TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Brandon Rauba

Scala 3 jõudlustestimine

Bakalaureusetöö (9 EAP)

Juhendaja(d): Kalmer Apinis

Tartu 2023

Scala 3 jõudlustestimine

Lühikokkuvõte:

Laiatarbe programmeerimiskeeled toovad endaga kaasa palju võimalusi koodi struktureerimiseks. Uue keele või keele uue abstraktsioonivõimaluse kasutamisel võib tekitada kahtlusi, kas see teeb koodi aeglaseks. Antud bakalaureusetöös kontrollitakse, kas Scala 3 abstraktsioonivõimalused on aeglased või mitte. Töö raames kirjutatakse teste, millega saab võrrelda uusi funktsioone vanadega. Testide tulemused on kujutatud graafikutena, mille põhjal saab teha otsuse vastava võimaluse kasutamise osas. Lõpuks võetakse tulemused kokku ja leitakse vastus küsimusele, kas Scala 3 uued lahendused on kiiremad või aeglasemad.

Võtmesõnad:

Jõudlustestimine, kiirus, testimine

CERCS: P175, Informaatika, süsteemiteooria

Scala 3 benchmarking

Abstract:

High-level programming languages bring along numerous possibilities for structuring code. When using a new language or exploring a new abstraction feature within a language there may be doubts about the functions making the code slower. This bachelor's thesis controls if Scala 3 abstraction features are slow or not. During the work process there will be written tests that can be used to compare to older functions. Tests results will be converted on graphs to make a decision which function to use. In the end the results will be summarised and the answer to the question if Scala 3 new updates are slower or faster will be answered.

Keywords:

Benchmarking, speed, testing

CERCS: P175, Informatics, system theory

Sisukord

1. Sissejuhatus

Tänapäeva maailmas, kus keeli, millega programmeerida on tohutult palju ja keeltes olevaid meetodeid kordades rohkem, on väga raske valida, milline keel on koodi kirjutamiseks kõige otstarbekam. Programmeerimiskeele kiirus ja jõudlus on üheks näitajaks, mille põhjal valitakse, mis keeles programmi kirjutada [1]. Sellepärast on antud töö suunatud sellele, et otsustada, millised keele uuendusi kasutusele võtta.

Käesoleva töö uuritavaks keeleks on Scala 3. Keele valik tuleneb sellest, et Tartu Ülikooli poolt kirjutatud Java baitkoodi analüsaator Põder on selles keeles ja on kahtlusi, et uued meetodid Scala 3-s pruugi olla kiiremad vanemast versioonist. Töö eesmärk ongi teha kindlaks, kas kahtlused Scala 3 kiiruse osas on õigustatud või mitte.

Esimeses peatükis antakse ülevaade, milliseid vahendeid on töös kasutatud. Lisaks sõnastatakse täpsemalt lahti, mis eesmärki antud töö täpsemalt täidab. Peatükis 2.1 kirjeldatakse, mida kujutab endast jõudlustest. Peatükk 2.2 annab lühikese ülevaate Põdrast. Peatükid 2.3 ja 2.4 kirjeldavad vastavalt Scala 3 keelt ja ScalaMeter raamistiku valikut. Viimane alapeatükk seletab lähemalt, mis probleemile antud töö lahendust otsib.

Teises peatükis kirjeldatakse testide ülesehitust ja loogikat. Esimene alapeatükk annab üldise ülevaate, mis põhimõtteid kõikide testide kirjutamisel on kasutatud. Peatükid 3.1.1, 3.1.2 ja 3.1.3 kirjeldavad nende põhimõtete olulisusi. Järgmised alapeatükid kirjeldavad iga testi olemust ja mida vastava testiga mõõta taheti. Tulemuste ja arutelu peatükis antakse ülevaade testide tulemustest ja arutatakse mõõdetud tulemuste üle.

2. Taust

Selles peatükis tutvustatakse, mis asi on jõudlustest ja antakse ülevaade kasutatud vahenditest ja tarkvaradest. Põhiline eesmärk on põhjendada, miks kasutati just ScalaMeter raamistikku testide kirjutamisel. Väiksemad eesmärgid on tekitada arusaama, mis on jõudlustestimine ja kuidas on need olulised loodava lahenduse loomiseks.

2.1 Jõudlustest

Jõudlustestimine on oluline protsess, mis on vajalik tarkvaraarenduse ja süsteemi haldamise jaoks. See aitab hinnata süsteemi, rakenduse või tarkvara toimimist erinevates tingimustes ning tuvastada selle jõudlusega seotud kitsaskohti ja puudujääke[2]. Käesoleva töö peamiseks vaatepunktiks on vaadata, kas uuemad funktsioonid ja süntaksid on efektiivsuse poolest paremad, kui seda on enne uuendusi kasutusel olevad meetmed.

Jõudlustestid jagunevad kahte tüüpi. Testid, mis on mõeldud kogu programmi testimiseks nimetatakse makro jõudlustestideks ehk inglise keeles *macro benchmarking*. Teised testid, mida kasutati ka antud uurimustöös, on mõeldud eraldiseisvate koodijuppide testimiseks. Neid teste nimetatakse mikro jõudlustestideks ehk inglise keeles *micro benchmarking*. Mikro jõudlustestide juures tuleb olla eriti tähelepanelik, et kood oleks võimalikult puhas, kuna kompileerides koodi võivad kõik read, mis näiteks jäävad kasutamata, muuta jõudlustesti tulemusi[3].

