Funktionaalisen- ja olioparadigman suorituskykyvertailu Scala-kielessä

LuK-tutkielma
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Tietojenkäsittely
2020
Jaakko Paju

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos

JAAKKO PAJU: Funktionaalisen- ja olioparadigman suorituskykyvertailu Scala-kielessä

LuK-tutkielma, 25 s., 0 liites.
Tietojenkäsittely
Huhtikuu 2020

Funktionaalisesta ohjelmoinnista lainattuja ominaisuuksia lisätään jatkuvasti perinteisiin imperatiivisiin ohjelmointikieliin. Yksi funktionaalisen ohjelmoinnin keskeisimmistä käsitteistä on muuttumattomat arvot. Muuttumattomien arvojen ja tietorakenteiden käyttö saattaa lisätä kopioimisen tarvetta ja samalla heikentää ohjelman suorituskykyä.
Tutkielman tarkoituksena on perehtyä funktionaalisten ohjelmien suorituskykyyn vaikuttaviin seikkoihin sekä verrata funktionaalisen paradigman suorituskykyä imperatiiviseen paradigmaan. Vertailut tehdään Scala-kielellä, sillä se tukee kumpaakin edellä mainittua paradigmaa. Tutkielmassa keskitytään järjestettyjen muuttumattomien ja muuttuvien kokoelmien suorituskyvyn vertailuun. Suorituskykyä tutkitaan useamman eri mittauksen pohjalta ja mittauksien tuloksia vertaillaan toisiinsa.
Mittauksissa ei havaittu säännönmukaisia eroja muuttumattomien ja muuttuvien kokoelmien suorituskyvyssä, vaikka muuttuvat kokoelmat olivat joissain mittauksissa hieman muuttumattomia suorituskykyisempiä. Suurin vaikutus suorituskykyyn on tarkoituksenmukaisen tietorakenteen valitsemisella riippumatta onko kyseessä muuttuva vai muuttumaton kokoelma.
Tutkielman pohjalta ei voida tehdä johtopäätöksiä funktionaalisen paradigman kokonaisvaltaisesta vaikutuksesta ohjelman suorituskykyyn. Kokonaisvaltaisten vaikutusten arvioimiseksi tulisi tutkia myös muita funktionaalisen paradigman keskeisiä käsitteitä, kuten sulkeumia, hahmontunnistusta ja rekursiota.

Asiasanat: suorituskyky, funktionaalinen ohjelmointi, ohjelmointiparadigma, Scala

Sisältö

1 Johdanto

Funktionaalinen ohjelmointi kasvattaa suosiotaan jatkuvasti. Useisiin yleiskäyttöisiin ja alun perin imperatiivisiin ohjelmointikieliin on lisätty ominaisuuksia funktionaalisesta ohjelmoinnista. Esimerkiksi suositut oliokielet Java, Python ja C++ ovat kaikki lainanneet funktionaalisesta ohjelmoinnista anonyymit sekä korkeamman asteen funktiot.

Funktionaalisen paradigman keskeisiä käsitteitä ovat muuttumattomat arvot ja tietorakenteet. Muuttumattomien arvojen käyttäminen usein lisää kopioimisen tarvetta verrattuna ohjelmiin, joissa käytetään muuttuvia arvoja. Kopioimisen seurauksena muistia pitää varata ja vapauttaa useammin kuin muuttuvia arvoja käytettäessä. Tämä herättää kysymyksen funktionaalisten ohjelmien suorituskyvystä verrattuna imperatiivisiin ohjelmiin.

Tutkielmassa pyritään saamaan vastaus siihen, millaisia vaikutuksia suorituskykyyn funktionaalisen paradigman käytöllä on verrattuna olioparadigmaan. Suorituskykyvertailut keskittyvät Scala-ohjelmointikieleen, sillä se tukee kumpaakin paradigmaa, jolloin suorituskyvyn vertailu paradigmojen välillä on mielekästä ja suoraviivaista. Tarkastelu kohdistuu muuttumattomiin tietorakenteisiin ja niihin liittyviin algoritmeihin. Tutkielma on suoritettu perehtymällä aihetta käsittelevään kirjallisuuteen ja suorituskykymittauksiin.

Luvussa 2 esitellään molemmat vertailun kohteena olevat paradigmat. Luvussa 3 esitellään Scala-kielen rakenteet, ja miten ne tukevat kumpaakin paradigmaa. Luvussa 4 tarkastellaan Scalan standardikirjaston muuttumattomien kokoelmien suorituskykyä ja verrataan sitä muuttuvien kokoelmien suorituskykyyn. Viimeisenä luvussa 5 esitellään johtopäätökset ja kootaan tutkielman tulokset.

2 Ohjelmointiparadigmat

Ohjelmointiparadigmat luokittelevat ohjelmointikieliä sen perusteella, miten kieli on suunniteltu mallintamaan ongelmia ja minkälaisia mekanismeja kieli tarjoaa näiden ongelmien ratkaisuun. Yhdessä ohjelmointikielessä voi olla vaikutteita useammasta paradigmasta. Tällaista kieltä kutsutaan *moniparadigmaiseksi* kieleksi. Ohjelmointiparadigmat voi jakaa karkeasti kahteen yläluokkaan: *imperatiivisiin* ja *deklaratiivisiin*. [1, Luku 6]

Imperatiiviset kielet keskittyvät ohjelmointiongelman ratkaisuun määrittelemällä *miten* tietokoneen tulisi ratkaista laskettava ongelma. Ohjelmat siis rakentuvat peräkkäisistä käskyistä, jotka muokkaavat ohjelman tilaa ja näin ratkaisevat ongelman. Imperatiivisista kielistä puhutaan matalan tason kielinä, sillä ongelman ratkaisu mallinnetaan niissä tietokoneen näkökulmasta. [2, Luku 1]

Deklaratiiviset kielet ratkaisevat ongelman vastaamalla kysymykseen *mitä* tietokoneen tulisi tehdä ongelman ratkaisemiseksi. Käytännössä tämä tarkoittaa ongelman ratkaisun kuvailemista ilman toteutuksen yksityiskohtiin uppoutumista. Deklaratiiviset mielletään yleensä korkean tason kielinä, sillä ne mallintavat ongelmanratkaisua ohjelmoijan näkökulmasta. [2, Luku 1]

2.1 Olioparadigma

Olio-ohjelmoinnin perusajatus on kuvata ongelmaa olioiden avulla, jotka kukin kuvaavat jotakin ongelma-alueen käsitettä. Olioiden tarkoitus on kapseloida kuvaamansa käsitteen tieto ja tila sisäänsä sekä tarjota operaatioita kapseloidun tiedon muokkaamiseen ja tar-

kasteluun. Olioparadigma on osa imperatiivisia paradigmoja, sillä ongelmanratkaisu tapahtuu peräkkäisillä komennoilla, jotka muuttavat olioiden ja samalla koko ohjelman tilaa kohti ratkaistua ongelmaa. [2, Luku 1] [1, Luku 10]

2.2 Funktionaalinen paradigma

Funktionaaliset ohjelmat rakentuvat yksittäisistä funktioista, joita yhdistämällä luodaan yhä isompia funktiota. Näillä funktioilla ei ole tilaa, ja ne ovat puhtaita eli eivät aiheuta *sivuvaikutuksia*. Sivuvaikutuksia voivat olla esimerkiksi muuttujan arvon muuttaminen, poikkeuksen nostaminen, tiedoston lukeminen tai kirjoittaminen, pyyntö tietokantaan tai verkon ylitse sekä näytölle piirtäminen. Tilaton ja puhdas funktio palauttaa tilanteesta riippumatta tietyllä syötteellä aina saman arvon. [3, Luku 1]

Ohjelmointikieli ilman sivuvaikutuksia olisi käytännössä hyödytön, joten funktionaaliset ohjelmointikielet tarjoavat erilaisia mekanismeja sivuvaikutusten hallintaan. Funktionaalisen ohjelmoinnin katsotaan yleensä kuuluvan deklaratiiviseen paradigmaan. [1, Luku 11]

