T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Raivo Laanemets

Lõimeanalüüs Goblintis

Magistritöö (30 ECT)

Juhendajad: prof Varmo Vene
Vesal Vojdani, PhD
Kalmer Apinis, MSc

Autor: ..... "....." mai 2011

Juhendaja: ....."....." mai 2011

Lubada kaitsmisele

Professor: ..... "....." mai 2011

TARTU 2011

Sisukord

Sissejuhatus

Järjest enam luuakse keerulisi ja kriitilisi arvutisüsteeme ning seetõttu on oluline muuta programmides potsentsiaalselt esinevate vigade hulk võimalikult väikseks. Üks moodus vigade arvu vähendamiseks ja nende vältimiseks on staatiliste koodianalüsaatorite kasutamine.

Staatiline programmianalüüs (*static analysis*) üritab välja selgitada programmide omadusi programme mitte käivitamata, sest programmi käivitamisel esile tulevad vead võivad olla omased ainult konkreetsele käivitusele ning paljud potsentsiaalselt eksisteerivad vead ei pruugi üldse välja tulla. Üheks selliseks raskesti avastatavaks vigade liigiks on lõimede (*threads*) kasutamise vead.

Andmejooksuks (*data race*) nimetatakse olukorda, kus kahe või enama lõime koos töötamise tulemus ühel koodilõigul erineb sellest, kui nad läbiksid seda lõiku ühe kaupa. Andmejooksu näiteks on globaalse muutuja väärtuse suurendamine ühe võrra, kus tekib järgmine situatsioon: 1. esimene lõim loeb muutuja väärtuse; 2. teine lõim loeb sama väärtuse; 3. esimene lõim suurendab väärtust ja kirjutab selle tagasi; 4. teine lõim suurendab väärtust ja kirjutab selle tagasi. Kui teine lõim oleks oodanud kuni esimene oma operatsioonid lõpuni teeb, siis oleks globaalse muutuja väärtus suurenenud korrektselt kahe võrra pärast mõlema lõime läbimist, ning esimese lõime tööd poleks teine üle kirjutanud.

Andmejooksude vältimiseks piisab, kui sellised koodiosad kaitsta lukkudega (*mutex, lock*), mis lubavad konkreetset koodilõiku täita ainult ühel lõimel korraga.

Goblint (varasema nimega Goblin) on C-keelsete programmide andmejooksude analüüsimiseks loodud raamistik [Voj06]. Selliseid analüsaatoreid nimetatakse ka kraasijateks (*linters*). Goblint baseerub programmi andmevoo analüüsidel (*data flow analysis*). Niisugust tüüpi analüüsid on levinud ka kompilaatorites

koodi optimeerimiseks [Kil73]. Neid analüüse iseloomustab programmi olekute ja omaduste abstrahheerimine ning lähendamine matemaatiliste struktuuride abil, mida nimetatakse võredeks (*lattice*).

Goblint realiseerib andmejookude analüüsi, eeldades et kõik pöördumised programmis globaalsete muutujate poole (kus vähemalt üks lõim kirjutab muutujasse) peavad olema kaitstud lukkudega. See lähenemine on korrektne (*sound*), sest kindlustab andmejooksude puudumise. Antud tingimust saab nõrgendada programmiosade jaoks, mille kohta teame kindlalt, et seal ei saa mitu lõime koos töötada. Reaalsetes programmides puuduvad sageli nendes kohtades lukud ning analüsaator annab valehäire.

Üheks viisiks leida mitteparalleelselt käivitatavaid programmiosi on lõimede loomise ja lõpetamise (*create/join*) struktuuri uurimine. Seda tüüpi analüüsi nimetame siin ja edaspidi lõimeanalüüsiks (*thread analysis*). Selline analüüs on teatud määral komplementaarne MHP (*may-happen-in-parallel*) analüüsidega, mille eesmärgiks on välja selgitada potsentsiaalselt paralleelselt töötavad programmiosad. Miks selline analüüs võiks täpsemaks muuta andmejooksude tuvastamist, on välja toodud allikates [Voj06, Voj10]. Näiteks saab reaalse programmikoodi jagada nn. initsialiseerimisfaasiks ja tööfaasiks. Initsialiseerimise ajal väärtustab põhilõim globaased muutujad ning alles seejärel loob alamlõimi. On selge, et initsialiseerimise faasis võivad lukud puududa, sest töötab ainult pealõim.

Osaliselt on selliste olukordade arvestamine Goblintisse sisse programmeeritud. Vaadeldakse eraldi kahte olukorda: 1. töötab vaid pealõim; 2. töötab mitu lõime. Käesolevas töös üritame seda ideed edasi arendada, arvestades rohkemate olukordadega.

Töö ülesehitus

Esimene peatükk annab ülevaate staatiliselt analüüsist, mis põhineb programmi andmevoo võrranditel ning nende lahendamisel. Toome võrrandite aluseks oleva võrestruktuuride kirjelduse ja näitame, kuidas võrrandeid primitiivse *worklist* algoritmi abil saab lahendada. Praktikas kasutatavad lahendajad (sealhulgas Goblintis) on keerulisemad, annavad suurema vabaduse võrrandite koostamisel, on kiiremad, jne, aga kõik nad toetuvad sarnastele printsiipidele.

Järgmine osa tööst kirjeldab lähemalt Goblintit ja uute analüüside spetsifitseerimist. Tutvustame üldiselt Goblinti ülesehitust ja kasutamist. Valdavalt käsitleme ainult kolmanda peatüki jaoks vajalikku infot.

Sellele järgnevas töö põhiosas, kolmandas peatükis, seletame põhjalikumalt lahti lõimeanalüüsi probleemi. Tutvustame *POSIX* lõimede paketti. Pärast seda anname kaks põhilist juhtumit, kus me saame midagi väita mitteparalleelselt töötavate lõimede kohta, spetsifitseerime vastava andmevooanalüüsi ning üritame anda põhjenduse meie analüüsi korrektsuse kohta.

Töö lõpetame implementatsiooni kirjeldusega.

Peatükk 1

Staatiline programmianalüüs

Staatilise analüüsi eesmärgiks on programmi omaduste välja selgitamine ilma programmi ennast käivitamata. Seda kasutatakse kompilaatorites, silurites (*debuggers*) ja vigade otsijates (kraasijad). Erinevalt dünaamilisest analüüsist (*dynamic analysis*), mis sisuliselt on programmi käivitamine erinevate sisenditega, saab staatilise analüüsiga tõestada programmi omadusi. Dünaamilise analüüsiga (mis ka leiab kasutust) on võimalik tuvastada vea esinemist, aga ei saa garanteerida nende puudumist.

Traditsiooniliselt on staatilist analüüsi kasutatud kompilaatorites optimieerimiseks. Näiteks tuvastatakse programmipunktides, milliseid muutujaid sellest punktist alates enam ei vajata (*liveness analysis*). Niiviisi saab kokku hoida mälupesi.

Programmivigade otsimise analüüsid on tihti keerulisemad kui need, mida kasutatakse kompilaatorites, sest tuleb arvestada terve programmi koodiga, st. funktsioonide omavaheliste kutsetega. Tavapärane optimeerimine kompilaatorites vaatleb tihti funktsioone eraldiseisvana. C-keeles ja teistes sarnastes keeltes, kui funktsiooni kutse võib toimuda läbi viitade, tuleb lisaks meid huvitava omaduse analüüsile teostada ka viitade analüüs (osutianalüüs, *pointer analysis*) [Ray05].

Siin vaadeldav programmianalüüsi meetodid kasutavad nn. monotoonsete funktsioonide raamistikku (*monotone framework*). Need meetodid on välja kujunenud programmide optimiseerimise tehnikast, mida kasutatakse kompilaatorites [Kil73].

Sellist lähenemist iseloomustab universaalsus ning analüüside spetsifitseerimise mugavus. Kõigepealt defineerime sobivad omadused, mis meid programmipunktides huvitavad. Seejärel kirjeldame, millisel viisil instruktsioonid nende täitmisel teisendavad neid omadusi. Meid huvitavaks omaduseks võib olla näiteks muutuja initsialiseerimine (kontrollime, kas avaldises kasutava muutuja väärtus on defineeritud) või täisarvmuutujate väärtusvahemikud (*interval analysis*).

