TARTU ÜLIKOOL
Loodus- ja täppisteaduste valdkond
Arvutiteaduse instituut
Informaatika õppekava

Karl Marten Mägi

Modulaarne staatiline programmianalüüs

Bakalaureusetöö (9 EAP)

Juhendajad: Kalmer Apinis, PhD
Vesal Vojdani, PhD

Tartu 2021

Modulaarne staatiline programmianalüüs

Lühikokkuvõte:
Staatiline programmianalüüs on viis uurida programmi käitumist ja omadusi. Modulaarne analüüs viitab programmi osade eraldi analüüsimisele ja saadud tulemuste kombineerimisele. See aitab mahukate analüüside korral aega säästa, sest on võimalik tükeldatud analüüsi osi taaskasutada.

Töö käigus antakse eestikeelne ülevaade modulaarsest analüüsist, selle impleementatsioonist programmianalüüsi raamistikus Põder ning täiendatakse raamistiku Põder. Esmalt parandati vigu, mis tekkisid tulemuste laadimisel. Lisati laetavate tulemuste verifitseerimine, et nendes saaks kindlam olla, ning versioonihaldus, et oleks võimalik ohutult laadida tulemusi kasutaja arvuti välistest allikatest. Tulemusena on raamistik Põder kasutajale mugavam ja tulemuste salvestamine ning laadimine on töökindlam.

Võtmesõnad:
Modulaarne analüüs, staatiline analüüs, võred, Põder

CERCS: P175 Informaatika, süsteemiteooria

Modular static program analysis

Abstract:
Static program analysis is a method for studying a program's behaviour and characteristics. Modular analysis means analysing program parts separately and then combining the received results. It saves time when analysing lengthy programs because one can reuse respective analysis results.

This thesis focuses on giving an Estonian overview of modular analysis, documenting its implementation in program analysis framework Põder and improving the framework Põder. First, various issues with loading saved results were fixed. To improve confidence in loaded results, a verification process was added to the beginning of an analysis. Additionally, version support was added to Põder, so it would be safer to load results from online sources where versions may differ. As a result Põder is more comfortable to use and loading results is reliable.

Keywords:
Modular analysis, static analysis, lattices, Põder

CERCS: P175 Informatics, systems theory

Sisukord

1 Sissejuhatus

Programmianalüüs on automaatne programmide käitumise ja sellega seotud omaduste analüüs. Staatilisus viitab sellele, et analüüs teostatakse ilma programmi käivitamata. Analüüsid võimaldavad kontrollida, kas programm töötab vastavalt tingimustele, mis omakorda aitab programmist vigasid leida ja osaliselt korrektsust tõestada. Enamasti ei ole nendele küsimustele täpseid ja kindlaid vastuseid võimalik anda, sest erinevaid võimalike sisendeid on liiga palju. Seetõttu otsitakse ligikaudseid vastuseid abstraktsel tõlgendusel (interpretatsioonil) põhineva analüüsi abil [1].

Modulaarne analüüs võimaldab erinevaid programmi fragmente eraldi analüüsida. Iga analüüsi osa sisaldab ainult seda infot, mis tuleneb vastavast programmi fragmendist. See võimaldab osasid salvestada ning teiste analüüside käigus ka taaskasutada. Kui näiteks standardteekide analüüse salvestada, on võimalik neid tulevaste analüüside käigus laadida ja uuesti kasutada. Täpsemad analüüsid on enamasti ajakulukad, seega on taaskasutamisega säästetud aeg täpsemateks tulemusteks väga kasulik [2].

Lõputöö üheks eesmärgiks on anda eestikeelne teoreetiline ülevaade protseduuripõhisest modulaarsest analüüsist. Lisaks teoreetilisele ülevaatele on siht dokumenteerida programmianalüüsi raamistiku Põderi modulaarset analüüsi. Praktilise töö käigus on plaanis parandada Põderi raamistikku, et programmi osade salvestamine ja laadimine töötaks ning ei põhjustaks vigu teistes raamistiku analüüsides.

Teine peatükk kirjeldab vajalikku teoreetilist tausta, et mõista järgnevat peatükki, mis keskendub konkreetselt protseduuride analüüsile ja selle kasudele. Neljandast peatükist saab ülevaate, kuidas Põderis modulaarne analüüs toimib ning kirjeldatud on ka programmi osade salvestamine, laadimine ja taaskasutamine. Praktilise arenduse tulemustest annab ülevaate viies peatükk.

2 Teoreetiline taust

Töö teoreetiline taust annab üldise ülevaate programmide analüüsimisest. Selle koostamisel on lähtutud põhiliselt Kalmer Apinise doktoritööl „Frameworks for analyzing multi-threaded C“ [3] ja Jens Knoopi ning Bernhard Steffeni tööl „The Interprocedural Coincidence Theorem“ [4]. Lisatud näited on inspiratsiooni saanud raamatust „Compiler design. Analysis and transformation“ [5].

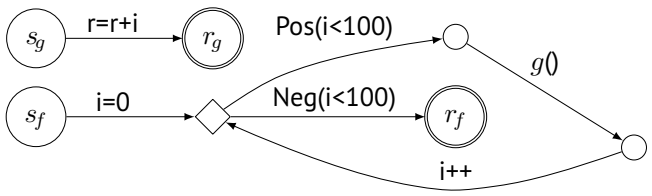
2.1 Programmid

Enne programmide analüüsi on oluline teada, kuidas analüüsitavaid programme esitatakse. Need koosnevad protsetuuridest, mille hulgas on ka $main$ protseduur. Iga protseduuri kohta on voograaf (N_p, E_p, s_p, r_p), kus:

- N_p tähistab lõpliku tippude hulka;
- $E_p \subseteq N_p \times L \times N_p$ on lõplik hulk märgendatud servi. Märgendite hulgas L on käsud, mida programm täidab, nagu näiteks tingimuslaused, omistamised ja teiste protseduuride kutsed;
- $s_p \in N_p$ ja $r_p \in N_p$ on vastavalt alg- ja lõpptipud.

Oletatakse, et algtipust s_p on servade kaudu võimalik saada igasse tippu $u \in N_p$ ja igast tipust u on võimalik saada lõpptippu r_p. Joonisel 1 on näide funktsioonide voograafidest. Kui antud tingimus kehtib, liigutakse Pos serva pidi, ja kui ei kehti, liigutakse Neg serva pidi.

```
int i, r;
void g(){
  r = r + i;
}
void f(){
  i = 0;
  while (i<100){
    g(); i++;
  }
}
```



Joonis 1. Kaks funktsiooni ja nende voograafid [3]

2.2 Semantika

Programmi tätimisel liigutakse voograafi servade kaudu tipust tippu, muutes sealjuures programmi olekut. Programmil on voograafi tippudest koosnev kutsepinu, kus esimene tipp kutsepinus on praegune tipp ja sellele järgnevad tipud, kuhu tuleb liikuda pärast käesoleva protseduuri lõpptippu jõudmist.

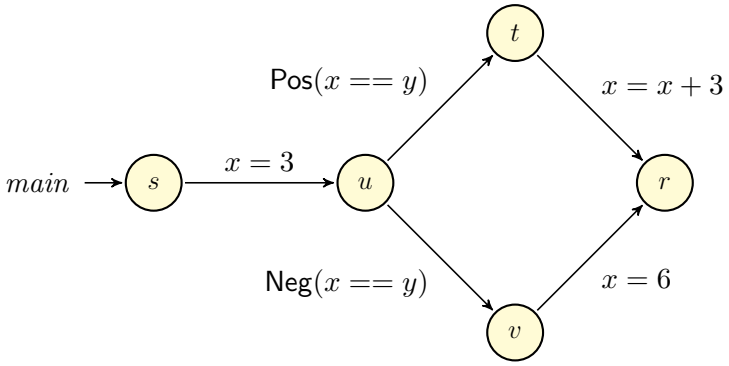
Sammu täitmist kujutatakse binaarse seosega $(\twoheadrightarrow) \subseteq (Pinu \times S) \times (Pinu \times S)$. $Pinu$ kujutab kutsepinude hulka ning S programmi olekute hulka. Seos näitab, kuidas programmi olek muutub sammu tegemisel. Keskendume programmi olekutes hetkel peamiselt muutujatele ning nende väärtustustele. Protseduurikutsed käsitletakse jaotises 3.1.

