VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

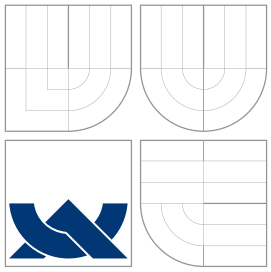
OPTIMALIZACE V ZADNÍ ČÁSTI ZPĚTNÉHO PŘEKLADAČE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

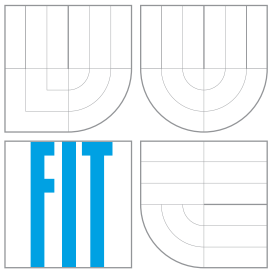
AUTOR PRÁCE
AUTHOR

JAROSLAV KOLLÁR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE V ZADNÍ ČÁSTI ZPĚTNÉHO PŘEKLADAČE

OPTIMIZATIONS IN THE DECOMPILER'S BACK-END

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAROSLAV KOLLÁR
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR ZEMEK
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá návrhem a tvorbou optimalizací v zadní části zpětného překladače. Úlohou těchto optimalizací je vylepšit čitelnost produkovaného zdrojového kódu. V úvodu jsou poskytnuty základní informace o reverzním inženýrství a zpětných překladačích, které slouží pro účel uvedení do dané problematiky. Poté následuje analýza produkovaného kódu zpětným překladačem s cílem obeznámit čtenáře s navrženými optimalizacemi pomocí názorných ukázek. Dále následuje hlavní část této práce, která se věnuje popisu návrhu a implementace jednotlivých navržených optimalizací. Poté následuje popis technik, které byly využity při testování. V závěru práce jsou shrnuty dosažené výsledky a jejich přínos.

Abstract

This bachelor's thesis describes the design and implementation of optimizations in the retargetable decompiler's back-end. The purpose of these optimizations is to improve readability of the produced source code. In the introduction, basic information about reverse engineering and decompilation is provided. Then, there is an analysis of the source code produced by the decompiler to familiarize the reader with the proposed optimizations. After that, there is the main part of this work, which describes the design and implementation of the proposed optimizations. It is followed by mentioning the techniques that were used for testing. The present work is concluded by a summary of the achieved results and their benefits.

Klíčová slova

Reverzní inženýrství, zpětný překlad, zadní část zpětného překladače, optimalizace, čitelnost zdrojového kódu.

Keywords

Reverse engineering, decompilation, decompiler's back-end, optimization, source code readability.

Citace

Jaroslav Kollár: Optimalizace v zadní části zpětného překladače, bakalářská práce, Brno, FIT VUT v Brně, 2013

Optimalizace v zadní části zpětného překladače

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, pouze za odborného vedení pana Ing. Petra Zemka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal. Všechny citované obrázky jsem převzal se souhlasem autora.

.....
Jaroslav Kollár
30. dubna 2013

Poděkování

Děkuji svému vedoucímu Ing. Petru Zemkovi za odborné vedení, za poskytnuté rady a za čas, který mi při tvorbě práce věnoval.

Obsah

Kapitola 1

Úvod

Dôležitým pojmom v rámci informačných technológií je v dnešnej dobe bezpečnosť. Medzi jedno z najväčších rizík je možné zaradiť rôzne druhý škodlivého softvéru. Typickými príkladmi sú klasické počítačové vírusy, trójske kone, internetové červy a iné. Všeobecné označenie takýchto programov je možne pomocou pojmu malvér. Typické úlohy škodlivého softvéru je najčastejšie získavanie citlivých dát od užívateľov, poškodzovanie súborov, znepríjemňovanie práce na počítači a podobné [23]. Príkladom možnej obrany je nespúšťanie a otváranie nedôveryhodných materiálov. Ďalšou osvedčenou technikou je využívanie antivírusových programov. Antivírusové programy prehľadávajú súbory, sledujú správanie sa aplikácií a aktivitu systému na pozadí. Pri týchto činnostiach súčasne porovnávajú či napríklad chovanie niektorej aplikácie neodpovedá správaniu sa napadnutého programu. V súboroch zase hľadajú sekvenciu inštrukcií, ktorá odpovedá nejakému malvéru [9]. Jednou z techník, ktorú môžu použiť tvorcovia aplikácii proti malvéru, je softvérové reverzné inžinierstvo. Ako príklad je možne uviesť využitie spätného prekladu nad získaným binárnym súborom. O spätný preklad sa postará spätný prekladač, pričom jeho úlohou je z tohto súboru vygenerovať program vo vysokoúrovňovom jazyku. Následne je možné tento kód preskúmať a vyhodnotiť jeho činnosť [29]. Táto technika je výhodná v tom, že daný súbor nie je potrebné ani spustiť.

V tejto práci je možné nájsť základné informácie o reverznom inžinierstve a jeho použití v rámci informačných technológii, ako aj informácie o spätnom preklade. Hlavná časť práce je venovaná optimalizáciám v zadnej časti spätného prekladača vyvíjaného v rámci projektu Lissom. Cieľom práce bolo navrhnúť a vytvoriť optimalizácie, pričom hlavnou motiváciou tejto činnosti bolo zlepšiť čitateľnosť produkovaného výsledného kódu. Ako prvé bolo však potrebné vykonať analýzu produkovaného kódu. Následne na základe získaných poznatkov a postrehov bolo možné navrhnúť jednotlivé optimalizácie. Po dokončení návrhu nasledovala implementácia. Táto implementácia bola priebežne testovaná jednotkovými testami (angl. unit tests) a referenčnými testami.

Kapitola 2 obsahuje základné informácie o reverznom inžinierstve a spätných prekladačoch. Popis spätného prekladača vyvíjaného v rámci projektu Lissom sa nachádza v kapitole 3. Analýze produkovaného kódu pred vytvorenými optimalizáciami v tejto práci sa venuje kapitola 4. Návrhu a popisu jednotlivých optimalizácií sa venuje kapitola 5. Kapitola 6 popisuje implementáciu navrhnutých optimalizácií. Popis techník, ktoré boli využité pri testovaní optimalizácií, sa nachádza v kapitole 7. Záverečná kapitola 8 obsahuje zhodnotenie dosiahnutých výsledkov a diskutovanie o budúcom možnom vývoji.

Kapitola 2

Reverzné inžinierstvo a spätný preklad

Táto kapitola obsahuje základne informácie o reverznom inžinierstve. Okrem toho je možné dočítať sa o princípoch spätného prekladu, pretože sa jedná o jednu z techník, ktoré reverzné inžinierstvo využíva. Prvá podkapitola sa venuje softvérovému reverznému inžinierstvu a jeho použitiu v praxi. Druhá podkapitola obsahuje popis a štruktúru spätného prekladača. V tejto podkapitole sú uvedené aj výhody využitia spätného prekladača oproti disassembleru. Cieľom tejto kapitoly je uviesť čitateľa do danej problematiky.

Pod pojmom reverzné inžinierstvo chápeme proces, ktorého úlohou je získať chýbajúce znalosti o skúmanom predmete. Najčastejšie sa jedná o znalosti, ktoré nie sú voľne šírené. V niektorých prípadoch sa môže jednať o informácie, ktoré boli zničené alebo stratené. Tento pojem sa datuje už od dôb priemyslovej revolúcie a nie je striktne viazaný na modernú dobu. S modernou dobou však prichádza softvérové reverzné inžinierstvo. Dôležitá je aj otázka právnej legálnosti. V niektorých štátoch sú určité aktivity spadajúce pod reverzné inžinierstvo považované za nezákonné [7].

2.1 Softvérové reverzné inžinierstvo a jeho použitie

Softvérové reverzné inžinierstvo (SRE) a jeho použitie môžeme rozdeliť na dva prípady. V prvom prípade je v praxi používané ako analýza určitého softvérového systému s cieľom získať implementačné detaily prípadne jeho návrh. Môže byť aplikovaná buď iba na jeho určitú časť alebo na celý systém [2].

V druhom prípade sa jedná o prevod informácie z nízkoúrovňového formátu (napr. aplikačného binárneho kódu) na čitateľnejšiu formu na určitej úrovni abstrakcie. Typickými príkladmi nástrojov využívané pri SRE je spätný prekladač a disassembler[1] [1]. Spätný preklad je reverzný proces bežného prekladu, pričom jeho cieľom je vytvorenie vysokoúrovňového kódu z binárneho kódu [5].

Dnešné použitie týchto techník sa najčastejšie využíva na detekciu malvéru, prípadne vyhľadávanie bezpečnostných chýb. Niektoré organizácie ho však zneužívajú aj na vylepšovanie

[1] Prevádza strojový kód do symbolického zápisu v assembleru [20].

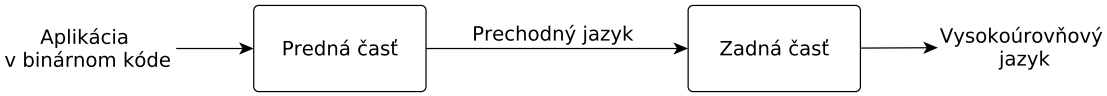
vlastných technológií analyzovaním technológií vyvinutých konkurenciou. Ku skomplikovaniu využitia SRE je možné použiť tzv. obfuskátor[2] kódu [22]. Ďalšou možnosťou je využitie tzv. protektorov. Tieto protektory napríklad na výsledný binárny kód použijú kompresie alebo ho pomocou šifrovacích techník zašifrujú a vložia kód na jeho rozšifrovanie [19].

2.2 Spätný prekladač

Cieľom spätného prekladača je preklad binárneho kódu závislého na určitej platforme do zdrojového kódu vysokoúrovňového jazyka. S týmto procesom je však spojený jeden významný problém. Kód v binárnej podobe obvykle neobsahuje explicitné informácie akými sú napríklad definície dátových štruktúr, názvy premenných, prípadne informácie o iných dátových typoch. Často sa vedú polemiky o tom či je vôbec možné zostrojiť kvalitný spätný prekladač [7]. Vytvorenie zdrojového kódu vysokoúrovňového jazyka, ktorý identicky vykonáva funkcie pôvodného programu, je pre niektoré jazyky možné [5]. Tento kód sa však vo väčšine prípadoch nepodobá na originálny zdrojový kód a býva veľmi ťažko čitateľný. Vo výslednom kóde chýbajú napríklad komentáre, názvy pôvodných premenných, deklarácie dátových štruktúr. Takisto jedným z dôvodov, prečo výsledný kód po spätnom preklade neodpovedá štruktúre pôvodného kódu, sú optimalizácie vykonávané prekladačom [7].

Jedna z možných štruktúr spätného prekladača je uvedená na obrázku 2.1. Táto štruktúra je veľmi podobná prekladaču. Jednotlivé časti však pracujú v opačnom poradí. Úlohou prednej časti je dekódovať inštrukcie v nízkoúrovňovom jazyku a ich prevod do prechodnej reprezentácie. Prechodná reprezentácia predstavuje vyššiu formu abstrakcie, než pôvodný nízkoúrovňový jazyk. Jednou z jej hlavných výhod je, že obsahuje menší počet inštrukcií, než pôvodný jazyk závislý na architektúre. Nezávislosť tejto reprezentácie na nejakej konkrétnej architektúre je možno chápať ako ďalšiu veľkú výhodu. Kód generovaný v prednej časti môžeme prezentovať pomocou tzv. grafu toku riadenia. Základným prvkom tohto grafu je blok. Na každý blok sa dá odkázať a je ukončený skokovou inštrukciou. Tieto neštruktúrované grafy sú neskôr využívane v analýze tokov riadenia, ktorá z nich vytvára štruktúrované grafy konštrukcií typických pre vysokoúrovňové jazyky. Zadná časť následne z prechodnej reprezentácie vygeneruje zvolený vysokoúrovňový jazyk [7].

Spätný prekladač je možné považovať za ďalší vývojový stupeň po disassembleru. Vstupom disassembleru je spustiteľný binárny súbor a výstupom je kód v jazyku assembler. Úlohou spätného prekladača je vytvoriť kód na vyššej úrovni abstrakcie než assembler. Kód na vyššej úrovni abstrakcie je jednoduchšie čitateľný a analyzovateľný. Hlavnou myšlienkou spätného prekladu je získať čo najpodobnejší výstupný kód s kódom, ktorý bol preložený do binárnej podoby. Spätný prekladač je však možné použiť aj na vytvorenie výstupného kódu v inom jazyku, než v akom bol pôvodný kód napísaný [26].



Obr. 2.1: Štruktúra spätného prekladača.

[2]Slúži k prevodu zdrojového kódu do zdrojového kódu v tom istom jazyku, avšak vykoná na ňom niekoľko zmien tak, aby bol takýto kód ťažko čitateľný. Obidva kódy sú však funkčne ekvivalentné. [22].

Kapitola 3

Spätný prekladač vyvíjaný v rámci projektu Lissom

Táto kapitola sa venuje popisu spätného prekladača vyvíjaného v rámci projektu Lissom. Prvá podkapitola obsahuje krátku zmienku o LLVM a prechodnej reprezentácii LLVM IR, ktorá je využívaná v tomto spätnom prekladači. V druhej podkapitole je možné nájsť informácie o BIR reprezentácii, ktorá predstavuje základnú reprezentáciu používanú v zadnej časti spätného prekladača [25]. Predposledná podkapitola obsahuje informácie o štruktúre spätného prekladača vyvíjaného v rámci projektu Lissom, pričom najväčší dôraz bude kladený na zadnú časť spätného prekladača, pretože obsahuje optimalizácie navrhnuté v tejto bakalárskej práci. Posledná kapitola popisuje optimalizácie, ktoré boli navrhnuté pred touto prácou a popisuje triedy, ktoré využívajú k svojej funkčnosti. Tieto triedy budú používané i v optimalizáciách navrhnutých v tejto práci.

