Antti-Juhani Kaijanaho

Muistinhallinta siivousmenetelmien avulla

Tietotekniikan (ohjelmistotekniikka)
LuK-tutkielma
5.10.2001

Jyväskylän yliopisto

Tietotekniikan laitos

Tiivistelmä

Tämä LuK-tutkielma käsittelee muistinhallintaa siivousmenetelmien näkökulmasta. Tutkielmassa tarkastellaan muistinsiivouksen menetelmiä erityisen muistinhallinnan abstraktin mallin antamassa kontekstissa.

Title in English: Memory management using garbage collection

Avainsanat: Muistinsiivous, muistinhallinta, muistinhallinnan abstrakti malli, merkkaa ja lakaise, pysäytä ja kopioi, ikäperustainen siivous, vähittäinen siivous, osoittimien laputus

Keywords: Garbage collection, memory management, abstract model of memory management, mark and sweep, stop and copy, generational collection, incremental collection, pointer tagging

Tekijän yhteystiedot:
Antti-Juhani Kaijanaho
Helokantie 1 A 16
40640 JYVÄSKYLÄ
sähköposti: gaia@iki.fi

Copyright © 2001 Antti-Juhani Kaijanaho
Tekijä antaa kaikille luvan tehdä tästä tutkielmasta sisällöltään muuttamattomia kappaleita ja saattaa ne haluamallaan tavalla yleisön saataville. Muunlaisesta käytöstä tulee neuvotella tekijän kanssa erikseen.

Sisältö

1 Johdanto

Tämä tutkielma käsittelee *muistinsiivousta*[1] (engl. *garbage collection*). Siivoustekniikoita käytetään yleisesti muistinhallintaan lähes kaikissa funktionaalisissa, logiikkapohjaisissa ja oliopohjaisissa kielissä muistinhallinnan menetelmänä. Tunnetuin esimerkki on epäilemättä Java [8], mutta siivous on alunperin peräisin funktionaalisesta ohjelmointikielestä nimeltään LISP [11].

Muistinsiivous on myös ohjelmistotuotannon kannalta tärkeä. Manuaalinen muistinhallinta on virhealtista ja joko kytkee modulit liian läheisesti yhteen tai pakottaa modulin käyttäjän tekemään datasta turhia kopioita. Muistinsiivouksen käyttäminen poistaa muistivuodon ja roikkuvien osoittimien ongelmat, jotka ovat yleisimmät virhelähteet ohjelmia kirjoitettaessa. [9]

Tämä tutkielma jakautuu kolmeen osaan. Luvut 2 ja 3 esittelevät tutkielmassa käytettävän muistinhallinnan mallin ja tutkielmassa käsiteltävän ongelman. Luvut 5 ja 7 käsittelevät muistinsiivouksen perustekniikat. Lopuksi luku 6 esittelee kursorisesti nykyaikaisen muistinsiivouksen algoritmityypit.

2 Muistinhallinnan abstrakti malli

Tämä luku esittelee tässä tutkielmassa käytettävän muistinhallinnan abstraktin mallin. Vaikka monet sen yksityiskohdat (mm. muistin jakaminen muuttujiin) kuuluvat alan yleistietoon ja jotkin muut yksityiskohdat (mm. muistin mallittaminen graafina) esiintyvät yleisesti alan kirjallisuudessa (esim. [9]), ei tämä malli kokonaisuutena vastaa mitään kirjallisuudessa esitettyä.

Mallin tarkoituksena ei ole kuvata sitä, miten käytännön muistinhallinta muistin näkee, vaan sen tarkoituksena on olla mahdollisimman selkeä ja ilmaisuvoimainen väline eri muistinhallintamenetelmien ja -ongelmien esittelemiseen.

2.1 Muutin ja ohjelma

Tämän tutkielman käsitteistössä *ohjelma* koostuu *muuttimesta* (*mutator*) ja muistinhallinnasta. Muutin on ohjelman ”hyötykuorma” – se osa ohjelmaa, joka tekee, mitä ohjelman odotetaan tekevän. Se käyttää muistinhallinnan palveluita tehtävänsä suorittamiseen. Muistinhallinnan kannalta tuo ohjelman osa vain käy muuttelemassa tietorakenteita, joita muistinhallinta yrittää hallita (ks. luku 7.6) – siitä nimi ”muutin”.

[1] Suomen kielessä termi ”siivous” ei ole tässä merkityksessä kovinkaan yleisessä käytössä, mutta se mainitaan ATK-sanakirjassa kyseessä olevan käsitteen suomenkieliesnä nimenä. Lisäksi tuota nimeä on käytetty Tietojenkäsittelyopin laitoksella viitisentoista vuotta sitten tehdyssä opinnäytetyössä [12], joten päätin – perinteen vuoksi – käyttää tuota termiä tässäkin tutkielmassa.

2.2 Muistin rakenne

Muistimäärä, jonka ohjelma voi ottaa käyttöön, on aina rajattu. Tämä rajattu muistimäärä jaetaan kahteen osaan: *vapaaseen* ja *varattuun* muistiin. Varattu muisti puolestaan jakautuu *muuttujiksi*, jotka ovat toisistaan riippumattomia vektoreita, jotka sisältävät *alkioita*. Alkiot voivat joko viitata toisiin muuttujiin, jolloin ne ovat *osoittimia* tai ne voivat sisältää jotain hyödyllistä dataa, jolloin ne ovat *atomaarisia*. Muuttuja, jossa on vain atomaarisia alkioita, on itsekin *atomaarinen*. Osoitinalkio voi myös olla *tyhjä* (*null*), jolloin se ei viittaa mihinkään muuttujaan. Tässä tutkielmassa oletetaan yksinkertaisuuden vuoksi, että alkiot ovat muistinhallinnan kannalta kaikki saman kokoisia[2].

2.3 Luonnoskieli

Tässä tutkielmassa käytetään erityistä luonnoskieltä esittämään eri algoritmeja. Kielen ohjausrakenteet on kirjoitettu suomen kielellä ja lienevät selviä. Lohkorakenne ilmaistaan sisennyksellä. Aliohjelmien parametrienvälitys tapahtuu viitesemantiikalla (*call by reference*).

Luonnoskielessä käytettävät nimet esittävät muuttujia kuten missä tahansa ohjelmointikielessä. Nimiä ei tarvitse esitellä. Operaattori "$\leftarrow$" sitoo vasemmaksi operandiksi kirjoitettuun nimeen oikeanpuolisen operandin arvon. Saman nimen uudelleensitominen on destruktiivinen operaatio.