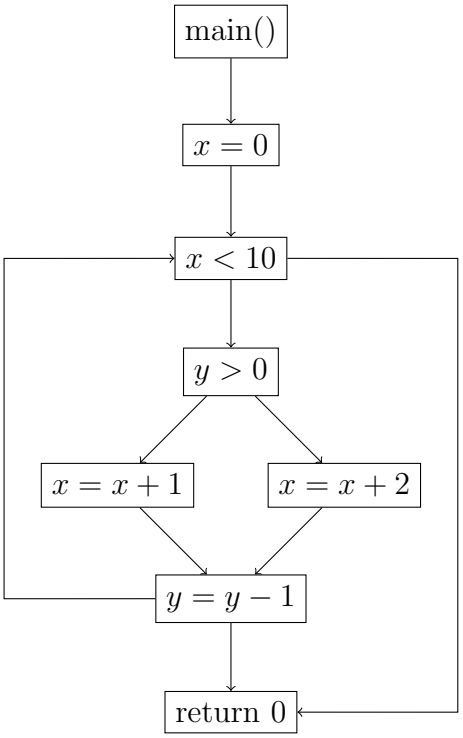
Seda tüüpi analüüsidega on võimalik tuvastada mitmeid erinevaid omadusi. Meid ei huvita konkreetsed muutujate väärtused programmipunktides, sest nende leidmine pole üldisel juhul arvutuslikult võimalik. Selle asemel vaatleme näiteks muutujate absraktseid väärtusi (positiivne, negatiivne, initsialiseeritud, konstantne, jne.). Saame analüüsida ka otseselt muutujatega mitteseotud omadusi, näiteks seda, kas programm on loonud ja lõpetanud lõimi. Lõimede analüüsile keskendub käesoleva kolmas peatükk, mis on ka käesoleva töö põhiosaks.

Enne lõimeanalüüsi juurde asumist vaatleme konkreetsemalt üle programmide andmevoo analüüsides kasutatavad põhimõtted. Alustame juhtvoograafist, jätkame programmiomaduste väljendamise vahenditega, milleks kasutatakse võresid, ning lõpetame algoritmi kirjeldusega, mis arvutab välja kehtivad omadused igas programmipunktis.

1.1 Juhtvoograaf

Programmikoodi hoitakse analüsaatoris juhtimisvoograafina (N, E, e_f, r_f). Mis koosneb tippudest N, suunatud servadest $E \subseteq N \times N$, algtipust $e_f \in N$ ja lõpptipust $r_f \in N$. Servad E vastavad programmis esinevatele instruktsioonidele. Igale programmis defineeritud protseduurile vastab üks selline graaf. Graafi servad on seotud üleminekufunktsioonidega, mis esitavad instruktsiooni täitmisest tulenevat efekti programmi olekule instruktsiooni täitmisel.

```
int main() {
  int y;
  int x = 0;
  while (x < 10) {
    if (y > 0) {
     x = x + 1;
    } else {
     x = x + 2;
    }
    y = y - 1;
  }
  return 0;
}
```



Joonis 1.1: näidisprogramm ja selle juhtvoograaf

Juhtvoograafis esinevad servad iga võimaliku ülemineku jaoks ühest programmipunktist teise. Iga graafi punktiga seome muutuja (andmevoomuutuja), mis väljendab meid huvitavat programmi seisu, kui programmi täitmine on jõudnud sellesse tippu. Programmiolek mingis tipus võib sõltuda tipu eellastest (*forward analysis*) või tipu järglastest (*backward analysis*). Kompilaatorites on tarvis mõlemat tüüpi analüüse, kuid enamus vigade tuvastamise analüüse (nullviitade analüüs, andmetüüpide analüüs, konstantanalüüs jt.) on esimest tüüpi analüüsid.

1.2 Võrestruktuurid

Omadusi juhtvoograafis esitatakse kasutades võrestruktuure. Võreks formaalse definitsiooni esitame allpool, põhjalikumalt võib nende kohta lugeda allikast [Nie99].

Seost $\sqsubseteq: L \times L \to \{\text{true}, \text{false}\}$ hulgal L nimetatakse *osaliseks järjestuseks*, kui $\sqsubseteq$ on refleksiivne ($\forall l \in L : l \sqsubseteq l$), transitiivne ($\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow$

$l_1 \sqsubseteq l_3$) ja antisümmeetriline ($\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$). Hulka L koos osalise järjestusega $\sqsubseteq$ nimetatakse *osaliselt järjestatud hulgaks*.

Lisaks on meil iga selle hulga alamhulga jaoks teatud omadustega elemendid, mida me nimetame vastavalt *ülemtõkkeks* ja *alamtõkkeks*.

Osaliselt järjestatud hulga L alamhulga Y *ülemtõkkeks* nimetatakse elementi $l \in L$, kui iga elemendi $l' \in Y$ korral $l' \sqsubseteq l$ ja *alamtõkkeks*, kui iga $l' \in L$ korral $l \sqsubseteq l'$.

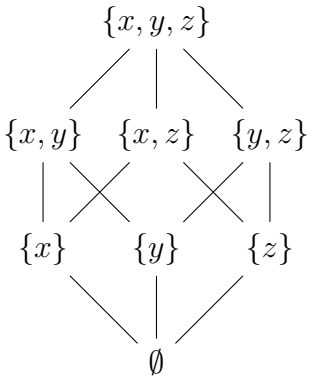
Ning omakorda saab ülemtõkete ja alamtõkete hulgast välja tuua eriliste omadustega elemendid *ülemraja* ja *alamraja*.

Alamhulga Y ülemtõket l nimetatakse hulga Y *ülemrajaks* (vähimaks ülemtõkkeks), kui iga Y ülemtõkke l' korral $l \sqsubseteq l'$ ja tähistatakse $\sqcup Y$ (*join*). Alamhulga Y alamtõket l nimetatakse hulga Y *alamrajaks* (suurimaks alamtõkkeks), kui iga Y alamtõkke l' korral $l' \sqsubseteq l$ ja tähistatakse $\sqcap Y$ (*meet*). Igal alamhulgal ei pruugi ülemraja või alamraja olla, küll on need aga olemas täielikus võres.

1.2.1 Täielik võre

Kui osaliselt järjestatud hulgal L järjestusega $\sqsubseteq$ on iga alamhulgal $Y \subseteq L$ ülemraja ja alamraja, siis nimetame hulka L koos järjestusega $\sqsubseteq$ *täielikuks võreks*. Täielikus võres leiduvad elemendid $\bot = \sqcup\emptyset = \sqcap L$ (*vähim element*) ja $\top = \sqcap\emptyset = \sqcup L$ (*suurim element*).

Lihtsamaid võresid saab esitada *Hasse diagrammide* abil. Seda illustreerib joonis 1.2. Siin on tegemist kolmeelemendilise hulgaga $\{x, y, z\}$ ning seos $\sqsubseteq$ on defineeritud kui alamhulga seos $\subseteq$.



Joonis 1.2: võre näide.

Seose $a \sqsubseteq b$ esitamiseks joonistatakse Hasse digrammil element b ülespoole elemendist a ning ühendatakse joonega. Joonisel 1.2 esitatud võrega sarnane võre leiab näiteks kasutamist initsialiseerimata muutujate analüüsis, kus ta esitab muutujaid, millele on programmis juba väärtus antud.

Programmianalüüsis võib seose $a \sqsubseteq b$ all mõista, et väärtus a on täpsem (või võrdne) kui väärtus b. Samuti saab mõelda, et väärtus b on üldisem kui a, s.t. kui olekut kirjeldab a, siis kirjeldab seda ka b, mida võib mõista ka faktide loogilise implikatsioonina, s.t., $a \sqsubseteq b$ siis ja ainult siis, kui $a \implies b$.

Lihtsamatest võredest saab keerulisemaid ehitada. Olgu $L_1, L_2, \ldots, L_n$ võre. Siis rahuldab võre tingimusi ka võrede korrutis [Sch08]:

$$L_1 \times L_2 \times \ldots \times L_n = \{(x_1, x_2, \ldots, x_n) | x_i \in L_i\},$$

kus $\sqsubseteq$ on defineeritud punktiviisiliselt. Sarnaselt defineeritakse võrede summa:

$$L_1 + L_2 + \ldots + L_n = \{(i, x_i) | x_i \in L_i \setminus \{\bot, \top\}\} \cup \{\bot, \top\},$$

kus $(i, x) \sqsubseteq (j, y) \Leftrightarrow i = j \wedge x \sqsubseteq y$.

Lisaks, kui me võtame hulga A, siis saame defineerida võre $L = \text{flat}(A) = A \cup \{\bot, \top\}$, kus järjestus $\sqsubseteq$ on defineeritud iga $a \in A$ jaoks kui $\bot \sqsubseteq a \wedge a \sqsubseteq \top$.