Jedným z hlavných cieľov projektu Lissom je vytvoriť spätný prekladač nezávislý na architektúre (MIPS, ARM, Intel x86, atď.) a súborových formátoch (ELF, PE, Mach-O, atď.), generujúci zvolený vysokoúrovňový jazyk. Jeho využitie je plánované pri analýze malvéru napríklad pre mobilné zariadenia, tablety a iné podobné zariadenia. Momentálne sú podporované architektúry MIPS, ARM a Intel x86. Medzi podporované objektové formáty patrí ELF a PE. Výstup je možný v jazyku C alebo v modifikovanej verzii jazyka Python [25].

3.1 LLVM a LLVM IR

LLVM systém bol pôvodne navrhnutý ako inovatívny framework pre prekladače. Obsahuje množinu jazykovo nezávislých inštrukcií, veľké množstvo vstavaných optimalizačných algoritmov a prechodnú reprezentáciu LLVM IR [28]. Pre prechodnú reprezentáciu LLVM IR platí, že každá premenná je priradená iba raz na jednom mieste v programe. Jedná sa o základnú reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízkoúrovňové operácie, flexibilitu a schopnosť prevodu z množstva vysokoúrovňových jazykov. Cieľom jeho tvorcov bolo dosiahnuť jednoduchý nízkoúrovňový jazyk, ktorý je typovaný a rozšíriteľný [15]. Príklad tohto jazyka je na obrázku 3.1.

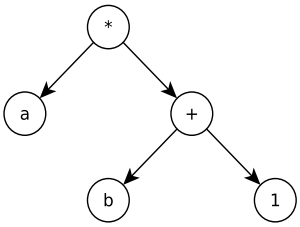
```
%1 = load i32* %a, align 4
%2 = add i32 %a, 1
ret i32 %2
```

Obr. 3.1: LLVM IR reprezentácia `return a + 1` [25].

3.2 BIR

Prechodná reprezentácia v zadnej časti spätného prekladača, skrátene BIR, je vnútornou reprezentáciou jazyka LLVM IR. Prechodná reprezentácia BIR umožňuje zachytiť všetky prvky do pamäti, pričom nie je potreba žiadnej textovej reprezentácie. Jej dôležitou schopnosťou je, že dokáže modelovať všetky konštrukcie jazyka LLVM IR a okrem toho je schopná zoskupovať viacero príkazov a výrazov do jedného, ktoré sú následne prevedené do vysokoúrovňového jazyka. Ďalšou vlastnosťou BIR je umožnenie štrukturovania kódu. Tieto postupy nie sú možné v LLVM IR, pretože LLVM IR je nízkoúrovňový jazyk. BIR následne slúži ako vstup pre konkrétny generátor výsledného jazyka [25].

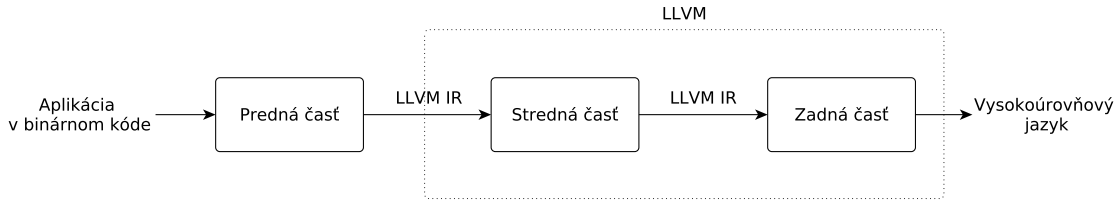
BIR obsahuje triedy, ktoré slúžia ako abstrakcia rôznych prvkov vysokoúrovňových jazykov. Napríklad funkcie sú reprezentované pomocou triedy `Function`. Telo funkcií je modelované ako abstraktný syntaktický strom. Príklad takéhoto syntaktického stromu pre výraz je zobrazený na ukážke 3.2. Medzi ďalšie významné triedy patria `Statement`, `Type` a `Expression`. Trieda `Statement` je bázovou triedou pre triedy reprezentujúce príkazy ako sú napríklad `switch` (`SwitchStmt`), `goto` (`GotoStmt`), a takisto cykly `for` (`ForLoopStmt`), `while` (`WhileLoopStmt`) a iné. Triedy ako `ArrayType`, `FloatType`, `StringType` a ďalšie dedia z bázovej triedy `Type`. Tieto triedy predstavujú rôzne dátové typy. `Expression` je bázovou triedou zastupujúcou výrazy. Z tejto triedy dedia ďalšie bázové triedy ako `Constant`, `UnaryOpExpr`, `BinaryOpExpr`, `CastExpr` a iné. Tieto uvedené 4 triedy a ďalšie už zastrešujú konkrétne triedy konštánt (napr. `ConstInt`, `ConstFloat` a iné), unárnych operátorov (napr. `NegOpExpr`, `NotOpExpr` a iné), binárnych operátorov (napr. `AddOpExpr`, `MulOpExpr` a iné), pretypovaní (napr. `IntToPtrCastExpr`, `IntToFPCastExpr` a iné) a iných častí výrazov [25].



Obr. 3.2: Reprezentácia výrazu `a * (b + 1)` v BIR [25].

3.3 Štruktúra spätného prekladača

Štruktúru tohto spätného prekladača je najlepšie vidieť na obrázku 3.3. Skladá sa z troch častí. Každá z týchto častí je samostatná a nezávislá na ostatných.

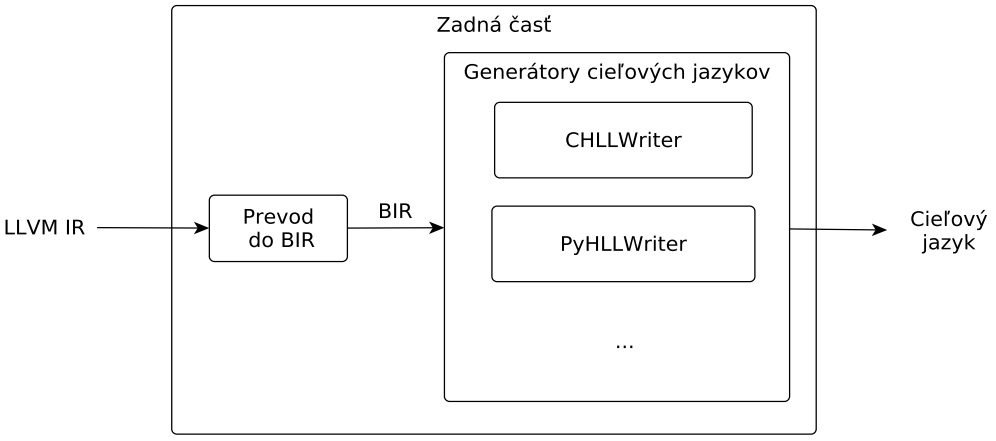


Obr. 3.3: Štruktúra spätného prekladača vyvíjaného v rámci projektu Lissom [25].

Predná časť spätného prekladača je jediná časť, ktorá je platformovo závislá. Hlavnou úlohou prednej časti je prevod binárneho kódu platformovo závislej aplikácie do sekvencie LLVM IR inštrukcií. O tento účel sa stará inštrukčný dekodér. Jeho funkcionalita je podobná disassembleru, avšak výstupom nie je kód v jazyku assembler, ale sekvencia inštrukcií odpovedajúca reprezentácii LLVM IR [27].

Táto sekvencia inštrukcií je však veľmi nízkoúrovňová. Každý základný blok predstavuje jednoduchú inštrukciu jazyka assembler. Hlavnou úlohou strednej časti je optimalizácia LLVM IR (napr. pomocou optimalizácií z LLVM) z dôvodu zlepšenia produkovaného výsledku zadnou časťou [27].

Poslednou časťou spätného prekladača je zadná časť. Jej výstupom je kód vysokoúrovňového jazyka. V zadnej časti dochádza k prevodu LLVM IR do BIR. Následne sú nad BIR vykonávane optimalizácie, ktorých úlohou je zlepšiť čitateľnosť produkovaného kódu. Práve do týchto optimalizácií patria aj optimalizácie navrhnuté v tejto práci. Produkovanie výsledného vysokoúrovňového kódu je realizované pomocou bázovej triedy `HLLWriter` a z nej dediacich tried pre konkrétne jazyky ako `CHLLWriter` a `PyHLLWriter`. Uvedené triedy pracujú s jazykom BIR a prevádzajú jeho konštrukcie do konkrétnych textových konštrukcií špecifických pre daný jazyk. Uvedený postup prevodu LLVM IR až na výsledný jazyk je zhrnutý na obrázku 3.4. Aktuálne podporované jazyky sú jazyk C a modifikovaná verzia jazyka Python. Tento Python je rozšírený napríklad o ukazatele a referenčné operátory z jazyka C. Keďže jazyk Python neobsahuje dátový typ pole ani štruktúra, tak sú tieto konštrukcie nahradené inými. Pole je nahradené dátovým typom list a štruktúra pomocou slovníku [25].

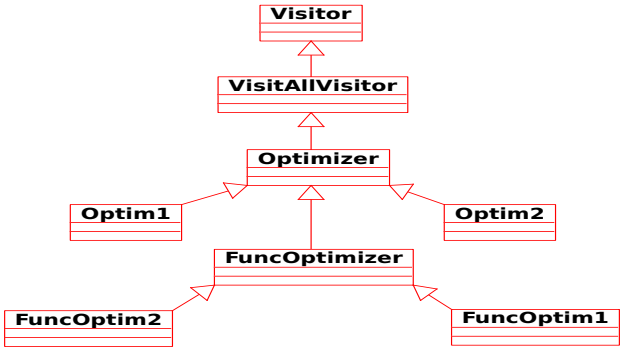


Obr. 3.4: Prevod LLVM IR do cieľového jazyka [25].

3.4 Optimalizácie v zadnej časti spätného prekladača

Pred začatím tejto práce už zadná časť spätného prekladača obsahovala rôzne optimalizácie. Jedná sa napríklad o optimalizáciu, ktorá sa podieľa na presune definícií globálnych premenných do funkcií, pokiaľ je to možné. Tým sa tieto premenné stávajú lokálnymi premennými funkcie. Ďalšia optimalizácia sa zaoberá odstránením zbytočných dereferencií nad operátorom získania adresy. Iná optimalizuje také priradenia, kde na ľavej strane je premenná a na pravej strane je priraďovaná tá istá premenná a nič iné. Okrem optimalizácií sa o úpravu výsledného kódu staral takzvaný postprocesor. Tento postprocesor bol používaný pre úpravu generovaného výstupného kódu do jazyka Python.

Princíp optimalizácií v zadnej časti spätného prekladača je ten, že pri prechode abstraktným syntaktickým stromom dochádza ku spracovaniu výrazov a následne potom podľa druhu optimalizácie a výrazu dochádza k jeho optimalizácii. Optimalizáciám však musí byť umožnený tento prechod stromom. Táto požiadavka je vyriešená pomocou dedičnosti. Vzťahy tejto dedičnosti sú znázornené na UML diagrame 3.5.



Obr. 3.5: UML diagram návrhu tried pre optimalizácie v zadnej časti spätného prekladača.

Ako prvá trieda uvedená v hierarchii najvyššie, je trieda `Visitor`. Cieľom tejto bázovej triedy je podpora pridávania operácií nad BIR tak, aby nemuselo dochádzať k úprave príslušných tried reprezentujúcich prvky BIR. Táto myšlienka vychádza z návrhového vzoru návštevník (angl. visitor) [17]. Prechod abstraktným syntaktickým stromom zabezpečuje trieda `VisitAllVisitor`. Táto trieda obsahuje metódy, ktoré sa vo východzej implementácii starajú o rekurzívny prechod celým týmto stromom. Tieto metódy je však možné vo vlastných optimalizáciách predefinovať (angl. override). Vďaka tomuto predefinovaniu môže dochádzať počas prechodu stromom k optimalizácii výrazov. Trieda `Optimizer` predstavuje základnú triedu pre takmer všetky optimalizácie. Z nej dedí trieda `FuncOptimizer`, ktorá je triedou pre všetky optimalizácie, ktoré sú aplikované na telách funkcií.

Niektoré optimalizácie, ako napríklad presun definícií globálnych premenných, využívajú k rozhodovaniu sa o optimalizácii nejakého výrazu takzvaný graf toku riadenia[1]. Prechod grafom tokom riadenia je implementovaný v triede `CFGTraversal`.

Ďalšou triedou, ktorá je veľmi často využívaná, je trieda `ValueAnalyzer`. Táto trieda obsahuje metódy, ktoré napríklad umožňujú získanie všetkých premenných využitých vo výraze, overuje výskyt pola, referenčného a dereferenčného operátora a iné.

[1] Graf toku riadenia predstavuje v informatike reprezentáciu používajúcu grafovú notáciu, pričom zobrazuje všetky cesty kadiaľ môže program vo svojom kóde prechádzať počas jeho vykonávania [8].

Kapitola 4

Výstup spätného prekladača pred vytvorenými optimalizáciami

Táto kapitola je úvodom do hlavnej časti obsahu tejto práce. Obsahuje porovnanie a analýzu výstupov produkovaných spätným prekladačom pred optimalizáciami navrhnutých v tejto práci a po nich. Tieto ukážky budú ukázané na jednoduchých a názorných častiach zdrojového kódu. Uvedené príklady sú prevzaté z reálneho kódu a sú dostupné na priloženom CD. V ukážkach sú pomenované názvy premenných pomocou anglických názvov ovocia. Takéto názvy sú čitateľnejšie a prehľadnejšie, než názvy premenných typu `var1, var2` a podobne. Z tohto dôvodu sa v spätnom prekladači využíva tento spôsob pomenovania premenných. Podrobný popis návrhu konkrétnych uvedených optimalizácií sa nachádza v kapitole 5. Implementačné riešenia sú uvedené v kapitole 6. Kapitola 7 popisuje spôsob testovania navrhnutých optimalizácií.

