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

Vesal Vojdani

Mitmelõimeliste C-programmide kraasimine analüsaatoriga Goblin

Magistritöö (40 ap)

Juhendaja: dots. Varmo Vene

TARTU 2006

Sisukord

Sissejuhatus

Programmivigade hind USA majandusele on ligikaudu 60 miljardit dollarit aastas [12] ehk viiekordne Eesti SKP. Kolmandik sellest rahast saaks väidetavalt kokku hoida, kui oleks kasutusel paremad programmide testimise raamistikud.

Definitsioon (Kraasija)**.** Kõiki süsteeme, mis üritavad võimalikult iseseisvalt programmivigu avastada, nimetame kraasijateks (*linter*).

Meie keskendume selles töös mitmelõimeliste programmide kraasimisele. Ühelt poolt on see eriti raske, kuna peab arvestama lõimede kõikide põimumistega. Teiselt poolt kaasneb mitmelõimelise paradigmaga teatud programmivigu, mida on eriti sobilik staatilise analüüsiga tuvastada. Hea näide on andmejooksud (*data races*) ja tupikud (*deadlocks*), sest puhtalt asjaolu, et programmis esineb andmejooks, ei tähenda veel, et see viga programmi käivitamisel ilmneb. Sellistes olukordades, kus Murphy seadus ei kehti, on programmianalüüs väga kasulik, sest testidest ei piisa vea leidmiseks.

Viimastel aastatel on isegi üles kerkinud firmad, kes pakkuvad kraasimisteenuseid. Kõige tähelepanuväärsemad selle valdkonna tegijad on Fortify Software, Agitar ja Coverity. Üsna aktiivselt uuritakse seda valdkonda ka ülikoolides, kuid enamasti ei taheta koodi avalikustada, kuna ülikooli juures loodud *startup*-firma müüb seda toodet. Edukad kraasimisüsteemid integreeruvad hästi arenduskeskkondadesse ning nende kasutamine on programmeerijate jaoks mugav.

Fortify tegeleb eelkõige turvalisuse probleemidega ning põhimõte on programmi analüüsida kasutades tuntud turvaaukude andmebaasi. Mõnes mõttes töötab see nagu viirustõrje. (Firma asutajad olid ka pärit Symantecist.) Võrreldes vabavaraliste kraasijatega on andmebaas palju aktuaalsem. Selle koostamiseks on tööle võetud USA parimad häkkerid ning andmebaasi uuendatakse pidevalt.

Agitari põhitoode Agitator on Eclipse'i lisakomponent, mis kombineerib erinevaid analüüsi meetodeid, et Java programmide jaoks automaatselt invariante leida. Põhitööna teeb see dünaamilist programmianalüüsi: klass instrumenteeritakse ja selle meetodeid kutsutakse nii rajaväärtustega (*null*, *maxint* jne) ning ka tüüpiliste väärtustega. Selle tulemusena tuletatakse rida potentsiaalseid invariante, mida kasutaja võib ühe hiireklõpsatusega teha üksustestideks. Firma loojad Alberto Savoia ning Roongko Doong on kaua tegelenud kommertsiaalsete testimisraamistikega ning Agitatorit on palju kiidetud Java kogukondades.

Coverity on Stanfordi Ülikooli juures loodud firma. Nad müüvad C-keele programmide kraasijat Coverity Prevent ja on oma analüüsidega suutnud leida reaalseid vigu vabavaralistes projektides, muuhulgas Linuxi tuumas. Nad nimetavad oma lähenemist metakompileerimiseks. See on sisuliselt staatiliste programmianalüüside genereerimine, millega selles magistritöös pikemalt tegeleme, aga neil on analüüside spetsifitseerimine automaadi terminites, mis on kasutajasõbralikum ning võimaldab süsteemi spetsialistidel endil analüüse kirjutada. Nagu Agitatorgi integreerub toode sihtrühma arenduskeskkonda, mis on antud juhul gcc/make.

Tartu Ülikooli juures ei ole veel sellist firmat, aga ikkagi tutvume ka siin valminud süsteemiga. Goblin on üldine raamistik C-programmide kraasimiseks, mis võimaldab analüüside kirjutamist kõrgtasemelises programmeerimiskeeles O'Caml. See põhineb Trieri andmejooksude analüsaatoril [13] ning kasutab andmevooanalüüsi, et tuvastada võimalikke andmejookse programmi lähtekoodis.

Meie lähenemise eripäraks on selle range teoreetiline alus, mistõttu analüüs on korrektne (*sound*). See ei tähenda, et meie analüüs kunagi ei eksi — see oleks algoritmiteoreetiliselt võimatu saavutus. Analüüs eksib aga ainult ühes suunas: ta on ebatäpne, kuid korrektne. Kui analüüs ütleb, et programmis ei ole andmejooksu, siis on kindel, et andmejooksu ei esine. Vastasel juhul *võib* programm olla vigane ja kasutaja peab analüsaatori poolt väljatoodud koodiridu üle vaatama.

Ebakorrektne (*unsound*) analüüs seevastu eksib mõlemas suunas, sellel on lisaks vale-positiividele ka vale-negatiivid (s.t mõni viga võib jääda avastamata), mistõttu selline analüüs ei saa kunagi garanteerida, et programm on korrektne. Kõikide omaduste jaoks ei ole võimalik piisavalt täpseid korrektseid analüüse defineerida. Coverity kasutab mõnede programmivigade tuvastamiseks ebakorrektseid analüüse ning nad väidavad, et programmianalüüsi

ülesanne ei ole programmi korrektsust tõestada, vaid võimalikult palju vigu leida. Kuid on valdkondi (transport, energeetika, meditsiin jpm), kus kulutatakse palju raha programmide korrektsuse tõestamiseks.

Töö ülesehitus on järgmine. Esimene peatükk tutvustab andmevooanalüüsi. Teine peatükk tegeleb mitmelõimeliste programmide analüüsiga ning tutvustab osalise invariandi tuletamise meetodit. Viimane peatükk tegeleb konkreetsemalt analüsaatoriga Goblin.

Peatükk 1

Andmevooanalüüs

Goblin on andmevooanalüüside loomise raamistik. Andmevooanalüüs on lihtne ja loomulik programmi staatilise analüüsimise meetod, mida kasutatakse peaaegu igas kompilaatoris programmide optimeerimiseks ja vigade leidmiseks. Kompileerimisprotsessis kasutatavad analüüsid on üldjuhul üsna lihtsad, sest nende lubatud tööaeg on piiratud. Iseseiseva analüsaatoriga võime proovida palju keerulisemaid ning täpsemaid analüüse.

Olenemata omadusest, mida kasutaja tahab analüüsida, peab C-keele programmide korral läbi viima teatud baasanalüüsi. Protseduuri väljakutsed võivad toimuda viitade kaudu ning seetõttu peab teadma, kuhu nad viitavad. Goblin üritab võimalikult täpselt arvutada kõikide muutujate väärtusi. Kasutaja poolt spetsifitseeritud analüüsis ei pruugita seda informatsiooni otseselt kasutada, kuid Goblin vajab ikkagi seda informatsiooni, et protseduuridevahelist analüüsi teostada.

Selliseid protseduuridevahelisi analüüse, mille korral eristatakse funktsioonikutseid sõltuvalt parameetrite väärtustest, nimetatakse kontekstitundlikeks (*context sensitive*) analüüsideks. Informatsiooni programmi muutujate kohta saab konstantanalüüsi teel. Seda võib teha paralleelselt konkreetse analüüsiga või eraldi faasina enne pärisanalüüsi. Selles peatükis kasutame näitena konstantanalüüsi ning järgmises peatükis vaatame ühte konkreetset kasutajaanalüüsi.

Andmevooanalüüs on sisuliselt programmiga seotud võrrandite ligikaudne lahendamine. Alustame konstantanalüüsi spetsifitseerimisest, jõuame nende võrranditeni ning vaatame, kuidas neid saab lahendada.

1.1 Konstantanalüüs

Konstantanalüüsi (*Constant Propagation, CP*) ülesanne on võimalikult täpselt kindlaks teha muutujate väärtused. Vaatame programmi *seisundit* enne iga avaldise täitmist. Konstantanalüüs peab siduma iga programmipunktiga informatsiooni muutujate väärtuste kohta. See on antud analüüsi kontekstis programmi seisundiks.

Definitsioon 1.1 (Domeen)**.** Kõikide võimalike seisundite hulka nimetatakse analüüsi domeeniks. Tavaliselt tähistatakse seda tähega $\mathbb{D}$.

Analüüsi spetsifitseerimiseks peab kõigepealt defineerima analüüsi domeeni ja siis näitama, kuidas programmi avaldised teisendavad domeeni objekte. Konstantanalüüsi domeeni element on funktsioon, mis omistab programmis esinevatele muutujatele nende väärtusi. Tähistagu Var programmis esinevate muutujate hulka ning tähistagu Val väärtuste tüüpi (näiteks täisarvud $\mathbb{Z}$). Konstantanalüüsi korral on domeen $\mathbb{D} = \text{Var} \to \text{Val}$, mida edaspidi esitame järgmiselt. Alustame mõnest konstantsest funktsioonist, näiteks $d \in \mathbb{D}$ ning lisame sellele väärtusi:

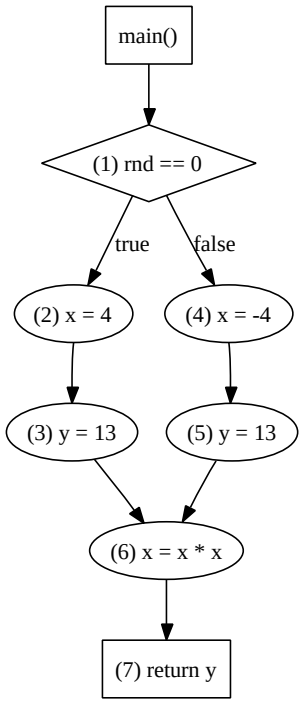
$$d[\mathtt{x} \mapsto n](z) = \begin{cases} n & \text{kui } z = \mathtt{x} \\ d(z) & \text{vastasel korral} \end{cases}$$

Andmevooanalüüsi põhimõte on informatsiooni edasi saata mööda programmi juhtimisstruktuuri. Moodustame programmist plokkskeemi, mida nimetame juhtimisvoograafiks (*Control Flow Graph*). See on suunatud graaf, mille tippudes on avaldised ja informatsioon programmi juhtimise kohta on väljendatud graafi servade abil. Joonisel 1.1 on üks näidisprogramm koos juhtimisvoograafiga.

Definitsioon 1.2 (Juhtimisvoograaf)**.** Juhtimisvoograaf on suunatud graaf $G = (N, E, s, T)$, kus N on lõplik tippude hulk, $E \subseteq N \times N$ on servade hulk, $s \in N$ on algtipp ja $T \subseteq N$ on lõpptippude hulk. Tippude n_1 ja n_2 vahel on serv $(n_1, n_2) \in E$ parajasti siis, kui programm võib liikuda tipust n_1 tippu n_2.

Analüüs peaks igale tipule $n \in N$ seadma vastavusse seisundi $d \in \mathbb{D}$. Tipu seisund väljendab teatud väidet programmi kohta, mis peab *alati kehtima* enne tipule vastava koodirea täitmist. Analüüs on seega funktsioon $N \to \mathbb{D}$. Konstantanalüüsi korral on selleks funktsiooniks $\mathsf{CP}\colon N \to (\text{Var} \to \text{Val})$, kus

```
int main() {
    int x, y, rnd;
    if (rnd == 0) {
        x = 4;
        y = 13;
    } else {
        x = -4;
        y = 13;
    }
    x = x * x;
    return y;
}
```



Joonis 1.1: Analüüsitava C-programmi kood ning juhtimisvoograaf

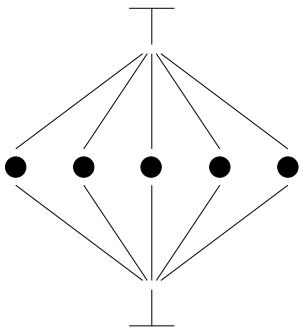
$\mathsf{CP}(n)$ on funktsioon cp_n: Var $\to$ Val, mis annab iga programmis esineva muutuja kohta tema väärtuse, kui programmi täitmine on jõudnud tipuni n. Kuna programmi muutujad ei ole kõik konstantsed ning nende täpne jälgimine on võimatu, peame rahulduma ligikaudse analüüsiga ning väärtuste hulk Val on keerulisema struktuuriga kui lihtsalt täisarvude hulk $\mathbb{Z}$. Me tahame näiteks väljendada seda, et mõne muutuja väärtus on meie jaoks tundmatu.

Selleks, et formaalselt rääkida ligikaudsest ent korrektsest analüüsist, nõuame, et domeen $\mathbb{D}$ oleks osaliselt järjestatud. Seega $\mathbb{D} = (D, \sqsubseteq)$, kus $a \sqsubseteq b$ tähendab, et seisundi b poolt väljendatud väide programmi oleku kohta on korrektne seisundi a suhtes ehk iga kord, kui programmi olekut kirjeldab a, siis kirjeldab seda ka b. Seisundi b väide on enamasti üldisem kui see, mida väljendab seisund a. Võib ka öelda, et seisundid on järjestatud informatsiooni kadumise suunas — mida suurem, seda üldisem.

Abstraktsete väärtuste hulga Val valikust sõltub konstantanalüüsi domeeni järjestus. Vaatame joonisel 1.1 oleva funktsiooni juhtimisvoograafi: enne kuuenda tipu avaldise täitmist on küll muutuja `y` konstantne, aga muutuja `x` on kas 4 või -4. Kui võtame väärtuste hulgaks Val $= \mathcal{P}(\mathbb{Z})$, siis saame

öelda $cp_6(x) = \{-4, 4\}$, mis väljendab järgmist väidet: *kui programm jõuab kuuenda tipuni, kuulub muutuja* `x` *väärtus hulka* $\{4, -4\}$. Väärtuste hulk Val on järjestatud sisalduvuse järgi. Näiteks on $cp_6(\mathtt{x}) = \{-4, 4, 50\}$ samuti korrektne väide, aga asjatult ebatäpne. Kõige üldisem väide on väärtuste hulga suurim element $\mathbb{Z}$, mis väljendab seda, et väärtust ei ole teada. Analüüsi tulemus peab olema korrektne ja nii täpne kui võimalik.

Kui muutuja väärtuste hulk on väga suur, siis ei ole kasulik jälgida tema kõikvõimalikke väärtusi hulkadena, mistõttu me võtame abstraktsema domeeni Val = $\mathit{flat}(\mathbb{Z})$. Operaatori *flat* poolt moodustatud järjestus on joonisel 1.2: unustatakse baashulga $\mathbb{Z}$ järjestus ning lisatakse kaks elementi *top* ($\top$) ja *bottom* ($\bot$), et $\forall n \in \mathbb{Z} : \bot \sqsubseteq n \sqsubseteq \top$. Element *top* väljendab seda, et väärtus on tundmatu, ning *bot* tähistab väärtuse puudumist (näiteks nulliga jagamise tõttu). Sellise väärtuste hulgaga saame ülaloleva näite korral, et $cp_6(\mathtt{y}) = 13$, aga $cp_6(\mathtt{x}) = \top$.



Joonis 1.2: Struktuur $\mathit{flat}(\mathbb{Z})$

Domeen $\mathbb{D} = \mathrm{Var} \to \mathit{flat}(\mathbb{Z})$ on ise järjestatud nii nagu funktsioonid ikka järjestatakse: $f \sqsubseteq g \iff \forall x : f(x) \sqsubseteq g(x)$. Käesolevat domeeni $\mathbb{D}$ kutsutakse Kildalli domeeniks, kuna seda kasutas esimesena Gary L. Kildall [5].

Kui oleme domeeni valinud, siis saame kirjeldada funktsiooni $\mathsf{CP} \colon N \to \mathbb{D}$. Kõigepealt peame selle funktsiooni matemaatiliselt defineerima ning leidma selle arvutamiseks algoritmi. Järgmises jaotises sõnastame andmevooprobleemi formaalselt ning püüame seda lahendada kõikide teede ühendi võtmise meetodiga (*Merge Over all Paths, MOP*).

1.2 Kõikide teede ühend

Me tahame formaalsemalt defineerida funktsiooni CP. Teeme seda programmis esinevate avaldiste kaudu. Iga avaldis mõjutab programmi seisundit teatud viisil. Konstantanalüüsi korral jälgime omistusi ning püüame nii hästi kui võimalik avaldisi väärtustada. Igal analüüsil on teatud algseisund $\iota \in D$, mis on antud juhul see, et kõikide muutujate väärtused on teadmata (keeles C on initsialiseerimata muutuja väärtus defineerimata) ehk $\forall \mathtt{x} \in \mathrm{Var} : \iota(\mathtt{x}) = \top$. Sellega alustame ning liigume mööda graafi rakendades läbitud avaldistele vastavaid teisendusi.

Täpsemalt, me seome iga juhtimisvoograafi *servaga* teatud üleminekufunktsiooni (*transfer function*), mis teostab üleminekut ühest seisundist teise. Funktsioon on seotud servaga ja mitte tipuga, kus antud avaldis oli. Näiteks tingimusavaldise `if x==0` korral on võimalik järeldada, et `true`-harus $\mathtt{x} = 0$, aga `false`-harus $\mathtt{x} \neq 0$. Seetõttu vastavad sellest avaldisest väljuvatele servadele erinevad üleminekufunktsioonid.

Definitsioon 1.3 (Üleminekufunktsioonid)**.** Programmi juhtimisvoograafi üleminekufunktsioonide komplekti tähistame $tf \colon E \to (\mathbb{D} \to \mathbb{D})$, kus tf on funktsioon, mis seab igale servale e vastavusse tema üleminekufunktsooni $tf_e \colon \mathbb{D} \to \mathbb{D}$.

