TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Henrik Innos

Bitioperatsioonide analüüsi täiendamine Goblintis

Bakalaureusetöö (9 EAP)

Juhendaja:
Simmo Saan, MSc

Tartu 2025

Bitioperatsioonide analüüsi täiendamine Goblintis

Lühikokkuvõte:

Goblint on abstraktsel interpretatsioonil põhinev staatiline analüsaator, mille peamine rakendusala on mitmelõimelised C programmid. Töö eesmärk on Goblinti staatilise analüüsi täpsuse parandamine, täiendades selle bitioperatsioonide käsitlust. Töös formuleeritakse komplekt täisarvude omadusi, mis iseloomustavad nende käitumist, kui nende masinesitusele rakendada bitthaaval operatsioone, ning neid kasutatakse bitthaaval operatsioonide abstraktsioonide täpsustamiseks Goblinti täisarvudomeenides. Samuti lisati Goblintile bittväljade analüüsi funktsionaalsus. Tulemusena suudeti edukalt muuta Goblinti analüüse täpsemaks: implementatsiooni evalveeriti SV-COMP mõõtlusalustel, mille tulemusena Goblint suutis lahendada 26 uut ülesannet, ning selle korrektsus tõestati formaalselt.

Võtmesõnad: staatiline analüüs, abstraktne interpretatsioon, bitioperatsioonid, Goblint

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

Improving Bitwise Operation Analysis in Goblint

Abstract:

Goblint is a static analyzer, based on abstract interpretation, specializing in multi-threaded C programs. The goal of this thesis is to improve the precision of Goblint's static analysis by enhancing its handling of bit operations. The thesis establishes a set of integer properties that characterize their behaviour when subjected to bitwise operations on their machine representations, which were used to enhance the abstractions of logical bitwise operations in Goblint's integer domains. Additionally, support for the analysis of bit-fields was added. The precision of Goblint's analyses was successfully improved: the implementation was evaluated on SV-COMP benchmarks, resulting in Goblint being able to solve 26 additional tasks, and its correctness was formally proved.

Keywords: static analysis, abstract interpretation, bitwise operations, Goblint

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

Sisukord

1. Sissejuhatus

Tarkvara verifitseerimine on oluline, kindlustamaks, et toodetud tarkvara vastab oodatud omadustele ning ei tee vigu. Eriti oluline on see näiteks kompilaatorites, draiverites ja teistes kriitilistes süsteemides, kus on oluline korrektsus ja töökindlus. Üks viis programmide omadusi verifitseerida on staatiliselt ehk programmi käivitamata. Staatiline analüüs võimaldab näidata programmide omadusi, mis on tõestatavalt korrektsed, sest programmiolekuid vaadeldakse matemaatiliselt formuleeritud raamistikus [1]. Üks meetod korrektse staatilise analüüsi konstrueerimiseks, on abstraktne interpretatsioon, milles vaadeldakse programmi kõrgemal abstraktsiooni tasemel [2]. Seeläbi kaotatakse täpsuses, ent valides kohase abstraktsiooni, on võimalik efektiivselt konstrueerida korrektne analüüs programmi omadustele [1].

Goblint on abstraktsel interpretatsioonil põhinev staatiline analüsaator, mille peamine rakendusala on mitmelõimelised C programmid [3]. Arvuti- ja tarkvarasüsteemid muutuvad aina keerulisemaks, mistõttu on ka vaja pidevalt täpsemaid ja efektiivsemaid tarkvara verifitseerijaid. Töö eesmärk on parandada Goblinti staatilise analüüsi täpsust, täiendades selles bitioperatsioonide käsitlust. Goblintis on dokumenteeritud ebatäpsused teatavatel SV-COMP mõõtlusalustel, mis olid suuniseks tööd implementeerides.[1] Töö raames modifitseeritakse Goblinti abstraktseid täisarvudomeene, täiendades nendes loogiliste bitioperatsioonide, täpsemalt bitthaaval konjunktsiooni (AND), disjunktsiooni (OR), välistava disjunktsiooni (XOR) ja eituse (NOT) ning parempoolse loogilise bitinihke, analüüse. Samuti lisatakse Goblintile C bittväljade analüüsi funktsionaalsus. Kõik implementeeritud täiendused testitakse funktsionaalselt, käsitsi kirjutatud testide ja SV-COMP mõõtlusalustega, ning nende korrektsuses veendumiseks kasutatakse Goblinti domeeniteste ja automaatset teoreemitõestajat Z3.

Töö sisu on jaotatud kolme peatükki. Peatükis 2 kirjeldatakse asjakohaseid C keele eripärasid, mis on seotud bitioperatsioonide analüüsiga, ning formuleeritakse abstraktse interpretatsiooni teooria ja sellega seotud matemaatilised struktuurid, millele tugineb töö praktiline implementatsioon. Peatükis 3 kirjeldatakse, kuidas abstraktsel interpretatsioonil põhinev staatiline analüüs on Goblintis implementeeritud, ning näidatakse töö raames loodud praktilisi täiendusi. Peatükis 4 kirjeldatakse, kuidas verifitseeriti, et kõik lubatud funktsionaalsus on olemas ja tehtud täiendused kirjeldavad korrektset staatilist analüüsi.

[1] https://github.com/goblint/analyzer/issues/1586

2. Taust

Selles peatükis kirjeldatakse asjakohaseid C programmeerimiskeele eripärasid. Samuti antakse ülevaade staatilise programmianalüüsi teooriast ning formuleeritakse matemaatilised struktuurid, millel põhineb töö praktilise osa implementatsioon.

2.1 Bitioperatsioonid C keeles

Töös keskendutakse bittväljade ja loogiliste bititeisenduste, täpsemalt bitthaaval AND, OR, XOR, NOT operatsioonide ning parempoolse loogilise bitinihke, analüüsile. Käesolevas alapeatükis selgitatakse lahti asjakohased C keele eripärad, mis on seotud eelnimetatud kontseptsioonide analüüsiga.

2.1.1 Tähistused ja abifunktsioonid

Edaspidiseks on defineeritud järgnevad abisümbolid ja -funktsioonid. Olgu $\mathcal{T}$ kõigi C keele täisarvuliste tüüpide (ingl *integer types*) hulk

$$\mathcal{T} = \{\texttt{int}, \texttt{short}, \texttt{char}, \texttt{long}, \texttt{long long}, \texttt{signed int}, \texttt{unsigned int}, ...\}.^{2}$$

Kuna töö keskendub C programmide analüüsile, olgu käsitletavate väärtuste ruumi kirjeldamiseks defineeritud hulk $\mathbb{Z}_t$ iga tüübi $t \in \mathcal{T}$ jaoks:

$$\mathbb{Z}_t = \{x \in \mathbb{Z} \mid x \text{ kuulub tüüpi } t\}.$$

Tüüpide jaoks olgu defineeritud funktsioonid $\max, \min : \mathcal{T} \to \mathbb{Z}$:

$$\max(t) = \max(\mathbb{Z}_t)$$
$$\min(t) = \min(\mathbb{Z}_t),$$

mis tagastavad vastavalt tüübi maksimaalse ja minimaalse väärtuse. Edaspidi, kui t väärtus ei ole määratletud, on eeldatud, et t on suvaline $\mathcal{T}$ element.

Praktilise osa implementatsiooni kirjelduseks peatükis 3 on kasutusel kolm abifunktsiooni. Funktsioon *numbits* : $\mathbb{Z} \to \mathbb{N}$ tagastab täisarvu mälus esitamiseks vajaliku bittide arvu ehk argumendi x korral tagastatakse naturaalarv n nii, et $2^{n-1} \leq |x| < 2^n$ (kui $x = 0$, siis

[2]Terviklik tüüpide loetelu: https://en.cppreference.com/w/c/language/type

tagastatakse 0). Funktsiooni *numbits* abil on defineeritud abifunktsioonid min_bv, max_bv : $\mathbb{Z} \to \mathbb{Z}$ nii

$$min\_bv(n) = \begin{cases} -1 & \text{, kui } n = 0 \\ -2^{numbits(|n|-1)} & \text{, muidu} \end{cases}$$

$$max\_bv(n) = \begin{cases} 2^{numbits(n)} - 1 & \text{, kui } n > 0 \\ 2^{numbits(|n|-1)} - 1 & \text{, muidu.} \end{cases}$$

Funktsioonid *min_bv* ja *max_bv* tagastavad vastavalt minimaalse ja maksimaalse (märgiga) väärtuse, mis on võimalik saada nende arvude siseste bittide mistahes loogilistel teisendustel. Vaadeldes arvu kui fikseeritud pikkusega bitivektorit, on sellesse mahtuvad minimaalne ja maksimaalne väärtus vastavalt -2^n ja $2^n - 1$, kus n on bitivektori pikkus. Funktsiooni *numbits* argumendiks on $|n| - 1$ selle pärast, et käsitletakse täiendkoodis täisarve ning funktsioon *numbits* ei ole täiendkoodis arvude korral täiesti täpne.[3] Seda eripära seletatakse lähemalt peatükis 2.1.3. Töös analüüsitakse just loogilisi bitioperatsioone, mistõttu saab funktsioone *min_bv* ja *max_bv* kasutada läbivalt loogiliste bitioperatsioonide rakendamisel saadud tulemuste piiride määratlemiseks.

2.1.2 Bittväljad

Kernighan ja Ritchie [4] on märkinud, et C programmide korral võib mälu olla piiratud, mis juhul on vaja mitu objekti pakkida ühe masinsõna sisse. Tavaline viis sedasi pakendatud väärtustele ligi pääsemiseks on kasutada bitimaske ja loogilisi bitioperatsioone, kuid C keeles on samuti võimalik selliseid välju defineerida ja manipuleerida otseselt, määrates neile nime ja suuruse. Säärane objekt on õpikus defineeritud kui **bittväli** (ingl *bit-field*). Bittvälja saab deklareerida C keele *struct*[4] või *union*[5] objekti deklaratsiooni sees [5]. Deklaratsioon on kujul `tunnus: laius`, kus laiuse vähim väärtus on 0 ning suurim väärtus on muutuja andmetüübi laius bittides.

Bittväljad võivad olla kolme tüüpi [5]:

1. **`unsigned int`** märgita täisarvude jaoks, mille korral võimalikud väärtused jäävad lõiku $[0, 2^n - 1]$, kus n on välja laius;

[3] Funktsioon *numbits* on pärit OCamli moodulist zarith (https://ocaml.org/p/zarith/1.10/doc/Z/index.html).
[4] https://en.cppreference.com/w/c/language/struct
[5] https://en.cppreference.com/w/c/language/union

```c
struct S {
    unsigned int a : 3;
    signed int b: 2;
    int c: 3;
};
```

Joonis 1. Näide bittväljade kasutamisest koodis.