Hulga A ja võre L abil saab defineerida ka kujutisvõre (*map lattice*):

$$A \mapsto L = \{[a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots, a_n \mapsto x_n] | x_i \in L\},$$

kus $\sqsubseteq$ defineerime komponenthaaval. Kujutisvõre kasutame töös kolmandas peatükis (abstraktsete) lõimede sidumiseks nende (abstraktse) olekuga.

1.3 Üleminekufunktsioonid

Selleks et kirjeldada programmi instruktsioonide mõju programmi olekule, kasutame üleminekufunktsioone. Esitame meid huvitavad oleku väärtusi võrega $\mathbb{D}$ (seda nimetatakse ka analüüsi domeeniks). Seejärel seome programmipunktist väljuvad kaared funktsioonidega kujul $f : \mathbb{D} \to \mathbb{D}$. Selline funktsioon väljendab instruktsiooni (näiteks omistamine) tulemust sõltuvalt programmipunkti sisendist.

Kui programmipunkti siseneb rohkem kui üks andmevoog (kaks või enam sisenevat serva), siis saame nende tulemused ühendada operaatoriga $\sqcup$. Operaatori rakendamisel arvutatakse välja kõigi sisenevate servade andmevoogude ülemraja. Sisuliselt arvutatakse välja kõige üldisem olek, mis kirjeldab kõiki punkti sisenevaid andmevooge.

1.4 Andmevoovõrrandid

Olgu meil programmipunktidele vastavad muutujad $x_1, x_2, \ldots, x_n$ ja neile vastavate üleminekufunktsioonide komplekt

$$F_1(x_1, x_2, \ldots, x_n), F_2(x_1, x_2, \ldots, x_n), \ldots, F_n(x_1, x_2, \ldots, x_n),$$

siis saame võrratuste süsteemi:

$$\begin{array}{ll} x_1 & \sqsupseteq F_1(x_1, x_2, \ldots, x_n) \\ x_2 & \sqsupseteq F_2(x_1, x_2, \ldots, x_n) \\ \vdots & \qquad\qquad \vdots \\ x_n & \sqsupseteq F_n(x_1, x_2, \ldots, x_n) \end{array}$$

Selline süsteem on ekvivalentne võrrandisüsteemiga:

$$\begin{aligned} x_1 &= x_1 \sqcup F_1(x_1, x_2, \ldots, x_n) \\ x_2 &= x_2 \sqcup F_2(x_1, x_2, \ldots, x_n) \\ \vdots & \qquad\qquad \vdots \\ x_n &= x_n \sqcup F_n(x_1, x_2, \ldots, x_n) \end{aligned}$$

Kahe viimase süsteemi ekvivalentsus järeldub $x \sqsupseteq y$ ja $x = x \sqcup y$ ekvivalentsusest, mis omakorda tuleneb sellest et hulga $\{x, y\}$ ülemiseks rajaks saab $x \sqsupseteq y$ korral olla vaid x. Viimane süsteemi esitusviis võetakse mõnikord aluseks ka kitsenduste lahendusalgoritmides, sest ta sobib ka kasutamiseks olukordades, kus ühe muutuja kohta võib olla rohkem kui üks kitsendus.

Võrrandite paremas pooles esinevatele funktsioonidele $F_1, F_2, \ldots, F_n$ erilisi tingimusi me ei sea peale monotoonsuse. Funktsiooni f monotoonsus tähendab järgmise seose kehtimist:

$$\forall x, y \in L : x \sqsupseteq y \Rightarrow f(x) \sqsupseteq f(y).$$

Sisuliselt tähendab see tingimus, et üleminekufunktsioon ei tohi ebatäpsema oleku x pealt arvutada täpsemat olekut kui täpsema oleku y pealt. Kõik konstantsed funktsioonid on monotoonsed, samuti on seda operaator $\sqcup$ ja monotoonsete funktsioonide kompositsioon [Sch08, Nie99, Nie07].

1.5 Vähim püsipunkt

Andmevõrrandite lahendamisel huvitab meid võimalikult täpne lahendus. Eespool vaadeldud süsteemi üheks lahendiks on näiteks $x_1 = \top, x_2 = \top, \ldots, x_n = \top$. Selline lahend on maksimaalselt ebatäpne ja seega analüüsi seisukohast kasutu.

Üldiselt oleme huvitatud võimalikult täpsest lahendist, mis rahuldab kõiki kitsendusi. Sellist lahendit nimetatakse vähimaks püsipunktiks (*least fixed point*). Funktsiooni püsipunktiks nimetame iga väärtus x, mis rahuldab $x = f(x)$. Monotoonse funktsiooni f vähimat püsipunkti $\mathrm{fix}(f)$ saab arvutada n.n. Kleeni iteratsiooni teel:

$$\mathrm{fix}(f) = \bigsqcup_{i \geq 0} f^i(\bot).$$

Arvestades eelnevat, tuleb funktsioonina f siin mõista üleminekufunktsioonide $F_1, F_2, \ldots, F_n$ kollektsiooni $\vec{F}$ ning väärtusena vektorit $\vec{x} = (x_1, x_2, \ldots, x_n)$ (korrutisvõre). Üheks püsipunktiks on ka eespool esitatud $x_1 = \top, x_2 = \top, \ldots, x_n = \top$, aga ta ei pruugi olla sellistest väärtustest järjestuse $\sqsupseteq$ suhtes vähim.

Vähim püsipunkt annab meile lahendi, mis rahuldab kõiki kitsendusi ja samas kindlustab see programmi kohta võimalikult täpse info. Sellist lahendit saab leida iteratiivsete meetoditega. Enamus neist baseerub järgnevalt kirjeldatud *worklist* algoritmil.

1.5.1 WORKLIST algorithm

Kitsenduste lahendamiseks on küllaltki efektiivne WORKLIST algoritm [Sch08]. Algoritmi kasutab ära asjaolu, et uuesti tuleb arvutada ainult need kitsendused, mille paremas pooles esinevate muutujate väärtused on muudetud.

Esitagu sellist kitsendustevahelist sõltuvust funktsioon dep : $C \to 2^C$. Funktsiooni dep kasutatakse mingi kitsenduse uuesti arvutamisel nende kitsenduste leidmiseks, mis potentsiaalselt sõltuvad sellest kitendusest ning tuleks uuesti arvutada.

```
1:  x_1 = ⊥, ..., x_n = ⊥
2:  q ← [c_1, c_2, ..., c_n]
3:  while q ≠ [] do
4:      c_i ← q
5:      y = F^(c_i)(x_1, ..., x_n)
6:      if y ≠ x_i then
7:          for c ∈ dep(c_i) do
8:              q ← c
9:          end for
10:         x_i = y
11:     end if
12: end while
```

Joonis 1.3: WORKLIST algoritm

Algoritmis antakse kõigepealt algväärtus $\bot$ muutujatele $x_1, x_2, \ldots, x_n$ ning listatakse kõik kitsendused $c_1, c_2, \ldots, c_n$ tööjärjekorda q. Seejärel hakatakse

iteratiivselt arvutama kitsendusi, võttes kitsenduse c järjekorrast q ja arvutades sellele vastava üleminekufunktsiooni $F^{(c_i)}$. Kui tulemus erines varasemast väärtusest x_i, uuendatakse x_i väärtust ja lisatakse tööjärjekorda kõik kitsendused, mis sõltuvad sellest muutujast. Algoritm lõpetab töö, kui tööjärjekord on tühi.

Siin esitatud algoritm on küllaltki primitiivne ning seda saab suurel määral täiendada. Näiteks võib kiirema koonduvuse huvides välja eraldada sõltuvuste poolt indutseeritud graafis seotud tipud ning nende jaoks tulemused varem arvutada, kui teiste tippude jaoks. Samuti pole mõnedel juhtudel kasutatava lõpptulemuse saamiseks tarvilik kõikide kitsenduste lahendamine. Sellest ja muudest täiendustest annab ülevaate artikkel [Fec99]. Seal kirjeldatud algoritm on aluseks võetud ka analüsaatoris Goblint.

Kui programmianalüüsi vaadelda kooskõlas programmi formaalse semantikaga, siis nimetatakse seda abstraktseks interpretatsiooniks [Cou77]. Abstraktne interpretatsioon on programmi omaduste korrektne tõestamine. Eespool kirjeldatud andmevoo analüüsi meetodid kuuluvad ka abstraktse interpretatsiooni alla, kui neid vaadelda näiteks programmi denotatsioonisemantika suhtes.

