УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Бојана Мандић

ОПТИМИЗАЦИЈА ВРЕМЕНСКЕ ЕФИКАСНОСТИ ПРОГРАМА КОЈИ КОРИСТЕ БИБЛИОТЕКУ NUMPY

мастер рад

Београд, 2019.

Ментор:

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Милена Вујошевић Јаничић, доцент
Универзитет у Београду, Математички факултет

др Милан Банковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Садржај

Глава 1

Увод и организација рада

Програмски језик Python и библиотека NumPy се последњих година интензивно користе за писање програма који се примењују у разним областима као што су истраживање података, машинско учење, нумеричка и научна израчунавања и слично. За природу проблема којима се поменуте области баве, временска ефикасност решења углавном има велики значај, али се у оквиру ових области често срећу проблеми чије решавање захтева комплексна и временски захтевна израчунавања.

Оптимизација временске сложености извршавања програма може бити ручна или аутоматизована. Ручну оптимизацију је углавном најкорисније реализовати кроз погодан избор алгоритама, структура података и добар дизајн, док покушаји ручне оптимизације нижег нивоа често могу да имају и супротан ефекат од жељеног. Такође, оптимизације нижег нивоа најчешће имају негативан утицај на читљивост кôда. Аутоматизована оптимизација спроводи се уз помоћ специјалних програма који се називају *оптимизатори*. Оптимизатори су најчешће уграђени у програмске преводиоце и користе се у завршним фазама превођења. Оптимизатори омогућавају програмеру да лакше успостави баланс између читљивости и ефикасности кôда.

У овом раду биће представљен приступ аутоматизованој оптимизацији заснован на научном раду Equality Saturation: A New Approach to Optimization [3]. Овај рад проблем оптимизације кôда дефинише као проблем оптимизације се линеарним ограничењима који се решавају уз помоћ псеудо-буловских решавача. Аутори овог рада приказали су уопштени приступ оптимизацији, који су реализовали за програмски језик Java. Овај рад бавиће се реализацијом поменутог приступа за програмски језик Python и библиотеку Numpy.

Прво поглавље обухвата преглед најчешће коришћених приступа аутоматизованој оптимизацији кôда. Друго поглавље посвећено је програмском језику Python и библиотеци Numpy. У оквиру овог поглавља, представљен је и модул `AST`, који је коришћен у имплементацији. У наредном поглављу представљен је алат CLIPS, такође коришћен у имплементацији. У четвртом поглављу описан је приступ представљен у поменутом научном раду. Последње поглавље обухвата основне детаље имплементације приступа за програмски језик Python и библиотеку Numpy.

Глава 2

Оптимизација Компилатора

2.1 Структура компилатора

Задатак програмских преводиоца је да преведу кôд програма са једног језика на други, не мењајући при томе значење програма. Најчешћи случај је да се кôд изворног програмског језика преводи на неки језик нижег нивоа, као што су асемблер или машински кôд, у циљу генерисања извршног кôда. Процес превођења састоји се од *фазе анализе* и *фазе синтезе*. Фазу анализе чине лексичка, синтаксна и семантичка анализа, а фазу синтезе генерисање циљног кôда и оптимизација. Сваки од поменутих корака генерише сопствену репрезентацију програма који се преводи и прослеђује је даље као улаз за наредни корак.

Лексичка анализа из изворног кôда састављеног од низа карактера издваја лексичке целине - лексеме, које представљају основне језичке елементе, тј. речи језика, и представља их у облику токена - уређених парова које чине назив токена (на пример, кључна реч, идентификатор, оператор, сепаратор) и вредност лексеме. Издвајање, провера валидности лексема и њихово класификовање као токена изводи се уз помоћ правила дефинисаних регуларним изразима. *Синтаксичка анализа* или парсирање представља процес који од низа токена гради синтаксичко стабло - графовску репрезентацију синтаксне структуре изворног кôда. У овом кораку упоређују се редослед токена и граматика језика изворног кôда како би се проверила синтаксна исправност наредби и израза изграђених од задатих токена. Током *семантичке анализе* проверава се да ли је конструисано синтаксичко стабло исправно у односу на правила језика. На пример, проверава се да ли су операције компатибилне са типовима својих операнада,

да ли је променљива која се користи у неком изразу претходно декларисана и друга правила која је тешко, а често и немогуће проверити приликом парсирања.

Пре генерисања циљног кôда, изворни кôд се најчешће преводи у додатну репрезентацију - међукôд, који служи да обезбеди већу независност фазе анализе и фазе синтезе и омогући да фаза анализе буде јединствена и машински независна. Осим тога, оваква репрезентација може да буде погодна за примену разних оптимизационих техника. *Генерисање кôда* подразумева превођење међукôда на циљни кôд, узимајући у обзир радно окружење и скуп инструкција машине на којој се кôд генерише. *Оптимизација* представља модификацију кôда у циљу ефикасније употребе процесора или меморијских ресурса, не мењајући значење изворног кôда. Корак оптимизације се може изводити и над међукôдом и над циљним кôдом.

2.2 Оптимизација

Оптимизација представља трансформисање кôда у кôд који мање користи ресурсе као што су процесорско време, меморија или чак батерија. Основни захтев за овакву трансформацију је да резултујући кôд буде семантички еквивалентан полазном. Оптимизовање временске ефикасности углавном се реализује на рачун просторне ефикасности и обратно, мада је најчешћи циљ оптимизације ипак минимизација времена извршавања програма. У смислу временске и просторне сложености, оптимизација је најзахтевнији корак у процесу превођења. Доказано је да су неки проблеми оптимизације кôда НП-комплетни, а неки чак и неодлучиви.

Оптимизација се може извршавати у различитим фазама превођења. Оптимизација над међукôдом је машински независна, па не узима у обзир карактеристике циљне архитектуре и ефикасност њених инструкција и не приступа апсолутним меморијским локацијама и регистрима, за разлику од оптимизације која се извршава над генерисаним циљним кôдом. Такође, оптимизација може да се извршава и над синтаксним стаблом.

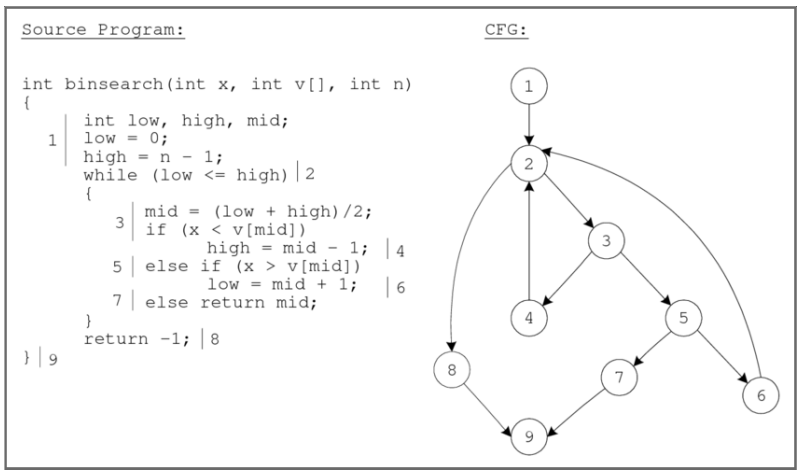
Традиционални приступ оптимизацији компилатора је секвенцијална примена оптимизационих техника, тако да свака од њих трансформише резултат добијен применом претходне. Већина оптимизационих техника коју модерни компилатори користе може да се сврста у неку од следећих група: локална

оптимизација, глобална оптимизација, интерпроцедурална оптимизација, оптимизација генератора кôда и функционална оптимизација.

Локална и глобална оптимизација

Оптимизације овог типа заснивају се на анализи контроле тока функције или процедуре у програму. Контрола тока представља се *графом контроле тока.*

Граф контроле тока је графовска репрезентација свих путева кроз инструкције којима програм може да прође у фази извршавања. Скуп чворова у оваквом графу чине *базични блокови.* Сваки базични блок састоји се из инструкција које се увек извршавају секвенцијално, тј. само прва инструкција у блоку може бити циљ скока (условног или безусловног) из неке инструкције и само последња инструкција у блоку може бити скок на неку инструкцију. Додатни услов који употпуњује дефиницију базичног блока је да сваки базични блок садржи максимални скуп инструкција за које важе наведена својства. Дакле, базични блок представља скуп инструкција које се увек извршавају редом, од прве до последње.



Слика 2.1: Пример графа контроле тока

Локалну оптимизацију чине оптимизационе технике које се примењују на сваки базични блок појединачно, не узимајући у обзир међусобне повезаности блокова. Ово је једноставнија врста оптимизације. Неке од типичних техника локалне оптимизације су алгебарско поједностављивање израза (нпр. `x = y + 0` ⇒ `x = y`), слагање константи, (нпр. `x = 1 + 2` ⇒ `x = 3`), редук-

ција снаге, која представља замену операције неком другом операцијом која даје исти резултат, а брже се извршава (нпр. `i * 4` ⇒ `i << 2`, za celobrojno `i`). Локална оптимизација често се изводи над *троадресним кôдом*. Троадресни кôд је врста међукôда у коме свака инструкција има облик `x := y op z`, где су `y` и `z` променљиве, константе или привремене променљиве које генерише компилатор, а `op` било који оператор.

Глобална оптимизација примењује оптимизационе технике над читавим графом контроле тока. Иако њен назив може да сугерише другачије, глобална оптимизације оптимизује унутрашњу структуру функције или процедуре као целине, не узимајући у обзир структуру читавог програма и међусобне зависности функција које се у њему користе, па се често назива и интрапроцедуралном оптимизацијом. У ову групу спадају бројне оптимизације петљи и гранања, као што су: измештање кôда који се у петљи не мења изван петље, поједностављивање индукцијске променљиве (променљиве која је дефинисана линеарном функцијом бројача петље), одмотавање петљи, сажимање петљи или наредби гранања које имају исти услов...

Постоје и оптимизационе технике које имају и своју локалну и глобалну верзију. Ту спадају елиминација мртвог кôда, пропагација константи, пропагација променљивих и елиминација заједничких подизраза. Глобалне верзије поменутих техника су знатно компликованије за имплементацију, јер је потребно из статичког кôда извести информације о динамичком понашању програма. За решавање овог проблема користе се *глобалне анализе*. Њихов задатак је да провере да ли су у конкретној тачки у програму испуњена унапред дефинисана својства неопходна да би се нека оптимизација извела. На пример, да бисмо знали који делови кôда могу безбедно да се елиминишу, морамо да будемо сигурни да они нису ни у ком случају релевантни за крајњи исход програма. Најчешћи приступ проблему детекције мртвог кôда је анализа живости променљивих. Променљива се дефинише као жива у конкретној тачки у програму ако ће у даљем току програма њена вредност бити прочитана пре него што јој се додели нова вредност. Док је над базичним блоком, због секвенцијалности извршавања инструкција које он садржи, лако проценити да ли је нека променљива жива у задатој тачки, гранања и петље могу условити компликовану структуру графа контроле тока и учинити овај процес знатно сложенијим. Такође, у случају пропагације константи, ако желимо да у неком изразу променљиву `x` заменимо константом `k`, морамо да будемо сигурни да на сваком

путу у графу који води до тог израза последња додела променљивој `x` буде управо `x := k`. Слично важи и за пропагацију променљивих.

Интерпроцедурална оптимизација

Ову групу чине технике које примењују оптимизације на нивоу читавог програма. Ту спадају неке од проширених варијанти типичних оптимизација као што су интерпроцедурална елиминација мртвог кôда, интерпроцедурална пропагација променљивих и константи, али и веома значајна техника *увлачења дефиниција функција (inlining)*, која се изводи како би се смањила учесталост релативно скупих позива функција. Међутим, претерана примена ове технике може да има и негативан утицај на ефикасност, јер повећава дужину кôда и тако узрокује његово споро учитавање из меморије. Праву меру примене увлачења дефиниција функција није тривијално одредити. Најчешће се избегава увлачење дефиниција функција које садрже петље.

Оптимизација генератора кôда

Задатак генератора кôда је да међукôд преслика у циљни кôд. Како би генерисао што ефикаснији циљни кôд, генератор кôда пролази кроз процесе избора кôда, алокације регистара и планирање редоследа извршавања инструкција.

У фази избора кôда се секвенце операција међукôда преводе у секвенце инструкција циљног кôда. Међукôд углавном користи примитивне аритметичке операције, логичке операције и операције поређења, а циљна архитектура може да комбинује више примитивних операција у једну инструкцију, па се инструкције међукôда често могу на више начина пресликати у инструкције циљног кôда. Зато на архитектурама са комплексним скуповима инструкција добар избор кôда може да има велики утицај на ефикасност генерисаног кôда.

Алокација регистара бави се доделом регистара процесора променљивим које се користе у програму. Читање садржаја регистара и упис у регистре су операције које се знатно брже изршавају од читања и уписа у меморију, али број променљивих које програм користи је често знатно већи од броја расположивих регистара, па се у фази алокације регистара бира које ће променљиве у фази извршавања бити смештене у регистре, а које у меморију. Две променљиве могу бити смештене у исти регистар само ако се не преклапају периоди у којима се оне користе у програму.

Узимајући у обзир међусобне зависности инструкција, некада је могуће променити редослед извршавања инструкција тако да семантика програма остане иста. Планирање редоследа извршавања инструкција изводи се како би се што боље искористила особина *проточне обраде* коју поседују модерни процесори. Проточна обрада је могућност процесора да извршава више инструкција током једног откуцаја системског сата. Ако је некој инструкцији неопходан резултат неке друге инструкције како би се извршила, проточна обрада омогућава да се у периоду чекања несметано изврши нека инструкција која је независна од претходних, па се на тај начин смањује и укупно време извршавања.

Глава 3

Програмски језик Python

Програмски језик Python креирао је деведесетих година прошлог века Гвидо ван Росум. То је језик опште намене који подржава више стилова програмирања, као што су императивни, објектно-оријентисани и функционални стил. Спада у језике вишег нивоа и омогућава писање веома читљивог кôда, што је један од разлога његове велике популарности. Динамички је типизиран и поседује сакупљаче отпадака за аутомаско управљање меморијом. Python програми се углавном интерпретирају, али постоје и компилатори који Python кôд преводе у машински кôд, у циљу бржег извршавања програма.