2.2 Põder

Põder on staatilise analüüsi raamistik, mis analüüsib Java baitkoodi. Põdra eesmärk on tuvastada programmis sisalduvad vead ilma programmi käivitamata. Kuna Põder analüüsib Java baitkoodi, siis on võimalik sellega ka analüüsida kõiki JVM-i (*Java Virtual Machine*) baitkoodi programme. Analüsaatori loogika on üles ehitatud nii, et luuakse graaf, kus tipud on asukohad kontrollitavas programmis ja kaared nende vahel kirjeldavad instruktsioonide plokke, kuidas graafil edasi liikuda.

Põder on kirjutatud Scala keeles ja on avatud lähtekoodiga. Antud töö eesmärk on uurida, kas algselt kasutatud Scala funktsioonid on kiiremad või aeglasemad kui uue Scala 3 funktsioonid.

Lisad I alt on võimalik näha lühidemo Põdra tööst, kuid rohkem süvitsi antud töös Põdra loogikaga ei minda.

2.3 Scala

Scala on Java sarnane programmeerimiskeel, mis seob objekt-orienteeritud ja funktsionaalse programmerimise. Scala on loodud töötama koos teiste puhtalt objekt-orienteeritud programmeerimiskeeltega nagu Java. Seetõttu on ka võimalik Scala klassidega kutsuda välja Java meetodeid, luua Java objekte ilma lisakoodi kirjutamata. Scala esimene versioon avaldati 2003. aasta novembris, kuid keele arendustega alustati juba aastal 2001[4].

Antud töö raames võrreldakse Scala teise versiooni ja kõige uuema ehk kolmanda versiooni võimaluste (i.k. *feature*) võimekust.

2.4 ScalaMeter

ScalaMeter on üks võimalikest raamistikest, mida kasutada, et teha jõudlusteste erinevate väiksemate koodijuppide peal. ScalaMeter on mõeldud JVM platvormidele, mis tähendab, et seda saab kasutada nii Java kui ka Scala puhul[5].

Antud töö raames valiti ScalaMeter kuna erinevate materjalide ja blogi arvustuste põhjal osutus see raamistik kõige kergemini implementeeritavaks. ScalaMeter on suuteline tagastama erinevaid andmeid teostatud testi kohta. Neid kahte argumenti arvesse võttes valitigi selleks tööks see raamistik.

Lisaks ScalaMeterile prooviti ka SBT-ga (*Scala Build Tool*) koodi testida. SBT implementeerib JMH (*Java Microbenchmark Harness*) raamistiku SBT-sse, mis teoorias peaks laskma jõudlusteste kirjutada ja käivitada otse SBT projektist, kuid testide kirjutamise perioodil ei õnnestunud seda korrektselt implementeerida. Seetõttu jäi testide kirjutamiseks ScalaMeter raamistik.

2.5 Probleemi kirjeldus

Probleemi algus tuleneb sellest, et algselt kui analüsaator Põder kirjutati ei olnud veel Scala 3 väljastatud. Peale seda kui Scala 3 avalikustati, on tehtud Põdra koodis muudatusi, kuid siiani on ainult loodetud selle peale, et uuendused ka päriselt kiiremad on. Seetõttu otsustasid töö autor ja töö juhendaja, et oleks vaja testida neid funktsioone ja süntakseid, mida Scala 3 nüüd pakub. Töö eesmärk ei olnud tõestada, et tehtud uuendused on kiiremad kui vanemad funktsioonid, vaid leida kinnitust sellele, kas edaspidi Põdra arenduses on mõttekas kasutada uusi funktsioone. Uute funktsioonide kasutamine tähendab seda, et mõningal määral on süntaks tehtud küll lihtsamaks, et koodi kirjutada, kuid koodi tööle

saamiseks on vaja teha mõningaid muudatusi, et kood kompileeruks ja töötaks vastavalt vajadusele.

3. Testide ülesehitus

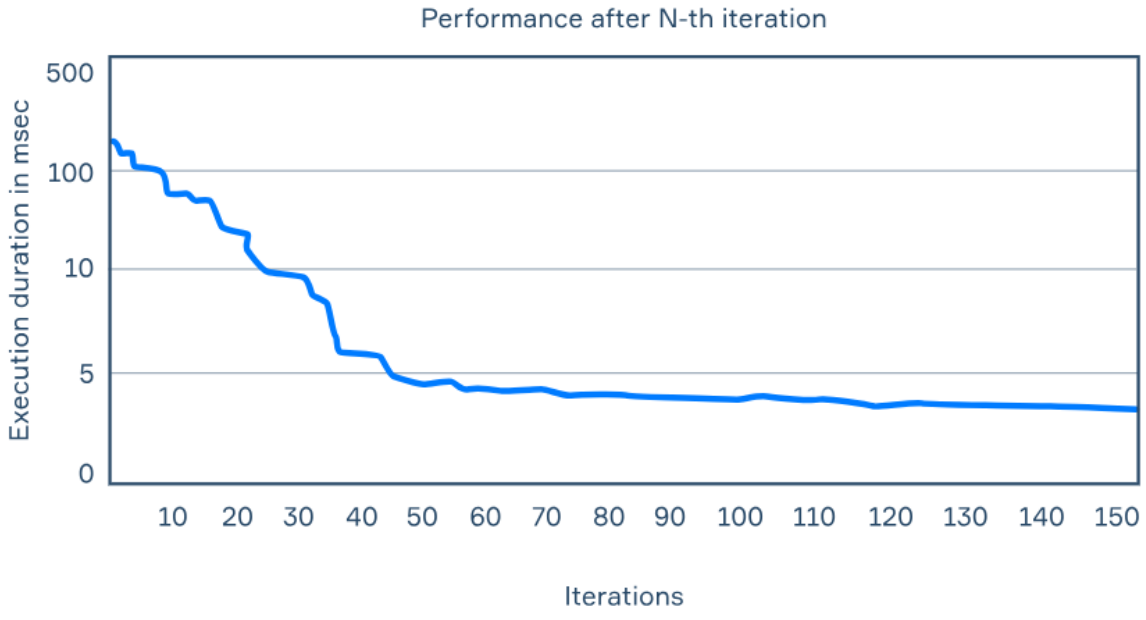
Järgmises peatükis tutvustatakse, milliseid loogikaid testide kirjutamisel järgiti ja antakse ülevaade, millised testid erinevate klasside ja funktsioonide kohta tulid. Kõik kasutatud jõudlustestid on leitavad GitHub repositooriumist Lisad II alt.

