VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

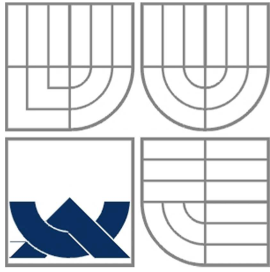
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ZADNÍ ČÁST ZPĚTNÉHO PŘEKLADAČE PRODUKUJÍCÍ KÓD V JAZYCE C

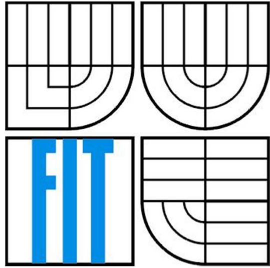
BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE MARTIN URBAN
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ZADNÍ ČÁST ZPĚTNÉHO PŘEKLADAČE PRODUKUJÍCÍ KÓD V JAZYCE C

C BACK-END FOR A DECOMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN URBAN
AUTHOR

VEDOUCÍ PRÁCE ING. PETR ZEMEK
SUPERVISOR

BRNO 2008

Abstrakt

Práce popisuje implementaci zadní části zpětného překladače produkujícího kód v jazyce C. Obsahuje základní informace o principech a využití reverzního inženýrství v oblasti informačních technologií i mimo něj. Hlavním cílem je vytvořit zadní část zpětného překladače, která bude generovat kód ekvivalentní vůči vstupu, který bude opět přeložitelný do binární formy se zachováním stejné funkčnosti jako zdrojový binární kód. Výstupem je implementace tříd v jazyce C++, vykonávající popisovanou činnost jako součást obecného dekompilátoru, který je vyvíjený v rámci projektu Lissom.

Abstract

This thesis deals with the implementation of the back-end of the decompiler, which produces a code in C language. It contains basic information about the principals and using of the reverse engineering either in the area of information technology or apart from it. The main goal is to create the back-end of the decompiler which would generate a code that would be equivalent against the input and will be translatable into a binary code. Functionality of the output code will be conserved in state of the functionality of the source code. The output is the implementation of the classes in C++ language. It does described activity as a part of the general decompiler which is developed in terms of the project Lissom.

Klíčová slova

Reverzní inženýrství, zpětný překlad, překladač, dekompilátor, Lissom, LLVM IR, jazyk C.

Keywords

Reverse engineering, decompilation, compiler, decompiler, Lissom, LLVM IR, C language.

Citace

Urban Martin: Zadní část zpětného překladače produkující kód v jazyce C, bakalářská práce, Brno, FIT VUT v Brně, 2012

Zadní část zpětného překladače produkující kód v jazyce C

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Zemka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Martin Urban
15.5.2012

Poděkování

Na tomto místě bych rád poděkoval mému vedoucímu Ing. Petru Zemkovi za odborné rady, vedení a cenné připomínky, které mi poskytoval při vypracování bakalářské práce.

© Martin Urban, 2012
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Obsah 4
Úvod 6
1 Reverzné inžinierstvo a dekompilácia 7
1.1 Reverzné inžinierstvo 7
1.2 Softvérové reverzné inžinierstvo 7
1.3 Využitie reverzného inžinierstva 8
1.3.1 Nebezpečný softvér 8
1.3.2 Kryptografia 9
1.3.3 Správa digitálnych práv 9
1.3.4 Disassembling a dekompilácia 9
1.3.5 Vývoj softvéru 9
1.3.6 Zisk strateného kódu a úpravy aplikácií bez zdrojových kódov 9
1.3.7 Migrácia kódu 10
1.3.8 Verifikácia 10
1.4 Legálnosť reverzného inžinierstva 10
1.5 Dekompilácia 10
2 Dekompilátor projektu Lissom 12
2.1.1 Jazyk LLVM IR – vnútorná reprezentácia 12
2.1.2 Front-end 13
2.1.3 Middle-end 13
2.1.4 Back-end 13
3 Pridanie výstupu do jazyka C 15
3.1 Pridanie nového jazyka 15
3.2 Oblasti, pre ktoré chýba v aktuálnom riešení podpora 15
3.2.1 Informácie o typoch a pretypovaní 16
3.2.2 Hlavičkové súbory externých knižníc 16
3.2.3 Deklarácie funkcií 16
4 Návrh pridania podpory jazyka C 17
4.1 Získanie hlavičkových súborov k funkciám 17
4.2 Generovanie prototypov funkcií 17
4.3 Návrh dátových typov 18
4.3.1 Celočíselné dátové typy 18
4.3.2 Dátové typy s pohyblivou rádovou čiarkou 19
4.3.3 Dátový typ pole 19

4.3.4 Dátový typ štruktúra ... 19
4.3.5 Ostatné typy ... 19
4.4 Pretypovanie ... 20
5 Implementácia ... 21
5.1 Databáza hlavičkových súborov ... 21
5.2 Reprezentácia dátových typov ... 21
5.2.1 Spoločné črty implementácie reprezentácie typov ... 21
5.2.2 Typy s pohyblivou rádovou čiarkou ... 22
5.2.3 Polia a štruktúry ... 22
5.2.4 Neznámy typ ... 22
5.3 Výrazy pretypovania ... 22
5.4 Generátor výstupného kódu v jazyku C ... 23
5.4.1 Spracovanie výrazov, návrhový vzor návštevník ... 23
5.4.2 Hlavičkové súbory, užívateľské typy, globálne premenné ... 24
5.4.3 Prototypy funkcií a externé funkcie ... 25
5.4.4 Bloky, riadiace štruktúry a výrazy ... 25
5.4.5 Deklarácie, definície a použitie premenných, funkcie ... 25
5.4.6 Textová reprezentácia typov ... 26
5.4.7 Výrazy pretypovania ... 26
6 Výsledky a vyhodnotenie ... 28
6.1 Sada testov ... 28
6.2 Rozbor zaujímavých výsledkov testov ... 28
6.3 Zhrnutie problémov funkčnosti ... 31
Záver ... 33
Literatúra ... 34
Zoznam príloh ... 35

Úvod

V oblasti teórie prekladačov sa čím ďalej, tým viac do popredia dostávajú pojmy ako reverzné inžinierstvo, spätný preklad alebo dekompilácia. Jedná sa o proces, pri ktorom dochádza k opačnému deju ako v prípade prekladu. Pri spätnom preklade dochádza k transformácii spustiteľného binárneho súboru na vyššiu formu reprezentácie. Hlavnou náplňou tejto práce je práve výstup do jazyka vyššej reprezentácie – jazyka C. [1]

Od spätných prekladačov sa v budúcnosti očakáva významný pokrok v oblasti prevencie proti nebezpečnému softvéru, veľké využitie v kryptoanalýze, umožnenie migrácie programov medzi jednotlivými platformami alebo programovacími jazykmi, ale aj pri vývoji softvéru. [2]

Cieľom tejto bakalárskej práce je navrhnúť a implementovať zadnú časť spätného prekladača (tzv. back-end). Úlohou tejto časti dekompilátoru je produkovať výstup v cieľovom jazyku. Pre tento účel bude skúmané a študované aktuálne riešenie spätného prekladača, hlavne zadnej časti a existujúceho riešenia výstupu do jazyka Python. Na základe týchto vedomostí bude do súčasnej implementácie doplnená podpora ďalšieho programovacieho jazyka – C.

Práca vzniká ako súčasť výskumného projektu Lissom prebiehajúceho na Fakultě informačních technológií v Brně.

Prvá kapitola práce slúži k oboznámeniu sa s reverzným inžinierstvom a dekompiláciou. Obsahuje príklady využitia a podkapitolu o riešení legálnosti reverzného inžinierstva. S dekompilátorom projektu Lissom, a jeho časťami sa oboznámime v nasledujúcej kapitole. Oboznámime sa i so zadnou časťou a rôznymi reprezentáciami kódu na ceste od binárneho programu až po kód vo vyššom programovacom jazyku. Obsah tretej kapitoly sa už týka priamo možnosti pridania výstupu do ďalšieho vyššieho programovacieho jazyka, vrátané vytipovania oblastí, v ktorých v súčasnom riešení chýba podpora. Ďalšie tri kapitoly popisujú konkrétny návrh a implementáciu riešenia pridania podpory jazyka C. Toto riešenie je otestované sadou testov, ktorých výsledky sú diskutované a vyhodnotené. V závere sú diskutované nedostatky, budúci vývoj a možné vylepšenia tejto práce.

1 Reverzné inžinierstvo a dekompilácia

Obsah tejto kapitoly vychádza zo zdrojov [2] a [3]. V tejto kapitole načrtneme pojem reverzného inžinierstva. Následne si ho zaradíme do oboru informačných technológií a bližšie sa zoznámime s princípom dekompilácie.