Osoitinta, joka osoittaa muuttujaan, merkitään luonnoskielessä kirjaimella. Jos `x` on epätyhjä osoitin, niin `x[i]` on sen päästä löytyvän muuttujan *i*:s alkio (laskeminen aloitetaan nollasta). Osoittimen `x` päästä löytyvän muuttujan kaikki epätyhjiä osoittimia sisältävät alkiot muodostavat joukon `Osoittimet(x)`. Tyhjää osoitinta esittää nimi `null`. Jos muistinhallintamenetelmä tarvitsee omia, muuttimelle näkymättömiä kenttiä, ne ovat käytettävissä negatiivisilla indekseillä.

Luonnoskielessä on osoitinaritmetiikka sallittua. Tätä käytetään esimerkeissä lähinnä muistinhallinnan sisäisten kenttien piilottamiseen muuttimelta.

2.4 Muistinhallinnan primitiivioperaatiot

Osalla tässä tutkielmassa esitellyistä muistinhallinntamenetelmistä on käytettävissään kaksi primitiivistä operaatiota: `p := Varaa(koko)` ja `Vapauta(p)`. Ensiksi mainittu ottaa palan vapaata muistia ja tekee siitä varatun muuttujan, jossa on *koko* alkiota. Muistialue sidotaan nimeen *p*. Jos varausoperaatio epäonnistuu, nimeen *p* sidotaan tyhjä osoitin. Jälkimmäinen operaatio ottaa sille annetun muuttujan ja tekee siitä vapaan muistin osan. Vapautusoperaatio onnistuu aina.

Käytännössä muistinhallinta toteuttaa nämäkin operaatiot itse, mutta niiden toiminta yleisessä tapauksessa ei kuulu tämän tutkielman alueeseen; kiinnostuneen

[2]Itse asiassa tämä oletus pätee kaikissa yleisissä tietokoneissa, jos alkiot ovat kaikki konesanan pituisia, kuten usein on.

lukijan kannattaa tutustua Wilsonin ym. artikkeliin [15]. Osa tarkasteltavista algoritmeista ovat kuitenkin sellaisia, ettei näitä operaatioita voi niistä irroittaa; niissä yhteyksissä näitä primitiivioperaatioita ei käytetä.

Varattujen muuttujien koon summa ei voi koskaan ylittää ohjelman käytettävissä olevan muistin kokonaismäärää. Varausoperaatio voi kuitenkin epäonnistua, vaikka varattujen muuttujien koon summa ynnä varattavan muuttujan koko olisikin pienempi kuin käytettävissä olevan muistin kokonaismäärä, sillä vapautusoperaatiot aiheuttavat muistin *pirstoutumista*.

2.5 Nimigraafi

Ohjelman muistiavaruus voidaan ajatella suunnatuksi, väritetyksi graafiksi. Graafiin lisätään valkoinen solmu aina kun uusi muuttuja luodaan; kun muuttuja vapautetaan, sitä vastaava solmu värjätään mustaksi. Kaaret esittävät epätyhjiä osoittimia. Syy solmujen väritykseen on se, että oikeassa muistisysteemissä vapautettuun muistialueeseen voi jäädä roikkumaan osoittimia, ja mallin tulee voida esittää tämä ongelma.

Muutin saa koskea vapaasti kaikkiin valkoisiin solmuihin; ne ovat sen omaisuutta. Sen sijaan mustiin solmuihin se ei saa koskea, vaikka se niihin pääsisikin käsiksi. Muistialue, joka joskus edusti kyseistä mustaa solmua voi olla jo uusiokäytössä jonakin toisena muuttujana, ja siihen koskeminen voi muuttaa muuttimen toimintaa ennustamattomasti ja virheitä tuottavasti. Toisaalta kyseinen muistialue voi olla muistinhallinnan tai jopa edellä esiteltyjen primitiivioperaattorien sisäisessä käytössä, ja sen muuttaminen voi sekoittaa näiden toiminnan. Joissakin järjestelmissä osa mustista solmuista on erityisesti suojattu niin, että niiden käyttäminen aiheuttaa virheilmoituksen ja tyypillisesti ohjelman suorituksen päättymisen.

Tietty joukko graafin valkoisia solmuja muodostaa *juurijoukon*[3]. Mitä tahansa tietä juurijoukon alkiosta johonkin solmuun sanotaan ko. solmun eli muuttujan *nimeksi* – ja tästä syystä koko graafia sanotaan tässä tutkielmassa *nimigraafiksi*. Erityisesti jokainen triviaali tie juurijoukon alkiosta siihen itseensä on kyseisen alkion nimi. Huomaa, että muuttujalla voi olla useita nimiä, ja joillakin muuttujalla – lähinnä syklisen tietorakenteen alkioilla – voi olla jopa numeroituvasti ääretön määrä nimiä.

2.6 Muistinhallinnan kolme perusstrategiaa

Perinteisesti ohjelmointikielten toteutukset tarjoavat kolme tapaa hallita ohjelman muistinkäyttöä [9, luku 1.1].

Yksinkertaisin tapa on käyttää *staattista* dataa: ohjelman käynnistyessä se luodaan `Varaa`-operaatiolla ja ohjelman päättyessä se tuhotaan `Vapauta`-operaatiolla. Staat-

[3] Tyypillisesti von Neumannin koneissa tämän joukon muodostavat täsmälleen koneen rekisterit sekä aktivaatiopino.

tiset muuttujat kuuluvat aina ohjelman juurijoukkoon. Muutin ei voi vaikuttaa staattisen muuttujan tilanvaraukseen ajon aikana.

Yleisin tapa on käyttää *pinosta varattua* muistia, joka on yleensä kytketty ohjelmointikielen aliohjelmakonseptiin. Aliohjelman *paikallisista* nimistä luodaan aliohjelman suorituksen alkaessa uudet, *aktiiviset* ilmentymät `Varaa`-operaatiolla. Rekursiivisille aliohjelmille luodaan uudet paikallisten nimien ilmentymät jokaista aliohjelman dynaamista ilmentymää kohti. Aliohjelmasta poistuttaessa sen aktiiviset paikalliset muuttujat tuhotaan `Vapauta`-operaatiolla. Näin pinosta varatut muuttujat ovat olemassa täsmälleen ne omistavan aliohjelman dynaamisen keston ajan

Koska ohjelmointikielissä tyypillisesti aliohjelmakutsut tapahtuvat LIFO-tapaan (*Last In, First Out*) – eli sen aliohjelman, jonka suoritus aloitettiin viimeisenä, suoritus päättyy aina ensimmäisenä – niiden paikalliset muuttujat syntyvät ja kuolevat myös LIFO-tapaan. Niinpä yleensä aliohjelmien paikallisten muuttujien muistinvaraus toteutetaan erityisellä *aktivaatiopinolla* (vertaa lukuun 7.5), johon laittaminen luo uuden paikallisen muuttujan ja josta poistaminen tuhoaa poistettavan muuttujan.