Konstantanalüüsi peamine töö on programmis esinevate avaldiste abstraktne väärtustamine. Iga avaldise jaoks peab leidma sellise üleminekufunktsiooni, mis teisendab programmi olekut nii, nagu programmi täitmiselgi. Joonisele 1.1 vastavad järgmised üleminekufunktsioonid:

$$tf_{(1,2)}(d) = d[\mathtt{rnd} \mapsto 0] \qquad tf_{(1,4)}(d) = d[\mathtt{rnd} \mapsto \top]$$
$$tf_{(2,3)}(d) = d[\mathtt{x} \mapsto 4] \qquad tf_{(4,5)}(d) = d[\mathtt{x} \mapsto -4]$$
$$tf_{(3,6)}(d) = d[\mathtt{y} \mapsto 13] \qquad tf_{(5,6)}(d) = d[\mathtt{y} \mapsto 13]$$
$$tf_{(6,7)}(d) = d[\mathtt{x} \mapsto d(\mathtt{x})^2]$$

Kui meil on olemas funktsioonide komplekt tf, siis võime anda tähenduse igale graafis esinevale teele. Tee semantikaks on funktsioon, mis näitab, kuidas seisund muutub, kui programm liigub mööda seda teed.

Definitsioon 1.4 (Tee)**.** Juhtimisvoograafi $G = (N, E, s, t)$ teeks π tipust n_1 tippu n_k nimetame paarikaupa ühiste otspunktidega servade jada, mis

algab tipust n_1 ja lõpeb tipuga n_k: $\pi = (n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$, kus $(n_i, n_{i-1}) \in E$

Definitsioon 1.5 (Tee semantika)**.** Tee π semantika $[\![\pi]\!]_{tf}$ on vastavate üleminekufunktsioonide kompositsioon mööda seda teed:

$$[\![\varepsilon]\!]_{tf} = id_{\mathbb{D}\to\mathbb{D}}$$
$$[\![e_1, \ldots, e_n]\!]_{tf} = [\![e_2, \ldots, e_n]\!]_{tf} \circ tf(e_1)$$

Programmi juhtimisvoograafis on reeglina rohkem kui üks võimalik tee, kuidas programmi täitmine võib kulgeda. Meid huvitavad kõik võimalikud teed juhtimisvoograafis ning peame arvestama kõigi nende teede semantikatega. Erinevaid teid pidi tulev informatsioon tuleb ühendada. Vaatame jälle funktsiooni juhtimisvoograafi (joonis 1.1). Programm võib jõuda kuuenda tipuni kahte teed pidi ning ühel juhul $\mathtt{x} = 4$ ja teisel juhul $\mathtt{x} = -4$. Me nägime, et informatsiooni ühendamine sõltub domeeni valikust.

Mõlema domeeni korral *otsime kõige täpsemat seisundit, mis oleks kõikide teede jaoks korrektne*, ehk eeldefineeritud järjestuse terminites *vähimat seisundit, mis on kõikidest nendest suurem.* See on võreteooria terminites nende ülemine raja (tähistame $\sqcup$). Kui väärtuste hulk Val $= (\mathcal{P}(\mathbb{Z}), \subseteq)$, siis on ülemise raja operatsiooniks hulkade ühendi võtmine ning $cp_6(\mathtt{x}) = \{4\} \cup \{-4\} = \{-4, 4\}$. Kui me kasutame aga Kildalli domeeni, siis $cp_6(\mathtt{x}) = 4 \sqcup -4 = \top$. Kuna analüüsis tahetakse, et domeen oleks järjestatud ja võetakse ülemine raja, siis nõutakse, et domeen oleks võre (*lattice*).

Definitsioon 1.6 (Täielik võre)**.** Osaliselt järjestatud hulka $D = (A, \sqsubseteq)$ nimetatakse täielikuks võreks, kui igal hulga A alamhulgal leidub ülemine raja ($\bigsqcup$) ja alumine raja ($\bigsqcap$). Elemendid $\bot = \bigsqcap A$ ja $\top = \bigsqcup A$ on vastavalt võre vähim ja suurim element. Kaheelemendilise hulga ülemist raja $\bigsqcup\{a, b\}$ tähistame $a \sqcup b$. Analoogiliselt $\bigsqcap\{a, b\} = a \sqcap b$.

Võre struktuurist sõltub, kuidas informatsiooni ühendatakse. Alamhulkade võre on natuke täpsem kui Kildalli domeen. Me oleme seda võret lähendanud (*approximate*) natuke abstraktsema võrega. Praegu on intuitiivselt arusaadav, et meie lähend on korrektne. Abstraktse interpretatsiooni teooria [1] annab aga aproksimeerimisele formaalse aluse Galois' vastavuste (*Galois connection*) kaudu.

Kui domeeniks on võre, siis võib andmevooprobleemi lahenduse formaalselt sõnastada. Analüüs pidi näitama muutujate väärtusi enne iga avaldise

täitmist ehk iga tipu $n \in N$ jaoks tuleb leida seisund $\mathsf{CP}(n) \in \mathbb{D}$. Me arvutame $\mathsf{CP}(n)$ järgmise meetodiga: iga graafis esineva tee kohta, mis jõuab tipuni n, tuleb rakendada selle tee semantilist funktsiooni algseisundile ι ning lõpuks võtta kõikide tulemuste ülemine raja.

Definitsioon 1.7 (Kõikide teede ühend)**.** Andmevooprobleemi lahend kõikide teede ühendi võtmise meetodil $\mathsf{MOP}\colon N \to \mathbb{D}$ on iga tipu n korral

$$\mathsf{MOP}(n) = \bigsqcup \left\{ [\![\pi]\!]_{tf}\,(\iota) \mid \pi \text{ on tee algtipust } s \text{ tipuni } n \right\}$$

Kuna meie näidisprogrammis ei esine ühtegi tsüklit, siis on võimalikke teid juhtimisvoograafis üsna vähe. Me võime seetõttu kergesti arvutada iga tipu n jaoks $\mathsf{MOP}(n)$ väärtuse:

$$\begin{aligned}
&\mathsf{MOP}(n_1) = \iota \\
&\mathsf{MOP}(n_2) = \iota[\mathtt{rnd} \mapsto 0] && \mathsf{MOP}(n_4) = \iota \\
&\mathsf{MOP}(n_3) = \iota[\mathtt{rnd} \mapsto 0, \mathtt{x} \mapsto 4] && \mathsf{MOP}(n_5) = \iota[\mathtt{x} \mapsto -4] \\
&\mathsf{MOP}(n_6) = \iota[\mathtt{y} \mapsto 13] \\
&\mathsf{MOP}(n_7) = \iota[\mathtt{x} \mapsto 16, \mathtt{y} \mapsto 13]
\end{aligned}$$

Tsükliga programmi korral on olukord keerulisem. Kõikide teede ühendi arvutamiseks võime sooritada indekseeritud sügavutiotsingut üle juhtimisvoograafi. Alustame algseisundiga algtipu juurest ja iga tipu külastamisel rakendame läbitud serva üleminekufunktsiooni ning salvestame tulemuse indeksis. Analüüsitav tee lõpeb kas lõpptipus või siis, kui jõuame tipuni, mida oleme antud seisundiga juba külastanud. Kui analüüsi domeen on lõplik, siis lõpetab algoritm töö.

Konstantanalüüsi korral on domeen sisuliselt lõpmatu ning sellise meetodiga oleks väga ebaefektiivne analüüsi tulemust korrektselt arvutada. Tsükliga programmi korral peab halvimal juhul tsükli läbi analüüsima loenduri kõikvõimalike väärtustega. Kui seda mitte teha, ei pruugi analüüs enam olla korrektne. See ei tähenda, et ebakorrektne (*unsound*) analüüs on kasutu. Stanfordi metakompileerijad [4] kasutavad väga edukalt sügavutiotsingut programmide analüüsimiseks.

Kui me siiski tahame korrektset analüüsi, siis on olemas teine lähenemine, kus ei arvutata üle teede. Selle asemel kasutame üleminekufunktsioone võrrandite moodustamiseks ning otsime kõige täpsema (järjestuse terminites

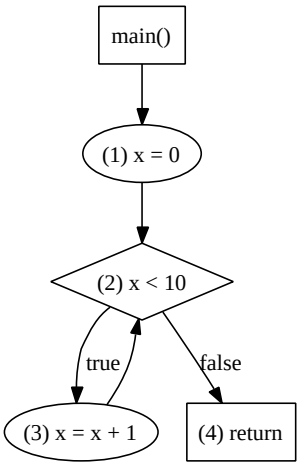
vähima) lahendi sellele süsteemile.

1.3 Vähim püsipunkt

Me oleme otsitud funktsiooni CP arvutamiseks defineerinud funktsiooni MOP, mille jaoks on olemas lahendusalgoritm väikestele domeenidele. Nüüd vaatame meetodit, mida saab kasutada ka lõpmatul domeenil programmide analüüsimiseks. Selleks peab täpsusest loobuma; aga nõuame, et leitud lahend, mida tähistame LFP (*Least Fix Point*), oleks korrektne MOPi suhtes: $\forall n : \mathsf{MOP}(n) \sqsubseteq \mathsf{LFP}(n)$.

Selle asemel, et vaadata läbi kõiki võimalikke teid, analüüsime iga tipu juures ainult sissetulevaid servi ning ühendame informatsiooni juba seal. Arvutame iga tipu seisundi tema vahetute eelkäijate seisundite põhjal. Saame sissetulevate servade kaudu teatud sõltuvused seisundite vahel, mis moodustavad võrrandisüsteemi. Kuna erinevalt MOP lahendi arvutamisest, ei jälgita vähima püsipunkti arvutamisel iga tipu juures tervet sissetulevat andmevoogu, siis öeldakse, et analüüs ei ole vootundlik (*flow-sensitive*).

```
void main() {
    int x, y;
    x = 0;
    while (x < 10) {
      x++;
    }
}
```



Joonis 1.3: Näide tsükliga programmist

Definitsioon 1.8 (Vähim püsipunkt)**.** Andmevooprobleemi lahend vähima püsipunkti arvutamise teel $\mathsf{LFP}\colon N \to \mathbb{D}$ on järgmise võrrandisüsteemi vähim

lahend.

$$\mathsf{LFP}(n) = \begin{cases} \iota & \text{kui } n = s \\ \bigsqcup \left\{ tf_{(n',n)}(\mathsf{LFP}(n')) \mid (n',n) \in E \right\} & \text{vastasel korral} \end{cases}$$

Joonisel 1.3 on programm, mis sisaldab üht lihtsat tsüklit. Selle juhtimisvoograafis on potentsiaalselt lõpmata palju teid. Vähima püsipunkti arvutamiseks peame lahendama ühe rekurrentse võrrandisüsteemi. Antud näitele vastab järgmine süsteem:

$$\begin{aligned} \mathsf{LFP}(n_1) &= \iota \\ \mathsf{LFP}(n_2) &= tf_{(1,2)}(\mathsf{LFP}(n_1)) \sqcup tf_{(3,2)}(\mathsf{LFP}(n_3)) \\ \mathsf{LFP}(n_3) &= tf_{(2,3)}(\mathsf{LFP}(n_2)) \\ \mathsf{LFP}(n_4) &= tf_{(2,4)}(\mathsf{LFP}(n_2)) \end{aligned}$$

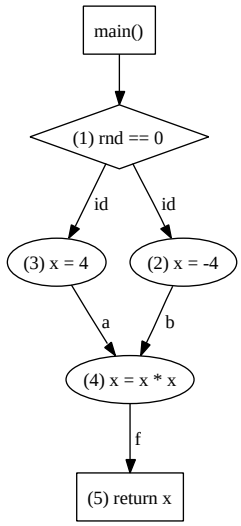
Kildalli domeeniga on analüüsi tulemus äärmiselt ebatäpne. Antud programmi korral on muutuja `x` väärtuse leidmine ka targema domeeni korral üsna raske. Näiteks võib kasutada intervalldomeeni, kuid siis peab võrrandisüsteemi lahendamiseks kasutama keerulisemaid meetodeid [2], kui hetkel Goblin võimaldab. Väga keerulisi invariante on võimalik leida kasutades domeeniks kumeraid hulktahukaid.

Kuna Goblinis teostame konstantanalüüsi ennekõike selleks, et viitadega C-programmide korral suuta analüüse teostada, siis sellisest (eba)täpsusest piisab. Veendume nüüd, et selline lähenemine on korrektne ($\forall n : \mathsf{MOP}(n) \sqsupseteq \mathsf{LFP}(n)$). Järgnev osa on analüüside spetsifitseerimisel väga oluline, sest üleminekufunktsioonid peavad analüüsi korrektsuse tagamiseks rahuldama teatud omadusi.

Võtmekoht on välja toodud joonisel 1.4. Tähistame $a' = a(\iota)$ ja $b' = b(\iota)$, siis on lõpptipus $\mathsf{MOP}(n_5) = f(a') \sqcup f(b')$ ja $\mathsf{LFP}(n_5) = f(a' \sqcup b')$. Vähima püsipunkti lahendi leidmisel võetakse igal sammul ülemine raja, kõikide teede ühendi korral ainult lõpus. Analüüsi korrektsus ja ka täpsus sõltub sellest, kuidas üleminekufunktsioonid jälgivad võre struktuuri. Kui funktsioon on distributiivne ehk $\forall x, y \in D : f(x \sqcup y) = f(x) \sqcup f(y)$, siis on vähima püsipunkti lahend ja kõikide teede ühendi lahend koguni võrdsed. Kui funktsioonid ei ole distributiivsed, siis on ikkagi lootust teha korrektset analüüsi. Hädavajalik tingimus on monotoonsus.

Definitsioon 1.9 (Monotoonsus)**.** Funktsiooni $f \colon D \to D$ nimetatakse mo-

```
int main() {
    int x,  rnd;
    if (rnd == 0) {
        x = 4;
    } else {
        x = -4;
    }
    x = x * x;
    return x;
}
```



Joonis 1.4: MOP vs. LFP täpsus

notoonseks, kui $\forall d, d' \in D : d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d)$.

Analüüsi, mille kõik üleminekufunktsioonid on monotoonsed (distributiivsed), nimetatakse monotoonseks (distributiivseks) analüüsiks. Monotoonsus on tegelikult väga loomulik tingimus. Nõuame, et üleminekufunktsioon ei tohiks ebakindlamast seisundist midagi täpsemalt järeldada, kui ta suudab kindlamast seisundist. Kui funktsioon f on monotoonne, siis on $f(x \sqcup y) \sqsupseteq f(x) \sqcup f(y)$, sest

$$\begin{matrix} x \sqsubseteq x \sqcup y \\ y \sqsubseteq x \sqcup y \end{matrix} \Rightarrow \begin{matrix} f(x) \sqsubseteq f(x \sqcup y) \\ f(y) \sqsubseteq f(x \sqcup y) \end{matrix} \Rightarrow f(x) \sqcup f(y) \sqsubseteq f(x \sqcup y)$$

See ongi intuitiivne põhjus, miks monotoonsete üleminekufunktsioonidega analüüsi korral $\forall n : \mathsf{MOP}(n) \sqsubseteq \mathsf{LFP}(n)$, s.t monotoonse analüüsi korral on LFP korrektne MOPi suhtes. Distributiivse analüüsi korral langevad nad kokku, kui võrel on veel üks kitsendus, mis puudutab kasvavaid ahelaid.

Definitsioon 1.10 (Kasvav ahel)**.** Kasvavaks ahelaks nimetatakse võre elementide jada $x_1, x_2, \ldots$, kui $\forall i : x_i \sqsubseteq x_{i+1}$. Analoogilise definitsiooni saab anda ka kahanevale ahelale. Ahelat nimetatakse statsionaarseks, kui $\exists k : \forall i \geq k : x_i = x_k$.

Definitsioon 1.11 (Kasvavate ahelate tingimus)**.** Võre rahuldab kasvavate ahelate tingimust (*Ascending Chain Condition*), kui see ei sisalda ühtegi lõp-

matult kasvavat ahelat, s.t kõik kasvavad ahelad peavad olema statsionaarsed (*all ascending chains stabilize*). Kui võre rahuldab lisaks kahanevate ahelate tingimust, mida defineeritakse analoogiliselt, siis öeldakse, et võre on lõpliku kõrgusega.

See tingimus garanteerib, et püsipunkt oleks algoritmiliselt leitav, mistõttu programmianalüüsiks kasutatavad domeenid ei sisalda lõpmatult kasvavaid ahelaid. Tingimus tagab ka LFP ning MOP lahendite kokkulangevuse. Ülaltoodud võrrandisüsteemis oli meil ainult vaja ühendada kaks teed, mistõttu distributiivsusest piisas. Üldjuhul on vaja ühendada kuitahes palju teid ja funktsioon peaks olema afiinne, s.t iga mittetühja alamhulga $X \subseteq D$ korral $f\left(\bigsqcup X\right) = \bigsqcup\{f(x) \mid x \in X\}$[1] Kui võre ei sisalda ühtegi lõpmatult kasvavat ahelat, siis järeldub afiinsus distributiivsusest [8, lk. 79]. Formaalselt saab sõnastada järgmise korrektsuse teoreemi.

