TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Andre Sinisalu

Java programmide staatiline intervallanalüüs raamistikus Põder

Bakalaureusetöö (9 EAP)

Juhendaja(d): Kalmer Apinis

Tartu 2019

Java programmide staatiline intervallanalüüs raamistikus Põder

Lühikokkuvõte:

Staatiline programmianalüüs on meetod programmide uurimiseks eesmärgiga tuvastada programmis vigu või kontrollida koodi kvaliteeti ilma programmi käivitamata. Käesoleva bakalaureusetööga luuakse staatilise analüüsi raamistikku Põder intervallanalüüsi moodul, millega on võimalik kontrollida ja analüüsida täisarvuliste muutujate käsitlust Java programmides. Selleks antakse lugejale ülevaade staatilise analüüsi teoreetilisest lahendusest ja matemaatilisest võreteooriast ning kirjeldatakse JVM (*Java Virtual Machine*) baitkoodi. Töö praktilises osas luuakse eelnevalt kirjeldatud teooriale toetudes intervallanalüüsi moodul. Moodulit testitakse testprogrammide komplektiga näidates, et moodul on suuteline JVM-baitkoodiks kompileeritud programmides leidma täisarvuliste muutujate käsitlemisel tekkinud vigu.

Võtmesõnad:

Staatiline analüüs, intervallanalüüs

CERCS: P170 (Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine)

Static Interval Analysis of Java Programs Using the Framework Põder

Abstract:

Static program analysis is a method for studying program's behaviour with the intent of detecting bugs or verifying code quality. The main goal of this thesis is to create an interval analysis module in the framework Põder, which could be used for detecting bugs and inefficiencies in the usage of integer variables in Java programs. The reader is first given an overview of static analysis, the mathematical lattice theory and Java Virtual Machine (JVM) bytecode. After that, the interval analysis module will be created based on the previously described theory. The module will be tested with a test program suite proving that the module can successfully be used for detecting bugs related to the usage of integer variables.

Keywords:

Static analysis, interval analysis

CERCS: P170 (Computer science, numerical analysis, systems, control)

Sisukord

1 Sissejuhatus

Staatilist analüüsi on kompilaatorites programmide optimeerimiseks kasutatud 1970. aastatest. Ajapikku on avastatud, et staatiline analüüs on kasulik ka programmist vigade leidmiseks, programmi tegevuse verifitseerimiseks kui ka koodiredaktorites erinevate funktsionaalsuste toetamiseks. Näiteks kasutatakse staatilist analüüsi koodis meetodite vahel liikumiseks, koodi refaktoreerimiseks ja koodijuppide automaatseks laiendamiseks. Seega püüab staatiline analüüs vastata võimalikult täpselt võimalikult paljudele küsimustele programmi kohta ilma programmi käivitamata.

Rice'i teoreem ütleb, et ei leidu Turningi masinat, mis suudaks alati vastata õigesti küsimustele programmi mitte-triviaalsete semantiliste omaduste kohta [1]. Sellest järeldub, et ei ole võimalik luua ideaalset staatilist analüsaatorit, mis suudaks suvalise programmi korral vastata kõikidele küsimustele korrektselt.

Olgu meil näiteks ülesandeks määratleda suvalises programmis, kas teatud muutuja omab programmi käitusaja vältel alati sama väärtust või on tema väärtus muutlik. Triviaalselt võiksime vastata sellele küsimusele alati negatiivselt, kuid sel juhul ei oleks analüsaatorist palju kasu. Lihtsamate näidete puhul on vastuse andmine kerge, vt joonis 1, seega võiksime oletada, et suudame luua algoritmi, mis suudab alati vastata sellele küsimusele korrektselt.

```
int x = 1;
if (true)
    x = 2;
```

Joonis 1. Näidisprogramm, *x*-i väärtus muutub ajas.

Sarnaselt on meil aga võimalik konstrueerida programm, kus TM(n) simuleerib n-indat Turingi masinat, vt joonis 2.

```
int x = 1;
if (TM(n))
    x = 2;
```

Joonis 2. Näidisprogramm, *x*-i väärtus ei pruugi muutuda ajas.

Näeme, et x on konstantne ainult juhul, kui n-is Turingi masin oma tööd ei lõpeta. Kui meie eelnevalt loodud algoritm oleks olemas, siis me suudaksime anda alati korrektse vastuse Turingi masinate termineerumise probleemile (*Halting problem*). Turing ise on tõestanud, et see ei ole võimalik – oleme jõudnud vastuoluni. Seega võime järeldada, et staatilise analüsaatori loomine on protsess, kus tuleb teha kompromisse täpsuse ja korrektsuse vahel.

Käesoleva uurimistöö eesmärgiks on luua staatilise analüüsi tarkvara, mis suudab Java programmides leida täisarvude käsitlemisel tekkinud vigu. Selleks kasutatakse raamistikku Põder, mis võimaldab teostada nii lihtsamaid kui ka keerukamaid Java baitkoodil põhinevaid analüüse. Täisarvude käsitlemisel tekkinud vigade efektiivseks leidmiseks kasutab autor intervallanalüüsi.

2 Staatilise analüüsi ülevaade

Staatiline analüüs on meetod programmide uurimiseks nende lähtekoodi järgi eesmärgiga tuvastada programmist vigu ja ebaefektiivsusi ning need kasutaja jaoks välja tuua. Samuti saab staatilist analüüsi kasutada koodi kvaliteedi kontrolliks ning erinevateks spetsiifilisteks muutujate ja käsuvoo analüüsideks.

Staatilist analüüsi teostatakse enne tarkvara käivitamist tema lähtekoodi uurides. Lähtekoodiks võib olla nii masinkood kui ka mõni kõrgema taseme programmeerimiskeel. See tähendab, et staatilist analüüsi saab läbi viia ka masinatel, kus antud programmi käivitada ei ole võimalik või on turvakaalutluste põhimõttel keelatud.

Staatilise analüüsi protsess on automatiseeritud. Käsitsi vigade otsimine ja leidmine on ajakulukas, ebaefektiivne ning tekkida võivad inimlikud vead. Automaatne protsess on aga kiirem ja põhjalikum, olles suuteline uurima tervet koodibaasi ja käsuvood. See vabastab inimressurssi ning kiirendab arendusprotsessi.

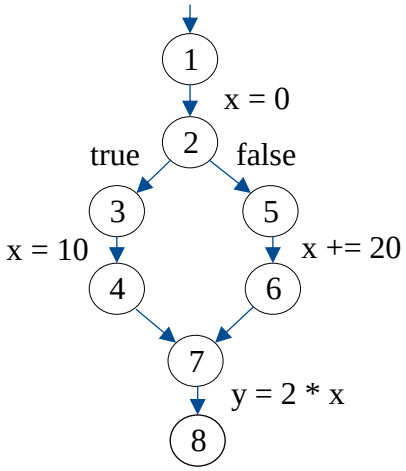
2.1 Intervallanalüüsi näide

Järgnevalt teostame käsitsi intervallanalüüsi joonisel 3 asuvale programmile, kus on esitatud programmi lähtekood ning kõrval tema juhtvoograaf. Juhtvoograafis on tippudeks programmipunktid ning servadeks programmis täidetavad käsud. Tipud on nummerdatud, et analüüsi oleks lihtsam jälgida. Hiljem näeme, et tegelik analüüs tegeleb samuti programmi juhtvoograafiga, kus programmi käsulausete asemel vaadeldakse masinkoodi instruktsioone.

- Punktis 1 algab programmi täitmine. Sellel hetkel ei ole veel defineeritud ühtegi lokaalset muutujat, seega ei ole siin midagi analüüsida.
- Punktis 2 defineeritakse muutuja *x* ning omistatakse talle väärtus 0. Seega loome muutuja *x*-i jaoks intervallvahemiku, kus seame alumiseks ja ülemiseks piirväärtuseks 0-i. $x \in [0, 0]$. Samuti peaksime analüüsima järgmise tingimuslause tingimust. Kuna *Math.random()* $\in [0, 1]$, siis peame uurima mõlemat tingimuslause haru.
- Punktis 3 oleme jõudnud tingimuslause tõesesse harusse. Praegusel hetkel kehtib jätkuvalt $x \in [0, 0]$.
- Punktis 4 määratakse *x*-ile uus väärtus. $x \in [10, 10]$.
- Punktis 5 oleme jõudnud tingimuslause väära harusse. $x \in [0, 0]$.
- Punktis 6 lisatakse *x*-i väärtusele 20. Selleks tuleb liita antud väärtus mõlemale piirmäärale. $x \in [20, 20]$.
- Punktis 7 liituvad tingimuslause mõlemad harud, seega tuleb arvestada mõlema haru analüüsil leitud tulemusi. Et me kumbagi haru ei saanud varem vabastada, siis tuleb kombineerida kaks *x*-i intervalli, $x \in [10, 10] \vee x \in [20, 20]$. On võimalik jätkata analüüsi arvestades mõlemat intervalli eraldi, kuid lihtsuse mõttes saame neid kombineerides tulemuseks $x \in [10, 20]$.

- Punktis 8 luuakse uus muutuja *y*, mis saadakse *x*-i väärtuse kahega korrutamisel. Seega saame analüüsi tulemuseks $x \in [10, 20] \wedge y \in [20, 40]$.

```
int x = 0;
if (Math.random() <= 0.5)
    x = 10;
else
    x += 20;
int y = 2 * x;
```



Joonis 3. Testprogrammi lähtekood ning juhtvoograaf.

2.2 Võred

Osaliselt järjestatud hulk on selline hulk *S*, mille on defineeritud binaarne seos $\sqsubseteq$ nõnda, et järgnevad tingimused oleks täidetud:

- refleksiivsus: $\forall x \in S : x \sqsubseteq x$,
- transitiivsus: $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$,
- antisümmeetrilisus: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$. [2]

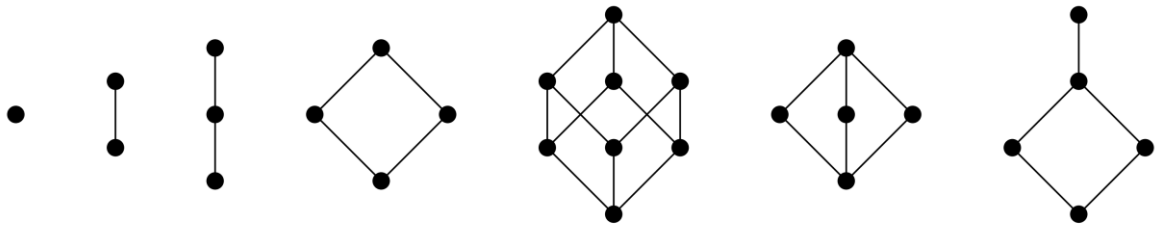
Antud binaarne operaator tähistab järjestusseost. Programmis on elemendid järjestatud nende väite tugevuse ehk implikatsiooni järgi. Kirjutis $x \sqsubseteq y$ näitab, et kui programmi oleku kohta kehtib *x*, siis kehtib selle kohta ka *y*. Näiteks, kui mingis programmipunktis kehtib väide $0 \leq a \leq 3$, siis võib analüsaator ilma korrektsust kaotamata väita nii $a \to [0, 3]$ kui ka $a \to [-\infty, 100]$. Mõistagi eelistame, et analüsaator annab täpsema vastuse.

Olgu *X* hulga *S* osaliselt järjestatud alamhulk, $X \subseteq S$.

Sel juhul nimetatakse hulga *X* **ülemiseks tõkkeks** sellist elementi $c = \bigsqcup X$, mille korral $\forall x \in X : x \sqsubseteq c$. Vähimat sellist elementi nimetatakse **ülemiseks rajaks**, s.t *X*-i iga ülemise tõkke *y* korral $y \sqsubseteq c$.

Sarnaselt nimetatakse hulga *X* **alumiseks tõkkeks** sellist elementi $c = \bigsqcap X$, mille korral $\forall x \in X : c \sqsubseteq x$. Vähimat sellist elementi nimetatakse **alumiseks rajaks**, s.t *X*-i iga ülemise tõkke *y* korral $y \sqsubseteq x$.

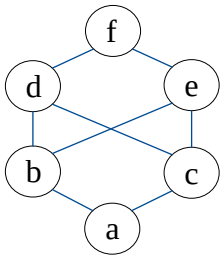
Täielik võre on osaliselt järjestatud hulk, mille igal alamhulgal leidub ülemine ja alumine raja. Võresid saab kujutada Hasse diagrammiga, kus graafi tippudeks on võre elemendid ning servadeks nendevahelised suhted. Näiteks on joonisel 4 näha osaliselt järjestatud hulkasid, mis on võred. [2]



Joonis 4. Osaliselt järjestatud hulgad, mis on võred.[1]

Võre kahe elemendi vahel ülemise ja alumise raja leidmist tähistatakse vastavalt kirjapildis $a \sqcup b$ (i.k *join*) ja $a \sqcap b$ (i.k *meet*). Staatilise analüüsi juures kasutatakse ülemise raja leidmist programmi juhtimisvoo hargnemisel tekkinud mitme erineva oleku taas ühendamiseks.

Joonisel 5 on aga näha osaliselt järjestatud hulk, mis ei ole võre, sest tippudel *b* ja *c* on ülemisteks tõketeks tipud *d*, *e* ja *f*, kuid ükski nendest ei ole võre ülemine raja.



Joonis 5. Osaliselt järjestatud hulk, mis ei ole võre.

Igal täielikul võrel on unikaalne suurim element $\bigsqcup S = \top$ ning unikaalne vähim element $\bigsqcap \emptyset = \bot$, mida vastavalt kutsutakse *top*'iks ja *bottom*'iks. *Top* märgib staatilise analüüsi juures kõige ebatäpsemat olekut ehk siis olukorda, kus programmi seisundi kohta puudub igasugune info. Selline olek võib tekkida olukorras, kus analüsaator on otsustanud, et programm võib olla igasuguses olekus, või siis ka olukorras, kus analüsaator on suutnud näidata, et programm on mingites kindlates olekutes, kuid ei ole teisi olekuid suutnud välistada. *Bottom* kirjeldab kokkuleppeliselt seega sellist vastupidist seisundit, kus kõik ülejäänud seisundid on välistatud. Praktikas tähistabki see seisu, kuhu programm täitmisel kunagi ei jõua. [3]

2.3 Staatiline analüüs

Programmi staatilise analüüsi teostamiseks tuleb läbi vaadata kõik programmipunktid ning määrata iga punkti kohta võimalikult täpne seisund tema muutujate ja oleku kohta, milles võib programm selles punktis olla. Pärast analüüsi saab selle infoga uurida täpsemalt programmi täitmise käiku ning leida programmis anomaaliaid.

Selleks seatakse esmalt igale programmipunktile vastavusse üks võre element, milleks analüüsi alguses on element *bottom*. Seda tehakse eeldusel, et kui programmis leidub punkt,

[1] https://cs.au.dk/~amoeller/spa/spa.pdf

kuhu programm oma töö käigus jõuda ei saa, siis selle punkti olek peakski vastavalt eelnevale definitsioonile olema *bottom*.

Programmianalüüsi ei ole mõtet alustada punktist, kuhu programm ei pruugi oma töö käigus kunagi jõuda, seega valitakse analüüsi alguseks selline punkt, mis programmi käivitamisel kindlasti kasutust leiab. Enamasti on selleks punktiks programmi peameetodi alguspunkt. See seatakse vastavusse mõne võre mitte-*bottom* elemendiga. Näiteks võib selleks olla element *top*, mis näitab, et sellesse punkti programm jõuab, kuid analüsaatoril puudub täpsem info punkti seisundi kohta. Samuti on võimalik sellise meetodi puhul analüsaatorisse kaasa anda täiendavat infot, mida analüsaator teisiti teada ei saa. Näiteks programmi käivitamisel kasutatavaid parameetreid või infot keskkonnamuutujate kohta.

Järgnevalt läbitakse programm vastavalt tema juhtvoograafile ning uuendatakse programmipunktidele vastavusse seatud elemente. Väärtuseid uuendatakse seni, kuni propageeritav väärtus juhtvoograafi serva siht-tipule on suurem kui selle vana väärtus. Kui serva siseneb rohkem kui üks serv, võetakse uuendamise väärtuseks kasutusele kõikide nende servade elementide ülemine raja. Lisaks valitakse igas tsüklis välja üks juhtvoograafi tipp, mille väärtuse uuendamisel väärtust vastavalt kas kitsendatakse või laiendatakse.

Kui väärtuste uuendamine on lõppenud, siis on garanteeritud, et igale programmipunktile vastav väide selles punktis kehtib ning programmi töö ja seisundite analüüs on lõppenud. Sellele järgnevalt saab teostada lisaanalüüsi, et tuvastada programmis punktid, kuhu programm oma töö käigus jõuda ei saa, või kontrollida muutujate väärtusi ja funktsioonidele antavaid argumente.

2.4 Intervallanalüüs

Intervallanalüüs on selline staatiline analüüs, mis leiab igale täisarvulisele muutujale tema vähima ja suurima väärtuse igas programmipunktis. See tähendab, et muutujale määratakse arvteljelt lõik, mida vastavalt kas siis suurendatakse või vähendatakse. Selline analüüs võimaldab tuvastada programmis massiivide indekseerimise ning täisarvuliste muutujate üle- ja alatäitumisega seotud probleeme. Samuti on võimalik teostada juhtimisvoo analüüsi.

Intervallanalüüsi teostamiseks kasutatakse **intervalldomeeni**, vt joonis 6, järgmine:

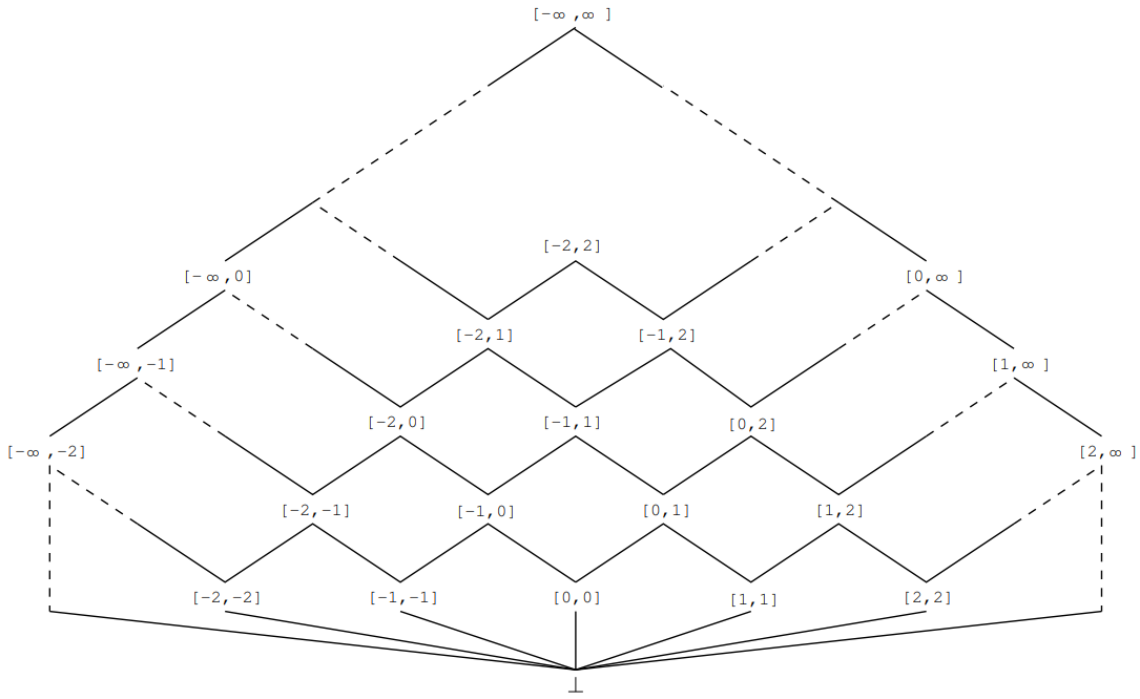
$$\text{Interval} = \{[l,u] | l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}.$$

Intervalldomeenil käitub binaarne seos $\sqsubseteq$ loogiliselt:

$$[l_1,u_1] \sqsubseteq [l_2,u_2] \Leftrightarrow l_2 \leq l_1 \wedge u_1 \leq u_2. \text{ [3]}$$

Samuti on intervalldomeenil defineeritud ülemise raja leidmise operaator $\sqcup$.

$$\begin{gathered} \bot \sqcup\ i = \bot \\ i\ \sqcup \bot = \bot \\ [l_1,u_1] \sqcup [l_2,u_2] = \ [\min(l_1,l_2), max(u_1,u_2)] \end{gathered}$$



Joonis 6. Intervalldomeeni Hasse diagramm.[2]

2.5 Laiendamine ja kitsendamine

Intervalldomeen on lõpmatu kõrgusega, kuna sisaldab endas järgnevat ahelat:

$$[0,0] \sqsubset [0,1] \sqsubset [0,2] \ldots$$

See tähendab, et tsüklitega programme uurides ei pruugi me naiivse lähenemisega intervallanalüüsi teostades piisavalt kiirelt ja efektiivselt lahenduseni jõuda. Näiteks järgneva programmi, vt joonis 7, analüüsiks kuluks meil täpselt nii palju samme, kui mitu korda selle tsükli keha läbitakse. Sisuliselt simuleeritakse läbi terve programm, mis aga ei ole staatilise analüüsi ideedega kooskõlas.

```
int x = 0;
while (x < 1000)
      x += 1;
```

Joonis 7. Tsükliga programmi näidis.

Sellise probleemi lahendamiseks toome teadlikult analüüsi sisse teataval määral ebatäpsust, et kiirendada sellistel puhkudel lahenduse leidmist ning muuta tsüklite analüüs garanteeritult lõplikuks. Selleks defineerime **laiendamise operaatori ∇**, mida klassikaliselt intervallide puhul defineeritakse järgnevalt:

$$[l_1, u_1] \nabla [l_2, u_2] = [l, u], \text{ kus}$$

$$l = \begin{cases} l_1, & \text{kui } l_1 \leq l_2 \\ -\infty, & \text{muidu} \end{cases} \quad \text{ja} \quad u = \begin{cases} u_1, & \text{kui } u_2 \leq u_1 \\ +\infty, & \text{muidu} \end{cases}$$

[2] https://cs.au.dk/~amoeller/spa/spa.pdf

Laiendamist kasutame iga kord, kui jõuame analüüsiga tsükli päise juurde. Seal kasutame laiendamist praeguse analüüsi tulemuse ja eelmisel tsükli läbimisel saadud tulemuse vahel. Kui antud muutuja on tsükli vahepealse läbimisega kas suurenenud või vähenenud, siis laiendamise tulemusel saame teda kasvatada kohe vastavas suunas lõpmatusse. Sellega garanteerime, et joonisel 7 toodud programmi tsükli analüüsiks kulub meil vaid 2 iteratsiooni. [3]

Joonisel 7 välja toodud programmi analüüsil laiendamisega saame tsükli kehas tulemuseks $x \to [0, +\infty]$, mis aga ei ole just kõige parem tulemus. Täpsuse taastamiseks on mõnel juhul võimalik kasutada kitsendamist, mille tulemusel jõuaksime tagasi täpsema lahenduseni $x \to [0, 999]$. Selleks defineerime **kitsendamise operaatori Δ**, mille klassikaline definitsioon on järgmine:

$$[l_1, u_1] \,\Delta\, [l_2, u_2] = [l, u], \text{ kus}$$

$$l = \begin{cases} l_2, & \text{kui } l_1 = -\infty \\ l_1, & \text{muidu} \end{cases} \quad \text{ja} \quad u = \begin{cases} u_2, & \text{kui } u_1 = +\infty \\ u_1, & \text{muidu} \end{cases}$$

Kitsendamist saame kasutada joonisel 7 välja toodud programmi analüüsis tsükli kehasse sisenemisel, sest tingimuse $x < 1000$ puhul teame, et tsükli kehas peab muutuja *x* ülemine piirväärtus olema rangelt väiksem 1000-st. Seega saame rakendada kitsendamist, et jõuda tulemuseni $x \to [0, +\infty] \,\Delta\, [0, 999] = [0, 999]$.

2.6 Raamistik Põder

Põder on JVM (*Java Virtual Machine*) programmide staatiline analüsaator, mille eesmärk on tuvastada programmis sisalduvad vead seejuures programmi käivitamata. Selleks kasutatakse andmevooanalüüsi, kus analüüsitavat programmi käsitletakse kui graafi, mille tipud on programmi käsud ning kaared nende vahel kirjeldavad viise, kuidas programm saab liikuda ühest avaldisest teise [4]. Graafi loomiseks kasutatakse programmi JVM baitkoodi, seega on raamistikus Põder võimalik universaalselt analüüsida kõiki JVM-i jaoks kompileeritavaid programme.

Põder on kirjutatud Scala-s ning on avatud lähtekoodiga [5]. Uute analüüside implementeerimiseks tuleb luua analüüsiks sobilik võre ehk domeen ning seejärel kirjutada klass, mis programmi domeeni vastavalt JVM baitkoodi instruktsioonidele muudab.

2.7 JVM baitkood

Java Virtual Machine (JVM) on abstraktne masin, mis suudab interpreteerida JVM baitkoodi arvuti protsessori jaoks arusaadavateks instruktsioonideks. JVM on implementeeritud tarkvaraliselt kindla spetsifikatsiooni järgi, mis tagab selle, et JVM baitkood täidetakse olenemata olemasolevast riistvarast alati samamoodi ning korrektselt. Seega tekitab JVM eraldi kihi riistvara ja tarkvara vahele lubades programmeerijatel ühe korra koodi kirjutades seda programmi mitmel erineval platvormil käivitada. [6]

JVM baitkoodiks nimetatakse JVM-i masinkoodi. Need on instruktsioone, mida JVM vastava riistvara protsessorile edasi teisendab. JVM-i jaoks on baitkood ühe-baidi pikkuste

instruktsioonide ja nende argumentide jada, mille lihtsamaks jälgimiseks on igale instruktsioonile antud teatud mnemooniline vaste, vt tabel 1. [7]

Tabel 1. Baitkoodi mnemoonika näide.