Medzi jednu z prvých činností, ktorú bolo potrebné vykonať, bola analýza produkovaného kódu pred navrhnutými optimalizáciami v tejto práci. Tento kód bol už čiastočne optimalizovaný pomocou niektorých navrhnutých optimalizácií a pomocou postprocesoru. Ten je založený na textovej úprave, takže nezvláda zložitejšie konštrukcie. Jedným z cieľov optimalizácií bolo nahradiť tento postprocesor do univerzálnejšej podoby tak, aby navrhnuté optimalizácie zvládali zložitejšie konštrukcie a boli použiteľné i pre jazyk C, prípadne ďalšie jazyky.

Ako prvá uvedená optimalizácia je **optimalizácia redundantných zátvoriek**. Jej hlavným cieľom je odstrániť redundantné zátvorky v čo najväčšom množstve. Vzor výstupu pred vytvorením optimalizácie a po vypnutí postprocesoru je vidieť na obrázku 4.1.

```
1   banana = (((v4 + (v3 + (v2 + v12))) - v5) - v6)
2   plum = (1.23e+2 * ((((8.0e+0 * apple) + 3.2e+1) << 1) & 2046))
```

Obr. 4.1: Ukážka kódu výstupu spätného prekladača pred odstránením redundantných zátvoriek. Uvedený kód je v jazyku Python.

Prostredníctvom priority operátorov a rôznych matematických pravidiel, ako je napríklad asociativita a komutativita operátorov, je možné redukovať počet zátvoriek. Výsledok je vidieť na obrázku 4.2. Podrobný popis tejto problematiky sa nachádza v podkapitole 5.1

```
1 banana = v4 + v3 + v2 + v12 - v5 - v6
2 plum = 1.23e+2 * (8.0e+0 * apple + 3.2e+1 << 1 & 2046)
```

Obr. 4.2: Ukážka kódu výstupu spätného prekladača po odstránení redundantných zátvoriek. Uvedený kód je v jazyku Python.

Ďalšia optimalizácia je zameraná na **zjednodušenie aritmetických výrazov**. Kód na obrázku 4.3 ukazuje niekoľko výrazov, ktoré môžu byť zjednodušené. Úlohou tejto optimalizácie je takisto nahradiť funkčnosť zmieňovaného postprocesoru a vylepšiť ju. Uvedená ukážka zobrazuje výstup po vypnutí postprocesoru.

```
1   peach = 0 + 1 + 5;
2   jambul = damson + -48;
3   result = 1 * lemon;
4   plum = apple * 4;
5   banana = apple + apple;
```

Obr. 4.3: Ukážka kódu výstupu spätného prekladača pred optimalizáciou zjednodušenia aritmetických výrazov.

Riadok 1 obsahuje numerický výraz, ktorý je možné jednoducho sčítať a tým dosiahnuť zmenšenie výrazu. Na riadku 2 je uvedený príklad na kombináciu operátoru plus a unárneho mínusu. Tieto dva operátory sa dajú nahradiť za odčítanie. Na riadku 3 je vidieť prípad násobenia číslom 1. Takéto násobenie neovplyvňuje výsledok, preto výsledný výraz bude bez tohto násobenia. Na riadku 4 ide len o kozmetickú úpravu. Prirodzenejší zápis pri násobení je taký, že číslo je prvý operand a premenná druhý operand. V poslednom riadku ide o sčítanie dvoch rovnakých operandov. Takýto výraz je možné nahradiť výrazom typu `2 * operand`. Výsledok uvedených úprav je ukázaný na obrázku 4.4.

```
1   peach = 6;
2   jambul = damson - 48;
3   result = lemon;
4   plum = 4 * apple;
5   banana = 2 * apple;
```

Obr. 4.4: Ukážka kódu výstupu spätného prekladača po optimalizácii zjednodušenia aritmetických výrazov.

Ako je vidieť, tak vďaka týmto úpravám dochádza k zlepšeniu čitateľnosti aritmetických výrazov. Uvedené úpravy nie sú všetky, ktoré boli navrhnuté v tejto optimalizácii. Jej návrh a kompletný popis sa nachádza v podkapitole 5.2.

Optimalizácia zameraná na **presun definícií premenných k najbližšiemu prvému použitiu definovanej premennej** produkuje veľký prínos pre čitateľnosť produkovaného kódu. Pôvodný výstup vo väčšine prípadoch obsahoval definície lokálnych premenných na začiatku funkcie, ako je uvedené na obrázku 4.5.

```
1   uint32_t factorial(uint32_t apple) {
2       int32_t banana;
3       int32_t result;
4       if (apple < 0) {
5           return -1;
6       }
7       if (apple == 0) {
8           result = 1;
9       } else {
10          if (apple >= 1) {
11              banana = 1;
12              result = banana;
13              return result;
14          }
15      }
16  }
```

Obr. 4.5: Ukážka kódu výstupu spätného prekladača pred optimalizáciou zameranou na presun definícií premenných k najbližšiemu prvému použitiu definovanej premennej.

Vo veľmi veľa prípadoch je možné tieto definície presunúť rovno do priradenia hodnoty pre definovanú premennú, prípadne čo najbližšie k jej prvému použitiu. Výsledok tejto optimalizácie je zobrazený na obrázku 4.6. Ako je možné vidieť, tak vďaka tejto optimalizácii sa nenachádza veľké množstvo definícií premenných na začiatku funkcie. Výstupný kód je takto pre programátora jednoduchšie analyzovateľný. Na riadku 5 je vidieť, že došlo k presunu definície premennej `result`. Je to najvhodnejšie miesto, kde mohla byť umiestnená, pretože táto premenná je použitá v tele podmienky `if`, ale aj vo vetve `else`. Definícia premennej `banana` bola presunutá na riadok 10 rovno do priradenia hodnoty, pretože táto premenná už nie je použitá nikde mimo vetvy `else`. Viac o tejto optimalizácii je možné dočítať sa v podkapitole 5.3.

```
1   uint32_t factorial(uint32_t apple) {
2       if (apple < 0) {
3           return -1;
4       }
5       int32_t result;
6       if (apple == 0) {
7           result = 1;
8       } else {
9           if (apple >= 1) {
10              int32_t banana = 1;
11              result = banana;
12              return result;
13          }
14      }
15  }
```

Obr. 4.6: Ukážka kódu výstupu spätného prekladača po optimalizácii zameranej na presun definícií premenných k najbližšiemu prvému použitiu definovanej premennej.

Cieľom **optimalizácie zámeny bitových operácií za logické operácie** je zameniť bi-

tový súčet za logický súčet [13] a bitový súčin za logický súčin [12]. Výskyt bitového súčtu a bitového súčinu v kóde pred optimalizáciou je z toho dôvodu, že LLVM IR nepodporuje logické súčet a logický súčin [15]. Táto zámena však má význam len pre podmienku v príkazoch `while`, `if`, prípadne `else if` a v príkaze `switch`. Je to z toho dôvodu, že logické operátory sú práve najčastejšie používané v podmienkach, oproti tomu bitové zase vo výrazoch. Výstupný kód pred vytvorením tejto optimalizácie je uvedený na obrázku 4.7.

```
1    if (plum > orange | orange == 0) {}
2    if (jambul > 9 & durian > 25) {}
```

Obr. 4.7: Ukážka kódu výstupu spätného prekladača pred optimalizáciou zámeny bitových operácií za logické operácie.

Riadok 1 obsahuje podmienku, v ktorej môže byť bitový súčet nahradený za logický súčet. Na riadku 2 je možné zase uskutočniť náhradu bitového súčinu za logický súčin. Tieto náhrady však nie je možné uskutočniť v každom prípade. Podrobný popis tejto problematiky je uvedený v podkapitole 5.4. Ukážka na obrázku 4.8 obsahuje výstupný kód vyprodukovaný spätným prekladačom po aplikovaní tejto optimalizácie.

```
1    if (plum > orange || orange == 0) {}
2    if (jambul > 9 && durian > 25) {}
```

Obr. 4.8: Ukážka kódu výstupu spätného prekladača po optimalizácii zámeny bitových operácií za logické operácie.

Ako posledná uvedená optimalizácia je **optimalizácia bitových posunov**. Jej užitočnosť je z toho dôvodu, že pri preklade zo zdrojového kódu často dochádza vďaka prekladaču ku zámene operácii násobenia a delenia za bitové posuny, ktoré sú jednoznačne rýchlejšie [11]. Vo výslednom kóde produkovanom spätným prekladačom sú však tieto bitové posuny menej čitateľnejšie, než ich pôvodná podoba v tvare násobenia a delenia. Časť kódu uvedená na obrázku 4.9 demonštruje výstup spätného prekladača pred touto optimalizáciou.

```
1    return apple >> 4;
2    return apple << 2;
```

Obr. 4.9: Ukážka kódu výstupu spätného prekladača pred optimalizáciou bitových posunov.

Riadok 1 v ukážke obsahuje bitový posun doprava, pre ktorý platí nasledujúce: bitový posun doprava o n bitov je rovný deleniu 2^n. Pre bitový posun doľava uvedený na riadku 2 odpovedá vzťah: bitový posun doľava o n bitov je rovný vynásobeniu o 2^n [6]. Tieto substitúcie však nie je možné vykonať v každom prípade. Podrobnejší popis tejto problematiky je uvedený v podkapitole 5.5. Výsledný kód po aplikovaní optimalizácie je uvedený na obrázku 4.10. Je vidieť, že tento výstup je čitateľnejší než pôvodný s bitovými posunmi.

```
1    return apple / 16;
2    return apple * 4;
```

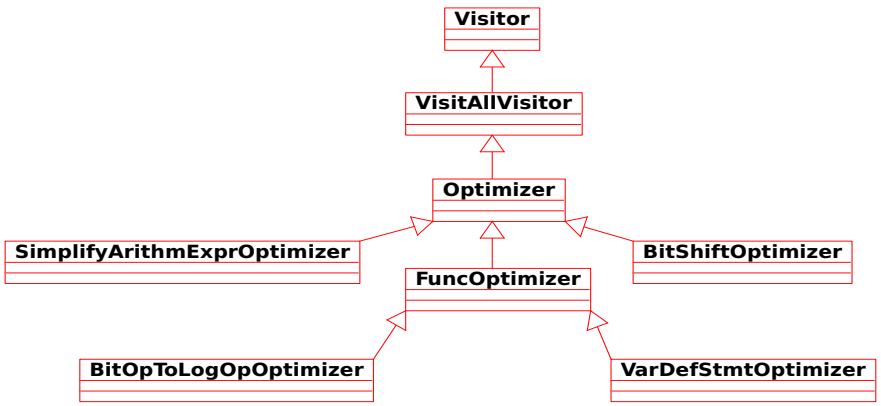
Obr. 4.10: Ukážka kódu výstupu spätného prekladača po optimalizácii bitových posunov.

Kapitola 5

Návrh optimalizácií

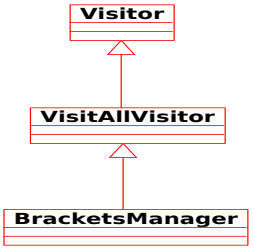
V tejto kapitole sa nachádza podrobný popis návrhu jednotlivých optimalizácií. Pri týchto návrhoch je kladený dôraz na funkčnú ekvivalentnosť výsledného kódu pred a po optimalizácii, a takisto na nezávislosť optimalizácií na konkrétny jazyk. Každá podkapitola sa venuje jednej optimalizácii, pričom obsahuje popis tejto optimalizácie, problémy ktoré boli riešené, ako aj podmienky za ktorých môže byť vykonaná. Implementačné riešenia optimalizácií sú popísané v nasledujúcej kapitole 6. Spôsobom, akým boli tieto optimalizácie testované sa venuje kapitola 7. Ukážky výstupov pred optimalizáciami a po optimalizáciách už boli uvedené v kapitole 4, preto v jednotlivých podkapitolách už nebudú uvádzané.

Optimalizácie navrhnuté v tejto práci takisto používajú prechod abstraktným syntaktickým stromom, ako tie navrhnuté už pred touto prácou. Presun definícií premenných čo najbližšie k prvému použitiu (`VarDefStmtOptimizer`) a optimalizácia bitových operácií na logické operácie (`BitOpToLogOpOptimizer`) sa uskutočňujú nad telami funkcií. Z tohto dôvodu dedia z triedy `FuncOptimizer`. Zjednodušovanie aritmetických výrazov (`SimplifyArithmExprOptimizer`) a optimalizácia bitových posunov (`BitShiftOptimizer`) sa využíva ako nad telami funkcií, tak aj nad definíciami globálnych premenných. Na základe tohto faktu dedia z triedy `Optimizer`. Vzťahy dedičnosti týchto optimalizácií sú znázornené na UML diagrame 5.1.



Obr. 5.1: Hierarchia dedičností pre optimalizácie dediace z triedy `Optimizer` a `FuncOptimizer`.

Optimalizácia redundantných zátvoriek (`BracketsManager`) v diagrame vyššie uvedená nie je, pretože táto optimalizácia sa uskutočňuje až pri generovaní cieľového kódu. Uvedený UML diagram 5.2 zobrazuje vzťahy dedičnosti pre túto optimalizáciu.

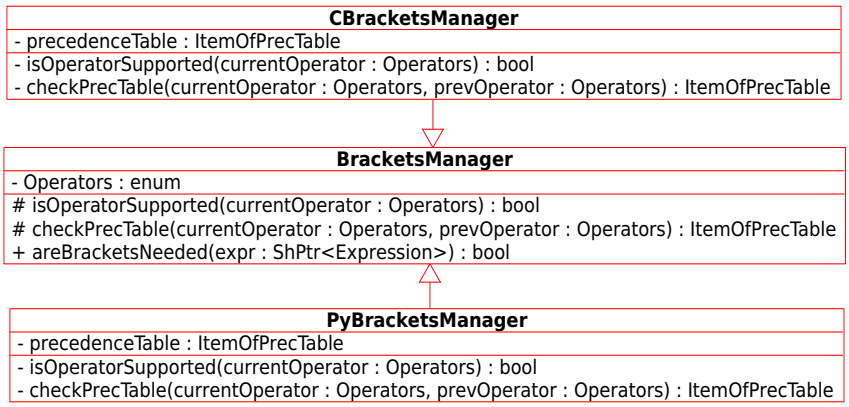


Obr. 5.2: Hierarchia dedičností pre optimalizáciu redundantných zátvoriek.