Võttes programmi algolekuks $s_0 \subseteq S$ (olekud), saab leida programmi kõikide võimalikkude olekute hulga $\mathcal{S} \subseteq Pinu \times S$, kui rakendada programmile korduvalt seost "$\twoheadrightarrow$", alustades olekust $([s_{main}], s_0) \in Pinu \times S$. Saame seda transitiivse sulundiga (tärniga) korrektselt kirjutada järgnevalt:

$$\mathcal{S} = \{y \mid ([s_{main}],\ s_0) \twoheadrightarrow^* y\}.$$

Vahepeal võivad meid huvitada ainult kindlad olekud, mis tekivad, läbides mingit kindlat tippu u. Sellisel juhul saame võtta protseduuri olekud $\mathcal{S}$-i paaridest, mille kutsepinud sisaldavad u tippu:

$$\mathcal{S}[u] = \{s \mid ([u],\ s) \in \mathcal{S}\}.$$



Joonis 2. Hargnemisega voograaf

Näide 1. Vaatame näiteks joonisel 2 kujutatud voograafi. Kui kutsepinu on $s :: r$ ja programmi olekus S on muutuja väärtused $\{x = 8, y = 4\}$, siis pärast ühe sammu tegemist läbi s ja u vahelise serva on kutsepinu $u :: r$ ja olek S muutub ning sisaldab nüüd väärtuseid $\{x = 3, y = 4\}$, sest läbitud serv muutis x-i väärtust.

Voograafi tipu r mõned võimalikud muutujate olekud $\mathcal{S}[r]$ on $\{x = 6,\ y = 5\}$, mis saadakse kui eelnevalt näiteks $y = 5 \neq 3$, ja $\{x = 6,\ y = 3\}$, mis saadakse vastasel juhul $(y = 3)$. Alati $x = 6$, samas muutuja y võimalusi on lõpmatu.

Nägime, kuidas programmi olekuid esitada. Vaatame edasi, kuidas lõpmatuid olekuid abstraheerida, et neid mõistlikult kirjeldada.

2.3 Abstraktsioon

Praktikas on $\mathcal{S}$-i leidmine mõistliku ajaga tihti võimatu, seega tuleb korrektselt abstraheerida. Esiteks on oluline tutvuda mõnede mõistetega.

Osaline järjestus: Hulk D koos operaatoriga $(\sqsubseteq) \subseteq D \times D$ on osaline järjestus, kui $\sqsubseteq$ relatsioon on refleksiivne, transitiivne ja antisümmeetriline. Need omadused on järjestamiseks vajalikud.

Vähim ülemtõke: Osalise järjestuse $(D,\ \sqsubseteq)$ korral, d on vähim ülemtõke mingile alamhulgale X kui:

- d on hulga ülemtõke, ehk iga $x \in X$ korral $x \sqsubseteq d$;
- kõik teised ülemtõked on vähemalt sama suured kui d.

Vähimat ülemtõket nimetatakse ka ülemrajaks.

Täielik võre: Paar $(\mathbb{D},\ \sqsubseteq)$ on täielik võre, kui paar on osaline järjestus ja igal $\mathbb{D}$ alamhulgal X on vähim ülemtõke $\sqcup\ X\ \in\ \mathbb{D}$. Vähima ülemtõkega $(\sqcup)$ defineerime binaarse vähima ülemtõke $x \sqcup y = \sqcup\{x,\ y\}$, vähima elemendi $\bot\ =\ \sqcup\ \emptyset$ ja suurima elemendi $\top = \sqcup\ \mathbb{D}$. Edaspidi kutsutakse täieliku võre lihtsalt võreks.

Me tahame abstraheerida semantikat $\mathcal{S}$ ja iga tipu u kohta „nõrgendada“ informatsiooni $\mathcal{S}[u]$-s. Selle jaoks valime täieliku võre $(\mathbb{D},\ \sqsubseteq)$, kus iga element $d \in \mathbb{D}$ esindab efektiivselt mingit (potentsiaalselt lõpmatut) hulka programmi olekutest. Järjestamise relatsioon $\sqsubseteq$ peab töötama järgnevalt. Kui kahe elemendi korral $d_1, d_2 \in \mathbb{D}$ kehtib $d_1 \sqsubseteq d_2$, siis d_1 peab täpsemat infot andma, näiteks rohkem muutujaid on kindla väärtusega, ja d_2 on rohkem abstraheeritud element, kus osadel muutujatel, mis olid enne defineeritud, kindel väärtus puudub.

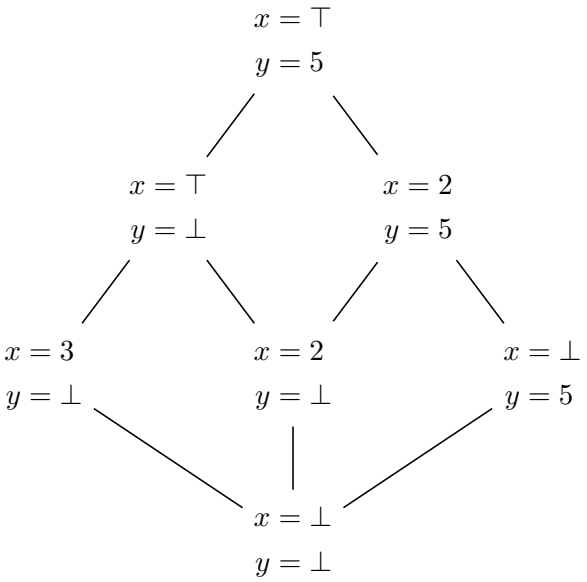
Suhe võre elemendi ja olekute vahel kujutatakse märgiga $\triangle \subseteq S \times \mathbb{D}$. Vähim element $(\bot)$ ei peaks kirjeldama ühtegi hulka ehk ei leidu $s \in S$, mille korral kehtib $s\ \triangle\ \bot$. Samas suurim element $(\top)$ peaks kõiki olekuid kirjeldama, nii et iga $s \in S$ korral kehtib $s\ \triangle\ \top$. Lisaks peab suhe kindlasti väljendama võre järjestust ehk kui võre element d_1 on olekuga s seotud, siis ka kõik suuremad elemendid on selle olekuga seotud:

$$s\ \triangle\ d_1 \wedge d_1 \sqsubseteq d_2 \Longrightarrow s\ \triangle\ d_2.$$

Näide 2. Võtame näiteks kaks programmi muutujate olekut $\{x = 6,\ y = 5\}$ ja $\{x = 6,\ y = 3\}$, mida kirjeldavad võre elemendid d_1 ja d_2. Vähim ülempiir d_1-st ja d_2-st oleks $d_3 \in \mathbb{D}$, mis kirjeldab olekut $\{x = 6\, y = \top\}$, kus x-i väärtust on abstraheeritud.

Hea on kujutada võret Hasse diagrammina — nii nagu joonisel 3, kus alustame alt hästi spetsiifilistest olekutest ja siis edasi üles minnes abstraheerime väärtuseid. Joonisel olevad jooned tähistavad funktsiooni ($\sqsubseteq$).

Siinkohal tuleks märkida, et võre elementideks võivad olla teistsugused ning keerulisemad andmed ja andmestruktuurid, aga lihtsamaks arusaamiseks kirjeldab näide muutujaid hoidvat võre.



Joonis 3. Võre, mis kirjeldab võimalike programmi muutujate olekuid

Funktsiooni $\gamma \in \mathbb{D} \to 2^S$ saame kasutada, et saada hulga kõiki olekuid, mida võre element d kirjeldab:

$$\gamma(d) = \{s \in S \mid s \triangle d\}.$$

Võre loomisel tuleks silmas pidada, et iga oleku hulga S alamhulga X kohta leidub võres element $a(X) \in \mathbb{D}$, mis on vähim element, et kirjeldada kõiki olekuid $s \in X$. Nii tekib a ja y vahel olukord, kus $\gamma(d) = X$ ja $a(X) = d$, mis on tulevikus kasulik, kui tahame mingi võre elemendiga seotud konkreetsetele olekutele rakendada piiranguid.