3.1 Üldine loogika

Antud töös ei testita kogu programmi töötamist, mistõttu on kõik testid mikro jõudlustestid. Valitud funktsioonide ja klasside kohta on tehtud kaks eraldi testi, mis mõlemad täidavad sarnast ülesannet, kuid kasutades uusi ja vanemaid käsklusi töö läbiviimiseks.

3.1.1 Soojendus

Nagu ka sportlastel on vaja arvutitel ennast üles soojendada. Seetõttu tuleb jõudlustestide kirjutamisel kasutada soojendusringe, mis aitavad JVM-l kompileeruda. JVM ülesehitus on keeruline ja erinevate klasside üles laadimine võib võtta aega. Kui JVM on juba mingi arv ringe käinud, siis tulemused hakkavad stabiliseeruma ja võib usaldada saadud tulemusi[6]. Antud töös tahetakse teada, kas kahel sarnase tööeesmärgiga programmidel on kiiruse vahe ja sellepärast tehti ka soojendus, et saada programmi võimalikult täpne töötamiskiirus ilma arvestamata JVM eripärasustega. Joonis 1 peal on näha, et mida rohkem kordusi läbitakse seda paremaks ja stabiilsemaks muutub programmi läbivuse aeg. Antud kontekstis tuleb mõista, et kuskil on programmi läbimisajal piir ja liialt palju soojendus ringe ei ole mõttekas teha.



Joonis 1. Jõudlus peale n-dat kordust[6].

3.1.2 Kordused

Sarnaselt soojendusel on kordused tähtsad, et saada võimalikult täpne vastus. Ühe tulemuse mõõtmine ei anna antud töös piisavalt kindlat tulemust, mille pealt võrrelda, kas üks või teine oli kiirem. Kuna JVM-i seesmiseid protseduure on väga keeruline ette ennustada jääb ainukeseks lahenduseks jooksutada kõiki teste mingi n arv kordi.

Antud töö testide jooksutamise ajal märgati ka, et jooksutades mingeid teste järjest mitu korda, siis testide tulemused halvenesid. See võib tuleneda näiteks sellest, et arvuti protsessor töötab pideva koormuse all kui teste jooksutada ja kui järgmine test kohe otsa käivitada, siis ei laeta uuesti kõiki klasse sama kiirelt ära või tekib üldine jõu puudus arvutil. Selle probleemi vältimiseks käivitas autor teste käsitsi kümme korda järjest ja kandis kõik tulemused Excelisse.

3.1.3 Mõõtmine

Jõudlustestide puhul on palju erinevaid väärtusi mida mõõta. Käesoleva töö eesmärgist sõltuvalt valiti selleks väärtuseks programmi läbimise aeg. Mõõtmiseks kasutatakse ScalaMeter raamistikku, mis automaatselt mõõdab ära, kui kaua aega kulus programmi läbimiseks. Kuna testimiste käigus avastati kiiruse kadu mitme testi järjestikul jooksutamisel, siis võeti kasutusele kolm erinevat väärtust. Excelisse kantud andmetest leiti, mis oli kümne jooksutamise keskmine kiirus, kiireim kiirus ja aeglaseim kiirus.

3.2 Retroaktiivne klasside laiendamine

Varasemalt Scala 2-s ei olnud võimalik laiendusmeetodeid ilma kaudsete teisenduste või klasside kaudu kirjeldada. Nüüd Scala 3-s on laiendusmeetodid toodud keelde sisse, mis annab võimaluse parematakse veateadeteks ja tüübi järeldusteks[7].

Antud uuenduse kiiruse mõõtmiseks koostati kaks sarnast testi. Scala 2 funktsioonidega testis koostati üks klass, mis võtab parameetrid x, y ja raadius. Klassi sees on meetod *circumference*, mis arvutab ja tagastab saadud parameetritega ringi ümbermõõdu.

```
case class nonExtensionCircle(x: Double, y: Double, radius: Double) {
  def circumference: Double = radius * math.Pi * 2
}

def circleClass(): Double = {
  val circle = nonExtensionCircle(0.0, 0.0, 10.0)
  circle.circumference
}
```

Joonis 2. Ilma laienduseta klass *nonExtensionCircle* koos lisaks kirjeldatud meetodiga *circleClass*.

Sarnaselt on ka Scala 3 funktsioonidega jõudlustest, kuid klass, mis võtab parameetrid x, y ja raadius on laiendatud funktsiooniga *circumference*.

```
case class Circle(x: Double, y: Double, radius: Double)
extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2
```

Joonis 3. Laiendatud klass Circle meetodiga circumference.

3.3 Kontekstifunktsioonid

Kontekstifunktsioonid on funktsioonid, millel on ainult kontekstiga parameetrid[8]. Järngevates testides koostati tabel ridade ja lahtritega.

Scala 2 funktsioone kasutades tuli tabelile lisada kõik read ja lahtrid käsitsi, kuna puudus võimalus neid läbi konteksti lisada. Klassid *Table, Cell* ja *Row* on samad nii Scala 2 kui ka Scala 3 puhul, mida näeb Joonisel 5.

```
val t = Table()
val r1 = Row()
val r2 = Row()
val r3 = Row()
t.add(r1)
r1.add(new Cell("top left"))
r1.add(new Cell("top right"))
t.add(r2)
r2.add(new Cell("bottom left"))
r2.add(new Cell("bottom right"))
t.add(r3)
```

Joonis 4. Tabelisse ridade ja lahtrite lisamine Scala 2 puhul.

Scala 3 puhul on kolm lisa meetodit *table, row* ja *cell*, mis on kontekstifunktsioonid. Antud funktsioonid võtavad parameetriks antud hetkel käsil oleva tabeli ja seejärel lisavad sinna vastavalt koodile, kas uue rea või uue lahtri.

```
case class Cell(value: String)
case class Row(cells: Seq[Cell] = Seq.empty) {
  def add(cell: Cell): Row = copy(cells = cells :+ cell)
}
case class Table(rows: Seq[Row] = Seq.empty) {
  def add(row: Row): Table = copy(rows = rows :+ row)
}

def table(init: Table ?=> Unit) =
  given t: Table = Table()
  init
  t

def row(init: Row ?=> Unit)(using t: Table) =
  given r: Row = Row()
  init
  t.add(r)

def cell(str: String)(using r: Row) =
  r.add(new Cell(str))
```

Joonis 5. Scala 3 konteksti kaudu tabelisse ridade ja lahtrite lisamine.

3.4 Enumereerimine

Enumereerimine on ka varasemalt Scala 2-s olnud kasutusel, kuid selle süntaksit muudeti. Enumereerimise mõte on kirjeldada tüüpi, millel on üks või rohkem väärtusi[9].

Varasemalt pidi kirjeldatud objekti laiendama *Enumeration* klassiga, et kasutada selle funktsionaalsust. Seega esimeses testis on kirjeldatud objekt *Color1*, mida laiendab klass *Enumeration* ja mille väärtused on kolm värvi: punane, roheline ja sinine. Testis kutsutakse kõik kolm värvi välja ja mõõdetakse nende välja kutsumise kiirust.

```
object Color1 extends Enumeration {
  type Color = Value

  val Red: Color1.Value = Value("Red")
  val Green: Color1.Value = Value("Green")
  val Blue: Color1.Value = Value("Blue")
}
```

Joonis 6. Objekt Color1, mida laiendab klass *Enumeration* ja mis sisaldab erinevaid väärtusi.

Uue süntaksiga saab enumereerimist kasutada lihtsasti märksõnaga enum, mis annab võimaluse kirjeldada erinevate väärtusega objekti. Hiljem on võimalik kirjeldatud objekti väärtusi välja kutsuda.

```
enum Color:
  case Red, Green, Blue
```

Joonis 7. Uuendatud süntaks enumereerimise jaoks.

```
val red = Color.Red
val green = Color.Green
val blue = Color.Blue
```

Joonis 8. Uue süntaksiga muutujate väärtustamine.

3.5 Suletud tüübi aliased

Suletud tüübi eesmärk on pakkuda rohkem abstraktsiooni. See võimaldab luua uusi tüüpe, mis erinevad neile aluseks antud kujutisest, mis tähendab, et on võimalik kergesti luua uusi tüüpe, millel on ainulaadsed omadused ja käitumismustrid.

Käesoleva testi eesmärk oli võrrelda *opaque type alias* ehk suletud tüüpi aliast ja tavalist *type alias* ehk lihtsalt aliase kasutamist. Selles jõudlustestis kasutati erinevaid funktsioone nagu näiteks: *apply, safe, toDouble*, + ehk liitmine ja * ehk korrutamine. Ühe jõudlustesti ajal mõõdeti kõikide funktsioonide kiirused suletud kui ka tavalise aliase tüübi puhul. Antud testide ainsaks erinevuseks jäi üks ainus rida, kus kirjeldati *Logarithm* tüüpi. Scala 3 uue funktsiooni puhul oli selleks reaks *opaque type Logarithm = Double* ja vanema versiooni testimiseks *type Logarithm = Double*.

3.6 Lõikumistüübid

Tüübi kirjeldamise puhul kasutades sümbolit “&” luuakse lõikumistüüp, mis tähendab, et kirjeldatud parameeter peab olema mõlemat tüüpi samaaegselt. See tähendab, et kui on kaks tüüpi A ja B, siis kõik tüübi A meetodid ja tüübi B meetodid on üheselt kättesaadavad uue muutuja kaudu, mis on tüüpi A&B[10].

Sarnane kirjeldusviis oli juba varasemalt olemas Scala 2-s, kus sümboli “&” asemel tuli kasutada süntaksit *with*. Programmi töö mõttes on need kaks lahendust väga sarnased. Uue meetodi puhul luuakse lõikumistüüp ja vanema meetodi puhul ühildatakse kaks tüüpi, mis tähendab, et põhi erinevus peitub ainult selles, kuidas kompilaator neid käsitleb.

Testide puhul loodi kaks erinevat omadustüüpi *Resettable* ja *Growable*. Mõlemad tunnused omavad ühte funktsiooni vastavalt *reset*() ja *add*(). Vanema lahenduse testimisel kasutati funktsiooni f2, mis võtab parameetriks x, mille tüübiks on *Resettable with Growable*.

Uue lõikumistüübi testimiseks loodi funktsioon f, mis võtab parameetriks x ja x tüübiks on *Resettable & Growable*. Joonise 9 peal on näha mõlema kirjeldamisviis süntaksid.

```
def f(x: Resettable & Growable[String]) =
  x.reset()
  x.add("first")


def f2(x: Resettable with Growable[String]) =
  x.reset()
  x.add("first")
```

Joonis 9. Lõikumistüübi ja kahe tüübi ühildamise süntaksi erinevused.

3.7 Kattetüübid

Kattetüüpe saab kasutada sõltuvate tüüpidega meetodite määratlemiseks. Kattetüüp taandub ühele oma parempoolsetest külgedest, sõltuvalt selle uuritava tüübist[11].

Käesoleva uuenduse testimiseks loodi kaks meetodit *leafElem2* ja *leafElem*, kus vastavalt siis esimene ei kasutanud kattetüübi funktsiooni ja teine kasutas. Koodis kutsuti välja mitu korda erinevate väärtustega funktsiooni, et näha, kas ja kui palju kattetüübi kasutamine aega võidab.

```
def leafElem[X](x: X): LeafElem[X] = x match
  case x: String      => x.charAt(0)
  case x: Array[t]    => leafElem(x(0))
  case x: Iterable[t] => leafElem(x.head)
  case x: AnyVal      => x


def leafElem2[X](x: X): Any = x match
  case x: String      => x.charAt(0)
  case x: Array[t]    => leafElem(x(0))
  case x: Iterable[t] => leafElem(x.head)
  case x: AnyVal      => x
```

Joonis 10. Kattetüübi ja ilma kattetüübita funktsioonid.

Joonisel 10 on näha, et esimeses koodijupis on tagastus meetod *LeafElem* ja teises koodijupis on *Any*, mis ei anna nii täpset tagasiside, kui seda annab kattetüübiga meetod.

3.8 Omadustüübi parameetrid

Varasemalt ei olnud võimalik omadustüüpidele parameetreid kaasa anda. Selleks, et kasutada mingeid parameetreid, tuli need omadustüübi sees kirjeldada ja hiljem väärtustada. Scala 3 uuendustega on üritatud omadustüübid muuta rohkem klasside sarnaseks ja nüüd on võimalik ka anda kaasa parameetreid[12].

Antud uuenduse testimiseks loodi üks testjuhtum, kus oli omadustüüp *Greeting2*, mis sisaldas endas muutujat *name* ja funktsiooni msg, mis tagastab lihtsa lause. Hiljem väärtustatakse klassis C2, mida laiendab *Greeting2*, muutuja *name* ja kutsutakse välja funktsioon msg.

```
trait Greeting2:
  val name: String
  def msg = s"How are you, $name"

class C2 extends Greeting2:
  val name = "Bob"
  println(msg)
```

Joonis 11. Ilma parameetriteta omadustüübi muutujate väärtustamine klassis C2.

Teisel juhul loodi omadustüüp *Greeting*, mis võttis parameetriks ühe sõna tüüpi muutuja *name* ja sisaldas endas ühte funktsiooni msg. Sarnaselt esimesele testile kutsuti ka selle omadustüübi funktsioon msg välja klassis C.

```
trait Greeting(val name: String):
  def msg = s"How are you, $name"

class C extends Greeting("Bob"):
  println(msg)
```

Joonis 12. Parameetritega omadustüübi muutujate väärtustamine klassis C.

3.9 Sisseehitatud meetod

Sisseehitatud meetod ehk *inlined method* on tihti kasutatud kompileerimisaja metaprogrammeerimise meetod. Kui real on võtmesõna *inline*, siis kompileerimise ajal laiendatakse selle meetodi sisu. See tähendab, et kutse meetodile asendatakse hoopis selle meetodi sisuga. Staatiliselt asendatakse ka meetodi parameetrit esitatud parameetritega[13]. Selline tegevus peaks teoorias kiirendama programmi jooksutamise ajal meetodi kutsumist. *Inline* meetodite kasutamisel tuleb jälgida, et meetod ei oleks liialt keeruline ja pikk, sest muidu võib kompileerimisel tekkida probleeme ja programmi kiirus kannatab selle pealt, et kompileerimine võtab kauem aega kui tavaliselt. Lisaks meetoditele on ka võimalik muutujaid ja parameetreid *inlineda*.

Antud töös testiti *inline* meetodit lihtsa logimisfunktsiooni peal. Logimisfunktsioonis kutsuti välja faktoriaal arvutamise funktsioon, mis tagastas arvu kakskümmend faktoriaali. Arvu valik tehti selle järgi, et arvutus ei oleks liiga lihtne ja samas ka ei võtaks liiga palju

aega. Peale faktoriaali arvutamist kirjutas logimisfunktsioon üles arvutuse tulemuse. Mõlema testi puhul oli tegu sama funktsiooniga ja ainsaks vaheks oli *inline* võtmesõna funktsiooni alguses ja lisaks ka parameeter, mis oli *inlined*.

```
def nonInlinedLogged[T](level: Int, message: => String)(op: T): T =
  println(s"[$level]Computing $message")
  val res = op
  println(s"[$level]Result of $message: $res")
  res


inline def inlinedLogged[T](level: Int, message: => String)(inline op: T): T =
  println(s"[$level]Computing $message")
  val res = op
  println(s"[$level]Result of $message: $res")
  res
```

Joonis 13. Logimis funktsioonid ilma *inline* võtmesõnata kui ka *inline* võtmesõnaga.

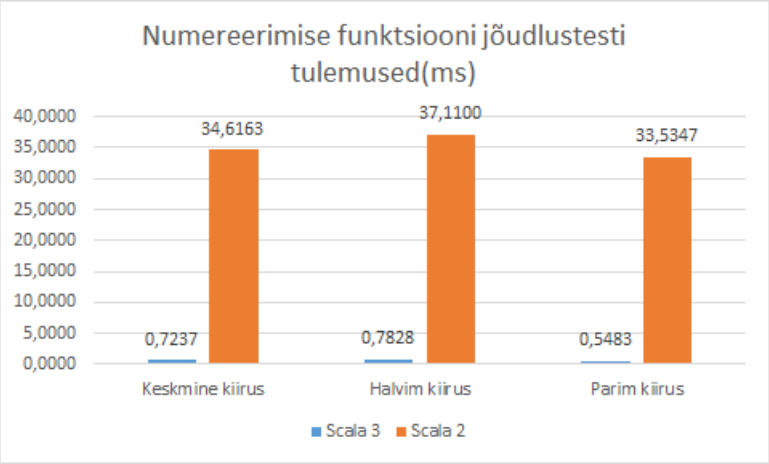
4. Tulemused ja arutelu

Selles peatükis antakse tagasiside testide tulemuste kohta. Võrreldakse testide graafikuid ja leitakse, milline uuendus kõige rohkem efektiivne oli ja milline kõige vähem.

4.1 Testide tulemused

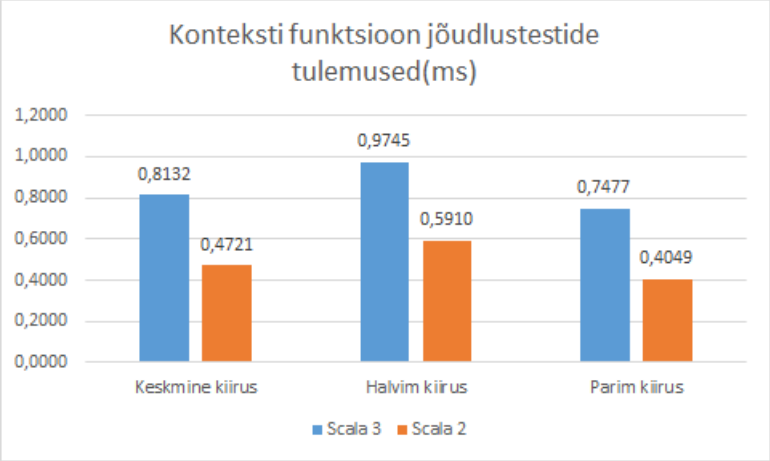
Antud töös mõõdeti kokku kaheksa erineva uuenduse tulemused. Kõiki tulemusi mõõdeti kümme korda käsitsi ja tulemused kanti Excelisse, mis on leitav Lisad II all olevast repositooriumist.

Enumereerimis funktsioon oli tunduvalt kiirem kasutades uut Scala 3 lähenemist. Joonis 14 pealt on näha, et enumeerimine Scala 3 süntaksiga on ligi viiskümmend korda kiirem kui Scala 2-s.



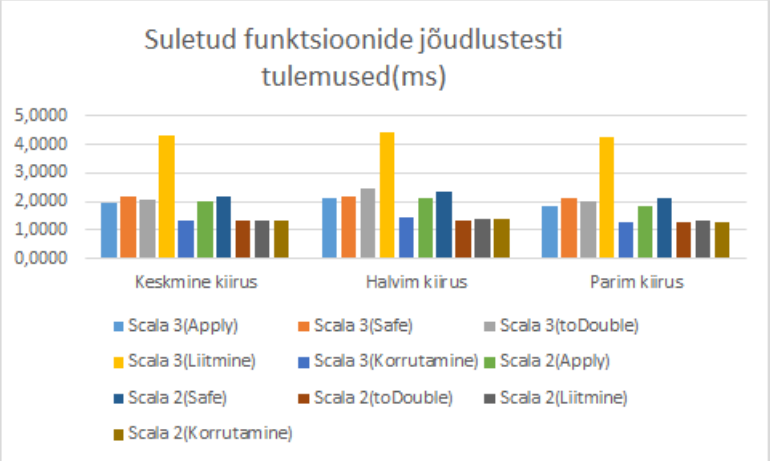
Joonis 14. Numereerimise funktsiooni jõudlustesti tulemused.

Teine vastand sellisele tulemusele oli konteksti süntaksiga. Joonisel 15 on näha, et kasutades vanema Scala 2 käsitsi lisamise funktsioone saadi tulemused kaks korda paremad kui seda uue Scala 3 konteksti süntaksiga.



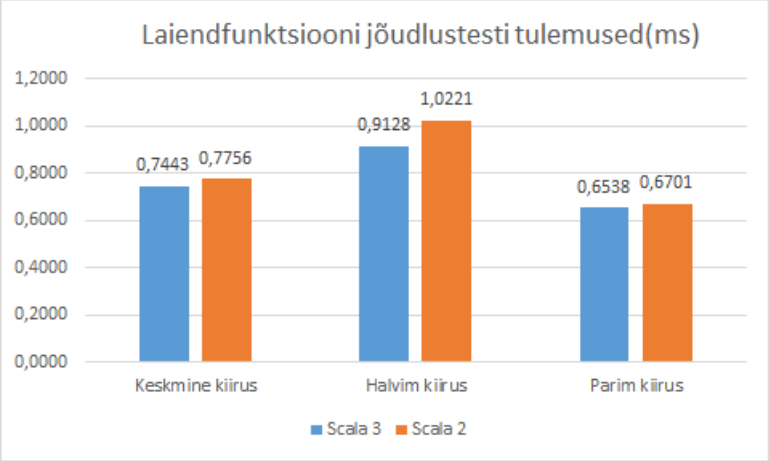
Joonis 15. Konteksti funktsiooni jõudlustestide tulemused.

Lisaks konteksti süntaksile oli ka suletud funktsioonide tulemused kohati halvemad. Joonisel 16 on näha, et teatud funktsioonid töötasid Scala 3 uuenduste puhul tunduvalt aeglasemalt kui Scala 2 funktsioonidega. Kuna selles testjuhtumis kasutati mitut erinevat funktsiooni ühes jõudlustestis, siis on graafikule kantud kõikide funktsioonide tulemused.

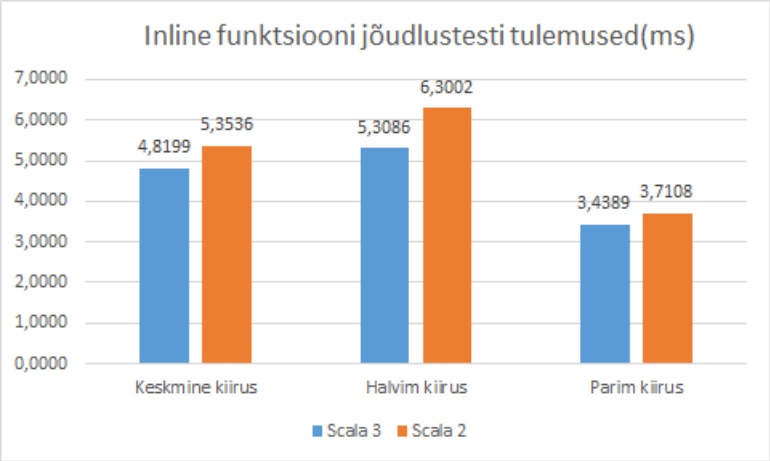


Joonis 16. Suletud funktsioonide jõudlustesti tulemused.

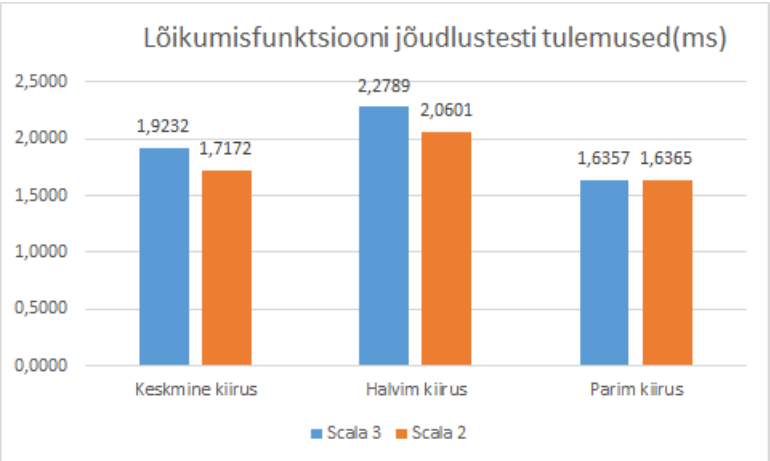
Ülejäänud uuendused on liiga võrdsete tulemustega, et öelda kumb variant efektiivsemalt töötab. Joonistel 17, 18, 19, 20 ja 21 on graafikud, mis näitavad ülejäänud uuenduste testide tulemusi. Selgelt on näha, et kõik tulemused on väga võrdsed ja selliste tulemustega ei ole võimalik öelda kumba süntaksit eelistada. Küll aga võib selle pealt öelda, et kui kasutada kõige uuemaid funktsioone ja süntakseid, siis ei kaota sellega kiiruse poolest.



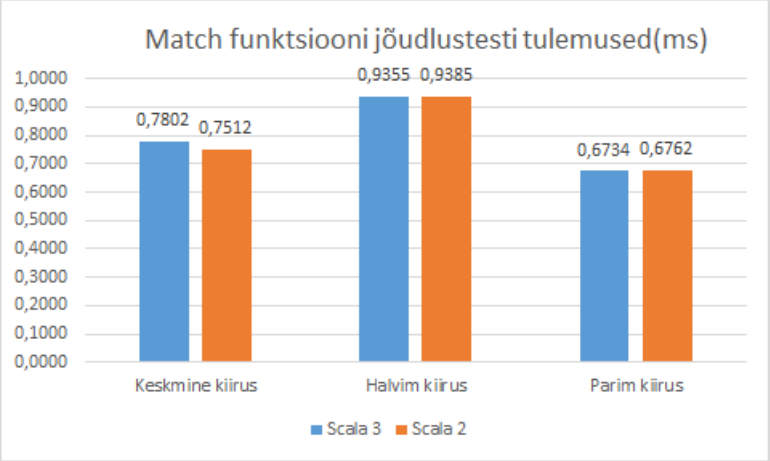
Joonis 17. Laiendfunktsiooni jõudlustesti tulemused.



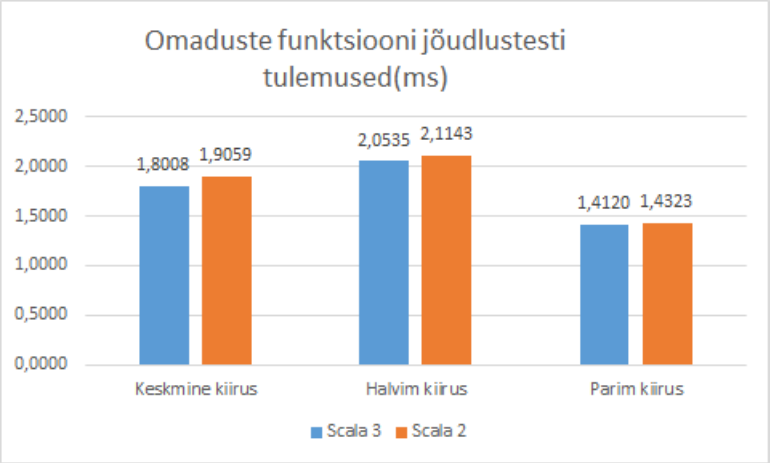
Joonis 18. Inline funktsiooni jõudlustesti tulemused.



Joonis 19. Lõikumisfunktsiooni jõudlustesti tulemused.



Joonis 20. Match funktsiooni jõudlustesti tulemused.



Joonis 21. Omaduste funktsiooni jõudlustesti tulemused.