5.1 Optimalizácia redundantných zátvoriek

Jednou z funkcií už zmieňovaného postprocesoru bola funkcia odstraňovania redundantných zátvoriek. Toto odstraňovanie však nebolo stopercentné. Postprocesor bol založený len na textovej úprave, takže si nedokázal poradiť so zložitejšími konštrukciami. Úlohou tejto optimalizácie bolo nahradiť tuto funkčnosť postprocesoru a vylepšiť ju.

Pri návrhu tejto optimalizácie bolo potrebné myslieť na jej budúcu rozšíriteľnosť pre ostatné jazyky tak, aby bola čo najjednoduchšia. Z tohto dôvodu bolo navrhnuté riešenie zobrazené na UML diagrame 5.3. Bázová trieda `BracketsManager` obsahuje väčšie množstvo metód, než sú znázornené na ukážke, avšak pre jednoduchosť a vysvetlenie princípu zobrazené metódy postačia. Hlavnú úlohu v tomto návrhu zohráva takzvaný návrhový vzor šablónová metóda (angl. template method) [18]. Vďaka nemu môže byť vykonaná väčšina algoritmu v triede `BracketsManager` a časti závislé na konkrétnom jazyku prebiehajú v konkrétnych podtriedach pre daný jazyk. Každý jazyk má práve svoju jednu vlastnú podtriedu dediacu z triedy `BracketsManager`.



Obr. 5.3: UML diagram návrhu tried pre optimalizáciu redundantných zátvoriek.

Pri vytváraní tejto podtriedy je potrebné do nej umiestniť precedenčnú tabuľku (atribút `precedenceTable`). Prístup do nej je možný pomocou dvojice operátorov. Indexovaný prvok obsahuje informáciu o tom, či je medzi dvoma operátormi potrebné umiestniť zátvorku alebo nie. Tabuľka je vytvorená na základe priority operátorov, ktorú každý programovací jazyk definuje podľa svojho. V prípade, že operátory majú rovnakú prioritu, využívajú sa matematické vzťahy ako je asociativita operátorov a komutativita operátorov. Uvedené tri vlastnosti stačia takmer na úplne odstránenie redundantných zátvoriek. Existujú však kombinácie operátorov, pre ktoré tieto tri vlastnosti nestačia. Napríklad kombinácia operátoru plus, ktorý predchádza binárnemu operátoru mínus. V takýchto prípadoch je možné došpecifikovať precedenčnú tabuľku. Vďaka takémuto došpecifikovaniu sa vylepšuje minimalistickosť počtu redundantných zátvoriek. Pre vytváranie precedenčnej tabuľky bol vytvorený skript, ktorý dokáže na základe vyššie zmienených troch vlastností, prípadne došpecifikovaniu kombinácie operátorov, pre ktoré nemá byť umiestnená zátvorka, automaticky vygenerovať túto tabuľku. Ďalším krokom pri vytváraní podtriedy je predefinovanie dvoch zdedených metód, ktorých úlohou je pristúpiť do zmienenej precedenčnej tabuľky (metóda `checkPrecTable`). Úlohou druhej je overiť, či je daný operátor podporovaný v tomto jazyku (metóda `isOperatorSupported`). Toto overenie prebieha na základe prístupu do precedenčnej tabuľky. Je vidieť, že pri zavedení nového jazyka je potrebné naozaj minimum práce, aby mohla byť táto optimalizácia aplikovaná i na nový jazyk.

Vzhľadom na to, že bola popísaná hlavná myšlienka tejto optimalizácie, je možné uviesť jej princíp. V prvom rade sa uskutoční takzvaný predprechod, ktorého úlohou je prejsť všetky výrazy v abstraktnom syntaktickom strome. Pri tomto prechode dochádza vždy k uloženiu si aktuálneho operátora, ktorý sa pri prechode na nasledujúci operátor stane predchádzajúcim operátorom. S ukladaním aktuálneho operátora sa takisto na vrchol zásobníku ukladá smer zanorenia sa v abstraktnom syntaktickom strome. Tento smer sa využíva v prípade operátorov s rovnakou prioritou a využíva sa pri rozhodovaní na základe asociativity operátorov. Pokiaľ je aktuálny operátor a predchádzajúci operátor zhodný a platí pre tento operátor pravidlo komutativity, v tomto prípade nemá zmysel uvažovat o asociativite operátorov. V takomto prípade nie je potrebné umiestniť zátvorky. Následne je pre aktuálny operátor volaná metóda `checkPrecTable`, vďaka ktorej pomocou uloženého predchádzajúceho operátora a aktuálneho dôjde k vyhodnoteniu o potrebe zátvoriek. Výsledok je uložený do slovníku, kde kľúčom je výraz a hodnotou potrebnosť zátvoriek. Tento predprechod je volaný z tried slúžiacich pre generovanie BIR už do konkrétnych cieľových jazykov. Následne pri vypisovaní konkrétnych výrazov dochádza k dotazovaniu sa do zmieňovaného slovníka pomocou výrazu a na základe obsahu hodnoty dôjde, prípadne nedôjde k výpisu zátvoriek. Tento dotaz je uskutočnený pomocou metódy `areBracketsNeeded`.

Princíp tejto optimalizácie bude demonštrovaný na výraze `((3 * a) + b)`. V takejto podobe by bol tento výraz pred aplikovaním tejto optimalizácie. Ako prvý operátor, ktorý sa nachádza pri prechode abstraktným syntaktickým stromom je operátor plus. Skontroluje sa predchádzajúci operátor, avšak v tomto prípade nie je žiaden. Tento prípad reprezentuje začiatok výrazu, a preto nie je potrebné umiestniť zátvorky okolo celého výrazu. Následne dôjde k uloženiu si aktuálneho operátoru plus a smeru zanorenia sa v abstraktnom syntaktickom strome, ktorý je v tomto prípade doľava. Potom dôjde k zanoreniu sa nižšie v abstraktnom syntaktickom strome, vďaka čomu je získaný operátor krát. Následne dôjde k prístupu do precedenčnej tabuľky a vzhľadom na to, že operátor krát má väčšiu prioritu ako predchádzajúci operátor plus, tak nie je potrebné umiestnenie zátvoriek. Výsledný výraz po optimalizácii bude teda vyzerať nasledovne: `3 * a + b`.

Vďaka tomuto algoritmu dôjde k takmer úplnému odstráneniu redundantných zátvoriek. Stopercentné odstránenie redundantných zátvoriek je závislé na došpecifikovaniu všetkých operátorov, pre ktoré nestačia uvedené tri vlastnosti v návrhu tejto optimalizácie. Pre niektorých programátorov môže takýto optimalizovaný výraz bez takmer žiadnych zátvoriek pripadať neprehľadne. Napriek tomu bolo rozhodnuté pristúpiť k riešeniu, v ktorom sa dáva prednosť minimalistickému počtu zátvoriek, pretože takéto prípady sú pre užívateľa jednoducho rozhodnuteľné na základe nahliadnutia do tabuliek priority operátorov. Naopak, ak by dochádzalo k nútenému vkladaniu zátvoriek do niektorých výrazov, tak by mohlo dochádzať k zbytočnému generovaniu zátvoriek a zvýšeniu neprehľadnosti výsledku.

5.2 Zjednodušovanie aritmetických výrazov

Hlavným cieľom tejto optimalizácie je zjednodušiť najčastejšie sa objavujúce aritmetické výrazy, ktoré je možno zapísať lepšie čitateľnou podobou. Zjednodušenia sú vykonávané nad nasledujúcimi operáciami: súčet, rozdiel, súčin, podiel, bitový súčin, bitový súčet, exkluzívny súčet a modulo.

Uvedené tabuľky 5.1 až 5.8 v závere tejto podkapitoly zobrazujú príklady, ktoré výrazy boli pre jednotlivé operácie zjednodušené. Pokiaľ je v príklade uvedená premenná, tak táto premenná pre jednoduchosť zastupuje ľubovoľný výraz. Ak je v príklade uvedené číslo, tak tento operand musí pozostávať z celého čísla, prípadne desatinného. Pokiaľ sa v nejakom operande v uvedených príkladoch vyskytuje číslo jedna alebo číslo nula, tak tento operand musí mať takúto hodnotu aby mohlo dôjsť k danej optimalizácii.

V uvedených príkladoch pri vykonávaní operácie nad dvoma číslami je potrebná kontrola, či sa jedná o čísla s rovnakým počtom bitov. Je to z toho dôvodu, že výsledok môže spôsobiť pretečenie, avšak pri zámene za číslo s rozšírením na potrebný počet bitov by sme o toto pretečenie prišli. Ďalej je potrebné zabezpečiť kontrolu správnosti výsledku pri vykonávaní operácií nad dvoma číslami. Jedná sa napríklad o problémy s pretečením a podtečením. V prípade, že dôjde k takémuto stavu, optimalizácia nesmie prebehnúť. Optimalizácia výrazov nie je možná takisto v prípadoch, kde záporné číslo je najmenšie možne na prislúchajúcom počte bitov. Jedná sa napríklad o výrazy `a - -128`, `-128 / -1` a podobné. V týchto príkladoch je číslo -128 uložené na 8 bitoch. Pokiaľ by však došlo k obráteniu hodnoty na kladné číslo 128, nebolo by možné toto číslo uložiť na ôsmich bitoch v prípade, že by sa jednalo o ukladanie čísiel v doplnkovom kóde [16]. Pri delení číslom 0 alebo pri operácii modulo, kde pravý operand je číslo 0 tiež nesmie dôjsť k optimalizácii. Vyhodnotenie operácie nad dvoma číslami však môže prebehnúť len vtedy, pokiaľ sa jedná o dva operandy obsahujúce celé číslo alebo dva operandy zložené z desatinného čísla. Vzájomná kombinácia nie je možná z dôvodu, že niektoré jazyky nemusia podporovať vyhodnocovanie operácií, kde jeden operand je desatinné číslo a druhý operand celé číslo. Pri celočíselnom delení nesmie dôjsť k optimalizácii v prípade, že výsledok delenia zanecháva zvyšok. Je to z toho dôvodu, pretože pri niektorých jazykoch je pri celočíselnom delení výsledok celé číslo, pri iných desatinné. Všetky navrhnuté optimalizácie však musia byť jazykovo nezávislé, preto tento prípad nie je možné optimalizovať. Pri bitovom súčine, bitovom súčte a exkluzívnom súčte, pokiaľ operand obsahuje číslo, tak je možné vykonať optimalizácie iba na celých číslach. Uvedené operácie nepodporujú desatinné čísla, preto by nebolo vhodné optimalizovať takéto výrazy.

Možnosť optimalizácie výrazu `1 ^ (a == b)` na `a != b` je z toho dôvodu, že výsledkom porovnania je vždy hodnota `true` alebo `false`. V našom prekladači predstavuje hodnota

`true` jednobitové číslo jedna a hodnota `false` jednobitové číslo nula. Exkluzívny súčet pomocou čísla 1 s výsledkom porovnania spôsobí obrátenie hodnoty výsledku porovnania na opačnú. Toto chovanie spôsobuje takisto logický operátor `not`, takže takýto výraz by bolo možné nahradiť výrazom `!(a == b)`. Tento výraz je možné ešte zoptimalizovať na `a != b`.

Algoritmus pre túto optimalizáciu je navrhnutý nasledovne:

- Prechádzaj po jednom výrazy v abstraktnom syntaktickom strome, ktoré predstavujú súčet, rozdiel, súčin, delenie, bitový súčin, bitový súčet, exkluzívny súčet a modulo.
- Pre nájdený výraz vyhodnoť, či je možné vykonať niektorú z optimalizácií v nižšie uvedených tabuľkách.
- Výsledok vyhodnotenia:
  - V prípade, že si našiel možnú optimalizáciu, skontroluj podmienky jej vykonania. Tieto podmienky boli uvedené vyššie.
  - V prípade, že pre daný výraz neexistuje optimalizácia, choď na ďalší výraz.
- V prípade, že podmienky optimalizácie vyhovujú, uskutočni optimalizáciu a zapamätaj si, že došlo ku zmene. Následne choď na ďalší výraz.
- V prípade, že už neexistuje ďalší výraz, skontroluj či došlo ku nejakej zmene.
- Výsledok kontroly zmeny:
  - V prípade, že došlo ku zmene, prejdi znovu celý abstraktný syntaktický strom. Tento krok spôsobí ďalšie optimalizovanie už zoptimalizovaných výrazov.
  - V prípade, že ku zmene nedošlo, ukonči optimalizáciu.

Pred úpravou	Po úprave
`a * 0`	`0`
`0 * a`	`0`
`a * 1`	`a`
`1 * a`	`a`
`2 * 5`	`10`
`a * 5`	`5 * a`

Tabuľka 5.1: Optimalizácie nad operáciou súčinu.

Pred úpravou	Po úprave
`0 / a`	`0`
`a / 1`	`a`
`10 / 5`	`2`
`a / a`	`1`

Tabuľka 5.2: Optimalizácie nad operáciou podielu.

Pred úpravou	Po úprave
`a + 0`	`a`
`0 + a`	`a`
`a + - b`	`a - b`
`2 + 5`	`7`
`- a + b`	`b - a`
`2 + (2 - a)`	`4 - a`
`3 + (a - 2)`	`1 + a`
`2 + (2 + a)`	`4 + a`
`2 + (a + 2)`	`4 + a`
`(2 - a) + 4`	`6 - a`
`(a - 2) + 4`	`a + 2`
`(2 + a) + 4`	`6 + a`
`(a + 2) + 4`	`a + 6`

Tabuľka 5.3: Optimalizácie nad operáciou súčtu.

