TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut
Teoreetilise informaatika õppetool

Simple-Pascali programmide optimeerimine Haskellis andmevooanalüüsi teegiga Hoopl

Bakalaureusetöö

Üliõpilane: Tõnn Talvik
Üliõpilaskood: 073849 IAPB
Juhendaja: prof. Tarmo Uustalu

Tallinn
2013

If you optimize everything, you will always be unhappy.

Donald Ervin Knuth

Autorideklaratsioon

Deklareerin, et käesolev lõputöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud.

....

(kuupäev) (lõputöö kaitsja allkiri)

Annotatsioon

Käesolev töö tutvustab andmevooanalüüsi teegi Hoopl kasutamist Haskell platvormil, lisades optimeerimise lihtsustatud Pascali kompilaatorile *Simple-Pascal*. Luuakse teegi kasutamiseks sobilik vahe-esitus. Sooritatakse mh konstantide voltimise ja levitamise, koopiate levitamise, ühiste alamavaldiste eemaldamise, surnud omistamiste eemaldamise analüüsid. Kirjeldatakse iga optimeerimise andmevoo fakt, võre, siirdefunktsioon ja vahe-esituse ümberkirjutamine.

Abstract

This thesis describes the application of the dataflow analysis library Hoopl on Haskell platform by adding an optimizer to the simplified Pascal compiler *Simple-Pascal*. An intermediate representation is created to apply the library. Constant evaluation and propagation, copy propagation, common subexpression elimination and dead assignment elimination optimization passes are performed. Dataflow analysis facts, lattice, transfer function and intermediate representation rewriting are described for each pass.

Sisukord

Koodiloend

Mõisted ja lühendid

- arvutama — ingl. evaluate
- AST — abstraktne süntaksipuu (ingl. abstract syntax tree)
- f-n — funktsioon
- IR — vahe-esitus (ingl. intermediate representation)
- sooritus — terviklik optimeerimise teostus (ingl. optimization pass)
- sõne — tähemärkide jada (ingl. string)

1 Sissejuhatus

Teema valik on ajendatud huvist kasutada Haskellit ning avardada teadmisi funktsionaalprogrammeerimise paradigmast. Kompilaatori tegemine pakub hulgaliselt näiteprobleeme, sh optimeerimise. Pascal oli minu esimene koolis õpitud programmeerimiskeel ja kuna igapäeva töös kasutan selle edasiarendust Delphit, siis valisin optimeeritavaks keeleks just Pascali — avaneb võimalus näha Pascali keelt ja kompilaatorit „teiselt poolt“.

Programmi kompileerimise, kus sisendiks on programmeerija poolt koostatud kõrgtaseme programmikeeles lähtetekst ja väljundiks on masinal käivitatav arvutiprogramm, saab jagada järgnevateks osadeks:

1. lähteteksti süntaksi kontroll ja jaotamine lekseemideks,
2. lekseemide süntaktiline parsimine ASTiks ja tüüpide kontroll,
3. ASTi transleerimine vahe-esitusse edasiseks analüüsimiseks,
4. vahe-esitusel põhinev optimeerimine konkreetsest käitusmasinast sõltumatult,
5. masinaspetsiifiline optimeerimine ja masinkeeles programmi genereerimine.

Esimese kahe osa (v.a. tüüpide kontroll) realiseerimiseks saab kasutada Haskelli platvormiga kaasa tulevaid vahendeid:

- leksikalist analüsaatorit *alex*[1] ja parseri generaatorit *happy*[2], mis vastavad keele C vahenditele *lex* ja *yacc*, või
- parserkombinaatorite teeki *Parsec*[3].

Neljanda osa katmiseks on tehtud Haskelli teek Hoopl[4], mis kasutab andmevooanalüüsi meetodit. Andmevooanalüüsis kogutakse andmevoo fakte programmi eri punktide kohta. Programmist koostatakse juhtvoo graaf, mille iga sõlm seab faktidele kitsendusi. Saadud kitsenduste süsteem lahendatakse, leides vähim püsipunkt vastavas võres. Kogutud faktide põhjal optimeeritakse programm. Hea ülevaate andmevooanalüüsist annab [5]. Andmevooanalüüsi Hoopl-teegi näitel selgitab ka [1].

Ilya V. Portnov on loonud Pascali laadse keele kompileerimiseks lihtsale magasiniga virtuaalmasinale (ingl. simple stacked virtual machine, SSVM) kompilaatori *Simple-Pascal*. Kuna see kompilaator ei järgi Pascal-keele standardit[5] ning on oluliselt piiratum (nt puuduvad eel- ja järelkontrolliga tsüklid), siis tuleb seda keelt pidada **lihtsustatud Pascaliks**. *Simple-Pascali* lähtekoodi põhjal koostatud grammatika, mida suudetakse parsida, on toodud laiendatud Backus-Naur kujul lisas A.

Kompilaator käitub järgnevalt: lähtekood parsitakse Parsec'iga, teostatakse tüüpide kontroll ja väljastatakse virtuaalmasinal käivitatav baitkood, mida eelnevalt ei optimeerita.

[1] `http://www.haskell.org/alex/`

[2] `http://www.haskell.org/happy/`

[3] `http://legacy.cs.uu.nl/daan/parsec.html`

[4] `http://research.microsoft.com/en-us/um/people/simonpj/papers/c--/hoopl-haskell10.pdf`

[5] ISO 7185: The original Pascal standard, `http://pascal-central.com/docs/iso7185.pdf`

Käesolev töö lisab *Simple-Pascali* kompilaatorile Hoopl-teegil põhineva optimeerimise: eeltoodud loendis osad 3 ja 4. Seejuures peetakse optimaalseks programmi, mis käivitades annab õige tulemuse, st sama tulemuse, mis annaks see programm ilma optimeerimiseta — optimeerimine peab olema konservatiine — ning teeb seda vähema või sama arvu masinkeele käskude täitmisega. Ei tagata käskude arvu minimaalsust, st võib leiduda veelgi „optimaalsem“ programm. Ei uurita muutmälu ega teiste ressursside kasutust.

2 ASTide tõlkimine vahe-esitusse

Selle osa eesmärk on transleerida *Simple-Pascali* abstraktne süntaksipuu vahe-esitusse, mis on sobilik edasiseks analüüsimiseks Hoopl-teegiga.

2.1 Ülevaade ASTist

S*imple-Pascali* abstraktse süntaksipuu ülesehitus Haskelli tüübikonstruktoritena on toodud koodis 1. Puu sõlmed on varustatud tüübioperaatori `:~` abil annotatsioonidega (read 1-4): parsimisel lisatakse sõlmedele nende asukoht lähtetekstis (rida ja veerg); tüübikontrollis lisatakse tüübiinformatsioon.

Simple-Pascal toetab 3 tüüpi literaale: täisarvud, sõned ja tõeväärtused (kood 1, rida 6).

Avaldised koosnevad (kood 1): muutujatest (rida 9), massiivi elemendi või kirje välja väärtuse kasutamisest (read 10-11), literaalidest (rida 12), funktsiooni väljakutsetest argumentidel (rida 13) ning binaarsetest tehetest (read 16-20) avaldistel (rida 14). Tüüp `Id` on samaväärne sõnega.

Omistamiste vasakul poolel võib olla (kood 1): muutuja (rida 23), massiivi element (rida 24) või kirje väli (rida 25).

Laused on järgnevad (kood 1): omistamine (rida 28), protseduuri väljakutse argumentidel (rida 29), funktsioonist väärtusega tagasipöördumine (rida 30), tsükli katkestamine (rida 31) ja jätkamine (rida 32), programmi või protseduuri lõpetamine (rida 33), kui-siis-muidu hargnemine (rida 34) ja loendav tsükkel (rida 35).

Simple-Pascali programm (kood 1) koosneb konstantidest (rida 38), tüüpidest (rida 39), globaalsetest muutujatest (rida 40), funktsioonidest (sh. protseduuridest, mis on realiseeritud `void` tüüpi funktsioonidena; rida 41) ja programmi kehast (rida 42), milleks on lausete jada.

Käesolev töö tegeleb protseduuri siseste optimeerimistega. Lihtsuse huvides pole siinkohal toodud `Function` tüübi esitust Haskellis. Samuti on välja jäetud tüüp `Type`.

2.2 Hoopl-teegi nõuded vahe-esitusele

Hoopl-teek töötleb juhtvoo graafe (ingl. control-flow graph), mis koosnevad: a) servadega ühendatud plokkidest (ingl. blocks), b) liimitud graafidest (ingl. spliced graphs). Plokk omakorda koosneb ühest või mitmest järjestikusest sõlmest (ingl. node). Sõlm kui kõige väiksem juhtvoo üksus vastab madaltaseme keeles käsule (lae, salvesta, opereeri, hargne vms) või kõrgtaseme keeles lihtlausele (ingl. statement): kui-siis-muidu, omista vms. [4, lk. 2]

Juhtvoo kulgemist sõlme, plokki või graafi sisenemisel ja väljumisel tähistatakse kujudega avatud (open, `O`) ja suletud (closed, `C`):

Kood 1: *Simple-Pascali* abstraktne süntaksipuu

```
data Annotate node ann = Annotate
  { content :: node
  , annotation :: ann }
type node :~ ann = Annotate (node ann) ann

data Lit = LInteger Integer | LString String | LBool Bool

data Expression a =
    Variable Id
  | ArrayItem Id (Expression :~ a)
  | RecordField Id Id
  | Literal Lit
  | Call Id [Expression :~ a]
  | Op BinOp (Expression :~ a) (Expression :~ a)

data BinOp =
    Add | Sub
  | Mul | Div | Mod
  | Pow
  | IsGT | IsLT | IsEQ | IsNE

data LValue a =
    LVariable Id
  | LArray Id (Expression :~ a)
  | LField Id Id

data Statement a =
    Assign (LValue :~ a) (Expression :~ a)
  | Procedure Id [Expression :~ a]
  | Return (Expression :~ a)
  | Break
  | Continue
  | Exit
  | IfThenElse (Expression :~ a) [Statement :~ a] [Statement :~ a]
  | For Id (Expression :~ a) (Expression :~ a) [Statement :~ a]

data Program a = Program
  { progConsts :: [(Id, Expression :~ a)]
  , progTypes :: Map Id Type
  , progVariables :: [Annotate Symbol a]
  , progFunctions :: [Function :~ a]
  , progBody :: [Statement :~ a] }
```

- sisenemisel (ingl. entry): avatud — leidub unikaalne eelkäija; suletud — eelkäijaid on null või rohkem,
- väljumisel (ingl. exit): avatud — leidub unikaalne järeltulija; suletud — järeltulijaid on null või rohkem,

seejuures eelkäija on koht, kust juhtvoog lähtub, ning järeltulija koht, kuhu juhtvoog suubub. [4, lk. 2-3]

Plokki sisenemise kuju on määratud ploki esimese sõlme sisenemise kujuga ja väljumise kuju on määratud ploki viimase sõlme väljumise kujuga. Ploki sisene juhtvoo kulg peab järgima sõlmede järjestust ja kujusid, st ploki siseselt ei saa olla väljumisel suletud sõlme, millele järgneb veel mõni sõlm, ega sisenemisel suletud sõlme, millele eelneb mõni sõlm. Kahte plokki saab avatud otsest siduda üheks terviklikuks plokiks. [4, lk. 3]

Graaf sisaldab otstest suletud plokkide kogumit. Juhtvoo kulg ploki sees on ilmne, kuid plokkide vaheline kulg on määratud graafi servadega. Tüübiklass `NonLocal` seab vastavusse sisenemisel suletud sõlme ja tema unikaalse märgendi (ingl. label); väljumisel suletud sõlme ja vahetult järeltulevate sõlmede märgendite hulga, kuhu võib juhtvoog edasi kanduda. Sellega määrakse graafi servad. [4, lk. 4]

Graaf võib täiendavalt omada sisenemispunkti, sisenemisel avatud plokki, ning väljumispunkti, väljumisel avatud plokki, mis muudavad vastavalt graafi sisenemise ja väljumise kujusid. Kahte graafi saab avatud otsest siduda (operaator `<*>`) üheks terviklikuks graafiks sarnaselt plokkidega nii, et juhtvoog kandub väljumisel avatud graafist edasi sisenemisel avatud graafi. Lisaks saab graafe suletud otsest liimida (operaator `|*><*|`) suuremaks graafiks, kuid selline liimimine ei mõjuta juhtvoo kulgu. [4, lk. 4]

Hoopl-teek pakub teegi kasutajale graafi ja ploki tüübid, mis on polümorfsed sõlme tüübi suhtes, ja nendega seonduvad funktsioonid. Sõlme tüübi realiseerimine on kasutaja ülesanne, kusjuures tuleb täita teegi poolt seatud nõuded:

- sõlme sisenemise ja väljumise kujud peavad olema määratud,
- sõlme tüüp peab kuuluma tüübiklassi `NonLocal`.

2.3 Vahe-esituse kirjeldus

Järgnevalt otsitakse ASTi konstruktsioonidele vasted, mis rahuldavad Hoopl-teegi nõudeid vahe-esitusele.

Kuna avaldise AST võib sisaldada funktsioonide väljakutseid, mis kannavad juhtvoo üle väljakutsutava funktsiooni kehale, millest tagasipöördumisel jätkatakse avaldise arvutamist pooleli jäänud kohast, siis avaldist sisaldava lause AST ei saa olla sõlm — tagasipöördumise ehk sisenemise kohti tekiks mitu ja sõlm ei saaks omada unikaalset märgendit. Samuti ei saa funktsioonide väljakutseid teha avaldise arvutamisest väljaspool, kuna funktsioonidel võivad olla kõrvalmõjud (globaalsete muutujate muutmine, vms), mis muudaksid avaldise lõpptulemust. Seega peab iga funktsiooni väljakutse asuma eraldi sõlmes.

Koodis 2 on toodud lihtsustatud avaldise IR-i tüüp (read 1-6), mis võimaldab tervikliku avaldise ASTi jaotada mitmesse vahe-esituse sõlme. Avaldise AST läbitakse arvutamise järjekorras ning iga puu sõlme väärtus omistatakse abimuutujale `TempVar`. Funktsiooni väljakutse käsitlus on toodud edasises, kuid väljakutse lõpptulemuse väärtus omistatakse samuti `TempVar`-ile, mis võimaldab seda kasutada avaldise vahe-esituse osana. Literaalid

(`Lit`), binaarsed tehted (`BinOp`) ja tüüp `Id` on võetud muutusteta *Simple-Pascalist*. Loobutud on annotatsioonidest.

Kuna omistamislause vasaku poole AST sisaldab omakorda avaldist masiivi elemendi omistamisel, siis vahe-esituses tuleb see avaldis jaotada osadeks ning omistamislause vasakul poole IR-is viidata avaldisele vastavale abimuutujale (kood 2, rida 10). Muutuja ja kirje välja kasutamine omistamislause vasakul poole IR-is on sarnane omistamislause ASTiga (kood 2, read 9 ja 11).