2.3 Paradigmoille tyypillisiä ominaisuuksia

Lauseke on ohjelmointikielen rakenne, jonka suoritus tuottaa arvon. Lauseke voi koostua joko arvosta tai operaattorista, jota on sovellettu yhteen tai useampaan lausekkeeseen. Esimerkiksi `3+2` on numeerinen lauseke, jossa on yksi operaattori `+`, jota sovelletaan kahteen numeeriseen arvoon `3` ja `2`. Lausekkeen arvo voidaan sijoittaa muuttujaan, antaa parametriksi funktioon tai sen arvo voidaan palauttaa funktiosta. Lauseke on minkä tahansa ohjelmointikielen perusrakenne, eli lausekkeita on funktionaalisissa sekä oliokielissä. [1, Luku 6]

Funktionaalisessa ja olioparadigmassa molemmissa on muuttujia, mutta niitä käsitellään eri tavoilla. Funktionaalisissa kielissä muuttujaan sijoitettua arvoa ei voi muuttaa

alustuksen jälkeen, kun taas imperatiivissa kielissä muuttujan arvoa voi yleensä muuttaa. Sama pätee tietorakenteisiin: imperatiivissa kielissä niiden muuttaminen alustamisen jälkeen on sallittua, funktionaalisissa ei. [3, Luku 3] Funktionaalisissa kielissä funktioita kohdellaan kuin mitä tahansa muitakin arvoja: niitä voi sijoittaa muuttujiin, antaa parametrina toiseen funktioon tai käyttää funktion palautusarvona. Oliokielissä tämä ei aina ole mahdollista. [1, Luku 6]

Komento on ohjelmointikielen rakenne, jonka suoritus ei aina tuota arvoa ja saattaa aiheuttaa sivuvaikutuksia. Komennot saattavat esimerkiksi muuttaa olioiden ja muuttujien tilaa, lukea käyttäjän tuottamaa syötettä tai nostaa poikkeuksen. Komentojen suorittamisella saattaa olla sivuvaikutuksia, joten niiden suoritusjärjestys on merkityksellinen. Imperatiivisiin kieliin komennot kuuluvat oleellisesti, mutta funktionaalisiin eivät.

Imperatiivisissa kielissä toisto toteutetaan silmukoilla. Silmukkaa suoritetaan ennalta määritelty määrä tai kunnes silmukan suoritusta säätelevä ehtolause muuttuu epätodeksi. Ehtolauseen arvo määräytyy muuttujan arvon mukaan, jota muutetaan silmukan suorituksen aikana. Funktionaalisissa kielissä muuttujien arvoa ei voi muuttaa, ja siitä syystä silmukat olisivat hyödyttömiä. Silmukoiden sijaan käytetään rekursiota, eli funktio kutsuu itseään, kunnes funktioon määritelty perustapaus saavutetaan. [1, Luku 6 ja 11]

Kummankin paradigman ominaisuuksissa on omat ongelmakohtansa. Imperatiivisia ohjelmia on tavallisesti vaikea säikeistää, sillä jos useat säikeet käsittelevät samaa oliota samanaikaisesti, saattaa olion tila olla odottamaton. Funktionaalisessa ohjelmassa samaa ongelmaa ei ole, koska luomisen jälkeen olion tila ei muutu, ja sitä voi turvallisesti käyttää useampi säie. [4, Luku 6]

Toiston toteuttaminen rekursion avulla toimii hyvin, kunhan rekursio ei ajaudu liian syväksi. Rekursiivinen funktiokutsu kasvattaa kutsupinoa, kunnes pino vuotaa yli ja ohjelma kaatuu. Tämä tarkoittaa, että käsiteltävän datan määrä ei voi olla suuri tai kääntäjän pitää pystyä tekemään häntäoptimointia. Häntäoptimointi on tapa, jolla rekursiivinen kutsu voidaan muuttaa silmukaksi, jos se on funktion viimeinen operaatio. [4, Luku 8]

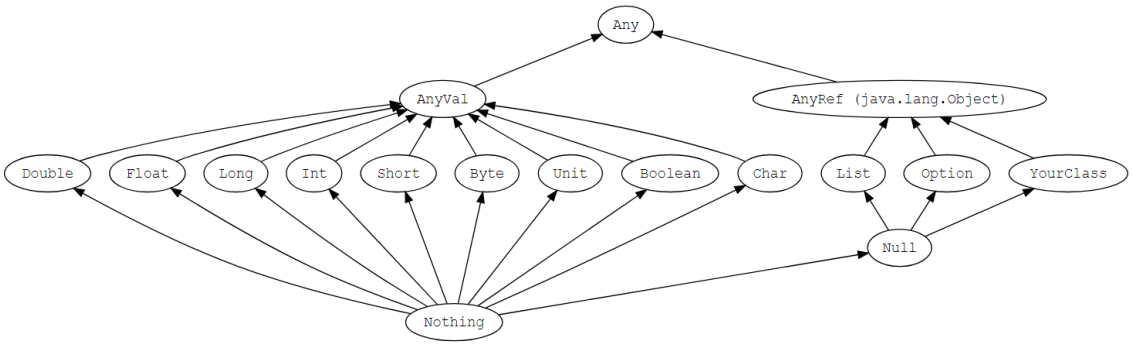
3 Scala

Scala on staattisesti tyypitetty moniparadigmainen käännettävä ohjelmointikieli, joka yhdistää funktionaalisen ohjelmoinnin ja olio-ohjelmoinnin ominaisuuksia. Scala on korkean tason kieli ja sen syntaksi on kompaktia ja eleganttia. Scalan kääntäjä ja tyyppijärjestelmä takaavat tyyppiturvallisuuden kääntämisen aikana muutamaa poikkeusta lukuun ottamatta. Scala-koodi on tarkoitettu käännettäväksi Javan tavukoodiksi, ja se mahdollistaa Java-kirjastojen käyttämisen suoraan Scalasta. Tavukoodia ajetaan Javan virtuaalikoneessa (JVM). Tässä tutkielmassa tarkastellaan Scalan versiota 2.12.10. [5, Introduction] [4, Luku 2]

3.1 Muuttujat

Scalassa on tavallisten muuttujien lisäksi vakioita, jotka ovat muuttujia, joiden arvoa ei alustuksen jälkeen voi muuttaa. Vakion määrittelyyn käytetään avainsanaa **`val`** ja muuttujan määrittelyyn **`var`**. Funktionaalisessa Scalassa käytetään pääasiassa **`val`**-avainsanalla määriteltyjä muuttujia, ja imperatiivisessa tyylissä voidaan käyttää molempia.

Muuttujan nimen jälkeen määritellään muuttujan tyyppi, joka erotetaan muuttujan nimestä kaksoispisteellä. Esimerkiksi **`val`** `x: Int` **`=`** `1` alustaa kokonaislukuvakion x, jonka arvo on 1. Muuttujan tyyppimäärittelyn voi jättää myös kirjoittamatta, sillä Scala-kääntäjä osaa yleensä päätellä muuttujan tyypin asetetun arvon perusteella. Edellisen esimerkin voi siis halutessaan kirjoittaa muodossa **`val`** `x` **`=`** `1`, ja Scala osaa päätellä muuttujan olevan tyypiltään kokonaisluku. [5, Basics]



Kuva 3.1: Scalan luokkahierarkia

Muuttujat ovat myös staattisesti tyypitettyjä, joten muuttujan tyyppi ei voi muuttua ajon aikana. Scalassa myös funktiot ovat arvoja, joten niitä voidaan sijoittaa muuttujiin. Esimerkiksi `val add1 = (x: Int) => x + 1` luo *funktioliteraalin* ja sijoittaa sen muuttujaan nimeltä add1. Funktioita käsitellään tarkemmin luvussa 3.4. [4, Luku 1]

3.2 Tietotyypit

Scala on puhtaasti oliokieli, eli kaikki arvot ja muuttujat ovat olioita. Jokainen Scalan tietotyyppi perii luokan `Any` ja luokka `Nothing` on jokaisen tietotyypin alaluokka. Scalassa luokka voi olla arvoluokka tai viiteluokka. Arvoluokat perivät luokan `AnyVal` ja niihin ei voi sijoittaa null-arvoa. Viiteluokat perivät luokan `AnyRef`, joka on tyyppialias Javan luokalle `Object`. Jokaisen viiteluokan alaluokka on `Null`. Tietotyyppien hierarkia on kuvattu kuvassa 3.1. [4, Luku 5]

Alkeistietotyyppejä kuvaavat arvoluokat `Byte`, `Short`, `Int`, `Long`, `Char`, `Float`, `Double`, `Boolean` ja `Unit`. Kaikki paitsi `Unit` vastaavat Javan vastaavia käärelukkia. `Unit` on erityinen tietotyyppi, joka kuvastaa funktion tyhjää paluuarvoa. Sen voi ajatella vastaavan Javan void-avainsanaa. Merkkijonoja Scalassa kuvaa tietotyyppi `String`, joka on tyyppialias Javan merkkijonoa kuvaavalle tietotyypille. [4, Luku 5]