Pred úpravou	Po úprave
`a - 0`	`a`
`0 - a`	`-a`
`a - - b`	`a + b`
`2 - 5`	`-3`
`a - a`	`0`
`3 - (2 - a)`	`1 + a`
`2 - (a - 2)`	`4 - a`
`3 - (a + 2)`	`1 - a`
`3 - (2 + a)`	`1 - a`
`(3 - a) - 2`	`1 - a`
`(a - 2) - 4`	`a - 6`
`(5 + a) - 4`	`1 + a`
`(a + 5) - 6`	`a - 1`

Tabuľka 5.4: Optimalizácie nad operáciou rozdielu.

Pred úpravou	Po úprave
`0 | a`	`a`
`a | 0`	`a`
`10 | 22`	`30`

Tabuľka 5.5: Optimalizácie nad operáciou bitového súčtu.

Pred úpravou	Po úprave
`0 & a`	`0`
`a & 0`	`0`
`10 & 22`	`2`

Tabuľka 5.6: Optimalizácie nad operáciou bitového súčinu.

Pred úpravou	Po úprave
`0 ^ a`	`a`
`a ^ 0`	`a`
`10 ^ 22`	`28`
`1 ^ (a == b)`	`a != b`
`(a == b) ^ 1`	`a != b`

Tabuľka 5.7: Optimalizácie nad operáciou bitového exkluzívneho súčtu.

Pred úpravou	Po úprave
`0 % a`	`0`

Tabuľka 5.8: Optimalizácie nad operáciou modulo.

5.3 Presun definícií premenných čo najbližšie k prvému použitiu

Cieľom tejto optimalizácie je presun definícií premenných k čo najbližšiemu prvému použitiu definovanej premennej. Táto optimalizácia je zatiaľ použiteľná len pre jazyk C, pretože druhý podporovaný jazyk, ktorým je modifikovaný Python definície premenných nepodporuje. V budúcnosti však môže byť použitá i pre iné jazyky. Optimalizácia sa zameriava na definície premenných v telách funkcií. Táto optimalizácia optimalizuje len lokálne premenné, pretože globálne premenné sú používané vo väčšine prípadoch vo viacerých telách funkcií. Pokiaľ je však globálna premenná použitá len v jednej, tak takáto globálna premenná je za určitých podmienok presunutá do funkcie a stáva sa z nej lokálna premenná. Tento presun je implementovaný v optimalizácii, ktorá nebola navrhnutá v tejto práci. Pri tejto optimalizácii je dôraz kladený na zachovanie ekvivalentnosti funkcionality kódu pred a po optimalizácii. Ďalej je potrebné zabezpečiť, aby nedošlo k chybnému presunu definície tak, že tento kód bude nepreložiteľný.

Presun definícií premenných je možné vykonať dvoma spôsobmi. Po prvé sa jedná o presun definície do prvého priradenia ľubovoľnej hodnoty do definovanej premennej. S týmto však súvisí nasledujúci problém. Presun definície premennej do prvého priradenie nesmie nastať vtedy, pokiaľ v tomto priradení je použitá tá istá definovaná premenná na pravej strane. Pokiaľ by k tomu došlo, tak priradzovaná hodnota by mala nedefinovanú hodnotu. Napríklad pre kód `int a; a = a + 1;` nesmie dôjsť k optimalizácii na výraz `int a = a + 1;`.

Pokiaľ však prvý prípad nie je možný, tak v druhom prípade je snaha o umiestnenie definície čo najbližšie k jej prvému použitiu. S týmto však súvisí problém demonštrovaný na ukážke 5.4. V tomto prípade nemôže dôjsť k presunu definície `int a` na riadok 5 pred príkaz `printf`. V prípade že by k tomu došlo, z tejto premennej by sa stala lokálna premenná pre telo príkazu `if`. Avšak tá istá premenná použitá na riadku 7 by ostala nedefinovaná. Najvhodnejšie je teda presunúť túto definíciu premennej na riadok 3, pred príkaz `if`.

Ďalší problém, s ktorým musí algoritmus počítať, je nepovoliť presun definícií štruktúr a polí. Je to z toho dôvodu, že napríklad priradenie konkrétnej hodnoty do jednej bunky pola neodpovedá priradeniu tejto hodnoty do jeho definície. Tento istý problém nastáva i v prípade štruktúr. Takisto nie je možné presunúť priradenie hodnoty do jedného prvku štruktúry priamo do definície príslušnej štruktúry.

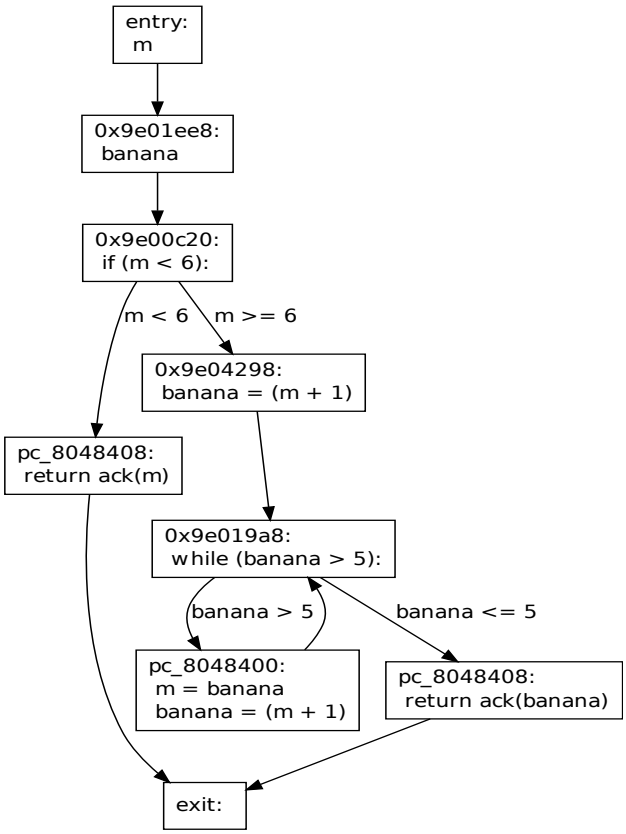
```
1    int a;
2    int b = 5;
3    printf("%d", b);
4    if (b > 5) {
5        printf("%d", a);
6    }
7    return a;
```

Obr. 5.4: Ukážka problému presunu definície premennej k jej najbližšiemu prvému použitiu.

Pri návrhu algoritmu však bolo ešte potrebné premyslieť komplikácie, ktoré mohli spôsobovať ukazatele. Pri vytváraní ukazateľu na premennú je potrebné túto premennú použiť v definičnej časti. Z toho dôvodu je toto miesto označené ako prvé použitie premennej a presun jej definície za tento ukazateľ nemôže nastať.

Umiestnenie definície do tela cyklu prípadne cyklickej slučky tvorenej príkazom `goto`, spôsobuje zmenu funkčnej ekvivalentnosti kódu. Stačí si predstaviť jednoduchý príklad s cyklom `while`. V prípade, že by došlo k umiestneniu definície premennej do tohto cyklu, tak pri každej iterácii dôjde k vytvoreniu novej premennej, čo však neodpovedá pôvodnej definícii premennej, ktorá sa vytvorila len raz. Z tohto dôvodu nesmie dôjsť k presunu definície premennej do nejakej cyklickej slučky.

Pre analýzu umiestnenia definície premennej boli použité grafy toku riadenia. Ukážkovému grafu toku riadenia na obrázku 5.5 odpovedá kód na obrázku 5.6.



Obr. 5.5: Príklad grafu toku riadenia pre kód z uvedenej ukážky 5.6.

```
1   uint32_t ack(uint32_t m) {
2       uint32_t banana;
3       if (m < 6) {
4           return ack(m);
5       }
6       banana = m + 1;
7       while (banana > 5) {
8           m = banana;
9           banana = m + 1;
10      }
11      return ack(banana);
12  }
```

Obr. 5.6: Príklad kódu pre graf toku riadenia uvedený na 5.5.

Najzložitejšia časť algoritmu je označiť čo najvhodnejší uzol v grafu toku riadenia pre umiestnenie definície premennej. Princíp tohto algoritmu je nasledovný:

I. Nájdi všetky uzly, v ktorých je použitá definovaná premenná. Označ pomocou ●.

II. Každý takto nájdený uzol označ ešte aj pomocou ■ a rekurzívne prejdi všetkých predchodcov týchto uzlov a označ ich tiež pomocou ■.

III. Následne začni od začiatočného uzlu rekurzívne prechádzať všetkých jeho následníkov, pričom:

   (a) Ak nájdeš uzol, v ktorom je premenná použitá (označený pomocou ●), pričom toto použitie nepredstavuje definíciu tejto premennej, tento uzol je výsledný. Pokračuj bodom IV.

   (b) Ak aktuálny uzol má aspoň dvoch následníkov označených ■, tak tento uzol je výsledný. Pokračuj bodom IV.

   (c) Ak neexistuje žiaden následník označený ■, tento uzol je výsledný. Pokračuj bodom IV.

   (d) Ak aktuálny uzol má aspoň dvoch predchodcov označených ■, tento uzol je výsledný. Pokračuj bodom IV.

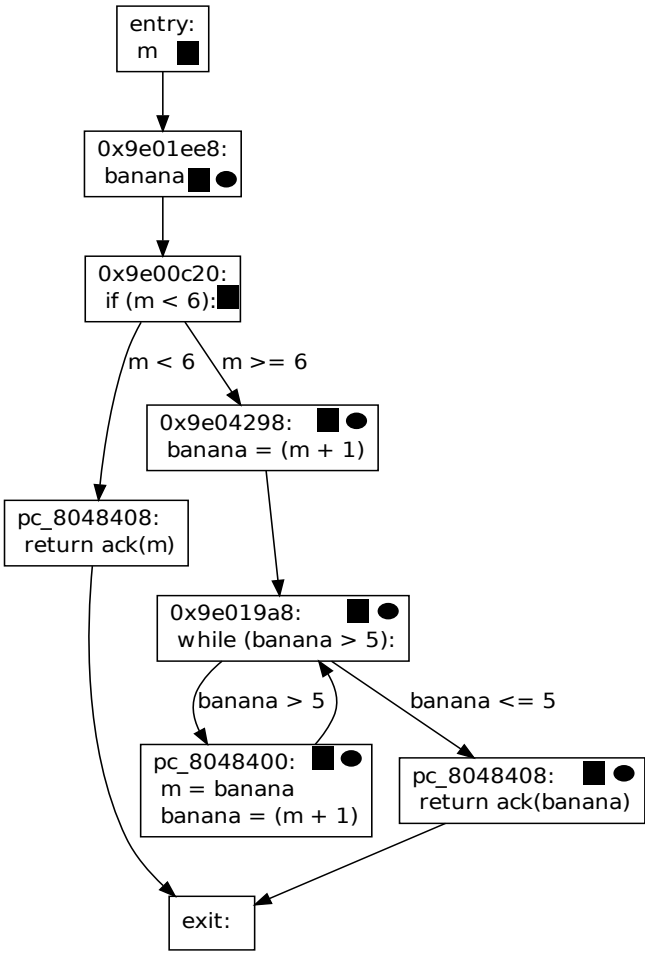
   (e) V prípade, že existuje práve jeden následník označený ■ a práve jeden predchodca označený ■, pokračuj v uzle následníka označeného ■ a vykonávaj ďalší krok na základe podmienok z bodu III.

IV. Presuň definíciu premennej do výsledného uzlu a vykonaj príslušný druh optimalizácie. Podmienky v akých prípadoch môže dôjsť buď k presunu definície do priradenia premennej alebo do tohto uzlu čo najbližšie k použitiu premennej boli zmienené vyššie.

Princíp tohto algoritmu je taký, že prejdením všetkých predchodcov uzlov, v ktorých je definovaná premenná použitá a ich označením, dosiahneme označenie uzlov odkiaľ je daná premenná dosiahnuteľná. Pomocou kontroly či daný uzol má aspoň dvoch predchodcov označených ■, je možné zistiť výskyt cyklickej slučky, pretože cyklická slučka má vždy jedného predchodcu, ktorý sa do nej dostáva a druhého, s ktorým sa do nej vracia. Nájdenie

aspoň dvoch následníkov označených ■ znamená, že tento uzol predstavuje miesto, od ktorého je premenná použitá v oddelených častiach kódu a presun do jednej z nich by spôsobil, že v jednom bloku by premenná nebola definovaná.

Výsledkom tohto algoritmu na už uvedenom grafu toku riadenia 5.5 je nasledujúci obrázok 5.7. Zamerajme sa na optimalizáciu premennej `banana`. Uzly označené ● predstavujú uzly, v ktorých bola premenná použitá. Uzly označené ■ zase miesta, z ktorých je daná premenná dosiahnuteľná, čiže predchodcov uzlov, v ktorých bola premenná použitá. Optimalizácia sa zastaví v tomto príklade na bode III po (a). Jedná sa o uzol `0x9e04298`. V tomto prípade priradenie spĺňa podmienku, že premenná na ľavej strane nie je použitá na pravej strane priradenia a preto môže dôjsť k presunutiu definície premennej `banana` rovno do tohto priradenia.



Obr. 5.7: Názorná ukážka označovania uzlov v grafe toku riadenia algoritmom pre túto optimalizáciu.

Použitý algoritmus však nezoptimalizuje úplne všetky prípady. V určitých prípadoch na základe podmienky v bode III po (b) dôjde predčasne k označeniu výsledného uzla. Takýchto prípadov je však veľmi málo. Algoritmus bol napriek tomu vybraný ako vhodný, pretože vo väčšine prípadoch dôjde k optimalizácii a zachováva ekvivalentnosť funkcionality kódu a takisto jeho správnosť. V budúcnosti by vylepšenie tohto algoritmu mohlo priniesť ešte o niečo lepší výsledok.

5.4 Optimalizácia bitových operácií na logické operácie

Táto optimalizácia sa zameriava náhradou bitového súčtu za logický súčet a bitového súčinu za logický súčin. Optimalizácia je zameraná na zámeny v podmienkach cyklu `while`, `switch`, `if` a `else if`. V týchto podmienkach sú typickejšie použitia logických operátorov namiesto binárnych.