Koodis 2 on toodud vahe-esituse lausele vastav sõlme tüüp, kus parameetrid `e` ja `x` näitavad vastavalt sõlme sisenemise ja väljumise kuju. Ainuke sisenemisel suletud sõlm on unikaalse märgendiga varustatud `SLabel` (rida 14). Otstest avatud sõlm `STempAssign` arvutab avaldise IR-i `SExpression` väärtuse ja omistab selle abimuutujale `TempVar` (rida 15). Sõlm `SLValAssign` omistab vasakule poolele `LVal` abimuutujale `TempVar` vastava väärtuse (rida 16). Laused `Break` ja `Continue` on vahe-esituses asendatud hüppega konkreetsele märgendile — `SGoto` (rida 18). Kui-siis-muidu lause `SIfThenElse` kannab juhtvoo vastavalt `TempVar`-iga tähistatud väärtusele üle kas esimesele (väärtus on tõene) või teisele (väär) märgendile (rida 19). Protseduuri väljakutse `SProcedure` edastab `Id`-ga tähistatud protseduuri kehale argumentide väärtustele vastavad abimuutujad ning pärast keha täitmist naaseb juhtvoog määratud märgendile (rida 20). *Simple-Pascal* ei toeta protseduuride / funktsioonide väljundparameetreid. Funktsiooni väljakutse on analoogne protseduuri väljakutsele, kuid tagastatav väärtus omistatakse `TempVar`-ile (rida 21). Funktsioonist väärtusega tagasipöördumise lause `Return` on lihtsuse huvides välja jäetud. Programmi keha (või ka protseduuri keha) lõppeb sõlmega `SExit` (rida 17), mis lõpetab programmi (protseduuri) täitmise.

Eelnevalt toodud andmekonstruktorid määravad sõlme sisenemise ja väljumise kujud. Koodis 2 ridadel 23-29 on sõlme tüüp `Stmnt` tehtud tüübiklassi `NonLocal` eksemplariks: pannakse vastavusse sisenemisel suletud sõlmede märgendid (defineerides sõlme `entryLabel`) ja väljumisel suletud sõlmedele vahetult järgnevate sõlmede märgendite hulga (defineerides sõlme `successors`). Sellega rahuldab vahe-esituse sõlme tüüp Hoopl-teegi poolt esitatud nõudeid.

Lisaks on sõlme tüüp `Stmnt` tehtud tüübiklassi `HooplNode` eksemplariks (kood 2, read 31-33). Selleks on näidatud, millist sõlme konstruktorit (`mkLabelNode`) peab funktsioon `mkLabel` kasutama, et märgendist saaks teha sisenemisel suletud graafi, mis koosneb sellest sõlmest; millist konstruktorit (`mkBranchNode`) peab funktsioon `mkBranch` kasutama, et märgendist teha väljumisel suletud graaf, mis koosneb vastavast sõlmest. Funktsioonid `mkLabel` ja `mkBranch` hõlbustavad transleerimisel graafide koostamist.

Programm vahe-esituses (kood 2) koosneb algusmärgendist (rida 35), mis vastab juhtvoo esimesele sõlmele, ning otstest suletud juhtvoo graafist (rida 39), milles sõlmeks on eespool kirjeldatud sõlme tüüp `Stmnt`. Väli `progVars` sisaldab globaalmuutujaid kodeeritud omistamislause vasakpoole jadana (rida 37). Väli `tempVars` sisaldab kasutatud abimuutujaid (rida 38). Programmi konstantide, tüüpide ja funktsioonide / protseduuride definitsioonide AST ei leia edasistes protseduurisisestes andmevooanalüüsides kasutust ja seega pole neid vahe-esitusse üle kantud.

Kood 2: Vahe-esituse abstraktne süntaksipuu

```
data SExpression =
    SVariable Id
  | SArrayItem Id TempVar
  | SRecordField Id Id
  | SLiteral Lit
  | SOp BinOp TempVar TempVar

data LVal =
    LVar Id
  | LArr Id TempVar
  | LFld Id Id

data Stmnt e x where
  SLabel :: Label → Stmnt C O
  STempAssign :: TempVar → SExpression → Stmnt O O
  SLValAssign :: LVal → TempVar → Stmnt O O
  SExit :: Stmnt O C
  SGoto :: Label → Stmnt O C
  SIfThenElse :: TempVar → Label → Label → Stmnt O C
  SProcedure :: Id → [TempVar] → Label → Stmnt O C
  SFun :: TempVar → Id → [TempVar] → Label → Stmnt O C

instance NonLocal Stmnt where
  entryLabel (SLabel l) = l
  successors SExit = []
  successors (SGoto l) = [l]
  successors (SIfThenElse _ t f) = [t, f]
  successors (SProcedure _ _ l) = [l]
  successors (SFun _ _ _ l) = [l]

instance HooplNode Stmnt where
  mkBranchNode = SGoto
  mkLabelNode = SLabel

data IProgram = IProgram
  { entry :: Label
  , progVars :: [LVal]
  , tempVars :: [TempVar]
  , body :: Graph Stmnt C C}
```

2.4 Märgendite ja abimuutujate genereerimine

Märgendite tüüp `Label` on kirjeldatud teegis Hoopl. Seda tüüpi väärtusi on võimalik genereerida unikaalsust tagavas monaadis `SimpleUniqueMonad` funktsiooniga `freshLabel`.

Avaldiste ASTide eri sõlmedes arvutatud väärtused on üldjuhul erinevad ja seega peab neile üldjuhul vastama ka erinevad abimuutujad. Sarnaselt märgenditega peab olema võimalus genereerida unikaalseid abimuutujaid. Hoopl-teegi liides ei ekspordi kõike unikaalsusmonaadiga seonduvat ja seega ei saa neid taaskasutada `TempVar`-i realiseerimiseks. Haskelli viki [3] pakub olekumonaadil põhineva unikaalsusmonaaditeisendaja `UniqueT`, milles funktsioon `fresh` tagastab unikaalse täisarvu. Teisendajale on lisatud funktsioon `used`, mis tagastab nimekirja seni kasutatud täisarvudest.

Väljumisel suletud sõlmede konstrueerimiseks vajalikud märgendid on võimalik genereerida vahetult lause ASTi transleerides, välja arvatud `Break` ja `Continue` lause puhul - nende märgendid genereeritakse tsükli lause juures. Seega tuleb neid märgendeid pidada meeles, kuni jõutakse tsüklilause transleerimisest tsüklile vastava `Break` või `Continue` lauseni. Selleks sobib kasutada keskkonnamonaadi teisendajat, kus keskkonna tüübiks on `Map String Label`.

Kuna transleerimisel on vajalik unikaalsete märgendite ja abimuutujate genereerimine ning `Break`/`Continue` märgendite meeles pidamine, siis transleerimine peab toimuma monaadis, mis need omadused ühendab. Koodis 3 on toodud `TranslationMonad`, mis kombineerib märgendite genereerimiseks `SimpleUniqueMonad`-i, abimuutujate genereerimiseks teisendaja `UniqueT` ja märgendite hoidmiseks `ReaderT` teisendaja (read 1-4). Funktsioon `freshT` tagastab uue abimuutuja (read 6-7), `usedTs` tagastab juba kasutatud abimuutujad (read 9-10), funktsioon `freshL` uue märgendi (read 12-13) ja funktsioon `getL` tagastab keskkonnast sõne argumendile vastava märgendi (read 15-16), eeldusel, et vaste leidub. Funktsioon `localL` muudab keskkonda (read 18-21), lisades sinna jada sõne-märgendi vaste paare.

Eelnevalt kasutatud funktsioon `mapUniqueT` on täiendus `UniqueT` teisendajale (kood 3, read 23-27), et saaks rakendada funktsiooni monaadilisele väärtusele jättes seejuures teisendaja sisese monaadi puutumata. Koodis 3 toodud realisatsioon üldjuhul seda eesmärki ei täida, kuna sisemist monaadi seotakse (ingl. bind ehk `>>=` ja ←) kaks korda ja seega on monaadi kõrvalmõjudega muudetud. Erijuhul, kus sisemiseks monaadiks on `ReaderT e SimpleUniqueMonad`, tähendab see, et raisatakse mõned unikaalsed märgendid. Selline kaotus ei sega transleerimist ega hilisemat analüüsimist.

2.5 ASTide tõlkimine vahe-esitusse

2.5.1 Avaldiste transleerimine

Järgnevalt on näidatud avaldise ASTi transleerimine abimuutuja ja otstest avatud graafi paariks (kood 4, rida 1). Graafi sõlmedes arvutatakse avaldise väärtuse ja omistatakse see vastavale abimuutujale. Kui avaldise annoteeritud ASTi sisuks on literaal `l` (rida 3), siis funktsiooniga `freshT` genereeritakse abimuutuja `t` (rida 4). Funktsiooniga `mkMiddle` muudetakse otstest avatud sõlm otstest avatud graafiks (read 5, 9, 13, 18, 24). Sõlm `STempAssign` omistab abimuutujale `t` IR-avaldise `SLiteral l` väärtuse (rida 5). Sarnaselt

Kood 3: Transleerimismonaad

```
type TranslationMonad =
  UniqueT (ReaderT (Map String Label) SimpleUniqueMonad)
runTranslationMonad =
  runSimpleUniqueMonad . (flip runReaderT Map.empty) . evalUniqueT

freshT :: TranslationMonad TempVar
freshT = fresh

usedTs :: TranslationMonad [TempVar]
usedTs = used

freshL :: TranslationMonad Label
freshL = lift $ lift freshLabel

getL :: String → TranslationMonad Label
getL k = lift $ asks (fromJust . Map.lookup k)

localL :: [(String, Label)] →
            TranslationMonad a → TranslationMonad a
localL ls = mapUniqueT (local
  (\e → foldl (\r (s, l) → Map.insert s l r) e ls))

mapUniqueT :: (Monad m) => (m a → m b) → UniqueT m a → UniqueT m b
mapUniqueT f (UniqueT m) = UniqueT $ mapStateT
  (\m → do (_, s) ← m
           b ← f $ m >>= return . fst
           return (b, s)) m
```

käitutakse, kui avaldise ASTis on muutuja `v` (rida 7): abimuutujale omistatakse avaldise IR-i `SVariable v` väärtus (rida 9); või kui avaldise ASTis on kirje `r` väli `f` (rida 11): abimuutujale omistatakse avaldise IR-i `SRecordField r f` väärtus (rida 13).

Kui avaldise ASTis on massiivi `a` elemendi, mille indeks on määratud avaldisega `e`, väärtus (rida 15), siis kõige pealt tuleb funktsiooniga `e2g` transleerida avaldis `e`, millele vastab abimuutuja `u` ja graaf `e'` (rida 16). Genereeritakse kogu avaldisele vastav abimuutuja `t` ja koostatakse graaf `g`, mille sõlmeks on abimuutujale `t` avaldise IR-i `SArrayItem a u` väärtuse omistamine (rida 18). Kogu avaldisele vastav graaf saadakse graafide `'e` ja `g` sidumisel avatud otstest (rida 19).

Kui avaldise ASTis on binaarne operatsioon `o` rakendatud avaldistele `el` ja `er` (rida 20), siis transleeritakse need avaldised abimuutuja-graafi paarideks: vastavalt `(tl, gl)` ja `(tr, gr)` (read 21-22). Genereeritakse kogu avaldisele vastava abimuutuja `t` (rida 23). Koostatakse graaf `g`, mille sõlmes omistatakse abimuutujale `t` vastav avaldise IR-i arvutus `SOp o tl tr`, mis teostab abimuutujatele `tl` ja `tr` vastavate väärtustega tehte `o` (rida 24). Kogu avaldisele vastav graaf saadakse graafide `gl`, `gr` ja `g` sidumisel avatud otstest (rida 25).

Kui avaldise ASTis on funktsiooni `f` väljakutse argumentide jadaga `es` (rida 26), siis tuleb transleerida argumentidele vastavad avaldised (rida 27). Argumentide avaldistele vastavad abimuutujate jada on muutuja `ts` ja graafide jada on muutuja `es'`, mis seotakse `catGraphs` funktsiooniga terviklikuks otstest avatud graafiks `es''` (rida 30). Genereeritakse märgend `l` (rida 28), kuhu juhtvoog naaseb peale funktsiooni väljakutset. Märgendist `l` tehakse `mkLabel` funktsiooniga graaf `r` (rida 32), mis on sisenemisel suletud. Funktsiooni väärtusele vastab genereeritud abimuutuja `t` (rida 29). Funktsiooni väärtus arvutatakse sõlmes `SFun t f ts l`, mis `mkLast` funktsiooniga tehakse sisenemisel avatud ja väljumisel suletud graafiks `g` (rida 31). Argumentidele vastav graaf `es''` ja funktsiooni väljakutsele vastav graaf `g` seotakse operaatoriga `<*>` ja saadud väljumisel suletud graaf seotakse sisenemisel suletud graafiga `r`, kasutades operaatorit `|*><*|` (rida 33).

2.5.2 Lausete transleerimine

Koodis 5 on näidatud lause ASTi transleerimine otstest avatud graafiks (rida 1), mille sõlmed on IR-i laused. Kui annoteeritud lause sisuks on vasakule poolele `lval` avaldise `exp` omistamine (rida 2), siis tuleb avaldis transleerida funktsiooniga `e2g` (rida 3): väärtusele vastab abimuutuja `t` ja graaf `eg`. Vasaku poole ASTile seatakse vastavusse vahe-esituse vasak pool `lv` ja sellega seotud graaf `lg` (rida 4). Kui vasakul poolel on muutuja `v` või kirje `r` väli `f` (read4-5), siis seotud graaf `lg` on tühigraaf ning vasteks `lv` on vastavalt `LVar v` või `LFld r f`. Kui vasakul poolel on massiivi `a` element (rida 7), mille indeks on määratud avaldisega `e`, siis tuleb avaldis transleerida abimuutujaks `t` ja graafiks `g`, mis on ühtlasi vasaku poolega seotud graafiks `lg`. Vasaku poole vasteks on IR-i vasak pool `LArr a t` (rida 8). Graaf `ag` on funktsiooniga `mkMiddle` koostatud sõlmest `SLValAssign lv t` (rida 9). Kogu omistamislausele vastab graaf, mis saadakse graafide `eg`, `lg` ja `ag` tervikuks sidumisest (rida 10).

Kui lause ASTis on protseduuri `p` väljakutse argumentide jadaga `es` (kood 5, rida 11), siis lause transleerimine on analoogne funktsiooni väljakutse transleerimisega avaldiste juures (kood 4, rida 25) — ainult abimuutujat ei genereerita, kuna protseduuri väljakutse ei tagasta väärtust.

