TARTU ÜLIKOOL
Loodus- ja täppisteaduste valdkond
Arvutiteaduse instituut
Informaatika õppekava

Heigo Tornik

Nõrgad sõltuvused kõrvalmõjudega kitsendussüsteemides

Bakalaureusetöö (9 EAP)

Juhendajad: Simmo Saan, MSc
Vesal Vojdani, PhD

Tartu 2025

Nõrgad sõltuvused kõrvalmõjudega kitsendussüsteemides

Lühikokkuvõte: Bakalaureusetöös vaadeldakse staatilise analüüsi metoodikat abstraktset interpretatsiooni, täpsemalt kõrvalmõjudega kitsendussüsteeme. Kitsendussüsteemidega saab abstraktselt kirjeldada programme ning neid lahendades laskuvate lahendajatega, näiteks TD_{side}-iga, kirjeldada mitmeid omadusi, näiteks esinevaid vigu. Töös antakse teoreetiline taust kitsendussüsteemide ja laskuva lahendaja TD_{side} kohta, ning käsitletakse ebatõhusust TD_{side}-is, kus lõime loov programmipunkt sõltub loodavast lõimest, tingides ebavajalikke taasarvutusi. Ebatõhususe lahendamiseks kirjeldatakse nn "nõrgad sõltuvused". Täiendatakse kitsendussüsteemi paremat poolt, lisades uue funktsiooni $demand$ ning kirjeldatakse vajalikud täiendused TD_{side}-is uute kitsendussüsteemide lahendamiseks. Töös loodi TD_{side} põhjal TD_{weak} kahes variatsioonis: ahne ja laisk. Nõrgad sõltuvused Goblintis implementeerituna vähendasid kitsendussüsteemide paremate poolte väärtustamise arvu keskmiselt 28.87% laisa variatsiooni korral ja 13.70% ahne korral.

Võtmesõnad:
staatiline analüüs, Goblint

CERCS: P170. Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine.

Weak Dependencies for Side-Effecting Constraint Systems

Abstract: The bachelor's thesis examines the static analysis method abstract interpretation, specifically side-effecting constraint systems. Constraint systems can be used to describe programs in an abstract manner and, by solving them with top-down solvers, such as TD_{side}, several properties, like occuring errors, can be described. The thesis provides a theoretical background on constraint systems and the top-down solver TD_{side}, and describes the inefficiency in TD_{side}, where the program point that creates a thread depends on the thread being created, causing unnecessary recalculations. To solve the inefficiency, so-called "weak dependencies" are described. The right-hand side of the constraint system is supplemented by adding a new function $demand$ and the necessary additions to TD_{side} to solve these new constraint systems are described. In the thesis TD_{weak} was created based on TD_{side} in two variations: eager and lazy. Weak dependencies, when implemented in Goblint, reduced the number of right-hand side evaluations of constraint system by an average of 28.87% for the lazy variation and 13.70% for the eager variation.

Keywords:
static analysis, Goblint

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

Sisukord

Sissejuhatus

Staatilist programmianalüüsi (ingl *static program analysis*) kasutatakse programmide vigade tuvastamiseks ning turvalisuse ja tõhususe tõstmiseks, käivitamata programmi ennast, vaid selle lähtekoodi uurides.

Järgnev lõik tugineb Anders Møller ja Michael I. Schwartzbachi õpikule [1]. Oluline rakendus staatilisele analüüsile on mitmelõimeliste programmide tõrkeolukordade — trügimiste (ingl *race conditions*) ja tupikute (ingl *deadlocks*) — tuvastus, kuna enamik mitmelõimeliste programmide erinditest on tingitud sünkroniseerimise või lukustusmehanismide valest kasutusest. Analüüsi läbiviimise sellistel programmidel teeb keeruliseks eksponentsiaalselt kasvav võimalike täitmiste järjekord ning semaforide ja lukkude kasutamisest tingitud käitumine.

Kõrvaltoimetega kitsendussüsteemid (ingl *side-effecting constraint systems*) on võimaldanud kirjeldada protsessidevahelisi käitumisi. Sellised kitsendussüsteemid, võrreldes tavalistega, võimaldavad kirjeldada mõjutusi erinevate tundmatute vahel kitsendussüsteemides. Kitsendussüsteemide lahendamiseks on mitmeid algoritme, neist tõhusamad programmianalüüsi kontekstis on laskuvad lahendajad (ingl *top-down solvers*).

Goblinti, programmeerimiskeele C staatilise analüsaatori, teostuses on kasutatud laskuvat lahendajat, mille teostust on täiendatud toetamaks kõrvaltoimetega kitsendussüsteeme. Täiendatud lahendaja võimaldab seeläbi tõhusamalt analüüsida mitmelõimelisi programme kui Charlieri ja Hentenrycki laskuv lahendaja [2, 3]. Laskuva lahendaja on tõhus meetod protseduuride analüüsimiseks, kuid lõimede loomisel tekitab üleliigseid sõltuvusi (loova lõime analüüs sõltub loodud lõime omast).

Bakalaureusetöö esimene eesmärk on teostada nn “nõrgad sõltuvused”. Teostamiseks tuleb täiendada kõrvalmõjudega kitsendussüsteeme, seeläbi siduda lahti loodud-loova lõime omavaheline sõltuvus, kuid tagada analüüsi lõppedes saadud tulemuse õigsus. Kõrvalmõjudega kitsendussüsteemide muutmisel on seetõttu tarvis täiendada ka laskuvat lahendajat. Töö teine eesmärk on läbi võrdlusuuringu kontrollida, kas uue kitsendussüsteemi lahendamine täiendatud laskuva lahendajaga on tõhusam kui esialgne.

Bakalaureusetöö on jaotatud kolmeks osaks. Esimeses osas kirjeldatakse mitmelõimeliste programmide analüüsi, ning erinevaid kitsendussüsteeme ning nende mõistmiseks vajalikke definitsioone. Teine osa algab sissejuhatusega laskuvatesse lahendajatesse, millele järgneb ülevaade $\mathrm{TD}_{\mathrm{side}}$ solverist (kasutusel Goblintis) ning kirjeldatakse lahendaja korrektsust, lõpuks on kirjeldatud lahendaja ebatõhusust. Viimases osas tuuakse välja nõrkade sõltuvuste ja täiendatud lahendaja variatsioonide tööpõhimõtted, nende korrektsuse tõestus, võrdlusuuring, ning käsitletakse ka kitsaskohti.

1 Teoreetiline taust

Peatükis vaadeldakse mitmelõimelisi programme, eeliseid ühelõimeliste suhtes ning vigu, mis nendes võivad esineda. Teises alampeatükis kirjeldatakse kuidas saab analüüsida programme ning töös vaadeldavat metoodikat — abstraktset interpretatsiooni. Viimases alampeatükis kirjeldatakse kitsendussüsteeme, mida kasutatakse abstraktses interpretatsioonis, ning kitsendussüsteemide mõistmiseks olulisi teemasid — täielikud võred, funktsioonide monotoonsus ning kitsendussüsteemide lahend.

1.1 Mitmelõimelised programmid

Konkurrentsust kirjeldatakse kui süsteemi omadust, kus sündmused võivad toimuda üksteisest sõltumata [4]. Mitmelõimelisus on üks viisidest saavutada konkurrentsust. Arvuti arhitektuuris kirjeldatakse mitmelõimelisust kui keskprotsessori või protsessorituuma võimet täita korraga mitut protsessi või lõime (ehk programmiosa), sõltumata teistest protsessidest või lõimedest [5, 4]. Eelis ühelõimeliste programmide ees on tõhusam keskprotsessori kasutus — blokeerivate tegumite ajal on võimalik teostada teisi tegumeid, vähendades jõudeaega. Mitmelõimelisuse kasutusteks on näiteks aeglaste sisend-väljund seadmete kasutamine, interaktsioon inimestega, mitme võrgukliendi päringute teenindamine [6].

Järgnev lõik tugineb Sara Abbaspour Asadollah et al. [7] tehtud uuringule. Aastal 2017 avatud lähtekoodiga projektide põhjal tehtud juhtumianalüüsis leiti, et ligi 4% programmivigadest on tingitud konkurrentsusest, ent parandamiseks kuluv aeg võrreldes ülejäänud vigadega oli 24% pikem. Konkurrentsusest tulenevad vead moodustasid 40% mitte-korratavate vigade hulgast ning kategoriseeriti tihemini enim häirivateks vigadeks.

Joonisel 1 on kujutatud mitmelõimelist programmi, mis koosneb funktsioonidest `main` ja `t_foo`. Programm alustab tööd funktsioonist `main`, seejärel luuakse uus lõim `t_foo`, väärtustatakse muutujad `a`, `g` ja `b`, prinditakse kasutajale `h` ja `j` väärtused ning tagastatakse 0 programmi lõpus. Loodud lõim `t_foo`, sõltuvalt `g` väärtusest, väärtustab muutujad `h` ja `j`. Peale lõime loomist eeldame, et funktsioonid jooksevad samaaegselt kahel erineval lõimel. Sõltuvalt järjekorrast, kuidas kahe lõime samme täidetakse, võib väljund varieeruda. Näiteks näidisprogramm trükib muutujate `h` ja `j` väärtused, mis saavad olla ”0 0”, ”1 0” ja ”1 1”. Varieeruv täitmisjärjekord teeb raskemaks programmi mõistmist ning selle korrektsuses veendumist.

```
1  int g, h, j = 0
2  thread t_foo() {
3    if (g == 1) {
4      h = 1;
5      j = 1;
6    }
7  }
8
9  func main() {
10   thread_create(t_foo);
11   int a = 1;
12
13   g = 1;
14   int b = 2;
15   print(h, j);
16 }
```

Joonis 1: Pseudokoodis kirjutatud mitmelõimeline programm, mis sõltuvalt muutuja g väärtusest, väärtustab h ja j ning väljastab funktsiooni main lõppedes nende väärtused.

1.2 Mitmelõimeliste programmide analüüs

Tarkvara analüüsi metoodika valimiseks tuleb lähtuda analüüsi eesmärgist ja metoodika enda omadustest. Mitmelõimeliste programmide analüüsimises on oluline analüüsi tulemusena tõestada, et programmi täitmisel ei esineks vigu. Üks sellist eesmärki täitvatest metoodikatest on automaatverifitseerimine. Bakalaureusetöö raames vaadeldakse automaatverifitseerimist, kuna mitmelõimeliste programmide analüüsis on tähtis korrektsus (garantii analüüsi lõpuks vähemalt leida kõik päriselt esinevad vead), automatiseeritus ning lõimemodulaarsus (võimalus skaleerida metoodikat, teostades analüüsi lõimhaaval teistest lõimedest sõltumata) [8]. Automaatverifitseerimine hõlmab omakorda mitut metoodikat, üks nendest on abstraktne interpretatsioon. Töös on vaadeldavaks metoodikaks abstraktne interpretatsioon, mis võimaldab programmide kõikvõimalike käitumisi matkida abstraktsete olekutega, mis esindavad programmi seisundite hulka [9].

Skaleeritavus on automaatverifitseerimise metoodika valimisel tähtis, sest lõimede arvu kasvades tõuseb programmi väljundite arv eksponentsiaalselt. Analüsaatoris Astrée on kasutatud abstraktset interpretatsiooni erinevate organisatsioonide koodibaaside peal vigade leidmiseks - analüsaatorit on kasutatud Euroopa Kosmoseagentuuri, Airbus France, Bosch poolt - kinnitades meetodika skaleeritavust päriselulistele programmidele [10].

Korrektsuse puhul soovime kinnitada, et analüüsi lõpuks on leitud programmi kõik

võimalikud vead, kusjuures tuleb arvestada, et analüüs võib ka leida vigu, mida konkreetses programmis päriselt ei esine. Abstraktse interpretatsiooni korrektsust tõestati kaardistades programmi konkreetsete käskude semantika abstraktsetele [11]. Näidati, et abstraktsete süsteemide lahendeid arvutades saadud tulemused ning nendest järeldatud omadused kattuvad konkreetse programmi omaga.

Abstraktset interpretatsiooni saab teostada kahel viisil: analüüsida otse süntakspuu pealt või kasutades kitsendussüsteeme [12]. Töös vaadeldakse kitsendussüsteeme.

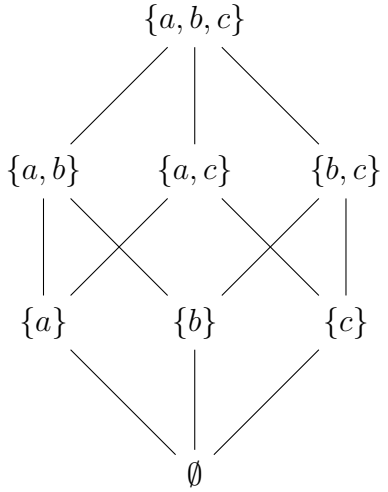
1.3 Kitsendussüsteemid

Kitsendussüsteem (ingl *constraint systems*) on matemaatiline struktuur, mis võimaldab kirjeldada seoseid võrrandisüsteemide erinevate muutujate ehk tundmatute vahel. Kitsendussüsteemid on üks vahenditest, mida kasutatakse programmide kirjeldamiseks abstraktsete mudelitena [1, 13].

Järgnev lõik tugineb Helmut Seidl, Reinhar Wilhelm ja Sebastian Hacki õpikule [14]. Tavalised kitsendussüsteemid on võrratussüsteemid, kus võrratuse vasakuks pooleks on tundmatud muutujad x_i ja paremaks pooleks on funktsioonid $f_i : \mathbb{D}^n \to \mathbb{D}$, kus $\mathbb{D}$ on täielik võre. Kitsendussüsteemi võrratusi kirjutame kujul $x_i \sqsupseteq f_i(x_1, ..., x_n) \qquad i = 1, ..., n$, kus $n \in \mathbb{N}$. Kitsendussüsteemide paremaks mõistmiseks tuleb esmalt selgitada järgnevat:

1. võresid ning võre omadusi ja tähtsust programmianalüüsi kontekstis;
2. kitsendussüsteemi paremat poolt;
3. kitsendussüsteemi lahendit ning seda tagavaid omadusi.