Baitkoodi instruktsioon	Mnemoonika
03	ICONST_0
3B	ISTORE_0
84 00 01	IINC 0, 1
1A	ILOAD_0
05	ICONST_2
68	IMUL
3B	ISTORE_0
A7 FF F9	GOTO -7

Lihtsustatult koosneb JVM käsuloendurist (*programm counter*), mis sisaldab järgmise täidetava instruktsiooni aadressi, kuhjast (*heap*), kus hoitakse programmi täitmisel tekkinud objekte, meetodialast (*method area*), kust laetakse mällu uusi instruktsioone, ja magasinist (*stack*), mis on ajutiste muutujate ja väärtuste jaoks [8].

Staatilise analüüsi jaoks on kõige huvitavam ala magasin. Iga meetodikutse järel luuakse magasini uus täitmiskeskkonna kaader (*frame*), kus hoitakse lokaalseid muutujaid ja operandide pinu. Konkreetsele lokaalsele muutujale viitamiseks kasutatakse muutujate indekseerimist, seega on mõistlik neid kujutada massiivina. [9]

2.8 JVM baitkoodi spetsifikatsioon

Järgnevalt kirjeldab autor täisarvuliseks intervallanalüüsiks vajalikke põhilisi JVM baitkoodi instruktsioone ja nende täpsemaid implementatsioone. Intervallanalüüs ei uuri viitasid, mäluaadresse ega muid andmetüüpe, seega ei ole antud nimestik lõplik, kuid siiski on see intervallanalüüsi implementeerimiseks piisav. Nimekirja loomiseks on kasutatud teost „Programming for the Java Virtual Machine“ ning antud hetkel viimast Java virtuaalmasina spetsifikatsiooni [10, 11].

2.8.1 Lokaalsed muutujad

JVM-i lokaalsed muutujad on sarnased Java programmide lokaalsetele muutujatele. Iga meetod võib kasutada kuni 65536 muutujat, mida kõiki salvestatakse lokaalsete muutujate massiivis. See tähendab, et muutujate poole ei pöörduta nende programmis määratud nime järgi, vaid massiivi järjekorranumbri järgi. Muutujaid järjestatakse nende kasutamise

järjekorra alusel, võttes arvesse seda, et *long* ja *double* tüüpi muutujad hõivavad massiivis kaks järjestikust asukohta. [11]

Käsk „**istore n“** asetab n-indasse massiivi pessa ajutiste väärtuste magasini kõige ülemise elemendi ning eemaldab selle magasinist.

Käsk „**iload n**“ laeb n-indast massiivi pesast muutuja mällu ning asetab selle kõige ülemiseks ajutiste väärtuste magasini elemendiks.

Mõlemad käsud eeldavad täisarvulise väärtuse olemasolu massiivis või magasinis. Samuti on saadaval „iload_n“ ja „istore_n“ käsud, kus n-i väärtus jääb 0-i ja 3 vahele, mida saab JVM-is implementeerida efektiivsemalt ning mis võtavad kompileeritud koodis vähem ruumi. Näiteks on samaväärsed käsud „iload_3“ ning „iload 3“.

2.8.2 Aritmeetika

Täisarvulise aritmeetika jaoks on reserveeritud neli baaskäsku: liitmine, lahutamine, korrutamine ja jagamine. Lisaks on defineeritud käsk jagatise jäägi leidmiseks ning samuti käsk arvu vastandarvu leidmiseks.

Käsud „**iadd**“, „**isub**“, „**imul**“, „**idiv**“, „**irem**“ eemaldavad magasinist kaks ülemist väärtust, leiavad vastavalt kas nende väärtuste summa, vahe, korrutise, jagatise või jäägi ning asetavad resultaadi tagasi magasini kõige ülemiseks elemendiks. Vahe, jagatise ja jäägi leidmisel leitakse resultaat teisest magasini elemendist. Kõik antud käsud tagastavad resultaadina magasini täisarvu.

Lisaks on eraldi defineeritud vastandarvu leidmise käsk „**ineg**“, mis on samaväärne magasini esimese elemendi korrutamisega „-1“-ga. See käsk on efektiivsem tavalisest korrutamisest, kuna suurem osa tänapäevastest protsessoritest omavad spetsiaalset arvu vastandarvu leidmiseks loodud efektiivset instruktsiooni.

Samuti on defineeritud käsk „**iinc n m**“, mis suurendab magasini *n*-indat elementi konstandi *m* võrra. Konstant võib olla nii negatiivne kui ka positiivne, mistõttu kohtab seda instruktsiooni tihti *for*-tsüklite juures.

2.8.3 Konstantväärtused

Konstantväärtused käituvad baitkoodis samaväärselt Java koodile. Konstantide lisamiseks magasini kasutatakse üldistatud käsku „**ldc n**“, mis lisab väärtuse *n* magasini esimeseks elemendiks. Antud käsku saab kasutada igat tüüpi väärtustega, näiteks *int*, *float*, *long*, *double* ja *string*, seega tuleb tüübi leidmiseks käsu argumenti spetsiaalselt uurida.

Kuna enamasti kasutavad programmid väiksemaid konstantväärtuseid, siis on nende magasini lisamiseks loodud ka efektiivsemad käsud „**sipush n**“, mis lisab *short* tüüpi väärtuse magasini ning seega vajab kahe-baidilist argumenti, „**bipush n**“, mis lisab *byte* tüüpi väärtuse magasini ning vajab vaid ühebaidilist argumenti, ning „**iconst_n**“, mis on spetsialiseeritud käsk vahemikus 0..5 väärtuse magasini lisamiseks ning seega ei vaja ühtegi argumenti.

2.8.4 Kontrollkäsud

Programmid peavad olema suutelised teatud koodijuppe korduvalt läbima või teatud koodijuppe vahele jätma. Kõrgema taseme keeltes on tihti programmeerijale kättesaadavad programmivoo juhtimiseks märksõnad *for*, *while* ja *if*, mis teevad voo kontrollimise lihtsaks, kuna neil struktuuridel on alati vaid üks algus- ja üks lõpp-punkt.

JVM selliseid struktuure otseselt ei toeta. Koodi ei ole võimalik jagada väiksemateks tükkideks ja struktuurideks, kuna nii vähim kui ka suurim ühik koodi JVM-is on instruktsioon. Tsüklite ja if-lausete teotamiseks kasutatakse JVM-is *goto* käsku, millega on võimalik teatud hulga instruktsioone edasi või tagasi koodis hüpata. *Goto* käsu parsimisega tegeleb raamistik Põder iseseisvalt, seega tuleb analüüsi loomiseks korrektselt implementeerida vaid tingimuslaused.

Tingimuslauseid on JVM-is kahte tüüpi. Käsk „**if_icmpx**“ eemaldab magasinist kaks ülemist väärtust ning võrdleb neid omavahel vastavalt x-iga määratud operaatori järgi. Käsk „**ifx**“ võrdleb magasini esimest väärtust konstandi 0-iga vastavalt *x*-iga määratud operaatori järgi. Operaatoreid on kokku kuus:

1) *EQ* – ekvivalentsus,
2) *NE* – mitte-ekvivalentsus,
3) *GT* – rangelt suurem,
4) *GE* – suurem või võrdne,
5) *LT* – rangelt väiksem,
6) *LE* – väiksem või võrdne.

3 Intervallanalüüsi teostus

Intervallanalüüsi moodul loodi raamistikus Põder, kasutades programmeerimiskeelt Scala. Loodud mooduli kood on kättesaadaval autori-nimelises harus raamistiku Põder koodirepositooriumis [12]. Repositooriumis on ka juhend mooduli kasutamiseks.

Mooduli hõlpsamaks loomiseks oli autoril kasutada juba raamistikus asuv ning töötav loogiline analüüs, mille tulemusi sai intervallanalüüsi mooduli tulemustega võrrelda. Samuti loodi enne töö alustamist erinevad testprogrammid, vt lisa 1, mida oleks moodulil tarvis korrektselt analüüsida. Testprogrammid muutusid järjest keerukamaks, et saaks funktsionaalsust järk-järgult implementeerida, kuid olid siiski piisavalt lihtsad, et analüüsi protsessi ja tulemusi oleks võimalik käsitsi kontrollida.

3.1 Analüüsi domeen

Analüüsi domeen näitab programmi olekut vastavas programmi punktis. Tegemist on andmestruktuuriga, mis sisaldab endas kõike analüüsiks vajalikku infot programmi antud seisundi kohta. Analüüsi domeen algväärtustatakse analüüsi alustades. Igal analüüsitaval sammul muudetakse eelmist domeeni vastavalt analüüsis läbitavale instruktsioonile ning analüüsi lõpuks näitab domeen analüüsi lõpptulemust. See tähendab, et intervallanalüüsi korral peab ta sisaldama endas sarnaselt JVM-ile selleks hetkeks defineeritud lokaalsete muutujate massiivi ning ajutiste muutujate magasini. Seega sai programmi domeeniks defineeritud kahe-elemendiline ennik, mis mustrisobituse eesmärgil sisaldab *Optional*-tüüpi massiivi ja sõnastikku, vt joonis 8. Massiivis hoitakse ajutisi muutujaid ning sõnastikus nimelisi lokaalseid muutujaid.

```
type IntervalDomain = (Option[List[Interval]],
                       Option[Map[String, Interval]])
```

Joonis 8. Programmi domeen.

Raamistiku loogika jaoks on vajalik, et programmi domeen käituks võrena, sest kõik eelnevalt välja toodud staatilise analüüsi teooria ning funktsioonid ja operaatorid kehtivad samuti ka programmi domeeni peal. Seega peab programmi domeenil määratletud olema *top* ja *bottom* väärtused ning järjestamise, kitsendamise ja laiendamise operaatorid.

Top-iks määratleme olukorra, kus programmi seisundi kirjeldamiseks puudub meil igasugune info, ning *bottom*-iks programmis tekkinud veaseisundi. Seega defineeritud programmi domeeni järgi saame määratleda neid joonisel 9 näidatud viisil.

```
val top = (Some(List()), Some(Map()))
val bottom = (None, None)
```

Joonis 9. *Top* ja *bottom* väärtused.

Laiendamise operaatoriks määratleme funktsiooni, mis elemendipõhiselt teostab intervallide laiendamist. Kuna domeeni üldine laiendamine ei ole võimalik, siis tuleb nii ajutiste muutujate massiivi kui ka lokaalsete muutujate sõnastikku vaadelda eraldi. Mõlemas andmestruktuuris tuleb eraldi läbi käia kõik muutujad ning neid vastavalt laiendada.

Seega tuleb vaadelda mõlemat olukorda mustrisobituse kaudu niimoodi, et kui ühel programmidomeenidest puudub informatsioon muutujate kohta, siis läheb käiku teise domeeni muutujate kohta kogutud informatsioon. Ning mõlema olemasolu korral toimub kitsendamine või laiendamine elemendipõhiselt kasutades funktsionaalprogrammeerimise printsiipe.

Et domeen koosneb erinevatest muutujatest, siis kasutatakse muutujate analüüsimiseks eelnevalt kirjeldatud intervallvõret, kus on määratletud samamoodi *top* ja *bottom* väärtused ning vajalikud üleminekufunktsioonid.

Intervallvõre implementeerimiseks defineerime *Num*-tüübilised klassid, mis hõlmavad endast kolme erinevat väärtust: $+\infty$, $-\infty$ või arvuline väärtus *IntNum(n)*. See võimaldab meil defineerida aritmeetilised funktsioonid *Num*-tüüpi väärtustele ning defineerida täisarvuline intervall $[a, b]$, kus nii a kui ka b on *Num*-tüüpi muutujad. Seega saame vastavalt eelnevale teooriale määrata intervalldomeeni *top* väärtuseks $[-\infty; +\infty]$ ning *bottom* väärtuseks $[+\infty; -\infty]$. Defineerime samuti ka liitmisfunktsionaalsuse, et saaksime hiljem intervalle omavahel liita ja lahutada, võrdlusfunktsionaalsuse võrdlusinstruktsioonide implementeerimiseks ning vähima ja suurima elemendi leidmise funktsioonid.

Näidiseks on joonisel 10 välja toodud programmi domeen, kus ajutiste muutujate massiivis asub parasjagu üks element intervallvahemikuga $[0, 1]$. See tähendab, et antud muutuja võib selles programmi punktis omada väärtusi 0 ja 1. Samuti on näha, et programmis on defineeritud ka üks lokaalne muutuja vahemikuga $[-\infty, +\infty]$, mis tähendab, et täpsemat väärtust selleks programmi punktiks talle veel omistatud ei ole.

```
val d = (Some(List(Interval(IntNum(0), IntNum(1)))),
         Some(Map("loc0" -> Interval(MInf, PInf))))
```

Joonis 10. Näidisdomeen.

3.2 Intervallvõre

Intervallvõre loome vastavalt eelnevalt kirjeldatud teooriale ja raamistikus kirjeldatud definitsioonile, vt joonis 11. Võre elementide tüübiks kasutame programmi domeeni *IntervalDomain*, seega implementeerime funktsioonid *join*, *leq*, *top*, *bot*, *widen*, *narrow* eelnevalt määratud definitsioone kasutades.

```
trait Lattice[T] {
  def join(x: T, y: T): T
  def leq(x: T, y: T): Boolean
  def top: T
  def bot: T
  def toJson[jv](x: T)(implicit j: JsonF[jv]): jv
  def widen(oldv: T, newv: T): T
  def narrow(oldv: T, newv: T): T
}
```

Joonis 11. Lihtsustatud võre definitsioon.

Funktsioon *toJson* on kasutusel võre ühe elemendi kuvamiseks kasutajale raamistiku kasutajaliideses. Üheks elemendiks on intervallanalüüsi puhul *IntervalDomain*, seega tuleb

funktsioonis luua nii ajutistest kui ka lokaalsetest muutujatest sõne, mida kuvatakse graafiliselt kasutajale ning mis annab ülevaate programmi olekust igal analüüsi sammul.

3.2.1 Funktsiooni *join* implementatsioon

Illustreerivaks näiteks vaatame funktsiooni *join* implementatsiooni, vt joonis 12, millele sarnaselt on implementeeritud ka ülejäänud funktsioonid.

Funktsiooni *join* kasutatakse kahe oleku omavahel ühendamisel, nt tingimuslause kahe erineva haru analüüsi tulemuste kokku liitmiseks. Esmalt kontrollitakse, kas kumbki olek on *top* või *bottom*. Kui üks olekutest on *bottom*, siis järelikult seda haru ei uuritud või jõuti seal harus vastuoluni ning seda olekut edasises analüüsis ei kasutata. Kui üks olekutest on *top*, siis see tähendab, et programmi oleku kohta ei oma analüsaator kindlat infot ning edasiseks analüüsiks propageeritakse samuti edasi *top* väärtust.

Ajutiste ja lokaalsete muutujate jaoks tuleb arvestada iga muutuja mõlema haru väärtustega. Konstanti *Interval(PInf, MInf)* kasutatakse juhul, kui ühes harus antud muutujal väärtus puudub. Kuna ühendamisel tuleb leida vähim võimalik vahemik, mis mahutab mõlemas harus analüüsi tulemusel leitud muutuja väärtuseid, siis sellise konstandi kasutamisega on võimalik koodi lihtsustada, kuid samas saavutada ikkagi vajalik parim tulemus.

```
override def join(x: IntervalDomain, y: IntervalDomain): IntervalDomain = {

  if (x == top || y == top) return top
  else if (x == bot) return y
  else if (y == bot) return x
  else if (x == y) return x

  val stack = (x._1, y._1) match {
      case (None, None) => None
      case (Some(e), None) => Some(e)
      case (None, Some(e)) => Some(e)
      case (Some(a), Some(b)) =>
        Some(
            a.zipAll(b, Interval(PInf, MInf), Interval(PInf, MInf))
              .map(e => joinIntervals(e._1, e._2))
        )
  }

  val lVars: Option[Map[String, Interval]] = (x._2, y._2) match {
      case (None, None) => None
      case (Some(e), None) => Some(e)
      case (None, Some(e)) => Some(e)
      case (Some(a), Some(b)) =>
        Some(
            (a.toSeq ++ b.toSeq)
              .groupBy(_._1)
              .mapValues(_.map(_._2).reduce(joinIntervals))
        )
  }

  (stack, lVars)
}
```

Joonis 12. Intervallvõre ⊔ (*join*) implementatsioon.

3.3 Hargnemise täpsem käsitlus

Programmides toimub juhtvoo hargnemine tingimuslausete juures. Iga tingimuslause kohta tekib kaks tingimuslause haru, mida tuleb eraldi analüüsida ning hiljem harude ühinemisel mõlema haru tulemused ühendada. Näiteks oli joonisel 3 näha, kuidas tingimuslause tulemusena muutuja x väärtus muutus vastavalt sellele, kumba haru programm läbis. Staatilises analüüsis programmi ei käivitata, seega tuleb lähtekoodi abil võimalikult täpselt määratleda seda, kas ja milliseid harusid programmi käivitamisel läbida võidakse.

Harud tekivad programmis enamasti kahel juhul: kas programmis kasutatakse spetsiifiliselt mõnda tingimuslauset või on programmis mõni tsükkel. Tsükli puhul luuakse samamoodi tsükli päisesse tingimuslause, mille järgi otsustatakse, kas programm siseneb tsükli kehasse või väljutakse tsüklist.

Nagu teoreetilises osas kirjeldati, siis JVM baitkoodis on tingimuslauseteks instruktsioonid, mis kas võrdlevad ajutiste muutujate pinus kahte ülemist väärtust või võrreldakse pinu kõige ülemist väärtust 0-iga. Selleks laetakse enne tingimuslause instruktsiooni ajutiste muutujate pinusse vastavad konstant- või siis lokaalsete muutujate väärtused. Et staatiline analüsaator analüüsib programmi käsk-käsu haaval, siis tingimuslause instruktsioonini jõudes ei oma analüsaator infot selle kohta, mis muutujaid võrreldakse, kuna ajutiste muutujate pinus asuvad vaid intervallväärtused. Sellega aga tekib probleem tsüklite kehas kitsendamise

kasutamisega, sest kitsendamise jaoks on vaja infot selle kohta, mis muutuja järgi tsüklisse sisenetakse. Näiteks kui tsükli tingimuseks on $x < 1000$, siis kitsendamise sooritamiseks peame täpselt teadma, et just muutuja *x* on see, mille intervallvahemiku ülemine väärtus on tsüklisse sisenemisel maksimaalselt 999.

Selle probleemi lahendamiseks võtame programmi domeeni kasutusele kolmanda andmestruktuuri, milles hoiame lisainformatsiooni ajutiste muutujate massiivis asuvate väärtuste kohta. Seega iga kord, kui laetakse uus väärtus ajutiste muutujate massiivi, lisame me ka lisainformatsiooni massiivi info selle väärtuse kohta. Joonisel 13 on näha, et ajutiste muutujate massiivi on kontrollkäsu täitmiseks laetud kaks intervallväärtust $[0, 5]$ ja $[3, 3]$. Lisainformatsiooni massiivi järgi saame nüüd täpselt määratleda, et esimene intervallvahemik $[0, 5]$ tähistab lokaalse muutuja „loc0“ väärtust ning teine tavalist konstantväärtust.

```
val d = (Some(List(
              Interval(IntNum(0), IntNum(5)),
              Interval(IntNum(3), IntNum(3)))
             ),
         Some(Map("loc0" -> Interval(0, 5))),
         Some(List(
              LVar(0),
              Const))
)
```

Joonis 13. Näidisdomeen koos lisainformatsiooni massiiviga.

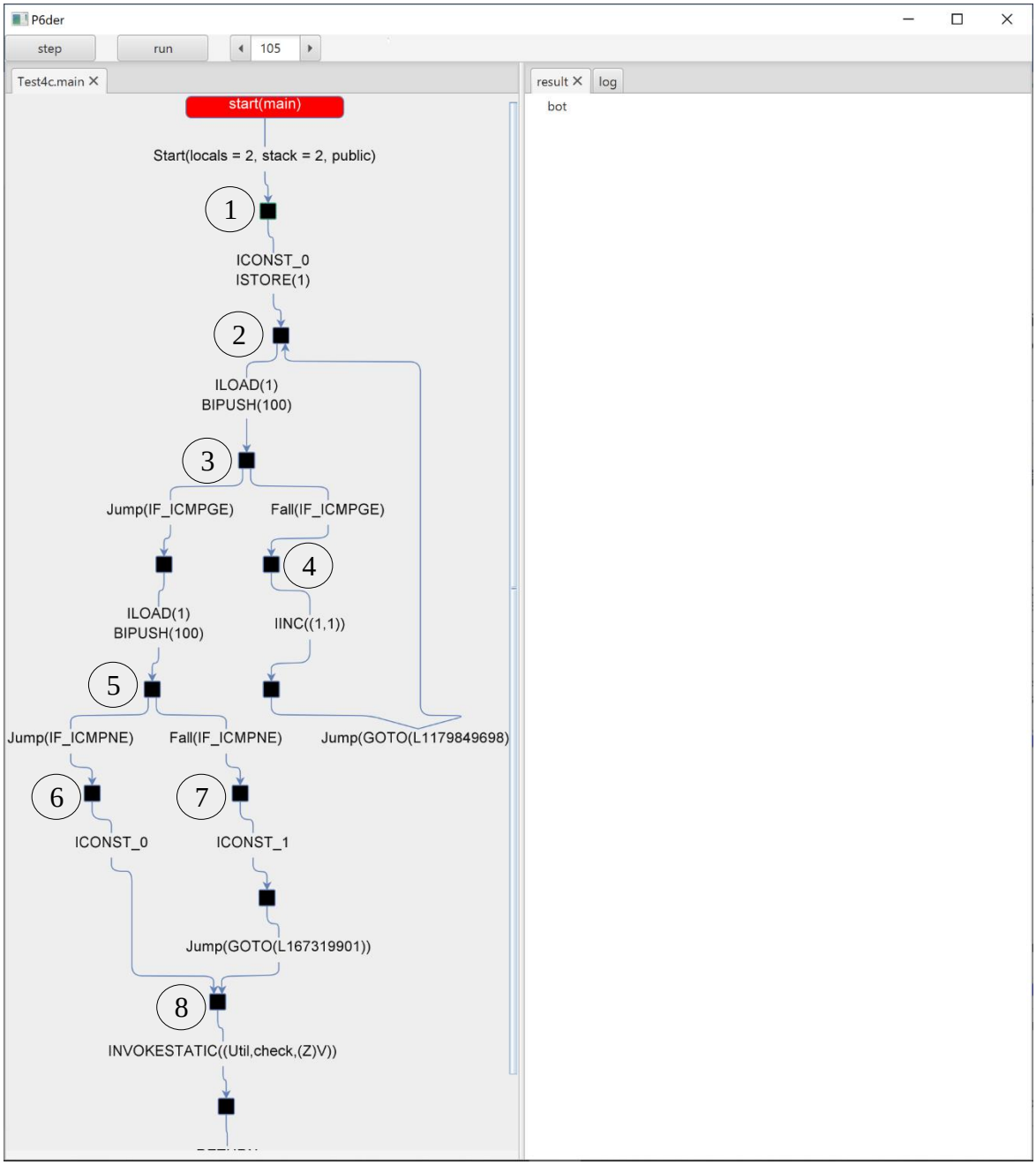
3.4 Testprogrammi intervallanalüüs

Vaatleme lähemalt ühe tsüklilise testprogrammi staatilise analüüsi käiku, kasutades loodud intervallanalüüsi moodulit ning raamistikku Põder. Testprogramm sai valitud selline, et ta hõlmaks üheaegselt nii lokaalsete kui ka ajutiste muutujate, lihtsa aritmeetika ning laiendamise ja kitsendamise käsitlust. Testprogramm sisaldab ka kahte tingimuslauset, kus esimesel korral tuleb uurida mõlemat tingimuslause haru ning teisel korral tuleb üks harudest analüüsist vabastada. Testprogrammi lähtekood on välja toodud joonisel 14.

```
class Test4c {
  public static void main(String[] args) {
    int x = 0;
    while (x < 100) x += 1;
    Util.check(x == 100);
  }
}
```

Joonis 14. Tsükliga testprogrammi lähtekood.

Staatilise analüüsi eesmärgiks on välja selgitada meetodi *Util.check()* väljakutsel kasutatava argumendi tõeväärtus. Selleks luuakse esmalt raamistiku Põder abil programmi juhtvoograaf, mida on näha joonise 15 vasakpoolsel paneelil, ning algväärtustatakse analüüsi domeen, mille tekstilist esitust on näha joonise 15 parempoolsel paneelil. Järgmisena teostatakse analüüs läbides programm instruktsioon-haaval ning lõplikku tulemust näeb kasutaja kas logidest või hiirega teda huvitava graafi tipu peale klõpsates.

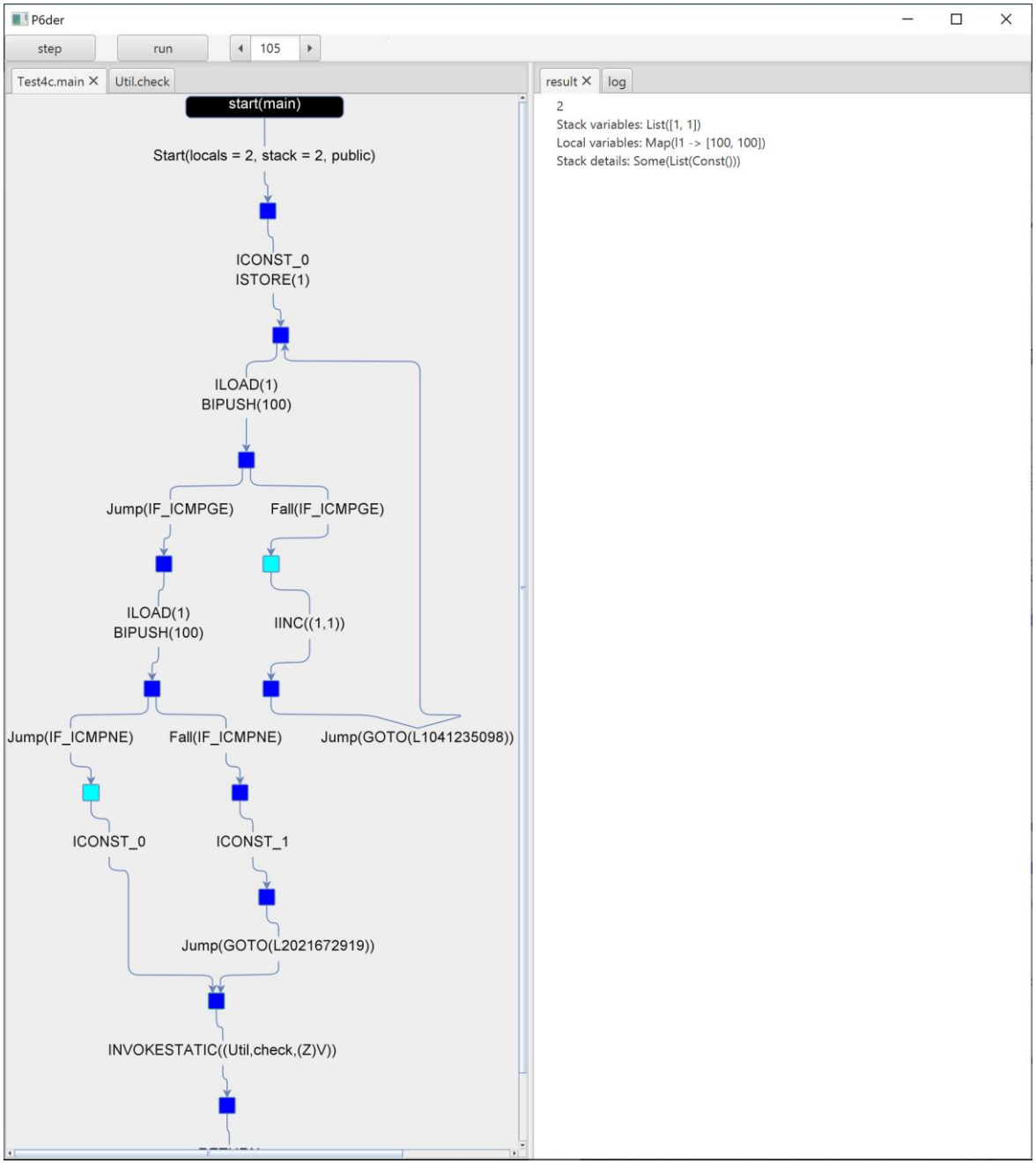


Joonis 15. Raamistiku Põder graafiline kasutajaliides.

- Punktis 1 algväärtustatakse domeen. Ajutiste ja lokaalsete muutujate massiiv ja sõnastik on tühjad.
- Punkti 2 esimest korda jõudes on defineeritud muutuja *x* ning väärtustatu ta 0-iga.
- Punktis 3 kontrollitakse esimesel korral, kas *x* on suurem kui 100. Kuna vastus on kindlasti väär, siis sisenetakse tsüklisse.
- Punktis 4 proovitakse teostada laiendamist vastavalt eelmisel läbimisel kasutatud domeeniga. Kuna esimesel korral punkti läbides ei ole eelmist domeeni olemas, siis kasutatakse vaikeväärtusena *bottom*-it ning laiendamise tulemusel domeen ei muutu.
- Järgnevalt suurendatakse *x*-i ühe võrra ning jõutakse tagasi punktidesse 2 ja 3. Et punktis 2 oli esimesel läbimisel $x \in [0, 0]$ ning praeguses domeenis on $x \in [1, 1]$, siis ühendatakse need väärtused ja teisel läbimisel punkti 3 jõudes kehtib $x \in [0, 1]$.

- Jällegi liigutakse punkti 4, kuid kuna seekord on *x*-i väärtus võrreldes eelmisega kasvanud, siis laiendamise tulemusel $x \in [0, +\infty]$. Seega vastavalt kehtib sama ka järgnevalt punktidesse 2 ja 3 jõudes.
- Punktis 3 ei saa seekord kumbagi haru kindlalt välistada, seega analüüsitakse tsükli keha, lootes teostada kitsendamist. Selleks nenditakse, et tsükli kehasse sisenetakse ainult juhul, kui $x < 100$. Kitsenduse tulemusel jõutakse järgneva tulemuseni. $x \in ([0, +\infty] \,\Delta\, [0, 99]) \rightarrow x \in [0, 99]$.
- Järgnevalt punkti 3 jõudes on lõpmatused kitsendamise tulemusel ära kaotatud ning $x \in [0, 100]$. Raamistik tunneb ära, et tsükli keha on läbi uuritud ning seega võib analüüsiga edasi liikuda. Selleks kasutatakse sarnaselt ära asjaolu, et tsüklist väljudes peab kehtima $x \geq 100$. Muutujate ülemised ja alumised piirväärtused leitakse võre alamhulkade alumist raja leides.
  $x \in ([100, +\infty] \sqcap [100, 100]) \rightarrow x \in [100, 100]$.
- Punktis 5 kontrollitakse *x*-i mitte võrdumist 100-ga. Et muutuja *x* on aga kindlasti 100, siis välistatakse vastav tingimuslause haru, propageerides sinna *bottom*, ning analüüsitakse ainult teist haru.
- Punkti 6 programm oma töö käigus ei jõua, seega siin on analüüsi domeeni väärtuseks *bottom*.
- Punktis 7 lisatakse ajutiste muutujate magasini intervall $[1, 1]$.
- Punktis 8 ühendatakse mõlema haru analüüsil leitud tulemused. Kuna üks domeenidest oli väärtustatud *bottom*-iks, siis läheb kasutusse teise haru domeeni. Näeme, et meetodi *Util.check()* väljakutse ajal on ajutiste muutujate magasini kõige ülemiseks elemendiks intervall $[1, 1]$, mis tähistab kindlalt tõeset väärtust ning mis funktsiooni välja kutsudes muutub antud funktsiooni argumendiks (vt joonis 16).

□



Joonis 16. Analüüsi lõpptulem raamistiku Põder graafilises kasutajaliideses.

3.5 Tulemused

Loodud analüsaatorit testis autor erinevate Java programmide kompileerimise tulemusel saadud JVM baitkoodidega. Kompileerimiseks kasutati viimast saadaolevat Java kompilaatorit (Java SE 12.0.1[3]). Samuti testiti analüsaatorit Scala programmidega, mille kompileerimiseks kasutati viimast Scala kompilaatorit (Scala 2.12.8[4]). Java testprogrammid jaotati nelja erinevasse kategooriasse, mille iga kategooria ühe programmi lähtekood on välja toodud lisas 1. Kategooriatesse jagamine võimaldas testimist üldistada ja mugavdada.

[3] https://www.oracle.com/technetwork/java/javase/downloads/jdk12-downloads-5295953.html
[4] https://www.scala-lang.org/download/2.12.8.html

Testprogrammid kontrollisid analüsaatori peamist funktsionaalsust. Selleks loodi eraldi abifunktsioon, mille argument peaks testides alati tõene olema. Analüsaator on suuteline tuvastama baitkoodis antud funktsiooni ning kontrollima, kas funktsioonile antav argument on tõene või väär, ning vastavalt sellele kasutajat teavitama.

Testgrupis 1 kontrolliti nii ajutiste kui ka lokaalsete muutujate kasutamist ning aritmeetika- ja võrdlusinstruktsioonide analüüsi. Nende testide läbimiseks tuli välja mõelda ja luua vastavad andmestruktuurid muutujate analüüsi ajaks mälus hoidmiseks ning implementeerida tavapärased aritmeetika- ja võrdlusoperatsioonid intervallide peal.

Testgrupis 2 kontrolliti spetsiaalsete väliste funktsioonide väljakutsumist ja tingimuslausete erinevate harude analüüsi. Spetsiaalseteks välisteks funktsioonideks nimetame funktsioone, mida ei ole mõtet eraldi analüüsida, kuna nende funktsioonide tulemus on ilma analüüsita teada. Enamasti on tegemist sisseehitatud funktsioonidega, näiteks *Math.random()*, kuid kasutaja võib ka määrata eraldi mõne funktsiooni spetsiaalseks. Selle testgrupi läbimiseks tuli osaliselt implementeerida funktsioonide väljakutse analüüs, mis hõlmas spetsiaalsete väliste funktsioonide tuvastajat, ning intervallvõrel pidi olema implementeeritud korrektne *join* funktsioon.

Testgrupis 3 kontrolliti tsüklite töötamist. Testiti nii *while*- kui ka *for*-tsükleid, mis kompileerides väga sarnasteks baitkoodideks muutuvad. Ilma laiendamise ja kitsendamiseta võib tsüklite analüüs võtta ebaproportsionaalselt kaua aega, seega tuli siin etapis implementeerida korrektsed laiendamis- ja kitsendamisfunktsioonid.

Testgrupis 4 kontrolliti tavaliste funktsioonide väljakutseid. Selleks tuli implementeerida raamistiku-spetsiifilised funktsioonide välja kutsumise meetodid, et analüsaator oskaks funktsioonide vahet käia ja vajaduse korral programmidomeeni uuendada või muuta.

Testides leiti, et intervallanalüüsi moodul on suuteline kõik testid läbima ning testitud funktsionaalsust korrektselt analüüsima. Seega töötab moodulis aritmeetika, tingimuslausete, tsüklite ja ka erinevate funktsioonide väljakutsete analüüs ning implementeeritud on kõik vajalikud JVM baitkoodi instruktsioonid.

3.6 Puudused

Mooduli teadaolevateks puudusteks on parasjagu suutmatus korrektselt analüüsida programmis erindite viskamisel ja püüdmisel tekkinud olukordi. Selleks on raamistikus vajalik funktsionaalsus ja võimekus olemas, kuid moodulis ei ole implementeeritud vastavad funktsioonid seisundi uuendamiseks ja taastamiseks.

Samuti ei ole implementeeritud mustrid iga keerulisema kitsendamise juhu jaoks. Moodul on küll suuteline jälgima, et ajutiste muutujate pinus on väärtus, mis tähistab kindlale lokaalsele muutujale lisatud või lahutatud teist väärtust, näiteks *Add(LVar(0), Const(5))*, kuid kitsendamise jaoks ei võeta sellist informatsiooni arvesse. See tähendab, et kitsendamine toimib näiteks tsükli tingimusega $x < 10$, kuid ei toimi tingimusega $x + 5 < 10$.

4 Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli luua raamistikku Põder staatilise analüüsi moodul, mis oleks suuteline intervallanalüüsi kasutades JVM baitkoodiks kompileeritud programmidel teostama täisarvuliste muutujate staatilist analüüsi. Intervallanalüüsi moodul implementeeriti programmeerimiskeeles Scala.

Töö teoreetilises osas kirjeldati staatilise analüüsi põhimõtteid ning anti ülevaade staatilise analüüsi loomiseks vajalikust matemaatilisest võreteooriast. Selleks vaadeldi täpsemalt võrede omadusi ja võredel defineeritud operaatoreid, mida hiljem intervallanalüüsi mooduli loomiseks kasutati. Samuti kirjeldati raamistikku Põder ning anti täpsem ülevaade JVM-ist (*Java Virtual Machine*) ja JVM baitkoodi instruktsioonidest.

Töö praktilises osas loodi staatilise analüüsi raamistikku Põder intervallanalüüsi moodul, mille ülesanne oli analüüsitavatel programmidel teostada täisarvuliste muutujate analüüsi. See tähendab, et moodul oli suuteline igas programmipunktis määratlema nii ajutiste kui ka lokaalsete täisarvuliste muutujate vähimat ja suurimat võimalikku väärtust. Selleks kompileeriti analüüsitav programm JVM baitkoodi, mis raamistiku Põder abil analüüsiks ette valmistati ning mille instruktsioonide läbivaatluse abil teostati programmile staatiline intervallanalüüs.

Antud töö praktilise osa tulemusel on võimalik teostada intervallanalüüsi programmidele, mis kompileerivad JVM baitkoodiks. Analüüsi käigus on võimalik nendes programmides tuvastada täisarvuliste muutujate käsitlemisel tekkinud vigu, näiteks tuvastada kättesaamatuid tingimuslausete harusid või kontrollida funktsioonide väljakutsetel kasutatavate argumentide õigsust. Samuti on analüüsi kasutajal võimalik iga programmipunkti seisundit eraldi käsitsi uurida, et leida rohkem vigu või kontrollida programmi töö käiku.

Lõputöös loodud moodulit on võimalik edasi arendada, lisades juurde korrektse erindite analüüsi võimekuse. Kui programmis visatakse erind, mis mujal kinni püütakse, siis võiks moodul olla suuteline sealt analüüsi jätkama. Samuti on võimalik asendada intervallanalüüs intervallhulkade analüüsiga, mis teatud olukordades võib lubada teostada praegusest täpsemat analüüsi.

5 Viidatud kirjandus

[1] R. J. v. Glabbeek. [Võrgumaterjal]. Kättesaadav aadressil: http://kilby.stanford.edu/~rvg/154/handouts/Rice.html. [Kasutatud 30. märts, 2019].

[2] V. Laan, „Võreteooria. Loengukonspekt. Sügis 2017“ [Võrgumaterjal]. Kättesaadav aadressil: https://courses.ms.ut.ee/MTMM.00.039/2017_fall/uploads/Main/kon.pdf. [Kasutatud 5. aprill, 2019].

[3] A. Møller ja M. I. Schwartzbach, „Static Program Analysis“ [Võrgumaterjal]. Kättesaadav aadressil: https://cs.au.dk/~amoeller/spa/spa.pdf. [Kasutatud 8. aprill, 2019].

[4] V. Vene, „Staatiline analüüs. Abstraktne interpretatsioon“ [Võrgumaterjal]. Kättesaadav aadressil: http://kodu.ut.ee/~varmo/SEM04/slides/stanal.pdf. [Kasutatud 30. märts, 2019].

[5] „Raamistik Põder. Lähtekood“ [Võrgumaterjal]. Kättesaadav aadressil: https://bitbucket.org/kalmera/poder. [Kasutatud 25. veebruar, 2019].

[6] B. Venners, „The lean, mean, virtual machine“ Javaworld, [Võrgumaterjal]. Kättesaadav aadressil: https://www.javaworld.com/article/2077184/the-lean--mean--virtual-machine.html. [Kasutatud 30. aprill, 2019].

[7] B. Venners, „Bytecode basics“, Javaworld [Võrgumaterjal]. Kättesaadav aadressil: https://www.javaworld.com/article/2077233/bytecode-basics.html. [Kasutatud 30. aprill, 2019].

[8] G. Nolan, Decompiling Java, 2004.

[9] T. Römer ja R. Raudjärv, „Java baitkood“ [Võrgumaterjal]. Kättesaadav aadressil: http://kodu.ut.ee/~isotamm/PKeeled/RaudjRr/baitkood.pdf. [Kasutatud 18. aprill, 2019].

[10] Oracle. [Võrgumaterjal]. Kättesaadav aadressil: https://docs.oracle.com/javase/specs/jvms/se12/html/index.html. [Kasutatud 18. aprill, 2019].

[11] A. Wesley, Programming for the Java Virtual Machine, 2014.

[12] „Raamistik Põder. Intervallanalüüsi mooduli lähtekood“ [Võrgumaterjal]. Kättesaadav aadressil: https://bitbucket.org/kalmera/poder/branch/sinisalu-interval-analysis. [Kasutatud 7. aprill, 2019].

Lisad

I. Testprogrammid

```
class Util {
  public static void check(boolean b) {
    assert(b);
    // Analüüs peaks meetodi `Util.check` kutsel väljastama
    // hoiatuse, kui argument pole true.
  }
}

class Test1 {
  public static void main(String[] args) {
    int x = 1;
    Util.check(x==1);
    x = 3;
    Util.check(x!=1);
    Util.check(x==3);
    int y = x + 1;
    Util.check(y==4);
    Util.check(y>x);
    x = (3*y)/2;
    Util.check(x==6);
  }
}

class Test2 {
  public static void main(String[] args) {
    int x = 0;
    if (Math.random() <= 0.5) {
      x = 10;
      Util.check(x==10);
    } else {
      x = 20;
      Util.check(x==20);
    }
    Util.check(x>=10);
    Util.check(x<=20);
  }
}

class Test3 {
  public static void main(String[] args) {
    int x = 0;
    for (;Math.random() * 2 <= 1; x++) { }
    Util.check(x>=0);
    Util.check(!(x<1000));

    for (x = 0; x < 100; x++) { }
    Util.check(x==100);
  }
}
```

Joonis 17. Testprogrammid 1, 2, 3 ning abifunktsioon.

```
class Test4 {
  private static int incr(int i) {
    return i+1;
  }
  private static int viis() {
    return 5;
  }
  private static void m(int q, int k) {
    Util.check(q==20);
  }
  public static void main(String[] args) {
    int x = 1 + viis();
    Util.check(x==6);

    x = 20;
    int y = 5;
    m(x, y);

    x = 30;
    x = incr(incr(x));
    Util.check(x==32);
  }
}
```

Joonis 18. Testprogramm 4.

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Andre Sinisalu**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose
   Java programmide staatiline intervallanalüüs raamistikus Põder,
   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.

Andre Sinisalu

10.05.2019