmin suoritustehokkuuden kannalta ja toisaalta saavutettujen hyötyjen perusteella. Tässä aliluvussa tutkitaan aluksi rajapintoja yhdistävän optimointialgoritmin kahden erilaisen toteutuksen suoritusaikaa yleistä tapausta matkivalla synteettisellä syötteellä, ja seuraavaksi esitellään saavutettuja hyötyjä viidellä samasta sovelluskehyksestä erikoistetulla mobiiliohjelmalla.

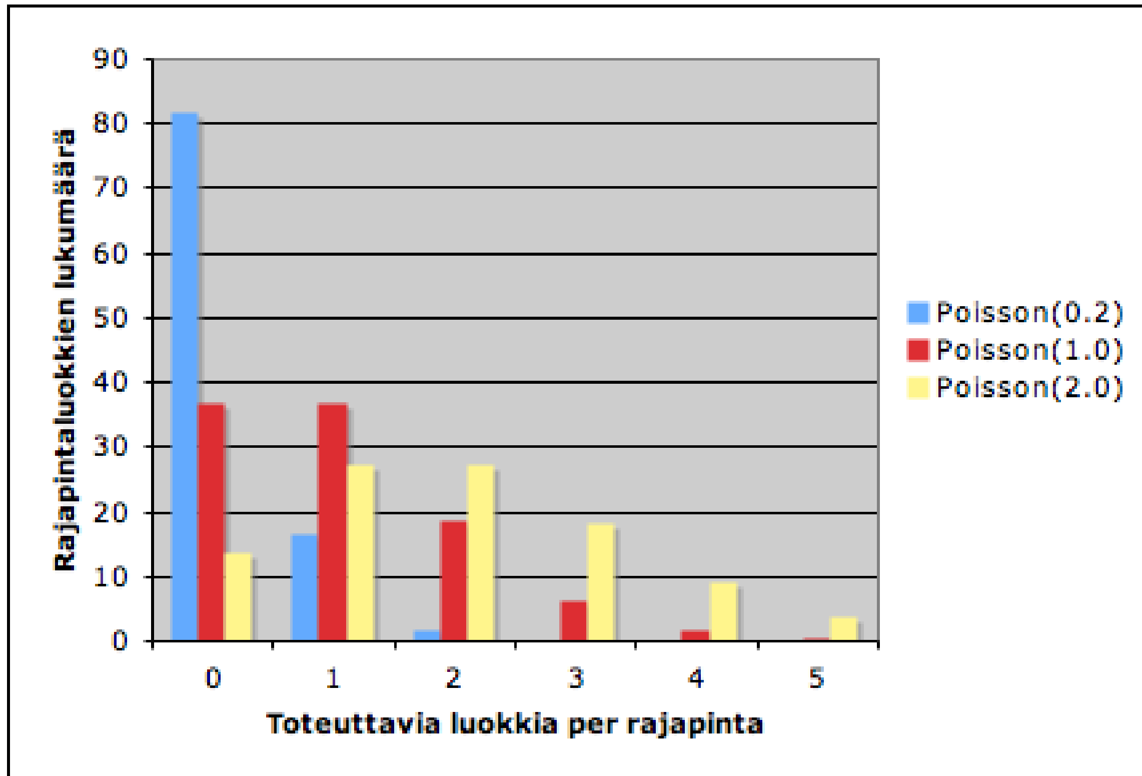
Suoritusaika

Päättelykoneiston avulla toteutetun ja suoraan Javalla kirjoitetun analyysialgoritmin suoritusaikaa verrattiin tuottamalla synteettisiä ohjelmia, jotka syötettiin kahdelle olemassa olevalle toteutukselle. Generoiduissa ohjelmissa luokkahierarkiassa esiintyvien rajapintaluokkien määrä vaihtelee kymmenen ja sadan välillä. Lisäksi niille generoitiin nollasta viiteen (kaavassa esiintyvä k) toteuttavaa luokkaa Poissonin kaavan

$$P(k) = \frac{\lambda^k}{k!}e^{-\lambda} \tag{4.1}$$

mukaan jakaantuneesti siten, että kaavan λ:ksi annettiin 0,2. Generointimenetelmän tarkoituksena on siis tuottaa satunnainen luokkahierarkia, jota voidaan käyttää algoritmin suorituskyvyn analysointiin. λ:n arvoa vaihtamalla voitaisiin generoida toisilla tavoin jakaantuneita luokkahierarkioita.