Teoreem 1.12 (Kam ja Ullman)**.** *Kui üleminekufunktsioon $tf(e)$ on monotoonne iga $e \in E$ korral, siis on $\forall n : \mathit{MOP}(n) \sqsubseteq \mathit{LFP}(n)$. Kui üleminekufunktsioonid on kõik distributiivsed ja kõik kasvavad ahelad on statsionaarsed, siis $\forall n : \mathit{MOP}(n) = \mathit{LFP}(n)$.*

Selleks, et analüüs oleks üldse korrektne, peab see olema monotoonne. Distributiivsus on natuke rangem nõue, kuid suur osa kompilaatorites kasutatavatest analüüsidest on distributiivsed. Andmejooksude analüüs, millega tutvume järgmise peatüki lõpus, on samuti distributiivne. Nende analüüside korral on MOP lahend saavutatav. Mittedistributiivse analüüsi korral (näiteks konstantanalüüs) võib informatsioon kaduda. Pöördume tagasi meie esimese näite juurde (joonis 1.1). Viimase tipu juures kaotab püsipunkti arvutaja informatsiooni muutuja `x` kohta, kuid suudab vähemalt leida muutuja `y` väärtuse:

$$\mathsf{MOP}(n_5) = \iota[\mathtt{x} \mapsto 16, \mathtt{y} \mapsto 13] \qquad \mathsf{LFP}(n_5) = \iota[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 13]$$

Alustasime sellega, et tahtsime muutujate väärtusi jälgida. Selle asemel oleme saanud väga palju võreteooriat, aga nüüd lõpuks oleme jõudnud funktsioonini LFP, mis on efektiivselt arvutatav ja meie jaoks piisavalt täpne. Vaatame nüüd, kuidas seda võrrandisüsteemi lahendatakse.

[1] Monotoonsusest järeldub, et $f\left(\bigsqcup X\right) \sqsupseteq \bigsqcup\{f(x) \mid x \in X\}$, aga teistpidi võrratus kehtib distributiivse funktsiooni korral siis, kui võres ei ole lõpmatult kasvavaid ahelaid.

1.4 Püsipunkti leidmine

Võrrandisüsteemi lahendamise saab sõnastada teatava funktsiooni püsipunkti leidmise ülesandena. Olgu $\vec{x} = (x_1, \ldots, x_k) \in \mathbb{D}^k$, kus kõik juhtimisvoograafi tipud on esitatud vektorina ja $k = |N|$. Meil on siis antud võrrandisüsteem:

$$\begin{aligned} x_1 &= F_1(\vec{x}) \\ x_2 &= F_2(\vec{x}) \\ &\ldots \\ x_k &= F_k(\vec{x}) \end{aligned}$$

Defineerime vektorfunktsiooni $\vec{F}\colon \mathbb{D}^k \to \mathbb{D}^k$ selliselt, et iga $\vec{x} \in \mathbb{D}^k$ korral on funktsiooni tulemus $\vec{F}(\vec{x}) = (F_1(\vec{x}), \ldots, F_k(\vec{x}))$.

Püsipunkt on selline $\vec{x} \in \mathbb{D}^k$, et $F(\vec{x}) = \vec{x}$. Vähim püsipunkt on võre $\mathbb{D}^k$ järjestuse mõttes vähim ($\vec{x}, \vec{y} \in \mathbb{D}^k$ korral $\vec{x} \sqsubseteq \vec{y}$ parajasti siis, kui $\forall i : x_i \sqsubseteq y_i$). Meie näite (joonis 1.3) võrrandisüsteemile vastab funktsioon $\vec{F} = (F_1, F_2, F_3, F_4)$, kus

$$\begin{aligned} F_1(\vec{x}) &= \iota \\ F_2(\vec{x}) &= tf_{(1,2)}(x_1) \sqcup tf_{(3,2)}(x_3) \\ F_3(\vec{x}) &= tf_{(2,3)}(x_2) \\ F_4(\vec{x}) &= tf_{(3,4)}(x_2) \end{aligned}$$

Ülesanne on nüüd leida selline $\vec{x} \in \mathbb{D}^4$, mille korral $\vec{F}(\vec{x}) = \vec{x}$. Kõigepealt veendume, et selline $\vec{x}$ üldse leidub. Olgu $F\colon L \to L$ funktsioon üle suvalise võre L. Funktsiooni väärtus kohal l võib olla suurem, väiksem, võrdne või mittevõrreldav elemendiga l. Elementide hulka, kus funktsioon on kasvav (*extensive*), kahanev (*reductive*) või püsiv, tähistame vastavalt

$$\begin{aligned} \mathrm{Ext}(F) &= \{l \in L \mid F(l) \sqsupseteq l\} \\ \mathrm{Red}(F) &= \{l \in L \mid F(l) \sqsubseteq l\} \\ \mathrm{Fix}(F) &= \{l \in L \mid F(l) = l\} = \mathrm{Ext}(F) \cap \mathrm{Red}(F) \end{aligned}$$

Hulk $\mathrm{Fix}(F)$ on võre mingi alamhulk. Sellel hulgal on ülemine ja alumine raja olemas, isegi kui ta oleks tühi. Tähistame $\mathrm{lfp}(F) = \bigsqcap \mathrm{Fix}(F)$ ning

$\mathrm{gfp}(f) = \bigsqcup \mathrm{Fix}(F)$. Järgmine teoreem väidab, et need elemendid on tõepoolest funktsiooni püsipunktid, s.t $\mathrm{lfp}(F) \in \mathrm{Fix}(F)$. Seega $\mathrm{Fix}(F)$ ei ole ikkagi tühi ja püsipunktide hulga alumine raja ongi vähim püsipunkt.

Teoreem 1.13 (Tarski püsipunktiteoreem)**.** *Olgu $(L, \sqsubseteq)$ võre. Kui funktsioon $F\colon L \to L$ on monotoonne, siis kehtib*

$$\mathrm{lfp}(F) = \bigsqcap \mathrm{Red}(F) \in \mathrm{Fix}(F)$$
$$\mathrm{gfp}(F) = \bigsqcup \mathrm{Ext}(F) \in \mathrm{Fix}(F)$$

Püsipunkti leidmise aluseks on järgmine teoreem, mida me siin nimetame Kleene'i teoreemiks. Kleene'i teoreem on tegelikult sõnastatud pideva funktsiooni kohta üle täieliku järjestatud hulga (*Complete Ordered Set*), aga alltoodud versioon on talle piisavalt sarnane. Tegeliku teoreemiga võib tutvuda Nielsenite võrguraamatus [9, lk. 104–105]. Võre kohta on see teoreem peaaegu triviaalne, sest Tarski teoreemi põhjal on juba teada, et püsipunkt on kindlasti olemas. Peab ainult veenduma, et selline iteratiivne meetod püsipunktini jõuab ning leiab ka vähima püsipunkti.

Teoreem 1.14 (Kleene'i püsipunktiteoreem)**.** *Olgu $(L, \subseteq)$ võre, mille kõik kasvavad ahelad on statsionaarsed. Kui $F\colon L \to L$ on monotoonne, siis leidub $k \in \mathbb{N}$ nii, et $F^k(\bot) = F^{k+1}(\bot)$ on funktsiooni F vähim püsipunkt.*

Põhimõtteliselt võiks püsipunkti arvutada teoreemis toodu eeskirja alusel. Alustame domeeni vähima elemendiga ning rakendame järjest funktsiooni F kuni jõuame püsipunktini. Kuid Goblinis me ei itereeri järjest vektorfunktsiooni, vaid kasutame kaootilist iteratsiooni. Selle asemel, et vektorfunktsiooni F rakendada kõikidele muutujatele korraga, võime liikuda edasi komponenthaaval ning kasutame järgmise muutuja arvutamisel juba arvutatud muutujate uusi väärtusi. Sellist meetodit kasutatakse ka lineaarvõrrandisüsteemi lahendamisel Gauss-Seideli meetodil. Järgnevates võrrandites on vasakul tavaline iteratsioon ja paremal Seideli[2] iteratsioon. Järgmise itera-

[2]Philipp Ludwig von Seideli järgi mitte Helmut Seidl'i järgi, kes on järgmise jaotise algoritmi üks autoritest.

tsioonisammu tulemus on tähistatud tähega y.

$$\begin{aligned} y_1 &= F_1(x_1, x_2 \ldots, x_k) \\ y_2 &= F_2(x_1, x_2 \ldots, x_k) \\ &\ldots \\ y_k &= F_k(x_1, x_2 \ldots, x_k) \end{aligned} \qquad \begin{aligned} y_1 &= F_1(x_1, x_2 \ldots, x_k) \\ y_2 &= F_2(y_1, x_2 \ldots, x_k) \\ &\ldots \\ y_k &= F_k(y_1, y_2, \ldots, x_k) \end{aligned}$$

Kaootiline iteratsioon on tegelikult veelgi kaootilisem kui Seideli iteratsioon. Kui y_1 on arvutatud, siis ei pea üldsegi minema järgmise komponendi juurde, vaid võib jätkata suvalise komponendiga. Me võime ka teatud muutujatega edasi minna mitu iteratsiooni enne kui jätkame teiste muutujatega. Tsükliga programmi analüüsides on mõistlik tsüklid stabiliseerumiseni ära analüüsida ning alles siis minna edasi tsüklile järgneva koodiga. Kui jätkatakse, kuni kõik komponendid on stabiliseerunud, siis on selge, et on jõutud võrrandisüsteemi püsipunktini. See püsipunkt on ka vähim püsipunkt, olenemata lahendamisel valitud itereerimise strateegiast, millest sõltub vaid püsipunktini jõudmise kiirus. Vaatame nüüd üht eriti efektiivset lahendamise algoritmi.

1.5 Kitsenduste süsteemide lahendamine

Selles jaotises vaatame püsipunkti leidmise algoritmi, mis on Goblinis kasutatud lahendaja aluseks. Selle algoritmi autorid on Christian Fecht ja Helmut Seidl [3]. Me oleme lahendanud võrrandisüsteeme kujul $\vec{x} = F(\vec{x})$, aga Goblinis kasutame kitsenduste (*constraints*) süsteeme $\vec{x} \sqsupseteq F(\vec{x})$, kuna see võimaldab efektiivsemat algoritmi. Kõigepealt vaatame, kuidas saab võrrandisüsteeme arvutis esitada, siis vaatame kitsenduste süsteeme ning veendume, et süsteemidel on sama lahend, aga arvutamine on efektiivsem.

Võrrandisüsteem $\vec{x} = F(\vec{x})$ on siiamaani olnud vektorite terminites. Meil on vektorfunktsioon $F\colon \mathbb{D}^k \to \mathbb{D}^k$ ning otsime vektorit $\vec{x} \in \mathbb{D}^k$, mis seda võrdust rahuldab. Vektorite asemel on mugavam võrrandit sõnastada funktsionaalselt. Olgu V võrrandisüsteemi muutujate hulk. Vektorfunktsiooni korral olid meil muutujad nimedeta ja ainult järjekorra järgi eristatavad, aga mõttes olime neid samastanud juhtimisvoograafi tippudega. Me võime ka praegu võtta muutujateks V juhtimisvoograafi tippude hulga N. Funktsionaalne esitus võimaldab aga ka potentsiaalselt lõpmatu muutujate hulgaga võrrandisüstee-

mi lahendada ning seda kasutame järgmises peatükis protseduuridevahelisel analüüsil.

Võrrandisüsteemi lahendiks on vektori $\vec{x}$ asemel funktsioon $\sigma\colon V \to \mathbb{D}$, s.o muutujate väärtustus. Võrrandisüsteemi võrrand oli kujul $x_i = F_i(\vec{x})$, kus võrrandi parem pool on funktsioon $F_i\colon \mathbb{D}^k \to D$. Nüüd tuleb selleks kõrgemat järku funktsioon hulgast $R = (V \to D) \to D$. Meie võrrandite paremad pooled on funktsioonid muutujate väärtustusest domeenisse ning süsteem ei ole midagi muud kui funktsioon $\mathcal{E}\colon V \to R$, mis seab igale muutujale vastavusse tema parema poole. Vaatame näitena tavalist lineaarset võrrandisüsteemi ning selle funktsionaalset esitust.

$$\begin{aligned} x &= y + z & x &\mapsto \lambda\sigma.\ \sigma(y) + \sigma(z) \\ y &= z + 5 & y &\mapsto \lambda\sigma.\ \sigma(z) + 5 \\ z &= 5 & z &\mapsto \lambda\sigma.\ 5 \end{aligned}$$

Vaatame nüüd kitsenduste süsteeme. Programmianalüüsis on meil tihti võrrandid kujul $x = f(y) \sqcup g(z)$. Kitsenduste süsteemis väljendame sama tingimust kahe kitsendusega: $x \sqsupseteq f(y)$ ja $x \sqsupseteq g(z)$. Kirjutame joonisele 1.3 vastava süsteemi ümber kitsenduste süsteemina. Siin on meil funktsiooni LFP asemel võrrandisüsteemi muutujateks x_n $(n \in N)$:

$$\begin{aligned} x_1 &= \iota & x_1 &\sqsupseteq \iota \\ x_2 &= tf_{(1,2)}(x_1) \sqcup tf_{(3,2)}(x_3) & x_2 &\sqsupseteq tf_{(1,2)}(x_1) \\ & & x_2 &\sqsupseteq tf_{(3,2)}(x_3) \\ x_3 &= tf_{(2,3)}(x_2) & x_3 &\sqsupseteq tf_{(2,3)}(x_2) \\ x_4 &= tf_{(3,4)}(x_2) & x_4 &\sqsupseteq tf_{(3,4)}(x_2) \end{aligned}$$

Võrrandisüsteemi iga lahend on ka kitsenduste süsteemi lahend, aga kitsenduste süsteemi lahend ei pruugi olla võrrandisüsteemi lahend. Nende süsteemide vähim lahend langeb aga kokku. Seda intuitsiooni illustreerib võrrand $x = y \sqcup z$, mille korral x peabki olema (ülemise raja definitsiooni järgi) vähim element, mis rahuldab kitsendusi $x \sqsupseteq y$ ja $x \sqsupseteq z$.

Tihti eelistatakse kitsenduste süsteemide lahendamist. Vaatame taas võrrandit $x = f(y) \sqcup g(z)$. Oletame, et meil on juba arvutatud muutuja x väärtus ning järgmisel iteratsioonil muutub muutuja y väärtus. Me peaksime nüüd uuesti arvutama terve võrrandi kaasaarvatud funktsioonikutse $g(z)$. Kitsen-

duste süsteemi korral peame ainult uuesti vaatama kitsendust $x \sqsupseteq f(y)$.

Kitsenduste süsteemi lahendaja peab endiselt arvutama lahendit kujul $\sigma\colon V \to \mathbb{D}$, mis seab igale süsteemi muutujale vastavusse tema väärtuse. Kitsendused on meil antud funktsioonina $\mathcal{C}\colon V \to \mathcal{P}(R)$, mis iga muutuja jaoks annab sellele vastavate paremate poolte hulga. Paremad pooled on ise samamoodi esitatud nagu võrrandisüsteemi korral, s.t $R = (V \to \mathbb{D}) \to \mathbb{D}$. Joonisel 1.5 on algoritmi pseudokood.

Algoritm koosneb kahest abifunktsioonist *solve* ja *eval*, kuid alustame põhiprotseduuriga, mis on pseudokoodi lõpus. Kõigepealt algväärtustatakse olulised abimuutujad. Muutuja *todo* sisaldab neid kitsendusi, millega ei ole veel arvestatud. Esialgu on selleks kõik süsteemi kitsendused. Muutujas *infl* hoiame sõltuvusi kitsenduste vahel, mida me jälgime dünaamiliselt (iga kord kui parema poole väärtustame, arvutame sõltuvused ümber, sest nad võivad muutuda). Paisktabelite initsialiseerimise järel kutsutakse välja abifunktsioon *solve* kõikidel muutujatel, millest oleme huvitatud (hulk I). Nende väärtuste leidmiseks peab paljude teiste muutujate väärtused samuti arvutama ning seda tehakse ainult siis, kui neid vaja läheb.

Muutuja väärtuse leidmisel abifunktsiooniga *solve* vaadatakse kõigepealt, kas tegemist on uue muutujaga või mitte. Kui muutuja on uus, siis luuakse tema jaoks vastavad kirjed σ ja *infl* tabelites. Järgmisel real tõstetakse antud muutuja jaoks veel töötlemata kitsendusi tööjärjekorda W ning algväärtustatakse muutuja *new*, millele pärast rakendatakse kõiki kitsendused, et leida muutuja uus väärtus. Kitsenduse arvutamiseks aga ei kasutata otse paisktabelit σ, vaid selle ümber pannakse funktsioon *eval*, mis kitsenduse arvutamisel leiab rekursiivselt kasutatud muutuja väärtuse ning jälgib sõltuvusi.

Kui kõik kitsendused on rakendatud, siis juhul, kui uus seisund erineb eelmisest, peab lisama tööjärjekorda kõik kitsendused, mille arvutamine sõltub käesolevast muutujast, ning rekursiivselt uurima nende kitsenduste vasakutes pooltes olevaid muutujaid. Algoritm jätkab niimoodi tööd, kuni tööjärjekord on tühi.

Selles peatükis võtsime läbi teooria, mille alusel Goblin analüüsib üksikuid funktsioone. See on suhteliselt standardne käsitlus: iga andmevooanalüsaator toetub selles peatükis esitatud ideedele. Protseduurivahelises analüüsis on aga palju erinevaid lähenemisviise ja järgmises peatükis tutvume Goblinis kasutatud lahendusi protseduurivahelise mitmelõimelise analüüsi probleemidele.

sisend:
$\mathcal{C}$ – kitsenduste hulk
I – süsteemi muutujad, millest oleme huvitatud

väljund:
σ – võrrandisüsteemi lahend

proc $solve(x : V)$
begin
  if $x \notin dom(\sigma)$ **then** $\sigma(x) := \bot$; $infl(x) := \emptyset$ **fi**;
  $W := todo(x)$; $todo(x) := \emptyset$; $new := \sigma(x)$;
  foreach $f \in W$ **do** $new := new \sqcup f\ (\lambda y.\ eval((x, f), y))$ **od**;
  if $\sigma(x) \neq new$ **then**
    $\sigma(x) := new$; $U := \emptyset$;
    foreach $(y, f) \in infl(x)$ **do**
      $todo(y) := todo(y) \cup \{f\}$;
      $U := U \cup \{y\}$
    od;
    $infl(x) := \emptyset$;
    foreach $y \in U$ **do** $solve(y)$ **od**;
  fi;
end;

fun $eval(c : V \times R, y : V) : \mathbb{D}$
begin
  $solve(y)$; $infl(y) := infl(y) \cup \{c\}$;
  return $\sigma(y)$
end;

begin
  $\sigma := \emptyset$; $infl := \emptyset$;
  $todo := \mathcal{C}$;
  foreach $x \in I$ **do** $solve(x)$ **od**;
end

Joonis 1.5: Kitsenduste süsteemi lahendusalgoritm

Peatükk 2

Protseduuridevaheline mitmelõimeline analüüs

Siiamaani oleme tegelenud protseduurisisese (*intraprocedural*) analüüsiga. Nüüd tahame jõuda Goblinis realiseeritud protseduuridevahelise mitmelõimelise analüüsi raamistikuni.