Kood 4: Avaldiste transleerimine

```
e2g :: (Expression :~ a) →
         TranslationMonad (TempVar, Graph Stmnt O O)
e2g (content → Literal l) =
  do t ← freshT
     let g = mkMiddle $ STempAssign t (SLiteral l)
     return (t, g)
e2g (content → Variable v) =
  do t ← freshT
     let g = mkMiddle $ STempAssign t (SVariable v)
     return (t, g)
e2g (content → RecordField r f) =
  do t ← freshT
     let g = mkMiddle $ STempAssign t $ SRecordField r f
     return (t, g)
e2g (content → ArrayItem a e) =
  do (u, e') ← e2g e
     t ← freshT
     let g = mkMiddle $ STempAssign t $ SArrayItem a u
     return (t, e' <*> g)
e2g (content → Op o el er) =
  do (tl, gl) ← e2g el
     (tr, gr) ← e2g er
     t ← freshT
     let g = mkMiddle $ STempAssign t $ SOp o tl tr
     return (t, gl <*> gr <*> g)
e2g (content → Call f es) =
  do (ts, es') ← mapAndUnzipM e2g es
     l ← freshL
     t ← freshT
     let es'' = catGraphs es'
         g = mkLast $ SFun t f ts l
         r = mkLabel l -- where to return after call
     return (t, es'' <*> g |*><*| r)
```

Kood 5: Lausete transleerimine

```
s2g :: Statement :~ a → TranslationMonad (Graph Stmnt O O)
s2g (content → Assign lval exp) =
  do (t, eg) ← e2g exp
     (lv, lg) ← case content lval of
                   LVariable v → return (LVar v, emptyGraph)
                   LField r f → return (LFld r f, emptyGraph)
                   LArray a e → do (t, g) ← e2g e
                                   return (LArr a t, g)
     let ag = mkMiddle $ SLValAssign lv t
     return $ eg <*> lg <*> ag
s2g (content → Procedure p es) =
do (ts, es') ← mapAndUnzipM e2g es
   l ← freshL
   let es'' = catGraphs es'
       g = mkLast $ SProcedure p ts l
       r = mkLabel l -- where to return after call
   return $ es'' <*> g |*><*| r
s2g (content → IfThenElse e ts fs) =
  do endif ← freshL
     ltrue ← freshL
     lfalse ← freshL
     (t, g) ← e2g e
     tbranch ← ss2g ts
     fbranch ← ss2g fs
     return $
       g <*> mkLast (SIfThenElse t ltrue lfalse) |*><*|
       mkLabel ltrue <*> tbranch <*> mkBranch endif |*><*|
       mkLabel lfalse <*> fbranch <*> mkBranch endif |*><*|
       mkLabel endif

s2g (content → Continue) =
  do lsur ← freshL
     lcon ← getL "continue"
     return $
       mkBranch lcon |*><*|
       mkLabel lsur
s2g (content → Break) =
  do lsur ← freshL
     lbrk ← getL "break"
     return $
       mkBranch lbrk |*><*|
       mkLabel lsur
s2g (content → Exit) =
  do lsur ← freshL
     return $
       mkLast SExit |*><*| mkLabel lsur
```

Järgnevalt on käsitletud juhtu, kui lause ASTis on kui-siis-muidu hargnemine (kood 5, rida 18): kui avaldise `e` on väärtus on tõene, siis täidetakse lausete jada `ts`, vastasel juhul jada `fs`. Genereeritakse märgend `endif` (rida 19), kuhu juhtvoog naaseb peale tõese / väära haru täitmist. Kummagi lausete haru kohta genereeritakse märgend (`ltrue` / `lfalse`, read 20-21) ja transleeritakse laused graafiks (`tbranch` / `fbranch`, read 23-24). Avaldis transleeritakse abimuutujaks `t` ja graafiks `g` (rida 22), mis seotakse `SIfThenElse t ltrue lfalse` sõlmest tehtud graafiga (rida 26). Saadud graaf on väljumisel suletud ning seotakse operaatoriga `|*><*|` sisenemisel suletud graafiga, mis on tehtud märgendist `ltrue` (rida 27). Tulemuseks on väljumisel avatud graaf, mis seotakse graafiga `tbranch` ning seejärel väljumisel suletud graafiga (rida 27), mis on funktsiooniga `mkBranch` tehtud märgendist `endif`. Saadakse väljumisel suletud graaf. Sarnaselt seotakse väär haru (rida 28). Kui-siis-muidu lausele vastav graaf saadakse, sidudes seni koostatud graafile operaatoriga `|*><*|` sisenemisel suletud ja väljumisel avatud graaf `mkLabel endif` (rida 29).

Koodis 6 on transleeritud loenduriga tsükli lause (rida 1). Muutuja `i` algväärtus on määratud avaldisega `e0`. Tsükli keha ehk lausete jada `body` täidetakse, kui loendur `i` pole suurem avaldisega `eh` määratud väärtusest. Peale keha täitmist suurendatatakse loendurit `i` ühe võrra. Tsükli saab jagada osadeks: ettevalmistamine, kontroll, keha täitmine, loenduri suurendamine. Genereeritakse märgendid:

- rida 3 — `ltest` tähistab kontrolli osa; peale ettevalmistamist või loenduri suurendamist kandub juhtvoog üle kontrollile;
- rida 4 — `lbody` tähistab keha; juhtvoog kandub kehale vastavalt kontrolli tulemusele;
- rida 5 — `linc` tähistab loenduri suurendamist; peale keha või `Continue` lause täitmist kandub juhtvoog loenduri suurendamisele;
- rida 6 — `lafter` tähistab tsüklile järgnevat; juhtvoog satub siia, kas peale `Break` lause täitmist või vastavalt kontrolli tulemusele.

Tsükli osad saab koostada lausete ASTidest, mille transleerimist on kirjeldatud ülal. Funktsioonide `s2g` ja `e2g` tüübid eeldavad annoteeritud argumente. Selleks on kasutatud funktsiooni `a` (rida 8), mis annoteerib oma argumendi tsüklilause annotatsiooniga.

Loenduri `i` ülemine piir olgu salvestatud muutujasse `h` (kood 6, rida 9), mille nimi on koostatud selliselt, et vältida nimeruumi konflikti teiste muutujatega. Väärtustamiste vasakul pooltel `lvari` ja `lvarh` on vastavalt muutujad `i` ja `h` (read 10 ja 11). Lauses `s0` omistatakse muutujale `i` avaldise `e0` väärtus (rida 12) ja lauses `sh` omistatakse muutujale `h` avaldise `eh` väärtus (rida 13). Laused transleeritakse vastavalt graafideks `g0` ja `gh` (read 14 ja 15). Ettevalmistamise osa (rida 32) saadakse graafide `g0` ja `gh` sidumisest graafiga `mkBranch ltest`, mis kannab juhtvoo edasi kontrolli osale.

Kontrollis võrreldakse muutujaid `i` ja `h`. Neile vastavad avaldiste ASTid `vari` ja `varh` (kood 6, read 17 ja 18). Kontrollile vastab avaldis `test` (rida 19), mis transleeritakse kontrolli tulemust hoidvaks abimuutujaks `t` ja graafiks `gtest` (rida 20). Kontrolli lause olgu `sif` (rida 21): kui abimuutuja `t` on tõene, st muutuja `i` on suurem kui muutuja `h`, siis juhtvoog jätkab märgendist `lafter`, vastasel korral märgendist `lbody`. Kontrolli osa (rida 33) saadakse märgendist `ltest` koostatud graafi sidumisel graafidega `gtest` ja lausest `sif` tehtud graafiga.

Keha täidetakse graafis `gbody` (kood 6, read 23-24), mis saadakse funktsiooniga `localL` muudetud keskkonnas lausete jada `body` transleerimisel funktsiooniga `ss2g`. Keskkonnas

seotakse sõne „continue“ märgendiga `linc` ja sõne „break“ märgendiga `lafter` (read 23 ja 24). Graaf `gbody` seotakse vasakult keha märgendist koostatud algusgraafiga ja paremalt hargnemisega loenduri suurendamise osale (rida 34).

Loenduri suurendamiseks omistatakse lausega `sinc` vasakule poolele `lvari` (kood 6, rida 26), milles on muutuja `i`, väärtus (read 27-28), mis saadakse muutujale `i` literaali 1 liitmisel. See lause transleeritakse graafiks `ginc` (rida 29), mis seotakse vasakult algusgraafiga ja paremalt hargnemisega kontrolli osale (rida 35).

Saadud osade graafid seotakse suletud otstest terviklikuks graafiks, mis on otstest avatud ning mis vastab tsükli lausele (kood 6, read 32-36).

Koodis 5 on transleeritud juhtvoo suunamise laused: `Continue`, `Break` ja `Exit`. Esimesed 2 neist loevad keskkonnast vastava märgendi (read 33, 39) ja koostavad `mkBranch` funktsiooniga sellest väljumisel suletud graafi (read 35, 41). Kuna `s2g` funktsiooni tüüp nõuab väljumisel avatud graafi, siis genereeritakse surrogaatmärgend `lsur` (read 32, 38, 44), mis `mkLabel` funktsiooniga tehakse graafiks ja operaatoriga `|*><*|` seotakse eelnevalt koostatud graafi külge (read 35-36, 41-42, 46). Märgend `lsur` ei oma juhtvoo kulgemise seisukohalt sisu — juhtvoog ei jõua sinna kunagi ja hiljem on võimalik selline sõlm graafist eemalda. Lause `Exit` korral tehakse väljumisel suletud graaf funktsiooniga `mkLast SExit` (rida 46), mis samamoodi seotakse surrogaatmärgendist tehtud graafiga.

2.5.3 Programmi transleerimine

Järgnevalt on esitatud programmi ASTi transleerimine vahe-esitusse funktsiooniga `runAstToIr` (kood 7, read 1-2). Seejuures funktsioon `astToIr` on transleerimismonaadis (rida 4). Annoteeritud argumendi sisu sobitatakse *ViewPattern*-mustriga content. Sisust ehk programmi ASTi kirjest seotakse programmi keha `progBody` väli muutujaga `body` (rida 5) ning globaalsete muutujate deklaratsioonid `progVariables` muutujaga `vars` (rida 6). Genereeritakse märgend `lentry` (rida 7), mis on programmi vahe-esituse algusmärgendiks `entry` (rida 11), ning transleeritakse keha lausete jada body funktsiooniga `ss2g` otstest avatud graafiks `gs` (rida 8). Funktsiooniga `mkLast` muudetakse väljumisel suletud sõlm `SExit` väljumisel suletud graafiks (rida 14). Programmi IR-i kirje väli `body` on otstest suletud graaf, mis on operaatoriga `<*>` seotud kokku kolmest graafist (rida 14): `mkLabel`-iga algusmärgendist (`lentry`) tehtud algus-graafist, keha lausetele vastavast graafist `gs` ja `mkLast`-iga tehtud lõpugraafist. Programmi IR-i muutujad väljal `progVars` saadakse, rakendades f-ni `astVarToIrVars`-i igale globaalmuutuja deklaratsioonile ja aheldades saadud vasakpoolte jadad. Programmi IR-i abimuutujad `tempVars` on leitud funktsiooniga `usedTs` peale keha transleerimist. Kuigi kõik muutujad ei pruugi kehas kasutust leida, eriti pärast optimeerimisi, on see piisav edasisteks analüüsideks. Alternatiivina saaks koguda kasutatud muutujad läbides kehale vastava graafi `gs` Hoopl-teegi poolt pakutavate funktsioonidega `postorder_dfs` või `preorder_dfs` ja korrata kogumist peale optimeerimist.

Lausete ASTide jada transleeritakse otstest avatud graafiks (kood 7, rida 16), mis saadakse iga lause transleerimisel otstest avatud graafiks funktsiooniga `s2g` (rida 18) ja saadud graafide jada sidumisel funktsiooniga `catGraphs` (rida 19), mis seob jada elemendid operaatoriga `<*>` terviklikuks graafiks.

Koodis 7 on toodud tüübiinfoga annoteeritud muutuja muutmine vasakpoolte jadaks (rida 21). Kui muutuja on kirje tüüpi (rida 26), siis jadasse lisatakse muutuja nimi

Kood 6: Loendava tsükli lause transleerimine

```
s2g forStatement @ (content → For i e0 eh body) =
  do -- labels
     ltest ← freshL
     lbody ← freshL
     linc ← freshL
     lafter ← freshL
     -- initializiation part
     let a = flip Annotate $ annotation forStatement
         h = i ++ "~High"
         lvari = a $ LVariable i
         lvarh = a $ LVariable h
         s0 = a $ Assign lvari e0
         sh = a $ Assign lvarh eh
         g0 ← s2g s0
         gh ← s2g sh
     -- test part
     let vari = a $ Variable i
         varh = a $ Variable h
         test = a $ Op IsGT vari varh
         (t, gtest) ← e2g test
     let sif = SIfThenElse t lafter lbody
     -- body part
     gbody ← localL [("continue", linc)
                     , ("break", lafter)] (ss2g body)
     -- increment part
     let sinc = a $ Assign lvari $
                  a $ Op Add vari $
                    a $ Literal $ LInteger 1
     ginc ← s2g sinc
     -- loop
     return $
       g0 <*> gh <*> mkBranch ltest |*><*|
       mkLabel ltest <*> gtest <*> mkLast sif |*><*|
       mkLabel lbody <*> gbody <*> mkBranch linc |*><*|
       mkLabel linc <*> ginc <*> mkBranch ltest |*><*|
       mkLabel lafter
```

Kood 7: Programmi transleerimine

```
1  runAstToIr :: Program :~ TypeAnn → IProgram
2  runAstToIr = runTranslationMonad . astToIr
3
4  astToIr :: Program :~ TypeAnn → TranslationMonad IProgram
5  astToIr (content → Program { progVariables = vars
6                             , progBody = body }) =
7    do lentry ← freshL
8       gs ← ss2g body
9       temps ← usedTs
10      return $ IProgram
11        { entry = lentry
12        , progVars = concat $ map astVarToIrVars vars
13        , tempVars = temps
14        , body = mkLabel lentry <*> gs <*> mkLast SExit }
15
16 ss2g :: [Statement :~ a] → TranslationMonad (Graph Stmnt O O)
17 ss2g ss =
18   do gs ← mapM s2g ss
19      return $ catGraphs gs
20
21 astVarToIrVars :: Annotate Symbol TypeAnn → [LVal]
22 astVarToIrVars v =
23   let s = content v
24       t = annotation v
25   in case typeOf t of
26        TRecord fields → do f ← fields
27                            return $ LFld (symbolName s) (fst f)
28        _ → return $ LVar (symbolName s)
```

`symbolname s` kombineeritud kirje iga välja nimega `fst f` mähituna konstruktorisse `LFld` (rida 27). Ülejäänud muutuja tüüpide, sh massiivi, korral on jadaks muutuja nimest konstrueeritud `LVar` (rida 28). Kuigi massiivi jaoks on eraldi konstruktor `LArr`, pole seda kasutatud, kuna see eeldab teadmist masiivi indeksi kohta abimuutuja näol. Massiivi elemendi omistamine / kasutamine sisaldab endas aliaste probleemi: erinevad abimuutujad võivad viidata massiivi ühele indeksile ja seda tuleb optimeerimise analüüsides arvesse võtta.