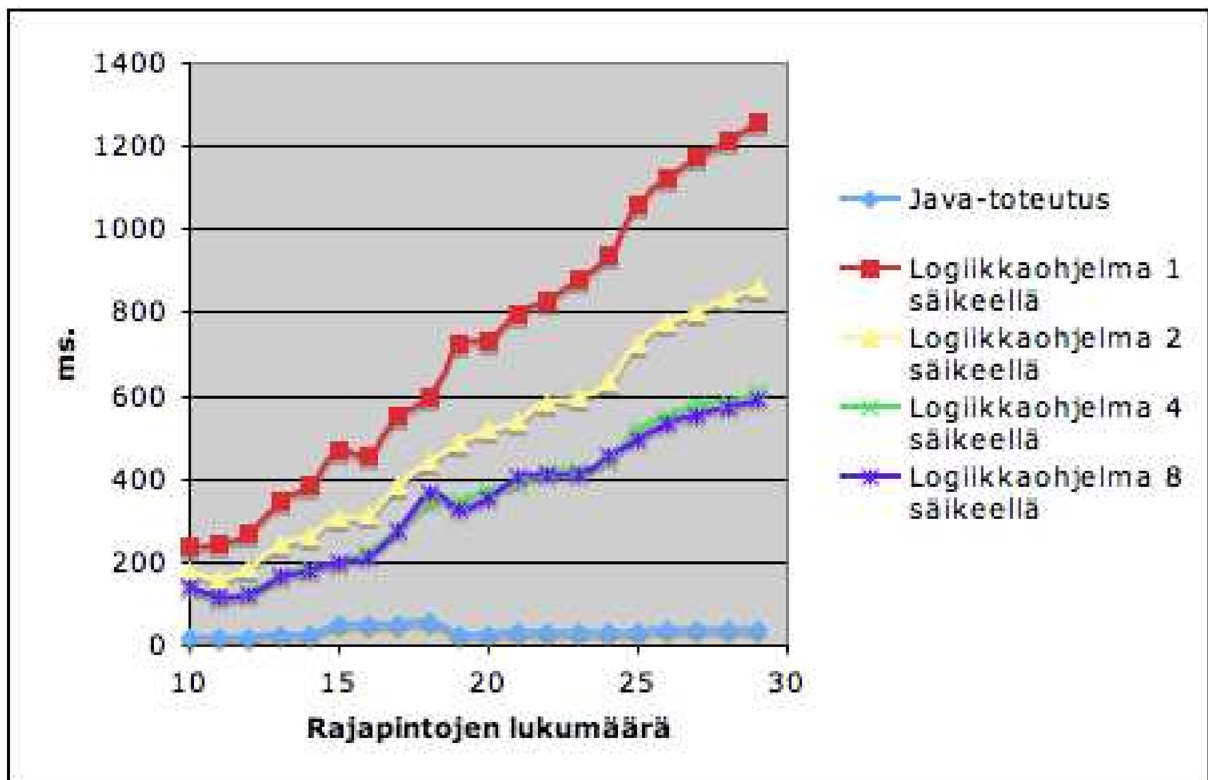
Esimerkki sadan rajapintaluokan ja niitä toteuttavien luokkien lukumääristä on kuvassa 4.7. Kuvasta nähdään, että kun $\lambda = 0,2$, niin sadasta rajapintaluokasta 81:lle ei generoitu lainkaan toteuttavia luokkia ja vain kahdella rajapintaluokalla ilmenee kaksi toteuttavaa luokkaa. Toisaalta, kun $\lambda = 2,0$, vain kolmellatoista rajapintaluokalla ei ole toteuttavia luokkia ja yli puolella rajapintaluokista on kaksi tai useampi toteuttava luokka. Koeasettelussa valittiin λ:n arvoksi mainittu $0,2$. Koeasetelma toistettiin myös muilla parametriarvoilla, jolloin tulokset olivat kuitenkin suoritusaikojen suhteiden osalta vastaavia.



Kuva 4.7: Koeasettelussa rajapintoja toteuttavien luokkien jakauma kun $\lambda = 0,2; 1,0; 2,0$

Koska logiikkakoneen kukin yksittäinen propositio tuottaa suoritusjärjestyksestä riippumatta konsistentteja vastauksia, voidaan päättelyprosessi helposti hajauttaa useammalle työskentelysäikeelle. Mikäli laitteistosta löytyy useampi prosessori, voi hajauttaminen selvästi nopeuttaa suoritusaikoja. Kuvassa 4.8 esitetään kuvan 4.6 ohjelman suoritusaikoja Poisson(0.2)-generoidulla luokkahierarkialla. Ohjelmat suoritettiin kullekin syötteelle kymmenen kertaa, ja tuloksista laskettiin keskiarvot. Sinisellä esitetään vertailukohtana olevan Java-ohjelman suoritusaikaa, ja punainen, keltainen, vihreä ja violetti näyttävät puolestaan suoritusaikoja suoritussäikeiden lukumäärän suhteen.

Tällä Emoolen kyselyllä käsin kirjoitettu toteutus vie suoritusnopeudessa selkeän voiton,



Kuva 4.8: Kysely class(x) and class(y) and interface(y) and implements(x, y)

sillä logiikkaohjelma on pseudokoodina algoritmin 4.1 mukainen.

Algoritmi on aikavaativuudeltaan selvästi eksponentiaalinen, sillä suorituksen aluksi kaikista luokista tehdään karteesinen tulo. Syötteen koon kasvaessa myös suoritusaika nousee nopeasti. Koe suoritettiin Helsingin yliopiston tietotekniikkaosaston neliprosessorisessa *kruuna*-palvelimessa, jolloin säikeiden määrän lisääminen nopeutti kokonaissuoritusaikaa aina neljään säikeeseen saakka.

Algoritmia voidaan parantaa kuormittamalla propositio *implements* seuraavasti:

```
IMPLEMENTS(JavaClass x, Variable interface)
```

Nyt propositiota *implements* käytetään listaamaan kaikki rajapinnat, jotka luokka x toteuttaa. Ensimmäisellä silmäyksellä saattaa näyttää siltä, että kaikki operaation monimutkaisuus piilotetaan kuormitetun metodin toteutukseen. Näin ei kuitenkaan ole, sillä sekä kaikkien luokkien listaaminen että tietyn luokan kaikkien rajapintojen listaaminen ovat itsessään hyvin yksinkertaisia operaatioita. Suurin monimutkaisuus operaatiossa on juurikin näiden kahden tiedon yhdistäminen.

```
Data    : Luokat on kaikkien ohjelmassa esiintyvien luokkien lista
foreach Luokka x ∈ Luokat do
    foreach Luokka y ∈ Luokat do
        if isInterface(y) then
            if x.implements(y) then
                list(y ← x);
            end
        end
    end
end
```

Algoritmi 4.1: Kuvan 4.6 ohjelma algoritmina

Implements-proposition kuormittamisen jälkeen algoritmia 4.1 vastaava Emoole-ohjelma yksinkertaistuu muotoon *class(x) and implements(x, y)*. Kuvassa 4.9 näytetään ohjelman suoritusaika ja vertailu vastaavaan Java-toteutukseen. Tällä kertaa hajautettuna suoritettu logiikkaohjelma voittaa suoritusnopeudessa muutamalla syötteellä, ja kokonaisuutenakaan ei ole pahasti tappiolla.

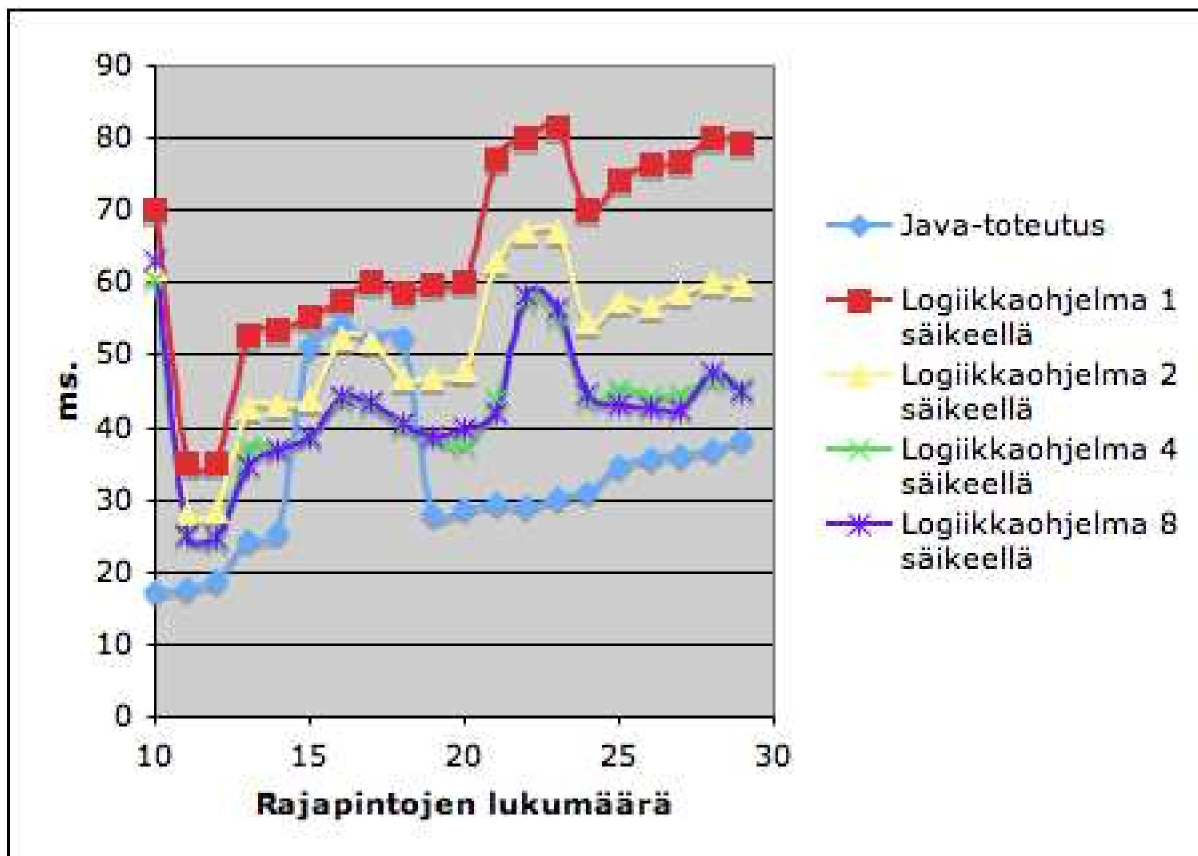
Yksinkertaistettu logiikkaohjelma on edelleenkin aikavaativuudeltaan eksponentiaalinen, sillä kullekin luokalle (lukumäärä m) listataan tuntematon määrä rajapintaluokkia (lukumäärä n). Käytännössä algoritmi on kuitenkin hyvin käyttäytyvä, sillä kunkin luokan toteuttamien rajapintojen lukumäärä on useimmiten pienempi kuin ohjelman kaikkien luokkien lukumäärä[5].

Parantelun tulokset ja päätelmiä

Itse rajapintoja yhdistävän algoritmin hyödyllisyyttä testattiin soveltamalla algoritmia viiteen kännykkäpelejä valmistavan Digital Chocolate- yhtiön kehittämän samasta sovelluskehyksestä johdettuun peliin. Koska käytetyssä kehyksessä hyödynnetään rajapintoja eristämään kännykkämallikohtaiset erot itse sovelluksen rakenteesta, saadaan käytetyllä algoritmilla kullekin mallille erikoistetusta ohjelmasta poistettua ylimääräisiä rajapintaluokkia automaattisesti.

Vastaava optimointi voitaisiin toki toteuttaa myös lähdekooditasolla, ohjelman esiprosessointivaiheena. Tällöin kuitenkin sitouduttaisiin käytettyyn lähdekieleen, eli tässä tapauksessa Javaan. Kun optimointi toteutetaan välikielen tasolla, voidaan ohjelmistoprosessin muissa vaiheissa hyödyntää sopivaksi katsottuja menetelmiä: mahdollisten lähdekielten

[5] Esimerkin kannalta patologisia, pelkistä rajapintaluokista koostuvia ohjelmia voidaan toki generoida



Kuva 4.9: Kysely class(x) and implements(x, y)

joukko laajenee kaikkiin niihin kieliin, joille löytyy Javan tavukoodia generoiva kääntäjä. Lisäksi välikieltä prosessoiva optimoija on paljon yksinkertaisempi toteuttaa kuin kokonaisuudessaan lähdekieltä ymmärtävä ja muokkaava optimoija.

Javalla kirjoitetuissa kännykkäsovelluksissa usein ensimmäinen vastaan tuleva rajoite on ohjelman koko, sillä monet kännykkämallit rajoittavat asennettavien ohjelmien kokoa jopa tarpeettoman paljon. Ohjelmat myös usein siirretään suoritusalustaan langattoman verkon yli, jolloin koko on ohjelman asennuksessa määräävä ajankäytön tekijä. Tämän vuoksi ohjelman staattisen koon mittaaminen on olennaisin elementti kännykkäohjelmia optimoidessa.

Optimoinnin tulokset näkyvät taulukossa 4.2.

Prosentuaalisesti rajapintaluokkia poistamalla ei päästä kovinkaan merkittäviin tilansäästölukuihin. On kuitenkin huomattava, että kyseessä on Javassa aiemmin käyttämätön optimointi, jolloin esitettyä algoritmia voidaan käyttää muiden optimointimenetelmien ohella.

Toiselta suunnalta saavutettuja hyötyjä voidaan tarkastella ajattelemalla vaihtoehtoishyö-

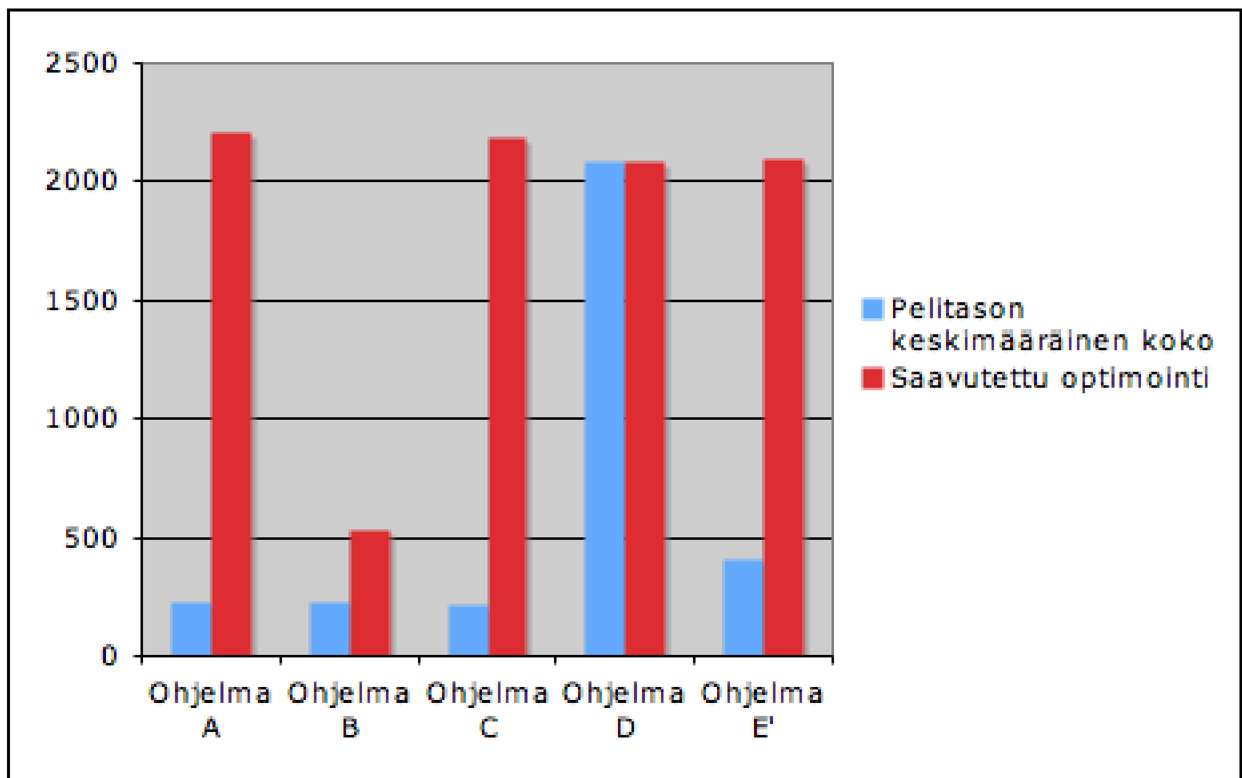
Ohjelma	Optimoimaton koko	Optimoitu koko	Tilansäästö	Prosenteissa
Ohjelma A	71,719 tavua	70,507 tavua	2,212 tavua	3,04%
Ohjelma B	68,305 tavua	67,772 tavua	533 tavua	0,78%
Ohjelma C	75,883 tavua	73,695 tavua	2,188 tavua	2,88%
Ohjelma D	55,799 tavua	53,711 tavua	2,088 tavua	3,74%
Ohjelma E	62,683 tavua	60,583 tavua	2,100 tavua	3,35%

Taulukko 4.2: Kokeelliset tulokset

tyä, eli mihin muuhun säästetty tila voidaan käyttää. Neljä viidestä kokeeseen osallistuneesta ohjelmasta on tasopohjaisia pelejä, joissa kaikki sovelluslogiikalta ylitse jäävä tila käytetään pelin tasojen tallentamiseen. Mitä enemmän tilaa on käytettäväksi, sitä laajempia ja monipuolisempia tasoja peliin voidaan sisällyttää. Tässä vertailussa sovellettu algoritmi pärjää hyvin: peleissä keskimääräinen tason koko vaihtelee 215 ja 400 tavun välillä, jolloin kahden kilotavun säästöllä peliin saadaan viidestä kymmeneen uutta tasoa. Tätä suhdetta esitetään kuvassa 4.10. Tämä ei kuitenkaan ole yleispätevä mittari, sillä esimerkiksi kokeen tapauksessa D ei tasoja käytetä lainkaan.

Koska aikaisemmin ei ole ollut mahdollista turvallisesti poistaa rajapintaluokkia ohjelmasta, ovat mobiilisovellusten suunnittelijat vältelleet turhaksi kokemiensa abstraktioiden käyttämistä. Kuitenkin jo hyvin yksinkertaisella analyysillä voidaan ohjelmistokehyksen erikoistamisrajapintaan kuuluvat metodit tunnistaa ja kokoonpanovaiheessa poistaa. Oletettavasti ohjelman luokkamäärää pienentämällä saavutetaan pieniä yleishyötyjä monissa suorituksen vaiheissa, sillä pienempi määrä luokkia päätyy ladattavaksi, ja esimerkiksi kukin suoraan luokan toteutukseen viittaava metodikutsu on tavukooditasolla kaksi tavua lyhyempi kuin rajapintatoteutuksen kautta tapahtuva.

Erityisen logiikkakielen käyttäminen optimointien kuvaamiseen on mielenkiintoinen lähestymistapa optimointeihin liittyvän monimutkaisuuden hallitsemiseksi. Perinteisestä poikkeava metodien sidontatapa mahdollistaa luontevan logiikan komponenttimetodien kirjoittamisen. Nykyinen toteutus on kuitenkin ominaisuuksiltaan lähempänä teknologiademoa kuin yleispätevää ohjelmointijärjestelmää. Esimerkin perusteella on kuitenkin todettava, että ensimmäisen kertaluokan logiikkaan rajoittuva kieli on ilmaisuvoimaltaan kovin rajoittunut tavoitteisiin nähden. Siinä missä tavoitteena olisi saada kääntäjän optimointisääntöjä kirjoitettua deklaratiivisesti, onnistuttiin logiikkakielellä hakemaan luokkahierarkiasta kaikki rajapinnat ja ne toteuttavat luokat, mutta esimerkiksi toteuttavien luokkien lukumäärän laskeminen jouduttiin edelleenkin toteuttamaan perinteisesti Javalla. Kokonaiskuvan kannalta käytetty logiikkakieli tuntuu toistaiseksi vain sekoittavan asiaa.



Kuva 4.10: Optimoinnin tuottama tilansäästö suhteessa kunkin sovelluksen keskimääräiseen tason kokoon

Toisaalta järjestelmän suunnittelun yhteydessä tehty sovellusalueen mallintaminen ei varmastikaan ole mennyt hukkaan. Etsiessään ja toteuttaessaan logiikkajärjestelmän propositioita ohjelmoija joutuu miettimään sovellusalueen abstraktioita ja käsitteitä. Tämän vuoksi lähestymistapaa ei kannattane täysin hylätä, vaikkakin logiikkakielellä onnistuttiinkin kuvaamaan vain pieni osa kokonaissovelluksen toiminnasta. Mielenkiintoinen jatkokehityssuunta voisi olla esimerkiksi modaalilogiikan primitiivien yhdistäminen erilaisten luvussa 3 esitettyjen analyysimenetelmien tuottamaan tietoon.

Nopeudessa logiikkakielen toteutus pärjäsi yllättävän hyvin, sillä alkuperäisesti kielen suorituksen oletettiin olevan kahta tai kolmea kertaluokkaa Java-toteutusta hitaampaa. Osasyynä hyvään suorituskykyyn lienee kuitenkin optimoidun kyselyn yksinkertaisuus. Tämän vuoksi esiteltyä logiikkakonetta ja sen mahdollistamaa logiikkakyselyiden integroimista oliokieleen kannattanee kuitenkin tutkia tarkemmin muissa yhteyksissä.

5 Yhteenveto

Kääntäjän tuottaman kohdekoodin optimoiminen nopeuden, tilankäytön tai muun mitattavan suureen suhteen on ohjelmistojen tuotannossa tärkeä vaihe. Ohjelmien nopea suoritus on joissakin tapauksissa käyttäjän implisiittinen toive, mutta monesti ohjelmia joudutaan optimoimaan esimerkiksi sovitettaessa ohjelmaa toimimaan suoritusalustan tai sovellusympäristön asettamien rajoitteiden puitteissa. Koska käsin tapahtuva ohjelman matalan tason virittely ei ole kustannustehokasta, on kääntäjiin ohjelmoitu menetelmiä optimoinnin automatisointiin.

Matemaattisessa mielessä sana 'optimointi' tarkoittaa parhaimman mahdollisen muodon löytämistä. Ohjelmia käännettäessä tämä on kuitenkin mahdoton lopputulos. Tämän vuoksi kääntäjän suorittamilla optimoinneilla tarkoitetaan niitä toimenpiteitä, joilla tuotettua ohjelmakoodia parannellaan siten, että se vastaa vähintään hyvän ihmisohjelmoijan tuottamaa koodia.

Tutkielmassa käsiteltiin oliokielten välikielten tasolla tapahtuvaa optimointia käyttäen Javan tavukoodia esimerkkinä ja keskittyen pääasiassa staattisella tasolla tapahtuviin paranteluihin. Nykyisten oliokielten dynaamisuuden vuoksi puhtaasti staattiselle muuntelulle ei ole suuriakaan mahdollisuuksia, sillä optimoivan muunnoksen tulee aina säilyttää alkuperäisen ohjelman semantiikka. Tämän vuoksi suurin osa nykyisestä oliokielten optimointitutkimuksesta keskittyy suoritusaikaiseen, dynaamiseen optimointiin (Just-in-Time). Java-ympäristössä on kuitenkin myös staattiselle parantelulle mahdollisuuksia, pääasiassa kännykkäkäyttöön tarkoitetuissa ohjelmissa, joissa suljetun maailman oletus pätee. Kohdealueen haasteena ovat myös vähäiset käytettävissä olevat laiteresurssit, joten pienetkin parantelut ovat tervetulleita.

Tutkielman yhteydessä toteutettiin luokkahierarkiaa uudelleenjärjestelevä optimointi, jonka todettiin viiden esimerkkipeliohjelman tapauksessa pienentävän ohjelman kokoa keskimäärin 2,75%. Säästyneeseen tilaan voitiin esimerkkitapauksissa pakata noin kymmenen pelimaailman kenttää, kun tavallisessa tapauksessa pelin mukana toimitetaan noin kolmekymmentä kenttää. Prosentuaalisesti pieni tilansäästö näyttelee siten kuitenkin sovelluksen sisällön kannalta huomattavasti suurempaa roolia.

Teollisuuskäyttöön otetun parantelualgoritmin lisäksi tutkielmassa käsiteltiin mahdollisuutta määritellä ja suorittaa parantelualgoritmeja logiikan tasolla. Edellä mainitun parantelun esiehdot onnistuttiin kuvaamaan ensimmäisen kertaluokan logiikan kaavalla, joka syötettiin tutkielman puitteissa toteutetulle logiikkakoneelle. Vertailtaessa esiehtojen tarkastusalgoritmin Java- ja logiikkatoteutuksia huomattiin, että logiikkakaavaan perustu-

va tarkastus pystyy pärjäämään suoritusnopeudessa Javalla toteutetulle ohjelmalle. Siinä missä Javalla toteutettu algoritmi suorittuu sarjana peräkkäisiä käskyjä voidaan logiikkakaavaan perustuva tarkastus hajauttaa vaivatta useammalle prosessorille, ja siten hyötyä suoritusajassa.

Työn puitteissa toteutettu logiikkakonetta on mahdollista jatkokehittää moneen suuntaan: esimerkiksi ensimmäisen kertaluokan logiikkaa vahvempien logiikkajärjestelmien ratkaisimeksi. Toinen mielenkiintoinen kehityssuunta olisi tutkia, kuinka ratkaisualgoritmi saataisiin hajautettua useammassa Java- virtuaalikoneessa suoritettavaksi. Koneen käyttöaluetta ei ole mitenkään erityisesti rajattu optimointitehtävissä käytettäväksi, joten sitä voidaan soveltaa myös muissa yhteyksissä. Tällaisenaan logiikkakone on kelvollinen apuväline mm. optimointitehtävissä esiintyvien kombinointialgoritmien peruspalasten mallintamiseen ja abstrahoimiseen sekä toimiva teknologiademo epäperinteisen viestinvälitysmekanismin hyödyntämisestä.

Lähteet

AGT89 Aho, A. V., Ganapathi, M. ja Tjiang, S. W. K., Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11,4(1989), sivut 491–516.

AH96 Aigner, G. ja Hölzle, U., Eliminating virtual function calls in C++ programs. *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, osa 1098 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 1996, sivut 142–166, URL `citeseer.ist.psu.edu/aigner96eliminating.html`.

AIS+77 Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. ja Angel, S., *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

AK02 Allen, R. ja Kennedy, K., *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

Ale87 Alexander, C., *The Timeless Way of Building*. Oxford University Press, 1987.

All70 Allen, F. E., Control flow analysis. *ACM SIGPLAN Notices*, 5,7(1970), sivut 1–19.

App98 Appel, A. W., *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

ASU86 Aho, A., Sethi, R. ja Ullman, J. D., *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

BC90 Bracha, G. ja Cook, W., Mixin-based inheritance. *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, Meyrowitz, N., toimittaja, Ottawa, Canada, 1990, ACM Press, sivut 303–311, URL `citeseer.ist.psu.edu/bracha90mixinbased.html`.

BDB00 Bala, V., Duesterwald, E. ja Banerjia, S., Dynamo: a transparent dynamic optimization system. *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*.

ACM Press, 2000, sivut 1–12, URL `http://dx.doi.org/10.1145/349299.349303`.

Bel73 Bell, J. R., Threaded code. *Communications of the ACM*, 16,6(1973), sivut 370–372.

Bla99 Blanchet, B., Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34,10(1999), sivut 20–34. URL `citeseer.nj.nec.com/blanchet99escape.html`.

Bro75 Brooks, Jr., F. P., The mythical man-month. *ACM SIGPLAN Notices*, 10,6(1975).

Bro95 Brooks, Jr., F. P., *The Mythical Man-Month, 20th Anniversary Edition*. Addison-Wesley, 1995.

Cha92 Chambers, C., Object-oriented multi-methods in Cecil. *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, Madsen, O. L., toimittaja, osa 615 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 1992, sivut 33–57, URL `citeseer.ist.psu.edu/174216.html`.

CMH91 Chang, P. P., Mahlke, S. A. ja Hwu, W. W., Using profile information to assist classic code optimizations. *Software – Practice and Experience*, 21,12(1991), sivut 1301–1321. URL `citeseer.ist.psu.edu/chang91using.html`.

Con58 Conway, M. E., Proposal for an UNCOL. *Communications of the ACM*, 1,10(1958), sivut 5–8.

CPL83 Conroy, T. J. ja Pelegri-Llopart, E., An assessment of method-lookup caches for Smalltalk-80 implementations. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983, sivut 239–247.

Dah01 Dahm, M., Byte code engineering with the BCEL API, 2001. URL `citeseer.ist.psu.edu/dahm01byte.html`. http://citeseer.ist.psu.edu/dahm01byte.html.

DGC95 Dean, J., Grove, D. ja Chambers, C., Optimization of object-oriented programs using static class hierarchy analysis. *ECOOP '95: Proceedings of the*

9th European Conference on Object-Oriented Programming, osa 952 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 1995, sivut 77–101, URL citeseer.ist.psu.edu/dean94optimization.html.

DLS+01 Dutchyn, C., Lu, P., Szafron, D., Bromling, S. ja Holst, W., Multi-dispatch in the Java virtual machine: Design and implementation. *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, San Antonio, Texas, 2001, sivut 77–92.

DS84 Deutsch, L. P. ja Schiffman, A. M., Efficient implementation of the Smalltalk-80 system. *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah, 1984, sivut 297–302, URL citeseer.ist.psu.edu/deutsch84efficient.html.

Ert95 Ertl, M. A., Stack caching for interpreters. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, New York, NY, USA, 1995, ACM Press, sivut 315–327.

Fer95 Fernández, M. F., Simple and effective link-time optimization of Modula-3 programs. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. ACM Press, 1995, sivut 103–115.

GDGC95 Grove, D., Dean, J., Garrett, C. ja Chambers, C., Profile-guided receiver class prediction. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Austin, Texas, 1995, sivut 108–123, URL citeseer.ist.psu.edu/grove95profileguided.html. ACM SIGPLAN Notices 30, 10 (Oct.).

GG83 Griswold, R. E. ja Griswold, M. T., *ICON Programming Language*. Prentice Hall PTR, 1983.

GHJV93 Gamma, E., Helm, R., Johnson, R. E. ja Vlissides, J. M., Design patterns: Abstraction and reuse of object-oriented design. *Proceedings of the 7th European Conference on Object-Oriented Programming*, Nierstrasz, O., toimittaja, osa 707 sarjasta *Lecture Notes in Computer Science*. Springer, 1993, sivut 406–431.

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

Gos95 Gosling, J., Java intermediate bytecodes. *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. ACM Press, 1995, sivut 111–118.

Gou02 Gough, K. J., *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall PTR, 2002.

GR83 Goldberg, A. ja Robson, D., *Smalltalk-80: the Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

Gro03 Grothoff, C., Walkabout revisited: The runabout. *ECOOP 2003 - Object-Oriented Programming*, Cardelli, L., toimittaja, osa 2743 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 2003, sivut 103–125, URL `http://www.ovmj.org/runabout/runabout.ps`.

GS00 Gay, D. ja Steensgaard, B., Fast escape analysis and stack allocation for object-based programs. *9th International Conference on Compiler Construction (CC'2000)*, osa 1781 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 2000, URL `citeseer.ist.psu.edu/gay00fast.html`.

Ham03 Hamilton, J., Language integration in the Common Language Runtime. *ACM SIGPLAN Notices*, 38,2(2003), sivut 19–28.

HCU91 Hölzle, U., Chambers, C. ja Ungar, D., Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, osa 512 sarjasta *Lecture Notes in Computer Science*, London, UK, 1991, Springer-Verlag, sivut 21–38, URL `citeseer.ist.psu.edu/hlzle91optimizing.html`.

HCU92 Hölzle, U., Chambers, C. ja Ungar, D., Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27,7(1992), sivut 32–43.

HH97 Hookway, R. J. ja Herdeg, M., Digital fx!32: Combining emulation and binary translation. *Digital Technical Journal*, 9,1(1997), sivut 3–12.

HU94 Hölzle, U. ja Ungar, D., Optimizing dynamically-dispatched calls with run-time type feedback. *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, New York, NY, USA, 1994, ACM Press, sivut 326–336.

Höl94 Hölzle, U., *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. Väitöskirja, 1994. URL `citeseer.nj.nec.com/hlzle94adaptive.html`.

IKY+00 Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. ja Nakatani, T., A study of devirtualization techniques for a Java just-in-time compiler. *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2000, sivut 294–310.

ISO03 ISO/IEC, ISO/IEC 23271:2003 Common Language Infrastructure (CLI), 2003.

Joh79 Johnson, S. C., Yacc: Yet another compiler compiler. Teoksessa *UNIX Programmer's Manual*, osa 2, Holt, Rinehart, and Winston, 1979, sivut 353–387, URL `citeseer.ist.psu.edu/johnson79yacc.html`.

Ken81 Kennedy, K. *Program Flow Analysis: Theory and Applications*, luku 1. A Survey of Data Flow Analysis Techniques. Prentice Hall, 1981.

KH02 Krall, A. ja Horspool, R. N., Optimizations for object-oriented languages. Teoksessa *The Compiler Design Handbook*, CRC Press, 2002, sivut 219–246.

Knu71 Knuth, D. E., An empirical study of FORTRAN programs. *Software — Practice and Experience*, 1,2(1971), sivut 105–133.

Koo94 Koopman, P., A preliminary exploration of optimized stack code generation. *Journal of Forth Application and Research*, 6,3(1994), sivut 241–251. URL `citeseer.ist.psu.edu/koopman92preliminary.html`.

KV92 Koskimies, K. ja Vihavainen, J., The problem of unexpected subclasses. *Journal of Object-Oriented Programming*, 6(1992), sivut 53–59.

Lin76 Linden, T. A., The use of abstract data types to simplify program modifications. *Proceedings of the 1976 conference on Data : Abstraction, definition and structure*. ACM Press, 1976, sivut 12–23.

LS Lesk, M. E. ja Schmidt, E., Lex – a Lexical Analyzer Generator. Tekninen raportti 39.

LUW99 Lance, D., Untch, R. H. ja Wahl, N. J., Bytecode-based java program analysis. *ACM-SE 37: Proceedings of the 37th Annual Southeast Regional Conference (CD-ROM)*, New York, NY, USA, 1999, ACM Press, sivu 14.

LY99 Lindholm, T. ja Yellin, F., *The Java Virtual Machine Specification, Java 2 Platform*. Addison Wesley, 1999.

McK65 McKeeman, W. M., Peephole optimization. *Communications of the ACM*, 8,7(1965), sivut 443–444.

ME98 Maierhofer, M. ja Ertl, M. A., Local stack allocation. *Compiler Construction*, Koskimies, K., toimittaja, osa 1383 sarjasta *Lecture Notes in Computer Science*. Springer, 1998, sivut 189–203.

Mel88 Mellender, F., An integration of logic and object-oriented programming. *SIGPLAN Notices*, 23,10(1988), sivut 181–185.

Mey92 Meyer, B., *Eiffel: The Language*. Prentice-Hall, 1992.

Moo65 Moore, G. E., Cramming more components onto integrated circuits. *Electronics*, 38,8(1965), sivut 114–117.

MS98 Mikhajlov, L. ja Sekerinski, E., A study of the fragile base class problem. *Proceedings of the 12th European Conference on Object-Oriented Programming*, osa 1445 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 1998, sivut 355–382.

Muc97 Muchnick, S. S., *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

ND78 Nygaard, K. ja Dahl, O.-J., The development of the SIMULA languages. *The First ACM SIGPLAN Conference on History of Programming Languages*, Los Angeles, CA, 1978, ACM Press, sivut 245–272.

Nii80 Niiniluoto, I., *Johdatus tieteenfilosofiaan: Käsitteen- ja teorianmuodostus*. Otava, 1980.

Nov03 Novillo, D., Tree SSA – a new optimization infrastructure for GCC. *Proceedings of the 2003 GCC Developers' Summit*, Ottawa, Canada, May 2003, sivut 181–193.

Par72 Parnas, D. L., On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15,12(1972), sivut 1053–1058.

Poh04 Pohjalainen, P., Restructuring optimisations for object-oriented mobile applications. *Proceedings of the HiPC Workshops 2004 (CD-ROM)*, Bangalore, India, December 2004.

Pre95 Pree, W., *Design Patterns for Object-Oriented Software Development*. ACM Press/Addison-Wesley Publishing Co., 1995.

Pro95 Proebsting, T. A., Optimizing an ansi c interpreter with superoperators. *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 1995, ACM Press, sivut 322–332.

PS82 Papadimitriou, C. H. ja Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

PS94 Palsberg, J. ja Schwartzbach, M. I., *Object-oriented type systems*. John Wiley and Sons, 1994.

Rau78 Rau, B. R., Levels of representation of programs and the architecture of universal host machines. *MICRO 11: Proceedings of the 11th Annual Workshop on Microprogramming*, Piscataway, NJ, USA, 1978, IEEE Press, sivut 67–79.

Ric53 Rice, H. G., Classes of recursively enumerable sets and their decision problems. *Transactions of the Americal Mathematical Society*, 83(1953), sivut 25–29.

Ric71 Richards, M., The portability of the BCPL compiler. *Software – Practice and Experience*, 1,2(1971), sivut 135–146.

Rob65 Robinson, J. A., A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12,1(1965), sivut 23–41.

RTV01 Riggs, R., Taivalsaari, A. ja Vandenbrink, M., *Programming for Wireless Devices with the Java 2 Platform, Micro Edition*. Addison-Wesley, 2001.

Sch94 Schwartzbach, M., Developments in object-oriented type systems, January 1994. URL `citeseer.ist.psu.edu/schwartzbach94developments.html`.

Sco00 Scott, M. L., *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

Seb99 Sebesta, R. W., *Concepts of Programming Languages*. Addison-Wesley Longman, neljäs painos, 1999.

Sin03 Singer, J., JVM versus CLR: a comparative study. *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*. Computer Science Press, 2003, sivut 167–169.

Sta99 Stallman, R. M., *Using and Porting the GNU Compiler Collection, For GCC Version 2.95*. Free Software Foundation, Cambridge, USA, 1999. URL `citeseer.ist.psu.edu/article/stallman01using.html`.

Sta04 Stallman, R. M., *GNU Compiler Collection Internals*. Free Software Foundation, Boston, USA, 2004. http://gcc.gnu.org/onlinedocs/gccint.

Ste84 Steele, G. L., *Common LISP: The Language*. Digital Press, Newton, MA, USA, 1984.

Str88 Stroustrup, B., What is object-oriented programming? *IEEE Software*, 5,3(1988), sivut 10–20.

Str94 Stroustrup, B., *The Design and Evolution of C++*. Addison-Wesley, 1994.

SWT+58 Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O. ja Steel, T., The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, 1,8(1958), sivut 12–18.

Sys00 Systä, T., *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Väitöskirja, Tampereen yliopisto, 2000. URL `http://acta.uta.fi/pdf/951-44-4811-1.pdf`.

Tai92 Taivalsaari, A., Kevo – a prototype-based object-oriented language based on concatenation and module operations. Tekninen raportti TR-LACIR 92-02, University of Victoria, 1992.

Tai93a Taivalsaari, A., Concatenation-based object-oriented programming in Kevo. *Actes de la 2eme Conference sur la Representations Par Objets RPO'93*. EC2, France, June 1993, sivut 117–130.

Tai93b Taivalsaari, A., *A Critical View of Inheritance and Reusability in Object-oriented Programming*. Väitöskirja, University of Jyväskylä, 1993.

Tai96 Taivalsaari, A., On the notion of inheritance. *ACM Computing Surveys*, 28,3(1996), sivut 438–479.

US87 Ungar, D. ja Smith, R. B., Self: The power of simplicity. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 1987, sivut 227–242.

VKB01 Veldema, R., Kielmann, T. ja Bal, H. E., Optimizing Java-specific overheads: Java at the speed of C? *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, osa 2110 sarjasta *Lectures Notes in Computer Science*, London, UK, 2001, Springer-Verlag, sivut 685–692.

Weg87 Wegner, P., Dimensions of object-based language design. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 1987, sivut 168–182.

Wir71 Wirth, N., The design of a PASCAL compiler. *Software – Practice and Experience*, 1,4(1971), sivut 309–333.

WS94 Whitfield, D. ja Soffa, M. L., The design and implementation of Genesis. *Software — Practice and Experience*, 24,3(1994), sivut 307–325.

WS97 Whitfield, D. L. ja Soffa, M. L., An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19,6(1997), sivut 1053–1084. URL `citeseer.nj.nec.com/whitfield97approach.html`.

A Liitteet

A.1 Luokkahierarkian muodostava algoritmi Javalla

Liitteessä esitetään luokkahierarkian rajapintojen osalta muodostava algoritmi. Toteutus hyödyntää BCEL-luokkakirjastoa [Dah01].

```
public void listInterfaces(Map memory) {
    // allLoadedClasses defined elsewhere,
    // lists all classes in the program
    JavaClass[] classList = allLoadedClasses();

    for(int i=0; i<classList.length; i++) {
        if( classList[i].isClass() ) {
            JavaClass[] interfaces;
            interfaces = classList[i].getAllInterfaces();

            for(int j=0; j<interfaces.length; j++) {
                listClassToInterface(memory,
                                     interfaces[j],
                                     classList[i]);
            }
        }
    }
}

private void listClassToInterface(Map memory,
                                  JavaClass iface,
                                  JavaClass clazz) {
    Object ob = memory.get( iface.getClassName() );

    if(ob == null) {
        List list = new ArrayList();
        list.add( clazz.getClassName() );
        memory.put(iface.getClassName(), list);
    } else if(ob instanceof List) {
        List list = (List)ob;
```

```
        if( !list.contains(clazz.getClassName()) ) {
            list.add( clazz.getClassName() );
        }
    } else {
        throw new RuntimeException("Logic error");
    }

}
```

A.2 Poistettavat rajapintaluokat listaava Java-ohjelma

Algoritmi käyttää liitteessä A.1 esitettyjä metodeja hyväkseen.

```
/*
 * Analysis part -
 *  traverse over the class hierarchy in order to find
 *  interfaces that have only one implementing class
 */
public void analyseClassHierarchy()
{
    /*
     * interfaceMap = ([ "interfaceName":
     *                       ({ "implementinClass1", .. }),
     *                 ]);
     */
    Map interfaceMap = new java.util.HashMap();
    listInterfaces(interfaceMap);

    /*
     * if(interfaceMap["ifName"].length() == 1) {
     *   mergeableMap["ifName", 0] = interfaceMap["ifName"][0]
     * }
     *
     */
    this.mergeableMap = new java.util.HashMap();
    findInterfacesWithSingleImplementation(interfaceMap);
}

public void findInterfacesWithSingleImplementation(Map memory) {
    boolean printedSometing = false;
    List    loadedClasses   = Arrays.asList(allLoadedClasses());

    for(Iterator iter = memory.keySet().iterator(); iter.hasNext(); )
    {
        String key    = iter.next().toString();
        Object value  = memory.get(key);
```

```
            if(value instanceof List) {
                List      list      = (List)value;
                JavaClass interFace = Repository.lookupClass(key);

                if(list.size() == 1 &&
                   loadedClasses.contains(interFace))
                {
                    System.out.println("- " + key + " is mergeable");
                    printedSometing = true;
                    mergeableMap.put(key, list.get(0));
                }
            } else {
                throw new RuntimeException("Logic error");
            }
        }

        if(!printedSometing) {
            System.out.println("No mergeable interfaces found.\n");
        }
    }
```