2. `signed int` märgiga täisarvude jaoks, mille korral võimalikud väärtused asuvad lõigus $[-2^{n-1}, 2^{n-1} - 1]$;

3. `int` võib sisaldada nii märgiga kui ka märgita täisarve, olenevalt implementatsioonist. Oluline on siinkohal märgata, et märksõna `int` tähendus on selles kontekstis erinev, kui C keeles mujal, kus see tähendab implitsiitselt `signed int` [5]. Seega võimalikud väärtused langevad lõiku $[0, 2^n - 1] \cup [-2^{n-1}, 2^{n-1} - 1] = [-2^{n-1}, 2^n - 1]$.

Bittväljad võivad olla ka muud tüüpi, kuid need on defineeritud implementatsiooni põhjal ning neid töös arvesse ei võeta [5]. Näiteks joonisel 1 on väljade `a`, `b` ja `c` võimalike väärtuste hulgad vastavalt $[0, 7]$, $[-1, 1]$ ja $[-4, 7]$.

2.1.3 Negatiivsete arvude esitamine

Negatiivsete arvude esitamiseks kahendkoodis on kolm peamist viisi: märgiga otsekood (ingl *sign-and-magnitude*), pöördkood (ingl *one's complement*) ja täiendkood (ingl *two's complement*). C keeles pole fikseeritud konkreetset esitusviisi ning see oleneb kompilaatori implementatsioonist. Siiski, ülekaalukalt kõige levinuim nendest on täiendkood ning alates C23 standardist [5] on sellest saanud ametlik standard märgiga täisarvude esitamiseks. Selles töös on samuti edaspidi eeldatud, et kõik negatiivsed arvud on täiendkoodis.

Kahendarvu vastandarvu leidmiseks täiendkoodis tuleb pöörata kõik arvu bitid ning tulemusele liita 1, arvestamata ületäitumisega. Tabelis 1 on näited täiendkoodis esituste kohta. Tegu on märgiga täisarvudega, seega kõrgeim bitt (ingl *most significant bit*) on märgibitt, mille puhul 0 tähendab positiivset ning 1 negatiivset arvu. Siinkohal saab märgata, et täiendkoodis n-kohalise kahendarvuga maksimaalne esindatav väärtus on $2^{n-1} - 1$ ning minimaalne -2^{n-1}. Seega negatiivseid arve mahub n biti sisse ühe võrra rohkem kui positiivseid.

Alapeatükis 2.1.1 kirjeldatud funktsioon *numbits* on negatiivse argumendi korral defineeritud läbi tema vastandarvu: $\mathit{numbits}(-i) = \mathit{numbits}(i)$, kuid see pole täiendkoodis arvude korral täiesti

Tabel 1. Näited 8-bitistest arvudest täiendkoodis.

Kümnendarv	Kahendarv	Vastandarv täiendkoodis
0	00000000	00000000
1	00000001	11111111
8	00001000	11111000
127	01111111	10000001
-128	10000000	10000000

täpne. Funktsiooni definitsioon põhineb matemaatilistel täisarvudel ning seeläbi ei arvestata täiendkoodis masinesituste eripäradega, nagu märgibiti olemasolu. Näiteks $numbits(-128) = 8$, kuid tegelikult mahub -128 ära sama arvu bittide sisse, kui 127 ning $numbits(127) = 7$ (vt tabel 1). Funktsioonid *min_bv* ja *max_bv* eeldavad samuti, et argument on täiendkoodis ning arvestamaks selle erisusega on nendes funktsiooni *numbits* kasutusel antud argumendiks $|n| - 1$. Näited funktsioonide käitumisest eri argumentidel on tabelis 2.

Tabel 2. Funktsioonide *numbits*, *min_bv* ja *max_bv* väärtused argumentidel.

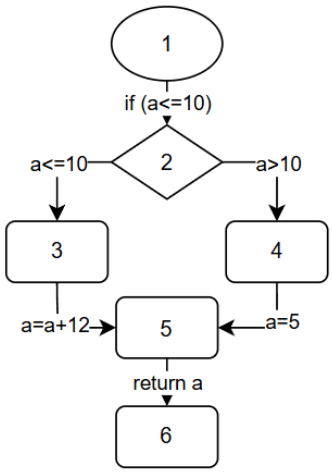
Argument	numbits	min_bv	max_bv
0	0	-1	1
1	1	-1	1
-1	1	-1	1
128	8	-128	255
-128	8	-128	127

2.2 Staatiline analüüs

Miné [1] on kirjeldanud staatilisi analüsaatoreid kui programme, mis võtavad sisendiks teise programmi ning tagastavad informatsiooni selle käitumise kohta. Staatilise analüüsi eripära on see, et staatilised analüsaatorid ei käivita uuritavat programmi, vaid analüüsivad selle lähtekoodi. Kuna kõigi võimalike programmi täitmiste läbi kontrollimine ei ole realistlik, vaadeldakse staatilises analüüsis programmi mingil kõrgemal abstraktsiooni tasemel. Autor toob välja, et üks meetod korrektse staatilise analüüsi defineerimiseks on **abstraktne interpretatsioon**, sest selle

```c
int main() {
    unsigned int a;
    if (a <= 10) {
        a = a + 12;
    } else {
        a = 5;
    }
    return a;
}
```

(a) C keeles.



(b) Vastav juhtvoograaf.

Joonis 2. Näidisprogramm.

abil saab matemaatiliselt väljendada seost üldistatud analüüsi ja esialgse programmi semantika vahel.

2.2.1 Abstraktne interpretatsioon

Abstraktse interpretatsiooni olemust on selgitanud Cousot [2]. Programmid kirjeldavad arvutusi mingis konkreetsete väärtuste ruumis. Abstraktne interpretatsioon kirjeldab arvutusi abstraktsete väärtuste ruumis nii, et abstraktsete arvutuste tulemuste põhjal saab teha järeldusi nendele vastavate konkreetsete väärtuste kohta.

Üks selline abstraktsioon on näiteks täisarvude abstraheerimine intervallideks, mida on kirjeldanud järgnevalt Miné [1]. Intervallanalüüsis programmi käsud abstraheeritakse intervallide aritmeetilisteks operatsioonideks. Näiteks muutujale väärtuse liitmisel liidetakse see tema alumisele ja ülemisele tõkkele. Võrratust sisaldav tingimuslause kitsendab intervalli tõkkeid, juhul kui võrreldav arv ei asu intervallist väljaspool.

Ideed illustreerib programm joonisel 2a ja sellele vastav juhtvoograaf joonisel 2b. Tegu on lihtsa programmiga, mille olekuid iseloomustab muutuja `a` võimalike väärtuste hulk vastavates olekutes. Programmi staatiliseks analüüsiks saab muutuja võimalike väärtuste hulga abstraheerida intervalliks. Programmi alguses on algväärtustamata muutuja `a` abstraktne väärtus intervall $[0, 2^{32} - 1]$. Tingimus `a <= 10` kitsendab intervalli nii, et tingimuslause *then* harus $a \in [0, 10]$ ja *else* harus $a \in [11, 2^{32} - 1]$. Arvutuste puhul kasutatakse intervallide aritmeetikat. Parema haru läbimise järel on tulemuseks ühe väärtusega intervall $[5, 5]$ ning vasaku haru tulemuseks $[12, 22]$. Tingimuslause järel ühendatakse harud minimaalseks intervalliks, mis sisaldab mõlema

```
1  int main() {
2      unsigned int a;
3      /* a ∈ {0, 1, 2, ..., 2^32 − 1}*/
4      if (a <= 10) {
5          /* a ∈ {0, 1, 2, ..., 10} */
6          a = a + 12;
7          /* a ∈ {12, 13, 14, ..., 22} */
8      } else {
9          /* a ∈ {11, 12, 13, ..., 2^32 − 1}*/
10         a = 5;
11         /* a ∈ {5} */
12     }
13     /* a ∈ {5, 6, 7, ..., 22} */
14     return a;
15 }
```

Joonis 3. Näidisprogramm täiendatud muutuja võimalike väärtustega

haru tulemusi ehk intervalliks $[5, 22]$. Analüüsi tulemus on seega, et programmi täitmise lõpus vastab muutujale `a` intervall $[5, 22]$. Sellest saab järeldada, et programmi täitmise lõpuks kuulub `a` väärtus hulka $\{x \in \mathbb{Z} \mid 5 \leq x \leq 22\}$. Analüüsi illustreerib joonis 3, kus on kujutatud võimalike muutuja väärtuste hulgad igas programmi olekus ehk iga käsu täitmise järel.

Abstraktsel interpretatsioonil on kaks olulist omadust, mida on kirjeldanud Miné [1]:

1. Abstraktne interpretatsioon on korrektne (ingl *sound*). Korrektsus programmianalüüsi kontekstis tähendab, et analüüsi tulemusel saadud omadused kehtivad ka programmi töötamise käigus. Abstraktsioon on üldistus programmi olekutest, seega kõik võimalikud programmi olekud kuuluvad selle hulka. Sellest järelduvalt, kui mingi omadus kehtib abstraktsete olekute hulgal, kehtib see ka kõigis konkreetsetes programmi olekutes. Seega kui abstraktse analüüsi käigus ei teki viga, siis ka programmis ei saa viga tekkida [6].

2. Abstraktne interpretatsioon on ebatäpne. Abstraktne interpretatsioon võib tagastada valepositiivseid tulemusi, ehk abstraktses olekus leitakse viga, olgugi et programmis seda päriselt ei teki. Sellises olukorras programm on korrektne, kuid abstraktsioon on liiga üldine, et seda tõestada. Siis on omaduse tõestamiseks vaja leida täpsem abstraktsioon.

2.2.2 Osalised järjestused

Edaspidi formuleeritakse matemaatiliselt eelmises alapeatükis ebaformaalselt kirjeldatud kontseptsioon. Selles ning järgnevates alapeatükkides välja toodud definitsioonid on pärit Miné õppematerjalist [1] ning sellest on võetud ka eeskuju definitsioonidest lähtuvate analüüside koostamisel.

Osaline järjestus $\sqsubseteq$ hulgal X on seos $\sqsubseteq \in X \times X$, mis on

1. refleksiivne: $\forall x \in X : x \sqsubseteq x$
2. antisümmeetriline: $\forall x, y \in X : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
3. transitiivne: $\forall x, y, z \in X : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.

Hulka X koos sellel defineeritud osalise järjestusega $\sqsubseteq$ nimetatakse **osaliselt järjestatud hulgaks** ning tähistatakse $(X, \sqsubseteq)$.