3.1 Библиотека Numpy

Библиотека Numpy написана је у C-у и Python-у и користи се за разне врсте научног израчунавања. Између осталог, садржи модуле специјализоване за линеарну алгебру, Фуријеове трансформације и генерисање псеудо-случајних бројева и омогућава велики број ефикасних функционалности за рад са векторима, матрицама и низовима виших димензија. Numpy такође садржи и алате за једноставно интегрисање C/C++ и Fortran кôда.

Numpy вишедимензиони низ

Најзначајнија класа Numpy библиотеке је свакако `ndarray` класа. Ова класа представља хомогени вишедимензиони низ, што значи да сви елементи овог низа морају бити истог типа. Неки од основних атрибута класе вишедимензионог низа су `dtype`, који представља тип елемената које низ садржи и `shape`, који

представља уређену n-торку дужина сваке од n димензија низа. Операције над објектима `ndarray` класе извршавају се веома ефикасно. Основни разлог за то је чињеница да се објекти ове класе у меморију смештају у континуалним блоковима заједно са *шемом индексирања*, која је дефинисана управо атрибутима `shape` и `dtype`. Шема индексирања представља основну информацију потребну за приступ елементима низа преко индекса. Још неки корисни атрибути ове класе су и `ndim` - број димензија низа и `size` - укупна дужина низа, а библиотека Numpy садржи и бројне векторске, матричне и тензорске операције које се ефикасно извршавају над објектима вишедимензионог низа.

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
```

Слика 3.1: Пример коришћења класе ndarray

Један значајан термин везан за Numpy вишедимензиони низ је *broadcasting*. Овај термин се односи на дефинисање аритметичких операција над низовима са различитим вредностима атрибута `shape`. Пре него што се изврши нека аритметичка операција над два вишедимензиона низа, проверавају се дужине димензија првог и другог низа по паровима. Дужине димензија се сматрају компатибилним ако су једнаке, или ако је бар једна од њих 1. Број димензија два низа не мора да буде једнак. У случају низова неједнаког броја димензије поређење парова се извршава десним поравнавањем - од упаривања последњих димензија оба низа до прве димензије низа са мањим бројем димензија. Пример димензија операнада и резултата неке аритметичке операције добијеног broadcasting правилима представљен је на слици 3.2. Када је нека од димензија операнада једнака 1, одговарајућа димензија резултата биће једнака одгова-

рајућој димензији другог операнда. Интуитивно, димензија дужине један се продужава како би се поклопила са дужом димензијом копирањем елемента који на тој димензији садржи. Наравно, у реалности се не извршава никакво копирање и broadcasting омогућава ефикасније извршавање аритметичких операција над низовима са различитим вредностима атрибута `shape` него када би се исти резултат реализовао коришћењем петљи.

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Слика 3.2: Broadcasting

3.2 AST парсер

Модул AST (Abstract Syntax Tree) је парсер за Python написан у Python-u и омогућава грађење апстрактног синтаксичког стабла од задатог Python кôда. Резултујуће стабло добија се додавањем `ast.PyCF_ONLY_AST` flag-а позиву уграђене `compile()` функције или позивањем `ast.parse()` функције.

Класа AST је базна класа свим класама овог модула које представљају чворове апстракстног синтаксичког стабла. За сваки симбол леве стране правила извођења граматике постоји одговарајућа класа чвора и додатно, за сваки симбол десне стране правила извођења постоји класа која наслеђује класу симбола леве стране из којег тај симбол може да се изведе.

Представљање сваког елемента Python кôда објектом омогућава велику флексибилност за анализирање и модификацију кôда Python програма. Класе којима припадају ови објекти могу се поделити на:

- *литерале* - у које спадају класе `Num`, `Str`, `NameConstant` (која се користи за представљање вредности `True`, `False` и `None`), `Bytes`, `Tuple`, `List`, `Set`, `Dict`, `Generator`
- *променљиве* - међу којима је најважнија класа `Name`, која као атрибуте садржи идентификатор (назив променљиве) и контекст. Контекст мора бити нека од опција `ast.Load()`, `ast.Store()` или `ast.Del()`
- *изразе* - у које спадају изрази као што су функцијски позиви, променљиве, ламбда фукнције и нека операције. Класу `expr` наслеђују класе

унарних и бинарних аритметичких и логичких оператора и оператора поређења, функцијских позива, али и оператора индексирања и приступа атрибутима и класе comprehension израза за листе, скупове, генераторе и речнике.

- *наредбе* - наредбе доделе, штампања, наредбе укључивања заглавља, итд.
- *контролу тока* - `If`, `For`, `While` конструкције, конструкције за бацање и хватање изузетака и `Break` и `Continue` наредбе
- *функције и класе* - дефиниције функција и класа, аргументи функција, наредбе `Return` и `Yield`

AST модул такође поседује и функцију `walk()` за обилажење стабла. Постоји и механизам за напреднији обилазак стабла, који омогућава класа `NodeVisitor`. Потребно је наследити ову класу и предефинисати функције `visit_NodeClassName`, где `NodeClassName` треба заменити одговарајућим називом чвора класе, а сама функција треба да дефинише акције које желимо да буду предузете при посети конкретног чвора. На сличан начин фунцкионише и класа `NodeTransformer`, коју користимо када при обиласку стабла желимо да модификујемо чворове. Још неке од корисних функција су и функције `iter_fields()` и `iter_child_nodes()` које омогућавају итерирање над пољима или чворовима који су директни наследници задатог чвора, и функција `dump()` која омогућава штампање синтаксичког стабла.

Глава 4

Алат CLIPS

Алат CLIPS (C Language Integrated Production System) развијен је у NASA-Johnson Space центру и представља један од најпопуларнијих алата за грађење *експертских система*. Експертски системи представљају програме дизајниране да симулирају процес доношења одлука и решавања проблема који би применио доменски експерт за домен којем задати проблем припада.

Експертски системи су најчешће засновани на извођењу закључака на основу унапред дефинисаних правила, која се углавном састоје из *if* и *then* дела или леве и десне стране правила. Дакле, лева страна правила дефинише услове под којима се правило извршава, а десна страна представља скуп акција које прате извршавање правила. Експертски системи засновани на правилима функционишу уз помоћ *базе знања* и *механизма за закључивање*. База знања садржи скуп правила закључивања, а механизам за закључивање служи да, на основу правила закључивања које се налазе у бази знања и унапред дефинисаних чињеница, изведе нове чињенице или изврши неке друге акције.

Механизам за закључивање у оквиру CLIPS-а имплементиран је уз помоћ RETE алгоритма[1], specijalizovanog za efikasno решавање проблема *поклапања шаблона*, тј. проблема примене већег броја правила на већи број чињеница.

4.1 Дефинисање чињеница

Чињенице (енгл. Facts) у CLIPS-у представљају једну од основних форми представљања информације и могу бити уређене или неуређеме. састоје се из:

[1] `https://en.wikipedia.org/wiki/Rete_algorithm`

- имена
- уређене n-торке вредности, у случају уређених чињеница, или скупа парова облика (име-атрибута, вредност атрибута), у случају неуређених чињеница

```
(numbers 1 2 3)
(point (x-coordinate 103) (y-coordinate -5))
```

Слика 4.1: Пример уређене и неуређене чињенице

Вредности у оквиру чињеница су литерали неког од примитивних типова којима CLIPS располаже (float, integer, symbol, string).

Неке од операција над чињеницама су додавање у листу чињеница, брисање из листе чињеница и модификација и реализују се редом командама `assert`, `retract` и `modify`.

Коришћењем `deftemplate` конструкције, могуће је креирати структуру чињенице додељивањем назива сваком пољу у оквиру чињенице.

```
(deftemplate object
   (slot name)
   (slot location)
   (slot on-top-of)
   (slot weight)
   (multislot contents))
```

Слика 4.2: пример deftemplate конструкције

Кључном речју `slot` прецизира се да конкретно поље чињенице садржи тачно једну вредност, док `multislot` поља могу садржати нула или више вредности.

Синтакса ове конструкције дата је на слици 4.4.

Могуће је постављати ограничења пољима као што су тип или опсег вредности које поље може да садржи, као и прецизирати подразумевану вредност поља која се додељује када се при инстанцирању чињенице вредност тог поља не наведе.

4.2 Дефинисање класа и креирање инстанци

У оквиру CLIPS-а дефинисан је и Clips Object-Oriented Language (COOL), који омогућава подршку концепата објектно оријентисаног стила програмирања

```
(deftemplate <deftemplate-name> [<comment>]
   <slot-definition>*)
<slot-definition> ::= <single-slot-definition> |
                      <multislot-definition>
<single-slot-definition>
               ::= (slot <slot-name>
                         <template-attribute>*)
<multislot-definition>
               ::= (multislot <slot-name>
                              <template-attribute>*)
<template-attribute> ::= <default-attribute> |
                         <constraint-attribute>
<default-attribute>
               ::= (default ?DERIVE | ?NONE | <expression>*) |
                   (default-dynamic <expression>*)
```

Слика 4.3: синтакса deftemplate конструкције

као што су наслеђивање, енкапсулација и полиморфизам и дефинисање реактивних класа које такође могу да представљају шаблоне за поклапање у оквиру леве стране правила.

Класе се дефинишу уз помоћ `defclass` конструкције, чија је синтакса слична `deftemplate` конструкцији, али у случају дефинсања класа имамо и могућности избора базне класе, прецизирања која поља класе желимо да буду реактивна, прецизирање која поља класе желимо да буду приватна, дефинисање класних метода...

Инстанце класе се креирају уз помоћ `make-instance` наредбе.

```
(defclass A (is-a USER)
   (slot a1 (visibility private))
   (slot a2))

(defclass B (is-a A)
   (slot b1 (pattern-match non-reactive))
   (slot b2))
```

Слика 4.4: Пример дефинисања класа. Класа A наслеђује класу USER, која представља базну класу свим класама у CLIPS-у, а класа B наслеђује класу A. Поље a1 означено је као приватно, а поље b1 као нереактивно.

4.3 Дефинисање правила

Као што је већ поменуто, правила се састоје од леве и десне стране, одвојене знаком `=>`.

Кондиционални елементи леве стране правила могу да се састоје од шаблона чињеница или објеката чија поља могу да буду представљена константним вредностима или променљивама. Коришћење везаних променљивих омогућава да се

```
(defrule <rule-name> [<comment>]
[<declaration>]
<conditional-element>*
=>
<action>*)
```

Слика 4.5: Синтакса за дефинисање правила.

промењљиве користе на више места и у левој и у десној страни правила, референшући на исту вредност. Кондиционални елементи могу да буду и произвољни предикати састављени од уграђених или кориснички дефинисаних функција. Додатну флексибилност у изградњи леве стране правила омогућавају логички везници који могу да се користе над кондиционалним елементима.

Десна страна правила представља секвенцу акција које треба да се изврше када се правило активира. Ове акције могу да буду додавање, брисање или модификација чињеница или објеката или извршавање уграђених или кориснички дефинисаних функција, уз могућност коришћења процедуралне контроле тока извршавања уз помоћ конструкција облика `for`, `if` и `while`.

```
(deffunction distance (?x1 ?y1 ?x2 ?y2)
   (bind ?dx (- ?x1 ?x2))
   (bind ?dy (- ?y1 ?y2))
   (sqrt (+ (* ?dx ?dx) (* ?dy ?dy)))
)