Täielik võre on osaliselt järjestatud hulk, mille kõikidel alamhulkadel eksisteerib alamraja ja ülemraja, ning millel on vähim element $\bot$ ja suurim element $\top$ [14]. Joonisel 2 toodud näite puhul on tegu täieliku võrega - hulga $\{a, b, c\}$ astmehulk on järjestatud sisalduvuse ($\subseteq$) alusel, $\bot = \emptyset$ ja $\top = \{a, b, c\}$. Käsitledes tundmatute väärtuseid täieliku võre elementidena, saab väita programmianalüüsi kontekstis, et kitsendussüsteemide lahendamisel parimad ehk täpsemad lahendused on väikseimad elemendid võres [14]. Tehetemärk $\sqsupseteq$ kirjeldab kitsendussüsteemide ja programmianalüüsi tulemuse täpsust. Näiteks $a \sqsupseteq b$, tähendab, et a on suurem või sama suur element võres kui b, mida saab lugeda, kui "a on vähem täpsem või sama täpne kui b", või "a on sama täpne või ülehinnang b väärtusest".



Joonis 2: Täielik võre $L = (P(\{a, b, c\}), \subseteq)$

Järgnev lõik tugineb Helmut Seidl, Reinhar Wilhelm ja Sebastian Hacki õpikule [14]. Funktsioon f_i kirjeldab tundmatu x_i sõltuvusi teiste tundmatute suhtes. Funktsioon f_i on monotoonne funktsioon. Monotoonsus kirjeldab funktsioonide korral omadust säilitada järjestust võre elemendide vahel, kui esimene element kuulub teise, formaalselt $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$.

Püsipunkt on lahend funktsioonile $f(x) = x$ [15]. Näiteks võrrandi $f(x) = (x - 2)^2$ korral on kaks püsipunkti $x = 4$ ja $x = 1$, samas funktsioonil $f(x) = x + 1$ pole ühtegi püsipunkti. Hulkade korral kirjeldatakse sarnaselt püsipunkte: $F(X) = X$, kus F on funktsioon hulkadest hulkadesse ja X on suvaline hulk.

Järgnev lõik tugineb Helmut Seidl, Reinhar Wilhelm ja Sebastian Hacki õpikule [14]. Tuginedes Knaster-Tarski teoreemile leidub igale monotoonsele funktsioonile $f : \mathbb{D} \to \mathbb{D}$ püsipunkt, mis on vähim lahendus, mis rahuldab võrratust $x \sqsupseteq f(x)$. Kuna kitsendussüsteemides on tegu monotoonsete funktsioonidega, saab kitsendussüsteemi funktsioonest moodustada liitfunktsioon $f : \mathbb{D}^n \to \mathbb{D}^n$, kus $f(x_1, ..., x_n) = (y_1, ..., y_n)$ ning $y_i = f_i(x_1, ..., x_n)$, seeläbi on saadud liitfunktsioon monotoonne ning saab tervele kitsendussüsteemile leida püsipunkti.

Vaadeldakse näitena täielikku võre $L = (P(\{a, b, c\}), \subseteq)$, $\sqsubseteq = \subseteq$, ning kitsendussüsteemi joonisel 3.

$$
\begin{aligned}
x_1 &\supseteq x_2 \cup \{a\} \\
x_2 &\supseteq \{b\} \cup x_1 \\
x_3 &\supseteq \{a, b\} \cap x_4 \\
x_4 &\supseteq \{a, c\} \cap x_1
\end{aligned}
$$

Joonis 3: Tavaline kitsendussüsteem, täielik võre $L = (P(\{a, b, c\}), \subseteq)$.

Kitsendussüsteemi (joonisel 3) vähima püsipunkti leidmiseks kasutatakse naiivset püsipunkti iteratsiooni, kus määratakse igal sammul tundmatute väärtuseks eelmisel sammul arvutatud väärtused. Saadakse tabel 1, kus on saavutatud 5. sammuks püsipunkt ehk leitud vähim lahend kitsendussüsteemile.

Tabel 1: Tavalise kitsendussüsteemi (joonis 3) lahendamise tulemus igal sammul.

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$
x_2	$\emptyset$	$\{b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$
x_3	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$
x_4	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$	$\{a\}$

Kitsendussüsteemide kirjeldamisel on võimalik kasutada ka funktsionaalset kuju, kus võrrandisüsteemi paremaid pooli kirjutatakse kui **fun** $get \rightarrow e$, funktsiooni defineeritakse kui $f_x : (V \rightarrow \mathbb{D}) \rightarrow \mathbb{D}$ ning funktsionaalses kujus e on konkreetse parema poole definitsioon ja V tundmatute hulk [3]. Näite puhul (joonisel 3) saadakse uuel kujul võrrandisüsteem joonisel 4. Kitsendussüsteemi lahendades jäävad lahenduskäik ja tulemus samaks, mis tabelis 1.

$$
\begin{aligned}
x_1 &\supseteq \textbf{fun}\ get \rightarrow get\ x_2 \cup \{a\} \\
x_2 &\supseteq \textbf{fun}\ get \rightarrow \{b\} \cup get\ x_1 \\
x_3 &\supseteq \textbf{fun}\ get \rightarrow \{a, b\} \cap get\ x_4 \\
x_4 &\supseteq \textbf{fun}\ get \rightarrow \{a, c\} \cap get\ x_1
\end{aligned}
$$

Joonis 4: Tavaline kitsendussüsteem funktsionaalsel kujul, täielik võre $L = (P(\{a, b, c\}), \subseteq)$.

Järgnev lõik tugineb Helmut Seidl ja Ralf Vogleri artiklile [3]. Tavaliste kitsendussüsteemide puhul kirjeldatakse terve kitsendussüsteemi tundmatute väärtustusi kui

$\sigma : V \to \mathbb{D}$. Kuna kitsendussüsteeme lahendades ei pruugi olla võimalik mõistliku ajaga leida püsipunkti, loetakse lahendiks ka saadud tundmatute väärtusi, mis ülehindavad tegeliku püsipunkti väärtust. Sellist lahendit nimetatakse täielikuks ülelähendiks (ingl *total post-solution*), formaalselt σ on kitsendussüsteemi ülelähend, kui iga $x \in V$ korral $\sigma\, x \sqsupseteq f_x\, \sigma$. Täielik tähendab, et igale muutujale on garanteeritud tulemus, ning ülelähend tähendab, et muutujate väärtused ei pruugi olla täpsed, kuid kindlalt ülehindavad nende väärtust.

Kõrvalmõjudeta kitsendussüsteemide probleemiks on toodud välja, et kuigi mitmelõimeliste programmide puhul on võimalik täpselt kirjeldada programmi, siis suuremate või lõimerohkete programmide puhul muutub kitsendussüsteemide esitus praktiliselt kasutuskõlbmatuks [13].

1.4 Kõrvalmõjudega kitsendussüsteemid

Kitsendussüsteemid kõrvalmõjudega täiendavad tavalisi kitsendussüsteeme, võimaldades lahendamise käigus väärtustada teisi tundmatuid. Sellised kitsendussüsteemid võimaldavad näiteks samaaegselt jooksvate lõimede mõjutusi teineteisele kirjeldada [16]. Alljärgnev tugineb Kalmer Apinis, Helmut Seidl ja Vesal Vojdani artiklile [17]. Ühe väljundina kõrvalmõjudega kitsendussüsteemid lihtsustavad analüüsi perspektiivis dünaamiliste funktsioonide kutseid, näiteks vaadeldes programmi, kus programmi seisust sõltuvalt määratakse viida väärtuseks erinevaid funktsioone, mida järgmistes sammudes välja kutsutakse. Tavaliste kitsendussüsteemidega parim täpsus, mida saaks saavutada, on hulk kõikide võimalike funktsioonidega, mida programm võib välja kutsuda, koos kõikvõimalike programmiolekutega. Võrreldes kõrvalmõjudega kitsendussüsteemidega, saadakse kõik võimalikud funktsioonid, kuid täpsemalt määratletud programmiolekutega, millega funktsioone kutsutakse.

Alljärgnev tugineb Helmut Seidl ja Ralf Vogleri artiklile [3]. Kõrvalmõjudega kitsendussüsteemide paremaks pooleks on funktsioon $f_x : (V \to \mathbb{D}) \to (V \to \mathbb{D} \to unit) \to \mathbb{D}$, kus V on kitsendussüsteemi tundmatute hulk. Funktsiooni kirjutatakse funktsionaalsel kujul kui **fun** $get\ set \to e$, kus $get : V \to \mathbb{D}$ on tundmatu hetkeväärtuse pärimise funktsioon, $set : V \to \mathbb{D} \to unit$ on kõrvalmõju ehk tundmatu väärtustamise funktsioon, ja e on konkreetse kitsenduse definitsioon.

Vaadeldakse lihtsat kõrvalmõjudega kitsendussüsteemi joonisel 5, kus g algväärtus on $\emptyset$. Lahendades kitsendussüsteemi saadakse tabel 2, kus püsipunkti saavutatakse sammul 6. Tabelisse on muutuja g väärtuse kujutamiseks lisatud eraldi rida.

$$
\begin{aligned}
x_1 &\supseteq \textbf{fun}\ get\ set\ \ \rightarrow get\ x_2 \cup \{a\} \\
x_2 &\supseteq \textbf{fun}\ get\ set\ \ \rightarrow set\ g\ (get\ x_1); \{b\} \cup get\ x_1 \\
x_3 &\supseteq \textbf{fun}\ get\ set\ \ \rightarrow \{a, b\} \cap get\ x_4 \\
x_4 &\supseteq \textbf{fun}\ get\ set\ \ \rightarrow \{a, c\} \cap (get\ g)
\end{aligned}
$$

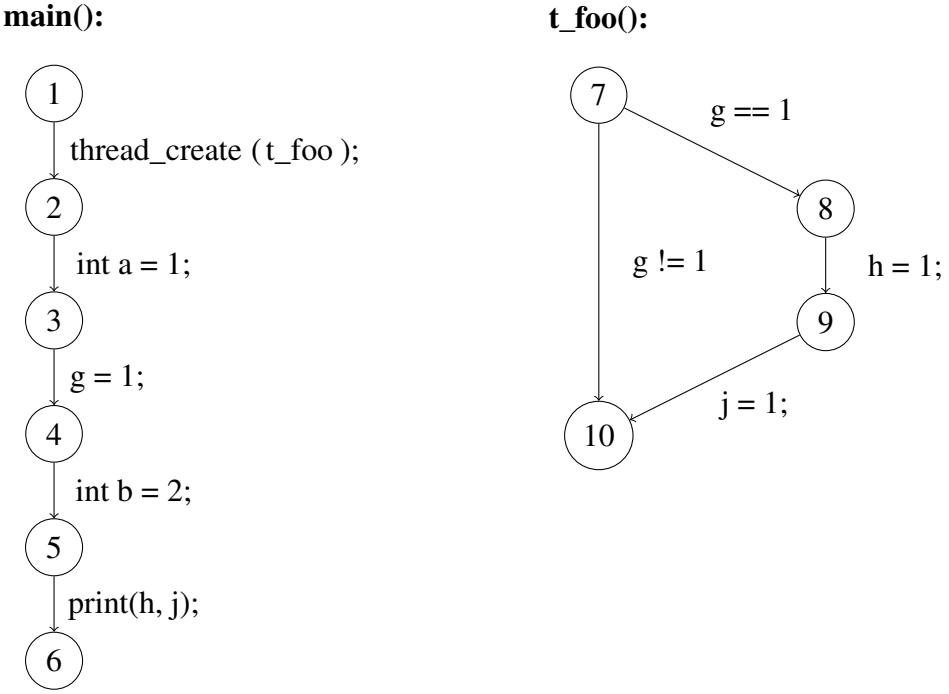
Joonis 5: Kõrvalmõjudega kitsendussüsteem, täielik võre $L = (P(\{a, b, c\}), \subseteq)$

Tabel 2: Kõrvalmõjudega kitsendussüsteemi (joonis 5) lahendamise tulemus igal sammul.

	0	1	2	3	4	5
x_1	$\emptyset$	$\{a\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$
x_2	$\emptyset$	$\{b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$
x_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$
x_4	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$	$\{a\}$
g	$\emptyset$	$\emptyset$	$\{a\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$

Sarnaselt tavalistele kitsendussüsteemidele, loetakse lahendit σ täielikuks ülelähendiks kui $\forall\ x\ \in\ V, \sigma\ x\ \sqsupseteq\ f_x\ \sigma\ set$, kus iga kutse $set\ y\ d$ korral $\sigma\ y\ \sqsupseteq\ d$ ja $y\ \in\ V$ [3]. Sedasi kinnitakse, et kõik tundmatute väärtused ja kõrvalmõjudest saadud väärtustused ülehindavad tegelikke väärtuseid.

Kitsendussüsteemide kasutamiseks programmide analüüsiks esiteks konstrueeritakse programmist juhtvoograaf. Programmi kirjeldatakse kui lõplikku hulka P, mille elementideks on protsessid (joonise 1 puhul näiteks meetodid `main` ja `t_foo`), igale protsessile p vastab unikaalne juhtvoograaf. Suvalist juhtvoograafi kirjeldatakse kui (N_p, $N_p \times L \times N_p$), N_p on protseduuri p programmipunktide hulk ja L on lause või tingimusvalvur (ingl *conditional guard*) [17]. Konstrueeritakse juhtvoograaf joonise 1 põhjal ning saadakse joonis 6.



Joonis 6: Näidisprogrammile (joonis 1) vastav juhtvoograaf.