Minimaalne struktuur, mis on vajalik konkreetsete ja abstraktsete maailmate vahelise seose kirjeldamiseks on konkreetsete väärtuste hulk $(\mathcal{D}, \sqsubseteq)$, abstraktsete väärtuste hulk $(\mathcal{D}^\#, \sqsubseteq^\#)$ ja konkretiseerimisfunktsioon (ingl *concretization function*) γ. **Konkretiseerimisfunktsioon** $\gamma : (\mathcal{D}, \sqsubseteq) \to (\mathcal{D}^\#, \sqsubseteq^\#)$ on monotoonne[6] funktsioon, mis kujutab iga abstraktse elemendi konkreetseks elemendiks. Nii siin kui ka edaspidi kasutatakse ülaindeksit $^\#$, et tähistada hulga, väärtuse, funktsiooni vms abstraktsiooni.

Joonisel 2a kujutatud programmi korral iseloomustavad programmi olekuid muutuja `a` võimalikud väärtused, seega $(\mathcal{D}, \sqsubseteq) = (\mathcal{P}(\mathbb{Z}_t), \subseteq)$, kus $\mathcal{P}(\mathbb{Z}_t)$ on $\mathbb{Z}_t$ kõigi alamhulkade hulk, ning $(\mathcal{D}^\#, \sqsubseteq^\#) = (\{[a, b] \mid a \in \mathbb{Z}_t, b \in \mathbb{Z}_t, a \leq b\}, \subseteq)$. Näites on t väärtus `unsigned int`, kuid analüüs kehtiks ka suvalise tüübi korral. Selleks, et hulgas $\mathcal{D}^\#$ leiduks vastav element hulga $\mathcal{D}$ elemendile $\emptyset$, tuleb abstraktsete väärtuste hulka lisada element $\bot$. Seejärel saab defineerida abstraktsed väärtused

$$(\mathcal{D}^\#, \sqsubseteq^\#) = (\{[a, b] \mid a \in \mathbb{Z}_t, b \in \mathbb{Z}_t, a \leq b\} \cup \{\bot\}, \subseteq)$$

[6] *Monotoonsus* tähendab siin kontekstis, et $a_1 \sqsubseteq^\# a_2 \Rightarrow \gamma(a_1) \sqsubseteq \gamma(a_2)$.

ning konkretiseerimisfunktsiooni

$$\gamma([a,b]) = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$$
$$\gamma(\bot) = \emptyset.$$

Oluline on märgata, definitsioonis $\mathcal{D}^\#$ elemendid on täisarvude paarid, mis tähistavad intervalli ülemist ja alumist tõket, mitte intervallid (ehk täisarvude hulgad) ise. Säärane esitus on arvutisüsteemides märkimisväärselt efektiivsem. Konkretiseerimisfunktsioon kujutab täisarvude paari vastavaks täisarvude hulgaks.

Abstraktsiooni korrektsus on defineeritud järgmiselt. $c^\# \in \mathcal{D}^\#$ on $c \in \mathcal{D}$ korrektne abstraktsioon parajasti siis, kui $c \leq \gamma(c^\#)$. Joonise 2a näidisprogrammi korral on tõepoolest analüüs korrektne, sest väärtuste hulga ja intervallide vaheline seos on identsusseos.

2.2.3 Võred

Olgu a ja b osaliselt järjestatud hulga X elemendid. Elementide a ja b **ülatõkkeks** nimetatakse igat elementi $c \in X$, mille korral $a \sqsubseteq c$ ja $b \sqsubseteq c$. Analoogiliselt, elementide a ja b **alatõkkeks** nimetatakse iga elementi $c \in X$, mille puhul $c \sqsubseteq a$ ja $c \sqsubseteq b$. Elementi kutsutakse **vähimaks ülatõkkeks** (ingl *least upper bound* või *join*) ja tähistatakse $a \sqcup b$, kui see on vähim element, mis on suurem elementidest a ja b. Elementi kutsutakse **suurimaks alatõkkeks** (ingl *greatest lower bound* või *meet*) ja tähistatakse $a \sqcap b$, kui see on suurim element, mis on väiksem elementidest a ja b.

Osaliselt järjestatud hulgad on minimaalseim struktuur, mis on vajalik abstraktsioonide kirjeldamiseks, kuid mitmete järjestatud hulkade struktuur on sellest rikkalikum ning seda saab programmianalüüsis ära kasutada. Säärased struktuurid on näiteks võred.

Võre on osaliselt järjestatud hulk, millel igal kahel elemendil leidub vähim ülatõke ja suurim alatõke. **Tõkestatud võre** on võre, milles leiduvad vähim ja suurim element. Neid tähistatakse vastavalt $\bot$ (ingl *bot*) ja $\top$ (ingl *top*).

Eelmises peatükis formuleeritud intervallide analüüsis saab märgata, et seose $\subseteq$ järgi suurim element ehk $\top$ on $[\min(t), \max(t)]$. Tulemuseks on tõkestatud võre

$$(\{[a,b] \mid a \in \mathbb{Z}_t, b \in \mathbb{Z}_t, a \leq b\} \cup \{\bot\}, \subseteq, \bot, [\min(t), \max(t)])\,.$$

Suurima alatõkke ja vähima ülatõkke leidmine on võre elementidel defineeritud järgnevalt:

$$[x_1, x_2] \sqcap [y_1, y_2] = \begin{cases} [\max(x_1, y_1), \min(x_2, y_2)] & \text{, kui } \max(x_1, y_1) \leq \min(x_2, y_2) \\ \bot & \text{, muidu} \end{cases} \quad (1)$$

$$[x_1, x_2] \sqcup [y_1, y_2] = [\min(x_1, y_1), \max(x_2, y_2)]. \quad (2)$$

Kui kumbki argumentides on $\bot$, siis kehtib järgmine:

$$x \sqcup \bot = \bot \sqcup x = x$$

$$x \sqcap \bot = \bot \sqcap x = \bot.$$

2.2.4 Abstraktne domeen

Selleks, et viia läbi staatilist analüüsi abstraheerimise abil, on vaja samuti kõigile konkreetsetele operatsioonidele määrata vastav abstraktne operatsioon. Olgu $\mathcal{D}$ konkreetsete väärtuste hulk ja $\mathcal{D}^\#$ selle abstraktsioon. Olgu $\oplus : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ suvaline binaarne operatsioon ja $\oplus^\# : \mathcal{D}^\# \times \mathcal{D}^\# \to \mathcal{D}^\#$ selle abstraktsioon. $\oplus^\#$ on **korrektne** siis, kui

$$\forall X^\#, Y^\# \in \mathcal{D}^\# : \gamma(X^\#) \ \oplus \ \gamma(Y^\#) \sqsubseteq \gamma(X^\# \oplus^\# Y^\#). \quad (3)$$

See tähendab, et kõik väärtused, mis tekivad operatsiooni konkreetsel sooritamisel, saavad kaetud operatsiooni abstraktsel sooritamisel.

Abstraktne domeen koosneb järgnevatest elementidest:

- abstraktne väärtuste hulk $\mathcal{D}^\#$
- osaline järjestus $\sqsubseteq^\#$ hulgal $\mathcal{D}^\#$
- konkretiseerimisfunktsioon $\gamma : \mathcal{D}^\# \to \mathcal{D}$
- Vähim element $\bot^\# \in \mathcal{D}^\#$ ja suurim element $\top^\# \in \mathcal{D}^\#$
- korrektsed abstraktsioonid binaarsetele operaatoritele.[7]

Loetletud omadustega abstraktne domeen on minimaalne matemaatiline struktuur, mis on vajalik, et üles ehitada staatiline analüüs, mis on konstruktsioonilt korrektne.

[7]Definitsiooni on lihtsustatud, tuues välja ainult elemendid, mis on asjakohased selles töös.

3. Goblinti analüüsid

Goblint[8] on C programmide staatiline analüsaator, mis põhineb abstraktsel interpretatsioonil [3]. Loogiliste bitioperatsioonide abstraktsioonide implementatsioon Goblinti täisarvudomeenides oli eelnevalt pigem triviaalne. Töö eesmärgiks on Goblinti analüüsi täiendamine täpsemate bitioperatsioonide abstraktsioonidega. Operatsioonid, mille abstraktsioone täpsustati, olid bitthaaval AND (&), OR (|), XOR (^) ja NOT (~) ning parempoolne bitinihe ($\gg$). Töö raames täiendati nelja Goblinti abstraktset domeeni: intervallide domeen, intervallihulkade domeen, DefExc domeen ja Enums domeen. Muudatuste jaoks tehti tõmbetaotlus Goblinti repositooriumi (vt Lisa III).

3.1 Intervallide domeen

Goblinti intervallide domeeni võrestruktuur ning konkretiseerimine on defineeritud analoogiliselt sellele, nagu seda on tehtud peatükkides 2.2.2 ja 2.2.3. Edaspidi kutsutakse intervalli $[x_1, x_2]$ **negatiivseks**, kui $x_2 < 0$ ehk $\forall x \in [x_1, x_2] : x < 0$, ja **mittenegatiivseks**, kui $x_1 \geq 0$ ehk $\forall x \in [x_1, x_2] : x \geq 0$.

Bitthaaval AND defineeriti neljal võimalikul juhul:

- kui mõlemad intervallid on mittenegatiivsed (k.a `unsigned` tüüpi väärtused), siis

$$[x_1, x_2] \mathbin{\&^\#} [y_1, y_2] = [0, \min(x_2, y_2)] \tag{4}$$

- kui mõlemad intervallid on negatiivsed, siis

$$[x_1, x_2] \mathbin{\&^\#} [y_1, y_2] = [\mathit{min\_bv}(\min(x_1, y_1)), 0]$$

- kui $x_1 \geq 0$ ja $y_1 < 0$ (analoogiline argumentide vahetamisel), siis

$$[x_1, x_2] \mathbin{\&^\#} [y_1, y_2] = \left[0, x_2\right] \tag{5}$$

- muidu,

$$[x_1, x_2] \mathbin{\&^\#} [y_1, y_2] = [\mathit{min\_bv}(\min(x_1, y_1)), \max(x_2, y_2)]. \tag{6}$$

[8] https://goblint.in.tum.de/

Kõige üldisem viis määratleda tulemuse piire on kasutades funktsiooni *min_bv*. Mida suurem on arvu absoluutväärtus, seda rohkem on vaja tema esitamiseks bitte, seega negatiivsed tulemused on piiritletud intervallide vähima elemendi bitiesituse pikkusega. Positiivsetele väärtustele leiti täpsemad tõkked.