Scalan alkeistietotyypit muutetaan käännösvaiheessa primitiivityypeiksi, esimerkiksi

Scalan `Int` käännetään 32-bittiseksi sanaksi, aivan kuten Javan `int`. Tämä takaa yhteensopivuuden Java-kirjastojen kanssa. Se parantaa myös suorituskykyä, sillä primitiiviarvolle ei tarvitse allokoida kekomuistia ajon aikana. [4, Luku 6]

3.3 Kontrollirakenteet

3.3.1 Ehtolauseet

Ehtolauseita kirjoitetaan Scalassa *if/else*-lausekkeiden avulla. Lauseketta voi käyttää imperatiiviseen tyyliin eli jos ehto arvioituu todeksi, suoritetaan if-lohko, muuten else-lohko.

```
val x = true
if (x) {
    println("True")
} else {
    println("False")
}
// Tulostaa "True"
```

Scalan if-lausekkeella on myös arvo, eli if-lauseke palauttaa arvon. Arvon palauttavaa if-lauseketta voidaan käyttää myös funktionaalisessa ohjelmoinnissa. Esimerkiksi `val y = if (x >= 0) "positive" else "negative"` asettaa muuttujan `y` arvoksi joko merkkijonon "positive"tai "negative", riippuen muuttujan `x` arvosta. Huomattavaa on myös, että if-lauseke voidaan kirjoittaa yhdelle riville ilman aaltosulkeita. [6, Luku 2.1]

3.3.2 Silmukat

Silmukoita Scalassa on kolmenlaisia: **`while`**, **`do`** ja **`for`**. While-silmukka suorittaa ohjelmalohkoansa 0-n kertaa, kunnes annettu ehto arvioituu epätodeksi. Do-silmukka toimii vastaavalla tavalla, mutta sitä suoritetaan 1-n kertaa.

```
val x = false
while (x)
  println("while")
```

```
do
  println("do")
    while (x)
```

Yllä oleva esimerkki ei suorita while-silmukkaa kertaakaan, ja do-silmukan kerran. [6, Luku 2.5]

Scalan **for**-lauseke on monipuolisempi kuin monissa muissa kielissä. Sitä voi käyttää perinteiseen tapaan silmukkana, jolloin for-lausekkeen *generaattori* antaa määriteltyjä arvoja yksi kerrallaan muuttujan `i` kautta. Lauseketta voidaan käyttää myös usean generaattorin kanssa ja arvoja voidaan suodattaa lausekkeen sisällä. Kuten if-lauseke, myös for-lauseke voi palauttaa arvon. Seuraavaksi esimerkki kummastakin käyttötavasta.

```
for (i <- 1 to 10)
    println(i)
```

Silmukkana lauseketta voi käyttää yllä olevan esimerkin tapaan tulostamaan kaikki numerot yhdestä kymmeneen.

```
val res = for {
    x <- 1 to 10
    if x % 2 == 0
} yield (x * 2) // Vector(4, 8, 12, 16, 20)
```

Yllä olevassa esimerkissä luodaan kokonaisnumerovektori yhdestä kymmeneen, suodatetaan parilliset arvot, palautetaan jäljelle jääneet arvot kahdella kerrottuna ja asetetaan palautettu vektori muuttujaan `res`. Tällainen **for**-lauseke on yleinen funktionaalisessa Scalassa. [6, Luku 2.6]

3.4 Metodit ja funktiot

Jokainen operaatio Scalassa on metodikutsu. Myös tavanomaiset aritmeettiset operaattorit on toteutettu metodikutsuina. Esimerkiksi kahden kokonaisluvun yhteenlasku `1 + 2` on lyhyempi versio metodikutsusta `1.+(2)`. Scalassa on mahdollista määritellä funktio kahdella tavalla: olion metodiksi tai funktioliteraaliksi. Metodi voidaan määritellä **def**-

avainsanalla. Scalassa funktio palauttaa automaattisesti viimeisen lausekkeensa arvon, joten **`return`**-avainsanan käyttö on vapaaehtoista ja sitä käytetään vain harvoin. [6, Luku 1.4] [5, Basics]

3.4.1 Funktioliteraalit

Funktioliteraali on funktio, jota käsitellään kuten mitä tahansa muuttujaa. Yleensä funktioliteraali sijoitetaan muuttujaan tai annetaan parametriksi toiselle funktiolle. Kuten muillakin muuttujilla, funktioilla ja metodeilla on Scalassa aina tyyppi. Tyypin voi määritellä eksplisiittisesti, mutta monesti Scala osaa päätellä funktion tyypin. Seuraavassa esimerkissä määritellään kummallakin tavoilla funktio, joka kertoo, onko parametrina annettu kokonaisluku parillinen. Funktion tyyppi on siis `Int => Boolean`. [4, Luku 8]

```
def isEven1(x: Int): Boolean = x % 2 == 0
val isEven2: (Int => Boolean) = x => x % 2 == 0
```

3.4.2 Korkeamman asteen funktiot

Funktio voi ottaa parametriksi toisen funktion tai palauttaa funktion. Tällaisia funktioita kutsutaan *korkeamman asteen funktioiksi*. Korkeamman asteen funktiot ovat yksi funktionaalisen ohjelmoinnin kulmakivistä. Seuraavassa esitellään korkeamman asteen funktio `mapValue`, joka ottaa kaksi parametria: kokonaisluvun ja funktion kokonaisluvusta kokonaislukuun. `MapValue` soveltaa funktiota kokonaislukuun ja palauttaa tuloksen.

```
def mapValue(v: Int, m: Int => Int): Int = m(v)

def double(x: Int): Int = x * 2

mapValue(2, double)
mapValue(2, _ * 3)
```

Kuten yllä olevasta esimerkistä huomataan, annetaan `double`-funktio parametrinä ihan kuten mikä tahansa muukin muuttuja. Esimerkin viimeisellä rivillä käytetään funktioliteraalia `_ * 3`, joka on lyhyempi muoto funktioliteraalille `x => x * 3`. [6, Luku 12]

3.5 Luokat ja oliot

Scalassa on useita erilaisia luokka- ja oliotyyppejä sekä tapoja luoda instansseja luokista. Tässä luvussa esitellään pääpiirteittäin erilaiset luokkatyypit.

3.5.1 Class

Luokka voidaan määritellä Scalassa **class**-avainsanalla. Luokka voi olla tyhjä tai se voi sisältää metodeita ja muuttujia. Luokasta luodaan instanssi **new**-avainsanalla. Luokan sisältämät kentät voidaan määritellä konstruktorissa heti luokan nimen jälkeen suluissa. Konstruktorissa määritellyt kentät ovat oletuksena **private val**. Luokasta saadaan uusi instanssi **new**-avainsanan avulla. Ihmistä kuvaava luokka ja instanssi voidaan määritellä seuraavalla tavalla:

```
class Person(val name: String, var age: Int) {
    def isAdult = age >= 18
}
val p = new Person("Matti", 30)
```

Luokassa Person on kaksi kenttää: julkinen muuttumaton kenttä nimi ja julkinen muutettava ikä. Lisäksi luokassa on metodi, joka kertoo, onko henkilö täysi-ikäinen. [5, Classes]

3.5.2 Case class

Scalassa on myös erityisiä case-luokkia, joita käytetään yleensä kuvaamaan muuttumatonta dataa. Case-luokka määritellään kuten tavallinen luokka, mutta konstruktorissa määriteltävät kentät ovat oletuksena `public` **val**. Case-luokkaan voidaan määritellä metodeita samoin kuin tavalliseenkin luokkaan. Case-luokan instanssia luotaessa ei tarvitse käyttää **new**-avainsanaa. Puhelinmallia kuvaava case-luokka ja instanssi voidaan määritellä seuraavalla tavalla:

```
case class Phone(brand: String, model: String)
val p = Phone("Google", "Pixel4")
```

Scala-kääntäjä lisää case-luokkiin `toString`-, `hashCode`-, `equals`- ja `copy`-metodit oletustoteutuksilla. Lisäksi kääntäjä lisää *kumppaniolioon* `apply`-tehdasmetodin, jonka avulla uusien instanssien luonti tapahtuu. [4, Luku 15]