Lõpuks on vaja iga voograafi tipu u olekute $\mathcal{S}[u]$ jaoks leida abstraheering, nii et iga olek $s \in \mathcal{S}[u]$ oleks kirjeldatud mingi sama võre elemendiga $\mathcal{S}^\sharp[u] \in \mathbb{D}$. Niisugune abstraheering säästab aega ja kirjeldab samas piisavalt kasulikku infot.

3 Protseduuride analüüs

Järgnev peatükk annab põhjalikuma ülevaate, kuidas analüüsida protseduure ja võimalusel taaskasutada teatud osa protseduuride analüüsist. Selleks toetutakse Sharir ja Pnueli poolt kirjeldatud *funktsionaalse lähenemise* meetodile [6].

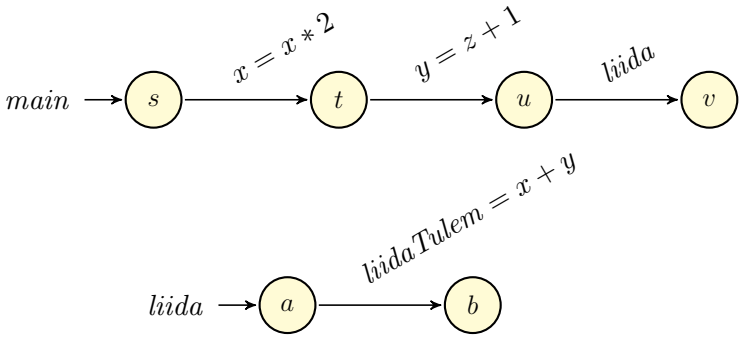
3.1 Protseduuride semantika

Protseduurivaheline semantika kirjeldab, kuidas andmevoog liigub protseduuridest sisse ja välja. Sealjuures on oluline, et erinevad kutsed samale funktsioonile ei põhjustaks andmevoogude segunemist. Konkreetses semantikas on selleks kasutusel kutsepinu. Kui toimub protseduuri kutse programmipunktis u, siis lisatakse pinusse kutsutava protseduuri algtipp ning hiljem lõpptipust välja tulles saab pinus vaadata, et naasta tuleb programmipunkti u.

Näide 3. Uurime järgnevat pseudokoodi:

```
global x = x * 2;              def liida():
global y = z + 1;                  global liidaTulem = x + y
liida();
```

Joonisel 4 on kujutatud vastavad koodi põhjal loodud voograafid. Vaatame $main$ meetodi voograafil tipude u ja v vahelist serva, mis sisaldab funktsiooni $liida$. Kutsepinu tipus u on $u :: []$.



Joonis 4. Kaks voograafi, kus ülemine on üldiselt programmi koodi kohta ja alumine on programmis välja kutsutud funktsiooni $liida$ kohta.

Läbides serva, sisenetakse funktsiooni $liida$ ning kutsepinuks saab $a :: v :: []$. Pärast funktsiooni lõpptippu b peab naasma programmipunkti, kus vastavat funktsiooni välja kutsuti. Kutsepinu on enne tagasihüpet $b :: v :: []$, seega saab minna tagasi punkti v.

Nüüd abstraheerime funktsiooni kutsed järgmiselt. Olgu protseduur f, mille algustipp on u ja lõpptipp on v. Tähistagu s mingit programmi olekut. Meid huvitab kõik

võimalikud täitmised $(u :: c, s) \twoheadrightarrow^* (v :: c, s')$, kus kutsemagasini tipus on algselt u ja jõuame protseduuri lõppu niimoodi, et kõik vahepealsed funktsiooonikutsed on naasnud ja magasini tipus on vaid v. Kõikide selliste (s, s') paaride hulk võtab kokku protseduuri f mõju. Järgnevalt vaatame, kuidas seda relatsiooni (s, s') abstraheerida protseduuri kokkuvõttena.

3.2 Protseduuri kokkuvõte

Võtame esmalt hulga $\mathbb{F}$, mis hoiab endas funktsioone $\mathbb{D} \to \mathbb{D}$. Hulk $\mathbb{F}$ sisaldab muutuseid, mida mingid programmi read või ka protseduurid teevad, mitte konkreetseid väärtusi, mis nende ridade või protseduuridega muutuvad. Näiteks kui ühes võre $\mathbb{D}$ elemendis on muutujate olek $\{x = 2\}$, mis läbi mingi programmi käsu muutub olekuks $\{x = 3\}$, siis hulgas $\mathbb{F}$ tähistaks seda kujul $\{x = x' + 1\}$. Kutsume edaspidi neid muutusi kokkuvõtteks.

Võime oletada, et on olemas nii globaalsed ja lokaalsed muutujad. Globaalsete muutujatega saame simuleerida protseduuride parameetreid ja tagastusväärtusi, mistõttu jätame need välja, et saaksime järgnevat võrrandsüsteemi lihtsamini esitada. Kirjeldagu funktsioon $\circ \in \mathbb{F} \to \mathbb{F} \to \mathbb{F}$ kokkuvõtete järjestikust komponeerimist, ehk operatsiooni $(f \circ g)(x) = f(g(x))$. Loome võrrandsüsteemi üle hulga $\mathbb{F}$, mille vähim lahendus kirjeldab protseduuride kokkuvõtteid:

$$\{s_f\} \sqsupseteq id_{\mathbb{D}} \qquad f \in Proc \tag{1}$$

$$\{v\} \sqsupseteq \{r_f\} \circ \{u\} \qquad e \equiv (u, f(), v) \in E \tag{2}$$

$$\{v\} \sqsupseteq [\![s]\!]^\sharp \circ \{u\} \qquad (u, s, v) \in E \tag{3}$$

Vaatame võrrandeid:

(1) $\{s_f\}$ tähistab protseduuri f alguspunkti kokkuvõtet. Sinna peab kuuluma ainult identsusfunktsioon $id_{\mathbb{D}} \in \mathbb{F}$, millest saame järeldada, et protseduuri kokkuvõtte leidmisel välist infot arvesse ei võeta.

(2) See võrrand näitab, kuidas saada kokkuvõte, kui liikuda programmis serva pidi, mis sisaldab mingi protseduuri f kutset. r_f tähistab protseduuri lõpptippu ning $\{r_f\}$ tähistab protseduuri f kokkuvõtet. Saadud kokkuvõte tuleb siduda nüüd protseduuri kutsele eelneva tipu u kokkuvõtega $\{u\}$ ja saame kokkuvõtte $\{v\}$, mis näitab olukorda pärast protseduuri f kutset.

(3) Siin võrrandis näeme, kuidas tegutseda lihtsalt ühe programmi käsu s korral. $[\![s]\!]^\sharp \in \mathbb{F}$ näitab vastava programmi käsu kokkuvõtet, mille sidumisel eelneva käsu kokkuvõtega $\{u\}$, saame kokkuvõtte pärast käsu s täitmist.

Näide 4. Vaatame jälle joonisel 4 kujutatud programmi. Tahame analüüsida muutujaid erinevates programmi punktides, seega võre $\mathbb{D}$ kirjeldab muutujate olekuid. Alustame

liikumist voograafis. Liikudes tipust s tippu t, muutuja x alati kahekordistub. Kui kokkuvõte oli enne tühi, siis nüüd on kokkuvõte $\{x = x' * 2\}$. Nüüd liigume edasi tipust t tippu u ja saame selle käsu mõju, mis on $\{y = z' + 1\}$. See tuleb siduda eelneva tipu kokkuvõttega ja saadakse $\{x = x' * 2,\ y = z' + 1\}$.