Ako prvý bude uvedený popis **náhrady bitového súčtu**. Na uvedenom príklade 5.8 budú demonštrované všetky prípady a podmienky, za ktorých môže dôjsť, prípadne nemôže dôjsť k zámene za logický súčet.

```
1   int a = 5;
2   int c = 0;
3   int d = 2;
4   int *p = 0;
5   int h[5];
6   if (a | d) {
7   } else if (a | (d / c)) {}
8   } else if (a | (d % c)) {}
9   } else if (a | (c + func())) {}
10  } else if (a | (h[7] > 2)) {}
11  } else if (a | (d / -1)) {}
12  } else if (a | (-1 * d)) {}
13  } else if (p == 0 | *p == 5) {}
```

Obr. 5.8: Ukážka kódu s príkladmi podmienok pre optimalizáciu bitového súčtu.

Jediný prípad, kedy môže dôjsť k optimalizácii je podmienka na riadku 6. Princíp tejto zámeny je ten, že pokiaľ aspoň jeden operand obsahuje nenulovú hodnotu, resp. hodnotu ktorá nie je rovná `false`, tak vďaka logickému súčtu je vždy výsledná hodnota rovná nenulovej hodnote a teda `true`. Stačí si predstaviť bitovú podobu dvoch čísel pri ktorom je aspoň jedno nenulové, tak výsledná hodnota po bitovom súčte bude vždy nenulová [13].

Nie vždy je však možné náhradu za logický súčet vykonať. Je to z toho dôvodu, že niektoré jazyky používajú tzv. skrátene vyhodnocovanie výrazov. Pri bitovom súčte dôjde vždy k vyhodnoteniu obidvoch operandov, ale pri zámene za logický súčet stačí aby prvý operand mal hodnotu `true` a druhý sa už nikdy nevyhodnotí. Riadok 7 poukazuje na delenie číslom nula. V prípade tejto podmienky na uvedenom riadku spôsobí takéto delenie chybu. Pokiaľ by však došlo k zámene bitového súčtu v tejto podmienke za logický súčet, tak by mohol nastať prípad, v tomto príklade aj nastáva, že vďaka skrátenému vyhodnoteniu by nedošlo k spôsobeniu chyby delenia, a tak by výstup nebol ekvivalentný s pôvodným pred zámenou. Z uvedených dôvodov vyplýva, že je potrebná kontrola pravého operandu logického súčtu pri vyčísliteľných hodnotách na hodnotu nula. Pri nevyčísliteľných hodnotách je potrebné zakázať tuto optimalizáciu pri akomkoľvek delení, pretože napríklad obsah premennej môže kľudne byť hodnota nula. Rovnaký prípad nastáva aj na riadku 8, kde takisto nesmie dôjsť k modulu hodnotou 0. Riadok 9 obsahuje volanie funkcie v pravom operande bitového súčtu. Ani tento prípad nie je možné optimalizovať kvôli dôvodu skráteného vyhodnocovania. Pokiaľ by došlo k zámene, objavovali by sa situácie, kedy by nedošlo k volaniu funkcie tak, ako pri pôvodnom bitovom súčte. Z tohto dôvodu je teda nutné kontrolovať výskyt volania funkcií v pravom operande. Na riadku 10 je vidieť známy problém pod názvom prístup za hranice pola. Uvedený príklad spôsobí prístup do nealokovanej

časti pamäte, naopak v prípade nahradeniu bitového súčtu za logický by k tomuto chovaniu nedošlo. Pokiaľ by došlo na riadku 11 k náhrade bitového súčtu za logický súčet, stratili by sme vďaka skrátenému vyhodnocovaniu chybu, ktorá by mohla nastať pri delení číslom -1 na zápornom čísle, ktoré je najmenšie možne na prislúchajúcom počte bitov (napríklad `-128 / -1` na 8 bitoch). V prípade doplnkového kódu je maximálne zobraziteľné číslo vždy o jedno menšie, než to záporné [16]. Ten istý prípad nastáva aj pri násobení číslom 1, uvedenom na riadku 12. Problém však nenastáva, pokiaľ je násobený/delený operand iného typu, ako znamienkového celého čísla. V príklade na riadku 13 dôjde v tomto prípade k prístupu do nealokovanej pamäte. V prípade nahradenia za logický súčet by však vďaka skrátenému vyhodnocovaniu k nemu nedošlo. Toto teda vedie ku kontrole pravého operandu na výskyt dereferenčného operátoru.

Pri **náhrade bitového súčinu** za logický súčin je už však situácia zložitejšia. Výsledok bitového súčinu nad číslom jedna a dva je nula, čo znamená v programátorskej terminológii hodnotu `false`. Naopak nad číslom dva a dva je výsledok štyri, čo predstavuje hodnotu `true`. Ako je vidieť, tak náhrada bitového súčinu za logický súčin nie je možná vo väčšine prípadoch. Jediná možná náhrada nastáva vtedy, pokiaľ sú obidva operandy typu `bool` [12]. V našom prekladači predstavuje hodnota `true` jednobitové číslo jedna a hodnota `false` jednobitové číslo nula. Z toho vyplýva, že pokiaľ obidva operandy nadobúdajú hodnotu `true`, ich logický súčin je opäť hodnota `true`. Pokiaľ je jeden operand nepravdivý, prípadne obidva, hodnota logického súčinu je `false`. Toto správanie teda odpovedá rovnakému ako pri logickom súčine. Ďalšie podmienky, v ktorých nesmie dôjsť k optimalizácii na logický súčin, sú rovnaké ako pri podmienkach optimalizácie bitového súčtu na logický súčet.

Princíp algoritmu pre túto optimalizáciu je nasledovný:

- Prechádzaj po jednom výrazy v abstraktnom syntaktickom strome, ktoré predstavujú príkazy `if`, `while` a `switch` pokiaľ nejaké ešte existujú.
- Pre nájdený výraz analyzuj jeho podmienku, v prípade príkazu `if` analyzuj všetky príslušné `else if` podmienky.
- V analyzovaných podmienkach hľadaj bitový súčet/súčin.
- Skontroluj pre nájdený bitový súčet/súčin jeho operandy podľa vyššie uvedených podmienok.
- V prípade, že optimalizácia môže byť uskutočnená:
  - V prípade bitového súčtu nahraď bitový súčet za logický súčet.
  - V prípade bitového súčinu nahraď bitový súčin za logický súčin.

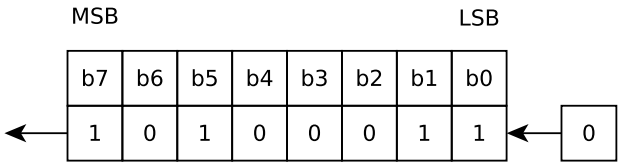
5.5 Optimalizácia bitových posunov

Jej hlavnou úlohou je zameniť bitové posuny za operácie násobenia a delenia. Táto optimalizácia je užitočná z toho dôvodu, že pri preklade zo zdrojového kódu často dochádza vďaka prekladaču ku zámene operácii násobenia a delenia za bitové posuny, ktoré sú jednoznačne rýchlejšie [11]. Pôvodná podoba v tvare násobenia a delenia je však pre bežného programátora čitateľnejšia.

Princíp zámeny bitových posunov za operácie násobenia a delenia je nasledujúci:

- Bitový posun doprava o n bitov je rovný deleniu 2^n.
- Bitový posun doľava o n bitov je rovný vynásobeniu o 2^n [6].

Tento prevod však nie je možné vykonať v každom prípade. Ako prvý bude uvedený rozbor podmienok pre **bitový posun doľava** či už aritmetický alebo logický. Princíp bitového posunu doľava je znázornený na obrázku 5.9. Pri bitovom posune doľava nemá zmysel uvažovať o tom či sa jedná o aritmetický alebo logický posun. Je to z toho dôvodu, že vždycky sprava je vložený práve jeden bit s hodnotou nula a pri všetkých ostatných bitoch dôjde k posunu práve o jeden bit doľava, čo spôsobí vyradenie pôvodného bitu na pozícii `b7` [11]. Skratka `MSB` v ukážkach predstavuje najvýznamnejší bit(angl. most significant bit) a `LSB` najmenej významný (angl. least significant bit).



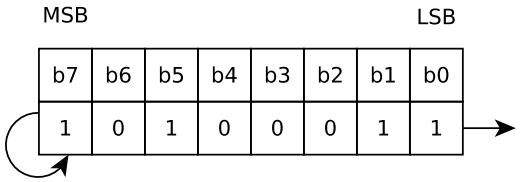
Obr. 5.9: Bitový posun doľava (aritmetický/logický).

Nasledujúci zoznam popisuje požiadavky na vlastnosti operandov, ktoré musia spĺňať, aby mohlo dôjsť k prevodu bitového posunu doľava (aritmetického/logického) na násobenie:

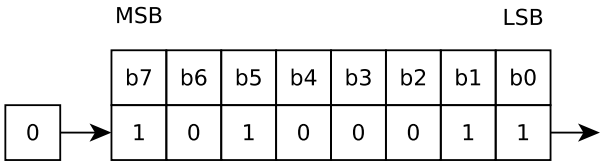
- Ľavý aj pravý operand musí byť typu `integer`. Je to z toho dôvodu, že bitové posuny nad desatinnými číslami nie sú povolené. V prípade, že by sa takýto bitový posun objavil vo výstupe spätného prekladača a my by sme ho nahradili za násobenie, došlo by k nekorektnej náhrade, pretože prvý prípad s bitovým posunom spôsobí chybu, avšak náhrada s násobením už nie.
- Ľavý aj pravý operand môže byť ako bezznamienkový, tak znamienkový. Jediný problém pri znamienkovom operande spôsobuje pretečenie. Dochádza k nemu práve vtedy, keď na pozícii najvýznamnejšieho bitu dôjde k zámene hodnoty bitu z jedna na nula, prípadne opačne [11]. Toto pretečenie však vznikne ako aj pri bitovom posune, tak aj pri násobení. Z tohto dôvodu to nie je potrebné špeciálne riešiť.
- Pravý operand môže byť len číslo typu `integer` väčšie alebo rovné od nuly. Záporné čísla nie sú povolené, pretože bitový posun nie je možné uskutočniť nad zápornými číslami a náhradou za násobenie by došlo k strate tejto chybovosti. Nič iné ako číslo nie je povolené preto, aby mohlo dôjsť k overeniu nezápornosti pravého operandu. Ďalší dôvod je ten, že v prípade nepoznania hodnoty premennej nie je aktuálne možné v našom spätnom prekladači vytvoriť výraz tvaru 2^a, pretože chýba podpora pre vytvorenie zápisu mocniny.

Druhý bitový posun je **bitový posun doprava**. V tomto prípade má zmysel uvažovať o tom či je aritmetický alebo logický. Hlavný rozdiel medzi nimi je ten, že aritmetický

bitový posun doprava zachováva na rozdiel od logického hodnotu znamienkového bitu. Obrázok 5.10 ukazuje princíp aritmetického bitového posunu doprava. Pri tomto posune dochádza ku kopírovaniu hodnoty najvýznamnejšieho bitu na pozíciu `b7`, pričom zároveň dochádza k posunu ostatných bitov práve o jeden bit doprava. Naopak logický bitový posun doprava, uvedený na obrázku 5.11, vkladá zľava bit s hodnotou nula na pozíciu `b7`, pričom dochádza k posunu všetkých ostatných bitov práve o jeden bit doprava, čím dôjde k vyradeniu bitu na pozícii `b0` [11].



Obr. 5.10: Aritmetický bitový posun doprava.



Obr. 5.11: Logický bitový posun doprava.

Na základe vyššie uvedených znalostí je možné uviesť zoznam požiadavkou na operandy pre bitový posun doprava či už aritmetický alebo logický.

- Pravý operand musí byť typu znamienkového alebo bezznamienkového `integeru`. Tento operand môže však byť len číslo väčšie alebo rovné od nuly. Uvedené dôvody sú rovnaké, ako v návrhu bitového posunu doľava.
- Ľavý operand môže byť buď typu bezznamienkového `integeru` v prípade prvkov výrazu pre ktoré nie sme schopní zistiť presnú číselnú hodnotu, alebo typu bezznamienkového, alebo znamienkového `integeru` pre čísla väčšie, alebo rovné od nuly. Tieto požiadavky na operand sú z toho dôvodu, že pri logickom posune doprava nedochádza pri záporných číslach k zachovaniu znamienkového bitu, čím by náhrada za delenie nespôsobila rovnaký výsledok ako zmienený posun [11]. V prípade aritmetického posunu doprava pre záporné čísla nastáva problém, že pre nepárne čísla nedochádza k zaokrúhleniu smerom k nule, ale naopak [24]. Nasledujúca ukážka 5.12 lepšie vysvetľuje tento problém. Výstupom riadku dva je hodnota nula, avšak výsledkom riadku tri je hodnota mínus jedna.

```
1    int a = -1;
2    printf("a / 4 = %d\n", a / 4);
3    printf("a >> 2 = %d\n", a >> 2);
```

Obr. 5.12: Ukážka problému pri aritmetickom posune doprava na nezáporných číslach.

Po uvedení všetkých problémov spojených s touto optimalizáciou je možné uviesť princíp algoritmu pre túto optimalizáciu.

- Prechádzaj po jednom výrazy v abstraktnom syntaktickom strome, ktoré predstavujú bitový posun doprava/doľava.
- Skontroluj pre nájdený výraz jeho operandy podľa vyššie uvedených podmienok v návrhu.
- V prípade, že optimalizácia môže byť uskutočnená:
  - V prípade bitového posunu doľava zameň tento posun za násobenie a pravý operand nahraď hodnotou $2^{operand}$.
  - V prípade bitového posunu doprava zameň tento posun za delenie a pravý operand nahraď hodnotou $2^{operand}$.