Idee valemi (4) taga on järgmine. Kuna intervallides võivad peaaegu alati leiduda arvud kujul $x = 010101\ldots$ ja $y = 001010\ldots$, mille puhul bitthaaval konjunktsioon on võrdne nulliga, siis intervalli alumine raja on 0. Olgu $x \in [x_1, x_2]$ ja $y \in [y_1, y_2]$ suvalised n-kohalised kahendarvud $x = a_n a_{n-1} \ldots a_1 a_0$, $y = b_n b_{n-1} \ldots b_1 b_0$. Kahendarvu kümnendsüsteemi teisendamise valemi järgi

$$\begin{aligned} x &= a_n \cdot 2^n + \ldots a_1 \cdot 2^1 + a_0 \cdot 2^0 && \leq x_2 \\ y &= b_n \cdot 2^n + \ldots b_1 \cdot 2^1 + b_0 \cdot 2^0 && \leq y_2. \end{aligned}$$

Olgu $z = a \,\&\, b = c_n \ldots c_1 c_0$. Bitthaaval konjunktsiooni definitsioonist tuleneb, et $c = a \,\&\, b = 1 \Rightarrow a = 1 \wedge b = 1$ ning

$$\forall i \in \{0, 1, \ldots, n\} : \quad \begin{cases} c_i \cdot 2^i \leq a_i \cdot 2^i \\ c_i \cdot 2^i \leq b_i \cdot 2^i. \end{cases}$$

Seega

$$\begin{aligned} z = c_n \cdot 2^n + \ldots c_1 \cdot 2^1 + c_0 \cdot 2^0 &\leq x_2 \\ z &\leq y_2 \\ z &\leq \min(x_2, y_2). \end{aligned}$$

Valemi (5) saab tuletada sarnaselt. Kui $y \in [y_1, y_2]$ on mittenegatiivne, siis eelneva tõestuse põhjal $x \in [x_1, x_2]$ korral $x \,\&\, y \leq x_2$. Kui y on negatiivne, saab teha konjunktsiooni ainult kõrgeimate bittide peal. Tulemuseks on y', mis on positiivne arv, mille kõrgeimast bitist madalamal on bitid identsed y omadega. Eelnevast tuletusest: $x \,\&\, y' \leq x_2$. Kuna y ja y' erinevad vaid kõrgeima biti poolest ning x kõrgeim bitt on 0, siis $x \,\&\, y = x \,\&\, y' \leq x_2$. Analoogiliselt on näidatav ka valemi (6) ülemise raja leidmine.

Bitthaaval OR defineeriti samuti neljal võimalikul juhul:

- kui mõlemad intervallid on mittenegatiivsed (k.a `unsigned` tüüpi väärtused), siis

$$[x_1, x_2] \mid^{\#} [y_1, y_2] = [\max(x_1, y_1), \mathit{max\_bv}(\max(x_2, y_2))] \tag{7}$$

- kui mõlemad intervallid on negatiivsed, siis

$$[x_1, x_2] \mid^{\#} [y_1, y_2] = [\max(x_1, y_1), 0] \tag{8}$$

- kui $x_2 < 0$ ja $y_2 \geq 0$ (analoogiline argumentide vahetamisel), siis

$$[x_1, x_2] \mid^{\#} [y_1, y_2] = [x_1, 0] \tag{9}$$

- muidu,

$$[x_1, x_2] \mid^{\#} [y_1, y_2] = [\min(x_1, y_1), \mathit{max\_bv}(\max(x_2, y_2))]. \tag{10}$$

Üldiseim viis ülemise raja leidmiseks on kasutada funktsiooni *max_bv*, sest bitthaaval disjunktsiooni tulemus ei saa võtta mälus rohkem ruumi kui intervallide maksimaalne element. Alumiste rajade puhul sai leida täpsemad tõkked.

Idee valemi (7) alumise raja leidmise taga on järgmine. Olgu $x \in [x_1, x_2]$ ja $y \in [y_1, y_2]$ suvalised n-kohalised kahendarvud $x = a_n \dots a_1 a_0$ ja $y = b_n \dots b_1 b_0$. Kahendarvu kümnendsüsteemi teisendamise valemi järgi

$$\begin{aligned} x &= a_n \cdot 2^n + \dots a_1 \cdot 2^1 + a_0 \cdot 2^0 && \geq x_1 \\ y &= b_n \cdot 2^n + \dots b_1 \cdot 2^1 + b_0 \cdot 2^0 && \geq y_1. \end{aligned}$$

Olgu $z = a \mid b = c_n \dots c_1 c_0$. Bitthaaval disjunktsiooni definitsioonist tuleneb, et $c = a \mid b = 0 \Rightarrow a = 0 \wedge b = 0$ ning

$$\forall i \in \{0, 1, \dots, n\} : \quad \begin{cases} c_i \cdot 2^i \geq a_i \cdot 2^i \\ c_i \cdot 2^i \geq b_i \cdot 2^i. \end{cases}$$

Seega

$$\begin{aligned} z = c_n \cdot 2^n + \dots c_1 \cdot 2^1 + c_0 \cdot 2^0 &\geq x_2 \\ z &\geq y_2 \\ z &\geq \max(x_2, y_2). \end{aligned}$$

Valemis (8) mõlema argumendi märgibitt on 1, seega tulemus on kindlasti negatiivne ja ülemiseks rajaks saab määrata 0. Alumise tõkke leidmine käib analoogiliselt eelnevaga, kasutades täiendkoodis negatiivse väärtuse arvutamiseks valemit $x = -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$. Valemi (9) tõesust saab näidata järgnevalt. Kui $y \in [y_1, y_2]$ on negatiivne, siis on eelnevalt näidatud, et suvalise x korral $x \mid y \geq x_1$. Kui y on mittenegatiivne, saab teha disjunktsiooni ainult kõrgeimatel bittidel, tulemuseks on y', mis on negatiivne arv. Eelnevast tuletusest: $x \mid y' \geq x$. Kuna y ja y' erinevad ainult kõrgeima biti poolest ning x kõrgeim bitt on 1, siis $x \mid y = x \mid y' \geq x_1$. Kasutades eelnevaid tulemusi, saab näidata ka valemi (10) tuletust.

Bitthaaval XOR defineriti järgnevalt:

- kui mõlemad intervallid on mittenegatiivsed, siis

$$[x_1, x_2] \,\hat{\wedge}^{\#}\, [y_1, y_2] = [0, \mathit{max\_bv}(\max(x_2, y_2))]$$

- kui mõlemad intervallid on negatiivsed, siis

$$[x_1, x_2] \,\hat{\wedge}^{\#}\, [y_1, y_2] = [0, \mathit{max\_bv}(\min(x_1, y_1))]$$

- kui üks intervall on negatiivne ja teine mittenegatiivne, siis

$$[x_1, x_2] \,\hat{\wedge}^{\#}\, [y_1, y_2] = [l, 0]$$

- muidu,

$$[x_1, x_2] \,\hat{\wedge}^{\#}\, [y_1, y_2] = [l, u]\,, \tag{11}$$

kus

$$\begin{aligned} l &= \min\big(\{\mathit{min\_bv}(i) \mid i \in \{x_1, y_1\}\} \cup \{-\mathit{max\_bv}(i) - 1 \mid i \in \{x_2, y_2\}\}\big) \\ u &= \max\big(\mathit{max\_bv}(i) \mid i \in \{x_1, x_2, y_1, y_2\}\big). \end{aligned}$$

Siin kasutatakse samuti väärtuste piiride leidmiseks argumentide suurust mälus, kuid vaja on teatavaid lisatingimusi ja -üldistusi, sest XOR-i käitumist on keerulisem kirjeldada. Raja l leidmisel kasutatakse positiivsetel intervallide rajadel funktsiooni *max_bv*, sest $-\mathit{max\_bv}(i) - 1$ on täpselt ühe kahe astme võrra väiksem kui $\mathit{min\_bv}(i)$. See on vajalik, sest XOR töötab teatavate negatiivse ja positiivse argumendi paaride korral omapäraselt. Näiteks rakendades ka ülemistele rajadele funktsiooni *min_bv* oleks

$$[-8, 3] \,\hat{\wedge}^{\#}\, [-3, 8] = [-8, 15],$$

kuid $-8 \,\hat{\wedge}\, 8 = -16$ (vt joonis 4) ning seega abstraktsioon oleks ebakorrektne. Kirjeldatud valemiga saadakse korrektne tulemus $[-16, 15]$.

Bitthaaval NOT defineeriti järgnevalt:

$$\sim^{\#}[x_1, x_2] = [\sim x_2, \sim x_1].$$

Täiendkoodis täisarvude puhul on NOT operatsioon defineeritud kui $\sim x = -x - 1$. Rakendades NOT-i igale intervalli elemendile on saadud minimaalne väärtus $\sim x_2$ ja maksimaalne $\sim x_1$.

$$\begin{array}{rl} 11111000 & = -8 \\ \hat{}\,00001000 & = 8 \\ \hline 11110000 & = -16 \end{array}$$

Joonis 4. Bitthaaval XOR erijuhul.

Parempoolne bitinihe, eeldusel, et tegu on märgita täisarvudega või mõlemad intervallid on mittenegatiivsed, defineeriti valemiga

$$[x_1, x_2] \gg^{\#} [y_1, y_2] = \left[0, \frac{x_2}{2^{y_1}}\right].$$

C keeles on bittide paremale nihutamise definitsioon $x \gg y = \frac{x}{2^y}$, eeldusel, et x ja y on kas märgita või mittenegatiivsed täisarvud [5]. Intervallide korral on tulemuse ülemise raja määramiseks vaja leida vastavalt kummastki intervallist väärtused x ja y, nii, et $\frac{x}{2^y}$ oleks maksimaalne. Need väärtused on vastavalt x_2 ja y_1.

3.2 Intervallihulkade domeen

Intervallihulkade domeeni ülesehitus on sarnane intervallide domeenile. Konkreetsete väärtuste hulgaks on siinkohal aga $\mathcal{D} = \mathcal{P}(\mathcal{P}(\mathbb{Z}))$. Abstraktsete väärtuste hulgaks on vastavalt $\mathcal{D}^{\#} = \mathcal{P}(\{[a, b] \mid a \in \mathbb{Z}_t, b \in \mathbb{Z}_t, a \leq b\})$. Lisatingimusteks on, et intervallid ei tohi kattuda ning ei tohi asetseda vahetult kõrvuti. Domeeni vähim element on $\emptyset$ ja suurim element $\{[\min(t), \max(t)]\}$.

Operatsioonid on defineeritud analoogiliselt intervallide domeeniga, tehted tehakse läbi igal argumentide otsekorrutise elemendil ning tulemused ühendatakse. Kõik operatsioonid, mis defineeriti intervallide domeenis, implementeeriti ka intervallihulkade omas.

3.3 DefExc domeen

DefExc domeen ehk määratud-välistatud domeen (ingl *definite-excluded domain*) esitab väärtusi kas konstandina või hulgana välistatud väärtustest. Domeeni elemendid saavad olla kahte tüüpi:

1. konstant $c \in \mathbb{Z}_t$
2. hulk välistatud väärtusi $[\![r_1, r_2]\!] \setminus \mathcal{E}$, kus $\mathcal{E} \subseteq \mathbb{Z}_t$ ja

$$[\![r_1, r_2]\!] := \begin{cases} [0, 2^{r_2} - 1] & \text{, kui } r_1 = 0, \\ [-2^{|r_1|}, 2^{r_2} - 1] & \text{, muidu.} \end{cases} \tag{12}$$

```
1  int main {
2      unsigned int a;
3      int b;
4
5      if (b == 1 || b == 2) {
6          b = 0;
7      }
8
9      return 0;
10 }
```

Joonis 5. Näidisprogramm illustreerimaks DefExc domeeni.