3 Optimeerimine

Selle osa eesmärk on vahe-esituse optimeerimine teegiga Hoopl.

3.1 Hoopl-teegi analüüsi sooritus: andmevoo võre, ülekandefunktsioon ja vahe-esituse ümberkirjutamine

Teek Hoopl muundab juhtvoo graafe. Kasutaja ülesanne on lisaks sõlme tüübile koostada:

- faktide andmetüüp, mis on aluseks andmevoo võrele DataflowLattice ;
- ülekandefunktsioon `FwdTransfer` / `BwdTransfer`, mis sõlmest koostab fakti muundaja: sisendiks on sõlme sisenev fakt ja väljundiks on muundatud fakt;
- ümberkirjutamisefunktsioon FwdRewrite / BwdRewrite, mis sõlme ja fakti sisendi korral annab väljundiks graafi, mis asendab sõlme.

Need koondatakse analüüsi sooritusse kirjesse vastavalt analüüsi suunale: pärisuunas FwdPass ja vastusuunas BwdPass. [4, lk. 4-5]

Andmevooanalüüsi fakt mingis programmi punktis kinnitab väite kehtivust selles punktis sõltumata teest, kuidas sellesse punkti jõuti, kuid arvestades analüüsi suunda, st millises järjekorras punkte läbitakse. Faktide hulk peab moodustama võre ja selleks tuleb näidata võre alumine element `fact_bot` ja kuidas leida kahe elemendi ülemraja `fact_join`. Jõudlusest lähtudes peab `fact_join` lisaks rajale tagastama ka fakti muutumise lipu `ChangeFlag`: `SomeChange`, kui ülemraja on erinev esialgsest faktist, või `NoChange`, kui esialgne fakt ongi ülemrajaks. [4, lk. 5-6]

Hoopl-teek põimib omavahel ülekandefunktsiooniga faktide kogumise ja vahe-esituse ümberkirjutamise [4, lk. 9]. Ühes soorituses tehakse ümberkirjutamist korduvalt ja soorituse implementatsioonis vea leidmine, võrreldes vahe-esituse alg- ja lõpptulemust, ei ole lihtne. Hoopl-teek võimaldab analüüsi sooritust siluda, kasutades „kütuse“-monaadi tüübiklassi `FuelMonad`. Iga ümberkirjutamine kasutab ära ühe ühiku kütust ja ümberkirjutamist jätkatakse seni, kuni jätkub kütust. Muutes kütuse alghulka, saab kontrollida ümberkirjutamiste arvu, kuni leitakse viga. [4, lk. 11] Käesolev töö silumist ei uuri ja analüüs sooritatakse „lõputu kütuse“ monaadis `InfiniteFuelMonad`, seega ümberkirjutamist jätkatakse, kuni vahe-esituse ümberkirjutamine ei muuda tulemust.

Koodis 8 on toodud triviaalne analüüsi sooritus, mis sisuliselt ei optimeeri vahe-esitust. Sooritus `noOptPass` põhineb pärisuuna analüüsil `FwdPass` (read 1-2) ja kasutab andmevoo võret `noOptLattice` (rida 3), ülekandefunktsiooni `noOptTransfer` (rida 4) ja ümberkirjutamist `noOptRewrite` (rida 5), mis on defineeritud allpool.

Fakti andmetüüp on `NoOptFact` (kood 8, rida 7), mille ainus andmekonstruktor `NA` tähistab fakti sisutust. Andmevoo võre `noOptLattice` alumiseks elemendiks on `NA` (rida 12) ja vähim ülemtõke leitakse funktsiooniga `join` (rida 13), mis tagastab alati algse fakti ilma muutusteta (read 15-18). Võre nimi `fact_name` on toodud dokumenteerimise huvides (rida 11). Funktsiooni `join` esimene argument on mõeldud silumiseks ja näitab, millise märgendi juures toimub ülemraja leidmine.

Ülekandefunktsioon `noOptTransfer` (kood 8, read 20-21) koostatakse funktsioonist `ft`, mis on polümorfne sõlme tüübi suhtes (rida 23): `Fact` on tüübi perekond, mille eksemplarideks on üksik `NoOptFact` tüüpi fakt ja märgendite-faktide vastend `FactBase`. Väljumisel avatud sõlme puhul on tulemuseks üksik fakt. Väljumisel suletud sõlme puhul on tulemuseks märgendite ja faktide vastend, iga väljuva serva kohta üks fakt. Antud analüüsi soorituses kanname sõlme siseneva fakti `f` ilma muutusteta üle (read 24-26, 28-32). Funktsioon `mapEmpty` tähistab tühja vastendit (rida 27), `mapSingleton` vastendab

argumendid (read 31-32) ja `mkFactBase` koostab märgendi-fakti paaride jadast vastendi (read 29-30).

Vahe-esituse ümberkirjutamine toimub „kütuse“-monaadis funktsiooniga `fr` (kood 8, rida 36), seejuures `mkFRewrite` kasutamine hoolitseb, et järgitakse kütuse kasutamise reegleid. Kuna sisulist ümberkirjutamist ei toimu, siis tagastatakse sõlmest ja faktist sõltumata alati `Nothing` (rida 39), mis tähendab, et sõlm kirjutatakse muutusteta väljund-graafi.

Vahe-esituse programmi ümberkirjutamist „lõputu kütuse“ monaadis ja uute märgendite genereerimiseks `SimpleUniqueMonad`-is on tähistatud lühidalt sooritusena `Pass` (kood 8, 41). Sooritus `optNothing` seob programmi algusmärgendi muutujaga `e` ja keha muutujaga `b` (rida 44), seejärel analüüsib ja kirjutab vahe-esituse ümber funktsiooniga `analyzeAndRewriteFwd` (rida 45), kasutades:

- soorituse kirjet `noOptPass`,
- kehasse sisenemise punktide hulka, milleks antud juhul on sisenemisel suletud märgend `e`,
- keha `b`,
- algväärtustatud faktide hulka, milleks on märgendi `e` vastend faktiga `NA`.

Tulemusena saadakse ümberkirjutatud keha `body'`, mis asendab vahe-esituse programmi keha (rida 47). Kuigi sisulist ümberkirjutamist ei tehta, kantakse väljund-graafi `body'` üle ainult need plokid, mida läbiti ülekande funktsiooniga lähtudes sisenemise punktidest. Seega viskab `optNothing` sooritus minema saavutamatud plokid. Saavutamatu koodi elimineerimine võib võimaldada teiste optimeerimiste kasutamise nagu näiteks „koodi sirgendamine“ (ingl. code straightening) [2, lk. 580].

3.2 Konstandi voltimine ja levitamine

Konstandi voltimine seisneb avaldiste, mille operandideks on literaalid, arvutamises kompileerimise ajal. Konstandi levitamine tähendab muutuja, millele on omistatud literaal, kasutamise asendamist tema väärtusega, eeldusel, et muutujale pole vahepeal omistatud mõnda muud väärtust. Kuna fakte kogutakse samas järjekorras nagu läbitakse programmi täitmise ajal sõlmi, siis on tegu pärisuuna analüüsiga.

Vahe-esituses võib vaadelda kahte liiki muutujaid, mida on tähistatud kokkuvõtvalt tüübiga `Var` (kood 12, rida 1): abimuutujad, tähistatud konstruktoriga `TVar`, ja programmi muutujad, st kõik, mis saab olla omistamislause vasakul poolel, tähistatuna `PVar`. Fakt sisaldab muutuja ja literaalide vastendamist (rida 2). Kasutatud on Hoopl-teegi abitüüpi `WithTop`, mis lisab tüübile võre moodustamiseks tippelemendi. Seega, kui muutuja on „konstant“, st omab igal juhul üht kindlat väärtust, siis fakt vastandab muutujale andmekonstruktoris `PElem` mingi literaali `Lit`. Kui muutuja ei ole konstant, siis faktile vastab element `Top`. Kui muutujat pole veel analüüsitud, siis on samuti tegu mittekonstandiga ehk `Top`-iga. Kahe fakti vähim ülemtõke leitakse funktsiooniga `fact_join` (rida 8):

Kood 8: Analüüsi sooritus *no-optimization*

```
noOptPass :: FuelMonad m => FwdPass m Stmnt NoOptFact
noOptPass = FwdPass
  { fp_lattice = noOptLattice
  , fp_transfer = noOptTransfer
  , fp_rewrite = noOptRewrite }

data NoOptFact = NA -- Not Applicable

noOptLattice :: DataflowLattice NoOptFact
noOptLattice = DataflowLattice
  { fact_name = "no optimization"
  , fact_bot = NA
  , fact_join = join }
  where
    join :: Label ->
              OldFact NoOptFact -> NewFact NoOptFact ->
              (ChangeFlag, NoOptFact)
    join _ (OldFact o) _ = (NoChange, o)

noOptTransfer :: FwdTransfer Stmnt NoOptFact
noOptTransfer = mkFTransfer ft
where
  ft :: Stmnt e x -> NoOptFact -> Fact x NoOptFact
  ft (SLabel _) f = f
  ft (STempAssign _ _) f = f
  ft (SLValAssign _ _) f = f
  ft (SExit) _ = mapEmpty
  ft (SGoto l) f = mapSingleton l f
  ft (SIfThenElse _ tl fl) f = mkFactBase noOptLattice [ (tl, f)
                                                       , (fl, f)]
  ft (SProcedure _ _ l) f = mapSingleton l f
  ft (SFun _ _ _ l) f = mapSingleton l f

noOptRewrite :: forall m. FuelMonad m =>
                  FwdRewrite m Stmnt NoOptFact
noOptRewrite = mkFRewrite fr
  where
    fr :: Stmnt e x -> NoOptFact -> m (Maybe (Graph Stmnt e x))
    fr _ _ = return Nothing

type Pass = IProgram -> InfiniteFuelMonad SimpleUniqueMonad IProgram

optNothing :: Pass
optNothing p@(IProgram { entry = e, body = b}) =
  do (body', _, _) <- analyzeAndRewriteFwd
                        noOptPass (JustC [e]) b (mapSingleton e NA)
     return $ p { body = body' }
```

Kood 9: Konstandi voltimine ja levitamine — andmevoo võre

```
1  data Var = TVar TempVar | PVar LVal
2  type ConstFact = Map Var (WithTop Lit)
3
4  constLattice :: DataflowLattice ConstFact
5  constLattice = DataflowLattice
6    { fact_name = "temp ← literal"
7    , fact_bot = Map.empty
8    , fact_join = joinMaps (extendJoinDomain constFactAdd) }
9    where
10     constFactAdd _ (OldFact old) (NewFact new)
11       = if new == old then (NoChange, PElem new)
12         else (SomeChange, Top)
```

- `constFactAdd` ühendab omavahel kaks literaali, andes tulemuseks `Top`, kui need erinevad, ja `PElem` literaali, kui nad on samad,
- `extendJoinDomain` laiendab ühendamise tippelementi säilitades kogu võrele, st kui kasvõi üks ühendatavatest on tippelement, siis on ka ühendamise tulemuseks tippelement,
- `joinMaps` ühendab faktide vastandamised, kasutades elementide ühendamiseks eespool toodud funktsioone.

Fakti tüübist ja faktide ühendamise funktsioonist koostatakse võre `constLattice`, mille põhjas on tühi vastandus.

Koodis 10 on toodud ülekandefunktsioon `varHasLit` (rida 1).

Abimuutujale `t` omistamine sõlmes `STempAssign` (kood 10, rida 5), lisab fakti `f` teadmise selle muutuja `TVar t` kohta: kui lihtsustatud avaldis `e` on literaal `x` (rida 7), siis on tegu konstandiga `PElem x`, vastasel korral muutuja ei ole konstant (rida 8), mida tähistatakse tippelemendiga `Top`.

Vasakule poolele `lv` abimuutuja `t` omistamisel sõlmes `SLValAssign` (kood 10, rida 9) muudetakse faktis `f` vasaku poole `PVal lv` vaste samaväärseks abimuutuja `TVar t` vastega. Kuna fakt on algväärtustatud, st abimuutujale leidub vaste, saab kasutada funktsiooni `fromJust` otsingu tulemuse tõlgendamiseks (rida 10).

Kui vaadeldavaks sõlmeks on kui-siis-muidu hargnemine (kood 10, rida 14), siis funktsiooniga `mkFactBase` koostatakse märgendite-faktide vastend (rida 15), kusjuures tõese haru, märgendi `tl`, korral lisandub fakti `f` teadmine abimuutuja `t` tõesuse kohta (rida 16) ning väära haru, märgendi `fl`, korral vastavalt vääruse kohta (rida 17).

Protseduuri / funktsiooni väljakutse või hüpe märgendile ei lisa teadmisi muutujate kohta ja seega kannab ülekandefunktsioon fakti `f` muutumatult edasi funktsiooniga `mapSingleton` märgendile `l` (kood 10, read 13, 18, 19).

Märgendiga plokki sisenemine sõlmega `SLabel` ei muuda fakti `f` (kood 10, rida 11).

Protseduuri lõpetamine sõlmega `SExit` paneb piiri faktide kogumisele funktsiooniga `mapEmpty` (kood 10, rida 12).

Kood 10: Konstandi voltimine ja levitamine — ülekandefunktsioon

```
1  varHasLit :: FwdTransfer Stmnt ConstFact
2  varHasLit = mkFTransfer ft
3    where
4      ft :: Stmnt e x → ConstFact → Fact x ConstFact
5      ft (STempAssign t e) f =
6        Map.insert (TVar t) (case e of
7                               SLiteral x → PElem x
8                               _ → Top) f
9      ft (SLValAssign lv t) f = Map.insert (PVar lv)
10                                  (fromJust $ Map.lookup (TVar t) f) f
11     ft (SLabel _) f = f
12     ft (SExit) _ = mapEmpty
13     ft (SGoto l ) f = mapSingleton l f
14     ft (SIfThenElse t tl fl) f =
15       mkFactBase constLattice
16         [ (tl, Map.insert (TVar t) (PElem (LBool True)) f)
17         , (fl, Map.insert (TVar t) (PElem (LBool False)) f) ]
18     ft (SProcedure _ _ l) f = mapSingleton l f
19     ft (SFun _ _ _ l) f = mapSingleton l f
```

Koodis 11 on toodud vahe-esituse ümberkirjutamine `simplify`.

Kui sõlmeks on kui-siis-muidu hargnemine vastavalt abimuutuja `t` tõeväärtusele (kood 11, rida 6) ja abimuutujale `t` leitakse funktsiooniga `lookup` faktist `f` konstantne vaste (rida 7), siis asendatakse kogu sõlm hüppega konkreetsele harule (read 8-9): vastavalt `t` tõeväärtusele kas tõene haru `tl` või väär haru `fl`. Kui vastet ei leita, siis `Maybe` monaad annab tulemuseks `Nothing`, mida Hoopl-teek tõlgendab teadmisena, et sõlme ei muudetud.