Tavallisia luokkia käytetään varsinkin oliotyylisessä ohjelmoinnissa, kun taas case-luokat ovat käytössä erityisesti funktionaalisessa Scalassa, kun halutaan käyttää muuttumattomia olioita. Tavallisten luokkien ja case-luokkien yhtäsuuruusvertailut eroavat toisistaan merkittävästi. Tavallisia luokkia verrataan viittauksen mukaan, eli osoittaako viittaus samaan olioon JVM:ssä. Case-luokkien yhtäsuuruutta verrataan olion kenttien arvon mukaan. [4, Luku 15]

3.5.3 Singleton-oliot

Useat muut oliokielet tarjoavat mahdollisuuden kirjoittaa staattisia luokkia tai staattisia kenttiä luokkiin. Scalassa tällaista staattisuutta ei ole, vaan vastaava toiminnallisuus mahdollistetaan erityisien *singleton*-olioiden kautta. Singleton-oliosta on ajon aikana olemassa vain yksi instanssi, ja Scala luo sen automaattisesti. Singleton-olion on mahdollista periä luokka. Viittaaminen olioon tapahtuu yksinkertaisesti olion nimellä. [5, Singleton objects]

Singleton-oliota voidaan kutsua myös kumppanioliokси, jos se on määritelty saman nimisen luokan yhteydessä. Kumppanioliot näkevät ilmentymiensä yksityiset kentät, ja ilmentymät näkevät kumppaniolionsa yksityiset kentät. Luvun 3.5.2 Phone-luokalle voidaan luoda kumppaniolio seuraavalla tavalla:

```
object Phone {
    def iphone11 = new Phone("Apple", "iPhone 11")
}
val iphone = Phone.iphone11
```

Tässä tapauksessa kumppanioliossa on vain yksi metodi, jonka avulla voidaan luoda helposti instansseja. [4, Luku 4]

3.5.4 Piirretyypit

Piirretyyppejä käytetään Scalassa määrittelemään olioiden tarjoamien palveluiden rajapintoja ja mahdollistamaan ohjelmakoodin uudelleenkäytettävyyttä. Uusi piirretyyppi luodaan **`trait`**-avainsanalla. Piirretyyppien on myös sallittua periä toisia piirretyyppejä tai luokkia. Myös singleton-olioihin on mahdollista liittää piirretyyppejä. Piirretyypin voi ajatella olevan Javan rajapintaluokan ja abstraktin luokan yhdistelmä. [4, Luku 6 ja 12]

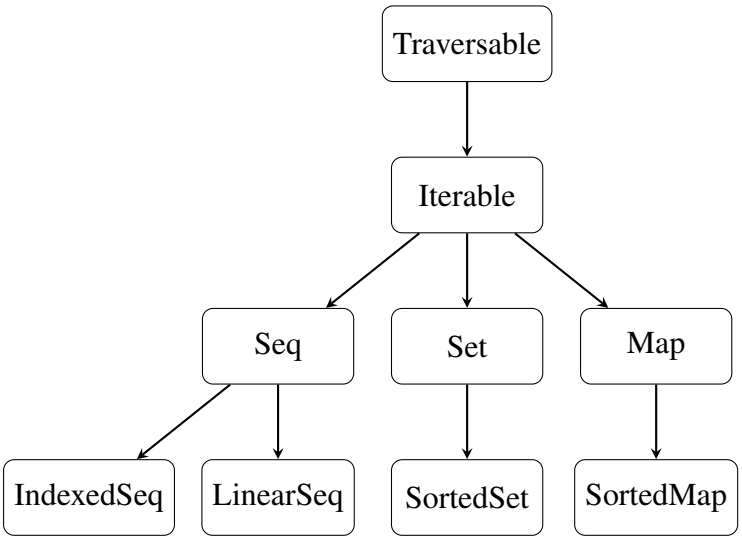
Piirretyypin jäseniksi voidaan määritellä metodeita tai muuttujia. Jäsenet voivat olla abstrakteja tai konkreetteja. Abstraktille jäsenelle tulee toteuttavan luokan tarjota toteutus, kun taas konkreeteille jäsenille ei. Yhdessä piirreluokassa voi olla sekä abstrakteja että konkreetteja jäseniä. Tietyissä erityistapauksissa metodi voidaan merkitä abstraktiksi, vaikka sille on annettu toteutus piirretyypissä. Piirretyyppeihin voidaan myös määritellä abstrakteja tyyppijäseniä **`type`**-avainsanalla. [6, Luku 10] [4, Luku 20]

4 Kokoelmat

Tutkimuksen kohteena on Scalan standardikirjaston kokoelmat versiosta 2.8 versioon 2.12. Standardikirjastossa on toteutus useimmille yleisimmille kokoelmaluokille kuten taulukoille, listoille, joukoille(`Set`) ja assosiaatiotauluille(`Map`). Scalan kokoelmaluokat ovat paketissa `scala.collection`. Muuttumattomat kokoelmat ovat alipaketissa `immutable` ja muuttuvat alipaketissa `mutable`. [7]

Muuttumattoman kokoelman sisältämät alkiot eivät voi muuttua alustamisen jälkeen. Tämä tarkoittaa, ettei alkioiden lisääminen, poistaminen tai uudelleenjärjestäminen ole mahdollista. Kokoelman muuttamista muistuttavat operaatiot, kuten `map`, `reverse`, `fold` ja kokoelmien yhdistäminen, palauttavat aina uuden kokoelman jättäen alkuperäisen kokoelman ennalleen. Käytännössä kuitenkaan aina ei kaikkia muuttumattoman kokoelman alkioita ei tarvitse kopioida, vaan tietorakenteet pyrkivät käyttämään hyväkseen rakenteellista jakamista parantaakseen suorituskykyä ja optimoidakseen muistinkäyttöä. Muuttuvissa kokoelmissa on nimensä mukaisesti mahdollista vaihtaa alkioiden järjestystä, lisätä ja poistaa alkioita kokoelmasta luomatta uutta kokoelmaa. [7] [4, Luku 22]

Kokoelmaluokat noudattavat piirreluokkien määrittelemää hierarkiaa, joka on paketissa `scala.collection`. Hierarkia on sama sekä muuttumattomille että muuttuville kokoelmille. Ylimpänä hierarkiassa on `Traversable`-piirreluokka, jolla on yksi välitön aliluokka, `Iterable`. `Iterable`:lla on kolme välitöntä aliluokkaa `Seq`, `Set` ja `Map`, jotka edustavat järjestettyjä kokoelmia, joukkoja ja assosiaatiotauluja. Kuvassa 4.1 näytetään lisäksi vielä muutama aliluokka. [7]



Kuva 4.1: Kokoelmien hierarkia

4.1 Järjestetyt kokoelmat

Suorituskykyvertailuun valittiin muuttumattomista kokoelmista `List`, joka tyypillinen rekursiivinen linkitetty rakenne funktionaalisessa ohjelmoinnissa, ja `Vector`, jossa alkion haku indeksin mukaan on tehokkaampaa. Muuttuvista kokoelmista valittiin `Array`, joka on kiinteän kokoinen taulukko, `ArrayBuffer`, joka on dynaamisesti kasvava taulukko ja `ListBuffer`, joka on dynaamisesti kasvava linkitetty rakenne. Kaikki kyseiset muuttuvat tietorakenteet ovat tyypillisiä olio-ohjelmoinnissa käytettyjä tietorakenteita.