Juhtvoograafi (joonis 6) põhjal luuakse kõrvalmõjudega kitsendussüsteem konkreetse programmi kohta. Arvestatakse, et $V = N_{main} \cup N_{t\_foo}$ ehk hulk V sisaldab kõiki protseduuride `main` ja `t_foo` programmipunkte. Eeldatakse, et $d_0 \in \mathbb{D}$ kirjeldab programmiolekut enne `main` väljakutset ning igale programmi käsule $s \in L$ on antud abstraktne funktsioon $[\![s]\!]^\sharp \in \mathbb{D} \to \mathbb{D}$, mis kirjeldab kuidas programmi käsk teisendab eelmise oleku järgmiseks olekuks. Tulemuseks on joonisel 7 kujutatud kõrvalmõjudega kitsendussüsteem.

$$
\begin{aligned}
[s_1] &\sqsupseteq set\ [g]\ [\![0]\!]^\sharp; set\ [h]\ [\![0]\!]^\sharp; set\ [j]\ [\![0]\!]^\sharp; d_0 \\
[s_2] &\sqsupseteq \textbf{let}\ d = get\ [s_1]\ \textbf{in}\ set\ [s_7]\ \varnothing;\ get\ [s_{10}];\ d \\
[s_3] &\sqsupseteq [\![a = 1]\!]^\sharp\ (get\ [s_2]) \\
[s_4] &\sqsupseteq \textbf{let}\ d = get\ [s_3]\ \textbf{in}\ set\ [g]\ [\![1]\!]^\sharp;\ d \\
[s_5] &\sqsupseteq [\![b = 2]\!]^\sharp\ (get\ [s_4]) \\
[s_6] &\sqsupseteq get\ [h];\ get\ [j];\ get\ [s_5] \\
[s_8] &\sqsupseteq \textbf{if}\ ([\![1]\!]^\sharp \sqsubseteq\ get\ [g])\ \textbf{then}\ get\ [s_7]\ \textbf{else}\ \bot \\
[s_9] &\sqsupseteq \textbf{let}\ d = get\ [s_7]\ \textbf{in}\ set\ [h]\ [\![1]\!]^\sharp; d \\
[s_{10}] &\sqsupseteq \textbf{if}\ (get\ [g] \not\sqsubseteq\ [\![1]\!]^\sharp)\ \textbf{then}\ get\ [s_7]\ \textbf{else}\ \bot \\
[s_{10}] &\sqsupseteq \textbf{let}\ d = get\ [s_9]\ \textbf{in}\ set\ [j]\ [\![1]\!]^\sharp; d
\end{aligned}
$$

Joonis 7: Juhtvoograafile (joonis 6) vastav kõrvalmõjudega kitsendussüsteem.

Joonisel 7 on programmipunktid kujul $[s_n]$ kus n on naturaalarv vahemikus $[1, 10]$ ning igale programmipunktile vastab vähemalt üks parem pool. Vaadeldes ühte programmipunktile vastavat paremat poolt, näiteks $[s_2]$, saab näha kuidas joonisel 6 kujutud käsud sellega ühtivad. Programmipunkti paremaks pooleks on

$$\textbf{let}\ d = get\ [s_1]\ \textbf{in}\ set\ [s_7]\ \varnothing;\ get\ [s_{10}],$$

mis kirjeldab lõime loomise käsku `thread_create(t_foo)`. Esmalt päritakse eelmise programmipunkti tulemust $get\ [s_1]$, antud juhul programmi algolekut, mis määratakse muutjasse d. Järgmisena kuna lõime funktsioon `t_foo()` ei nõua ühtegi argumenti, määratakse $set\ [s_7]\ \varnothing$ ning päritakse lõime lõpp-punkti tulemust $get\ [s_{10}]$. Viimasena tagastatakse parema poole tulemusena $get\ [s_1]$ (muutuja d kaudu).

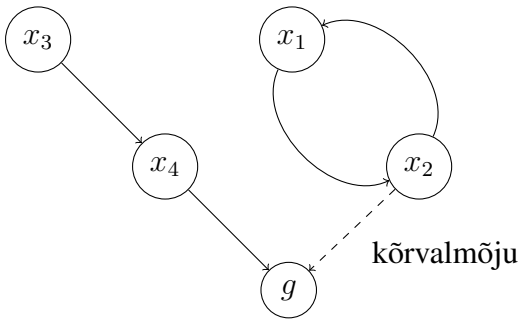
Kõrvalmõjudega kitsendussüsteeme on kasutatud programmeerimiskeele C staatilises analüsaatoris Goblint ning kasutusvõimalusi on laiendatud primitiivide (näiteks mutexsite) analüüsimiseks. Kõrvalmõjud võimaldavad eraldiseisvalt ülejäänud programmist analüüsida lõime spetsiifilisi olekuid [3, 16].

2 TD_{side} lahendaja ja selle ebatõhusus

Kitsendussüsteemide lahendite arvutamiseks on mitmeid algoritme nagu naiivne püsipunkti iteratsioon (ingl *naive fixpoint iteration*), kõik kordamööda iteratsioon (ingl *round-robin iteration*), tegevuskava iteratsioon (ingl *worklist iteration*) ja laskuv lahendaja (ingl *top-down solver*) [2, 14]. Töös vaadeldakse variatsiooni laskuvast lahendajast, mis on programmide analüüsimise aspektis tõhusam teistest loetletud algoritmidest, kuna arvutatakse vaid nõutud programmipunkte ning taasarvutused toimuvad vaid muutunud võrrandisüsteemide paremat pooltel [2].

2.1 Laskuva lahendaja käitumine

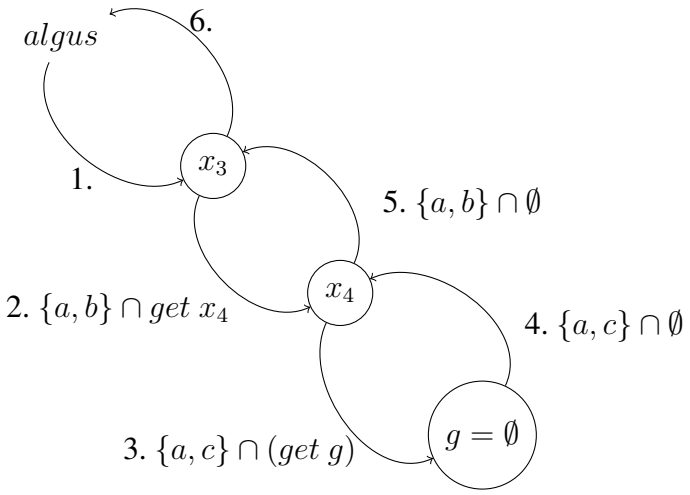
Laskuvate lahendajate eesmärk on analüüsida programmi alustades etteantud programmipunktist, mis on tavapäraselt programmi lõpp-punkt, kuid võib olla mistahes programmipunkt [2]. Lahendaja selgitab välja järgnevad analüüsi vajavaid programmipunktid, lahendades hetkel analüüsitava programmipunkti paremat poolt. Joonise 5 võrrandisüsteemi programmipunktide nõudmist ehk tundmatute omavahelisi sõltuvusi kujutatakse joonisel 8. Tundmatust väljuv pidev nool kirjeldab, millised teised tundmatud selle paremas pooles on ehk millest see sõltub (näiteks x_3 sõltub x_4-st). Lisaks on katkendliku joonega noolega kujutatud kõrvalmõju ühest tundmatust teise.



Joonis 8: Võrrandisüsteemile joonisel 5 vastavad tundmatute omavahelised sõltuvused ja kõrvalmõjud.

Tuginedes joonisel 8 kujutatud sõltuvustele ja kõrvalmõjule, saab laskuva lahendaja üldist tööpõhimõtet — lahendada vaid nõutud programmipunkte — kujutada kasutades Baudouin Le Charlier ja Pascal Van Hentenryck kirjeldatud universaalset laskuvat lahendajat, mida on kohendatud töö jaoks [2]. Universaalsel lahendajal on defineeritud tabelid σ (tundmatute väärtused), `infl` (tundmatu mõjutused teistele tundmatutele) ja `called` (tundmatud, mida hetkel arvutatakse). Lahendamise alguses on kõik hulgad tühjad. Lahendaja võtab sisendiks huvipakkuva tundmatu `x` ning iga võrrandisüsteemi

parema poole arvutamisel, kui leitakse tundmatu `y`, lisatakse tundmatu hulka `called`, tabelis `infl` lisatakse `y`-ile kuuluvasse hulka `x`, ning lahendatakse `y` kohe, tulemuse saamisel eemaldatakse `y` hulgast `called`. Lahendamine jätkub kuni hulga `called` ja tabelite σ ja `infl` väärtused ei muutu võrreldes eelmise sammuga. Joonisel 8 kujutatud sõltuvuste ja joonisel 5 esitatud tundmatute paremate poolte põhjal, alustades huvipakkuvast tundmatust x_3, saadakse universaalse laskuva lahendaja korral joonisel 9. Kuna lahendaja ei lahenda tundmatuid x_2 ja x_1, on need jooniselt välja jäetud.



Joonis 9: Võrrandisüsteemi joonisel 5 lahendamine sammhaaval.

Joonisel 9 sammul 1 alustatakse kitsendussüsteemi lahendamist tundmatust x_3. Sammul 2 arvutatakse x_3 paremat poolt kuni jõutakse tundmatuni x_4, seejärel hakatakse kohe lahendama x_4 paremat poolt (samm 3). Jõutakse 3. sammul x_4 võrrandis punkti $get\ g$, misjärel tagastatakse kohe selle väärtus. Sammudel 4 ja 5 on tundmatute väärtused asendatud eelmiste sammude tulemustega. Sammul 6 tagastatakse kitsendussüsteemile leitud lahend. Joonisel 9 on kujutatud igal sammul hulga `called` ja tabeli `infl` väärtust, ning tabeli σ korral on iga tundmatute väärtuste esitatud tabelis 3 eraldi reana.

Tabel 3: Kõrvalmõjudega kitsendussüsteemi (joonis 5) hulkade `called`, `infl` ja iga tundmatu väärtus igal sammul, kui lahendatakse kohandatud universaalse laskuva lahendajaga.

	1	2	3	4	5	6
`called`	$\{x_3\}$	$\{x_3, x_4\}$	$\{x_3, x_4, g\}$	$\{x_3, x_4\}$	$\{x_3\}$	$\emptyset$
`infl`	$\emptyset$	$\{x_4 \to x_3\}$	$\{x_4 \to x_3, g \to x_4\}$	...	...	$\{x_4 \to x_3, g \to x_4\}$
x_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Kuna laskuvad lahendajad lahendavad tundmatuid nõudmise baasil, ei pruugi kõikidele kitsendussüsteemi tundmatutele tulemust arvutada. Lahendite käsitlemisel tuleb arvestada lahendaja poolt tagastatava tundmatute väärtustuste kui ka hulgaga, mis kirjeldab tundmatuid, millele on leitud ülelähend. Laskuvad lahendajad seega tagastavad paari $(\sigma, stable)$, kus $stable$ on hulk tundmatuid, mille puhul on saavutatud hea ülehinnang nende tegelikust väärtusest. Selliseid lahendeid nimetatakse osalisteks ülelähenditeks.

2.2 TD_{side} lahendaja käitumine

TD_{side} lahendaja (ingl TD_{side} *solver*) põhineb L. Charlier ja P. Van Hentenrycki poolt kirjeldatud laskuval lahendajal [2]. Laskuvat lahendajat on täiendatud kahes aspektis: lisatud on laiendamine ja kitsendamine, mis garanteerib termineerumise ka mittemonotoonsete võrrandisüsteemide korral ning täiendatud toetamaks kõrvalmõjudega võrrandisüsteeme, mis on ühe väljundina võimaldanud analüüsida mitmelõimelisi programme [3].

Lahendaja meetodite ülevaade tugineb Helmut Seidl ja Ralf Vogleri artiklile [3]. TD_{side} (lisa I) koosneb neljast meetodist: `destabilize`, `eval`, `solve` ja `side`. TD_{side} lahendaja algab `entry` väljakutsest programmipunktiga, mida soovitakse analüüsida. Programmide analüüsis on tavaliselt selleks üks alguspunkt — programmi funktsiooni `main` lõpp-punkt (joonise 6 näitel punkt number 6). Meetodis `solve` teostatakse lahendi leidmist ühe tundmatu jaoks. Meetodis on kirjeldatud tundmatu paremat poolt kirjeldav abstraktne funktsioon $f_x^\sharp$, milles kasutatakse meetodeid `eval` ja `side`. Meetodi `eval` kaudu päritakse teisi sõltuvaid tundmatuid ning jälgitakse, mis tundmatud üksteisele mõju avaldavad. Meetod `side` abil saab uuendada teiste tundmatute väärtusi esialgse tundmatu väärtuse arvutamise vältel. Tundmatu väärtuste muutumisel funktsioon `destabilize` eemaldab tundmatu ja sellest sõltuvad tundmatud stabiilsete tundmatute hulgast, tagades järgmisel analüüsi iteratsioonil tundmatu uuesti analüüsimise. Analüüs jätkub kuni tundmatute väärtused enam ei muutu ehk on leitud ülelähend tervele kitsendussüsteemile.

Tuginedes Helmut Seidl ja Ralf Vogleri artiklis kirjeldatud TD_{side}i implementatsioonile [3] tuleb lahendaja samm-sammult kirjeldamiseks tuleb esmalt selgitada hulkasid ja tabeleid (joonisel 10), mis on algoritmi igas etapis kättesaadavad ja muudetavad:

1. σ – tabel, programmipunktid ja nende väärtused;
2. `infl` – tabel, mille võti on programmipunkt ning igale võtmele vastab hulk programmipunkte, mis sõltuvad võtmest;
3. `called` – hulk, mida kasutatakse, et eristada arvutamise vältel olevaid programmipunkte;
4. `stable` – hulk programmipunktidest, mis on saavutanud hulka lisamise hetkel hea ligikaudse (ingl *good approximation*) tulemuse programmipunktile;
5. `point` – hulk kõikide programmipunktidega, kus kasutatakse laiendamist;
6. `leaf` – hulk, mis kirjeldab kõiki programmipunkte, millel ei ole kõrvalmõjusid teistesse programmipuntidesse ja mis ei sõltu teistest programmipunktidest, näiteks joonise 1 puhul oleks sellisteks programmipunktiks funktsiooni `t_foo` alguspunkt number 7.

```
1  val σ       : (𝒳, 𝔻) Map.t
2  val infl    : (𝒳, 𝒳 Set.t) Map.t
3  val called  : 𝒳 Set.t
4  val stable  : 𝒳 Set.t
5  val point   : 𝒳 Set.t
6  val leaf    : 𝒳 Set.t
7  val Set.create  : unit -> 'a Set.t
8  val Map.create  : (unit -> 'b) -> ('a, 'b) Map.t
9
10 val (!) : ('a, 'b) Map.t -> 'a -> 'b
11
12 val (:=) : ('a, 'b) Map.t * 'a -> 'b -> unit
13 val (∈)  : 'a -> 'a Set.t -> bool
14 val (+=) : 'a Set.t -> 'a -> unit
15 val (-=) : 'a Set.t -> 'a -> unit
16
17 let stable = Set.create ()
18 let called = Set.create ()
19 let point  = Set.create ()
20 let infl   = Map.create Set.create
21 let σ      = Map.create (fun () -> ⊥)
```

Joonis 10: TD_{side} hulkade ja tehete definitsioonid OCamli-stiilis pseudokoodis.

