TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Mirjam Iher

Nõrgima eeltingimuse staatiline analüüs pinukeeltele

Bakalaureusetöö (9 EAP)

Juhendaja: Kalmer Apinis, PhD
Juhendaja: Vesal Vojdani, PhD

Tartu 2019

Nõrgima eeltingimuse staatiline analüüs pinukeeltele

Lühikokkuvõte:

Staatiline analüüs on üks viis programmide uurimiseks. Vastandina dünaaamilisele analüüsile ei pea staatilise analüüsi jaoks analüüsitavat koodi käivitama. Seetõttu võimaldab staatiline analüüs ohutumat ning üldisemat analüüsi kui dünaamiline.

Käesolevas töös keskendutakse nõrgima eeltingimuse staatilisele analüüsile alt üles meetodil. Selle meetodi korrektsuse näitamiseks kasutatakse analüüsitavate programmide abstraksete mäluga pinumasinatena mudelleerimist. Alt-üles lähenemise eeliseks traditsiooniliste ülalt-alla meetodi ees on programmi semantika ebaoluliste osade analüüsi vältimine.

Töö väljundina teostatakse staatilise analüüsi raamistikus Põder nõrgima eeltingimuse alt üles leidmise analüüs.

Võtmesõnad:
staatiline analüüs, nõrgim eeltingimus, pinumasin, Java baitkood

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

Weakest precondition static analysis for stack machines

Abstract:

Static analysis is a way to inspect software. As opposed to dynamic analysis, code does not need to be executed in order to be analysed. This property allows for safer and more general analysis than dynamic analysis.

This thesis focuses on weakest precondition computation through a backwards static analysis. To show the soundness of this approach, the programs to be analysed are modelled as memory-augmented stack machines. The advangate of the backwards method compared to traditional forward analyses is avoiding analysing inconsequential properties of the program's semantics during verification.

As a practical contrubition of this thesis, the weakest precondition analysis is implemented in the Põder static analyisis framework.

Keywords:
static analysis, weakest precondition, stack machine, Java bytecode

CERCS:

P170 Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus

Nõrgima eeltingimuse staatiline analüüs on üks oluline viis programmide analüüsimiseks. See võimaldab programme analüüsida neid käivitamata ning üleliigseid osasid analüüsimata jättes. Nõrgima eeltingimuse leidmiseks kasutatakse alt-üles meetodit. Käesolevas töös implementeeritakse alt-üles meetod staatilise analüüsi raamistikus Põder, mis tugineb juhtimisvoograafi analüüsil. Üldiselt on levinumad teised staatilised analüüsid, kus programme uuritakse nende käitlusjärjekorras (näiteks intervallanalüüs). Ka alt üles meetodi valimise puhul on tavaline valik nõrgima eeltingimuste tuletamine kõrgemal abstraktsioonitasemel. Käesoleva töö raames koostatud analüüsi praktiline osa käsitleb aga tavalisi Java baitkoodi instruktsioone.

Töö eesmärk implementeerida nõrgima eeltingimuse analüsaator lihtsustatud Java baitkoodile [11], anda ülevaade pinukeeltest ja staatilisest analüüsist ning näidata, et programmi korrektsust võib olla mõistlik hinnata nõrgima eeltingimuse leidmisega meetodil. Programmi korrektsust on võimalik hinnata sobivalt valitud programmispetsiifiliste omaduste kehtimise kontrollimisega. Otsitakse minimaalseid eeldusi, mis peavad programmi alguses kehtima, et programmi lõpus kehtiks tingimus ψ. Neid eeldusi nimetatakse nõrgimaks eeltingimuseks. Kui nõrgimaks eeltingimuseks on samaselt tõene väide, siis tingimus ψ kehtib alati ning programm on korrektne.

Staatiline analüüs on programmide analüüsimine ilma neid käivitamata [3]. Pinukeeled on arvutusmudelid, kus vahetulemusi säilitatakse pinus [5]. Käesolevas töös uuritakse täpsemalt nõrgima eeltingimuse staatilist analüüsi mäluga pinukeeltel.

Töö esimeses sissejuhatavas peatükis antakse üldine ülevaade käsitletavatest teemadest. Teises peatükis tehakse ülevaade pinukeeltest ning spetsialiseerutakse mäluga pinukeeltele, millele antakse ka abstraktne definitsioon. Kolmandas peatükis kirjeldatakse lühidalt erinevaid staatilisi analüüse ning võrreldakse ülalt-alla ning alt-üles meetodeid. Neljandas peatükis keskendutakse alt-üles meetodile ehk leitakse nõrgimat eeltingimust. Esialgu vaadeldakse eeltingimusi abstraksetel kõrgkeeltel, edasi minnakse defineeritud pinukeelte ning Java baitkoodi analüüsimise juurde. Viiendas peatükis antakse tuuakse näiteid implementatsioonikoodist ning selgitatakse, kuidas analüüsitakse programme nõrgima eeltingimuse analüsaatoriga. Lõpetuseks tuuakse välja, mis õnnestus teostada.

2. Pinukeeled

Pinukeel on arvutusmudel, kus vahetulemusi salvestatakse pinus.

2.1 Pinu

Pinuks nimetatakse

- tühipinu $\sqcup$ või
- paari (s, ξ), kus $s \in S$ ning $\xi \in \Xi$,

kus S on pinusümbolite hulk ning Ξ kõigi pinude hulk. Saab defineerida pinusse elemendi lisamise tehte $\downarrow\colon S \times \Xi \to \Xi$, kus

$$s \downarrow \xi := (s, \xi),$$

mistahes $s \in S$ ja $\xi \in \Xi$ korral. Vastupidiselt, pinust pealmise elemendi võtmiseks on tehe $\uparrow\colon \Xi \to (S \cup \{\text{␣}\}) \times \Xi$, kus

$$\uparrow \sqcup := (\text{␣}, \sqcup), \qquad \uparrow (s, \xi) := (s, \xi).$$

Tehe $\uparrow$ annab üldjuhul tulemuseks paari võetud elemendist ning alles jäänud pinust. Tühipinust elementi võtta ei saa, sel juhul on vastusepaari esimeseks elemendiks vaikeelement ␣, mis väljendab pinusümboli puudumist.

Matemaatiliselt on mugav vaadelda pinu paarina pinusümbolist ja sabapinust, kuid tegemist on siiski mudeliga tavalisest pinust, kus pinusümbolid on üksteise peal pinus. Kusjuures paari esimene element on pinu pealmine, alumised pinusümbolid on sabapinus.

Üldiselt pinu kasutavad arvutusmudelid ei luba pinu teisiti mõjutada kui vaid pinusse peale elementi lisades või sealt pealt elementi võttes. Teisit öeldes, lubatud on vaid tehted $\uparrow$ ning $\downarrow$, sabapinu elementide mõjutamine pole lubatud.

2.2 Mäluga pinukeeled

Pinukeele arvutusmudelit kasutavates programmeerimiskeeltes kirjutatud programmid on **pinumasinad**. Tuntuim keel, mis pinukeele mudelit kasutab, on Forth [7]. Vaid pinukeele mudeli kasutamisel on probleem, et samu pinuelemente on tihti vaja pinus hoida mitmes kohas.

Mäluga pinukeeled on sellised pinukeeled, kus arvutuste vahetulemusi saab lisaks pinule hoida ka mälus. Selline arvutusmudel aitab vähendada andmete dubleerimist.

Java baitkood, mida analüüsib käesoleva töö praktiline osa, kasutab samuti mäluga pinukeele arvutusmudelit. Lähtudes Greibach'i [5] artiklis antud pinumasina definitsioonist, vaadeldakse antud töös Java baitkoodi mäluga pinukeelna, mille programmid on järgnevalt defineeritud mäluga pinumasinad:

Definition 1. Mäluga pinumasin on kaheksik $(Q, \Sigma, S, \delta, \xi_0, q_0, \lambda_0, Q_F)$, kus

- Q on olekute hulk;
- Σ on sisendsümbolite hulk;
- S on pinusümbolite hulk;
- $\delta : Q \times \Xi \times (\mathbb{N} \to S) \times \Sigma \to Q \times \Xi \times (\mathbb{N} \to S) \times \{0, 1\}$ on üleminekufunktsioon[1], kus 0 tähistab, et sisendsümbolit ei kasutatud ära ning 1 tähistab, et sisendsümbol kasutati ära;
- $\xi_0 \in \Xi$ on algpinu;
- $q_0 \in Q$ on algolek;
- $\lambda_0 \in (\mathbb{N} \to S)$ on mälu algolekfunktsioon;
- $Q_F \subseteq Q$ on lõppolekute hulk.