1.6 Kontekstitundlikkus

Terve programmi analüüsimisel seotakse protseduuridele vastavad voograafid ühiseks protseduuridevaheliseks graafiks kokku. Seda tuleb teha osaliselt andmevoovõrrandite lahendamise ajal, kui funktsioone saab kutsuda läbi viitade. Ilma viitade väärtust teadmata ei saa kindlaks teha, milliseid funktsioone sellises kohas välja kutsutakse.

Teine probleem on olekute segunemine funktsiooni sisenedes. Olekud sama funktsiooni erinevatest väljakutsetest segunevad kutsutava funktsiooni algustipus. Selle vältimiseks luuakse igale sisenemisele oma andmevoomuutuja sõltuvalt andmevoo väärtusest millega sisenetakse. Baasanalüüsi korral eraldatakse andmevoo väärtusest, mis jälgib muutujate väärtusi need, mis ei mõjuta funktsiooni tulemust, näiteks pole kättesaadav funktsiooni argumentide kaudu [Api09, Voj06].

Peatükk 2

Goblint

Selles peatükis vaatame analüsaator Goblinti kirjeldust. Siin esitatud kirjeldus baseerub allikatel [Voj06, Api09, Api07, Voj10]. Esialgu oli mõeldud Goblint ainult andmejooksude analüüsiks, kuid praeguseks on tegemist küllaltki universaalse vahendiga C-keelsete programmide jaoks.

Goblinti baseerus oma algusajal Trieri andmejooksude analüsaatoril, mis kasutas unikaalset lähenemist andmejooksudele nn. globaalsete invariantide kaudu [Sei03]. Trieri süsteemi oli paraku liiga ebamugav arendada universaalsemaks. Goblinti raamistiku töötas välja V.Vojdani oma magistritöös [Voj06].

Seejärel lisati analüsaatorisse K.Apinise poolt abstraktsed massiividomeenid, mis muutsid massiive sisaldava programmi analüüsi täpsemaks [Api07]. Hiljem lisas K.Apinis Goblintisse rohkem analüüse (initsialiseerimata muutujad, nullviidad) ning muutis raamistiku universaalsemaks (vähem spetsifitseeritud andmevooanalüüsidele), samuti muutus lihtsamaks uute analüüside loomine [Api09].

Oma doktoritöös lisas V.Vojdani raamistikku paremad võimalused dünaamiliste struktuuride (lingitud listid) käsitlemiseks. Sellised programmi poolt töö ajal loodavad struktuurid, mis võivad sisaldada ka lukke, on andmejooksude analüüsi seisukohast keerulised [Voj10].

2.1 Goblinti struktuur

Analüsaator koosneb kolmest põhikomponendist: kasutajaliides, C-keele parser CIL (*C Intermediate Language*) ja analüsaator ise. Kasutajaliideseks on arenduskeskkond Eclipse koos CDT (*C/C++ Development Tooling*) ja Goblinti täiendustega. Goblint ise on koostatud programmeerimiskeeles Ocaml. Analüsaatorit saab käivitada ka käsurealt, mis muudab võimalikuks süsteemi integreerimise teiste süsteemidega, näiteks programmi ehitusskriptidega (*GNU Make*).

Kasutajaliides seob analüüsikomponendi poolt väljastatud hoiatused ning vead koodi redaktorisse, tuues esile need read, kus probleemid ilmnevad. Kasutajaliidesest võib lugeda lähemalt töödest [Api07, Api09].

Enne juhtimisvoo graafide moodustamist lihtsustatakse programmi lähekood. Lihtsustamise ajal teisendatakse protseduuride kehad kujule, kus igas protseduuris on ülimalt üks tagastuskäsk. Samuti muudetakse avaldised kõrvalefektidest (näiteks tuuakse omistamised eraldi välja — C-keel võib neid avaldistes sisaldada) vabaks. Selline programmitöötlus aitab kergendada hilisemat tööd.

Goblint eristab juhtvoo servadena muutujale väärtuse omistamist, hargnemist, sisenemist funktsiooni kehasse, väljumist funktsiooni kehast, ja funktsiooni väljakutset, funktsiooni kutset ja juhtimisvoo tagastuse funktsiooni kutsest. Konkreetsest analüüsist olenevalt ei pruugi kõik need funktsioonid tähtsust omada ja mõned neist võivad olla identsusfunktsioonid, kus üleminek analüüsi tulemusele mõju ei avalda.

Eraldi saab käsitleda funktsioone, mille kirjeldust programm ei sisalda, näiteks teegifunktsioonid. Sellised funktsioone identifitseeritakse nime järgi. Töös kasutame seda ka lõimede loomise ja lõpetamisega seotud funktsioonide *pthread_create* ja *pthread_join* käsitlemiseks.

Juhtimisvoo graaf ei ole staatiline, sest C-keel võimaldab funktsioone välja kutsuda viitade kaudu. Sellised väljakutsutavad funktsioonid leitakse viidaanalüüsiga (*pointer analysis*). Goblintis kasutatav andmevoo võrrandite lahendamise algoritm võimaldab lahendamise ajal uusi võrrandeid lisada, see muudab võimalikuks täielikuma juhtvoograafi koostamise vastavalt viitade analüüsi (osalise) lahendi leidmise ajal.

2.2 Baasanalüüs

Goblintis toimub viitade analüüs baasanalüüsi osana. Baasanalüüs lahendab lokaalsete muutujate olekut, seades igale muutujale vastavusse tema hetkeoleku võre elemendi, mis koosneb väärtuse tüübist ja tüübile vastava spetsiaalse võre elemendist.

Baasanalüüs sooritatakse samaaegselt põhianalüüsiga vastavalt sellele, millist baasanalüüsi poolt võimalikku pakutavat informatsiooni on põhianalüüsis tarvis.

Samuti on võimalik keerulisem analüüside kombineerimine. Mitu analüüsi saab kokku panna viisil, kus üks täiendab teist, arvestades automaatselt analüüside omavahelisi sõltuvusi.

2.3 Uute analüüside loomine

Uut analüüsi luues tuleb kõigepealt otsustada võre andmestruktuuri üle, mis peab olema sobiv analüüsitava omaduse väljendamiseks. Võresid esitavate andmestruktuuride koostamise jaoks pakub Goblint mitmesuguseid abistruktuure. Soovitud domeeni saab üles ehitada kombineerides primitiivseid struktuure. Vajadusel saab ka täiesti uue võrestruktuuri luua, defineerides sellele sobiva esituse Ocaml vahenditega ning implementeerides osalise järjestuse predikaadi või ülemise raja võtmise operatsiooni.

Pärast domeeni koostamist lisatakse üleminekufunktsioonide definitsioonid juhtvoo servadele ning fikseeritakse analüüsi domeeni algväärtus, võttes selleks näiteks vähima elemendi. Uute analüüside loomist kirjeldab kõige paremini allikas [Api09], mis sisaldab lisaks põhjalikke näiteid Goblintis juba realiseeritud analüüside üleminekufunktsioonidest.

Goblinti arendustöö tõttu võivad üleminekufunktsioonide tüübid ja argumendid muutuda, seetõttu on kõige täpsema info saamiseks tarvis lugeda süsteemi lähtekoodi, mille saab laadida alla Goblinti koduleheküljelt [Gob11]. Lähtekoodi jagatakse BSD litsentsi alusel, mis annab vabaduse analüsaatori kasutamiseks ja täiendamiseks.

Peatükk 3

Mitteparalleelsed lõimed

Järgnev osa tööst keskendub lõimeanalüüsi spetsifitseerimisele. Kõigepealt kirjeldame, millisel viisil lõimede loomine/lõpetamine toimub ja anname täpsema probleemi kirjelduse. Seejärel vaatame kahte üldjuhtu, milles programmipunktid ei saa paralleelselt töötada. Neist esimene kerkib esile lõimede loomisel ning teine lõimede lõpetamisel. Edasises osas spetsifitseerime analüüsi, mis üritab leida programmipunktid ja lõimed, kus üks nendest juhtudest kehtib. Peatüki lõpetame implementatsiooni kirjeldusega.

3.1 Pthread lõimede pakett

Pthread (*POSIX threads*) on POSIX (*Portable Operating System Interface for Unix*) standardisse kuuluv API (*application programming interface*) lõimedega manipuleerimiseks. See sisaldab suhteliselt madala taseme vahendeid lõimede loomiseks, lõpetamiseks ja sünkroniseerimiseks.

Käesoleva töö piires huvitavad meid otseselt ainult Pthread paketti kuuluvad funktsioonid `pthread_create` ja `pthread_join`. Nendest esimest kasutatakse lõime loomiseks ja teist lõime lõpetamiseks. Joonistel 3.1 ja 3.2 on vastavalt esitatud mõlema funktsiooni signatuur.

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg)
```

Joonis 3.1: funktsiooni `pthread_create` signatuur

```
int pthread_join(pthread_t thread,
                 void **value_ptr)
```

Joonis 3.2: funktsiooni `pthread_join` signatuur

Siin tuleb tähele panna, et mõlemad funktsioonid kasutavad andmestruktuuri `pthread_t`. Selle andmestruktuuri täpne kirjeldus pole oluline. Vajalik on ainult teada, et selle kaudu identifitseeritakse loodud lõimi. Teine oluline komponent on funktsiooniviit `start_routine`, mis määrab loodud lõime jaoks peafunktsiooni, mida täitma hakata. Programmianalüüsis võib juhtuda, et viit `start_routine` viitab potsentsiaalselt mitmele erinevale funktsioonile. Sellega tuleb arvestada ning analüüsis kuidagi ühendada andmed nendest funktsioonidest, eeldades et tegeliku käivitamise ajal võib esineda viidas ükskõik milline nendest funktsioonidest.

Teine ebamugavus *pthread* lõimede loomise/lõpetamise juures on asjaolu, et lõime lõpetava funktsiooni kutse võib esineda programmi mistahes osas. Samuti ei pea lõime lõpetama see lõim, kes ta lõi. Sellised võimalused annavad rohkem vabadust lõimedega manipuleerimisel, kui strukuurne lähenemine (*structured concurrency, fork/join model*), aga muudab analüüsi keerukamaks. Samas võib ka eeldada, et vigade esinemise tõenäosus programmis on suurem, näiteks jäetakse osa lõimi lõpetamata, lõpetatakse vale lõim, kutsutakse *pthread_join* sama lõime peal mitu korda jms.

3.2 Probleemi kirjeldus

Oletame, et meil on korrektne andmejooksude analüüs, mis annab vääraid tulemusi programmiosade kohta, mis tegelikult koos töötada ei saa. Sellisteks

```
1:  int myglobal;
2:
3:  void *t_fun(void *arg) {
4:    myglobal=42;
5:  }
6:
7:  int main(void) {
8:    pthread_t t1, t2;
9:    myglobal = 1;
10:   pthread_create(&t1, NULL, t_fun, NULL);
11:   pthread_join (t1, NULL);
12:   myglobal = 2;
13:   pthread_create(&t2, NULL, t_fun, NULL);
14:   pthread_join (t2, NULL);
15:   myglobal = 3;
16:   return 0;
17: }
```

Näide 3.1: andmejooksuvaba programm

programmiosadeks on töö alustades initsialiseerimine, kui lõimi pole veel loodud, samuti töö lõpus, kus varem loodud lõimed ära lõpetatakse, ning seejärel vabastatakse põhilõime poolt ressursid. Siin annab andmejooksude analüüs vea, kuigi teised lõimed tegelikult ei tööta ning andmejooks pole võimalik.

3.2.1 Näide

Alljärgnev programm illustreerib hästi tekkivat probleemi. Selle koodi on siin lihtsustatud ruumi säästmise eesmärgil. Täieliku koodi koos päisefailide jm. juurdekuuluva osaga leiab failist *tests/10-synch/05-two_threads_nr.c* [1]. Siin on välja toodud ainult probleemiga seotud osad.

Ridadel 3, 8, 11 ja 14 toimuvad globaalse muutuja `myglobal` kasutamised loeb analüsaator andmejooksudeks, sest nad pole kaitstud ühise lukuga. Andmejookse tegelikult ei esine, sest põhilõime poolt loodud lõimed `t1` ja `t2` ei jookse kunagi koos. Samuti ei kasuta põhilõim muutujat `myglobal` samal ajal kui `t1` või `t2` jookseb.

Lõime eristamiseks analüüsis kasutatakse lõime loomisel identifikaatoriks oleva

[1] Siin ja edaspidi on failide suhtelised teed Goblinti distributsiooni suhtes.

muutuja aadressi. See on mugav esituviis, kuigi siin tekib probleem, kui sama muutujat kasutatakse mitme lõime loomiseks. Kõigepealt vaatame, millised meid huvitavad olukorrad võivad programmis esineda.

3.3 Mitteparalleelsuse üldjuhud

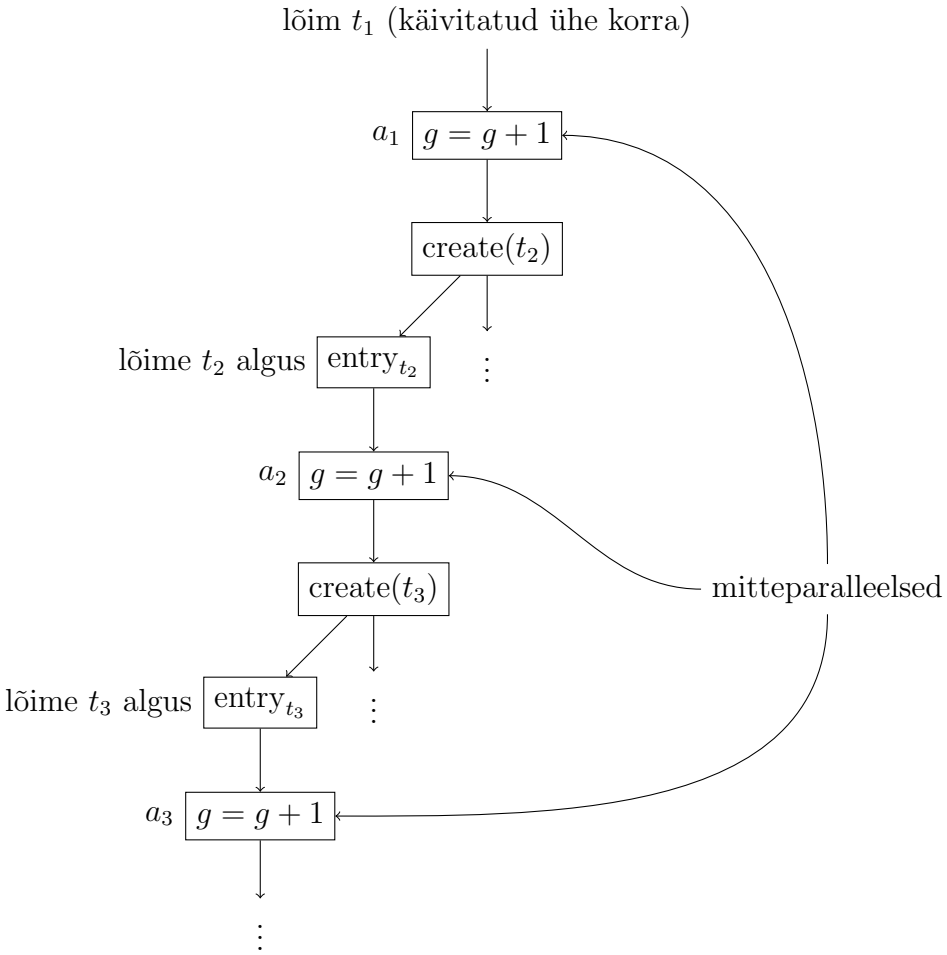
Joonisel 3.3 on näidatud esimene olukord (juhtum A), kus esinevad üksteise paralleelset käivitamist välistavad programmiosad. Eeldusel, et lõim t_1 käivitatakse programmis vaid ühe korra, ei saa programmiosad a_1, a_2 ja a_3 paariti paralleelselt töötada, sest a_1 käivitamise ajal ei tööta veel lõimed t_2 ja t_3 (a_2 ja a_3 välistatud), ning a_2 käivitamise ajal ei tööta veel lõim t_3 (a_3 välistatud). Kui t_1 on loodud rohkem kui üks kord, siis võivad tekkida paralleelsed jooksud mitme t_1 instantsi vahel ning punktide a_1, a_2 ja a_3 üksteist välistavuse kohta ei saa midagi väita.

Juhtum A kirjeldab näiteks olukorda, kus pealõim algväärtustab erinevate lõimede andmed ja siis käivitab neid. See juhtub ka Linuxi kernel koodi analüüsil ja Goblint ei saa hetkel sellega hakkama.

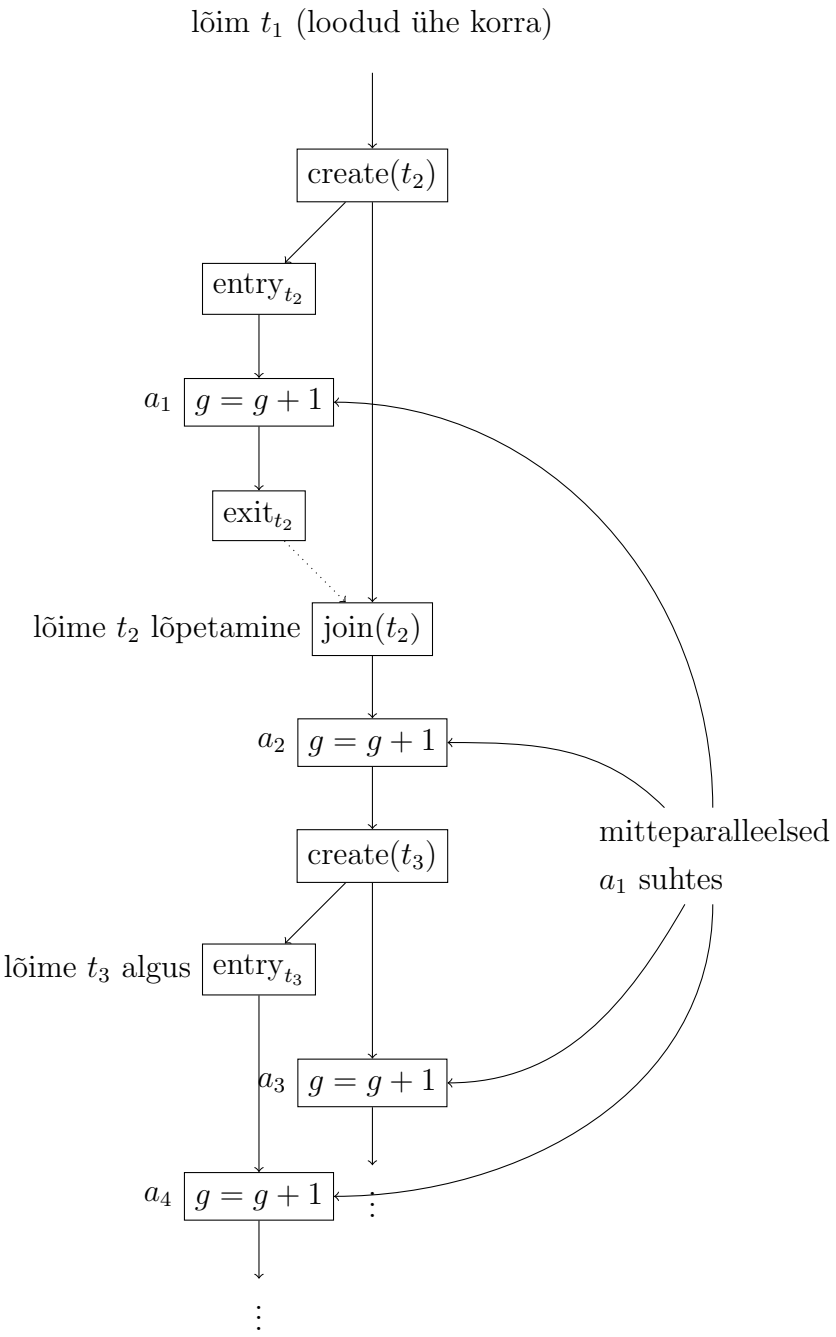
Järgmisel joonisel, 3.4, on teine olukord (juhtum B), kus saab midagi väita mitteparalleelsuse kohta. Eeldusel, et lõim t_1 käivitatakse programmis vaid ühe korra, ei saa ükski programmiosadest a_2, a_3 ja a_4 joosta paralleelselt a_1-ga, sest lõim t_2 on nende punktide käivitamise ajal lõpetatud. Punktide a_2, a_3 ja a_4 paariviisi välistuse kohta ei väida me hetkel midagi. Taas kord, kui t_1 on loodud mitu korda, siis ei saa üldse midagi väita, sest paralleelsed jooksud tekivad mitme t_1 instantsi tõttu.

Lisaks loomise/lõpetamise struktuurile, välistavad paralleelsed jooksud ka lukud, mida me selles analüüsis ei vaatle.

Arvestades kahte ülemist juhtumit, peavad analüüsis kasutatavad andmestruktuurid ja algoritmid võimaldama väljendada lõimede loomise ajalugu ning hierarhiat ning samuti asjaolu, et mingit lõime on loodud ühe või mitu korda.



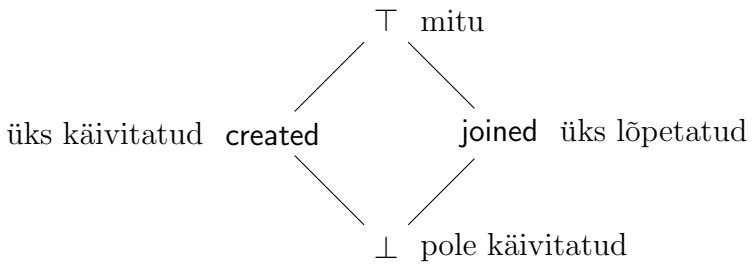
Joonis 3.3: mitteparalleelsete juhtum A.



Joonis 3.4: mitteparalleelsete juhtum B.

3.4 Lõimeolekute võre

Lõime olekut kujutav võre on esitatud joonisel 3.5. Võre elemendid on hulgast $\{\bot, \mathsf{joined}, \mathsf{created}, \top\}$, mis tähistavad erinevaid lõime olekuid.



Joonis 3.5: lõimeanalüüsis kasutatav võre.

Pole käivitatud ($\bot$) - lõimemuutujaga pole ühtegi seotud lõime käivitatud. See on analüüsis ühtlasi ka algolekuks.

Üks lõpetatud (joined) - lõimemuutujaga seotud lõime töö on lõpetatud (lõim on varem loodud ja seejärel on rakendatud funktsiooni *pthread_join* ühe korra).

Üks loodud (created) - lõimemuutujaga on seotud loodud lõim (on rakendatud funktsiooni *pthread_create* ühe korra).

Mitu ($\top$) - lõimemuutujaga on seotud mitu töötavat või lõpetatud lõime. See on kõige ebatäpsem olek, millest ei saa midagi mõistlikku järeldada töötavate lõimede kohta.

3.5 Analüüsi domeen

Siin ja edaspidi töös kasutame järgmist väljendusviisi vektorite olekute uuendamiseks:

$$f[x:n](y) = \begin{cases} n, & \text{kui } y = x, \\ f(y), & \text{muul juhul.} \end{cases}$$

Igale programmipunktile $n \in N$ vastab analüüsi olek $S : T \to T \to L$, mis määrab seni kehtivad lõimede olekud.

Täpsemalt, S sisaldab vektorit iga lõime t kohta lõimedest, mida ta ise on loonud. Näide: $[\mathsf{main} : [t_1 : \mathsf{created}, t_2 : \mathsf{joined}]]$ (pealõim on loonud alamlõimed t_1 ja t_2 ning lõpetanud t_2).

Topelt-kujutisvõre (vt. ka peatükki 1.2.1) kasutamise tingib vajadus lõimede loojate jälgimiseks juhtumite A ja B kirjeldusest. Järgnevalt toome domeeniga teisendusoperatsioone teostavad üleminekufunktsioonid.

3.5.1 Üleminekufunktsioonid

Lõime loomine

Lõime t' loomisel annab analüüs sellele edasi oma praeguse oleku. Olgu analüüsi olek S programmipunktis vahetult enne lõime t' loomist. Lõime t' sisendolek S' arvutatakse siis funktsiooniga:

$$\mathrm{create}(t, t', S) = \begin{cases} S[t : S(t)[t' : \mathsf{created}]], & \text{kui } S(t)(t') = \bot, \\ S[t : S(t)[t' : \top]], & \text{muul juhul.} \end{cases}$$

Lõime lõpetamine

Lõime t' lõpetamisel on kaks võimalikku olukorda: 1) lõime lõpetab tema looja; 2) lõpetajaks on mingi teine lõim.

$$\mathrm{join}(t, t', S) = \begin{cases} S[t : S(t)[t' : \mathsf{joined}]], & \text{kui } S(t)(t') = \mathsf{created}\,, \\ S[t : S(t)[t' : \top]], & \text{muul juhul.} \end{cases}$$

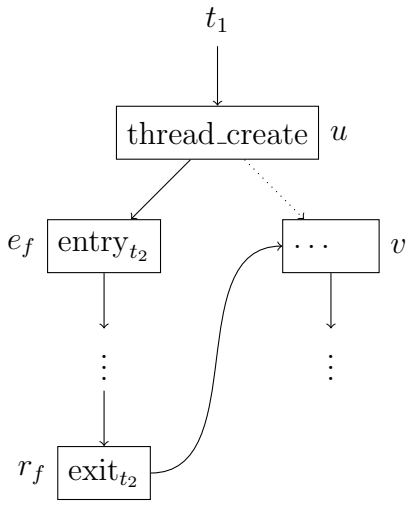
Muud üleminekufunktsioonid

Ülejäänud funktsioonid on identsusfunktsioonid ja analüüsi olekut nad ei muuda.

3.5.2 Kitsenduste süsteem

Kitsenduste süsteemi osa, mis käib lõime loomise kohta, koostame vastavalt juhtvoograafile 3.6. Lõime loomist vaatleme kui lõimele vastava funktsiooni

f väljakutset. Joonisel vastab lõime loomisele serv (u, v) (tähistatud punktiiriga). Analüüsi seisukohalt on vajalik, et loomise serva lõpp-punktis v sisalduks kogu info loodud lõime t_2 kohta, mis peab sisaldama ka t_2 enda poolt loodud ja lõpetatud lõimi.



Joonis 3.6: juhtvoog lõime loomisel.

Kasutades eespool defineeritud üleminekufunktsioone ja arvestades andmevoogu joonisel 3.6, saame järgmised kitsendused iga loomisserva $(u, \mathsf{create}(t_2, f), v)$ ja lõime t_1 kohta:

$\mathcal{A}[t_2, e_f] \sqsupseteq \text{create}(t_1, t_2, \mathcal{A}[t_1, u])$	loomisserv (olek loodud lõimele)
$\mathcal{A}[t_1, v] \sqsupseteq \mathcal{A}[t_2, r_f]$	lõime loojas järgmine üleminek

Lõime lõpetamisel on olukord lihtsam, eraldi funktsioonikutseid pole tarvis arvestada.

Jällegi tekib siin iga lõpetamisserva $(u, \mathsf{join}(t_2), v)$ ja iga lõime t_1 puhul kitsendus:

$$\mathcal{A}[t_1, v] \sqsupseteq \text{join}(t_1, t_2, \mathcal{A}[t_1, u])$$

Ning kõikides teistes servades propageerime andmevoo väärtuse muutmata kujul serva kaudu edasi ja saame järgneva kitsenduse iga lõime t kohta:

$$\mathcal{A}[t, v] \sqsupseteq \mathcal{A}[t, u] \text{ kõik muud servad } (u, v)$$

Oleme ühesõnaga spetsifitseerinud potentsiaalselt lõpmatu kitsenduste süsteemi. Seda saab aga lahendada kasutades lokaalset lahendusalgoritmi, mis ainult lahendab neid süsteemi muutujad, mis on vaja, et teada saada programmi lõpptulemus. Seega, me kutsume lahendajat välja muutujaga $\mathcal{A}[\mathsf{main}, r_{\mathsf{main}}]$ ning seega lahendame ainult neid lõimede loomised, mis programmis tegelikult toimuvad.

3.5.3 Mitteparalleelsed programmipunkid

Olgu meil vaatluse all kaks suvalist programmipunkti koos neid läbivate lõimedega (n_1, t_i) ja (n_2, t_j). Olgu nendes analüüsi tulemused $\mathcal{A}_i$ ja $\mathcal{A}_j$.

Olgu punkt n_2 ($\mathcal{A}_j$) see, mis sisaldab täielikumat infot lõimede kohta (kui punkt n_1 on andmevoograafis eespool, siis ei sisalda ta infot hiljem loodud lõimede kohta).

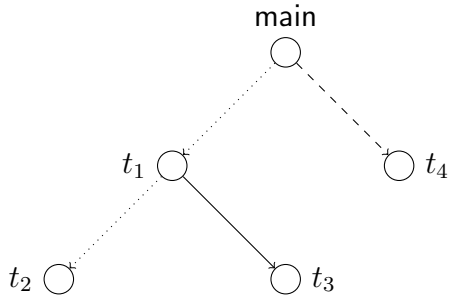
Lõimeseoste puu

Andmeid vektorites $\mathcal{A}_i$ ja $\mathcal{A}_j$ saab käsitleda puuna, mis väljendab lõime loomiste/lõpetamiste struktuuri. Näiteks vektorile $[\mathsf{main} : [t_1 : \mathsf{joined}, t_4 : \top], t_1 : [t_2 : \mathsf{created}, t_3 : \mathsf{joined}]]$ vastav loomise suhteid esitav puu asub joonisel 3.7. Joonisel tähistame loodud ja lõpetatud lõime pideva joonega; loodud, aga mitte lõpetatud lõime punktiirjoonega ning mitme instantsi loomist ($\top$) kriipsjoonega.

Puu juurtipuks on alati põhilõim main ning mingi tipu vahetud alluvad on need lõimed, mida tipp ise vahetult loob/lõpetab. Serv kahe tipu vahel esitab suhet vastavalt lõimeolekute võre väärtusele.

Käsitlust puuna kasutame siin ainult oma idee lihtsamaks seletamiseks. On mõned seosed, näiteks lõimede rekursiivne loomine, kui lõim t_1 loob lõime t_2 ja t_2 loob omakorda t_1, mida ei saa lõpliku puuna väljendada. Vektorina esituks kirjeldatud situatsioon kui $[\ldots, t_1 : [t_2 : \mathsf{many}], t_2 : [t_1 : \mathsf{many}], \ldots]$.

Joonisel 3.7 toodud näites saab väita, et t_3 ei jookse paralleelselt lõimega t_2 programmipunktis, milles see puu esineb, sest põhilõim main on unikaalne, t_1 lõpetab t_3 (vt. ka joonist 3.4). Lisaks on t_2 loodud ainult üks instants lõime poolt, mis on lõpetanud t_3. Antud puu põhjal ei väida me midagi teiste programmipunktide ega lõimede kohta.



Joonis 3.7: lõimede loomissuhte esitamine puuna.

Nüüd üldistame lähenemist.

Olgu main $= t^{(1)} \to t^{(2)} \to \ldots \to t^{(n)} = t_k$ tee analüüsi $\mathcal{A}$ olekule vastavas puus juurtipust main tipuni t_k. Lõime t_k on loodud ühe korra, kui kehtib seos $\mathcal{A}(t^i)(t^{i+1}) =$ created ja $t_i \neq t_{i+1}$ iga $i = 1, \ldots, n-1$ korral ning antud tee on ainuke tee tipuni t_k. Tähistame predikaadiga $\mathbf{unique}_{\mathcal{A}}(t_k)$ sellise tingimuse kehtimist tipule t_k puus $\mathcal{A}$.

Juhtum A

See olukord rakendub, kui lõim t_i pole loodud punktis $(n_2, t_j, \mathcal{A}_2)$ või lõim t_j pole loodud punktis $(n_1, t_i, \mathcal{A}_1)$.

Esimese juhu kehtimiseks peab iga t korral olema analüüsi tulemus $\mathcal{A}_2(t)(t_i) = \bot$. Teise juhu jaoks peab iga t korral olema analüüsi tulemus $\mathcal{A}_1(t)(t_j) = \bot$, s.t. vaadeldavas programmipunktis ei leidu lõime, mis looks vastavalt t_i või t_j.

Juhtum B

Vaaltluse all on taas kaks punkti $(n_1, t_i, \mathcal{A}_1)$ ja $(n_2, t_j, \mathcal{A}_2)$. Sarnaselt eelnevaga jaotame juhtumi kaheks: t_j on lõpetatud olekus $\mathcal{A}_1$ või t_i on lõpetatud olekus $\mathcal{A}_2$. Juhtumi B kehtimiseks peab siis üks neist kehtima.

Defineerime puu p jaoks funktsiooni lca_p, mis leiab kahe tipu lähima ühise eellase.

$$\mathrm{lca}_p(t_i, t_j) = \begin{cases} t_k, & \text{kui } t_k \text{ on } t_i \text{ ja } t_j \text{ lähim eellane puus } p \\ \mathsf{nil}, & \text{kui } t_i \text{ või } t_j \text{ puus } p \text{ ei esine} \end{cases}$$

Vaatleme esimest juhtu ning anname lõime lõpetamise kohta täpsemad tingimused. Olgu $t = \mathrm{lca}_{\mathcal{A}_1}(t_i, t_j)$ lähim ühine eellane olekule $\mathcal{A}_1$ vastavas puus. Kui $t = \mathsf{nil}$, siis pole kas t_i või t_j veel loodud (kui mõlemad on loodud, siis n.ö. halvimal juhul on ühiseks lähimaks eellaseks main). Sellisel juhul me olukorda edasi ei vaatle. Me loeme lõime t_j olema lõpetatud olekus $\mathcal{A}_1$ lõime t suhtes, kui servad teel $t = t^{(1)} \to t^{(2)} \to \ldots \to t^{(n)} = t_j$ on kõik tüüpi joined ja kehtib $\mathbf{unique}_{\mathcal{A}_1}(t)$. Teel tipust t teise lõimeni t_i peavad servad olema tüüpi $\mathsf{created}$ (ekvivalentne tingimusega $\mathbf{unique}_{\mathcal{A}_1}(t_i)$). Juhtumit illustreerib eespool esitatud puu näide 3.7. Teisel juhul on juhtumi kehtivust teostav kontroll sarnane.

3.6 Implementatsioon

Analüüsi spetsifikatsioon, mis sisaldab üleminekufunktsiooni, asub failis *src/analyses/thread.ml*. Seal paiknevad üldised üleminekufunktsioonid, mis kasutavad analüüsi domeeni koodifailis *src/cdomains/concDomain.ml* realiseeritud abifunktsioone.