(defrule calculate-distance
   (point ?x1 ?y1)
   (point ?x2 ?y2)
=>
   (assert
      (distance (distance ?x1 ?y1 ?x2 ?y2))
   )
)
```

Слика 4.6: Пример дефинисања функције и правила.

4.4 Основни циклус извршавања правила

Када је скуп правила у бази знања изграђен и када се креирају листе чињеница и инстанци реактивних класа, извршавање правила може да се покрене наредбом `run`.

Правила чији је услов леве стране испуњен смештају се у *агенду*, одакле се бира следеће правило које ће бити извршено. Редослед извршавања правила која се налазе у агенди установљава се на основу приоритета правила (које је могуће дефинисати при креирању правила постављањем атрибута `salience` на неку од вредности у опсегу од -10 000 до 10 000), или на основу *стратегије за*

разрешавање конфликта. У оквиру CLIPS-а дефинисано је неколико стратегија за разрешавање конфликта, а подразумевана стратегија је `DEPTH` стратегија која подразумева да се увек прво извршава правило које је последње смештено у агенду. Подразумевана `salience` вредност сваког правила је 0.

Циклус извршавања правила може да се прекине у два случаја - ако је достигнуто унапред постављено ограничење броја правила које могу да се изврше или ако су сва правила чији је услов леве стране испуњен већ извршена.

Глава 5

Проблем SAT

Проблем SAT или проблем испитивања задовољивости (енгл. Boolean satisfiability problem) представља проблем одлучивања да ли за задату формулу исказне логике у конјуктивној нормалној форми (КНФ) постоји интерпретација у којој формула има истинитосну вредност 1. Уколико оваква интерпретација не постоји, формула је незадовољива.

Проблем SAT је први проблем за који је доказано да је НП-комплетан (Кук-Левинова теорема). Доказ Кук-Левинове теореме показује да сваки проблем одлучивања из класе НП може бити сведен на проблем SAT за формуле у КНФ. Проблем SAT има значајну практичну примену у различитим областима, а посебно у вештачкој интелигенцији, аутоматском доказивању теорема, верификацији софтвера и дизајну електронских кола.

5.1 Формулација проблема и варијанте

Претпоставка проблема је да је формула чија се задовољивост испитује исказна формула у КНФ. То значи да је формула састављена од конјункције *клауза*. Клаузе су састављене од дисјункције *литерала* који представљају променљиве, које евентуално могу бити негиране. На пример, формула

$$(\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \tag{5.1}$$

је формула у КНФ. Уз помоћ правила Булове алгебре, сваку формулу исказне логике могуће је свести на КНФ форму, међутим, могуће је да добијена формула буде експоненцијално дужа од полазне.

Валуација је функција која исказним променљивим додељује истинитосне вредности 0 или 1. Истинитосна вредност формуле при датој валуацији дефинише се на основу логичких везника и за валуацију кажемо да представља модел неке формуле, ако је њена истинитосна вредност при тој валуацији 1. Формула 5.1 је задовољива, јер валуација која свакој од променљивих x_1, x_2 и x_3 додељује истинитосну вредност 0 сведочи да постоји интерпретација у којој ова формула има истинитосну вредност 1, тј. поменута валуација је један модел ове формуле.

У пракси се често срећу и неки од специјалних случајева проблема SAT као што су, на пример, kSAT - варијанта проблема SAT у коме су све клаузе састављене од тачно `k` литерала. Посебно популарна варијанта је проблем 3SAT, за које су развијене посебне технике решавања, али је и овај проблем NP-комплетан, док варијанта проблема SAT где је свака клауза састављена од највише 2 литерала због своје једноставне структуре омогућава решавање у полиномијалном времену. XOR-SAT представља варијанту проблема SAT где клаузе формуле представљају ексклузивне дисјункције литерала. Овај проблем је могуће представити и као систем линеарних једначина по модулу 2, па је могуће решити га и Гаусовом методом елимнације у кубној временској сложености.

Неке од надоградњи проблема SAT укључују испитивање задовољивости формула логике првог или другог реда, испитивање задовољивости формула са задатим ограничењима и решавање проблема 0-1 целобројног програмирања, тј. псеудо-буловског проблема (проблема где ограничења представљају линеарне једнакости или неједнакости променљивих, а променљивама могу бити додељене само вредности 0 или 1).

5.2 SAT решавачи

НП-комплетност проблема SAT значи да сви данас познати алгоритми за његово решавање имају експоненцијалну сложеност у најгорем случају. Међутим, развијене су бројне хеуристике и алгоритми који у неким случајевима веома ефикасно решавају овај проблем. Најефикаснији модерни SAT решавачи могу се, на основу приступа решавању, поделити у две класе - *учење клауза вођено конфликтима (енгл. conflict-driven clause learning - CDCL)*, које представља унапређену верзију DPLL алгоритма, заснованог на систематичној претрази уз

помоћ backtracking-а и *стохастичку локалну претрагу*, у коју спадају WalkSat и GSAT алгоритми.

Такође су популарне и применљиве и непотпуне SAT методе. Алгоритми непотпуних метода не претражују цео простор могућих решења, већ се заснивају на одређеној стратегији претраге могућих решења. Уколико се решење не налази у простору који се претражује, овакви решавачи ће пријавити незадовољивост, чак и ако то није исправна одлука. Алгоритми засновани на стохастичкој локалној претрази најчешће су непотпуни.

Глава 6

Приступ оптимизацији заснован на анализи еквиваленција

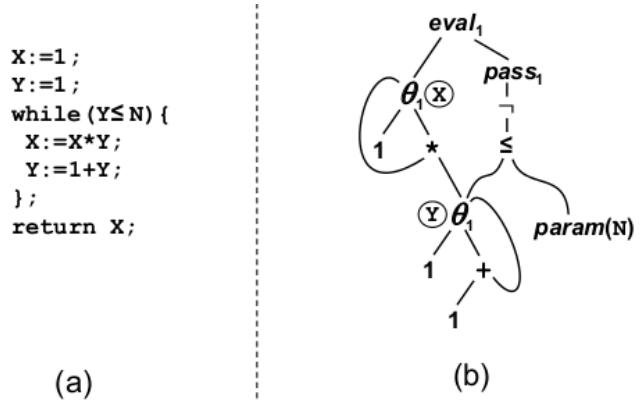
У поглављу о оптимизацији поменуто је да традиционални приступ оптимизацији програмских преводиоца подразумева секвенцијалне примене оптимизационих техника. То значи да свака оптимизациона техника добија верзију програма коју је трансформисала претходно примењена оптимизациона техника, па у општем случају различити редоследи примене оптимизационих техника доводе до различитих крајњих резултата. Могуће су и ситуације у којима примена неких оптимизационих техника отвара пут ка примени неких других оптимизационих техника, али и ситуације у којима примена неких оптимизационих техника онемогућава примену других. Проблем зависности квалитета оптимизације од редоследа примене оптимизационих техника познат је као *проблем уређивања фаза (енгл. phase ordering problem)*. Још један недостатак овог приступа је и чињеница да се приликом одлучивања да ли у конретном тренутку применити одређену оптимизациону технику тешко може узети у обзир какве ће последице ова одлука имати на будуће отимизације.

Аутори рада Equality Saturation: A New Approach to Optimization [3], представили су приступ који превазилази поменути проблем. Приступ је заснован на израчунававању скупа могућих оптимизованих верзија полазног програма из кога се затим бира оптимални кандидат. Основу реализације овог приступа чини погодна структура изабрана за ефикасно представљање већег броја верзија програма одједном. Ова структура омогућава да се избор оптимизационих техника одложи до тренутка када се размотре све могућности и онда изабере најбоља од њих. Опис и формализација приступа биће представљени у на-

ставку.

6.1 Граф програмских израза

Процес оптимизације програмских преводиоца често се изводи над неком репрезентацијом програма (на пример, међукôд, граф контроле тока, синтаксичко стабло) која је погодна за примене одређених оптимизационих техника. У овом приступу, програм се најпре преводи у репрезентацију која се назива *граф програмских израза (ГПИ)*. ГПИ је усмерени граф, који својом структуром подсећа на синтаксичко стабло, али овај граф није ацикличан. ГПИ садржи две врсте чворова - чворове који представљају операторе (+, ∗, функцијске позиве или константе које представљају нуларне операторе) и специјалне чворове контроле тока. Сваки чвор је усмереним гранама повезан са чворовима који представљају његове аргументе. Пример кôда који израчунава факторијел броја N и његова ГПИ репрезентација дати су на слици 6.1.



Слика 6.1: Пример кôда и његове ГПИ репрезентације

Променљива која варира у петљи представља се уз помоћ θ чвора. θ чвор заправо представља секвенцу вредности које променљива узима у петљи. Први аргумент θ чвора је иницијална вредност, а други аргумент је вредност коју променљива добија у наредној итерацији изражена у односу на вредност из претходне итерације. Како су вредности свих итерација неке променљиве у петљи представљене једним θ чвором, у овом случају се у графу јављају циклуси. Сваки циклус који постоји у ГПИ мора да пролази кроз чвор који представља друго дете θ чвора. Вредност коју променљива добија после извршавања петље представља се уз помоћ *eval* чвора. Овај чвор такође има два аргумента, тј.

два детета. Први аргумент је секвенца вредности, а други индекс који представља редни број итерације у којој се петља прекида. Дакле, вредност *eval* чвора је елемент секвенце његовог првог аргумента на позицији која одговара вредности његовог другог аргумента. *pass* чвор је чвор који за задату секвенцу буловских вредности израчунава индекс првог елемента чија је вредност *true* и најчешће се користи као други аргумент *eval* чвора. На слици 6.1 променљиве `X` и `Y` варирају у петљи и свака од њих је представљена засебним θ чвором. Индекс 1 који се на слици налази уз *eval*, *pass*, и θ чворове означава дубину угнежђења петље.

За разлику од представљања променљивих у петљи у ГПИ које се реализује *θ-eval-pass* конструкцијом, за представљање променљивих чија вредност зависи од неког услова гранања довољан је један чвор. То је чвор ϕ, који има три аргумента. Први аргумент представља услов гранања, други вредност у случају испуњености услова, а трећи вредност у случају неиспуњености услова.

ГПИ је *референцијално транспарентна* репрезентација програма, у смислу да вредност сваког чвора зависи искључиво од вредности чворова који су у графу достижни из тог чвора (тј. нема бочних ефеката). Осим тога, ГПИ представља ток сваке променљиве у програму, па је такође и *потпун*, у смислу да садржи све неопходне информације и да нема потребе додатно анализирати још неку репрезентацију, као што је, на пример, граф контроле тока. Иако је ГПИ референцијално транспарентан, то не значи да програми који имају бочне ефекте не могу да се представе у овој форми. У случају програма са бочним ефектима, користи се још једна специјална врста чворова која служи за представљање стања хипа. У овом раду биће речи о реализацији приступа заснованог на анализи еквиваленција само над функцијама које немају бочне ефекте и које у оквиру своје дефиниције не садрже позиве функција које имају бочне ефекте, па детаљи везани за реализацију над функцијама са бочним ефектима неће бити даље разматрани.

ГПИ репрезентација кôда неке функције садржи и посебне врсте чворова који представљају повратне вредности те функције. Ако функција има једну повратну вредност, њена ГПИ репрезентација представља коренски граф, чији је корен чвор који представља ту повратну вредност. У програмском језику Python, уз наредбу `return` можемо навести више од једне повратне вредности, али се у том случају наведене повратне вредности имплицитно групишу у n-торку, што значи да функције у Python-у увек имају највише једну повратну

вредност. Поред тога, како су функције без бочних ефеката и без повратне вредности бескорисне, за овај рад су од интереса само функције које имају тачно једну повратну вредност.

Формализација графа програмских израза и његове семантике

У оквиру описа ГПИ репрезентације сваки чвор графа представљен је као одређени оператор чији су аргументи његова деца чворови. Формално, сваком чвору додељује се одговарајућа *семантичка функција* која одређује вредност коју тај чвор израчунава, тј. његову семантичку вредност. Могући су и случајеви у којима се израчунавање семантичке функције неког чвора никад не завршава, па им је вредност недефинисана. Овакве вредности се обележавају са $\perp$ и за сваки тип вредности τ над којима оперишу функције представљене ГПИ чворовима, формално се дефинише тип $\tau_\perp = \tau \cup \perp$.

Како постоје и случајеви у којима један чвор не представља једну вредност, већ секвенцу вреднсоти (тј. вредности свих итерација петље), семантичка вредност оваквих чворова не зависи само од аргумената, већ и од редног броја итерације. Због тога се формално уводи скуп идентификатора петљи $\mathcal{L}$, на коме је дефинисано и уређење - за $\ell, \ell' \in \mathcal{L}$ важи $\ell < \ell'$, ако је ℓ' угнежђена у ℓ. Над скупом $\mathcal{L}$ уводи се фунцкија итерацијског индекса $\mathbf{i}$ који сваком $\ell \in \mathcal{L}$ додељује редни број итерације (на пример $\mathbf{i} = [\ell_1 \to 10,\ \ell_2 \to 3]$ треба да представи стање програма у којем је петља ℓ_1 на десетој итерацији, а ℓ_2 на трећој). Нека је $\mathbb{I} = \mathcal{L} \to \mathbb{N}$ скуп свих итерацијских индекса, за сваки дефинисани тип $\tau_\perp$ формално се дефинише тип $\tilde{\tau} = \mathbb{I} \to \tau_\perp$. Семантичке функције чворова оперишу над овако дефинисаним типовима.

Формална дефиниција семантичких функција чворова који представљају контролу тока приказана је на слици 6.2. Скуп $\mathbb{B}$ представља скуп буловских вредности, а функција $monotonize_\ell$ служи да задату секвенцу вредности преслика у секвенцу у којој су после прве недефинисане вредности и све остале вредности недефинисане. Нотација $\mathbf{i}[\ell \to i]$ означава функцију која враћа вредност исту као $\mathbf{i}$ за све улазне вредности, осим што за улаз ℓ враћа вредност i.

ГПИ представља уређену четворку (N, E, λ, r), где је:

- N - скуп чворова

- $\lambda : N \to F$ - функција која пресликава чвор у семантичку функцију из задатог скупа семантичких функција F
- $E : N \to N^*$ - функција која сваки чвор пресликава уређену n-торку његове деце.
- r чвор који представља повратну вредност функције

$$\boxed{\phi : \widetilde{\mathbb{B}} \times \widetilde{\tau} \times \widetilde{\tau} \to \widetilde{\tau}}$$

$$\phi(cond, t, f)(\mathbf{i}) = \begin{cases} \textbf{if } cond(\mathbf{i}) = \bot & \textbf{then } \bot \\ \textbf{if } cond(\mathbf{i}) = \textbf{true} & \textbf{then } t(\mathbf{i}) \\ \textbf{if } cond(\mathbf{i}) = \textbf{false} & \textbf{then } f(\mathbf{i}) \end{cases}$$

$$\boxed{\theta_\ell : \widetilde{\tau} \times \widetilde{\tau} \to \widetilde{\tau}}$$

$$\theta_\ell(base, loop)(\mathbf{i}) = \begin{cases} \textbf{if } \mathbf{i}(\ell) = 0 & \textbf{then } base(\mathbf{i}) \\ \textbf{if } \mathbf{i}(\ell) > 0 & \textbf{then } loop(\mathbf{i}[\ell \mapsto \mathbf{i}(\ell) - 1]) \end{cases}$$

$$\boxed{eval_\ell : \widetilde{\tau} \times \widetilde{\mathbb{N}} \to \widetilde{\tau}}$$

$$eval_\ell(loop, idx)(\mathbf{i}) = \begin{cases} \textbf{if } idx(\mathbf{i}) = \bot & \textbf{then } \bot \\ \textbf{else } monotonize_\ell(loop)(\mathbf{i}[\ell \mapsto idx(\mathbf{i})]) \end{cases}$$

$$\boxed{pass_\ell : \widetilde{\mathbb{B}} \to \widetilde{\mathbb{N}}}$$

$$pass_\ell(cond)(\mathbf{i}) = \begin{cases} \textbf{if } \mathscr{I} = \emptyset & \textbf{then } \bot \\ \textbf{if } \mathscr{I} \neq \emptyset & \textbf{then } \min \mathscr{I} \end{cases}$$

$$\text{where } \mathscr{I} = \{i \in \mathbb{N} \mid monotonize_\ell(cond)(\mathbf{i}[\ell \mapsto i]) = \textbf{true}\}$$

where $monotonize_\ell : \widetilde{\tau} \to \widetilde{\tau}$ is defined as:

$$monotonize_\ell(value)(\mathbf{i}) = \begin{cases} \textbf{if } \exists\, 0 \le i < \mathbf{i}(\ell).\ value(\mathbf{i}[\ell \mapsto i]) = \bot & \textbf{then } \bot \\ \textbf{if } \forall\, 0 \le i < \mathbf{i}(\ell).\ value(\mathbf{i}[\ell \mapsto i]) \neq \bot & \textbf{then } value(\mathbf{i}) \end{cases}$$

Слика 6.2: Дефиниције семантичких функција

6.2 Граф програмских израза са еквиваленцијама

Основна идеја приступа оптимизацији заснованог на анализи еквиваленција је да структрира више могућих оптимизованих верзија програма одједном. ГПИ репрезентација представља само једну верзију програма, али служи и као полазна тачка за креирање репрезентације која може да представи више верзија. Ова репрезентација назива се *граф програмских израза са еквиваленцијама (Е-ГПИ)*. Основна особина која разликује Е-ГПИ репрезентацију од ГПИ репрезентације је да су њени чворови груписани у класе еквиваленције, на основу једнакости семантичких вредности чворова. Када се кôд програма преведе у ГПИ, од њега се креира иницијални Е-ГПИ у коме су све класе еквиваленције једночлане, а затим се у Е-ГПИ извођењем еквиваленција проширују и додају

нове класе. Еквиваленције се закључују на основу унапред задатих аксиома (на пример, $a+0 \Leftrightarrow a$). Свака аксиома састоји се из леве и десне стране, где је лева страна шаблон (који може да садржи и слободне променљиве) представљен у графовској форми. Када се лева страна неке аксиоме пронађе као подграф у оквиру Е-ГПИ, тада се додаје нова еквиваленција између леве и десне стране аксиоме. У случају аксиоме $a+0 \Leftrightarrow a$, додаје се еквиваленција чвора који представља оператор + и чвора који везује променљива a. У неким случајевима је приликом додавања еквиваленције потребно и инстанцирати нове чворове (на пример, у случају аксиоме $a - a \Leftrightarrow 0$ и ако у тренутку извођења еквиваленције на основу ове аксиоме у Е-ГПИ не постоји чвор који представља константу 0, тада се овај чвор додаје у Е-ГПИ). Захваљујући референцијалној транспарнтности, еквиваленције се изводе на нивоу чворова, што омогућава да замена неког чвора чвором који му је еквивалентан не мења семантику програма. На слици 6.3 приказан је пример оригиналне и оптимизоване верзије кôда, а на слици 6.4 одговарајуће ГПИ и Е-ГПИ репрезентације (прецизније представљени су делови који учествују у оптимизацији, тј. корен графа представљен је изразом `i * 5` над којим се позива функција `use()`).