Tuginedes Helmut Seidl ja Ralf Vogleri artiklis kirjeldatud TD_{side}i implementatsioonile [3], alustatakse algoritmi meetodite selgitamist lahendaja alguspunktist (joonisel 11 ning seejärel kirjeldatakse ülejäänud meetodid lahti:

1. funktsioon `entry` (joonisel 11), argumendiks on huvipakkuv programmipunkt `x`, funktsioon lahendab `x`-i ning tagastab lahendamise lõppedes hulgad σ (programmipunktide väärtused) ning `stable` (kõik hea ligikaudse tulemuse saavutanud programmipunktid);

```
70 let entry x = point += x;
71               solve ∇ x;
72               (σ, stable)
```

Joonis 11: TD_{side} meetod entry OCamli-stiilis pseudokoodis.

2. funktsioon `solve` (joonisel 12), argumendid on `p`, laiendamis- või kitsendamisoperaator, ja `x`, lahendatav programmipunkt; funktsioon arvutab `x`-ile vastava võrrandisüsteemi paremat poolt, juhul kui seda juba ei arvutata ja ei ole hea ligikaudne tulemus saavutanud; kui lahendamise vältel `x` ei saavuta ligikaudse tulemus lahendab seda uuesti laiendades, kui aga laiendamisega ei muutu tulemus, lahendatakse kitsendades (eeldusel, et ei ole eelmine samm juba kitsendatud, muul juhul lahendatakse aktiivse `p` järgi;

```
47 and solve p x =
48     if x ∉ stable && x ∉ called then (
49         stable += x;
50         called += x
51         let tmp = f♯_x (eval x) side in
52         called -= x;
53         let tmp =
54             if x ∈ point then p (!σ x) tmp
55             else tmp
56         in
57         if x ∉ stable then solve ∇ x
58         else if !σ x = tmp then
59             if p = ∇ && x ∈ point then (
60                 stable -= x;
61                 solve Δ x
62             )
63         else (
64             σ, x := tmp;
65             destabilize x;
66             solve p x
67         )
```

```
68      )
```

Joonis 12: TD_{side} meetod solve OCamli-stiilis pseudokoodis.

3. funktsioon `eval` (joonisel 13), argumendid on `x` ehk programmipunkt, mille võrrandisüsteemi paremat poolt hetkel lahendatakse, ja `y`, mis on programmipunkt, mida `x`-i võrrandisüsteemi paremas pooles kasutatakse; funktsioon lisab `y`-i hulka `point`, kui seda hetkel juba lahendatakse või on tegu `leaf`-i kuuluva programmipunktiga, muul juhul lahendatakse `y`, ning lisatakse `y`-i sõltuvuseks `x` ja tagastatakse `y` arvutatud väärtus;

```
30 let rec eval x y =
31     if y ∈ (called ∪ leaf) then
32         point += y;
33     else
34         solve ∇ y;
35     !infl y += x;
36     !σ y
```

Joonis 13: TD_{side} meetod eval OCamli-stiilis pseudokoodis.

4. funktsioon `side` (joonisel 14), argumendid on `x` ehk programmipunkt, mille võrrandisüsteemi paremat poolt hetkel lahendatakse ja `d` ehk väärtus, mis mõjutab programmipunkti `x`, kusjuures `x`-i paremas pooles võib `d` puhul olla konstantne väärtus, kui ka teise programmipunkti väärtus; funktsioon arvutab uue väärtuse `x`-ile, rakendades laiendamist seni arvutatud väärtuse ja mõjutuse peal, kui `x`-i uus väärtus erineb eelmisest uuendatakse `x`-i ja `x` loetakse stabiilseks, kuid seda mõjutavad programmipunktid eemaldatakse `stable`-ist:

```
38 and side x d =
39     let tmp = (!σ x) ∇ d in
40     if (!σ x) ≠ tmp then (
41         σ, x := tmp;
42         stable += x;
43         destabilize x
44     )
```

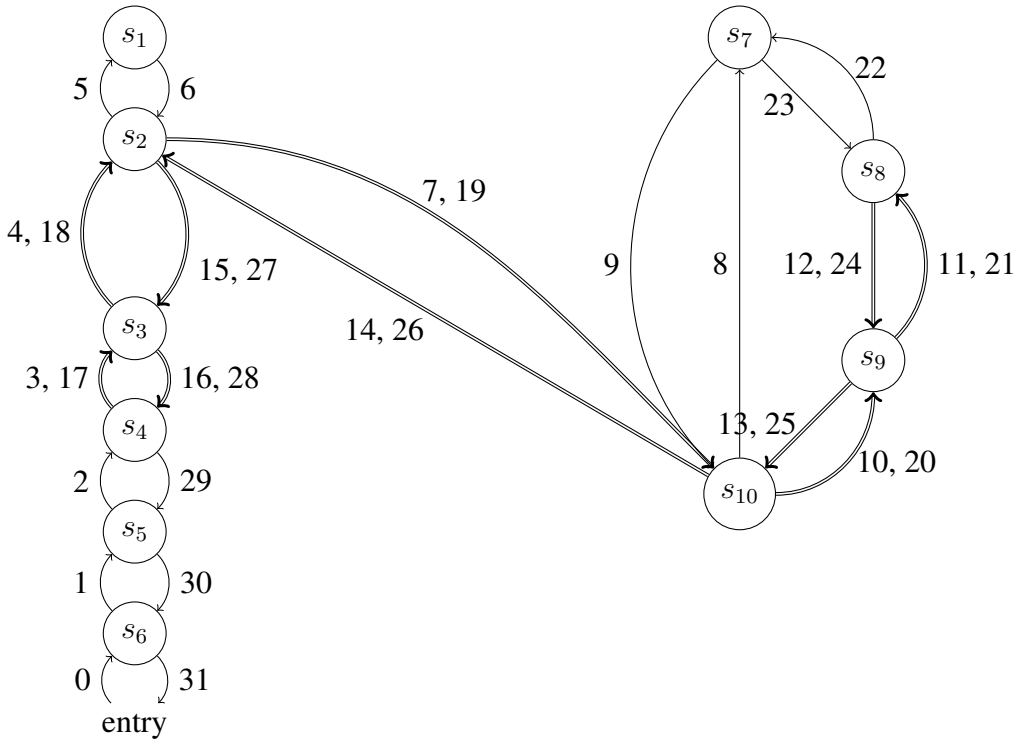
Joonis 14: TD_{side} meetod side OCamli-stiilis pseudokoodis.

5. funktsioon `destabilize` (joonisel 15), argumendiks on tundmatu `x`; funktsioon eemaldab `x`-ist sõltuvad tundmatud hulkadest `infl` ja `stable`, ning juhul kui eemaldatuid tundmatuid ei lahendata väljakutse hetkel, kutsutakse `destabilize` nendega välja.

```
22 let rec destabilize x =
23     let w = !infl x in
24     infl, x := Set.create ();
25     Set.iter (fun y ->
26         stable -= y;
27         if y ∉ called then destabilize y
28     ) w
```

Joonis 15: TD_{side} meetod destabilize OCamli-stiilis pseudokoodis.

Vaadeldakse TD_{side} algoritmi tööd näidisvõrrandisüsteemi peal (joonisel 7), lahendaja samme kujutatakse sarnaselt universaalse laskuva lahendajale (joonisel 9). TD_{side} `entry` funktsioonile antakse sisendiks programmipunkt $[s_6]$ ehk programmi lõpp-punkt. Globaalsete muutujate väärtuseks on intervallide võre elemendid.



Joonis 16: TD_{side} programmipunktide lahendamise järjekord.

Joonisele 16 vastab tabel 4, mis kirjeldab muutujate g, h ja j väärtuseid ja hulka $infl$. Tabelis on loetavuse huvides hulga $infl$ korral viidatud sammule kujul "(n)", kus n on samm millele viidatakse. Viitamise korral on $infl$ väärtuseks viidatud sammu ja praeguse ühend.

Tabel 4: TD_{side} kitsendussüsteemi (joonis 5) lahendamise $infl$ ja tundmatute g, h, j väärtused igal sammul, kus need muutuvad.

	g	h	j	$infl$
0	$\bot$	$\bot$	$\bot$	$\emptyset$
1	$\bot$	$\bot$	$\bot$	$h \to [s_6]$, $j \to [s_6]$
⋮	$[0,0]$	$[0,0]$	$[0,0]$	(1)
6	$[0,0]$	$[0,0]$	$[0,0]$	(1), $[s_1] \to [s_2]$
⋮	$[0,0]$	$[0,0]$	$[0,0]$	(6)
9	$[0,0]$	$[0,0]$	$[0,0]$	(6), $[s_7] \to [s_{10}]$
10	$[0,0]$	$[0,0]$	$[0,0]$	(9), $[s_g] \to [s_9]$
11	$[0,0]$	$[0,0]$	$[0,0]$	(10)
12	$[0,0]$	$[0,0]$	$[0,0]$	(10), $[s_8] \to [s_9]$, $g \to [s_8]$
13	$[0,0]$	$[0,0]$	$[0,0]$	(12), $[s_9] \to [s_{10}]$
14	$[0,0]$	$[0,0]$	$[0,0]$	(13), $[s_{10}] \to [s_2]$
15	$[0,0]$	$[0,0]$	$[0,0]$	(14), $[s_2] \to [s_3]$
16	$[0,1]$	$[0,0]$	$[0,0]$	(15), $[s_3] \to [s_4]$
17	$[0,1]$	$[0,0]$	$[0,0]$	$h \to [s_6]$, $j \to [s_6]$, $[s_1] \to [s_2]$, $[s_7] \to [s_{10}]$
⋮	$[0,1]$	$[0,0]$	$[0,0]$	(17)
20	$[0,1]$	$[0,0]$	$[0,0]$	(17), $[s_g] \to [s_9]$
22	$[0,1]$	$[0,0]$	$[0,0]$	(20), $[s_g] \to [s_8]$
23	$[0,1]$	$[0,0]$	$[0,0]$	(22), $[s_7] \to [s_8]$
24	$[0,1]$	$[0,0]$	$[0,0]$	(23), $[s_8] \to [s_9]$
25	$[0,1]$	$[0,1]$	$[0,0]$	(24), $[s_9] \to [s_{10}]$
26	$[0,1]$	$[0,1]$	$[0,1]$	(25), $[s_{10}] \to [s_2]$
27	$[0,1]$	$[0,1]$	$[0,1]$	(26), $[s_2] \to [s_3]$
28	$[0,1]$	$[0,1]$	$[0,1]$	(27), $[s_3] \to [s_4]$
29	$[0,1]$	$[0,1]$	$[0,1]$	(28), $[s_4] \to [s_5]$
30	$[0,1]$	$[0,1]$	$[0,1]$	(29), $[s_5] \to [s_6]$
31	$[0,1]$	$[0,1]$	$[0,1]$	(30)

Joonise 16 ja tabeli 4 põhjal tõstetakse esile lahendaja omadust laskuvalt lahendada:

- lahendaja liigub sammudel 1–5 programmi lõpp-punktist algusesse ning sealt hakkab arvutatud tulemusi tagastama (samm 6);
- lõime nõudmisel (sammul 7) läbitakse lõim laskuvalt sammudel 8–14;
- g väärtuse muutumisel, minnakse ümber arvutama programmipunkte, millest $[s_4]$ sõltus ehk laskutakse kuni lõime loomiseni ning arvutatakse uuesti ümber lõime (sammud 17–28).

2.3 TD_{side} korrektsus

Kitsendussüsteeme lahendavatel algoritmidel on oluline kinnitada nende korrektsus ehk kui algoritm tagastab vastuse, on vastus korrektne lähend (ingl *sound approximation*) konkreetse võrrandisüsteemi vähimast osalisest lahendusest (ingl *least partial solution*). Vaadeldakse Helmut Seidl ja Ralf Vogleri [3] teoreemi TD_{side} korrektsusest:

Teoreem 1. Eeldame, et $E^\sharp$ on kõrvalmõjudega abstraktsete võrrandite süsteem ja x on tundmatu süsteemis $E^\sharp$. Eeldame, et hulgad $stable$ ja $called$ on tühjad ja tabelis $infl$ iga tundmatu väärtuseks on $\emptyset$. Eeldame, et `solve` ∇ `x` arvutatakse TD_{side} korral $E^\sharp$ põhjal. Olgu σ ja $stable$ andmestruktuuride tulemusi kirjeldavad peale algoritmi lõppemist. Siis $x \in stable$ ja $(\sigma, stable)$ on $E^\sharp$-i parandatud osaline ülelähend.

Teoreem TD_{side} korrektsusest kirjeldab, et algolekus algoritm, alustades võrrandisüsteemi $E^\sharp$ tundmatust x , lahendab abstraktsete võrrandite süsteemi ning tagastab paari $(\sigma, stable)$. Funktsioon σ määrab igale tundmatule väärtuse, ning hulk $stable$ kirjeldab kõiki tundmatuid, mis kindlalt ülehindavad (ingl *over-approximate*) tundmatu tegelikku väärtust.

2.4 Kõrvalmõjudega kitsendussüsteemide ebatõhusus

Kõrvalmõjudega kitsendussüsteemide lahendamisel esineb ebatõhusus lõimede analüüsis — lõime looja sõltub loodud lõimest — mis on tingitud *top-down solver*i omadusest lahendada vaid nõutud võrrandisüsteeme. Probleem ilmneb kui peale loodud lõime analüüsi lõppemist analüüsitakse programmipunkte, mis mitmel taasarvutamisel ei muutu, ning kohatakse kõrvalmõju loodud lõimes. Kõrvalmõjudega kitsendussüsteemide omadusest väärtustada muutujaid sõltumata, mis punktis on jooksev analüüs, tingib olukorra, kus tuleb taasarvutada programmipunkte, mille tulemus ei muutu. Näiteks joonisel 1 kujutatud programmi korral toimuv analüüs algab funktsiooni `main` lõpus sammul `return 0`, jõudes sammuni `pthread_create` luuakse uus lõim `t_foo`, millest eelnev samm `int a = 1` määratakse sellest sõltuvaks. Analüüs jätkub kuni funktsioonide definitsioonide alguseni – esialgu analüüsitakse läbi lõim, seejärel `main`, mille järel jõudes sammu `g = 1` lahendamiseni tuleb kõrvalmõjude tõttu uuesti lahendada lõim, mis `g` väärtusest sõltub. Ebatõhusus seisneb selles, et tuleb lõime loomispunkti ja seda mõjutanud sammu vahel olevaid käske uuesti analüüsida (joonisel 1 sammu `int a = 1`).

