TARTU ÜLIKOOL

Arvutiteaduse instituut

Informaatika õppekava

Ronald Judin

Laiendamisoperaatorid abstraktses interpretaatoris Goblint

Bakalaureusetöö (9 EAP)

Juhendaja: Simmo Saan

Tartu 2024

Laiendamisoperaatorid abstraktses interpretaatoris Goblint

Lühikokkuvõte:

Programmide staatilist analüüsi kasutatakse, et tarkvara korrektsuses ja ohutuses veenduda. Abstraktne interpretatsioon on staatilise analüüsi meetod, mida kasutab abstraktne interpretaator Goblint. Laiendamine on abstraktses interpretatsioonis viis tsüklite analüüsi koondumise kindlustamiseks. Goblintis on defineeritud palju laiendamisoperaatoreid. Selles töös on neid kirjeldatud ning realiseeritud uusi. Realiseeritud laiendamisoperaatoreid testiti edukalt.

Võtmesõnad:

staatiline analüüs, abstraktne interpretatsioon, Goblint

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

Widening Operators in the Abstract Interpretator Goblint

Abstract:

Static analysis of programs is used to ensure the correctness and safety of software. Abstract interpretation is a method of static analysis used by the abstract interpretator Goblint. Widening is a method in abstract interpretation to ensure the convergence of analysis of cycles. There are many widening operators defined in Goblint. This paper describes them and the implementation of new ones. The newly implemented widening operators were successfully tested.

Keywords:

static analysis, abstract interpretation, Goblint

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

Sisukord

Sissejuhatus

Automaatne staatiline analüüs on tarkvaraarenduses laialdaselt kasutuses [1]. Staatilise analüüsi automatiseerimine soodustab koodis vigade leidmist ja parandamist, parandab tarkvara turvalisust, võimaldab tööaja tõhusamat kasutamist ning lihtsustab kvaliteedikontrolli [2].

Staatilise analüüsi meetodite seas on abstraktne interpretatsioon, mille defineerisid formaalselt Cousot ja Cousot aastal 1977 [3]. Abstraktse interpretatsiooni raames tsüklite analüüsi termineeruvuse tagamiseks kasutatakse laiendamist. Funktsioone, mille alusel laiendatakse, nimetatakse laiendamisoperaatoriteks.

Abstraktset interpretatsiooni kasutab teiste seas Tartu Ülikooli ja Müncheni Tehnikaülikooli koostöös arendatav abstraktne interpretaator Goblint [4]. Bakalaureusetöö eesmärk on Goblintis realiseeritud laiendamisoperaatorite võrdlus abstraktse interpretatsiooni kirjandusega ja Goblinti täiendamine uute laiendamisoperaatoritega. Töö raames analüüsitakse kirjanduses defineeritud laiendamisoperaatoreid ja realiseeritakse neid Goblintis. Realiseeritud laiendamisoperaatoreid võrreldakse varem olemasolevatega, kasutades mõõdikuna laiendamist kasutava staatilise analüüsi täpsust.

Bakalaureusetöö on jagatud kolmeks peatükiks. Esimeses peatükis antakse teoreetiline ülevaade staatilise analüüsi, abstraktse interpretatsiooni ja laiendamise kohta. Teises peatükis tutvustatakse abstraktset interpretaatorit Goblint ja kirjeldatakse selles olemasolevaid laiendamisoperaatoreid. Kolmandas peatükis kirjeldatakse laiendamisoperaatoreid, mida Goblintis ei olnud, nende implementeerimise protsessi ja testimise tulemusi.

Töö lisas I on viide Goblinti lähtekoodile ja selle töö raames valminud täiendustele.

1. Teoreetiline ülevaade

Arvutiprogrammide staatiline analüüs on programmide analüüs ilma neid käivitamata. Staatilist analüüsi saab teha käsitsi või automaatselt [5]. Automaatne staatiline analüüs on käsitsi analüüsist ajaliselt tõhusam. Teisalt ei saa automaatne staatiline analüüs üldjuhul olla täiuslik, sest Rice'i teoreemi järgi ei ole Turingi-täielike keelte ükski mittetriviaalne omadus lahenduv [6].

Järgnev lõik tugineb Minéle [5]. Abstraktne interpretatsioon on semantilise staatilise analüüsi meetod. Teisisõnu on abstraktse interpretatsiooni eesmärk programmide omaduste formaalne tõestamine. Abstraktne interpretatsioon hõlmab programmi lihtsustatud kujul ehk abstraheeritult esitamist ja analüüsimist. Lihtsustatud programmi analüüs on algse programmi analüüsist vähem kulukas ja annab mingit infot algse programmi käitumise kohta. Õigesti abstraheeritud programmi staatiline analüüs on korrektne: kui analüüs programmi käitumises mingit otsitavat aspekti ei tuvasta, siis ei ole seda programmis tõepoolest. Samas pole abstraktsel interpretatsioonil põhinev analüüs täielik. See tähendab, et analüüs võib anda valepositiivseid tulemusi.

Programmis esinevad muutujad võivad programmi täitmise käigus saada erinevaid väärtusi. Ühe muutuja võimalikud väärtused on kirjeldatavad osalise järjestusega [3]. Hulga X osaline järjestus $\sqsubseteq$ on seos $\sqsubseteq \subset X \times X$, mis on

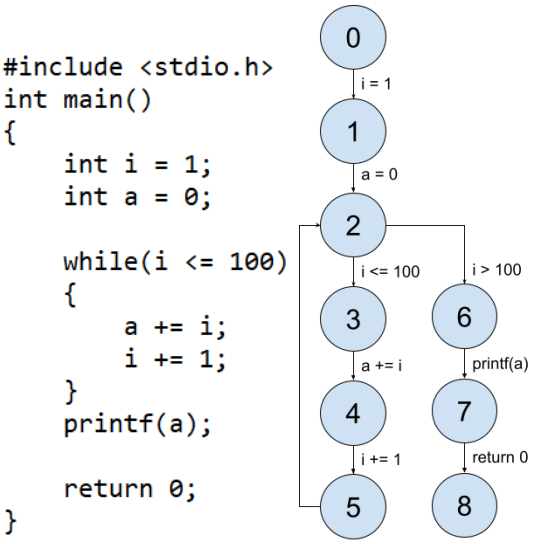
1. refleksiivne: $\forall x \in X: x \sqsubseteq x$;
2. antisümmeetriline: $\forall x, y \in X: (x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow x = y$;
3. transitiivne: $\forall x, y, z \in X: (x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow x \sqsubseteq z$.

Järjestatud hulka võib esitada kujul $(X, \sqsubseteq)$, kus X on hulk ja $\sqsubseteq$ sellel defineeritud järjestus, mis võib olla osaline või täielik.

Ühe muutuja võimalikke väärtuseid kirjeldav osaliselt järjestatud hulk on nende väärtuste kõigi hulkade hulk, mis on järjestatud sisalduvusseose järgi [5]. Abstraktse interpretatsiooni käigus arvutatakse programmi iga muutuja jaoks selle võimalike väärtuste mingi ülemhulk igal programmi täitmise sammul [7]. Selleks valitakse esmalt muutuja iga võimaliku väärtuse jaoks mingi seda kirjeldav element ehk abstraktne väärtus (näiteks täisarvulise muutuja

väärtust 1 võib esitada selle märgina + või intervallina [1,1] [3]), täidetakse programmi käsud saadud abstraktseid väärtusi kasutades ja võetakse kõikvõimalike abstraktsete väärtuste vähim ülemtõke [5].

Ülalkirjeldatut näitlikustab joonisel 1 kujutatud programmi analüüs.



Joonis 1. C programm, mis arvutab ja väljastab esimese 100 arvu summa, ja selle programmi juhtvoograaf.

Joonisel 2 on kujutatud selle programmi lihtsa intervallanalüüsi tulemus.

```
#include <stdio.h>
int main()
{
    int i = 1;
    int a = 0;
    // i ∈ [1, 1], a ∈ [0, 0]

    while(i <= 100)
    {
        // i ∈ [1, 100], a ∈ [0, 4950]
        a += i;
        i += 1;
        // i ∈ [2, 101], a ∈ [1, 5050]
    }
    // i ∈ [101, 101], a ∈ [0, 5050]
    printf(a);

    return 0;
}
```

Joonis 2. Joonisel 1 kujutatud programmi intervallanalüüs.

Konstantide omistamiste järel programmi alguses saavad muutujad i ja a konkreetsed väärtused, mis on abstraktselt kirjeldatavad üheelemendiliste intervallidena. Tsükli sees täiendatakse muutujate i ja a võimalikke väärtusi igal täitmiskorral. Muutujale a muutuja i väärtuse liitmist kirjeldatakse abstraktselt intervallide [1, 100] ja [0, 4950] liitmisena. Intervallide [l1, h1] ja [l2, h2] summa on defineeritud järgmiselt:

$$[l1, h1] + [l2, h2] = [l1 + l2, h1 + h2], \text{ kusjuures } -\infty + \_ = -\infty \text{ ja } \infty + \_ = \infty.$$

Seega on kahe intervalli summa alampiiriks nende alampiiride summa ja ülempiiriks nende ülempiiride summa.