Kui sõlmeks on abimuutujale `t` tehte `SOp o l r` tulemuse omistamine (kood 11, rida 10) ning abimuutujatele `l` ja `r` leidub mõlemale konstantne vaste (read 11 ja 12), vastavalt `l'` ja `r'`, siis funktsiooniga `eval` arvutatakse operatsiooni `o l' r'` tulemus, mis omistatakse abimuutujale `t` kui literaal (rida 13). Tulemuse arvutamisel kompileerimise ajal tuleb arvestada eriolukordade: nulliga jagamine, ületäitumine, ujukomakäsu erandid, jm [2, lk. 329-331]. Käesolev töö ei käsitle mainitud situatsioone.

Kui abimuutujale `t` omistatakse lihtsustatud avaldis `e` (kood 11, rida 14), mis valvuriga `isExprVar` kontollitult sisaldab programmi muutujat (rida 15), siis f-niga `toPVar` teisendatakse avaldis `e` programmi muutujaks, millele otsitakse faktist `f` vaste `x` (rida 16). Kui `x` on konstant, siis asendatakse kogu sõlm abimuutujale `t` literaali `SLiteral x` omistamisega (rida 17).

Ülejäänud sõlmede puhul ümberkirjutamist ei tehta (kood 11, rida 18).

Funktsioon `lookup` otsib muutujale `t` faktist `f` vaste (kood 11, read 20-22): tulemuseks on kas literaal `v` mähituna andmekonstruktorisse `Just` (rida 23) või muutuja mittekonstantsuse puhul `Nothing` (rida 24).

Funktsioon `returnG` muudab sõlme tüüpi argumendi graafiks ja tagastab selle `Maybe`-monaadis (kood 11, read 8, 13, 17).

Kood 11: Konstandi voltimine ja levitamine — ümberkirjutamine

```
simplify :: forall m. FuelMonad m => FwdRewrite m Stmnt ConstFact
simplify = mkFRewrite fold
  where
    fold :: Stmnt e x → ConstFact → m (Maybe (Graph Stmnt e x))
    -- fold if-then-else into goto
    fold (SIfThenElse t tl fl) f = return $
      do (LBool t') ← lookup (TVar t) f
         returnG $ SGoto $
           if t' then tl else fl
    fold (STempAssign t (SOp o l r)) f = return $
      do l' ← lookup (TVar l) f
         r' ← lookup (TVar r) f
        returnG $ STempAssign t $ SLiteral (eval (o, l', r'))
    fold (STempAssign t e) f
      | isExprVar e = return $
        do x ← lookup (toPVar e) f
           returnG $ STempAssign t $ SLiteral x
    fold _ _ = return Nothing

lookup :: Var → ConstFact → Maybe Lit
lookup t f =
  case Map.lookup t f of
    Just (PElem v) → Just v
    _ → Nothing
```

Kood 12: Konstandi voltimine ja levitamine — sooritus

```
1  optConst :: Pass
2  optConst p@(IProgram { entry = e
3                       , progVars = vs
4                       , tempVars = ts
5                       , body = b}) =
6    do (body', _, _) ← analyzeAndRewriteFwd
7                          constPropPass (JustC [e]) b $
8                          mapSingleton e (initFact (vs, ts))
9       return $ p { body = body' }
10
11 initFact :: ([LVal], [TempVar]) → ConstFact
12 initFact (vs, ts) = Map.fromList $
13   [(TVar t, Top) | t ← ts] ++
14   [(PVar v, Top) | v ← vs]
```

Koodis 12 on toodud konstandi voltimise ja levitamise sooritus `optConst`. Võrreldes `optNothing` sooritusega (kood 8, read 42-46), on kasutatud sooritusе kirjet `constPropPass`, mis koondab võre `constLattice`, ülekandefunktsiooni `varHasLit` ja ümberkirjutamise `simplify`, ning algväärtustatud fakti `initFact` (kood 12, read 11-14), mis saadakse programmimuutujate `ps` ja abimuutujate `ts` vastendamisel tippelemendiga `Top`, st kõik muutujad on alguses mittekonstantsed.

3.3 Koopia levitamine

Kui programmi muutujale on omistatud mõni teine programmi muutuja, siis sisaldab muutuja originaalmuutuja väärtuse koopiat ning järgnevates sõlmedes võib muutuja kasutamise asendada originaalmuutuja kasutamisega, juhul kui originaali pole vahepeal muudetud. Analüüsi suund ühtib normaalse programmi täitmise suunaga.

Vaatluse all on Pascal-keele laused kujul `X:=Y`, kuid vahe-esituses on nad kujul `t←Y; X←t`. Seega peab andmevoo fakt talletama muutujate, nii abi- kui ka programmimuutujate, vastandamised. Kuna originaalmuutujale võidakse omistada uus väärtus, siis tuleb faktis vaste tühistada, arvestades andmevoograafi serval tühistamist liitumissõlmes. Selleks on vaste mähitud `WithTop` tüübikonstruktorisse.

Koodis 13 on toodud fakti tüüp `CopyFact` (rida 1) ja sellega seonduvad abifunktsioonid: `copy` lisab fatki `f` vaste muutuja `src` kui originaali ja muutuja `dst` kui koopia vahel (read 3-4), `kill` tühistab faktis `f` muutuja `dst` vaste (read 6-7), muutes selle `Top`-iks.

Vahe-esituses tehakse vahet abi- ja programmimuutuja tüüpidel, st üks ei saa olla sõlmes seal, kus saab teine, seega ümberkirjutamisel tuleb muutuja asendada sama tüüpi muutujaga. Kuna omistamislausetes ei saa muutujale omistada sama tüüpi muutujat, siis on faktis alati vastendatud abimuutujat programmimuutujaga või vastupidi. Seega sama tüüpi koopia leidmiseks tuleb otsida topelt. Funktsioon `lookupCopy` otsib faktist `f` muutuja `v` originaali `v'`, millele otsitakse omakorda originaal `v''`, mis on sama tüüpi kui muutuja `v` (kood 13, read 9-12). Kui sellist koopiate ahelat ei leidu, siis tagastatakse `Nothing`.

Kood 13: Koopia levitamine — andmevoo fakt

```
1  type CopyFact = Map Var (WithTop Var)
2
3  copy :: Var → Var → CopyFact → CopyFact
4  copy src dst f = Map.insert dst (PElem src) f
5
6  kill :: Var → CopyFact → CopyFact
7  kill dst f = Map.insert dst Top f
8
9  lookupCopy :: CopyFact → Var → Maybe Var
10 lookupCopy f v = do PElem v' ← Map.lookup v f
11                    PElem v'' ← Map.lookup v' f
12                    return v''
13
14 lookupTemp :: CopyFact → TempVar → TempVar
15 lookupTemp f t = fromTVar $ fromMaybe (TVar t) $
16   lookupCopy f (TVar t)
17
18 lookupTemps :: CopyFact → [TempVar] → [TempVar]
19 lookupTemps f = map $ lookupTemp f
```

Abimuutuja otsimiseks on lisaks funktsioon `lookupTemp` (kood 13, read 14-16), mis tagastab faktist `f` abimuutuja `t` originaali või `t` enda, kui ei leidunud kopeeritud abimuutujat. Funktsioon `lookupTemps` laiendab f-ni `lookupTemp` abimuutujate jadale (read 18-19).

`CopyFact` ja `ConstFact` on oma ülesehituselt sarnased ning neist konstrueeritud võred, `copyLattice` ja `constLattice`, on samuti sarnased — siinkohal pole `copyLattice` esitatud.

Koodis 14 on toodud ülekandef-n `copyTransfer`. Sõlmes `STempAssign` abimuutujale `t` avaldise `e` omistamisel (rida 5):

- kui avaldis `e` sisaldab programmimuutujat, siis f-niga `copy` tehakse koopia programmimuutujast abimuutujasse (rida 6),
- vastasel korral, nt kui avaldiseks on tehe või literaal, tühistatakse f-niga `kill` abimuutuja `t` vaste (rida 7).

Koopia levitamises esineb aliaste probleem massiivide korral. Kuna faktist `CopyFact` otsimine toimub muutujate järjekorra alusel ja muutujaid järjestatakse nimeliselt, mitte aga väärtuseliselt, siis: a) väheneb analüüsi agressiivsus, st faktist ei pruugi leida kõiki koopiaid, b) analüüs ei säilita konservatiivsust, sest vaste tühistamine ei toimi, ja optimeerimine võib anda vale tulemuse. Antud töös on aliaste analüüsil põhinevaid optimeerimisi välditud säilitades konservatiivsus, ent ohverdades agressiivsust: sõlmes massiivi elemendile omistamisel ei muudeta teadmisi koopiate kohta (kood 14, rida 8).

Teistes vasakpooltele omistamistel sõlmes `SLValAssign` lisatakse fakti `f` f-niga `copy` abimuutujast `t` koopia vasakusse poolde `lv` (kood 14, rida 9).

Kood 14: Koopia levitamine — ülekandefunktsioon

```
1  copyTransfer :: FwdTransfer Stmnt CopyFact
2  copyTransfer = mkFTransfer ft
3    where
4      ft :: Stmnt e x → CopyFact → Fact x CopyFact
5      ft (STempAssign t e) f
6        | isExprVar e = copy (toPVar e) (TVar t) f
7        | otherwise = kill (TVar t) f
8      ft (SLValAssign (LArr _ _) _) f = f
9      ft (SLValAssign lv t) f = copy (TVar t) (PVar lv) f
10     ft (SLabel _) f = f
11     ft (SExit) _ = mapEmpty
12     ft (SGoto l ) f = mapSingleton l f
13     ft (SIfThenElse _ tl fl) f = mkFactBase copyLattice [ (tl, f)
14                                                         , (fl, f)]
15     ft (SProcedure _ _ l) f = mapSingleton l f
16     ft (SFun t _ _ l) f = mapSingleton l f
```

Eeldusel, et abimuutujale omistatakse väärtus ainult üks kord, pole vaja sõlmes `SFun` abimuutuja vastet tühistada (kood 14, rida 16), sest see on nii juba fakti algväärtustamisel. Kuna vahe-esituse koostamisel luuakse igale abimuutujale omistamisel unikaalne abimuutuja ja käesoleva töö ümberkirjutamistes ei muudeta abimuutujate omistamist, siis see eeldus on täidetud.

Ülejäänud sõlmedes ei toimu muutujale väärtuse omistamist ja seega kantakse fakt muutusteta üle (kood 14, read 10, 12, 15, 16).

Koodis 15 on toodud koopia levitamise ümberkirjutamine `copyRewrite`. Kui sõlmes `STempAssign` omistatakse abimuutujale `t` avaldis `e` (rida 5), mis sisaldab programmi muutujat, siis otsitakse faktist `f` kopeeritud programmi muutuja `lv`, kasutades f-ni `lookupCopy` (rida 6). Kui see leitakse, tagastatakse ümberkirjutatud sõlm `STempAssign` (rida 7), milles omistatakse abimuutujale `t` programmimuutujast `lv` f-niga `fromLVal` koostatud IR-avaldis.

Kui sõlmes `STempAssign` omistatakse abimuutujale `t` tehte `o` operandidega `l` ja `r` väärtus (kood 15, rida 8), siis f-niga `lookupTemp` otsitakse võimalusel kopeeritud abimuutujad, vastavalt `l'` ja `r'` (read 9 ja 10), ja sõlm kirjutatakse ringi leitud operandidega (rida 11).

Sõlme `SLValAssign`, kus vasakul poolel on massiivi element indeksiga `i`, saab ringi kirjutada, kui leidub originaalne abimuutuja `i'`, mida `i` kopeerib (kood 15, read 12-15).

Sõlme `SLValAssign`, kus vasakule poolele omistatakse abimuutuja `t` väärtus (kood 15, rida 16), saab ümberkirjutada, kui `t` on koopia abimuutujast `t'` (rida 17). See reegel kehtib ka masiivi elemendi omistamise kohta.

Sõlme `SIfThenElse`, mille tõeväärtus on määratud abimuutujaga `t` (kood 15, rida 19), saab ümberkirjutada, kui leidub kopeeritud abimuutuja `t'` (rida 20).

Protseduuride ja funktsioonide sõlmed kirjutatakse ringi (kood 15, read 22-27), muutes argumentide jada `ts` f-niga `lookupTemps`, mis on kirjeldatud eespool.

Kood 15: Koopia levitamine — ümberkirjutamine

```
1  copyRewrite :: forall m. FuelMonad m => FwdRewrite m Stmnt CopyFact
2  copyRewrite = mkFRewrite rw
3    where
4      rw :: Stmnt e x → CopyFact → m (Maybe (Graph Stmnt e x))
5      rw (STempAssign t e) f | isExprVar e = return $
6        do (PVar lv) ← lookupCopy f (toPVar e)
7           returnG $ STempAssign t (fromLVal lv)
8      rw (STempAssign t (SOp o l r)) f = return $
9        let l' = lookupTemp f l
10           r' = lookupTemp f r
11       in returnG $ STempAssign t (SOp o l' r')
12     rw (SLValAssign (LArr a i) t) f
13       | i /= i' = return $
14           returnG $ SLValAssign (LArr a i') t
15         where i' = lookupTemp f i
16     rw (SLValAssign lv t) f = return $
17       do (TVar t') ← lookupCopy f (TVar t)
18          returnG $ SLValAssign lv t'
19     rw (SIfThenElse t tl fl) f = return $
20       do (TVar t') ← lookupCopy f (TVar t)
21          returnG $ SIfThenElse t' tl fl
22     rw (SProcedure p ts l) f = return $
23       let ts' = lookupTemps f ts
24       in returnG $ SProcedure p ts' l
25     rw (SFun t p ts l) f = return $
26       let ts' = lookupTemps f ts
27       in returnG $ SFun t p ts' l
28     rw _ _ = return Nothing
```

Ülejäänud sõlmedes ümberkirjutamist ei tehta (kood 15, rida 28).

Koopia levitamise analüüsi sooritus `optCopy` on sarnane sooritusega `optConst`: muutub kasutatud võre ja fakti tüüp, kuid graafi sisenemispunkt ja fakti algväärtustamine jääb samaks.

3.4 Ühiste alamavaldiste eemaldamine

Kui avaldised sisaldavad ühiseid alamavaldisi, siis eeldusel, et alamavaldiste tulemus vahepeal ei muutu, võib nende alamavaldiste korduva arvutamise asemel arvutada neist esimese, salvestada tulemus ja järgnevate arvutamisel kasutada salvestatud väärtust. [2, lk. 378] märgib, et mõningates olukordades võib tulemuse salvestamine olla kulukam kui uuesti arvutamine. Käesoleva töö vahe-esituses on avaldised nagunii lõhutud lihtsustatud alamavaldiseks, millede tulemused salvestatakse abimuutujatesse, ja seega antud optimeerimine ei tekita kulu registrite või mälupöördumiste kasutamise kaudu.

