

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:

Staatilise programmianalüüsiga uuritakse programme lähtekoodi põhjal, ilma neid käivitamata. Andmevooanalüüsiga määratakse programmi võimalikud seisundid, mida vaadeldakse ligikaudselt ja mis moodustavad abstraktse domeeni. Teostatav analüüs on abstraktse interpretatsiooni teooria kohaselt korrektne (ingl. *sound*), kui kasutatav domeen rahuldab teatud matemaatilisi omadusi. Staatilise analüsaatori implementeerimisel võib juhtuda, et domeenides esineb vigu, mis rikuvad analüüsi ja selle korrektsuse. Töös koostatakse omaduste komplekt, mida kasutatakse Goblint analüsaatorist omaduspõhise testimise abil vigade leidmiseks. Selleks implementeeritakse Goblintis vajalik domeenide testimise raamistik ja elementide generaatorid. Lõpuks viiakse läbi testimine ja esitatakse leitud vead. Sellega näidatakse, et omaduspõhist testimist on võimalik efektiivselt rakendada abstraktsetest domeenidest vigade leidmiseks.

Võtmesõnad:

staatiline analüüs, andmevooanalüüs, abstraktne interpretatsioon, võred, Goblint

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Property-based Testing of Abstract Domains

Abstract:

Static program analysis studies programs based on their source code, without executing them. Data-flow analysis determines a program's possible states, which are approximated and which make up an abstract domain. The analysis is sound according to the theory of abstract interpretation, if the domain satisfies certain mathematical properties. When implementing a static analyzer it may happen that the domains contain bugs, which ruin the analysis and its soundness. Here a set of properties is compiled, which are used to find bugs from the Goblint analyzer via property-based testing. For this the necessary domain testing framework and element generators are implemented in Goblint. Finally the testing is conducted and found issues presented. This shows that property-based testing can effectively be used to find bugs from abstract domains.

Keywords:

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

CERCS:

P170 Computer science, numerical analysis, systems, control

Sisukord

1	Sissejuhatus	5
2	Andmevooanalüüs	7
2.1	Analüüsi näide	7
2.2	Võred	8
2.3	Alamhulkade domeen	9
2.4	Kujutusdomeen	10
2.5	Algoritmiline analüüs	11
2.6	Intervalldomeen	13
3	Domeeni omadused	16
3.1	Võre omadused	16
3.2	Laiendamine ja kitsendamine	17
3.3	Abstraktsiooni korrektsus	19
4	Omaduspõhine testimine	21
4.1	Põhimõtted	21
4.2	Goblint	22
4.3	Alt-üles lähenemine	23
4.4	Ülalt-alla lähenemine	24
4.5	Raskused testimisel	25
4.6	Tulemused	26
5	Kokkuvõte	30
	Viidatud kirjandus	31
	Lisad	32
I	Tsüklita näiteprogrammi iteratiivne analüüs	32
II	Lähtekood	33
III	Testitud Goblinti domeenid	34
IV	Litsents	35

Unsolved issues

\sqcup, \sqcap, \top	10
Üksik lause kuhugi mujale	11
Täpsemalt defineerida üleminekufunktsioonide ülemraja?	12
Otsekorrutise domeeni eraldi defineerimine?	12
Ülejäänud domeenid ka definitsiooni kujul?	14

1 Sissejuhatus

Üks võimalus arvutiprogrammi käitumise uurimiseks on dünaamiline analüüs: programm käivitatakse ja huvitav olukord kutsutakse esile. Teine võimalus on staatiline analüüs: programmi ei käivitata, vaid selle käitumise kohta tehakse järeldusi lähtekoodi põhjal. Staatilist programmianalüüsi teostatakse mitmel põhjusel:

1. Kompilaatorid analüüsivad kompileeritavaid programme, et neid automaatselt optimeerida. Näiteks asendatakse korduvalt esinevaid avaldise abimuutujatega, vältides sama avaldise mõttetut taasväärtustamist (ingl. *common subexpression elimination*). Seejuures on oluline, et programmi käitumine optimeerimise käigus ei muutuks.
2. Mõned programmide vead esinevad väga spetsiifilistel juhtudel või on raskesti esilekutsutavad (nt mitmelõimelisusega seotud vead), mistõttu dünaamilise analüüsiga on neid keeruline avastada. Näiteks FindBugs^{TM1} otsib Java programmide vigu.
3. Programmi korrektsuse näitamisel, tõestamaks, et selles teatud tüüpi vigu kindlasti ei esine. Näiteks Astrée² võimaldab tõestada, et C programmis puuduvad määramata ja implementatsioonispetsiifilisest käitumisest põhjustatud täitmisaegsed vead.

Andmevooanalüüs (ingl. *data-flow analysis*) on üks intuitiivne meetod staatilise programmianalüüsi teostamiseks. Selle keskseks ideeks on võimalike programmi seisundite määramine selle programmi igal täitmise sammul. 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 kättesaadavate omistamiste hulk (ingl. *available assignments*) [2:12].

Üldiselt pole võimalik staatilise analüüsiga alati täpselt määrata programmi seisundit, sest see oleks samaväärne programmi käivitamisega. Seetõttu vaadeldakse ligikaudseid seisundeid, mille uurimist nimetatakse abstraktseks interpretatsiooniks (ingl. *abstract interpretation*) ja see põhineb rangelt 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.

¹<http://findbugs.sourceforge.net/>

²<http://www.astree.ens.fr/>

Ligikaudsed seisundid, mis vastavad konkreetsetele seisunditele, moodustavad **abstraktse domeeni**. Näiteks võib arvuhulkade asemel vaadelda intervalle või uurida lineaarseid piire muutujate vahel (polüeedri domeen). Abstraktse interpretatsiooni õigeks toimimiseks on vajalik, et abstraktne domeen 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 analüüsides tulemused usaldusväärsed ja korrektsed.

Goblint on mitmelõimeliste C programmide staatiline analüsaator, mille eesmärk on tõestada, et programmis puuduvad mitmelõimelisusega seotud vead, kasutades selleks andmevooanalüüsi. Goblinti autoritele on teada, et analüsaator ei käitu alati oodatud viisil, vaid võib teha vigu, mis võivad rikkuda analüüsi korrektsuse.