1.1. Abstraktsed domeenid

Osaliselt järjestatud hulga $(X, \sqsubseteq)$ kahe elemendi x ja y ülemtõke on element z, mille puhul $x \sqsubseteq z$ ja $y \sqsubseteq z$. Element z on elementide x ja y vähim ülemtõke ehk ülemraja, kui lisaks iga elemendi w puhul $x \sqsubseteq w \wedge y \sqsubseteq w \Rightarrow z \sqsubseteq w$. Võre on osaliselt järjestatud hulk, mille igal kahel elemendil on vähim ülemtõke ja suurim alamtõke. Võre on täielik, kui selle igal elementide hulgal on vähim ülemtõke ja suurim alamtõke.

Miné [5] järgi järeldub täieliku võre definitsioonist, et selles on alati suurim ja vähim element. Muutujate väärtuste esitamiseks täieliku võrena võib nende osaliselt järjestatud hulgale suurima ja/või vähima elemendi puudumisel need kunstlikult lisada. Suurim element $\top$ tähendab, et muutuja konkreetse väärtuse kohta ei ole midagi teada. Vähim element $\bot$ tähendab Miné järgi programmi täitmise käigus võimatut olukorda ja võib analüüsi vahetulemuseks olla, kui mõnda käsku programmis ei täideta kunagi.

Sõltuvalt sellest, millist infot staatilisest analüüsist taotletakse, tuleb analüüsiks valida abstraktne domeen. Abstraktne domeen koosneb Miné [5] järgi järgmistest osadest:

1. arvutis esitatavate abstraktsete väärtuste hulk $D^{\#}$;
2. hulgal $D^{\#}$ defineeritud tõhus osaline järjestus $\sqsubseteq^{\#}$;
3. hulgal $D^{\#}$ defineeritud korrektsed ja tõhusad vähima ülemtõkke ja suurima alamtõkke funktsioonid;
4. vähim element $\bot^{\#} \in D^{\#}$ ja suurim element $\top^{\#} \in D^{\#}$;

5. abstraktsed operatsioonid domeeni elementidega ehk üleminekufunktsioonid, nt omistamine ja liitmine;
6. laiendamisoperaator ∇ (defineeritakse allpool).

Joonisel 2 kirjeldatud intervallanalüüsi tarbeks valitud abstraktsel domeenil on need osad järgmised:

1. intervallide hulk $\{[x, y] \mid x, y \in \mathbb{Z} \cup \{\infty, -\infty\} \wedge x \leq y\} \cup \{\bot\}$;
2. sisalduvusseos: $[l_1, h_1] \sqsubseteq [l_2, h_2]$, kui $l_1 \geq l_2$ ja $h_1 \leq h_2$;
3. vähim ülemtõke: $[l_1, h_1] \sqcup [l_2, h_2] = [\min l_1\ l_2, \max h_1\ h_2]$;
   suurim alamtõke: $[l_1, h_1] \sqcap [l_2, h_2] = [\max l_1\ l_2, \min h_1\ h_2]$;
4. kunstlik vähim element $\bot$ ja triviaalne suurim element $\top = [-\infty, \infty]$;
5. ülalkirjeldatud intervallide omistamine ja liitmine;
6. laiendamisoperaator defineeritakse samuti allpool.

Paljud programmid sisaldavad tsükleid. Staatilises analüüsis uuritakse teiste programmi olekute seas tsüklite invariante - omadusi, mis on tõesed tsükli iga täitmiskorra eel ja järel. Invariandid selgitatakse abstraktses interpretatsioonis välja tsükli täitmist simuleeriva funktsiooni vähima püsipunkti arvutamise kaudu [5].

Tabel 1. Näidisprogrammi staatilise analüüsi protsess tsükli sees.

Täitmiskord	Alguses	Lõpus
1	$i \in [1, 1], a \in [0, 0]$	$i \in [1, 2], a \in [1, 1]$
2	$i \in [1, 2], a \in [0, 1]$	$i \in [1, 3], a \in [1, 3]$
3	$i \in [1, 3], a \in [0, 3]$	$i \in [1, 4], a \in [1, 6]$
⋮	⋮	⋮
100	$i \in [1, 100], a \in [0, 4950]$	$i \in [1, 101], a \in [1, 5050]$
101	$i \in [1, 100], a \in [0, 4950]$	$i \in [1, 101], a \in [1, 5050]$

Funktsiooni f püsipunkt on selline punkt x, et f(x)=x. Püsipunkti arvutamiseks võetakse programmi kõikide olekute hulk, milles programm võib vahetult enne tsüklit olla, ja

rakendatakse sellel iteratiivselt tsükli täitmist simuleerivat funktsiooni, kuni jõutakse püsipunktini [5].

Näidisprogramm on enne tsüklit olekus, kus i = 1 ja a = 0. Tabel 1 kirjeldab muutujate i ja a arvutatud võimalikke väärtusi tsükli erinevate täitmiskordade alguses ja lõpus. Püsipunkt leitakse abstraktse funktsiooni 101. rakendamisel.

Püsipunkti sel viisil arvutamine võib võtta ebamõistlikult palju aega, kui tsüklit täidetakse programmi töö käigus palju kordi, või mitte lõppeda, kui programmis on lõpmatu tsükkel [8].

1.2. Laiendamisoperaatorid

Ahel on osaliselt järjestatud hulga täielikult järjestatud alamhulk. Võre võib olla lõpliku või lõpmatu kõrgusega. Lõpmatu kõrgusega võres leidub lõpmatuid ahelaid ja lõpliku kõrgusega võres ei leidu. Intervallide sisalduvusseose järgi järjestatud võre on lõpmatu kõrgusega, sest $[0, 0] \sqsubseteq [0, 1] \sqsubseteq [0, 3] \sqsubseteq [0, 6] \sqsubseteq \ldots \sqsubseteq [0, \infty]$. Lõpliku kõrgusega on näiteks lame täisarvude võre, kus iga kaks täisarvu on omavahel võrreldamatud ning suurim ja vähim element on kunstlikud.

Lõpmatu kõrgusega võrede korral on kasulik püsipunkti arvutamiseks vajalike iteratsioonide arvu vähendada. Cousot ja Cousot [3] defineerisid selle tarbeks laiendamisoperaatori ∇. Kahendoperaator $\nabla: A \times A \rightarrow A$ on laiendamisoperaator abstraktses domeenis $(A, \sqsubseteq)$, kui:

1. see arvutab mingi ülemtõkke: $\forall x, y \in A: x \sqsubseteq x \nabla y$ ja $y \sqsubseteq x \nabla y$;
2. ja see tagab koondumise: iga jada puhul $(y^i)_{i \in \mathbb{N}}$ hulgas A stabiliseerub $(x^i)_{i \in \mathbb{N}}$, mida arvutatakse valemiga $x^0 := y^0$, $x^{i+1} := x^i \nabla y^{i+1}$, lõpliku ajaga: $\exists k \geq 0: x^{k+1} = x^k$.

Kuna laiendamisoperaator ei arvuta alati vähimat ülemtõket, muudab selle kasutamine programmi analüüsi ebatäpsemaks, kuid analüüsi lõppemise kindlustamiseks piisab laiendamisoperaatori kasutamisest ühes kohas igas programmi tsüklis [8].

Laiendamisoperaatori mõiste lai definitsioon loob eeldused erinevate laiendamisoperaatorite defineerimiseks, et neid abstraktses interpretatsioonis kasutada. Näidisprogrammi puhul võib laiendamisoperaatoriks valida järgmise:

$$[l_1, h_1] \bigtriangledown [l_2, h_2] = [\text{kui } l_1 \leq l_2, \text{ siis } l_1, \text{ muidu } -\infty; \text{ kui } h_1 \geq h_2, \text{ siis } h_1, \text{ muidu } +\infty].$$

Laiendamisel vaadeldakse intervalli ülem- ja alampiiri eraldi. Kui esimese intervalli alampiir l_1 on teise omast l_2 väiksem või sellega võrdne, on l_1 ka laiendamise tulemuseks. Kui l_1 on l_2-st suurem, on tulemuseks negatiivne lõpmatus, et vältida olukorda, kus alampiiri nihutatakse iga järjestikuse iteratsiooniga madalamale, kui intervallide jadas $([l_i, h_i])_{i \in \mathbb{N}}$ alampiir l_i monotoonselt kahaneb. Analoogselt, kuid vastupidiselt toimub laiendamine intervalli ülempiiriga.

Joonisel 3 on kujutatud näidisprogrammi intervallanalüüsi, mille käigus on laiendamist kasutatud.

```
#include <stdio.h>
int main()
{
    int i = 1;
    int a = 0;
    // i ∈ [1, 1], a ∈ [0, 0]

    while(i <= 100)
    {
        // i ∈ [1, ∞], a ∈ [0, ∞]
        a += i;
        i += 1;
        // i ∈ [2, ∞], a ∈ [1, ∞]
    }
    // i ∈ [101, ∞], a ∈ [0, ∞]
    printf(a);

    return 0;
}
```

Joonis 3. Joonisel 1 kujutatud programmi intervallanalüüs koos laiendamisega.