Kood 16: Ühiste alamavaldiste eemaldamine — andmevoo fakt

```
1  data AvailExprFact = AvailExprFact
2    { refs :: Map TempVar SExpression
3    , firstUsage :: Map SExpression (Maybe TempVar)
4    }
5
6  join _ (OldFact old) (NewFact new)
7    = let combine = \u v → if u == v then v else Nothing
8          ref = Map.union (refs new) (refs old)
9          first = Map.unionWith combine
10                   (firstUsage new) (firstUsage old)
11         f = AvailExprFact { refs = ref, firstUsage = first }
12         ch = changeIf (old /= f)
13     in (ch, f)
14
15 lookupFirst :: AvailExprFact → TempVar → Maybe TempVar
16 lookupFirst f t =
17   do w ← Map.lookup t (refs f)
18      v ← Map.lookup w (firstUsage f)
19      v
```

Andmevoo faktis `AvailExprFact` vastendatakse väljal `refs` abimuutujad ja seniseks arvutatud lihtsustatud avaldised, mille väärtused on neisse abimuutujatesse salvestatud (kood 16, rida 2). Väli `firstUsage` vastendab lihtsustatud avaldise ja esimese abimuutuja (rida 3), kuhu arvutatud väärtus salvestati. Kasutatud on `Maybe` mähist, et tähistada vasakule poolele omistamisele vastava avaldise tühistamist konstruktoriga `Nothing`. Kuna abimuutujatele antakse väärtus ainult üks kord, siis `refs` välja puhul pole tühistamine oluline.

Koodis 16 on näidatud võre `availLattice` ülemraja leidmine f-niga `join` (rida 6). Faktide `old` ja `new` väljad `refs` liidetakse muutujaks `ref` (rida 8), väljade `firstUsage` liitmisel muutujaks `first` tuleb arvestada, et kui vähemalt ühes neist on vaste tühistatud, siis liitunud vastendis oleks samuti vaste `Nothing` (read 9-10). Muutujatest `ref` ja `first` koostatakse fakt `f` (rida 11), mis on ülemrajaks (rida 13). Vihje `ChangeFlag` saadakse faktide `old` ja `f` võrdlemisest (rida 12).

Faktist otsimine käib funktsiooniga `lookupFirst` (kood 16, rida 15): väljalt `refs` otsitakse abimuutujale `t` vastav lihtsustatud avaldis `w` (rida 17), millele otsitakse väljalt `firstUsage` esimene abimuutuja `v` (rida 18), kuhu avaldise väärtus on salvestatud. Ümberkirjutamises saab abimuutuja `t` asemel kasutada abimuutujat `v`, kui selline leidub. Sarnaselt koopia levitamise optimeerimisega, on kasulikud abif-nid `lookupTemp`, mis eemaldab `lookupFirst` mähisest `Maybe`, ja `lookupTemps`, mis toimib abimuutujate jadal.

Koodis 17 on toodud ülekandef-n `availTransfer`. Aliaste probleemi vältimiseks ei muuda abimuutujale massiivi elemendile omistamine fakti (rida 5).

Abimuutujale teiste lihtsustatud avaldiste väärtuste omistamine lisab fakti väljale `refs` vaste abimuutuja `t` ja lihtsustatud avaldise `e` vahel (kood 17, rida 7). Väljale `firstUsage`

Kood 17: Ühiste alamavaldiste eemaldamine — ülekandefunktsioon

```
1  availTransfer :: FwdTransfer Stmnt AvailExprFact
2  availTransfer = mkFTransfer ft
3    where
4      ft :: Stmnt e x → AvailExprFact → Fact x AvailExprFact
5      ft (STempAssign _ (SArrayItem _ _)) f = f
6      ft (STempAssign t e) f =
7        f { refs = Map.insert t e (refs f)
8          , firstUsage = Map.insertWith
9              (\n o → if o == Nothing then n else o)
10             e (Just t) (firstUsage f) }
11     ft (SLValAssign lv _) f =
12       f { firstUsage = Map.insert (fromLVal lv)
13                           Nothing (firstUsage f) } -- kill
14     ft (SLabel _) f = f
15     ft (SExit) _ = mapEmpty
16     ft (SGoto l ) f = mapSingleton l f
17     ft (SIfThenElse _ tl fl) f = mkFactBase availLattice [ (tl, f)
18                                                          , (fl, f)]
19     ft (SProcedure _ _ l) f = mapSingleton l f
20     ft (SFun _ _ _ l) f = mapSingleton l f
```

salvestatakse seos avaldise `e` ja abimuutuja `t` vahel, kui seal juba ei ole mõnda muud abimuutujat (read 8-10).

Vasakule poolele omistamine tühistab fakti väljal `firstUsage` programmi muutujale vastava lihtsustatud avaldise (kood 17, read 11-13).

Ülejäänud sõlmedes kantakse fakt muutusteta üle nagu pärisuuna analüüsi ülekandefnides `noOptTransfer` või `copyTransfer`. Kuigi funktsiooni väljakutse sõlmes `SFun` omistatakse abimuutujale väljakutse tulemus, siis pole faktis selle abimuutuja tühistamine oluline, kuna abimuutujat väärtustatakse ainult üks kord. Väljakutse tulemust ei tohi salvestada hilisemaks ümberkirjutamiseks, kuna funktsiooni väljakutse võib omada kõrvalmõjusid ja seega tuleb iga väljakutse uuesti täita.

Koodis 18 on toodud ülekandef-n `commonRewrite`. Kui sõlmes kasutatakse abimuutujat ja leidub mõni muu abimuutuja, mille väärtus on sama (read 9, 11, 15, 16, 18, 22, 24, 27), siis saab sõlme ringi kirjutada, kasutades leitud abimuutujat. Võrreldes `copyRewrite`-ga ei otsita programmimuutujatele ümberkirjutamiseks asendust.

3.5 Surnud omistamiste eemaldamine

Omistamise sõlm on surnud, kui omistatud muutuja on selles sõlmes surnud. Muutuja on sõlmes surnud, kui järgnevates sõlmedes, kuni muutujale uue väärtuse omistamiseni või muutuja skoobi lõpuni, ei ole muutujat kasutatud. Surnud sõlmed võib graafist eemaldada, muutmata programmi tähendust.

Kood 18: Ühiste alamavaldiste eemaldamine — ümberkirjutamine

```
commonRewrite :: forall m. FuelMonad m =>
                   FwdRewrite m Stmnt AvailExprFact
commonRewrite = mkFRewrite rw
  where
    rw :: Stmnt e x → AvailExprFact → m (Maybe (Graph Stmnt e x))
    rw (SLValAssign (LArr a i) t) f
      | i /= i' = return $
        returnG $ SLValAssign (LArr a i') t
      where i' = lookupTemp f i
    rw (SLValAssign lv t) f = return $
      do t' ← lookupFirst f t
         returnG $ SLValAssign lv t'
    rw (STempAssign t (SOp o l r)) f = return $ returnG $
      STempAssign t $ SOp o l' r'
        where l' = lookupTemp f l
              r' = lookupTemp f r
    rw (STempAssign t (SArrayItem a i)) f = return $
      do i' ← lookupFirst f i
         returnG $ STempAssign t (SArrayItem a i')
    rw (SIfThenElse t tl fl) f = return $
      do t' ← lookupFirst f t
         returnG $ SIfThenElse t' tl fl
    rw (SProcedure p ts l) f = return $ returnG $
      let ts' = lookupTemps f ts
      in SProcedure p ts' l
    rw (SFun t p ts l) f = return $ returnG $
      let ts' = lookupTemps f ts
      in SFun t p ts' l
    rw _ _ = return Nothing
```

Kood 19: Surnud omistamiste eemaldamine — andmevoo võre

```
1  type Live = Set Var
2
3  liveLattice :: DataflowLattice Live
4  liveLattice = DataflowLattice
5    { fact_name = "Live variables"
6    , fact_bot = Set.empty
7    , fact_join = add
8    }
9    where add _ (OldFact old) (NewFact new) = (ch, j)
10         j = new `Set.union` old
11         ch = changeIf (Set.size j > Set.size old)
```

Liikudes graafis vastupidiselt normaalse programmi täitmise suunaga, saab faktis talletada elus muutujad, st muutujad, millede väärtust kasutatakse, ja muutujale omistamisel selle suretada, st eemaldada faktist. Koodis 19 on esitatud andmevoo fakt `Live` (rida 1) ja sellest ehitatud võre `liveLattice` (rida 3). Võre alumine element on tühi hulk (rida 6) — kõik muutujad on surnud. Kahe fakti vähim ülemtõke leitakse hulkade ühendamisega (rida 10), kusjuures `ChangeFlag` tüüpi lipp `ch` saadakse Hoopl-teegi f-niga `changeIf` hulkade suuruste võrdlusest (rida 11).

Vastusuuna analüüsi ülekandefunktsioon `liveness` koostatakse f-niga `mkBTransfer` (kood 20, rida 2). Võrreldes pärisuuna analüüsi f-niga `mkFTransfer` on parameetri tüüp teine. Pärisuuna analüüsis on sisendiks sõlm ja sõlme sisenev fakt ning väljundiks tüübi perekonna `Fact` eksemplar sõlmest väljumisel; vastusuuna analüüsis on sisendiks sõlm ja `Fact` eksemplar sõlmest väljumisel ning väljundiks fakt sõlme kohta (rida 4). Järgnevalt on kirjeldatud vastusuuna analüüsi osa, funktsiooni `live`, juhtumid.

Sõlm `SLabel` ei muuda muutuja elusust ja seega fakti ei muudeta (kood 20, rida 5).

Sõlmes `STempAssign` suretatakse (kood 20, rida 6), st eemaldatakse hulgast, abimuutuja `t` faktist `f`, saades fakt `f'` (rida 13), milles elustatakse, st lisatakse hulka, vastavalt avaldises `e` leiduvad abi- või programmimuutujad (read 8, 10, 11). Massiivimuutuja lisatakse hulka kui tavaline muutuja (rida 9), vältimaks aliaste probleemi elemendile omistamise ümberkirjutamises.

Omistades vasakule poolele `lv` abimuutuja `t` väärtus (kood 20, rida 15), lisatakse `t` fakti, kust seejärel kustutatakse vasakust pooles olev programmimuutuja (rida 18). Kui vasakul poolel on massiivi element, siis lisatakse indeks `i` abimuutujana hulka, kuid massiivimuutujat hulgast ei eemaldata (rida 17). Seega, kui kasvõi ühte elementi massiivist kasutatakse, on kogu massiiv koos kõigi elementidega elus.

Väljumisel suletud sõlme korral (kood 20, read 20, 21, 23, 25) on `live` teiseks sisendiks märgendite-faktide vastend `FactBase`, kust Hoopl-teegi f-niga `lookupFact` leitakse konkreetsele märgendile vastav fakt. F-n `fact` täiendab f-ni `lookupFact` vaikeväärtusega, tühi hulk, kui märgendi vaste puudus (read 30-31). Sõlmes `SGoto` otsitakse märgendile `l` vaste ning kuna sõlm ei muuda muutujate elusust, ongi see väljundiks (rida 20).

Kui-siis-muidu sõlmes (kood 20, rida 21) ühendatakse tõese ja väära haru märgenditele

Kood 20: Surnud omistamiste eemaldamine – ülekandefunktsioon

```
liveness :: BwdTransfer Stmnt Live
liveness = mkBTransfer live
  where
    live :: Stmnt e x → Fact x Live → Live
    live (SLabel _) f = f
    live (STempAssign t e) f =
      case e of
        SVariable _ → Set.insert (toPVar e) f'
        SArrayItem a i → Set.insert (PVar (LVar a)) $
                         Set.insert (TVar i) f'
        SRecordField _ _ → Set.insert (toPVar e) f'
        SOp _ l r → Set.insert (TVar l) $ Set.insert (TVar r) f'
        _ → f'
      where f' = Set.delete (TVar t) f
    live (SLValAssign lv t) f =
      case lv of
        LArr _ v → Set.insert (TVar v) f'
        _ → Set.delete (toPVar $ fromLVal lv) f'
      where f' = Set.insert (TVar t) f
    live (SGoto l) f = fact f l
    live (SIfThenElse t tl fl) f =
      Set.insert (TVar t) (fact f tl `Set.union` fact f fl)
    live (SProcedure _ ts l) f =
      fact f l `Set.union` Set.fromList (map TVar ts)
    live (SFun t _ ts l) f =
      (Set.delete (TVar t) (fact f l)) `Set.union`
      Set.fromList (map TVar ts)
    live SExit _ = Set.empty

fact :: FactBase Live → Label → Live
fact f l = fromMaybe Set.empty $ lookupFact l f
```

Kood 21: Surnud omistamiste eemaldamine — ümberkirjutamine

```
1  deadAsstElim :: forall m . FuelMonad m => BwdRewrite m Stmnt Live
2  deadAsstElim = mkBRewrite d
3    where
4      d :: Stmnt e x → Fact x Live → m (Maybe (Graph Stmnt e x))
5      d (STempAssign t _) live
6        | not ((TVar t) `Set.member` live) = return $ Just emptyGraph
7      d (SLValAssign (LArr a _) _) live = return $
8        if (PVar (LVar a)) `Set.member` live
9        then Nothing
10       else Just emptyGraph
11     d (SLValAssign lv _) live
12       | not ((PVar lv) `Set.member` live) = return $ Just emptyGraph
13     d _ _ = return Nothing
```

vastad faktid ning lisatakse sinna tõeväärtuse abimuutuja kui elus muutuja (rida 22).

Protseduuri ja funktsiooni väljakutse lisavad väljakutsele järgnevale märgendile vastavale faktile argumentide jada abimuutujad (kood 20, read 24, 26-27). Funktsiooni väljakutse lisaks suretab abimuutuja (rida 26), kuhu salvestatakse väljakutsel tagastatav väärtus.

Sõlmele `SExit` vastab alati tühi hulk elusaid muutujaid (kood 20, rida 28), kuna pole järgmist sõlme, kus neid saaks kasutada.

Ümberkirjutamine `deadAsstElim` on toodud koodis 21. Funktsioon `mkBRewrite` tagab kütuse reeglitepärase kasutamise vastusuuna analüüsi ümberkirjutamises `BwdRewrite` (rida 2). Muutujale omistamise sõlm asendatakse tühja graafiga (read 6, 10, 12), kui muutuja ei ole elus muutujate hulgas. Ülejäänud olukordades ümberkirjutamist ei toimu (read 9, 13).

3.6 Läbilangevate märgenditete eemaldamine

Kui plokk algab märgendiga, lõppeb hüppega mõnele muule plokile ja ei sisalda ühtki sõlme, siis on plokk sisuliselt tühi ning vastava ploki märgend „langeb läbi“. Sellise ploki võib graafist eemaldada ja hüpped sellele plokile asendada hüppega järgnevale plokile, st märgendile, kuhu oleks tühja ploki lõpus hüpatud.