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

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 Goblinit ja selle domeene nii, et omaduspõhist testimist oleks võimalik teostada, implementeeriti omaduste testid ja viidi läbi testimine.

2 Andmevooanalüüs

Andmevooanalüüsi ja abstraktse interpretatsiooni põhimõtete selgitamiseks parim viis on illustreeriva näite läbitegemine. Seejärel samm-sammult formaliseeritakse tehtu matemaatiliselt, kuna sissejuhatuses antud informaalne definitsioon on ebapiisav mingisuguse teooria arendamiseks.

2.1 Analüüsi näide

Selleks olgu edaspidi vaatluse all programm jooniselt 1, kus on kõrvuti C-keelse lähetekoodi 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 kindlasti $x \in \{0, 1, 2\}$ ³. Lisaks on vahepeal deklareeritud väärtustamata muutuja y , millel C keele semantikas vaikeväärtust pole ja seetõttu $y \in \mathbb{Z}$ [3].
- Punktis 3, kus tingimus oli tõene, $x = 0$.
- Punktis 4, kus tingimus oli väär, saab eelnevast hulgast välistatud nulli eemaldamisel öelda, et $x \in \{1, 2\}$.
- Punktis 5 on täpselt teada, et $y = 5$. Lisaks on selles harus jätkuvalt $x = 0$.
- Punktis 6, teades võimalikke muutuja x väärtusi, saab öelda, et liitmistehte tulemusena $y \in \{2, 3\}$.
- Punktis 7, kus kaks võimalikku tingimuslause haru uuesti kokku saavad, tuleb arvestada mõlema haru võimalike väärtustega, seega muutuja $y \in \{2, 3, 5\}$.
- Punktis 8, teades võimalikke muutuja y väärtusi, saab öelda, et korrutamistehte tulemusena $z \in \{4, 6, 10\}$.

Tehtud analüüs on võimalikult täpne, mis tähendab, et igas punktis iga muutujaga on seostatud väikseim väärtuste hulk. Iseenesest poleks vale mõnes programmi punktis mõne muutujaga seostada suuremat võimalike väärtuste hulka, kuid sellisest analüüsist