Kapitola 6

Implementačné riešenie optimalizácií

Táto kapitola sa venuje popisom implementácie jednotlivých optimalizácií. Každá jedna podkapitola sa venuje práve jednej optimalizácii. Všetky optimalizácie sú implementované v jazyku C++.

Prechod abstraktným syntaktickým stromom je implementovaný pomocou návrhového vzoru návštevník. Prechod týmto stromom je uskutočnený metódami `visit(ShPtr<BIRObject> par)`, ktoré sú implementované v triete `VisitAllVisitor`. Vstupným parametrom týchto metód sú je BIR objekty. Predefinovaním týchto metód vo vlastnej optimalizácii je možné upraviť správanie sa tohto prechodu, a takisto počas tohto prechodu vykonávať optimalizáciu. `ShPtr` predstavuje zdieľaný ukazateľ a jedná sa o `typedef` na `std::shared_ptr<>`. Výhoda týchto ukazateľov oproti bežným je tá, že tieto ukazatele sa automaticky samé postarajú o uvoľnenie pamäte objektov, na ktoré ukazujú [4].

V optimalizáciách často dochádza k náhrade nejakého výrazu iným. K tomuto účelu slúži statická metóda `replaceExpression(ShPtr<Expression> oldExpr, ShPtr<Expression> newExpr)`, ktorá nahradí starý výraz za nový. Táto metóda sa nachádza v triede `Expression`.

Uchovávanie a vytváranie čísel je možné pomocou tried z LLVM. Trieda `APInt` predstavuje celé číslo a trieda `APFloat` predstavuje desatinné číslo. Tieto triedy obsahujú veľké množstvo metód, ktoré je možné vykonávať nad číslami. Jedná sa napríklad o overovanie či je číslo rovné nule, či je číslo negatívne, prípadne sčitovanie dvoch čísel, odčítavanie a veľké množstvo ďalších metód. V optimalizáciách pri práci s číslami sú práve tieto metódy používané.

Umiestnenie triedy pre optimalizáciu redundantných zátvoriek je nasledujúce:
`/decompiler/decdev/backend/llvm/lib/Target/Decompiler/HLL/`.
Táto trieda je umiestnená medzi generátory cieľových kódov, pretože je z nich volaná a upravuje výstup už tesne pred jeho generovaním. Umiestnenie ostatných optimalizácií je v:
`/decompiler/decdev/backend/llvm/lib/Target/Decompiler/Optimizer/`.
Adresár `Optimizer` obsahuje všetky ostatné optimalizácie.

6.1 Optimalizácia redundantných zátvoriek

Túto optimalizáciu predstavuje bazová trieda `BracketsManager`. Pre konkrétne jazyky sú vytvorené podtriedy. Pre jazyk C je to trieda `CBracketsManager` a pre modifikovaný jazyk Python `PyBracketsManager`. Pre túto implementáciu bolo potrebné predefinovať všetky metódy `visit`, kde parametrom sú prvky BIR, ktoré sa môže vyskytovať vo výrazoch. Jedná sa napríklad o operátor plus, väčšie rovné, ale aj polia, štruktúry a iné. Takisto bolo potrebné predefinovať túto metódu, kde parametrom bolo pretypovanie.

Hlavným problémom, ktorý bolo potrebné vyriešiť, bol problém uchovania si predchádzajúceho operátora. Pri prechode abstraktným syntaktickým stromom nie je možné zistiť začiatok a ani koniec výrazu. Z tohto dôvodu bol zvolený zásobník, ktorý je v C++ reprezentovaný pomocou `std::stack<>`. Prázdny zásobník predstavuje začiatok výrazu. Pri každom prechode nejakým výrazom dochádza k uloženiu typu výrazu pomocou `enum Operators` na vrchol zásobníku. Po zanorení do tohto výrazu, predstavuje vrchol zásobníku predchádzajúci operátor. Pri rekurzívnom vynáraní, zase dochádza k odstráneniu vrcholu zásobníku. Vďaka tomuto mechanizmu predstavuje vždy vrchol zásobníku predchádzajúci operátor.

Prístup do precedenčnej tabulky je možný pomocou predchádzajúceho a aktuálneho operátoru. Táto precedenčná tabuľka je reprezentovaná pomocou dvojrozmerného pola. Ukladanie si informácie o tom či je pre daný výraz potrebné umiestniť, prípadne neumiestniť zátvorku, je realizované pomocou `std::map<>`. Kľúčom do tejto `std::map<>` je adresa výrazu a hodnotou informácia o potrebe zátvoriek.

6.2 Zjednodušovanie aritmetických výrazov

Optimalizácie je implementovaná v triede `SimplifyArithmExprOptimizer`. K opakovanému prechodu abstraktným syntaktickým stromom po všetkých funkciách a globálnych premenných, pokiaľ nastavajú nejaké optimalizácie, bolo potrebné predefinovať metódu `doOptimization()` z triedy `Optimizer`. K získavaniu výrazov obsahujúcich operátori plus, mínus, krát, deleno, bitový súčin, bitový súčet, exkluzívny súčet a modulo bolo potrebné predefinovať nasledujúce metódy ako sú `visit(ShPtr<AddOpExpr> expr)`, `visit(ShPtr<SubOpExpr> expr)`, `visit(ShPtr<MulOpExpr> expr)`, `visit(ShPtr<DivOpExpr> expr)`, `visit(ShPtr<BitAndOpExpr> expr)`, `visit(ShPtr<BitOrOpExpr> expr)`, `visit(ShPtr<BitXorOpExpr> expr)`, `visit(ShPtr<ModOpExpr> expr)`.

Ku kontrole správnosti výsledku pri vykonávaní operácii nad dvoma číslami prispieva aj LLVM vďaka svojím funkciám pre tieto operácie. Vďaka tomu je možné zachytiť vypočítanie nesprávneho výsledku, pretečenie a podtečenie.

6.3 Presun definícií premenných čo najbližšie k prvému použitiu

Trieda `VarDefStmtOptimizer` obsahuje hlavnú programovú časť tejto optimalizácie. Ďalšia trieda, ktorá je využívaná pre túto optimalizáciou a bola pre ňu implementovaná, je trieda `NoInitVarDefAnalyzer`. V tejto triede je implementovaná verejná statická metóda

`getNoInitVarDefStmts(ShPtr<Function> func)`. Vďaka nej je možné získať všetky definície premenných vo funkcii predanej parametrom, ktoré neobsahujú inicializačnú časť. Všetky tieto definície sú vrátene pomocou C++ kontajneru `std::set<>`. Tieto definície sú získane na základe predefinovania metódy `visit(ShPtr<VarDefStmt> stmt)`.

Pre túto optimalizáciu bol ešte naimplementovaný prechod grafom tokom riadenia. Tento prechod je implementovaný v triede `NodesOfVarUseCFGTraversal`. V tomto prechode dochádza k získaniu všetkých uzlov, kde bola daná definovaná premenná použitá. K implementovaniu prechodu grafu tokom riadenia bola využitá dedičnosť z triedy `CFGTraversal`, ktorá obsahuje metódu `visitStmt(ShPtr<Statement> stmt)`. Predefinovaním tejto metódy sa zabezpečila jej bežná funkčnosť a to prechod grafom tokom riadenia po jednotlivých príkazoch, avšak takisto sa pre každý príkaz overuje použitie definovanej premennej. Toto overenie prebieha na základe triedy `ValueAnalyzer`, ktorá obsahuje podporu pre získanie všetkých premenných, ktoré sú použité v danom príkaze. Na základe týchto premenných dochádza k prístupu do `std::map<>`, ktorá bola naplnená pred spustením prechodu tak, že kľúčom je adresa definovanej premennej a hodnotou definícia premennej. V prípade, že dôjde ku nájdeniu prvku, následne je použitý prístup do druhej `std::map<>`, v ktorej na základe kľúču, ktorým je nájdená adresa definície premennej z predchádzajúcej `std::map<>`, dôjde k uloženiu aktuálneho uzlu. Získanie aktuálneho uzlu, v ktorom sa nachádza skúmaný príkaz, je možné pomocou metódy `getNodeForStmt(ShPtr<Statement> stmt)` získanej z triedy `CFGTraversal`. Prvotne bol tento prístup cez dve `std::map<>` nahradený pomocou metódy z `ValueAnalyzer`, ktorá kontrolovala použitie premennej v príkaze. Ukázalo sa však, že aktuálne riešenie je rýchlejšie, preto bolo pristúpené k nemu. Získanie všetkých uzlov použitia premennej je možné pomocou verejnej statickej metódy `getNodesOfUseVariable(const VarDefStmtSet &setOfVarDefStmt, ShPtr<CFG> cfg, ShPtr<ValueAnalyzer> va)`, pričom prvý parameter je `std::set<>` obsahujúci všetky definície premenných, ktorých použitia definovaných premenných chceme hľadať. Druhý parameter predstavuje vytvorený graf toku riadenia pre danú funkciu, v ktorej chceme hľadať. Tretí parameter predstavuje objekt triedy `ValueAnalyzer`. Návratovou hodnotou tejto metódy je `std::map<>`, kde kľúčom do nej je adresa príkazu definície premennej a hodnotou je `std::set<>` obsahujúci všetky uzly použitia definovanej premennej. Využívanie `std::set<>` bolo z toho dôvodu, že je to kontajner obsahujúci iba unikátne hodnoty [3] a nebolo potrebné implementovať kontrolu na prvok, ktorý sa už v ňom nachádza.

Označovanie uzlov predchodcov od uzlu, kde bola definovaná premenná použitá ako aj hľadanie výsledného uzlu umiestnenia definície je implementované pomocou rekurzie.

Náhradu príkazu za iný umožňuje statická metóda `replaceStatement(ShPtr<Statement> oldStmt, ShPtr<Statement> newStmt)`. Priradenie príkazu pred aktuálny je uskutočnené pomocou metódy `prependStatement(ShPtr<Statement> stmt)`. Tieto metódy sa nachádzajú v triete `Statement`.

Na záver bola optimalizácia ešte testovaná na rýchlosť. Ukázalo sa, že jej časová náročnosť je príliš veľká. Z tohto dôvodu bolo pristúpené k pár riešeniam, ktoré zrýchlili jej rýchlosť. Pre testovanie rýchlosti bol použitý zdrojový kód, ktorého veľkosť je 335.8 kB. Tento kód obsahuje veľké množstvo premenných, takže bol vhodný pre testovanie. Optimalizácia na tomto súbore pôvodne trvala približne 17 sekúnd. Po vykonaní rôznych úprav pre zvýšenie rýchlosti optimalizácie sa potrebný čas zmenšil približne na 1,5 sekundy. Najväčší prínos pre zrýchlenie algoritmu mala zmena spôsobu získavania uzlov, v ktorých bola premenná použitá. V pôvodnom návrhu bol pre každú jednu premennú vytvorený nový prechod grafu

toku riadenia a v prípade nájdenia takýchto uzlov, boli tieto uzly uložené. Nový prístup vytvoril len jeden prechod, pričom v každom uzle kontroloval výskyty všetkých definovaných premenných a následne si ukladal pre konkrétnu premennú príslušné uzly. Ďalšie zrýchlenie prinieslo predávanie si hodnôt do metód pomocou konštantných referencií, vďaka čomu nedochádza k zbytočnému kopírovaniu hodnôt. Vyradenie definícií polí a štruktúr boli presunuté pred analýzu v grafe toku riadenia, čim zbytočne nad týmito prvkami nedochádzalo ku kontrole ich použití. Tieto uvedené optimalizácie pre urýchlenie boli najvýznamnejšie, ale boli vykonané ešte aj iné.

6.4 Optimalizácia bitových operácií na logické operácie

Tuto optimalizáciu predstavuje trieda `BitOpToLogOpOptimizer`. Pre získavanie príkazov, ktoré môžu obsahovať nejakú podmienku, boli predefinované nasledujúce metódy: `visit (ShPtr<IfStmt> stmt), visit(ShPtr<SwitchStmt> stmt), visit(ShPtr<WhileLoopStmt> stmt)`. Následne po získaní podmienok z týchto príkazov je nad týmito podmienkami spustený prechod abstraktným syntaktickým stromom.

Pre získavanie výrazov obsahujúcich operátori bitového súčtu a bitového súčinu boli predefinované metódy `visit(ShPtr<BitOrOpExpr> expr)` a `visit(ShPtr<BitAndOpExpr> expr)`.

Pre kontrolu, či v nejakom operande nedochádza k deleniu potencionálnou nulou bola predefinovaná metóda `visit(ShPtr<DivOpExpr> expr)`. Pri delení bolo potrebné ešte vyriešiť problém delenia číslom -1. Toto riešenie sa nachádza takisto v tejto metóde. Ďalším nežiaducim prípadom bolo modulo hodnotou nula. Pre túto kontrolu bola predefinovaná metóda `visit(ShPtr<ModOpExpr> expr)`. Problém spojený s násobením číslom jedna je riešený pomocou predefinovania metódy `visit(ShPtr<MulOpExpr>) expr`.

K overovaniu či pravý operand neobsahuje prístup do pola, volanie funkcie, prípadne výskyt dereferenčného operátora bola využitá trieda `ValueAnalyzer`. Táto trieda okrem iného sprostredkúvava metódy, pomocou ktorých na predanom výraze skontroluje príslušný výskyt uvedených nežiaducich prvkov.

6.5 Optimalizácia bitových posunov

Trieda `BitShiftOptimizer` obsahuje implementáciu tejto optimalizácie. K prechodu abstraktným syntaktickým stromom po všetkých funkciách a globálnych premenných bolo potrebné predefinovať metódu `doOptimization()` z triedy `Optimizer`.

Pre získavanie výrazov obsahujúcich operátori bitového posunu doprava a bitového posunu doľava bolo potrebné predefinovať metódy z triedy `VisitAllVisitor`. Jednalo sa o nasledujúce metódy `visit(ShPtr<BitShrOpExpr> expr)` a `visit(ShPtr<BitShlOpExpr> expr)`. V týchto metódach dochádza k overovaniu možnosti optimalizácie jednotlivých výrazov, podľa podmienok uvedených v návrhu tejto optimalizácie.

