VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

STRUKTUROVÁNÍ KÓDU V ZADNÍ ČÁSTI ZPĚTNÉHO PŘEKLADAČE

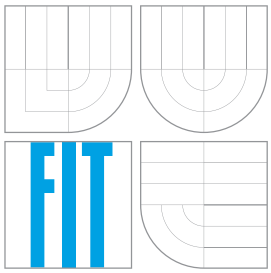
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID HRBEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

STRUKTUROVÁNÍ KÓDU V ZADNÍ ČÁSTI ZPĚTNÉHO PŘEKLADAČE

CODE STRUCTURING IN THE DECOMPILER'S BACK-END

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DAVID HRBEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR ZEMEK
SUPERVISOR

BRNO 2014

Abstrakt

Cílem této práce bylo navrhnout a implementovat algoritmus pro strukturování kódu v zadní části zpětného překladače projektu Lissom. Zabývá se problémem eliminace nepřímých skoků (branch/goto) z nízkoúrovňového kódu s využitím vysokoúrovňových konstrukcí, jako jsou podmíněné příkazy (if, switch) a cykly (for, while). Práce obsahuje teoretický úvod do problematiky zpětného překladu, informace o zpětném překladači projektu Lissom, návrh algoritmu pro strukturování kódu, popis jeho implementace, popis sady testovacích úloh a shrnutí výsledků.

Abstract

The goal of this thesis was to design and implement an algorithm for code structuring in Lissom decompiler's back-end. This algorithm eliminates indirect jumps (branch/goto) from low-level code with a use of high-level constructs, such as conditional statements (if, switch) and loops (for, while). This thesis contains an introduction into the topic of decompilation, some information about the Lissom project's decompiler, a proposal of the structuring algorithm, details of its implementation, testsuite description and results summary.

Klíčová slova

zpětný překlad, zpětný překladač projektu Lissom, LLVM IR, BIR, strukturování kódu, zadní část zpětného překladače

Keywords

decompilation, Lissom project's decompiler, LLVM IR, BIR, code structuring, decompiler's back-end

Citace

David Hrbek: Strukturování kódu v zadní části zpětného překladače, bakalářská práce, Brno, FIT VUT v Brně, 2014

Strukturování kódu v zadní části zpětného překladače

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Zemka.

.....
David Hrbek
20. května 2014

Poděkování

Děkuji svému vedoucímu Ing. Petru Zemkovi za rady, tipy a veškerou pomoc při vedení mé bakalářské práce.

Obsah

Kapitola 1

Úvod

Tato bakalářská práce se zabývá strukturováním kódu v zadní části zpětného překladače projektu Lissom. Zpětný překladač je nástroj, jehož úkolem je získat z nízkoúrovňového kódu ekvivalentní kód s vyšší úrovní abstrakce, například z binárního kódu jeho reprezentaci ve vysokoúrovňovém jazyce, třeba jazyce C. Postup zpětného překladu je inverzní k postupu klasického překladu, kdy ze zdrojového kódu získáme spustitelný program pro danou platformu.

Zadní část zpětného překladače zajišťuje převod vnitřní reprezentace překládaného programu na požadovanou výstupní reprezentaci. V rámci této práce je navržen a implementován algoritmus provádějící v zadní části eliminaci nepřímých skoků (`branch/goto`) z nízkoúrovňového kódu s využitím vysokoúrovňových konstrukcí, jako jsou podmíněné příkazy (`if, switch`) a cykly (`for, while`). Důvodem pro tuto transformaci je to, že kód ve kterém je tok řízení určován pouze nepřímými skoky je pro člověka velice obtížně čitelný. Ze vzniklého kódu strukturovaného pomocí vysokoúrovňových konstrukcí lze mnohem snáze pochopit funkci daného programu.

Cílem práce je vytvořit a implementovat algoritmus, který v nízkoúrovňové reprezentaci správně identifikuje vysokoúrovňové konstrukce a vytvoří vysokoúrovňovou reprezentaci daného programu se správnou strukturou. Strukturování kódu probíhá v rámci převodu jedné vnitřní reprezentace překládaného kódu na druhou. Algoritmus se zabývá pouze vytvořením správné struktury zadaného programu, nikoliv zpětným překladem dalších částí.

Rozvržení práce je následující. V prvních dvou kapitolách je ustaveno teoretické pozadí této práce. Konkrétně v kapitole 2 se nachází obecný popis zpětného inženýrství a jeho využití v oblasti softwaru, tedy zpětného překladu. V kapitole 3 je pak popsána struktura zpětného překladače vyvíjeného v rámci projektu Lissom a technologií, na kterých tento překladač stojí.

Kapitola 4 krátce shrnuje současný stav zpětného překladače projektu Lissom a především cíle této práce.

V kapitole 5 je navržen algoritmus na strukturování kódu, který je stěžejní pro tuto práci. Jeho cílem je nalézt v kódu strukturovaném pouze pomocí skoků vysokoúrovňové řídicí konstrukce (podmíněné příkazy, cykly). Navrhovaný algoritmus hledá tyto konstrukce pomocí průchodu grafem toku řízení (angl. *control-flow graph*) a do jisté míry vychází z algoritmů, které jsou již implementovány a vyzkoušeny v konkurenčních zpětných překladačích.

V kapitole 6 je popsána implementace navrženého algoritmu ve zpětném překladači projektu Lissom. Kapitola 7 popisuje testovací sadu, pomocí které byla ověřována správnost implementace strukturování kódu. Kapitola 8 shrnuje dosažené výsledky a možná budoucí rozšíření.

Kapitola 2

Zpětné inženýrství

V této kapitole se zaměříme na zpětné inženýrství a jeho využití v informatice, především pak na převod binárního kódu do vyšší formy reprezentace. Seznámíme se také s faktory ovlivňujícími kvalitu výsledku, problémy provázejícími tento proces a možnostmi jejich řešení. Informace prezentované v této části byly převzaty z [1], [2] a [3].

Zpětné neboli též reverzní inženýrství (angl. *reverse engineering*) je proces, jehož cílem je zjistit princip fungování daného zařízení, případně vytvořit dokumentaci, na jejímž základě je možné toto zařízení vyrobit či dále modifikovat. Je zjevné, že zpětné inženýrství se nevztahuje pouze k počítačům a zpětnému překladu. Naopak, jeho historie sahá zřejmě až do dob průmyslové revoluce (viz [1], s. 3) a bylo bohatě využíváno k průmyslové špionáži, například v době studené války.

Kapitola je rozdělena následovně. Sekce 2.1 popisuje využití zpětného inženýrství v oblasti IT, sekce 2.2 pak překladač a průběh překladu a sekce 2.3 zpětný překladač, problémy zpětného překladu (sekce 2.3.1) a existující zpětné překladače (sekce 2.3.2).

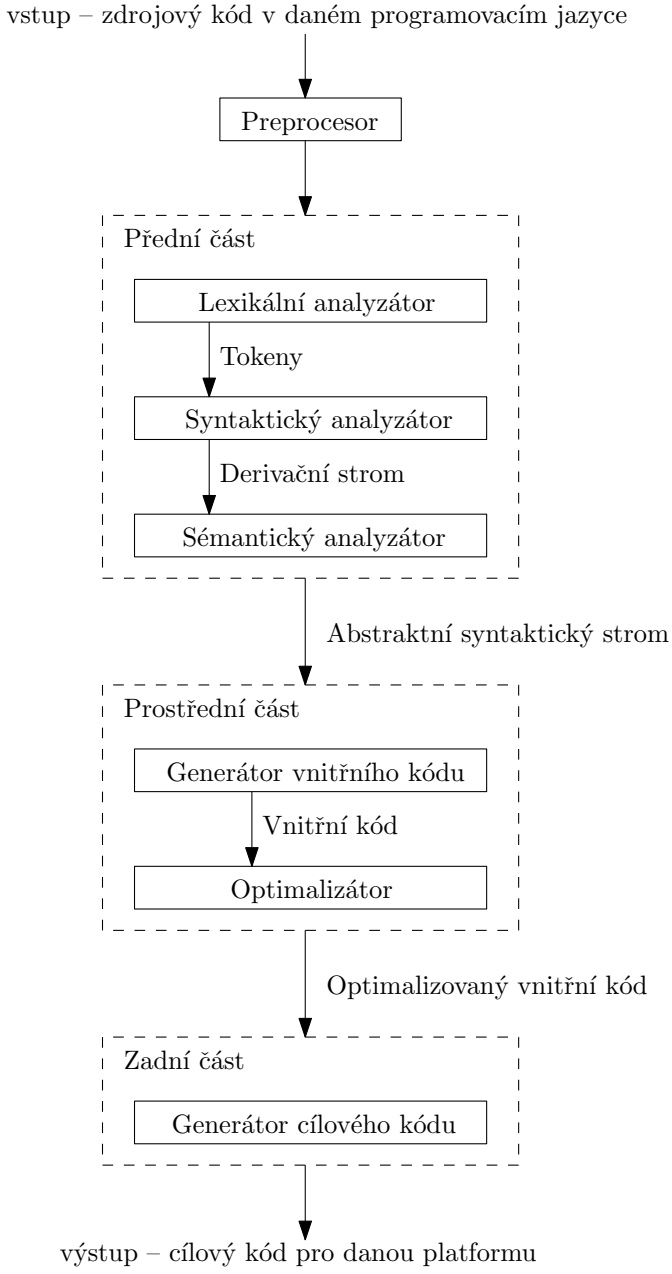
2.1 Zpětné inženýrství v IT

Pro účely této práce nás bude zajímat použití zpětného inženýrství pouze v oblasti počítačů, konkrétně v oblasti softwaru. Zde je úkolem tohoto procesu získat z kódu s nižší úrovní abstrakce kód s vyšší úrovní abstrakce. Například z binárního souboru určeného ke spuštění na daném typu procesoru získat reprezentaci ve vyšší formě, třeba v nějakém programovacím jazyce snáze čitelném pro člověka. Tuto oblast zpětného inženýrství nazýváme zpětný překlad.

2.2 Překladač

Standardní proces tvorby počítačového programu se skládá z několika kroků. Nejdříve je třeba na základě zadání rozmyslet strukturu programu a jeho funkce, poté napsat zdrojový kód. Obvykle je použit některý z vysokoúrovňových jazyků, jejichž výhodou je především platformní nezávislost a vyšší úroveň abstrakce, která více odpovídá tomu, jak si daný problém a jeho řešení představí člověk.

Procesor ovšem tomuto kódu nerozumí a neumí jej vykonat. Je proto potřeba jej pomocí překladače (též kompilátoru, ovšem v této práci se budeme držet označení překladač) přeložit do podoby, kterou umí procesor vykonat. Tento cílový kód (obvykle binární) je



Obrázek 2.1: Typické schéma překladače

funkčně ekvivalentní s kódem zdrojovým. Klasické schéma překladače má tři hlavní fáze a některé podpůrné kroky. Je zobrazeno na obrázku 2.1.

Nejdříve projde zdrojový kód preprocesorem, který jej upraví tím, že odstraní komentáře, rozgeneruje makra, připojí externí soubory, atd.

Přední část (angl. *front-end*) tvoří lexikální analyzátor (angl. *scanner*), syntaktický analyzátor (angl. *parser*) a sémantický analyzátor. Probíhá kontrola správnosti syntaxe vstupního kódu vzhledem k definici daného programovacího jazyka. Pokud je vše v pořádku, je vygenerován abstraktní syntaktický strom, na jehož základě je vytvořena vnitřní reprezentace, například tříadresní kód.

Další část tvoří optimalizátor, jehož úkolem je co nejvíce vylepšit vlastnosti výsledného kódu, obvykle se zaměřením na dobu jeho vykonávání. Toho lze dosáhnout třeba nahrazením některých instrukcí vyžadujících více hodinových cyklů ekvivalentními méně náročnými instrukcemi nebo eliminací tzv. mrtvého kódu, tedy části programu, která nemůže být nikdy vykonána.

Zadní část (angl. *back-end*) odpovídá za převod vnitřní reprezentace do cílového jazyka, většinou strojového kódu. Jeho tvar je závislý na cílové platformě, tedy použitém procesoru a jeho instrukční sadě. Zadní část se také snaží o co nejlepší využití specifických vlastností dané cílové platformy ve prospěch výkonu výsledného kódu.

Tímto postupem získáme objektové soubory jednotlivých modulů programu, které jsou spojeny pomocí linkeru do výsledného spustitelného souboru.

Jak z popisu vyplývá, není optimalizátor ani vnitřní reprezentace v překladači nezbytnou součástí. Je možné přímo přeložit validní vstupní kód pomocí přední a zadní části na cílový kód, nicméně je vhodné nechat překladač kód i optimalizovat.

2.3 Zpětný překladač

Cílem zpětného překladače je provést opačný proces překladu normálního, tedy zpětný překlad (též dekompilaci, ovšem v této práci se budeme držet označení zpětný překlad). Na vstupu je typicky spustitelný soubor, výstupem je funkčně ekvivalentní kód na vyšší úrovni abstrakce. Bohužel je v absolutní většině případů nemožné získat kód totožný s původním zdrojovým kódem, jehož překladem vznikl daný spustitelný soubor, neboť při překladu původního zdrojového kódu na spustitelný program dochází ke ztrátě mnoha informací (například v důsledku optimalizací, nahrazení vysokoúrovňových řídicích konstrukcí skoky, atd.). Důraz je kladen na to, aby získaný kód popisoval program se stejným chováním.

Existují také programy nazvané disassemblery, které též provádějí zpětný překlad, ovšem pouze z binárního kódu do assembleru, tedy na úroveň jednotlivých instrukcí, které jsou závislé na instrukční sadě procesoru. Oproti tomu zpětný překladač obvykle převádí binární kód na vysokoúrovňový programovací jazyk a výsledek je platformě nezávislý a odpovídající disassembler zařizuje jen první část zpětného překladu.

Použití zpětného překladače může mít mnoho důvodů, například analýza chování programu. Nebo jsme ztratili či nemáme přístup ke zdrojovým kódům. Dále lze výstup zpětného překladače využít při hledání chyb a zranitelných míst programu, případně k nalezení změn provedených překladačem při překladu. V neposlední řadě může posloužit jako nástroj umožňující migraci binárního kódu, tedy přenášení programů na jiné architektury, pokud máme k dispozici jen spustitelný soubor pro jednu konkrétní architekturu, případně jako nástroj pro převod zdrojových kódů do jiného programovacího jazyka.

Zpětné překladače lze také použít při nelegálních aktivitách, například pro odstranění ochran proti kopírování programů a podobně. S tím souvisí téma legality zpětného překladu. Co člověk smí a co ne je upraveno především místními zákony (v České republice autorský zákon číslo 121/2000 Sb., konkrétně §66, viz [4]), případně licenční smlouvou mezi autorem programu a jeho uživatelem.

2.3.1 Problémy zpětného překladu

Zpětný překlad je proces podstatně složitější než klasický překlad. Je to způsobeno tím, že při překladu dochází ke ztrátě mnoha informací přítomných v původním zdrojovém kódu. Míra úspěchu zpětného překladu tak závisí za prvé na dokonalosti zpětného překladače a za

druhé, a to především, na informacích přítomných v daném spustitelném souboru a dodržení konvencí při jeho tvorbě.

Už v preprocesoru ztrácíme komentáře, makra a další. Informace o typech proměnných, jejich jménech a jménech funkcí máme ve výsledném binárním souboru k dispozici pouze částečně, například díky symbolům nebo případným ladicím informacím. Také všechny vysokoúrovňové konstrukce jako podmíněné větve nebo cykly jsou nahrazeny nízkoúrovňovými instrukcemi, například skoky podmíněnými příznakovými registry. Některé operace s výrazy jsou v rámci optimalizace nahrazeny sekvencemi instrukcí, které zajistí stejné chování, ale pro člověka je podstatně méně čitelný původní význam. Spoustu problémů také nelze bez dalších informací rozhodnout.

Autoři škodlivých programů se také často snaží zkomplikovat zkoumání binárních souborů (viz [2], s. 4). Jak zdrojový kód, tak výsledný binární soubor může být obfuskován, tedy schválně zatemněn, například přejmenováním proměnných, zrušením struktury kódu smazáním bílých znaků, přidáním nesouvisejícího kódu a dalšími způsoby. Binární soubory mohou být také zabaleny pomocí různých packerů a chráněny proti ladění. Také pokud autor využívá při psaní kódu nestandardní konstrukce a postupy, zpětný překlad tím značně ztíží.

Binární soubor, u kterého chceme provést zpětný překlad, může být různých formátů. Mezi běžné formáty patří například PE, ELF nebo Mach-O a pro zpětný překlad je nutné formát souboru správně identifikovat. Binární soubory se také liší podle toho, pro jaký procesor byly vytvořeny. Existuje velké množství procesorů s různými architekturami lišícími se mezi sebou například instrukční sadou, bitovou šířkou a endianitou.

Zdrojový kód lze zapsat v nepřeberném množství programovacích jazyků a přeložit množstvím různých překladačů, jejichž chování lze ovlivnit také různými přepínači. Například pokud máme k dispozici ladicí informace (u překladače `gcc` přepínač `-g`) nebo symboly, můžeme být z těchto informací schopni získat původní jména funkcí v originálním zdrojovém kódu.

Všechny tyto věci ovlivňují výsledný binární soubor a pokud o nich nemáme informace, lze při zpětném překladu jen obtížně dosáhnout dobrých výsledků.

2.3.2 Existující zpětné překladače

Protože zpětný překlad není v informatice novým tématem, existuje už celá řada zpětných překladačů. Jedním z prvních pokusů byl zpětný překladač `dcc` [2] pro architekturu i80286 a spustitelné soubory MS-DOS. Na základě tohoto zpětného překladače vznikl open-source projekt Boomerang [5] s myšlenkou vytvořit zpětný překladač, který by podporoval různé procesory a formáty souborů.

Ze zpětných překladačů podporujících větší množství platforem stojí za zmínku například zdarma dostupný REC [6] a komerční Hex-Rays [7], který je nástavbou na disassembler IDA. Další z aktivně vyvíjených je projekt SmartDec [8], který podporuje vstup i výstup v jazyce C++ a v neposlední řadě také zpětný překladač projektu Lissom [9], který je podrobněji popsán v následující kapitole, neboť praktická část této práce se zabývá vývojem tohoto zpětného překladače.

Kapitola 3

Zpětný překladač projektu Lissom

Zpětný překladač vyvíjený na Fakultě informačních technologií Vysokého učení technického v Brně v rámci projektu Lissom si klade na tomto poli poměrně vysoké cíle. Důvodem k jeho vzniku byl velký rozmach mobilních zařízení (smartphonů, tabletů, atd.), které používají různé architektury a platformy. Jeho hlavním úkolem je sloužit při statické analýze škodlivého softwaru (*malware*), přičemž tato analýza je platformě nezávislá (viz [10], s. 72–73). Více informací o tomto zpětném překladači lze nalézt především na jeho internetových stránkách (viz [9]), kde je stručně popsán, v sekci *Publications* jsou veřejné články týkající se návrhu a vývoje zpětného překladače a lze si také vyzkoušet samotný zpětný překlad. Při popisu struktury zpětného překladače bylo čerpáno především z [10] a interních dokumentů projektu.

Tvůrci malware se v minulosti zaměřovali téměř výhradně na PC s architekturou x86 a operačním systémem Windows. Ovšem v posledních letech (respektive desetiletí) začínají uživatelé stále častěji používat k manipulaci s daty i zařízení s jinými parametry. Tvůrci škodlivých programů se tomuto trendu přizpůsobili a nyní se je snaží dohnat i firmy poskytující zabezpečení (viz [10], s. 73).

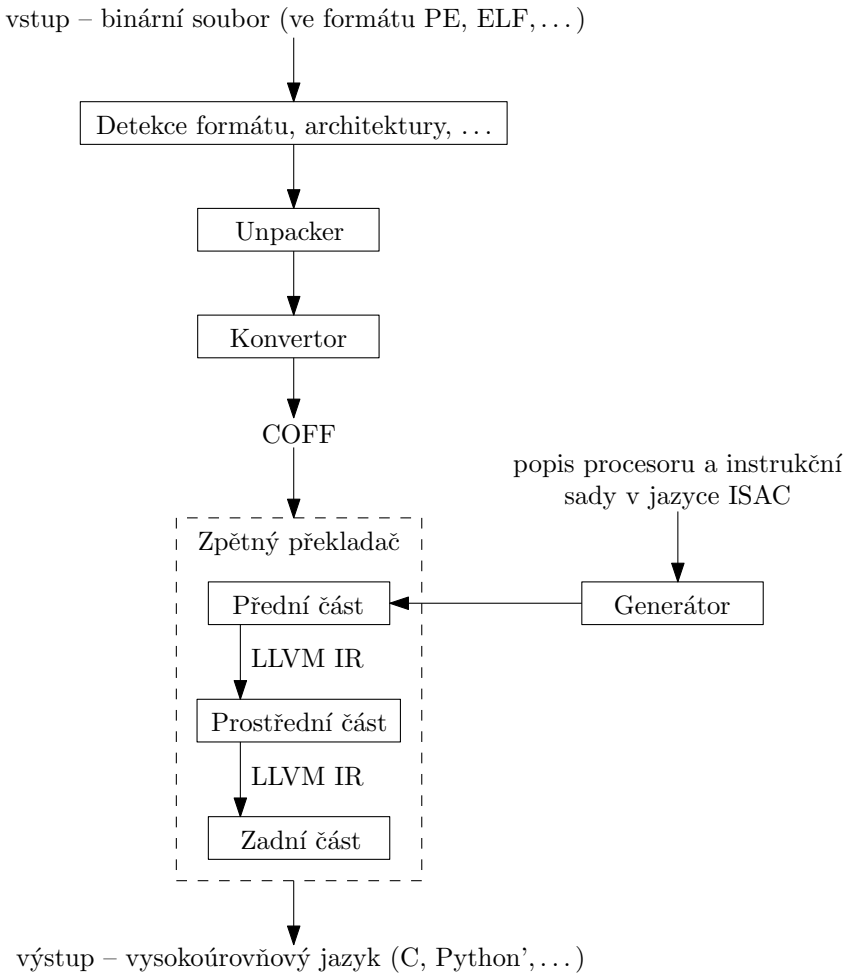
Aby zpětný překladač zvládl držet krok s dobou, je jeho důležitou vlastností to, že je generický. Lze tedy snadno přidat nové architektury, jejichž binární soubory je schopen přeložit. V současné době podporuje zpětný překladač architektury Intel x86, MIPS a ARM. Také je možné relativně snadno přidat na výstup další vysokoúrovňový jazyk. Tyto vlastnosti vyplývají z použité struktury zpětného překladače, kdy je kladen důraz na modularitu, tedy oddělení jednotlivých částí, přičemž po většinu průběhu zpětného překladu je použita platformě nezávislá vnitřní reprezentace.

Více o struktuře zpětného překladače je v sekci 3.1, konkrétně o přední části v sekci 3.1.1, o prostřední části v sekci 3.1.2 a o zadní části v sekci 3.1.3. V sekci 3.2 je stručně popsán projekt LLVM, vnitřní reprezentace LLVM IR (sekce 3.2.1), použitá ve zpětném překladači projektu Lissom, a její ukázka (sekce 3.2.2).

3.1 Struktura zpětného překladače

Zpětný překladač má podobně jako překladač tři hlavní fáze, tedy přední, prostřední a zadní část. Schéma zpětného překladače projektu Lissom je zobrazeno na obrázku 3.1. Na vstupu máme platformě závislý binární soubor, který je předzpracován ve třech krocích. Nejdříve je zjištěn formát souboru a architektura, pro kterou byl vytvořen a také jaký překladač, případně packer, byl při jeho tvorbě použit. Potom je soubor v případě, že byl během tvorby

komprimován, rozbalen pomocí unpackeru a pomocí konverzních algoritmů je platformě závislý soubor převeden na formát souboru odvozený od formátu COFF. Tento převod je výhodný, protože dále pracujeme s jednotným formátem souboru a není tedy třeba řešit různé varianty pro různé formáty souborů.



Obrázek 3.1: Schéma zpětného překladače projektu Lissom

3.1.1 Přední část

Nyní vstupujeme do přední části (angl. *front-end*), kde dochází k převodu binárního kódu do vnitřní reprezentace v jazyce LLVM IR (tento jazyk je blíže popsán v sekci 3.2). Aby byl tento převod možný, musíme znát binární kódování jednotlivých instrukcí dané instrukční sady a také to, co jednotlivé instrukce dělají. Tyto informace jsou automaticky generovány z popisu procesoru v jazyce ISAC (viz [11]). Tento jazyk je také vyvíjen v rámci projektu Lissom. Popis procesoru je složen ze dvou částí. Za prvé popisu prostředků procesoru, tedy jeho registrů a hierarchie paměti. A za druhé popisu chování jednotlivých instrukcí. Pomocí těchto informací je automaticky vygenerován nástroj podobný disassembleru a jednotlivé instrukce jsou převedeny na odpovídající instrukci (respektive sekvenci instrukcí) v jazyce LLVM IR.

Přední část se také v kódu snaží nalézt úseky vložené ze staticky linkovaných knihoven. Detekce je inspirována technologií FLIRT (zkratka z angl. *Fast Library Identification and Recognition Technology*) používanou ve zpětném překladači Hex-Rays. Kód staticky linkovaných knihoven je z překládaného kódu vypuštěn a příslušné objektové soubory, které dané knihovny obsahují (mohou být různých formátů), jsou extrahovány, převedeny na společný formát a zabaleny do jednoho archivu. Oproti technologii FLIRT je výhoda v tom, že veškerá další práce se týká jednoho formátu souborů (viz [10], s. 80).

Dále je v přední části provedena statická analýza kódu se zaměřením na analýzu toku řízení (angl. *control-flow*), v jejímž rámci jsou například detekovány funkce a je vygenerována nízkoúrovňová vnitřní reprezentace v jazyce LLVM IR.

3.1.2 Prostřední část

Na vstupu do prostřední části (angl. *middle-end*) máme kód v jazyce LLVM IR na nízké úrovni. Vzhledem k tomu, že v přední části je při převodu z binárního kódu každá instrukce dekódována zvlášť, nemáme zatím žádnou představu o vyšších konstrukcích v programu, jako například cyklech. Každá původní instrukce odpovídá jednomu základnímu bloku.

V rámci několika průchodů dochází k řadě optimalizací. Jsou využívány jak optimalizace vyvinuté již v rámci projektu LLVM, tak optimalizace vyvinuté pro samotný zpětný překladač. Například jsou vypuštěny zbytečné instrukce.

Celkově je cílem prostřední části především optimalizovat vnitřní reprezentaci pro další práci v zadní části.

3.1.3 Zadní část

Cílem zadní části (angl. *back-end*) je převést vnitřní LLVM IR reprezentaci přes BIR (zkratka angl. *back-end intermediate representation*) na cílový vysokoúrovňový jazyk. Zpětný překladač podporuje v současné době jazyky C (konkrétně C podle normy ISO C99) a Python', tedy Python (verze 3) rozšířený o některé konstrukce jazyka C (například ukazatele a příkazy `goto` a `switch`).

Samotný převod do vysokoúrovňového jazyka je proveden průchodem grafu toku řízení a na základě informací získaných analýzou v předchozích částech se zpětný překladač snaží použít co nejvhodnější vysokoúrovňové konstrukce. O konverzi se stará program `llvmir2hll`, který využívá také knihovnu `BIR` pro práci s vnitřní reprezentací v zadní části.

Kromě vytvoření kódu v cílovém jazyce se zadní část stará také o další analýzy a optimalizace a celkové zlepšení vlastností finálního výstupu. Jsou generovány grafy toku řízení (angl. *control-flow graph*) jednotlivých funkcí a grafy volání (angl. *call graph*) a další statistické informace.

3.2 LLVM

LLVM je projekt zastřešující vývoj modulárních překladačů a dalších nízkoúrovňových nástrojů (assemblerů, debugerů), které jsou kompatibilní s již existujícími nástroji [12]. Zabývá se jak klasickými statickými překladači, tak tzv. *just-in-time* překladači. Mezi známé produkty patří například překladač Clang.

Oproti konkurenčním projektům se liší především vnitřní architekturou, kdy je kladen důraz na modularitu a možnost znovuvyužití již vytvořených částí, čemuž odpovídá velké množství knihoven s kvalitním rozhraním. Vnitřní reprezentace kódu v překladači může být

tří typů. Za prvé textová IR (zkratka z angl. *intermediate representation*), za druhé binární (přeložená textová) a za třetí vnitřní reprezentace v paměti počítače, na které probíhají optimalizace. Všechny tři jsou vzájemně ekvivalentní a lze převést jednu na druhou bez ztráty informací. Nyní se blíže podíváme na textovou reprezentaci LLVM IR a některé její další vlastnosti.

3.2.1 LLVM IR

Tato reprezentace má několik zajímavých vlastností. Například je to univerzální, na jazyku a architektuře nezávislá instrukční sada, jejíž instrukce jsou ve tříadresní formě a jsou velmi podobné většině RISC procesorů. Dále je to nezávislý systém typů. Proměnné mají svůj typ a také instrukce kontrolují správnost typů u použitých operandů. Explicitní přetypování je možné pomocí instrukce `cast`. K dispozici jsou základní typy void, boolean, celočíselné (*integer*) různých velikostí a desetinné (*float*). A také odvozené typy ukazatel (*pointer*), pole (*array*), struktura (*structure*) a funkce (*function*). Složením těchto typů lze vytvářet složitější datové typy v určitém programovacím jazyce.

Registry, jejichž podoba je závislá na daném procesoru, jsou nahrazeny virtuálními registry (proměnnými) automaticky pojmenovanými číslem, počínaje nulou. Proměnné mají vlastnost *Static Single Assignment*, což znamená, že je jim hodnota přiřazena právě jednou při jejich vzniku. Pokud se v původním programu mění hodnota proměnné, je v LLVM IR vytvořena nová proměnná a do ní je přiřazena nová hodnota. To výrazně usnadňuje analýzu závislostí mezi proměnnými. Pokud je v původním programu konstrukce `if/else` a v obou větvích programu je nějakým různým způsobem upravována stejná proměnná (tedy o aktuální hodnotě proměnné po skončení konstrukce `if/else` rozhoduje to, kterou větví šel tok řízení programu), použije se v LLVM IR v následujícím základním bloku tzv. ϕ instrukce (angl. *phi-instruction*), která do nové proměnné přiřadí hodnotu na základě toho, ze kterého základního bloku bylo tomuto bloku předáno řízení (tedy podle toho, která větev byla vykonána) a dále je pracováno s touto novou proměnnou. ϕ instrukce je použita i v ukázce kódu na obrázku 3.3, který bude popsán dále.

Každý základní blok končí ukončující instrukcí (angl. *terminator instruction*), která určuje následníka nebo následníky bloku. Jedná se buď o jednu z instrukcí `br` (*branch*), `ret` (*return*), nebo `switch`, nebo o méně obvyklé ukončující instrukce `indirectbr` (nepřímý skok), `invoke`, `resume` a `unreachable`. Více o ukončujících instrukcích lze nalézt v manuálu LLVM [13]. Jednotlivé instrukce také mohou mít přiřazeny metadata, která nesou další informace využitelné při dalším zpracování. Například zpětný překladač Lissom si pomocí metadat přenáší původní jména proměnných (pokud jsou ve spustitelném programu dohledatelná), která poté využije při generování výsledného kódu.

Výhodou této reprezentace je také to, že v rámci projektu LLVM pro ni byla vyvinuta celá řada sofistikovaných optimalizací.

3.2.2 Ukázka kódu

V této části je pro názornost uvedena část kódu v reprezentaci LLVM IR. Jedná se o rekurzivní implementaci funkce faktoriál. Příklad je převzat z [10]. Na obrázku 3.2 je funkce zapsána v jazyce C a na obrázku 3.3 je potom tato funkce v LLVM IR.

Na prvním řádku obrázku 3.3 začíná definice funkce `factorial` klíčovým slovem `define`, dále pokračuje návratovým typem `i32`, což značí 32 bitové celé číslo. Pak je uveden název funkce a v závorce seznam dvojic typ a název parametru. V tomto případě je funkce volána s jedním parametrem, který je opět typu 32 bitové celé číslo. Proměnné jsou v LLVM IR

```
1  int factorial(int n) {
2      if(n == 0)
3          return 1;
4      return n * factorial(n - 1);
5  }
```

Obrázek 3.2: Rekurzivní implementace funkce faktoriál v jazyce C

```
1  define i32 @factorial (i32 %n) {
2  entry:
3      %0 = icmp eq i32 %n, 0
4      br i1 %0, label %bb2 , label %bb1
5  bb1:
6      %1 = add i32 %n, -1
7      %2 = icmp eq i32 %1, 0
8      br i1 %2, label %factorial_.exit , label %bb1.i
9  bb1.i:
10     %3 = add i32 %n, -2
11     %4 = call i32 @factorial (i32 %3)
12     %5 = mul i32 %4, %1
13     br label %factorial_.exit
14 factorial_.exit:
15     %6 = phi i32 [ %5, %bb1.i ], [ 1, %bb1 ]
16     %7 = mul i32 %6, %n
17     ret i32 %7
18 bb2:
19     ret i32 1
20 }
```

Obrázek 3.3: Rekurzivní implementace funkce faktoriál v LLVM IR

uvozeny znakem `%`. Na druhém řádku je návěští značící začátek základního bloku daného jména (v tomto případě `entry`). Návěští jsou také na řádcích 5, 9, 14 a 18. Za návěštím následuje posloupnost instrukcí daného základního bloku.

Například na řádku 3 se volá instrukce `icmp`, která porovnává na základě zadaných parametrů `eq` a `i32` dvě 32 bitová celá čísla na rovnost, v tomto případě hodnotu proměnné `%n` s konstantou 0. Výsledek typu boolean je uložen do proměnné `%0`.

Na řádku 4 je poté volána instrukce `br`, v tomto případě s podmínkou typu `i1`, tedy boolean, získanou voláním předchozího porovnání. Pokud je podmínka splněna, dojde ke skoku na návěští `bb2`. Pokud není, pokračuje vykonávání programu od návěští `bb1`.

Další instrukce použité v této ukázce jsou instrukce `add`, která sečte dvě hodnoty zadaného typu, instrukce `mul`, která dvě hodnoty zadaného typu vynásobí, poté následuje instrukce `phi`. `phi` je reprezentace ϕ instrukcí v jazyce LLVM IR. ϕ instrukce, jak již bylo popsáno dříve, vrací hodnotu zadaného typu na základě toho, ze kterého základního bloku bylo tomuto bloku předáno řízení. V tomto případě vrací buď hodnotu proměnné `%5`, pokud vykovávání bloku `factorial_.exit` následuje vykonávání bloku `bb1.i`, nebo konstantu 1, pokud předchozím vykonaným blokem byl blok `bb1`. Dvojic proměnná, respektive konstanta, a příchozí blok může být ve ϕ instrukci i více než dvě, jako v tomto případě. Jejich

počet je dán počtem předchůdců daného základního bloku. Teoreticky jich může být i neomezeně mnoho. Případné ϕ instrukce v základním bloku vždy předcházejí volání dalších příkazů.

Na řádku 13 je příklad nepodmíněného skoku pomocí instrukce `br`, který vede na zadané návěští `factorial_.exit`.

Poslední nezmíněné funkce jsou funkce `call` a `ret`. Funkce `call` (řádek 11) provede volání zadané funkce s případnými parametry. V tomto případě se jedná o rekurzivní volání funkce `factorial` a návratová hodnota je uložena do proměnné `%4`. Funkce `ret` vrací hodnotu daného typu volající funkci v místě jejího volání instrukcí `call`.

Kapitola 4

Cíl práce a popis současného stavu

V současné době je strukturalizace kódu ve zpětném překladači projektu Lissom řešena ve třídě `OrigLLVMIR2BIRConverter`. Jedná se o konvertor, který převádí vnitřní reprezentaci v LLVM IR na vnitřní reprezentaci zadní části BIR. Základ tohoto konvertoru je použit i při implementaci nového konvertoru, tedy třídy `StructLLVMIR2BIRConverter`. Důvodem práce na novém konvertoru jsou problémy toho původního při zpětném překladu některých příkladů, kdy výstup nemá správnou strukturu či by šel strukturovat lépe.

Cílem práce je vytvořit konvertor, který generuje pouze strukturu překládaného programu, nikoliv veškerý jeho kód. Na místa, kde by se měl vykonávat kód jednotlivých základních bloků jsou pouze vloženy komentáře se jménem návěští daného bloku, který by v tomto místě měl být. Nový konvertor používá původní implementaci pro konverzi podmínek podmíněných příkazů (`if`, `switch`) z LLVM IR do BIR a také pouze mírně modifikované řešení ϕ instrukcí. Nejlépe je požadovaný tvar výstupu konvertoru vidět na obrázku 4.1, kde je výstup v jazyce C pro zpětný překlad kódu z příkladu 3.3.

```
1  int factorial(int n) {
2      // entry
3      if(n == 0) {
4          // bb2
5          return 1;
6      }
7      // bb1
8      if(apple == 0) {
9          banana = 1;
10     }
11     else {
12         // bb1.i
13         banana = plum;
14     }
15     // factorial_.exit
16     return banana;
17 }
```

Obrázek 4.1: Výstupní kód v jazyce C pro zpětný překlad LLVM IR kódu z příkladu 3.3

Na řádcích 9 a 13 je vidět řešení ϕ instrukce z řádku 15 v příkladu z obrázku 3.3, kdy je požadovaná hodnota, kterou má proměnná nabývat v bloku `factorial_.exit` přiřazena na

konci obou bloků, které do bloku `factorial_.exit` přicházejí. Kvůli ϕ instrukci je dokonce vytvořena větev příkazu `if`, ve které dochází pouze k přiřazení správné hodnoty proměnné. Jména použitých proměnných jsou generována generátorem jmen proměnných zpětného překladače, který implicitně využívá anglické výrazy pro různé ovoce. Ve výstupu není vidět souvislost mezi jednotlivými proměnnými, protože aritmetické operace, které s nimi manipulují, nejsou generovány, stejně jako další instrukce jednotlivých základních bloků.

Jak již bylo řečeno, cílem práce je návrh a implementace algoritmu pro strukturování kódu. Cílem algoritmu není vygenerovat kód ekvivalentní se zadaným kódem, ale pouze správnou strukturu tohoto kódu.

Kapitola 5

Návrh strukturování kódu

V této kapitole je přiblížena problematika strukturování kódu v zadní části zpětného překladače, především pak návrh algoritmu pro strukturování kódu. Strukturování je prováděno v rámci konvertoru LLVM IR (vstupní reprezentace zadní části) na BIR (vnitřní reprezentace použitá v zadní části, ze které je následně generován výstupní kód v daném výstupním jazyce).

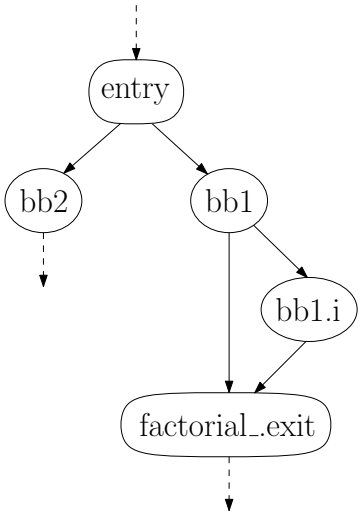
Původní vysokoúrovňové konstrukce tvořící strukturu programu lze v překládaném kódu nalézt zkoumáním grafu toku řízení (*control-flow* graf), který je blíže popsán v sekci 5.1. Cílem je rozlišit cykly a podmíněné příkazy, v případě jazyka C konkrétně konstrukce `if`, `if/else`, `switch`, `for`, `while` a řídicí příkazy `break` a `continue`. Pokud není možné nějakou část překládaného programu ekvivalentně popsat s využitím těchto vysokoúrovňových konstrukcí, bude tato část ve výstupním jazyce využívat příkazy `goto`.

V sekci 5.1 jsou popsána některá specifika, kterými se liší grafy toku řízení v LLVM IR od teoretických grafů toku řízení, které jsou popsány v sekci 5.1. Sekce 5.2 popisuje navrhovaný algoritmus pro vyhledávání řídicích konstrukcí. Konkrétně sekce 5.2.1 vymezuje základní pojmy z oblasti grafů a jejich zpracování prohledáváním do hloubky. V sekci 5.2.2 je pak uveden pseudokód navrhovaného algoritmu se stručným popisem.

5.1 Graf toku řízení

Graf toku řízení (angl. *control-flow graph*, dále značen běžně používanou zkratkou CFG) je orientovaný graf s jedním kořenem, ve kterém základní bloky programu tvoří uzly a orientované hrany tok řízení mezi těmito bloky (viz [14], s. 1 a 2). Může být definován jako graf $G = (N, E, r)$, kde N je množina uzlů grafu, E je množina orientovaných hran a $r \in N$ je kořen grafu. Ukázku CFG pro funkci faktoriál z příkladu na obrázku 3.3 můžeme vidět na obrázku 5.1. Vstupní přerušovaná šipka značí předání řízení funkci pomocí instrukce `call` a výstupní přerušované šipky značí vrácení řízení a návratové hodnoty instrukcí `ret`.

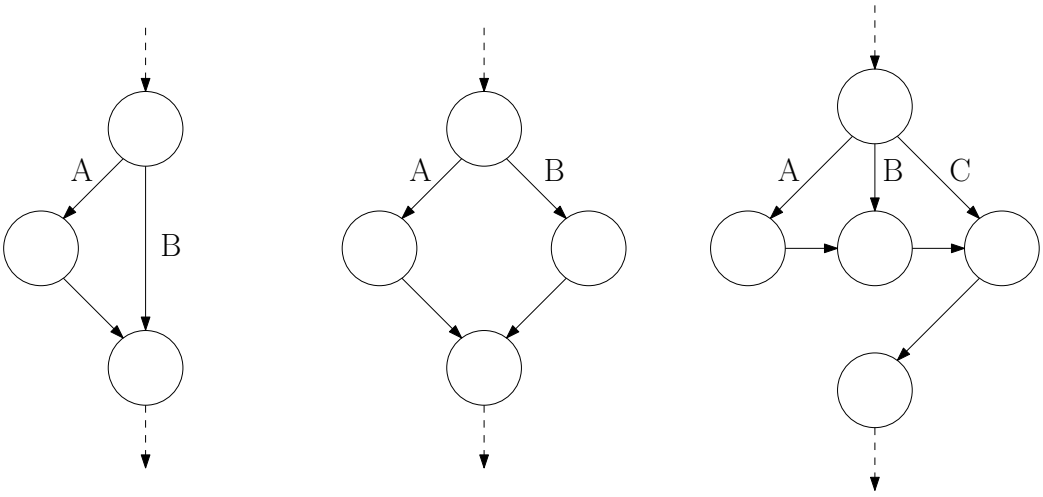
Na vstupu do zadní části zpětného překladače máme program v LLVM IR, kde je základní blok tvořen posloupností instrukcí ukončených buď instrukcí skoku na další blok, nebo instrukcí `ret`, tedy návratem řízení do bloku, odkud byla daná funkce volána. Strukturování je řešeno v rámci jednotlivých funkcí, takže tok řízení mezi bloky je určován buď instrukcí `br` (*branch*), nebo instrukcí `switch`. U instrukce `br` může skok na další blok být jak nepodmíněný (existuje jeden daný blok, kterým bude vykonávání programu pokračovat), tak podmíněný (na základě podmínky pokračuje vykonávání programu jedním ze dvou



Obrázek 5.1: CFG funkce faktoriál z příkladu 3.3

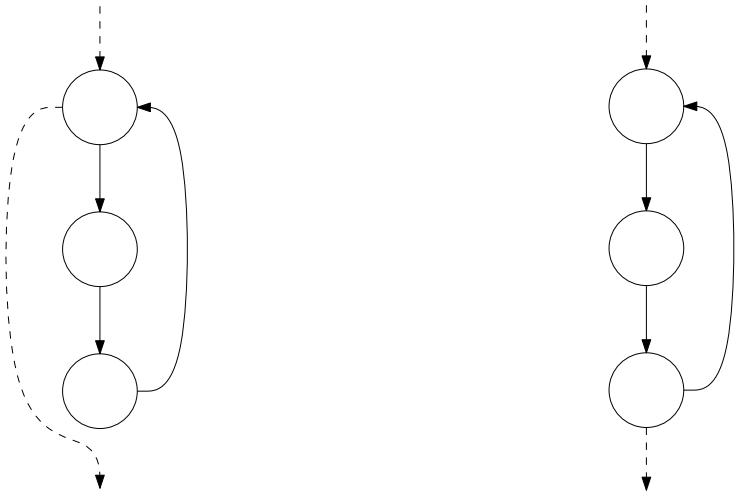
možných bloků). U instrukce `switch` je řízení předáno na základě hodnoty řídicí proměnné do dalšího bloku.

Na obrázcích 5.2 a 5.3 jsou znázorněny jednotlivé vysokoúrovňové řídicí konstrukce tak, jak vypadají v CFG. Jednotlivé konstrukce do sebe samozřejmě mohou být libovolně zanořeny a tvořit složitější programovou strukturu. Cykly typu `while` a `for` jsou na úrovni CFG reprezentovány stejně a jejich odlišení je možné až na základě datové analýzy, kdy hledáme indukční proměnnou odlišující cyklus `for`.



Obrázek 5.2: Příklady CFG konstrukcí `if`, `if/else` a `switch`

přerušované šipky naznačují pokračování CFG. Vstupní přerušovaná šipka jde do uzlu, který budeme nadále nazývat hlavičkou dané konstrukce. Výstupní přerušovaná šipka jde z uzlu, který budeme nadále nazývat koncovým uzlem dané konstrukce. V případě cyklů `while`, respektive `for`, se jedná o tentýž uzel. Uzly, kterými prochází řízení programu mezi hlavičkou a koncovým uzlem dané konstrukce, tvoří tělo této konstrukce.



Obrázek 5.3: Příklady CFG cyklů `while`, respektive `for`, a `do/while`

První znázorněnou konstrukcí na obrázku 5.2 je `if`. Program vykoná větev A tohoto CFG, pokud je splněna podmínka v hlavičce konstrukce. Pokud podmínka splněna není, jde řízení z hlavičky přímo do koncového uzlu konstrukce (větev B).

Druhou znázorněnou konstrukcí na obrázku 5.2 je `if/else`. Program vykoná větev A tohoto CFG, pokud je splněna podmínka v hlavičce konstrukce. Pokud podmínka splněna není, vykoná program větev B.

Třetí znázorněnou konstrukcí na obrázku 5.2 je `switch` se dvěma větvemi s podmínkou `case` (větve A a B) a jednou `default` větví (větev C). Tok řízení přes `switch` bez použití `break` je tzv. *fallthrough*. Tento `switch` nepoužívá příkaz `break`, provede se tedy vždy větev splňující podmínku i všechny následující větve. Program vykoná větev A tohoto CFG (a všechny následující), pokud je splněna podmínka pro první `case`. Program vykoná větev B tohoto CFG (a všechny následující), pokud je splněna podmínka pro druhý `case`. Pokud ani jedna podmínka splněna není, provede se větev C (`default`). Pokud by `default` větev neexistovala a nebyla splněna ani jedna podmínka `case`, šel by tok řízení přímo z hlavičky do koncového uzlu konstrukce.

Na obrázku 5.3 jsou zobrazeny cykly `while`, respektive `for`, a `do/while`. Obě konstrukce mají na obrázku jediný možný vstup i jediný možný výstup toku řízení (jsou to tzv. *single-entry* a *single-exit* cykly). Případné cykly s více možnými výstupy toku řízení je možné vyjádřit použitím příkazu `break`. U cyklu `while`, respektive `for`, je podmínka na pokračování cyklu vyhodnocena na začátku, tělo cyklu tedy nemusí být vykonáno ani jednou. Hlavička a koncový uzel cyklu tohoto typu je tentýž uzel. U cyklu `do/while` je podmínka na pokračování cyklu vyhodnocena na konci, tělo cyklu je tedy vykonáno nejméně jednou.

CFG v LLVM IR

Grafy toku řízení ve vnitřní reprezentaci LLVM IR zpětného překladače nemají vždy tak ideální vlastnosti, jak jsou popsány v předchozí části. Jedním ze specifik je například to, že zanořené konstrukce `if`, respektive `if/else`, mohou sdílet koncový uzel.

Cykly typu `while`, respektive `for`, se v LLVM IR na vstupu zadní části nevyskytují. Jedná se totiž o kód na nízké urovní, kdy jsou cykly reprezentovány pomocí nepřímých skoků a návěští. Cykly mají koncovou podmínku testovánu většinou na konci, někdy i v těle cyklu. Pokud se v původním kódu jedná o cyklus s podmínkou na začátku, kdy při překladu

není jasné, zda bude tělo cyklu alespoň jednou provedeno, je tato podmínka otestována konstrukcí typu `if` a daný cyklus je převeden na cyklus s podmínkou na konci prováděný v těle této konstrukce `if`. Takto je zajištěno, že pokud podmínka není splněna hned na začátku, tělo konstrukce `if` se nevykoná a řízení je předáno koncovému uzlu této konstrukce.

Pokud tělo cyklu tvoří pouze jeden základní blok, potom je tento blok ukončen podmíněnou instrukcí `br` (má tedy dva následníky). Jedním následníkem je blok sám (hrana CFG vede z tohoto uzlu opět do něj) a druhým pak uzel, kterým pokračuje vykonávání programu po skončení cyklu.

Ukázku instrukce `switch` v LLVM IR můžeme vidět na obrázku 5.4. Za názvem instrukce následuje typ řídicí proměnné a samotná proměnná. V tomto případě se jedná o 32 bitovou celočíselnou proměnnou `%tmp1`. Poté následuje návěští, na které je odskočeno v případě, že řídicí proměnná nenabývá žádné z hodnot zadaných dále. Toto návěští odpovídá `default` větvi konstrukce `switch` v jazyce C, případně bloku za touto instrukcí, pokud `default` větev nemá. Dále je uvedeno pole dvojic hodnota a návěští (řádky 2, 3 a 4), kdy je vždy porovnána hodnota řídicí proměnné s danou konstantou a pokud jsou shodné, je odskočeno na návěští uvedené za touto konstantou.

```
1 switch i32 %tmp1, label %bb9 [
2     i32 0, label %bb3
3     i32 1, label %bb5
4     i32 2, label %bb7
5 ]
```

Obrázek 5.4: Ukázka instrukce `switch` v LLVM IR

5.2 Algoritmus pro vyhledání řídicích konstrukcí

Navrhovaný algoritmus pro vyhledání řídicích konstrukcí vychází z algoritmu pro vyhledávání smyček v CFG vytvořeného na univerzitě v Pekingu (viz [15]). Tento algoritmus byl jako základ navrhovaného algoritmu vybrán z několika důvodů. Především by měl být oproti ostatním algoritmům zjišťujícím strukturu CFG rychlejší a jednodušší na paměťovou náročnost i implementaci. V článku [15] je rychlost algoritmu porovnána s algoritmem Havlak-Tarjan, respektive Ramalingam-Havlak-Tarjan. Tyto algoritmy tvoří základ většiny algoritmů pro zjišťování struktury CFG. Jsou založeny na vyhledávaní intervalů v grafu (viz [16]) a i přesto, že teoreticky mají kvadratickou, respektive téměř lineární časovou náročnost, na CFG reálných programů je jejich výkon horší.

Cílem navrhovaného algoritmu je na základě průchodu CFG metodou prohledávání do hloubky (angl. *depth-first search*, dále značeno běžně používanou zkratkou DFS) vytvořit pro každý základní blok (uzel CFG) anotaci nesoucí informace, zda je daný blok hlavičkou nějaké vyhledávané konstrukce, nebo koncovým uzlem nějaké vyhledávané konstrukce, nebo je v těle některé vyhledávané konstrukce. A pokud ano, která z řídicích konstrukcí to je a pokud je blok součástí těla konstrukce, či jejím koncovým uzlem, který blok je hlavičkou této konstrukce.

O CFG překládané funkce musíme předpokládat, že se jedná o obecný graf. I když v dnešní době již většina programátorů píše strukturované kódy, tedy bez použití příkazu

`goto`, dochází při překladu v rámci optimalizací často k tomu, že i původně strukturovaný CFG obsahuje neredukovatelné podgrafy (viz. [15], s. 2).

5.2.1 Definice pojmů

V této části jsou zavedeny pojmy použité v dalším textu. Stručnou teorii z oblasti CFG lze nalézt v článcích [14] a [15].

Prohledávání do hloubky (DFS) je metoda průchodu grafem, kdy jsou navštíveny všechny uzly v následujícím pořadí. Aktuální uzel (začínáme kořenem grafu) označíme jako navštívený a do seznamu uzlů k dalšímu zpracování zařadíme na začátek seznam jeho následníků. Z tohoto seznamu nezpracovaných uzlů vyjmeme první uzel, který bude zpracován jako další (v příštím kroku se z něj stane aktuální uzel).

Pokud začínáme DFS kořenem grafu a všechny uzly grafu jsou dostupné, potom hrany procházené algoritmem DFS tvoří tzv. *depth-first spanning tree* (dále značeno DFST, viz [15], s. 5).

Vezmeme-li uzel U, potom cestu v DFST z kořenu grafu do uzlu U nazveme *depth-first search path* (dále značeno DFSP) a značíme DFSP(U). Pokud máme uzel V takový, že uzel U leží v cestě DFSP(V), potom DFSP(U, V) je část cesty DFSP(V) z uzlu U do uzlu V (viz [15], s. 5).

V CFG rozlišujeme 3 typy hran (viz. [15], s. 6). Jsou to dopředné hrany (angl. *forward edges*), zpětné hrany (angl. *back edges*) a příčné hrany (angl. *cross edges*). Dopředné hrany jsou ty, které vedou z aktuálního uzlu do uzlu, který ještě nebyl zpracován. Zpětné hrany jsou ty, které vedou z aktuálního uzlu U do uzlu V, přičemž uzel V již byl zpracován a V je v cestě DFSP(U). Příčné hrany jsou zbylé hrany, tedy ty, které vedou z aktuálního uzlu U do uzlu V, přičemž uzel V již byl zpracován a V není v cestě DFSP(U).

Dominátor (angl. *dominator*) uzlu U je takový uzel V, že všechny cesty z kořene CFG do uzlu U procházejí uzlem V. Bezprostřední dominátor (angl. *immediate dominator*) uzlu U je takový uzel V, že neexistuje uzel W takový, aby V bylo dominátorem W a W dominátorem U, pokud $U \neq V \neq W$ (viz [14], s. 4).

Silně souvislá oblast grafu (angl. *strongly connected region*, dále značeno SCR) je takový podgraf S grafu G, kde pro libovolnou dvojici uzlů U, V podgrafu S existuje v podgrafu S orientovaná cesta z uzlu U do uzlu V a z uzlu V do uzlu U. Maximální SCR je největší podgraf daného CFG splňující vlastnost SCR (viz. [15], s. 6).

Vnější smyčka (angl. *outermost loop*) CFG je maximální SCR daného CFG. První zpracovaný uzel tohoto SCR metodou DFS je hlavičkou této smyčky a zbylé uzly tohoto SCR tvoří její tělo. Pokud označíme uzly tohoto SCR (uzly vnější smyčky) jako množinu L a hlavičku této smyčky jako uzel H, pak vnitřní smyčka (angl. *inner loop*) je SCR na podgrafu tvořeném uzly z množiny $\{L - \{H\}\}$. Pokud máme uzel U nějakého CFG, pak jeho nejvnitřnější smyčka (angl. *innermost loop*) je nejmenší SCR, jehož je uzel U součástí. Seznam hlavičkových uzlů smyček k danému uzlu U je pak tvořen hlavičkou jeho nejvnitřnější smyčky, hlavičkou smyčky, ve které se nachází hlavička nejvnitřnější smyčky, atd. Tento seznam tak nese informaci o vzájemném zanoření smyček a tvoří tzv. *loop-nesting forest* daného CFG (viz. [15], s. 6).

5.2.2 Pseudokód algoritmu

V této části je pseudokód algoritmu použitého pro získání informací potřebných pro strukturování překládaného kódu. Základ algoritmu je tvořen dvěma rekurzivními funkcemi, které

provádějí DFS průchod CFG dané funkce. Pseudokód prvního průchodu je na obrázku 5.5 a druhý průchod pak na obrázku 5.7.

```
1  block getCFGStructure(block b0, int DFSPPos)
2    označit b0 jako zpracovaný uzel;
3    b0.DFSPPos = DFSPPos;  // označení pozice b0 v DFSP
4    if(počet následníků b0 > 1)
5      označit b0 jako hlavičku konstrukce if, resp. switch;
6    if(blok b0 končí instrukcí ret)
7      označit, že hlavička větve, ve které je blok b0, má větev
           končící instrukcí ret;
8    foreach(block b in successors(b0))
9      if(uzel b nebyl zatím zpracován)  // případ A
10         block innermostHeader = traverse_DFS(b, DFSPPos + 1);
11         označit hlavičku cyklu innermostHeader k bloku b0;
12     else
13       if(b.DFSPPos > 0)  // případ B
14         označit b jako hlavičku cyklu;
15         označit hlavičku cyklu b k bloku b0;
16         označit blok b0 jako koncový blok cyklu, resp. blok
                   s continue;
17       else if(b nemá hlavičku cyklu)  // případ C
18         označit b jako konec konstrukce if, respektive switch;
19         označit konec b k hlavičce konstrukce;
20       else
21         block h = b.innermostLoopHeader;
22         if(h.DFSPPos > 0)  // případ D
23           označit b jako konec konstrukce if, respektive switch;
24           označit konec b k hlavičce konstrukce;
25           označit hlavičku cyklu h k bloku b0;
26         else  // případ E
27           označit blok b0 jako blok s goto;
28           while(h.innermostLoopHeader != NULL)
29             h = h.innermostLoopHeader;
30             if(h.DFSPPos > 0)
31               označit hlavičku cyklu h k bloku b0;
32               break;
33   b0.DFSPPos = 0;
34   return b0.innermostLoopHeader;
```

Obrázek 5.5: Pseudokód prvního průchodu CFG

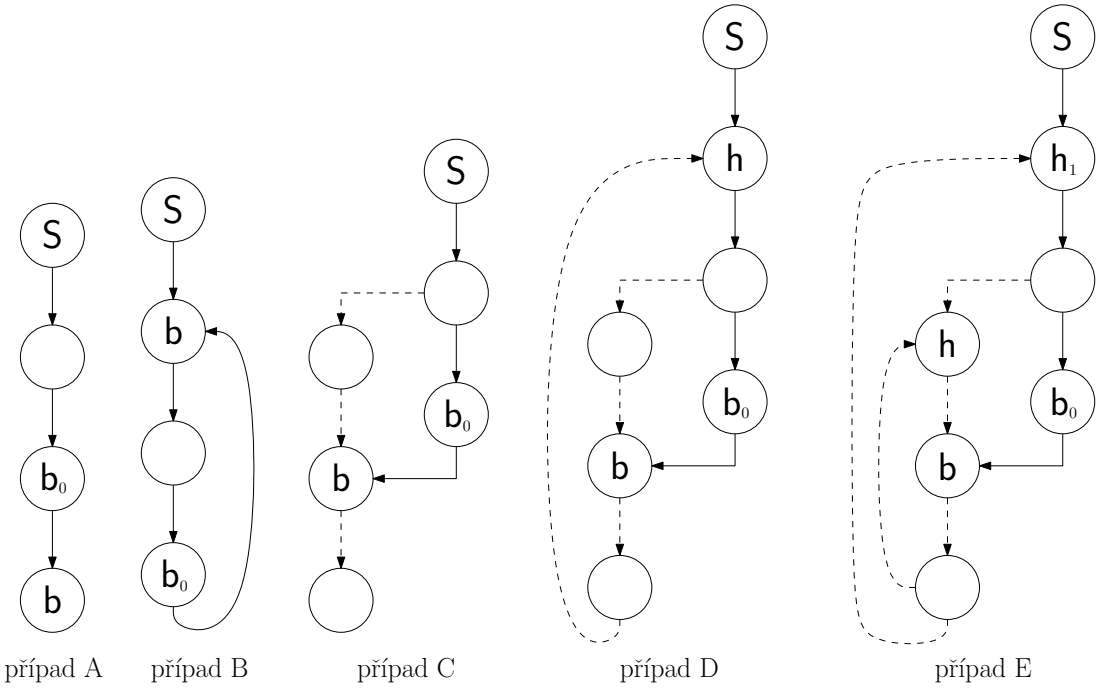
V prvním průchodu jsou identifikovány především cykly. Jsou označeny hlavičky cyklů a konce cyklů (tedy bloky, ze kterých vede zpětná hrana do hlavičky daného cyklu). U každého bloku je také poznačena hlavička cyklu, do kterého tento blok patří, případně fakt, že do žádného cyklu nepatří. Pokud je blok v cyklu, který je zanořen v dalších cyklech, potom je označena hlavička nejvnitřnějšího cyklu daného bloku. Pokud je více bloků, ze kterých vedou zpětné hrany do jedné hlavičky cyklu, potom pouze jeden blok je koncovým blokem daného cyklu a zpětná hrana ostatních těchto bloků je strukturována příkazem `continue`.

Dále jsou v prvním průchodu nalezeny bloky, které jsou hlavičkami konstrukcí `if/else` a `switch`. Jsou to bloky, které mají více následníků. Ne všechny tyto nalezené hlavičky

jsou hlavičkami kompletních konstrukcí. Některé tyto hlavičky mají například jednu větev ukončenou příkazem `continue`. Algoritmus dále identifikuje bloky, ve kterých se sbíhá více hran. V úvahu již nejsou brány zpětné hrany, ale pouze příčné. Tyto bloky jsou koncovými uzly hlavičkových uzlů konstrukcí `if/else` a `switch`. Pomocí pomocné funkce je k danému koncovému bloku nalezena a označena odpovídající hlavička.

První průchod také vyhledává větve CFG končící blokem s instrukcí `ret` a značí tento fakt do bloku, z něhož daná větev vychází. Pokud je nalezena příčná hrana, která vede do cyklu, v jehož těle není blok odkud hrana vychází, pak musí být tento tok řízení realizován pomocí příkazu `goto`. Do počátečního bloku této hrany je poznačeno, že přechod do daného následníka je proveden pomocí `goto` a také do hlavičky větve obsahující tuto hranu je uloženo, že je má větev ukončenou příkazem `goto`.

Funkce `getCFGStructure` prochází metodou DFS rekurzivně všechny uzly daného CFG. Jako parametr dostává blok (uzel grafu) určený ke zpracování a jeho hloubku v DFST. Vrací blok, který je hlavičkou nejvnitřnějšího cyklu zadaného bloku. Funkce označí zadaný blok jako zpracovaný a rekurzivně zpracuje jeho dosud nezpracované následníky. Obecně může nastat jeden z případů znázorněných na obrázku 5.6 a popsaných níže. Blok $b0$ je aktuálně zpracovávaný blok a blok b je jeho právě zkoumaný následník.



Obrázek 5.6: Případy strukturalizace CFG

V případě A je blok b dosud nezpracován a hrana $\langle b0, b\rangle$ je dopředná hrana. Zavoláme tedy rekurzivně funkci `getCFGStructure` na blok b a v případě, že se nám z této funkce vrátí odkaz na blok, který je hlavičkou cyklu uzlu b, pak tuto hlavičku označíme jako hlavičku cyklu uzlu $b0$ (uzel $b0$ je v těle cyklu s touto hlavičkou).

V případě B je blok b již zpracován a hrana $\langle b0, b\rangle$ je zpětná hrana. To znamená, že blok b je hlavičkou cyklu a blok $b0$ je v těle tohoto cyklu a je jeho koncovým blokem.

V případě C je blok b již zpracován a hrana $\langle b0, b\rangle$ je příčná hrana. To znamená, že uzel b spojuje více větví CFG, tedy je koncovým blokem konstrukce `if` (respektive `switch`). Ani blok $b0$, ani b nejsou v těle cyklu.

V případě D je blok b již zpracován a hrana $\langle b0, b\rangle$ je příčná hrana. To znamená, že uzel b spojuje více větví CFG, tedy je koncovým blokem konstrukce `if` (respektive `switch`). Blok b má již označenou hlavičku cyklu, jehož je součástí. Blok $b0$ je také součástí tohoto cyklu a proto je označen jako součást cyklu s touto hlavičkou.

V případě E je blok b již zpracován a hrana $\langle b0, b\rangle$ je příčná hrana. Hlavička nejvnitřnějšího cyklu bloku b neleží v DFSP($b0$). To znamená, že hrana $\langle b0, b\rangle$ je dalším vstupem do tohoto cyklu (angl. *re-entry edge*), kterou lze vytvořit pouze použitím příkazu `goto`. Celý tento případ může, tak jako na obrázku, být součástí jiného cyklu, nebo také nemusí.

Druhý průchod CFG označí u všech bloků hlavičku nejvnitřnější konstrukce `if/else`, respektive `switch`, v jejímž těle se blok nachází. Také označí první blok za koncovým blokem cyklu, který nepatří do těla tohoto cyklu (je to následník konce cyklu, který není koncem zpětné hrany). Tento blok je třeba znát při generování BIR reprezentace. Označeny jsou také větve končící příkazem `continue`, větve končící příkazem `break` a hrany, které je třeba generovat s použitím příkazu `goto` a větve, které těmito hranami končí.

```
1  void tagInnermostBranchHeader(block b0, block h)
2    označit b0 jako zpracovaný uzel;
3    označit h jako hlavičku větve, v níž leží b0;
4    if(b0 je koncový uzel cyklu)
5      označit následníka b0, který není zpětnou hranou, jako první
           blok po cyklu s hlavičkou b0.innermostLoopHeader;
6    if(b0 končí příkazem continue)
7      označit, že hlavička větve b0 má větev ukončenou continue;
8    foreach(block b in successors(b0))
9      if(<b0, b> je zpětná hrana)
10       continue;
11     if(uzel b již byl zpracován)
12       nalézt a označit společnou hlavičku větví b0 a b;
13     if(b vede z cyklu b0 do jiného bloku, než je první blok za
           cyklem || b vede do těla jiné konstrukce if, resp. switch)
14       označit hranu <b0, b> jako hranu s goto;
15       označit, že hlavička větve b0 má větev ukončenou goto;
16     if(b vede z cyklu b0 do prvního bloku za cyklem && b0 není
           koncový uzel cyklu)
17       označit b0 jako blok s break;
18       označit, že hlavička větve b0 má větev ukončenou break;
19     if(b0 je hlavičkou kompletní konstrukce if, resp. switch)
20       tagInnermostBranchHeader(b, b0);
21     else if(b0 je koncem konstrukce if, resp. switch)
22       tagInnermostBranchHeader(b, b0.innermostBranchHeader.
             innermostBranchHeader);
23     else
24       tagInnermostBranchHeader(b, h);
```

Obrázek 5.7: Pseudokód druhého průchodu CFG

Funkce `tagInnermostBranchHeader` prochází metodou DFS rekurzivně všechny uzly daného CFG. Přijímá dva parametry a nevrací žádnou hodnotu. Prvním parametrem je odkaz na zpracovávaný blok a druhý odkaz na blok, který máme zpracovávanému bloku přiřadit jako nejvnitřnější hlavičku větve.

Pokud je zpracovávaný blok $b0$ koncovým uzlem cyklu a má následníka b, který není koncem zpětné hrany $\langle b0, b\rangle$, potom je blok b označen jako první blok po cyklu, jehož hlavičkou je hlavička nejvnitřnějšího cyklu bloku $b0$. Pokud blok $b0$ končí příkazem `continue`, pak označíme, že větev, ve které blok $b0$ leží, končí příkazem `continue`.

Poté jsou prozkoumáni všichni následníci b bloku $b0$. Pokud je hrana $\langle b0, b\rangle$ zpětná hrana, potom je další zpracování následníka b přeskočeno. Jestliže už byl uzel b zpracován (je koncový uzel konstrukce `if`, respektive `switch`), potom označíme uzel b jako koncový uzel hlavičky, ze které vycházejí všechny větve, které se v uzlu b sbíhají.

Pokud hrana $\langle b0, b\rangle$ vede ven z cyklu do bloku, který není prvním blokem po konci tohoto cyklu, pak se jedná o hranu tvořenou příkazem `goto`. Také pokud hrana $\langle b0, b\rangle$ vede do těla jiné konstrukce `if`, respektive `switch`, jedná se o hranu tvořenou příkazem `goto`. V obou případech je ještě označeno, že větev, v níž leží blok $b0$, končí příkazem `goto`.

Pokud hrana $\langle b0, b\rangle$ vede ven z cyklu do bloku, který je prvním blokem po konci tohoto cyklu a zároveň blok $b0$ není koncovým blokem tohoto cyklu, pak se jedná o hranu tvořenou příkazem `break`. Větev, v níž leží blok $b0$, je označena jako větev končící příkazem `break`. V praxi je třeba ještě kontrolovat, zda nechceme použít příkaz `break` pro ukončení vykonávání cyklu v těle konstrukce `switch`. V této konstrukci má příkaz `break` jinou funkci a je proto třeba nahradit takovéto výskyty příkazu `break` příkazem `goto`.

Kapitola 6

Implementace

Hlavní část implementace konvertoru je řešena ve třídě `StructLLVMIR2BIRConverter` (viz sekce 6.1). Použity jsou také implementace jednotlivých řídicích příkazů ve vnitřní reprezentaci BIR (viz sekce 6.2). Vzhledem k tomu, že cílem práce bylo vytvořit pouze strukturu překládaného kódu, je část konvertoru, která se stará o zbytek převodu LLVM IR na BIR, převzata z původní implementace (viz 6.1.4). Také nejsou generovány všechny příkazy překládaného kódu, ale pouze ty, které určují strukturu programu, tedy `if`, `if/else`, `switch`, `break`, `continue`, `return`, `goto` a cykly. Mezi těmito příkazy jsou pouze v komentářích uvedeny návěští bloků z LLVM IR, jejichž instrukce (kromě ϕ instrukcí a ukončujících instrukcí) by měly být v tomto místě vygenerovány.

6.1 Implementace konvertoru

V této části je popsána implementace konvertoru zajišťující strukturalizaci kódu. Strukturalizace probíhá v rámci konverze těla dané funkce v metodě `visitAndAddFunction` a probíhá ve třech fázích. V prvních dvou fázích dojde k průchodům CFG dané funkce, tak jak byly popsány v sekci 5.2, konkrétně v sekci 5.2.2. Ve třetí fázi je pak na základě informací získaných prvními dvěma průchody vytvořena BIR reprezentace struktury dané funkce. Průběh jednotlivých fází je stručně popsán v následujících sekcích. Pro další detaily implementace je vhodné projít přímo zdrojový kód.

Informace o každém základním bloku se ukládají do struktury typu `BBInfoStruct`, která má v sobě ukazatel na tento základní blok, ukazatele na hlavičku nejvnitřnějšího cyklu, konec nejvnitřnějšího cyklu, první blok za koncem cyklu, hlavičku začátku větve CFG, ve které se daný blok nachází, a ukazatel na konec této větve. Pro větve konstrukce `switch` má nastaven ukazatel na hlavičku této konstrukce. Poté tato struktura obsahuje proměnné typu boolean nesoucí informace, zda je blok hlavičkou cyklu, hlavičkou větve, koncem cyklu, koncem větve, je hlavičkou a má větev ukončenou příkazem `break`, `continue`, `goto`, nebo `return` a další. Je zde také uložen ukazatel na ukončující instrukci daného bloku a číslo udávající hloubku zanoření bloku v daném CFG. Součástí struktury je také vektor s ukazateli na následníky daného bloku a ukazatel na začátek bloku instrukcí, které tvoří daný blok v reprezentaci BIR.

6.1.1 První průchod

První průchod je zajištěn voláním metody `getCFGStructure`. Činnost této metody je naznačena v pseudokódu na obrázku 5.2.2. Tato metoda dostane při prvním volání strukturu

s informacemi o prvním základním bloku zpracovávané funkce. U tohoto bloku uloží ukazatel na ukončující instrukci, uloží její typ a počet následníků bloku a ty poté jeden podruhém zpracovává. Všechny dvojice základní blok a jeho informační struktura jsou uloženy do instanční proměnné typu `std::map`. Před tvorbou nové informační struktury pro daný blok je nejdříve v této mapě zkontrolováno, jestli už pro tento blok informační struktura neexistuje a pokud ano, je vrácen ukazatel na ni. Pokud ne, je vytvořena nová struktura a dvojice blok, struktura vložena do mapy.

Následník je zpracován na základě toho, o který případ z obrázku 5.6 se jedná. Všechny získané informace jsou ukládány do informační struktury zpracovávaného bloku. Pokud je blok koncovým uzlem cyklu a jeho následník hlavičkou tohoto cyklu, pak je volána metoda `tagLoopHeader`, která označí hlavičku cyklu k danému bloku tak, aby bylo zachováno správné pořadí zanoření hlaviček cyklů. Pokud je blok koncovým uzlem větve, pak je volána metoda `tagBranchHeader`, která označí tento koncový uzel ke správné hlavičce konstrukce, ze které spojující se větve vycházejí.

Poměrně složitým problémem je nalezení správného konce konstrukce `switch`. Prvním následníkem konstrukce `switch` je vždy blok, který je buď začátkem větve `default` nebo je hledaným koncovým blokem dané konstrukce `switch`, tedy prvním blokem, přes který jde tok řízení ze všech větví dané konstrukce `switch` (všechny větve se zde scházejí). Problém je, že z LLVM IR nelze poznat, o který z těchto dvou případů se jedná. Pokud je v původním kódu, který zpětně překládáme, v konstrukci `switch` větev `default` bez příkazu `break` a je následována alespoň jednou další větví `case` (tzn. je tzv. *fall-through*), pak první následník instrukce `switch` v LLVM IR není hledaným koncovým blokem. Koncovým blokem je v takovém případě poslední větev `case`, kterou je potřeba správně identifikovat. Nebo pokud některá větev `case` končí příkazem `break` a větev `default` existuje, pak také nemusí být vykonána a koncový blok konstrukce `switch` je jiný, než je první následník této instrukce v LLVM IR.

6.1.2 Druhý průchod

Druhý průchod je proveden voláním metody `tagInnermostBranchHeader`. Opět se jedná o průchod, jehož princip byl popsán v sekci 5.2.2. Metoda má dva parametry, prvním je odkaz na informační strukturu zpracovávaného bloku a druhým odkaz na hlavičku větve, kterou chceme k danému bloku označit jako hlavičku, do jejíž větve blok patří. Při prvním volání má tedy druhý parametr hodnotu `NULL`, protože první blok funkce se nenalézá v žádné konstrukci `if` nebo `switch`. Rekurzivně zpracujeme další bloky. Pokud je blok hlavičkou konstrukce `if` nebo `switch` (má více následníků), pak jako druhý parametr pro tyto následníky je předán tento blok. Pokud je naopak blok koncovým uzlem konstrukce, pak jeho následník má stejnou hlavičku, jako hlavička dané konstrukce (vynoříme se o jednu úroveň výš). Pokud má blok jednoho následníka a není koncem konstrukce, pak je hlavička následníka stejná jako hlavička bloku.

Také jsou v této fázi nalezeny a uloženy informace o prvních blocích po skončení cyklu, a větvích končících příkazy `continue`, `break`, `goto` nebo `return`.

6.1.3 Třetí průchod

V tomto průchodu dochází na základě získaných informací ke generování BIR reprezentace překládaného kódu. Opět se jedná o rekurzivní průchod CFG. Je zařizován následujícími metodami:

- `generateStmts` – Základní metoda pro generování BIR reprezentace překládaného kódu. Je volána pro první blok dané funkce a poté rekurzivně i pro další bloky. Metoda prozkoumá blok a pokud je tento blok hlavičkou cyklu, zavolá na něj metodu `visitLoop`. Pokud blok není hlavičkou cyklu, ale je hlavičkou konstrukce `if`, respektive `switch`, pak je na něj zavolána metoda `visitIfStmt`, respektive `visitSwitchStmt`. Pokud blok není hlavičkou, pak je na něj zavolána metoda `visitBasicBlock`. Metoda vrací ukazatel na první příkaz BIR reprezentace dané funkce.

- `visitLoop` – Metoda je volána, pokud je blok hlavičkou cyklu. Jsou vygenerovány příkazy daného bloku. Pokud je blok zároveň hlavičkou konstrukce `if`, respektive `switch`, potom je další zpracování řešeno metodami `visitIfStmt`, respektive `visitSwitchStmt`. Pokud je jen hlavičkou cyklu, je třeba zkontrolovat, zdali blok není zároveň koncem cyklu. Pokud ano, pak je tělo cyklu tvořenou pouze tímto jedním blokem a je třeba vygenerovat koncovou podmínku a `break` pro ukončení cyklu. Pokud blok není zároveň koncem cyklu, pak je pomocí metody `visitBranchOrReturn` vygenerováno tělo cyklu.

- `visitSwitchStmt` – Metoda je volána, pokud je blok hlavičkou konstrukce `switch`. Je nalezena řídicí proměnná a generovány jednotlivé větve `case` a pokud existuje, tak na správné místo mezi větve `case` také větev `default`. Příkazy koncového bloku této konstrukce jsou označeny jako následník dané konstrukce `switch`.

- `visitIfStmt` – Metoda je volána, pokud je blok hlavičkou konstrukce `if`. Pokud má tato konstrukce i koncový blok, pak jsou vygenerovány podmíněné větve a první příkaz koncového bloku je označen jako následník dané konstrukce `if`. Pokud hlavička koncový blok nemá, pak má pouze jednu podmíněnou větev končící některým z příkazů `continue`, `break`, `goto` nebo `return`. Je tedy vygenerována podmínka a tato větev. Jako následník konstrukce `if` je poté označena druhá větev.

- `visitBasicBlock` – Metoda se volá pro zpracování bloků, které mají pouze žádného, nebo jednoho následníka (případně dva, ale jeden tvoří zpětnou hranu). Metoda vygeneruje příkazy pro daný blok a pokud končí `continue`, `break`, `goto` nebo `return`, pak i tento příkaz. Pokud má blok následníka, pak na něj zavolá metodu `generateStmts`.

- `visitBranchOrReturn` – Metoda přijímá tři parametry. Prvním je blok ke zpracování. Druhým je hledaný koncový blok větve a třetím hledaný koncový blok cyklu. Pokud je zadaný blok hledaným koncem větve, pak vrátí `NULL` a ukončí tak procházení této větve. Pokud končí zadaný blok příkazem `return`, pak je tento vygenerován pomocí metody `generateReturnStmt` a vrácen.

- `generateBlockStmts` – Vygeneruje prázdný příkaz s metadaty nesoucími jméno návěští zadaného základního bloku. Vrací ukazatel na tento prázdný příkaz. V této metodě by měly být do budoucna generovány příkazy daného základního bloku.

- `generateLoopEndBreakStmt` – Metoda pro generování podmínky `if` a příkazu `break` pro ukončení cyklu v místě koncového bloku daného cyklu. Vrací ukazatel na vygenerovanou konstrukci `if`. Tato metoda je redundantní v případě, že budou podmínky ukončení cyklu nastaveny přímo v příkazu cyklu.

- `generateContinueStmt` – Vygeneruje a vrátí odkaz na příkaz `continue`.

- `generateBreakStmt` – Vygeneruje a vrátí odkaz na příkaz `break`.
- `generateEmptyGotoStmt` – Metoda přijímá parametr typu `BBInfoStruct`, který je informační strukturou bloku, na který má daný skok vést. Vygeneruje příkaz `goto` a jako jeho cíl zadá prázdnou instrukci, protože BIR reprezentace cíle skoku nemusí být v daný moment ještě vygenerována. Proto metoda vloží dvojici odkaz na daný příkaz `goto` a informační struktura cíle skoku do instanční proměnné typu `std::map` a cíle skoků jsou nastaveny až když je BIR reprezentace kompletně vygenerována. Vrací odkaz na vygenerovaný příkaz `goto`.
- `generateReturnStmt` – Vygeneruje příkaz `return` s návratovou hodnotou, nebo bez ní, pokud je funkce typu void. Vrací ukazatel na vygenerovaný příkaz.

Po skončení tohoto průchodu je ještě zavolána funkce `setGotoTargets`, která vygenerovaným příkazům `goto` zadá cíl jejich skoku. Cíl skoku není příkazu `goto` zadán přímo, protože v době generování příkazu nemusí být znám (viz výše). Seznam příkazů `goto` a cíle jejich skoků jsou uloženy v instanční proměnné typu `std::map`. Pokud cíl skoku stále není znám (cíl je v CFG přístupný pouze příkazem `goto`), pak je tato větev vygenerována v tomto místě.

Ve všech případech, kdy má zpracovávaný následník více předchůdců, je třeba kontrolovat, zdali není potřeba generovat do aktuálního bloku kopie ϕ instrukcí, aby byly v následníkovy k dispozici správné hodnoty proměnných. Generování kopií ϕ instrukcí zařizuje metoda `getPHICopiesForSuccessor`. Je třeba také dávat pozor na zvláštní případy, jako například když generujeme kopie ϕ instrukcí v hlavičce konstrukce `if`, kdy následník hlavičky je přímo koncový uzel dané konstrukce (tato konstrukce `if` má pouze jednu podmíněnou větev). V takovém případě je třeba přidat větev, ve které budou tyto kopie ϕ instrukcí, protože pokud by byly vygenerovány přímo do hlavičky konstrukce `if`, pak by jimi byla ovlivněna i druhá větev, což je nežádoucí.

6.1.4 Převzaté části

Některé funkce a podpůrné třídy byly převzaty z originální implementace konvertoru a jsou popsány v této části. Jedná se především o řešení ϕ instrukcí, některé podpůrné třídy pro převádění podmínek skoků na výrazy BIR (třída `LLVMConverter`) a práci s proměnnými (třída `VarsHandler`) a také o základní funkce konvertoru, které zajišťují rozdělení překládaného kódu na jednotlivé funkce a předávají pak výslednou BIR reprezentaci k dalšímu zpracování.

V LLVM IR jsou ϕ instrukce vyhodnocovány paralelně, nicméně v BIR je vyhodnocování postupné. Proto pokud je vyhodnocena jedna ϕ instrukce a poté další, jejíž výsledek je na této první závislý, dostaneme z první ϕ instrukce již novou hodnotu, což je nežádoucí. Je třeba závislé ϕ instrukce seřadit tak, aby byly vyhodnocovány v pořadí, kdy jsou nejdříve vyhodnoceny závislé ϕ instrukce a až poté ϕ instrukce, na kterých tyto instrukce závisí. Převzaté funkce zajišťující seřazení ϕ instrukcí jsou následující:

- `dependsOn` – Funkce přijímá dva parametry typu `llvm::PHINode` a vrací `true`, pokud první zadaná ϕ instrukce závisí na druhé, tedy výsledek první ϕ instrukce závisí na vyhodnocení druhé ϕ instrukce. Pokud na sobě ϕ instrukce v zadaném pořadí nezávisejí, vrací funkce `false`.
- `getPHINodes` – Funkce přijímá parametr typu `llvm::BasicBlock` a vrací vektor ϕ instrukcí nacházejících se v tomto bloku.

- `canBeOrdered` – Funkce přijímá jako parametr vektor ϕ instrukcí a vrací `true`, pokud je možné dané ϕ instrukce seřadit. To není možné v případě, kdy je jedna ϕ instrukce závislá na druhé a tato druhá na první. Závislost může být i tranzitivní. V tomto případě vrací funkce `false`.
- `isReachable` – Funkce přijímá dva parametry typu `llvm::PHINode` a jeden parametr typu `llvm::BasicBlock` a vrací `true`, pokud je druhá zadaná ϕ instrukce v daném základním bloku z první zadané ϕ instrukce dosažitelná. Pokud není, vrací `false`.
- `performOrderingOfDependentPHINodes` – Funkce přijímá jako první parametr vektor ϕ instrukcí a druhý parametr typu `llvm::BasicBlock` a provádí na základě vzájemných závislostí seřazení zadaných ϕ instrukcí v zadaném bloku.
- `orderDependentPHINodes` – Přetížená funkce přijímá buď parametr typu `llvm::BasicBlock`, nebo `llvm::Function`, nebo `llvm::Module` a nad zadaným základním blokem, respektive funkcí, respektive modulem provede seřazení ϕ instrukcí.

Více o těchto funkcích lze nalézt v komentářích ve zdrojovém kódu.

Převzaté metody zajišťující chod konvertoru v rámci zpětného překladače jsou následující:

- konstruktor
- `create` – Metoda zajišťuje vytvoření nové instance konvertoru.
- `getId` – Metoda vrací řetězcový identifikátor konvertoru.
- `convert` – Metoda provede konverzi modulu, který je zadán parametrem typu `llvm::Module` a vrátí výsledný modul.
- `getAddressFromLabelName` – Metoda přijímá jako parametr řetězec, který je návěštím základního bloku a vrací řetězec s adresou tohoto základního bloku, nebo pouze návěští, pokud se adresu bloku nepodaří získat.
- `visitAndAddFunctions` – Metoda pomocí volání metody `visitAndAddFunction` konvertuje všechny funkce zpracovávaného modulu.
- `visitAndAddGlobalVariables` – Metoda projde všechny globální proměnné a uloží je do výsledného modulu.
- `visitAndAddFunctionDeclarations` – Metoda projde pomocí volání metody `visitAndAddFunction` všechny funkce a uloží jejich deklarace do výsledného modulu.
- `visitAndAddFunction` – Metoda navštíví funkci zadanou parametrem typu `llvm::Function` a konvertuje ji do výsledného modulu. Pokud je zadán i druhý parametr typu boolean s hodnotou `true`, pak je generována pouze deklarace dané funkce. Pokud ne, pak je zpracováno i tělo funkce. V této části dochází v rámci konverze na BIR ke strukturalizaci kódu.
- `getFunctionParams` – Metoda vrací seznam parametrů funkce zadané parametrem typu `llvm::Function`.

Některé metody jsou převzaty kompletně, některé byly modifikovány pro potřeby třídy `StructLLVMIR2BIRConverter`.

Dále jsou použity metody:

- `addStatementToStatementBlock` – Metoda přidá blok příkazů zadaný prvním parametrem do bloku příkazů zadaného druhým parametrem na místo zadané třetím parametrem.
- `getPHICopiesForSuccessor` – Metoda přijímá dva parametry. Blok, pro který hledá závislé ϕ instrukce v zadaném následníkovi. Pokud je některá z ϕ instrukcí v následníkovi závislá na zadaném bloku, pak je tato možnost přiřazení hodnoty do dané proměnné uložena do bloku příkazů, který je vrácen. Tento blok příkazů je pak obvykle připojen na konec zadaného bloku, aby v případě, že bude tok řízení do následníka předán ze zadaného bloku, byly v následníkovi k dispozici správné hodnoty všech proměnných.
- `mergeStatements` – Metoda třídy `Statement`, která sloučí dva zadané bloky příkazů.

6.2 Třídy z vnitřní reprezentace zadní části

V této sekci jsou popsány jednotlivé třídy reprezentující jednotlivé řídicí konstrukce ve vnitřní reprezentaci BIR použité v rámci této práce. Tyto třídy nebyly implementovány v rámci této práce, jsou pouze využity při konverzi LLVM IR na BIR. Všechny uvedené třídy dědí od třídy `Statement`. V konvertoru jsou využity i některé další třídy, které však nereprezentují řídicí konstrukce a tak nejsou podrobněji popsány. Použita je například třída `Function`, která implementuje BIR reprezentaci funkce. Nebo třídy `ConstBool` a `ConstInt`, které reprezentují BIR implementaci konstantní proměnné typu boolean, respektive konstantní celočíselné proměnné. Využívána je také implementace sdíleného ukazatele `ShPtr<`*typ*`>`, kde *typ* je typ objektu, na který daný ukazatel odkazuje (například příkaz `Statement` nebo výraz `Expression`).

6.2.1 BreakStmt

Implementace příkazu `break`. Příkaz je vytvořen voláním metody `create` bez parametrů.

6.2.2 ContinueStmt

Implementace příkazu `continue`. Příkaz je vytvořen voláním metody `create` bez parametrů.

6.2.3 EmptyStmt

Implementace prázdného příkazu. Příkaz je vytvořen voláním metody `create`, která má parametr typu `ShPtr<Statement>` odkazující na příkaz následující za tímto prázdným příkazem. Parametr není povinný. V konvertoru je prázdný příkaz používán tak, že jsou mu pomocí metody `setMetadata`, zděděné ze třídy `Statement` přiřazena metadata, pomocí kterých jsou vypisovány dodatečné informace na výstupu.

6.2.4 GotoStmt

Implementace příkazu `goto`. V implementaci konvertoru jsou používány metody `create` a `setTarget`. Metoda `create` vytvoří novou instanci třídy `GotoStmt`. Přijímá jeden parametr typu `ShPtr<Statement>`, který odkazuje na příkaz, na který vede daný skok. Vzhledem k tomu, že ve chvíli kdy jsou instance této třídy v konvertoru vytvářeny nemusí být instrukce, na kterou skok vede, ještě vytvořena, je parametr metody `create` při volání instance třídy `EmptyStmt`.

Skutečný cíl skoku je zadán metodou `setTarget` až poté, co jsou vygenerovány všechny příkazy dané funkce a je tedy jisté, že i cíl pro daný skok je již vygenerován. Metoda `setTarget` také přijímá parametr typu `ShPtr<Statement>`.

6.2.5 IfStmt

Implementace konstrukce `if`, respektive `if/else`. V implementaci konvertoru jsou používány metody `create` a `setElseClause`. Metoda `create` vytvoří novou instanci třídy `IfStmt`. Přijímá tři parametry. První parametr typu `ShPtr<Expression>` určuje podmínku konstrukce `if`, druhý a třetí parametr jsou typu `ShPtr<Statement>` a odkazují na první příkaz těla konstrukce `if`, respektive na první příkaz za danou konstrukcí `if`.

Metoda `setElseClause` přijímá jeden parametr typu `ShPtr<Statement>`, který odkazuje na první příkaz větve `else`, kterou chceme k dané konstrukci `if` připojit.

6.2.6 ReturnStmt

Implementace příkazu `return`. Příkaz je vytvořen voláním metody `create` se dvěma parametry. Jeden je typu `ShPtr<Expression>` a jedná se o výraz určující návratovou hodnotu daného příkazu `return`. Druhý parametr typu `ShPtr<Statement>` odkazuje na první příkaz následující za tímto příkazem `return`. Ani jeden z parametrů není povinný.

6.2.7 SwitchStmt

Implementace konstrukce `switch`. V implementaci konvertoru jsou používány metody `create`, `addClause` a `addDefaultClause`. Metoda `create` vytvoří novou instanci třídy `SwitchStmt`. Přijímá dva parametry. První parametr typu `ShPtr<Expression>` určuje řídicí proměnnou a druhý parametr typu `ShPtr<Statement>` odkazuje na první příkaz za danou konstrukcí `switch`.

Metoda `addClause` připojí k dané konstrukci `switch` novou větev `case`. Přijímá dva parametry. První je typu `ShPtr<Expression>` a určuje hodnotu, kterou má nabývat řídicí proměnná, aby byla tato větev `case` vykonána. Druhý je typu `ShPtr<Statement>` a odkazuje na první příkaz větve `case`.

Metoda `addDefaultClause` přijímá jeden parametr typu `ShPtr<Statement>`, který odkazuje na první příkaz větve `default`, kterou chceme k dané konstrukci `switch` připojit.

6.2.8 WhileLoopStmt

Implementace konstrukce `while`. V implementaci konvertoru je používána metoda `create`, která vytvoří novou instanci třídy `WhileLoopStmt`. Přijímá tři parametry. První parametr typu `ShPtr<Expression>` určuje podmínku při jejímž splnění je provedeno tělo cyklu, druhý a třetí parametr jsou typu `ShPtr<Statement>` a odkazují na první příkaz těla cyklu, respektive na první příkaz za danou konstrukcí `while`.

Kapitola 7

Testování

V této kapitole je popsána testovací sada použitá při kontrole funkčnosti vytvořeného konvertoru. Testy jsou prováděny pomocí shellového skriptu, převzatého z již existujících testovacích sad ke zpětnému překladači a modifikovaného pro potřeby otestování nového konvertoru. Testy jsou z větší části krátkými programy, zaměřenými pouze na to, aby otestovaly zpětný překlad CFG určité struktury, případně různých kombinací a zanoření těchto jednoduchých struktur. Je zde ale také několik příkladů větších programů, které mají za cíl otestovat konvertor na reálných aplikacích.

Zpětný překlad je prováděn ze souboru s příponou `.ll` obsahujícího kód programu v jazyce LLVM IR. Tento soubor je získán přeložením původního kódu v jazyce C pomocí zpětného překladače přímo do jazyka LLVM IR. Překlad do LLVM IR by šel provést i přes překlad do binárního kódu a následný zpětný překlad do LLVM IR, ovšem tato varianta není použita, protože `.ll` soubory generované z binárního kódu jsou méně přehledné, než soubory získané přímo překladem z C. Například identifikátory proměnných a návěští jsou při generování LLVM IR z binárního kódu odlišeny hexadecimálním číslem, které v binárním kódu určuje jejich adresu. Pokud však chceme LLVM IR reprezentace manuálně zkoumat, jsou tyto identifikátory obtížně rozlišitelné a je proto použito generování LLVM IR přímo z původního zdrojového kódu.

Celkem obsahuje testovací sada 34 testů. Všechny zdrojové soubory v jazyce C, reprezentace v jazyce LLVM IR a referenční výstupy v jazyce Python' a C jsou umístěny v adresáři `testsuite`. 27 testů jsou krátké programy zaměřené na testování překladu určité konstrukce. Zbylých 7 testů v adresáři `largerFiles` bylo převzato z již existujících testovacích sad zpětného překladače a jejich cílem je otestovat, zda zvládne konvertor úspěšně projít i rozsáhlejší programy. Testy jsou spouštěny s následujícími parametry:

- `--backend-unify-labels` – Zajišťuje sjednocení názvů návěští. Z neznámých důvodů u testu `IZP-proj4` některé názvy návěští nepřevede na jednotný tvar a proto je u tohoto testu vypsána při porovnávání výstupů s referenčními výstupy chyba.
- `--backend-no-time-varying-info` – Zakazuje výpis metadat s časovými údaji.
- `--backend-no-debug` – Zakazuje výpis informací o průběhu překladu.
- `--backend-no-opts` – Vypnutí optimalizací v zadní části.
- `--backend-llvmir2bir-converter=struct` – Použití implementovaného konvertoru ke konverzi LLVM IR na BIR.

- `--keep-unreachable-funcs` – Tento parametr je použit jen u testů z adresáře `largerFiles` a zajišťuje, že funkce, které jsou nedostupné, jsou zachovány. Implicitně jsou nedostupné funkce z výsledného kódu vymazány, ale protože konvertor negeneruje příkazy volání funkcí, jsou všechny funkce mimo `main` nedostupné a jejich kód tak standardně není ve výsledném souboru.

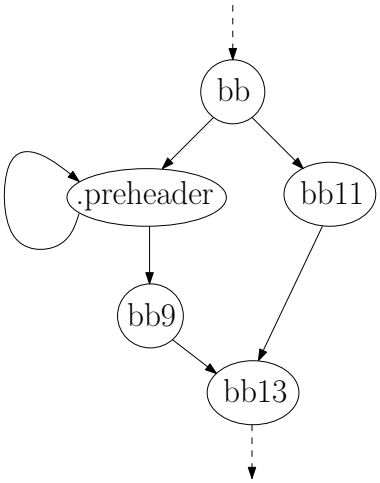
Většina parametrů sjednocuje tvar výstupu testů kvůli automatickému porovnání s referenčními výstupy. Optimalizace jsou vypnuty, protože jinak by ve výstupním kódu, který tvoří pouze příkazy vytvářející strukturu programu, byly některé části odstraněny. Očekávaný tvar výstupního kódu je znázorněn na obrázku 4.1 a popsán v kapitole 4. Zdrojový kód tohoto příkladu je na obrázku 3.2. Níže jsou uvedeny některé vybrané testy, konkrétně jejich zdrojové kódy, CFG a výstupy.

ifElseWithNestedLoop

Program s cyklem v jedné větvi konstrukce `if`.

```
1  int main()
2  {
3      srand(time(0));
4      int a = rand() % 2;
5      if(a == 1) {
6          for(int i=0; i<10; ++i) {
7              a = rand() % 2;
8              printf("%d\n", a);
9          }
10         printf("After loop.");
11     }
12     else
13         printf("Else body.");
14     return 0;
15 }
```

Obrázek 7.1: Zdrojový kód testu `ifElseWithNestedLoop`



Obrázek 7.2: CFG testu `ifElseWithNestedLoop`

```
1  int main() {
2      // bb
3      if (apple % 2 == 1) {
4          while (true) {
5              // .preheader
6              if (banana == 10) {
7                  break;
8              }
9              lemon = banana;
10         }
11         // bb9
12         return 0;
13     } else {
14         // bb11
15         return 0;
16     }
17 }
```

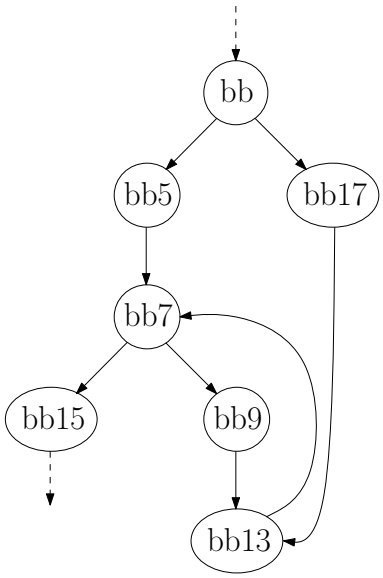
Obrázek 7.3: Výsledný kód testu `ifElseWithNestedLoop`

gotoToLoop

Program používající příkaz `goto` pro skok do těla cyklu. Konkrétně je příkazem `goto` realizována hrana $\langle bb17, bb13 \rangle$ z grafu na obrázku 7.5.

```
1  int main() {
2      srand(time(0));
3      int a = rand() % 2, i;
4      if(a == 1) {
5          printf("Before loop.\n");
6          for(i=0; i<10; ++i) {
7              a = rand() % 2;
8              printf("%d\n", a);
9              label: ;
10         }
11         printf("After loop.\n");
12     }
13     else {
14         printf("Using goto.\n");
15         i = 5;
16         goto label;
17     }
18     return 0;
19 }
```

Obrázek 7.4: Zdrojový kód testu `gotoToLoop`



Obrázek 7.5: CFG testu `gotoToLoop`

```
int main() {
    // bb
    if (apple % 2 != 1) {
        // bb17
        banana = 5;
        goto label1;
    }
    // bb5
    lemon = 0;
    while (true) {
        // bb7
        if (lemon >= 10) {
            // bb15
            return 0;
        }
        // bb9
        banana = lemon;
      label1:
        // bb13
        lemon = plum;
    }
}
```

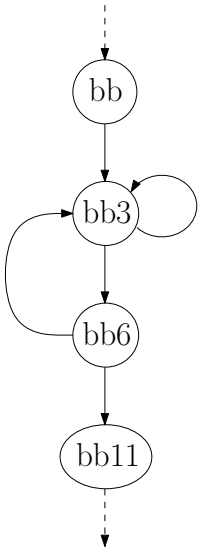
Obrázek 7.6: Výsledný kód testu `gotoToLoop`

loopWithContinue

Program používající příkaz `continue`. Konkrétně je příkazem `continue` realizována zpětná hrana $\langle bb3, bb3 \rangle$ z grafu na obrázku 7.8.

```
int main()
{
    srand(time(0));
    int a;
    for(int i=0; i<10; ++i) {
        if(i == 5)
            continue;
        a = rand() % 2;
        printf("%d", a);
    }
    printf("After loop.");
    return 0;
}
```

Obrázek 7.7: Zdrojový kód testu `loopWithContinue`



Obrázek 7.8: CFG testu `loopWithContinue`

```
1  int main() {
2      // bb
3      apple = 0;
4      while (true) {
5          // bb3
6          if (apple == 5) {
7              apple = 6;
8              continue;
9          }
10         // bb6
11         if (banana == 10) {
12             break;
13         }
14         apple = banana;
15     }
16     // bb11
17     return 0;
18 }
```

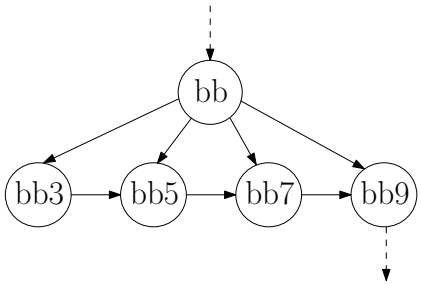
Obrázek 7.9: Výsledný kód testu `loopWithContinue`

switchWithDefaultAtTheEndNoBreak

Program s konstrukcí `switch` bez příkazů `break` a větví `default` na konci. Vzhledem k tomu, že větev `default` je provedena vždy, je tato větev zrušena a její kód přesunut za konstrukci `switch` (viz výsledný kód na obrázku 7.12).

```
1  int main()
2  {
3      srand(time(0));
4      int a = rand() % 10;
5      switch(a) {
6          case 0:
7              printf("case 0\n");
8          case 1:
9              printf("case 1\n");
10         case 2:
11             printf("case 2\n");
12         default:
13             printf("default\n");
14     }
15     printf("After switch.");
16     return 0;
17 }
```

Obrázek 7.10: Zdrojový kód testu `switchWithDefaultAtTheEndNoBreak`



Obrázek 7.11: CFG testu `switchWithDefaultAtTheEndNoBreak`

```
1  int main() {
2      // bb
3      switch (x % 10) {
4          case 0: {
5              // bb3
6          }
7          case 1: {
8              // bb5
9          }
10         case 2: {
11             // bb7
12             break;
13         }
14     }
15     // bb9
16     return 0;
17 }
```

Obrázek 7.12: Výsledný kód testu `switchWithDefaultAtTheEndNoBreak`

Kapitola 8

Závěr

Tato kapitola shrnuje dosažené výsledky a navrhuje možná rozšíření nebo vylepšení implementace strukturalizačního algoritmu.

Cílem práce bylo nastudovat problematiku zpětného inženýrství, seznámit se se zpětným překladačem projektu Lissom, jazykem LLVM IR použitým k vnitřní reprezentaci při zpětném překladu a implementací vnitřní reprezentace BIR. Touto části se zabývají kapitoly 2 a 3. Dalším bodem zadání byl návrh algoritmu provádějícího strukturování kódu. Tento algoritmus byl na základě nastudovaných materiálů navržen v kapitole 5. Kapitola 6 se pak zabývá implementací tohoto algoritmu v rámci zpětného překladače projektu Lissom. Při vývoji byla také využívána postupně se rozšiřující sada testů popsaných v kapitole 7. Na základě výsledků testů byly odladěny nalezené chyby v implementaci.

V současné implementaci existuje několik míst, která by šla rozšířit nebo vylepšit. Jedním ze slabých míst algoritmu je použití rekurze, které by při zpětném překladu větších souborů mohlo způsobovat problémy s nedostatkem paměti na zásobníku.

Druhá možnost vylepšení je při identifikaci cyklů. Algoritmus vytváří všechny cykly typu `while(true)` a ukončení provádění cyklu je řešeno příkazem `break` při splnění ukončující podmínky. Cykly tohoto typu jsou nyní transformovány na cykly `while`, respektive `for` s ukončující podmínkou pomocí existujících optimalizací v zadní části zpětného překladače. Bylo by však možné cykly generovat ve správném tvaru již při konverzi z LLVM IR na BIR a optimalizace cyklů v zadní části by šlo vynechat.

Další vylepšení by spočívalo ve snížení paměťové náročnosti pro ukládání informací o jednotlivých základních blocích. V současné implementaci jsou informace o každém bloku uloženy ve struktuře stejného typu bez ohledu na to, zda jsou pro daný blok všechny její položky potřebné, nebo ne.

Testovací sada by se dala rozšířit o další příklady různých CFG. Především u větších funkcí vznikají složité CFG a i přes návrh použitého algoritmu a použité testy nelze zcela vyloučit, že existují CFG, u kterých by měl algoritmus se strukturalizací problém. V případě chyby je třeba dohledat proč k ní dochází a slabé místo opravit.

Literatura

[1] EILAM, Eldad. *Reversing: Secrets of Reverse Engineering*. Indianapolis: Wiley, 2005. ISBN 07-645-7481-7.

[2] CIFUENTES, Cristina. *Reverse Compilation Techniques*. Brisbane, 1994. Dostupné z: `http://zyloid.com/recomposer/files/decompilation_thesis.pdf`. Dizertační práce. Queensland University of Technology.

[3] Decompiler Design. *Backer Street Software* [online]. 2011 [cit. 2013-11-05]. Dostupné z: `http://www.backerstreet.com/decompiler/introduction.htm`

[4] Česká republika. Zákon č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon). In: *Sbírka zákonů*. 2000, roč. 2000, 36, s. 1672. Dostupné z: `http://portal.gov.cz/app/zakony/zakon.jsp?q=121/2000&par=66`

[5] Boomerang: A general, open source, retargetable decompiler of machine code programs. *The Boomerang Decompiler Project* [online]. 2002–2006 [cit. 2014-05-19]. Dostupné z: `http://boomerang.sourceforge.net/`

[6] Reverse Engineering Compiler. *Backer Street Software* [online]. 1997–2012 [cit. 2013-11-12]. Dostupné z: `http://www.backerstreet.com/rec/rec.htm`

[7] *Hex-Rays* [online]. 2013 [cit. 2013-11-12]. Dostupné z: `http://www.hex-rays.com/`

[8] *SmartDec* [online]. 2013 [cit. 2013-11-12]. Dostupné z: `http://decompilation.info/`

[9] LISSOM. *Retargetable Decompiler* [online]. 2013 [cit. 2013-11-12]. Dostupné z: `http://decompiler.fit.vutbr.cz/`

[10] ĎURFINA, Lukáš, Jakub KŘOUSTEK, Petr ZEMEK, Dušan KOLÁŘ, Tomáš HRUŠKA, Karel MASAŘÍK a Alexander MEDUNA. Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. In: KIM, Tai-hoon, Hojjat ADELI, Rosslin John ROBLES a Maricel BALITANAS. *Information Security and Assurance: International Conference, ISA 2011, Brno, Czech Republic, August 15–17, 2011*. New York: Springer, 2011, s. 72–86. ISBN 978-3-642-23140-7.

[11] MASAŘÍK, Karel. *Systém pro souběžný návrh technického a programového vybavení počítačů*. Brno, 2008. Dostupné z: `http://www.fit.vutbr.cz/study/DP/PD.php?id=177&file=t`. Dizertační práce. FIT VUT v Brně.

[12] LLVM. LATTNER, Chris. *The Architecture of Open Source Applications* [online]. 2012 [cit. 2013-10-30]. Dostupné z: `http://www.aosabook.org/en/llvm.html`

[13] LLVM Language Reference Manual. *LLVM Project* [online]. 2003–2014 [cit. 2014-04-21]. Dostupné z: `http://llvm.org/docs/LangRef.html`

[14] CIFUENTES, Cristina. A Structuring Algorithm for Decompilation. In: *XIX Conferencia Latinoamericana de Informática* [online]. Buenos Aires, 1993 [cit. 2014-01-18]. Dostupné z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.3657&rep=rep1&type=pdf`

[15] WEI, Tao, Jian MAO, Wei ZOU a Yu CHEN. A New Algorithm for Identifying Loops in Decompilation. In: NIELSON, Hanne Riis a Gilberto FILÉ. *SAS'07 Proceedings of the 14th international conference on Static Analysis*. Heidelberg: Springer, 2007, s. 170–183. ISBN 3-540-74060-0.

[16] CIFUENTES, Cristina. Structuring Decompiled Graphs. In: *International Conference on Compiler Construction* [online]. Linköping, 1996 [cit. 2014-05-19]. Dostupné z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.8004&rep=rep1&type=pdf`