Välistatud element koosneb kahest osast: **välistushulk** (ingl *exclusion set*) $\mathcal{E}$ ehk hulk väärtusi, mis muutujal ei saa olla, ning **bitiintervall** $[\![r_1, r_2]\!]$, mis kirjeldab bittide vahemikku, millesse väärtus peab mahtuma. Positiivne raja näitab maksimaalset bittide arvu, millesse peavad mahtuma võimalikud positiivsed väärtused ning negatiivne raja analoogiliselt piirab negatiivseid väärtusi. Negatiivse raja puhul on bittide mahtuvus raja absoluutväärtus. Kuna $[\![r_1, r_2]\!]$ tähistab intervalli, siis $\sqcap$ ja $\sqcup$ töötavad sellel samamoodi kui tavalistel intervallidel. Bitiintervallidel on Goblinti implementatsioonis lisakitsendus: $\forall [\![r_1, r_2]\!] : r_1 \leq 0 \ \& \ r_2 > 0$ ehk intervall peab sisaldama nulli.[9]

Näiteks saab kasutada DefExc domeeni joonisel 5 kujutatud programmi analüüsimiseks. Programmi lõpuks on muutujate `a` ja `b` abstraktsioonide väärtused vastavalt $[\![0, 32]\!] \setminus \emptyset$ ja $[\![-31, 31]\!] \setminus \{1, 2\}$.

Konstantidel on operatsioonid defineeritud otseselt ja täpselt, rakendades nendele vastavat tehet. Täiendusi tehti tehete abstraktsioonidel, kus vähemalt üks argument on välistatud element. Binaarsete bititeisenduste korral on keeruline teha järeldusi tulemuse välistushulga $\mathcal{E}$ kohta, sest tegu pole injektiivsete funktsioonidega. Välistushulka kuulumine tähendab, et $e \in \mathcal{E} \Leftrightarrow e \notin \gamma(\mathcal{E})$[10], kus $\gamma(\mathcal{E}) = \mathbb{Z}_t \setminus \mathcal{E}$. Suvaliste $e_1 \in \mathcal{E}_1, e_2 \in \mathcal{E}_2$ ja $\oplus \in \{\&, |, \hat{}\}$ korral ei saa järeldada,

[9] https://github.com/goblint/analyzer/pull/1726/commits/c5c752eec48582532e13d87ae5735de364071bf9

[10] Konstantide konkretiseerimine on siin otsene ehk γ on identsusfunktsioon, seega võib γ ja ülaindeksid $^{\#}$ lihtsuse mõttes konstantide puhul ära jätta.

et $e_1 \oplus e_2 \in \mathcal{E}_1 \oplus^{\#} \mathcal{E}_2$. Tehte $\oplus$ mitteinjektiivsuse tõttu võivad leiduda $c_1 \notin \mathcal{E}_1$ ja $c_2 \notin \mathcal{E}_2$ nii, et $c_1 \oplus c_2 = e_1 \oplus e_2$. Sellisel juhul $c_1 \in \gamma(\mathcal{E}_1)$ ja $c_2 \in \gamma(\mathcal{E}_2)$ ning $c_1 \oplus c_2 \in \gamma(\mathcal{E}_1) \oplus \gamma(\mathcal{E}_2)$[11]. Kuna $c_1 \oplus c_2 = e_1 \oplus e_2$, siis eeldades, et $e_1 \oplus e_2 \in \mathcal{E}_1 \oplus^{\#} \mathcal{E}_2$, järeldub $c_1 \oplus c_2 \in \mathcal{E}_1 \oplus^{\#} \mathcal{E}_2 \Rightarrow c_1 \oplus c_2 \notin \gamma(\mathcal{E}_1 \oplus^{\#} \mathcal{E}_2)$. Tekkis vastuolu $\oplus^{\#}$ korrektsusega valemi (3) järgi. Eelnevaga arvestava korrektse välistushulga leidmiseks ei leidu efektiivset viisi, seega binaarsete operatsioonide korral kujutatakse välistushulgad triviaalseks tühjaks hulgaks.

Bitthaaval AND välistatud ja määratud elemendi vahel defineeriti järgnevalt:

$$(\llbracket r_1, r_2 \rrbracket \setminus \mathcal{E}) \mathbin{\&^{\#}} c = \begin{cases} \llbracket 0, \min(r_2, \mathit{numbits}(c)) \rrbracket \setminus \emptyset & \text{, kui } c \geq 0 \text{ ja } r_1 = 0 \\ \llbracket 0, \mathit{numbits}(c) \rrbracket \setminus \emptyset & \text{, kui } c \geq 0 \\ \llbracket 0, r_2 \rrbracket \setminus \emptyset & \text{, kui } r_1 = 0 \\ \llbracket -b, b \rrbracket \setminus \emptyset & \text{, muidu} \end{cases}$$

kus $b = \max(\mathit{numbits}(c), |r_1|, |r_2|)$.

Kahe välistatud elemendi vahel defineeriti bitthaaval konjunktsioon järgnevalt:

$$(\llbracket p_1, p_2 \rrbracket \setminus \mathcal{E}_1) \mathbin{\&^{\#}} (\llbracket r_1, r_2 \rrbracket \setminus \mathcal{E}_2) = \begin{cases} \llbracket 0, \min(p_2, r_2) \rrbracket \setminus \emptyset & \text{, kui } p_1 \geq 0 \text{ ja } r_1 \geq 0 \\ \llbracket 0, p_2 \rrbracket \setminus \emptyset & \text{, kui } p_1 \geq 0 \\ \llbracket 0, r_2 \rrbracket \setminus \emptyset & \text{, kui } r_1 \geq 0 \\ (\llbracket p_1, p_2 \rrbracket \sqcup \llbracket r_1, r_2 \rrbracket) \setminus \emptyset & \text{, muidu.} \end{cases} \tag{13}$$

Bitthaaval OR välistatud ja määratud elemendi vahel defineeriti kahel juhul:

$$(\llbracket r_1, r_2 \rrbracket \setminus \emptyset) \mathbin{|^{\#}} c = \begin{cases} (\llbracket r_1, r_2 \rrbracket \sqcup \llbracket 0, \mathit{numbits}(c) \rrbracket) \setminus \emptyset & \text{, kui } c \geq 0 \\ \llbracket -b, b \rrbracket \setminus \emptyset & \text{, kui } c < 0, \end{cases}$$

kus $b = \max(\mathit{numbits}(c), |r_1|, |r_2|)$.

Kahe välistatud elemendi korral on definitsioon

$$(r_1 \setminus \mathcal{E}_1) \mathbin{|^{\#}} (r_2 \setminus \mathcal{E}_2) = (r_1 \sqcup r_2) \setminus \emptyset. \tag{14}$$

[11]Tehte $\oplus$ hulkade vahel saab defineerida kui $X \oplus Y = \{x \oplus y \mid x \in X, y \in Y\}$.

Bitthaaval XOR välistatud ja määratud elemendi defineeriti kui

$$(\llbracket r_1, r_2 \rrbracket \setminus \mathcal{E}) \,^{\wedge\#}\, c = \begin{cases} \llbracket 0, b \rrbracket \setminus \emptyset & \text{, kui } r_1 \geq 0 \text{ ja } c \geq 0 \\ \llbracket -b, b \rrbracket \setminus \emptyset & \text{, muidu,} \end{cases}$$

kus $b = \max(\mathit{numbits}(c), |r_1|, |r_2|)$.

Kahe välistatud elemendi vahel defineeriti

$$(\llbracket p_1, p_2 \rrbracket \setminus \mathcal{E}_1) \,^{\wedge\#}\, (\llbracket r_1, r_2 \rrbracket \setminus \mathcal{E}_2) = \begin{cases} \llbracket 0, \max(p_2, r_2) \rrbracket \setminus \emptyset & \text{, kui } p_1 \geq 0 \text{ ja } r_1 \geq 0 \\ \llbracket -b, b \rrbracket \setminus \emptyset & \text{, muidu,} \end{cases} \tag{15}$$

kus $b = \max(|p_1|, p_2, |r_1|, r_2)$.

Bitthaaval AND-i, OR-i ja XOR-i implementatsioonides on mitmeid sarnasusi vastavate intervallide domeeni implementatsioonidega. Analüüs on vähem granulaarne, sest bitiintervallideks abstraheerimine on täisarvude intervallidest robustsem. Bitiintervallide implementatsiooni tõttu pole ka võimalik vaadelda juhte, kus intervallid oleksid negatiivsed. Samuti saab märgata sarnasusi operaatori abstraktsiooni siseselt, kui argumentideks on välistatud ja määratud element või kaks välistatud elementi. Sarnasused tulenevad sellest, et konstanti c saab domeenis vaadelda ekvivalentse välistatud elemendina:

$$c \sim \begin{cases} \llbracket 0, \mathit{numbits}(c) \rrbracket \setminus (\mathbb{Z}_t \setminus \{c\}) & \text{, kui } c \geq 0 \\ \llbracket -\mathit{numbits}(c), \mathit{numbits}(c) \rrbracket \setminus (\mathbb{Z}_t \setminus \{c\}) & \text{, kui } c < 0. \end{cases}$$

Bitthaaval NOT defineeriti välistatud elemendil:

$$\sim^{\#}(\llbracket r_1, r_2 \rrbracket \setminus \mathcal{E}) = \llbracket -r_2, -r_1 \rrbracket \setminus \{\sim c \mid c \in \mathcal{E}\}.$$

Bittide pööramisel ei saa minna väärtus mälus suuremaks, seega suurimast võimalikust väärtusest saab operaatori rakendamise järel vähim võimalik väärtus ja vastupidi. Valem kehtib ainult siis, kui $-r_2 \leq 0$ ja $-r_1 > 0$, et intervallid vastaksid Goblinti implementatsiooni reeglitele.

3.4 Enums domeen

Enums domeeni elemendid on väga sarnased DefExc domeeni omadele. Element saab olla kas võimalike konstantide hulk (ingl *enumeration*) $\mathcal{I} \in \mathcal{P}(\mathbb{Z})$ või välistatud väärtuste hulk $\llbracket r_1, r_2 \rrbracket \setminus \mathcal{E}$, nagu see on defineeritud eelmises alapeatükis.