Antud töös ei keskenduta avaldiste tüüpidele, seega saab võtta pinusümboliteks ning sisendsümboliteks täisarvud. Neid kitsendusi arvestades on joonisel 1 kujutatud funktsioon võrdne mäluga pinumasinaga $P = (Q, \mathbb{Z}, \mathbb{Z}, \delta, \sqcup, \text{START}, \lambda_0, \{\text{RETURN}\})$, kus

[1] Käesolevas töös käsitletakse arvu 0 naturaalarvuna, st. $\mathbb{N} \equiv \{0, 1, 2, 3, 4, \dots\}$

- $Q = \{\text{START}, \text{ILOAD(0)}, \text{ICONST\_2}, \text{ISUB}, \text{ICONST\_2'}, \text{IADD}, \text{ISTORE(1)}, \text{ILOAD(0)'}, \text{ILOAD(1)}, \text{IF\_ICMP}, \text{CONST\_0}, \text{CONST\_1}, \text{PÕDER}, \text{RETURN}\}$;
- $\lambda_0(x) = \begin{cases} a, & \text{kui } x = 0, \\ 0, & \text{muidu;} \end{cases}$
- ning δ on defineeritud parajasti järgnevate sisenditega:

$$
\begin{array}{rrcccrrcc}
\delta(& \text{START}, & \xi, & \lambda_0, & \sigma) & = (& \text{ILOAD(0)}, & \xi, & \lambda_0, \quad 0); \\
\delta(& \text{ILOAD(1)}, & \xi, & \lambda_0, & \sigma) & = (& \text{ICONST\_2}, & (\lambda_0(0), \xi), & \lambda_0, \quad 0); \\
\delta(& \text{ICONST\_2}, & \xi, & \lambda_0, & \sigma) & = (& \text{ISUB}, & (2, \xi), & \lambda_0, \quad 0); \\
\delta(& \text{ISUB}, & (s_0, (s_1, \xi)), & \lambda_0, & \sigma) & = (& \text{ICONST\_2'}, & (s_1 - s_0, \xi), & \lambda_0, \quad 0); \\
\delta(& \text{ICONST\_2'}, & \xi, & \lambda_0, & \sigma) & = (& \text{IADD}, & (2, \xi), & \lambda_0, \quad 0); \\
\delta(& \text{IADD}, & (s_0, (s_1, \xi)), & \lambda_0, & \sigma) & = (& \text{ISTORE(1)}, & (s_1 + s_0, \xi), & \lambda_0, \quad 0); \\
\delta(& \text{ISTORE(1)}, & (s_0, \xi), & \lambda_0, & \sigma) & = (& \text{ILOAD(0)'}, & \xi, & \lambda_1, \quad 0),
\end{array}
$$

$$\text{kus} \quad \lambda_1(x) = \begin{cases} s_0, & \text{kui } x = 1, \\ \lambda_0(x), & \text{muidu;} \end{cases}$$

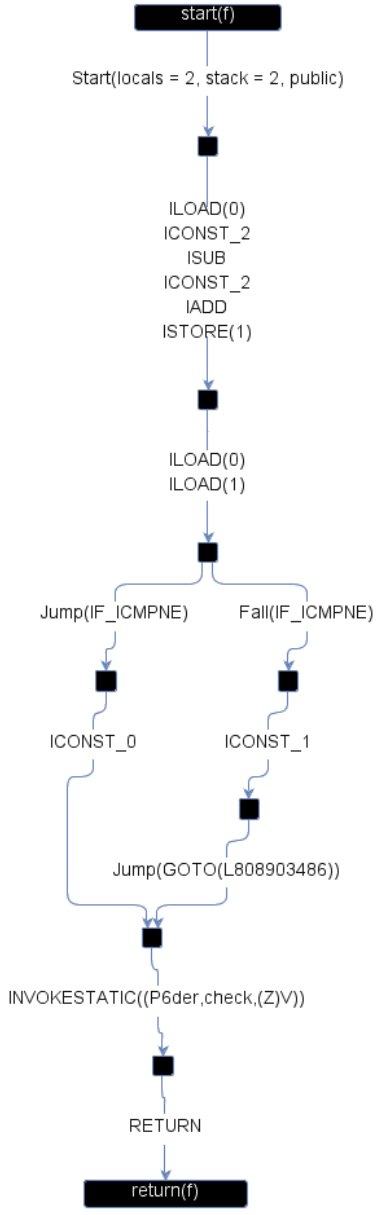
$$
\begin{array}{rrcccrrcc}
\delta(& \text{ILOAD(0)'}, & \xi, & \lambda_1, & \sigma) & = (& \text{ILOAD(1)}, & (\lambda_1(0), \xi), & \lambda_1, \quad 0); \\
\delta(& \text{ILOAD(1)}, & \xi, & \lambda_1, & \sigma) & = (& \text{IF\_ICMP}, & (\lambda_1(1), \xi), & \lambda_1, \quad 0); \\
\delta(& \text{IF\_ICMP}, & (s_0, (s_1, \xi)), & \lambda_1, & \sigma) & = (& q, & \xi, & \lambda_1, \quad 0);
\end{array}
$$

$$\text{kus} \quad q = \begin{cases} \text{ICONST\_1,}, & \text{kui } s_0 = s_1, \\ \text{ICONST\_0,}, & \text{kui } s_0 \neq s_1; \end{cases}$$

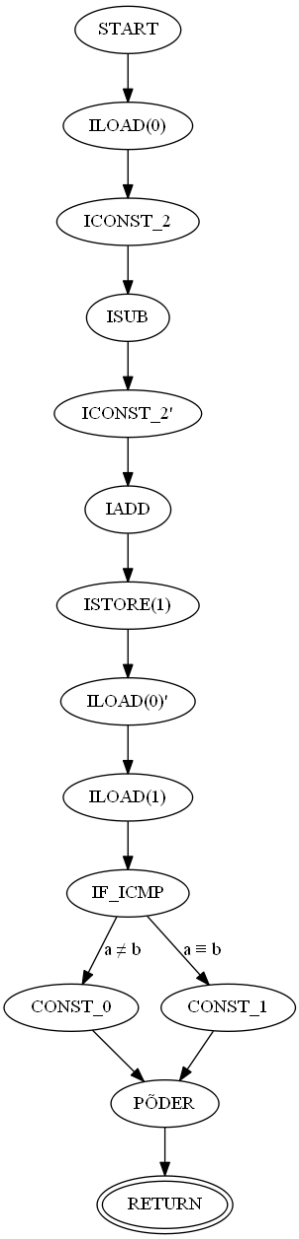
$$
\begin{array}{rrcccrrcc}
\delta(& \text{CONST\_0}, & \xi, & \lambda_1, & \sigma) & = (& \text{PÕDER}, & (0, \xi), & \lambda_1, \quad 0); \\
\delta(& \text{CONST\_1}, & \xi, & \lambda_1, & \sigma) & = (& \text{PÕDER}, & (1, \xi), & \lambda_1, \quad 0); \\
\delta(& \text{PÕDER}, & (s_0, \xi), & \lambda_1, & \sigma) & = (& \text{RETURN}, & \xi, & \lambda_1, \quad 0).
\end{array}
$$

```
public static void f(int a) {
  int b = a - 2 + 2
  assert(a == b)
}
```

Joonis 1. Lihtne analüüsitav Java funktsioon.



(a) Põdra analüüsi väljundi juures kujutatud instruktsioonide graaf.



(b) Pinumasina olekud.

Joonis 2. Joonisel 1 [2] kujutatud programmi graafid.

Joonisel 2b on toodud mäluga pinumasina P olekuteskeem. Olekuskeemil olekutele kirjutatud IADD, ICONST_2, jt on pinumasina kontekstis vaid olekute tähised, masina töö

[2] Põdraga analüüsides kastuatakse tingimuse c kehtimise kontrollimiseks `assert(c)` asemel `P6der.check(c)`

ei muutuks kuidagi, kui olekute nimed oleks abstraksed A, B, C jt. Sellised nimed on valitud, et aidata lugejal näha funktsiooni δ poolt defineeritud muutuste seost joonisel 2a kujutatud Põdra analüüsi väljundi juures oleva instruktsioonigraafiga. Edaspidi käsitletakse olekut M pinumasina kontekstis ning instruktsiooni M Java baitkoodi kontekstis samaväärsena, järgnevalt on kirjeldatud nendevaheline seos.

Kui $\delta(\mathrm{M}, \xi, \lambda, \sigma) = (\mathrm{M'}, \xi', \lambda', b)$, siis sisuliselt funktsioon δ väljendab, et kui rakendada instruksiooni M pinule ξ ja mälule λ saaksime sisendsümbol σ korral tulemuseks pinu ξ' ning mälu λ'. Vastuseneliku komponent $b \in \{0, 1\}$ väljendab sisendsümboli σ säilimist sisendis. Vastuseneliku olek M' ei ole otseselt instruktsiooni M täitmisega seotud, vaid väljendab, et instruktsiooni M täitmisele järgneb instruktsiooni M' täitmine.