Liikudes tipust u tippu v, kutsutakse protseduuri $liida$. Esiteks tuleb leida protseduuri kokkuvõte $\{r_f\}$. Funktsioon $liida$ on õnneks lihtne, selle kokkuvõte on ainult $\{liidaTulem = x + y\}$. Nüüd tuleb see viia efektiks tipus u. Kuna x ja y on mõlemad $\mathbb{F}$-is ümber defineeritud, siis muutujad asendades ja eelneva kokkuvõtega kombineerides saame, et kokkuvõte tipus v on $\{x = x' * 2,\ y = z' + 1,\ liidaTulem = x' * 2 + z' + 1\}$.

3.3 Kokkuvõte kasutamine

Nüüd vaatame, kuidas protseduuri kokkuvõtteid kasutada, et kirjeldada konkreetseid programmi olekuid, mis jõuavad mingi programmi punktini v. Kirjeldame kõrvutamisega funktsiooni $\mathbb{F} \to \mathbb{D} \to \mathbb{D}$, millega ühendame programmi kokkuvõtteid seisunditega ja seisundeid omavahel. Loome uue võrrandsüsteemi, mis on seekord üle võre $\mathbb{D}$, mille vähim lahendus kirjeldab programmi abstraktseid seisundeid vastavates programmi punktides:

$$[s_{main}] \sqsupseteq d_0 \tag{4}$$

$$[s_f] \sqsupseteq [u] \qquad e \equiv (u, f(), v) \in E \tag{5}$$

$$[v] \sqsupseteq \{r_f\}\ ([u]) \qquad e \equiv (u, f(), v) \in E \tag{6}$$

$$[v] \sqsupseteq [\![s]\!]^\sharp\ ([u]) \qquad (u, s, v) \in E \tag{7}$$

Uurime võrrandeid:

(4) $d_0 \in \mathbb{D}$ kirjeldab programmi algseisu protseduuri $main$ käivitamisel.

(5) Võrrand näitab, kuidas saadakse mingi protseduuri f algseisu $[s_f]$. See on lihtsalt protseduurile eelneva programmi punkti u seisund. Paneme ka tähele, et kui kokkuvõtte saamiseks ei võetud protseduuri välist infot arvesse, siis nüüd programmi seisundi leidmisel peab seda arvesse võtma.

(6) Siin võrrandis näeme, et programmi seisu saamiseks pärast funktsiooni f kutsumist, peame funktsiooni kokkuvõtte $\{r_f\}$ kombineerima seisundiga $[u]$ enne programmi kutset. Mõtleme olukorrast, kus mitu erinevat analüüsitavat programmi kasutavad sama protseduuri. Täheldame, et kui funktsiooni kokkuvõte on juba ühe analüüsi käigus leitud ja alles hoitud, siis on võimalik seda taaskasutada teiste programmide analüüsis, sest kokkuvõte ei võta arvesse protseduuri välist infot. Kui kokkuvõte sõltuks programmi seisundist, siis peaks igakord kokkuvõte uuesti leidma, sest seisund on tõenäoliselt programmidel erinev.

(7) Võrrand näitab lihtsalt, et kui täidetakse programmi käsk s, siis tuleb see efekt kombineerida seisundiga $[u]$ enne käsu kutsumist, et saada seisund $[v]$, mis tähistab seisundit pärast käsku s.

Näide 5. Leidsime eelnevalt joonisel 4 kujutatud programmile kokkuvõte. Rakendame nüüd kokkuvõtet, et leida programmi olek pärast vastava koodi läbimist. Oletame, et programmi olek enne läbimist oli $\{x = 3,\ z = 1\}$. Rakendades kokkuvõtet $\{x = x' * 2,\ y = z' + 1,\ liidaTulem = x' * 2 + z' + 1\}$ saame, et olek pärast läbimist on $\{x = 6,\ z = 1,\ y = 2,\ liidaTulem = 8\}$

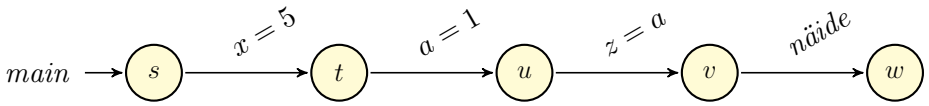
Näide 6. Vaatame nüüd, kuidas on võimalik kokkuvõtet taaskasutada protseduuride analüüsimisel. Las olla eelnevalt analüüsitud pseudokood ühe protseduuri kood, mida kutsume uuesti $main$ meetodist:

```
def näide():
    global x = x * 2;
    global y = z + 1;
    liida();
```

```
def main():
    global x = 5;
    global a = 1;
    global z = a;
    näide();
```

Tahame endiselt muutujaid analüüsida ehk võre $\mathbb{D}$ kirjeldab ikka muutuja olekuid. Alustame joonisel 5 kujutatud $main$ protseduuri voograafi läbimist. Programmi algseisu kirjeldav võre element $d_0 \in \mathbb{D}$ on algul lihtsalt $\{x = \top,\ a = \top,\ z = \top\}$, sest siin näites programm sisendeid ei saa.

Liigume tipust s tippu t ja saame käsu s mõju $\{x = 5\}$. Käsule eelneva tipu seisundis kehtib $x = \top$. Kombineerides selle käsu s kokkuvõtega, saadakse tipu t seisundiks $\{x = 5,\ a = \top,\ z = \top\}$. Samamoodi liigutakse edasi läbi tipu u tipuni v ja saadakse $[v]$ seisundiks $\{x = 5,\ a = 1,\ z = 1\}$.



Joonis 5. Programmi $main$ protseduuri voograaf

Minnes edasi tipust v tippu w, kutsutakse välja protseduur $näide$. Meid hetkel ei huvita seisundid protseduuri sees ning tahame leida ainult seisundi $[w]$. Mäletame, et protseduuri kokkuvõte on juba eelnevalt leitud, mida on nüüd võimalik uuesti kasutada. Protseduuri kokkuvõte oli $\{x = x' * 2,\ y = z' + 1,\ liidaTulem = x' * 2 + z' + 1\}$. Muutujad asendatakse praeguse seisundiga $\{x = 5,\ a = 1,\ z = 1\}$ ja saame, et seisund $[w]$ on $\{x = 10,\ a = 1,\ z = 1,\ y = 2,\ liidaTulem = 12\}$.

Kui tahaksime minna protseduuri $näide$ sisse, siis peab leidma protseduuri algoleku praeguse programmi oleku põhjal. Nüüd hakkame sellest seisundist läbima protseduuri voograafi samamoodi nagu eelnevalt läbisime protseduuri $main$ voograafi. See tähendab, et $näide$ keha analüüsitakse, alustades seisundiga $\{x = 5,\ a = 1,\ z = 1\}$.

Eelnevad näited annavad ülevaate, kuidas kokkuvõtteid esitada ja leida. Need võimaldavad protseduuride analüüse taaskasutada, mis aitab tihedamini kasutatavate protseduuride puhul säästa aega, sest neid ei pea igakord uuesti arvutama. See on kasulik, sest protseduuride analüüs on tihti aeganõudev protsess.

4 Modulaarne analüüs raamistikus Põder

Programm Põder [7] on programmikeeles Scala kirjutatud raamistik, mis võimaldab Java klassifailisid analüüsida. Raamistiku töötas välja Kalmer Apinis koos teiste Tartu Ülikooli töötajatega. Programmiga on tegeletud ka teiste akadeemiliste tööde käigus [8, 9, 10].

4.1 Modulaarne analüüs

Modulaarne analüüs on üks mitmetest analüüsi tüüpidest programmis Põder. Avades joonisel 6 kujutatud kompileeritud Java klassifaili Põderis modulaarse analüüsiga, saadakse graaf, mis on kujutatud joonisel 7.