Aktiiviset paikalliset muuttujat kuuluvat elinaikanaan ohjelman juurijoukkoon.

Joustavin tapa on käyttää *dynaamista* muistia. Ohjelma voi luoda dynaamisia muuttujia milloin vain (yleensä eksplisiittisellä aliohjelmakutsulla). Manuaalisesti hallinnoidussa dynaamisen muistin järjestelmässä ohjelma joutuu myös itse huolehtimaan käyttämättömän varatun muistin vapauttamisesta, mutta järjestelmän kannalta se voi tehdä sen milloin vain.

2.7 Muuttimen ja muistinhallinnan rajapinta

Muutin käyttää muistia neljän operaation kautta:

p ← **`Uusi(`*`koko`*`)`** Luo uusi dynaaminen muuttuja, johon mahtuu *koko* alkiota. Osoitin tähän muuttujaan sidotaan nimeen *p*.

`Poista(`*`p`*`)` Poista dynaaminen muuttuja, jonka osoite on sidottu nimeen *p*, ja sido nimi *p* uudelleen tyhjään osoittimeen.

a ← **`Lue(`*`p, i`*`)`** Lue muuttujan *p* kentän numero *i* sisältö ja sido se nimeen *a*.

`Päivitä(`*`p, i, a`*`)` Päivitä muuttujan osoitinkentän *i* arvoksi *a*.

Kaksi ensimmäistä operaatiota käsittelevät vain dynaamisia muuttujia mutta kaksi seuraavaa operoivat millä tahansa muuttujalla.

Yksinkertaisuuden vuoksi oletetaan, että kaikki näiden operaatioiden kohteena olevat kentät ovat osoitinkenttiä.

Esityksen yksinkertaistamiseksi asetetaan muuttimelle seuraavat rajoitukset: Muutin ei koskaan käytä luonnoskielen sijoitusoperaatiota "←" suoraan, vaan kaikki sijoitukset tehdään `Päivitä`-operaatiolla. Lisäksi muutin sijoittaa aina ennen juurijoukon muuttujan kuolemaa kyseisen muuttujan kaikkiin osoitinkenttiin nolla-osoittimen. Nämä rajoitukset eivät ole vakavia, sillä luonnoskieli on tarkoitettu vain

muistinhallinnan kuvaamiseen; todellisessa tilanteessa käytetään oikeaa ohjelmointikieltä, jossa Nämä vaatimukset on helppo toteuttaa läpinäkyvästi kääntäjää muuttamalla.

3 Muistinhallinnan ongelma

Muistin hallinnoinnin tavoitteena on pitää huoli siitä, että juurijoukon transitiivinen sulkeuma sisältää ainoastaan valkoisia solmuja (varattua muistia) ja että tämän transitiivisen sulkeuman ulkopuolella on vain mustia solmuja (vapaata muistia). Jos valkoisesta solmusta on kaari mustaan solmuun, kyseinen osoitin *roikkuu* (*dangle*). Jos puolestaan juurijoukon transitiivisen sulkeuman ulkopuolella on valkoinen solmuy, on kyseessä *muistivuoto*.

Muistivuoto ei ole sinänsä kovin vakava ongelma. Niin kauan kun muistia on jäljellä niin, että uusia dynaamisia muuttujia voidaan luoda, muistivuodosta ei ole minkäänlaisia seuraamuksia. Useat automaattisen muistinhallinnan menetelmät itse asiassa antavat muistivuodon kasvaa juuri niin suureksi kuin mahdollista ja poistavat sen juuri viime hetkellä. Jos muisti kuitenkin täyttyy kokonaan muistivuodosta eikä asialle tehdä mitään, ei ohjelma voi jatkaa suoritustaan.

Roikkuvat osoittimet ovat ohjelman luotettavuuden kannalta erittäin vakava ongelma. Jos vapautetulla muuttujalla on nimi (eli ohjelma pääsee siihen käsiksi jonkin osoittimen kautta), ohjelma voi yrittää käyttää sitä. Koska se on vapaa (musta), astuu ohjelma lohikäärmeiden maahan, ja paras, mitä sille voi sattua on, että se kaatuu välittömästi. Huonommassa tapauksessa se tuhoaa muistinhallinnan (ja joskus jopa käyttöjärjestelmän) sisäisiä tietorakenteita, mistä tuloksena on tyypillisesti ohjelman sekoaminen.

4 Manuaalinen muistinhallinta ei ole ratkaisu

Aikaisemmin useimmat ohjelmointikielet vaativat, että ohjelma huolehtii omasta dynaamisen muistin käytöstään itse ja erityisesti huolehtii varattujen dynaamisten muuttujien vapauttamisesta. Toisin sanoen kielet vaativat, että ohjelmat käyttävät `Poista`-operaatiota aina tarpeen mukaan. Tämä on ongelmallista, sillä manuaalinen muistin vapautus mahdollistaa roikkuvat osoittimet, kun muuttuja vapautetaan yhden nimen kautta (ja samalla se nimi poistetaan) mutta muuttujalle jää vielä monta muuta nimeä (ks. esim. [13, luku 5.10.3.1]).

Joissakin sovelluksissa manuaalinen hallinta on välttämättömyys. Tällaisia ovat lähinnä kovia ajantasaisuusvaatimuksia asettavat sovellukset, joille myöhästynyt vastaus on yhtä kuin väärä vastaus. Tällaisia ovat esimerkiksi matalan tason tietoliikennejärjestelmät, käyttöjärjestelmän laiteajurit sekä esimerkiksi tietynlaiset tietokonepelit.

Muilla sovellusalueilla on parempi käyttää automaattista dynaamisen muistin hallintaa, jolloin `Poista`-operaatiota ei tarvitse (eikä edes ole hyvä) käyttää. Tämän tutkielman aiheena ovat tämän toteuttamisessa käytettävät menetelmät.

5 Siivouksen perusalgoritmit

Tämä luku esittelee siivouksen klassiset menetelmät: viitelaskuritekniikan, merkkaustekniikan ja kopioivan tekniikan. Luku perustuu lähinnä Jonesin ja Linsin kirjaan [9] ja Wilsonin laajennettuun katsausartikkeliin [14].