Implementatsioonid Enums domeenis olid sarnased DefExc domeeni omadega. Kuna välistatud elemendid Enums domeenis on identse ülesehitusega, siis on ka nende vahelised abstraktsed

operatsioonid identselt defineeritud. Operatsioonid, juhul kui argumendiks on konstantide hulk, defineeriti järgmiselt:

- **bitthaaval AND**:

$$[\![r_1, r_2]\!] \setminus \mathcal{E} \mathbin{\&^\#} \mathcal{I} = \bigsqcup_{i \in \mathcal{I}} ([\![r_1, r_2]\!] \setminus \mathcal{E}) \mathbin{\&^\#} i$$

- **bitthaaval OR**:

$$[\![r_1, r_2]\!] \setminus \mathcal{E} \mathbin{|^\#} \mathcal{I} = \bigsqcup_{i \in \mathcal{I}} ([\![r_1, r_2]\!] \setminus \mathcal{E}) \mathbin{|^\#} i$$

- **bitthaaval XOR**:

$$[\![r_1, r_2]\!] \setminus \mathcal{E} \mathbin{\wedge^\#} \mathcal{I} = \bigsqcup_{i \in \mathcal{I}} ([\![r_1, r_2]\!] \setminus \mathcal{E}) \mathbin{\wedge^\#} i$$

- **bitthaaval NOT**:

$$\sim^\# \mathcal{I} = \{\sim i | i \in \mathcal{I}\},$$

kus välistatud elemendi ja konstandi vahelised operatsioonid on defineeritud identselt eelmise alapeatükiga. Operatsioonid tehakse läbi iga hulga elemendiga eraldi ning leitakse saadud tulemuste ühine vähim ülatõke. Hulga elementide ühise vähima ülatõkke leidmist tähistab sümbol $\bigsqcup$, mis on üheselt määratud, tänu tehte $\sqcup$ assotsiatiivsusele [1]. Bithaaval NOT-i definitsioon konstantide loendil on otsene ja täpne, rakendades operaatorit igale hulga elemendile.

3.5 Bittväljade analüüs

Töö raames lisati Goblintile samuti bittväljade analüüsi funktsionaalsus. Bittväljade kasutamine realiseeriti igas eelnevalt nimetatud Goblinti domeenis. Bittväljad on relevantsed domeenide suurima elemendi ehk $\top$ arvutamisel. Suurimat elementi arvutatakse domeenides muutuja andmetüübi alusel, sõltuvalt muutuja võimalikust suurusvahemikust mälus. Bittväljade abil on võimalik muutujaid pakendada mäluüksustesse, mille suurus on väiksem nende andmetüübi omast. See kitsendab abstraktse domeeni suurimat elementi ning seeläbi kõiki domeeni kuuluvaid võimalikke väärtusi. Sel eesmärgil täiendati olemasolevat domeenide funktsiooni $\text{top\_of}_t : \mathcal{D}^\#$, mis tagastab $\top$ väärtuse sõltuvalt muutuja andmetüübist t. Funktsioonile lisati valikuline bittvälja parameeter. Seega on funktsioonil alternatiivne signatuur:

$$\text{top\_of}_t : \mathbb{N} \ \to \mathcal{D}^\#.$$

Kui bittvälja parameetrit ei ole argumendina korrektselt kujul antud, siis käitub funktsioon täpselt samamoodi nagu enne ehk tagastab andmetüübi suurusvahemiku. Bittvälja parameeter peab olema naturaalarv ning ei tohi ületada sellele vastava muutuja andmetüübi suurust.

Intervallide domeenis (ja analoogiliselt intervallihulkade domeenis) on funktsioon defineeritud kahel võimalusel, eeldusel, et argumendina on antud korrektne bittvälja väärtus b:

$$\text{top\_of}_t(b) = \begin{cases} [-2^{b-1}, 2^b - 1] & \text{, kui } t \text{ on märgiga} \\ [0, 2^b - 1] & \text{, kui } t \text{ on märgita.} \end{cases} \tag{16}$$

Goblinti parser ei tee semantiliselt vahet `signed int` ja `int` tüüpi muutujatel, sest peaaegu kõikjal C keeles tähendab `int` implitsiitselt `signed int` [5]. Nagu on välja toodud peatükis 2.1.2, siis bittväljade puhul tähendab `int` tüüp seda, et see võib sisaldada nii märgiga kui ka märgita väärtust. Seega on kõigi märgiga täisarvude jaoks ülemine tõke ikkagi $2^b - 1$, mitte $2^{b-1} - 1$, isegi kui väärtus on eksplitsiitselt `signed` tüüpi.

DefExc ja Enums domeenides on funktsioon defineeritud samuti sõltuvalt tüübist t:

$$\text{top\_of}_t(b) = \begin{cases} [\![-(b-1), b]\!] \setminus \emptyset & \text{, kui } t \text{ on märgiga} \\ [\![0, b]\!] \setminus \emptyset & \text{, kui } t \text{ on märgita.} \end{cases}$$

Sarnaselt intervallide domeenile on ka siin märgiga andmetüüpide puhul ülemine tõke sama, mis märgita andmetüüpidel: b, mitte $b - 1$, isegi kui muutuja on eksplitsiitselt `signed` tüüpi.

Lisaks parandati ka Goblinti kasutajakogemust, kuvades hoiatussõnum, kui programmis üritatakse bittväljale määrata väärtust, mis sinna ei mahu. Bittväljale liiga suure väärtuse määramine ei põhjusta kompileerimisviga ning see mis sellisel juhul tehakse, on implementatsioonist sõltuv [5]. Üks võimalus selleks on väärtuse algusest ära kärpida piisavalt bitte, et väärtus mahuks. Igal juhul ei ole enamasti tegu soovitava olukorraga, seega hoiatav sõnum on siin asjakohane.

Olgu `a` tüüpi t bittväli suurusega b. Kui väljale `a` üritatakse omistada väärtust n, kontrollitakse tingimust $n^\# \sqsubseteq^\# \text{top\_of}_t(b)$, kasutades eelnevalt defineeritud funktsiooni top_of_t alternatiivset implementatsiooni. $n^\#$ on siin omistatavale väärtusele vastav abstraktsioon analüüsitavas domeenis. $\top$ on abstraktse domeeni suurim element ning kui omistatav väärtus ei ole sellest väiksem, ei kuulu see domeeni ning tegu on ebakorrektse omistamisega. Kui eelnev tingimus kehtib, omistatakse väärtus, kui mitte, väljastatakse hoiatus kujul:

```
[Warning][Analyzer] Assigned value n exceeds the representable range of a
↪  b-bit bit-field.
```

Näiteks analüüsides programmi joonisel 6 intervallide domeenis, on bittvälja `b` abstraktse domeeni suurim väärtus $[-8, 15]$. Seos $\sqsubseteq^\#$ defineeritakse intervallide

```
1  struct S {
2      int b: 4;
3  }
4  int main() {
5      struct S s;
6      int x;
7      if (-20 > x || x  > 20) {
8          x = 0;
9      }
10     unsigned int y;
11     y = y & 5;
12     s.b = 16;
13     s.b = -10;
14     s.b = x;
15     s.b = y;
16 }
```

Joonis 6. Näidis bittväljadele ebakorrektsete väärtuste omistamisest.

```
[Warning][Analyzer] Assigned value 16 exceeds the representable range of a
↪  4-bit bit-field.
[Warning][Analyzer] Assigned value -10 exceeds the representable range of
↪  a 4-bit bit-field.
[Warning][Analyzer] Assigned value [-20,20] exceeds the representable
↪  range of a 4-bit bit-field.
```

Joonis 7. Näited bittväljadele ebakorrektsete väärtuste omistamise hoiatustest.

võrel kui $\subseteq$ (vt peatükk 2.2.3). Ridadel 14-15 väljastab Goblint hoiatuse, sest $[16, 16] \not\subseteq [-8, 15]$ ja $[-10, -10] \not\subseteq [-8, 15]$. Muutujale `x` vastab omistamise hetkel intervall $[-20, 20]$ ning $[-20, 20] \not\subseteq [-8, 15]$ seega väljastatakse ka siis hoiatus. Muutujale `y` vastab omistamise hetkel intervall $[0, 5]$ ja $[0, 5] \subseteq [-8, 15]$ seega omistamine on korrektne ning hoiatust ei väljastata. Väljastatud hoiatused on kujutatud joonisel 7.

4. Verifitseerimine

Käesolevas peatükis kirjeldatakse, kuidas veenduti, et oodatud funktsionaalsus on realiseeritud ja implementatsioon on korrektne valemi (3) järgi. Funktsionaalsust testiti käsitsi kirjutatud testide ja SV-COMP mõõtlusaluste abil. Korrektsuses veendumiseks kasutati Goblinti domeeniteste ja automaatset teoreemitõestajat Z3[12].

4.1 SV-COMP võrdlustestid

SV-COMP ehk *Competition on Software Verification* on võistlus, kus hinnatakse C ja Java programmide analüsaatoreid nende täpsuse, korrektsuse ja efektiivsuse põhjal. 2025. aasta võistluse aruande järgi oli 14. SV-COMP osaluse poolest ajaloo suurim võrdluslik hindamine oma erialal. Võistlusel testitakse analüsaatoreid mõõtlusalustel (ingl *benchmarks*), milleks on C või Java programmid, ning hinnatakse, kas analüsaator tagastas oodatud tulemuse. Hiliseimas aruandes toodi välja, et C programmide analüsaatoreid evalveeriti 33 353 mõõtlusaluse põhjal [7].

Goblintis on dokumenteeritud ebatäpsus kolme SV-COMP mõõtlusaluse suhtes kaustas *array-crafted*: *bAnd*, *bor* ja *xor*.[13] Joonisel 8 on kujutatud mõõtlusaluse *bAnd* testprogrammi relevantne osa. Programmi analüüsil hoiatab Goblint, et toimub märgiga täisarvu ületäitumine (ingl *signed integer overflow*). See juhtub sellepärast, et funktsiooni tagastustüüp on `int`, kuid kehas tagastatakse muutuja `res`, mis on tüüpi `long long`. Käesolevas programmis ei ole see aga oodatud tulemus. Real 4 saab muutuja `res` väärtuseks `int` tüüpi väärtuse `x[0]` ning edaspidi omistatakse sellele ainult bitthaaval konjunktsioone iseenda ja teiste `int` tüüpi muutujate vahel. Kuna bitthaaval konjunktsiooni käigus ei saa tulemuse suurus bittides olla suurem mõlema argumendi suurusest, ei saa muutuja `res` väljuda `int` tüübi väärtusvahemikust ning ületäitumist ei toimu.

Mõõtlusalused *bor* ja *xor* on analoogilised mõõtlusalusega *bAnd*, need erinevad vaid konjunktsiooni asemel vastavalt disjunktsiooni ja XOR-i kasutamise poolest. Goblint annab nende puhul samased valed tulemused ning nende seletus on analoogiline.