`List` on abstrakti luokka, joka kuvaa yhteen suuntaan linkitettyä rekursiivista muuttumatonta listaa, jolla on kaksi konkreettista toteutusta: **`case class`** `::[B](head ↪ : B, tail: List[B])` ja **`case object`** `Nil`. Epätyhjää listaa kuvaavassa luokassa `::` on kaksi jäsentä: alkio, jota kutsutaan listan *pääksi*, ja *häntä* joka kuvaa listan muita alkioita. Tyhjää listaa ja samalla listan loppumista kuvaa singleton-olio `Nil`. `Nil` perii luokan `List[Nothing]`, jolloin saman olion on mahdollista kuvata minkä tahansa listan viimeistä alkiota. Esimerkiksi luvut 1 ja 2 sisältävä lista voidaan ilmaista näin: `1 :: 2 :: Nil`. Listan `head`-ja `tail`-operaatiot sekä listan alkuun lisääminen tapahtuvat vakioajassa. Aikakompleksisuus alkion hakemiselle indeksin perusteella

on lineaarinen. [8] [7]

`Vector` on taulukon ominaisuuksia jäljittelevä muuttumaton tietorakenne, joka on toteutettu puurakenteena, jonka solmut ovat korkeintaan 32-alkioisia taulukoita. Lehtisolmujen taulukot sisältävät vektorin varsinaisia alkioita, ja muut solmut sisältävät alemman tason taulukoita. Jos esimerkiksi alustetaan uusi 70-alkioinen vektori, on se kolmen taulukon puu, jossa juurisolmuna olevalla taulukolla on kolme lapsitaulukkoa, joista kahdessa on 32 alkiota ja yhdessä 6 alkiota. Tämä mahdollistaa taulukoiden rakenteellisen jakamisen eri vektori-instanssien kesken, mikä vähentää kopioimista ja muistinkäyttöä. Satunnaisten haku-, muokkaus- ja poisto-operaatioiden aikakompleksisuus on log(32, N), eli operaatioiden voidaan ajatella tapahtuvan lähes vakioajassa. [7] [9, Luku 4]

`Array` on Scalassa erityinen kokoelma, joka mahdollistaa primitiivityyppisten, viittaustyyppisten ja geneeristen alkioiden tallentamisen. Primitiivityyppisiä alkioita käsitellään ilman alkioiden käärimistä olioiksi. Esimerkiksi kokonaislukuja sisältävä `Array[Int]` kääntyy Javan `int[]`-taulukoksi. Kuten Javassa, taulukon koko ei voi muuttua alustamisen jälkeen. Verrattuna Javan taulukoihin `Array` tarjoaa kuitenkin huomattavasti enemmän operaatioita. Haku- ja muokkausoperaatiot tapahtuvat vakioajassa. Häntäoperaatio vie kuitenkin lineaarisen ajan, sillä jokainen alkio, pois lukien ensimmäinen, joudutaan kopioimaan uuteen taulukkoon. [7]

`ArrayBuffer` on dynaamisesti kasvava listarakenne, joka käyttää alkioiden tallentamiseen taulukkoa. Tästä johtuen useimmat operaatiot vastaavat aikakompleksisuudeltaan `Array`:ta. Lisäksi `ArrayBuffer` mahdollistaa alkion lisäämisen taulukon alkuun, loppuun sekä keskelle. Alkuun ja keskelle lisäämisen aikakompleksisuus on lineaarinen. Muut operaatiot tapahtuvat pääasiassa vakioajassa, poislukien täyteen taulukkoon lisääminen, joka vie lineaarisen ajan. [7]

`ListBuffer` on dynaamisesti kasvava muuttuva kokoelma, joka on toteutettu linkitettynä rakenteena. Poiketen `List`:stä, `ListBuffer` mahdollistaa alkioiden lisäämisen rakenteen alkuun ja loppuun vakioajassa. Satunnaisen alkion haku- ja muokkausoperaa-

tiot sekä rakenteen keskelle lisääminen ovat aikakompleksisuudeltaan lineaarisia. Myös häntäoperaatio vie lineaarisen ajan, sillä kaikki hännän alkiot täytyy kopioida uuteen tietorakenteeseen. [7]

4.2 Suorituskykyvertailut

Tutkielmassa on käytetty kokoelmien suorituskykymittauksia kahdesta eri lähteestä: Toby Hobsonin[10] ja Li Haoyi:n[11] mittauksista. Järjestettyjen kokoelmien tapauksessa kiinnostuksen kohteena on yleisimpien operaatioiden suorituskyky: kokoelman luominen, iterointi, alkion satunnainen haku sekä lisääminen ja häntäoperaatio.

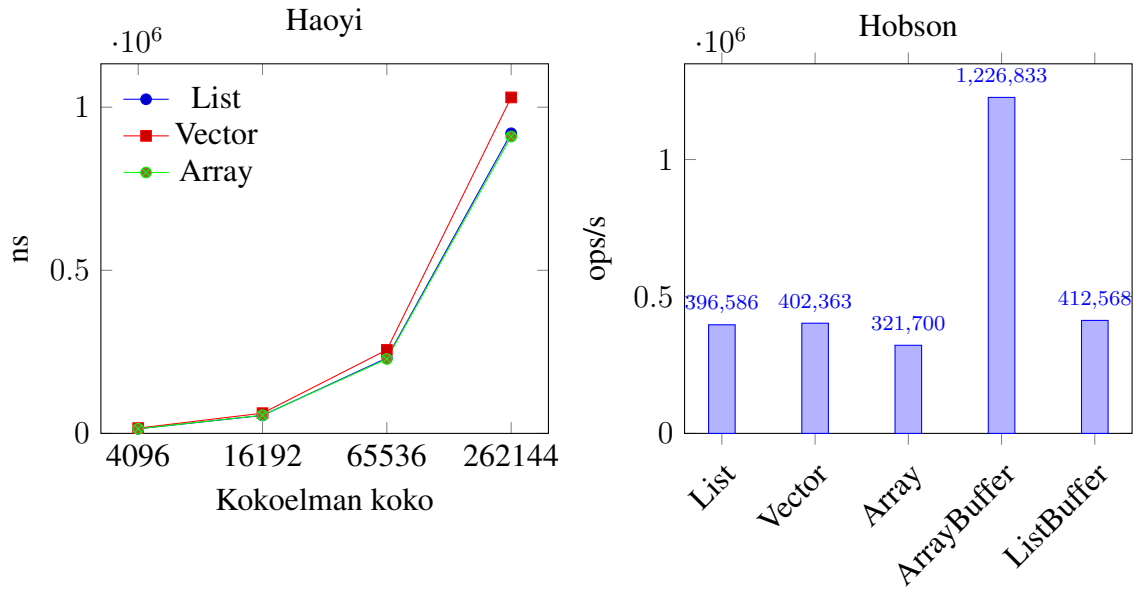
Hobson[10] on käyttänyt mittauksien toteutuksessa OpenJDK-tiimin kehittämää *Java Microbenchmark Harness:ia* (JMH), joka sisältää useita työkaluja suorituskykymittausten tekemistä varten. Yhden testausiteraation aikana testataan 1000-alkioinen kokoelma. Jokaista mittausta varten on viisi JVM-instanssia, joita jokaista lämmitetään kymmenen iteraation verran, ja itse mittaus sisältää kymmenen iteraatiota. Mittausten yksikkönä on operaatioiden määrää sekunnissa (ops/s), jolloin suuri luku tarkoittaa hyvää suorituskykyä. [9]

Haoyi ei käytä mittauksissaan[11] JMH:ta. Suorituskykyä on mitattu eri kokoisilla kokoelmilla, ja kaavioihin on valittu kyseisen operaation kannalta mielekkäät koot kokoelmille. Ennen jokaista yksittäistä mittausta pakotetaan virtuaalikoneen roskien keräys, ja virtuaalikonetta lämmitetään ajamalla yksi iteraatio mittausta, jonka jälkeen tehdään itse mittaus. Yksi mittauskierros sisältää yhden mittauksen jokaiselle kokoelman ja koon yhdistelmälle. Kaikki mittaukset sisältäviä mittauskierroksia tehdään yhteensä seitsemän kappaletta, joista lasketaan keskiarvo. Mittausten yksikkö on operaation suorittamiseen kuluva aika nanosekunneissa, eli pieni arvo tarkoittaa hyvää suorituskykyä.

4.2.1 Iterointi

Iteroinnissa kokoelman jokainen alkio käydään läpi. Operaation oletettu aikakompleksisuus on lineaarinen. Iterointioperaatio ei muokkaa iteroitavaa kokoelmaa, joten muuttuvien ja muuttumattomien kokoelmien suorituskyvyissä ei pitäisi olla merkittäviä eroja. Lähteenä käytettävissä suorituskykymittauksissa ei tutkittu rinnakkaisten kokoelmien suorituskykyä, joten iterointi tapahtui järjestyksessä.

Haoyin mittauksissa[11] `Array`:n iteroiminen **`while`**-silmukalla sekä `ArrayBuffer`:n iteroiminen tapahtui mittaamattoman lyhyessä tai jopa negatiivisessa ajassa, joten on kyseisiä mittauksia ei otettu huomioon. Kaaviosta 4.2 voidaan huomata, että suorituskyvyissä ei ollut suuria eroavaisuuksia, paitsi Hobsonin mittauksissa[10] `ArrayBuffer` oli 3-4 kertaa muita kokoelmia nopeampi. Haoyin mittauksissa[11] merkittävää eroa ei ollut, vaikkakin `Array` oli hieman muita nopeampi.