```
i := 0;
while (...) {
   use(i * 5);
   i := i + 1;
   if (...) {
      i := i + 3;
   }
}
```

(a)

```
i := 0;
while (...) {
   use(i);
   i := i + 5;
   if (...) {
      i := i + 15;
   }
}
```

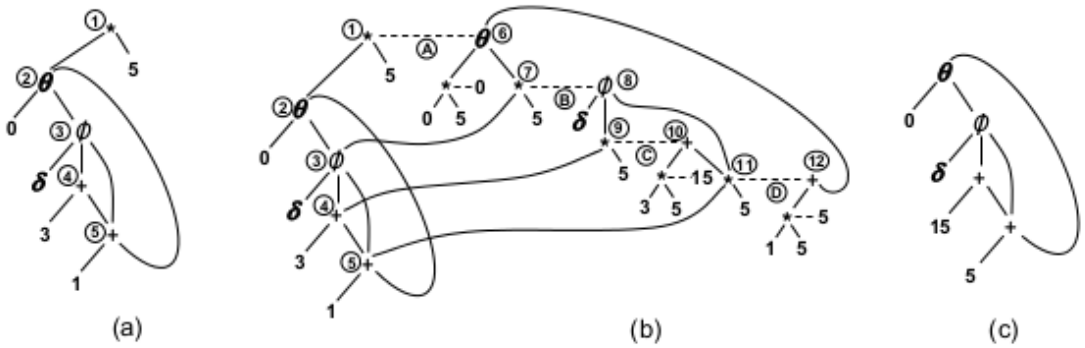
(b)

Слика 6.3: Пример полазног кôда и кôда након примене оптимизационе технике познате као редукција снаге

На слици 6.4 су еквивалентни чворови повезани гранама обележеним испрекиданим линијама. За извођење еквиваленција у овом примеру коришћене су следеће аксиоме:

1. $(a + b) * c \Leftrightarrow a * c + b * c$
2. $\theta(a, b) * c = \theta(a * c, b * c)$
3. $\phi(a, b, c) * d = \phi(a, b * d, c * d)$

Грана A додата је на основу друге аксиоме, грана B на основу треће, а гране C и D на основу прве. Осим наведене три аксиоме, у примеру су коришћене и аксиоме пропагације константи $0 * 5 \Leftrightarrow 0$, $3 * 5 \Leftrightarrow 15$ и $1 * 5 \Leftrightarrow 5$.



Слика 6.4: (a) и (c) приказују редом делове ГПИ репрезентација полазне и оптимизоване верзије кôда са слике 6.3, а (b) представља Е-ГПИ са изведеним еквиваленцијама. Услов гранања над којим оперише чвор ϕ у овом случају није релевантан, па је због једноставности означен са δ.

Овај пример показује како Е-ГПИ структурира више верзија програма који желимо да оптимизујемо одједном, што омогућава да се одлука о избору чворова представника класа еквиваленција донесе након што су размотрене све могуће опције и да различити редоследи примене аксиома не доводе до различитих крајњих резултата. Заправо, Е-ГПИ је компактна репрезентација експоненцијално много верзија програма, тј. ако посматрамо чвор n који, на пример, има три детета, која редом припадају класама еквиваленција C_1, C_2 и C_3, тада за чвор n постоји укупно $|C_1| \cdot |C_2| \cdot |C_3|$ могућих варијанти избора деце овог чвора.

Формализација графа програмских израза са еквиваленцијама

Е-ГПИ је уређена петорка (N, E, λ, r, C), где N, E и λ имају исто значење као и у случају ГПИ. Поред тога, над скупом чворова N дефинише се и релација еквиваленције $\sim$. За $n_1, n_2 \in N$ важи $n_1 \sim n_2$, ако чворови n_1 и n_2 имају једнаке семантичке вредности. Релација $\sim$ разбија скуп N на међусобно дисјунктне класе еквиваленције. C представља скуп класа еквиваленције релације $\sim$, а r класу еквиваленције чвора који представља повратну вредност функције.

6.3 Конверзије између програмског кôда и ГПИ репрезентације

Како би представили идеју процеса превођења кôда императивног програмског језика у ГПИ репрезентацију и превођења ГПИ репрезентације у кôд, аутори овог приступа оптимизацији дефинисали су једноставан императивни програмкси језик SIMPLE.

$$
\begin{aligned}
p &::= \texttt{main}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \{s\} \\
s &::= s_1; s_2 \mid x := e \mid \textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{while}\ (e)\ \{s\} \\
e &::= x \mid op(e_1, \ldots, e_n)
\end{aligned}
$$

Слика 6.5: Граматика програмског језика SIMPLE

Програм језика SIMPLE састоји се из функције `main` са декларисаним параметрима и њиховим типовима, типом повратне вредности и телом ове функције. Сваки исказ `s` има форму наредбе доделе, наредбе облика `if-then-else` или `while` петље или представља надовезивање исказа. Изрази могу бити или променљива или неки примитивни оператор. Програмски језик `SIMPLE` може имати произвољан скуп примитивних оператора који оперишу над произвољним скупом типова. Једини захтев је да скуп типова садржи и тип `bool`, коме припадају изрази који представљају услове гранања и петљи. Скуп променљивих садржи и једну специјалну променљиву `retvar`, која садржи повратну вредност функције `main` на крају њеног извршавања.

Једноставна синтакса програмског језика SIMPLE омогућава и релативно једноставне алгоритме конверзије, али ако би се његова синтакса допунила компликованијом контролом тока, коју омогућавају, на пример, наредбе `break` и `continue`, ови алгоритми постају приметно сложенији. Целокупан процес конверзије са доказима коректности представљен је у раду посвећеном техничким детаљима везаним за ове конверзије [2]. У наставку ће бити укратко приказани алгоритми конверзије над програмским језиком SIMPLE, који служе као смерница за реализацију овог процеса над програмима неког конкретног императивног језика.

Превођење императивног кôда у ГПИ репрезентацију

Псеудокôд алгоритма превођења кôда програма програмског језика `SIMPLE` у ГПИ репрезентацију представљен је на слици 6.6.

```
1:  function TranslateProg(p : Prog) : N =
2:      let m = InitMap(p.params, λx. param̄(x))
3:      in TS(p.body, m, 0)(retvar)

4:  function TS(s : Stmt, Ψ : map[V,N], ℓ : ℕ) : map[V,N] =
5:      match s with
6:          "s1;s2" ⇒ TS(s2, TS(s1, Ψ, ℓ), ℓ)
7:          "x := e" ⇒ Ψ[x ↦ TE(e, Ψ)]
8:          "if (e) {s1} else {s2}" ⇒ PHI(TE(e, Ψ), TS(s1, Ψ, ℓ), TS(s2, Ψ, ℓ))
9:          "while (e) {s}" ⇒
10:             let vars = Keys(Ψ)
11:             let Ψ_t = InitMap(vars, λv. TemporaryNode(v))
12:             let Ψ' = TS(s, Ψ_t, ℓ + 1)
13:             let Ψ_θ = THETA_{ℓ+1}(Ψ, Ψ')
14:             let Ψ'_θ = InitMap(vars, λv. FixpointTemps(Ψ_θ, Ψ_θ(v)))
15:             in EVAL_{ℓ+1}(Ψ'_θ, pass̄s_{ℓ+1}(¬̄(TE(e, Ψ'_θ))))

16: function TE(e : Expr, Ψ : map[V,N]) : N =
17:     match e with
18:         "x" ⇒ Ψ(x)
19:         "op(e1, ..., ek)" ⇒ ōp(TE(e1, Ψ), ..., TE(ek, Ψ))

20: function PHI(n : N, Ψ1 : map[V,N], Ψ2 : map[V,N]) : map[V,N] =
21:     Combine(Ψ1, Ψ2, λ t f . φ̄(n, t, f))

22: function THETA_{ℓ:ℕ}(Ψ1 : map[V,N], Ψ2 : map[V,N]) : map[V,N] =
23:     Combine(Ψ1, Ψ2, λ b n . θ̄_ℓ(b, n))

24: function EVAL_{ℓ:ℕ}(Ψ : map[V,N], n : N) : map[V,N] =
25:     InitMap(Keys(Ψ), λv . eval̄_ℓ(Ψ(v), n))