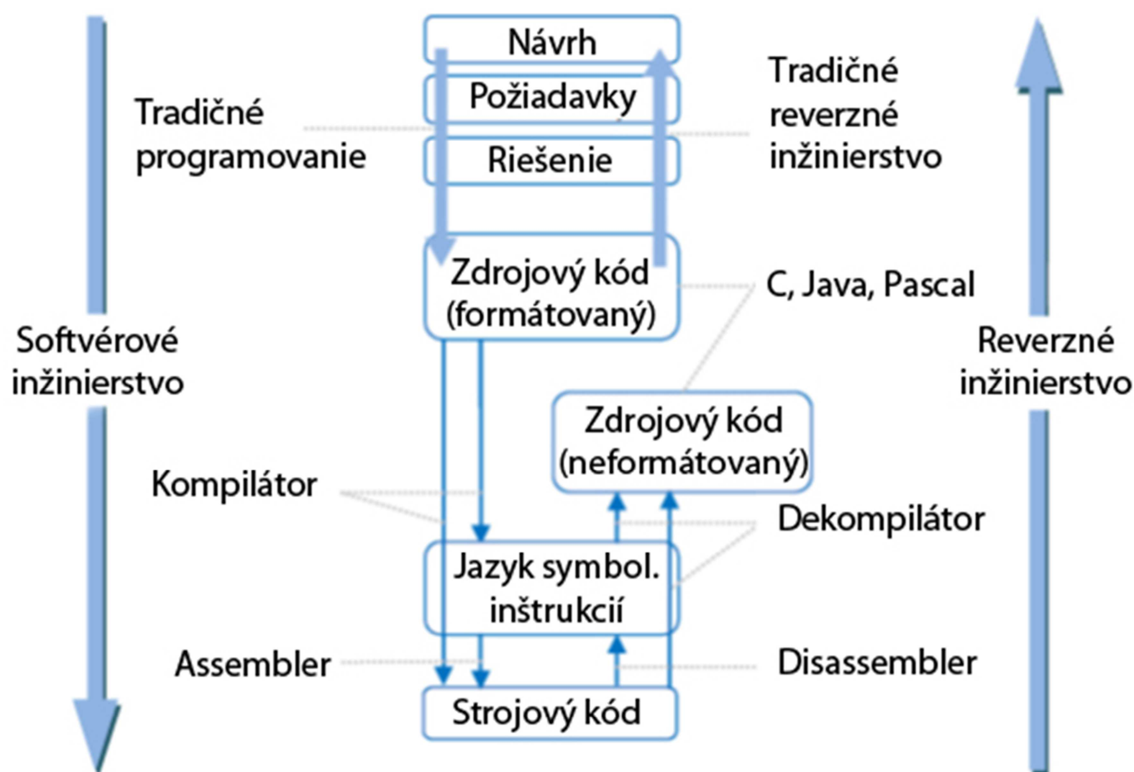
1.1 Reverzné inžinierstvo

Neustále používame obrovské množstvo rôznych aplikácii a pritom sa vôbec nezamýšľame ako to vlastne funguje. Pripájame sa na rôzne stránky, inštalujeme programy či utility. Ale ak sa aspoň na chvíľu zastavíme a pomyslíme si, ako by to či ono mohlo fungovať, tak to je prvý krok k reverznému inžinierstvu. Ak sa programátor pozrie na kód iného programátora, alebo dokonca na svoj kód po určitom čase, tak môžeme hovoriť opäť o reverznom inžinierstve. Jedná sa o rôzne postupy a nástroje, ktorými získavame informácie o čom reálne daný software je. Formálne reverzné inžinierstvo môžeme definovať ako „proces analýzy predmetu systému s cieľom identifikovať komponenty systému, ich vzájomne vzťahy a vytvoriť reprezentáciu systému v inej forme alebo na vyššej úrovni abstrakcie“ (IEEE 1990).

Koncept reverzného inžinierstva tu však bol už dávno pred počítačmi. Takmer vo všetkých vedných oboroch dochádza k skúmaniu pôvodu, významu či štruktúry daného javu alebo predmetu. Napríklad vo fyzike to je skúmanie stavby látok, molekúl, atómov a stovky ďalších problémov. V medicíne môžeme spomenúť stavbu buniek alebo v genetike rozbor DNA. Reverzné inžinierstvo nájdeme napríklad aj v bežnom živote, keď sa nám pokazí nejaké zariadenie (rádio, televízor, auto). Okamžite začneme zisťovať, ako by to mohlo fungovať a ako by sa to dalo opraviť. Toto všetko je reverzné inžinierstvo.

1.2 Softvérové reverzné inžinierstvo

Softvér a softvérové inžinierstvo patrí v súčasnosti medzi najkomplexnejšie technológie a technologické postupy. [3] Reverzné inžinierstvo sa využíva na to, aby sme sa do tohto „balíčka“ mohli pozrieť. Tento proces si môžeme predstaviť ako opačný proces k softvérovému inžinierstvu. Obidva tieto procesy majú takmer identické fázy (návrh, implementácia, zdrojový kód až po výsledný produkt), akurát v opačnom chronologickom poradí. Podmienkou softvérového reverzného inžinierstva je, že ak na výsledok jeho procesu uplatníme opäť proces opačný, snažíme sa dostať produkt, ktorý sa čo najviac blíži pôvodnému produktu. To znamená, že reprezentácie vo všetkých fázach musia byť absolútne ekvivalentné.



Obrázok 1.1 – Prehľad jednotlivých metód reverzného inžinierstva (prevzaté z [2])

1.3 Využitie reverzného inžinierstva

Reverzné inžinierstvo v informačných technológiách má široké spektrum použitia. Niektoré najpodstatnejšie oblasti obsahujú nasledujúce podkapitoly. Hlavné uplatnenie nájdeme v oblasti bezpečnosti a vývoja softvéru.

1.3.1 Nebezpečný softvér

V súčasnej dobe na nás na internete číhajú všelijaké nástrahy v podobe počítačových vírusov, červov a iného nebezpečného softvéru. Tvorcovia týchto bezpečnostných hrozieb využívajú reverzné techniky na objavenie slabých miest vášho systému. Na druhej strane sa reverzné inžinierstvo využíva pri tvorbe antivírusových systémov, v rámci ktorých sa používa na analýzu, zabezpečenie proti napadnutiu či odstráneniu týchto hrozieb. Čím je reverzný proces kvalitnejší a nová reprezentácia lepšie porozumiteľná pre človeka, tým je zabezpečenie kvalitnejšie a rýchlejšie dostupné.

1.3.2 Kryptografia

Kryptografia alebo šifrovanie je náuka o utajovaní správ prevodom do formy, ktorá je čitateľná len so špeciálnymi vedomosťami. Reverzné inžinierstvo má využitie hlavne v kryptoanalýze (lúštenie zašifrovaných správ, overovanie kvality a spoľahlivosti šifry). Jedná sa o využívanie rôznych metód (vrátane reverzného inžinierstva) na prelomenie šifier a poukázanie na potrebu jej výmeny alebo aktualizácie.

1.3.3 Správa digitálnych práv

Správa digitálnych práv alebo DRM (Digital Rights Management) je pojem, ktorý zastrešuje metódy na zabezpečenie dodržiavania autorských práv, licenčných podmienok a ochranu proti nelegálnemu šíreniu. Tzv. piráti alebo „crackeri" využívajú techniky na odhalenie týchto DRM metód k odstráneniu ochrany.

1.3.4 Disassembling a dekompilácia

Disassembler je program, ktorý prevádza binárny kód do vyššej reprezentácie v podobe assembleru, teda jazyka symbolických inštrukcií, ktorý je pre programátora oveľa lepšie čitateľný. Prevod do ešte vyššej reprezentácie, teda do niektorého z vyšších programovacích jazykov (napr. C) sa nazýva dekompilácia. Táto téma je bližšie popísaná v kapitole 2.5.

1.3.5 Vývoj softvéru

Vo vývoji je použitie reverzného inžinierstva hneď v niekoľkých fázach a oblastiach. Môže slúžiť ku skúmaniu nedokumentovaného kódu, kódu tretích strán alebo k získavaniu technológií z cudzích produktov. Najväčší význam však má pri vývoji nástroj zvaný debugger, ktorý slúži na hľadanie chýb, zdokonaľovanie a ladenie programu. Podporuje najrôznejšie metódy, ktoré programátorovi pomáhajú odhaliť chyby a nedostatky v programoch, tým že môže po krokoch sledovať beh svojho programu.

1.3.6 Zisk strateného kódu a úpravy aplikácií bez zdrojových kódov

Občas sa môže vyskytnúť situácia, kedy môžeme mať aplikáciu vo forme spustiteľného kódu, ale zdrojový kód nám z nejakých dôvodov chýba. Buď sa časom doslova stratil, stratil sa z dôvodu technických závad alebo napríklad firma stratí zamestnanca a s ním i kompletné alebo čiastočné zdrojové kódy. Dekompiláciou môžeme tento kód získať späť. Ale niekedy ani nepožadujeme samotný kód, ale snažíme sa získať znalosti, čo niektorá aplikácia vykonáva alebo ako to vykonáva. Často môže týmto dôjsť k doplneniu nekvalitnej alebo chýbajúcej dokumentácie aplikácie.

1.3.7 Migrácia kódu

Preklad z vyššieho programovacieho jazyka do spustiteľného kódy je samozrejmosť. Ak dokážeme dekompilovať spustiteľné kódy do rôznych vyšších programovacích jazykov, tak nič nám nebráni v prechodoch medzi týmito jazykmi – migráciou.

Napíšeme aplikáciu napríklad v jazyku C, preložíme aplikáciu do binárneho kódu. V prípade, že máme dekompilátor, ktorý produkuje kód v jazyku Python, po spätnom preklade získame ekvivalentný kód k pôvodnej reprezentácii v jazyku C. Prebehla migrácia kódu z jazyka C do jazyka Python.

1.3.8 Verifikácia

Verifikácia je dôležitý proces u prekladačov, kedy zisťujeme či výstup je konzistentný so vstupom. Preto je veľmi zaujímavé využitie dekompilátoru pri verifikácii správnosti implementácie prekladačov.

1.4 Legálnosť reverzného inžinierstva

Na tieto témy sa vedú nekonečné diskusie, ale už z predchádzajúcich príkladov sme mali pocit, že nie všetky spôsoby využívania sú úplne v poriadku. Negatívnou stránkou je nepochybne „cracking“, pirátstvo alebo plagiátorstvo, kedy väčšinou dochádza k hrubému porušovaniu autorských zákonov, licenčných podmienok alebo patentov. Touto činnosťou dochádza obvykle ku krádežiam, získavaniu informácii v niečí prospech alebo ich zneužívaniu. Ak sa neporušuje nič z týchto zákonov a podmienok môžeme povedať, že reverzné inžinierstvo nie je nelegálne.

1.5 Dekompilácia

Dekompilátor alebo spätný prekladač je nepochybne jeden z najzaujímavejších nástrojov reverzného inžinierstva. Jednoducho by sme mohli povedať, že jeho funkcia je úplne rovnaká ako má prekladač (kompilátor) akurát v opačnom poradí. Prekladač programovacieho jazyka prevádza program vo vyššom programovacom jazyku do strojovej podoby. Dekompilátor však generuje kód vo vyššom programovacom jazyku a prijíma ako vstup zdrojový kód, ktorý sa analyzuje oveľa horšie.