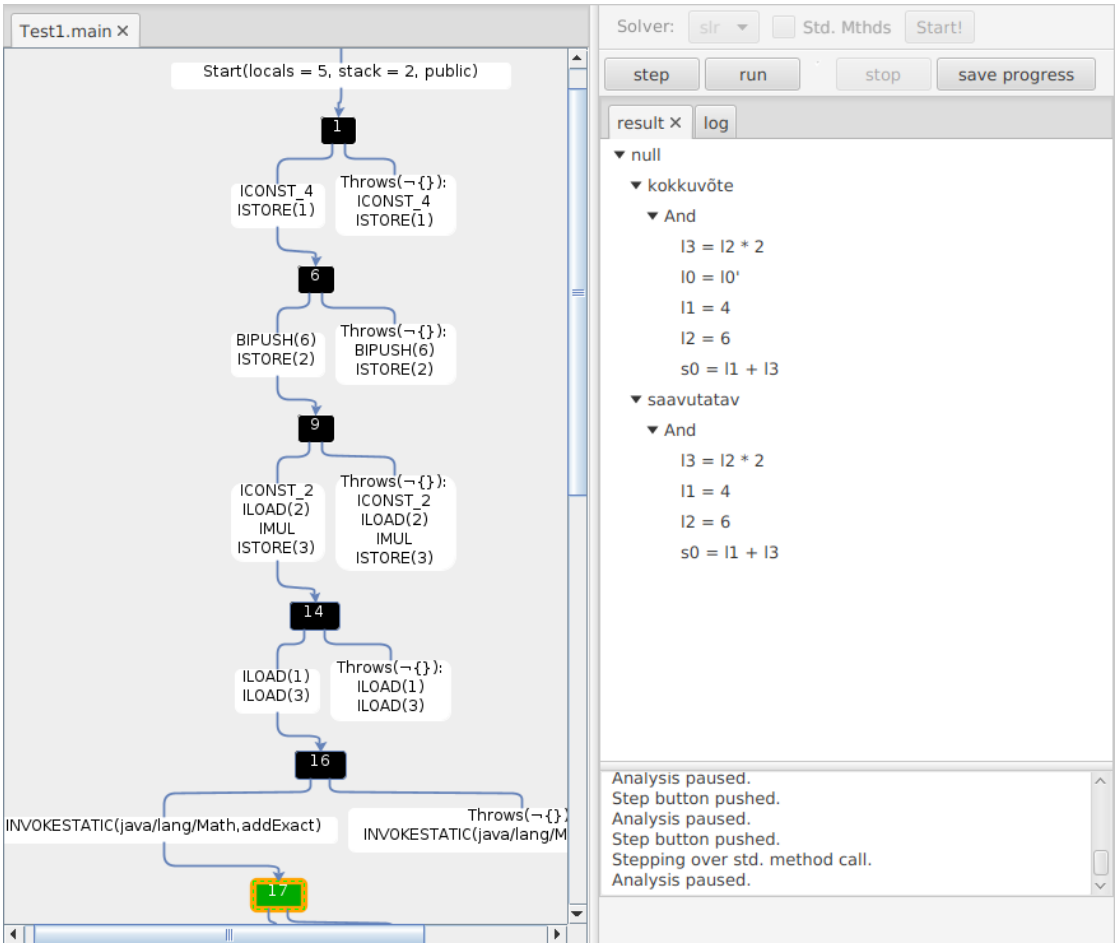
```
class Test1 {
    Test() {
    }

    public static void main(String[] var0) {
        byte var1 = 4;
        byte var2 = 6;
        int var3 = var2 * 2;
        int var4 = Math.addExact(var1, var3);
        P6der.check(var4 == 15);
    }
}
```

Joonis 6. Dekompileeritud Java klassifail $Test1.class$

Voograafi servad on Java baitkoodi käsud, mida programm täidab. Tippude infot on näha paremal ja seal kirjeldatakse programmi kokkuvõtet ning saavutatavat seisundit vastavas programmipunktis. Graafis on võimalik edasi astuda nupuga $step$ ning niimoodi analüüsitakse järgmist käsku ja seotakse see eelneva tipuga. Voograafi pildil oleme juba mõned sammud teinud ja näeme programmi seisu pärast meetodi $addExact$ kutsumist.

Iga tipu kohta näidatakse kokkuvõtet ja saavutatud olekut, mis on võre $LogicLattice$ kujul. $LogicLattice$ hoiab endas loogikalauset, mis näitab kehtivaid tingimusi, kui oleme vastavasse programmipunkti jõudnud. See loogikalause koosneb esiteks And (Ja), Or (Või) and Not (Eitus) tehetest.

Kui on vaja hoida mitut väidet, mis samaaegselt tipus kehtivad, siis And tehtega on võimalik seda teha kujul $And(al)$, kus al on list nendest tehetest. Näeme joonisel 7, et And-i "sees" on näiteks väited muutujate kohta. Samamoodi saab Or-i ja Not-iga käituda. $Or(al)$ näitab, et üks listi al väidetest kehtib ja $Not(q)$ näitab, et mingi väide q ei kehti. Lisaks tuleks märkida, et neid ja tulevasi tehteid saab omavahel kombineerida ja üksteise sisse panna, ehk tehte $And(al)$ korral võib väidete listi al mõni tehetest olla Not tehe.



Joonis 7. *Test1.class* põhjal loodud voograaf

Lisaks oleks vaja hoida võrdus-, korrutus- ja teisi aritmeetilisi tehteid. Selleks kasutatakse tehet App, millele antakse funktsioon ja tehetest koosnev avaldis, millele seda funktsiooni rakendatakse. Muutujate esitamiseks kasutatakse $LVar(p, s)$, kus p tähistab muutuja hoidmise asukohta ning s tähistab, kas tegemist on muutuja seisuga funktsiooni alguses. Muutuja võib meetodi sees muutuda, aga tihti on oluline kokkuvõtet väljendada algseisu kaudu. Mäletame teooria osast, et muutuja x korral tähistati muutuja algseisu kujul x'. Samamoodi näidatakse algseisu ka Põderi *result* aknas. Arvkonstante tähistatakse $IVal(nr)$, kus nr on konstandi väärtus.

Joonisel 7 olev võrrand $l3 = l2 * 2$ on sel juhul hoitud järgnevalt ($ArrayBuffer$-iga tähistame listi):

```
App(Fun(=,FunType(IntType,FunType(IntType,PropType))),
    ArrayBuffer(
      App(LVar(3,false,IntType),ArrayBuffer()),
```

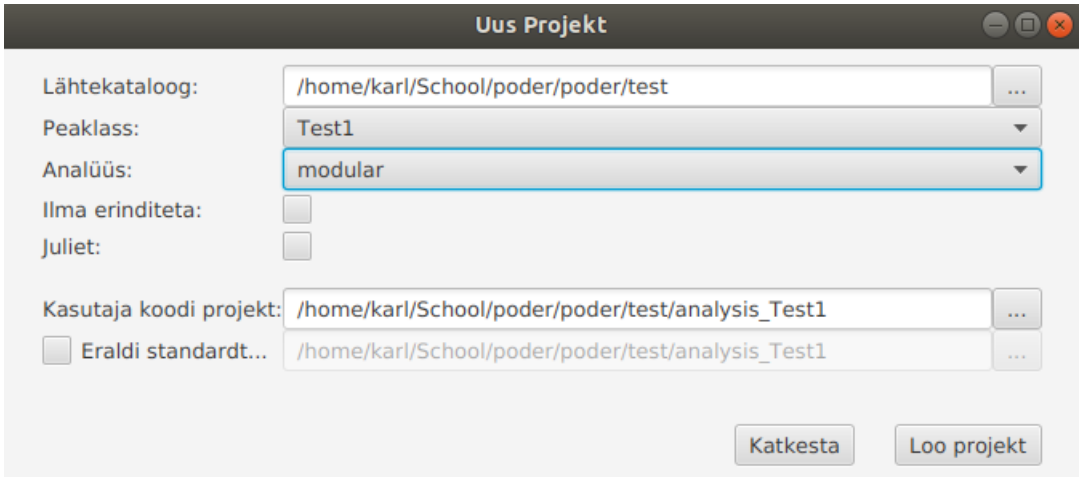
```
App(Fun(*,FunType(IntType,FunType(IntType,IntType))),
    ArrayBuffer(
        App(LVar(2,false,IntType),ArrayBuffer()),
        App(IVal(2),ArrayBuffer())))))
```

Lisaks $LVar$ muutujatele on Põderi vaates näha programmi pinul olevaid $SVar$ väärtusi (tähistatud $s\{nr\}$). Graafil kujutatud tipp kirjeldab programmi olekut pärast funktsiooni $addExact$ kutset, kus saadud tulemus asub pinul ja pole veel muutujasse salvestatud. $LogicLattice$-s saab defineerida ka tehteid, aga praeguse näite kontektsis pole need olulised.

Kokkuvõtvalt võib öelda, et modulaarse analüüsiga saadakse voograaf, kus tipud näitavad võre $LogicLattice$ olekut. See olek kirjeldab tingimusi, mis kehtivad, kui jõutakse vastavasse programmipunkti.

4.2 Tulemuste salvestamine ja laadimine Põderis

Uue Java klassi analüüsimiseks peab esiteks looma uue projekti, mille käigus on võimalik spetsifitseerida kaks kausta, ühte salvestatakse kasutaja kood ja teise salvestatakse standardteegi meetodid. Need kaustad võivad olla ka samad, aga juhul kui tahetakse analüüsida mitut erinevat Java klassi, mis kasutavad samasid standardteegi meetodeid, on kasulik luua neile ühine standardteegi kaust, et standardteegi meetodite analüüsi arvelt aega säästa.



Joonis 8. Uue projekti loomise vaade

Valitud kaustadesse salvestatakse tipud klassi ja meetodi järgi ehk kõik tipud, mida nägime joonisel 7 lähevad $Test1.main([Ljava.lang.String;)V$ kausta ning iga tipu jaoks on tema numbriga tähistatud fail.

Meetodi $addExact$ jaoks eraldi voograafi ei tekkinud, sest me ei märkinud, et tahame standardteegi meetodeid analüüsida (joonisel 7 $Std.Mthds$ linnuke on märkimata). Analüüs tehti taustal ning selle käigus saadud tulemused salvestatatakse kausta $java.lang.Math.addExact(II)I$.

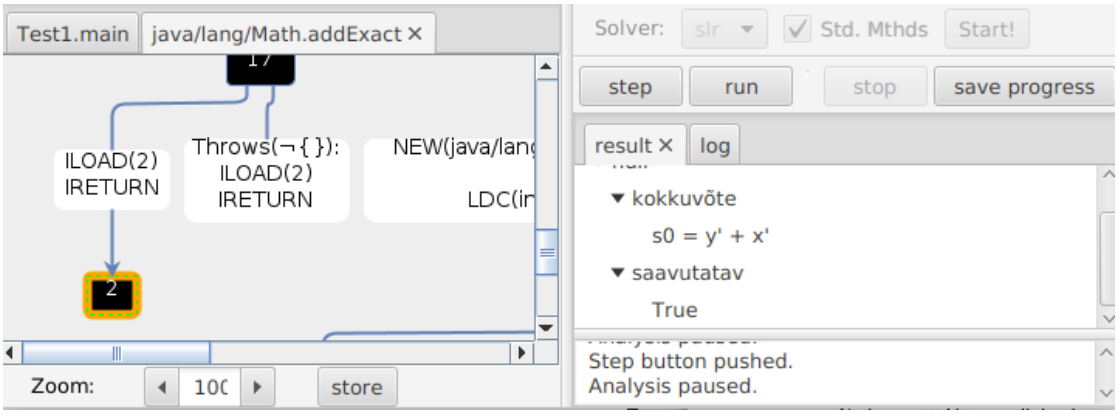
Tipu faili salvestatakse analüüsis nähtav kokkuvõte ja saavutatav olek. Lisaks sellele salvestatakse tõeväärtus, mis ütleb, kas tegemist on $main$ meetodi tipuga.

Küsimuseks jääb, millal salvestamine toimub. Eelnevalt salvestas Põder automaatselt pärast analüüsi lõpetamist lahendatud tipud. Antud töö käigus lisati programmi kasutajale võimalus otsustada, kas lahendus salvestatakse või mitte. Kui analüüs on veel pooleli, saab $save\ progress$ nupule vajutades salvestada selleks hetkeks lahendatud tipud. Kui analüüs on lõpetatud, asendub $save\ progress$ $save\ result$-ga ning on võimalik salvestada lõpptulemused. Selline võimalus on kasulik, sest mittemodulaarsete analüüside tulemusi ei ole vajalik alati ära salvestada. Kui eelnevalt pidi tulemuste salvestamiseks analüüsi lõpuni tegema, siis nüüd saab ka pooleliolevat analüüsi salvestada.

Tulemuste sisse laadimine toimub avatud analüüsis samme tehes. Põder üritab enne tipu lahendamist avada meetodi voograafi vastava tipu faili ja kui see leitakse, saab seal olevat infot kasutada. Kui ei ole tegemist $main$ meetodi tipuga, oletatakse, et see pole muutunud ja seda tippu üle ei kontrollita. Juhuks kui analüüsitava klassi $main$ meetod on muutunud, kontrollitakse selle tipud üle.

4.3 Analüüsi tulemuste taaskasutamine

Vaatame täpsemalt, kuidas on võimalik taaskasutada meetodite analüüse. Mäletame, et klassi Test1 analüüsi käigus analüüsiti ka meetodit $addExact$. Analüüsi salvestamisel talletati joonisel 9 nähtav $addExact$ meetodi lõpptipp vastavasse kausta ($java.lang.Math$ $.addExact(II)I$).



Joonis 9. Meetodi $addExact$ voograafi lõpptipp

Faili sisse salvestatakse tõeväärtus, mis näitab, kas on tegemist $main$ meetodi tipuga.

Lisaks sellele salvestatakse saavutatav oleks, mis on praegusel juhul lihtsalt $True$, ja kokkuvõte $s0 = y' + x'$. Kui teist korda $addExact$ serva läbida ja projekti kaustas on selle meetodi tipud olemas, ei pea seda uuesti analüüsima. Põder laeb lõpptipu kokkuvõtte ja see kombineeritakse eelneva tipuga.

Kombineerimise teostab joonisel 10 kujutatud $compose$ funktsioon. Muutuja f on funktsiooni kokkuvõte, mis on praeguse näite korral koodi tasandil järgnev:

```
App(Fun(=,FunType(IntType,FunType(IntType,PropType))),
    ArrayBuffer(
        App(SVar(0,IntType),ArrayBuffer()),
        App(Fun(+,FunType(IntType,FunType(IntType,IntType))),
            ArrayBuffer(
                App(LVar(1,true,IntType),ArrayBuffer()),
                App(LVar(0,true,IntType),ArrayBuffer())))))
```

Alguses asendatakse meetodi $addExact$ kokkuvõttes $SVar(0)$ $RetVar$-iga, sest kui hakkatakse $addExact$ kokkuvõtet ühildama ülemfunktsiooni (siinkohal Test1 klassi $main$) kokkuvõtega, on vaja enne ülemfunktsiooni $SVar(0)$ siduda meetodi $addExact$ kokkuvõtega, sest see pinu peal olev väärtus on üks meetodile sisse antud parameetritest. Kui asendust $RetVar$-iga ei toimuks, oleks mingil hetkel kokkuvõttes kaks $SVar(0)$ elementi, üks tähistaks parameetrit ning teine tähistaks meetodi tulemust, mis ei ole korrektne.

```
def compose(args:Seq[(Int,LVar)], f:l): l => l = inForall{
  x =>
    val retVar =  RetVar
    val f1 = repl(Map[Val,l](SVar(0) -> and(retVar)))(f)
    def r(x: l): l = {
      args.foldRight(x) {
        case ((i,v),b) => repl(Map[Val,l](v -> and(SVar(i))))(b)
      }
    }
    val hmm = and(x,forgetLocals(r(f1)))
    val q = popStack(hmm,args.length)
    val w = simplify(repl(Map[Val,l](retVar -> and(SVar(0))))(
       pushStack(q)))
    w
  }
```