Kuva 4.2: Iterointi

4.2.2 Satunnainen haku

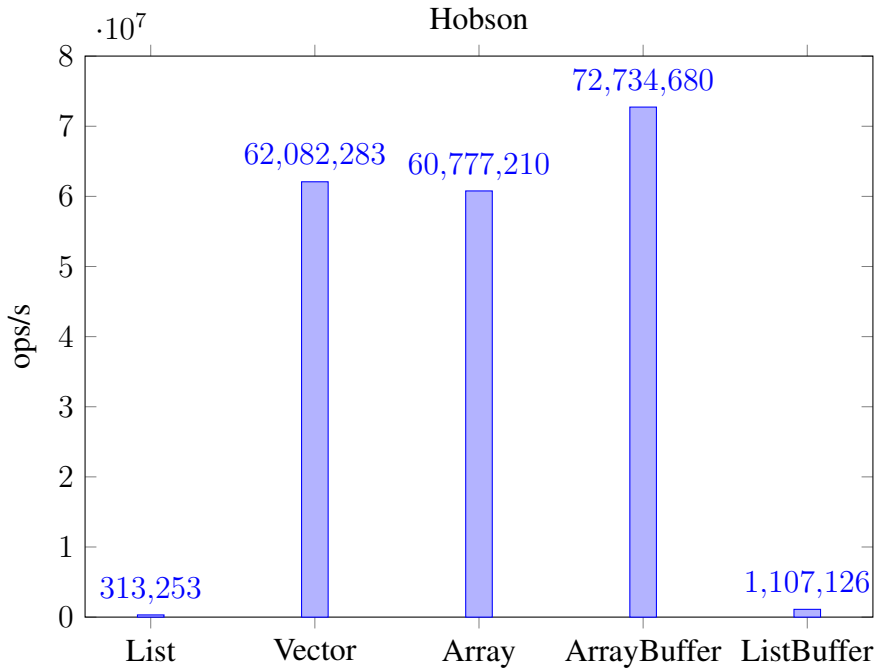
Nimensä mukaan satunnainen haku tarkoittaa alkion hakemista kokoelmasta satunnaisen indeksin perusteella. Linkitetyissä rakenteissa haku vaatii alkioiden käymistä läpi peräkkäin, kunnes haettu indeksi saavutetaan, jolloin aikakompleksisuus on lineaarinen. Taulukkopohjaisissa tietorakenteissa vastaavaa rajoitetta ei ole, vaan alkion muistiosoite voidaan laskea suoraan indeksin perusteella ja haun voi olettaa tapahtuvan vakioajassa. Tästä syystä on perusteltua olettaa taulukkojen olevan huomattavasti linkitettyjä rakenteita suorituskykyisempiä.

Hobsonin mittaukset[10] kaaviossa 4.3 tukevat tätä hypoteesia. `ListBuffer` sekä `List` ovat jopa useita satoja kertoja hitaampia kuin taulukoihin pohjautuvat kokoelmat. Hieman yllättäen `ListBuffer` on kuitenkin noin 3,5 kertaa nopeampi kuin `List`, vaikka `ListBuffer` käyttää sisäisesti alkioiden tallentamiseen `List`:iä. Taulukon ominaisuuksia mukaileva `Vector` ja `Array` olivat suorituskyvyltään lähes identtisiä.
`ArrayBuffer` oli noin 15 % nopeampi kuin muut taulukkopohjaiset kokoelmat, mikä on hieman yllättävää, sillä sisäisesti alkiot tallennetaan käyttäen `Array`:ta.

4.2.3 Kokoelman luominen

Kokoelmia rakennetaan useasti lisäämällä alkioita yksi kerrallaan kokoelmaan. Joskus kokoelma luodaan toisen kokoelman tai kokoelmatyypin alkioista. Taulukkoja (Scalassa `Array`) luodessa täytyy yleensä päättää, miten suuri taulukko luodaan. Mikäli täyteen taulukkoon halutaan lisätä alkioita, täytyy luoda uusi suurempi taulukko ja kopioida kaikki alkiot alkuperäisestä taulukosta uuteen taulukkoon ja lisätä uusi alkio uuteen taulukkoon. Tästä syystä taulukon luominen alkio kerrallaan johtaa neliölliseen aikakompleksisuuteen.

Haoyin vertailuissa[11] (kaavio 4.4) on testattu useita tapoja kokoelmien luomiseen. `Array:+` kuvastaa tilannetta, jossa luodaan kokoelma lisäämällä alkio täyteen taulukkoon. Oletetusti sen suorituskyky on koko vertailun huonoin. `Array`-prealloc kuvaa ti-



Kuva 4.3: Satunnainen haku

lannetta, jossa luodaan valmiiksi oikean kokoinen taulukko, ja sen suorituskyky on vertailun paras.

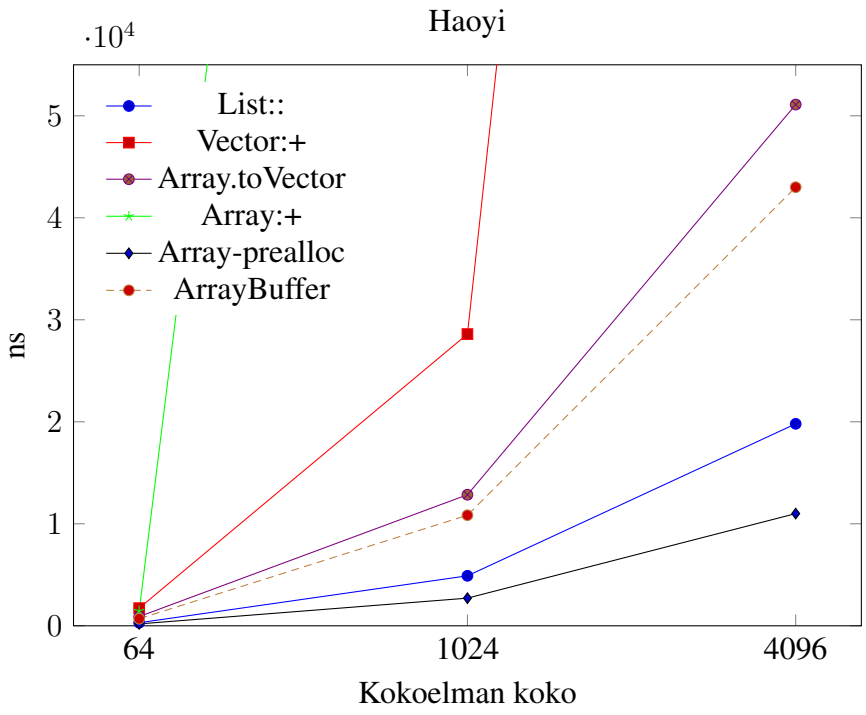
`List::` rakentaa linkitetyn rakenteen lisäämällä alkioita linkitetyn listan alkuun, eli alkio lisätään `::`-olion pääksi ja hännäksi lisätään lista. Operaation aikakompleksisuus on vakio. [7] Hayoin mittaukset osoittavat tämän todeksi, ja ainoastaan `Array`-prealloc oli `List`:aa suorituskykyisempi.

Kun `ArrayBuffer`:n sisäinen taulukko tulee täyteen, luodaan uusi, kaksi kertaa alkuperäistä suurempi, taulukko ja vanhan taulukon arvot kopioidaan uuteen. Valtaosa lisäyksistä tapahtuu vakioajassa, mutta lisäys sisäisen taulukon ollessa täysi vie lineaarisen ajan. Suorituskyky on odotetusti linkitettyä rakennetta huonompi, mutta hieman `Array.toVector`:ia parempi.

`Vector:+` aikakompleksisuuden ilmoitetaan olevan käytännössä vakio [7], mutta vakion voi olettaa olevan suurempi kuin `ArrayBuffer`:lla, sillä osa puurakenteen solmuista joudutaan kopioimaan. Suurempi vakiokerroin näkyy mittauksissa selvästi. Eri-

tyisesti luotaessa yli suuria kokoelmia, kasvaa kopioinnin tarve, kun sisäiseen puurakenteeseen lisää tasoja. Mittauksissa suorituskyky heikkenee merkittävästi yli 1024 alkion kokoelmissa.