Lisaks on oluline märgata, et näiteks olekud ICONST_2 ja ICONST_2' ei ole samad olekud, küll aga väljendab funktsioon δ sama instruktsiooni ICONST_2 täitmist, olenemata sellest, kas esimene argument on ICONST_2 või ICONST_2'. Sama instruksiooni jaoks mitme oleku loomine on vajalik eelnevalt mainitud järjestusseose jaoks. Selle meetodi puuduseks on, et peame samade instruktsioonide tegevust pinu ja mäluga mitmekordselt kajastama, kuid seeeest pääseme instruktsioonide järjekorra eraldi hoidmisest.

Definitsioonis 1 kirjeldatud „mäluga pinumasin“ on õigusega pinumasin: kõik arvutused toimuvad pinus, mälu on vaid pinuelementide hoidmiseks. Kuigi funktsioon δ kirjeldab vaid enne ja peale instruktsiooni olevat pinu, opereeritakse pinuga sisuliselt siiski vaid tehete $\uparrow$ ning $\downarrow$ abil. Kui funktsiooni δ teine argument on ξ_0 ning tulemusneliku teine element on ξ_f, siis mistahes teiste argumentide korral, on võimalik pinu ξ_0 teisendada tehete $\uparrow$ ning $\downarrow$ mingis kombinatsioonis rakendamisel pinuks ξ_f, kus $\downarrow$ eismene argument on mingi sobivalt valitud pinusümbol.

Näiteks kui funktsiooni δ esimene argument on IADD, siis rakendatakse argumentpinule ξ kaks korda tehet $\uparrow$, väljavõetud elemendid liidetakse ning seejärel rakendatakse liitmise tulemusele ja viimasele tulemuspinule tehet $\downarrow$. Seega tulemusneliku teine argument

$$\xi_4 = (s_1 + s_0) \downarrow \xi_3,$$

kus $(s_1, \xi_3) = \uparrow \xi_2$ ning $(s_0, \xi_2) = \uparrow \xi$, kus ξ on funksiooni δ teine argument.

3. Staatiline analüüs

Programmi staatiline analüüs on programmi valitud omaduste analüüsimine programmi käivitamata [3]. Vastandina, dünaamilist analüüsi tehakse analüüsitava programmi käitlusajal.

3.1 Staatilise analüüsi eelised

Staatilisel analüüsil on dünaamilise ees mitmeid eeliseid [14]:

- Välisest allikast pärit koodi käivitamisel on alati risk, et see teeb midagi ohtlikku. Staatilist analüüsi saab kasutada nende ohtude ehk turvaaukude leidmiseks [9].
- Testida saab vaid lõpliku hulga sisendparameetritega, staatilise analüüsita pole garantiid, et kood on korrektne.

  Soovitakse kontrollida joonisel 1 kujutatud lihtsa Java funktsiooni lõpus olevat tingimust $a = b$ abstrakse sisendparameetri a korral. Tingimuse, mida staatilise analüüsiga võib saada vähese hulga teisendustega täielikult kontrollida, dünaamilise analüüsiga kontrollimine tähendaks selle tingimuse kehtimise kontrollimist mitme konktreetse a väärtuse korral. Olenevalt sisenditüübist võib programmide kõikvõimalike sisenditega testimine olla kulukas või võimatu. Antud näite dünaamiliselt täielikuks kontrollimiseks tuleks seda katsetada 2^{32} parameetri a väärtuse korral, samas kui staatilise analüüsiga piisab järgnevast lihtsustusest:

$$\begin{aligned} a = b &\rightsquigarrow a = a - 2 + 2 \\ &\rightsquigarrow a = a \\ &\rightsquigarrow \top \end{aligned}$$

- Programmi käitumine võib sõltuda käivituskeskkonnast.

  See võib tekitada nii juhuslikke erisusi programmi kasutajate vahel kui ka anda pahatahtlikule programmile võimaluse oma tegevust testkeskkonnas peita.

- Kompilleerimisaegse staatilise analüüsi käigus leitud infot saame kasutada koodi optimeerimiseks.

Staatilisi analüüse tehakse programmi erinevate omaduste kontrollimiseks. Analüüsist saadud kasu sõltub suuresti sellest, mida täpsemalt analüüsime.

3.2 Staatilise analüüsi liigid

Programmikoodis võib üritada analüüsida kõike, näiteks, kas a-tähti on rohkem muutujanimedes või võtmesõnades. Järgnevalt on lühidalt kirjeldatud levinumaid staatilisi analüüse.

3.2.1 Andmevooanalüüs

Andmevooanalüüs (*data-flow analysis*) on staatiline analüüs, kus programmi igal täitmise sammu järel vaadeldakse võimalikke programmiseisundeid, milleks lihtsatel programmidel on programmiseisunditeks kõigi kasutatud muutujate väärtused, kuid seisunditeks võivad olla ka mistahes konkreetsele analüüsile olulised tunnused [12]. Andmevooanalüüsimisel käsitletakse analüüsitavat programmi suunatud graafina, kus programmi igal täitmise sammul liigutakse mööda kaart uude tippu (või mõnikord tsükli puhul juba külastatusse), tippudes toimub võimalike seisundite analüüs; seda graafi nimetatakse **juhtimisvoograafiks** (*control-flow graph*) [12].

3.2.2 Tüübikontroll

Tüübikontroll on staatilise analüüsi meetod, kus enne programmi käivitamist kontrollitakse tüüpide korrektsust. Programmeerimiskeelt nimetatakse **tüübitud keeleks**, kui igale muutujale on antud mittetriviaalne tüüp, vastasel juhul on tegemist **tüüpimata keelga** [2]. Vaid tüübitud keelte puhul on võimalik rääkida staatilisest tüübikontrollist. Selliste programmide korral saab tüübikontrolliga ennetada paljusid käivitusaegseid tüübivigu. Tüübitud keeli, mis läbivaid tüübikontrolli, nimetakse **hästitüübituks**, neid, mis tüübikontrollis põruvad, nimetatakse **halvastitüübituks** [2].

Tüübikontrollijate puhul on oluline, et nad lõpetaks töö mistahes programmi korral ning ei loeks korrektseks ühtki programmi, mis pole tegelikult tüübikorrektne, see aga toob Turingi-täielike programmeerimiskeelte puhul paratamatult kaasa mõningate tüübikorrektsete programmide mittelubamise [10]. Tüübikontrolli läbinud programm ei tähenda veel, et kogu programm oleks korrektne, viimase kindlustamiseks on vaja staatilisest tüübikontrollist tugevamaid vahendeid.

3.2.3 Programmide verifitseerimine

Programmide verifitseerimine on staatilise analüüsi erijuht, kus üritatakse tõestada, et kehtivad valitud programmispetsiifilised omadused, mis moodustavad **programmi spetsifikatsiooni** [1, 8]. Programmide verifitseerija annab garantiisid programmi spetsifikatsioonide kehtimise kohta [4]. Programmide korrektsust saab uurida erinevatel viisidel, järgnevalt kirjeldatakse kahte levinumat võimalust programmide verifitseerimiseks.

Ülalt-alla meetod Selle meetodi puhul hakatakse programmi tööd algusest analüüsima, ning vaadatakse lõpus, mida saab nende eeldustega garanteerida. Selle meetodi rakendamisel võib olla probleemiks, kui on vaja arvestada paljude erinevate eeldustega. Eelnevalt kirjeldatakse, et joonisel 1 toodud programmi analüüsimiseks on vaja sisuliselt läbi vaadata 2^{32} erinevat a väärtust. Selle probleemi vältimiseks kasutatakse ülalt-alla analüüsi puhul intervallanalüüsi, programmi igas punktis vaadatakse iga täisarvulise muutuja kohta, mis on tema minimaalne ning maksimaalne võimalik väärtus [3]. Intervallanalüüs võimaldab suure hulga väärtusi korraga analüüsida, kuid võime kaotada täpsust, kui näiteks x saab olla väiksem kui -2 või suurem kui 2 (sel juhul saame intrevalliks kõik täisarvud, kuigi tegelikult 0, 1 ja -1 pole võimalikud).

Alt-üles meetod Vastandina, alt-üles meetodi puhul otsutatakse, milles soovitakse programmi mingis punktis kindel olla ning uuritakse, mida on selle garanteerimiseks vaja eeldada. Neid eeldusi nimetatakse **nõrgimaiks eeltingimuseks**, kui nad garanteerivad parajasti soovitut, mitte rohkemat. Seda meetodit käsitletakse pikemalt nõrgima eeltingimuse peatükis 4.