Laiendamine muudab siin hinnangud muutujate i ja a väärtustele tsükli sees ja järel oluliselt ebatäpsemaks, kaotades kummagi puhul ülempiiri. Täpsust on võimalik säilitada keerukamate laiendamisoperaatoritega või kasutades kitsendamist. Kitsendamisoperaator arvutab lõpliku ajaga mingi oma argumentide vahel oleva elemendi ning selle teatud tingimustel kasutamine muudab analüüsi täpsemaks [7]. Edaspidi keskendub töö ainult laiendamisoperaatoritele.

2. Goblint ja selle laiendamisoperaatorid

Goblintit on tutvustanud Vojdani jt [4]. Goblint on programmeerimiskeeles OCaml kirjutatud staatiline analüsaator mitmelõimelistele C-keeles programmidele. Goblint suudab tuvastada võimalikke andmejookse, täisarvude ületäitumist jm. Goblint kasutab staatiliseks analüüsiks abstraktset interpretatsiooni. Goblinti teoreetiline alus tähendab, et selle analüüs on õigesti implementeeritult korrektne: kui analüüsi tulemusena programmis mingit liiki potentsiaalselt ohtlikke olukordi ei tuvastata, siis ei leidu neid programmis tõepoolest. Goblinti analüüs pole aga täielik: see võib anda valepositiivseid tulemusi.

Goblinti arhitektuuri on kirjeldanud Apinis [9]. Abstraktsed domeenid on kaustades *domains* ja *cdomains*. Kaust *cdomains* sisaldab C-le omaste kontseptsioonide abstraktseid domeene. Abstraktsetel domeenidel defineeritakse üleminekufunktsioonid, mis lubavad minna ühest programmi olekust teise. Nende seas on omistamine, tingimuslauses hargnemine jt. Abstraktsetest domeenidest ja üleminekufunktsioonidest moodustuvad analüüsi spetsifikatsioonid ehk kitsenduste süsteemid. Need asuvad kaustas *analyses*. Goblinti arhitektuur on modulaarne, mis tähendab, et ühel programmil saab korraga teha mistahes analüüse. Sisendprogramm teisendatakse juhtvoograafiks, mille servadel on lubatud üleminekufunktsioonid. Lahendajat kasutades lahendatakse kitsenduste süsteem sisendprogrammi suhtes. Lahendajad asuvad kaustas *solvers*. Laiendamist võib vaja minna lahendamisel tsükleid käsitledes.

Info Goblinti laiendamisoperaatorite kohta ja joonised pärinevad Goblinti lähtekoodist (vt lisa I).

2.1. Laiendamisoperaatorite ülevaade

Joonisel 4 on kujutatud Goblinti osalise järjestuse moodulisignatuuri *PO*, millega defineeritakse suurem osa abstraktsest domeenist. Osalise järjestuse elemendid on tüüpi t. Osalisel järjestusel on defineeritud järjestusfunktsioon `leq`, mis kontrollib, kas esimene argument on teisest mitterangelt väiksem. Funktsioonid `join` ja `meet` leiavad vastavalt kahe argumendi vähima ülemtõkke ja suurima alamtõkke. Laiendamisoperaator on defineeritud funktsiooniga `widen`. Funktsioon `narrow` on kitsendamisoperaator.

```
module type PO =
sig
  include Printable.S
  t -> t -> bool
  val leq: t -> t -> bool
  t -> t -> t
  val join: t -> t -> t
  t -> t -> t
  val meet: t -> t -> t
  t -> t -> t
  val widen: t -> t -> t (** [widen x y] assumes [leq x y]. Solvers guarantee this

  t -> t -> t
  val narrow: t -> t -> t

  (** If [leq x y = false], then [pretty_diff () (x, y)] should explain why. *)
  unit -> t * t -> Pretty.doc
  val pretty_diff: unit -> (t * t) -> Pretty.doc
end
```

Joonis 4. Goblinti osalise järjestuse moodul failis *lattice.ml.*

Abstraktne domeen saadakse, muutes osalise järjestuse võreks, mida kirjeldab Goblinti moodulisignatuur *Lattice.S* (joonis 5). Selleks tuleb osalises järjestuses defineerida suurim ja vähim element (vastavalt funktsioonid `top` ja `bot`) ning nendeks olemise kontrollimise funktsioonid (vastavalt `is_top` ja `is_bot`).

```
module type S =
sig
  include PO
  unit -> t
  val bot: unit -> t
  t -> bool
  val is_bot: t -> bool
  unit -> t
  val top: unit -> t
  t -> bool
  val is_top: t -> bool
end
```

Joonis 5. Goblinti võre moodul failis *lattice.ml.*

Goblintis on defineeritud 107 laiendamisoperaatorit. Selles töös liigitatakse need neljaks.

1. Triviaalsed laiendamisoperaatorid: `let widen = join`, `let widen = meet`, `let widen x y = y` ja nende variandid. `Join`-funktsioon ei taga ise

koondumist, kuid laiendamine on üldjuhul nii defineeritud vaid lõpliku kõrgusega domeenides, sest neis ei saa abstraktne väärtus lõputult suureneda.

2. Laiendamisoperaatorid, mis delegeerivad laiendamise teistele, kasutamata unikaalset loogikat. Need võivad olla muud samas domeenis defineeritud laiendamisoperaatorid või domeenides, millest antud domeen tuletatud on, näiteks failis *lattice.ml* defineeritud paaride domeen *ProdConf* tuletatakse loomisel kahest algdomeenist. Samuti delegeerivad laiendamist sellised laiendamisoperaatorid, mille ülesanne on olemasoleva laiendamise kohta silumiseks või muul eesmärgil infot väljastada, näiteks failis *addressDomain.ml*.
3. Mittetriviaalsed laiendamisoperaatorid. Need on näiteks tõeväärtusdomeenis *MayBool* failis *boolDomain.ml* ning intervalldomeenis *FloatIntervalImpl* failis *floatDomain.ml*.
4. `Widen`-nimelised funktsioonid Goblintis, mis on implementeerimata ja annavad käitamisel alati vea. Neid selles töös ei käsitleta.

Tabel 2. Goblintis leiduvate laiendamisoperaatorite liigid ja arvud.

Laiendamisoperaator	Arv Goblintis
Triviaalne	24
Delegeeriv	60
Mittetriviaalne	21
Implementeerimata	2

2.2. Mittetriviaalsed laiendamisoperaatorid

Käesolevas peatükis kirjeldatakse täpsemalt mittetriviaalseid laiendamisoperaatoreid Goblintis.

2.2.1. Tõeväärtused

Abstraktse domeeni lihtsaim praktiline näide on tõeväärtusdomeen *MayBool* (joonis 6). Selle domeeni võre koosneb kahest elemendist: tõesest ja väärast. Vähim element on siin väär ja suurim on tõene. Sellest järeldub, et väär on tõesest väiksem. Kahe tõeväärtuse vähim ülemtõke on nende disjunktsioon - tõeväärtustest võetakse suurem - ja suurim alamtõke

vastavalt konjunktsioon - tõeväärtustest võetakse väiksem. Kuna tõeväärtuste võre on lõpliku kõrgusega, piisab siin laiendamise ja kitsendamise võrdsustamisest vastavalt vähima ülemtõkke ja suurima alamtõkkega, kuigi seda pole ilmutatult tehtud.

```
module MayBool: Lattice.S with type t = bool =
struct
  include Bool
  unit -> t
  let bot () = false
  t -> t
  let is_bot x = x = false
  unit -> t
  let top () = true
  t -> t
  let is_top x = x = true
  t -> t -> t
  let leq x y = x == y || y
  t -> t -> t
  let join = (||)
  t -> t -> t
  let widen = (||)
  t -> t -> t
  let meet = (&&)
  t -> t -> t
  let narrow = (&&)
end
```

Joonis 6. Domeen *MayBool* failis *boolDomain.ml.*

Analoogse struktuuriga on teine Goblinti tõeväärtusdomeen *MustBool*, kus tõene on vähim ja väär suurim element.