Tehnilistel põhjustel on domeeni andmetüübiks $2^T \times (T \to T \to L)$. Lõimi hulgast T identifitseerimine lõimemuutuja aadressi (*l-value*) kaudu, millele vastab Goblintis andmestruktuur `Basetype.Variables`. Domeeni esimest komponenti 2^T kasutatakse hetkel vaadeldava lõime identifikaatori hoidmiseks, teine komponent $T \to T \to L$ vastab otseselt eespool kirjeldatud oleku struktuurile.

Topelt-kujutisvõre $T \to T \to L$ esitamiseks kombineerisime ühe kujutisvõre (vt. peatükk 1.2.1) teise sisse, kasutades struktuuri `MapDomain.MapBot`. Samuti sai olemasolevat koodi ära kasutada lõime oleku komponendi L jaoks. flat(A)-tüüpi domeeni loomiseks on olemas moodul `Lattice.Flat`. Baashulgana A kasutame tõeväärtusdomeeni `IntDomain.MakeBooleans`, kus väärtus true = $\mathsf{created}$ ja false = joined. `Lattice.Flat` rakendamine kaotab nende omavahelise järjestuse ning lisab elemendid $\bot$ ja $\top$.

Sobivad struktuurid moodulites `Basetype`, `MapDomain`, `InDomain` ja `Lattice` olid olemas enne töö kirjutamist ning neid tarvitatakse sarnasel viisil ka teiste analüüside domeenide koostamiseks.

Implementatsioon ei sisalda käesoleva töö raames integreerimist andmejooksude analüüsiga, sellega tegeleb Goblinti meeskond edaspidi. Implementatsioon sisaldub Goblinti koodihoidlas, millele saab juurdepääsu Goblinti kodulehe [Gob11] kaudu.

Peatükk 4

Kokkuvõte

Andmevooanalüüse kasutatakse programmide omaduste staatiliseks analüüsiks. Sellist tüüpi lähenemist iseloomustab universaalsus ning see leiab kasutamist kompilaatorites, koodi silujates, vigade otsijates jpt. töövahendites. Töös selgitasime andmevooanalüüsiga seotud olulisi mõisteid: juhtvoograaf, võre, vookitsendused, kitsenduste süsteem ja selle lahendamine.

Goblint on C-keelsete programmide andmevooanalüüside raamistik, mis on eelkõige mõeldud andmejooksude analüüsiks. Goblint pakub head tuge uute analüüside loomiseks lähtekoodi eeltöötluse, parsimise ja juhtvoograafi koostamise näol, mis on iga analüüsi lahutamatuks koostisosaks. Lisaks on implementeeritud baasanalüüs, mis analüüsib viitade väärtusi. C-keeles võivad toimuda funktsioonide väljakutsed läbi viitade, seetõttu on viitade analüüs tingimata vajalik. Töös andsime põgusa ülevaate Goblintist.

Andmejooksude analüüs inspekteerib programmi lähekoodi, et teha kindlaks, kas kõik pöörumised globaalsete muutujate poole on kaitstud lukkudega. Reaalsetes programmides leiduvad sektsioonid nagu initsialiseerimine, kus peale põhilõime pole veel ühtegi lõime loodud. Lihtsamal juhul oskab Goblint selliste olukordadega arvestada. Käesoleva töö eesmärk oli täiendada seda osa, spetsifitseerides lõimeanalüüsi, mis arvestab keerulisemat lõimede loomise/lõpetamise struktuuri.

Töö põhiosas spetsifitseerisime andmevooanalüüsi lõimede loomise/lõpetamise struktuuri uurimiseks eesmärgiga need realiseerida Goblintis. Tõime välja kaks peajuhtumit, kus esineb lõimede mitteparalleelne töö. Nendest esimene on

seotud lõime loomisega ning teine lõpetamisega. Arvestades neid juhtumeid, töötasime välja meetodi ja vajalikud tingimused nende juhtumite tuvastamiseks programmikoodis. Enne töö lõpliku versiooni valmimist üritasime kasutada lihtsalt hulkasid, mis pidid väljendama töötavaid ja lõpetatud lõimesid, kuid selle lähenemise juures ilmnesid ebamugavused mitme lõime loomise või lõpetamise info väljendamisega samas programmipunktis, sest analüüsis vastab abstraktsele lõime identifikaatorile mitu tegelikku lõime.

Töö raames implementeerisime analüüsi üleminekufunktsioonid, domeeni ja abifunktsioonid.

Thread analysis in Goblint

Master thesis (30 ECTS credits)

Raivo Laanemets

Abstract

This thesis presents a static dataflow analysis to infer thread identities and their creation hierarchy in POSIX threaded C programs. The analysis has been implemented within the Goblint race detection analyzer, which may use the provided information to exclude potential races between accesses that cannot occur concurrently. Previously, this was a major cause of false alarms as real-world programs often contain sections of initialization and finalization code where the parent thread is not running concurrently with its child threads. In these sections, the parent thread need not acquire locks to access shared variables.

The thesis consists of four parts. In the first chapter we introduce dataflow approach. The next part gives some details about the Goblint. The third, main chapter, specifies the thread analysis, and finally we say couple of words about the implementation (which is still work-in-progress).

Kirjandus

[Aho07] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman, *Compilers: Principles, Techniques, & Tools*. Pearson Education 2007.

[Api07] K.Apinis, *Abstraktsed massiividomeenid analüsaatorile Goblint*, Bakalaureusetöö. Tartu Ülikool 2007

[Api09] K.Apinis, *Programmianalüüs Goblint raamistikus*, Magistritöö. Tartu Ülikool 2009

[Cou77] P.Cousot, R.Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238—252, Los Angeles, California, 1977. ACM Press, New York.

[Fec99] C.Fecht, H.Seidl, *A Faster Solver for General Systems of Equations*. Science of Computer Programming 1999, p. 2-35.

[Gob11] http://goblint.cs.tum.edu/, *Goblinti koduleheküg*, Viimati vaadatud 23.05.2011.

[Kil73] G.A.Kildall, *A Unified Approach to Global Program Optimization*. POPL '73 Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1973.

[Nie07] H.R.Nielson, F.Nielson, *Semantics with Applications: An Appetizer*. Springer-Verlag 2007.

[Nie99] F.Nielson, H.R.Nielson, C.Hankin, *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg 1999.

[Ray05] D.Rayside *Points-To Analysis*. Loengumaterjalid. 2005.

[Sch08] M.I.Schwartzbach, *Lecture Notes on Static Analysis*. Loengumaterjalid.

[Sei03] H.Seidl, V.Vene ja M.Müller-Olm, *Global invariants for analyzing multithreaded applications*. Proc. of the Estonian Academy of Sciences: Phys., Math., 52(4):413–436, 2003.

[Voj06] V.Vojdani, *Mitmelõimeliste C-programmide kraasimine analüsaatoriga Goblint*, Magistritöö. Tartu Ülikool 2006

[Voj10] V.Vojdani, *Static Data Race Analysis of Heap-Manipulating C Programs*, Doktoritöö. Tartu Ülikool 2010

Lisad

Lisa 1: Goblinti lähtekood