Ebatõhusust vaadeldakse joonisel 16, kus sammul 16, peale `g` väärtustamist lahendaja eemaldab hulgast `stable` kõik programmipunktid, millest $[s_4]$ sõltus. Eemaldatakse seeläbi programmipunkt $[s_3]$, mille väärtus mitme lahendamisega ei muutu, kuid taasarvutamisega tuleb uuesti lahendada.

3 Uus nõrkade sõltuvustega lahendaja TD_{weak}

Peatükis tutvustatakse nõrkasid sõltuvusi. Esimeses alampeatükis kirjeldatakse nõrkade sõltuvuste tööpõhimõtet, tingimusi, mida nõrgad sõltuvused peavad täitma, ning täiendusi nii kõrvalmõjudega kitsendussüsteemidele kui ka TD_{side} lahendajale. Teine alampeatükk esitab tõestused tingimustele, mida on võimalik kinnitada ilma võrdlusanalüüsita. Viimases alampeatükis on välja toodud võrdlusanalüüsi tulemused ning järeldused.

3.1 Nõrkade sõltuvuste tööpõhimõte

Nõrgad sõltuvused on viis kirjeldada lõimi loovate programmipunktide sõltuvusi loodavatest lõimedest nii, et need eristuksid teistest sõltuvustest. Eesmärgiks on eemaldada tugev seos loodud ja loodava lõime vahel, et vältida TD_{side} lahendaja puhul lõime analüüsist sõltumatute programmipunktide taasarvutamist, vähendades seeläbi kogu analüüsi parema poole lahendamiste arvu. Nõrkade sõltuvuste lisamisel tuleb, sarnaselt kõrvalmõjudega kitsendussüsteemidele, tagada korrektsus ning keskmiselt vähendada võrrandisüsteemide paremate poolte arvutamiste arvu.

Nõrkade sõltuvuste korral täiendatakse kõrvalmõjudega kitsendussüsteemide igat paremat poolt lisades uue meetodi $demand$ kujul $V \to unit$ ehk võrrandisüsteemi paremaks pooleks saadakse funktsioon $f \in (V \to \mathbb{D}) \to (V \to \mathbb{D} \to unit) \to (V \to unit) \to \mathbb{D}$. Funktsionaalsel kujul kirjutatakse **fun** $get\ set\ demand \to e$. Täiendatakse näidisprogrammile (joonisel 1) loodud kõrvalmõjudega kitsendussüsteemi (joonisel 7), asendades mõjutatud programmipunkti $[s_2]$ võrratuse uuega (joonisel 17).

$$[s_2] \sqsupseteq \textbf{let}\ d = get\ [s_1]\ \textbf{in}\ set\ [s_7]\ \varnothing;\ demand\ [s_{10}];\ d$$

Joonis 17: Juhtvoograafile (joonis 6) vastav nõrkade sõltuvustega ning kõrvalmõjudega kitsendussüsteem.

Nõrkade sõltuvustega täiendatud laskuvale lahendajale viidatakse edaspidi kui TD_{weak}. Laskuvasse lahendajasse TD_{weak} lisatakse uus muutuja `weak`, mille väärtuseks määratakse analüüsi alguses tühi hulk (joonisel 18).

```
1 val weak : χ Set.t
2 let weak = Set.create ()
```

Joonis 18: Nõrgade sõltuvuste hulkade ja definitsioonide täiendused lisatud TD_{weak}-i, kirjutatud OCamli-stiilis pseudokoodis.

Nõrkade sõltuvuste kasutamisel saadakse realiseerida lahendajat kahel viisil:

1. laisa (ingl *lazy*) variatsioonina (lisa II) - kus hulka `weak` lisatud tundmatud lahendatakse alles siis kui kõik teised programmipunktid on hea ligikaudse tulemuse saavutanud;

2. ahne (ingl *eager*) variatsioonina (lisa III) - kus hulka `weak` lisamise järel kohe lahendatakse programmipunkt, ning kui peaks lahendaja lõpus leiduma ebastabiilseid programmipunkte, lahendatakse need ka lahendaja lõpus, oluline erinevus `eval`-ist on see, et hulka `infl` ei lisata sõltuvust.

Vaadeldakse esialgu laiska variatsioon lahendajast TD_{weak}. Laisa variatsiooni korral luuakse uus meetod `demand`, kus lisatakse hulka `weak` nõutud programmipunkt (joonisel 19).

```
25 let demand x y =
26     weak += y;
```

Joonis 19: TD_{weak} laisa variatsiooni funktsioon `demand`, kirjutatud OCamli-stiilis pseudokoodis.

TD_{weak} ahne variatsiooni funktsioon `demand` lisab programmipunkti hulka `weak` ning lahendab seda, kutsudes meetodit `solve` (joonisel 20).

```
25 let demand x y =
26     weak += y;
27     solve ∇ y;
```

Joonis 20: TD_{weak} ahne variatsiooni funktsioon `demand`, kirjutatud OCamli-stiilis pseudokoodis.

Mõlema variatsiooni puhul täiendatakse funktsiooni `entry`, kontrollides, kas hulka `weak` lisatud programmipunkt `w` on hea ligikaudse tulemuse saavutanud. Kui ei ole, lahendatakse programmipunkti uuesti, kutsudes funktsiooni `entry` välja `w`-ga. Sarnaselt kontrollitakse ja lahendatakse programmipunkti `x` (joonisel 21).

```
75 let rec entry x = point += x;
76                   solve ∇ x;
77                   Set.iter (fun w ->
78                      if w ∉ stable then entry w;
79                   ) weak;
80                   if x ∉ stable then entry x
81                   else
82                   (σ, stable)
```

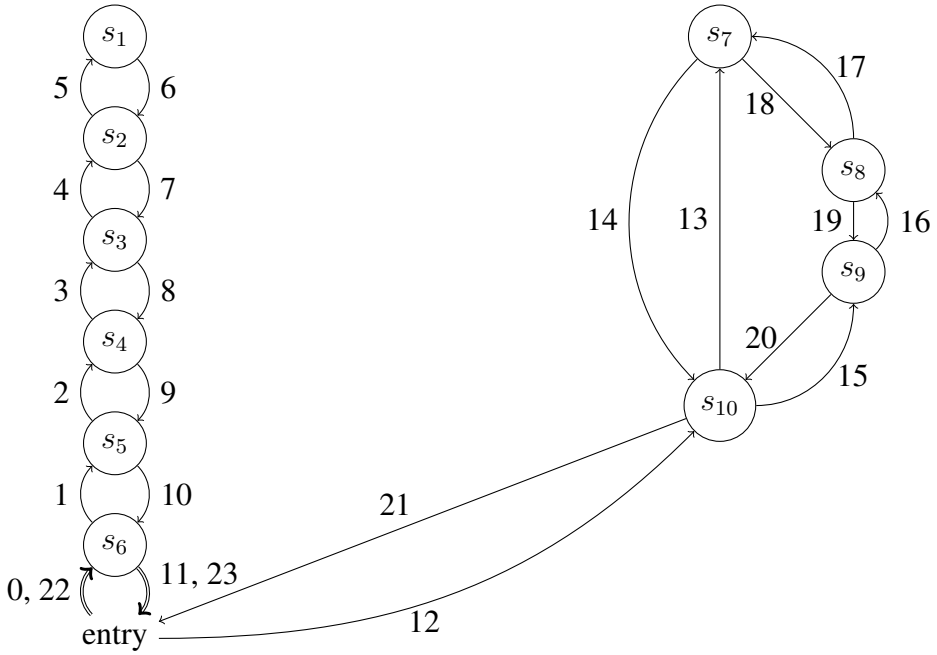
Joonis 21: TD_{weak} funktsioon `entry`, kirjutatud OCamli-stiilis pseudokoodis.

Viimase muudatusena lisatakse funktsioonis `solve` võrrandisüsteemi parema poole funktsioonile kolmanda parameetrina funktsioon `demand` (joonisel 22).

```
52 and solve p x =
53     if x ∉ stable && x ∉ called then (
54         stable += x;
55         called += x
56         let tmp = f♯_x (eval x) side demand in
57         called -= x;
58         let tmp =
59             if x ∈ point then p (!σ x) tmp
60             else tmp
61         in
62         if x ∉ stable then solve ∇ x
63         else if !σ x = tmp then
64             if p = ∇ && x ∈ point then (
65                 stable -= x;
66                 solve Δ x
67             ) else ()
68         else (
69             σ, x := tmp;
70             destabilize x;
71             solve p x
72         )
73     )
```

Joonis 22: TD_{weak} funktsiooni `solve` täiendus, kirjutatud OCamli-stiilis pseudokoodis.

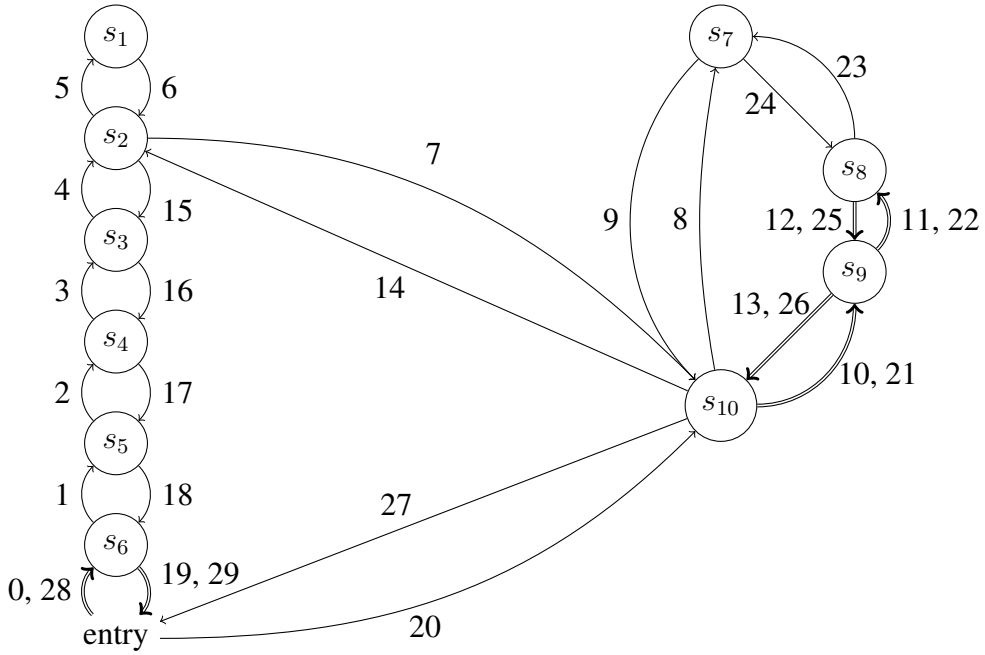
Saadud algoritmi variatsioonid võimaldavad lahendada nõrkade sõltuvustega kitsendussüsteeme. Garanteeritud on tugeva sõltuvuse eemaldamine lõime loova programmipunkti ning lõime vahel. Vaadeldakse näidisprogrammi läbimist TD_{weak} laisa variatsiooniga (joonisel 23).



Joonis 23: TD_{weak} laisa variatsiooni programmipunktide lahendamise järjekord.

Lahendades kitsendussüsteemi TD_{weak} laisa variatsiooniga, märgati järgmist:

1. lahendaja laskub sammudel 1–5 funktsiooni `main` algusesse ning tagastab kohatud paremate poolte tulemused sammudel 6–11, seeläbi muutuja `g` väärtuseks määratakse sammul 6 $[0, 0]$ ning täiendatakse tulemuseks $[0, 1]$ sammul 9;
2. sammul 7 lisatakse lõime lõpp-punkt $[s_{10}]$ hulka `weak`;
3. peale sammu 11 on $[s_6]$ lisatud hulka `stable`, kuid $[s_{10}]$ ei ole, ning hakatakse lõime tulemust arvutama;
4. sammudel 12–21 läbitakse iga lõimele vastava tundmatu paremaid pooli ühe korra;
5. sammudel 22 ja 23 läbitakse uuesti $[s_6]$, kuna sammudel 19 ja 20 täiendati muutujate `h` ja `j` väärtusi;
6. peale sammu 23 on leitud osaline ülelähend tervele kitsendussüsteemile.



Joonis 24: TD_{weak} ahne variatsiooni programmipunktide lahendamise järjekord.

Lahendades kitsendussüsteemi TD_{weak} ahne variatsiooniga, märgati järgmist:

1. lahendaja laskub sammudel 1–5 funktsiooni `main` algusesse ning sammul 6 tagastab $[s_1]$ parema poole väärtuse;
2. sammul 7 lisatakse lõime lõpp-punkt $[s_{10}]$ hulka `weak` ning hakatakse lahendama lõime;
3. sammudel 7–14 läbitakse lõime esimest korda;
4. sammudel 15–19 läbitakse tagastatakse funktsiooni `main` programmipunktide paremate poolte tulemused, kusjuures sammul 17 täiendatakse muutuja `g` väärtust, aga lõime ja `main` vahelise sõltuvuse puudumise tõttu eemaldatakse lõime programmipunkte hulgast ainult `stable`;
5. sammudel 20–27 taasarvutatakse lõime programmipunktid, mis eemaldati hulgas `stable`;
6. sammudel 28 ja 29 läbitakse uuesti $[s_6]$, kuna sammudel 25 ja 26 täiendati muutujate `h` ja `j` väärtusi;
7. sammuks 29 leitakse osaline ülelähend tervele kitsendussüsteemile.

Mõlema TD_{weak} variatsiooni puhul välditakse programmipunkti $[s_3]$ ümberarvutamist ja vähendatakse paremate poolte lahendamiste arvu. Kui TD_{side} lahendas kitsendussüsteemi 31 sammuga (joonisel 16), saavutavad TD_{weak} variatsioonid parema tulemuse — TD_{weak} ahne lahendab 29 sammuga (joonisel 24), TD_{weak} laisk lahendab 23 sammuga (joonisel 23).