2.2.2. Intervallid

Teoreetilises ülevaates kirjeldatud intervallide domeeni elementideks on Goblinti failis *floatDomain.ml* ujukomaarvude intervallid, kunstlik vähim ja suurim element, positiivne ja negatiivne lõpmatus ning NaN, mis tähistab defineerimata tehete, näiteks nulliga jagamise tulemust. Intervallide domeeni laiendamisoperaator (vt joonis 7) on teoreetilises ülevaates kirjeldatu edasiarendus, milles on lisaks defineeritud reeglid suurima elemendi, vähima elemendi, NaNi ja lõpmatuste käsitlemiseks.

```
t -> t -> t
let widen v1 v2 = (**TODO: support 'threshold_widening' option
  match v1, v2 with
  | Top, _ | _, Top -> Top
  | Bot, v | v, Bot -> v
  | Interval (l1, h1), Interval (l2, h2) ->
    (**If we widen and we know that neither interval contains
       | because a widening with +-inf/nan will always result in
    let low = if l1 <= l2 then l1 else Float_t.lower_bound in
    let high = if h1 >= h2 then h1 else Float_t.upper_bound in
    norm @@ Interval (low, high)
  | NaN, NaN -> NaN
  | MinusInfinity, MinusInfinity -> MinusInfinity
  | PlusInfinity, PlusInfinity -> PlusInfinity
  | _ -> Top
```

Joonis 7. Domeeni *FloatIntervalImpl* laiendamisoperaator failis *floatDomain.ml.*

```
ikind -> t -> t -> t
let widen ik x y =
  match x, y with
  | None, z | z, None -> z
  | Some (l0,u0), Some (l1,u1) ->
    let (min_ik, max_ik) = range ik in
    let threshold = get_interval_threshold_widening () in
    let upper_threshold u =
      let ts = if get_interval_threshold_widening_constants () = "comparisons" then
        WideningThresholds.upper_thresholds () else ResettableLazy.force widening_thresholds in
      let u = Ints_t.to_bigint u in
      let t = List.find_opt (fun x -> Z.compare u x <= 0) ts in
      BatOption.map_default Ints_t.of_bigint max_ik t
    in
    let lower_threshold l =
      let ts = if get_interval_threshold_widening_constants () = "comparisons" then
        WideningThresholds.lower_thresholds () else ResettableLazy.force widening_thresholds_desc in
      let l = Ints_t.to_bigint l in
      let t = List.find_opt (fun x -> Z.compare l x >= 0) ts in
      BatOption.map_default Ints_t.of_bigint min_ik t
    in
    let lt = if threshold then lower_threshold l1 else min_ik in
    let l2 = if Ints_t.compare l0 l1 = 0 then l0 else Ints_t.min l1 (Ints_t.max lt min_ik) in
    let ut = if threshold then upper_threshold u1 else max_ik in
    let u2 = if Ints_t.compare u0 u1 = 0 then u0 else Ints_t.max u1 (Ints_t.min ut max_ik) in
    norm ik @@ Some (l2,u2) |> fst
```

Joonis 8. Domeeni *IntervalFunctor* laiendamisoperaator failis *intDomain.ml.*

Joonisel 8 on kujutatud analoogset laiendamisoperaatorit täisarvude domeeni jaoks, kus lisaks on kasutusel künnised. Laiendamiskünnised on lõplik täisarvude hulk T koos positiivse ja negatiivse lõpmatusega, mida kasutades on võimalik defineerida järgmine laiendamisoperaator [10]:

$$[a, b] \nabla [a', b'] = [\text{kui } a' < a, \text{ siis } \max\{l \in T \mid l \leq a'\}, \text{ muidu } a;$$
$$\text{kui } b' > b, \text{ siis } \min\{h \in T \mid h \geq b'\}, \text{ muidu } b].$$

See tähendab, et laiendamisel valitakse võimalusel ülemtõkke piirideks lõplikud arvud. Analoogselt saab laiendamiskünniseid koostada teiste domeenide jaoks. Täisarvude domeeni tarbeks valitud künnised võivad järgida eri täisarvuliste andmetüüpide suurimaid ja vähimaid väärtusi. Näiteks sobivad künnisteks 8-, 16-, 32- ja 64-bitise märgiga täisarvu suurimad väärtused: 127, 32767, 2147483647 ja 9223372036854775807 [11]. Samuti võib analüüs künnised tuletada süntaktiliselt, otsides programmi tekstist arvulisi konstante. Kui konstantide selline otsimine on ebatõhus, võib otsingut piirata programmis sisalduvate võrdlusavaldistega. Goblintis on künniste süntaktiliseks otsinguks mõlemal viisil seade olemas.

Domeenis *IntervalFunctor* on võimalik laiendada künniseid kasutades või mitte. Künniseid toetava laiendamisfunktsiooni esimene argument on täisarvu tüüp, mille suurimat ja vähimat väärtust kasutatakse künniste kindlaksmääramisel.

2.2.3. Hulgad

Osaliselt järjestatud elementidega hulkade domeeni *HoareDomain.Set* (joonis 9) laiendamisoperaator kasutab funktsiooni `product_widen`, mis laiendab kõiki ühe hulga elemente kõigi teise hulga elementidega, kasutades nende elementide tüübile vastavat laiendamisfunktsiooni:

$$X \nabla Y = \{x \nabla y \mid x \in X \wedge y \in Y\}.$$

```
(elt -> elt -> elt) -> t -> t -> t
let product_widen op a b = (* assumes b to be bigger than a *)
  let xs,ys = elements a, elements b in
  List.concat_map (fun x -> List.map (fun y -> op x y) ys) xs |> fun x -> reduce (union b (of_list x))
t -> t -> t
let widen = product_widen (fun x y -> if B.leq x y then B.widen x y else B.bot ())
```

Joonis 9. Domeeni *Set* laiendamisoperaator failis *hoareDomain.ml*.

Bagnara järgi [12] pole siin tegemist definitsioonile vastava laiendamisega, sest see ei garanteeri koondumist. Hulka abstraheeritakse tema maksimaalsete elementide järgi, kuid

maksimaalsete elementide arv hulgas võib laiendamise käigus tõkestamatult kasvada, näiteks võib hulka juurde tekkida üheelemendilisi intervalle, mis üksteises ei sisaldu.

Mooduli *SetEM* laiendamisoperaator (joonis 10) garanteerib koondumise ka siis, kui maksimaalsete elementide arv hulgas tõkestamatult kasvab. Seda tehakse Egli-Milneri osalise järjestuse abil, mille järgi hulk X on hulgast Y väiksem, kui hulk X on tühi või see on hulga Y alamhulk ja iga hulga Y elemendi puhul leidub hulgas X sellest väiksem element [12]. Hulkade X ja Y laiendamisel võrreldakse neid Egli-Milneri järgi. Kui hulk X on hulgast Y Egli-Milneri järgi väiksem, laiendatakse mooduliga *Set* analoogse laiendamisoperaatoriga. Vastasel juhul ühendatakse kõik hulkade X ja Y elemendid üheks üheelemendiliseks hulgaks S (intervallide hulkade puhul on hulgas S hulkade X ja Y kõigi intervallide vähim ülemtõke) ning laiendatakse hulki X ja S.

```
t -> t -> t
let join_em s1 s2 =
  join s1 s2
  |> elements
  |> BatList.reduce E.join
  |> singleton

(** Connector-based widening.
    See Bagnara, Section 6. *)
t -> t -> t
let widen s1 s2 =
  assert (leq s1 s2);
  let s2' =
    if leq_em s1 s2 then
      s2
    else
      join_em s1 s2
  in
  widen s1 s2'
```

Joonis 10. Domeeni *SetEM* laiendamisoperaator ja abifunktsioon `join_em` failis *hoareDomain.ml.*

Failis *disjointDomain.ml* defineeritakse domeene sisemise jaotusega hulkadele. Domeen *PairwiseSets* kirjeldab hulkade ehk ämbrite hulki, mille elemendid võivad olla seotud ehk kongruentsed teises ämbrites olevate elementidega. Joonisel 11 on kujutatud domeeni *PairwiseSets* laiendamisoperaatorit, mis valib iga esimeses hulgas s_1 oleva ämbri b_1 jaoks

ämbri b_2 teisest hulgast s_2, milles leidub mõne ämbri b_1 elemendiga kongruentne element, ja laiendab ämbreid b_1 ja b_2:

$$s_1 \bigtriangledown s_2 = \{b_1 \bigtriangledown b_2 \mid b_1 \in s_1 \wedge b_2 \in s_2 \wedge \exists (e_1 \in b_1, e_2 \in b_2).\ C(e_1, e_2)\}.$$

Kuna ühes hulgas ei leidu mitut omavahel kongruentset ämbrit (kõik omavahel kongruentsed elemendid on samas ämbris), ei saa ämbrite arv laiendamise käigus suureneda.

```
t -> t -> t
let widen s1 s2 =
  Lattice.assert_valid_widen ~leq ~pretty_diff s1 s2;
  let f b2 (s1, acc) =
    let e2 = B.choose b2 in
    let (s1_match, s1_rest) = S.partition (fun e1 -> C.cong (B.choose e1) e2) s1 in
    let b' = match S.choose s1_match with
      | b1 ->
        assert (S.cardinal s1_match = 1);
        B.widen b1 b2
      | exception Not_found -> b2
    in
    (s1_rest, S.add b' acc)
  in
  let (s1', acc) = S.fold f s2 (s1, empty ()) in
  assert (is_empty s1'); (* since [leq s1 s2], folding over s2 should remove all s1 *)
  acc (* TODO: extra union s2 needed? *)
```

Joonis 11. Domeeni *PairwiseSets* laiendamisoperaator failis *disjointDomain.ml*.

Analoogne laiendamisoperaator on defineeritud sõnastike hulkade domeenis *PairwiseMap*.