3.3 Alt-üles ning ülalt-alla meetodi võrdlus

Üks küllaltki selge alt-üles meetodi eelis on mittehuvitava sisu analüüsimise vältimine, kui lihtsustame tõestamist vajavaid tingimusi piisavalt agressiivselt. Näiteks joonisel 3 kujutatud koodi puhul vajab alt-üles analüüs vaid kolme viimast rida peameetodist, et teada, et funktsiooni lõpus kontrollitav tingimus ei kehti, sest siis oleks see kehtinud juba tingimuslause, mille kehas a omale tingimust $a < 0$ mitterahuldava väärtuse saab, juures. See lihtsustab edasist analüüsi, sest edasi ülespoole minnes vajame vaid, et eksisteeriks tulemus, pole oluline milline.

```
public static int f(int arv) {
    return 2 * arv - 100;
}

public static void main(String[] args) {
    int a = 10;
    int b = a * 15;
    b = 2 * b * b;
    b = f(f(b));

    if (a < 10) {
        a = 100;
    }
    assert(a < 10);
}
```

Joonis 3. Osaliselt analüüsimiseks mitteolulise sisuga Java kood.

Vähem selgena kehtib sama eelis ka ilma tingimuste lihtsustamiseta. Kuigi lihtsustamisega pääsetakse joonisel 3 toodud koodi analüüsides kolme reaga, saab ka ilma lihtsustamiseta vältida b kohta käivaid ridu, sest need ei mõjuta a kohta käiva tingimuse kehtimist. Selle eelise väljatoomine on oluline, sest lihtsustamisel on oma hind. Seetõttu on võimalik, et on vaja otsustada, kas ja millistes vahesammudes lihtsustamisega tegeleda.

Üleliigse info analüüsi vältimise tõttu võimaldab alt-üles meetod kitsamat anaüüsi kui ülalt-alla meetod, sest saab kontrollida täpselt parasjagu huvitavate tingimuste kehtimist. Vastandina, ülalt-alla analüüsi puhul peab arvestama kõikvõimalike sisenditaga, kuid boonusena saab mõnikord allajõudes rohkem infot. Seega tuleb valida olukorrale vastav meetod analüüsimiseks, kui võimalikke sisendeid on vähe, kuid on vaja kontrollida paljude tingimuste kehtimist, on ülalt-alla meetod mõistlikum. Kui huvi pakub vaid üks tingimus, kuid võimalikke sisendeid on palju, tasub valida alt-üles meetod.

Üks alt-üles meetodi puudus on keerukus võrreldes ülalt-alla meetodiga. Ülalt alla analüüsides saab lihtsalt liikuda järjest mööda programmiinstruktsioone ning nende tegevust modelleerida, alt üles meetodil aga on vaja iga instruktsioon nii-öelda ümber pöörata, et saada vastavale instruktsioonile eelnev seis. Tavaliste instruktsioonide puhul on see lihtsasti ületatav probleem, kuid näiteks tsüklitega on keerulisem.

```
public static void main(String[] args) {

    int a = sala(3);

    while (a < 10) {
        a = suurenda(a);
    }
    assert(a != 10);
}
```

Joonis 4. Tsükliga Java programm.

Joonisel 4 kujutatud programmi ülalt alla analüüsimisel saame kindlalt ilma tsükli sisu vaatamata öelda, et pärast tsükli läbimist tingimus $a < 10$ enam ei kehti, sest muidu ei oleks tsüklist väljutud. Vastupidiselt, alt üles analüüsides ei saa kuigi palju öelda muutuja a väärtuse kohta enne tsüklit. Näiteks, kui soovitakse kontrollida, kas tsükli järel kehtib tingimus $a \neq 10$, siis on ainult teada, et kui $a > 10$, siis tsüklisse üldse ei minda ja kontrollitav tingimus $a \neq 10$ kehtib. Nende juhtude, kus tsükli keha täitma minnakse, analüüsimine on keerukam.

4. Nõrgima eeltingimuse leidmine

Programme alt üles meetodil uurides valitakse, milline tingimus või tingimused peavad programmi punktis x kindlasti kehtima ning otsitakse, mis on vähim, mida selle tingimuse kehtimiseks on vaja programmi punktis x', kusjuures programmi peab käivitamisel jõudma punkti x' enne kui punkti x.

4.1 Eeltingimuse tuletamine

Enne kui alapeatükis 4.2 vaadeldakse eeltingimusi pinukeeltel, käsitletakse siinkohal eeltingimuste tuletamist lihtsatel kõrgema taseme programmeerimiskeelte lausetel. Kui on teada, et lause B täitmise järel peab kehtima tingimus ψ, siis eeltingimuseks ψ' nimetatakse sellist tingimust, mille kehtimisel vahetult enne lauset B, peab lause B täitmise järel alati kehtima tingimus ψ.

Eeltingimuste moodustamisel kasutatakse tavalisi loogilisi sümboleid $\neg$ (eitus), $\wedge$ (konjunktsioon ehk loogiline „ja“), $\vee$ (disjunktsioon ehk loogiline „või“), $\Rightarrow$ (implikatsioon) ning $\Longleftrightarrow$ (ekvivalents). Lisaks on vaja tingimusi kujul $\alpha[Q/R]$, mis tähistab tingimuse α struktuuriga tingimust erisusega, et kasutame avaldist Q kohtades, kus tingimuses α on avaldis R.

Eeltingimusi saab leida ka mitmele lausele korraga. Moodustagu laused C_1, C_2, C_3, …, C_n, kus n on mistahes naturaalarv, lausetejada $C = \{C_1, C_2, C_3, \ldots, C_n\}$. Soovitakse leida eeltingimust ψ', mille kehtimisel vahetult enne lausetejada C kehtiks lausetejada C täitmise järel kindlasti teadaolev tingimus ψ_n. Selleks tuleb leida tingimuse ψ_n kehtimist garanteeriv eeltingimus ψ_{n-1} lausele C_n. Edasi tuleb leida tingimust ψ_{n-1} garanteeriv eeltingimus lausele C_{n-1}. Selliselt jätkates saab lõpuks leida lausele C_1 eeltingimuse ψ_0, mis on ka kogu lausetejada eeltingimus ψ'. Lausetejada käsitletakse edaspidi ühe võimaliku lausena, lausetejada eeltingimust oskame leida eelkirjeldatud meetodil. Arvestades, et programmid on lausetejadad, on vaja programmi nõrgima eeltingimuse leidmiseks osata vaid üksikute lausete nõrgimaid eeltingimusi leida.

Klassikaline loogika ütleb, et kui tingimusest α saab järeldada tingimuse γ, siis ka tingimusest $\alpha \wedge \beta$ saab järeldada tingimuse γ kehtimise. Selleks, et mitte lisada eeltingimusse ebaolulist infot, räägitakse nõrgimast eeltingimusest. Deterministlike keelte (ehk keelte, mille puhul sisendit teades on alati vaid üks võimalik programmiläbimistee) puhul on

nõrgimaks eeltingimuseks selline eeltingimus ψ', mille kehtimine on garanteeritud, kui teame, et kehtib järeltingimus ψ [15]. Tuginedes Huth jt [6] raamatule on järgnevalt analüüsitud erinevate lausete puhul, milline eeltingimus peab kehtima, et selle lause järel kehtiks tingimus ψ.

4.1.1 Omistuslaused

Olgu lause B kujul $x := A$ tähenduses, et muutujale x omistatakse avaldise A väärtus. Teame, et selle omistamise toimumise järel on muutuja x väärtus avaldise A väärtus. Seega kui tingimuses ψ öeldakse midagi muutuja x väärtuse kohta, peab see enne kehtima avaldise A kohta. Samas muutuja x kohta vahetult enne avaldist B nõudmisi pole, sest muutujale x antakse avaldises B uus väärtus. Järelikult sobib nõrgimaks eeltingimuseks $\psi[A/x]$, mida on kujutatud ka joonisel 5a.

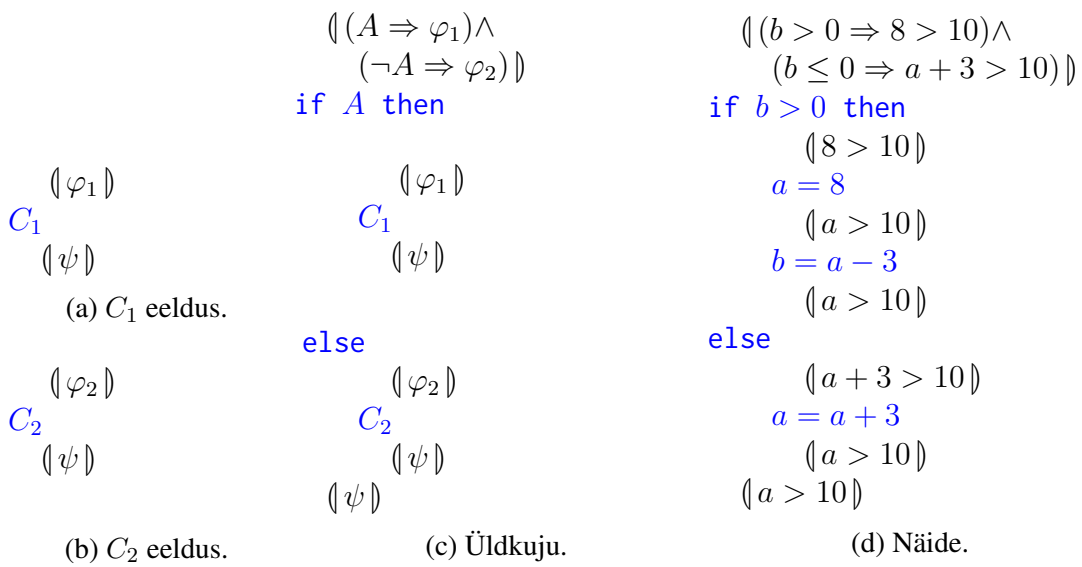
$\{\!\{ \psi[A/x] \}\!\}$	$\{\!\{ y > 0 \}\!\}$	$\{\!\{ y - 9 > 0 \}\!\} \equiv \{\!\{ y > 9 \}\!\}$
$x := A$	$x := y$	$x = y - 9$
$\{\!\{ \psi \}\!\}$	$\{\!\{ x > 0 \}\!\}$	$\{\!\{ x > 0 \}\!\}$
(a) Üldkuju.	(b) Lihtne näide.	(c) Lihtsustamisega näide.