3.2 TD_{weak} korrektsus

TD_{weak} korrektsuse näitamiseks tuleb kinnitada, et peale algoritmide lõppu saadud lahend $(\sigma, stable)$ on osaline ülelahend abstraktsest võrrandisüsteemist $E^\sharp$. TD_{weak} variatsioonide korral saab kinnitada, et ühelõimeliste programmide korral käitub algoritm täpselt nagu TD_{side}, mitmelõimeliste puhul lisatakse loodava lõime lõpp-punkt hulka `weak` ja toimub järgnev:

- TD_{weak} ahne variatsioon lahendab kohe lisatud lõime lõpp-punkti;
- TD_{weak} laisk variatsioon peale lisamise midagi täiendavat ei tee.

Mõlema variatsiooni puhul, peale esialgse programmipunkti lahendamist, on kõik seni kohatud programmipunktid hulgas $stable \cup weak$. Kuna funktsioon `entry` peale programmipunkti lahendamist kontrollib, et $\{x\} \cup weak$ oleks hulgas $stable$, jätkub $\{x\} \cup weak$ olevate programmipunktide lahendamine, kuni kõik on hulgas $stable$, mistõttu saadakse teoreem:

Teoreem 2. Eeldame, et $E^\sharp$ on abstraktne võrrandite süsteem nõrkade sõltuvustega ja x on tundmatu süsteemis $E^\sharp$. Eeldame, et hulgad $stable$, $called$ ja $weak$ on tühjad ning $infl$ määrab igale tundmatule väärtuse $\emptyset$. Eeldame, et lahendame TD_{weak} variatsiooni funktsiooni `entry x` kutse. Olgu σ ja $stable$ algoritmi tulemus selle lõppedes. Siis, $x \in stable$ ja $\forall w \in weak, w \in stable$, ning $(\sigma, stable)$ on süsteemi $E^\sharp$ osaline ülelähend.

Tõestus. Olgu X võrrandisüsteemi $E^\sharp$ tundmatute hulk ja $x \in X$ huvipakkuv tundmatu.

Näitame TD_{weak} variatsioonide puhul järgmist:

1. laisa variatsiooni korral on võimalik teisendada võrrandisüsteem ümber kujule, mida saab lahendada TD_{side}-iga;
2. ahne variatsiooni korral tugineme laskuva lahendaja omadustele kõrvalmõjude korral taasarvutada ebastabiilseid tundmatuid ning olukorrale, kus mitme `solve` väljakutse korral ei muutu analüüsi tulemus valeks.

Käsitleme esmalt laiska variatsiooni. Kirjeldame lahendaja TD_{weak} funktsiooni $entry$ kutset kui TD_{side} poolt lahendatava tundmatu x' paremat poolt:

$$
\begin{aligned}
&f_{x'}\ get\ set = \\
&\quad \textbf{let}\ d = get\ x\ \textbf{in} \\
&\quad \textbf{for}\ w \in\ get\ [weak]: \\
&\qquad get\ w; \\
&\quad d
\end{aligned}
$$

Abitundmatuga tekib uus võrrandisüsteem $E'^{\sharp}$, mille tundmatute hulk on $X' = X \cup \{x', [weak]\}$, kus $[weak]$ väärtus on võrest 2^X, kus tundmatute hulgad on osaliselt järjestatud sisalduvuse alusel. Kõikide võrrandisüsteemi paremates pooltes, kus kasutame $demand$-i, asendame funktsiooniga $set\ [weak]\ \{y\}$ ehk iga tundmatu y lisatakse tundmatusse `[weak]`, sisuliselt matkides funktsiooni $demand\ set$-ina. Muudatused võimaldavad matkida TD_{weak}i laiska käitumist TD_{side}-ga.

Tuginedes teoreemile 1 teame, et lahendaja TD_{side} lahendab võrrandisüsteemi $E'^{\sharp}$ alustades tundmatust x' ning tagastab osalise ülelähendi ja $x' \in stable$ ning seeläbi on $x \in stable$ ja $\forall w \in get\ [weak], w \in stable$, kinnitades, et iga $demand$-iga nõutud tundmatu lahendatakse. Järelikult on TD_{weak} laisk variatsioon korrektne.

Teisalt käsitleme ahnet variatsiooni. Me saame kirjeldada ahnet variatsiooni kui laiska variatsiooni erinevusega, et funktsioonis `demand y` on täiendav `solve` ∇ `y`. Funktsiooni `solve` ∇ `y` kutse garanteerib, et selle järel `y` $\in stable$ ning hulka $stable \setminus called$ allesjäänud tundmatute väärtused ei ole muutunud valeks [18, Teoreem 6]. Kui lahendamise jooksul mingid tundmatud hulgast $stable$ eemaldatakse, millest endiselt sõltutakse, siis lahendatakse need tundmatud funktsioonis `entry`, kuni on kõik seni kohatud tundmatud hulgas $stable$. Järelikult on TD_{weak} ahne variatsioon korrektne. ■

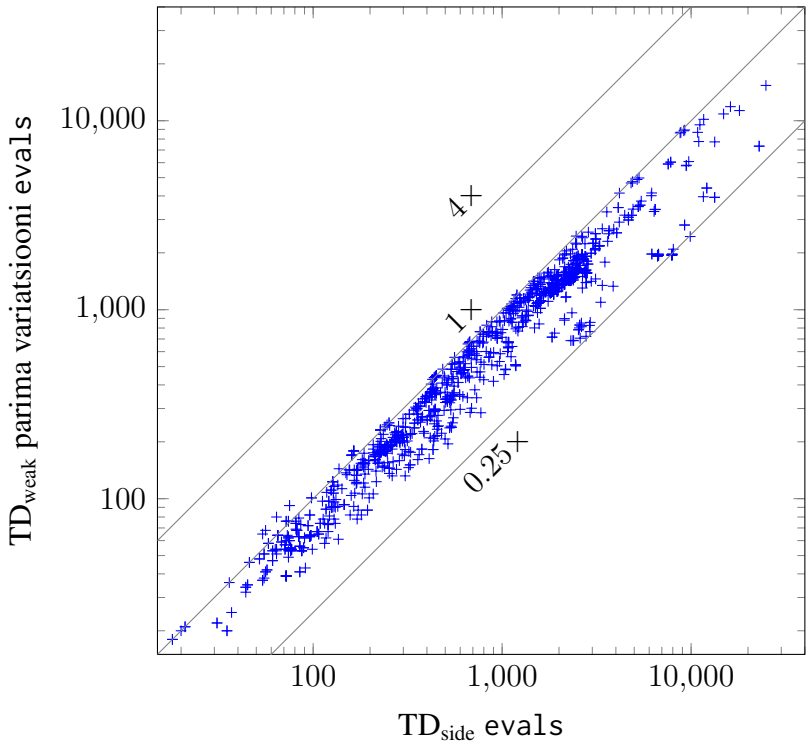
3.3 Võrdlusuuring

Nõrkade sõltuvuste lisamise eesmärk oli vähendada mitmelõimeliste programmide puhul programmipunktide taasarvutusi. Nõrgad sõltuvused implementeeriti Goblintis (lisa IV) ning võrdlusuuringuks (lisa V) kasutati 'SV-COMP 2025' andmestikku [19]. Iga testi korral anti lahendajatele 60 sekundit aega ning 1000 MB mälu. Võrdlusuuringus võrreldi Goblinti poolt kogutavat `evals` väärtust, mille väärtus oli analüüsi algul 0, kuid iga tundmatu parema poole lahendamisel tõsteti väärtust ühe võrra. TD_{weak} variatsioone võrreldakse TD_{side} vastu. Väärtust `evals` kasutatakse algoritmide tõhususe võrdlemiseks ajapõhiste meetodite asemel. Kuna väärtus ei sõltu keskkonnast ega riistvarast, saame analüüsi korrates alati sama tulemuse ning kindlamini esitada väiteid lahendajate tõhususe muutumise kohta.

’SV-COMP 2025’ andmestikust vaadeldi vaid teste, milles oli mitmelõimelisust kasutatud (3181 testi). Kuna täiendatud algoritmidesse lisatud `demand` meetodit ei kasutata ühelõimeliste programmide puhul, pole nende testide tulemused analüüsis asjakohased. Mitmelõimelistest testidest on välistatud testid, mille korral vähemalt üks kolmest lahendajast (TD_{side}, TD_{weak} ahne variatsioon või TD_{weak} laisk variatsioon) ei jõudnud ajavahemiku vältel analüüsi lõppu. Samuti on välistatud võrdlusest testid, mis ei tagastanud õiget vastuse võrreldes TD_{side}-iga. Kokku välistati analüüsis 1847 testi ning vaadeldi 1334 testi tulemust.

TD_{weak} laisa variatsiooni puhul saadi järgmised tulemused võrreldes TD_{side}-iga (joonisel 25):

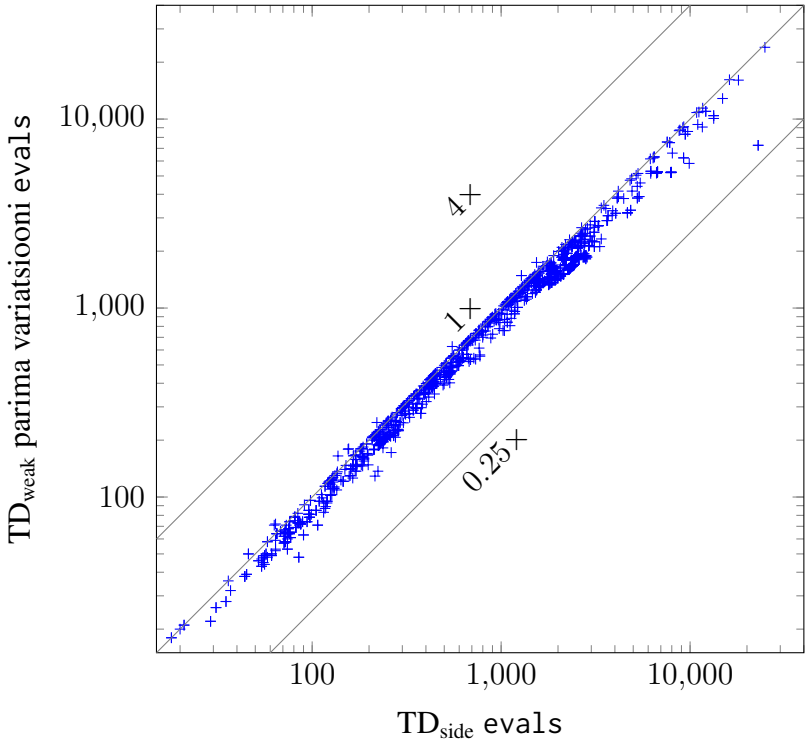
- keskmiselt kõikide testide peale paranes tulemus 28.87%;
- tulemus paranes 95.58% juhtudest (1275 testi), nende seas keskmiselt 30.37%;
- tulemus halvenes 1.87% juhtudest (25 testi), nende seas keskmiselt 8.36%;
- tulemus jäi samaks 2.55% juhtudest (34 testi).



Joonis 25: TD_{weak} laisa variatsiooni võrdlus TD_{side}-iga.

TD_{weak} ahne variatsiooni puhul saadi järgmised tulemused võrreldes TD_{side}-iga (joonisel 26):

- keskmiselt kõikide testide peale paranes tulemus 13.70%;
- tulemus paranes 94.98% juhtudest (1267 testi), keskmiselt 14.56%;
- tulemus halvenes 0.90% juhtudest (12 testi), keskmiselt 13.80%;
- tulemus jäi samaks 4.12% juhtudest (55 testi).

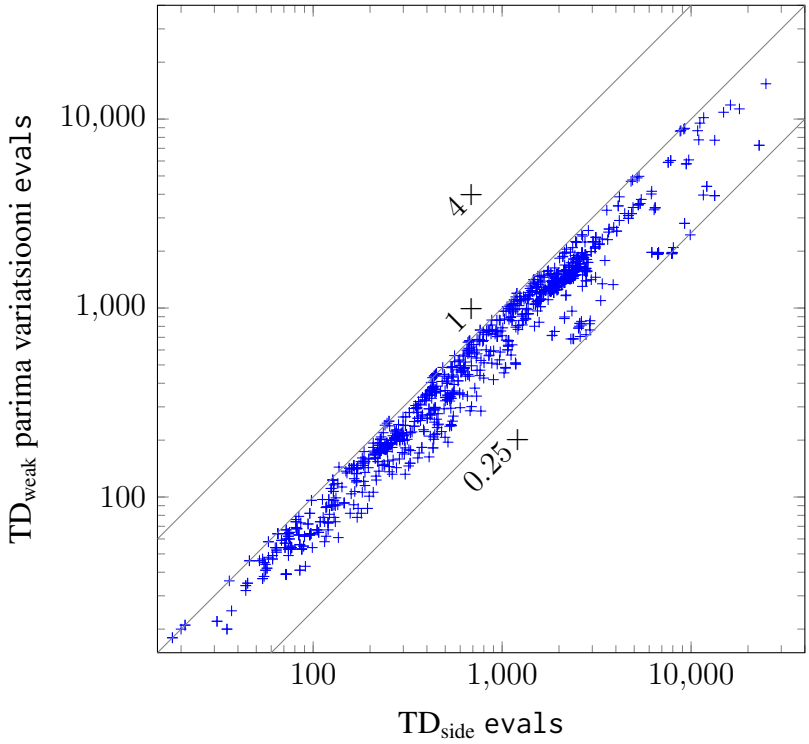


Joonis 26: TD_{weak} ahne variatsiooni `evals` väärtuste võrdlus TD_{side}-iga.

TD_{weak} mõlema variatsiooni seast parimat tulemust valides saadi järgmised tulemused võrreldes TD_{side}-iga (joonisel 27):

- keskmiselt kõikide testide peale paranes tulemus 29.35%;
- tulemus paranes 97.53% juhtudest (1301 testi), keskmiselt 30.10%;
- tulemus halvenes 0.07% juhtudest (1 testil, “goblint-regression/28-race_reach_46-escape_racefree.yml”), keskmiselt 5.11%;

- tulemus jäi samaks 2.40% juhtudest (32 testi) TD_{side}-iga võrreldes.