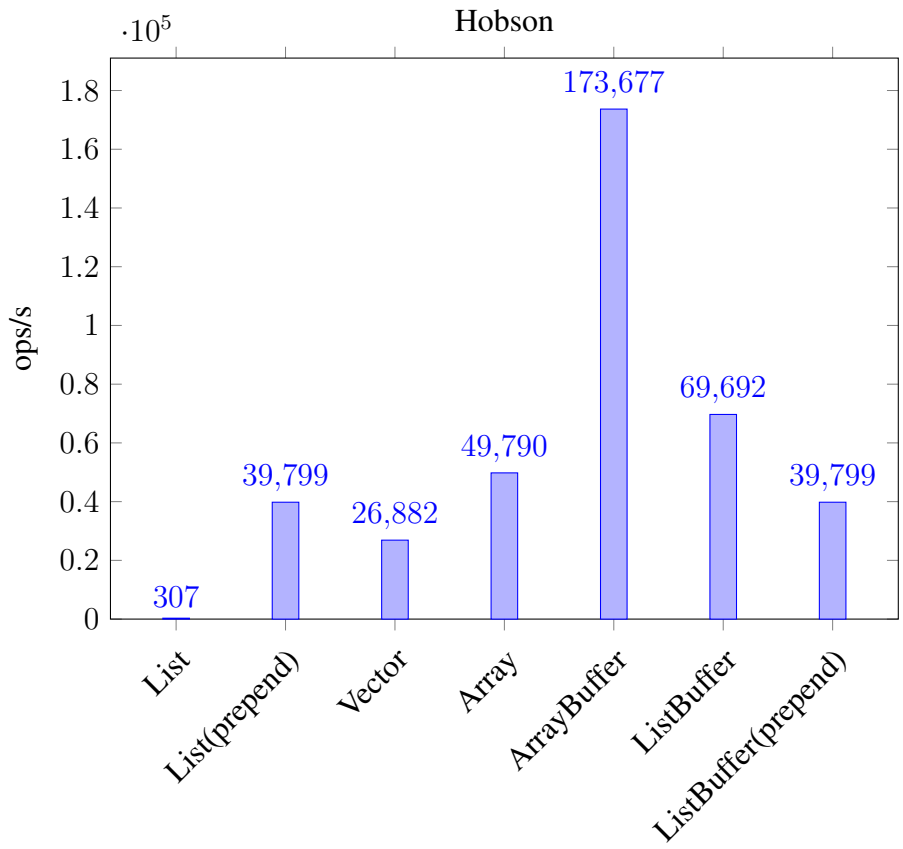
Suorituskykyisempi tapa luoda `Vector` on luoda toinen kokoelma ja kutsua sen `toVector`-metodia. Haoyin[11] mittausten `Array.toVector` on tilanne, jossa luodaan `Array` pre-alloc-tyylisesti ja luodaan sen pohjalta `Vector`. Suorituskyky on huomattavasti parempi kuin lisäämällä alkioita yksi kerrallaan.



Kuva 4.4: Kokoelman luominen

4.2.4 Kokoelman loppuun lisääminen

Asetelma Hobsonin mittauksessa[10] kokoelman loppuun lisäämisestä on hieman samankaltainen kuin Haoyin[11] kokoelman luomisen mittauksessa. Muuttumattoman linkitetyn listan (`List`) tapauksessa täytyy jokainen alkio kopioida uuteen kokoelmaan ja uusi alkio lisätä juuri luodun kokoelman viimeiseksi. Operaation aikakompleksisuus on lineaarinen. Kaaviosta 4.5 voidaan huomata, että suorituskyky on selvästi vertailun heikoin.



Kuva 4.5: Kokoelman loppuun lisääminen

`List`-prepend tarkoittaa tilannetta, jossa alkio lisätään rakenteen alkuun, ja kokoelman järjestys käännetään. Tällä tavoin suorituskyky on jopa yli 100 kertaa parempi. Parannus suorituskyvyssä on hieman yllättävä, sillä järjestyksen kääntävän operaation aikakompleksisuus on vakio. Eroa saattaisi selittää `reverse`-metodin toteutus, jossa käytetään while-silmukkaa, kun taas `:+`-metodissa käytetään erityistä `CanBuildFrom`-oliota.

Muuttuvasta linkitetystä rakenteesta, `ListBuffer`:sta on mittaus kummallakin edellisessä kappaleessa mainitulla tavalla. Suoraan kokoelman loppuun lisäämisellä saavutetaan paras suorituskyky, ja se on jopa yli 70 % parempi kuin lisäämällä alkuun ja kääntämällä kokoelman järjestys. Prepend-tyylillä suorituskyky on identtinen `List`-prepend kanssa.

Taulukoista `ArrayBuffer` suoriutuu selvästi parhaiten ollen yli kolme kertaa pa-

rempi kuin seuraavaksi paras taulukkorakenne `Array`, vaikka kyseisessä mittauksessa luodaan valmiiksi oikean kokoinen `Array`. Tulos on yllättävä, sillä tuloksien voisi olettaa olevan lähes identtisiä. Mittauksen [10] toteutuksessa on käytetty `Array`-luokan metodia `zipWithIndex`, joka muuttaa `Array`:n `WrappedArray`:ksi. Tämä voisi osaltaan selittää tulosta.

Aikaisemman testin tavoin `Vector` suoriutuu tästäkin mittauksesta huonommin kuin muut taulukkomaiset kokoelmat ollen noin 85 % huonompi kuin `ArrayBuffer`. Syynä lienee kopioinnin tarve jokaisen alkion lisäämisen yhteydessä.

4.2.5 Häntäoperaatio

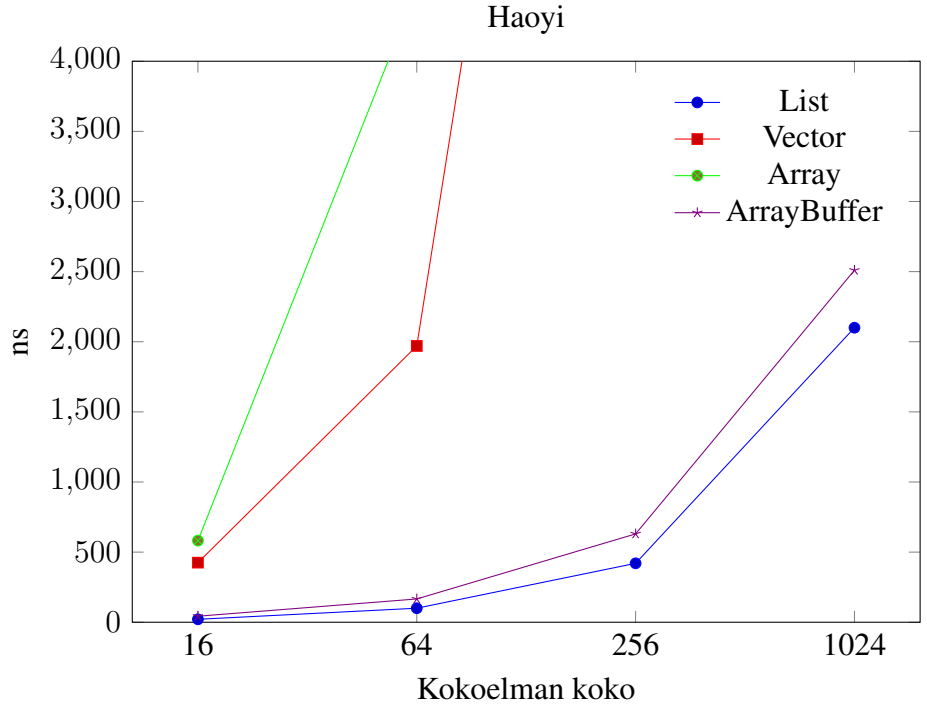
Häntäoperaatio palauttaa kokoelman, jossa on kaikki paitsi ensimmäinen alkio. Muuttuvissa kokoelmissa tämä pakottaa yleensä kopioimaan kokoelman hännän uuteen kokoelmaan, jolloin aikakompleksisuus on lineaarinen. Muuttumattomissa kokoelmissa kopioimista voidaan vähentää tai jopa välttää kokonaan hyödyntämällä rakenteellista jakamista. Muuttumattoman `List`:n häntäoperaation aikakompleksisuus on vakio, sillä linkitetyn rakenteen häntään saadaan viittaus yksinkertaisesti palauttamalla ensimmäisen alkion seuraaja. `Vector`:n häntäoperaation aikakompleksisuudeksi ilmoitettu on käytännössä vakio[7], vaikka todellisuudessa joitain alkioita täytyy kopioida.