Intervallide hulkade domeen *IntervalSetFunctor* võimaldab täisarvu võimalikke väärtusi esitada mitme vahemikuga. Intervallide hulkade X ja Y laiendamisel (joonis 12) vastendatakse iga hulga X intervall x ühe hulga Y intervalliga y, mis sisaldab intervalli x, ühendatakse kõik intervalliga y vastendatud intervallid, et moodustuksid vastendpaarid, ühendatakse kõrvutiasetsevad teineteisele lähenevad vastendpaarid ning tehakse saadud vastendpaaridel eelpool kirjeldatud laiendamine. Laiendamisoperaator eeldab, et hulk X on hulgast Y väiksem, s.t iga intervall x sisaldub mingis intervallis y. Goblintis kehtib see eeldus alati.

```
ikind -> t -> t -> (int_t * int_t) list
let widen ik xs ys =
  let (min_ik,max_ik) = range ik in
  let threshold = get_bool "ana.int.interval_threshold_widening" in
  let upper_threshold (_,u) =
    let ts = if GobConfig.get_string "ana.int.interval_threshold_widening_constants" = "comparisons" then
      WideningThresholds.upper_thresholds () else ResettableLazy.force widening_thresholds in
    let u = Ints_t.to_bigint u in
    let t = List.find_opt (fun x -> Z.compare u x <= 0) ts in
    BatOption.map_default Ints_t.of_bigint max_ik t
  in
  let lower_threshold (l,_) =
    let ts = if GobConfig.get_string "ana.int.interval_threshold_widening_constants" = "comparisons" then
      WideningThresholds.lower_thresholds () else ResettableLazy.force widening_thresholds_desc in
    let l = Ints_t.to_bigint l in
    let t = List.find_opt (fun x -> Z.compare l x >= 0) ts in
    BatOption.map_default Ints_t.of_bigint min_ik t
  in
  (*obtain partitioning of xs intervals according to the ys interval that includes them*)
  let rec interval_sets_to_partitions (ik: ikind) (acc : (int_t * int_t) option) (xs: t) (ys: t)=
    match xs,ys with
    | _, [] -> []
    | [], (y::ys) -> (acc,y):: interval_sets_to_partitions ik None [] ys
    | (x::xs), (y::ys) when Interval.leq (Some x) (Some y) ->
      interval_sets_to_partitions ik (Interval.join ik acc (Some x)) xs (y::ys)
    | (x::xs), (y::ys) -> (acc,y) :: interval_sets_to_partitions ik None  (x::xs) ys
  in
  let interval_sets_to_partitions ik xs ys = interval_sets_to_partitions ik None xs ys in
  (*merge a pair of adjacent partitions*)
  let merge_pair ik (a,b) (c,d) =
    let new_a = function
      | None -> Some (upper_threshold b, upper_threshold b)
      | Some (ax,ay) -> Some (ax, upper_threshold b)
    in
    let new_c = function
      | None -> Some (lower_threshold d, lower_threshold d)
      | Some (cx,cy) -> Some (lower_threshold d, cy)
    in
    if threshold && (lower_threshold d +. Ints_t.one) >. (upper_threshold b) then
      [(new_a a,(fst b, upper_threshold b)); (new_c c, (lower_threshold d, snd d))]
    else
      [(Interval.join ik a c, (Interval.join ik (Some b) (Some d) |> Option.get))]
  in
  let partitions_are_approaching part_left part_right = match part_left, part_right with
    | (Some (_, left_x), (_, left_y)), (Some (right_x, _), (right_y, _)) ->
      (right_x -. left_x) >. (right_y -. left_y)
    | _,_ -> false
  in
  (*merge all approaching pairs of adjacent partitions*)
  let rec merge_list ik = function
    | [] -> []
    | x::y::xs  when partitions_are_approaching x y -> merge_list ik ((merge_pair ik x y) @ xs)
    | x::xs -> x :: merge_list ik xs
  in
  (*expands left extremity*)
  let widen_left = function
    | [] -> []
    | (None,(lb,rb))::ts -> let lt = if threshold then lower_threshold (lb,lb) else
      min_ik in (None, (lt,rb))::ts
    | (Some (la,ra), (lb,rb))::ts  when lb <. la ->  let lt = if threshold then lower_threshold (lb,lb) else
      min_ik in (Some (la,ra),(lt,rb))::ts
    | x  -> x
  in
  (*expands right extremity*)
  let widen_right x =
    let map_rightmost = function
      | [] -> []
      | (None,(lb,rb))::ts -> let ut = if threshold then upper_threshold (rb,rb)
      else max_ik in (None, (lb,ut))::ts
      | (Some (la,ra), (lb,rb))::ts  when ra <. rb -> let ut = if threshold then upper_threshold (rb,rb) else
        max_ik in (Some (la,ra),(lb,ut))::ts
      | x  -> x
    in
    List.rev x |> map_rightmost |> List.rev
  in
  interval_sets_to_partitions ik xs ys |> merge_list ik |> widen_left |> widen_right |> List.map snd
```

Joonis 12. Domeeni *IntervalSetFunctor* laiendamisoperaator failis *intDomain.ml*.

Järgneb näide. Kehtigu, et X = {[2, 3], [5, 7], [10, 12]} ja Y = {[2, 7], [9, 17], [20, 24]}. Neile funktsiooni `interval_sets_to_partitions` rakendades tekivad järgmised vastendpaarid:

`interval_sets_to_partitions` ik X Y =
= ([2, 7], [2, 7]), ([10, 12], [9, 17]), (None, [20, 24]).

Intervallid [2, 3] ja [5, 7] ühendatakse vastendamisel intervalliga [2, 7]. Intervallile [20, 24] ei vastendu miski. Saadud vastendpaaridele (olgu nende tähis Z) funktsiooni `merge_list` rakendades ühendatakse paarid ([2, 7], [2, 7]) ja ([10, 12], [9, 17]), sest parempoolsed intervallid on arvteljel teineteisele lähemal kui vasakpoolsed, s.t vastendpaarid lähenevad teineteisele:

`merge_list` ik Z = ([2, 12], [2, 17]), (None, [20, 24]).

See muudab järgneva laiendamise täpsemaks. Laiendatakse kumbagi vastendpaari (olgu nende tähis W). Esmalt laiendatakse intervallide alampiire (`widen_left`), seejärel ülempiire (`widen_right`). Intervallid, millele ükski hulga X intervall ei vastendu (selles näites [20, 24]), laiendatakse mõlemale poole lähima künniseni või selle puudumisel lõpmatusse, et vältida olukorda, kus igal tsükli sammul tekib hulka uusi intervalle:

`widen_right` (`widen_left` W) = ([2, 12], [2, ∞]), (None, [-∞, ∞]).

Laiendamise tulemuseks on hulk, mis sisaldab vastendpaaride teisi intervalle:

`widen` ik X Y = {[2, ∞], [-∞, ∞]}.

Selles näites saadud tulemust peaks normaliseerima. Normaliseeritult on laiendamise tulemuseks hulk {[-∞, ∞]}. Laiendamisoperaator ise tulemust ei normaliseeri.

2.2.4. Ülejäänud

Massiivide domeen *Partitioned* toetab massiivi jagamist kolmeks osaks. Seda kasutatakse massiivi tsüklilisel läbimisel, et eristada juba käsitletud ning praegu ja tulevikus käsitletavaid elemente. Domeeni laiendamisoperaator (joonisel 13) võimaldab jagatud massiivide osade eraldi laiendamist. Jagatud massiivi iga osa laiendatakse terve jagamata massiiviga või jagatud massiivi vastava osaga:

$$(xl, xm, xr) \bigtriangledown y = (xl \bigtriangledown y, xm \bigtriangledown y, xr \bigtriangledown y);$$
$$x \bigtriangledown (yl, ym, yr) = (x \bigtriangledown yl, x \bigtriangledown ym, x \bigtriangledown yr);$$
$$(xl, xm, xr) \bigtriangledown (yl, ym, yr) = (xl \bigtriangledown yl, xm \bigtriangledown ym, xr \bigtriangledown yr).$$

Kahte jagatud massiivi võib osade kaupa laiendada üksnes siis, kui need on jagatud sama avaldise järgi. Näiteks ei saa laiendada tsüklimuutuja järgi indekseeritavat massiivi ja konstandi järgi jagatud massiivi, sest laiendama peab samal indeksil asuvaid elemente, mistõttu on tarvilik erinevalt jagatud massiivide erinevad osad ühendada. Kui kahe massiivi jagamise alused erinevad, tuleb neid laiendada tervikutena.

```
let widen (x:t) (y:t) = normalize @@ match x,y with
  | Joint x, Joint y -> Joint (Val.widen x y)
  | Partitioned (e,(xl, xm, xr)), Joint y -> Partitioned (e,(Val.widen xl y, Val.widen xm y, Val.widen xr y))
  | Joint x, Partitioned (e,(yl, ym, yr)) -> Partitioned (e,(Val.widen x yl, Val.widen x ym, Val.widen x yr))
  | Partitioned (e,(xl, xm, xr)), Partitioned (e',(yl, ym, yr)) ->
    if CilType.Exp.equal e e' then Partitioned (e,(Val.widen xl yl, Val.widen xm ym, Val.widen xr yr))
    else Joint (Val.widen (join_of_all_parts x) (join_of_all_parts y))
```

Joonis 13. Domeeni *Partitioned* laiendamisoperaator failis *arrayDomain.ml*.

Domeenis *Partitioned* on ka teine laiendamisfunktsioon `smart_widen_with_length`, mis kutsub välja funktsiooni `smart_op`. Funktsioon `smart_op` (joonis 14) saab argumendiks tehtava operatsiooni (antud juhul laiendamise), massiivide pikkuse, kaks massiivi ja kaks funktsiooni, mis vastavalt massiivide jagamise aluseid väärtustavad. Kui massiivide jagamise alused ei ole võrdsed, kontrollib funktsioon, kas kummagi massiivi jagamise alus on kindlasti 0 või ühe võrra väiksem massiivi pikkusest. Sel juhul on selle massiivi vastavalt vasak- või parempoolne osa tühi ning selle võib laiendamisel samuti tühjaks jätta. Ülejäänud kaks osa laiendatakse kogu teise massiiviga:

$$([], xm, xr) \bigtriangledown y = ([], xm \bigtriangledown y, xr \bigtriangledown y);$$

$$(xl, xm, []) \bigtriangledown y = (xl \bigtriangledown y, xm \bigtriangledown y, []);$$

$$x \bigtriangledown ([], ym, yr) = ([], x \bigtriangledown ym, x \bigtriangledown yr);$$

$$x \bigtriangledown (yl, ym, []) = (x \bigtriangledown yl, x \bigtriangledown ym, []).$$

```
(value -> value -> value) -> idx option -> t -> t -> (exp -> Z.t option) -> (exp -> Z.t option) -> t
let smart_op (op: Val.t -> Val.t -> Val.t) length x1 x2 x1_eval_int x2_eval_int =
  normalize @@
  let must_be_length_minus_one v = match length with
    | Some l ->
      begin
        match Idx.to_int l with
        | Some i ->
          v = Some (BI.sub i BI.one)
        | None -> false
      end
    | None -> false
  in
  let must_be_zero v = v = Some BI.zero in
  let op_over_all = op (join_of_all_parts x1) (join_of_all_parts x2) in
  match x1, x2 with
  | Partitioned (e1, (xl1, xm1, xr1)), Partitioned (e2, (xl2, xm2, xr2)) when Basetype.CilExp.equal e1 e2 ->
    Partitioned (e1, (op xl1 xl2, op xm1 xm2, op xr1 xr2))
  | Partitioned (e1, (xl1, xm1, xr1)), Partitioned (e2, (xl2, xm2, xr2)) ->
    if get_string "ana.base.partition-arrays.keep-expr" = "last" ||
      get_bool "ana.base.partition-arrays.smart-join" then
      let op = Val.join in (* widen between different components isn't called validly *)
      let over_all_x1 = op (op xl1 xm1) xr1 in
      let over_all_x2 = op (op xl2 xm2) xr2 in
      let e1_in_state_of_x2 = x2_eval_int e1 in
      let e2_in_state_of_x1 = x1_eval_int e2 in
      (* TODO: why does this depend on exp comparison?
         probably to use "simpler" expression according to constructor order in compare *)
      (* It is mostly SOME order to ensure commutativity of join *)
      let e1_is_better = (not (Cil.isConstant e1) && Cil.isConstant e2) ||
      Basetype.CilExp.compare e1 e2 < 0 in
      if e1_is_better then (* first try if the result can be partitioned by e1e *)
        if must_be_zero e1_in_state_of_x2  then
          Partitioned (e1, (xl1, op xm1 over_all_x2, op xr1 over_all_x2))
        else if must_be_length_minus_one e1_in_state_of_x2  then
          Partitioned (e1, (op xl1 over_all_x2, op xm1 over_all_x2, xr1))
        else if must_be_zero e2_in_state_of_x1 then
          Partitioned (e2, (xl2, op over_all_x1 xm2, op over_all_x1 xr2))
        else if must_be_length_minus_one e2_in_state_of_x1 then
          Partitioned (e2, (op over_all_x1 xl2, op over_all_x1 xm2, xr2))
        else
          Joint op_over_all
      else  (* first try if the result can be partitioned by e2e *)
      if must_be_zero e2_in_state_of_x1 then
        Partitioned (e2, (xl2, op over_all_x1 xm2, op over_all_x1 xr2))
      else if must_be_length_minus_one e2_in_state_of_x1 then
        Partitioned (e2, (op over_all_x1 xl2, op over_all_x1 xm2, xr2))
      else if must_be_zero e1_in_state_of_x2 then
        Partitioned (e1, (xl1, op xm1 over_all_x2, op xr1 over_all_x2))
      else if must_be_length_minus_one e1_in_state_of_x2 then
        Partitioned (e1, (op xl1 over_all_x2, op xm1 over_all_x2, xr1))
      else
        Joint op_over_all
    else
      Joint op_over_all
  | Joint _, Joint _ ->
    Joint op_over_all
  | Joint x1, Partitioned (e2, (xl2, xm2, xr2)) ->
    if must_be_zero (x1_eval_int e2) then
      Partitioned (e2, (xl2, op x1 xm2, op x1 xr2))
    else if must_be_length_minus_one (x1_eval_int e2) then
      Partitioned (e2, (op x1 xl2, op x1 xm2, xr2))
    else
      Joint op_over_all
  | Partitioned (e1, (xl1, xm1, xr1)), Joint x2 ->
    if must_be_zero (x2_eval_int e1) then
      Partitioned (e1, (xl1, op xm1 x2, op xr1 x2))
    else if must_be_length_minus_one (x2_eval_int e1) then
      Partitioned (e1, (op xl1 x2, op xm1 x2, xr1))
    else
      Joint op_over_all
```

Joonis 14. Domeeni *Partitioned* laiendamisoperaatori osa `smart_op` failis *arrayDomain.ml*.

Domeen *Compound* hoiab eri tüüpi väärtusi ja defineerib funktsioone nende omavaheliseks käsitlemiseks. Domeeni laiendamisoperaator (joonis 15) toetab lisaks sama tüüpi muutujate laiendamisele ka aadressi ja täisarvu laiendamist. Seejuures tuleb kindlustada, et väiksemat elementi laiendatakse suuremaga, asendades teise argumendi esimese ja teise argumendi vähima ülemtõkkega. Sellest laiendamisoperaatorist on puudu näiteks lõime ja täisarvu ning lõime ja aadressi korrektne käsitlemine: praegu ignoreeritakse vastavalt täisarvu ja aadressi ning tulemuseks on muutmata lõim.

```
t -> t -> t
let rec widen x y =
  match (x,y) with
  | (`Top, _) | (_, `Top) -> `Top
  | (`Bot, x) | (x, `Bot) -> x
  | (`Int x, `Int y) -> (try `Int (ID.widen x y) with
    IntDomain.IncompatibleIKinds m -> Messages.warn ~category:Analyzer "%s" m; `Top)
  | (`Float x, `Float y) -> `Float (FD.widen x y)
  | (`Int x, `Address y)
  | (`Address y, `Int x) -> `Address (match ID.to_int x with
      | Some x when BI.equal x BI.zero -> AD.widen AD.null_ptr (AD.join AD.null_ptr y)
      | Some x -> AD.(widen y (join y not_null))
      | None -> AD.widen y (AD.join y AD.top_ptr))
  | (`Address x, `Address y) -> `Address (AD.widen x y)
  | (`Struct x, `Struct y) -> `Struct (Structs.widen x y)
  | (`Union (f,x), `Union (g,y)) -> `Union (match UnionDomain.Field.widen f g with
      | `Lifted f -> (`Lifted f, widen x y) (* f = g *)
      | x -> (x, `Top))
  | (`Array x, `Array y) -> `Array (CArrays.widen x y)
  | (`Blob x, `Blob y) -> `Blob (Blobs.widen x y)
  | (`Thread x, `Thread y) -> `Thread (Threads.widen x y)
  | (`Int x, `Thread y)
  | (`Thread y, `Int x) -> `Thread y (* TODO: ignores int! *)
  | (`Address x, `Thread y)
  | (`Thread y, `Address x) -> `Thread y (* TODO: ignores address! *)
  | (`Mutex, `Mutex) -> `Mutex
  | (`JmpBuf x, `JmpBuf y) -> `JmpBuf (JmpBufs.widen x y)
  | _ -> warn_type "widen" x y; `Top
```

Joonis 15. Domeeni *Compound* laiendamisoperaator failis *valueDomain.ml.*

Domeenis *Compound* on ka teine laiendamisfunktsioon `smart_widen`, mis laiendab tüüpe *array* (massiiv), *union* (ühend) ja *struct* (struktuur) teisiti. Massiivide laiendamine delegeeritakse vastavale massiividomeenile.

C-keeles on ühend viis eri tüüpi muutujate hoidmiseks samas mälupesas [13]. Ühend sisaldab eri tüüpi välju, millest väärtust saab korraga hoida vaid üks. Moodul *Compound* laiendab

kahte ühendit, kui nende väärtustatud väli on sama tüüpi, ja tagastab muidu suurima elemendi.

Struktuuride laiendamine delegeeritakse vastavale domeenile, funktsioonile `widen_with_fct`. Struktuur on viis muutujate grupeerimiseks [13]. Struktuuri väljad võivad korraga väärtuseid hoida. Struktuuride analüüsiks on Goblintis fail *structDomain.ml*. Lihtne struktuuridomeen *Simple* delegeerib struktuuri väljade laiendamist: välja kahe väärtuse laiendamiseks kasutatakse mingi selle välja andmetüübi domeeni laiendamisoperaatorit.

Kui programmis defineeritakse struktuur, mille kaks täisarvulist välja on võrdsed, ja neid muudetakse tsüklis nii, et need jäävad igal sammul võrdseks, esitab analüüs neid ometi intervallidena ning võrdsuse tingimus ei ilmne. Selle info säilitamiseks tuleb väljade võimalikud väärtused esitada hulkadena. Seda võimaldab domeen *Sets*. Domeeni *Sets* laiendamisoperaator (joonis 16) kasutab funktsiooni `widen_with_fct`, mille sees on defineeritud funktsioon `product_widen`. Funktsioon `product_widen` võtab argumentideks funktsiooni (antud juhul laiendamise) ja kaks struktuurivälju kirjeldavat hulka ning laiendab hulkade elemente paarikaupa vastavalt etteantud funktsioonile.

```
(value -> value -> value) -> t -> t -> t
let widen_with_fct f =
  let product_widen op a b = (* assumes b to be bigger than a *) (* from HS.product_widen *)
    let xs,ys = HS.elements a, HS.elements b in
    List.concat_map (fun x -> List.map (fun y -> op x y) ys) xs |> fun x -> HS.of_list (List.append x ys)
  in
  product_widen (fun x y -> if SS.leq x y then (SS.widen_with_fct f) x y else SS.bot ())

t -> t -> t
let widen = widen_with_fct Val.widen
```

Joonis 16. Domeeni *Sets* laiendamisoperaator failis *structDomain.ml.*

Analoogne laiendamisoperaator on võtmetega hulkadega struktuuridomeenis *KeyedSets*.

Goblinti failis *apronDomain.apron.ml* rakendatakse teeki Apron, mis sisaldab erinevaid arvuliste andmetüüpide abstraktseid domeene [14]. Heterogeenseid keskkondi, mis ei pruugi sisaldada samu muutujaid, kirjeldab moodul *DHetero*. Selle laiendamisoperaator (joonis 17) kontrollib, kas selle argumentide keskkonnad on võrdsed. Kui argumentide keskkonnad ei ole võrdsed, tagastatakse teine argument, sest selle keskkond on suurem ehk kannab rohkem infot. Vastasel juhul delegeeritakse künnisteta laiendamine vastavale Aproni domeenile.

```
Man.mt A.t -> Man.mt A.t -> Man.mt A.t
let widen x y =
  let x_env = A.env x in
  let y_env = A.env y in
  if Environment.equal x_env y_env then (
    if GobConfig.get_bool "ana.apron.threshold_widening" then (
      if Oct.manager_is_oct Man.mgr then (
        let octmgr = Oct.manager_to_oct Man.mgr in
        let ts = ResettableLazy.force widening_thresholds_apron in
        let x_oct = Oct.Abstract1.to_oct x in
        let y_oct = Oct.Abstract1.to_oct y in
        let r = Oct.widening_thresholds octmgr (Abstract1.abstract0 x_oct) (Abstract1.abstract0 y_oct) ts in
        Oct.Abstract1.of_oct {x_oct with abstract0 = r}
      )
      else (
        let exps = ResettableLazy.force WideningThresholds.exps in
        let module Convert =
          SharedFunctions.Convert (V) (Bounds(Man)) (struct let allow_global = true end) (Tracked) in
        (* this implements widening_threshold with Tcons1 instead of Lincons1 *)
        let tcons1s = List.filter_map (fun e ->
            let no_ov = IntDomain.should_ignore_overflow (Cilfacade.get_ikind_exp e) in
            match Convert.tcons1_of_cil_exp y y_env e false no_ov with
            | tcons1 when A.sat_tcons Man.mgr y tcons1 ->
              Some tcons1
            | _
            | exception Convert.Unsupported_CilExp _ ->
              None
          ) exps
        in
        let tcons1_earray: Tcons1.earray = {
          array_env = y_env;
          tcons0_array = tcons1s |> List.enum |> Enum.map Tcons1.get_tcons0 |> Array.of_enum
        }
        in
        let w = A.widening Man.mgr x y in
        A.meet_tcons_array_with Man.mgr w tcons1_earray;
        w
      )
    )
    else
      A.widening Man.mgr x y
  )
  else
    y (* env increased, just use joined value in y, assuming env doesn't increase infinitely *)
```

Joonis 17. Domeeni *DHetero* laiendamisoperaator failis *apronDomain.apron.ml.*

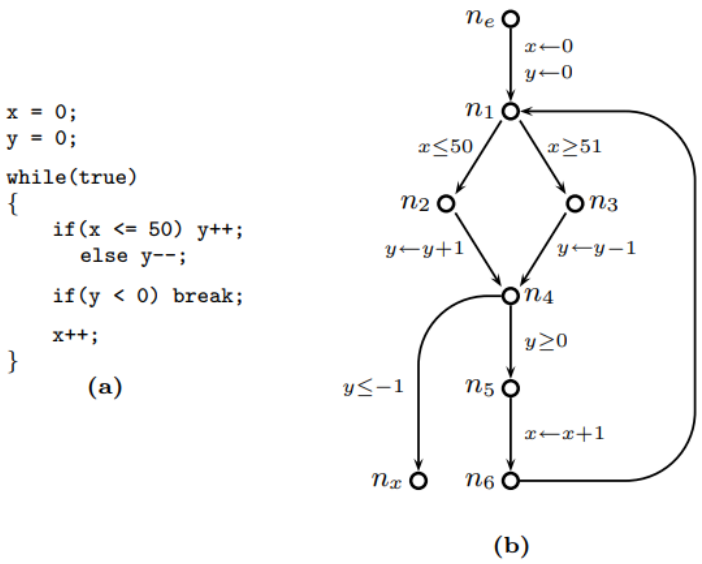
Künnistega laiendamine on kaheksanurkade puhul erinev teistest Aproni domeenidest. Kaheksanurkade domeeni on künnistega laiendamine sisse ehitatud; argumentidel tehakse vajalikud tüübiteisendused ning delegeeritakse künnistega laiendamine Aproni kaheksanurkade domeenile. Teistele Aproni domeenidele tuleb delegeerida tavaline laiendamine ning tagastada selle tulemuse ja laiendamiskünniste suurim alamtõke.

3. Goblinti täiendamine

Kirjandusest valiti kaks laiendamisoperaatorit, mida Goblintis senini ei olnud: ettevaatav laiendamine ja viitega laiendamine. Valik tehti implementeerimise oodatava raskusastme järgi, eelistades jõukohasust. Need realiseeriti Goblintis koos vajalike muudatustega.

3.1. Ettevaatav laiendamine

Gopan ja Reps [15] on välja töötanud ettevaatava laiendamise (ingl *lookahead widening*), mis annab mitmefaasiliste tsüklite analüüsil tavalisest laiendamisest täpsemaid tulemusi. Ettevaataval laiendamisel arvutatakse tsükli iga faasi jaoks eraldi lahendus ning võetakse kõigi lahenduste vähim ülemtõke.



Joonis 18. Mitmefaasilise tsükliga programm (a) ja selle juhtvoograaf (b) [15].

Joonisel 18 on kujutatud mitmefaasilist tsüklit sisaldavat programmi. Tsüklis muudetakse muutujate x ja y väärtuseid. Muutujat x suurendatakse kogu tsükli käitamise ajal. Muutujat y suurendatakse tsükli keha esimestel täitmistel ja seejärel vähendatakse. Muutuja y muutumise suuna järgi võib tsüklis eristada kahte faasi.

Tavalisel laiendamisel ei seata muutujate x ja y võimalikele väärtustele ülempiiri, sest tsükli faase vaadeldakse samaaegselt, mistõttu pole muutujale x tsükli sees võimalik kitsendusi

seada. Ettevaataval laiendamisel analüüsitakse tsükli kumbagi faasi eraldi, seega on võimalik öelda, millises olekus programm faaside vahel on, ja sellest täiendavaid kitsendusi tuletada. Näiteks võib tsükli esimese faasi jaoks leida invariandi, et x ja y on võrdsed. Sellele tuginedes saab analüüs kindlalt väita, et x on teise faasi järel 102.

Ettevaatav laiendamine on võimalik abstraktses domeenis, mille elementideks on paarid ja kunstlik vähim element. Paari esimene element on peaväärtus ehk kõikide seni analüüsitud tsükli faaside lahendus ja teine element pilootväärtus ehk hetkel analüüsitava faasi lahendus. Seejuures peab pilootväärtus olema peaväärtusega võrdne või sellest suurem ning peaväärtus ei või ise olla oma domeeni vähim element. Laiendamine toimub igas tsükli faasis; laiendatakse üksnes pilootväärtust. Faasi lõppemisel omistatakse pilootväärtus peaväärtuse kohale.

3.2. Viitega laiendamine

Miné [5] on kirjeldanud viitega ehk loenduriga laiendamist (ingl *delayed widening*), mille raames toimub laiendamine tsükli mingil täitmiskorral, mis ei ole esimene. See annab täpsemaid tulemusi lühikeste tsüklite puhul ja selliste tsüklite puhul, mille esimesed täitmiskorrad erinevad ülejäänutest.

$V \leftarrow 0;$
while $[0, 1] = 0$ **do**
    if $V = 0$ **then** $V \leftarrow 1$ **endif**; ...
done

Joonis 19. Tsükkel, mis muudab muutujat ainult esimesel korral [5].

Joonisel 19 on kujutatud programmi, milles deklareeritakse muutuja V ja muudetakse seda tsükli sees vaid ühel, esimesel täitmiskorral. Hariliku laiendamise järel kirjeldab muutujat V intervall [0, ∞], mis on väga ebatäpne. Viitega laiendamisel toimub laiendamine alles tsükli teisel või mõnel järgmisel täitmiskorral, mil V ei muutu, ja analüüsi tulemuseks on intervall [0, 1], mis on märkimisväärselt täpsem.

Viitega laiendamiseks on vaja paaride domeeni, mille esimene element kirjeldab analüüsitavat muutujat ja teine element on täisarvuline loendur, mille väärtus on analüüsi alguses igas

programmi juhtvoograafi punktis 0. Samuti peab domeenis olema defineeritud arv n (viide), mis näitab, mitme tsükli täitmiskorra võrra tuleb laiendamist edasi lükata. Laiendamisel kontrollitakse, kas loenduri väärtus on n, ja tehakse harilik laiendamine, kui on, vastasel juhul suurendatakse vaadeldavas juhtvoograafi punktis loendurit ühe võrra ja jätkatakse laiendamata.

3.3. Implementatsioon

Gopani ja Repsi [15] ettevaatav laiendamine realiseeriti faili *lattice.ml* moodulis *LookaheadConf*, kus defineeriti abstraktne domeen. Ettevaatava laiendamise domeen saab argumendiks ühe algdomeeni ja loob selle põhjal ülalkirjeldatud paaride domeeni koos vastava laiendamisoperaatoriga. Implementatsiooni lihtsuse huvides defineeriti domeeni vähim element triviaalselt algdomeeni vähimate elementide paarina, mitte ei lisatud kunstlikku.

Goblinti laiendamisoperaatorid eeldavad üldjuhul, et esimene argument on teisest mitterangelt väiksem. Selle kindlustamiseks asendab analüüside lahendaja laiendamistehted $x \triangledown y$ tehetega $x \triangledown (x \sqcup y)$. Ettevaatav laiendamine seda eeldust ei tee ning nimetatud asendus kahjustab oluliselt analüüsi täpsust. Seetõttu lisati lahendajasse võimalus asendus tegemata jätta ja defineeriti domeenis vähim ülemtõke selliselt, et seda võimalust kasutataks.

Domeeni analüüsis rakendamiseks defineeriti sellel üleminekufunktsioonid faili *constraints.ml* moodulis *Lookahead*. Algdomeeni üleminekufunktsioone rakendatakse ettevaatava laiendamise domeeni paari mõlemal elemendil, v.a juhul, kui peaväärtus on algdomeeni vähim element. Sel juhul on tulemuseks triviaalne vähim element.

Ettevaatava laiendamise implementatsiooni testiti Gopani ja Repsi näidisprogrammi peal, kasutades algdomeenina Aproni hulknurkade domeeni, mis võimaldas analüüsida ka muutujate x ja y vahelist seost. Ettevaatava laiendamisega analüüs oli ettevaatava laiendamiseta analüüsist täpsem: muutujate x ja y võimalike väärtuste vahemikud olid lõplikud.

Viitega laiendamine realiseeriti faili *lattice.ml* moodulis *DelayedConf*, kus defineeriti samuti abstraktne domeen. Viitega laiendamise domeen luuakse sarnaselt teise domeeni põhjal,

moodustades algdomeenile loenduri lisamisega paarid ja defineerides võreoperatsioonid, sh laiendamise vastavalt. Loendurit domeeni elementide võrdlemisel ei arvestata. Viite määramiseks lisati Goblintisse vastav seade.

Viitega laiendamise domeeni üleminekufunktsioonid defineeriti faili *constraints.ml* moodulis *Delayed*. Algdomeeni üleminekufunktsioone rakendatakse viitega laiendamise domeeni paari esimesel elemendil. Loenduri väärtus ei kandu edasi: loendur saab suureneda ainult programmi sama punkti olekute laiendamist kasutaval uuendamisel.

Viitega laiendamise implementatsiooni testiti Miné näidisprogrammi peal, kasutades algdomeenina täisarvude domeeni. Kui hariliku laiendamise tulemusena kirjeldas muutuja V väärtust intervall $[0, 2^{31}-1]$, siis viitega laiendamisega oli analüüsi tulemuseks intervall [0, 1]. Seega tegi viitega laiendamine analüüsi täpsemaks.

Kumbagi uut laiendamismeetodit ei ole veel laiemalt hinnatud, mistõttu ei ole teada, mil määral neist Goblinti rakendamisel kasu on.

Kokkuvõte

Bakalaureusetöö eesmärk oli Goblintis realiseeritud laiendamisoperaatorite võrdlus abstraktse interpretatsiooni kirjandusega ja Goblinti täiendamine uute laiendamisoperaatoritega. Töös anti teoreetiline ülevaade abstraktsest interpretatsioonist ning lähemalt laiendamisest. Seejärel kirjeldati Goblinti tööpõhimõtet ja selles olevaid laiendamisoperaatoreid. Viimaks valiti kirjandusest välja kaks laiendamisoperaatorit, mida Goblintis ei olnud, ja realiseeriti need koos abstraktsete domeenide ja üleminekufunktsioonidega. Realiseeritud laiendamisoperaatorid toimivad ühelõimeliste programmide puhul korrektselt ja võimaldavad varasemast täpsemat analüüsi. Sellegipoolest on vaja neid täiendavalt suurema arvu programmide ja mitmelõimeliste programmide peal testida.

Viidatud kirjandus

[1] Beller, M., Bholanath, R., McIntosh, S. & Zaidman, A. (2016). Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Osaka, Japan, 2016, pp. 470-481. doi:10.1109/SANER.2016.105

[2] Karpov, A. (2023). 5 reasons why static analysis is important for business. https://pvs-studio.com/en/blog/posts/1046/ (15.05.2024)

[3] Cousot, P. & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints). *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. Los Angeles, California, USA, January 1977 (pp. 238–252). New York: ACM Press. doi:10.1145/512950.512973

[4] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V. & Vogler, R. (2016). Static race detection for device drivers: the Goblint approach. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. New York, NY, USA, (pp. 391–402). New York: ACM Press. https://doi.org/10.1145/2970276.2970337

[5] Miné, A. (2017). Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. Foundations and Trends in Programming Languages, 2017, 4 (3-4), pp. 120-372. ff10.1561/2500000034ff. ffhal-01657536

[6] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2): pp. 358–366. doi:10.1090/s0002-9947-1953-0053041-6

[7] Blanchet, B. (2002). Introduction to Abstract Interpretation. https://bblanche.gitlabpages.inria.fr/absint.pdf (10.05.2024)

[8] Seidl, H. Wilhelm, R. & Hack, S. Compiler Design. DOI: 10.1007/978-3-642-17548-0, Springer-Verlag Berlin Heidelberg 2012

[9] Apinis, K. (2014). Frameworks for analyzing multi-threaded C. (Doktoritöö, Technische Universität München, Fakultät für Informatik). Technische Universität München.

[10] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X. (2003). A static analyzer for large safety-critical software. *SIGPLAN Not.,* 38(5), pp. 196–207. https://doi.org/10.1145/780822.781153

[11] cppreference. C language. Arithmetic types (i.a.). https://en.cppreference.com/w/c/language/arithmetic_types (17.04.2024)

[12] Bagnara, R., Hill, P.M. & Zaffanella, E. (2006). Widening operators for powerset domains. *Int J Softw Tools Technol Transfer* **8**, pp. 449–466. https://doi.org/10.1007/s10009-005-0215-8

[13] Isotamm, A. (2009). *Programmeerimine C-keeles.* Tartu: Tartu Ülikooli Kirjastus. http://hdl.handle.net/10062/16819

[14] Jeannet, B. & Miné, A. (2009). APRON: A Library of Numerical Abstract Domains for Static Analysis. In: Bouajjani, A., Maler, O. (eds) Computer Aided Verification. CAV 2009. Lecture Notes in Computer Science, vol 5643. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-02658-4_52

[15] Gopan, D. & Reps, T. (2006). Lookahead Widening. In: Ball, T., Jones, R.B. (eds) Computer Aided Verification. CAV 2006. Lecture Notes in Computer Science, vol 4144. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11817963_41

Lisad

I. Goblinti lähtekood

Goblinti lähtekood on kättesaadav Githubis aadressil https://github.com/goblint/analyzer. Käesoleva töö autori täienduste ülevaade on Githubis aadressil https://github.com/RonaldJudin/analyzer/commit/bde9a06aa0d35db4322e67583b01f7810ece3064.

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Ronald Judin,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose
   "Laiendamisoperaatorid abstraktses interpretaatoris Goblint",

   mille juhendaja on Simmo Saan,

   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 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.

Ronald Judin

15.05.2024