³C keele semantika on siinkohal keerukas, kuid (üldiselt võimalikud) negatiivsed jäägid on antud juhul välistatud, sest `rand()` funktsioon ei tagasta negatiivseid arve.

```

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.

oleks vähem kasu, sest see tooks sisse ebavajalikku ebatäpsust. Järgnevalt ongi eesmärk matemaatiliselt formaliseerida sellise võimalikult täpse analüüsi teostamine, mis omakorda oleks aluseks analüüsi teoreetiliseks uurimiseks ja automatiseerimiseks.

2.2 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, 1]:

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 $\sqcup X$ ja $\sqcap X$. 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 tärpsimat ühendatud seisundit, mis ongi ühendatavate seisundite ülemine raja.

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

Täielikus võres (A, \sqsubseteq) leidub vähim element $\perp = \sqcap A$ ja suurim element $\top = \sqcup A$, mida nimetatakse vastavalt *bottom*'iks ja *top*'iks. Tegelikult tuleneb täielikes võredes kõik alumise rajaga seonduv automaatselt ülemisest rajast [2:18].

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 \perp kirjeldab kokkuleppeliselt võimatut seisundit, milleks tüüpiliselt on saavutamatu (ingl. *unreachable*) programmi punkt ehk punkt, kuhu programmi täitmisel mitte kunagi ei jõutagi. Edaspidi tähistagu \mathbb{D} mingit parasjagu vaadeldavat domeeni, millega käib ilmutamata kaasas täielik võre $(\mathbb{D}, \sqsubseteq)$.

2.3 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 [4:6], kus

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

Ülal käsitsi tehtud joonise 1 programmi analüüsis oligi iga muutuja väärtusi analüüsitud alamhulkade domeeni $\mathbb{D} = \mathcal{P}(\mathbb{Z})$ 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 tehmed:

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

Abstraktne aritmeetika Lause $y = x + 1$ abstraktseks teostamiseks 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.

2.4 Kujutusdomeen

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

$$\mathbb{D} = (\text{Var} \rightarrow \text{Val})_{\perp} = (\text{Var} \rightarrow \text{Val}) \cup \{\perp\}.$$

Domeeni elementideks on kujutused muutujate hulgast nende abstraktsete väärtuste hulka, mis on väga analoogilised konkreetsete muutujate väärtustustega funktsionaalselt kirjeldatuna. Element \perp on tehiskult lisatud, et domeen moodustaks täieliku võre, ja tähendab saavutamatu programmi punkti. Osaline järjestus domeenis \mathbb{D} on iga $D_1, D_2 \in \mathbb{D}$ korral defineeritud kui

$$D_1 \sqsubseteq D_2 \iff D_1 = \perp \vee \forall x \in \text{Var}. D_1(x) \sqsubseteq D_2(x).$$

\sqcup, \sqcap, \top ?

Joonisel 1 tehtud näites olid muutujad $\text{Var} = \{x, y, z\}$ ja väärtused domeenis $\text{Val} = \mathcal{P}(\mathbb{Z})$. 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*) [1] ja samas näites võivad need olla järgnevad:

$$\begin{aligned} tf_{1,2}(d) &= d[x \mapsto \{0, 1, 2\}][y \mapsto \mathbb{Z}], \\ tf_{2,3}(d) &= d[x \mapsto d(x) \cap \{0\}], & tf_{2,4}(d) &= d[x \mapsto d(x) \setminus \{0\}], \\ tf_{3,5}(d) &= d[y \mapsto \{5\}], & tf_{4,6}(d) &= d[y \mapsto d(x) + \{1\}], \\ tf_{7,8}(d) &= d[z \mapsto \{2\} \cdot d(y)], \end{aligned}$$

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

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

kus kirjutis $d[x \mapsto v]$ tähendab funktsiooni uuendamist (ingl. *function update*) ehk funktsiooni, mis argumentil x omab väärtust v ja ülejäänud argumentidel sama väärtust, mis funktsioon d . Aritmeetilised tehted hulkade vahel on sellised nagu nad on sisemises domeenis Val . Seejuures üleminekufunktsioonid saab keele semantika ning lausete ja avaldiste struktuuri mallide järgi mehaaniliselt defineerida ehk konkreetse programmi analüüsil toimub see automatiseeritult.

Kujutusdomeeni kasutamisega saab lihtsama domeeni, mis kirjeldab ühte muutujat korraga, laiendada kõigile muutujatele korraga.

Üksik lause kuhugi mujale

2.5 Algoritmiline analüüs

Võrede, nende abstraktsete tehete ja üleminekufunktsioonide abil on formaliseeritud suur osa esialgses näites intuiitiivselt 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 [2]:

1. Iga programmi punkti i jaoks võtta kasutusele üks muutuja x_i , mis vastab otsitavale programmi abstraktsele seisundile selles punktis.
2. Iga juhtimisvoograafi serva $k \rightarrow i$ jaoks defineerida fikseeritud protsessi abil üleminekufunktsioon $tf_{k,i}$. Nõnda saab iga serva jaoks moodustada võrratuse $x_i \sqsubseteq tf_{k,i}(x_k)$. 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 $x_1 \sqsupseteq \iota$, kus ι kirjeldab algseisundit.

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

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

kus f_1, \dots, 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 punktiivisiliselt.

Otsekorrutise domeeni eraldi defineerimine?

Sellises võres on muutujaks $\bar{x} = (x_1, \dots, x_n)$ ning funktsioonid ühendatud kokku ühte funktsiooni $\bar{f}(\bar{x}) = (f_1(\bar{x}), \dots, f_n(\bar{x}))$. 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 monotoonused, 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 $f : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ nimetatakse **monotoonseks**, kui iga $a, b \in \mathbb{D}_1$ korral

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

4. Saadud võrratusele (ehk kaudselt algele 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. Seda on lõige lihtsam leida Kleene püsipunktialgoritmi abil, mis alustab väärtusest $\bar{x}_0 = \perp = (\perp, \dots, \perp)$ 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:

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

kus algseisund ι on konstantne kujutus väärtustega \emptyset . Nende iteratiivne lahendamine on samm-sammult toodud lisas I. On näha, et algoritmiline analüüs jõudis oodatud tulemuseni. Nagu näha, siis Kleene algoritm 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* ja *worklist* algoritmid [2:24,83].

Sellel põhjal 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 ülimalt 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 abstraherimise uurimisel.

2.6 Intervalldomeen

Intervalldomeen on domeen, milles täisarvude väärtuste abstraherimiseks 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

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

Ülejäänud domeenid ka definitsiooni kujul?

Sellises domeenis

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

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.

Analoogiliselt alamhulkade domeeniga, tuuakse siin programmide analüüsimiseks sisse:

Konstantide abstraherimine 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:

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

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

Valides eelnevas joonisel 1 tehtud näites seekord $\text{Val} = \mathbb{I}_\perp$, 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	\perp	\perp	\perp
x_2	$[0, 2]$	$[-\infty, +\infty]$	\perp
x_3	$[0, 0]$	$[-\infty, +\infty]$	\perp
x_4	$[1, 2]$	$[-\infty, +\infty]$	\perp
x_5	$[0, 0]$	$[5, 5]$	\perp
x_6	$[1, 2]$	$[2, 3]$	\perp
x_7	$[0, 2]$	$[2, 5]$	\perp
x_8	$[0, 2]$	$[2, 5]$	$[4, 10]$

3 Domeeni omadused

Selles peatükis tuuakse üldised abstraktsete domeenide omadused, mis kataks võimalikult palju universaalselt kontrollitavaid aspekte. Nende hulka kuuluvad täielike võrede, programmianalüüsi spetsiifilised ja abstraktsiooni korrektsuse omadused.

3.1 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 (definiitsioonist 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 $b \sqsubseteq a \sqcup b$ (definiitsioonist 3);
- $a \sqcap b \sqsubseteq a$ ja $a \sqcap b \sqsubseteq b$ (definiitsioonist 4);
- kui $a \sqsubseteq c$ ja $b \sqsubseteq c$, siis $a \sqcup b \sqsubseteq c$ (definiitsioonist 3);
- kui $c \sqsubseteq a$ ja $c \sqsubseteq b$, siis $c \sqsubseteq a \sqcap b$ (definiitsioonist 4);

Rajade tehete omadused [4:6, 5:39]

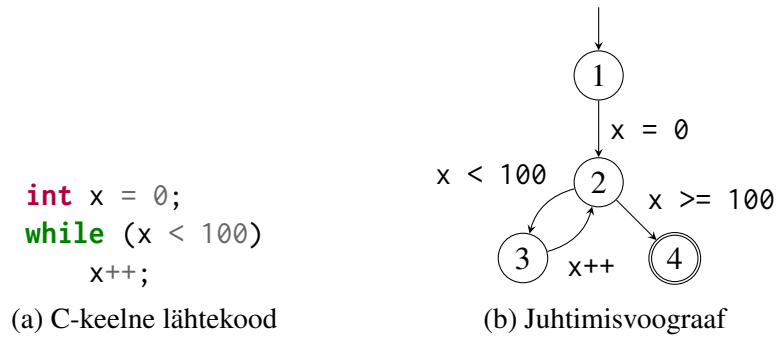
- $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ ja $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$ (assotsiatiivsus);
- $a \sqcup b = b \sqcup a$ ja $a \sqcap b = b \sqcap a$ (kommutatiivsus);
- $a \sqcup a = a$ ja $a \sqcap a = a$ (idempotentsus);
- $a \sqcup (a \sqcap b) = a$ ja $a \sqcap (a \sqcup b) = a$ (neelduvus);

Vähima ja suurima elemendi omadused [6]

- $\perp \sqsubseteq a$;
- $a \sqsubseteq \top$;
- $a \sqcup \perp = a$;
- $a \sqcap \top = a$;

Järjestuse ja rajade tehete seosed Järgnevad on ekvivalentsed [5:39]:

1. $a \sqsubseteq b$,
2. $a \sqcup b = b$,
3. $a \sqcap b = a$.



Joonis 2. Tsükliga programmi näidis.

3.2 Laiendamine ja kitsendamine

Kui joonisel 1 olev programm on tsükliteta ja selle analüüsi saaks teostada ka ilma võrratuste süsteemi püsipunkti leidmata, siis üldiselt sama lihtsalt ei saa. Huvitavamad programmid sisaldavad tsükleid, mille analüüsimiseks võib püsipunkti leidmisel põhinev algoritm siiski kasulik olla. Olgu nüüd vaatluse all tsükliga programm joonisel 2 ja domeen $\mathbb{D} = (\text{Var} \rightarrow \mathbb{I}_\perp)_\perp$ nagu viimati, siis võrratuste ja võrrandite süsteemid on järgnevad:

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

kus algseisund ι on konstantne kujutus väärtustega \perp . Nende vähim püsipunkt (tabelis 3a) leitakse Kleene püsipunktialgoritmi 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. Keerulisemate programmide puhul on isegi võimalik, et analüüs ei termineeru [2:60].

Laiendamine Probleemi põhjuseks on asjaolu, et intervalldomeen ei rahulda kasvavate jadade stabiliseerumise tingimust (ingl. *ascending chains condition*), sest selles leiduvad lõpmatud rangelt kasvavad ahelad [2:56], näiteks

$$[0, 0] \sqsubset [0, 1] \sqsubset [0, 2] \sqsubset \dots \sqsubset [0, +\infty],$$

kus \sqsubset on vastav range järjestuse seos ehk kirjutis $a \sqsubset b$ tähendab, et $a \sqsubseteq b$ ja $a \neq b$ [4:3].

Püsipunkti leidmise igal iteratsioonil tsüklis muutuja väärtuste lõik suureneb ühe võrra seetõttu stabiliseerumiseks läheb soovitus rohkem iteratsioone. Probleemi lahendamiseks muudetakse analüüsimeetod nii, et püsipunkti leidmine toimiks kiirendatult, tuues meelega sisse täiendava ebatäpsuse. Sellist võtet nimetatakse laiendamiseks (ingl. *widening*).

Lahendatav võrratus $\bar{x} \sqsubseteq \bar{f}(\bar{x})$ kirjutatakse samaväärsel akumulatsioonikujul $\bar{x} = \bar{x} \sqcup \bar{f}(\bar{x})$, 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 [2:62]:

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

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

Intervallide laiendamise idee seisneb selles, et intervall suurenedes suureneb kohe vastavas suunas lõpmatusse, garanteerides, et intervall saab suurenedes ülimalt kaks korda, mis muudab rangelt kasvavad ahelad lõplikeks. Sellise modifikatsiooniga süsteemi vähim püsipunkt (tabelis 3b) leitakse juba Kleene algoritmi 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 kahanevate jadade stabiliseerumise tingimusega (ingl. *descending chains condition*) ja lõpmatute rangelt kahanevate ahelatega [2:65], 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 [2:66]:

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

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

Intervallide kitsendamise idee seisneb selles, et intervall vähenedeks ainult lõpmatusest, garanteerides, et intervall saab väheneda ülimalt kaks korda, mis muudab rangelt kahanevad ahelad lõplikeks. Uuritud näitel kiirendatud kitsendamine jõuab sama tulemuseni

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

(a) Iteratsiooni lahend		(b) Laiendamise tulemus		(c) Kitsendamise tulemus	
	x		x		x
x_1	\perp	x_1	\perp	x_1	\perp
x_2	$[0, 100]$	x_2	$[0, +\infty]$	x_2	$[0, 100]$
x_3	$[0, 99]$	x_3	$[0, +\infty]$	x_3	$[0, 99]$
x_4	$[100, 100]$	x_4	$[100, +\infty]$	x_4	$[100, 100]$

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:

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

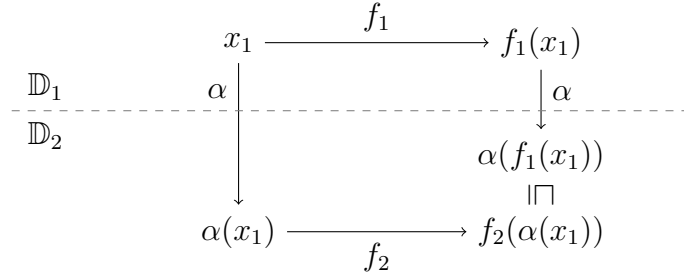
Kusjuures need operaatorid pole tingimata assotsiatiivsed, kommutatiivsed, jne nagu nende algsed analoogid. Neid kasutatakse täpselt nii nagu ülal toodud, kus vasak operand on vana seisund ja parem operand on uus seisund. Sellel asjaolul põhineb nende defineerimine ja seda ka nähtud intervalldomeeni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

3.3 Abstraktsiooni korrektsus

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 $\alpha : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ ja $\gamma : \mathbb{D}_2 \rightarrow \mathbb{D}_1$, 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 .



Joonis 3. Definitsiooni 8 tingimust 5 illustreeriv skeem.

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.

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 konkreetsetel 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 laiemalt võreteoorias [7].

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

$$\begin{aligned}
\alpha(x_1) &= \begin{cases} \perp, & \text{kui } x_1 = \emptyset, \\ [\min x_1, \max x_1], & \text{muidu;} \end{cases} \\
\gamma(x_2) &= \begin{cases} \emptyset, & \text{kui } x_2 = \perp, \\ \{l, \dots, u\}, & \text{kui } x_2 = [l, u]. \end{cases}
\end{aligned}$$

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.

4 Omaduspõhine testimine

Omaduspõhine testimine (ingl. *property-based testing*) on programmide testimise meetodika, mis põhineb soovitud omaduste verifitseerimisel juhuslikult genereeritud testandmetel. Selle lähenemise populariseeris Haskellis 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 väljamõttlemist. See võimaldab samu omadusi kontrollida kõigil domeenidel, sõltumata isegi nende elementide tüübist.

4.1 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, järjestite 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 järjest jne).

Juhuslikult genereeritud argumendid võivad olla „suured“: suured arvud, pikad sõned või järjestid, 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 järjestist 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 vähendamise 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  $\sqsubseteq$  *)
  val join: t -> t -> t (* tehe  $\sqcup$  *)
  val meet: t -> t -> t (* tehe  $\sqcap$  *)
  val bot: unit -> t (* element  $\perp$  *)
  val is_bot: t -> bool
  val top: unit -> t (* element  $\top$  *)
  val is_top: t -> bool
  val widen: t -> t -> t (* tehe  $\sqsubseteq$  *)
  val narrow: t -> t -> t (* tehe  $\sqcap$  *)
end