Eelnevalt nimetatud mõõtlusaluseid kasutati suunisena domeenide täiendamisel. Nende eeskujul implementeeriti DefExc domeenis valemid (13), (14) ja (15) ning intervallide domeenides

[12] https://github.com/Z3Prover/z3
[13] https://github.com/goblint/analyzer/issues/1586

```c
int bAnd (int x[100]) {
  int i;
  long long res;
  res = x[0];
  for (i = 1; i < 100; i++) {
    res = res & x[i];
  }
  return res;
}
```

Joonis 8. SV-COMP mõõtlusaluse *bAnd* testprogramm.[14]

valemid (6), (10) ja (11). Pärast neid implementatsioone Goblint enam ületäitumise hoiatusi ei andnud ning määras muutujaile korrektse väärtusvahemiku.

Töö valmimisel evalveeriti seda samuti tervel SV-COMP mõõtlusaluste kogumil ning võrreldes varasemaga, suutis Goblint lahendada 26 uut ülesannet, mis on loetletud lisas I. Nende seas aga ei olnud selles peatükis nimetatud mõõtlusaluseid, sest, kuigi hoiatus kadus ning väärtusvahemik on korrektne, oli testide läbimisel veel lisatingimusi, mille implementeerimine on selle töö skoobist väljas.

4.2 Funktsionaalsed regressioonitestid

Goblinti regressioonitestide seas on implementeeritud funktsionaalsed testid, mis kindlustavad, et funktsionaalsus töötab eeldatult [8]. Testid on C programmide kujul ning kontrollitakse, et Goblint analüüsib neid oodatud täpsusega. Loodi kuus testide komplekti, üks iga uuritava operatsiooni jaoks ning üks bittväljade jaoks. Igas komplektis testitakse läbi ükshaaval kõik eelnevalt välja toodud juhud eri väärtustel, sealhulgas piirjuhud, kui muutuja väärtus võib olla näiteks `0`, `-1`, `1`, `__LONG_LONG_MAX__` või `-__LONG_LONG_MAX__-1`.

Joonisel 9 on kujutatud funktsionaalne test bittväljade jaoks. Ridadel 11–12 kontrollitakse, et bittvälja väärtus jääb vahemikku, mis on määratletud valemiga (16). Ridadel 14–16 kontrollitakse, et Goblint väljastab hoiatuse, kui bittväljale üritatakse määrata väärtust, mis sellesse ei mahu.

[14] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/458b5991f0771aa940491eacb4fb998616e33c72/c/array-crafted/bAnd1.i

```c
struct S {
    int a: 3;
};

int main(void) {
    int x;
    struct S s;
    if (x < -30 || x > 30) {
        x = 0;
    }
    __goblint_check(s.a <= 7);
    __goblint_check(s.a >= -4);
    s.a = 1; // NOWARN
    s.a = 8; // WARN
    s.a = -5; // WARN
    s.a = x; // WARN
    return 0;
}
```

Joonis 9. Bittväljade funktsionaalne test (lühendatud).

Goblint peaks hoiatama ka siis, kui bittvälja üritatakse mahutada muutujat, mille väärtus võib, aga ei pruugi, mahtuda bittvälja sisse. Real 13 kontrollitakse, et korrektse väärtuse korral ei anta valeteadet. Annotatsioon `WARN` ridade järel annab testiskriptile teada, et nende ridade korral peaks Goblint andma hoiatuse, ning annotatsioon `NOWARN`, et rea korral hoiatust tekkida ei tohi. Tõepoolest, kui testi jooksutada, siis Goblint väljastab vajalike annoteeritud ridade puhul hoiatused (vt joonis 10). Kolmandas hoiatuses on omistatava väärtuse kohal paarina selle abstraktsed väärtused DefExc ja intervallide domeenides.

```
[Warning][Analyzer] Assigned value 8 exceeds the representable range of a
↪  3-bit bit-field.
[Warning][Analyzer] Assigned value -5 exceeds the representable range of a
↪  3-bit bit-field.
[Warning][Analyzer] Assigned value (Unknown int([-7,31]), [-30,30])
↪  exceeds the representable range of a 3-bit bit-field.
```

Joonis 10. Bittväljade testi (vt joonis 9) jooksutamisel väljastatavad Goblinti hoiatused.

4.3 Domeenitestid

Täisarvudomeenide operatsioonide korrektsuse testimiseks on Goblintis implementeeritud domeenide omaduspõhised testid [8]. Omaduspõhine testimine (ingl *property-based testing*) on testimise metoodika, mille käigus verifitseeritakse programmi omadusi juhuslikult genereeritud testandmetel [9]. Operatsioonide korrektsus on määratletud valemiga (3), kuid selles kasutatud konkretiseerimisfunktsiooni γ ei ole võimalik mälu ja arvutusvõimsuse piiranguid arvestades asjalikult realiseerida, sest see võib hõlmata väga suurte hulkade genereerimist. Korrektsuse saab defineerida alternatiivselt kasutades abstraheerimisfunktsiooni $\alpha : \mathcal{D} \to \mathcal{D}^{\#}$ [2][15]:

$$\forall X, Y \in \mathcal{D} : \alpha(X \ \oplus \ Y) \sqsubseteq \alpha(X) \oplus^{\#} \alpha(Y).$$

Domeenide testimise metoodikat on kirjeldanud Saan [9]. Funktsioonide korrektsus esitatakse predikaatidena, lähtudes välja toodud definitsioonist. Predikaatidele genereeritakse juhuslikult argumente, sealhulgas ka sageli probleeme tekitavaid piirjuhte. Kui predikaat on kõigil argumentidel tõene, läheb test läbi ning omadus loetakse kehtivaks. Kui mingil argumendil predikaat on väär, siis test kukub läbi ning vastunäide kuvatakse kasutajale. Domeenitestid töös ühtegi viga ei leidnud.

Omaduspõhised testid on efektiivne viis domeenis vigade leidmiseks, kuid nende juhusliku olemuse tõttu ei anna need täielikku kindlust, et omadused kehtivad. On võimalik, et leidub kontranäide, mille korral korrektsus ei kehti, kuid see jääb juhuvalimist välja. Selle võimaluse elimineerimiseks on vaja kasutada tugevamat meetodit.

4.4 Korrektsuse tõestamine

Selleks, et kindlustada defineeritud abstraktsete operatsioonide korrektsus, kasutati automaatset teoreemitõestajat Z3. Z3 valiti sellepärast, et selles on võimalik tõestada omadusi bitivektorite kohta ning sellel on kasutajasõbralik Pythoni liides. Samuti kasutati Pythoni teeki pytest[16], mille abil muudeti tõestused modulaarseks ning mugavaks kasutada.

Z3 on SMT lahendaja. SMT probleem (ingl *satisfiability modulo theories problem*) on üldistus lausearvutuse kehtestatavuse probleemist (SAT), lisades sellele juurde aritmeetika, fikseeritud suurusega bitivektorid ja teised teooriad [10]. SMT lahendajad teevad kindlaks nende teooriate

[15] Definitsiooni on lihtsustatud jättes välja teatavad funktsiooni α lisaomadused.
[16] https://pypi.org/project/pytest/

valemite kehtestatavuse [10]. Bitivektorite lahendamiseks kasutab Z3 *bit-blasting* tehnikat, mis tähendab, et bitivektor teisendatakse lausearvutuse valemiks, kus igale bitile vastab lausemuutuja [11].

Abstraktsete operatsioonide korrektsuse kontrollimiseks antakse lahendajale kitsenduste süsteem, mis kirjeldab abstraktset domeeni ning otsitakse vastunäidet kontrollitavale omadusele. Kui lahendaja leiab väärtused, mille korral kitsenduste süsteem on kehtestatav, siis järelikult omadus ei kehti ning selle põhjal loodud abstraktsioon ei ole korrektne. Kui süsteem ei ole kehtestatav, siis järelikult omadus kehtib kõigi võimalike väärtuste korral ning abstraktsioon on korrektne. Siinkohal ei kasutata korrektsuse tõestamiseks kumbagi eelnevalt defineeritud valemit, vaid Z3 garanteerib korrektsuse, kontrollides kitsenduste süsteemi kehtestatavust mehaaniliselt kõigil võimalikel bitivektorite väärtustel.

Joonisel 11 on kujutatud korrektsuse tõestus abstraktsele mittenegatiivsete intervallide bitthaaval disjunktsioonile, defineeritud valemiga (7). `x` ja `y` on suvalised väärtused vastavalt kummastki intervallist bitivektori kujul. Intervalle on tõestuses jäljendatud, kasutades muutujaid `lx` ja `ux` (`ly` ja `uy`), mis sümboliseerivad vastavalt esimese (teise) intervalli alumist ja ülemist raja. Funktsioon `initialize_interval` loob muutujad intervallide rajade jaoks ning lisab tõestajale kitsendused, et need jäljendaksid intervalli (`lx <= x <= ux`). Real 6 lisatud kitsendus jäljendab seda, et mõlemad intervallid peavad olema mittenegatiivsed. Real 8 leitakse suvaliste bitivektorite `x` ja `y` bitthaaval disjunktsioon. Funktsiooni `BV2Int`, mis teisendab bitivektori täisarvuks, teine argument **`True`** tähendab seda, et kahendarv tõlgendatakse kui märgiga täisarv täiendkoodis. Ridadel 9–10 arvutatakse vastavalt ülemine ja alumine raja valemi (7) järgi. Real 12 lisatakse kitsendus, et tulemus peab kuuluma leitud intervalli. Kuna otsitakse vastunäidet, on kitsendus mähitud `Not` konnektiivi sisse. Tõestuse lõpus kontrollitakse, et lahend oleks väärtusega `unsat` ehk kitsenduste süsteem ei oleks kehtestatav.