Joonis 10. Modulaarse analüüsi funktsioon $compose$

Funktsiooni r sees asendatakse meetodi $addExact$ kokkuvõttes olevad argumendid järjest vastavate $SVar$ pinu väärtustega. Praeguse näite puhul asendatakse $LVar(1)$

$SVar(1)$-iga ja $LVar(0)$ $SVar(0)$-iga. Edasi ühendatakse saadud kokkuvõte funktsiooni and abil ülemfunktsiooni kokkuvõtega ning saadakse:

```
And(ArrayBuffer(
    ....
    ....
    App(Fun(=,FunType(IntType,FunType(IntType,PropType))),
        ArrayBuffer(
            App(SVar(1,IntType),ArrayBuffer()),
            App(LVar(1,false,IntType),ArrayBuffer()))),
    App(Fun(=,FunType(IntType,FunType(IntType,PropType))),
        ArrayBuffer(
            App(SVar(0,IntType),ArrayBuffer()),
            App(LVar(3,false,IntType),ArrayBuffer()))),
    App(Fun(=,FunType(IntType,FunType(IntType,PropType))),
        ArrayBuffer(
            App(RetVar,ArrayBuffer()),
            App(Fun(+,FunType(IntType,FunType(IntType,IntType))),
                ArrayBuffer(
                    App(SVar(1,IntType),ArrayBuffer()),
                    App(SVar(0,IntType),WrappedArray()))))))))
```

Näha on kahte pinu peal olevat väärtust ($SVar(1) = LVar(1)$ ja $SVar(0) = LVar(3)$) ning all meetodi $addExact$ kokkuvõtet. Kutsudes $popStack$ funktsiooni, asendatakse $SVar(1)$ $LVar(1)$-iga ja $SVar(0)$ $LVar(3)$-iga ning kaotatakse pinupealsed väärtused. $RetVar$ muudetakse tagasi $SVar(0)$-iks ja saadakse kokkuvõte, mida oli näha ka joonisel 7.

Põder toimib samamoodi, kui panna meetodi $addExact$ kaust mingi teise klassi projekti kausta või luua klassidele ühine standardteegi meetodite kaust. Ka teise klassi analüüsi käigus üritatakse esiteks leida projekti kaustast meetodi $addExact$ kaust ja selle sees olevad tipud. Kui need leitakse, saab meetodi kokkuvõtte kombineerida vastava programmipunkti kokkuvõttega ning ei pea meetodi $addExact$ analüüsi kordama.

5 Täiendused raamistikule Põder

Modulaarse analüüsi salvestamine ja laadimine juba suuresti toimis ning oli võimalik meetodite kokkuvõtteid taaskasutada. Probleemiks oli, et tippude laadimine ei toiminud osade teiste analüüside korral, nii et tulemuste laadimine polnud kasutusel, et teised analüüsid töötaksid.

Esmalt parandasingi ära erinevad vead, et tippude laadimine toimiks ka teiste analüüside korral. Analüüsi vaatesse lisati eelnevalt mainitud salvestamise nupp, et teha salvestamine valikuliseks ja oleks võimalik salvestada pooleliolevaid analüüse. Lisaks väiksematele parandustele implementeerisin ka laetavate tippude verifitseerimise ja projektide versioonihalduse. Täiendused on leitavad allikast [11].

5.1 Verifitseerimine

On oluline, et kasutaja saaks kindel olla laetavate tippude korrektsuses, seetõttu lisasin Põderile nende verifitseerimise. Verifitseerimine toimub enne analüüsi algust. Verifitseerimise funktsioonile antakse parameetrina algtipud. Seejärel hakatakse tippe järjest lahendama joonisel 11 kujutatud $solve\_one$ funktsiooni abil.

Hulgas ds hoitakse juba lahendatuid tippe. Kontrollitakse, kas käsitletav tipp v on juba lahendatud ja kui ei ole, lisatakse v hulka ds ning lahendatakse tipp funktsiooniga tf. Funktsioonile tf antakse parameetritena edasi ka funktsioonid get ja set.

```
def solve_one(v: Var): D = {
  var d1 = L.bot
  if (!(ds contains v)) {
    ds += v
    d1 = s.tf(get(v, _), set(v, _), inVerification = true)(v)
  }
  val prevSol = previousSol(v)
  if (prevSol.isDefined) {
    val d = prevSol.get._1
    checkLeq(v, d, d1)
    d
  } else {
    L.top
  }
}
```

Joonis 11. Verifitseerimise abifunktsioon $solve\_one$

Funktsiooni get kutsutakse, kui tahetakse saada mingi tipu väärtust. Verifitseerimisel tähendab see tipu lahendamist $solve\_one$-iga. Funktsioon $solve\_one$ tagastab, kas laetud väärtuse või selle puudumisel võre suurima elemendi $\top$ ($L.top$). Suurim element võres on

tõeväärtus $True$ ehk selle tagastamisel verifitseerimine jätkub ja oletatakse, et programmi olekut kirjeldava tipuni on võimalik jõuda. Funktsioon set on tipu väärtuse seadmiseks, verifitseerimisel lisatakse tipp kontrolli vajavate tippude hulka.

Pärast tipu lahendamist funktsiooniga tf proovib Põder projekti kaustast laadida olemasolevat tipu lahendust. Õnnestumisel saab võrrelda laetud lahendust $prevSol$ praegu lahendatud väärtusega $d1$.

Funktsiooniga $checkLeq$ kontrollitakse, et lahendatud võre element $d1$ on väiksem või võrdne laetud elemendist. Nagu kirjeldatud teoreetilises taustas, võrreldakse võre elementisid nende täpsuse alusel. Nii võib $d1$ olla väiksem. See tähendab, et ta on üldisem ja ebatäpsem, aga ei tähenda, et laetud element $prevSol$ on vale. $d1$ ei tohi olla samas suurem või mittevõrdne, sest siis on laetud väärtus, mida ka hiljem analüüsis kasutatakse, kas ebatäpsem või vale. Sellisel juhul hoiatab Põder kasutajat sõnumiga ning kasutaja saab ise otsustada, kuidas edasi käituda.

Verifitseerimine lisab kindlust, aga sellega kaasnevad ka mõned probleemid. Verifitseerimisel tehakse vähem kui reaalse lahendamise käigus, mistõttu on ka ajakulu väiksem. Vaatamata sellele tuleb märkida, et verifitseerimisega väheneb osaliselt laetavate tippudega saadav ajakasu. Plaanis on ajakulu leevendada. Võimalik on teha verifitseerimine valikuliseks või teha seda taustal, et kasutaja saaks analüüsiga jätkata.

5.2 Versioonihaldus

Eelnevas peatükis nägime, et salvestamine ja laadimine käib Põderis projekti kaustade kaudu. Töö käigus täiendati projekte versioonide toetusega ja analüüside jaotusega.

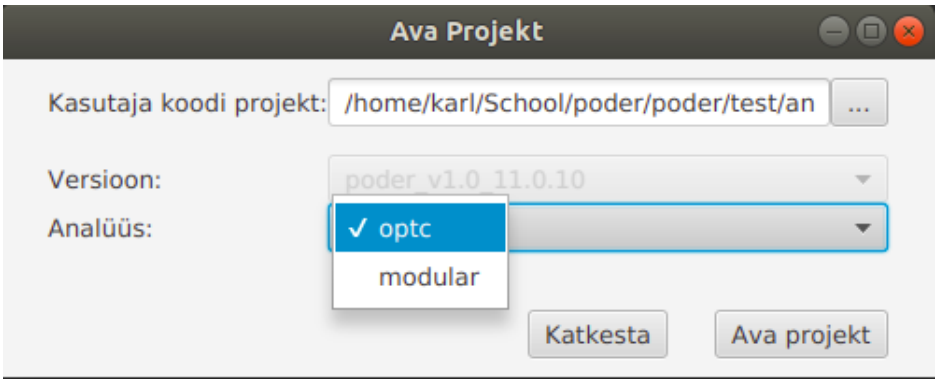
Versioonide toetus projektidele on eriti kasulik tulevikuks, kui Põderil on erinevaid versioone ja tahetakse analüüsi tulemusi lugeda läbi võrgu. Põderi ja Java versioonide tõttu võivad kaks sama klassi analüüsi väga palju erineda ning kui Põderis laetakse teistes versioonides tehtud analüüsi, võib Põder kokku joosta või analüüsis tekivad vead. Seetõttu on näiteks internetist laadimisel oluline teada vastava kasutaja Põderi ja Java versioone, millega aitab versioonihaldus.