26: function Combine(m1 : map[a,b], m2 : map[a,c], f : b * c → d) : map[a,d] =
27:     InitMap(Keys(m1) ∩ Keys(m2), λk. f(m1[k], m2[k]))
```

Слика 6.6: Алгоритам превођења

Нотација $\overline{a}(n_1, ..., n_k)$ користи се да означи ГПИ чвор чија је семантичка функција a, како би се направила дистинкција између функција псеудокôда и семантичких функција чворова.

Основна структура која се користи у процесу превођења је *контекст чворова* Ψ, који представља речник састављен од уређених парова облика $x : n$, где x представља промењљиву програмског језика SIMPLE, а n ГПИ чвор. Над овом структуром дефинисане су функције:

- `InitMap`(K, f), која иницијализује речник од задатог скупа кључева K и функције f која пресликава кључеве у вредности

- `Keys`(Ψ), која враћа скуп кључева речника Ψ,
- Функција $\Psi[k \to d]$ за ажурирање речника, која враћа ажурирани речник Ψ са постављеном вредношћу d уз кључ k,
- Функција $\Psi(k)$ за читање вредности Ψ која у речнику одговара кључу k.

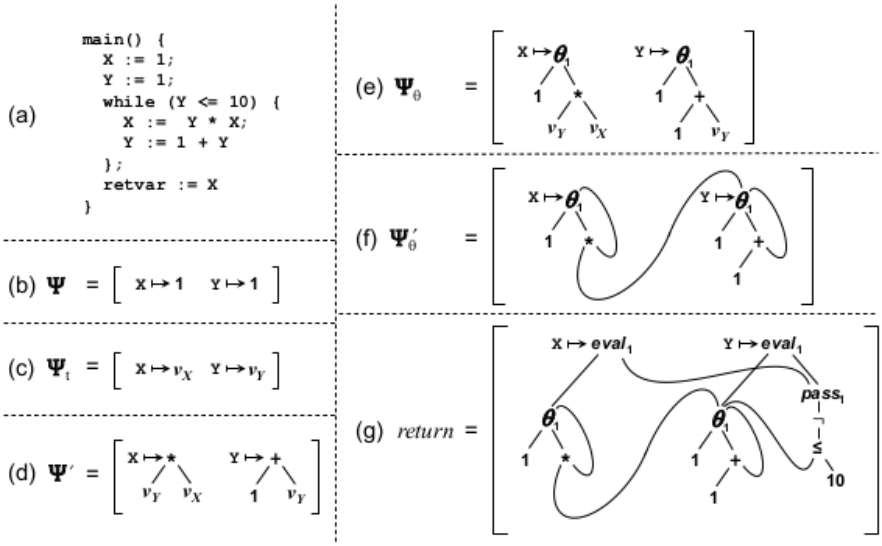
У оквиру псеудокôда користе се и типови *Prog*, *Stmt*, *Expr* и V, да представе редом програм, исказ, израз и променљиву. За задати програм p, $p.body$ и $p.params$ представљају редом тело `main` функције програма p и листу њених параметара.

Параметарски чворови, облика $\overline{param}(x)$, служе за представљање параметара функције.

Изрази се у ГПИ чворове преводе уз помоћ функције `TE()`, у односу на текући контекст Ψ. У случају да је израз e променљива, повратна вредност ове функције је одговарајући чвор који се налази у контексту уз кључ који представља назив те променљиве, а у случају да је израз e оператор, рекурзивно се преводи сваки од његових аргумената, а затим се конструише чвор чија семантичка функција одговара том оператору.

Искази у процесу превођења уз помоћ функције `TS` ажурирају текући контекст. Осим самог исказа и контекста, додатни аргумент ове функције је и дубина угнежђења тренутног исказа. Надовезани искази се преводе рекурзивно, тако што се контекст који је ажуриран превођењем првог исказа користи као контекст у односу на који се преводи други исказ. Додела израза променљивој у контекст уводи нови пар чији је кључ променљива, а вредност израз преведен у ГПИ чвор. У случају наредби облика `if-then-else`, услов гранања се преводи у чвор уз помоћ `TE()`, а за обе гране се затим рекурзивно креирају два одговарајућа контекста, над којима се уз помоћ функције `PHI()` креирају ϕ чворови, који представљају чворове гранања.

Процес превођења петљи је најкомпликованији случај и пример овог процеса представљен је на слици 6.7. У оквиру овог процеса израчунава се пет додатних контекста. У текућем контексту Ψ се налазе променљиве иницијализоване пред извршавање петље, контекст Ψ_t садржи привремене копије сваке од променљивих, а затим се над овим контекстом и телом петље позива фукнција `TS()`, да израчуна контекст Ψ' који представља итерацију петље. Контекст Ψ_θ затим уз помоћ функције `THETA()` израчунава чворове вредности променљиве у петљи, тј. θ чворове над задатим контекстом, док $\Psi_{\theta'}$ служи да замени привремене



Слика 6.7: Пример превођења наредбе while

променљиве оригиналним (уз помоћ функције `FixpointTemps`, чија имплементација није приказана). Над овако креираним контекстом позива се функција `EVAL()` која, уз рекурзивно израчунавање услова прекидања петље, употпуњује $\theta - eval - pass$ конструкцију.

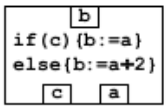
Напомена: у процесима превођења петљи и гранања, функције `PHI()` и `THETA()` граде ϕ и θ чворове над свим вредностима које се налазе у текућем контексту. Зато се чворови облика $\phi(C, A, A)$ (чвор који чија семантичка вредност не зависи од услова гранања) и $T = \theta(I, T)$ (чвор који се не мења у итерацији), који из овог разлога могу да буду присутни у резултујућем ГПИ, замењују редом са A и I.

Програми језика SIMPLE функцијом `TranslateProg` започињу процес превођења постављањем чворова који одговарају аргументима `main` функције у иницијални контекст, а затим се тело `main` функције преводи уз задати иницијални контекст, са постављеном дубином угнежђења на 0. Повратна вредност функције `TranslateProg` је вредност која у резултујућем контексту одговара променљивој `retvar` и ова вредност представља корен ГПИ репрезентације програма.

Превођење ГПИ репрезентације у императивни кôд

Смер конверзије из ГПИ репрезентације у императивни кôд је сложенији процес од обратног смера. Најзахтевнији део овог процеса узрокован је при-

суством чворова који омогућавају представљање контроле тока, тј. чворова ϕ, θ, *eval* и *pass*. Зато се у овом смеру конверзије уводи нови тип чвора - *чвор наредбе*. Чвор наредбе служи да одговарајућу наредбу s представи као примитивни оператор који, за задату листу улазних вредности, извршавањем наредбе s производи листу излазних вредности. Улазним и излазним вредностима су у случају чворова наредби додељени и називи променљивих.

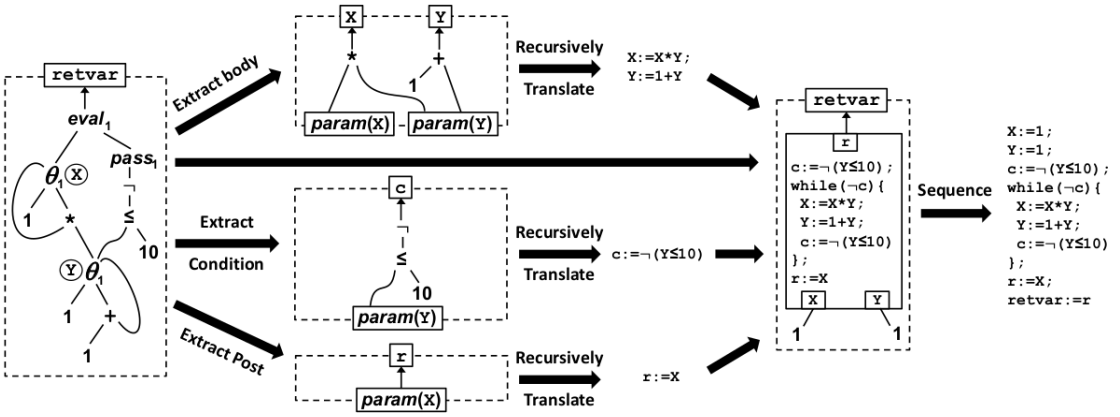


Слика 6.8: Пример чвора наредбе. Листу улазних вредности чине чворови обележени променљивама c и a, а излазна вредност променљивом b.

Основна идеја на којој се заснива коришћење чворова овог облика је замена *eval*, *pass* и θ чворова чворовима наредби `while` и замена ϕ чворова чворовима наредби `if-then-else`.

Алгоритам конверзије почиње заменом *eval*, *pass* и θ чворова чворовима наредби и ова замена резултује ацикличном ГПИ репрезентацијом. Над њом се даље извршава замена свих ϕ чворова, како би се добила ГПИ репрезентација која се састоји само из примитивних оператора. ГПИ репрезентација са оваквом структуром омогућава генерисање кôда секвенцијалним надовезивањем наредби одређених чворовима почевши од листова ГПИ репрезентације. У случају чворова наредби, надовезује се наредба коју дати чвор представља, а у осталим случајевима, за дати чвор који представља неки оператор, надовезује се наредба доделе која новој променљивој додељује тај оператор примењен над његовим аргументима. Алгоритам за превођење је рекурзиван и као улаз очекује контекст чворова, а као излаз враћа исказ програма SIMPLE (иницијални контекст се креира као једночлани речник састављен од кључа који чини променљива `retvar`, а вредност корен ГПИ репрезентације).

Превођење петљи чини прву фазу превођења и изводи се прво над спољашњим *eval* чворовима, док се угнежђени *eval* чворови преводе рекурзивно, у оквиру превођења тела петљи. За сваки спољашњи *eval* чвор e проналазе се θ чворови који су достижни из овог чвора путевима који не садрже друге *eval* чворове и који имају исту дубину угнежђења као e и смештају се у скуп S. Затим се за сваки чвор из S креира нова променљива која ће у програму фигурисати у петљи одређеној чвором e.



Слика 6.9: Пример превођења θ, *eval* и *pass* чворова

На слици 6.9, оба θ чвора су достижна из *eval* чвора, па се сваком од њих додељује нова променљива (у овом примеру X и Y). Затим се процес превођења наставља креирањем три нова контекста чворова Ψ_i, Ψ_c и Ψ_r, који служе да представе редом итерацију петље, услов прекидања петље и вредности после извршавања петље.

Контекст чворова Ψ_i конструише се тако што се за све чворове из S додају парови кључева, које чине одговарајуће нове креиране променљиве, и вредности, које чине *модификоване копије* другог детета (аргумента θ чвора који представља вредност у итерацији) сваког од ових чворова. Модификована копија неког чвора подразумева копију коренског подграфа који почиње у том чвору, али су сви чворови из скупа S замењени параметарским чворовима променљивих које су им додељене. Овај контекст чворова служи да представи тело петље.

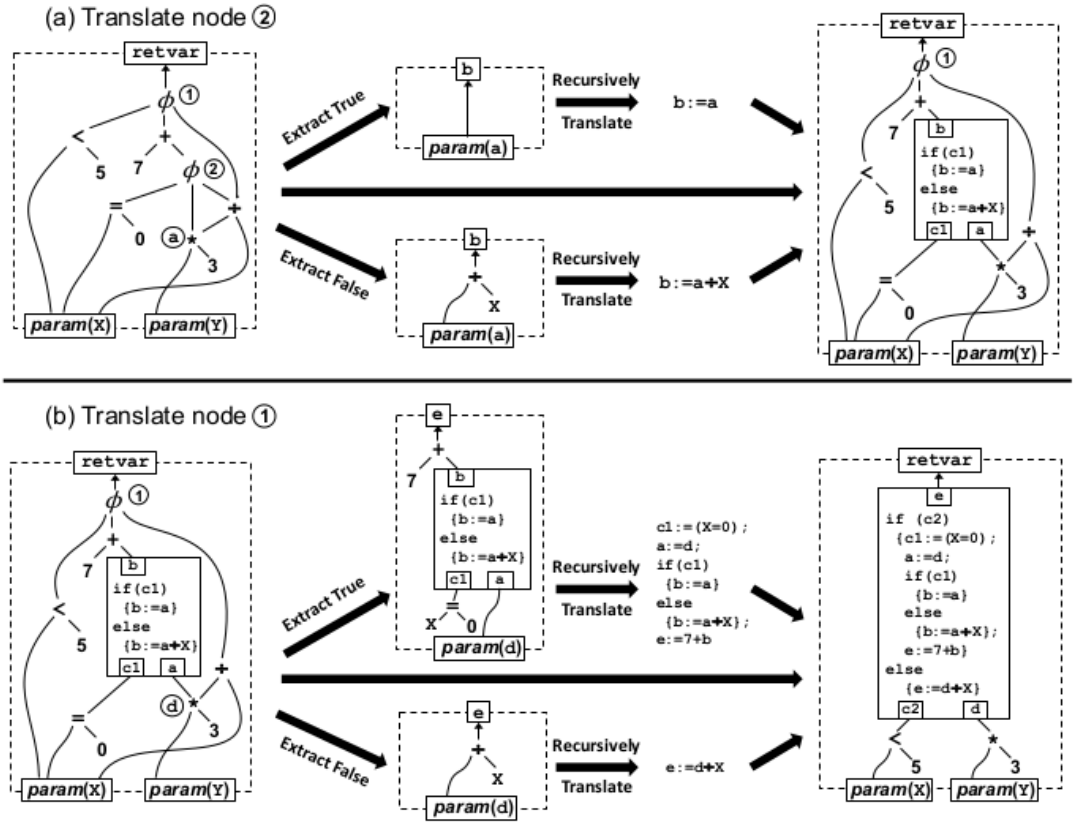
На сличан начин се креира и контекст чворова Ψ_c, за представљање услова прекидања петље. Ψ_c садржи само једну нову променљиву c као кључ, чија је вредност модификована копија детета *pass* чвора, који одговара чвору e.

Такође, креира се и контекст чворова Ψ_r, који служи да представи вредност добијену после извршавања петље, изражену преко променљивих коришћених у петљи. Овај контекст се гради од модификоване копије првог детета чвора e, коме се као кључ додељује нова променљива r.

Контексти чворова Ψ_i, Ψ_c и Ψ_r се затим преводе рекурзивно. Нека су резултати њиховог превођења редом искази s_i, s_c и s_r. Исказ s_c представља наредбу доделе која додељује услов прекидања петље променљивој c, s_i представља це-

локупно тело петље, а s_r наредбу доделе крајње вредности . Следећи корак је креирање чвора наредбе s, где је $s = s_c$; `while`$(\neg x_c)\{s_i; s_c\}$; s_r. Чвору наредбе се затим за сваки θ чвор t из S као улаз надовезује прво дете чвора t (аргумент који представља иницијалну вредност), при чему се уз сваки од улаза обележава променљивом додељеном том чвору. Излаз овог чвора обележава се променљивом r.

Превођење гранања спроводи се заменом ϕ чворова `if-then-else` наредбама. После превођења петљи, ГПИ постаје ацикличан и у првом кораку преводе се ϕ чворови који немају друге ϕ чворове као наследнике. За сваки овакав ϕ чвор p проналазе се чворови који представљају заједничке наследнике другог и трећег детета (`true` и `false` аргумената) чвора p и смештају се у скуп S. Као и у случају превођења петљи, сваком чвору из S се додељује нова променљива.



Слика 6.10: Пример превођења ϕ чворова

На слици 6.10(а), чвор `*` је наследник и другог и трећег детета ϕ чвора који се преводи, па се чвору `*` додељује нова променљива `a` (у овом примеру, и чвор који представља константу `3` припада скупу S, али на слици није обележен

променљивом, јер се та променљива у даљем процесу не користи). Скуп S представља чворове који ће се извршити у оба случаја, независно од услова гранања. Слично као и у процесу превођења петљи, и у овом случају се креирају додатни контексти чворова - Ψ_t и Ψ_f, који представљају `true` и `false` случајеве.

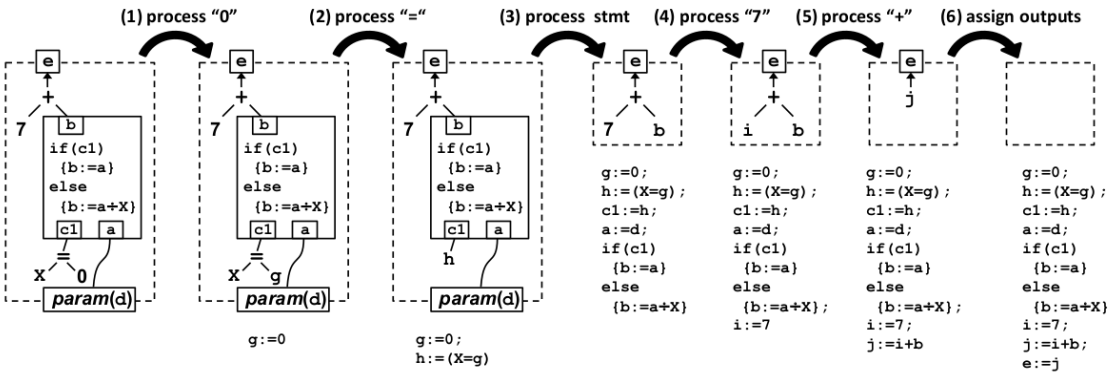
Како би се креирали контексти чворова Ψ_t и Ψ_f, прво се креирају модификоване копије (у односу на скуп S) другог и трећег детета чвора p - t и f. Копије t и f представљају `true` и `false` случајеве изражене преко променљивих које представљају вредности које се израчунавају у оба случаја. Затим се креира нова променљива x_p која се користи као кључ у оба контекста и контруишу се контексти $\Psi_t = \{x_p : t\}$ и $\Psi_f = \{x_p : f\}$.

Контексти чворова Ψ_t и Ψ_f преводе се рекурзивно. Нека су резултати њиховог превођења редом искази s_t и s_f. Чвор p се затим замењује чвором наредбе s, где је $s =$ `if` (x_c) $\{s_t\}$ `else` $\{s_f\}$, а x_c нова креирана променљива. Прво дете чвора p надовезује се на улаз чвора наредбе који се обележава променљивом x_c, а затим се надовезују и сви чворови из S, обележени променљивама које су им додељене. Излаз чвора наредбе обележава се са x_p.

Надовезивање наредби је последња фаза у процесу превођења контекста чворова у кôд у којој су сви чворови контекста који се преводи или чворови примитивних оператора или чворови наредби. Ова фаза започиње иницијализацијом променљиве s на празну наредбу. Надовезивање започиње од чворова листова. Основна идеја је да се сваки чвор у овом редоследу преведе у наредбу, а затим се замени параметарским чвором. У случају да је чвор који се тренутно разматра примитивни оператор, тј. има облик $\overline{op}(\overline{param}(x_1), ..., \overline{param}(x_k))$ (јер су му у тренутку разматрања сва деца замењена параметарским чворовима), тада се на s надовезује наредба $x = op(x_1, ...x_k)$, где је x нова креирана променљива.

У случају да је чвор који се тренутно разматра чвор наредбе, приступ је сличан, али се деца чворови овог чвора додељују променљивама којима су обележени улази тих чворова у оквиру чвора наредбе (што омогућава иницијализацију свих променљивих које се у наредби користе), а потребно је надовезати и саму наредбу коју чвор наредбе садржи.

Описани процес превођења ГПИ репрезентације у императивни кôд представља само основни алгоритам превођења који производи резултат који није ефикасан и који садржи и неке непотребне наредбе или поновљања делова кôда. Како би се овај проблем превазишао, потребно је надоградити основни алгори-



Слика 6.11: Пример процеса надовезивања наредби

там. Целокупан поступак приказан је у научном раду посвећеном техничким детаљима процеса конверзија [2].

6.4 Избор оптималне верзије програма

Када се извођењем еквиваленција уз помоћ Е-ГПИ репрезентације представе различите семантички еквивалентне верзије програма, следећи корак је избор оптималне верзије. Да би оваква оптимизација могла да се изведе, потребно је увести одређену метрику на основу које можемо да упоредимо неке две верзије програма и одредимо која од њих је оптималнија. У том циљу, над скупом Е-ГПИ чворова дефинише се функцију $\mathbb{F} : N \to \mathbb{N}$, која сваком чвору додељује цену (која би требало да одговара брзини извршавања операције која је датим чвором представљена) и тада се избор оптималне верзије програма своди на минимизацију функције $\sum_{n \in K} \mathbb{F}(n)$ по скупу $K \subseteq N$, тако да чворови из K чине валидан ГПИ који је семантички еквивалентан полазном.

У оквиру докторске дисертације аутора Мајкла Степа [1] формално је дефинисан проблем избора оптималног валидног ГПИ из задатог Е-ГПИ, а затим је доказано да је овај проблем НП-тежак. У наставку је дата формулација овог проблема која дефинише рестрикције неопходне за испуњење услова валидности при избору ГПИ као ограничења исказне логике и псеудо-буловска ограничења.

Формулација проблема избора оптималног ГПИ као псеудо-буловског проблема

Нека је $G = (N, E, \lambda, r, C)$ Е-ГПИ и $\mathbb{F} : N \to \mathbb{Z}$ тежинска функција дефинисана над његовим чворовима.

Променљиве за псеудо-буловски проблем дефинишу се на следећи начин:

- За сваки чвор $n \in N$ уводи се променљива B_n
- За сваку класу $c \in C$ уводи се променљива B_c
- За сваки пар класа $(c_1, c_2) \in C \times C$ уводи се променљива $B_{c_1 \to c_2}$

Ограничења за псеудо-буловски проблем дефинишу се на следећи начин:

(1) $B_r = 1$.

Ово ограничење означава услов да класа која представља повратну вредност функције мора да буде изабрана.

(2) За сваку класу $c \in C$:

$B_c = \sum_{n \in C} B_n$.

Услов да се за сваку класу еквиваленције може изабрати највише један чвор представник те класе.

(3) За свако $n \in N$, где је $E(n) = (c_1, ..., c_m)$:

$B_n \Rightarrow B_{c_i}$, за свако i.

Ово ограничење означава да избор неког чвора имплицира да свака од класа еквиваленција његове деце такође мора бити изабрана.

(4) За свако $n \in N$, где је $E(n) = (c_1, ..., c_m)$ и c класа еквиваленције којој n припада (осим у случају где је $\lambda(n) = \theta$ и $i = 2$):

$B_n \Rightarrow B_{c \to c_i}$, за свако i.

Ово ограничење означава да избор неког чвора имплицира тачност променљиве која означава грану усмерену од класе тог чвора ка чвору класе детета тог чвора (осим у случају другог детета θ чвора).

(5) За сваке три различите класе $c_1, c_2, c_3 \in C$:

$B_{c_1 \to c_2} \wedge B_{c_2 \to c_3} \Rightarrow B_{c_1 \to c_3}$.

(6) За сваку класу $c \in C$, нека је $T_c = \{n \in c \mid \lambda(n) = \theta\}$:

$B_{c \to c} \Rightarrow \bigvee_{n \in T_c} B_n$.

Променљиве облика $B_{c_1 \to c_2}$ уведене су како би се избегли циклуси у графу који нису валидни и користе се да означе да класа c_1 достиже класу c_2 у графу кроз пут који не садржи друго дете неког θ чвора. Ограничење (5) представља правило транзитивности над променљивама облика $B_{c_1 \to c_2}$. Ограничење (6) представља услов да, ако нека класа c достиже саму себе путем који не садржи θ чворове, тада из класе c мора бити изабран неки θ чвор (у случају да класа c не садржи θ чворове, услов се своди на $B_{c \to c} \Rightarrow False$, што значи да у овом случају није дозвољено да класа достиже саму себе) и ово ограничење заједно са претходна два ограничења условљава да граф може садржати само циклусе који садрже друго дете неког θ чвора.

Циљна функција за псеудо-буловски проблем дефинише се као:

$$min \sum_{n \in N} B_n \cdot \mathbb{F}(n)$$

Глава 7

Имплементација оптимизатора

У овој глави биће изнети неки од најважнијих детаља имплементације оптимизатора заснованог на анализи еквиваленција за програмски језик Python и Numpy библиотеку. Кроз примере ће бити приказани основни кораци процеса оптимизације и постигнути резултати. Оптимизатор као улаз очекује фајл у коме су функције које желимо да оптимизујемо (тј. било које функције без бочних ефеката и које не позивају друге функције које имају бочне ефекте) обележене анотацијом `@opt`. Остале функције након рада оптимизатора остају непромењене.

7.1 Превођење Python кôда у ГПИ репрезентацију

Модул за превођење кôда у ГПИ репрезентацију имплементиран је по узору на алгоритам конверзије представљен у раду који обухвата техничке детаље процеса конверзије [2]. Чворови који учествују у ГПИ репрезентацији Python кôда представљени су одговарајућим класама. Чворове за представљање контроле тока чине класе `THETA`, `EVAL`, `PASS` и `PHI`, за сваку класу AST модула која представља неки израз Python програма постоји одговарајућа класа ГПИ чвора, а посебну класу чине и чворови који представљају параметре функција. Чвор који представља numpy вишедимензиони низ такође је представљен засебном класом. Базна класа свим поменутим класама је класа `PEGNode` (Program Expression Graph Node).

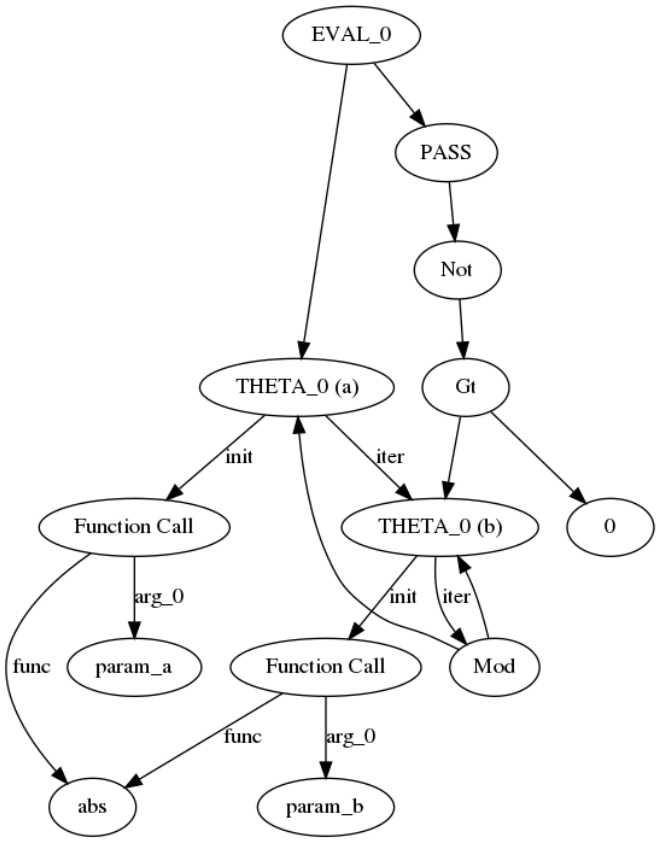
На слици 7.1 приказан је пример превођења имплементације Еуклидовог

алгоритма за израчунавање највећег заједничког делиоца. За визуализацију конструисаног графа коришћена је Python библиотека graphviz. Гране које повезују `THETA` чворове са њиховом децом означене су са: `init`, у случају детета које представља иницијалну вредност овог чвора и `iter`, у случају детета које представља итерацију. Функцијски позиви имају један или више аргумената. Први аргумент чвора који представља функцијски позив је функција која се позива. Грана која повезује чвор функцијског позива са чвором функције која се позива означена је са `func`. У примеру са слике, чвор који представља функцију која се позива је у оба случаја идентификатор, тј. назив функције. Чвор функцијског позива не мора увек бити идентификатор, на пример, он може бити и чвор који представља корен репрезентације ламбда функције или чвор који представља корен репрезентације атрибута (на пример, када би се у примеру који се разматра користила функција `numpy.abs()` уместо `abs()`). Гране које повезују чвор функцијског позива са његовом децом која представљају аргументе тог функцијског позива означене су са `arg_i`.

Једна специфичност везана за програмски језик Python су `for` петље, које су увек колекцијски базиране, тј. кораци итерације представљају пролазак кроз *итерабилни објекат*. Итерабилни објекат је сваки објекат који поседује имплементиран метод `__iter__()`, који враћа итератор на тај објекат. Итератор се креће кроз итерабилни објекат уз помоћ функције `next()`. Ова функција враћа вредност коју итератор реферише и помера итератор за један корак унапред. Пример ГПИ репрезентације функције која садржи `for` петљу која итерира над опсегом целобројних вредности од `0` до `n` приказан је на слици 7.2. Иницијална вредност `THETA` чвора који представља променљиву `i` је позив функције `next()` над итератором *range* објекта (што заправо представља прву вредност на коју реферише итератор), док се итерацијска вредност `THETA` чвора добија позивом функције `next()`. Услов прекидања петље је негирани услов припадности опсегу од `0` до `n` вредности коју променљива `i` садржи.

Напомена: приказана репрезентација није у потпуности семантички исправна, јер је иницијална вредност `THETA` чвора прва вредност на коју итератор реферише, док се у итерацији позива фукнција `next()`, чији аргумент мора бити итератор, а не вредност коју итератор реферише. Ова неправилност настала је као последица чињенице да функција `next()` производи бочне ефекте, међутим оваква репрезентација изабрана је зато што не доводи до било каквих двосмислености (јер функције којима се овај рад бави не позивају функције са бочним