5.1 Viitelaskuri

Viitelaskuritekniikassa (*reference counting*) jokaiseen muuttujaan sijoitetaan ylimääräinen muuttimelle näkymätön kenttä, *viitelaskuri* (*reference count*). Sen tehtävänä on ilmaista, kuinka monta osoitinta osoittaa kyseiseen muuttujaan (eli mikä sen tuloaste on nimigraafissa). Tämän tiedon ylläpitämiseksi jokainen osoittimen muutos vaatii viitelaskurikentän muuttamisen: kun osoitinta muutetaan, osoittimen vanhan arvon osoittaman muuttujan viitelaskuria on pienennettävä yhdellä, ja uuden arvon osoittaman muuttujan viitelaskuria on kasvatettava yhdellä. Jos jonkin muuttujan viitelaskuri muuttuu nollaksi, ei muuttujaan ole enää osoittimia, ja se voidaan vapauttaa.

Viitelaskuritekniikka voidaan toteuttaa seuraavasti:

```
Uusi(koko):
    rv ← Varaa(koko + 1)
    jos rv = null:
        virhe "Muisti loppui"
    rv ← rv + 1 // alussa on yksi kenttä viitelaskurina
    rv[−1] ← 1
    palauta rv

Poista(p):
    // ei tehdä mitään

Lue(p, i):
    palauta p[i]

Päivitä(p, i, uusi):
    vanha ← p[i]
    p[i] ← uusi
    jos uusi osoittaa dynaamiseen muuttujaan:
        uusi[−1] ← uusi[ − 1] + 1
```

```
    jos vanha osoittaa dynaamiseen muuttujaan:
      VähennäViitelaskuria(vanha)

VähennäViitelaskuria(p):
    p[-1] ← p[-1] - 1
    jos p[−1] = 0:
      kaikille q joukossa Osoittimet(p):
        VähennäViitelaskuria(q)
      Vapauta(p)
      p ← null
```

Tärkeää tässä algoritmissa on huomata, että kun jonkin muuttujan viitelaskuri menee nollaan, käydään vähentämässä sen alkioiden osoittamien muuttujien viitelaskureita, ja tarvittaessa ne tuhotaan. Näin jos jokin poiston kohteeksi joutuva muuttuja on jonkin ison asyklisen rakenteen ainoa (käänteinen) reitti juurijoukkoon, joutuu `Päivitä`-operaatio poistamaan samalla koko puurakenteen.

Viitelaskuritekniikka pitää muistigraafin täysin konsistenttina koko ajan: se estää sekä muistivuodot (yhdellä merkittävällä poikkeuksella) että roikkuvat osoittimet. Sen hyväksi puoleksi on myös laskettava se, että muistinhallinta limittyy yhteen muuttimen toiminnan kanssa, jolloin sen hintakin tulee jaettua tasaisesti muun suorituksen lomaan. Ainoa poikkeus tähän on tilanne, jossa poistettavaksi tulee ison asyklisen rakenteen ”juuri”.

Viitelaskuritekniikan pahin ongelma on se, ettei se kykene poistamaan syklisiä rakenteita. Tarkastellaanpa esimerkin vuoksi tilannetta, jossa kaksi yhden osoitinalkion sisältävää muuttujaa viittaavat toisiinsa. Tällöin molempien tuloaste nimigraafissa on aina vähintään 1 (esimerkin toisesta muuttujasta tuleva osoitin on edelleen olemassa), joten molemmat muuttujat jäävät muistiin, vaikka ne eivät kuuluisikaan juurijoukon transitiiviseen sulkeumaan. Esimerkki yleistyy miten suurelle sykliselle rakenteelle tahansa.

Sykliongelma voidaan ratkaista yhdistämällä viitelaskuritekniikka johonkin myöhemmin esiteltävään siivousmenetelmään: viitelaskuria käytetään normaaliin tapaan asyklisten rakenteiden poistamiseen, ja toista siivousmenetelmää käytetään silloin tällöin syklisten rakenteiden siivoamiseen.

5.2 Merkkaa ja lakaise

Merkkaa ja lakaise -siivoin on siivousalgoritmeista vanhin. Sen kehitti John McCarthy alkuperäistä LISP-systeemiä varten [11, luvun 4 kohta c] Perusmuodossaan se antaa muistivuodon kasvaa johonkin rajaan asti (tyypillisesti kunnes `Varaa`-operaatio epäonnistuu), minkä jälkeen se selvittää, millä muuttujilla on nimi ja merkitsee muut vapaaksi muistiksi.

Siivousprosessi on siis kaksiosainen: ensin etsitään nimelliset ja sitten vapautetaan nimettömät muuttujat. Käytännössä nimelliset merkitään *merkkibitillä*, joka mene-

telmän perusalgoritmissa tallennetaan osana itse muuttujaa. Ensimmäinen osa, nimellisten etsiminen, tapahtuu ensin merkitsemällä kaikki juurijoukon alkiot ja sitten joko syvyyshakua tai leveyshakua käyttämällä merkitsemällä kaikki nimelliset muuttujat.

Merkkaa ja lakaise -algoritmi voidaan toteuttaa vaikkapa seuraavaan tapaan:

```
merkki ← tosi

Uusi(koko):
    rv ← Varaa(koko + 1)
    jos rv = null:
      GC()
      rv ← Varaa(koko + 1)
      jos rv = null:
        virhe "Muisti loppui"
    rv ← rv + 1
    rv[−1] ← ei merkki
    palauta rv

Poista(p):
    // ei tehdä mitään

Lue(p, i):
    palauta p[i]

Päivitä(p, i, uusi):
    p[i] ← uusi

GC():
    kaikille x joukossa juurijoukko:
      Jäljitä(x)
    kaikille x joukossa varattujen dynaamisten muuttujien joukko:
      jos x[−1] ≠ merkki:
        Vapauta(x)
    merkki ← ei merkki

Jäljitä(p):
    jos p[−1] ≠ merkki:
      p[−1] ← merkki
      kaikille q joukossa Osoittimet(p):
        Jäljitä(q)
```

On hyvä huomata, että järjestelmissä, joissa muuttujaan mahtuu yksi yksittäinen bitti niin, ettei kokonaista alkiota tarvitse sitä varten varata eikä se häiritse muutin-

ta, voidaan merkki (dynaamisen muuttujan alkio -1 ylläolevassa algoritmissa) tallettaa tuossa bitissä. Merkkibitti voidaan myös tallettaa muuttujasta erilleen jonkinlaiseen merkkibiteistä koostuvaan bittikarttaan sellaisissa järjestelmissä, joissa bittikartasta voidaan helposti lukea, mitä muuttujaa mikäkin bitti edustaa.