Haoyin[11] häntäoperaation suorituskykyä mittaavassa tutkimuksessa aloitetaan n-kokoisesta kokoelmasta ja tehdään häntäoperaatiota jäljelle jäävälle kokoelmalle, kunnes jäljellä on tyhjä kokoelma. Mittauksen tulos on kulunut aika, jolloin pieni arvo tarkoittaa hyvää suorituskykyä. Kaaviossa 4.6 on mittauksen tulokset.

`List` suoriutuu mittauksissa oletetusti parhaiten. `ArrayBuffer` suoriutui varsin yllättäen vain hieman `List`:iä huonommin, vaikka aikakompleksisuuden voisi olettaa olevan lineaarinen. `Vector` suoriutui edellä mainittuja kokoelmia huomattavasti huonommin.

Selvästi huonoiten suoriutui kuitenkin `Array`. Tulos on oletettu, sillä jokaisella hän-

täoperaatiolla kaikki paitsi taulukon ensimmäinen alkio täytyy kopioida uuteen taulukkoon. `Vector` on hieman `Array`:ta parempi, mutta sen suorituskyky on 256 alkion kokoelmalla jo noin 20 kertaa huonompi kuin kahdella suorituskykyisimmällä kokoelmalla.



Kuva 4.6: Häntäoperaatio

4.3 Johtopäätökset

Iteroinnissa kaikki kokoelmat suoriutuivat verrattain tasavertaisesti, pois lukien `ArrayBuffer`:n loistava suorituskyky muihin verrattuna. Muiden kokoelmien välistä vaihtelua iteroinnin suorituskyvyssä saattaisi selittää esimerkiksi kääreluokan käyttö `Array`:n yhteydessä tai `Vector`:n puumainen rakenne, joka johtaa muutamaan epäsuoraan viittaukseen.

Satunnaisessa haussa taulukkopohjaiset kokoelmat olivat odotetusti suorituskykyisempiä kuin linkitetyt rakenteet, riippumatta siitä oliko kyseessä muuttuva vai muuttumaton kokoelma. Taulukkopohjaisten kokoelmien välisiä eroja selittää mahdollisesti alkioiden

löytymättömyys prosessorin välimuistista, jota todennäköisesti tapahtuu enemmän puumaisesti toteutetun `Vector`:n kohdalla kuin taulukkona toteutetussa `ArrayBuffer`:ssa.

Kokoelman luomisen ja loppuun lisäämisen suorituskyvyissä oli huomattavia eroja riippuen käytettävästä metodista. Taulukkopohjaisen kokoelman luominen `:+`-metodin avulla oli `Array`:n tapauksessa satoja kertoja hitaampaa kuin muilla tavoilla. Käyttötarkoitukseen sopivan kokoelman ja operaation valitsemisen merkitys on siis todella merkittävä.

Myöskään häntäoperaation suorituskyvyssä muuttuvien ja muuttumattomien kokoelmien välille ei muodostunut merkittävää eroa `List`:n ja `ArrayBuffer`:n suoriutuessa hyvin sekä `Array`:n ja `Vector`:n suoriutuessa heikommin.

Muuttumattomista kokoelmista `List` tarjoaa hyvän suorituskyvyn, kunhan tarvetta satunnaiselle haulle tai kokoelman loppuun lisäämiselle ei ole. `Vector` puolestaan tarjoaa hyvän suorituskyvyn satunnaisessa haussa, mutta häntäoperaatiota tulisi välttää sekä kokoelman rakentamiseen täytyy kiinnittää huomiota. Muuttuviin taulukkokokoelmiin verrattuna erityisesti loppuun lisäämisen ja kokoelman luomisen suorituskyvyssä joudutaan maksamaan pientä hintaa muuttumattomuudesta.

Kokonaisvaltaisesti parhaan suorituskyvyn tarjoaa muuttuva `ArrayBuffer`, joka suoriutuu kaikista operaatioista loistavasti, poislukien kokoelman luominen, jossa suorituskyky on keskinkertainen. Muuttuva linkitetty rakenne `ListBuffer` suoriutuu yleisesti hieman paremmin kuin vastaava muuttomaton kokoelma `List`, mutta ero on pieni.

Tutkielmassa käytettyjen lähteiden perusteella ei noussut esiin säännönmukaisia eroavaisuuksia muuttuvien ja muuttumattomien kokoelmien suorituskyvystä. Kuhunkin käyttötarkoitukseen sopivan kokoelman valitseminen on suorituskyvyn kannalta todella tärkeää. Pahimmillaan käyttötarkoitukseen epäoptimaalisen kokoelman valinta voi johtaa yli sata kertaa huonompaan suorituskykyyn verrattuna käyttötarkoitukseen sopivaan kokoelmaan.

5 Yhteenveto

Tutkielmassa pyrittiin selvittämään funktionaalisen ohjelmointiparadigman vaikutuksia suorituskykyyn verrattuna imperatiiviseen paradigmaan. Vaikutuksia suorituskykyyn pyrittiin tarkastelemaan vertailemalla järjestettyjen muuttuvien ja muuttumattomien kokoelmaluokkien suorituskykyjä toisiinsa. Suorituskykyä tarkasteltiin tutustumalla lähteenä käytettyihin suorituskykymittauksiin. Tarkasteluun valittiin kullekin paradigmalle tyypillisiä kokoelmaluokkia. Johtopäätöksiä kokoelmaluokkien suorituskyvystä tehtiin vertailemalla lähteiden mittauksien tuloksia.

Selviä eroja muuttuvien ja muuttumattomien kokoelmien suorituskyvylle ei havaittu, vaikka muuttuvat kokoelmat näyttäisivätkin suoriutuvan useissa operaatioissa hieman muuttumattomia paremmin. Suorituskyvyn kannalta tärkein tekijä oli kuitenkin kuhunkin käyttötarkoitukseen sopivan tietorakenteen valinta. Tarkempien säännönmukaisuuksien selvittämiseksi tulisi mittauksia tehdä useamman kokoelmaluokan, operaation ja skenaarion yhdistelmistä. Myös järjestämättömien kokoelmien, kuten joukkojen ja assosiaatiotaulujen, suorituskykyä tulisi vertailla.

Kokonaisvaltaista johtopäätöstä funktionaalisen paradigman suorituskykyvaikutuksista imperatiiviseen paradigmaan verrattuna ei tämän tutkielman tulosten perusteella voi tehdä. Jatkotutkimuskohteena kokoelmien osalta voisivat olla laiskat ja rinnakkaiset kokoelmat. Muita tutkielman tekemisen aikana esiin nousseita jatkotutkimuksen aiheita ovat hahmontunnistus, sulkeumat, funktioiden osittainen soveltaminen sekä silmukoiden ja rekursion suorituskyvyn vertailu.

Lähdeluettelo

[1] M. Gabbrielli ja S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2010.

[2] M. L. Scott, *Programming Language Pragmatics, 4th Edition*. Morgan Kaufmann, 2015.

[3] P. Chiusano ja R. Bjarnason, *Functional Programming in Scala*. Manning, 2015.

[4] M. Odersky, L. Spoon ja B. Venners, *Programming in Scala, 3rd Edition*. Artima, 2016.

[5] Tour of Scala, url: `https://docs.scala-lang.org/tour/tour-of-scala.html` (viitattu 03. 01. 2020).

[6] C. Horstmann, *Scala for the Impatient, 2nd Edition*. Addison-Wesley Professional, 2017.

[7] Scala Collections v2.8-2.12, url: `https://docs.scala-lang.org/overviews/collections/` (viitattu 12. 02. 2020).

[8] Scala API v2.12, url: `https://www.scala-lang.org/files/archive/api/2.12.x` (viitattu 03. 01. 2020).

[9] V. Theron ja M. Diamant, *Scala High Performance Programming*. Packt, 2016.

[10] T. Hobson. (2016). Scala Collections Performance, url: `https://www.tobyhobson.com/posts/scala/collections-performance/` (viitattu 18. 02. 2020).

[11] L. Hayoi. (2016). Benchmarking Scala Collections, url: http://www.lihaoyi.com/post/BenchmarkingScalaCollections.html (viitattu 18. 02. 2020).