```
def gcd(a, b):

    a = abs(a)
    b = abs(b)

    while b > 0:
        tmp = b
        b = a % b
        a = tmp

    return a
```

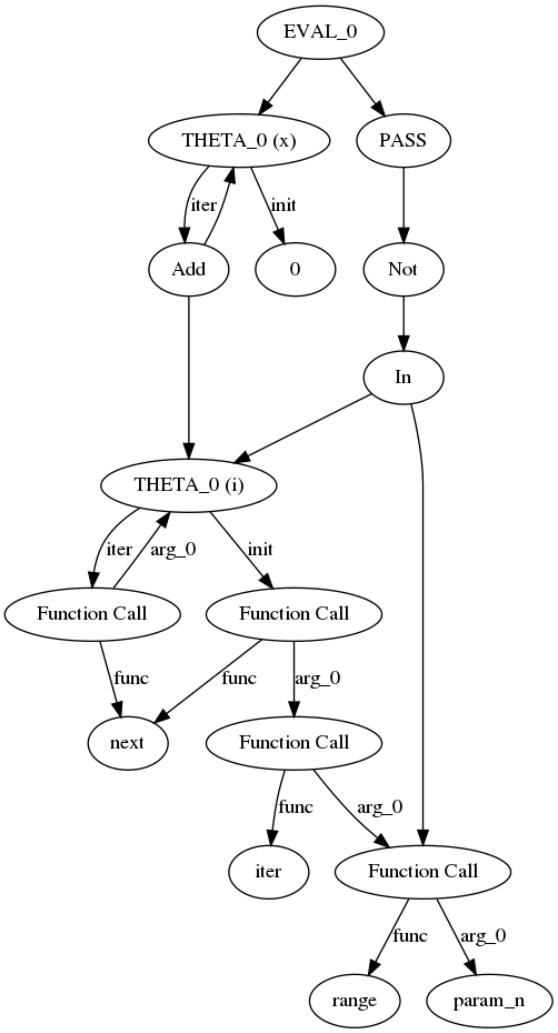


Слика 7.1: Пример кôда функције и његове ГПИ репрезентације. Функција abs() коришћена је због илустративности примера.

ефектима, па се функција `next()` може јавити искључиво у оквиру представљања `for` петље), а притом је погодна за коришћење у изградњи аксиома и анализи еквиваленција.

Већ у процесу изградње ГПИ репрезентације програмског кôда имплицитно се извршавају две оптимизационе технике. Прва је елиминација заједничких подизраза, јер ГПИ не садржи копије чворова. На пример, садржи по један чвор константе за сваку константу која се у програму користи, у случају да два функцијска позива имају једнаке аргументе и позивајуће функције, у ГПИ ће постојати само један чвор који представља оба функцијска позива, итд. Друга оптимизациона техника која се имплицитно спроводи је елиминација мртвог кôда, јер било који део кôда који не утиче на повратну вредност функције неће бити представљен у ГПИ, зато што је ГПИ коренски граф са кореном који представља повратну вредност и садржи само чворове који су достижни из

```
def f(n):

    x = 0

    for i in range(n):
        x += i

    return x
```



Слика 7.2: Пример репрезентације for петље

корена.

7.2 Извођење еквиваленција

Процес извођења еквиваленција спроводи се над Е-ГПИ репрезентацијом на основу задатих аксиома, тј. правила, а затим се у оквиру Е-ГПИ претражују шаблони који представљају леве стране правила. Овај процес реализован је грађењем CLIPS система за закључивање. Кораци реализације биће објашњени у наредне две секције. У првој секцији која следи биће приказано представљање Е-ГПИ чворова у оквиру CLIPS-а, док ће у наредној бити речи о аксиомама и формату у коме су представљене.

Дефинисање инстанци чворова за поклапање шаблона

Како би се започело са извођењем еквиваленција на основу задатих аксиома, неопходно је креирати Е-ГПИ од ГПИ репрезентације функције која се оптимизује. Овај поступак је једноставан - број класа еквиваленције чворова иницијалног Е-ГПИ је једнак броју чворова у ГПИ, тј. свака класа је једночлана.

Процес извођења еквиваленција извршава се уз помоћ алата CLIPS. У поглављу посвећеном овом алату речено је да он омогућава креирање инстанци реактивних класа. За потребе овог процеса, дефинисана је класа која представља ГПИ чвор.