Joonis 5. Omistamise eeltingimus.

Lihtsatel näidete puhul (vt joonis 5b) piisab lihtsalt asendamisest, kuid joonisel 5c omistatakse muutujale x keerulisem avaldis, sel juhul on võimalik ka lihtsustada. Edasises ei tooda eraldi välja võimalikke lihtsustusi, leitakse lihtsalt mingi korrektse nõrgim eeltingimus.

4.1.2 Tingimuslaused

Tingimuslausetele nõrgima eeltingimuse leidmisel tuleb arvestada, et programmi täitmisel läbitakse olenevalt tingimuslause tingimusavaldise kehtimisest erinevaid lauseosasid. Tingimuslaused on kujul `if` A `then` C_1 `else` C_2, kus A on avaldis ning C_1 ja C_2 on laused. Soovitakse leida nõrgimat eeltingimust, mille korral selle tingimuslause järel kehtiks tingimus ψ. Joonistel 6a ja 6b on kujutatud eeldused, et lausete C_1 ja C_2 nõrgimad eeltingimused, mis garanteerivad vastavalt C_1 või C_2 täitmise järel tingimuse ψ kehtimise, on vastavalt φ_1 ja φ_2.



(a) C_1 eeldus.

(b) C_2 eeldus.

(c) Üldkuju.

(d) Näide.

Joonis 6. Tingimuslause nõrgim eeltingimus.

Vastavalt sellele, kas avaldis A on tõene või mitte, on vaja tingimuslause eeldusse lisada eeldusi φ_1 ja φ_2, vt. joonis 6c. Terveks tingimuslause joonisel 6 kujutatud nõrgimaks eeltingimuseks on

$$\frac{\langle\!|\,\varphi_1\,|\!\rangle\ C_1\ \langle\!|\,\psi\,|\!\rangle \qquad \langle\!|\,\varphi_2\,|\!\rangle\ C_2\ \langle\!|\,\psi\,|\!\rangle}{\langle\!|\,(A \Rightarrow \varphi_1) \wedge (\neg A \Rightarrow \varphi_2)\,|\!\rangle\ \texttt{if}\ A\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \langle\!|\,\psi\,|\!\rangle},$$

kus joone all on lause ja selle nõrgim eeltingimus, kui on vaja järeltingimuse ψ kehtimist, ning joone peal on selle eeltingimuse kehtimiseks vajalikud eeldused. Joonisel 6d kujutatud näites on eeldused täidetud, sest tingimuslause harudes olevate lausete nõrgimaid eeltingimusi saab lihtsalt leida.

Viimane tingimuslause nõrgima eeltingimuse kuju on toodud eelmainitud Huth jt [6] raamatu järgi. Käesoleva töö praktilises osas kasutatakse nõrgima eeltingimusena siintoodud eeltingimusega $(A \Rightarrow \varphi_1) \wedge (\neg A \Rightarrow \varphi_2)$ samaväärset tingimust $(A \wedge \varphi_1) \vee (\neg A \wedge \varphi_2)$. Samaväärsus kehtib, sest tingimustest A ja $\neg A$ peab alati üks kehtima. Viimast tingimust kasutatakse, sest see ühildub paremini kasutatava staatilise analüüsi raamistikuga Põder.

4.1.3 Tsüklilause

Olgu tsüklilause kujul $\texttt{while}\ (A)\ C$. Sarnaselt tingimuslause analüüsile saab teha eelduse, et $(\!|\,\varphi\,|\!)\ C\ (\!|\,\psi\,|\!)$. Kui avaldis A on väär, siis selleks, et tsükli järel kehtiks tingimus ψ, on vaja, et see kehtiks ka enne tsüklilauset, sest tsükli kehasse ei minda ning mingeid muutusi ei toimu. Kui avaldis A on tõene, siis tsükli viimase läbimise alguses kehtib eelduse järgi nõrgim eeltingimus φ. Olenevalt sellest, kas ka siis avaldis A on tõene, läbitakse tsüklit uuesti või mitte. Terveks nõrgimaks eeltingimuseks tuleb seega

$$\frac{(\!|\,\varphi\,|\!)\ C\ (\!|\,\psi\,|\!) \qquad (\!|\,\xi\,|\!)\ \texttt{while}\ (A)\ C\ (\!|\,\varphi\,|\!)}{(\!|\,(\neg A \Rightarrow \psi) \wedge (A \Rightarrow \xi)\,|\!)\ \texttt{while}\ (A)\ C\ (\!|\,\psi\,|\!)}.$$

Selline eeltingimus pole kuigi hea, sest eeldused pole lihtsamad kui esialgne küsimus. Sellist meetodit kasutades jääb analüüs alt-üles meetodil enamasti lõpmatusse tsüklisse, sest tsüklitingimus kontrollimise hetkel pole alt poolt tulles eeldusi, mis ütleks, kas tsüklitingimus A kehtib või ei. Selle probleemi vältimiseks kasutatakse **tsükliinvariante**. Tsükliinvariant on tsükli kohta tehtav oletus, mis kujutab endas tingimust, mis peab kehtima enne tsüklit ning iga tsüklikeha läbimise järel (järelikult ka terve tsüklilause lõpus).

$(\!|\,\mu\,|\!)$
$\texttt{while}\ (A)\ \{$
  $(\!|\,A \wedge \mu\,|\!)$
  $(\!|\,\varphi\,|\!)$
  C
  $(\!|\,\mu\,|\!)$
$\}$
$(\!|\,\neg A \wedge \mu\,|\!)$
$(\!|\,\psi\,|\!)$

(a) Üldjuht.

$(\!|\,x \leq 10\,|\!)$
$\texttt{while}\ (x < 10)\ \{$
  $(\!|\,x < 10 \wedge x \leq 10\,|\!)$
  $(\!|\,x + 1 \leq 10\,|\!)$
  $x := x + 1$
  $(\!|\,x \leq 10\,|\!)$
$\}$
$(\!|\,x \geq 10 \wedge x \leq 10\,|\!)$
$(\!|\,x = 10\,|\!)$

(b) Näide.

Joonis 7. Tsüklilause eeltingimus teadaoleva invariandiga.

Oletused, mis võiks olla tsükli invariant, tehakse vaadeldavast nõrgima eeltingimuse leidmise analüüsist väljaspool. Seega olgu leitud üks sobiv analüüsitava tsükli invariant μ. Selleks, et see invariant sobiks tsüklilause eeltingimuseks järeltingimuse ψ kehtimiseks, peab kehtima $\neg A \wedge \mu \Rightarrow \psi$ ning kui kasutada eelnevalt mainitud eeldust $(\!|\,\varphi\,|\!)\ C\ (\!|\,\psi\,|\!)$, siis peab ka kehtima, $A \wedge \mu \Rightarrow \varphi$. Eeltingimust on kirjeldatud joonisel 7.

Leitud eeltingimus μ on nõrgim, kui kehtivad nii samaväärsus $(\neg A \wedge \mu) \equiv \psi$ kui ka samaväärsus $(A \wedge \mu) \equiv \varphi$ (joonisel 7a rohelisega read). Kokkuvõttes, tsüklilause nõrgim eeltingimus järeltingimuse ψ kehtimiseks avaldub kujul:

$$\frac{(\!|\, A \wedge \mu \,|\!)\ C\ (\!|\, \mu \,|\!) \qquad \neg(A \wedge \mu) \equiv \psi}{(\!|\, \mu \,|\!)\ \texttt{while}\ (A)\ C\ (\!|\, \neg A \wedge \mu \,|\!)}.$$

Joonisel 7b toodud näites kehtivad mõlemad samaväärsused ning $x \leq 10$ on nõrgim eeltingimus, mis kindlustab, et peale tsüklit on muutuja x väärtus 10.