Kõigepealt vaatame, kuidas peab protseduuri väljakutseid käsitlema. Kuna funktsioonikutsed võivad toimuda viitade kaudu, laiendame baasanalüüsi, et viitadega hakkama saada. Siis valmistume mitmelõimeliseks analüüsiks ning vaatame Trieri lahendust lõimede põimumise probleemile: osalise invariandi leidmist. Peatükk lõpeb andmejooksude analüüsi defineerimisega.

2.1 Protseduuridevaheline analüüs

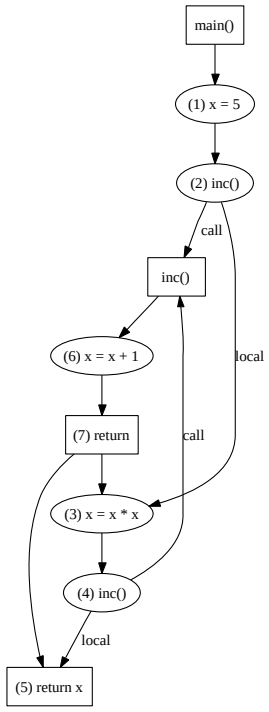
Protseduuridevahelise (*interprocedural*) analüüsi peamiseks probleemiks on funktsioonikutsete eraldamine. Joonisel 2.1 on püütud kahe protseduuri koodist teha ühine juhtimisvoograaf. Igal protseduuri kutsel on kaks serva, üks kohalik (*local*) serv, mis viib järgmisesse avaldisesse, ja üks kutse (*call*) serv, mis viib kutsutavasse protseduuri. Sellist juhtmisvoograafi nimetatakse protseduuridevaheliseks ülemgraafiks (*interprocedural supergraph*).

Probleem on aga selles, et protseduuri `inc` kutsutakse välja kahes kohas ning püsipunkti arvutamisel ühineb nende informatsioon. Kuigi on selge, et funktsiooni tulemus on 37, siis analüüs ei oska seda leida, kui me ei erista

```
int x;

void inc() {
  x++;
}

int main() {
  x = 5;
  inc();
  x = x * x;
  inc();
  return x;
}
```



Joonis 2.1: Protseduuri kutsekohtade segunemise probleem

funktsiooni `inc` erinevaid väljakutseid. Analüüsi tulemus oleks:

$$\sigma(n_1) = \iota \qquad \sigma(n_4) = \iota \qquad \sigma(n_6) = \iota$$
$$\sigma(n_2) = \iota[\mathtt{x} \mapsto 5] \qquad \sigma(n_5) = \iota \qquad \sigma(n_7) = \iota$$
$$\sigma(n_3) = \iota \qquad\qquad\qquad \sigma(n_8) = \iota$$

Informatsioon seguneb enne kuuendat tippu, mistõttu teiste tippude juures on muutuja `x` väärtus tundmatu.

Kui käsitlesime kõikide teede ühendit, siis mainisime ka indekseeritud sügavutiotsingu algoritmi, mis teostas vootundlikku (*flow sensitive*) analüüsi. Protseduuridevahelist analüüsi aga teemegi just selle ideega, et indekseerime kõik funktsiooni kutsed sõltuvalt seisundist, millega funktsiooni poole pöördutakse. Olgu $\mathcal{F}$ programmis esinevate funktsioonide hulk. Me tahame iga paari $\langle f, d\rangle \in \mathcal{F} \times \mathbb{D}$ kohta meeles pidada funktsiooni f tulemusseisundit, kui seda kutsuti välja seisundist d, sest funktsiooni parameetrid sõltuvad kutsekoha seisundist. Selleks, et eraldi analüüsida funktsiooni erinevate argumentidega kutseid, peab ka kehas olevate tippude juures seisundeid eraldi

hoidma.

Kuna meie võrrandisüsteemi lahendaja on piisavalt üldine, siis me ei pea uut algoritmi välja mõtlema, vaid piisab võrrandisüsteemi muutmisest. Kui enne oli muutujate hulgaks $V = N$, siis nüüd on kahte tüüpi muutujaid:

kutsete tulemused: $\langle f, d\rangle$, kus $f \in \mathcal{F}$ on funktsioon ja $d \in \mathbb{D}$ on seisund, millega funktsiooni välja kutsuti. Nendesse muutujatesse salvestame kutsete tulemusi.

tipud: $\langle n, d\rangle$, kus $n \in N$ on endiselt juhtimisvoograafi tipp. Kui tipp on funktsiooni f kehas, siis $d \in \mathbb{D}$ on seisund, millega funktsioon f välja kutsuti. Niimoodi on funktsiooni erinevad kutsed eraldatud.

Seega on muutujate hulk $V = (\mathcal{F} \cup N) \times \mathbb{D}$, mis on paljude analüüside korral lõpmatu.

Nüüd, kui võrrandisüsteem sisaldab protseduuridevahelist informatsiooni, huvitavad meid juhtimisvoograafis ainult protseduurisisesed servad. Meil ei ole vaja kutse servasid selles uues raamistikus, kuna meil on erilised võrrandisüsteemi muutujad kutsete tulemuste jaoks. Edaspidi me ülemgraafi ei joonista, vaid piisab programmis esinevate funktsioonide juhtimisvoograafide kõrvuti paigutamisest.

Näites on ainult kolm tippu, kus seisund muutub, ja meil on seetõttu järgmised kolm üleminekufunktsiooni:

$$\begin{aligned} tf_{(1,2)}(d) &= d[\mathtt{x} \mapsto 5] \\ tf_{(3,4)}(d) &= d[\mathtt{x} \mapsto (d(\mathtt{x}))^2] \\ tf_{(6,7)}(d) &= d[\mathtt{x} \mapsto d(\mathtt{x}) + 1] \end{aligned}$$

Vaatame, kuidas genereeritakse kitsendusi. Tuletame meelde, et lahendaja alustab hulgast I ning arvutab ainult nende muutujate väärtusi, mis on vajalikud. See võimaldab meil lahendada lõpmatuid süsteeme, sest lahendame süsteemi ainult nende muutujate suhtes, mis vastavad tegelikele programmi jooksudele. Meil tekivad iga seisundi $d \in \mathbb{D}$ jaoks järgmised kitsendused:

$\langle n, d\rangle \sqsupseteq tf_{(n',n)}\,\langle n', d\rangle$	$\forall (n', n) \in E$, kus n' on tavaline avaldis
$\langle n, d\rangle \sqsupseteq \langle f, \langle n', d\rangle\rangle$	$\forall (n', n) \in E$, kus n' on protseduuri f kutse
$\langle s, d\rangle \sqsupseteq d$	$\forall s \in N$, kus s on algtipp
$\langle f, d\rangle \sqsupseteq \langle t, d\rangle$	$\forall t \in N$, kus t on funktsiooni f lõpptipp

Joonisele 2.1 saaksime $\forall d \in \mathbb{D}$ korral järgmised kitsendused:

$$\begin{aligned}
\langle n_1, d\rangle &\sqsupseteq d & \langle n_6, d\rangle &\sqsupseteq d \\
\langle n_2, d\rangle &\sqsupseteq \langle n_1, d\rangle\,[\mathtt{x} \mapsto 5] & \langle n_7, d\rangle &\sqsupseteq \langle n_6, d\rangle\,[\mathtt{x} \mapsto \langle n_6, d\rangle\,(\mathtt{x}) + 1] \\
\langle n_3, d\rangle &\sqsupseteq \langle \mathtt{inc}, \langle n_2, d\rangle\rangle & \langle \mathtt{inc}, d\rangle &\sqsupseteq \langle n_7, d\rangle \\
\langle n_4, d\rangle &\sqsupseteq \langle n_3, d\rangle\,[\mathtt{x} \mapsto (\langle n_3, d\rangle\,(\mathtt{x}))^2] \\
\langle n_5, d\rangle &\sqsupseteq \langle \mathtt{inc}, \langle n_4, d\rangle\rangle \\
\langle \mathtt{main}, d\rangle &\sqsupseteq \langle n_5, d\rangle
\end{aligned}$$

Kui me lahendame antud süsteemi muutujate hulgaga $I = \{\langle \mathtt{main}, \iota\rangle\}$, siis lahendusalgoritm arvutaks järgmiste muutujate väärtusi:

$$\begin{aligned}
\sigma\,\langle n_1, \iota\rangle &= \iota & \sigma\,\langle n_6, \iota[\mathtt{x} \mapsto 5]\rangle &= \iota[\mathtt{x} \mapsto 5] \\
\sigma\,\langle n_2, \iota\rangle &= \iota[\mathtt{x} \mapsto 5] & \sigma\,\langle n_7, \iota[\mathtt{x} \mapsto 5]\rangle &= \iota[\mathtt{x} \mapsto 6] \\
\sigma\,\langle n_3, \iota\rangle &= \iota[\mathtt{x} \mapsto 6] & \sigma\,\langle \mathtt{inc}, \iota[\mathtt{x} \mapsto 5]\rangle &= \iota[\mathtt{x} \mapsto 6] \\
\sigma\,\langle n_4, \iota\rangle &= \iota[\mathtt{x} \mapsto 36] & \sigma\,\langle n_6, \iota[\mathtt{x} \mapsto 36]\rangle &= \iota[\mathtt{x} \mapsto 36] \\
\sigma\,\langle n_5, \iota\rangle &= \iota[\mathtt{x} \mapsto 37] & \sigma\,\langle n_7, \iota[\mathtt{x} \mapsto 36]\rangle &= \iota[\mathtt{x} \mapsto 37] \\
\sigma\,\langle \mathtt{main}, \iota\rangle &= \iota[\mathtt{x} \mapsto 37] & \sigma\,\langle \mathtt{inc}, \iota[\mathtt{x} \mapsto 36]\rangle &= \iota[\mathtt{x} \mapsto 37]
\end{aligned}$$

Meie analüüs on vootundlik protseduuri kutsete suhtes. Sellist vootundlikkust protseduuri kutsekoha seisundi suhtes nimetatakse kontekstitundlikkuseks (*context sensitivity*). Kuigi korrektne vootundlik protseduurisisene konstantanalüüs oli meie jaoks ebameeldiv, siis praegu ei ole olukord nii kriitiline. Erinevad seisundid, millega võib funktsiooni välja kutsuda püsipunkti arvutamise käigul, on üsna piiratud. Erineva seisundiga saab funktsiooni välja kutsuda ainult erinevatest kutsekohtadest või tsükli korral[1], kus võib samasse kutsekohta jõuda mitu korda. Kuna meie protseduurisisene analüüs muudab tipus oleva väärtuse ainult rangelt suuremaks (võre järjestuse mõttes), siis kontekstide arv samast kutsekohast sõltub võre kõrgusest, mitte võre elementide arvust, ning Kildalli domeeni kõrgus on lõplik ($2 * |\mathrm{Var}|$).

Protseduuridevahelise analüüsi idee peaks olema selge, aga me ei ole veel arvestanud parameetriedastusega ning selle tõsiasjaga, et reaalsetes C-programmides ei ole alati väljakutsutav funktsioon staatiliselt määratud.

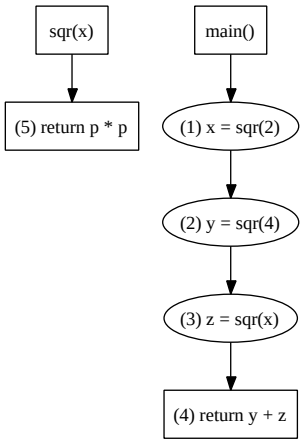
[1]Rekursiooni korral võib ka samast kohast funktsiooni kutsuda erineva seisundiga ja seda peab eraldi käsitlema.

2.2 Parameetrite edastus

Selles jaotises lahendame parameetriedastuse ja funktsiooni tulemuse salvestamise probleemi. Joonisel 2.2 on programm, kus funktsiooni kutsutakse välja erinevate argumentidega ning tulemust salvestatakse muutujates.

```
int sqr(int p) {
    return p*p;
}

int main() {
    int x,y,z;
    x = sqr(2);
    y = sqr(4);
    z = sqr(x);
    return y+z;
}
```



Joonis 2.2: Protseduuride vahelise analüüsi näidisprogramm

Iga protseduuri kutsele vastava tipu $n \in N$ jaoks defineerime järgmised kaks funktsiooni:

$$entry_{(n,n')} : \mathbb{D} \to \mathbb{D}$$
$$comb_{(n,n')} : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$$

Üht rakendame funktsiooni sisenemisel ja teist funktsioonist tagasipöördumisel. Kui toimub funktsiooni kutse, siis $entry_e$ peab looma kutse jaoks vastava algseisundi, kus formaalsetele parameetritele seatakse vastavusse analüüsi poolt arvutatud argumentide abstraktsed väärtused. Näidisprogrammi korral on seisunditeks esimesel tipul $\iota[\mathtt{p} \mapsto 2]$ ning teisel ja kolmandal tipul $\iota[\mathtt{p} \mapsto 4]$. Kutsekoha lokaalsete muutujate seisund on kutsutava protseduuri eest peidetud, kuna nad ei mõjuta funktsiooni `sqr` arvutamise tulemust.

Funktsiooni tulemusavaldisega arvestamiseks kujutame ette, et on olemas erilised servad $(t, f) \in \mathrm{Ret} = N \times \mathcal{F}$ lõpptipust t funktsioonile f. Selle serva üleminekufunktsioonid võiksid ühele erilisele muutujale (`ret`) omistada tulemusavaldise abstraktse väärtuse. Funktsiooni naasmisel peame tulemuse salvestama kohalikus muutujas. Selleks kombineerime kutse tulemuse ja kohaliku seisundi funktsiooniga $comb_e$. Näiteprogrammi jaoks saame järgmised

üleminekufunktsioonid:

$$
\begin{aligned}
&\mathit{entry}_{(1,2)}(d) = \iota[\mathtt{p} \mapsto 2] && \mathit{comb}_{(1,2)}(d_c, d_l) = d_l[\mathtt{x} \mapsto d_c(\mathtt{ret})]\\
&\mathit{entry}_{(2,3)}(d) = \iota[\mathtt{p} \mapsto 4] && \mathit{comb}_{(2,3)}(d_c, d_l) = d_l[\mathtt{y} \mapsto d_c(\mathtt{ret})]\\
&\mathit{entry}_{(3,4)}(d) = \iota[\mathtt{p} \mapsto d(\mathtt{x})] && \mathit{comb}_{(3,4)}(d_c, d_l) = d_l[\mathtt{z} \mapsto d_c(\mathtt{ret})]\\
&\mathit{tf}_{(4,\mathtt{main})}(d) = d[\mathtt{ret} \mapsto d(\mathtt{y}) + d(\mathtt{z})] && \mathit{tf}_{(5,\mathtt{sqr})}(d) = d[\mathtt{ret} \mapsto (d(\mathtt{p}))^2]
\end{aligned}
$$

Parameetrite edastus võib tunduda niivõrd loomuliku protsessina, et võib tekkida küsimus, miks iga analüüsi spetsifitseerimiseks peab uuesti defineerima *entry* ja *comb* funktsioone. Me oleme siin tegelenud konstantanalüüsiga ja need funktsioonid on seetõttu väga standardsed, aga mitte kõik analüüsid ei jälgi muutujate seisundeid. Paljude analüüside korral ei oma parameetriedastus üldse mõtet, küll aga võib tahta funktsioonile muudmoodi informatsiooni edastada ja tagasipöördumisel kohaliku seisundiga integreerida. Raamistik on piisavalt üldine, et seda lubada.