Kapitola 7

Testovanie optimalizácií

Po navrhnutí a implementovaní optimalizácií bolo potrebné overiť ich funkčnosť. V spätnom prekladači vyvíjanom v rámci projektu Lissom sú aktuálne podporované dve možnosti testovania. Vytváranie testov je potrebná činnosť, pretože počas vývoja tohto spätného prekladača dochádza k veľa úpravám už stavajúceho kódu, prípadne k implementovaniu nových prvkov. Tieto zmeny môžu mať dopad aj už na funkčné optimalizácie.

Prvý spôsob testovania je pomocou takzvaných referenčných testov. K vytvoreniu jedného testu je potrebný zdrojový súbor v jazyku C. Následne nad týmto súborom dochádza k prekladu a vygenerovaniu binárneho kódu, ktorý je spätným prekladačom preložený do výstupného súboru. Tento súbor je potom pomocou programu `diff` porovnávaný s referenčným výstupom. Takýchto testov už pred vytváraním optimalizácií v tejto práci bolo navrhnutých veľké množstvo. Po vytvorení optimalizácie bolo na týchto testoch vidieť rozdiely vo vygenerovanom kóde, ktoré spôsobovala nová optimalizácia. Referenčné výstupy bolo možné používať i ku hľadaniu nových možností optimalizácie. Pre optimalizáciu bitových posunov bolo v tejto práci navrhnutých niekoľko testov.

Druhým spôsobom testovania je testovanie pomocou jednotkových testov. Tieto testy sú písané pomocou testovacieho frameworku Google Test [10]. Výhoda týchto testov je tá, že dokážu testovať určité časti samostatne a nezávisle na ostatných, pokiaľ ich dokážeme odizolovať. Môžeme tak testovať napríklad výstup nejakej jednej metódy [21].V našom spätnom prekladači v týchto testoch vytvárame napríklad výrazy v BIR a následne nad týmito výrazmi spustíme konkrétnu optimalizáciu. Po aplikovaní optimalizácie dochádza ku kontrole výsledku a porovnávaniu očakávaných hodnôt. Príklad takéhoto testu je uvedený na obrázku 7.1. Na riadkoch 2 a 3 dochádza k vytváraniu aritmetického bitového posunu doprava, pričom hodnota ľavého operandu je -2 a pravého 5. Vytvorenie príkazu `return`, ktorého návratová hodnota je hodnota bitového posunu sa nachádza na riadku 5. Riadok 6 nastavuje telo funkcie na príkaz `return`. Na riadku 7 dochádza k aplikovaniu optimalizácie. Riadok 13 obsahuje kontrolu či výraz nebol zoptimalizovaný a ostal pôvodným bitovým posunom. Výsledok úspešného tohto testu je zobrazený na obrázku 7.2. Pokiaľ však tento test neprejde, napríklad z dôvodu, že dôjde k nesprávnemu optimalizovaniu výrazu, tak výsledok by vyzeral tak, ako je uvedený na obrázku 7.3.

Počet vytvorených testov pre optimalizácie navrhnuté v tejto práci:

- Optimalizácia redundantných zátvoriek - 53 jednotkových testov.

- Presun definícií premenných k najbližšiemu prvému použitiu definovanej premennej - 7 jednotkových testov.
- Zjednodušovanie aritmetických výrazov - 80 jednotkových testov.
- Optimalizácia bitových operácií na logické operácie - 15 jednotkových testov.
- Optimalizácia bitových posunov - 11 jednotkových testov a 10 referenčných testov.

Jednotkové testy pre optimalizáciu redundantných zátvoriek sa nachádzajú v adresári: `/decompiler/decdev/backend/llvm/unittests/Decompiler/HLL/`
Jednotkové testy pre ostatné optimalizácie vytvorené v tejto práci sú v adresári: `/decompiler/decdev/backend/llvm/unittests/Decompiler/Optimizer/`.
Umiestnenie referenčných testov pre optimalizáciu bitových posunov je v adresári: `/decompiler/testsuite/backend/HLL/Optimizations/BitShiftOptimizer/`.

Všetky referenčné testy boli vždy priebežne po navrhnutí optimalizácie ručne skontrolované, či nová optimalizácia nezaniesla nové chyby. Navrhnuté jednotkové testy všetky prechádzajú, a tak kontrolujú správnosť naimplementovaných optimalizácií.

```
1   TEST_F(BitShiftOptimizerTest, FirstNegNotOptimized) {
2       ShPtr<BitShrOpExpr> bitShr(BitShrOpExpr::create(
3           ConstInt::create(-2, 64), ConstInt::create(5, 64)
4       ));
5       ShPtr<ReturnStmt> returnStmt(ReturnStmt::create(bitShr));
6       testFunc->setBody(returnStmt);
7       Optimizer::optimize<BitShiftOptimizer>(module);
8
9       ShPtr<ReturnStmt> outReturnBody(cast<ReturnStmt>(
10          testFunc->getBody()));
11      ShPtr<BitShrOpExpr> outBitShr(cast<BitShrOpExpr>(
12          outReturnBody->getRetVal()));
13      ASSERT_TRUE(outBitShr);
14  }
```

Obr. 7.1: Príklad jednotkového testu pre optimalizáciu bitových posunov.

```
[ RUN      ] BitShiftOptimizerTest.FirstNegNotOptimized
[       OK ] BitShiftOptimizerTest.FirstNegNotOptimized (0 ms)
```

Obr. 7.2: Výsledok úspešného testu zobrazeného na obrázku 7.1.

```
[ RUN      ] BitShiftOptimizerTest.FirstNegNotOptimized
/decompiler/decdev/backend/llvm/unittests/Decompiler/Optimizer/
   BitShiftOptimizerTest.cpp:53: Failure
Value of: outBitShrOpExpr
  Actual: false
Expected: true
[  FAILED  ] BitShiftOptimizerTest.FirstNegNotOptimized (0 ms)
```

Obr. 7.3: Výsledok neúspešného testu zobrazeného na obrázku 7.1.

Kapitola 8

Záver

Cieľom tejto práce bolo preniknúť do tematiky reverzného inžinierstva a spätného prekladu. Ďalším cieľom bolo zoznámiť sa so spätným prekladačom vyvíjaným v rámci projektu Lissom a takisto jazykom LLVM IR, ktorý predstavuje základnú prechodnú reprezentáciu. Pre tento spätný prekladač boli v tejto práci navrhnuté optimalizácie za účelom zvýšenia čitateľnosti produkovaného kódu. Všetky navrhnuté optimalizácie boli otestované vytvorenými jednotkovými testami a skontrolované výstupy referenčných testov využívaných pri prvom spôsobe testovania.

V tejto práci bola popísaná problematika reverzného inžinierstva a popis princípu spätného prekladača. Ďalej sa v nej nachádza popis spätného prekladača vyvíjaného v rámci projektu Lissom. Pre tento spätný prekladač boli v tejto práci navrhnuté optimalizácie, ktorých cieľom je odstraňovanie redundantných zátvoriek, zjednodušovanie aritmetických výrazov, presun definícií premenných čo najbližšie k prvému použitiu, nahradzovanie bitových operácií za logické a nahradenie bitových posunov za operácie súčinu a podielu. Boli popísané všetky návrhy a implementačné riešenia týchto navrhnutých optimalizácií. Pre tieto optimalizácie bolo vytvorené množstvo jednotkových testov a niekoľko referenčných testov. K tejto práci bol napísaný odborný článok, s ktorým som sa zúčastnil súťaže EEICT [14] a umiestnil sa na druhom mieste v rámci informačných systémov bakalárskej formy štúdia.

Aktuálne produkovaný kód spätným prekladačom ešte obsahuje veľa námetov na možné optimalizácie. Vytvorením novej optimalizácie občas dochádza k možnosti vzniku inej. Pridávaním nových konštrukcií a jazykov do vyvíjaného spätného prekladača sa otvárajú takisto ďalšie nové možnosti. Jedna z ďalších možností budúceho vývoja je vylepšenie algoritmu optimalizácie, ktorej úlohou je presun definícií premenných k najbližšiemu použitiu definovanej premennej. Vďaka tomuto vylepšeniu by dochádzalo k bližšiemu umiestneniu definície premennej k jej prvému použitiu, pretože súčasný algoritmus v ojedinelých prípadoch označí uzol umiestnenia definície predčasne, čim nedôjde k umiestneniu definície premennej najbližšie ako je možné.

Literatura

[1] Reverse Engineering Resources. [online], [cit. 2012-12-29].
URL http://www.backerstreet.com/cg/work.htm

[2] Cipresso, T.: Software reverse engineering education. 2009.
URL http://scholarworks.sjsu.edu/etd_theses/3734

[3] cplusplus.com: std::set. [online], [cit. 2013-03-30].
URL http://www.cplusplus.com/reference/set/set/

[4] cplusplus.com: std::shared_ptr. [online], [cit. 2013-03-31].
URL http://www.cplusplus.com/reference/memory/shared_ptr/

[5] DebugMode: Decompilation. [online], Poslední modifikace 2001-10-22 [cit. 2012-12-29].
URL http://www.debugmode.com/dcompile/

[6] Department of Computer Science - University of Maryland: Bitshift Operators. [online], [cit. 2013-01-20].
URL
http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/bitshift.html

[7] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005, ISBN-10: 0-7645-7481-7.

[8] Frederic P. Miller, J. M., Agnes F. Vandome: *Control Flow Graph*. Alphascript Publishing, 2010, ISBN-10: 613-1-72029-0.

[9] Gerçek, B.: Jak funguje antivirový program? [online], [cit. 2013-3-31].
URL http://www.symantec.com/region/cz/resources/antivirus.html

[10] Google.com: Google C++ Testing Framework. [online], [cit. 2013-04-06].
URL http://code.google.com/p/googletest/

[11] harvestsoft: Bit Shifting Technique. [online], [cit. 2013-02-18].
URL http://harvestsoft.net/bitshift.htm

[12] IBM: Logical AND operator &&. [online], [cit. 2013-03-28].
URL http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2Fref%2Flogande.htm

[13] IBM: Logical OR operator ||. [online], [cit. 2013-03-28].
URL http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2Fref%2Flogore.htm

[14] Kollár, J.: Optimizations in a Retargetable Decompiler. In *Proceedings of the 19th Conference STUDENT EEICT 2013 Volume 1*, 2013, ISBN 978-80-214-4693-9, s. 185–187.

[15] LLVM Project: LLVM Language Reference Manual. [online], Poslední modifikace 2013-01-10 [cit. 2013-01-10].
URL http://llvm.org/docs/LangRef.html

[16] Marek, L.: Číselné soustavy - obecný úvod. [online], [cit. 2013-04-07].
URL http://www.cmsps.cz/~marlib/soustavy/soustavy.html

[17] Mičkat, P.: Visitor. [online], [cit. 2013-03-29].
URL http://www.algoritmy.net/article/1643/Visitor

[18] OOdesign.com: Template Method. [online], [cit. 2013-03-30].
URL http://www.oodesign.com/template-method-pattern.html

[19] Oreans Technologies: Themida Features. [online], [cit. 2012-04-08].
URL http://www.oreans.com/themida_features.php

[20] Rouse, M.: Disassemble. [online], Poslední modifikace září 2005 [cit. 2012-12-29].
URL http://whatis.techtarget.com/definition/disassemble

[21] Rouse, M.: unit testing. [online], Poslední modifikace únor 2007 [cit. 2013-03-29].
URL
http://searchsoftwarequality.techtarget.com/definition/unit-testing

[22] Russell, K. J.: Obfuscation. [online], Poslední modifikace říjen 2006 [cit. 2012-12-29].
URL http://searchsoftwarequality.techtarget.com/definition/obfuscation

[23] TechTerms.com: Malware. [online], [cit. 2013-2-17].
URL http://www.techterms.com/definition/malware

[24] Wikipedia contributors: Arithmetic shift. [online], Poslední modifikace 2013-03-23 [cit. 2013-03-29].
URL http://en.wikipedia.org/wiki/Arithmetic_shift

[25] Zemek, P.: *Design of a Language for Unified Code Representation*. Interná technická správa projektu Lissom, 2012.

[26] Ďurfina, L.; Křoustek, J.; Zemek, P.: Generic Source Code Migration Using Decompilation. In *10th Annual Industrial Simulation Conference (ISC'2012)*, EUROSIS, 2012, ISBN 978-90-77381-71-7, s. 38–42.

[27] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. *International Journal of Security and Its Applications*, 2011: s. 91–106, ISSN 1738-9976.

[28] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Design of an Automatically Generated Retargetable Decompiler. In *2nd European Conference of COMPUTER SCIENCE (ECCS'11)*, North Atlantic University Union, 2011, ISBN 978-1-61804-056-5, s. 199–204.

[29] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Detection and Recovery of Functions and Their Arguments in a Retargetable Decompiler. In *19th Working Conference on Reverse Engineering (WCRE 2012)*, IEEE Computer Society, 2012, ISBN 978-0-7695-4891-3, s. 51–60.

Dodatok A

Obsah DVD

- Text práce v elektronickej podobe.
- Zdrojové kódy implementácie optimalizácií navrhnutých v tejto práci.
- Spustiteľná verzia spätného prekladača.
- Referenčné testy využívané v spätnom prekladači pre globálne testovanie spätného prekladu.
- Referenčné testy vytvorené v rámci tejto práce pre optimalizáciu bitových posunov.
- Zdrojové kódy jednotkových testov k navrhnutým optimalizáciám v tejto práci.
- Súbor README s pokynmi pre spustenie spätného prekladu, spustenie testov a informáciami o adresároch na DVD.
- Skript, ktorý zjednodušuje prácu pri zobrazovaní výsledkov jednotkových a referenčných testov.
- Skript slúžiaci na generovanie precedenčnej tabuľky pre optimalizáciu redundantných zátvoriek.
- Rôzne balíčky ako sú gnuarm, pspsdk, LLVM-clang a iné, potrebné k spätnému prekladu.