4.2 Nõrgim eeltingimus mäluga pinukeeltele

Loeckx jt [13] defineerivad nõrgima eeltingimuse vooskeemi (*flowchart*) keeletele, sellele tuginedes defineeritakse käesolevas töös nõrgim eeltingimus definitsioonis 1 kirjeldatud mäluga pinukeeltele:

Definition 2. Nõrgimaks eeltingimuseks mäluga pinumasina $P = (Q, \Sigma, S, \delta, \xi_0, q_0, \lambda_0, Q_F)$ ja järeltingimuse $\varphi : (\Xi \times (\mathbb{N} \to S) \to \texttt{Bool})$ korral nimetatakse predikaati $\psi : (\Xi \times (\mathbb{N} \to S) \to \texttt{Bool})$, mille korral kehtib

$$\psi(\xi_0, \lambda_0) \iff \varphi(\xi_f, \lambda_f),$$

kus ξ_f ja λ_f on vastavalt pinu ja mälu olekud hetkel kui P (algpinuga ξ_0 ja algmäluga λ_0) jõuab (sisendist olenevalt) mingisse lõppolekusse, kusjuures nõrgimaks eeltingimuseks sobivad vaid sellised predikaadid ψ, mille korral mäluga pinumasin P jõuab mistahes sisendiga mingisse lõppolekusse.

Eelnevalt toodud näidetes (joonistel 5b, 5c, 6d ja 7b) on tingimustes kasutatud muutujanimesid. Mäluga pinumasina puhul peab arvestama, et tingimustes on nende asemel pinuasukohad ning mäluasukohad. Pinussümbolid on tähistatud s_0, s_1, s_2, . . . , alustades pinu pealmisest elemendist. Näiteks, kui tingimus on kujul $s_1 = 8$, siis see tähendab, et pinu pealtpoolt teine element peab olema sellel hetkel 8.

4.2.1 Lihtsad muutused pinus

Definitsioonis 1 kirjeldatud mäluga pinumasina üleminekufunktsioon δ võib kirjeldada erinevaid lihtsaid pinumodifikatsioone. Java baitkoodi instruktsioon ISUB võtab pinust

kaks pealmist elementi ning lisab pinusse nende vahe, selle instruktsiooni kirjeldus pinumasina definitsioonis oleks

$$\delta(\text{ISUB}, (s_0, (s_1, \xi))), \lambda, \sigma) = (\text{M}, (s_1 - s_0, \xi), \lambda, 0),$$

kus M on Java baitkoodi instruktsioon, mis järgneb instruktsioonile ISUB. Kui on vaja, et instruktsiooni ISUB järel kehtiks tingimus $s_0 = 8$, kus s_0 tähistab pinu pealmist elementi. Enne instruktsiooni ISUB toimumist pidi seega pinu kahe pealmise elemendi vahe olema 8. Teostatud nõrgima eeltingimuse leidja teisendab instruktsiooni ISUB alt üles läbides reegli $s_0 = 8$ reegliks $s_1 - s_0 = 8$. Sarnaselt toimitakse ka teiste lihtsate Java baitkoodi instruktsioonidega.

4.2.2 Mälu mõjutavad üleminekud

Mälu kirjeldab definitsioonis 1 funktsioon λ. Mälu mõjutavad Java baitkoodi instruktsioonid ILOAD(n) ning ISTORE(n), kus n on täisarv, mis näitab millisel indeksil mälu muudetakse. Mõjutatavat mäluvälja saab kirjeldada nimetusega l_n.

Instruktsioon ISTORE(3) võtab pinu pealmise elemendi ning paigutab selle mällu kohale 3. Vastav kirjeldus pinumasina keeles oleks

$$\delta(\text{ISTORE(3)}, (s_0, \xi)), \lambda, \sigma) = (\text{M}, \xi, \lambda_1, 0),$$

kus M on järgmine instruktsioon ning $\lambda_1(a) = \begin{cases} s_0, & \text{kui } a = 3, \\ \lambda(a), & \text{muidu.} \end{cases}$

Järelikult, element, mis enne instruktsiooni ISTORE(3) oli pinu pealmine, on peale seda instruktsiooni mälus kohal 3, kusjuures pole teada, mis oli mälus kohal 3 enne. Seega, kui peale seda instruktsiooni peab kehtima tingimus ψ, siis enne instruktsiooni oleva tingimuse leidmiseks tuleb kõik mäluindeksi l_3 esinemised asendada pinuindeksiga s_0, tingimuseks saaks $\psi[s_0/l_3]$.

Vastupidiselt instruktsioonile ISTORE(n) lisab instruktsioon ILOAD(n) pinu peale elemendi, mis on mälus kohal n. See instruktsioon mälu ei muuda. Nõrgima eeltingimuse leidmiseks tuleb asendada kõik pinuindeksi s_0 esinemised asendada mäluindeksiga l_0.

4.2.3 Pinuindeksite nihe

Eelnevalt alapeatükkides 4.2.1 ja 4.2.2 vaadeldi vaid otseselt instruktsioonist mõjutatud muutusi, mis kirjeldab pinu pealmist elementi. Kui aga on reegleid ka pinu teiste elementide kohta, siis peab arvestama sellega, et instruktsioon võib mõjutada (juhul, kui elementide arv pinus muutub) indekseid pinus. Pinu saab muuta vaid tehetega $\uparrow$ ning $\downarrow$, seega pinu alumiste elementide kohta käivaid reegleid saab parandada vaid pinuindeksite nihutamisega, mis kujutab endast kõigi pinuindeksite sama palju suurendamist või vähendamist.

Selleke, et instruktsiooni M järel kehtiks reegel α, peab enne M täitmist kehtima β, mis on sama reegliga α, kus on muudetud otseselt instruktsioonist mõjutatud reegliosad ning kus sügavamate pinusümbolite indekseid suurendatud ühe võrra iga instrukstioonis M oleva tehte $\downarrow$ kohta ning vähendatud ühe võrra iga seal oleva tehte $\uparrow$ kohta.

Eelmises alapeatükis 4.2.2 toodud näide, kuidas instruktsioon ISTORE(3) mõjutab reegleid, kehtib mainitud kujul vaid siis, kui reeglites pole ühtegi tingimust pinusümbolite kohta. Kuna ISTORE(3) võttis pinust ühe elemendi ära, siis kõigi reeglites mainitud pinusümbolite indekseid tuleb ühe võrra suurendada, et teha pinu peale ruumi pinusümbolile, mille ISTORE(3) pinust eemaldas, kuid mis seal enne seda instruktsiooni oli. Seega, nõrgimaks eeltingimuseks sobinuks tegelikult $(\texttt{suurenda\_pinuindekseid}(\psi))\,[s_0/l_3]$.

4.2.4 Funktsioonikutsed

Java funktsioonid võtavad oma argumendid pinu pealmistest elementidest, kuid funktsioonile tunduvad need kui mälumuutujad. Funktsioon ei näe välise keskkonna mälu ega pinu. Juhul kui funktsioon tagastab midagi, paneb ta selle lõpuks pinu peale, ja see element saab ka välise pinu pealmiseks elemendiks.

Vältimaks sama funktsiooni mitmekordset analüüsi (juhul, kui sama funktsiooni kutsutakse välja korduvalt), on mõistlik analüüsida funktsioone kontekstist sõltumatult. Funktsiooni sisse vaadatakse ainult programmiinstruktsioonides tagant ette liikudes esimest korda selle funktsioonini jõudes. Funktsiooni sisenemiseks on samuti vaja arvutada nõrgim eeltingimus ehk tingimus, mis peab kehtima kutsutud meetodi lõpuks, et kehtiks funktsiooni järel olev tingimus.

Java funktsioonid võivad olla tagastustüübita (`void`) või tagastustüübiga, selles töös käsitletakse vaid täisarve, seega olgu tagatustüübiga funktsioonid täisarvulise tagastustüübiga. Kui analüüs jõuab programmi lõpu poolt tulles esimest korda mingi funktsioonikutseni, siis tagastustüübiga funktsioonile antakse vaid reegel $\texttt{Ret} = s_0$. Välist programmi saavad mõjutada vaid tagastatav väärtus ning funktsioonis globaalsete muutujate muutmine. Käesolevas töös arvestatakse, et globaalsed muutujad pole lubatud. Tagastustüübiga funktsiooni sisenemisel on seega nõrgim eeltingimus, et pinu peal on tagastatav väärtus. Tagastustüübita funktsioonile antakse `true`, sest globaalsete muutujateta süsteemis on selliste funktsioonide puhul oluline vaid termineerumine.

Funktsiooni sees toimub analüüs tavapärasel moel, sammhaaval alt üles. Olenemata sellest, kas funktsiooni läbiti parasjagu selleni jõudes või olid funktsioonireeglid juba olemas, on peale funktsioonikutse alt üles läbimist vaja ühendada välise keskkonna ja väljakutsutud funktsiooni reeglid. Tagastustüübita funktsioon välise keskkonna reegleid ei mõjuta, sel juhul jäävad kehtima välised reeglid. Olgu täisarvulise tagastusüübiga funktsiooni reeglid φ ning välised reegilid ψ. Selleks, et saada sama tulemus kui iga kord funktsioonikutsel funktsiooni läbides saanuks, tuleb ühendamisel tekitada reegel $\varphi[\psi[a/s_0]/\texttt{Ret} = a]$ ehk reegel, mille struktuur on funktsioonireegli oma, kuid kõik avaldised kujul $\texttt{Ret} = a$ on asendatud reegli ψ struktuuriga, kus s_0 asemele on pandud avaldis a. Lisaks mainitule tuleb enne ühendamist arvestada ka pinuindeksite nihkega ning sellega, et välise programmi jaoks on funktsioonisisesed mälumuutujad pinus.

5. Teostusdetailid

Käesoleva töö praktiline osa on teostatud staatilise analüüsi keskkonnas Põder [11]. Põder on implementeeritud programmeerimiskeeles Scala, mis on tugevalt tüübitud funktsionaaset paradigma toetav objektorienteeritud Javaga ühilduv programmeerimiskeel. Põder kasutab oma analüüside jaoks andmevooanalüüsi, mida erinevaid programmiseisundeid uurides saab kasutada erinevate analüüside jaoks.

Põdras on implementeeritud mitmeid erinevaid analüüse, näiteks ülalpool mainitud intervallanalüüs. Käesoleva töö raames tehakse nõrgima eeltingimuse leidmise analüüsi alt-üles meetodil. Teostus on implementeeritud failis `WP.scala`. Nõrgima eeltingimuse leidmise analüüs ühildub teiste raamistikus Põder olevate analüüsidega, kuigi selle analüüsi spetsiifilisuse tõttu tuli teha raamistikus mitmeid kohandusi.

5.1 Implementatsioonikoodi näited

Analüsaatori koostamise põhiliseks tegevuseks on erinevate instruktsioonidega käitumise implementeerimine. Joonsel 8 on toodud ülalpool mainitud lihtinstruktsioonide implementatsioonid.

```
case xLOAD(_, i) => d : dl => {
  vS(Logic.repl(Map[Val, dl](SVar(0) -> app(LVar(i.x.get))))(d))
}
case xSTORE(_, i) => d => {
  Logic.repl(Map[Val, dl](LVar(i.x.get) -> app(SVar(0))))(sS(d))
}
case i: ICONST => d => {
  vS(Logic.repl(Map[Val, dl](SVar(0) -> app(IVal(i.n))))(d))
}
case xADD(x) => d => {
  Logic.repl(Map[Val, dl](SVar(1) -> app(Ident("+"), SVar(1),
      SVar(0))))(sS(d))
}
case xSUB(x) => d => {
  Logic.repl(Map[Val, dl](SVar(1) -> app(Ident("-"), SVar(1),
      SVar(0))))(sS(d))
}
```

Joonis 8. Mõningate lihtinstruktsioonide implementatsioon.

Nõrgima eeltingimuse leidmiseks on ette antud instruktsioon ning järeltingimus, mille kehtimist on vaja garanteerida. Esimesel toodud juhul `case xLOAD(_, i) => d : dl` on instruktsiooniks ILOAD(i) ning järeltingimuseks d, mis on tüüpi `dl`. Vastavalt peatükkides 4.2.2 ja 4.2.3 kirjeldatule asentakse nõrgima eeltingimuse saamiseks tingimuses d kõik pinuindeksid `SVar(0)` ehk s_0 mäluindeksitega `LVar(i.x.get)` ehk l_i ning seejärel vähendatakse (kasutades abifunktsiooni `vS`), kõiki ülejäänusid pinuindekseid ühe võrra, et täita selle elemendi koht, mis tuleb alles instruktsiooniga ILOAD(i). Vastupidiselt, juhul `case xSTORE(_, i) => d` ehk instruktsiooniks ISTORE(i) ning järeltingimuse d korral, suurendatakse kõigepealt abifunktsiooni `sS` abil kõiki pinuindekseid ühe võrra ning siis täidetakse tühjaks jäänud koht asendusega $l_i \longrightarrow s_0$, mis kirjeldab, et enne instruktsiooni ISTORE(i) oli hiljem mälus kohal l_i olev element pinu pealmine.

Instruktsiooni ICONST(n) väljendab juhtum `case i: ICONST => d`. Kui pinu peale lisatakse arv n, siis kõik, mida nõutakse pinu pealmise elemendi kohta, peab enne seda instruktsiooni kehtima arvu n kohta. Enne seda instruktsiooni olid teised pinus olevad elemendid kõrgemal, sest lisatavat konstanti veel ei olnud. Seega nõrgima eeltingimuse saamiseks asendatakse reeglis d kõik pinuindeksid s_0 arvuga n ning seejärel vähendatakse kõiki teisi pinuindekseid.

Juhtudel `case xADD(x) => d` ja `case xSUB(x) => d` võtavad instruktsioonid IADD ning ISUB kaks pinu pealmist elementi ja lisavad pinusse vastavalt summa või vahe. Eelnevalt oli seega pinus üks element rohkem, sellele elemendile ruumi tekitamiseks tuleb reeglites kõiki pinuindekseid suurendada ühe võrra, ning seejärel pärast instruktsiooni oleva pinu pealmise elemendi kohta käivaid reegleid muuta. Kuna kõigi pinuindeksite suurendamine tegi ka indeksist s_0 indeksi s_1, on vaja asendust $s_1 \longrightarrow s_1 \pm s_0$.

Kõik joonisel 8 kujutatud instruktsioonide implementatsioonid kasutavad pinuindeksite nihke abifunktsiooni `vS` või `sS`, mis vastavalt vähendavad või suurendavad kõiki reeglites esinevaid pinuindekseid.

Joonisel 9 kujutatud funktsioon `sS` võtab argumendiks reegli, ning tagastab esialgsega sarnase reegli, kus on kõiki pinuindekseid ühe võrra suurendatud. Ehk tehakse asendused `SVar(0)` $\longrightarrow$ `SVar(1)`, `SVar(1)` $\longrightarrow$ `SVar(2)` jne. Selleks kasutatakse klassi `Logic` meetodit `getAtoms`, mis teeb reeglist algosad. Neid algosasid läbi mustrisobitusega läbi vaadates leitakse kõik pinusümboleid tähistavad `SVar` tüüpi algosad. Iga leitud `SVar(n)`

kohta tehakse asendusreegel `SVar(n)` $\longrightarrow$ `SVar(n + 1)`. Saadud reeglite järgi teeb klassi `Logic` meetod `repl` vajalikud asendused.

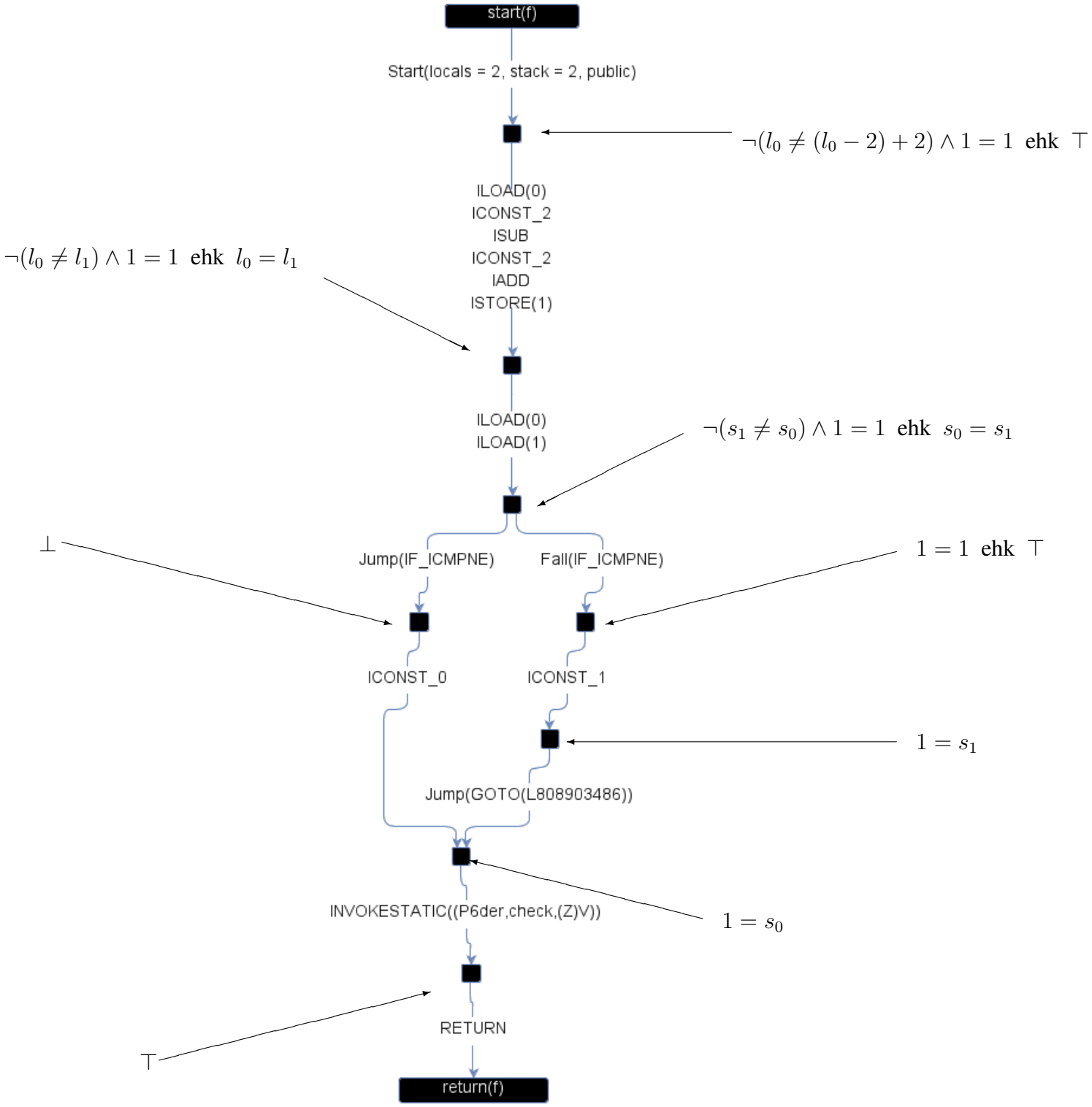
```
def sS(a: dl): dl = {

  def ss(s: Seq[Val]): List[(Val, dl)] = s match {
    case Seq() => List()
    case SVar(n) +: xs => (SVar(n), app(SVar(n + 1))) :: ss(xs)
    case x +: xs => ss(xs)
  }

  Logic.repl(ss(Logic.getAtoms(a)).toMap)(a)
}
```

Joonis 9. Pinuindeksite suurendamise funktsioon.

Klass `Logic` on raamistiku Põder klass, mida antud töö raames koostatud analüüs kõige rohkem otseselt kasutab. Kasutatakse nii juba mainitud meetodeid `getAtoms` ning `repl` kui ka reeglite ühendamiseks sobivaid loogikameetodeid `and`, `or` ja `not`.

Joonisel 1 kujutatud programm saab analüüsimisel väljundi, mis on kujutatud joonisel 10. Analüüsi käivitamisel läbitakse alt üles programmi instruktsioone ning muudab kehtima pidavaid tingimusi vastavalt läbitavatele instruktsioonidele. Teostatav implementatsioon ei tegele saadud avaldiste lihtsustamisega vahesammudes. Kui üles jõudes on tingimus samaselt tõene, siis kontrollitav avaldis kehtib, kui ei ole samaselt tõene, siis pole kontrollitava analüüsi kehtivus garanteeritud.

Enne kontrolli algust on eeltingimus samaselt tõene, sest pole vaja midagi kontrollida. Põdra analüüsi välja kutsudes tekib tingimus, et pinu peal peab olema 1. Seda väärtust ei saa joonisel vasakust harust tulla, seega seal on tingimus samaselt väär. Hargnemise ühendamisel tekib tingimus, et peab minema parempoolsesse harusse ja seal olev tingimus peab kehtima. Kui mõlemad harud oleks võimalikud, tekiks tingimus kujul $(A \wedge \varphi_1) \vee (\neg A \wedge \varphi_2)$, kus A on hargnemistingimus ning φ_1 ja φ_2 on kummagi haru tingimused. Funktsiooni alguses on nõrgim eeltingimus samaselt tõene, seega joonisel 1 kujutatud funktsioon `f` on korrektne.



Joonis 10. Joonisel 1 [3] kujutatud programmi analüüs.

[3] Põdraga analüüsides kastuatakse tingimuse c kehtimise kontrollimiseks `assert(c)` asemel `P6der.check(c)`

5.2 Tsüklitega programmide analüüs

Joonisel 11 on kujutatud tsükliga programmi, mida soovime analüüsida.

```
public class TestWhile {
    public static void main(String[] args) {
        int x = 5;
        P6der.invariant("(<= x 10)");
        while (x < 10) {
            x++;
        }
        P6der.check(x == 10);
    }
}
```

Joonis 11. Anaüüsitav tsükliga Java programm.

Tsükliga programmi puhul annab kasutaja lisaks kontrollitavale tingimusele kaasa ka tsükliinvariandi. Tsüklilauset analüüsib Põdra raamistikku lisatud nõrgima eeltingimuse leidja sarnaselt peatükis 4.1.3 kirjeldatud skeemile. Analüsaator arvestab kasutaja pakutud invariandi kehtimisega, kuid väljastab hoiatuse, kui see invariant ei sobi või kui ta ei suuda nõutud samaväärsusi tõestada. Kuid mistahes juhul antakse see invariant edasi järgmisele instruktsioonile.

6. Kokkuvõte

Staatiline analüüs on üks viis programmide uurimiseks. Vastandina dünaaamilisele analüüsile ei pea staatilise analüüsi jaoks analüüsitavat koodi käivitama.

Töö väljundina teostatakse staatilise analüüsi raamistikus Põder nõrgima eeltingimuse alt üles leidmise analüsaator. Töö autori panus on leitav allikast [11] muudatusteajaloost autorinimelistest kehtestustest (*commits*). Muudatusi tehti failis `WP.scala`, kus töö autor sobitas Põdra raamistikku erinevate Java baitkoodi instruktsioonide alt-üles analüüsi.

Käesolevas töös tuuakse välja et, staatilise analüüsi peamine eelis dünaamilise ees on ohutum anaüüs ning alt-üles lähenemise eeliseks traditsiooniliste ülalt-alla meetodi ees on programmi semantika ebaoluliste osade analüüsi vältimine. Lisaks tuuakse välja, kuidas erinevatel juhtudel nõrgimat eeltingimust leida ning selgitatakse, et tsüklite alt-üles anaüüsimisel tuleb termineeruva analüüsi saamiseks leida tsüklitele invariant ehk tingimus, mis peab kehtima enne tsüklit ning iga tsükliläbimiskorra järel. Käesolevas analüüsis saadakse see invariant väljaspoolt: analüsaatori kasutaja lisab selle tsüklile.

Töö praktilisest osast on puudu instruktsioonide vahesammudes avaldiste lihtsustamine. Ühe võimaliku edasiarendusena on võimalik otsustada, kas vahepealne lihtsustamine on mõistlik ning see vajadusel teostada. Lisaks saab praegune analüsaator hakkama vaid lihtsamate Java programmidega. Teise võimaliku edasiarendusena saaks nõrgima eeltingimuse analüsaatorit täiendada nii, et ta aksepteeriks rohkemaid erinevaid programme.

Viidatud kirjandus

[1] Mike Barnett ja K Rustan M Leino. „Weakest-Precondition of Unstructured Programs“ (2005).

[2] Luca Cardelli. „Type Systems“ (1996).

[3] Patrick Cousot ja Radhia Cousot. „Static determination of dynamic properties of programs“. Teoses: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod. 1976.

[4] Cormac Flanagan *et al.* „Extended static checking for Java“. *ACM Sigplan Notices* 37.5 (2002), lk. 234–245.

[5] Sheila Greibach. „Checking automata and one-way stack languages“. *Journal of Computer and System Sciences* 3.2 (1969), lk. 196–217.

[6] Michael Huth ja Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[7] Philip Koopman. „Stack computers: the new wave“ (1989).

[8] Barbara Liskov, John Guttag *et al. Abstraction and specification in program development*. Köide 180. MIT press Cambridge, 1986.

[9] Benjamin Livshits ja Monica S Lam. „Finding Security Vulnerabilities in Java Applications with Static Analysis“ (2005).

[10] Benjamin C Pierce ja C Benjamin. *Types and programming languages*. MIT press, 2002.

[11] *Põder. Lähtekood*. URL: `https://bitbucket.org/kalmera/poder/src/master/`.

[12] Simmo Saan. „Abstraktsete domeenide omaduspõhine testimine“. Bakalaureusetöö. Tartu Ülikool, 2016.

[13] Kurt Sieber. *The foundations of program verification*. Springer-Verlag, 2013.

[14] Jiang Zheng *et al.* „On the value of static analysis for fault detection in software“. *IEEE transactions on software engineering* 32.4 (2006), lk. 240–253.

[15] D Wood. „A note on top-down deterministic languages“. *BIT Numerical Mathematics* 9.4 (1969), lk. 387–399.

Lisad

I. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Mirjam Iher**,

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

   Nõrgima eeltingimuse staatiline analüüs pinukeeltel,

   mille juhendajad on Kalmer Apinis ja Vesal Vojdani,

   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.

Mirjam Iher
20. mai 2019. a.