Kitsenduste süsteemi tuleb nüüd genereerida nii, et iga seisundi $d \in \mathbb{D}$ jaoks saame järgmised kitsendused:

$$
\begin{aligned}
&\langle n, d\rangle \sqsupseteq \mathit{tf}_{(n',n)}\,\langle n', d\rangle && \forall (n', n) \in E, \text{ kus } n' \text{ on tavaline avaldis}\\
&\langle n, d\rangle \sqsupseteq \mathit{comb}_{(n',n)}\left(\left\langle f, \mathit{entry}_{(n',n)}\,\langle n', d\rangle\right\rangle, \langle n', d\rangle\right) && \\
& && \forall (n', n) \in E, \text{ kus } n' \text{ on protseduuri } f \text{ kutse}\\
&\langle s, d\rangle \sqsupseteq d && \forall s \in N, \text{ kus } s \text{ on algtipp}\\
&\langle f, d\rangle \sqsupseteq \mathit{tf}_{(t,f)}\,\langle t, d\rangle && \forall t \in N, \text{ kus } t \text{ on funktsiooni } f \text{ lõpptipp}
\end{aligned}
$$

Joonisel 2.1 olevale näitele vastavad $\forall d \in \mathbb{D}$ korral järgmised kitsendused:

$$
\begin{aligned}
\langle n_1, d\rangle &\sqsupseteq d\\
\langle n_2, d\rangle &\sqsupseteq \langle n_1, d\rangle\,[\mathtt{x} \mapsto \langle \mathtt{sqr}, \iota[\mathtt{p} \mapsto 2]\rangle\,(\mathtt{ret})]\\
\langle n_3, d\rangle &\sqsupseteq \langle n_2, d\rangle\,[\mathtt{y} \mapsto \langle \mathtt{sqr}, \iota[\mathtt{p} \mapsto 4]\rangle\,(\mathtt{ret})]\\
\langle n_4, d\rangle &\sqsupseteq \langle n_3, d\rangle\,[\mathtt{z} \mapsto \langle \mathtt{sqr}, \iota[\mathtt{p} \mapsto \langle n_3, d\rangle\,(\mathtt{x})\rangle\,(\mathtt{ret})]\\
\langle \mathtt{main}, d\rangle &\sqsupseteq \langle n_4, d\rangle\,[\mathtt{ret} \mapsto \langle n_4, d\rangle\,(\mathtt{y}) + \langle n_4, d\rangle\,(\mathtt{z})]
\end{aligned}
$$

$$
\begin{aligned}
\langle n_5, d\rangle &\sqsupseteq d\\
\langle \mathtt{sqr}, d\rangle &\sqsupseteq \langle n_5, d\rangle\,[\mathtt{ret} \mapsto (\langle n_5, d\rangle\,(\mathtt{p}))^2]
\end{aligned}
$$

Kui nüüd jälle lahendada süsteem muutujate hulgaga $I = \{\langle \mathtt{main}, \iota\rangle\}$, siis

saaksime järgmised muutujate väärtused:

$$\begin{aligned}
\sigma\langle n_1, \iota\rangle &= \iota & \sigma\langle n_5, \iota[\mathtt{p} \mapsto 2]\rangle &= \iota[\mathtt{p} \mapsto 2] \\
\sigma\langle n_2, \iota\rangle &= \iota[\mathtt{x} \mapsto 4] & \sigma\langle \mathtt{sqr}, \iota[\mathtt{p} \mapsto 2]\rangle &= \iota[\mathtt{ret} \mapsto 4] \\
\sigma\langle n_3, \iota\rangle &= \iota[\mathtt{x} \mapsto 4, \mathtt{y} \mapsto 16] & \sigma\langle n_5, \iota[\mathtt{p} \mapsto 4]\rangle &= \iota[\mathtt{p} \mapsto 4] \\
\sigma\langle n_4, \iota\rangle &= \iota[\mathtt{x} \mapsto 4, \mathtt{y}, \mathtt{z} \mapsto 16] & \sigma\langle \mathtt{sqr}, \iota[\mathtt{p} \mapsto 4]\rangle &= \iota[\mathtt{ret} \mapsto 16] \\
\sigma\langle \mathtt{main}, \iota\rangle &= \iota[\mathtt{x} \mapsto 4, \mathtt{y}, \mathtt{z} \mapsto 16, \mathtt{ret} \mapsto 32]
\end{aligned}$$

Paneme tähele, et neljanda tipu juures ei ole vaja muutuja $\langle \mathtt{sqr}, \iota[\mathtt{p} \mapsto 4]\rangle$ väärtust uuesti arvutada, sest protseduuri eest on lokaalsete muutujate seisund peidetud (kasutame ι mitte d). Viitade korral ei ole enam nii lihtne, sest lokaalsed muutujad võivad funktsiooni arvutamist mõjutada. Analüüs peaks tegelikult kindlaks tegema, millised muutujad on kättesaadavad, aga eeldame siin, et funktsioonid suhtlevad ainult kas parameetrite või globaalsete muutujate kaudu.

Näites ei olnud globaalseid muutujad. Nendega peab käituma nagu esimeses protseduuridevahelises näites. Protseduuri sisenemisel edastame globaalsete muutujate seisundi ning naasmisel peab funktsioon $comb(d_c, d_l)$ globaalide väärtusi võtma protseduuri tulemusseisundist d_c ning lokaalsete muutujate omi kohalikust seisundist d_l.

Ideaalses maailmas oleksime nüüd valmis protseduuridevahelise analüüsiga, kuid kuna protseduuri kutsed võivad toimuda viitade kaudu, siis peab veel analüüsi laiendama.

2.3 Viitade kaudu protseduurikutsed

C-keele süntaksi järgi on väljakustutav funktsioon antud tavalise avaldisena, mille väärtustamise tulemus peaks viitama mingile funktsioonile. Seetõttu peame jälgima funktsiooniviitasid. Kui me täisarvude korral ei ole huvitatud väärtuste hulkadest, siis viitade korral vajame täpsemat struktuuri. Vaatame joonisel 2.3 toodud programmi. Sisuliselt käitub ta nagu meie esimene näidisprogramm (joonis 1.1), aga harudele järgnev väljakutse toimub viida kaudu. Programmi staatilisel analüüsil me ei tea, kumb funktsioon kutsutakse. Me ei saa siin öelda, et $cp_{n_4}(\mathtt{fun}) = \top$, sest siis peaksime analüüsi katkestama.

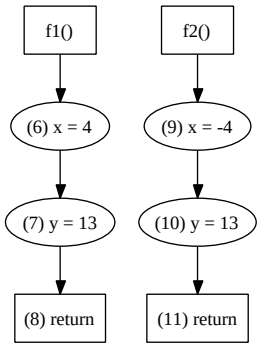
Lihtsuse mõttes eeldame, et võimalikud aadressid on ainult programmis

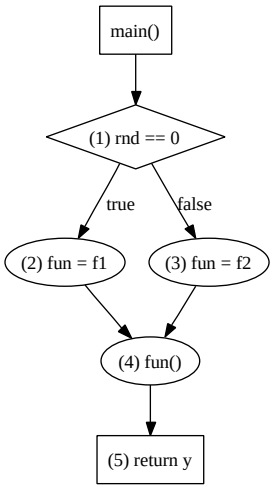
```
int x, y;

void f1() {
    x = 4;
    y = 13;
}

void f2() {
    x = -4;
    y = 13;
}

int main() {
  int rnd;
  void (*fun)();
  if (rnd == 0) {
      fun = f1;
  } else {
      fun = f2;
  }
  fun();
  return y;
}
```





Joonis 2.3: Viitade kaudu protseduurikutsetega programm

esinevate muutujate ja funktsioonide aadressid. Kõik muud aadressid võime koondada ühte spetsiaalsesse aadressi nimega `heap`. Sellisel juhul on aadresside hulk $\mathrm{Addr} = \mathrm{Var} \cup \mathcal{F} \cup \{\texttt{heap}\}$ lõplik ja kuna reaalselt kasutatud viitade hulk on tavaliselt väike, siis võime väärtuste hulgaks viitade korral võtta $\mathrm{Val}_{\mathrm{Addr}} = (\mathcal{P}(\mathrm{Addr}), \subseteq)$. Täisarvude korral võtame endiselt $\mathrm{Val}_{\mathbb{Z}} = \mathit{flat}(\mathbb{Z})$. Kuna muutuja tüübi järgi on võimalik neid eristada, siis muutujate hulk Var jaguneb samuti hulkadeks $\mathrm{Var}_{\mathbb{Z}}$ ja $\mathrm{Var}_{\mathrm{Addr}}$. Meie domeeniks on endiselt $\mathbb{D} = \mathrm{Var} \to \mathrm{Val}$, aga selliselt, et domeeni element $d \in \mathbb{D}$ rahuldab

$$\forall \mathtt{x} \in \mathrm{Var}_{\mathbb{Z}} : d(\mathtt{x}) \in \mathrm{Val}_{\mathbb{Z}}$$
$$\forall \mathtt{p} \in \mathrm{Var}_{\mathrm{Addr}} : d(\mathtt{p}) \in \mathrm{Val}_{\mathrm{Addr}}$$

Need tingimused garanteerivad, et domeen $\mathbb{D}$ on võre.

Vaatame nüüd, kuidas see võimaldab meil analüüsida funktsiooniviitadega programmi. Nüüd on $entry_e$ funktsiooni ülesandeks tulemusena anda ka funktsioonide hulk, kuhu viidatakse. Kuna Goblinis on erinevate funktsioonide formaalsed parameetrid unikaalsed, siis on vajalik iga kutsutava funktsiooni jaoks ka oma algseisund:

$$\begin{aligned} &entry_{(n,n')} \colon \mathbb{D} \to \mathcal{P}(\mathcal{F}) \times \mathbb{D} && \text{(Põhimõtteline lahendus)} \\ &entry_{(n,n')} \colon \mathbb{D} \to \mathcal{P}(\mathcal{F} \times \mathbb{D}) && \text{(Goblinis tehnilistel põhjustel)} \end{aligned}$$

Goblinis kasutatud $entry_e$ funktsiooni tüüp on lihtsalt hulk paaridest $\mathcal{F} \times \mathbb{D}$. Võrrandisüsteemis on ka funktsioonikutsete tulemused salvestatud muutujatesse, mis on täpselt sama tüübiga. Seetõttu on $entry_e$ funktsiooni tulemus võrrandisüsteemi muutujate hulk.

Kitsenduste süsteemi moodustamisel peab funktsioonide korral ühendama kõikide võimalike kutsete tulemused. Funktsiooni $entry_e$ rakendamisel saame muutujate hulga, mis vastab kõikide võimalike kutsete tulemustele. Me võtame nende ülemise raja:

$$\langle n, d\rangle \sqsupseteq comb_{(n',n)}\left(\bigsqcup entry_{(n',n)}\,\langle n', d\rangle\,,\ \langle n', d\rangle\right) \qquad \forall (n', n) \in E, \text{ kus } n' \text{ on protseduuri kutse}$$

Ülejäänud kitsendused luuakse nagu eelmises jaotises.

Vaatame nüüd näitele vastavat süsteemi. Toome siin välja ainult põhiprogrammile vastavad kitsendused, kuna muud on eelnevaga analoogilised. Me saame järgmised kitsendused ($\forall d \in \mathbb{D}$):

$$\begin{aligned} \langle n_1, d\rangle &\sqsupseteq d \\ \langle n_2, d\rangle &\sqsupseteq \langle n_1, d\rangle \qquad \langle n_3, d\rangle \sqsupseteq \langle n_1, d\rangle \\ \langle n_4, d\rangle &\sqsupseteq \langle n_2, d\rangle\,[\texttt{fun} \mapsto \{\texttt{f1}\}] \\ \langle n_4, d\rangle &\sqsupseteq \langle n_3, d\rangle\,[\texttt{fun} \mapsto \{\texttt{f2}\}] \\ \langle n_5, d\rangle &\sqsupseteq \langle n_4, d\rangle\,[\texttt{x} \mapsto d_c(\texttt{x}), \texttt{y} \mapsto d_c(\texttt{y})], \text{ kus} \\ &\qquad d_c = \bigsqcup \{\langle f, d_e\rangle \mid f \in \langle n_4, d\rangle\,(\texttt{fun})\} \text{ ja} \\ &\qquad d_e = \iota[\texttt{x} \mapsto \langle n_4, d\rangle\,(\texttt{x}), \texttt{y} \mapsto \langle n_4, d\rangle\,(\texttt{y})] \\ \langle \texttt{main}, d\rangle &\sqsupseteq \langle n_5, d\rangle\,[\texttt{ret} \mapsto \langle n_5, d\rangle\,(\texttt{y})] \end{aligned}$$

Selle süsteemi lahendi huvitavam osa on:

$$\begin{aligned}
\sigma\langle n_1, \iota\rangle &= \sigma\langle n_2, \iota\rangle = \sigma\langle n_3, \iota\rangle = \iota \\
\sigma\langle n_4, \iota\rangle &= \iota[\texttt{fun} \mapsto \{\texttt{f1}, \texttt{f2}\}] \\
\sigma\langle \texttt{f1}, \iota\rangle &= \iota[\texttt{x} \mapsto 4, \texttt{y} \mapsto 13] \\
\sigma\langle \texttt{f2}, \iota\rangle &= \iota[\texttt{x} \mapsto -4, \texttt{y} \mapsto 13] \\
\sigma\langle \texttt{main}, \iota\rangle &= \sigma\langle n_5, \iota\rangle = \iota[\texttt{fun} \mapsto \{\texttt{f1}, \texttt{f2}\}, \texttt{y} \mapsto 13]
\end{aligned}$$

Viitade juuresolekul muutub ka konstantanalüüs ise natuke keerulisemaks. Kui näiteks muutuja `x` jaoks on analüüs leidnud, et aadresside hulk on üheelemendiline {&a}, siis võib avaldise `*x = 4` korral lihtsalt uuendada muutuja `a` väärtuse $cp_{\text{uus}} = cp_{\text{vana}}[\texttt{a} \mapsto 4]$. Kui aga aadresside hulk on näiteks {&a, &b}, siis selle avaldise korral peab uueks seisundiks võtma

$$cp_{\text{uus}} = cp_{\text{vana}}[\texttt{a} \mapsto 4 \sqcup cp_{\text{vana}}(\texttt{a}),\ \texttt{b} \mapsto 4 \sqcup cp_{\text{vana}}(\texttt{b})],$$

kuna ei ole teada, millise muutuja seisund muutus.

Nüüd on protseduuridevahelise analüüsi kõik põhilised probleemid uuritud ja võime vaadata, kuidas teostatakse Goblinis mitmelõimelist analüüsi.

2.4 Mitmelõimeline analüüs

Mitmelõimeliste programmide analüüs on väga keeruline. Lõimede paralleelsel käivitamisel on teoreetiliselt võimalik, et neid täidetakse vaheldumisi paljudel erinevatel viisidel. Kui käivitada kaks 10-realist lõime, millel ei ole järjekorrale seatud kitsendusi (sünkronisatsioonipunkte jms), siis on 184 756 erinevat võimalust, mis järjekorras nende käske täita. Kui programm on vähegi paralleelne, siis on lootusetu kõikide põimumistega arvestada.

Programmi võimalikud täitmisjärjekorrad ei ole teineteisest nii radikaalselt erinevad, et neid kõiki peaks eraldi analüüsima. Lõimede käsud, mis ei tegele jagatud ressurssidega, on väga piiratud mõjuga. Nende erinevad põimumised ei ole analüüsi seisukohalt olulised. Mitmelõimelise analüüsi keskne probleem on kuidas kõikide võimalike põimumiste asemel teha midagi mõistlikumalt ajalise keerukuse mõttes.

Trieri lahendus [13] sellele probleemile on järgmine. Analüüsime lõimesid võimalikult iseseisvalt: püüame isoleerida selle, mis kuulub ainult üksikule

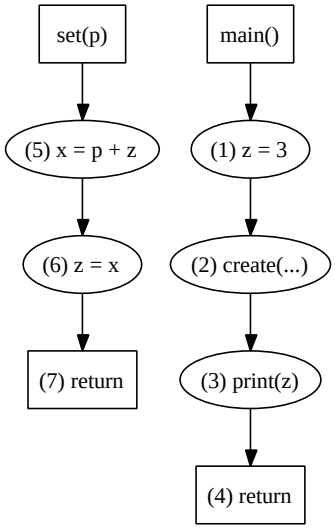
lõimele, ning selle, mis võib ka teiste lõimede tegevust mõjutada. Vaatame konkreetsemalt C-keele Posix-lõimesid. Suhtlus toimub globaalsete muutujate kaudu, sest lokaalsetele muutujatele pääseb ligi ainult konkreetne lõim. Tuletame globaalsete muutujate kohta *ühe* invariandi, mis kehtib programmi kogu täitmisaja jooksul.

Esmapilgul võib see tunduda lootusetult ebatäpne: invariandid, mis terve programmi vältel kehtivad, on ju ainult need, mis on deklareeritud võtmesõnaga `const`. Kuid on omadusi, mis tõepoolest peavad kehtima, iga kord kui globaalse muutujaga midagi tehakse. Andmejooksude analüüsis me uurime just seda tüüpi omadust: kas *kõik* pöördused antud globaalse muutuja poole on ühise lukuga kaitstud.

```
int z;

void set(int p) {
    int x;
    x = p + z;
    z = x;
}

int main() {
    int id;
    z = 3;
    create(&id, set, 5);
    print(z);
}
```



Joonis 2.4: Mitmelõimelise programmi näide

Raamistikuga tutvumiseks vaatame kõigepealt mitmelõimelist konstantanalüüsi. Joonisel 2.4 on programm, kus luuakse teises tipus uus lõim. Funktsioon `create` toimib sisuliselt nagu Posix-standardi `pthread_create` funktsioon. Kutse tulemusel luuakse uus lõim unikaalse identifikaatoriga, mis salvestatakse muutujas `id`, ning lõim käivitab funktsiooni `set` argumendiga 5. Kutsel on ka täisarvuline tulemusväärtus, mis näitab, kas lõime loomine õnnestus või mitte, aga seda näidisprogrammis ignoreeritakse.

Kui lõim on loodud, siis ta omistab mingil hetkel globaalsele muutujale `z`, ning kui põhiprogramm tahab kolmandal tipul muutuja `z` väärtust väljastada, siis me ei tea, kas muutuja väärtus on kolm või seitse. See sõltub sellest,

kes enne jõuab muutujani `z` — lõimed osalevad andmejooksus. Staatilisel analüüsil peame mõlema tulemusega arvestama. Kuna me ei tea, millal omistus võib toimuda, siis püüame globaalsetele muutujatele leida ühe invariandi, mis kehtiks kogu aeg ehk igal programmipunktil.

Kokku on meil globaalne informatsioon, mis peab kehtima kõikides programmipunktides, ning kohalik informatsioon, mis on seotud ühe konkreetse programmipunktiga. Olgu Var_G programmi globaalsete muutujate hulk ning Var_L lokaalsete muutujate hulk ($\mathrm{Var} = \mathrm{Var}_G \cup \mathrm{Var}_L$). Domeen jaguneb ka kaheks osaks $\mathbb{D} = \mathbb{D}_L \times \mathbb{D}_G$: kohalik seisund $\mathbb{D}_L$ ja globaalne seisund $\mathbb{D}_G$. Mitmelõimelise konstantanalüüsi korral on domeenid sisuliselt samad:

$$\mathbb{D}_L = \mathrm{Var}_L \to \mathit{flat}(\mathbb{Z}) \qquad\qquad \mathbb{D}_G = \mathrm{Var}_G \to \mathit{flat}(\mathbb{Z})$$

Lokaalsete muutujate väärtusi saame analüüsida täpselt nagu varem, kuid me peame globaalse olekuga arvestama. Selleks uurime uut liiki kitsenduste süsteeme. Siiamaani otsisime kitsenduste süsteemi $\vec{x} \sqsupseteq F(\vec{x})$ lahendit, kus F on vektorfunktsioon $\mathbb{D}^k \to \mathbb{D}^k$ ja k on süsteemi muutujate arv. Nüüd otsime süsteemi lahendit ühe globaalse invariandi $\tau \in \mathbb{D}_G$ korral. Süsteemi vektorfunktsioon $F\colon \mathbb{D}_L^k \times \mathbb{D}_G \to \mathbb{D}_L^k \times \mathbb{D}_G$ võtab seega arvesse globaalset seisundit ja võib ka seda muuta. Kitsenduste süsteem ise on kujul $(\vec{x}, \tau) \sqsupseteq F(\vec{x}, \tau)$, kus τ on süsteemi ühine globaalne seisund.

Programmianalüüsi jaoks on kitsenduste süsteemi muutujad endiselt $V = (N \cup \mathcal{F}) \times \mathbb{D}_L$ ning kasutaja poolt defineeritud üleminekufunktsioonidest moodustame kitsendusi. Kuid üleminekud võivad mõjutada globaalset seisundit, mistõttu üleminekufunktsioonide tüübid muutuvad:

$$\begin{aligned} \mathit{tf}_{(n,n')} &\colon \mathbb{D}_L \times \mathbb{D}_G \to \mathbb{D}_L \times \mathbb{D}_G \\ \mathit{entry}_{(n,n')} &\colon \mathbb{D}_L \times \mathbb{D}_G \to \mathcal{P}(\mathcal{F} \times \mathbb{D}_L) \\ \mathit{comb}_{(n,n')} &\colon \mathbb{D}_L \times \mathbb{D}_L \times \mathbb{D}_G \to \mathbb{D}_L \times \mathbb{D}_G \end{aligned}$$

Enne, kui me saame näitele vastavaid üleminekufunktsioone defineerida, peame vaatama, kuidas lõimede loomist käsitletakse. Funktsiooni `create` kutsumisel on kaks asjaolu, mida peame simuleerima: kutse enda otsene mõju seisundile ning loodud lõime potentsiaalne mõju globaalsele seisundile. Lõime loomisele vastava serva $(2, 3)$ jaoks peab seetõttu kasutaja defineerima kaks funktsiooni: $\mathit{tf}_{(2,3)}$ ning $\mathit{entry}_{(2,3)}$, kus esimene teostab kohalikku üleminekut ja teine peab andma kutsete hulga, mida loodud lõim võib käivitada. Näitele

saame järgmised üleminekufunktsioonid:

$$
\begin{aligned}
tf_{(1,2)}(d,\tau) &= (d, \tau[\mathtt{z} \mapsto 3]) & tf_{(5,6)}(d,\tau) &= (d[\mathtt{x} \mapsto d(\mathtt{p}) + \tau(\mathtt{z})], \tau) \\
tf_{(2,3)}(d,\tau) &= (d[\mathtt{id} \mapsto \top], \tau) & tf_{(6,7)}(d,\tau) &= (d, \tau[\mathtt{z} \mapsto d(\mathtt{x})]) \\
entry_{(2,3)}(d,\tau) &= \{\langle \mathtt{set}, \iota[\mathtt{p} \mapsto 5]\rangle\} & & \\
tf_{(3,4)}(d,\tau) &= (d,\tau) & & \\
tf_{(4,\mathtt{main})}(d,\tau) &= (d,\tau) & tf_{(7,\mathtt{inc})}(d,\tau) &= (d,\tau)
\end{aligned}
$$

Vaatame nüüd, kuidas nende abil genereeritakse kitsenduste süsteem. Kuigi igal pool on globaalne olek juures, on suurel määral kõik nagu protseduuridevahelise analüüsi korral. Me saame $\forall d \in \mathbb{D}$ korral:

$$
\begin{aligned}
(\langle n, d\rangle, \tau) &\sqsupseteq tf_{(n',n)}(\langle n', d\rangle, \tau) && \forall (n', n) \in E, \text{ kus } n' \text{ on tavaline avaldis} \\
(\langle n, d\rangle, \tau) &\sqsupseteq comb_{(n',n)}\left(\bigsqcup entry_{(n',n)}\, \langle n', d\rangle,\ \langle n', d\rangle,\ \tau\right) \\
& && \forall (n', n) \in E, \text{ kus } n' \text{ on protseduuri kutse} \\
\langle s, d\rangle &\sqsupseteq d && \forall s \in N, \text{ kus } s \text{ on algtipp} \\
(\langle f, d\rangle, \tau) &\sqsupseteq tf_{(t,f)}(\langle t, d\rangle, \tau) && \forall t \in N, \text{ kus } t \text{ on funktsiooni } f \text{ lõpptipp} \\
(\langle n, d\rangle, \tau) &\sqsupseteq tf_{(n',n)}(\langle n', d\rangle, \tau) && \forall (n', n) \in E, \text{ kus } n' \text{ on lõime loomine}
\end{aligned}
$$

Nendes võrrandites on lõimede loomisel kasutatud ainult funktsiooni tf, sest meid ei huvita lõime poolt käivitatava funktsiooni tulemusväärtus, vaid selle mõju globaalsetele muutujatele. Meie süsteem arvestab kõikide potentsiaalsete funktsioonikutsete mõjuga globaalsele olekule. Järgmises jaotises kasutame lõimede loomisel ka *entry* funktsiooni, et arvestada nende kutsetega, mis võivad programmis esineda. Vaatame konkreetsemalt näitele vastavaid kitsendusi, mida võib lihtsustada järgmiseks süsteemiks ($\forall d \in \mathbb{D}$):

$$
\begin{aligned}
\langle n_1, d\rangle &\sqsupseteq d & \langle n_5, d\rangle &\sqsupseteq d \\
\tau &\sqsupseteq \tau[\mathtt{z} \mapsto 3] & \langle n_6, d\rangle &\sqsupseteq \langle n_5, d\rangle\,[\mathtt{x} \mapsto \langle n_5, d\rangle\,(\mathtt{p}) + \tau(\mathtt{z})] \\
\langle n_3, d\rangle &\sqsupseteq \langle n_1, d\rangle\,[\mathtt{id} \mapsto \top] & \tau &\sqsupseteq \tau[\mathtt{z} \mapsto \langle n_5, d\rangle\,(\mathtt{x})] \\
\langle \mathtt{main}, d\rangle &\sqsupseteq \langle n_3, d\rangle & \langle \mathtt{inc}, d\rangle &\sqsupseteq \langle n_7, d\rangle
\end{aligned}
$$

Me oleme siiamaani defineerinud kitsendused funktsioonikutsete iga sisendoleku $d \in D_L$ korral. Kuigi kitsenduste süsteem on olnud lõpmatu, siis vastas igale süsteemi muutujale lõplik arv kitsendusi. Kuid ülalolevas süs-

teemis on lõpmatu kitsenduste pere, mille vasakul poolel on üks ja sama muutuja (globaalne olek τ):

$$\forall d \in \mathbb{D} : \tau \sqsupseteq \tau[\mathtt{z} \mapsto \langle n_5, d\rangle\,(\mathtt{x})]$$

Isegi, kui domeen oleks lõplik ja globaalne invariant oleks arvutatav, oleks see ikka liiga ebatäpne. Süsteemi täieliku invariandi arvutamisel arvestatakse ka funktsiooni `set` nende väljakutsetega, mida programmi käivitamisel ei esine. Me tahaksime iga sisendoleku $d \in \mathbb{D}$ asemel arvestada ainult olekuga $\iota[\mathtt{p} \mapsto 5]$, sest funktsiooni `set` ainus tegelik väljakutse on `set(5)`.

Süsteemi täieliku globaalse invariandi asemel tuleks arvutada osaline invariant. Viitade kaudu funktsioonikutsete korral saame alles analüüsi käigul teada, millised on funktsioonide tegelikud väljakutsed. Osalise invariandiga kitsenduste süsteemi korral peab looma kitsendused globaalsele olekule analüüsi käigus. Selleks tuleb kasutada uut algoritmi, millele me järgnevalt tähelepanu pöörame.

2.5 Osalise invariandi leidmine

Vaatame, kuidas saab eelmise peatüki lahendajat täiendada osalise invariandi arvutamiseks. Meil on $\forall d \in \mathbb{D}_L$ kohta rida kitsendusi, aga meid huvitavad ainult kitsendused, kus seisund d on saavutatav (*reachable*). Kuna meie funktsiooniviitade analüüs on selles suhtes konservatiivne, et funktsiooni väljakutsumisel arvestatakse reaalselt kutsutavate funktsioonide ülemhulgaga, siis võime vajalikud kitsendused globaalsele olekule dünaamiliselt genereerida püsipunkti arvutamisel. Osalise invariandi arvutamise teoreetilise aluse lõid Helmut Seidl, Varmo Vene ja Markus Müller-Olm [13] ning selle lähenemise korrektsust puudutavate küsimustega tuleks pöörduda nende poole.

Idee on lasta kohalike muutujate püsipunkti arvutamisel juhtida protsessi. Kui globaalne olek nende püsipunkti otsimise käigus muutub, peame süsteemi uuesti lahendama. Selliselt toimides seame globaalsele olekule põhimõtteliselt ainult neid kitsendusi, mis programmis reaalselt esinevad. Joonisel 2.5 on selle algoritmi pseudokood.

Algoritmi sisendiks on lahendatav süsteem $\mathcal{C}$, kus on vaja arvestada globaalse seisundiga. Sellest loome süsteemi $\mathcal{C}_0$, mida lahendame vana lahendajaga. Selles süsteemis kitsenduste paremad pooled enam ei anna tulemusena globaalse oleku muutusid, vaid hoiame globaalset olekut muutujas τ ning

```
fun wrap(f, σ)                          begin
begin                                     τ := ⊥;
  let ⟨d, η, s⟩ = f(σ, τ) in              repeat
    if η ⋢ τ then                           stable := true;
      stable := false;                      C0 := {x ⊒ λσ.wrap(f, σ) |
    fi;                                             x ⊒ f ∈ C};
    τ := τ ⊔ η;                           σ := Solve(C0, I);
    foreach v ∈ s do σ(v) od;             until stable
    return d;                           end
end;
```

Joonis 2.5: Lahendusalgoritmi idee

seda muudetakse globaalse imperatiivse muutujana. Kitsenduste paremate poolte tüübid on:

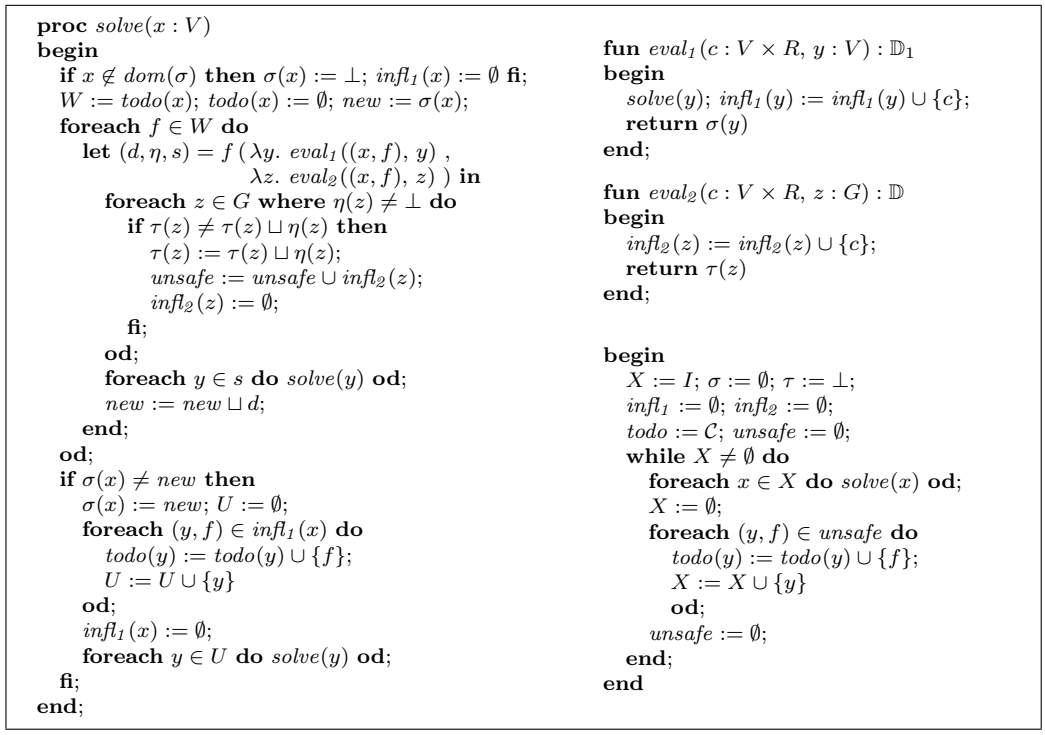
$f\colon (V \to \mathbb{D}) \to \mathbb{D})$	Süsteemi $\mathcal{C}_0$ parem pool
$f\colon (V \to \mathbb{D}_L) \times D_G \to \mathbb{D}_L \times \mathbb{D}_G \times \mathcal{P}(V)$	Süsteemi $\mathcal{C}$ parem pool

Süsteemi $\mathcal{C}$ üleminekufunktsioonid võivad anda teatud muutujate hulga, mida on vaja lahendada nende kõrvaltoime pärast. Me peame näiteks lõimede loomisel arvestama selle poolt kutsutava funktsiooni mõju globaalsele olekule. Kutsutavat funktsiooni aga ei ole süsteemi muutujate lahendamisel vaja, mistõttu eelmise jaotise kitsendustes ei olnud sellega arvestatud, aga praegu kasutame *entry* funktsiooni poolt antud muutujate hulka, et selle kutsele vastav muutuja saaks lahendusalgoritmi poolt läbitud.

Meie vana lahendusalgoritm arvutab võrrandisüsteemi muutuja väärtuse ainult siis, kui mõne üleminekufunktsiooni arvutamisel kasutatakse antud muutujat. Me võime seetõttu üleminekufunktsiooni niimoodi konstrueerida, et see lihtsalt vaatab loodud funktsioonikutsete väärtusi. Ning seda tehakse *wrap* funktsiooni lõpus.

Toodud algoritmi saab optimeerida, kui globaalsete muutujate domeen on kujul $\mathrm{Var}_G \to \mathbb{D}'$, kus $\mathbb{D}'$ on mingi võre. Sellisel juhul peab ümber arvutama ainult need süsteemi muutujad, mis tõepoolest sõltuvad muudetud globaalsest muutujast. Goblinis kasutame just sellist optimeeritud algoritmi, mis on toodud joonisel 2.6. Selle algoritmi kohta võib rohkem lugeda Trieri analüsaatorit tutvustavas artiklis [13].

Konstantanalüüsi tulemus on ikkagi liiga ebatäpne, et see meid siin hu-



Joonis 2.6: Optimeeritud lahendusalgoritm

vitaks. Nüüd pöördume andmejooksude analüüsi poole, kus meie analüüs annab ka globaalsete muutujate kohta huvitavat informatsiooni.

2.6 Andmejooksude analüüs

Kõigepealt selgitame, mis andmejooksude analüüs peaks tegema ja siis vaatame, kuidas analüüsi spetsifitseerida meie uues raamistikus. Programmi lõimed suhtlevad omavahel globaalsete muutujate kaudu. Kui need kriitilised sektsioonid ei ole keele sünkronisatsioonivahenditega kaitstud, võib tekkida andmejooks ning programm võib anda väga ootamatu tulemuse.

Definitsioon 2.1 (Andmejooks)**.** Andmejooks (*data race*) on olukord, kus rohkem kui üks lõim pöördub sama jagatud muutuja poole, ilma et pöörduste järjekorrale oleks seatud mõistlikke kitsendusi. Seetõttu *võib* käivatamisel tekkida olukord, kus pöörduste järjekord ei ole korrektne (ei vasta programmeerija ootustele).

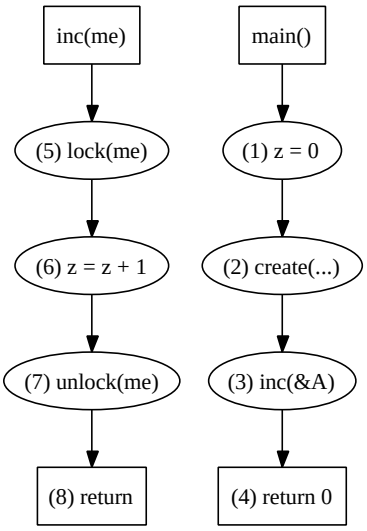
Selles jaotises tutvustame üht väga lihtsat andmejooksude analüüsi, mis

tuvastab üht väga ohtlikku liiki andmejooksu. Kui kasutatakse kõrgtasemelisi programmeerimiskeeli, siis on peamine oht see, et programmeerija võib eeldada, et mõni operatsioon on atomaarne, kuigi tegelikult koosneb see mitmest masinkoodi käsust ning nende vahel võib tekkida kontekstivahetus. Näiteks võiks eeldada, et Java täisarvude sisend/väljund on atomaarne, kuid tegelikult võib isegi 4-baidilise täisarvu väljastamine pooleli jääda. Operatsiooni atomaarsuse kindlustamiseks on keeltes sünkronisatsioonivahendid, mis piiravad koodi põimumist. C keeles kasutatakse selleks lukke ehk välistajaid (*mutex — mutually exclusive*). Programmi kood, mis on sama välistajaga lukustatud, ei saa omavahel põimuda.

```
int z;
mutex A, B;

void inc(mutex *me) {
    lock(me);
    z++;
    unlock(me);
}

int main() {
    int id;
    z = 0;
    create(&id, inc, &A);
    inc(&A);
    return 0;
}
```



Joonis 2.7: Mitmelõimelise programmi näide

Joonisel 2.7 on klassikaline näide kriitilise sektsiooniga programmist. Mõlemad lõimed suurendavad muutujat `z` ühe võrra, aga pöördused muutuja poole kaitstakse ühise lukuga `A`. Kui nad ei oleks kaitstud ühise lukuga, siis võib programmi käivitamisel kontekstivahetus tekkida just siis, kui üks lõim on jõudnud muutuja `z` väärtuse lugeda protsessori registrisse ning seal suurendada, kuid ei ole jõudnud tulemust mälusse salvestada enne, kui järgmine lõim muutuja väärtuse loeb. Seda olukorda illustreerib järgmine täitmisjärjekord:

Lõim 1:	load(z)	$\mathtt{z} = 0,\ reg_1 = 0$
Lõim 1:	inc	$\mathtt{z} = 0,\ reg_1 = 1$
Lõim 2:	load(z)	$\mathtt{z} = 0,\ reg_2 = 0$
Lõim 1:	store(z)	$\mathtt{z} = 1,\ reg_1 = 1$
Lõim 2:	inc	$\mathtt{z} = 1,\ reg_2 = 1$
Lõim 2:	store(z)	$\mathtt{z} = 1,\ reg_2 = 1$

Selliste sünkroniseerimisvigade tuvastamiseks on meil vaja teada, millised lukud on lukustatud, kui globaalse muutuja poole pöördutakse. Analüüs peab iga globaalse muutuja $g \in \mathrm{Var}_G$ korral leidma lukkude hulga, mis on alati lukustatud, kui g poole pöördutakse. See on funktsioon $\mathrm{Var}_G \to \mathcal{P}(\mathrm{Addr})$. Selleks peame aga kindlaks tegema hoitud lukud iga programmi punkti N kohta. See on kohalik seisund, sest kriitilise sektsiooni sisenemisel kuulub lukk just analüüsitavale lõimele.

$$\mathbb{D}_L = \mathcal{P}(\mathrm{Addr}) \qquad\qquad \mathbb{D}_G = \mathrm{Var}_G \to \mathcal{P}(\mathrm{Addr})$$

Näitele vastavad järgmised üleminekufunktsioonid:

$$\begin{aligned}
tf_{(1,2)}(d,\tau) &= (d, \tau[\mathtt{z} \mapsto d]) \\
entry_{(2,3)}(d,\tau) &= \{(\mathtt{inc}, d)\} \\
comb_{(2,3)}(d_c, d_l, \tau) &= (d_c, \tau) \\
entry_{(3,4)}(d,\tau) &= \{(\mathtt{inc}, d)\} \\
tf_{(3,4)}(d,\tau) &= (d, \tau) \\
tf_{(5,6)}(d,\tau) &= (d \cup \{d_{cp}(\mathtt{me})\}, \tau) \\
tf_{(6,7)}(d,\tau) &= (d, \tau[\mathtt{z} \mapsto d]) \\
tf_{(7,8)}(d,\tau) &= (d \setminus \{d_{cp}(\mathtt{me})\}, \tau)
\end{aligned}$$

Analüüs kasutab funktsiooni `inc` parameetri `me` väärtuse leidmiseks informatsiooni, mida annab konstantanalüüs. Seda on üleval tähistatud d_{cp} ja eeldame, et see informatsioon on kuidagi kättesaadav. Kuidas seda kättesaadavaks teha, on omaette raske probleem, mida ei ole veel Goblinis lahendatud. Goblin sooritab kombineeritud analüüsi, kus domeeniks on paar, mille esimene komponent on Kildalli domeeni element ja teine komponent on lukkude hulk ja iga üleminekufunktsioon tegeleb mõlemaga. Tulevikus tahaks jõuda modulaarsema lahenduseni.

Kitsendusi genereeritakse analoogiliselt eelnevaga. Kuid siin tuleb välja

üks probleem globaalse invariandi leidmisega. Analüüs võiks üritada kõikide põimumiste asemel olukorda natuke täpsemini analüüsida. Kuna esimest lõime luuakse alles teisel tipul, siis on selge, et sellele eelnevad käsud ei põimu uue lõime käskudega. Sellest võib järeldada, et tegelikult esimene tipp ei ohusta teisi. Samamoodi seavad ka sünkronisatsioonipunktid täitmisjärjekorrale kitsendusi. Ideaalis võiks juhtimisvoograafi kõik tippe osaliselt järjestada nii, et $n_1 < n_2$ tähendab, et alati jõutakse tipuni n_1 enne tippu n_2.

Meie lähenemine on üsna lihtne, meid huvitab ainult esimese lõime tekkimine, kuna enne seda toimub muutujate algväärtustamine, aga pärast me eeldame, et kõik toimub väga paralleelselt. Kui programm ainult aeg-ajalt loob ühe lõime tausttegevuse jaoks, siis oleme üsna ebatäpsed. Kui aga programmi lõimed on suhteliselt iseseisvad (näiteks serverlõimed), siis võib meil hästi minna. Antud näite jaoks on meie lähenemine muidugi piisavalt täpne.

Selle analüüsi tulemuse leidmiseks peab aga lugeja Goblini poole pöörduma ja ise katsetama, sest nüüd ongi lõpuks aeg tutvuda Goblini endaga.

Peatükk 3

Kraasija Goblin

Selles peatükis vaatame konkreetsemalt, kuidas mitmelõimeline analüüs on Goblinis realiseeritud. Analüsaatori kirjutamine, mis avastaks reaalsetes programmides vigu, on väga mahukas projekt. Selle implementatsioon on töö kirjutamise ajal jõudnud nii kaugele, et põhiline raamistik on valmis ning nüüd võib hakata uusi ideid katsetama ja ründama lahtisi probleeme programmianalüüsi valdkonnas. Autor loodab ka leida teisi inimesi, kes tahaksid programmianalüüsiga tegeleda.

Esialgne ülesanne oli täiendada Trieri analüsaatorit [13], kuid selle koodibaasi peal töötamine osutus autorile liiga raskeks. Selle asemel otsustati analüsaator algusest lõpuni ümber kirjutada. Sellist ettevõtmist õigustas võimalus kasutada uuemaid ja paremaid arendusvahendeid. Goblin on kirjutatud programmianalüüsi valdkonnas väga populaarseks saanud objektorienteeritud funktsionaalses keeles *Objective Caml*.

Goblin koosneb kolmest põhikomponendist: kasutajaliides, analüsaator ja C-keele parsija. Joonisel 3.1 on kujutatud komponentide suhtlus programmi analüüsimisel. C-keelte parsimiseks kasutame Berkeley Ülikoolis valminud C-keele analüüsimise raamistikku CIL[6], mis oluliselt lihtsustab juhtimisvoograafide loomist. Goblin teostab juhtimisvoograafi peal analüüsi ning tulemusi saab vaadata Eclipse'is.

Joonis 3.1: Goblini komponendid

Goblini sihtrühm jaguneb kolmeks, sõltuvalt kasutamise vajadusest. Toome mõned kasutuslood välja ning ülejäänud peatükk koosneb Goblini kirjeldamisest nende erinevate kasutajate perspektiividest.

1. Goblinit saab kasutada lihtsalt C-programmide kraasimiseks, kasutades Goblinis eeldefineeritud analüüse. Esimeses jaotises vaatame, kuidas saab analüüside tulemusi vaadata Eclipse'is.

2. Väga paljud nõuded süsteemtarkvarale on lihtsasti kontrollitavad programmianalüüsiga. Teises jaotises vaatame, kuidas kasutaja võib ise defineerida vajaliku analüüsi. Goblin loob selle spetsifikatsiooni põhjal mitmelõimeliste programmide analüsaatori, mis kontrollib just seda omadust.

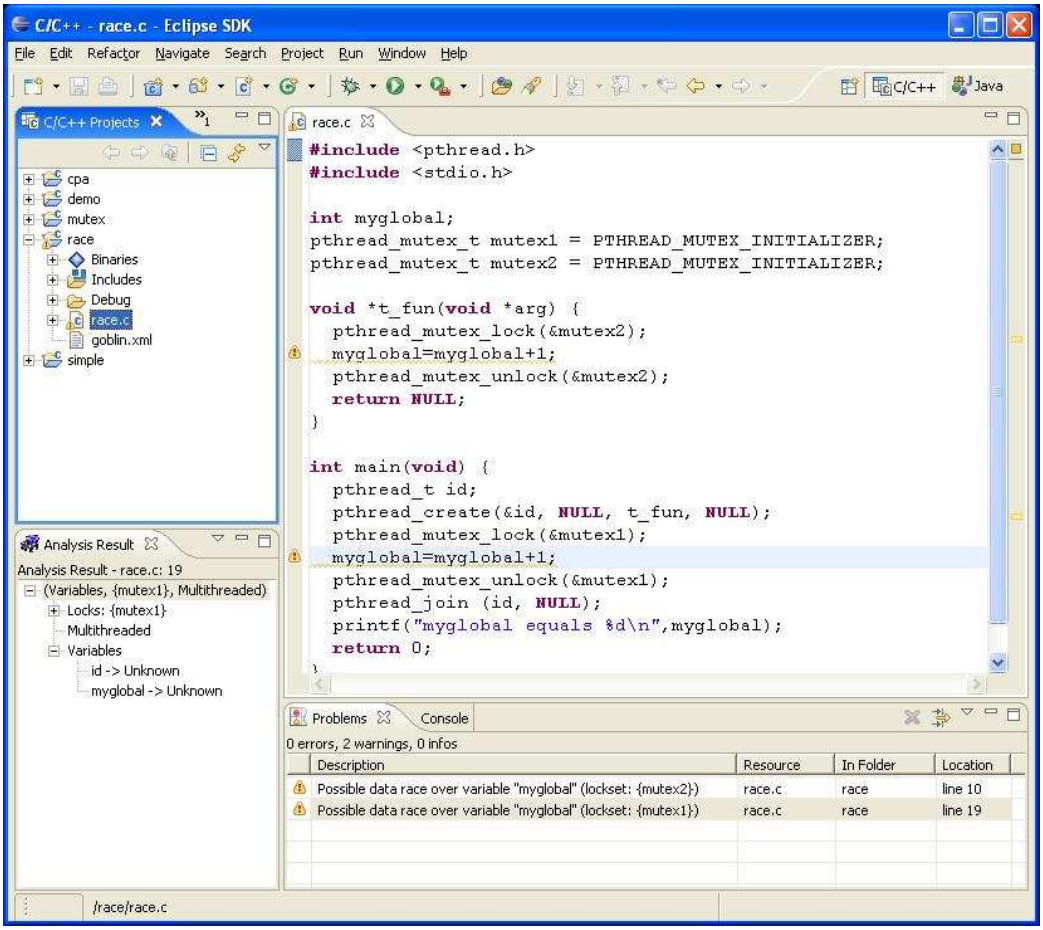
3. Goblin ei ole viimane sõna programmianalüüsis. Väga huvitavad probleemid on selles valdkonnas veel lahendamata. Ideede katsetamiseks on vaja vabavaralist analüüsi raamistikku: Goblin. Selle peatüki lõpus vaatame Goblini struktuuri ning tutvume lähitulevikus plaanitavate töödega.

Projekti veebiaadress on `http://www.ut.ee/~vesal/goblin`. Käesolev töö annab vaid üldpildi süsteemist ning põhjendab süsteemis tehtud valikuid. Kasutusjuhend ja muu dokumentatsioon on kättesaadav projekti leheküljelt.

3.1 Programmide analüüsimine

Goblini kasutamine programmianalüsaatorina on lihtne, sest see integreerub hästi arenduskeskkondadega. Hetkel on analüüsi tulemuste kuvamiseks võimalik kasutada Eclipse'i. Kui kasutaja on Goblini installeerinud ning oskab Eclipse'iga oma C-programme lahti teha, siis ta saab analüüse mugavalt käivitada ning tulemusi vaadata.

Tuletame meelde, et analüüs peab iga juhtimisvoograafi tipu $n \in N$ kohta leidma programmi seisundi $d \in \mathbb{D}$. Eclipse'is redigeerib aga kasutaja programmi lähtekoodi ning juhtimisvoograafi ei ole kuskil näha. Analüüsi tulemuse seostame seetõttu programmi ridadega ning näitame analüüsi seisundit enne vastava koodirea täitmist. Globaalne olek seostatakse vastava globaalse muutuja deklaratsiooniga. Joonisel 3.2 on ekraanipilt ühe lihtsa programmi analüüsist. Kasutaja valitud programmirea seisundit näidatakse all vasakul.



Joonis 3.2: Goblini kasutajaliides

Analüüs võib lisaks seisundi leidmisele ka hoiatusi tõstatada. Eclipse'il on olemas probleemide näitamiseks ja töötlemiseks tähistajad (*markers*), mis ilmuvad vigaste koodiridade juurde. Hoiatusi näidatakse ka eraldi probleemivaates, kus on nimekiri kõikidest kompileerimisvigadest ja muudest probleemidest. Hoiatuse peal klõpsutamine suunab probleemsele koodireale.

Eclipse on väga võimas arenduskeskkond, kuid peab mainima, et see on eelkõige Java arendusvahend ega ole väga levinud C/C++ arenduskeskkonnana, kuigi seda poolt arendatakse siiski väga aktiivselt. Eclipse'i keskkonna C-keele tugi kasutab kompilaatorina gcc-d ja silujana gdb-d. See on juba praegu üks parimaid keskkondi C-programmide graafiliseks silumiseks. Kuid väga paljud C-programmeerijad võivad jätta Goblini kasutamata, kuna nad ei tööta Eclipse'is.

Kui ainus eesmärk on saada hoiatusi, siis on olemas teine võimalus, kus kasutaja ei pea oma arenduskeskkonda muutma. Goblin võib käituda[1] täpselt nagu gcc, s.t kõiki käsureaparameetreid tõlgendatakse mõistlikult ning väljund jälgib gcc hoiatuste formaati. Kasutaja peab analüüsi teostamiseks oma *make*-failis kompilaatorina ajutiselt kasutama Goblinit. Suvaline arenduskeskkond (Eclipse kaasaarvatud) saab siis hoiatusi näidata.

Analüüside arendamisel on aga väga kasulik, et kiiresti saab analüüse käivitada ning vaadata tulemust iga programmirea kohta. Selle jaoks on Eclipse'i kasutajaliides ideaalne. Kuna kasutajaliides on põhiprogrammist eraldatud, siis ei pea kasutajaliidest uuesti käivitama, kui analüüsi on muudetud ja Goblin ümber kompileeritud. Kasutaja saab analüüse väga kiiresti arendada ja katsetada.

3.2 Analüüside loomine

Analüüsi spetsifikatsioon koosneb domeeni kirjeldusest ja üleminekufunktsioonide definitsioonidest. Analüüside generaator peab võimaldama kasutajal võimalikult mugavalt neid asju defineerida. Domeeni kirjelduse põhjal loob Goblin vajalikud andmestruktuurid domeeni elementide hoidmiseks ja töötlemiseks. Kasutaja kirjutab domeeni definitsiooni Goblini domeenikeeles DDL, mis on üsna sarnane matemaatilistele definitsioonidele. Keeles on kõigepealt mõned eeldefineeritud baasdomeenid, näiteks täisarvud ja programmis esinevate muutujate hulk. DDL sisaldab ka vahendeid, millega saab baasdomeenidest moodustada keerulisemad domeene. Töös käsitletud domeene saab DDLis defineerida järgmiselt (paremal):

$$\begin{aligned} \mathrm{Val} &= \mathit{flat}(\mathbb{Z}) & \texttt{Val} &= \texttt{Flat}\,(\texttt{Integers}) \\ \mathrm{Kildall} &= \mathrm{Var} \to \mathrm{Val} & \texttt{Kildall} &= \texttt{Map}\,(\texttt{Variables})\,(\texttt{Val}) \\ \mathrm{Mutex} &= \mathcal{P}(\mathrm{Addr}) & \texttt{Mutex} &= \texttt{Set}\,(\texttt{Addresses}) \end{aligned}$$

Kui kasutaja on domeeni defineerinud, siis võib ta defineerida üleminekufunktsioonid, mis teisendavad domeeni objekte. Me oleme töö teoreetilises osas üleminekufunktsioonid defineerinud iga konkreetse juhtimisvoograafi serva jaoks, kuid analüüs tuleb tegelikult spetsifitseerida iga potentsiaalselt võimaliku avaldise jaoks. Meie üleminekufunktsioonide tüübid on mitmelõi-

[1] Selle funktsionaalsuse toetamine on töö kirjutamise ajal veel pooleli.

melise analüüsi korral järgmised:

$$\begin{aligned} tf &: E \to (\mathbb{D}_L \times \mathbb{D}_G \to \mathbb{D}_L \times \mathbb{D}_G) \\ entry &: E \to (\mathbb{D}_L \times \mathbb{D}_G \to \mathcal{P}(\mathcal{F} \times \mathbb{D}_L)) \\ comb &: E \to (\mathbb{D}_L \times \mathbb{D}_L \times \mathbb{D}_G \to \mathbb{D}_L \times \mathbb{D}_G) \end{aligned}$$

Nad peavad iga võimaliku juhtimisvoograafi serva e korral andma vajaliku üleminekufunktsiooni. Servadeks on nüüd lihtsustatud C-keele avaldised. Goblini koodis on serva tüüp järgmine:

```
type edge = Assign of lval * exp
          | Proc of lval option * exp * exp list
          | Test of exp * bool
          | Ret of exp option
          | Entry of fundec
```

Assign servad vastavad lihtsatele omistusavaldistele kujul `x = 7`. Omistuse vasak pool on tüüpi `lval`, mis on CIL tüüp muutuja või aadressavaldise jaoks. Parema poole tüüp `exp` on puhas avaldis, s.t ilma kõrvalefektideta.

Proc on funktsioonikutsete jaoks. Kui kutse tulemus omistatakse muutujas, näiteks `x = f(3)`, siis on kolmiku esimene element omistuse vasak pool. Teine element on avaldis, mille tulemus on kutsutava funktsiooni aadress. Kolmas element on argumentide list.

Test vastab juhtimisvoograafi hargnemistele. See on paar, mis koosneb tingimusavaldisest ning tõeväärtusest.

Ret on eriline serv funktsiooni naasmisavaldise ja funktsiooni vahel, mille me lisasime eelmises peatükis selleks, et funktsiooni tulemusväärtustega arvestada.

Entry on sümmeetriline Ret-servaga ning seostab funktsiooni ja tema esimese avaldise. See sisaldab ka informatsiooni funktsiooni kohta, mida võib näiteks kasutada lokaalsete muutujate initsialiseerimiseks.

Üleminekufunktsioonide defineerimise keel TFDL on samuti *O'Caml*, mis saab selle ülesandega suurepäraselt hakkama. Struktuursete andmete teisendamine on näidiste sobitamisega (*pattern matching*) väga mugav. Joonisel 3.3

```
let tf edge st =
  match edge with
    | CFG.Assign (lval, exp) ->
        let x = getvar lval in
        let value = eval st exp in
          LDom.add st x value
    | CFG.Test (exp, tv) ->
        let inv = invariant exp tv in
          LDom.meet st inv
    | _ -> st
```

Joonis 3.3: TDFL näidiskood

on katkend konstantanalüüsi koodist. Ruumilistel põhjustel on globaalne olek siit kustutatud ja suur hulk abifunktsioone on defineerimata. Oluline on lihtsalt tähele panna, kui mugav on selles keeles üleminekufunktsioone kirja panna.

Keerulise C-programmi analüüsimiseks peab kasutaja ainult defineerima üleminekufunktsioonid mõnede lihtsate avaldiste jaoks ning Goblin teeb ülejäänud töö ise ära.

3.3 Süsteemi kirjeldus

Tutvume põgusalt Goblini ülesehitusega. Eesmärk on anda ülevaatlik pilt Goblini moodulitest, et kui lugejal on huvi ise programmianalüüsiga tegeleda ja soovib Goblinit edasi arendada, siis teaks, kust alustada. Vaatame kõigepealt Goblinis kasutatud komponente ja seejärel Goblini analüsaatorkomponenti ennast.

Selle töö näidisprogrammid on olnud väga lihtsad ning nende juhtimisvoograafide loomine on nii triviaalne, et me ei ole sellele tähelepanu pööranud. Tegeliku C programmi analüüsimine on aga üsna keeruline ning vajab palju eeltööd. CIL (C Intermediate Language) on sisuliselt selge struktuuriga C alamkeel, millega on palju mugavam töötada. Joonisel 3.4 on väike näide sellest, kuidas programmi lihtsustatakse. CILi kasutatakse peamiselt praktiliste analüüside ja programmiteisenduste tegemiseks. Selle kõige edukam rakendus on `ccured` [7], mis teisendab C-programme selliselt, et kõik mälupöördused on turvalised, lisades nii vähe täitmisaegseid teste kui võimalik.

```
int main() {
    int x, y, z;
    return &(x ? y : z) -
        &(x++, x);
}
```

```
int main() {
    int x,y,z, *tmp;
    if (x) {
        tmp = &y;
    } else {
        tmp = &z;
    }
    x = x + 1;
    return (tmp - &x);
}
```

Joonis 3.4: Näide CILi teisendustest

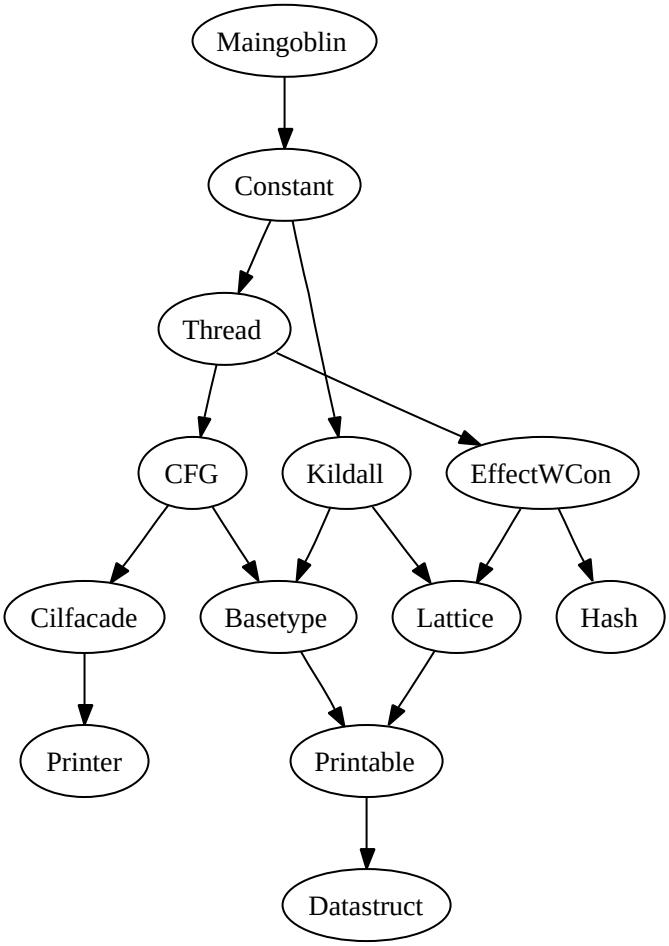
Meie soovime aga teostada põhjalikke andmevooanalüüse ja tahame programmivigu avastada. See ei ole CILi kõige tugevam külg, kus peamine rõhk on programmiteisendustel. Kuigi CILis on olemas ka püsipunkti algoritm, siis on üsna raske sellega teha omi protseduuridevahelisi analüüse. Seetõttu kasutame praegusel hetkel CILi rohkem süntaksanalüsaatorina, mis teeb C-programmidest abstraktse süntaksipuu.

Vaatame nüüd Goblinit ennast. Joonisel 3.5 on kujutatud Goblini moodulid, mis koos teostavad kõige lihtsamat mitmelõimelist konstantanalüüsi. Nooled tähistavad staatilisi sõltuvusi moodulite vahel. Vaatame lähemalt, mida iga moodul teeb:

Maingoblin on projekti peafail, millest tehakse Goblini käivitatav binaarfail. See peab käsureaparameetreid interpreteerima ja õiged C failid sisse lugema, sooritama eeltöötlust, rakendama analüüsi ning väljastama tulemuse. Eeltöötlust tehakse põhiliselt mooduli CilFacade funktsioonidega ning programmi analüüsimiseks kasutatakse antud juhul funktsioone moodulist Constant.

Constant sisaldab konstantanalüüsi spetsifikatsiooni alammoodulis `Spec`, mille domeen on defineeritud moodulis Kildall. Faili viimase käsuna defineeritakse alammoodul `Simple = Thread.Forward(Spec)`, mis kasutab moodulis Thread defineeritud analüüsimise raamistikku, et spetsifikatsioonist teha töötav analüüs. Selle tulemusena tekib muuhulgas funktsioon `Constant.Simple.analyze`, millega saab faile analüüsida.

Thread defineerib mitmelõimelise protseduuridevahelise analüüsi raamisti-



Joonis 3.5: Goblini struktuur

ku, mis on parametriseeritud analüüsi spetsifikatsiooni järgi. Andes moodulile spetsifikatsiooni, kus on üleminekufunktsioonid, genereerib ta vajaliku kitsenduste süsteemi ja kasutab selle lahendamiseks moodulit EffectWCon. Ta kasutab kitsenduste loomiseks juhtimisvoograafi, mis luuakse moodulis CFG.

CFG sisaldab juhtimisvoograafi tüübi definitsioone ning funktsioone, mis CILi abstraktsest süntaksipuust loovad meie jaoks sobiva graafi. Kuigi CIL teeb ise ka juhtimisvoograafe, on need abstraktse süntaksipuu sisse ehitatud. Selle loomine on nende jaoks rohkem programmiteisendus, kus näiteks `case`-lause asemel on kasutatud siirdekäske.

CilFacade võtab ühte moodulisse kokku kõik CILi funktsioonid, mida on vaja rakendada programmi analüüsimisel. Me kasutame CILi eelkõige süntaksipuu loomiseks, aga selles distributsioonis on lisaks väga palju kasulikke teisendusi peidetud, mida CilFacade funktsioonidega saab kasutada.

Printer sisaldab erilisi süntaksipuu printereid. See on eriti kasulik, et näha teisendusi, mida CIL on teinud lähtekoodiga enne kui see analüsaatorini on jõudnud.

EffectWCon ja Hash on kitsenduste süsteemi lahendaja moodulid. Moodulis Hash on lahendusalgoritmi abiandmestruktuurid. Lahendaja võtab sisendiks kitsenduste süsteemi, mis on antud funktsionaalsel kujul.

Kildall sisaldab domeeni definitsiooni. See on võrdlemisi lühike moodul, sest domeen on kirjeldatud kasutades Goblini domeeni defineerimise keelt DDL, mis on järgmiste moodulitega implementeeritud.

Basetype sisaldab DDLi baasdomeenide definitsioone. Domeen on moodul, millel on võre operatsioonide ja ilutrüki tugi. Lihtsamad baasdomeenid on mingi standardtüübi (näiteks täisarvude) triviaalsete definitsioonidega moodulid. Põhiline töö toimub siis, kui nendele rakendatakse DDLi funktsioone, et saada keerulisemad domeene.

Lattice ja järgmise kahe mooduliga defineeritakse domeenikeele DDL teisendajad, millega saab lihtsatest domeenidest keerulisemad. Tööjaotus on nende kolme mooduli vahel vertikaalne, kõikides moodulites on samade teisendajate definitsioonid. Moodul Lattice on kõige kõrgem kiht, kus lisatakse domeeniteisendajatele võre tasemel operatsioonid.

Printable lisab domeeniteisendajatele ilutrüki ja XMLi väljundi funktsionaalsuse. Eclipse'i kasutajaliidese komponent oskab siis seda ilusasti ekraanil näidata.

Datastruct on DDL teisendajate kõige madalam kiht, kus on defineeritud domeeni elementide hoidmise andmestruktuurid. Näiteks domeeniteisendaja `Map` korral on selleks paisktabel.

3.4 Planeeritavad täiustused

Programmianalüüs on noor ja huvitav valdkond, kus on palju lahtisi probleeme. Goblinis on töö kirjutamise ajal mõned praktilised probleemid veel lahendamata, aga põhimõtteliselt oleme jõudnud nii kaugele, et võib hakata selliseid probleeme uurima. Selles jaotises toome välja need kohad, kus Goblinit oleks vaja edasi arendada. Siin toodud probleemid on juba aktiivse uurimistöö teemad ning nende probleemide lahendamine vajab ka vastava teooria väljatöötamist.

Üks väga raske probleem programmianalüüsis on dünaamiliste andmestruktuuride analüüs ning sellega seoses kuhja analüüsimine. Kuhjal loodud objektid on dünaamiliselt allokeeritud, mistõttu peaks olema arusaadav, et staatiliselt on väga raske neid analüüsida. Kuhja analüüsmine on siiski väga oluline, aga Goblin käsitleb hetkel tervet kuhja ühe abstraktse väärtusena. Sisuliselt tähendab see seda, et kuhjas olevate objektide kohta ei ole midagi teada. Seda saab paremini teha, aga kuidas täpselt, on veel lahtine. Üks väga lootustandev lähenemine on O'Hearn ja Pymi separatsiooniloogika [10], mida John Reynolds on rakendanud kuhja analüüsimiseks [11].

Mitmelõimelise analüüsi korral on üheks keskseks probleemiks arusaamine, millal mõni muutuja on jagatud. Goblin eeldab praegu, et kõik globaalsed muutujad on jagatud, kuid see on ilmselt väga ebatäpne. Sellega natuke seotud probleem on lõimede põimumiste analüüs ning jällegi on Goblin äärmiselt ebatäpne, kuna eeldame, et pärast esimese lõime loomist võivad kõik lõimed omavahel põimuda ning me ei arvesta sünkronisatsioonipunktidega.

Võib-olla kõige huvitavam teema on aga analüüside kombineerimine. Goblinis on andmejooksude analüüs sisuliselt konstantanalüüsi sisse programmeeritud. Oleme alustanud tööd analüüside kombinaatorite suunas, kuid kahjuks on analüüside kombineerimine palju raskem, kui esialgu võib tunduda ning see töö on hetkel veel pooleli. Kasutajaanalüüside lisamiseks Goblinile on aga hädavajalik, et raamistik võimaldaks analüüside kombineerimist.

Lisaks ülalmainitud huvitavatele probleemidele on Goblinil hetkel teatud praktilisi puudujääke. Kuna Goblin kasutab protseduuridevahelise analüüsi jaoks nn funktsionaalset lähenemist, siis rekursiivse programmi korral võib analüüs jääda lõpmatusse tsüklisse. See on üsna raske probleem, aga selle peab muidugi lahendama. Võib ka teisi analüüsiraamistike ja algoritme lisada protseduuridevahelise analüüsi teostamiseks.

Lõpuks peab tunnistama, et kõige tähtsam töö on analüüside täpsuse pa-

randamine. Tuleb vaadata reaalsetes programmides kasutatud idioome, millega analüsaator ei saa hakkama ning lisada nende konstruktsioonide jaoks spetsiaalne käsitlus. Goblini projekti põhieesmärk on reaalsetes programmides reaalseid andmejookse tuvastada, kuid selle saavutamiseni on veel jäänud üsna palju tööd.

Kokkuvõte

Ajal, mil tarkvaratööstuses toimub üleminek järjestikuselt paralleelsele programmeerimisele, peavad uued keeled ja arendusvahendid tegelema paralleelse käivitamisega kaasnevate probleemidega. Antud töös esitletakse andmejooksude tuvastamise vahendit Posix-lõimedega C-programmides.

Goblin on andmevooanalüsaator, mis põhineb abstraktse interpretatsiooni teoorial [1]. Töö esimene peatükk tutvustab protseduurisisese analüüsi põhikontseptsioone, alates juhtimisvoograafidest kuni püsipunkti arvutamiseni. Lisaks kaetakse arusaamiseks vajalik võreteooria. Peatükk lõpeb ülevaatega Fechti ja Seidli loodud kitsenduste süsteemi lahendajast [3].

Teises peatükis minnakse protseduurisiseselt analüüsilt edasi protseduuridevahelisele mitmelõimelisele analüüsile. Mitmelõimeliste programmide analüüsimise raskus seisneb lõimede kõikvõimalike põimumistega arvestamises. Probleemi lahenduse aluseks on osalise globaalse invariandi teooria, mis arendati algselt Trieri andmejooksude analüsaatori jaoks [13].

Kolmandas peatükis antakse ülevaade analüsaatorist Goblin, mis kasutab süntaksanalüüsi teostamiseks CILi raamistikku [6]. Analüüsi tulemused kuvatakse Eclipse'is. Peatükk sisaldab ülevaadet kasutajaliidesest, lühikirjeldust analüüside spetsifitseerimisest keeles *Objective Caml* ja pilguheitu Goblini analüüsikomponendi sisse. Peatükk lõpeb aruteluga tulevikus planeeritavate täiustuste üle.

Linting multi-threaded C programs with the Goblin

Master of Science Thesis

Vesal Vojdani

Abstract

As the software industry is under a fundamental paradigm shift from sequential to parallel programming, new languages and software tools are required to deal with the complexity of parallel execution. This thesis presents a tool for detecting data races in Posix threaded C programs.

The Goblin is a data flow analyzer based on the theory of abstract interpretation [1]. The first chapter introduces the basic concepts in intra-procedural analysis. It moves gradually from control flow graphs to fix point equations covering some lattice theory on the way. The chapter ends with a description of a modern constraint solver proposed by Fecht and Seidl [3].

The second chapter takes the step from intra-procedural analysis to inter-procedural multi-threaded analysis. The difficulty with multi-threaded programs is to deal with all the possible interleaving of multiple threads. The Goblin uses a solution to this problem based on the theory of partial global invariants developed originally for the Trier Data Race analyzer [13].

The third and final chapter is devoted to the Goblin analyzer itself, which is built on the CIL framework [6] and has a user interface in Eclipse. The chapter contains an overview of the user interface, a brief description of how analyzes are specified in Objective Caml, and a look at the internals of the Goblin's analysis component. The chapter concludes the thesis with a discussion of future work.

Kirjandus

[1] Patrick Cousot ja Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, lk. 238–252. ACM Press, 1977.

[2] Patrick Cousot ja Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. M. Bruynooghe ja M. Wirsing (koostajad). *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*, Lecture Notes in Computer Science, 631. köide, lk. 269–295. Springer-Verlag, 1992.

[3] C. Fecht ja H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming (SCP)*, 35(2):137–161, 1999.

[4] Seth Hallem, Benjamin Chelf, Yichen Xie ja Dawson Engler. A system and language for building system-specific, static analyses. *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, lk. 69–82. ACM Press, 2002.

[5] Gary A. Kildall. A unified approach to global program optimization. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, lk. 194–206. ACM Press, 1973.

[6] George C. Necula, Scott McPeak, S. P. Rahul ja Westley Weimer. Cil: An infrastructure for C program analysis and transformation. *International Conference on Compiler Construction*, lk. 213–228, aprill, 2002.

[7] George C. Necula, Scott McPeak ja Westley Weimer. CCured: type-safe retrofitting of legacy code. *Symposium on Principles of Programming Languages*, lk. 128–139, 2002.

[8] F. Nielson, H. Riis Nielson ja C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[9] H. Riis Nielson ja F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[10] Peter W. O'Hearn ja David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[11] John Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millenial Perspectives in Computer Science, Proceedings of the Oxford–Microsoft Symposium in Honour of Sir Tony Hoare*, 1999.

[12] Research Triangle Institute (RTI). The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), 2002.

[13] 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.