```
(defclass node (is-a USER)
          (role concrete)
          (pattern-match reactive)
          (slot id (type INTEGER) (create-accessor read-write))
          (slot eq-class (type SYMBOL) (create-accessor read-write))
          (slot type (type SYMBOL))
          (multislot args)
          (multislot children (type SYMBOL) (create-accessor read-write) (default (create$ )))
          (multislot loop-variance-indices (type INTEGER) (default (create$ -1)))
          (slot loop-depth (type INTEGER) (default -1))
          (slot expr-type (type SYMBOL) (default unknown))
          (multislot shape (type INTEGER) (default (create$ -1)))
)
```

Слика 7.3: CLIPS реактивна класа ГПИ чвора

Атрибути које класа CLIPS чвора садржи су јединствени идентификатор чвора, класа еквиваленције којој чвор припада, тип чвора, аргументи који учествују у конструктору чвора (на пример, чвор који представља константу један има тип NumericNode, а аргумент 1), затим атрибут за чувење информације о деци чвора. Атрибут `loop-variance-indices` садржи информацију о томе у оквиру којих петљи чвор варира. Атрибут `expr-type` представља тип чвора као израза (на пример чвор који представља збир два чвора нумеричких константи има тип нумеричке константе). Потреба за овим атрибутом може се видети на следећем примеру - комутативност оператора `+` у програмском језику Python важи ако су његови аргументи нумеричке вредности, међутим, ако су његови аргументи две листе, онда овај оператор представља конкатенацију, која није комутативна операција, па у аксиомама морамо видити рачуна и о типу операнада. Да не бисмо морали да се ограничимо само на тип чвора у изградњи аксиома, уведен је атрибут типа израза који чвор представља, па

се на овај начин омогућава да се, нпр. аксиома комутативности у овом случају примени и на збир збирова нумеричких чворова, а не само директно на нумеричке чворове.

Атрибут `loop-depth` представља дубину угнежђења коју поседују само `THETA`, `EVAL` и `PASS` чворови, док атрибут `shape` представља вредност атрибута `shape` чвора који поседују само чворови чији је тип numpy вишедимензиони низ, у случају да је ова вредност позната у фази превођења. У осталим случајевима, ове вредности се подразумевано постављају на неважеће. У поглављу посвећеном алату CLIPS поменуто је да он омогућава и механизам изградње хијерархије класа, међутим, у овом случају су све класе ГПИ чворова представљене једном CLIPS реактивном класом чвора. Оваква одлука је донета због једноставније аутоматизације процеса превођења Python инстанци класа чворова у CLIPS инстанце, и обратног смера, у случају када се у току извођења еквиваленција креирају нове инстанце у оквиру CLIPS-а.

Параметризација аксиома

Основу приступа оптимизацији заснованог на анализи еквиваленција чини скуп задатих аксиома уз помоћ којих се еквиваленције изводе. Свака аксиома састоји се из леве и десне стране, где лева страна представља шаблон подграфа који треба пронаћи, а десна страна подграф семантички еквивалентан подграфу леве стране. Како би се креирао што бољи систем за закључивање еквиваленција, важно је да скуп аксиома буде богат. Креирање скупа аксиома и његово надограђивање олакшано је избором формата у коме аксиоме могу да се задају.

На слици 7.4 приказана је једна једноставна аксиома. Формат аксиоме састоји се из назива аксиоме (у овом примеру `'multiply-by-zero'`) који се додељује променљивој `axiom_name`, дефиниција везаних променљивих (чији називи треба да почињу доњом цртом) и њихових типова (у овом примеру фигурише само једна везана променљива, чији је тип `PEGNode`, што означава да чвор који одговара овој променљивој може бити било ког типа), леве и десне стране аксиоме које се додељују редом променљивама `lhs` и `rhs`. Једна од додатних опција која је омогућена приликом креирања аксиома је и постављање додатних ограничења везаних променљивих. У примеру са слике 7.4, постављено је ограничење да тип чвора као израза мора да буде нумеричка вредност, јер је, на пример, множење нулом numpy вишедимензионог на основу broadcasting

```
axiom_name = 'multiply-by-zero'

_a : PEGNode
_a.expr_type = 'NumNode'

lhs = _a * 0
rhs = 0
```

Слика 7.4: Аксиома која представља правило множења нумеричке вредности нулом

```
axiom_name = 'branch-with-true-condition'

_phi : PHINode
_t : PEGNode
_f : PEGNode

lhs = _phi(True, _t, _f)
rhs = _t
```

Слика 7.5: Аксиома која представља која се ослања на семантику PHI чвора

правила заправо покоординатно множење низа скаларом 0, док множење листе нулом даје празну листу као резултат.

На слици 7.5 приказана је аксиома у којој учествује и један чвор контроле тока - `PHI` чвор, који служи за представљање гранања. Леву страну ове аксиоме чини `PHI` чвор чији је услов `True`, а десну чвор који представља вредност овог израза у случају да је услов тачан.

Још један пример приказан на слици 7.6 представља аксиому која се односи на вредност трага матрице у специјалном случају димензија 2×2. У овом примеру имамо додато ограничење да аксиома може да се примени само у случају у коме је могуће статички одредити да матрица заиста има одговарајуће димензије. Без овог ограничења, аксиома не би била исправна, јер би у том случају везане променљиве `_aij` могле да буду, на пример, и листе, па се траг у том случају не би израчунавао на овај начин. Такође, ако бисмо уместо овог ограничења поставили ограничења да сваки од `_aij` чворова мора да буде нумеричка вредност, тај услов би био превише строг - у случају да, на пример, једна од `_aij` вредности представња параметар функције, у фази превођења њен тип је непознат, али ако остале променљиве представљају нумеричке изразе, онда

```
axiom_name = 'matrix-2x2-trace'

_arr : NPArrayNode
_a11 : PEGNode
_a12 :PEGNode
_a21 : PEGNode
_a22 : PEGNode

_arr.shape = (2, 2)

matrix = _arr([[_a11, _a12],
               [_a21, _a22]])


lhs = np.trace(matrix)
rhs = _a11 + _a22
```

Слика 7.6: Аксиома која представља правило израчунавања трага матрице димензије 2×2

знамо да и вредност представљена параметром мора да буде нумерички израз, јер је numpy вишедимензиони низ хомоген.

У приказаним примерима може се приметити да је синтакса формата за дефинисање аксиома веома блиска синтакси језика Python. За обележавање типова чворова користе се анотације које су део синтаксе Python верзије 3, а такође, изрази леве и десне стране аксиоме представљени су Python изразима, осим у случајевима коришћења контроле тока, у којима чворови контроле тока представљају као променљиве којима је додељен одговарајући тип и које чији аргументи представљају њихову децу чворове. Овакав формат аксиома је погодан, јер за сваки израз језика Python постоји одговарајућа класа ГПИ чвора, па се лева и десна страна аксиоме преводе у графовске репрезентације на сличан начин на који се и кôд функције преводи у ГПИ репрезентацију, али у овом случају имамо два корена - корене израза леве и десне стране, и ови графови могу имати и заједничке чворове, као што је на пример чвор `_t` у аксиоми са слике 7.5. Осим тога, овакав формат аксиома олакшава аутоматско креирање аксиома увлачења дефиниција кориснички дефинисаних функција. За овај процес коришћен је модул AST, који дефиницију функције преводи у приказани формат аксиоме. Овај поступак неће бити детаљније представљен. Када се аксиоме креирају и преведу у графовске репрезентације, графоске репрезентације се даље преводе у CLIPS правила.

7.3 Кодирање оптимизационог проблема у псеудо-буловску форму

Након креирања CLIPS инстанци чворова ГПИ репрезентације и дефинисања CLIPS правила, систем за закључивање је спреман за извођење еквиваленција. Један пример једноставног програма и закључених еквиваленција приказан је на слици 7.7. Међусобно еквивалентне класе уоквирене су зеленим правоугаоницима.

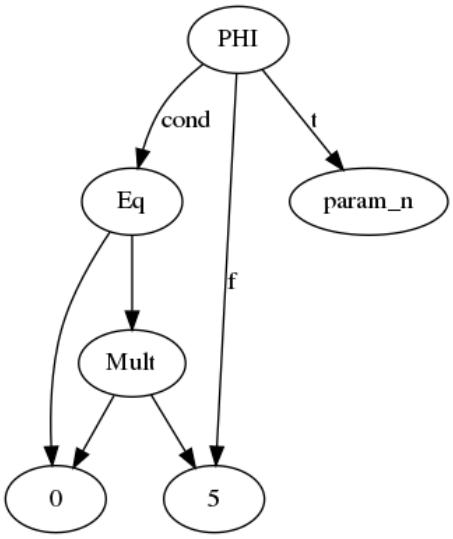
Прва изведена еквиваленција је еквиваленција чворова `0` и `Mult`, на основу правила множења нумеричке вредности нулом. Ова еквиваленција омогућила је да се даље закључи да је чвор `Eq` еквивалентан чвору `True` (у овом случају се чвор `True` додаје у граф, јер у иницијалном графу није постојао). Претходно изведена еквиваленција затим омогућава активацију правила са слике 7.5.

Корак који следи након процеса извођења еквиваленција је избор оптималне верзије програма. За ову сврху, коришћен је модул pysat, API који омогућава приступ за рад са неколико популарних SAT решавача (Minisat, Glucose, Lingeling, Maplesat, итд.). Формула се решавачу задаје као листа клауза, где су клаузе листе литерала, а литерали се представљају евентуално негираним целобројним вредностима. Овај модул такође омогућава и превођење задатих псеудо-буловских ограничења у формуле у КНФ форми, што је и коришћено у овом раду.

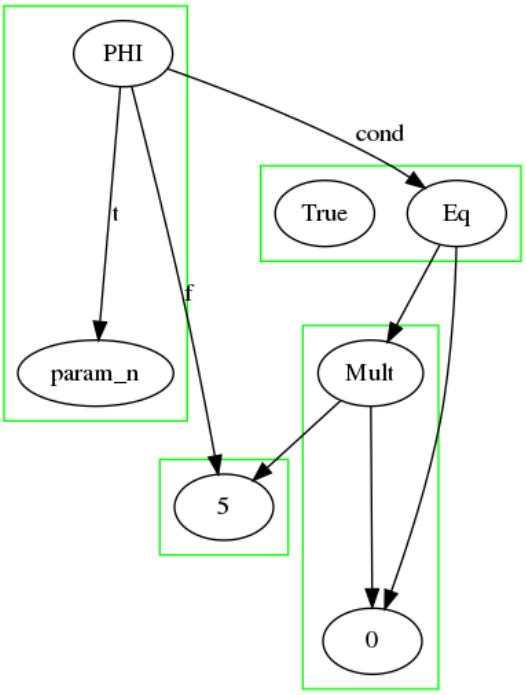
У претходном тексту представљен је оптимизациони проблем избора оптималне ГПИ репрезентације. Овај проблем је решен уз помоћ Minisat решавача. Сваком чвору додељена је променљива која представља његов јединствени идентификатор и дефинисане су функције које креирају и остале променљиве које су описане у оптимизационом проблему и пресликавају их у целобројне вредности. На основу ограничења представљених у поглављу 6.4, можемо се уверити да полазни ГПИ задовољава сва задата ограничења. Међутим, у овом случају, за било који избор (позитивних) тежина чворова, оптимални ГПИ је једночлани граф, који садржи само параметарски чвор `n`. Овај граф је подграф полазног ГПИ и задовољава сва ограничења. На пример, избором параметарског чвора `n` испуњен је услов да мора бити изабран неки од чворова класе повратне вредности функције (у овом случају класа чвора `PHI`), затим услов да, када се неки чвор изабере, мора се изабрати и по један чвор из класа еквиваленције сваког од његове деце чворова (избором параметарског чвора `n`

