TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Simmo Saan

Abstraktsete domeenide omaduspõhine testimine

Bakalaureusetöö (9 EAP)

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

Tartu 2018

Abstraktsete domeenide omaduspõhine testimine

Lühikokkuvõte:

One or two sentences providing a basic introduction to the field, comprehensible to a scientist in any discipline.

Two to three sentences of more detailed background, comprehensible to scientists in related disciplines.

One sentence clearly stating the general problem being addressed by this particular study.

One sentence summarising the main result (with the words “here we show´´ or their equivalent).

Two or three sentences explaining what the main result reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more general context.

Two or three sentences to provide a broader perspective, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion.

Võtmesõnad:
staatiline analüüs, andmevooanalüüs, abstraktne interpretatsioon, võred, Goblint

Veel märksõnu?

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

Õige CERCS?

Property-based Testing of Abstract Domains

Abstract:

Inglisekeelne lühikokkuvõte

Keywords:
static analysis, data-flow analysis, abstract interpretation, lattices, Goblint

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

Sisukord

Unsolved issues

Sissejuhatus

Viited sissejuhatuses?

Veel rohkem taustast, alustada kaugemalt: *static vs dynamic analysis*?

Staatiline programmianalüüs on võimalikult automaatne protsess, mis programmi lähtekoodi põhjal järeldab midagi selle programmi käitumise kohta. Staatilist analüüsi teostatakse mitmel põhjusel. Esiteks, programmi optimeerimise eesmärgil teostatakse analüüsi kompilaatorites, leidmaks kohti programmikoodis, mida on automaatse muudatusega võimalik optimeerida, ilma et sellest muutuks programmi käitumine. Teiseks, programmist vigade leidmise eesmärgil, leidmaks vigu, ilma et oleks tarvis programm käivitada ja vigane olukord esile kutsuda. Kolmandaks, programmi korrektsuse näitamiseks, veendumaks, et programm kindlasti käitub oodatud veatul moel.

Kolmas punkt ei erine teisest arusaadavalt

Mainida alternatiivseid staatilise analüüsi meetodeid?

Nimetada ja viidata olemasolevatele analüsaatoritele?

Andmevooanalüüs (ingl. *data-flow analysis*) on üks intuitiivne meetod staatilise programmianalüüsi teostamiseks. Selle keskseks ideeks on võimalike programmi seisundite, sh sageli muutujate võimalike väärtuste, määramine selle programmi igal täitmise sammul.

Üldiselt pole võimalik staatilise analüüsiga alati täpselt määrata programmi seisundit, sest see oleks samaväärne programmi käivitamisega. Seetõttu vaadeldakse ligikaudseid seisundeid, mis vastavad konkreetsetele seisunditele. Sellist ligikaudsete seisundite uurimist nimetatakse abstraktseks interpretatsiooniks (ingl. *abstract interpretation*) ja see põhineb rangel teoreetilisel alusel, millel on ligikaudsusele vaatamata head omadused. Nimelt, abstraktse interpretatsiooni teooria lubab, et analüüs on korrektne (ingl. *sound*), st kui uuritavast programmist otsitavat tüüpi viga ei leita, siis võib olla kindel, et seda seal päriselt ka ei ole.

Abstraktse interpretatsiooni õigeks toimimiseks on vajalik, et vaadeldavad ligikaudsed programmi seisundid, mis moodustavadki abstraktse domeeni, rahuldaks hulka omadusi, mis võimaldavad andmevooanalüüsi teostada sobiva võrrandisüsteemi lahendamise teel. Seetõttu on hädavajalik, et staatilist analüüsi teostav programm, analüsaator, ise oleks implementeeritud korrektselt, sest vastasel juhul pole teostatavate analüüside tulemused usaldusväärsed ja korrektsed.

Goblint on mitmelõimeliste C programmide staatiline analüsaator, mis keskendub mitmelõimelistes programmides esinevate vigade tuvastamisele, kasutades andmevooanalüüsi. Goblinti autoritele on teada, et analüsaator ei käitu alati oodatud viisil, vaid võib teha vigu, mis võivad rikkuda analüüsi korrektsuse.

Goblinti kirjeldus duplitseeritud viimasest peatükist

Midagi öelda omaduspõhise testimise kohta ikkagi?

Töö eesmärgiks on koostada ülevaatlik abstraktsete domeenide omaduste komplekt ja leida Goblintis implementeeritud domeenidest vigu, kontrollides vastavate omaduste kehtimist.

Esimeses peatükis antakse ülevaade abstraktse domeeni mõistest: esitatakse vajalikud definitsioonid ja tuuakse näiteid. Lisaks kirjeldatakse terviklikku andmevooanalüüsi ja domeeni rolli selles. Teises peatükis esitatakse abstraktsete domeenide matemaatilised omadused, mida testimisel kontrollida. Lisaks käsitletakse domeenide efektiivsuse ja abstraktsioonide korrektsuse temaatikat ning nende täiendavaid omadusi. Kolmandas peatükis selgitatakse omaduspõhise testimise metoodikat ja abstraktseid domeene Goblint analüsaatoris. Lõpuks viiakse läbi Goblinti domeenide testimine ja esitatakse tulemused.

Töö autori panuseks on võimalikult suure aga samas ka universaalse testitavate omaduste komplekti moodustamine. Praktilise osana täiendati Goblintit ja selle domeene nii, et omaduspõhist testimist oleks võimalik teostada, implementeeriti omaduste testid ja viidi läbi testimine.

1 Abstraktsed domeenid

Andmevooanalüüsiga püütakse võimalikult täpselt määrata programmi seisundit igas programmi punktis. Andmevooanalüüs kasutab programmi juhtimisvoograafi (ingl. *control-flow graph*), et järgida programmi seisundi muutumist selle võimalike töövoogude jooksul.

Definitsioon 1. Domeeniks nimetatakse programmi kõikvõimalike seisundite hulka [1].

Programmi seisundiks on ilmselt muutujate väärtused, aga ka mistahes keerulisemad uuritavad omadused, näiteks lõime hetkel hoitavate lukkude hulk [1] või kättesaadavadte omistamiste hulk (ingl. *available assignments*) [2:12]. Kuna staatilise analüüsiga üldiselt pole võimalik programmi seisundeid täpselt määrata, siis tehakse seda ligikaudsete seisundite abil, mis moodustavad **abstraktse domeeni**.

Selline informaalne definitsioon on ebapiisav mingisuguse teooria arendamiseks, mistõttu tegelikult vaadetakse domeene, mis moodustavad täieliku võre. Mida see täpselt tähendab, selgub pärast järgmist illustreerivat näidet.

1.1 Andmevooanalüüs

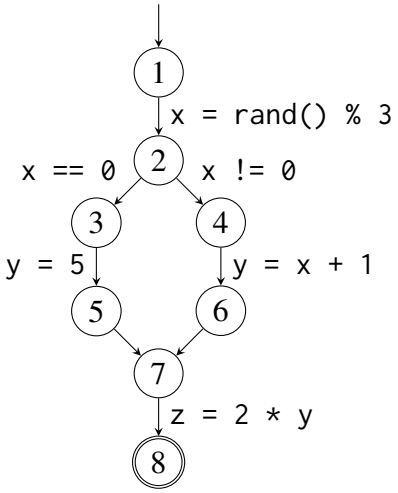
Andmevooanalüüsi ja abstraktse interpretatsiooni põhimõtete selgitamiseks parim viis on näite läbitegemine. Selleks olgu edaspidi vaatluse all programm jooniselt 1, kus on kõrvuti C-keelse lähtekoodi jupp ja selle juhtimisvoograaf, mille tippudes on programmi punktid, milles seisundeid uuritakse, ja servadel vastavad laused, mida täidetakse. Järgnevalt on käsitsi analüüsitud selle programmi täisarvuliste muutujate võimalike väärtusi igas programmi punktis:

- Punktis 1 pole muutujaid deklareeritud, seega nende väärtustest ei saa rääkida.
- Punktis 2 muutuja `x` väärtust üheselt määrata pole võimalik, sest see tuleb juhuarvu funktsioonist `rand()`, kuid jäägiga jagamise tulemusena kuulub see kindlasti hulka $\{0, 1, 2\}$[1]. Lisaks on vahepeal deklareeritud väärtustamata muutuja `y`, millel C keele semantikas vaikeväärtust pole ja seetõttu võib selle väärtus olla suvaline täisarv, st suvaline hulgast $\mathbb{Z}$ [3].
- Punktis 3, kus tingimus oli tõene, on `x` kindlasti ainult 0.
- Punktis 4, kus tingimus oli väär, saab eelnevast hulgast välistatud nulli eemaldamisel öelda, et muutuja `x` väärtus on hulgas $\{1, 2\}$.
- Punktis 5 on täpselt teada, et `y` on 5.

[1] C keele semantika on siinkohal keerukas, kuid (üldiselt võimalikud) negatiivsed jäägid on antud juhul välistatud, sest `rand()` funktsioon ei tagasta negatiivseid arve.

```c
int x = rand() % 3;
int y;
if (x == 0)
    y = 5;
else
    y = x + 1;
int z = 2 * y;
```

(a) C-keelne lähtekood



(b) Juhtimisvoograaf

Joonis 1. Tsüklita programmi näidis.

- Punktis 6, teades võimalikke muutuja x väärtusi, saab öelda, et muutuja y väärtus tehte tulemusena on hulgas $\{2, 3\}$.
- Punktis 7, kus kaks võimalikku tingimuslause haru uuesti kokku saavad, tuleb arvestada mõlema haru võimalike väärtustega, seega muutuja y väärtus kuulub hulka $\{2, 3, 5\}$.
- Punktis 8, teades võimalikke y väärtusi, saab öelda, et muutuja z väärtus tehte tulemusena on hulgas $\{4, 6, 10\}$.

Tehtud analüüs on võimalikult täpne, mis oleks ka automatiseeritud analüüsi eesmärgiks. Iseenesest poleks vale mõnes programmi punktis mõne muutujaga seostada suuremat võimalike väärtuste hulka, kuid sellisest analüüsist oleks vähem kasu, sest see tooks sisse ebavajalikku ebatäpsust. Järgnevalt ongi eesmärk matemaatiliselt formaliseerida sellise hea analüüsi teostamine, mis omakorda oleks aluseks analüüsi teoreetiliseks uurimiseks ja automatiseerimiseks.

1.2 Võred

Domeenide kirjeldamiseks kasutatakse matemaatilisi struktuure, mida nimetatakse võredeks. Kuigi võreteooria on arvestatav matemaatika haru, siis sügavamale laskumata on siin toodud vajalikud mõisted võredest aru saamiseks. Järgnevad eestikeelsed definitsioonid on refereeritud V. Laane loengukonspektist aines „Võreteooria“ [4], kuid tähistused on kohandatud programmianalüüsi kirjandusele omaseks [2:17]:

Definitsioon 2. Osaliselt järjestatud hulk on paar $(A, \sqsubseteq)$, kus A on hulk, millel on defineeritud binaarne seos $\sqsubseteq$, mis iga $a, b, c \in A$ korral rahuldab järgnevaid tingimusi:

- $a \sqsubseteq a$ (refleksiivsus),
- kui $a \sqsubseteq b$ ja $b \sqsubseteq c$, siis $a \sqsubseteq c$ (transitiivsus),
- kui $a \sqsubseteq b$ ja $b \sqsubseteq a$, siis $a = b$ (antisümmeetrilisus).

Domeenide puhul kasutatakse osalist järjestust seisundite täpsuse võrdlemiseks. Kokkuleppeliselt järjestatakse seisundid täpsemast ebatäpsema suunas — kirjutis $a \sqsubseteq b$ tähendab, et seisund a kirjeldab programmi olekut täpsemalt või sama täpselt kui seisund b. Teiste sõnadega, alati kui programmi olekut kirjeldab a, siis saab seda kirjeldada ka b-ga.
Järgnevalt olgu $(A, \sqsubseteq)$ osaliselt järjestatud hulk ja $X \subseteq A$.

Definitsioon 3. Elementi c nimetatakse hulga X **ülemiseks tõkkeks**, kui iga $x \in X$ korral $x \sqsubseteq c$. Vähimat ülemist tõket nimetatakse **ülemiseks rajaks**, st X-i iga ülemise tõkke d korral $c \sqsubseteq d$.

Definitsioon 4. Elementi c nimetatakse hulga X **alumiseks tõkkeks**, kui iga $x \in X$ korral $c \sqsubseteq x$. Suurimat alumist tõket nimetatakse **alumiseks rajaks**, st X-i iga alumise tõkke d korral $d \sqsubseteq c$.

Hulga X ülemist ja alumist raja tähistatakse vastavalt $\bigsqcup X$ ja $\bigsqcap X$. Kui $X = \{a, b\}$, siis tähistatakse ülemine ja alumine raja vastavalt $a \sqcup b$ ja $a \sqcap b$.

Domeenide puhul kasutatakse ülemisi tõkkeid mitme seisundi ühendamiseks, nii nagu seda oli vaja teha joonise 1 programmi punktis 7. Seejuures tahetakse loomulikult täpseimat ühendatud seisundit, mis ongi ühendatavate seisundite ülemine raja.

Milleks vaja alumist raja?

Definitsioon 5. Täielik võre on osaliselt järjestatud hulk, mille igal alamhulgal leidub ülemine ja alumine raja.

Täielikus võres $(A, \sqsubseteq)$ leidub vähim element $\bot = \bigsqcap A$ ja suurim element $\top = \bigsqcup A$, mida nimetatakse vastavalt *bottom*iks ja *top*iks.

Domeenidelt nõutaksegi, et need oleks täielikud võred, sest sellega kaasnevad ülal kirjeldatud võimalused seisunditega töötamiseks. Domeeni element $\top$ kirjeldab kõige üldisemat seisundit ehk seisundit, kus pole programmi oleku kohta mitte midagi teada. Element $\bot$ kirjeldab võimatut seisundit, milleks tüüpiliselt on saavutamatu (ingl. *unreachable*) programmi punkt ehk punkt, kuhu programmi täitmisel mitte kunagi ei jõutagi.

1.3 Alamhulkade domeen

Iga hulga S korral saab vaadelda selle kõigi alamhulkade hulka $\mathcal{P}(S)$, millel on loomulik osaline järjestus $\subseteq$. Osutub, et see on ka täielik võre, kus

- $a \sqsubseteq b \Leftrightarrow a \subseteq b$,
- $a \sqcup b = a \cup b$ ja $a \sqcap b = a \cap b$,
- $\bot = \varnothing$ ja $\top = S$.

Hulk X kirjeldab väärtust z parajasti siis, kui $z \in X$. Sellega on defineeritud, kuidas peaks seostama domeeni elemente konkreetsete võimalike väärtustega.

Description relation — kas üldse mõtet mainida? Vb jätta Galois ühenduste juurde.

Ülal käsitsi tehtud joonise 1 programmi analüüsis oligi iga muutuja väärtusi analüüsitud alamhulkade domeeni $\mathbb{D} = (\mathcal{P}(\mathbb{Z}), \subseteq)$ abil. Programmi punktis 7 toimunud muutuja y seisundite ühendamine on võrede terminites $\{5\} \sqcup \{2, 3\}$. Antud programmi analüüsimiseks on selle domeeniga vaja seostada kahte tüüpi tehted:

Konstantide abstraheerimine Lause `y = 5` abstraktseks teostamiseks, st muutujaga y domeeni elemendi seostamiseks, on esinev konstant vaja sobivalt abstraheerida. Täisarvude alamhulkade domeenis sobib konstantse väärtuse a jaoks ilmselt element $\{a\}$.

Abstraktne aritmeetika Lause `y = x + 1` abstraktseks teostamisks on vaja teostada liitmine muutuja x seisundi $\{1, 2\}$ ja konstandi seisundi $\{1\}$ vahel. Täisarvude alamhulkade domeenis saab elementide $A, B \in \mathcal{P}(\mathbb{Z})$ liitmise defineerida kui

$$A + B = \{a + b \mid a \in A, b \in B\}$$

ning sarnaselt võib defineerida ülejäänud tehted.

Nende matemaatiliste vahenditega ongi võimalik näites tehtu süstematiseerida. Siiski kirjeldab kasutatud domeen korraga ainult ühe muutuja väärtuseid, kuid muutujatevahelisi toiminguid otseselt mitte — seda peab ikkagi domeeniväliselt teostama. Oleks veelgi parem, kui tervet programmi olekut, st kõigi muutujate seisundeid korraga, saaks vaadelda ühe keerukama domeeni elementidena. Selleks võetaksegi kasutusele kujutused.

1.4 Kujutusdomeen

Olgu Var programmi muutujate hulk ja Val domeen, milles vaadeldakse üksiku muutuja seisundit. Sel juhul saab vaadelda abstraktsete muutujate väärtustuste domeeni [2:45]

$$\mathbb{D} = (\mathsf{Var} \to \mathsf{Val})_\bot = (\mathsf{Var} \to \mathsf{Val}) \cup \{\bot\}.$$

Domeeni elementideks on kujutused muutujate hulgast nende abstraktsete väärtuste hulka, mis on väga analoogilised konkreetsete muutujate väärtustustega funktsionaalselt

Tabel 1. Tsüklita näiteprogrammi (joonisel 1) analüüsi lõpptulemus kujutuste domeenis.

Punkt	x	y	z
1	$\top$	$\top$	$\top$
2	$\{0,1,2\}$	$\top$	$\top$
3	$\{0\}$	$\top$	$\top$
4	$\{1,2\}$	$\top$	$\top$
5	$\{0\}$	$\{5\}$	$\top$
6	$\{1,2\}$	$\{2,3\}$	$\top$
7	$\{0,1,2\}$	$\{2,3,5\}$	$\top$
8	$\{0,1,2\}$	$\{2,3,5\}$	$\{4,6,10\}$

kirjeldatuna. Element $\bot$ on tehislikult lisatud, et domeen moodustaks täieliku võre, ja tähendab saavutamatut programmi punkti. Osaline järjestus selles domeenis on defineeritud järgnevaga:

$$D_1 \sqsubseteq D_2 \quad \Longleftrightarrow \quad D_1 = \bot \quad \vee \quad \forall x \in \mathsf{Var}\ D_1(x) \sqsubseteq D_2(x).$$

$\sqcup, \sqcap, \top$?

Joonisel 1 tehtud näites olid muutujad $\mathsf{Var} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$ ja väärtused domeenis $\mathsf{Val} = \mathcal{P}(\mathbb{Z})$. Sellises kujutuste domeenis saabki kirjeldada tervet seisundit korraga ning sama analüüsi lõpptulemus oleks selline, nagu näidatud tabelis 1.

Kasutades kogu olekut kirjeldavaid domeeni elemente, on programmi laused ja avaldised juhtimisvoograafis kirjeldatavad funktsioonidena lähteseisundist sihtseisundisse. Neid nimetatakse **üleminekufunktsioonideks** (ingl. *transfer function*) ja samas näites võivad need olla järgnevad:

$$tf_{1,2}(d) = d[\mathsf{x} \mapsto \{0,1,2\}],$$
$$tf_{2,3}(d) = d[\mathsf{x} \mapsto \{0\}], \qquad tf_{2,4}(d) = d[\mathsf{x} \mapsto d(\mathsf{x}) \setminus \{0\}],$$
$$tf_{3,5}(d) = d[\mathsf{y} \mapsto \{5\}], \qquad tf_{4,6}(d) = d[\mathsf{y} \mapsto d(\mathsf{x}) + \{1\}],$$
$$tf_{7,8}(d) = d[\mathsf{z} \mapsto \{2\} * d(\mathsf{y})],$$

kus kirjutis $d[\mathsf{x} \mapsto \{0\}]$ tähendab funktsiooni uuendamist (ingl. *function update*) ehk sama funktsiooni d, mis argumendil x omab nüüd uut väärtust $\{0\}$. Aritmeetilised tehted hulkade vahel on sellised nagu nad on sisemises domeenis Val.

Funktsiooni uuendamise def?

Abstraktse aritmeetika jaoks eraldi operaatorid: $\oplus, \odot, \otimes$?

Seejuures üleminekufunktsioonid saab keele semantika ning lausete ja avaldiste struktuuri mallide järgi mehaaniliselt defineerida ehk konkreetse programmi analüüsil toimub see automatiseeritult.

Kujutusdomeeni kasutamisega on võimalik lihtsam lihtsam ühte muutujat korraga kirjeldav domeen laiendada kõigile muutujatele korraga. Seejuures muutub võimalikuks järgida andmete liikumist muutujate vahel, mis vastab juba paremini andmevooanalüüsi nimele.

1.5 Algoritmiline analüüs

Võrede, nende abstraktsete tehete ja üleminekufunktsioonide abil on formaliseeritud suur osa esialgses näites intuitiivselt tehtust, kuid täielikult automatiseeritud analüüsini on jäänud veel viimane samm, mis võimaldaks kogu analüüsi algoritmiliselt kirjeldada. Kõigepealt peavad paigas olema kaks asja:

1. Domeen $\mathbb{D}$, mille abil analüüsi teostatakse ja mis on võimeline kirjeldama uuritavaid programmi omadusi.
2. Üleminekufunktsioonide defineerimise protsess programmeerimiskeele lausetest.

Nende olemasolul saab konkreetse programmi analüüsi teostada järgnevate sammudega:

1. Iga programmi punkti i jaoks võtta kasutusele üks muutuja x_i, mis vastab otsitavale programmi abstraktsele seisundile selles punktis.

2. Iga juhtimisvoograafi serva $k \to i$ jaoks defineerida fikseeritud protsessi abil üleminekufunktsioon $tf_{k,i}$. Nõnda saab iga serva jaoks moodustada võrratuse $x_i \sqsupseteq tf_{k,i}(x_k)$. See tähendab, et analüüsis otsitav seisund võib olla ebatäpsem kui konkreetne üleminek ette näeb.

   Lisaks üleminekutele koostatakse võrratus ka programmi alguspunkti, milleks olgu x_1, jaoks kujul $x_1 \sqsupseteq \imath$, kus $\imath$ kirjeldab algseisundit.

3. Neist võrratustest moodustub süsteem, kus iga muutuja on vasakul täpselt ühel korral, selleks vajadusel võrratusi parema poole ülemraja järgi ühendades:

$$\begin{cases} x_1 \sqsupseteq f_1(x_1, \ldots, x_n), \\ \vdots \\ x_n \sqsupseteq f_n(x_1, \ldots, x_n), \end{cases}$$

   kus $f_1, \ldots, f_n$ on moodustatud võrratuste parematest pooltest (üleminekufunktsioonidest ja nende ülemrajadest).

   Täpsemalt defineerida üleminekufunktsioonide ülemraja?

   Sama saab lühemalt kirja panna, kui vaadelda hoopis täielikku võre $\mathbb{D}^n$, mille elementide positsioonil i on programmi punkti i seisund. Selle võre tehted on defineeritud punktiviisiliselt.

Otsekorrutise domeeni eraldi defineerimine?

Sellises võres on muutujaks $\bar{x} = (x_1, \ldots, x_n)$ ning funktsioonid ühendatud kokku ühte funktsiooni $\bar{f}(\bar{x}) = (f_1(\bar{x}), \ldots, f_n(\bar{x}))$. Sellega on kogu eelnev võrratuste süsteem ühendatud üheks ainsaks võrratuseks $\bar{x} \sqsupseteq \bar{f}(\bar{x})$ [2:21].

Lisaks eeldatakse, et funktsioonid f_i ja seega kaudselt funktsioon $\bar{f}$ on monotoonsed, mis tähendab, et nad säilitavad rakendamisel täpsusega määratud järjestust [2:20].

Definitsioon 6. Olgu $\mathbb{D}_1$ ja $\mathbb{D}_2$ osaliselt järjestatud hulgad. Funktsiooni $f : \mathbb{D}_1 \to \mathbb{D}_2$ nimetatakse **monotoonseks**, kui iga $a, b \in \mathbb{D}_1$ korral

$$a \sqsubseteq b \quad \Longrightarrow \quad f(a) \sqsubseteq f(b) \text{ [5:23].}$$

4. Saadud võrratusele (ehk kaudselt algsele võrratuste süsteemile) tuleb leida vähim lahend. Selle iga lahend on üks korrektne analüüs, mis iga programmi punkti kohta sisaldab selle abstraktset seisundit. Vähim sobiv lahend on ühtlasi täpseim võimalik korrektne analüüs, ehk see, millest enim kasu oleks.

   Saab näidata, et see on samaväärne võrrandi $\bar{x} = \bar{f}(\bar{x})$ vähima lahendi leidmisega, mida nimetatakse ka vähima püsipunkti leidmiseks. Selleks on kõige naiivne iteratiivne algoritm, mis alustab väärtusest $\bar{x}_0 = \overline{\bot} = (\bot, \ldots, \bot)$ ja järjest iga $\bar{x}_i$ korral arvutab $\bar{x}_{i+1} = \bar{f}(\bar{x}_i)$ kuni lõpuks jõutakse nii kaugele, et mingi i korral $\bar{x}_i = \bar{x}_{i+1}$, st funktsiooni väärtused stabiliseeruvad ja funktsiooni edasi rakendamine tulemust ei muuda. Leitud lahendist ongi võimalik välja lugeda analüüsi tulemused iga programmi punkti jaoks [2:21].

Joonise 1 programmi analüüsimisel oleks võrratuste ja võrrandite süsteemid järgnevad:

$$\begin{cases} x_1 \sqsupseteq \top \\ x_2 \sqsupseteq tf_{1,2}(x_1) \\ x_3 \sqsupseteq tf_{2,3}(x_2) \\ x_4 \sqsupseteq tf_{2,4}(x_2) \\ x_5 \sqsupseteq tf_{3,5}(x_3) \\ x_6 \sqsupseteq tf_{4,6}(x_4) \\ \left.\begin{array}{l} x_7 \sqsupseteq x_5 \\ x_7 \sqsupseteq x_6 \end{array}\right\} \Rightarrow x_7 \sqsupseteq x_5 \sqcup x_6 \\ x_8 \sqsupseteq tf_{7,8}(x_7) \end{cases} \quad \text{ja} \quad \begin{cases} x_1 = \top \\ x_2 = tf_{1,2}(x_1) = x_1[\mathsf{x} \mapsto \{0,1,2\}] \\ x_3 = tf_{2,3}(x_2) = x_2[\mathsf{x} \mapsto \{0\}] \\ x_4 = tf_{2,4}(x_2) = x_2[\mathsf{x} \mapsto d(\mathsf{x}) \setminus \{0\}] \\ x_5 = tf_{3,5}(x_3) = x_3[\mathsf{y} \mapsto \{5\}] \\ x_6 = tf_{4,6}(x_4) = x_4[\mathsf{y} \mapsto d(\mathsf{x}) + \{1\}] \\ x_7 = x_5 \sqcup x_6 \\ x_8 = tf_{7,8}(x_7) = x_7[\mathsf{z} \mapsto \{2\} * d(\mathsf{y})] \end{cases} .$$

Nende iteratiivne lahendamine on samm-sammult toodud lisas I. On näha, et algoritmiline analüüs jõudis oodatud tulemuseni. Nagu näha, siis naiivne iteratsioon on üsna

ebaefektiivne viis vähima püsipunkti leidmiseks, kuna väärtuste stabiliseerumine võib võtta palju samme, kusjuures iga kord arvutatakse uuesti ka need väärtused, mis enam niikuinii ei muutu. Loomulikult on olemas mitmeid palju efektiivsemaid meetodeid, näiteks *round-robin* iteratsioon ja *worklist* algoritm [2:24,83].

Algoritmide nimesid kuidagi tõlkida?

Sellegipoolest pole vaadeldud analüüs kuigi praktiline, sest võimalike arvuliste väärtuste hulgad võivad olla suured või lausa lõpmatud. Üldiselt pole võimalik lõpmatuid hulki programmis efektiivselt esitada ja nendega opereerida. Seetõttu loobutakse ülimast täpsusest alamhulkade kujul ja kasutatakse ebatäpsemaid domeene väärtuste kirjeldamiseks, mida on efektiivselt võimalik analüsaatorisse implementeerida. Alamhulkade domeene saab siiski muudeks analüüsideks mõistlikult kasutada ning neil on oluline roll domeenide abstraheerimise uurimisel.

1.6 Intervalldomeen

Intervalldomeen on domeen, milles täisarvude väärtuste abstraheerimiseks kasutatakse arvtelje lõike. Selline abstraktsioon on arvuhulkadega võrreldes efektiivsem, olles samaaegselt praktikas ka piisavalt täpne.

Definitsioon 7. Intervalldomeeniks [2:55] nimetatakse hulka

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\},$$

millel on osaline järjestus

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \quad \Longleftrightarrow \quad l_2 \leq l_1 \wedge u_1 \leq u_2.$$

Ülejäänud domeenid ka definitsiooni kujul?

Sellises domeenis

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min\{l_1, l_2\}, \max\{u_1, u_2\}],$$
$$[l_1, u_1] \sqcap [l_2, u_2] = [\max\{l_1, l_2\}, \min\{u_1, u_2\}], \quad \text{kui } \max\{l_1, l_2\} \leq \min\{u_1, u_2\}.$$

Intervallide järjestus on määratud nende omavahelise sisalduvuse kaudu ning ülem- ja alamraja on vastavalt lõikude ühend ja ühisosa. Lõigu otspunktides on lubatud vastavad lõpmatust kirjeldavad väärtused, mis võimaldavad rääkida selles domeenis suurimast elemendist $\top = [-\infty, +\infty]$. Kuna alumine raja on ainult tinglikult defineeritud, siis intervalldomeen ei moodusta täielikku võre. See pole probleem, kuna pisut täiendatud domeen $\mathbb{I}_\perp = \mathbb{I} \cup \{\perp\}$ osutub täielikuks võreks, kui seda vaja peaks olema.

Lõik $[l, u]$ kirjeldab täisarvu z parajasti siis, kui $l \leq z \leq u$.

Analoogiliselt alamhulkade domeeniga, tuuakse siin programmide analüüsimiseks sisse:

Konstantide abstraheerimine Konstantsele täisarvulisele väärtusele a vastab intervall $[a, a]$.

Abstraktne aritmeetika Intervallidel saab samuti defineerida erinevad aritmeetilised tehted, moodustades intervallaritmeetika. Näiteks lõikude $[l_1, u_1], [l_2, u_2] \in \mathbb{I}$ korral:

$$[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2],$$
$$[l_1, u_1] \cdot [l_2, u_2] = [\min\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}, \max\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}].$$

Siin on tehete defineerimine keerulisem kui alamhulkade domeenis, nagu korrutamisest paistab, kusjuures lisaks peab defineerima juhud, kus mõned otspunktidest on $-\infty$ või $+\infty$.

Kas peaks seda ka siin tegema?

Valides eelnevas joonisel 1 tehtud näites seekord $\mathsf{Val} = \mathbb{I}$, saab taaskord kirjeldada üleminekufunktsioonid, kasutades seekord intervallide operatsioone. Tabelis 2 on toodud samasuguse algoritmilise analüüsi lõpptulemus, kui seda oleks tehtud intervalldomeeni abil. Nagu näha, siis intervallide kasutamine on ebatäpsem kui alamhulkade kasutamine, sest kaob informatsioon selle kohta, kui mõni väärtus lõigu keskelt tegelikult ei saa esineda. Samas on analüüs ikkagi korrektne, sest ühtegi päriselt võimalikku väärtust pole ekslikult välistatud.

Tabel 2. Tsüklita näiteprogrammi (joonisel 1) analüüsi lahend intervalldomeenis.

	x	y	z
x_1	$\top$	$\top$	$\top$
x_2	$[0, 2]$	$\top$	$\top$
x_3	$[0, 0]$	$\top$	$\top$
x_4	$[1, 2]$	$\top$	$\top$
x_5	$[0, 0]$	$[5, 5]$	$\top$
x_6	$[1, 2]$	$[2, 3]$	$\top$
x_7	$[0, 2]$	$[2, 5]$	$\top$
x_8	$[0, 2]$	$[2, 5]$	$[4, 10]$

2 Domeeni omadused

Selles peatükis tuuakse üldised abstraktsete domeenide omadused, mis kataks kõik universaalselt kontrollitavad aspektid. Nende hulka kuuluvad täielike võrede, programmianalüüsi spetsiifilised ja abstraktsiooni korrektsuse omadused.

2.1 Võre omadused

Kuna nõutakse, et domeenid oleks täielikud võred, siis peavadki nad kõiksugu võrede tingimusi rahuldama. Järgnev on ülevaade erinevatest omadustest, mis täielikel võredel kehtima peaks:
Olgu $\mathbb{D}$ täielik võre, siis iga $a, b, c \in \mathbb{D}$ korral peavad kehtima järgnevad tingimused:

Osalise järjestuse omadused (definitsioonist 2)
- $a \sqsubseteq a$ (refleksiivsus);
- kui $a \sqsubseteq b$ ja $b \sqsubseteq c$, siis $a \sqsubseteq c$ (transitiivsus);
- kui $a \sqsubseteq b$ ja $b \sqsubseteq a$, siis $a = b$ (antisümmeetrilisus);

Rajade omadused [6]
- $a \sqsubseteq a \sqcup b$ ja $b \sqsubseteq a \sqcup b$ (definitsioonist 3);
- $a \sqcap b \sqsubseteq a$ ja $a \sqcap b \sqsubseteq b$ (definitsioonist 4);
- kui $a \sqsubseteq c$ ja $b \sqsubseteq c$, siis $a \sqcup b \sqsubseteq c$ (definitsioonist 3);
- kui $c \sqsubseteq a$ ja $c \sqsubseteq b$, siis $c \sqsubseteq a \sqcap b$ (definitsioonist 4);

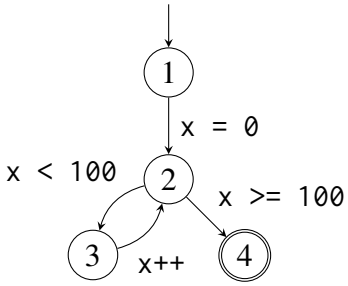
Rajade tehete omadused [4:6, 5:39]
- $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ ja $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$ (assotsiatiivsus);
- $a \sqcup b = b \sqcup a$ ja $a \sqcap b = b \sqcap a$ (kommutatiivsus);
- $a \sqcup a = a$ ja $a \sqcap a = a$ (idempotentsus);
- $a \sqcup (a \sqcap b) = a$ ja $a \sqcap (a \sqcup b) = a$ (neelduvus);

Vähima ja suurima elemendi omadused [6]
- $\bot \sqsubseteq a$;
- $a \sqsubseteq \top$;
- $a \sqcup \bot = a$;
- $a \sqcap \top = a$;

Järjestuse ja rajade tehete seosed Järgnevad on ekvivalentsed [5:39]:
1. $a \sqsubseteq b$,
2. $a \sqcup b = b$,
3. $a \sqcap b = a$.

```
int x = 0;
while (x < 100)
    x++;
```

(a) C-keelne lähtekood



(b) Juhtimisvoograaf

Joonis 2. Tsükliga programmi näidis.

2.2 Laiendamine ja kitsendamine

Kui joonisel 1 olev programm oli tsükliteta ja selle analüüsi saaks teostada ka ilma võrratuste süsteemi püsipunkti leidmata teostada, siis üldiselt see nii pole. Huvitavamad programmid sisaldavad tsükleid, mille korral ülalkirjeldatud algoritm on analüüsimiseks vajalik. Olgu vaatluse all programm jooniselt 2 ja domeen $\mathbb{D} = (\mathsf{Var} \to \mathbb{I})_\bot$ nagu enne, siis võrratuste ja võrrandite süsteemid on järgnevad:

$$\begin{cases} x_1 \sqsupseteq \top \\ x_2 \sqsupseteq tf_{1,2}(x_1) \sqcup tf_{3,2}(x_3) \\ x_3 \sqsupseteq tf_{2,3}(x_2) \\ x_4 \sqsupseteq tf_{2,4}(x_2) \end{cases} \quad \text{ja} \quad \begin{cases} x_1 = \top \\ x_2 = tf_{1,2}(x_1) \sqcup tf_{3,2}(x_3) = \\ \quad = x_1[\mathsf{x} \mapsto [0,0]] \sqcup x_3[\mathsf{x} \mapsto d(\mathsf{x}) + [1,1]] \\ x_3 = tf_{2,3}(x_2) = x_2[\mathsf{x} \mapsto d(\mathsf{x}) \sqcap [-\infty, 99]] \\ x_4 = tf_{2,4}(x_2) = x_2[\mathsf{x} \mapsto d(\mathsf{x}) \sqcap [100, +\infty]] \end{cases} .$$

Nende vähim püsipunkt (tabelis 3a) leitakse naiivse iteratsiooni 204. sammul. Itereerimisel sisuliselt tehakse läbi kõik 100 tsükli iteratsiooni, mistõttu sellise süsteemi püsipunkti leidmise ajaline keerukus on $O(k)$, kus k on tsükli iteratsioonide arv. Selline ebaefektiivsus on vastuvõetamatu, sest toimub justkui programmi kogu töö simuleerimine, mis on vastuolus staatilise analüüsi eesmärgiga.

Laiendamine Ebaefektiivsuses on süüdi asjaolu, et intervalldomeenis leiduvad **lõpmatud rangelt kasvavad ahelad**.

Selgitada kuidagi ahelaid

Püsipunkti leidmise igal iteratsioonil tsüklis muutuja väärtuste lõik suurenes ühe võrra seetõttu stabiliseerumiseks läks ebameeldivalt palju iteratsioone. Probleemi lahendamiseks muudetakse analüüsimist nii, et püsipunkti leidmine toimuks kiirendatult, tuues meelega sisse täiendava ebatäpsuse. Sellist võtet nimetatakse **laiendamiseks** (ingl. *widening*).

Lahendatav võrratus $\bar{x} \sqsupseteq \bar{f}(\bar{x})$ kirjutatakse samaväärsel akumuleerival kujul $\bar{x} = \bar{x} \sqcup \bar{f}(\bar{x})$, kus ülemraja asemele valitakse sobiv **laiendamise operaator** $\sqcup$ ja otsitakse vähim püsipunkt võrrandile $\bar{x} = \bar{x} \sqcup \bar{f}(\bar{x})$. Intervallidel defineeritakse see järgnevalt:

$$[l_1, u_1] \sqcup [l_2, u_2] = [l, u], \text{ kus}$$
$$l = \begin{cases} l_1, & \text{kui } l_1 \leq l_2, \\ -\infty, & \text{muidu} \end{cases} \qquad \text{ja} \qquad u = \begin{cases} u_1, & \text{kui } u_2 \leq u_1, \\ +\infty, & \text{muidu.} \end{cases}$$

Intervallide laiendamise idee seiseb selles, et intervall suurenedes suureneb kohe vastavas suunas lõpmatusse, garanteerides, et intervall saab suureneda ülimalt kaks korda, mis muudab kasvavad ahelad lõplikeks. Sellise modifikatsiooniga süsteemi vähim püsipunkt (tabelis 3b) leitakse juba naiivse iteratsiooni 6. sammul, mis ühest küljest on oluline võit, teisest kaotus, sest lahend pole nii täpne kui eelnevalt.

Kitsendamine Kuna $\bar{f}$ on monotoonne, siis selle rakendamine püsipunktile tulemust ebatäpsemaks ei muuda, aga võib ebatäpset laiendatud tulemust siiski parandada. Sellist võtet nimetatakse **kitsendamiseks** (ingl. *narrowing*) ning eelmisele laiendatud lahendile kitsendamist rakendades stabiliseerub tulemus (tabelis 3c) 4. sammul, jõudes sama tulemuseni, mis ilma laiendamist ja kitsendamist. Üldiselt see siiski nii ei pruugi olla, kuid tulemuse täpsust suurendab ikka.

Kitsendamisel võib tekkida analoogiline probleem **lõpmatute rangelt kahanevate ahelatega**, mistõttu funktsiooni korduv rakendamine mõistliku sammude arvuga ei stabiliseeru. Samas võib kitsendamist lõpetada mistahes hetkel ja analüüs on ikka korrektne. Kitsendamise stabiliseerumiseks kasutatakse analoogilist võtet: lahendatav võrratus kirjutatakse samaväärsel kujul $\bar{x} = \bar{x} \sqcap \bar{f}(\bar{x})$, kus alamraja asemele valitakse sobiv **kitsendamise operaator** $\sqcap$ ja otsitakse vähim püsipunkt võrrandile $\bar{x} = \bar{x} \sqcap \bar{f}(\bar{x})$. Intervallidel defineeritakse see järgnevalt:

$$[l_1, u_1] \sqcap [l_2, u_2] = [l, u], \text{ kus}$$
$$l = \begin{cases} l_2, & \text{kui } l_1 = -\infty, \\ l_1, & \text{muidu} \end{cases} \qquad \text{ja} \qquad u = \begin{cases} u_2, & \text{kui } u_1 = +\infty, \\ u_1, & \text{muidu.} \end{cases}$$

Intervallide kitsendamise idee seiseb selles, et intervall vähenedeb ainult lõpmatusest, garanteerides, et intervall saab väheneda ülimalt kaks korda, mis muudab kahanevad ahelad lõplikeks. Uuritud näitel kiirendatud kitsendamine jõuab sama tulemuseni kui tavaline kitsendamine (tabelis 3c). Üldiselt see nii ei pruugi olla, sest kiirendamisel tuuakse sisse meelega täiendav ebatäpsus.

Omadused Laiendamise ja kitsendamise operaatorid peavad analüüsi korrektsuse tagamiseks rahuldama teatud tingimusi.
Olgu $\mathbb{D}$ abstraktne domeen, siis iga $a, b \in \mathbb{D}$ korral peavad kehtima järgnevad tingimused:

Tabel 3. Tsükliga näiteprogrammi (joonisel 2) analüüsi lahendid erinevatel meetoditel.

(a) Iteratsiooni lahend

	x
x_1	$\top$
x_2	$[0, 100]$
x_3	$[0, 99]$
x_4	$[100, 100]$

(b) Laiendamise tulemus

	x
x_1	$\top$
x_2	$[0, +\infty]$
x_3	$[0, +\infty]$
x_4	$[100, +\infty]$

(c) Kitsendamise tulemus

	x
x_1	$\top$
x_2	$[0, 100]$
x_3	$[0, 99]$
x_4	$[100, 100]$

- $a \sqcup b \sqsubseteq a \sqcup b$ [2:61];
- $a \sqcap b \sqsubseteq a \sqcap b \sqsubseteq a$ [2:66].

Kusjuures need operaatorid pole tingimata assotsiatiivsed, kommutatiivsed, jne nagu nende algsed analoogid. Neid kasutatakse nii nagu ülal toodud, kus vasak operand on vana seisund ja parem operand on uus seisund. Sellel asjaolul põhineb nende defineerimine ja seda ka nähtud intervalldomeeni puhul.

2.3 Abstraktsiooni korrektsus

Consistent abstract interpretations – Cousot. Korrektsus? Kooskõlalisus?

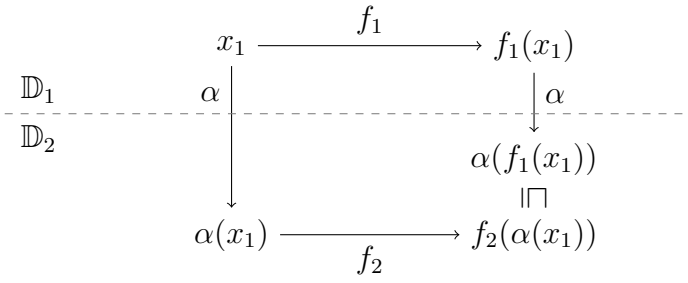
Abstraktse domeeni defineerimisel tekib küsimus, kas saadud domeen abstraheerib võimalikke olukordi korrektselt, st käitub mingis mõttes sama moodi. Näiteks intervalldomeen peaks olema täisarvude alamhulkade domeeni abstraktsioon, mille tehted käituks nii, nagu nad käituks konkreetsete alamhulkadega. Sellist olukorda kirjeldab järgmine definitsioon:

Definitsioon 8. Domeeni $\mathbb{D}_2$ nimetatakse domeeni $\mathbb{D}_1$ **korrektseks abstraktsiooniks**, kui leiduvad funktsioonid $\alpha : \mathbb{D}_1 \to \mathbb{D}_2$ ja $\gamma : \mathbb{D}_2 \to \mathbb{D}_1$, mis rahuldavad järgnevaid tingimusi [7:242]:

1. α on monotoonne,
2. γ on monotoonne,
3. iga $x_2 \in \mathbb{D}_2$ korral $x_2 = \alpha(\gamma(x_2))$,
4. iga $x_1 \in \mathbb{D}_1$ korral $x_1 \sqsubseteq \gamma(\alpha(x_1))$,
5. iga $x_1 \in \mathbb{D}_1$ korral $\alpha(f_1(x_1)) \sqsubseteq f_2(\alpha(x_1))$, kus f_1 on monotoonne tehe domeenis $\mathbb{D}_1$ ja f_2 sellele vastav monotoonne tehe domeenis $\mathbb{D}_2$.

Funktsiooni α nimetatakse abstraktsioonifunktsiooniks (ingl. *abstraction function*) ja funktsiooni γ konkretisatsioonifunktsiooniks (ingl. *concretization function*). Seega võib rääkida, et $\mathbb{D}_1$ on konkreetne domeen ja $\mathbb{D}_2$ selle abstraktne domeen.

Paremad tõlked?



Joonis 3. Definitsiooni 8 tingimust 5 illustreeriv skeem.

Tingimus 3 nõuab, et konkretiseerimine ei kaotaks täpsust. Tingimus 4 lubab, et abstraheerimine kaotab täpsust (aga ei suurenda seda). Tingimust 5 illustreerib joonis 3 ja see tähendab, et abstraktse tehte teostamine abstraktsel väärtusel pole täpsem, kui konkreetse tehte teostamine konkreetsel väärtusel.

Viimast tingimust saab samaväärselt sõnastada γ kaudu. Lisaks kogu definitsioonist järeldub domeenidevahline Galois' vastavus (ingl. *Galois connection*), millel on oluline roll abstraktse interpretatsiooni formaliseerimisel ja üleüldse võreteoorias [7].

Domeenide $\mathbb{D}_1 = \mathcal{P}(\mathbb{Z})$ ja $\mathbb{D}_2 = \mathbb{I}_\perp$ korral [7:243]

$$\alpha(x_1) = \begin{cases} \perp, & \text{kui } x_1 = \varnothing, \\ [\min x_1, \max x_1], & \text{muidu;} \end{cases}$$

$$\gamma(x_2) = \begin{cases} \varnothing, & \text{kui } x_2 = \perp, \\ \{l, \ldots, u\}, & \text{kui } x_2 = [l, u]. \end{cases}$$

Omadused Kui on teada, millist konkreetset domeeni mingi abstraktne domeen abstraheerib, ja nende vahel on defineeritud abstraheerimine ja konkretiseerimine, siis peavad kehtima definitsiooni 8 tingimused.

Isegi kui konkretiseerimisfunktsioon leidub ja on matemaatiliselt kirjeldatud, siis ei pruugi olla võimalik seda analüsaatoris implementeerida, sest abstraktsele väärtusele vastav konkreetne väärtus võib olla ebapraktiliselt mahukas või isegi lõpmatu. Nii on näiteks ülaltoodud alamhulkade ja intervallide domeenide puhul, kus $\gamma([-\infty, +\infty]) = \mathbb{Z}$. Konkretiseerimisfunktsiooni puudumisel analüsaatori implementatsioonis pole loomulikult sellega seonduvat võimalik verifitseerida, kuid definitsiooni tingimusi 1 ja 5 saab kontrollida ikka. Seejuures ongi kasulik tingimuse 5 toodud kuju, mitte selle konkretiseerimisega kuju.

3 Omaduspõhine testimine

Omaduspõhine testimine (ingl. *property-based testing*) on programmide testimise metoodika, mis põhineb soovitud omaduste verifitseerimisel juhuslikult genereeritud testandmetel. Selle lähenemise populariseeris Haskelli teek nimega QuickCheck [8], mis on nüüdseks implementeeritud ka paljude teiste programmeerimiskeelte jaoks ja mille kasutamine on funktsionaalprogrammeerimises laialdane.

Omaduspõhine testimine sobib hästi matemaatilist laadi omaduste, nagu eelmises peatükis toodud domeenide omaduste, kontrollimiseks, sest seda saab teha lihtsa tingimusega, vajamata sobivate testandmete väljamõtlemist. See võimaldab samu omadusi kontrollida kõigil domeenidel, sõltumata isegi nende elementide tüübist.

3.1 Põhimõtted

Testitavad omadused kirjeldatakse predikaatidena, millele omaduspõhise testimise raamistik genereerib hulga juhuslikke argumente, millel arvutab välja predikaadi väärtuse. Kui see on kõigil genereeritud argumentidel tõene, siis loetakse test läbituks ja omadus kehtivaks. Kui mõnel argumendil on predikaat väär, loetakse test läbikukkunuks ja omadus mittekehtivaks. Viimasel juhul kuvatakse probleemne argument ka testijale.

Omaduspõhise testimise raamistikes on tüüpiliselt juba defineeritud, kuidas vastava keele põhiliste andmetüüpide, sh paaride, järjendite jne, väärtusi juhuslikult genereerida. Puuduvate ja enda defineeritud andmetüüpide jaoks tuleb see ise juurde lisada. Seejuures „juhuslik“ ei pea tähendama täielikult juhuslikku protsessi, vaid võib alati genereerida ka hulka huvitavaid ja sageli probleeme tekitavaid erijuhte (nt arv null, tühi järjend jne).

Juhuslikult genereeritud argumendid võivad olla „suured“: suured arvud, pikad sõned või järjendid, sügavad puud jne. Võib kergesti juhtuda, et mõne omaduse kontranäide on seetõttu inimese jaoks raskesti hoomatav ja jääb ebaselgeks, mis on selle argumendi juures erilist, et omadus ei kehti. Vigade analüüsimise lihtsustamiseks kasutatakse omaduspõhises testimises vääravate argumentide lihtsustamist ehk vähendamist (ingl. *shrinking*). Kirjeldatakse, kuidas leitud väärtusest saada lihtsamaid, mis oleks algsega piisavalt sarnased, et nende korral võiks omadus samuti mitte kehtida. Näiteks järjendist mingite elementide eemaldamine, et pikkus sellest väheneks. Vähendatud argumentidel väärtustatakse predikaat uuesti, leidmaks neid, mille korral omadus ikka ei kehti, ja rakendatakse vähendamist taas nii palju kordi kui võimalik, et lihtsustada probleemset argumenti maksimaalselt, muutes selle paremini hoomatavaks.

Tüüpiliselt on ka põhiliste andmetüüpide vähendaminie testimisraamistikesse juba sisse ehitatud, aga sedagi on võimalik ise laiendada oma andmetüüpidele.

```
module type S =
sig
  type t (* domeeni elementide tüüp *)
  val equal: t -> t -> bool (* seos = *)
  val leq: t -> t -> bool (* seos ⊑ *)
  val join: t -> t -> t (* tehe ⊔ *)
  val meet: t -> t -> t (* tehe ⊓ *)
  val bot: unit -> t (* element ⊥ *)
  val is_bot: t -> bool
  val top: unit -> t (* element ⊤ *)
  val is_top: t -> bool
  val widen: t -> t -> t (* tehe ⨆ *)
  val narrow: t -> t -> t (* tehe ⨅ *)
end
```

Joonis 4. Domeeni (lihtsustatud) signatuur Goblinti moodulis `Lattice`.

3.2 Goblint

Goblint on mitmelõimeliste C programmide staatiline analüsaator, mis keskendub andmejooksude tuvastamisele [9]. Seda tehakse andmevooanalüüsi raamistikus, kasutades keerukamaid spetsiifilisi abstraktseid domeene. Goblinti autoritele on teada, et analüsaator ei käitu alati oodatud viisil, vaid võib teha vigu. Üheks võimalikuks probleemide allikaks on kasutusel olevad abstraktsed domeenid ja täpsemalt nende implementatsioonid. Leidmaks nende vigade allikat, testitigi käesoleva töö raames Goblintis implementeeritud domeenide omadusi.

Goblint ise on kirjutatud OCaml-is ja on avatud lähtekoodiga [10]. Selle omaduspõhiseks testimiseks valiti vastav OCaml-i teek QCheck [11], mis on sarnaste teekide hulgast enim kasutatud ja uuendatud.

Goblintis on domeenid implementeeritud OCaml-i moodulitena, mis muu hulgas sisaldavad domeeni elemendi tüüpi ning neil defineeritud seoseid ja tehteid, nagu näidatud joonisel 4. Domeenide omaduste testimiseks on esmalt vaja defineerida selle elementide generaator. Selleks lisati domeeni signatuuri järgneva signatuuriga funktsioon:

```
val arbitrary: unit -> t QCheck.arbitrary
```

Igas domeenis defineeritav `t QCheck.arbitrary` hõlmab endas nii tüüpi `t` elementide generaatorit (`t QCheck.Gen`) kui ka vähendajat (`t QCheck.Shrink`).

Kõik omadused peatükkidest 2.1 ja 2.2 implementeeriti omaduspõhiste testidena moodulis `DomainProperties`. Näiteks ülemraja kommutatiivsuse test näeb välja järgmiselt:

```
let join_comm = make ~name:"join comm" (pair arb arb)
                  (fun (a, b) -> D.equal (D.join a b) (D.join b a))
```

kus `D` on testitava domeeni moodul ja `arb` on mugavuse mõttes `D.arbitrary ()`.

Testid implementeeriti polümorfselt OCaml-i funktorite abil, mis võimaldab samu omadusi kontrollida mistahes domeenil. See oligi eesmärk, sest kõik need omadused peavad universaalselt kehtima. Domeeni testimiseks peab sellel siiski olema defineeritud `arbitrary`, mida saab teha üksnes manuaalselt ja mis seega on kõige ajamahukam osa Goblinti arvukate domeenide testimise juures. Sellele läheneti kahelt poolt.

3.3 Alt-üles lähenemine

Ühest küljest testiti erinevaid täisarvudega töötavaid domeene moodulist `IntDomain`, mis on üsna sõltumatud ja seetõttu saab neid eraldiseisvalt testida. Paljud teised domeenid kasutavad oma elementidena konkreetsest analüüsitavast programmist leitud muutujaid, funktsioone vms, mis teeb nende testimise raskeks, sest pole selge, kust võtta juhuslikke elemente. Täisarvude abstraktsioonide puhul seda probleemi ei esine.

Täisarvudomeenide korrektsus Täisarve abstraheerivad domeenid omavad Goblintis tavalisetele domeenidele lisaks mitmeid spetsiifilisi funktsioone, nagu näidatud joonisel 5. Olemasolevatele lisaks implementeeriti lihtne täisarvude alamhulkade domeen, mille suhtes testiti ka kõigi täisarvude domeenide korrektsust, kasutades peatükis 2.3 toodud tingimusi.

Hulga abstraheerimiseks kasutati neis domeenides olevat `of_int` funktsiooni, et üksikuid elemente abstraheerida ja seejärel ülemraja leida, mis vastaks kogu vaadeldavale arvuhulgale. See võimaldas testida kõigi tehete, nii ühekohaliste (nt `neg`) kui ka kahekohaliste (nt `add`), korrektsust alamhulkade kaudu defineeritute suhtes.

Kuna abstraktsiooni õigsuse testimiseks on domeenile lisaks vaja teist domeeni, mille abstraheerimist kontrollida, ja vastavaid tehteid mõlemas domeenis, siis sellist testimist üldisel tasandil teostada ei saa. Täisarvude domeenidel tehtu näitas, et selliseid omadusi on siiski võimalik testida, kuigi vajab olulist lisatööd.

Käivitamine Nende domeenide testimiseks loodi moodul `Maindomaintest`, mida saab Goblintist endast eraldi kompileerida (käsuga `make domaintest`) ja käivitada (käsuga `./goblint.domaintest -v`). Selles moodulis on nimekiri domeenidest, mida käivitamisel testitakse.

3.4 Ülalt-alla lähenemine

Teisest küljest testiti domeeni, mida Goblint päriselt etteantud programmi analüüsimisel kasutama hakkaks. See domeen pannakse erinevate seadistuste abil kokku paljudest

```
module type S =
sig
  include Lattice.S (* sisaldab domeeni signatuuri, vt joonis 4 *)
  val of_int: int64 -> t (* funktsioon α ühe arvu jaoks *)
  val neg: t -> t (* vastandarvu tehe, unaarne − *)
  val add: t -> t -> t (* liitmistehe, binaarne + *)
  val sub: t -> t -> t (* lahutamistehe, binaarne − *)
  (* ..., veel tehteid *)
end
```

Joonis 5. Täisarve abstraheeriva domeeni (lihtsustatud) signatuur Goblinti moodulis `IntDomain`.

domeenidest, mis oma osadena kasutavad teisi lihtsamaid domeene (nagu seda teeb kujutusdomeen), mis omakorda võivad olla veelgi lihtsamatest domeenidest komponeeritud. Sellised sõltuvusi arvestades saab analüüsiks kasutatavat domeeni ette kujutada kui puud, mille lehtedeks on lihtsad sõltuvusteta domeenid nagu eelnevalt testitud täisarvude domeenid. Kui alt-üles lähenemise puhul testiti sellise puu lehti, siis ülalt-alla lähenemises testitakse domeene alustades juurest.

Juurdomeeni elementide genereerimiseks oleks põhimõtteliselt vaja genereerida elemente kõigist teistest domeenidest, millest sõltutakse. See eeldab korraga kõigi Goblinti domeenide jaoks generaatorite lisamist, mis lisaks suurele töömahule võib olla ka ebaotstarbekas. Kui mingite alamdomeenide generaatorid defineerida triviaalselt, näiteks alati $\bot$ genereerimise teel, siis on võimalik tervet domeeni testida, ilma et oleks vaja kõiki generaatoreid lisada, sest mingite alamdomeenide elemente pole vaja genereerida. Selliselt testitigi domeenide hierarhia ülemist osa.

Mingi näide kõrgemat järku domeenist ja selle sõltuvustest/alamdomeenidest?

Käivitamine Selliselt testimiseks lisati Goblinti käsurealippude hulka uus nimega `dbg.test.domain`. Kui see on sisse lülitatud, siis enne etteantud programmi analüüsimist jooksutatakse analüüsiks kasutataval domeenil samad testid. Goblinti näidiskäivitus koos selle lipuga toimub järgneva käsuga:

```
./goblint --enable dbg.test.domain \
    ./tests/regression/04-mutex/01-simple_rc.c
```

Goblinti domeenide hierarhia osa, millele lisati generaatorid ja mida seeläbi sai testida, on toodud lisas II.

Tabel 4. Domeenide erinevatel meetoditel testimise tulemused.

Lähenemine	Võrdlemine	Testide arv		
		Kokku	Erindi teke	Mittekehtivad
Alt-üles	`equal`	468	~35	~75
	`leq`	468	~35	~69
Ülalt-alla	`equal`	27	0	~12
	`leq`	27	0	~12

Uuenda kui muutunud

3.5 Tulemused

Enamike omaduste testimisel on vaja kontrollida mingite domeeni elementide võrdumist. Esimese loomuliku variandina kasutati selleks domeenide `equal` funktsiooni (vt joonis 4). Kuna pole kindel, et kõigis Goblinti domeenides `equal` on defineeritud võre struktuuriga kooskõlaliselt, siis viidi samad testid läbi ka teisiti: testimisel kasutati elementide `a` ja `b` võrdumise tingimusena `leq a b && leq b a`. Eeliseks on see, et selline võrdlus ei eelda, et `equal` funktsioon oleks kooskõlaliselt defineeritud, aga puuduseks see, et eeldatakse, et järjestus on korrektne. Kuna tegu on kompromissiga, siis viidi samad testid läbi mõlema võrdlusmetoodikaga, ja tulemused on toodud tabelis 4.

Mõnede omaduste testimisel tekkisid erindid enne kui kehtivust kontrollida oli võimalik. Enamus juhtudel seetõttu, et domeen ei moodusta täielikku võre ning elementide $\bot$ ja $\top$ asemel antakse erind. Sel põhjusel lõpevad erindiga domeeni `Integers` testid, mis kontrollivad vähima ja suurima elemendiga seotud omadusi.

Omaduste mittekehtimise põhjusteks võis olla:

1. Omaduse mittekehtimine teoreetilisel tasandil. Sel põhjusel ei kehti domeeni `Trier` ülemraja assotsiatiivsus.
2. Omaduse dokumenteeritud mittekehtimine. Sel põhjusel tõenäoliselt ei kehti domeeni `CircInterval` rajade omadused, sest neid on nimetatud „pseudo-ülemrajaks“ ja „pseudo-alamrajaks“.
3. Omaduse sõltumine teistest mittekehtivatest omadustest samal või domeeni osaks oleval teisel domeenil. Sel põhjusel ei kehti domeeni `IntDomTuple` paljud omadused, kuna need ei kehti domeenidel, millest see koosneb.
4. Viga domeeni implementeerimisel.

Kuna töö eesmärgiks polnud vigade parandamine, siis põhjustesse oluliselt ei süvenetud, mistõttu ei saa kindlalt kõigi ebaõnnestunud testide kohta midagi väita. See nõuab omaette teadmisi konkreetsete domeenide ettenähtud tööpõhimõtetest ja tööd täpsete kontranäidete analüüsimiseks. Sellele vaatamata annavad läbiviidud testid suuna, kus tõenäoliselt probleemid võivad olla.

Kokkuvõte

Töö teoreetilise osana koostati ülevaatlik abstraktsete domeenide omaduste komplekt, mis ühest küljest on piisavalt üldine, et kehtida mistahes domeenil, ja teisest küljest on piisavalt täielik, et katta kõik, mida nii üldisel tasemel võimalik kontrollida. Sinna kuuluvad täielike võrede, programmianalüüsi spetsiifilised ja abstraktsiooni korrektsuse omadused.

Töö praktilise osana lisati Goblint analüsaatorile, täpsemalt selle domeenidele, omaduspõhise testimise võimekus, kasutades teeki QCheck. Implementeeriti üldrakendataval viisil kõigi omaduste testimine, lähenedes domeenide testimisele kahest suunast. Lõpuks viidi läbimine testimine ise.

Goblinti domeenide omaduspõhise testimise tulemusena leiti mitmeid domeene, millel mingid omadused ei kehtinud. Kuna töö eesmärgiks polnud leitud vigade parandamine, siis kõigi põhjused pole täpselt teada, kuid on olemas selged kohad, mis probleemide allikaks võivad olla.

Selle töö raames siiski ei täiendatud absoluutselt kõiki Goblinti domeene nii, et neid sisukalt testida saaks, mis tähendab, et tööd on võimalik jätkata, et kontrollitud oleks rohkemad domeenid. Lisaks on muidugi vaja põhjalikult analüüsida neid domeene, millest vigu leiti, et aru saada vigade täpsest sisust, ja lõpuks vead kõrvaldada. Viimane võib osutuda üpris keeruliseks, kui viga pole põhjustatud halvast implementatsioonist, vaid mittekehtivast teooriast selle taga.

Kuhu ja kuidas lisada oma Goblinti täiendused? GitHub link *fork*-ile? Selle lähtekood eraldi lisana lõputöö juurde?

Viidatud kirjandus

[1] Vojdani V. Mitmelõimeliste C-programmide kraasimine analüsaatoriga Goblin. Magistritöö. TÜ arvutiteaduse instituut, 2006. http://dspace.ut.ee/handle/10062/1253 (27. 02. 2018).

[2] Seidl H., Wilhelm R., and Hack S. Foundations and Intraprocedural Optimization. *Compiler Design: Analysis and Transformation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–114. https://doi.org/10.1007/978-3-642-17548-0_1 (02/27/2018).

[3] ISO/IEC. ISO International Standard ISO/IEC 9899:2011. Information technology – Programming languages – C. N1570 Commitee Draft. Apr. 12, 2011. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf (03/30/2018).

[4] Laan V. Võreteooria. Loengukonspekt. TÜ. Sügis 2017. https://courses.ms.ut.ee/MTMM.00.039/2017_fall/uploads/Main/kon.pdf (27. 02. 2018).

[5] Davey B. A. and Priestley H. A. Introduction to Lattices and Order. 2nd ed. Cambridge: Cambridge University Press, 2002.

[6] Might M. Order theory for computer scientists. http://matt.might.net/articles/partial-orders (03/03/2018).

[7] Cousot P. and Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252. http://www.di.ens.fr/~cousot/publications.www/CousotCousot-POPL-77-ACM-p238--252-1977.pdf (03/25/2018).

[8] Claessen K. and Hughes J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. http://doi.acm.org/10.1145/351240.351266 (04/23/2018).

[9] Vojdani V. et al. Static race detection for device drivers: the Goblint approach. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. ACM, 2016, pp. 391–402. http://goblint.in.tum.de (04/16/2018).

[10] Goblint. Lähtekood. https://github.com/goblint/analyzer (16. 04. 2018).

[11] QCheck. Lähtekood. https://github.com/c-cube/qcheck (16. 04. 2018).

Eemalda nocite

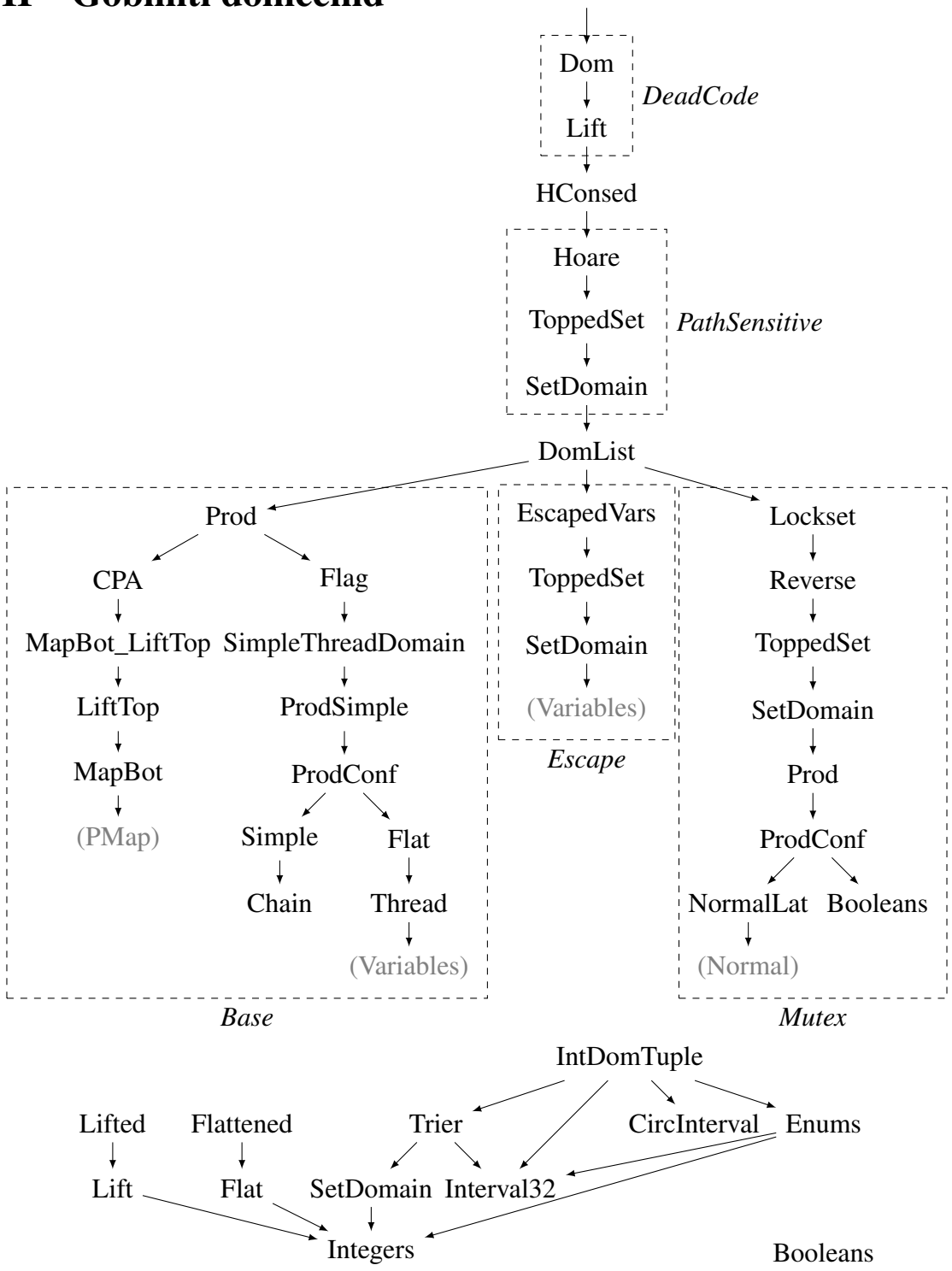
Lisad

I Tsüklita näiteprogrammi iteratiivne analüüs

Tabel. Näiteprogrammi (joonisel 1) analüüsi süsteemi iteratiivse lahendamise sammud ja lahend.

i	1	2			3			4			5			6		
		x	y	z	x	y	z	x	y	z	x	y	z	x	y	z
x_1	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
x_2	$\bot$	$\{0,1,2\}$	$\top$	$\top$	$\{0,1,2\}$	$\top$	$\top$	$\{0,1,2\}$	$\top$	$\top$	$\{0,1,2\}$	$\top$	$\top$	$\{0,1,2\}$	$\top$	$\top$
x_3	$\bot$		$\bot$		$\{0\}$	$\top$	$\top$	$\{0\}$	$\top$	$\top$	$\{0\}$	$\top$	$\top$	$\{0\}$	$\top$	$\top$
x_4	$\bot$		$\bot$		$\{1,2\}$	$\top$	$\top$	$\{1,2\}$	$\top$	$\top$	$\{1,2\}$	$\top$	$\top$	$\{1,2\}$	$\top$	$\top$
x_5	$\bot$		$\bot$		$\bot$			$\{0\}$	$\{5\}$	$\top$	$\{0\}$	$\{5\}$	$\top$	$\{0\}$	$\{5\}$	$\top$
x_6	$\bot$		$\bot$		$\bot$			$\{1,2\}$	$\{2,3\}$	$\top$	$\{1,2\}$	$\{2,3\}$	$\top$	$\{1,2\}$	$\{2,3\}$	$\top$
x_7	$\bot$		$\bot$		$\bot$			$\bot$			$\{0,1,2\}$	$\{2,3,5\}$	$\top$	$\{0,1,2\}$	$\{2,3,5\}$	$\top$
x_8	$\bot$		$\bot$		$\bot$			$\bot$			$\bot$			$\{0,1,2\}$	$\{2,3,5\}$	$\{4,6,10\}$

II Goblinti domeenid



III Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Simmo Saan**,

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

   Abstraktsete domeenide omaduspõhine testimine

   mille juhendajad on Vesal Vojdani ja Kalmer Apinis

   1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

   1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, pp.kk.aaaa

Kuupäev