Liikudes graafis vastusuunas, jättes meelde märgend, millele ploki lõpus hüpatakse, kuid mis tühistatakse vahepealsete sõlme korral, saab selle ploki algusmärgendile jõudes vastandada meelde jäetud märgendiga. Koodis 22 on toodud fakti tüüp `FallThroughFact` (read 1-2), mille konstruktori `FTF` väli `eq` vastandab läbilangevad märgendid ning väli `cur` hoiab meeles analüüsitava ploki lõpus toimuva hüppe sihtmärgendit, kui plokk lõppeb sõlmega `SGoto` ja kui vahepealsed sõlmed seda tühistanud pole. Tühjale faktile fastab f-n `unknownFact` (read 4-5). Vastusuuna analüüsis märgendile fakti otsimist on täiendatud f-nis `getFact` vaikimisi tühja faktiga (read 7-8). F-niga `exitingFact` salvestatakse märgend fakti (read 10-11) ning f-niga `setMiddles` tühistatakse see (read 13-14). Funktsioon `enteringFact` lisab fakti vaste vaadeldava märgendi ja antud ploki lõpusõlme vahel (read 16-21).

Kood 22: Läbilangevate märgendite eemaldamine — andmevoo võre

```
1  data FallThroughFact = FTF { eq :: Map Label Label
2                             , cur :: Maybe Label }
3
4  unknownFact :: FallThroughFact
5  unknownFact = FTF { eq = Map.empty, cur = Nothing }
6
7  getFact :: FactBase FallThroughFact → Label → FallThroughFact
8  getFact f l = fromMaybe unknownFact $ lookupFact l f
9
10 exitingFact :: Label → FallThroughFact → FallThroughFact
11 exitingFact l f = f { cur = Just l }
12
13 setMiddles :: FallThroughFact → FallThroughFact
14 setMiddles f = f { cur = Nothing }
15
16 enteringFact :: Label → FallThroughFact → FallThroughFact
17 enteringFact l f = f { eq = x, cur = Nothing }
18   where x = case cur f of
19               Just l' → Map.insert l l' eq'
20               _ → eq'
21         eq' = (eq f)
```

Vastusuuna ülekandef-n on toodud koodis 23. Sisenemisel suletud sõlm `SLabel` töötleb fakti f-niga `enteringFact` (rida 5). Mõlemast otsast avatud sõlmed `STempAssign`, `SLValAssign` märgivad f-niga `setMiddles` fakti, et tegu pole sisult tühja plokiga (read 6-7). Väljumisel suletud sõlmedest salvestatakse märgend f-niga `exitingFact` ainult sõlmes `SGoto` (rida 10), kuna sõlmedes `SProcedure` ja `SFun` tehakse sisulist tööd, mis kohe tühistaks salvestuse f-niga `setMiddles` (read 8, 14); sõlmes `SIfThenElse` puudub (üldjuhul) ühene märgend, millele ploki lõpus hüpatakse, seega märgendit ei salvestata ning tõese ja väära harudele vastavad märgendite vasted ühendatakse (read 12-13).

Koodis 24 on toodud ümberkirjutamine `fallThroughRewrite`. Väljumisel suletud sõlm on võimalik ringi kirjutada, kui leidub vaste sõlmes kasutatud märgendi(te)le (read 8, 14, 15, 18, 21).

3.7 Näiteprogrammi optimeerimine

Koodis 25 on esitatud *Simple-Pascali* näiteprogramm. Demonstreeritakse uue tüübi defineerimist, programmimuutujate deklareerimist, avaldiste arvutamist, kirje väljadele ja muutujatele omistamist ning nende väärtuste kasutamist, funktsiooni ja protseduuri väljakutseid, kui-siis-muidu hargnemist, loenduriga tsüklit ja selle katkestamist.

Koodis 26 on toodud selle programmi vahe-esitus. Märgendid on tähistatud `L`-iga, abimuutujad `t`-ga. Plokk algab vasakule joondatud märgendiga, ploki sisu on taandega

Kood 23: Läbilangevate märgendite eemaldamine — ülekandefunktsioon

```
fallThroughTransfer :: BwdTransfer Stmnt FallThroughFact
fallThroughTransfer = mkBTransfer bw
  where
    bw :: Stmnt e x → Fact x FallThroughFact → FallThroughFact
    bw (SLabel l) f = enteringFact l f
    bw (STempAssign _ _) f = setMiddles f
    bw (SLValAssign _ _) f = setMiddles f
    bw (SProcedure _ _ l) f = setMiddles (getFact f l)
    bw (SExit) _ = fact_bot fallThroughLattice
    bw (SGoto l) f = exitingFact l (getFact f l)
    bw (SIfThenElse _ tl fl) f =
      FTF { eq = (eq $ getFact f tl) `Map.union`
                 (eq $ getFact f fl), cur = Nothing }
    bw (SFun _ _ _ l) f = setMiddles (getFact f l)
```

Kood 24: Läbilangevate märgendite eemaldamine — ümberkirjutamine

```
fallThroughRewrite :: forall m. FuelMonad m =>
                      BwdRewrite m Stmnt FallThroughFact
fallThroughRewrite = mkBRewrite br
  where
    br :: Stmnt e x →
            Fact x FallThroughFact → m (Maybe (Graph Stmnt e x))
    br (SGoto l) facts = return $
      do l' ← Map.lookup l (eq (getFact facts l))
         returnG $ SGoto l'
    br (SIfThenElse t lt lf) facts = return $
      if ch then returnG $ SIfThenElse t lt' lf' else Nothing
      where
        look l = fromMaybe l $ Map.lookup l $ eq (getFact facts l)
        lt' = look lt
        lf' = look lf
        ch = (lt /= lt') || (lf /= lf')
    br (SProcedure p as l) facts = return $
      do l' ← Map.lookup l (eq (getFact facts l))
         returnG $ SProcedure p as l'
    br (SFun t f as l) facts = return $
      do l' ← Map.lookup l (eq (getFact facts l))
         returnG $ SFun t f as l'
    br _ _ = return Nothing
```

Kood 25: *Simple-Pascali* näiteprogramm

```
program example;
type
  TComplex = record
    a: integer;
    b: integer;
  end;
var
  C1, C2: TComplex;
  L1, L2, L3: integer;
  I: integer;
  X, Y: integer;
  N, P, R: integer;
function isqrt(X: integer): integer;
begin writeln("undefined"); end;

begin
  C1.a := 4;
  C1.b := 3;
  X := C1.a;
  Y := C1.b;
  if X > Y then
  begin
    C2.a := 5;
    C2.b := readln();
  end else begin
    C2.a := readln();
    C2.b := 12;
  end;
  L1 := isqrt(C1.a * C1.a + C1.b * C1.b);
  L2 := isqrt(C2.a * C2.a + C2.b * C2.b);
  N := (C2.a - C1.a) * (C2.a - C1.a) +
       (C2.b - C1.b) * (C2.b - C1.b);
  R := N;
  for I := 1 to 10 do
  begin
    P := R;
    R := (R + N / R) / 2;
    if R < P then
      writeln(R)
    else
      break;
  end;
  L3 := P;
  writeln(L3);
end.
```

viidud paremale. Ploki sisus vastab ühele sõlmele üks rida. Sõlmede täitmise järjekord on ülevalt alla; ploki viimane sõlm kannab juhtvoo tähistatud märgendile või lõpetab programmi töö. Plokkide omavaheline järjekord ei oma tähtsust. Programmi täitmist alustatakse alati märgendist `L1`. Funktsiooni `isqrt` vahe-esitust pole toodud, kuna optimeeritakse ainult programmi keha. Alljärgnevas optimeeritakse[6] seda vahe-esitust, kusjuures iga soorituse väljund on järgneva soorituse sisendiks.

Koodis 27 on näiteprogrammi vahe-esitus pärast konstantide voltimist ja levitamist sooritusega `optConst`. Plokis `L1` on konstant 4 levinud läbi `C1.a` abimuutujasse `t2`, sealt läbi programmimuutuja `X` abimuutujasse `t4`. Samamoodi levis konstant 3 läbi `C1.b` abimuutujasse `t3` ja sealt läbi programmimuutuja `Y` abimuutujasse `t5`. Abimuutujale `t6` on omistatud abimuutujate `t4` ja `t5` võrdluse tulemus ja kuna võrreldavad on konstantsed, siis võrdlus volditakse antud juhul konstandiks `True`. Seega järgnevas kui-siis-muidu hargnemises täidetakse alati tõene haru ja plokk märgendiga `L4` osutub surnud koodiks, mis visatakse minema (põhjendust vt. 3.1). Sarnaselt käitutakse ülejäänud plokkides. Näiteks plokis `L2`, volditi funktsiooni argumendile vastav avaldis lõplikult konstandiks. Tähelepanu tuleb juhtida plokile `L7`, kus f-ni argumendile vastav avaldis volditi pooleldi: kui poleks ära visatud plokki `L4`, ei oleks saanud avaldist voltida, kuna operandid ei ole mõlemas kui-siis-muidu harus konstantsed; kui oleks ära visatud plokk `L3` ploki `L4` asemel, oleks volditud avaldise teine pool `C2.b * C2.b`.

Koodis 28 eemaldati ühised alamavaldised. Plokis `L1` on muutujale `X` omistatud abimuutuja `t0`, kuna seal kasutatakse seda avaldist, literaal 4, esimest korda. Samamoodi omistatakse muutujale `Y` abimuutuja `t1`, kuna seal on esimest korda literaal 3. Plokis `L7` on abimuutujale `t24` omistamisel avaldise mõlemaks operandiks `C2.b`, st abimuutuja `t23` kasutamise asemel saab kasutada abimuutujat `t22`; plokis `L8` kasutatakse samuti abimuutujate `t34` ja `t37` asemel abimuutujat `t22`. Lisaks on plokis `L8` ühised alamavaldised abimuutujates `t36` ja `t39`, seega abimuutujale `t40` omistatud avaldise väärtuse arvutamiseks kasutatakse neist ainult esimest.

Koodis 29 on levitatud koopiaid. Plokkides `L3` ja `L5` kopeeritakse sisendist loetud väärtus, mis on salvestatud abimuutujas `t8`, kirje väljale `C2.b`. Ühiste alamavaldiste eemaldamise tulemusena vastas selle kirje välja väärtusele abimuutuja `t22`, mis koopia levitamisel plokkides `L7` ja `L8` asendakse originaaliga `t8`.

Koodis 30 on eemaldatud surnud omistamised. Eemaldati programmimuutujatele `L1` ja `L2` omistamised. Lihtne on veenduda näiteprogrammi põhjal koodis 25, et neid muutujaid ei ole kasutatud, kuid säilitati neile vastavad abimuutujad `t18` ja `t26`, kuna neile omistatakse väärtus funktsiooni väljakutsetega, mis võivad omada kõrvalmõjusid, mis peavad säilima. Paljud omistamised „surid“ eelnevate soorituste ümberkirjutamiste tulemusena — sellest tähelepanekust on tingitud soorituste järjekord.

Koodis 31 on toodud näiteprogrammi lõplikult optimeeritud vahe-esitus. Läbilangevate märgendite eemaldamise sooritustes muudeti ploki `L3` lõpus oleva funktsiooni väljakutsest naasemise märgendiks `L2`, kuna märgend `L5` langes läbi. Läbi langesid ka märgendid `L18`, `L20` ja `L21` — nende asemel on kasutatud vastavalt märgendeid `L11`, `L12` ja `L11`. Peale läbilängevate märgendite eemaldamist on vahe-esitust töödeldud sooritusega `optNothing`, mis viskas minema surnud koodi plokid `L5`, `L18`, `L20` ja `L21`.

[6] Optimeerija kood on saadaval veebis `http://www.tud.ttu.ee/~t073849/bsc/`

Kood 26: Näiteprogrammi vahe-esitus

```
L1:
    t0 ← 4
    ”C1”.”a” := t0
    t1 ← 3
    ”C1”.”b” := t1
    t2 ← ”C1”.”a”
    ”X” := t2
    t3 ← ”C1”.”b”
    ”Y” := t3
    t4 ← ”X”
    t5 ← ”Y”
    t6 ← t4 > t5
    if t6 then L3 else L4
L2:
    t11 ← ”C1”.”a”
    t12 ← ”C1”.”a”
    t13 ← t11 * t12
    t14 ← ”C1”.”b”
    t15 ← ”C1”.”b”
    t16 ← t14 * t15
    t17 ← t13 + t16
    t18 ← isqrt(t17) goto L7
L3:
    t7 ← 5
    ”C2”.”a” := t7
    t8 ← readln() goto L5
L4:
    t9 ← readln() goto L6
L5:
    ”C2”.”b” := t8
    goto L2
L6:
    ”C2”.”a” := t9
    t10 ← 12
    ”C2”.”b” := t10
    goto L2
L7:
    ”L1” := t18
    t19 ← ”C2”.”a”
    t20 ← ”C2”.”a”
    t21 ← t19 * t20
    t22 ← ”C2”.”b”
    t23 ← ”C2”.”b”
    t24 ← t22 * t23
    t25 ← t21 + t24
    t26 ← isqrt(t25) goto L8
L8:
    ”L2” := t26
    t27 ← ”C2”.”a”
    t28 ← ”C1”.”a”
    t29 ← t27 − t28
    t30 ← ”C2”.”a”
    t31 ← ”C1”.”a”
    t32 ← t30 − t31
    t33 ← t29 * t32
    t34 ← ”C2”.”b”
    t35 ← ”C1”.”b”
    t36 ← t34 − t35
    t37 ← ”C2”.”b”
    t38 ← ”C1”.”b”
    t39 ← t37 − t38
    t40 ← t36 * t39
    t41 ← t33 + t40
    ”N” := t41
    t42 ← ”N”
    ”R” := t42
    t43 ← 1
    ”I” := t43
    t44 ← 10
    ”I~High” := t44
    goto L9
L9:
    t45 ← ”I”
    t46 ← ”I~High”
    t47 ← t45 > t46
    if t47 then L12 else L10
L10:
    t51 ← ”R”
    ”P” := t51
    t52 ← ”R”
    t53 ← ”N”
    t54 ← ”R”
    t55 ← t53 / t54
    t56 ← t52 + t55
    t57 ← 2
    t58 ← t56 / t57
    ”R” := t58
    t59 ← ”R”
    t60 ← ”P”
    t61 ← t59 < t60
    if t61 then L19 else L20
L11:
    t48 ← ”I”
```

```
    t49 ← 1
    t50 ← t48 + t49
    ”I” := t50
    goto L9
L12:
    t63 ← ”P”
    ”L3” := t63
    t64 ← ”L3”
    writeln (t64) goto L23
L18:
    goto L11
L19:
    t62 ← ”R”
    writeln (t62) goto L21
L20:
    goto L12
L21:
    goto L18
L22:
    goto L18
L23:
    exit
```

Kood 27: Näiteprogrammi konstantide voltimine ja levitamine

```
L1:
    t0 ← 4
    ”C1”.”a” := t0
    t1 ← 3
    ”C1”.”b” := t1
    t2 ← 4
    ”X” := t2
    t3 ← 3
    ”Y” := t3
    t4 ← 4
    t5 ← 3
    t6 ← True
    goto L3
L2:
    t11 ← 4
    t12 ← 4
    t13 ← 16
    t14 ← 3
    t15 ← 3
    t16 ← 9
    t17 ← 25
    t18 ← isqrt(t17) goto L7
L3:
    t7 ← 5
    ”C2”.”a” := t7
    t8 ← readln() goto L5
L5:
    ”C2”.”b” := t8
    goto L2
L7:
    ”L1” := t18
    t19 ← 5
    t20 ← 5
    t21 ← 25
    t22 ← ”C2”.”b”
    t23 ← ”C2”.”b”
    t24 ← t22 * t23
    t25 ← t21 + t24
    t26 ← isqrt(t25) goto L8
L8:
    ”L2” := t26
    t27 ← 5
    t28 ← 4
    t29 ← 1
    t30 ← 5
    t31 ← 4
    t32 ← 1
    t33 ← 1
    t34 ← ”C2”.”b”
    t35 ← 3
    t36 ← t34 − t35
    t37 ← ”C2”.”b”
    t38 ← 3
    t39 ← t37 − t38
    t40 ← t36 * t39
    t41 ← t33 + t40
    ”N” := t41
    t42 ← ”N”
    ”R” := t42
    t43 ← 1
    ”I” := t43
    t44 ← 10
    ”I~High” := t44
    goto L9
L9:
    t45 ← ”I”
    t46 ← 10
    t47 ← t45 > t46
    if t47 then L12 else L10
```

```
L10:
    t51 ← ”R”
    ”P” := t51
    t52 ← ”R”
    t53 ← ”N”
    t54 ← ”R”
    t55 ← t53 / t54
    t56 ← t52 + t55
    t57 ← 2
    t58 ← t56 / t57
    ”R” := t58
    t59 ← ”R”
    t60 ← ”P”
    t61 ← t59 < t60
    if t61 then L19 else L20
L11:
    t48 ← ”I”
    t49 ← 1
    t50 ← t48 + t49
    ”I” := t50
    goto L9
L12:
    t63 ← ”P”
    ”L3” := t63
    t64 ← ”L3”
    writeln (t64) goto L23
L18:
    goto L11
L19:
    t62 ← ”R”
    writeln (t62) goto L21
L20:
    goto L12
L21:
    goto L18
L23:
    exit
```

Kood 28: Näiteprogrammi ühiste alamavaldiste eemaldamine

```
L1:
    t0 ← 4
    ”C1”.”a” := t0
    t1 ← 3
    ”C1”.”b” := t1
    t2 ← 4
    ”X” := t0
    t3 ← 3
    ”Y” := t1
    t4 ← 4
    t5 ← 3
    t6 ← True
    goto L3
L2:
    t11 ← 4
    t12 ← 4
    t13 ← 16
    t14 ← 3
    t15 ← 3
    t16 ← 9
    t17 ← 25
    t18 ← isqrt(t17) goto L7
L3:
    t7 ← 5
    ”C2”.”a” := t7
    t8 ← readln() goto L5
L5:
    ”C2”.”b” := t8
    goto L2
L7:
    ”L1” := t18
    t19 ← 5
    t20 ← 5
    t21 ← 25
    t22 ← ”C2”.”b”
    t23 ← ”C2”.”b”
    t24 ← t22 * t22
    t25 ← t17 + t24
    t26 ← isqrt(t25) goto L8
L8:
    ”L2” := t26
    t27 ← 5
    t28 ← 4
    t29 ← 1
    t30 ← 5
    t31 ← 4
    t32 ← 1
    t33 ← 1
    t34 ← ”C2”.”b”
    t35 ← 3
    t36 ← t22 − t1
    t37 ← ”C2”.”b”
    t38 ← 3
    t39 ← t22 − t1
```

```
    t40 ← t36 * t36
    t41 ← t29 + t40
    ”N” := t41
    t42 ← ”N”
    ”R” := t42
    t43 ← 1
    ”I” := t29
    t44 ← 10
    ”I~High” := t44
    goto L9
L9:
    t45 ← ”I”
    t46 ← 10
    t47 ← t45 > t44
    if t47 then L12 else L10
L10:
    t51 ← ”R”
    ”P” := t51
    t52 ← ”R”
    t53 ← ”N”
    t54 ← ”R”
    t55 ← t42 / t51
    t56 ← t51 + t55
    t57 ← 2
    t58 ← t56 / t57
    ”R” := t58
    t59 ← ”R”
    t60 ← ”P”
    t61 ← t59 < t60
    if t61 then L19 else L20
L11:
    t48 ← ”I”
    t49 ← 1
    t50 ← t45 + t29
    ”I” := t50
    goto L9
L12:
    t63 ← ”P”
    ”L3” := t60
    t64 ← ”L3”
    writeln (t64) goto L23
L18:
    goto L11
L19:
    t62 ← ”R”
    writeln (t59) goto L21
L20:
    goto L12
L21:
    goto L18
L23:
    exit
```

Kood 29: Näiteprogrammi koopia levitamine

```
L1:
    t0 ← 4
    ”C1”.”a” := t0
    t1 ← 3
    ”C1”.”b” := t1
    t2 ← 4
    ”X” := t0
    t3 ← 3
    ”Y” := t1
    t4 ← 4
    t5 ← 3
    t6 ← True
    goto L3
L2:
    t11 ← 4
    t12 ← 4
    t13 ← 16
    t14 ← 3
    t15 ← 3
    t16 ← 9
    t17 ← 25
    t18 ← isqrt(t17) goto L7
L3:
    t7 ← 5
    ”C2”.”a” := t7
    t8 ← readln() goto L5
L5:
    ”C2”.”b” := t8
    goto L2
L7:
    ”L1” := t18
    t19 ← 5
    t20 ← 5
    t21 ← 25
    t22 ← ”C2”.”b”
    t23 ← ”C2”.”b”
    t24 ← t8 * t8
    t25 ← t17 + t24
    t26 ← isqrt(t25) goto L8
L8:
```

```
    ”L2” := t26
    t27 ← 5
    t28 ← 4
    t29 ← 1
    t30 ← 5
    t31 ← 4
    t32 ← 1
    t33 ← 1
    t34 ← ”C2”.”b”
    t35 ← 3
    t36 ← t8 − t1
    t37 ← ”C2”.”b”
    t38 ← 3
    t39 ← t8 − t1
    t40 ← t36 * t36
    t41 ← t29 + t40
    ”N” := t41
    t42 ← ”N”
    ”R” := t41
    t43 ← 1
    ”I” := t29
    t44 ← 10
    ”I~High” := t44
    goto L9
L9:
    t45 ← ”I”
    t46 ← 10
    t47 ← t45 > t44
    if t47 then L12 else L10
L10:
    t51 ← ”R”
    ”P” := t51
    t52 ← ”R”
    t53 ← ”N”
    t54 ← ”R”
    t55 ← t41 / t51
    t56 ← t51 + t55
    t57 ← 2
    t58 ← t56 / t57
    ”R” := t58
    t59 ← ”R”
    t60 ← ”R”
    t61 ← t58 < t58
    if t61 then L19 else L20
L11:
    t48 ← ”I”
    t49 ← 1
    t50 ← t45 + t29
    ”I” := t50
    goto L9
L12:
    t63 ← ”P”
    ”L3” := t60
    t64 ← ”L3”
    writeln (t60) goto L23
L18:
    goto L11
L19:
    t62 ← ”R”
    writeln (t58) goto L21
L20:
    goto L12
L21:
    goto L18
L23:
    exit
```

Kood 30: Näiteprogrammi surnud omistamiste eemaldamine

```
L1:
    t1 ← 3
    goto L3
L2:
    t17 ← 25
    t18 ← isqrt(t17) goto L7
L3:
    t8 ← readln() goto L5
L5:
    goto L2
L7:
    t24 ← t8 * t8
    t25 ← t17 + t24
    t26 ← isqrt(t25) goto L8
L8:
    t29 ← 1
    t36 ← t8 − t1
    t40 ← t36 * t36
    t41 ← t29 + t40
    ”R” := t41
    ”I” := t29
    t44 ← 10
    goto L9
L9:
    t45 ← ”I”
```

```
    t47 ← t45 > t44
    if t47 then L12 else L10
L10:
    t51 ← ”R”
    t55 ← t41 / t51
    t56 ← t51 + t55
    t57 ← 2
    t58 ← t56 / t57
    ”R” := t58
    t60 ← ”R”
    t61 ← t58 < t58
    if t61 then L19 else L20
L11:
    t50 ← t45 + t29
    ”I” := t50
    goto L9
L12:
    writeln (t60) goto L23
L18:
    goto L11
L19:
    writeln (t58) goto L21
L20:
    goto L12
L21:
    goto L18
L23:
    exit
```

Kood 31: Optimeeritud näiteprogramm

```
L1:
    t1 ← 3
    goto L3
L2:
    t17 ← 25
    t18 ← isqrt(t17) goto L7
L3:
    t8 ← readln() goto L2
L7:
    t24 ← t8 * t8
    t25 ← t17 + t24
    t26 ← isqrt(t25) goto L8
L8:
    t29 ← 1
    t36 ← t8 − t1
    t40 ← t36 * t36
    t41 ← t29 + t40
    ”R” := t41
    ”I” := t29
    t44 ← 10
    goto L9
L9:
    t45 ← ”I”
    t47 ← t45 > t44
    if t47 then L12 else L10
L10:
    t51 ← ”R”
    t55 ← t41 / t51
    t56 ← t51 + t55
    t57 ← 2
    t58 ← t56 / t57
    ”R” := t58
    t60 ← ”R”
    t61 ← t58 < t58
    if t61 then L19 else L12
L11:
    t50 ← t45 + t29
    ”I” := t50
    goto L9
L12:
    writeln (t60) goto L23
L19:
    writeln (t58) goto L11
L23:
    exit
```

4 Kokkuvõte

Käesoleva töö eesmärgiks on lisada lihtsustatud Pascali kompilaatorile andmevooanalüüsil põhinev optimeerimine, kasutades selleks teeki Hoopl.

Töö esimene osa tutvustab kompilaatorit *Simple-Pascal* ja selle sisemist abstraktset süntaksipuud. Esitatakse nõuded andmevooanalüüsi teegi Hoopl rakendamiseks. Osa tulemuseks on Hoopl-teegiga töötlemiseks sobiv vahe-esitus ja vahend *Simple-Pascali* ASTi transleerimiseks sellesse vahe-esitusse.

Teises osas tutvustakse detailsemalt Hoopl-teegi kasutamist. Kirjeldatakse lihtsamate andmevooanalüüsil põhinevate optimeerimiste andmevoo faktid, faktide kogumine ülekande funktsiooniga ja vahe-esituse ümberkirjutamine lähtudes kogutud faktidest. Töö tulemusena valmis optimeerija, mis sooritab andmevooanalüüsil põhinevad konstantide voltimise ja levitamise, koopiate levitamise, ühiste alamavaldiste eemaldamise, surnud omistamiste eemaldamise lihtsustamised. Optimeerija tööd demonstreeritakse näiteprogrammi varal.

Edasiseid arengusuundi on mitmeid. Antud töös ei genereeritud optimeeritud vahe-esitusest virtuaalmasinal käivitamiseks sobivat masinkoodi, st kompilaatori *Simple-Pascal* ja antud töö optimeerija väljundid pole vahetult võrreldavad. Täiendada võib ka *Simple-Pascali* võimalusi ning lähendada seda Pascal-keelele. Optimeerimised saab laiendada protseduurile / funktsioonidele ning lisada interprotseduurseid optimeerimisi. Võib uurida toodud optimeerimiste jõudlust ning võrrelda neid imperatiivsete realisatsioonidega.

Viited

[1] Hoopl : Inside 206-105. [WWW] http://blog.ezyang.com/category/ghc/hoopl/ (05.07.2013)

[2] Muchnick, S. S. Advanced Compiler Design And Implementation. San Francisco: Morgan Kaufmann, 1997

[3] New monads/MonadUnique - HaskellWiki. [WWW] http://www.haskell.org/haskellwiki/New_monads/MonadUnique (05.05.2013)

[4] Ramsey N., Dias J., Jones, S. P. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation — *ACM Haskell Symposium, Baltimore MD, United States, 2010/09/30*

[5] Schwartzbach, M. I. Lecture Notes on Static Analysis. University of Aarhus, Denmark, 2006

A *Simple-pascali* grammatika EBNF[7]

Grammatika algussümboliks on mitteterminaalne (ingl. nonterminal) sümbol `program`.

```
identifier = { ws }, ident start, { ident letter }, { ws } ;
ident start = alpha—numeric | '_' ;
ident letter = alpha—numeric | '_' | ? apostrophe ? ;

ws = ? white space characters ? ;
semi = ';', { ws } ;
dot = '.', { ws } ;
comma = ',', { ws } ;
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
digit sequence = digit, { digit } ;
sign = '—';
integer = [sign], digit sequence ;

program = 'program', ws, identifier, semi, [constants], [types], [
   variables], [procedures and functions], block, dot;

constants = 'const', ws, constant declaration, { constant declaration }
   ;
constant declaration = identifier, '=', expression, semi ;

types = 'type', ws, type declaration, { type declaration } ;
type declaration = identifier, '=', type, semi ;
type = array type | record type | simple type ;

array type = 'array', { ws }, '[', integer, ']', { ws }, of, ws, { ws },
    type ;

record type = 'record', fields, 'end' ;
fields = field, { semi, field } , [ semi ] ;
field = identifier, ':', { ws }, type, { ws };

simple type = identifier;

variables = 'var', ws, variable list, { variable list } ;
variable list = variable names, ':', type , semi ;
variable names = identifier, { comma, identifier } ;

procedures and functions = { procedure | function } ;
procedure = 'procedure', ws, identifier, parameter list, semi, [
   variables ], block, semi ;

function = 'function', ws, identifier , '(', parameter list, ')', ':',
```

[7] Extended Backus–Naur Form, `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`

```
   identifier, semi, [ variables ], block, semi ;

parameter list = { identifier, colon, type };

block = 'begin', statements, 'end'

statements = statement, { semi, statement }, [ semi ] ;
statement = if then else | assignment | procedure call | 'break' | '
   continue' | 'exit' | return | for ;

if then else = 'if', expression, 'then', ws, ( block | statement ), [ '
   else', ( block | statement ) ] ;

assignment = left hand side, ':=', expression ;
left hand side = array item | record field | variable

array item = identifier, '[', expression, ']' ;
record field = identifier, dot, identifier ;
variable = identifier ;

procedure call = identifier, '(', [ expressions ], ')' ;

return = 'return', expression ;

for = 'for', identifier, ':=', expression, 'to', expression, 'do', (
   block | statement ) ;

expressions = expression, { comma, expression } ;
expression = simple expression, { relational operator, simple expression
    } ;
simple expression = term, { adding operator, term } ;

term = factor { multiplying operator, factor } ;
factor = '(', expression, ')'| literal | function call | array item |
   record field | variable ;
function call = identifier, '(', [ expressions ], ')' ;

adding operator = '+' | '—' ;
multiplying operator = '*' | '/' | '%' ;
relational operator = '=' | '!=' | '>' | '<' ;
```