4.2 Testide analüüs

Tehtud testide tulemused andsid ülevaate sellest, kas uued funktsioonid on aeglasemad või mitte. Tulemustest on näha, et peaaegu kõikidel juhtudel võib öelda, et kui keskmine kiirus oli kummalgi juhul parem, siis oli ka halvim ja parim kiirus parem. Käsitsi tehtud kümme kordust muutsid kõvasti testide tulemusi ja on üks võtmeväärtus, miks võib saadud tulemusi usaldada. Lisaks mõõdetud halvim ja parim kiirus näitavad, et koormuse all võib nende funktsioonide kiirus muutuda kuni 1 millisekundi võrra, mis ühekordsel läbimisel ei ole suur kadu, kuid mitmeid kordi sama funktsiooni läbides võib mõjutada üldist programmi kiirust.

Nagu eelnevalt ka antud töös mainitud tuleb mikro jõudlusteste tehes olla äärmiselt tähelepanelik, et kood oleks võimalikult optimeeritud ja puhas. Lisaks ei ole ainult funktsiooni läbimise aeg piisav argument, et öelda kogu tõde Scala 3 uuenduste kohta. Antud töö eesmärgiks oli saada teada, kas uuendused, mis tehti on ka vähemalt sama kiired

või kiiremad. Kui tahta teada mingi kindla uuenduse kohta peaks seda funktsiooni implementeerima mitut erinevat moodi ja jälgima ka teisi parameetreid. Kuna käesolevas töös ei olnud ühtegi kindlat uuendust, mida jälgida, siis viidi läbi kõikide uuenduste peal lihtsamad testid, et saada ülevaade funktsioonide kiiruste kohta.

5. Kokkuvõte

Töös anti lühiülevaade, mida kujutab endast jõudlustest. Lisaks jõudlustestile anti ülevaade testivast keelest ja raamistikust, mida testide kirjutamisel kasutati. Põhieesmärgiks oli leida kinnitus, et Scala 3 uuendused on vähemalt sama kiired või kiiremad kui vanemad funktsioonid ja süntaksid.

Töö jagunes praktiliseks osaks ja teoreetiliseks osaks. Praktilises osas kirjutati kokku kaheksa erinevat jõudlustesti. Iga uuenduse kohta kirjutati kaks testi, üks vanema funktsiooni kohta ja teine uuema funktsiooni. Teooria osas anti ülevaade, kuidas testid on kirjutatud, mis vahendeid kasutati ja analüüsiti testide tulemusi.

Jätkutööna saaks jõudlusteste veel täiendada ja uurida, mis põhjustel teatud funktsioonid ei olnud paremad kui vanemad funktsioonid. Lisaks saaks uurida võrdsete tulemustega testjuhtumite tulemusi. Millised tingimused mõjutavad nende testide tulemusi ja kas parema implementeerimise puhul saaks uuemad funktsioonid kiiremaks.

Viidatud kirjandus

[1] GeeksForGeeks. https://www.geeksforgeeks.org/how-to-choose-a-programming-language-for-a-project/ (04.08.2023)

[2] Forbes. https://www.forbes.com/sites/scottlenet/2018/12/12/the-importance-of-benchmarking/?sh=11c88f7b4245 (04.08.2023)

[3] Joseph Yossi Gil, Keren Lenz, Yuval Shimron. A microbenchmark case study and lessons learned. ACM Digital Library, 2011, 23. oktoober. https://dl.acm.org/doi/pdf/10.1145/2095050.2095100 (07.08.2023)

[4] Scala dokumentatsioon. https://scala-lang.org/files/archive/spec/2.13/ (06.08.2023)

[5] ScalaMeter GitHub. https://scalameter.github.io/ (04.08.2023)

[6] Hyperskill. https://hyperskill.org/learn/step/21044#benchmark-types (06.08.2023)

[7] Scala uuendused. https://docs.scala-lang.org/scala3/new-in-scala3.html (07.08.2023)

[8] Scala dokumentatsioon konteksti funktsioonidele. https://docs.scala-lang.org/scala3/reference/contextual/context-functions.html (07.08.2023)

[9] Scala dokumentatsioon enumereerimisele. https://docs.scala-lang.org/scala3/reference/enums/enums.html (07.08.2023)

[10] Scala dokumentatsioon lõikumistüüpidele. https://docs.scala-lang.org/scala3/reference/new-types/intersection-types.html (07.08.2023)

[11] Scala dokumentatsioon kattetüüpidele. https://docs.scala-lang.org/scala3/reference/new-types/match-types.html (07.08.2023)

[12] Scala dokumentatsioon omaduste parameetritele. https://docs.scala-lang.org/scala3/reference/other-new-features/trait-parameters.html (07.08.2023)

[13] Scala dokumentatsioon *inline* meetoditele. https://docs.scala-lang.org/scala3/guides/macros/inline.html (07.08.2023)

Lisad

I. Analüsaator Põder repositoorium

Lühidemo Põder programmi kasutamisest, mis on saadud Kalmer Apinis GitHub repositooriumist:

https://bitbucket.org/kalmera/poder/src/master/

https://bytebucket.org/kalmera/poder/raw/f86fbff50c91d1950e55efc6df8be66dd6f66f56/demo.gif

II. Scala 3 jõudlustestide repositoorium

Kõikide töös kasutatud testide repositoorium:

https://github.com/BrandonRauba/scalaTestid/tree/main/src/main/scala

Kõik saadud tulemused Exceli tabelisse kantuna:

https://github.com/BrandonRauba/scalaTestid/blob/main/testide_tulemused.xlsx

III. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Brandon Rauba**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   Scala 3 jõudlustestimine,

   mille juhendaja on *Kalmer Apinis*,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Brandon Rauba

11.08.2023