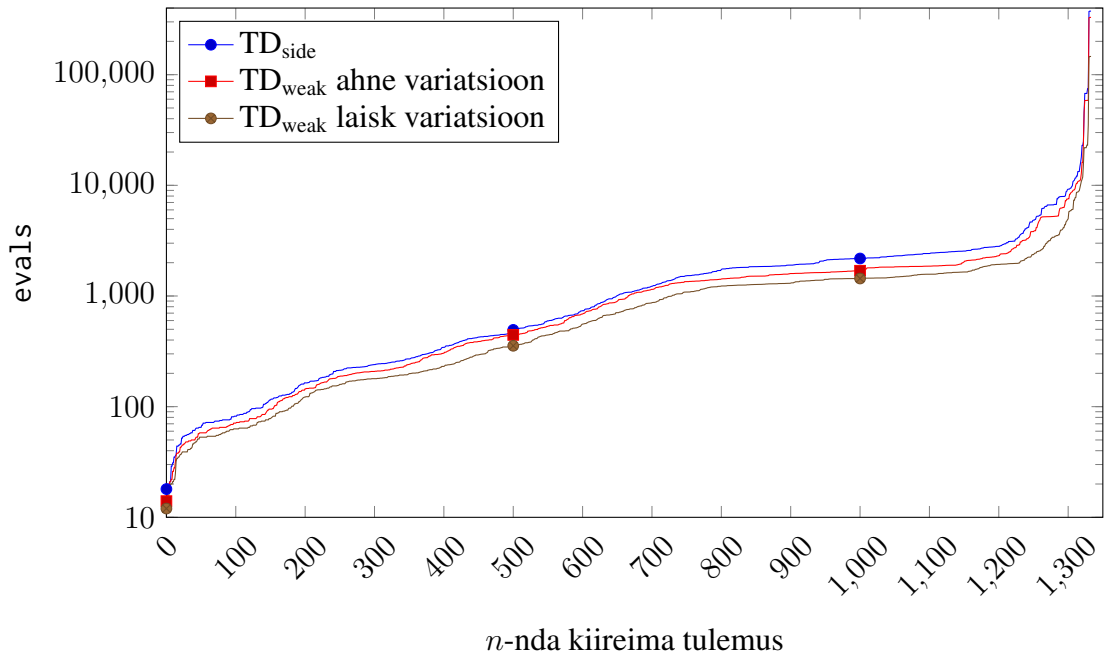


Joonis 27: TD_{weak} parima variatsiooni võrdlus TD_{side}-iga.

TD_{weak} variatsioone omavahel võrreldes:

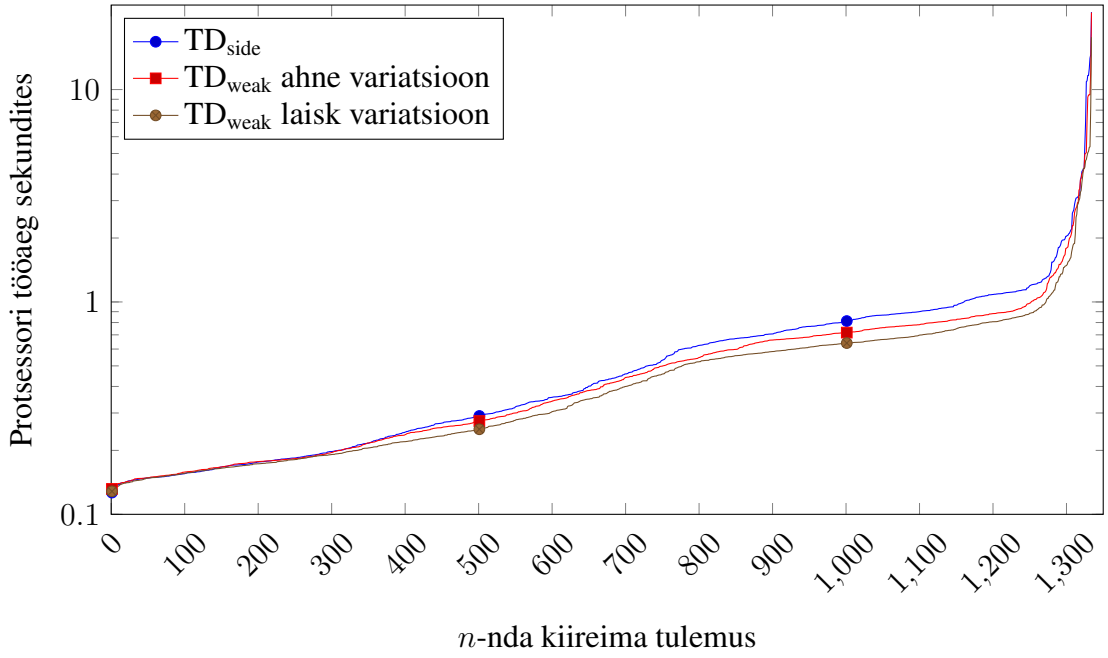
- laisa variatsiooni ja ahne variatsiooni tulemused jäid samaks 3.15% juhtudest (42 testi korral);
- laisk variatsioon oli ahnest variatsioonist parem 92.43% juhtudest (1233 testi korral);
- ahne variatsioon oli laisast variatsioonist parem 4.42% juhtudest (59 testi korral);

Järjestades testide tulemused `evals` väärtuse järgi, märgati ka tulemuste paranemist, kus TD_{side} nõudis rohkem taasarvutusi kui TD_{weak} ahne variatsioon ning see omakorda rohkem kui laisk variatsioon. Joonisel 28 kujutatakse kõikide laskuvate lahendajate testide `evals` väärtuseid kasvavas järjestuses. X-teljel kirjeldatakse n-ndat kiireimat testi ning y-teljel logaritmilisel skaalal `evals` arvu.



Joonis 28: Võrdlusuuringus kasutatud laskuvate lahendajate testide tulemused järjestatud `evals` järgi kasvavalt.

Kuigi lahendajate `evals` väärtust vaadeldi, et kindlamalt kinnitada erinevusi lahendajate tõhususes, on võimalik esile tuua ka ajakulu erinevus testide analüüsimisel. Võrdlusanalüüsi üheks väljundiks oli protsessori tööaeg, mille põhjal lahendajate tulemusi kasvavalt järjestades saadi joonis 29. Joonise x-teljel kirjeldatakse n-ndat kiireimat testi ning y-teljel logaritmilisel skaalal protsessori tööaega sekundites. Tulemuste põhjal saab väita, et nii joonisel 28 kui ka joonisel 29 saab täheldada sarnast paranemist TD$_{weak}$ variatsioonide korral õigustades `evals` kasutust ajapõhiste meetodite asemel.



Joonis 29: Võrdlusuuringus kasutatud laskuvate lahendajate testide tulemused järjestatud protsessori tööaja järgi kasvavalt.

Tulemuste põhjal saab väita, et mõlemad TD$_{weak}$ variatsioonid keskmiselt vähendasid arvutuste hulka vaadeldava testhulga peal. Suurimat muutust on märgata TD$_{weak}$ laisa variatsiooni puhul, kus keskmiselt langes arvutuste hulk 30.37%. Variatsioone omavahel võrreldes on niisamuti näha, et laisk variatsioon saavutab 92.43% juhtudest parema tulemuse kui ahne variatsioon. Seega saab väita, et TD$_{weak}$ laisk variatsioon on parem kui ahne.

3.4 Kitsaskohad

Mõlema kombinatsiooni parima tulemuse kombineerimisel leidus üks test, mis ei saavutanud paremat tulemust (lähtekood joonisel 30). Testi “goblint-regression/28-race_reach_46-escape_racefree.yml” puhul on tegu programmiga, kus meetodis `main` luuakse lõim, mille argumendiks antakse muutuja `i` viit. Lõimes kasutatakse muutuja `i` viita muteksi lukustamise ja avamise vahel ning meetodis `main` kontrollitakse, et pole esinenud trügimist muteksi lukustamise ja avamise vahel. Viimasena oodatakse, et loodud lõim lõpetaks töö, kui see pole seda veel teinud. Peatükis ei vaadelda lahendusi kitsaskohale, kuna see jääb töö fookusest välja.

```c
// PARAM:  --set lib.activated[+] sv-comp
#include <pthread.h>
#include "racemacros.h"

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *t_fun(void *arg) {
    int *p = (int *) arg;
    pthread_mutex_lock(&mutex1);
    access(*p);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

int main(void) {
    pthread_t id;
    int i = 0;
    pthread_create(&id, NULL, t_fun, (void *) &i);
    pthread_mutex_lock(&mutex1);
    assert_racefree(i);
    pthread_mutex_unlock(&mutex1);
    pthread_join(id, NULL);
    return 0;
}
```

Joonis 30: Testi “goblint-regression/28-race_reach_46-escape_racefree.yml” lähtekood programeerimiskeeles C.

Testi puhul ebatõhususe põhjuse selgitamiseks katsetati erinevate käskude asendamist. Tabelis 5 on toodud välja võrdlusuuringus muutmata kujul “SV-COMP 2025” sätetega test, muutmata kujul test Goblinti vaikesätetega ning muudatused testile, mida jooksutati Goblinti vaikesätetega. Muudatustes (lisa VI) toodi sisse globaalne muutuja g, algväärtusega 0. Muudatused hõlmasid järgnevat:

1. asendati kõik `i` kasutused muutujaga g;
2. lõimes määrati `p` väärtuseks g, meetodis `main` kontrolliti g väärtust, muutujat `i` antakse lõimele;

3. lõimes määrati `p` väärtuseks `g`, meetodis `main` kontrolliti `g` väärtust, lõimelt eemaldati parameeter;

4. eemaldati kõik muteksite kasutused ning asendati meetodid, kus kasutati mutekseid (`assert_racefree(i);` ja `access(*p);`).

Tabel 5: Kitsaskoha testi muudatuse `evals` väärtusted analüüsimise järel, kus parimad tulemused on esiletõstetud rasvases kirjas.

Test	TD_{side}	TD_{weak} ahne	TD_{weak} laisk
SV-COMP 2025 sätetega	**137**	165	144
Goblinti vaikesätetega	**81**	110	90
1. Muutuja `i` asendatud `g`-ga	**47**	**47**	56
2. Lõimes määrati `p` väärtuseks `g`, lõim parameetriga	**81**	93	90
3. Lõimes määrati `p` väärtuseks `g`, lõim parameetrita	**47**	**47**	56
4. Mutekseid ei kasutata	**19**	21	**19**

Muudetud testide analüüsimisel märgati, et lõimele lokaalse muutuja `i` viida andmisel TD_{weak} variatsioonidele, sõltumata viida kasutamisest lõimes, nõuab analüüsimiseks rohkem arvutusi kui TD_{side}. Järeldati, et lõimele lokaalse muutuja viida jagamisel on Goblintis lisakontrollid, mis TD_{weak}-i $demand$i kasutamisel tingib pikemaid arvutuskäike, kui originaalses implementatsioonis.

Kokkuvõte

Bakalaureusetöö esimene eesmärk oli teostada nõrgad sõltuvused ja täiendatud laskuv lahendaja, tagades samaaegselt nende õigsus analüüsitavate programmide lahendite tagastamisel. Teine töö eesmärk oli teostada võrdlusanalüüs, mille käigus võrrelda täiendusi esialgse lahendajaga. Töö tulemusena valmis kaks variatsiooni täiendatud laskuvast lahendajast, mille mõlema korrektsust tõestati.

Võrdlusanalüüsis selgus, et saadud variatsioonidest oli laisk variatsioon keskmiselt parem kui ahne variatsioon, vähendades kitsendussüsteemide paremate poolte arvutamist testhulga peal 28.87% võrreldes ahne variatsiooni 13.70%-iga. Kui kasutati mõlema lahendaja parimat tulemust vähenes arvutuste hulk keskmisel 29.35%, sealjuures laisk variatsioon oli parem 92.43% juhtudest võrreldes ahnega.

Mõlema variatsiooni kitsaskohaks oli lõimede vahel jagatava muutuja viida kasutamise, kus mõlemas lõimes kasutati või muudeti muutujat viida kaudu muteksi lukustamise ja avamise vahel. Töös käsitleti erinevaid muudatusi testile põhjuse selgitamiseks, kuid kitsaskoha lahendamine jäi töö fookusest välja.

Viidatud kirjandus

[1] Anders Møller ja Michael I. Schwartzbach. *Static Program Analysis*. 2024.

[2] Baudouin Le Charlier ja Pascal Van Hentenryck. *A Universal Top-Down Fixpoint Algorithm*. Tehniline raport. Brown University, 1992. https://dl.acm.org/doi/book/10.5555/864683.

[3] Helmut Seidl ja Ralf Vogler. „Three improvements to the top-down solver“. Teoses: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. 2018, lk. 1–14.

[4] *Mõistesüsteeme ja nende termineid esitav ingliskeelse turva- ja privaatsusteabe portaal AKIT*. https://akit.cyber.ee.

[5] Randal E. Bryant ja David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd. Pearson, 2015. ISBN: 013409266X.

[6] Andrew S. Tanenbaum ja Herbert Bos. *Modern Operating Systems*. 5. väljaanne. Inglismaa: Pearson Education, Inc., 2023.

[7] Sara Abbaspour Asadollah *et al.* „Concurrency bugs in open source software: a case study“. *Journal of Internet Services and Applications* 8 (2017). https://jisajournal.springeropen.com/articles/10.1186/s13174-017-0055-2.

[8] Vijay D'Silva, Daniel Kroening ja Georg Weissenbacher. „A Survey of Automated Techniques for Formal Software Verification“. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), lk. 1165–1178. https://ieeexplore.ieee.org/document/4544862.

[9] Patrick Cousot ja Radhia Cousot. „Abstract interpretation: past, present and future“. *CSL-LICS '14: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* 2 (2014), lk. 1–10. https://dl.acm.org/doi/10.1145/2603088.2603165.

[10] *Tarkvara valideerimise, verifitseerimise ja sertifitseerimise ettevõte AbsInt Astrée tutvustav leht*. https://www.absint.com/astree/index.htm.

[11] Patrick Cousot ja Radhia Cousot. „Abstract Interpretation Frameworks“. *Journal of Logic and Computation* 2.4 (1992), lk. 511–547. https://doi.org/10.1093/logcom/2.4.511.

[12] Antoine Miné. *Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation*. Now Foundations ja Trends, 2017. https://doi.org/10.1561/2500000034.