Lisaks versioonidele on kasulik ka analüüsi tüüpi eristada, sest siis saab teha samas projektis mitut erinevat analüüsi. Sama Java klassiga erinevaid analüüse tehes pidi eelnevalt iga analüüsi jaoks eraldi projekti looma, mis oli väga tülikas.

Täiendasin Põderi salvestamist, et tulemused jaotataks projekti kaustas versioonide ja analüüsi tüüpide järgi ära. Kui eelnevalt salvestati kujul *projekt/meetod/tipp*, siis nüüd salvestatakse järgnevalt *projekt/poder_v+java_v/analüüs/meetod/tipp*. Nii saab ühe projekti sees olla erinevaid versioone ja analüüse ning nad ei hakka üksteist segama. Neid kaustasid on hea teistele kasutajatele saata, sest versioon ja analüüs tuleb arusaadavalt kaustade struktuurist välja.

Muudetud sai ka projekti avamine, et see saaks uue kausta struktuuriga hakkama. Enne sai projekti kausta avada ja analüüs tuli automaatselt lahti. See toimib endiselt, kui

on kasutajaga samad versioonid ja projektis on ainult ühte tüüpi analüüs. Kui versioon erineb või on mitu analüüsi, avatakse joonisel 12 kujutatud aken.



Joonis 12. Projekti avamise aken

Joonisel 12 on näha projekti avamine, kus on kasutaja versiooniga tehtud kaks analüüsi. Projektis on veel teisi versioone, aga kui on leitud kasutaja versioon, siis teisi ei näidata, sest nad ei ole nii töökindlad. Kasutaja versiooni puudumisel Põder hoiataks, et analüüs ei pruugi valikus olevate versioonidega korralikult töötada, aga soovi korral annab võimaluse ikkagi ühe nendest avada. Erinevalt versioonidest tuleb alati valida üks analüüsidest, mis on tehtud vastava valitud versiooniga.

Kokkuvõtlikult hoiab Põderi projekt nüüd versioone eraldi kaustades. Versiooni kaustade sees hoitakse omakorda tehtud analüüside tüüpe, mis on samuti kaustadega eraldatud. Versioonide eraldamine võimaldab tulevikus internetist tulemusi laadida, sest laadides on versioon teada, mis aitab vältida versioonide erinevusest tulenevaid probleeme. Samas analüüside eraldus on juba praegu kasulik, sest see võimaldab ühe projekti sees mitmeid analüüse teha.

6 Kokkuvõte

Lõputöö käigus anti modulaarse analüüsi jaoks vajalik teoreetiline ülevaade. Esiteks defineeriti programmid ning näidati, kuidas kirjeldada programmi olekuid ja neid olekuid abstraheerida. Selle põhjal oli võimalik kirjeldada protseduuride analüüsi, mis võimaldab protseduure eraldi käsitleda ja seetõttu taaskasutada.

Peale selle dokumenteeriti ka modulaarset analüüsi programmianalüüsi raamistikus Põder. Kirjeldati analüüsi võre, mille kaudu esitatakse olekuid erinevates programmipunktides. Lisaks näidati, kuidas toimib Põderis analüüsi tulemuste salvestamine ning laadimine. Põhjalikumalt keskenduti meetodi analüüsi kombineerimisele programmi analüüsiga, kus vastavat meetodit välja kutsuti. Seeläbi näeb, kuidas on võimalik modulaarse analüüsiga meetodite analüüse erinevate analüüside käigus korduvalt kasutada.

Praktilise töö käigus täiendati Põderit, et toimiks tulemuste salvestamine ja laadimine. Tulemuste laadimine tekitas eelnevalt probleeme teiste analüüsidega. Seetõttu tehti kindlaks, et need protsessid töötavad ning ei sega teiste analüüside tööd.

Lisaks salvestamise ning laadimise parandamisele, täiendati Põderit verifitseerimisega, mis kontrollib, et laetud tulemused on tõesed. Selle tulemusena saab kasutaja olla laetavates tulemustes kindlam, aga kaob osa laadimisega võidetavast ajast. Tulevikus on võimalik ajakadu vähendada, tehes verifitseerimine valikuliseks või taustal töötavaks.

Põderi projektidele lisati ka versioonide toetus. See tähendab, et tulevikus, kui versioonid tekivad, on võimalik eristada analüüse versiooni järgi. Nii on võimalik analüüse näiteks läbi interneti laadida. Lisaks versioonidele jaotab Põder analüüse ka tüübi alusel. Eelnevalt ei saanud samas projektis mitu analüüsi olla, mistõttu pidi looma ühe programmi erinevatele analüüsidele eraldi projektid. Kui Põder ise analüüside eralduse teeb, säästetakse aega ja ei pea mitmeid projekte looma.

Tulevikus on võimalus Põderi salvestamisele ja laadimisele veel täiendusi teha. Lisaks eelnevalt mainitud paranduskohtadele, saab näiteks luua võimaluse piirata osade programmipunktide salvestamist.

Viidatud kirjandus

[1] Møller A., Schwartzbach M.I. *Static Program Analysis*. Aarhuse Ülikooli informaatika teaduskond, Taani. 2020.
https://cs.au.dk/~amoeller/spa/spa.pdf (21.02.21)

[2] Cousot P., Cousot R. Modular Static Program Analysis. Horspool R. N. (toim.), *Compiler Construction*, 2002, LNCS nr 2304. Berlin, Heidelberg: Springer.
https://doi.org/10.1007/3-540-45937-5_13 (21.02.21)

[3] Apinis K. *Frameworks for analyzing multi-threaded C.* Müncheni Tehnikaülikooli informaatika II teaduskonna doktoritöö. 2014.
http://www2.in.tum.de/bib/files/apinis14diss.pdf (12.12.20)

[4] Knoop J., Steffen B. The Interprocedural Coincidence Theorem. *Compiler Construction*, 1992, LNCS nr 641. Berlin, Heidelberg: Springer.
https://doi.org/10.1007/3-540-55984-1_13 (12.12.20)

[5] Seidl H., Wilhelm R., Hack S. *Compiler design. Analysis and transformation.* Berlin, Heidelberg: Springer, 2013.

[6] Sharir M., Pnueli A. Two approaches to interprocedural data flowanalysis. Muchnick S., Jones N. (toim.), *Program Flow Analysis:Theory and Application*, lk 189–233. Prentice-Hall, 1981.

[7] Põder. Lähtekood.
https://bitbucket.org/kalmera/poder/src/master/ (06.05.21)

[8] Mullari H. *Java baitkoodi sünkroniseerimise analüüs raamistikus Põder.* Tartu Ülikooli arvutiteaduse instituudi bakalaureusetöö. 2020.
https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=69894&year=2020 (07.05.21)

[9] Iher M. *Nõrgima eeltingimuse staatiline analüüspinukeeltele.* Tartu Ülikooli arvutiteaduse instituudi bakalaureusetöö. 2019.
https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=66604&year=2019 (07.05.21)

[10] Sinisalu A. *Java programmide staatiline intervallanalüüs raamistikus Põder.* Tartu Ülikooli arvutiteaduse instituudi bakalaureusetöö. 2019.
https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=66504&year=2019 (07.05.21)

[11] Põderi haru (inglise keeles *fork*). Lähtekood täiendustega.
https://bitbucket.org/karlike/poder/src/master/ (06.05.21)

Lisad

I. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Karl Marten Mägi**,

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

   Modulaarne staatiline programmianalüüs,

   mille juhendaja(d) on Vesal Vojdani ja Kalmer Apinis,

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

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

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

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

Karl Marten Mägi
07.05.2021