Lisas II on korrektsuse tõestus DefExc domeenis kahe välistatud elemendi konjunktsioonile (defineeritud valemiga (13)) viimasel juhul, kui argumendid ei vasta ühelegi välja toodud täiendavale eeldusele. Taas `x` ja `y` sümboliseerivad suvalisi bitivektoreid. Muutujad `x_lower_bit_range` ja `x_upper_bit_range` jäljendavad DefExc domeeni abstraktset elementi kujul $[\![i_1, i_2]\!]$. Kuna valemis kujutatakse välistushulk $\mathcal{E}$ tühjaks hulgaks, siis seda pole siinkohal vaja. Muutujad `x_lower_int_range` ja `x_upper_int_range` jäljendavad sellele vastavat täisarvuintervalli, definitsiooni (12) järgi. Analoogiliselt tehakse muutuja `y` jaoks. Nende intervallide esituste üles seadmiseks on funktsioon `initialize_bit_interval`, mis tagastab

```
def test_logor_both_non_negative():
    s = Solver()
    x, y = BitVecs("x y", BIT_VECTOR_LENGTH)
    lx, ux = initialize_interval(s, x, "x")
    ly, uy = initialize_interval(s, y, "y")
    s.add(lx >= 0, ly >= 0)

    dis = BV2Int(x | y, True)
    lower = Max(BV2Int(lx, True), BV2Int(ly, True))
    upper = BV2Int(max_val_bit_constrained(Max(ux, uy)), True)

    s.add(Not(And(lower <= dis,dis <= upper)))
    assert s.check() == unsat, f'Counterexample: {s.model()}'
```

Joonis 11. Kahe mittenegatiivse intervalli bitthaaval OR-i korrektsuse tõestus.

vajalikud muutujad ning lisab lahendajasse vajalikud kitsendused, et need jäljendaks täpselt vajalikke abstraktseid elemente. Real 25 leitakse vajalik tulemuse bitiintervall vastavalt valemile (13), milleks on intervallide vähim ülatõke (vt valem (2)). Ridadel 27–28 leitakse vähima ülatõkke esitus täisarvuintervallina, definitsiooni (12) põhjal. Real 30 leitakse väärtuse vaheline bitthaaval konjunktsioon ning real 32 lisatakse lahendajasse kitsendus, et konjunktsiooni tulemus peab jääma valemis defineeritud intervalli. Kui kitsenduste süsteem ei ole kehtestatav, siis test loetakse läbituks.

Kõik tõestused on leitavad eraldi Github repositooriumis (vt Lisa III). Ülejäänud omaduste tõestused on sarnased kirjeldatud näidetega. Kõik tõestused annavad positiivse tulemuse ehk implementeeritud loogika on korrektne.

5. Kokkuvõte

Töö eesmärk oli täiendada bitioperatsioonide käsitlust, et muuta Goblinti staatiline analüüs täpsemaks. Tarkvara verifitseerijate täpsuse parandamine on oluline, et käia kaasas tarkvarasüsteemide arenguga ning tõsta olemasolevate tarkvarade kvaliteedistandardeid.

Töös käsitleti nelja Goblinti abstraktset täisarvudomeeni ning nendes täiendati abstraktseid loogilisi bitioperatsioone: bitthaaval AND, OR, XOR ja NOT ning parempoolne loogiline bitinihe. Selle jaoks uuriti bitioperatsioonide omadusi täisarvudel, otsiti mustreid nende käitumises abstraktsete domeenide elementidel ning implementeeriti leitud omadused. Lisaks bitioperatsioonide abstraktsioonide täiendamisele, lisati Goblintile ka bittväljade analüüsi funktsionaalsus. Goblint suudab nüüd efektiivselt bittvälju analüüsida ning aitab arendajaid, väljastades hoiatusi kui bittvälja üritatakse omistada ebakorrektset väärtust.

Tulemusi evalveeriti SV-COMP mõõtlusalustel ning pärast töö valmimist suutis Goblint lahendada 26 uut ülesannet. Samuti koostati funktsionaalseks testimiseks komplekt testprogramme, mida Goblint suutis täienduste järel täpsemalt analüüsida. Kõigi implementeeritud analüüside korrektsus kinnitati Goblinti omaduspõhiste testide abil ning formaalsete tõestustega Z3 teoreemitõestajas.

Töö raames täiendatud analüüse saab ka tulevikus edasi arendada, sest arvatavasti ei saavutatud töö jooksul operatsioonidele kõige täpsem võimalik analüüs. Bitioperatsioonide analüüs täisarvudel on ilmselt sügavam selles töös käsitletust ning edasisel uurimisel on võimalik leida detailsemaid mustreid, mille põhjal saab Goblinti staatilist analüüsi veelgi täpsustada.

Viited

[1] Miné A. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. en. *Foundations and Trends® in Programming Languages* 4.3-4 (2017), lk 120–372. DOI: 10.1561/2500000034. http://www.nowpublishers.com/article/Details/PGL-034 (04.04.2025).

[2] Cousot P. ja Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. en. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*. Los Angeles, California: ACM Press, 1977, lk 238–252. DOI: 10.1145/512950.512973. http://portal.acm.org/citation.cfm?doid=512950.512973 (04.04.2025).

[3] Vojdani V., Apinis K., Rõtov V., Seidl H., Vene V. ja Vogler R. Static race detection for device drivers: the Goblint approach. en. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Singapore Singapore: ACM, august 2016, lk 391–402. DOI: 10.1145/2970276.2970337. https://dl.acm.org/doi/10.1145/2970276.2970337 (25.04.2025).

[4] Kernighan B. W. ja Ritchie D. M. The C programming language. 2nd ed. Englewood Cliffs, N.J: Prentice Hall, 1988.

[5] ISO/IEC JTC1/SC22/WG14. Programming languages — C (Working Draft). en. Tehniline raport N3096. International Organization for Standardization (ISO), 2023. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf (12.04.2025).

[6] Saan S. Abstraktne interpretatsioon. 2024. https://courses.cs.ut.ee/2024/AKTSP/spring/Main/HomePage?action=download&upname=abstraktne-interpretatsioon.pdf (07.12.2024).

[7] Beyer D. ja Strejček J. Improvements in Software Verification and Witness Validation: SV-COMP 2025. en. *Tools and Algorithms for the Construction and Analysis of Systems*. Toim. Gurfinkel A. ja Heule M. Köide 15698. Series Title: Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2025, lk 151–186. DOI: 10.1007/978-3-031-90660-2_9. https://link.springer.com/10.1007/978-3-031-90660-2_9 (12.05.2025).

[8] Goblint documentation. https://goblint.readthedocs.io/en/latest/ (30.04.2025).

[9] Saan, Simmo. Abstraktsete domeenide omaduspõhine testimine. Bakalaureusetöö. Tartu Ülikool, 2018. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=61841&year=2018 (30.04.2025).

[10] De Moura L. ja Bjørner N. Z3: An Efficient SMT Solver. en. *Tools and Algorithms for the Construction and Analysis of Systems*. Toim. Hutchison D., Kanade T., Kittler J., Kleinberg J. M., Mattern F., Mitchell J. C., Naor M., Nierstrasz O., Pandu Rangan C., Steffen B., Sudan M., Terzopoulos D., Tygar D., Vardi M. Y., Weikum G., Ramakrishnan C. R. ja Rehof J. Köide 4963. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, lk 337–340. DOI: 10.1007/978-3-540-78800-3_24. http://link.springer.com/10.1007/978-3-540-78800-3_24 (30.04.2025).

[11] Bjørner N., Moura L. de, Nachmanson L. ja Wintersteiger C. M. Programming Z3. en. *Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures*. Toim. Bowen J. P., Liu Z. ja Zhang Z. Cham: Springer International Publishing, 2019, lk 148–201. DOI: 10.1007/978-3-030-17601-3_4. https://doi.org/10.1007/978-3-030-17601-3_4 (30.04.2025).

Lisad

I Uued töö implementeerimise järel õnnestuvad SV-COMP mõõtlusalused

Programmi nimi	Omadus
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_change_pc8736x_gpio_configure	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_change_pc8736x_gpio_current	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_change_pc8736x_gpio_get	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_change_pc8736x_gpio_set	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_configure_pc8736x_gpio_current	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_configure_pc8736x_gpio_get	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_configure_pc8736x_gpio_set	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_current_pc8736x_gpio_get	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_current_pc8736x_gpio_set	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_get_pc8736x_gpio_set	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_open_pc8736x_gpio_change	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_open_pc8736x_gpio_current	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_open_pc8736x_gpio_get	no-overflow
pthread-driver-races/char_pc8736x_gpio_pc8736x_gpio_open_pc8736x_gpio_set	no-overflow
array-tiling/mlceu	no-overflow
array-tiling/mlceu2	no-overflow
bitvector/soft_float_1-2a.c.cil	no-overflow
bitvector/soft_float_1-3a.c.cil	no-overflow
bitvector/soft_float_2a.c.cil	no-overflow
bitvector/soft_float_3a.c.cil	no-overflow
bitvector/soft_float_4-2a.c.cil	no-overflow
bitvector/soft_float_4-3a.c.cil	no-overflow
bitvector/soft_float_5a.c.cil	no-overflow
termination-bwb/not-04	termination
array-tiling/mlceu2	unreach-call
hardware-verification-bv/btor2c-lazyMod.zaher	unreach-call

II DefExc domeeni korrektsuse tõestuse näide

```
def initialize_bit_interval(s, x, name="x"):
    x_int = BV2Int(x, True, "x")
    x_lower_bit_range, x_upper_bit_range = BitVecs(f'l{name} u{name}', BIT_VECTOR_LENGTH)

    s.add(x_lower_bit_range <= 0, x_upper_bit_range > 0)

    s.add(Abs(BV2Int(x_lower_bit_range, True)) <= BIT_VECTOR_LENGTH,
        x_upper_bit_range <= BIT_VECTOR_LENGTH,
        x_lower_bit_range <= x_upper_bit_range)

    x_lower_int_range = get_int_bound_from_bits(x_lower_bit_range)
    x_upper_int_range = get_int_bound_from_bits(x_upper_bit_range)

    s.add(x_int <= x_upper_int_range, x_int >= x_lower_int_range)
    return x_lower_bit_range, x_upper_bit_range, x_lower_int_range, x_upper_int_range

def test_logand_exc_exc_otherwise():
    s = Solver()
    x = BitVec("x", BIT_VECTOR_LENGTH)
    y = BitVec("y", BIT_VECTOR_LENGTH)

    x_lower_bit_range, x_upper_bit_range, x_lower_int_range, x_upper_int_range =
    ↪  initialize_bit_interval(s, x)
    y_lower_bit_range, y_upper_bit_range, y_lower_int_range, y_upper_int_range =
    ↪  initialize_bit_interval(s, y, "y")

    join_lower, join_upper = interval_join([x_lower_bit_range,
    ↪  x_upper_bit_range],[y_lower_bit_range, y_upper_bit_range])

    join_lower_int = get_int_bound_from_bits(join_lower)
    join_upper_int = get_int_bound_from_bits(join_upper)

    con = BV2Int(x & y, True)

    s.add(Not(And(con <= join_upper_int, con >= join_lower_int)))

    assert s.check() == unsat, f'Counterexample: {s.model()}'
```

III Lähtekood

Goblinti täiendamiseks tehti tõmbetaotlus Goblinti repositooriumile, mis on saadaval siin: https://github.com/goblint/analyzer/pull/1739.

Valemite tõestused Z3 lahendajaga asuvad eraldi GitHub repositooriumis: https://github.com/H-Innos/Z3.

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Henrik Innos**,

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

   „Bitioperatsioonide analüüsi täiendamine Goblintis",

   mille juhendaja on Simmo Saan,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada Tartu Ülikooli digitaalarhiivi kuni autoriõiguse kehtivuse lõppemiseni;

2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, 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.

Henrik Innos
14.05.2025