[13] Helmut Seidl, Varmo Vene ja Markus Müller-Olm. „Global invariants for analysing multi-threaded applications“. *Proceedings of the Estonian Academy of Sciences Physics Mathematics* 52 (4 2003), lk. 413–436.

[14] Helmut Seidl, Reinhard Wilhelm ja Sebastian Hack. *Compiler Design: Analysis and Transformation*. Berliin, Saksamaa: Springer-Verlag Berlin Heidelberg, 2012.

[15] Timothy Gowers, June Barrow-Green ja Imre Leader. *The Princeton companion to mathematics*. Princeton University Press, 2010.

[16] Michael Schwarz *et al.* „Improving Thread-Modular Abstract Interpretation“. Teoses: *Static Analysis*. Toim. Cezara Drăgoi, Suvam Mukherjee ja Kedar Namjoshi. Cham: Springer International Publishing, 2021, lk. 359–383. ISBN: 978-3-030-88806-0.

[17] Kalmer Apinis, Helmut Seidl ja Vesal Vojdani. „Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis“. Teoses: *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, lk. 157–172. ISBN: 978-3-642-35182-2.

[18] Helmut Seidl ja Ralf Vogler. „Three improvements to the top-down solver“. *Mathematical Structures in Computer Science* 31.9 (2021), lk. 1090–1134. https://doi.org/10.1017/S0960129521000499.

[19] Beyer Dirk ja Strejček Jan. *SV-Benchmarks: Benchmark Set for Software Verification (SV-COMP 2025)*. 2025.

Lisad

I TD_{side} pseudokood OCaml stiilis

```
1  val σ       : (𝒳, 𝔻) Map.t
2  val infl    : (𝒳, 𝒳 Set.t) Map.t
3  val called  : 𝒳 Set.t
4  val stable  : 𝒳 Set.t
5  val point   : 𝒳 Set.t
6  val Set.create  : unit -> 'a Set.t
7  val Map.create  : (unit -> 'b) -> ('a, 'b) Map.t
8
9  val (!) : ('a, 'b) Map.t -> 'a -> 'b
10
11 val (:=) : ('a, 'b) Map.t * 'a -> 'b -> unit
12 val (∈)  : 'a -> 'a Set.t -> bool
13 val (+=) : 'a Set.t -> 'a -> unit
14 val (-=) : 'a Set.t -> 'a -> unit
15
16 let stable = Set.create ()
17 let called = Set.create ()
18 let point  = Set.create ()
19 let infl   = Map.create Set.create
20 let σ      = Map.create (fun () -> ⊥)
21
22 let rec destabilize x =
23     let w = !infl x in
24     infl, x := Set.create ();
25     Set.iter (fun y ->
26         stable -= y;
27         if y ∉ called then destabilize y
28     ) w
29
30 let rec eval x y =
31     if y ∈ (called ∪ leaf) then
32         point += y;
33     else
34         solve ∇ y;
35     !infl y += x;
36     !σ y
37
38 and side x d =
```

```
39      let tmp = (!σ x) ∇ d in
40      if (!σ x) ≠ tmp then (
41          σ, x := tmp;
42          stable += x
43      )
44
45  and solve p x =
46      if x ∉ stable && x ∉ called then (
47          stable += x;
48          called += x
49          let tmp = f♯_x (eval x) (side x) in
50          called -= x;
51          let tmp =
52              if x ∈ point then p (!σ x) tmp
53              else tmp
54          in
55          if x ∉ stable then solve ∇ x
56          else if !σ x = tmp then
57              if p = ∇ && x ∈ point then (
58                  stable -= x;
59                  solve Δ x
60              ) else ()
61          else (
62              σ, x := tmp;
63              destabilize x;
64              solve p x
65          )
66      )
67
68  let entry x = point += x;
69                solve ∇ x;
70                (σ, stable)
```

Joonis 31: TD_{side} OCamli-stiilis pseudokoodis [3].

II $\mathrm{TD}_{\mathrm{weak}}$ laisk variatsioon OCaml-stiilis pseudokood

```
1  val weak    : 𝒳 Set.t
2  let weak    = Set.create ()
3
4  val σ       : (𝒳, 𝔻) Map.t
5  val infl    : (𝒳, 𝒳 Set.t) Map.t
6  val called  : 𝒳 Set.t
7  val stable  : 𝒳 Set.t
8  val point   : 𝒳 Set.t
9  val Set.create  : unit -> 'a Set.t
10 val Map.create  : (unit -> 'b) -> ('a, 'b) Map.t
11
12 val (!) : ('a, 'b) Map.t -> 'a -> 'b
13
14 val (:=) : ('a, 'b) Map.t * 'a -> 'b -> unit
15 val (∈)  : 'a -> 'a Set.t -> bool
16 val (+=) : 'a Set.t -> 'a -> unit
17 val (-=) : 'a Set.t -> 'a -> unit
18
19 let stable = Set.create ()
20 let called = Set.create ()
21 let point  = Set.create ()
22 let infl   = Map.create Set.create
23 let σ      = Map.create (fun () -> ⊥)
24
25 let demand x y =
26     weak += y;
27
28
29 let rec destabilize x =
30     let w = !infl x in
31     infl, x := Set.create ();
32     Set.iter (fun y ->
33         stable -= y;
34         if y ∉ called then destabilize y
35     ) w
36
37 let rec eval x y =
38     if y ∈ (called ∪ leaf) then
39         point += y;
40     else
41         solve ∇ y;
```

```
42      !infl y += x;
43      !σ y
44
45  and side x d =
46      let tmp = (!σ x) ∇ d in
47      if (!σ x) ≠ tmp then (
48          σ, x := tmp;
49          stable += x
50      )
51
52  and solve p x =
53      if x ∉ stable && x ∉ called then (
54          stable += x;
55          called += x
56          let tmp = f♯_x (eval x) (side x) (demand x) in
57          called -= x;
58          let tmp =
59              if x ∈ point then p (!σ x) tmp
60              else tmp
61          in
62          if x ∉ stable then solve ∇ x
63          else if !σ x = tmp then
64              if p = ∇ && x ∈ point then (
65                  stable -= x;
66                  solve Δ x
67              ) else ()
68          else (
69              σ, x := tmp;
70              destabilize x;
71              solve p x
72          )
73      )
74
75  let entry x = point += x;
76                solve ∇ x;
77                Set.iter (fun w ->
78                  if w ∉ stable then entry w;
79                ) weak;
80                (σ, stable)
```

Joonis 32: $\mathrm{TD}_{\mathrm{weak}}$ laisk variatsioon OCamli-stiilis pseudokoodis.

III $\mathrm{TD}_{\mathrm{weak}}$ ahne variatsioon OCaml-stiilis pseudokood

```
1  val weak    : 𝒳 Set.t
2  let weak    = Set.create ()
3
4  val σ       : (𝒳, 𝔻) Map.t
5  val infl    : (𝒳, 𝒳 Set.t) Map.t
6  val called  : 𝒳 Set.t
7  val stable  : 𝒳 Set.t
8  val point   : 𝒳 Set.t
9  val Set.create  : unit -> 'a Set.t
10 val Map.create  : (unit -> 'b) -> ('a, 'b) Map.t
11
12 val (!) : ('a, 'b) Map.t -> 'a -> 'b
13
14 val (:=) : ('a, 'b) Map.t * 'a -> 'b -> unit
15 val (∈)  : 'a -> 'a Set.t -> bool
16 val (+=) : 'a Set.t -> 'a -> unit
17 val (-=) : 'a Set.t -> 'a -> unit
18
19 let stable = Set.create ()
20 let called = Set.create ()
21 let point  = Set.create ()
22 let infl   = Map.create Set.create
23 let σ      = Map.create (fun () -> ⊥)
24
25 let demand x y =
26     weak += y;
27     solve y ∇;
28
29 let rec destabilize x =
30     let w = !infl x in
31     infl, x := Set.create ();
32     Set.iter (fun y ->
33         stable -= y;
34         if y ∉ called then destabilize y
35     ) w
36
37 let rec eval x y =
38     if y ∈ (called ∪ leaf) then
39         point += y;
40     else
41         solve ∇ y;
```

```
42      !infl y += x;
43      !σ y
44
45  and side x d =
46      let tmp = (!σ x) ∇ d in
47      if (!σ x) ≠ tmp then (
48          σ, x := tmp;
49          stable += x
50      )
51
52  and solve p x =
53      if x ∉ stable && x ∉ called then (
54          stable += x;
55          called += x
56          let tmp = f♯_x (eval x) (side x) (demand x) in
57          called -= x;
58          let tmp =
59              if x ∈ point then p (!σ x) tmp
60              else tmp
61          in
62          if x ∉ stable then solve ∇ x
63          else if !σ x = tmp then
64              if p = ∇ && x ∈ point then (
65                  stable -= x;
66                  solve Δ x
67              ) else ()
68          else (
69              σ, x := tmp;
70              destabilize x;
71              solve p x
72          )
73      )
74
75  let entry x = point += x;
76                solve ∇ x;
77                Set.iter (fun w ->
78                  if w ∉ stable then entry w;
79                ) weak;
80                (σ, stable)
```

Joonis 33: $\mathrm{TD}_{\mathrm{weak}}$ ahne variatsioon OCamli-stiilis pseudokoodis.

IV Goblintis implementeeritud nõrgad sõltuvused

Goblintis implementeeritud nõrgad sõltuvused on kättesaadavad tõmbekutsest GitHubist leheküljel `https://github.com/goblint/analyzer/pull/1743`.

V Nõrkade sõltuvuste võrdlusuuringu tulemused

Töös teostatud võrdlusuuringu tulemused on lisatud failina ”lisa-5-uuringu-tulemused.zip”. ZIP-failis on CSV-fail "table-generator-concurrency-cmp.table.csv"ning HTML-fail "table-generator-concurrency-cmp.table.html". CSV-failis saab tulemusi vaadata tabelina ning HTML-failis graafikutena.

VI Kitsaskohana märgitud testi katsetamisega tehtud muudatused

Kitsaskohana märgitud testi “goblint-regression/28-race_reach_46-escape_racefree.yml” katsetamisel tehtud muudatused. Lisatud rida on märgitud sinise tekstiga ning algab “+”-märgiga, eemaldatud rida on märgitud punase tekstiga ning algab “-”-märgiga.

```
1  // PARAM:  --set lib.activated[+] sv-comp
2  #include <pthread.h>
3  #include "racemacros.h"
4
5  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
6  pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
7
8  +int g = 0;
9  void *t_fun(void *arg) {
10   int *p = (int *) arg;
11   pthread_mutex_lock(&mutex1);
12   access(*p);
13   pthread_mutex_unlock(&mutex1);
14   return NULL;
15 }
16
17 int main(void) {
18   pthread_t id;
19   int i = 0;
20 - pthread_create(&id, NULL, t_fun, (void *) &i);
21 + pthread_create(&id, NULL, t_fun, (void *) &g);
22   pthread_mutex_lock(&mutex1);
23 - assert_racefree(i);
24 + assert_racefree(g);
25   pthread_mutex_unlock(&mutex1);
26   pthread_join(id, NULL);
27   return 0;
28 }
```

Joonis 34: Testis “goblint-regression/28-race_reach_46-escape_racefree.yml”. Muutuja i asendati globaalse muutujaga g, algväärtusega 0;

```
1  // PARAM:  --set lib.activated[+] sv-comp
2  #include <pthread.h>
3  #include "racemacros.h"
4
5  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

+int g = 0;
void *t_fun(void *arg) {
- int *p = (int *) arg;
+ int *p = (int *) g;
  pthread_mutex_lock(&mutex1);
  access(*p);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}

int main(void) {
  pthread_t id;
  int i = 0;
  pthread_create(&id, NULL, t_fun, (void *) &i);
  pthread_mutex_lock(&mutex1);
- assert_racefree(i);
+ assert_racefree(g);
  pthread_mutex_unlock(&mutex1);
  pthread_join(id, NULL);
  return 0;
}
```

Joonis 35: Testis “goblint-regression/28-race_reach_46-escape_racefree.yml”. Lisati globaalne muutuja g, mida muteksite vahel kasutati i ja selle viida asemel.

```
// PARAM:  --set lib.activated[+] sv-comp
#include <pthread.h>
#include "racemacros.h"

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

+int g = 0;
-void *t_fun(void *arg) {
+void *t_fun() {
- int *p = (int *) arg;
+ int *p = (int *) g;
  pthread_mutex_lock(&mutex1);
  access(*p);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
```

```
18
19 int main(void) {
20   pthread_t id;
21   int i = 0;
22 - pthread_create(&id, NULL, t_fun, (void *) &i);
23 + pthread_create(&id, NULL, t_fun, NULL);
24   pthread_mutex_lock(&mutex1);
25 - assert_racefree(i);
26 + assert_racefree(g);
27   pthread_mutex_unlock(&mutex1);
28   pthread_join(id, NULL);
29   return 0;
30 }
```

Joonis 36: Testis “goblint-regression/28-race_reach_46-escape_racefree.yml”. Lisati globaalne muutuja g, mida muteksite vahel kasutati i ja selle viida asemel. Lõimelt eemaldati parameeter.

```
1  // PARAM:  --set lib.activated[+] sv-comp
2  #include <pthread.h>
3  #include "racemacros.h"
4
5  -pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
6  -pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
7
8  void *t_fun(void *arg) {
9    int *p = (int *) arg;
10 - pthread_mutex_lock(&mutex1);
11 - access(*p);
12 + (*p)++;
13 - pthread_mutex_unlock(&mutex1);
14   return NULL;
15 }
16
17 int main(void) {
18   pthread_t id;
19   int i = 0;
20   pthread_create(&id, NULL, t_fun, (void *) &i);
21 - pthread_mutex_lock(&mutex1);
22 - assert_racefree(i);
23 + i == 0;
24 - pthread_mutex_unlock(&mutex1);
25   pthread_join(id, NULL);
26   return 0;
```

```
27 }
```

Joonis 37: Test “goblint-regression/28-race_reach_46-escape_racefree.yml”, kus on eemaldatud muteksid ja nende lukustamine ja avamine, ning asendatud meetodid, kus on mutekseid kasutatud.

VII Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Heigo Tornik**,

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

   Nõrgad sõltuvused kõrvalmõjudega kitsendussüsteemides,

   mille juhendajad on Simmo Saan ja Vesal Vojdani,

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

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

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

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

Heigo Tornik
15.05.2025