```

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

4.2 Goblint

Goblint on mitmelõimeliste C programmide staatiline analüsaator, mille eesmärk on tõestada, et programmis puuduvad mitmelõimelisusega seotud vead (nt andmejoosud) [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. Kõigi töö raames tehtud muudatuste ja täienduse lähtekoodi leiab lisast II.

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 funktsioon arbitrary signatuuriga:

```

val arbitrary: unit -> t QCheck.arbitrary

```

mille ainsaks argumendiks on ühiktüübi väärtus () ja mis tagastab t QCheck.arbitrary tüüpi väärtuse. Viimane hõlmab endas tüüpi t domeeni elementide generaatorit (t QCheck.Gen) ja vähendajat (t QCheck.Shrink) nii, nagu QCheck teek seda ette näeb.

Kõik omadused peatükkidest 3.1 ja 3.2 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 polümorfiselt 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 ning lõpuks sobivate generaatoritega varustatud domeenid on toodud lisas III.

4.3 Alt-üles lähenemine

Esiteks testiti erinevaid täisarvudega töötavaid domeene moodulist IntDomain, mis on üsna sõltumatud ja seetõttu saab neid eraldiseisvalt testida. Täpse loetelu leiab lisa III allosast. Paljud teised domeenid kasutavad oma elementidena konkreetsest analüüsita-vast programmist leitud muutujaid, funktsioone vms, mis teeb nende testimise raskeks, sest pole selge, kuidas korrektselt genereerida neid juhuslikke elemente. Täisarvude abstraktsioonide puhul seda probleemi ei esine.

Täisarvudomeenide korrektsus Täisarve abstraherivad domeenid omavad Goblintis tavalistele domeenidele lisaks mitmeid spetsiifilisi funktsioone, nagu näidatud joonisel 5. Olemasolevatele lisaks implementeeriti lihtne täisarvude alamhulkade domeen IntegerSet, mille suhtes testiti ka kõigi täisarvude domeenide korrektsust, kasutades peatükis 3.3 toodud tingimusi.

Hulga abstraherimiseks kasutati neis domeenides olevat of_int funktsiooni, et üksikuid elemente abstraherida ja seejärel ülemraja leida, mis vastaks kogu vaadeldavale arvuhulgale. Lisaks kasutatakse testitava domeeni elementi \perp tühihulga abstraktsioonina. Seejuures on eeldatud, et ülemraja on korrektselt defineeritud. See võimaldas testida kõigi tehete, nii ühekojaliste (nt neg) kui ka kahekojaliste (nt add), korrektsust alamhulkade kaudu defineeritute suhtes.

Kuna abstraktsiooni õigsuse testimiseks on domeenile lisaks vaja teist domeeni, mille abstraherimist 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 Maingroup, mida saab Goblintist endast eraldi kompileerida (käsuga make domainstest) ja käivitada (käsuga ./goblint.domainstest -v). Selles moodulis on järjend domeenidest, mida käivitamisel eraldiseisvalt testitakse.

```

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

```

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

4.4 Ülalt-alla lähenemine

Teiseks testiti tervet domeeni, mida Goblint päriselt etteantud programmi analüüsimisel standardseadistuses kasutama hakkaks. See domeen pannakse sõltuvalt seadistusest kokku mitme eri analüüsi (Base, Escape, Mutex) domeenidest, mis on omakorda moodustatud paljudest domeenidest. Sellised sõltuvusi arvestades saab analüüsiks kasutatavat domeeni ette kujutada kui puud või sõltuvusgraafi (ingl. *dependency graph*). Seda ongi tehtud lisas III, mille ülaosas on kujutatud neid tervikliku domeeni komponente, mida testiti.

Tervikliku domeeni elementide genereerimiseks oleks kaudselt vaja genereerida elemente kõigist teistest domeenidest, millest sõltutakse. See eeldab korraga kõigi Goblinti domeenide jaoks generaatorite lisamist, mis on väga töömahukas. Kui mingite domeenide generaatorid defineerida triviaalselt, näiteks alati \perp genereerimise teel, siis on võimalik terve domeeni elemente genereerida, ilma et oleks vaja absoluutselt kõiki generaatoreid lisada. See on võimalik, kuna triviaalse generaatoriga domeen ei kutsu välja sõltuvuseks oleva domeeni generaatorit ja seetõttu viimane ei pea generaatorit üldse implementeerima. Domeenide Variables, Normal ja PMap generaatorid selliselt hetkel implementeeritigi. See lähenemine võimaldab kasutada analüüsivast programmist leitud muutujaid, funktsioone vms nende domeenide generaatorites, mis vastavate konkreetsete ja keerukamate elementidega opereerivad.

Käivitamine Selliseks testimiseks lisati Goblinti käsurrealippude hulka uus nimega `dbg.test.domain`. Kui see on sisse lülitatud, siis enne etteantud programmi analüüsimist jooksutatakse analüüsiks kasutataval domeenil⁴ samad testid. Goblinti näidiskäivitus koos selle lipuga toimub järgneva käsuga:

⁴Täpsemalt „domeenidel“: Goblint kasutab analüüsis kahte erinevat domeeni, n-ö lokaalset ja globaalset.


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

4.5 Raskused testimisel

Implikatiivsed omadused Mõned peatükis 3.1 toodud omadustest on implikatsiooni kujul, see tähendab, et omadus peab kehtima ainult siis, kui mingid eeldused on täidetud. Nende testimise keerukuseks on see, et juhuslikke domeeni elemente genereerides ei pruugi need eeldust rahuldada. Sel juhul on implikatsioon küll tõene, aga ei väida midagi sisulist.

Kõige enam on see probleemiks osalise järjestuse antisümmeetrilisuse testimisel, milleks peaks juhuslikult genereerima domeeni elemendid a ja b nii, et $a \sqsubseteq b$ ja $b \sqsubseteq a$, enne kui saab kontrollida nõutud tingimust $a = b$. See on äärmiselt ebatõenäoline, kui testitavas domeenis on võrdlemisi palju elemente. Näiteks selle omaduse 100 (või isegi 10000) kordsel testimisel üle 32-bitiste täisarvude eeldust rahuldavaid argumente enamasti ei genereerita, mistõttu antisümmeetrilisus jääb praktiliselt kontrollimata.

Sellele vaatamata on implikatiivsete omaduste testid siiski Goblintis implementeeritud, sest need võivad probleeme avastada, kui järjestusseoses on mõni suurem viga.

Võrdlemine 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 kooskõllaliselt, 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 kooskõllaliselt defineeritud, aga puuduseks see, et eeldatakse, et järjestus on korrektne.

Võrduse ja osalise järjestuse vahelise ebakõla peaks tuvastama osalise järjestuse antisümmeetrilisuse test, kuid eelmises punktis kirjeldatud põhjustel ei pruugi see välja tulla. Testimise käigus paistis selline ebakõla välja vaid domeeni `CircInterval` vähima ja suurima elemendi puhul (testides „`bot is_bot`“, „`top is_top`“) ning järjestuse ja rajade vahelise ekvivalentsi juures (testides „`connect join`“, „`connect meet`“).

Laiendamine Esialgse testimise tulemused näitasid, nagu oleks kõigis testitud domeenides laiendamise operaator vigane, sest ei rahulda selle ainsat omadust peatükist 3.2. Osutub, et Goblinti domeenides olevat laiendamise operaatorit `widen` ei kasutata täpselt nii, nagu ülalkirjeldatud teooria seda ette näeb, vaid (ajaloolistel põhjustel) alati kujul `D.widen a (D.join a b)`, kus teiseks argumendiks on juba vastav ülemraja.

Tabel 4. Domeenide erinevatel meetoditel testimise ülevaatlilikud 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

Seega nõutud omaduse $a \sqcup b \sqsubseteq a \sqcup b$ asemel peaks kehtima omadus

$$\begin{aligned}
 a \sqcup (a \sqcup b) &\sqsubseteq a \sqcup (a \sqcup b) && \xRightarrow{\text{assotsiatiivsus}} \\
 (a \sqcup a) \sqcup b &\sqsubseteq a \sqcup (a \sqcup b) && \xRightarrow{\text{idempotentsus}} \\
 a \sqcup b &\sqsubseteq a \sqcup (a \sqcup b).
 \end{aligned}$$

Goblintis laiendamise operaatori testimiseks lõpuks kasutati viimast kuju. Seejuures eeldatakse, et ülemraja leidmise tehe on assotsiatiivne ja idempotentne, kuid neid kontrollitakse niikuinii vahetult omaette testidega.

4.6 Tulemused

Testide jooksumise ülevaatlilikud tulemused on toodud tabelis 4, kus on toodud mõlema lähenemise tulemused. Kuna võrdlusmetoodika puhul on tegemist kompromissiga, siis viidi samad testid läbi omakorda mõlemal moel. Läbikukkunud testide arvud on esitatud ligikaudselt, kuna kogu metoodika põhineb juhuslikkusel, mistõttu esineb väiksemaid variatsioone. Järgneb täpsem ülevaade probleemsetest domeenidest ja omadustest, kus mõningad probleemsete testide arvud võivad samuti varieeruda.

Erindi teke Mõnede omaduste testimisel tekkisid erindid enne kui kehtivust kontrollida oli võimalik. Enamus juhtudel seetõttu, et domeen tegelikult ei moodusta täielikku võre ning elementide \perp või \top asemel antakse erind. Erindite tekkimine domeenide kaupa:

1. Domeenis `IntegerSet`, mis implementeeriti täisarvude abstraktsioonide korrektsuse testimiseks, puudub \top , sest pole praktiline luua hulka kõigist täisarvudest ja vastaval testimisel seda pole ka vaja. Erind tekib kolmes suurima elemendi testis („top leq“, „top is_top“, „top meet“).
2. Domeenis `Integers` puuduvad \top ja \perp , mistõttu erind tekib lisaks kolmele suurima elemendi testile ka kolmes vähima elemendi testis („bot leq“, „bot is_bot“, „bot join“). Kuna \perp on vajalik ka abstraktsiooni korrektsuse testimisel, siis tekib samal põhjusel erind kõigis 22-s vastavas testis. Erindi teke on oodatud, sest lähtekoodis on selle domeeni juures kommentaar: „no top/bot, order is <=“.

3. Domeenis `CircInterval` tekib erind neljas abstraktsiooni korrektsust kontrollivas testis väljaselgitamata põhjusel:
 - Jagamise testis („valid div“) erind `Invalid_argument("compare: abstract value")`.
 - Kolmes loogikatehete testis („valid lognot“, „valid logand“, „valid logor“) erind `Failure("nativeint_of_big_int")`, kui argumendiks on täisarvu -1 abstraktsioon.

Mittekehtivad Mitmete domeenide ja omaduste testimisel selgus nende mittekehtivus. Detailsem ülevaade domeenide kaupa:

1. Domeen `Lifted` on ülalmainitud domeeni `Integers` täiendus ehk $\text{Integers} \cup \{\perp, \top\}$, siis selle testimisel erindeid ei teki, kuid 16 abstraktsiooni korrektsuse testi ikkagi ei kehti. Ka seda võib pidada ootuspäraseks, sest arvuline järjestus (nagu domeenis `Integers`) polegi korrektne täisarvude hulkade abstraktsioon, vaid on kasutusel muudel eesmärkidel.
2. Domeenis `Interval32`, mis töötab 32-bitiste täisarvude intervallidega, ei kehti arvudevahelise võrdusoperaatori abstraktsiooni korrektsus (testis „valid eq“), näiteks argumentidel

$$A = \{0\}, \quad B = \{2\,742\,196\,662\}.$$

Teine argument on juba vähendatud kujul ja on samamoodi suur korduval testi de jooksutamisel, mis vihjab, et probleem on põhjustatud abstraheeritava arvu suurusest. Probleemi põhjuseks võib olla see, et arv on väljaspool 32-bitiste märgiga täisarvude piire, ja sellise väärtusega ei peakski testima, kuid ülejäänud selle domeeni testides probleeme ei tekkinud.

3. Domeenis `Booleans`, mille elementideks kaks tõeväärtust, ei kehti bitinihke operaatorite abstraktsiooni korrektsus. Vasakule nihe (testis „valid shift_left“) näiteks argumentidel

$$A = \{-2\}, \quad B = \{-1\}$$

ja paremale nihe (testis „valid shift_right“) näiteks argumentidel

$$A = \{1\}, \quad B = \{1\}.$$

4. Domeenis `CircInterval` esineb mitmete omaduste mittekehtivus väljaselgitamata põhjustel:

- Võrdusseose transitiivsus (testis „equal trans“)⁵, näiteks argumentidel

$$a = [0, 0], \quad b = [], \quad c = [0, 1].$$

⁵Võrdusseose omadusi (refleksiivsus, sümmeetrilisus, transitiivsus) domeeni omaduste juures ei esitatud, kuid seoses võrdlemise problemaatikaga neid siiski otsustati lisaks testida.

- Kitsendamise omadus (testis „*narrow fst*“), näiteks argumentidel

$$a = [], \quad b = [0, 0].$$

- Ülem- ja alamraja assotsiatiivsus (testides „*join assoc*“, „*meet assoc*“). Lähtekoodis on kommentaaris neid operaatoreid nimetatud „pseudo-ülemrajaks“ ja „pseudo-alamrajaks“, mis vihjavad mingisugusele puudusele.
- Võrdlemise problemaatika juures mainitud neli omadust.
- Ülemraja mitteassotsiatiivsuse tõttu on rikutud ka 20 abstraktsiooni korrektsuse omaduse testimine.

5. Domeenis *Trier*, mille elementideks on üksikud täisarvud ja välistatud täisarvude hulgad (*exclusion set*), on rikutud ülemraja assotsiatiivsus (testis „*join assoc*“), näiteks argumentidel

$$a = 0, \quad b = 1, \quad c = \text{Not } \{3\}([-63, 63]).$$

Probleemi tuumaks on asjaolu, et kahe erineva konkreetse väärtuse ülemraja leidmisel unustatakse täielikult algsed väärtused, kuid väärtuse ja välistushulga ülemraja leidmisel mitte. Viga on teoorias, mitte implementatsioonis. Ülemraja mitteassotsiatiivne olemine rikub ka 11 abstraktsiooni korrektsusega seotud omadust.

Väga sarnase struktuuriga domeen *Enums*, mille elementideks on konkreetsete täisarvude hulgad ja välistatud täisarvude hulgad, seda omadust ei riku.

6. Domeen *IntDomTuple* sisuliselt moodustub nelja lihtsama domeeni valikulisest otsekorrutisest, nagu näidatud lisas III. Kui mingi omadus ei kehti selle mingis komponendis, siis põhjustab see kaudselt ka selle mittekehtimise *IntDomTuple* domeenil. Kuna komponentdomeenid pole veatud, siis pole kindel, kas *IntDomTuple* ise sisaldab vigu domeenide kokku panemisel või mitte. Kokku ei kehti sellel 20 omadust, millest 11 on seotud abstraktsiooni korrektsusega, mille testimiseks vajalik ülemraja assotsiatiivsus on samuti rikutud.

Samas ei kehti järjestusseose transitiivsus (testis „*leq trans*“), mis kõigil komponentidel on korras, näiteks argumentidel

$$a = \{\text{enums} : \top\}, \quad b = \{\}, \quad c = \{\text{enums} : \perp\}.$$

Tehtud ülevaatest on näha, et omaduste mittekehtimisel võib olla erinevaid põhjuseid:

1. viga domeeni implementeerimisel,
2. mittekehtimine teoreetilisel tasandil,
3. teadlik ja dokumenteeritud mittekehtimine,
4. sõltumine teistest probleemsetest domeenidest.

Kuna töö eesmärgiks polnud vigade parandamine, siis kõikidesse probleemidesse oluliselt ei süvenetud, mistõttu paljude läbikukkunud testide kohta ei saa kindlalt midagi väita. See nõuab omaette teadmisi konkreetsete domeenide ettenähtud tööpõhimõtetest, tööd täpsete kontranäidete analüüsimiseks ja probleemi kõrvaldamiseks. Sellele vaatamata annavad läbiviidud testid ja nende analüüs suuna, kust vigu otsida.

5 Kokkuvõte

Töö teoreetilise osana anti ülevaade andmevooanalüüsist ja abstraktsetest domeenidest, mida selleks kasutatakse. Seejärel koostati ülevaatlik abstraktsete domeenide omaduste komplekt, mis ühest küljest on piisavalt üldine, et kehtida mistahes domeenil, ja teisest küljest on piisavalt täielik, et katta võimalikult palju üldisel tasemel kontrollitavast. 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 üldtestitaval viisil kõik omadused ja lisati paljudesse Goblinti domeenidesse vajalikud juhuslike elementide generaatorid. Domeenide täiendamisele ja testimise läbiviimisele läheneti kahest suunast, millel on erinevad eelised ja puudused. Lisaks analüüsiti aspekte, mis teevad domeenide omaduspõhise testimise keerukaks. Goblintisse lisatud domeenide omaduspõhise testimise raamistiku saab kasutusele võtta selle edasises arenduses, et vältida domeenide implementeerimisel võimalikke vigu.

Goblinti domeenide omaduspõhise testimise tulemusena leiti mitmeid domeene, millel mingid omadused ei kehtinud. Teostati tulemuste analüüs, et leitud vigu klassifitseerida ja võimalikke põhjuseid tuvastada. Esitati omaduspõhisest testimisest leitud kontranäited, millest vigade olemasolu selgub ja mida peaks vigade parandamiseks põhjalikumalt analüüsima.

Selle töö raames siiski ei varustatud absoluutselt kõiki Goblinti domeene generaatoritega, mis neid testida võimaldaks, ja seetõttu on tööd võimalik jätkata, et kontrollitud oleks lõpuks kõik domeenid. Lisaks on muidugi vaja sügavamalt uurida neid domeene, millest vigu leiti, et aru saada vigade täpsest sisust, ja lõpuks vead kõrvaldada. Igal juhul on võimalik tehtut edasi arendada ja ära kasutada Goblinti parandamisel.

Läbiviidud testimine näitas, et omaduspõhist testimist on võimalik hästi rakendada staatilise analüsaatori abstraktsete domeenide omaduste kontrollimiseks ja neist vigade leidmiseks. Tõenäoliselt oleks omaduspõhist testimist võimalik kasutada ka teiste analüsaatori osade (nt üleminekufunktsioonide) korrektse toimimise kontrollimiseks, kuna abstraktsed domeenid pole kaugelki ainsad andmevooanalüüsi osad, milles vigu võib esineda.

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 Committee 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](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).

Lisad

I Tsüklita näiteprogrammi iteratiivne analüüs

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

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

II Lähtekood

Goblinti täiendamiseks tehti selle lähtekoodi repositooriumist [10] GitHub keskkonnas *fork*, mis asub aadressil

<https://github.com/sim642/goblint>.

Töö autori muudatused koos täpse versioonihaldusajalooga leiab selle Git repositooriumi *tag*'ilt „bsc-thesis“, mida näeb ka aadressilt

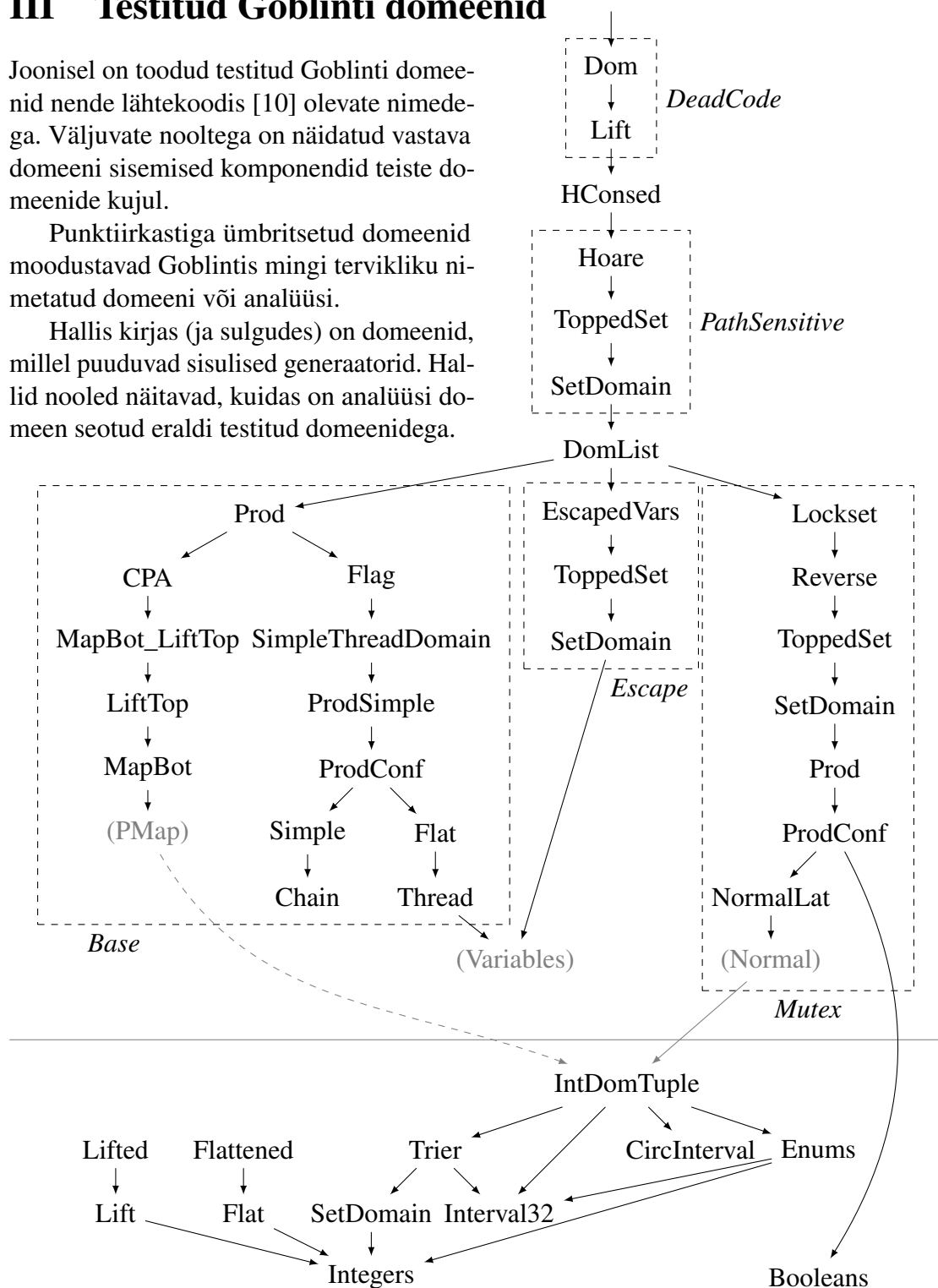
<https://github.com/sim642/goblint/tree/bsc-thesis>.

III Testitud Goblinti domeenid

Joonisel on toodud testitud Goblinti domeenid nende lähtekoodis [10] olevate nimedega. Väljuvate nooltega on näidatud vastava domeeni sisemised komponendid teiste domeenide kujul.

Punktiirkastiga ümbritsetud domeenid moodustavad Goblintis mingi tervikliku nimetatud domeeni või analüüsi.

Hallis kirjas (ja sulgudes) on domeenid, millel puuduvad sisulised generaatorid. Hallid nooled näitavad, kuidas on analüüsi domeen seotud eraldi testitud domeenidega.



IV 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 lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
 - 1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace 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, 14.05.2018