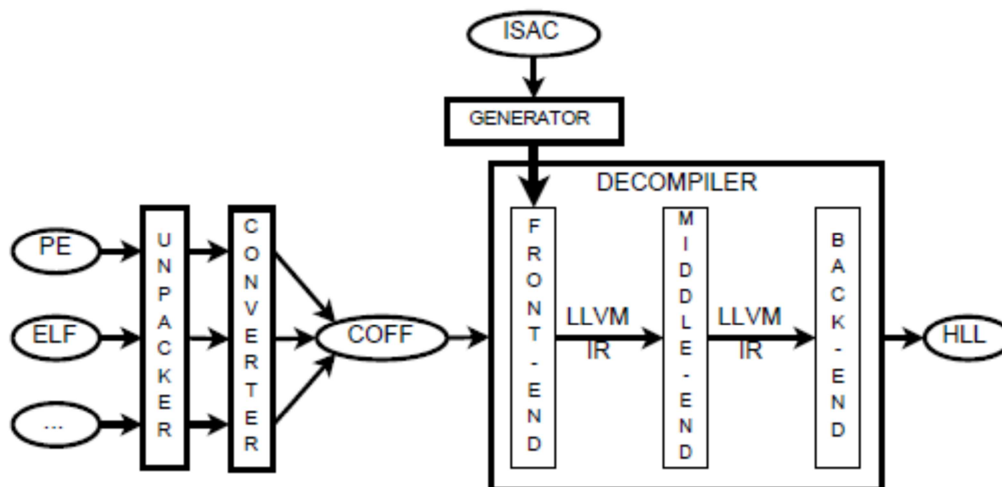
Pri prekladaní programu do strojovej podoby sa stráca obrovské množstvo informácií ako komentáre, informácie o štruktúrach, informácie o typoch, názvoch premenných a tiež dochádza k úprave štruktúry programu optimalizáciami prekladača. Tieto chýbajúce informácie spätný proces prekladu komplikujú a jeho analýza je oveľa náročnejšia, ale nie nemožná. Samozrejme nemôžeme očakávať, že výstup dekompilácie bude identický a rovnako dobre čitateľný ako kód napísaný

človekom, ale musí byť sémanticky správny a funkčnosťou identický zdrojovému programu. Všeobecne dekompilátor pozostáva z troch hlavných častí:

- *predná časť (front-end)* – táto časť na vstupe očakáva binárny kód alebo nejaký nízkoúrovňový jazyk – jazyk symbolických inštrukcií procesoru, ktorý dekóduje a prekladá na určitý druh nízkoúrovňovej vnútornej reprezentácie
- *stredná časť (middle-end)* - vnútorná reprezentácia (IR – Intermediate Representation) nižšej úrovne je optimalizovaná a prevádzaná na vnútornú reprezentáciu vyššej úrovne, jedná sa o „medzikrok“ medzi nižším a vyšším programovacím jazykom. Vnútorná reprezentácia vyššej úrovne oproti jazyku symbolických inštrukcií eliminuje málo využiteľné informácie a celková stavba je taká, aby čo najviac uľahčila prevod do vyššieho programovacieho jazyka.
- *zadná časť (back-end)* – prevod vnútornej reprezentácie na kód cieľového vyššieho programovacieho jazyka

2 Dekompilátor projektu Lissom

Informácie využité v tejto kapitole som čerpal v týchto zdrojoch [4], [5], [6] a [7].
Hlavnou úlohou dekompilátoru je analýza binárneho kódu a jeho transformácia do vyššieho programovacieho jazyka. Dekompilátor projektu Lissom sa vyznačuje tým, že sa jedná o všeobecný dekompilátor, teda nie je platformovo závislý. Funguje tak, že užívateľ popíše svoju architektúru v jazyku ISAC ADL (jazyk vyvinutý v rámci projektu Lissom pre popis rôznych architektúr mikroprocesorov) a následne sa automaticky vygenerujú potrebné nástroje na dekompiláciu pre potrebnú architektúru. Teda ak sa vrátime k rozdeleniu z predchádzajúcej kapitoly, tak platformovo závislá je iba predná časť (front-end). Celý proces dekompilácie popisuje obrázok 3.1. Na ňom vidíme, že binárna reprezentácia danej platformy (Windows PE, Unix ELF, ...), je často zbalená a chránená proti reverznému inžinierstvu, preto je nutná prvotná fáza rozbalenia a konvertovanie do objektového formátu založeného na COFF. Využijú sa spomínané vygenerované nástroje prednej časti z popisu ISAC ADL. Dekompilátor zložený z prednej, strednej a zadnej časti sa postará o prevedenie kódu do reprezentácie vyššieho programovacieho jazyka (Python, C...).



Obrázok 2.1 – Koncept všeobecného dekompilátoru Lissom (prevzaté z [4])

2.1.1 Jazyk LLVM IR – vnútorná reprezentácia

Pre pochopenie práce Lissom dekompilátoru je potrebné sa zoznámiť s jazykom vnútornej reprezentácie.

LLVM - Low-Level Virtual Machine - je to kompilačný zdrojovo a cieľovo nezávislý systém, ktorý umožňuje efektívnu optimalizáciu pri preklade a zostáva transparentný pre programátora. Využíva virtuálnu sadu inštrukcií LLVM IR. Jedná sa o nízkoúrovňovú reprezentáciu obohatenú o jazykovo nezávislé informácie o typoch, toku dát, nízkoúrovňových operáciách a prevádzaných akciách. Tento jazyk nie je závislý na zdrojovom kóde a ani na cieľovej architektúre.

LLVM IR obsahuje nekonečne mnoho typovaných virtuálnych registrov, ktoré môžu obsahovať hodnoty základných typov (*boolean, integer, float, pointer*). Tieto registre sú SSA (Static Single Assignment) [8], teda do každého registru môže byť priradená hodnota iba raz. Čo značne uľahčuje analýzu dátového toku. Opakovaný zápis je riešený špeciálnou inštrukciou *phi* (implementuje φ uzol SSA grafu). Na prenos medzi pamäťou a registrami sa využívajú inštrukcie *load* a *store* s operátormi typu ukazovateľ (*pointer*). Väčšina inštrukcií je 3-adresných (2 zdrojové operandy a 1 cieľový pre výslednú hodnotu). Je to jazyk nízkej úrovne, ale je veľmi obsiahly a s platformovo nezávislým dizajnom. Jeho ukážku môžeme vidieť na obrázku 2.2.
Tieto vlastnosti sú veľmi dobre využité vo vnútornej reprezentácii dekompilátoru a preto bol tento jazyk vybraný ako vnútorná reprezentácia pre dekompilátor. Detailný syntax a sémantika jazyka LLVM IR je definovaný v referenčnom manuáli [5].

```
Loop:              ; Nekonečný cyklus počítajúci od 0
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

Obrázok 2.2 – Ukážka kódu LLVM IR (prevzaté z [5])

2.1.2 Front-end

Hlavnou úlohou tejto časti je platformovo závislý binárny kód transformovať na sekvenciu príkazov vnútornej reprezentácie LLVM IR (kapitola 2.1.1). Jedná sa o niekoľko krokov. Najprv sa jednotlivé platformovo závisle binárne kódy konvertujú na jednotnú reprezentáciu. Tento interný objektový formát CCOFF formát súborov je vrátane konverzných algoritmov navrhnutý práve pre tento účel. V súčasnosti podporuje konverziu z formátov súborov Windows PE, Unix ELF, Symbian E32, a Android DEX. Následne prebieha extrakcia sémantiky, potrebná pre namapovanie inštrukčnej sémantiky LLVM IR. Potom sa generuje inštrukčný dekodér, ktorý vytvorí reprezentáciu v postupnosti LLVM IR inštrukcií. Vo finálnej fáze môže dôjsť k úprave LLVM IR kódu.

2.1.3 Middle-end

Vstupom tejto časti je LLVM IR kód. Táto časť má za úlohu vylepšiť tento kód a pripraviť ho pre export do vyššieho programovacieho jazyka. Jedná sa napr. o nahradenie časti kódov lepšie čitateľnými a pochopiteľnejšími inštrukciami, vyhľadanie a rozpoznanie konštánt, výrazov, pokročilejších štruktúr ako cykly, podmienky a niekoľko ďalších optimalizácii. Výstupom tejto časti je optimalizovaný LLVM IR kód.

2.1.4 Back-end

Pre nás najzaujímavejšia časť je transformácia LLVM IR kódu do vyššieho programovacieho jazyka.

Postupnosť akcií v zadnej časti dekompilátora je nasledujúca: Prevedie sa transformácia LLVM IR na našu vlastnú vnútornú reprezentáciu, nad ňou sa prevedú ďalšie optimalizácie a prevedie sa výpis do výstupného súboru (už vo vyššom programovacom jazyku).

Vlastná vnútorná reprezentácia back-endu

V zadnej časti sa ako posledný krok pred výstupom do vyššieho jazyka využíva vlastná vnútorná reprezentácia zadnej časti (ďalej ako IR back-endu). Táto reprezentácia slúži na prevod LLVM IR inštrukcií do interpretácie, ktorá je reprezentovaná abstraktným syntaktickým stromom. Dôležité je aby umožňovala čo najlepšiu optimalizáciu a čo najjednoduchší výstup výstupného jazyka.

Medzi optimalizácie obsiahnuté v back-ende patria napr. odstránenia priradenia výrazu samého sebe, konverzia prázdnych polí na prázdne reťazce, odstránenie „mŕtveho“ kódu, konverzie globálnych premenných na lokálne, odstránenie zbytočných postupností priradení, konverzia cyklov a podmienok na výhodnejšie formy, odstránenie prebytočných zátvoriek a iných.

Predstavme si niekoľko vybratých konverzií medzi LLVM IR hodnotami a IR back-endu. Hodnoty, konštanty, ukazovatele, *load* inštrukcie, unárne operátory, binárne operátory, porovnania sú konvertované na výrazy vlastnej IR back-endu. *Store* inštrukcia je konvertovaná na príkaz priradenia. Ďalej nesmieme zabudnúť na prevody dátových typov, volania funkcií a rôzne inicializácie.

Existujúci výstupný jazyk

V súčasnosti je možný prevod do jazyka Python (*http://212.4.138.237/trac/wiki/BackendHOWTO*), ktorý je ale mierne upravený. Je to blokovo štruktúrovaný interpretovaný objektovo orientovaný skriptovací jazyk so silnou typovou kontrolou. Na rozdiel od kompilovaných jazykov, typová kontrola prebieha až za behu programu. Oproti bežnému Pythonu sa v tomto jazyku vyskytujú konštrukcie, ktoré sa vyskytujú v jazyku C. Jedná sa o ukazovatele, dereferencie, získavanie adresy, príkaz *switch* (vetvenie na základe výrazu), označenie parametrov tromi bodkami vo funkciách s premenným počtom parametrov. Časť funkcionality z jazyka C je nahradená inými konštrukciami. Konkrétne namiesto ternárneho operátora sa využíva koštrukcia *if/else*, polia sú nahradené zoznamami a namiesto štruktúr sa využívajú slovníky. Cyklus *for* je použitý z jazyka Python s volaním funkcie *range(from, to+1, step)* namiesto cyklu známeho z jazyka C. Taktiež tento jazyk disponuje štandardným C príkazom *goto*.

3 Pridanie výstupu do jazyka C

Informácie, z ktorých vychádzam v tejto kapitole pochádzajú z internej dokumentácie projektu dekompilátoru Lissom [7] a zo zdrojových kódov súčasnej implementácie dekompilátoru.

Na tomto mieste je nutné spomenúť i existujúci LLVM C back-end (*http://llvm.org/*), ktorý by mohol mylne navodiť pocit, že vyvíjame existujúce. V skutočnosti je prístup k návrhu odlišný. Náš vyvíjaný C back-end v rámci všeobecného dekompilátoru má za cieľ generovať oveľa jednoduchší cieľový kód v jazyku C, namiesto skokových inštrukcií využívať riadiace štruktúry, zbaviť sa zbytočných pretypovaní, vynechať všetok nepotrebný kód a iné.

3.1 Pridanie nového jazyka

Dekompilátor projektu Lissom je na pridanie nového jazyka veľmi dobre prispôsobený, nakoľko celý dekompilátor je navrhnutý tak, aby jeho jednotlivé časti boli na sebe čo najmenej závislé. Súčasnú verziu dekompilátoru môžeme rozšíriť o nový jazyk pridaním nového generátoru cieľového kódu a samozrejme rozšírením vnútornej reprezentácie o časti, v ktorých chýba podpora. Prípadne doriešiť interné záležitosti jazyka.

Ako vzor pre pridanie jazyka C môže slúžiť aktuálny generátor do modifikovaného jazyka Pythonu, ktorý už dokonca obsahuje niektoré štruktúry jazyka C (kapitola 2.1.4). Spolu s doplnením podpory v oblastiach, kde chýba v aktuálnom riešení podpora, môžeme dosiahnuť pridanie jazyka C do súčasnej implementácie dekompilátoru.

V aktuálnom riešení sa vyskytujú konštrukcie, ktoré sa oproti jazyku C líšia minimálne. Jedná sa konkrétne o členenie do blokov, podpora polí, štruktúrované dátové typy, podpora ternárneho operátora alebo riadiacich konštrukcií, ktoré sa líšia len drobnými rozdielmi v syntaxi. Na druhej strane nájdeme i niekoľko vlastností jazyka C, ktorých podpora sa musí doplniť.

3.2 Oblasti, pre ktoré chýba v aktuálnom riešení podpora

Jazyk C má oproti jazyku Python isté odlišnosti, pre ktoré chýba podpora. V tejto kapitole sa pokúsim tieto oblasti vytipovať a zdôvodniť.

3.2.1 Informácie o typoch a pretypovaní

V jazyku Python nie je nutné premenné deklarovať lebo každá premenná sa chápe ako odkaz na objekt. Z toho dôvodu v súčasnom riešení chýbajú vo vlastnej vnútornej reprezentácii informácie o type konštánt a premenných.

V LLVM IR dekompilátoru Lissom sa generujú celočíselné typy, typy s pohyblivou rádovou čiarkou, ukazovateľom, poľom a štruktúrou. LLVM podporuje aj ďalšie typy ako napr. vektor, ale tie sa v dekompilátore negenerujú. Bude potrebné pridať konverziu týchto typov na vnútornú reprezentáciu back-endu.

Tiež bude nutné zabezpečiť pretypovanie v opodstatnených prípadoch. V LLVM je generovaných niekoľko inštrukcií, ktoré sú v súčasnej implementácií ignorované, ale v jazyku C zodpovedajú rôznym dátovým pretypovaniam.

Doplnenie tejto podpory nesmie nijako narušiť výstup jazyka Python a musí byť čo najviac nezávislý na konkrétnom výstupnom jazyku. Nebudeme napríklad vytvárať celočíselne typy bitových šírok, tak ako ich poznáme v jazyku C, ale vo vnútornej reprezentácii back-endu budeme reprezentovať typy ľubovoľnej bitovej šírky.

3.2.2 Hlavičkové súbory externých knižníc

V jazyku C je potrebné v prípade používania funkcií z externých knižníc, pripájať hlavičkové súbory týchto knižníc pomocou konštrukcie *#include <nazov.h>*. Bude nutné zabezpečiť, aby sa zoznam týchto hlavičiek generoval vždy na začiatku výstupného C súbor. Je dôležité, aby sa generovali len tie hlavičkové súbory, ktoré sú naozaj potrebné.

3.2.3 Deklarácie funkcií

V jazyku C sa môže vyskytnúť, že sa funkcia volá skôr ako je definovaná. V niektorých prípadoch sa to dá vyriešiť zmenou poradia funkcií, ale ani to nemusí stačiť. Funkcie sa môžu volať napríklad i navzájom, tak, že *funkcia A* volá *funkciu B* a *funkcia B* volá *funkciu A*. V tomto prípade je potrebné deklarovať funkcie pred ich použitím. Obvyklé sa generujú pred všetkými definíciami funkcií.

4 Návrh pridania podpory jazyka C

Tvrdenia v tejto kapitole vychádzajú z týchto zdrojov [3], [5] a [7].

V rámci zadnej časti aktuálneho riešenia funguje generovanie výstupu na nasledujúcom princípe: Prechádza sa vlastná vnútorná reprezentácia stromovej štruktúry, jej postupným prechodom sa vygeneruje príslušný výstupný text. Jedná sa o akýsi postupný proces, kde sa prechádza od všeobecnej reprezentácie ku stále konkrétnejšej. Napríklad máme výraz, ktorý je binárny, teda má 1 operátor a 2 operandy, ale jednotlivé operandy sú opäť výrazy, ktoré môžu reprezentovať napr. konštanty, premenné alebo ďalšie operandy s príslušným operátorom. Každý výraz sa takto rozgeneruje až do reprezentácie, ktorá je vyvedená na výstup. Tohto princípu sa neupustíme ani pri návrhu zadnej časti pre jazyk C. Rozdiel bude len vo výslednej textovej reprezentácii u prevzatých konštrukcií a do tejto reprezentácie (stromovej štruktúry) pridáme ďalšie vetvy reprezentujúce nové prvky jazyka C, ktorých návrhu sa týka táto kapitola.

4.1 Získanie hlavičkových súborov k funkciám

Vzhľadom k tomu, že vo vnútornej reprezentácii, či už v LLVM IR alebo vlastnej reprezentácii nemáme žiadne informácie o pripájaných knižniciach, tak nemáme inú možnosť ako k tomu využiť zoznam externých (neužívateľských) funkcií. Podľa názvu týchto funkcií získame názov knižnice, ku ktorej patrí.

Objavuje sa problém ako túto príslušnosť zistiť s platformovou nezávislosťou. Naskytuje sa nám možnosť vytvoriť databázu, kde bude spárovaná funkcia s príslušnou knižnicou. Je to riešenie nepraktické v tom, že táto databáza musí byť vytváraná a aktualizovaná ručne.

4.2 Generovanie prototypov funkcií

Pre zaručenie správneho generovania deklarácií funkcií bude najlepšie vygenerovať prototypy (deklarácie) všetkých užívateľských funkcií, aby sme sa vyhli stavu, že niektorá funkcia bude použitá skôr ako bola definovaná/deklarovaná. Prototyp funkcie obsahuje názov funkcie, návratový typ, parametre funkcie s príslušnými typmi a je ukončený bodkočiarkou. Napr. *int sucet(int a, int b);* je prototyp funkcie s názvom *sucet*, dvoma celočíselnými parametrami *a* a *b*, ktorá vracia celočíselný výsledok.

Umiestnenie prototypov je nutné pred definíciami všetkých funkcií, najlepšie bezprostredne pred definíciami (z dôvodu užívateľských typov, ktoré môžu byť v deklarácii použité a musia byť definované pred prototypmi funkcií).

4.3 Návrh dátových typov

Dátové typy sa v jazyku C vyskytujú na niekoľkých miestach. Svoje typy majú premenné, návratové hodnoty funkcií, parametre funkcií, zložky užívateľských štruktúrovaných dátových typov. Vzhľadom k tomu, že v aktuálnom riešení podpora typov chýba, tak sa jedná asi o najväčšiu prekážku v ceste k výstupu v jazyku C.

LLVM IR nám ponúka nasledujúce dátové typy:

- `void` – dátový typ veľkosti 0.
- Celočíselné: `i1, i2, i3, ..., i8, ..., i16, ..., i32, ..., i64, ...`
  (kde `i` označuje celočíselný typ a číslo reprezentuje počet bitov)
- S pohyblivou rádovou čiarkou: `half` (v C reprezentovaný ako typ *short*)(16), `float`(32), `double`(64), `x86_fp80`(80), `fp128`(128), `ppc_fp128`(128)
  (v zátvorkách sú informácie o veľkosti v bitoch, `fp128` a `ppc_fp128` sa líšia vo veľkosti mantisy, u `fp128` je 112 bitová a u `ppc_fp128` je 64 bitová mantisa)
- Polia
- Štruktúry
- Funkcie – tento dátový typ obsahuje typ návratovej hodnoty a parametrov
- Ukazovateľ
- Ďalšie: `vector, opaque, ...`

Dátové typy LLVM IR sa pomerne dobre prekrývajú s dátovými typmi v jazyku C. V nasledujúcich podkapitolách si predstavíme vznikajúce problémy jednotlivých dátových typov a navrhneme riešenie.

4.3.1 Celočíselné dátové typy

V jazyku C nie je pevne určená veľkosť celočíselných typov a navyše jazyk C nepodporuje celočíselné typy akejkoľvek bitovej šírky. Nehladiac na to budeme vo vlastnej vnútornej reprezentácie celočíselný dátový typ reprezentovať jednotne s veľkosťou určenou v bitoch a vo výstupe takto i reprezentovať. Ako ďalší ešte závažnejší problém je, že LLVM IR nám neposkytuje informácie či je číslo znamienkové alebo bezznamienkové. V súčasnom kontexte nenachádzam komplexné riešenie tohto problému na úrovni back-endu. Vo vyvíjanom riešení som sa rozhodol, že budem všetky celočíselné typy reprezentovať ako bezznamienkové, čo môže viesť v niektorých prípadoch (napr. porovnávanie dvoch čísel) k nekonzistencii vstupu a výstupu dekompilátoru.

4.3.2 Dátové typy s pohyblivou rádovou čiarkou

Naskytuje sa nám problém hlavne s typmi väčšej bitovej šírky ako podporuje jazyk C. Nakoľko sa tieto typy líšia okrem veľkosti v bitoch i veľkosťou mantisy, tak v IR back-endu budeme musieť udržiavať obe informácie. A tieto informácie v implementácii preniesť i do výstupu.

4.3.3 Dátový typ pole

Tento typ musí obsahovať typ položiek poľa a okrem toho samozrejme i počet týchto položiek. Polia navyše môžu byť viacrozmerné. Všetky tieto informácie je potrebné udržiavať. V rámci jazyka C je pri deklarácií potrebné poznať typ, rozmery a počty prvkov jednotlivých rozmerov. V použití pri prístupe ku konkrétnemu prvku, už tieto informácie potrebné nie sú, ale je stále nutné poznať počet rozmerov. Preto musí byť reprezentácia tohto typu čo najvšeobecnejšia s možnosťami získať všetky potrebné informácie.
Syntax polí v LLVM IR je `[<# elements> x <elementtype>]`. Viacrozmerné pole môže vyzerať napríklad takto: `[3 x [4 x i32]]`, vidíme, že nám v tejto reprezentácií nechýba žiadna z potrebných informácií.

4.3.4 Dátový typ štruktúra

Štruktúry z pohľadu jazyka C sú užívateľom definované dátové typy, ktorých jednotlivé zložky môžu mať rôzne dátové typy. Definujú sa kľúčovým slovom `struct` a príslušným menom štruktúry. Oproti poliam je nutné si v rámci štruktúr v IR back-endu udržiavať i názov štruktúry, typy jednotlivých zložiek štruktúr, názvy týchto zložiek. Navyše každý štruktúrovaný dátový typ musí byť pred použitím definovaný, takže sa hneď ukazuje potreba si udržiavať zoznam všetkých týchto štruktúr.

V rámci LLVM IR je syntax dátového typu štruktúra `type { <type list> }`. Hneď si všimneme, že sa tu nevyskytuje ani názov štruktúry a dokonca ani názvy zložiek. Tieto názvy preto musia byť vygenerované a keďže generované názvy nemajú žiadnu výpovednú hodnotu, tak som sa rozhodol, že bude vhodné zrušiť identické štruktúry (rovnaký počet, poradie a typy jednotlivých zložiek) a ponechať iba jednu reprezentáciu.

4.3.5 Ostatné typy

Aktuálne riešenie obsahuje podporu ukazovateľov tak ako ju poznáme z jazyka C, podobne je to i s funkciami (návratovým typom). Túto podporu je nutné len doplniť o správne typy.
U prázdneho typu `void` sa neobjavujú žiadne problémy. Ostatné typy ako `vector`, `opaque` sa v jazyku C nevyskytujú, takže ich implementáciu riešiť nebudem. V súčasnosti sa tieto typy v rámci dekomplilátoru negenerujú, ale ak by sa objavili v jazyku LLVM IR, tak budú reprezentované ako neznámy typ popisovaný v kapitole 5.2.4.

4.4 Pretypovanie

V LLVM IR sa nachádza niekoľko inštrukcií, ktoré slúžia k pretypovaniu hodnoty jedného typu na hodnotu iného typu.

Jedná sa konkrétne o tieto inštrukcie:

- *trunc* – konverzia väčšieho celočíselného typu na menší
```
%X = trunc i32 257 to i8            ; i8:1
```
- *zext* – Rozšírenie celočíselného dátového typu pridaním nulových bitov na začiatok
```
%X = zext i32 257 to i64            ; i64:257
```
- *sext* – Rozšírenie celočíselného dátového typu rozkopírovaním znamienkového bitu
```
%X = sext i8  -1 to i16             ; i16:65535
```
- *fptrunc* – konverzia väčšieho typu čísla s pohyblivou rádovou čiarkou na menší
```
%X = fptrunc double 123.0 to float  ; float:123.0
```
- *fpext* – Konverzia väčšieho typu čísla s pohyblivou rádovou čiarkou na menší
```
%X = fpext float 3.125 to double    ; double:3.125000e+00
```
- *fptoui* - Konverzia typu čísla s pohyblivou rádovou čiarkou na celočíselný bezznamienkový typ
```
%X = fptoui double 123.0 to i32      ; i32:123
```
- *fptosi* - Konverzia typu čísla s pohyblivou rádovou čiarkou na celočíselný znamienkový typ
```
%X = fptosi double -123.0 to i32      ; i32:-123
```
- *uitofp* – Konverzia celočíselný bezznamienkového dátového typu na typ čísla s pohyblivou rádovou čiarkou
```
%X = uitofp i32 257 to float         ; float:257.0
```
- *sitofp* – Konverzia celočíselný znamienkového dátového typu na typ čísla s pohyblivou rádovou čiarkou
```
%X = sitofp i32 257 to float         ; float:257.0
```
- *ptrtoint* – Konverzia celočíselného typu na ukazovateľ
```
%X = ptrtoint i32* %P to i8
```
- *inttoptr* – Konverzia ukazovateľu na celočíselný typ
```
%X = inttoptr i32 255 to i32*
```
- *bitcast*– Konvertuje jeden dátový typ na druhý bez zmeny bitov
```
%Y = bitcast i32* %x to sint*
```

Tieto konverzie dátových typov sa budú vyskytovať vo výrazoch, preto je nutné ich zaradiť do vnútornej reprezentácie back-endu, tak aby mohli vystupovať vo výrazoch.

U inštrukcií, ktoré pracujú so znamienkom celočíselného typu, som uvažoval nad spôsobom, či by to nemohlo pomôcť vo vyriešení problému znamienkových a neznamienkových typov. Tieto inštrukcie sa ale nevyskytujú vždy nad každou premennou, takže to tento problém nerieši.

5 Implementácia

Pred samotnou implementáciou generátoru výstupu v jazyku C, musíme najprv pridať potrebnú podporu do vnútornej reprezentácie backendu, preto i táto kapitola bude členená tak, že najprv popíšem zaujímavé časti ako som implementoval chýbajúcu podporu a nakoniec generátor samotného textového výstupu.

5.1 Databáza hlavičkových súborov

Ako som spomínal v návrhu, tak som implementoval databázu funkcií, ku ktorým sú priradené ich hlavičkové súbory. Trieda sa nazýva *CHeaders*, a v konštruktore sa plní štandardná C++ *std::map*, kde na každú funkciu je namapovaný jej príslušný hlavičkový súbor. Spracoval som databázu štandardnej knižnice jazyka C (1990 ISO), konkrétne hlavičkové súbory: *assert.h*, *ctype.h*, *locale.h*, *math.h*, *setjmp.h*, *signal.h*, *stdarg.h*, *stddef.h*, *stdio.h*, *stdlib.h*, *string.h*, *time.h.* Rozšírenie tejto databázy je jednoduché a to doplnením ďalších funkcií s príslušnými hlavičkovými súbormi.

Táto trieda obsahuje metódu *getHeader()*, ktorá príjme parameter ako reťazec názvu funkcie a vráti reťazec s názvom príslušného hlavičkového súboru. Každý hlavičkový súbor bude generovaný iba raz aj v prípade, že by sa objavovalo viacero funkcií z jedného hlavičkového súboru.

5.2 Reprezentácia dátových typov

Všetky dátové typy sú implementované ako triedy s postfixom *Type, konkrétne *IntType*, *FloatType*, *VoidType*, *TArrayType*, *TStructType*, *PointerType* a *UnknownType*. Všetký sú dedené z triedy *Type*. Inštancovanie týchto tried prebieha prevažne v module *LLVMConverter*, kde sa v metóde *llvmTypeToType()* konvertujú LLVM typy na typy vnútornej reprezentácie back-endu, ktoré sú reprezentované práve objektmi spomenutých tried.

5.2.1 Spoločné črty implementácie reprezentácie typov

Všetky typy využívajú návrhový vzor „singleton“, ktorý zabezpečuje aby sa vytvorila vždy maximálne jedna inštancia danej triedy. V implementácii typov, ktoré majú napríklad rôzne veľkosti, tak sa vytvára jedna inštancia každého objektu s konkrétnymi atribútmi. Teda napríklad jeden celočíselný *IntType* s veľkosťou 8 bitov, jeden s veľkosťou 32 bitov a pod. Tento prístup z jednej strany šetrí čas a pamäť a na druhej strane udržuje zoznam všetkých inštanciovaných variant konkrétneho typu, čo napríklad u štruktúr aj využijeme.

Každá z tried ma metódu na získanie veľkosti dátového typu v bitoch, u *FloatType* i pre získanie mantisy.

5.2.2 Typy s pohyblivou rádovou čiarkou

Na tomto type je najzaujímavejšie vytváranie len jednej inštancie typu s príslušnou veľkosťou. Komplikácia sa týka hlavne dvoch 128 bitových LLVM typov *fp128* a *ppc_fp128*. Teda veľkosť nám nestačí ako charakteristika jedného typu. Preto ako charakteristický atribút som použil súčet bitov veľkosti a mantisy.

5.2.3 Polia a štruktúry

Polia a štruktúry patria medzi najzaujímavejšie z typov. Polia vyžadujú pri vytvorení inštancie typ prvkov, už ako objekt vnútornej reprezentácie back-endu, nie LLVM IR. Rovnako vyžaduje i zoznam rozmerov s počtami prvkov. Ten je reprezentovaný kontajnerom C++ *std::vector*. Na získavanie týchto informácii som implementoval i príslušné metódy. Viac v [7].

Štruktúry majú zložky, ktoré sú reprezentované podobne ako rozmery u poľa vektorom. Tento vektor obsahuje typy jednotlivých zložiek. Môžeme si to predstaviť ako keby boli všetky zložky usporiadané za sebou a číslovane napr. od jednotky. Každá štruktúra môže mať meno, takže obsahuje verejné metódy na nastavenie i získanie mena. Štruktúry sú inštanciované konkrétne v 2 mapách *std::map*. V jednej sú inštancie podľa kľúča ako vektoru typov zložiek. To nám zaručuje i odstránenie duplicitných štruktúr ako bolo spomenuté v závere podkapitoly 4.3.4. Do druhej mapy sa ukladajú ako kľúč názvy štruktúr, na ktoré sa mapujú kľúče prvej mapy. Nakoľko sú názvy štruktúr generované tak to zaručuje, že i v tejto mape budú uložené v poradí ako vznikali. Poslednou zaujímavosťou je, že ak vytvoríme inštanciu a predáme prázdny vektor zložiek štruktúry, tak získame akurát prístup k mapám všetkých existujúcich inštancií štruktúr. Tieto vlastnosti využijeme v textovom výstupe do jazyka C, keď budeme potrebovať vypísať definície všetkých štruktúr v správnom poradí. Jedná sa len o dočasné riešenie, ktoré bude potrebné nahradiť elegantnejším spôsobom.

5.2.4 Neznámy typ

Trieda *UnknownType* nereprezentuje žiaden konkrétny typ, skôr naopak, reprezentuje prípadné typy, pre ktoré v súčasnej implementácii nebude podpora. Má ešte jeden význam pri vytváraní premenných v metódach, kde nemáme informáciu o type premennej. Vytvárame teda premenné tohto typu a potom v miestach, kde už typ premennej získať vieme, ho nahradíme správnym typom.

5.3 Výrazy pretypovania

Inštrukcie pretypovania LLVM IR sú na výrazy pretypovania vnútornej reprezentácie prevádzané v module LLVMConverter konkrétne v dvoch metódach kde sme implementovali vytváranie inštancií príslušných výrazov pretypovania: *visitCastInst()* a *llvmConstantToExpression().* Triedy týchto

výrazov majú ekvivalentné názvy príslušným inštrukciám s postfixom **Expr*, konkrétne *FPExtExpr, FPtoSIExpr, FPtoUIExpr, FPTruncExpr, IntToPtrExpr, PtrToIntExpr, SExtExpr, SItoFPExpr, TruncExp, UItoFPExpr, ZExtExpr a CastExpr (*reprezentuje inštrukciu *bitcast).* Konkrétny popis významu inštrukcií v LLVM IR, vrátane príkladov sa nachádza v kapitole 4.4.

Implementácia týchto výrazov je pomerne jednoduchá a priamočiara. Všetky triedy pretypovaní dedia z triedy *UnaryOpExpr*, ktorá reprezentuje výrazy s jedným operandom. Okrem operandu pri vytváraní potrebujú cieľový typ pretypovania. Triedy *ZExtExpr*, *SExtExpr*, *UItoFPExpr* a *SItoFPExpr* ešte navyše vyžadujú i zdrojový typ, ktorého prítomnosť je v súčasnej implementácií iba dočasná a jednoúčelová. Využíva sa len pre potreby textového výstupu v jazyku C kvôli zabezpečeniu správnej funkcie pretypovaní vzhľadom na už niekoľkokrát diskutovaný problém so znamienkovými a bezznamienkovými celočíselnými typmi a tým, že v aktuálnej verzii sú všetky typy generované ako bezznamienkové. Z toho vyplýva možnosť výskytu niektorých problémov.

Uveďme si príklad: Majme výraz: *(a > -1).* Ak by *a* bolo znamienkové číslo, tak môže byť výraz vyhodnotený ako pravda (*true*) i ako nepravda (*false*). Ak budeme číslo reprezentovať ako bezznamienkové, tak tento výraz by bol vždy pravdivý (*true*).

Zdrojový typ sa využíva pri konverzii medzi znamienkovými či bezznamienkovými typmi. Bližšie to rozoberiem v kapitole 5.4.7.

5.4 Generátor výstupného kódu v jazyku C

Úlohou tejto podkapitoly je popísať zaujímavé pasáže implementácie *CHLLWriter*, jedná sa o modul ktorého úlohou, je vykonať textový výstup vnútornej reprezentácie back-endu a vytvoriť zdrojový kód v jazyku C.

5.4.1 Spracovanie výrazov, návrhový vzor návštevník

V tejto chvíli je čas spomenúť, že sa v tejto implementácii využíva návrhový vzor *Návštevník* (*Visitor*) [9], ktorý slúži k oddeleniu algoritmov od objektovej štruktúry na ktorej pracuje. Využíva sa hlavne pre spracovanie a výstup výrazov ale i iných konštrukcií. Pre ďalšie pochopenie textu tejto práce stačí si predstaviť, že existuje nejaká metóda *accept()*, ktorá sa postará o navštívenie správnych metód až kým sa nedocieli korektný výstup.

Príklad:

Máme ukážkovu *visit* metódu z CHLLWriter:

```
void CHLLWriter::visit(ShPtr<AddOpExpr> expr) {
	out << "(";
	expr->getFirstOperand()->accept(this);
	out << " + ";
	expr->getSecondOperand()->accept(this);
	out << ")";
}
```

Majme výraz expr = ((1+2)+3), samozrejme je to vo vnútornej reprezentácii backendu, ale pre pochopenie funkčnosti na tom nezáleží.

Postup vyhodnotenia a výstupu výrazu:

1. Pri zavolaní metódy `visit(ShPtr<AddOpExpr> expr)` sa vypíše 1. zátvorka „("
2. getFirstOperand vráti (1+2), co je opäť výraz typu `ShPtr<AddOpExpr>`,
   nad ktorým sa volá metóda *accept()*
3. metóda *accept()* zabezpečí volanie metódy `visit(ShPtr<AddOpExpr> expr)`
4. Vytlačí sa opäť zátvorka „(" na výstupe už je „(("
5. `getFirstOperand` nám zrejme vráti konštantu typu `ShPtr<ConstInt>`,
   nad ktorým sa volá metóda accept()
6. Volá sa nasledujúca metóda, ktorá získa a vypíše hodnotu konštanty – výstup „((1"
   ```
   void CHLLWriter::visit(ShPtr<ConstInt> constant) {
          out << constant->getValue();
   }
   ```
7. Postupne sa takto spracuje celý výraz až na výstup sa dostane ((1+2)+3)

Treba si uvedomiť že metóda *accept()* nám zabezpečí, že sa bude volať presne tá metóda, ktorú požaduje výraz, bez akéhokoľvek upresňovania programátorom. Na tomto princípe funguje celé generovanie nielen výrazov v *CHLLWriter*, ale i v ostatných moduloch v rámci dekompilátoru.

5.4.2 Hlavičkové súbory, užívateľské typy, globálne premenné

Na začiatku C súboru musíme vygenerovať príslušné hlavičkové súbory. To získame tým, že si vytvoríme modul triedy *CHeaders*, z modulu *Module* získame príslušnou metódou jednotlivé funkcie. Následne zabezpečíme použitím vhodného C++ kontaineru (*std::set*) odstránenie duplicít. Pridáme hlavičky *stdint.h* a *stdbool.h*, ktoré sa budú generovať v každom výstupnom súbore. Prevedieme zápis všetkých potrebných hlavičiek do výstupu.

V hlavičke C súboru ďalej generujeme definície vlastných typov s pohyblivou rádovou čiarkou. Jedná sa o zlepšenie čitateľnosti výstupu, keď v názve súboru budeme vidieť i bitovú šírku a navyše sa jedná o dočasné riešenie nepodporovaných 128 bitových typov v jazyku C, ktoré aktuálne nahrádzame *long double* typom. Konkrétne teda pod vlastným typom *float32_t* sa rozumie typ *float*, pod *float64_t* typ *double* a *float80_t*, *float128_t*, *float128PPC_t* zodpovedajú typu *long double*.

V súvislosti s typmi nasledujú prípadne definície štruktúrovaných dátových typov. Tu sa využíva to, že si vlastne vytvoríme prázdny objekt typu *TStructType* (kapitola 5.2.3), ten nám umožní získať mapy všetkých inštancií štruktúr. Tieto mapy obsahujú názvy štruktúr a príslušné typy zložiek.

Týmto zložkám generujeme názvy tvaru *e#*, kde # je poradie ako je zložka uložená vo vektore typov zložiek. Tieto názvy sa správne generujú i pri prístupe k zložkám štruktúr vo výrazoch.

Globálne premenné sa generujú ako ostatné premenné ale s malým rozdielom. Globálne premenné sa vždy generujú ako ukazovatele [7]. Jednu úroveň ukazovateľov treba odfiltrovať a ako typ premennej berieme až v ukazovateli obsiahnutý typ.

5.4.3 Prototypy funkcií a externé funkcie

Pre výstup prototypov funkcií využívame metódu `visit(ShPtr<Function> func)`, v ktorej zisťujeme metodou *isDeclaration()* (vráti *true* ak funkcia nemá telo) či sa jedná o deklaráciu funkcie. Ak áno, voláme funkciu *emitFunctionPrototype()*, ktorá sa postará o výstup deklarácie funkcie.

Externé funkcie generujeme zakomentované aj napriek tomu, že ich nepotrebujeme. Využitie majú v prípade chýbajúcich hlavičkových súborov (napr. z dôvodu že sa nenachádzajú v databáze *CHeader*), kedy sa nám nájdenie problematickej funkcie podarí v oveľa kratšom čase a s väčším komfortom.

5.4.4 Bloky, riadiace štruktúry a výrazy

Implementácia týchto časti je vo väčšej miere prevzatá z implementácie riešenia výstupu do jazyka Python. Zmena je len v syntaxe. Čo sa týka blokov príkazov, tak jedná sa len o pridanie zložených zátvoriek, cykly *for, while*, podmienka *if-else* a vetvenie príkazom *switch* upravíme rovnako len doplnením zátvoriek okolo výrazov v podmienkach a rôznymi drobnými úpravami aby sme dosiahli výstup definovaný normou jazyka C99 [1].

Z operátorov vo výrazoch za zmienku stojí *StructIndexOpExpr*, ktorý má úlohu v prístupe k zložkám štruktúry, kde generuje „*.e*“ a nasledované poradovým číslom zložky štruktúry. Tiež by som upozornil, že *not* operátor a *neg* operátor sú rozdielne operátory, *not* má v C význam „*!*“ a *neg* reprezentuje znamienko mínus „-“.

5.4.5 Deklarácie, definície a použitie premenných, funkcie

Premenné sú deklarované vždy na začiatku funkcie pomocou príkazu `VarDefStmt`. Tým pádom typ premennej musíme riešiť len v rámci tejto metódy. Táto deklarácia je nutná hlavne z hľadiska toho, že sa nemôže stať aby sa lokálna premenná bloku používala mimo blok, lebo všetky sú deklarované v rámci celej funkcie. Viacnásobná deklarácia premennej s rovnakým názvom nehrozí vzhľadom k metódam spracovania premenných v nižších úrovniach dekompilátoru. V ostatných prípadoch pri použití premenných sa generuje len názov premennej.

Funkcia resp. jej názov je tiež premenná. Funkcia sa ale javí ako typ teda či v prototype alebo v definícii funkcie musí ísť na výstup typ, názov funkcie a parametre vrátane svojich typov, ktoré musíme generovať v rámci funkcie, nakoľko premenná typ negeneruje. Medzi funkciami je ešte zaujímavá generovaná hlavná funkcia programu s názvom *main*, ktorá sa generuje vždy buď ako *int main(void)* alebo *int main(int argc, char *argv[])*. V prípade ak by sa parametre funkcie volali ináč,

tak budú ich názvy nastavené na tieto. Prevádzame to z dôvodu aby prekladač nehlásil zbytočne varovania a taktiež pre lepšiu čitateľnosť kódu.

5.4.6 Textová reprezentácia typov

Číselné typy

U celočíselných typov som sa v implementácii rozhodol pre využitie typov z hlavičkového súboru *stdint.h*. Nakoľko obsahuje pevne definované veľkosti typov v bitoch, ktoré nie je platformovo závislé. Teda textová reprezentácia celočíselného typu generovaná ako „*uint##_t*“, kde *uint* znamená celočíselný bezznamienkový dátový typ (všetky celočíselné typy sú generované ako bezznamienkové – kapitola 4.3.1), *##* reprezentuje číslo s veľkosťou dátového typu v bitoch. Samozrejme v *stdint.h* nie sú definované typy všetkých variant odpovedajúcim LLVM IR, ale zatiaľ to pre naše účely postačuje.

U typov s pohyblivou radovou čiarkou sme v požiadavke mať v názve veľkosť dátového typu, podobne ako u typov celých čísel, vytvorili vlastné dátové typy v textovom tvare „*float##_t*“ s identickým významom ako u celých čísel. Vlastné typy sa definujú len v prípade potreby.

Ukazovatele

Ukazovatele vypisujú toľko hviezdičiek „*“, koľkokrát je v obsiahnutom type (ukazovateľ je ukazovateľ na obsiahnutý typ) opäť ukazovateľ, nakoniec volá metódu *accept()* nad obsiahnutým typom.

Ostatné typy

Pole vypisuje len typ prvkov pola. Hranaté zátvorky sa generujú za premennými na miestach, kde sú premenné deklarované alebo definované, teda v zložkách štruktúr, v deklaráciách premenných a globálnych premenných. Pre prístup k prvkom sa využíva výraz *ArrayIndexOpExpr*, ktorý negeneruje len hranaté zátvorky, ale i číslo prvku.

U štruktúr sa jedná o presne identický prístup, generuje sa len slovo *struct* a názov štruktúry, prístup k jednotlivým položkám je prostredníctvom výrazu *StructIndexOpExpr.*

Prázdny dátový typ má textovú reprezentáciu *void.*

5.4.7 Výrazy pretypovania

Výrazy pretypovania z kapitoly 5.3 generujeme takmer všetky ako cieľový typ v guľatých zátvorkách. Rozdiel je vo výraze *TruncExpr* pri pretypovaní na *bool*, kde je potrebné ešte pre identickú funkčnosť s inštrukciou LLVM IR urobiť pred pretypovaním, binárny súčin s jednotkou. Tým pádom, či hodnota bude true alebo false záleží len na poslednom bite ako to vyžaduje inštrukcia trunc v LLVM IR, nie na celom čísle ako to je v jazyku C. (viac informácií v [1] a [5]).

Zaujímavý je problém inštrukcií, ktoré pracujú so znamienkovým a bezznamienkovým celočíselným typom. Ako som spomínal v predchádzajúcom texte, tak všetky celočíselné typy *IntType* sú na výstup generované ako bezznamienkové. V prípade znamienkového rozšírenia S*ExtExpr,* je nutné najprv pretypovať bezznamienkový typ na znamienkový. Musíme generovať dve pretypovania, prvé na rovnako veľký ale znamienkový typ a až potom na cieľový typ, ktorý bude u pretypovania *SExtExpr* tiež znamienkový. Ak by sme to nespravili pretypovanie by sa správalo ako rozšírenie s doplnením nulových bitov *ZExtExpr.* Presne ten istý postup sa uplatňuje i u pretypovania *SItoFPExpr*. Mohlo by sa zdať, že u bezznamienkových verzií toto potrebné nie je, ale oplatí sa to minimálne z dôvodu, aby bolo možné rýchlo zmeniť generovanie všetkých celočíselných typov v *CHLLWriter* na znamienkové.

6 Výsledky a vyhodnotenie

Táto kapitola sa zaoberá hlavne otestovaním súčasnej implementácie zadnej časti spätného prekladača produkujúceho kód v jazyku C a zhrnutia toho čo funguje a čo nie.

6.1 Sada testov

Na DVD médiu v prílohe je obsiahnutá sada vstupných testov a príslušných vygenerovaných výstupov v zložke `/tests`. V zložke `/tests/decompile-c` sú referenčné výstupy skompilované na binárny kód a následne dekompilované. Použitá architektúra je MIPS (prekladač *psp-gcc* z *pspsdk* prístupné na URL: *http://sourceforge.net/projects/minpspw/*) a optimalizácia –O2. V zložke `/tests/decompile-c-nobin` sú vstupy skompilované a dekompilované len cez LLVM IR bez prevodu cez binárny kód, ktorá obsahujú viac informácií ako napr. názvy premenných, konštrukcie sú podobnejšie vstupným súborom a pod. Z tohto dôvodu sú tieto výstupy lepšie čitateľné a lepšie porovnateľné.

6.2 Rozbor zaujímavých výsledkov testov

Celková sada obsahuje 20 testov, z ktorej sa generuje 20+20 výstupov. K vygenerovaniu nových výstupov je možné použiť priložený skript pre linuxový shell. V tejto kapitole vyberiem z testovacích výstupov najzaujímavejšie výsledky.

Test01: ukážka kompletného vstupného i výstupného súboru

Vstupný kód:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  printf("Hello world!");
  return EXIT_SUCCESS;
}
```

Výstupný kód:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

/* ------------ Functions --------------- */

int main(void) {
  printf("Hello world!");
  return 0;
}

/* ------- External Functions -------- */

// uint32_t printf(uint8_t * var1, ...);
```

Test03: vstavané funkcie *scanf* a *printf* vstup a výstup

Vstupný kód:

```
int a,b;
printf("Type 2 integer numbers: ");
scanf("%d %d", &a, &b);
printf("\n%d+%d=%d\n",a,b,a+b);
```

Výstupný kód:

```
printf("Type 2 integer numbers: ");
a = 0;
b = 0;
scanf("%d %d", &(a), &(b));
printf("\n%d+%d=%d\n", a, b, (b + a));
```

Vstup a výstup sú takmer úplne rovnaké. V ďalších testoch sa objavujú rôzne konštrukcie, ktoré sú veľmi podobné čo sa týka i zdroju i cieľa. Jedná sa konkrétne o cykly *for* alebo *while*, niekedy s ekvivalentnou zámenou cyklov (Test07). Veľmi podobné bývajú i konštrukcie ako *switch* alebo *if-else*.

Test04: ukazuje problém s reprezentáciou štandardného vstupu/výstupu/chybového výstupu, teda nedokáže reprezentovať *stdin/stdout/stderr*, nakoľko by mali byť typu *STREAM** či *FILE**.

Test05: V tomto teste si môžeme všimnúť zaujímavé správanie, kedy sa v hlavnej funkcii nevolá funkcia *factorial()*, ale je vložená ako *inline*.

Test06: Vo výstupe sú jednotlivé prvky definované po jednom, čo by bolo vhodné určite optimalizovať.

Test08: Tento test nám ponúka ukážku definície štruktúr, porovnajme vstupy a výstupy:

Vstupný kód:

```
struct date
{
  int day;
  int month;
  int year;
};

struct person
{
  char name[11];
  char surname[11];
  struct date birth;
};
```

Vstupný kód:

```
struct struct1
{
  uint32_t e0;
```

```
    uint32_t e1;
    uint32_t e2;
  };

  struct struct2
  {
    uint8_t e0[11];
    uint8_t e1[11];
    struct struct1 e2;
  };
```

Vidíme, že obe zápisy sú úplne ekvivalentné, rozdiely sú akurát v názvoch či štruktúr alebo zložiek. Žiadny problém nevzniká ani pri zanorovaní štruktúr či volaní ako parameter funkcie:

```
void printfamily(struct struct2 person);  //z výstupného súboru
```

V tomto teste sa nám objavuje ešte jeden nedostatok, a to, že po dekompilácii sa nám v kóde objavili funkcie, pre ktoré nemáme príslušné hlavičkové súbory. Teda bez úpravy nie je možné kód preložiť.

Test09: Tento test nám ukazuje jeden z nevyriešených problémov (totožný s problémom v Teste04), nedokáže interpretovať typ *FILE**, ktorý slúži na prácu so súborovým vstupom a výstupom. Namiesto tohto vznikajú štruktúry ukazovateľov, ktoré ale samozrejme nevyhovujú ako parametre funkcií. Tento dekompilovaný súbor bez úpravy nie je možne preložiť.

Test10: Demonštruje použitie niekoľkonásobných ukazovateľov.

Test11 a 12: Teraz sa už definované funkcie používajú v hlavnej funkcii. Vo funkciách je ponechaná priama i nepriama rekurzia i po dekompilácii.

Test14: Jedná sa o ďalší zaujímavý test, ktorý nám demonštruje problém, ktorý vyplýva z toho, že všetky čísla sú generované ako bezznamienkové.

Vstupný kód:

```
volatile int cmp=-12;
printf("THIS IS PROBLEM:\n");
if (cmp<0) printf("%d < 0 - true\n", cmp);
if (cmp>0) printf("%d < 0 - false\n", cmp);
```

Výstupný kód:

```
cmp = -12;
puts("THIS IS PROBLEM:");
if ( (cmp < 0) )
{
  printf("%d < 0 - true\n", cmp);
}
if ( (cmp > 0) )
{
  printf("%d < 0 - false\n", cmp);
}
```

Po preklade kódu nám pôvodný program vypíše:

```
-12 < 0 - true
```

Dekompilovaný a znovu preložený program vypíše:

```
-12 < 0 – false
```

V tomto prípade sa jedná o neekvivalentné programy, čo je spôsobené diskutovaným problémom znamienkových a bezznamienkových čísel.

Test16: Reprezentuje využitie viacrozmerných polí na vykreslenie Ascii Artu v okne terminálu. Pôvodný i dekompilovaný (varianta bez prevodu cez binárny kód) a znovu preložený program podáva rovnaké výsledky. Kód dekompilovaný z binárneho kódu sa asi zo všetkých výstupov testou od výstupu prevádzaného len zec LLVM IR reprezentáciu líši najviac. Obsahuje *goto* inštrukcie, pre ktoré v súčasnom riešení chýba podpora.

Test17: Úspešne reprezentuje prácu s dynamickou pamäťou a textom i po dekompilácii.
Testy ktoré neboli spomenuté doplňujú demonštráciu výstupu, ale neprinášajú nič prekvapivé.

6.3 Zhrnutie problémov funkčnosti

V rámci tejto práce sa nepodarilo vysporiadať s nasledujúcim problémami, ktoré vyplývajú z implementácie alebo boli zistené počas testovania. Niektoré chyby boli opravené, tieto spomínať nebudem.

Tieto ťažkosti v niektorých prípadoch zapríčiňujú nepreložiteľný alebo neekvivalentný kód:

- Problém bitových veľkostí celočíselných typov a typov s pohyblivou rádovou čiarkou: Hlavným dôvodom je chýbajúca reprezentácia typov ľubovoľnej bitovej šírky v jazyku C. (kapitola 4.3.1)
- Problém znamienkových a bezznamienkových dátových typov. V LLVM IR chýba podpora: Bližšie dôvody sú konzultované v kapitole 4.3.1.
- Problém s chybnou interpretáciou ukazovateľov na súbor alebo stream: *FILE *, STREAM *, stdio/stdin/stderr:* Dôvodom je že v LLVM IR je interpretácia týchto ukazateľov ako štruktúry. Možné riešenie by bolo detekovať tieto štruktúry a interpretovať ich ako samostatný typ. Je to určite námet na ďalšie vylepšenie back-endu.
- Výskyt funkcií mimo štandardnú knižnicu C (1990 ISO): Pre iné knižnice zatiaľ nemáme podporu. Nie je vybudovaná dostatočne rozsiahla databáza, ktorá by pokrývala všetky hlavičkové súbory. Súčasná databáza slúži hlavne pre demonštračné účely. (kapitola 5.1)
- Problém zanorenia štruktúry samej do seba, čo tiež bráni v úspešnom preklade kódu: V jazyku C [1] nie je možná napr. konštrukcia typu:

```
struct s1 {
  int e1;
  struct s1 e2;
```

```
}
```

Pre očakávanú funkčnosť je nutné využiť príkaz *typedef*, ktorý nám už správnu funkčnosť zabezpečí:

```
typedef struct s1 Ts1;
struct s1 {
  int e1;
  Ts1 e2;
}
```

Riešením do budúcnosti by bola integrácia príkazu *typedef* do definícií štruktúr a podobný prístup k definovaniu štruktúrovaných dátových typov pomocou *typedef*, aký využívame pri prototypoch funkcií.

- Chýbajúca funkčnosť nepodmieneného skoku *goto*: V čase implementácie chýbala podpora a riešenie tohto problému nebolo náplňou tejto práce.

Ak sa v kóde nevyskytuje žiaden z týchto problémov, neprejavil sa počas testovania žiaden problém s prekladom a správnou funkčnosťou.

Záver

Cieľom tejto bakalárskej práce bolo naštudovať, navrhnúť a implementovať riešenie integrovateľné do vyvíjaného dekompilátoru bez narušenia pôvodnej funkčnosti. V tejto práci boli naznačené výhody a využitie dekompilátoru, z čoho vyplýva i význam celej tejto práce a získanie výstupu v jazyku C, ktorý je ďaleko čitateľnejší ako napríklad jazyk symbolických inštrukcií.

Pre implementáciu zadnej časti bolo nutné si naštudovať princípy spätného inžinierstva, dekompilácie, oboznámiť sa s vnútornými reprezentáciami v rámci dekompilátoru, hlavne sa naučiť syntax LLVM IR a zorientovať sa vo vnútornej reprezentácie backendu, preskúmať a zorientovať sa v zdrojových kódoch celého dekompilátoru a hlavne preskúmať možnosti pridania ďalších oblastí, v ktorých chýbala podpora.

V rámci práce sa doplnila do vnútornej reprezentácie zadnej časti podpora dátových typov a výrazov pre pretypovanie, ktoré bude využiteľné i v ďalších výstupných jazykoch.

Vzhľadom k rôznym komplikáciám a nevyriešeným problémom sa nepodarila implementácia 100% funkčného riešenia, vyskytujú sa problémy, ktoré sa budú do budúcnosti musieť riešiť. Medzi ďalšie úlohy a vylepšenia určite patrí korektné určenie znamienkových a bezznamienkových dátových typov, správna reprezentácia i typov mimo štandardné bitové šírky typov jazyka C, zredukovanie zbytočných počtov pretypovaní, posun deklarácií premenných v kóde na miesto, ktoré je čo najbližšie k prvému použitiu. Vlastne vhodné je celkové postupné zjednodušovanie, optimalizovanie a ladenie výstupu tak, aby bol čo najjednoduchšie čitateľný, a zároveň preložiteľný s funkčnosťou identickou dekompilovanému zdrojovému spustiteľnému programu.

Vzhľadom k faktu, že tento dekompilátor Lissom umožňuje výstup v 2 jazykoch, tak sa dá počítať s využitím dekompilátoru, okrem iného, na migráciu medzi jazykom Python a jazykom C. Ako už bolo niekoľkokrát spomenuté tak vývoj dekompilátoru do budúcna predstavuje veľmi zaujímavé využitie i na analýzu potencionálne nebezpečného softvéru, ale aj v mnohých iných sférach informačných technológii.

Literatúra

[1] Draft normy jazyka C99 (ISO/IEC 9899:1999) [online]. 2005 [cit. 2012-05-14]. Dostupné na URL: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.

[2] Křoustek, J.: *Analýza a převod kódů do vyššího programovacího jazyka*, diplomová práce, Brno, FIT VUT v Brně, 2009.

[3] Eilam, E.: *Reversing: Secrets of Reverse Engineering.* Wiley, 2005. ISBN: 978-0-7645-7481-8.

[4] Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Hruška, T., Masařík, K., Meduna, A.: *Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis*, In: International Journal of Security and Its Applications, roč. 5, č. 4, 2011, Daejeon, KR, s. 91-106, ISSN 1738-9976

[5] Lattner, Ch., Adve, V.: *LLVM Language Reference Manual* [online]. 2012 [cit. 2012-05-14]. Dostupné na URL: http://www.llvm.org/docs/LangRef.html.

[6] Lattner, Ch., Adve, V.: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation* [online]. 2004. [cit. 2012-05-14]. Dostupné na URL: http://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf.

[7] Interná dokumentace projektu Lissom. [cit. 2012-05-15]

[8] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K.: *Efficiently computing static single assignment form and the control dependence graph,*Trans. Prog. Lang. and Sys., s. 13(4):451–490,1991.

[9] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns.* Addison-Wesley, 1994. ISBN: 0-201-63361-2.

Zoznam príloh

Príloha 1. DVD so zdrojovými kódmi implementácie, testovacími vstupmi a referenčnými výstupmi.