Merkkaa ja lakaise -algoritmi on yksinkertaisimmassa muodossaan luonteeltaan *stop-and-go*-tyyppinen: kun siivous katsotaan aiheelliseksi, muutin pysäytetään siivouksen ajaksi. Isoilla muistimäärillä (suhteessa järjestelmän nopeuteen) tämä voi aiheuttaa havaittavia pysähdyksiä. Vanhat LISP-ohjelmat pysähtyivät kerran runsaassa minuutissa noin viideksi sekunniksi kerrallaan. Muistinsiivouksen huono maine tietotekniikan ammattilaisten keskuudessa johtunee lähinnä tästä.

Toisin kuin viitteidenlaskenta, merkkaa ja lakaise -algoritmi kykenee poistamaan myös sykliset rakenteet. Toisaalta haittana on `Jäljitä`-aliohjelman käyttämä rekursio: merkkaa ja lakaise tarvitseekin yksinkertaisessa muodossaan muistia talletettamaan syvyys- tai leveyshaun väliaikaiset tietorakenteet. Tämä ongelma voidaan kiertää käyttämällä vaikkapa osoittimienkääntöä [9, luku 4.3]

5.3 Pysäytä ja kopioi

Fenichel ja Yochelson [7] kehittivätkin ensimmäisen kopioivan siivoimen ei niinkään etsimään vapaata muistia[4] vaan pakkaamaan käytössä olevan muistin niin, ettei virtuaalimuisti joudu kovasti käyttämään apumuistia, mikä hidastaisi ohjelman suoritusta kovasti.

Kopioivat siivoimet jakavat muistin kahtia kahteen alueeseen: lähdeavaruus (*fromspace*) ja kohdeavaruus (*tospace*) – näitä kutsutaan yhteisesti *puoliavaruuksiksi* (*semispace*). Ne ovat samankokoisia, joten perustyyppistä kopioivaa siivointa käyttävä muutin voi käyttää korkeintaan puolet käytössä olevasta muistista. Kaikki muuttujat sijaitsevat siivouksien välisenä aikana kohdeavaruudessa, ja uudet muuttujat luodaan sinne. Siivouksen aluksi tehdään *vaihto* (*flip*), jossa lähdeavaruuden ja kohdeavaruuden roolit vaihtuvat. Vaihdon jälkeen kaikki muuttujat sijaitsevat lähdeavaruudessa.

Siivouksen aikana tärkeäksi operaatioksi nousee *kopiointi* (*forwarding*). Tämä operaatio ottaa lähdeavaruudessa olevan muuttujan, kopioi sen kohdeavaruuteen ja merkitsee lähtöavaruuteen jääneeseen kopioon, että kopiointi on tehty sekä lisää siihen osoittimen, joka osoittaa kohdeavaruuden osoitteeseen. Kuitenkin jos lähtöavaruudessa oleva kopio on jo kopioitu, ei kopiointioperaatio tee mitään (vaan kertoo kutsujalle, että kohdeavaruuden kopio on siinä, mihin viittaava osoitin osoittaa). Lisäksi osoitin, jolla muuttuja löydettiin, päivitetään joka tapauksessa osoittamaan uuteen kopioon.

Nykyaikaisen kopioivan siivoimen perusalgoritmi on peräisin Cheneyltä [5]. Algoritmissa ensin kopioidaan koko juurijoukko peräkkäin kohdeavaruuden alkuun (tai

[4] He ajattelivat, että virtuaalimuisti on käytännössä loputon ja että siivousta ei siksi oikeastaan tarvittaisi.

siis ne dynaamiset muuttujat, jotka ovat juurijoukon osoittimien päissä). Sitten aletaan käydä kohdeavaruutta järjestyksessä läpi: jokainen kohdalle osuneen muuttujan osoittimen päässä oleva muuttuja kopioidaan kohdeavaruuteen aiemmin kopioitujen perään. Kun kaikki kohdeavaruudessa olevat muuttujat on käyty läpi, on kaikki juurijoukon transitiiviseen sulkeumaan kuuluvat muuttujat kopioitu, ja lähdeavaruus sisältää pelkkää roskaa. Seuraavan vaihdon jälkeen tapahtuva siivous kirjoittaa niiden yli.

Cheneyn algoritmi voidaan toteuttaa seuraavasti:

```
lähdeavaruus ← muistin alkuosoite
kohdeavaruus ← muistin puolivälin osoite
muistinkoko ← kohdeavaruus − lähdeavaruus
varausptr ← kohdeavaruus
näkymättömiä ← 2

koko ← −2 // indeksi
forward ← −1 // indeksi

Uusi(varattavaMäärä):
  jos varausptr − kohdeavaruus + varattavaMäärä + näkymättömiä ≥ muistinkoko:
    GC()
  jos varausptr − kohdeavaruus + varattavaMäärä + näkymättömiä ≥ muistinkoko:
    virhe "Muisti loppui"
  rv ← varausptr
  varausptr ← varausptr + varattavaMäärä + näkymättömiä
  rv[koko] ← varattavaMäärä + näkymättömiä
  rv[forward] ← null
  palauta rv

Poista(p):
  // ei tehdä mitään

Lue(p, i):
  palauta p[i]

Päivitä(p, i, uusi):
  p[i] ← uusi

GC():
  lähdeavaruus, kohdeavaruus ← kohdeavaruus, lähdeavaruus
  scanptr ← kohdeavaruus
  varausptr ← kohdeavaruus
  kaikille p joukossa juurijoukko:
    jos p osoittaa dynaamiseen muuttujaan:
```

```
      Kopioi(p)
    muuten:
      kaikille q joukossa Osoittimet(p):
      jos q osoittaa dynaamiseen muuttujaan:
        Kopioi(q)
  kunnes scanptr = varausptr:
    p ← scanptr + näkymättömiä
    scanptr ← p[koko]
    Kopioi(p)

Kopioi(p):
  jos p osoittaa dynaamiseen muuttujaan:
    jos p[forward] = null:
      kohde ← varausptr
      varausptr ← varausptr + p[koko]
      kaikille i joukossa [−näkymättömiä, p[koko] − näkymättömiä[ ∩ ℤ:
        p[i] ← kohde[i]
      p[forward] ← kohde
  p ← p[forward]
```

Huomaa, että muuttujan luonti tässä tekniikassa on helppoa, koska vapaa tila on yhtenäinen alue, josta voidaan napsaista sopivankokoinen pala pois – melkein kuin pinosta. Jos siivouksen alkamisen käynnistää käyttöjärjestelmän muistinsuojaustekniikoita hyväksikäyttäen (eikä tätä tehdä kovin usein), on muistinvaraus täysin samanlaista kuin pinosta varaaminen – siis nopeaa.

5.4 Vertailua

Appel [1] on väittänyt, että kopioivaa siivointa käyttävä muistinhallinta on tietyissä tilanteissa jopa pinosta varaamista nopeampaa. Perusajatuksena on, että kopioiva siivoin ei koske siihen muistin osaan, joka ei kuulu juurijoukon transitiiviseen sulkeumaan, joten sen suoritusaika on riippumaton muistin koosta, ja kutakin varattua muuttujaa kohti siivouksen kustannus laskee, kun muistin määrä kasvaa.

Appelin analyysi on kuitenkin yksinkertaistus; kuten Jones ja Lins [9] huomauttavat, vakiot merkitsevät paljon. Käytännössä merkkaaminen on lakaisua kalliimpi toimenpide, ja yksittäisen muuttujan merkkaaminen on puolestaan halvempaa kuin kokonaisen muuttujan kopiointi. Näin ollen käytössä olevan muistin määrä joudutaan kasvattamaan todella korkealle, ennen kuin kopioivan keräimen hyöty tulee todella esille. Jo Appelin oma arvio on se, että hyöty saavutetaan vasta, kun muistia on vähintään seitsemän kertaa niin paljon kuin sitä minimissään tarvittaisiin.

Myös virtuaalimuistikysymykset tulevat esille. Vaikka kopioiva siivous kehitettiinkin aikoinaan nimenomaan virtuaalimuistin tehokasta käyttöä ajatellen [7], ei siitä saavuteta parasta mahdollista sivutuskäyttäytymistä. Kopiointi kyllä tiivistää käy-

tössä olevan muistin mahdollisimman vähille virtuaalimuistisivuille, mutta toisaalta jokainen siivouspari koskettaa kaikkia sivuja ja pakottaa ne takaisin keskusmuistiin. Tässä suhteessa merkkaavat algoritmit ovat parempia kuin kopioivat, varsinkin jos merkkibitit tallennetaan erillisissä bittivektoreissa; tällöin itse muuttujat sisältäviä sivut voidaan pitää poissivutettuina.

Toisaalta kopioivat siivoimet voivat kopioida muuttujat sopivaan järjestykseen niin, että todennäköisesti lähekkäin käytetyt muuttujat ovat samalla sivulla. Tämä onnistuu parhaiten käyttämällä syvyyshakua, joten Cheneyn algoritmi ei tähän sellaisenaan käy. Ongelman ratkaisevat kuitenkin useat Cheneyn algoritmin variaatiot, ks. [9, luku 6.6].

Viitelaskuritekniikan pahin ongelma on sen kykenemättömyys vapauttaa syklisiä tietorakenteita. Toinen iso ongelma on sen tehottomuus: tyypillinen imperatiivinen ohjelma muuttaa muuttujan kenttien arvoja paljon useammin kuin varaa tai vapauttaa dynaamisia muuttujia, joten viitelaskureiden jatkuva päivittäminen maksaa. Tämän vuoksi monet kielentoteuttajat ovat vältelleet viitelaskuritekniikkaa. Viitelaskureiden paras puoli, vapautuksen välittömyys, tekee toisaalta tekniikasta houkuttelevan ympäristöihin, joissa halutaan sekä automaattisen että manuaalisen muistinhallinnan edut samassa paketissa.

Siivousalgoritmin valinta on varsin paljon kiinni siitä, mitä ohjelmointityyliä sen on tarkoitus tukea. Kopioivat siivoimet sopivat funktionaaliseen tyyliin, jossa luodaan muuttujia hyvin usein mutta jossa suurin osa muuttujista muuttuu roskaksi ennen seuraavaa siivousta. Tällöin nimittäin siivouksesta elossa selviäviä muuttujia on suhteellisen vähän, ja Appelin seitsenkertaisen muistin rajaan päästään helposti. Imperatiivinen tyyli toimii paremmin joko viitelaskuritekniikan tai merkkaavan siivoimen kanssa.

6 Kohti parempaa vasteaikaa

Luvussa 5 esiteltiin muistinsiivouksen perusmenetelmät. 1980-luvulta alkaen on alettu kehittää parempia, näihin perusalgoritmeihin perustuvia menetelmiä. Tässä tutkielmassa ne esitellään kursorisesti; kiinnostuneen lukijan kannattaa tutustua Jonesin ja Linsin kirjaan [9].

6.1 Ikäperustainen siivous

Ikäperustainen siivous (*generational garbage collection*) perustuu niinsanotulle *heikolle ikähypoteesille* (*weak generational hypothesis*):

> Useimmat muuttujat kuolevat nuorena. [9, luku 7.1]

Toisin sanoen, suurin osa edellisen siivouksen jälkeen syntyneistä muuttujista ovat jo roskaa, kun siivous seuraavan kerran tulee kohdalle. Toisinaan puhutaan *lapsikuolleisuudesta* (*infant mortality*).

Ideana ikäperustaisessa siivouksessa on siivota ”lastentarha”, jonne suurin osa roskasta kerääntyy, usein ja käydä vanhemmat muuttujat läpi harvemmin. Näin tulee pienempi osa muistista käytyä läpi useimmilla siivouskerroilla, ja siivouskerrat nopeutuvat. Tästä seurauksena on vasteajan paraneminen, kun ohjelma ei pysähdy kerralla kovin pitkäksi aikaa. Yleensä ikäperustaisen siivouksen pohjalla on kopioiva algoritmi, ja ikäperustaisuus johtaa myös siihen, ettei pitkäikäisiä muuttujia kopioida yhtenään ympäriinsä.

Muisti jaetaan ikäperustaisessa siivouksessa kahteen tai useampaan *sukupolveen* (*generations*). Yksi sukupolvi siivotaan joka siivouskerralla, toinen siivotaan joka toisella (tai harvemmin) ja niin edelleen siten, että kullakin kerralla siivotaan k nuorinta sukupolvea kuin ne olisivat yksi sukupolvi (missä k on jokin pieni luku). Kun lastentarhasukupolvessa on jokin muuttuja ehtinyt elää riittävän pitkään, se todetaan pitkäikäiseksi ja ylennetään seuraavaan sukupolveen jollakin siivouskerralla. Vastaavasti muista sukupolvista ylennetään pitkäikäiset muuttujat seuraaviin sukupolviin, kunnes saavutaan siihen sukupolveen, jonka jälkeen ei enää ole uusia sukupolvia. Tavallisesti sukupolvia on kaksi.

Kun lastentarhaa siivotaan, juurijoukkoon täytyy laskea mukaan paitsi kaikki staattiset ja pinosta varatut muuttujat myös kaikki ne osoittimet, jotka lähtevät jostain vanhemmasta sukupolvesta lastentarhaan. Jos ohjelmassa ei käytetä lainkaan destruktiivista päivitystä, näitä osoittimia ei voi syntyä. Jos sitä käytetään vain harvoin, voidaan nämä hajaosoittimet vaikkapa pistää talteen johonkin listaan aina kun ne luodaan (tämä tehdään `Päivitä`-aliohjelmassa). Mikäli vanhojen muuttujien sisältämiä osoittimia muutetaan usein, täytyy niistä pitää kirjaa jollakin kehittyneemmällä tavalla tai sitten unohtaa koko ikäperustaisen siivoksen idea.

Ikäperustainen siivous sopii parhaiten tilanteisiin, joissa muuttujia luodaan paljon niin, että suurin osa niistä jätetään orvoiksi melkein heti ja jossa vanhojen muuttujien sisältämiä osoittimia ei juuri muutella. Tällaisia ohjelmia tuottaa erityisesti *funktionaalinen* ohjelmointityyli.

7 Siivoimen tarvitsemat ajonaikaiset tietorakenteet

Edellä on piilotettu joitakin olennaisia asioita muistinhallintamallin (ks. luku 2) abstraktion taakse. Tärkein kysymys on, miten siivoin erottaa atomit ja osoittimet toisistaan: erityisesti kopioiva siivous voi toimia oikein vain, jos se osaa erottaa ne toisistaan.

7.1 Osoittimien laputus

LISP ja muut dynaamisesti tyypitetyt kielet käyttävät osoittimien tunnistamiseen tyypillisesti *laputusta* (*pointer tagging*). Perusidea on varata konesanasta yksi bitti merkitsemään, onko kyseessä osoitin vai atomi. Vanhoissa järjestelmissä konesanaan mahtui yhden tai kahdenkin osoittimen tai atomin lisäksi yksi tai kaksi ylimää-

räistä bittiä, joten sekä merkkausalgoritmien merkkibitti että lappubitti oli luontevaa tallentaa näin [10, s. 412].

Uudemmissa järjestelmissä lappubitti joudutaan ottamaan siitä tilasta, johon voitaisiin tallettaa varsinaista dataakin. Osoittimia tämä ei yleensä häiritse, sillä osoittimet ovat tavallisesti kahdella jaollisia, jolloin lappubittinä voidaan käyttää osoittimen vähiten merkitsevää bittiä. On tosin mietittävä, onko osoittimen merkkinä ykkösvai nollabitti; Appelin [3] mukaan ykkösbitti on parempi, sillä suurin osa osoittimien kautta tähtäämisistä käyttävät jonkinlaista kiinteää siirrosta joka tapauksessa, ja ykkösen lisääminen tai vähentäminen ei tilannetta juuri muuta. Lisäksi nollabitin varaaminen atomaariselle datalle yksinkertaistaa huomattavasti aritmeettisia laskuja.

Laputuksen huono puoli nykyaikaisten järjestelmien kanssa on se, että se rajoittaa muuttujan alkioon mahtuvan atomaarisen datan kokoa. Usein joudutaankin iso atomaarinen data *laatikoimaan* (*box*): varaamaan sille tilaa jostakin siivoimen ulottumattomista ja tallentamaan siihen alkioon, jossa itse datan kuuluisi olla, osoitin, joka osoittaa tähän tilaan.

7.2 Muuttujankuvaimet

Toinen vaihtoehto on käyttää *muuttujankuvaimia* (*object descriptors*). Jokaiseen muuttujaan lisätään muuttimelle näkymätön kenttä, joka osoittaa tyyppikohtaiseen tietorakenteeseen, joka kertoo, mitkä muuttujan kentät ovat osoittimia ja mitkä eivät. Tämä tekniikka soveltuu parhaiten staattisesti tyypitetyille kielille, koska silloin kääntäjä kykenee helposti tuottamaan tuollaisen kuvaimen.

7.3 Tyypinkuvaimet

Kolmas vaihtoehto, joka soveltuu vain staattisesti tyypitetyille kielille, on tyyppiinformaation koodaaminen ohjelman staattisen datan joukkoon niin, että siivoin tulkitsee tyyppi-informaatiota sitä mukaa kun se seuraa osoitinrakenteita dynaamisessa muistissa. Prosessin alkuunpanemiseksi siivoimen täytyy tietää, mitä tyyppiä juurijoukon muuttujat ovat. Staattisilla muuttujilla se on triviaalia, mutta pinosta varatut muuttujat ovat ongelma. Tätä tarkastellaan luvussa 7.5. [2]

7.4 Räätälöidyt siivoimet

Neljäs vaihtoehto, joka myös soveltuu lähinnä staattisesti tyypitetyille kielille, on siivoimen räätälöinti (ks. esim. [6]). Tässä tekniikassa kääntäjä generoi esimerkiksi merkkaa ja lakaise -metodia varten jokaiselle tyypille oman merkkausaliohjelman, joka kykenee toimimaan ilman ajonaikaista dataa. Tämä metodi on variaatio tyyppikuvainmetodista, jota kuvattiin luvussa 7.3.

7.5 Erityistapaus: Pinossa olevat juurijoukon muuttujat

Lähes kaikkia edellämainittuja keinoja voidaan aivan hyvin käyttää myös pinon selaamiseen. Poikkeuksen muodostavat tyyppikuvaintekniikka ja räätälöidyt siivoimet, sillä ne itse tarvitsevat pinomuuttujien tyyppitietoa toimiakseen. Kuitenkaan mitään niistä ei kannata käyttää, sillä tyyppi-informaation tallettaminen pinoon itseensä tuhlaa pinomuistia ja hidastaa pinosta varaamista – ja parempiakin tapoja on.

Appel kuvaa erästä tällaista tapaa kääntäjätekniikan oppikirjassaan [4, s. 293]: Kääntäjä kokoaa staattiseen muistiin tietorakenteen, joka kuvaa aliohjelmien paluuosoitteet niitä vastaaviin aktivaatiotietueiden[5] kuvauksiin. Siivoin sitten osaa löytää kustakin aktivaatiotietueesta siihen tallennetun paluuosoitteen jollakin tavalla (esimerkiksi paluuosoite voi olla tietueen ensimmäinen alkio), ja hakee tietueen rakenteen kuvauksen yllä mainitusta tietorakenteesta.

7.6 Vähittäinen siivous

Siivouksen perusalgoritmeista merkkaa ja lakaise sekä pysäytä ja kopioi ovat *stop-and-go*-tyyppisiä, jolloin muuttimen toiminta pysäytetään siivouksen ajaksi. *Vähittäisen siivouksen* (*incremental garbage collection*) idea on tehdä kerrallaan vain vähän työtä niin, että tuo pysähdys olisi kerrallaan mahdollisimman lyhyt, jolloin ohjelman vasteaika paranee reilusti. Esimerkiksi saatetaan tehdä pieni määrä merkkausta joka aliohjelmakutsun kohdalla.

Vähittäisen siivouksen haaste on se, kuinka vähittäin etenevä siivous ja sen kanssa limittäin etenevä muutin saadaan kunnioittamaan toisiaan. Erityisesti tässä korostuu nimi ”muutin”: se kun muuttaa siivousprosessin selän takana niitä tietorakenteita, joita siivousprosessi yrittää seurata. Jonkinlainen yhteistyömekanismi on välttämätön.

Dijkstralta peräisin oleva *kolmivärimerkkaamisen* (*tricolour marking*) abstraktio on hyödyllinen tapa hahmottaa vähittäisen siivouksen toimintaa. Jokaisella muuttujalla on väri: musta, harmaa ja valkoinen. (Huomaa, että tällä värityksellä ei ole mitään tekemistä abstraktissa muistimallissa (luku 2) esitetyn värjäyksen kanssa.) Mustat muuttujat ovat sellaisia, jotka siivoin on jo ehtinyt käydä läpi ja jonka osoittimien päästäkin löytyvät muuttujat on ehditty vilkaista. Harmaat muuttujat ovat sellaisia, jotka siivoin on käynyt vilkaisemassa mutta joita se ei ole vielä ehtinyt käydä kokonaan läpi. Valkoiset muuttujat ovat sellaisia, joita siivoin ei ole vielä edes vilkaissut.

Kolmiväri-invariantti (*tricolour invariant*) sanoo, että minkään mustan muuttujan sisältämän osoittimen osoittama muuttuja ei ole valkoinen. Toisin sanoen, mustan värin aluetta ympäröi aina harmaa rintama, joka erottaa mustan valkoisesta. Jos muutin tekee jotain, joka rikkoisi tämän invariantin, sen pitää muutoksen tehtyään korjata asia jotenkin. Tämä toteutetaan *muureilla*.

[5] Aktivaatiotietue on muuttuja, joka sisältää aliohjelman paluuosoitteen ja aktiiviset paikalliset muuttujat.

Lukumuuri (*Read barrier*) on jokaiseen lukuoperaatioon (`Lue`) sisältyvä tarkistus siitä, että muutin ei pääse näkemään muistia sellaisessa tilassa, että sen muuttaminen rikkoisi kolmiväri-invariantin. Tarvittaessa lukumuuri tekee hieman siivoustyötä, jotta tällaiseen tilanteeseen päästäisiin.

Kirjoitusmuuri (*Write barrier*) on jokaiseen kirjoitusoperaatioon (`Päivitä`) sisältyvä tarkistus siitä, että jos muutin rikkoo invariantin, niin se myös korjaa sen. Tavallisesti tämä tarkoittaa sitä, että jokin mustaksi värjätty muuttuja värjätään uudestaan harmaaksi.

Jokainen yksittäinen vähittäisen siivouksen menetelmä käyttää joko lukumuuria tai kirjoitusmuuria. Molempia ei sama metodi koskaan tarvitse. Jotkin metodit toteuttavat muurin käyttäen käyttöjärjestelmän muistisuojausta apunaan ja jotkin toiset metodit vaativat toimiakseen erityistä laitteistotukea. Osa metodeista toteuttaa muurin täysin ohjelmallisesti ilman käyttöjärjestelmän tukea.

Muuriin liittyy aina (jollei sitä ole toteutettu laitteistotasolla) hitauselementti. Lukumuuri hidastaa jokaista lukua ja kirjoitusmuuri jokaista kirjoitusta. Kirjoitusmuuri sopii tilanteisiin, joissa muuttujia harvoin päivitetään, sillä uusien muuttujien alustusta ei tarvitse suojata muurilla. Lukumuuria käytetään erityisesti kopioivien vähittäissiivousmenetelmien kanssa, jolloin lukumuuri tarkistaa, että luettava muuttuja on kohdeavaruudessa ja tarvittaessa tekee tämän kopioinnin.

8 Yhteenveto

Muistinsiivous on jo yli neljäkymmentä vuotta vanha tekniikka: John McCarthy esitteli ensimmäisen merkkaa ja lakaise -menetelmän vuonna 1960 [11]. Sittemmin muistinsiivoustekniikka on kehittynyt valtavasti, ja perusalgoritmien lisäksi on kehitetty edistyksellisiä ikäpohjaisia ja vähittäisiä siivousmenetelmiä, jotka vähentävät siivouksen kustannuksia.

Tässä tutkielmassa tarkasteltiin muistinsiivouksen perusalgoritmeja sekä katsastettiin tehokkaampien algoritmien perusperiaatteet. Tämä kaikki esitettiin muistinhallinnan abstraktin mallin kontekstissa.

Kirjoittaja haluaa kiittää Janne Kujalaa tämän tutkielman luonnosversioiden lukemisesta ja kommentoimisesta.

Viitteet

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[2] Andrew W. Appel. Runtime tags aren't necessary. Tekninen raportti, Princeton University Computer Science, 1988.

[3] Andrew W. Appel. A runtime system. Tekninen raportti, Princeton University Computer Science, 1989.

[4] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[5] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] David L. Detlefs. *Concurrent, Atomoc Garbage Collection*. Tohtorinväitöskirja, Carnegie Mellon University Department of Computer Science, 1991.

[7] Robert R. Fenichel ja Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[8] James Gosling, Bill Joy, Guy Steele, ja Gilad Brancha. *The Java Language Specification*. Addison-Wesley, toinen laitos, 2000.

[9] Richard Jones ja Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, kolmas laitos, 1997.

[11] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[12] Erkki Sairanen. Muistinsiivous. Pro gradu -tutkielma, Jyväskylän yliopiston tietojenkäsittelyopin laitos, 1984.

[13] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, neljäs laitos, 1999.

[14] Paul R. Wilson. Uniprocessor garbage collection techniques. Julkaistaan lehdessä ACM Computing Surveys (67 sivua).

[15] Paul R. Wilson, Mark S. Johnstone, Michael Neely, ja David Boles. Dynamic storage allocation: A survey and critical review. Kirjassa *Proceedings of the 1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.