```
def f(n):

    x = 5
    y = x * 0

    if (y == 0):
        x = n

    return x
```

а)



б)



в)

Слика 7.7: а) Кôд функције, б) ГПИ репрезентација и в) Е-ГПИ репрезентација након извођења еквиваленција

овај услов је аутоматски испуњен, јер он нема деце), итд.

7.4 Додела тежина чворовима

За приступ оптимизацији представљен у овом раду, поред богатог скупа аксиома, кључан корак је и добра процена тежина чворова. Аутори приступа оптимизацији заснованог на анализи еквиваленција су чворовима доделили тежине статички, према типу чвора, тј. операције коју он представља. Тежине чворова који се налазе у оквиру неке петље се израчунавају степеновањем оригиналне тежине чвора дубином угнежђења петље у којој се тај чвор налази. Мана оваквог приступа је што не узима у обзир број итерација петље, који, наравно, у фази превођења не мора ни да буде познат. Статичка додела тежина чворова посебно није погодна за програмски језик Python, који је динамички типизиран, па у фази превођења често не можемо да знамо ни да ли, на пример, збир два параметарска чвора представља операцију сабирања две нумеричке вредности или можда два numpy вишедимензиона низа. Разлика у брзини извршавања поменуте две операције, наравно, може да буде јако велика.

Оптимизатор имплементиран уз овај рад додељује тежине чворовима динамички, уз помоћ линијског профајлера. Овај профајлер покреће програм и исписује информације о томе колико времена је програм у извршавању провео у свакој линији кôда. Да би ове информације могле да се искористе, функције обележене анотацијом `@opt` се пре покретања профајлера преводе у форму троадресног кôда. Процес превођења у форму троадресног кôда реализован је уз помоћ класе `NodeTransformer` модула AST. Затим се при изградњи ГПИ репрезентације чворовима додељују тежине на основу дужине извршавања линије у којој се налази операција која том чвору одговара. У случају линија које представљају заглавље петље, дужина њеног извршавања додељује се одговарајућем `PASS` чвору, док чворови константи и параметара функција увек имају тежину 1.

Пошто су у процесу извођења еквиваленција могући и случајеви у којима се инстанцирају нови чворови, тежина ових чворова је непозната. Како би се допуниле недостајуће тежине чворова, за аксиоме које инстанцирају нове чворове креирани су модели машинског учења за предвиђање тежина нових чворова. За сваку аксиому, тј. правило активирано у процесу извођења еквиваленција које инстанцира нове чворове, позива се модел тог правила коме се као улаз

задају тежине чворова леве стране тог правила, на основу којих модел предвиђа тежине нових чворова десне стране тог правила. Пример аксиоме која инстанцира нове чворове је аксиома која представља правило за рачунање детерминанте инверза матрице.

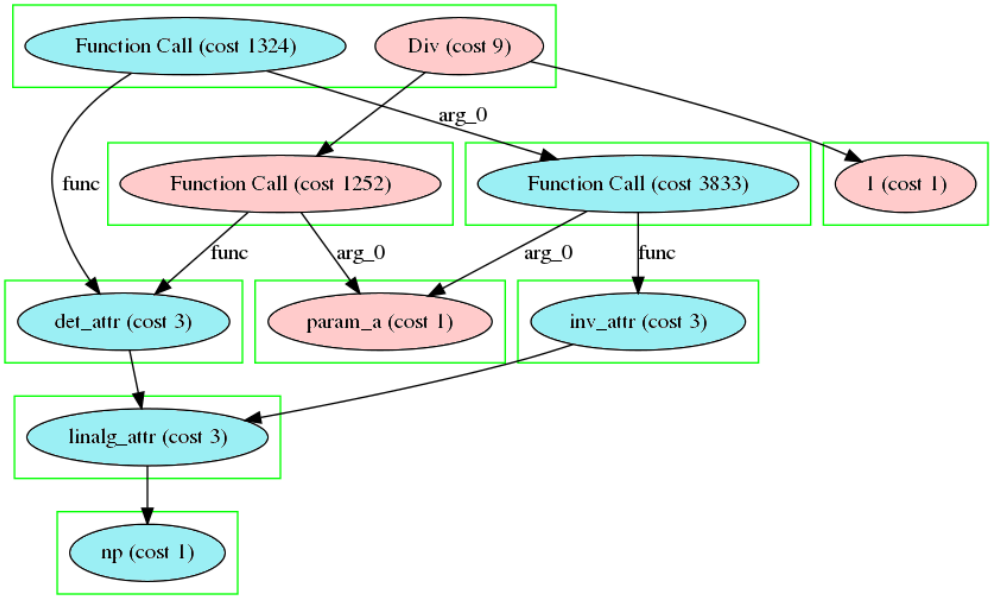
```
axiom_name = 'determinant-of-inverse'

_a : PEGNode

lhs = np.linalg.det(np.linalg.inv(_a))
rhs = 1 / np.linalg.det(_a)
```

Слика 7.8: Правило за рачунање детерминанте инверза матрице

На слици 7.9 приказан је пример функције која израчунава само детерминанту инверза задате матрице и Е-ГПИ репрезентација ове функције након покретања система за извођење еквиваленција. Једино активирано правило је управо правило са слике 7.8. Плавом бојом обојени су чворови који припадају иницијалном ГПИ функције `f()`, а црвени чворови су чворови инстанцирани на основу поменутог правила. Плавим чворовима тежине су додељене на основу информација које је линијски профајлер извео покретањем програма који је у овом случају позвао функцију `f()` над матрицом димензија 100×100. Тежине црвених чворова проценио је модел аксиоме на основу које су инстанцирани који је као улаз у овом случају имао све плаве чворове са слике.

Како би се једноставније креирали модели, у оквиру оптимизатора постоји и скрипт који је потребно модификовати у складу са одговарајућом аксиомом чији модел за предвиђање тежина чворова желимо да креирамо. Овај скрипт омогућава делимично аутоматизован процес креирања скупа података и тренирање модела, али више детаља о томе овде неће бити представљено. Модел из примера са слике трениран је над скупом матрица разних димензија и може се приметити да је тежина чвора који представља функцијски позив рачунања детерминанте матрице `a` релативно блиска тежини чвора који представља функцијски позив рачунања детерминанте матрице инверза ове матрице, што се може и очекивати, јер су ове две матрице истих димензија. Наравно, димензија матрице у овом случају није једини фактор који утиче на брзину извршавања функција које учествују у левој и десној страни ове аксиоме, али модели иницијалне верзије овог оптимизатора су једноставни и побољшање њиховог квалитета остављено је за даљи рад. Основна идеја оваквог приступа је по-

```
@opt
def f(a):

    inv_a = np.linalg.inv(a)

    return np.linalg.det(inv_a)


def main():

    a = np.random.randn(100, 100) * 5

    print(f(a))
```



Слика 7.9: Пример доделе тежина новим чворова у зависности од тежина чворова на основу којих су инстанцирани.

кушај креирања модела који апроксимирају временску сложеност десне стране на основу временске сложености леве стране аксиоме. Овакав приступ би посебно могао да буде користан у ситуацијама када се, на пример, за мање улазе брже извршава лева страна аксиоме, а за веће улазе десна, на основу чега би оптимизатор могао да одреди коју од ове две стране је адекватније применити у случају конкретног програма који се извршава.

7.5 Превођење ГПИ репрезентације у Python кôд

Након избора оптималне ГПИ репрезентације, ова репрезентација се преводи у Python кôд. Овај процес имплементиран је по узору на алгоритам представљен у секцији посвећеној конверзијама, али је надограђен и детаљима представљеним у раду са техничким детаљима процеса конверзије [2] који омогућавају избегавање сувишних делова кôда до којих оригинални алгоритам може да доведе. Осим тога, надограђени алгоритам обухвата и сажимање петљи и гранања са истим условом у процесу превођења, како би генерисао ефикаснији кôд. Над резултујућим кôдом имплементирана је и пропагација променљивих, уз помоћ модула staticfg, који креира граф контроле тока задатог изворног кôда. Примери генерисаног кôда биће представљени у наредном одељку.

7.6 Примери оптимизованих програма и резултати

Примери коришћени у претходном тексту углавном су били једноставни како би што јасније илустровали кораке који се извршавају у процесу оптимизације. У овом одељку биће представљени мало комплекснији примери и постигнути резултати.

На слици 7.10 приказан је један фајл у коме је дефинисана функција `f()` у којој постоји простор за оптимизацију. Променљивој `z` се у спољашњој петљи додељује вредност која не зависи од итерације петље. У унутрашњој петљи, променљивој `x` се у свакој итерацији додаје следећа вредност из опсега од `0` до `n`. Кôд који је генерисао оптимизатор приказан је на слици 7.11. Читава унутрашња петља је елиминисана, тј. замењена је условним изразом, који текућој вредности променљиве `x` (тј. `x_2` у оптимизованој верзији) додаје формулу за збир првих `n-1` природних бројева, у случају да је `n` веће од нуле, а иначе задржава почетну вредност. Може се приметити да је оптимизатор увео и додатне променљиве, које би у случају бољег распореда наредби могле да буду избегнуте, али тренутна имплементација процеса превођења ГПИ репрезентације у кôд даје овакав резултат. На сличан начин, променљива `z` која је у свакој од `m` итерација увећавана за вредност која не зависи од итерације, замењена је

условним изразом који на њену иницијалну вредност додаје `m * np.cos(m)` у случају да је `m` веће од `0`, тј. у случају да је извршена бар једна итерација петље оригиналног програма.

```python
# file.py

@opt
def f(n, m):

    x = 0
    y = 10
    z = 100

    for i in range(m):
        z += np.cos(m)
        y += x + 1

        for j in range(n):
            x += j

    return y + z


def main():

    print(f(100, 200))
```

Слика 7.10: Пример Python фајла задатог оптимизатору

```python
# output.py

def f(n, m):
    y_1 = 10
    x_2 = 0
    for i_3 in range(m):
        if_exp_18 = x_2 + (n - 1) * n // 2 if n > 0 else x_2
        op_20 = y_1 + 1 + x_2
        x_2 = if_exp_18
        y_1 = op_20
    return y_1 + (100 + m * np.cos(m) if m > 0 else 100)

def main():
    print(f(100, 200))
```

Слика 7.11: Оптимизована верзија задатог фајла

Аксиоме на основу којих је претходна оптимизација извршена представљене су на сликама 7.12 и 7.13.

Резултати постигнути у овом примеру приказани су на слици **??**. Прво је извршена провера да оба програма имају исти излаз, а затим су уз помоћ мо-

```
axiom_name = 'first_n_natural_numbers_addition'

_init : PEGNode
_up : PEGNode
_th1 : THETANode
_th2 : THETANode
_eval : EvalNode
_pass : PassNode

range_obj = range(_up)
iter_value = _th1(next(iter(range_obj)), next(_th1))

lhs = _eval(_th2(_init, _th2 + iter_value), _pass(not iter_value in range_obj))
rhs = _init + ((_up - 1) * _up) // 2 if _up > 0 else _init
```

Слика 7.12: Аксиома која замењује вредност којој се у петљи додају вредности из опсега од 0 до _up одговарајућом формулом

```
axiom_name = 'constant-step-loop-addition'

_init : PEGNode
_up : PEGNode
_th1 : THETANode
_th2 : THETANode
_eval : EvalNode
_pass : PassNode
_val : PEGNode

_init.expr_type = 'NumNode'

range_obj = range(_up)
iter_value = _th1(next(iter(range_obj)), next(_th1))

lhs = _eval(_th2(_init, _th1 + _val), _pass(not iter_value in range_obj))
rhs = _init + _up * _val if _up > 0 else _init

condition_1 = invariant(_val, _eval)
```

Слика 7.13: Аксиома која измешта променљиву која не варира у петљи изван петље. Ова аксиома има и условни израз додељен променљивој condition_1 који означава предикат да је вредност _val независна од дубине угнежђења _eval чвора, тј. да не варира у петљи одређеној тим чвором.

дула timeit измерена и времена извршавања оригиналне и оптимизоване верзије програма.

Још један пример фајла прослеђеног оптимизатору дат је на слици 7.15, а његова оптимизована верзија на слици 7.16. У овом примеру, функција `transform()` је остала непромењена, док је у функцији `compute()` примењено неколико аксиома, међу којима и аксиома увлачења дефиниције функције `transform()`. Из увучене дефиниције је елиминисан условни израз, а примењена је и акси-

```
In [1]: import file, output

In [2]: file.main()
Out[2]: 98505407.437535

In [3]: output.main()
Out[3]: 98505407.437535

In [4]: import timeit

In [5]: %timeit file.main()
100 loops, best of 3: 4.31 ms per loop

In [6]: %timeit output.main()
10000 loops, best of 3: 131 µs per loop
```

Слика 7.14: Резултати

ома множења вектора матрицом у специјалном тродимензионом случају. Ова аксиома приказна је на слици 7.17. Такође, може се приметити и да су синус и косинус задатог угла израчунати и смештени у промењиве, како би се избегло поновно израчунавање, јер се ове вредности у наставку кôда користе још по два пута.

```
# file.py

#@opt
def transform(x, angle, translation_vector):

    rotation_matrix = np.array([[0, 0, 1],
                                [np.cos(angle), -np.sin(angle),  0],
                                [np.sin(angle), np.cos(angle),  0]])

    result = np.dot(rotation_matrix, x)

    if not (translation_vector is None):
        result += translation_vector

    return result


#@opt
def compute(angle):

    vec = np.array([1, 3, 5])
    transformed = transform(vec, angle, None)

    return transformed


def main():

    return compute(10.3)
```

Слика 7.15: Пример Python фајла задатог оптимизатору

Постигнути резултати за овај пример приказани су на слици 7.18.

```python
# output.py

def transform(x, angle, translation_vector):
    rotation_matrix = np.array([[0, 0, 1], [np.cos(angle), -np.sin(angle),
        0], [np.sin(angle), np.cos(angle), 0]])
    result = np.dot(rotation_matrix, x)
    if not translation_vector is None:
        result += translation_vector
    return result


def compute(angle):
    func_call_2 = np.sin(angle)
    func_call_7 = np.cos(angle)
    return np.array([5, -func_call_2 * 3 + func_call_7, func_call_7 * 3 +
        func_call_2])


def main():
    return compute(10.3)
```

Слика 7.16: Оптимизована верзија задатог фајла

```
axiom_name = 'transform-vector-by-matrix-3d'

_mat : NPArrayNode
_a11 : PEGNode
_a12 : PEGNode
_a13 : PEGNode
_a21 : PEGNode
_a22 : PEGNode
_a23 : PEGNode
_a31 : PEGNode
_a32 : PEGNode
_a33 : PEGNode

_vec :NPArrayNode
_v1 : PEGNode
_v2 : PEGNode
_v3 : PEGNode

_mat.shape = (3, 3)
_vec.shape = (3,)

matrix = _mat([[_a11, _a12, _a13],
               [_a21, _a22, _a23],
               [_a31, _a32, _a33]])

vector = _vec([_v1, _v2, _v3])

lhs = np.dot(matrix, vector)
rhs = np.array([_a11 * _v1 + _a12 * _v2 + _a13 * _v3,
                _a21 * _v1 + _a22 * _v2 + _a23 * _v3,
                _a31 * _v1 + _a32 * _v2 + _a33 * _v3])
```

Слика 7.17: Правило множења вектора матрицом димензије 3×3

```
In [1]: import file, output

In [2]: file.main()
Out[2]: array([ 5.        ,  1.66223101, -2.69016506])

In [3]: output.main()
Out[3]: array([ 5.        ,  1.66223101, -2.69016506])

In [4]: import timeit

In [5]: %timeit file.main()
10000 loops, best of 3: 53 µs per loop

In [6]: %timeit output.main()
10000 loops, best of 3: 25 µs per loop
```

Слика 7.18: Времена извршавања оригиналног и оптимизованог фајла

Библиографија

[1] Michael Stepp. Equality saturation: Engineering challenges and applications. New York, NY, USA.

[2] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Translating between pegs and cfgs. New York, NY, USA.

[3] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM.