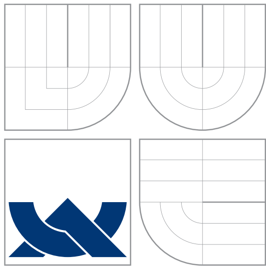
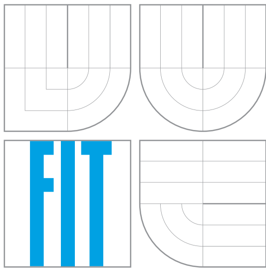


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

STRUKTUROVÁNÍ KÓDU V ZADNÍ ČÁSTI ZPĚTNÉHO PŘEKLADAČE

CODE STRUCTURING IN DECOMPILER BACK-END

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ PORWOLIK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETER MATULA

BRNO 2016

Zadání diplomové práce/18300/2015/xporwo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Porwolik Tomáš, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Strukturování kódu v zadní části zpětného překladače**
Code Structuring in Decompiler Back-End
Kategorie: Překladače

Pokyny:

1. Studujte problematiku zpětného inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se se zpětným překladačem společnosti AVG a jazykem LLVM IR, který je v překladači použit pro vnitřní reprezentaci dat. Dále se seznamte s vnitřní reprezentací použitou v zadní části zpětného překladače.
3. Navrhněte metodu strukturování vnitřní reprezentace, která se bude snažit eliminovat skoky (goto) s využitím podmíněných příkazů (if, switch) a cyklů (for, while).
4. Po konzultaci s vedoucím implementujte metodu navrženou v předchozím bodě.
5. Vytvořené řešení důkladně otestujte sadou minimálně dvaceti testů. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- Popis platformy LLVM [online]. 2015 [cit. 2015-08-19]. Dostupné na URL: <http://www.llvm.org>
- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- Interní dokumentace společnosti AVG.

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání a rozpracování čtvrtého bodu.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matula Peter, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá nástrojem pro zpětný překlad nízkoúrovňového strojového kódu do vyšší formy reprezentace, který je vyvíjen společností AVG Technologies. Cílem této práce je navrhnout a implementovat metodu strukturování vnitřní reprezentace v zadní části zpětného překladače, která se bude snažit eliminovat skoky s využitím podmíněných příkazů a cyklů. Je navržena metoda pro strukturování, která pracuje na základě opakovaného procházení grafu toku řízení a vyhledávání předdefinovaných vzorů. Ve všech případech však není možné strukturovat kód pouze s využitím podmíněných příkazů a cyklů. V takových případech je použito strukturování pomocí příkazu `goto`. Vytvořené řešení je srovnáno s původním řešením ve zpětném překladači. Dle výsledků je řešení rychlejší, lépe otestované, ale ve větším množství případů generuje nevalidní kód. Z hlediska strukturování jsou výsledky rozdílné a někdy je kód strukturován lépe, avšak někdy hůře.

Abstract

This thesis deals with a decompilation tool which converts low-level binary code to a high-level representation. This tool is being developed by AVG Technologies. The aim of this work is to design and implement a method for code structuring in the decompiler back-end. The designed method works by traversing the control-flow graph with matching of predefined patterns. It is not always possible to structure code using conditional statements and loops. Sometimes also `goto` statements must be used. The implemented solution is compared with the original solution in the decompiler. According to the results the new solution is faster, better tested, but in greater number of test cases generates invalid code. From the viewpoint of structuring the results are different and sometimes the code is structured better, but sometimes worse.

Klíčová slova

zpětné inženýrství, zpětný překlad, strukturování kódu, LLVM, AVG, BIR, goto

Keywords

reverse engineering, decompilation, code structuring, LLVM, AVG, BIR, goto

Citace

PORWOLIK, Tomáš. *Strukturování kódu v zadní části zpětného překladače.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Matula Peter.

Strukturování kódu v zadní části zpětného překladače

Prohlášení

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

....
Tomáš Porwolik
25. května 2016

Poděkování

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

© Tomáš Porwolik, 2016.
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

1 Úvod **3**

2 Zpětné inženýrství **5**
- 2.1 Definice . . . 5
- 2.2 Použití . . . 5
- 2.3 Postup provádění . . . 6
- 2.4 Používané nástroje . . . 6

3 Zpětný překladač společnosti AVG **8**
- 3.1 Struktura . . . 8
- 3.2 Předzpracování . . . 9
- 3.3 Přední část . . . 10
- 3.4 Optimalizační část . . . 11
- 3.5 Zadní část . . . 12

4 Jazyk LLVM IR **15**
- 4.1 Identifikátory . . . 15
- 4.2 Vysokoúrovňová struktura . . . 15
- 4.3 Typy . . . 17
- 4.4 Konstanty . . . 18
- 4.5 Instrukce . . . 18

5 Vnitřní reprezentace použitá v zadní části zpětného překladače **22**
- 5.1 Modul, funkce a bázové třídy . . . 22
- 5.2 Příkazy . . . 23
- 5.3 Výrazy . . . 24
- 5.4 Typy . . . 26

6 Rozbor současného stavu a stanovení cílů práce **27**
- 6.1 Popis současného konvertoru . . . 27
- 6.2 Stanovení cílů práce . . . 28

7 Návrh konvertoru **29**
- 7.1 Konvertor z LLVM IR do BIR . . . 30
- 7.2 Správce proměnných . . . 30
- 7.3 Konverze výrazů . . . 31
- 7.4 Konverze základního bloku . . . 33
- 7.5 Strukturování funkce . . . 34

8 Implementace konvertoru **39**
8.1 Třída `NewLLVMIR2BIRConverter` . . . 39
8.2 Třída `VariablesManager` . . . 39
8.3 Třída `LLVMValueConverter` . . . 39
8.4 Třída `LLVMTypeConverter` . . . 40
8.5 Třída `LLVMInstructionConverter` . . . 40
8.6 Třída `LLVMConstantConverter` . . . 41
8.7 Třída `BasicBlockConverter` . . . 41
8.8 Třída `StructureConverter` . . . 41

9 Testování **43**
9.1 Jednotkové testy . . . 43
9.2 Testy kvality . . . 44
9.3 Regresní testy . . . 45
9.4 Noční testy . . . 47

10 Zhodnocení **49**
10.1 Zhodnocení splnění cílů . . . 49
10.2 Srovnání výstupů nového a původního konvertoru . . . 50
10.3 Nalezené problémy ve zpětném překladači . . . 52

11 Závěr **53**

Literatura **54**

Přílohy **56**

A Ukázka zpětného překladu **57**

B Ukázka postupné redukce grafu toku řízení funkce **60**

Kapitola 1

Úvod

Při programování se typicky postupuje tak, že se zapíše kód programu v programovacím jazyce a tento kód se přeloží pomocí překladače na strojový kód zvolené platformy. Zpětné získání vysokoúrovňového kódu ze strojového kódu je však mnohem náročnější proces, jelikož při překladu na strojový kód dochází ke ztrátě velkého množství vysokoúrovňových konstrukcí. Proces zpětného získání vysokoúrovňového kódu se nazývá zpětný překlad a nástrojem pro jeho provedení je zpětný překladač. Výsledek zpětného překladu lze použít například pro analýzu škodlivých programů, a tedy pro tvorbu obranných prostředků proti nim.

Ve společnosti AVG Technologies (dále jen AVG) je vyvíjen rekonfigurovatelný zpětný překladač. Termín rekonfigurovatelný znamená, že je nezávislý na architektuře procesoru, pro kterou je možné provést zpětný překlad. Tento zpětný překladač dále podporuje více formátů vstupních souborů i více výstupních vysokoúrovňových jazyků.

Tento zpětný překladač se skládá ze čtyř základních částí. Jedná se o část předzpracování, přední část, optimalizační část a zadní část. Ve fázi předzpracování probíhá analýza vstupní aplikace pro získání informací o formátu spustitelného souboru, cílové architektuře nebo kódových a datových sekcích. Přední část slouží k dekódování instrukcí, detekci funkcí, datových typů a k mnoha dalším analýzám. Přední část generuje kód v jazyce LLVM IR, který je dále používán v optimalizační části a na vstupu zadní části. Optimalizační část provádí optimalizace kódu v jazyce LLVM IR a zadní část slouží především k vygenerování kódu v cílovém vysokoúrovňovém jazyce.

V rámci zadní části zpětného překladače je potřeba řešit strukturování kódu, což znamená převod z nízkoúrovňových instrukcí skoku na vysokoúrovňové konstrukce, jako jsou podmíněné příkazy a cykly. Jedná se o informace, které byly ztraceny při překladu programu a bez jejich přítomnosti by byl výsledný kód velmi obtížně čitelný.

Stávající řešení strukturování ve zpětném překladači společnosti AVG je však již zastaralé a obsahuje chyby ve strukturování. Mezi další nevýhody patří, že není založeno na žádném formálním algoritmu, a že není testované pomocí jednotkových testů.

Tato práce se zabývá konverzí kódu jazyka LLVM IR do vnitřní reprezentace používané zadní částí. Její součástí je převod nízkoúrovňových instrukcí skoku na vysokoúrovňové konstrukce, tedy podmíněné příkazy a cykly. Cílem práce je vytvořit nový konvertor, který bude založen na formálním algoritmu a bude řádně otestován.

Úkolem konvertoru je převést instrukce z jazyka LLVM IR na posloupnost příkazů vnitřní reprezentace používané zadní částí zvané zkráceně BIR. Konvertor musí řešit převod globálních proměnných, hlaviček všech funkcí a následně těl funkcí. V rámci konverze těl funkcí je potřeba dbát na čitelnost výsledného kódu.

Práce je rozvržena následovně. Po této úvodní kapitole následuje kapitola s popisem teoretického úvodu do zpětného inženýrství, které je zde definováno, jsou uvedeny příklady typického použití a nástroje, které se pro něj používají. Kapitola 3 představuje zpětný překladač vyvíjený společností AVG. Dále kapitola 4 popisuje jazyk LLVM IR, který je použit pro vnitřní reprezentaci v přední a optimalizační části zpětného překladače. Navazující kapitola 5 popisuje vnitřní reprezentaci používanou v zadní části zpětného překladače. V kapitole 6 je proveden rozbor současného stavu a jsou zde vytyčeny cíle práce.

V kapitole 7 je navrženo a popsáno řešení nového konvertoru, který řeší nedostatky současného konvertoru. Na tuto kapitolu navazuje kapitola 8, kde je popsána implementace navrženého konvertoru. Způsoby testování této implementace jsou uvedeny v kapitole 9. Zhodnocení konvertoru, který byl v rámci této práce navržen a implementován, je uvedeno v kapitole 10. Závěrečná kapitola 11 stručně shrnuje tuto práci a diskutuje budoucí vývoj.

Kapitola 2

Zpětné inženýrství

Tato kapitola se věnuje teoretickému úvodu do zpětného inženýrství. Je zde definován pojem zpětné inženýrství a dále popsány typické příklady jeho použití, jeho postup provádění a hlavní používané nástroje. Informace v této kapitole jsou čerpány z [9].

2.1 Definice

Zpětné inženýrství je proces získávání znalostí nebo konstrukčních plánů z čehokoliv vytvořeného člověkem. Myšlenka zpětného inženýrství vznikla již dlouho před příchodem počítačů, pravděpodobně v době průmyslové revoluce. Zpětné inženýrství je velmi podobné s vědeckým výzkumem. Rozdíl však spočívá v tom, že při vědeckém výzkumu je zkoumán přírodní jev, ale při zpětném inženýrství je zkoumán objekt vytvořený člověkem.

Zpětné inženýrství se obvykle provádí za účelem získání chybějících znalostí nebo konstrukčních plánů. Tyto informace mohou chybět, protože jsou ztracené, nebo protože jsou vlastnictvím někoho, kdo je neposkytne (např. konkurence).

Běžně se při zpětném inženýrství postupuje tak, že se fyzicky rozebere zkoumaný předmět a zjišťuje se, jak byl sestrojen. Získané informace jsou poté použity pro sestrojení podobného nebo lepšího výrobku.

Softwarové zpětné inženýrství funguje na stejném principu, ale jeho proces je pouze virtuální. Cílem je pochopení principu fungování programu nebo odhalení použitých algoritmů. Pro softwarové zpětné inženýrství je potřeba podrobných znalostí fungování počítače a vývoje software. Dále pak schopnost řešení šifer, programování a logické analýzy.

Následující sekce jsou zaměřeny pouze na softwarové zpětné inženýrství, jelikož ostatní odvětví zpětného inženýrství nejsou pro tuto práci podstatné.

2.2 Použití

Obecně se zpětné inženýrství používá nejčastěji pro vytvoření konkurenčního výrobku, ovšem při vývoji software tento přístup nemá smysl. Důvodem je příliš velká složitost software, takže by použití zpětného inženýrství bylo příliš nákladné. Softwarové zpětné inženýrství se běžně používá např. v těchto oblastech:

- **Škodlivý software** – na jedné straně stojí tvůrci škodlivého software, kteří používají zpětné inženýrství pro nalezení slabých míst v operačních systémech, internetových prohlížečích atd. Na druhé straně stojí tvůrci antivirového software, kteří používají

zpětné inženýrství ke zjištění, jak odstranit škodlivý software ze systému, nebo jak zabránit infikování systému tímto škodlivým softwarem.

- **Kryptografie** – zde se dá zpětné inženýrství využít k odhalení šifrovacího algoritmu pro algoritmy, kde je tajemstvím samotný šifrovací algoritmus (např. Caesarova šifra [16]). Dále lze zpětné inženýrství využít i u algoritmů, kde je algoritmus veřejný a tajemstvím je klíč. V takových případech lze zkoumat, zda konkrétní implementace algoritmu neobsahuje chyby, což může být využito k redukci možných kombinací, které je potřeba vyzkoušet pro získání klíče.

- **Správa digitálních práv** (anglicky *Digital rights management*, zkráceně DRM) – zpětné inženýrství lze využít pro obcházení DRM ochrany u přehrávačů nebo prohlížečů digitálního obsahu.

- **Kontrola kvality uzavřeného software** – zpětné inženýrství lze využít pro kontrolu kvality nebo hledání bezpečnostních chyb v uzavřeném software.

- **Vývoj konkurenčního software** – pro vývoj kompletního software nemá smysl využívat zpětné inženýrství, ale lze jej využít např. pro získání specifických algoritmů, které by jinak bylo velmi složité vyvinout.

2.3 Postup provádění

Existuje mnoho různých přístupů pro provádění zpětného inženýrství, ale obecně lze postup rozdělit do dvou etap. První etapou je pozorování systému při běhu programu. Druhou etapou je zkoumání nízkoúrovňového kódu.

Pozorování systému při běhu programu spočívá ve spuštění různých nástrojů a služeb operačního systému, které jsou využívány pro získávání informací, sledování vstupu a výstupu programu atd. Jelikož program komunikuje s okolním světem prostřednictvím operačního systému, lze při důkladné znalosti operačního systému zjistit mnoho užitečných informací.

Zkoumání nízkoúrovňového kódu je složitý proces, který vyžaduje znalost zpětného inženýrství, vývoje software, práce procesoru a operačního systému. Jedná se o zkoumání posloupností instrukcí v binárním souboru, což je náročné, ale lze k tomu využít nástrojů, které jsou popsány v následující sekci.

2.4 Používané nástroje

Bez použití nástrojů by nebylo možné provádět zpětné inženýrství. Tato sekce popisuje základní kategorie nástrojů, které lze použít, i když některé z nich nebyly vytvořeny za účelem zpětného inženýrství.

2.4.1 Nástroje pro sledování systému

Jedná se o celou řadu nástrojů, které jsou většinou vestavěny přímo v operačním systému. Tyto nástroje slouží ke shromažďování informací o běhu programu. Může jít o sledování síťové aktivity, přístupu k souborovému systému nebo využívání dalších prostředků operačního systému.

2.4.2 Disassemblery

Disassemblery jsou programy, které mají na vstupu binární spustitelný program a vygenerují textové soubory, které obsahují kód v jazyce symbolických instrukcí pro celý program nebo jeho části. Jazyk symbolických instrukcí (assembler) je textové mapování binárně zakódovaných instrukcí procesoru, ale disassembler musí řešit také rozlišení datových a kódových segmentů kódu, což je poměrně složitý problém. Disassembler je specifický pro každý procesor, ale existují i disassemblery s podporou více architektur procesorů.

2.4.3 Nástroje pro ladění programů za běhu (debuggery)

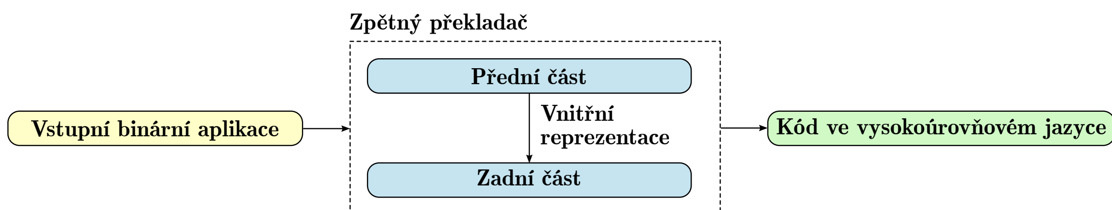
Debugger je program, který umožňuje vývojářům softwaru pozorovat jejich programy za běhu. Základní funkcí debuggeru je vkládání zarážek (anglicky *breakpoints*) a krokování v kódu. Zarážku lze nastavit na libovolný příkaz ve zdrojovém kódu a program spustit. Při dosažení instrukce se zarážkou se běh programu zastaví a je možné prozkoumat stav programu. Některé debuggery umožňují i běh v režimu disassembleru, což je u zpětného inženýrství obzvláště výhodné, protože je možné „krokovat“ program i na úrovni instrukcí procesoru.

2.4.4 Zpětné překladače

Zpětný překladač (nebo také dekompilátor) je program, který má na vstupu binární spustitelný program a na výstupu čitelný zdrojový kód ve vysokoúrovňovém jazyce. Myšlenkou zpětného překladače je obrácení postupu překladu programu za účelem získání zdrojového souboru, který je ekvivalentní vstupnímu binárnímu souboru. Obrácení postupu překladače však není možné provést přesně, protože během překladu se ze vstupního zdrojového kódu ztratí množství informací. Jedná se např. o informace o datových typech, názvech proměnných a funkcí, komentáře.

Základní struktura zpětného překladače je znázorněna na obrázku 2.1. Struktura je podobná překladači, ale jednotlivé části pracují v opačném pořadí. Cílem přední části je dekódování instrukcí a jejich převod do vnitřní reprezentace. Cílem zadní části zpětného překladače je z vnitřní reprezentace vygenerovat zdrojový kód ve vysokoúrovňovém jazyce.

Zpětný překladač lze považovat za další vývojový stupeň po disassembleru [11]. Úlohou zpětného překladače je vytvořit kód na vyšší úrovni abstrakce než v jazyce symbolických instrukcí. Dalším využitím zpětného překladače může být vytvoření zdrojového kódu v jiném programovacím jazyce, než ve kterém byl zdrojový program napsán [11].



Obrázek 2.1: Obecné schéma zpětného překladače

Kapitola 3

Zpětný překladač společnosti AVG

Ve společnosti AVG je v současné době vyvíjen zpětný překladač, který vznikl jako součást projektu Lissom [5] na Fakultě informačních technologií Vysokého učení technického v Brně. Tento zpětný překladač je rekonfigurovatelný, což znamená, že je nezávislý na architektuře procesoru, pro kterou byl vstupní binární soubor vytvořen. Zpětný překladač společnosti AVG má dále podporu pro více formátů binárních souborů a výstupní kód umožňuje generovat ve více vysokoúrovňových programovacích jazycích. Zpětný překlad je možné vyzkoušet na veřejně přístupných stránkách projektu [6].

Tato kapitola je zpracována na základě [11, 18] a je rozvržena následovně. Sekce 3.1 obsahuje popis struktury zpětného překladače a stručný popis jednotlivých částí. Následující sekce 3.2 podrobněji popisuje část předzpracování. Sekce 3.3 popisuje přední část, sekce 3.4 popisuje optimalizační část a poslední sekce 3.5 popisuje zadní část.

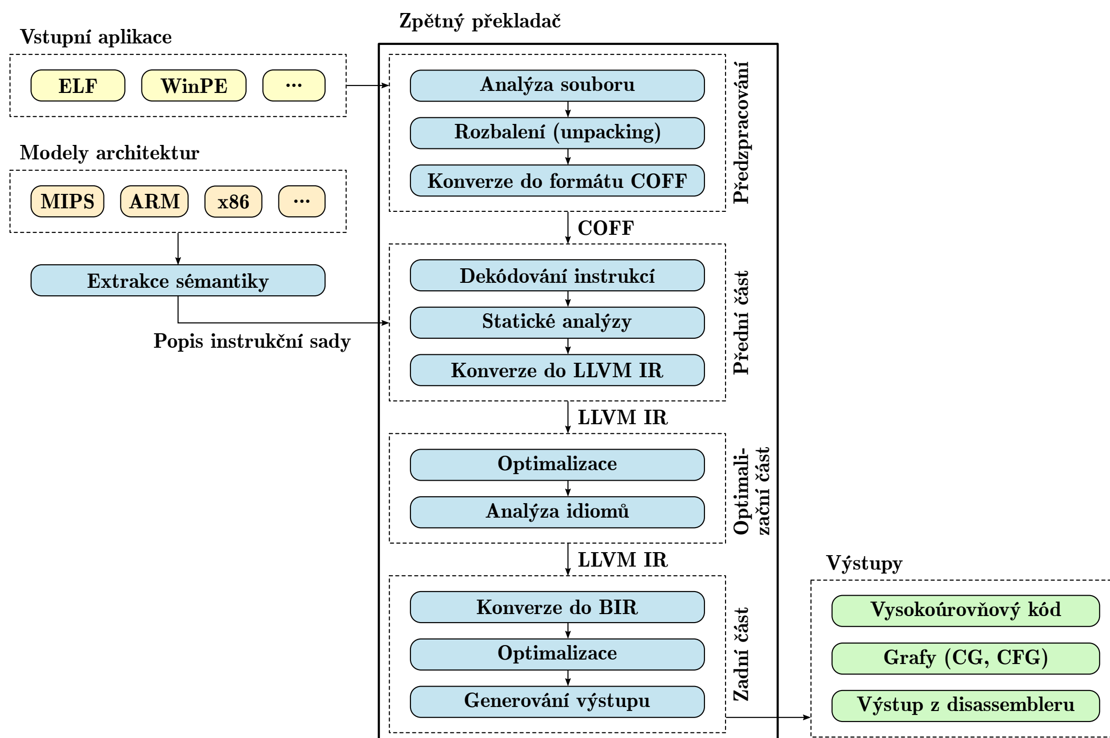
3.1 Struktura

Struktura zpětného překladače je velmi podobná struktuře klasického překladače a skládá se ze čtyř základních částí – předzpracování, přední část (anglicky *front-end*), optimalizační část (anglicky *middle-end*) a zadní část (anglicky *back-end*). Zjednodušená struktura zpětného překladače je znázorněna na obrázku 3.1.

Ve fázi předzpracování je vstupní aplikace analyzována pro získání co největšího množství informací – formát spustitelného souboru (např. ELF, WinPE), cílová architektura (např. Intel x86, ARM), informace o kódových a datových sekcích, ladicí informace, zdrojový programovací jazyk, použitý překladač atd. Dále zde probíhá detekce, zda byla aplikace zabalena tzv. packerem (typicky u škodlivého software) a probíhá zde rozbalení (anglicky *unpacking*). Poslední prováděnou akcí v rámci předzpracování je převod na jednotný formát spustitelného souboru COFF.

Následuje zpracování přední části zpětného překladače, kde probíhá dekódování instrukcí a jejich převod na instrukce jazyka LLVM IR (bude popsán níže). Dále jsou zde prováděny některé statické analýzy – detekce staticky linkovaného kódu, detekce funkcí a jejich argumentů, detekce volání funkcí, detekce datových typů atd. Přední část rovněž umožňuje generování výstupu z disassembleru.

Jazyk LLVM IR je nízkoúrovňový programovací jazyk podobný assembleru, který je součástí projektu LLVM [13]. Projekt LLVM je kompletní sada nástrojů pro tvorbu vlastních překladačů a je navržena tak, aby byla nezávislá na programovacím jazyku a dále, aby podporovala optimalizace po celou dobu běhu programu [12, 13]. Jazyk LLVM IR se využívá



Obrázek 3.1: Struktura zpětného překladače společnosti AVG, převzato z [11] a upraveno

jako vnitřní reprezentace v přední a optimalizační části zpětného překladače.

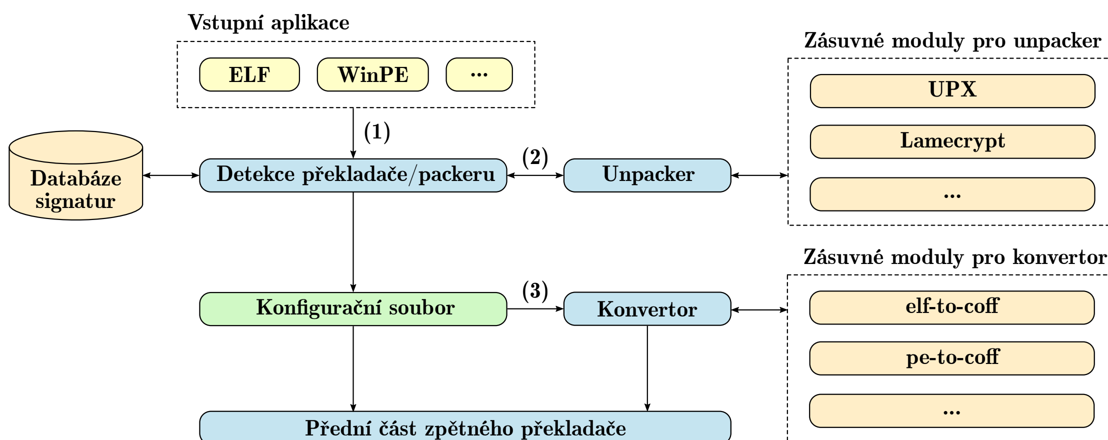
V optimalizační části probíhá několik optimalizací nad reprezentací v LLVM IR – optimalizace cyklů, propagování konstant, optimalizace nad grafem toku řízení atd. Dále zde probíhá detekce a transformace „instrukčních idiomů“, které do aplikací vkládají překladače v rámci optimalizací.

Poslední částí zpětného překladače je zadní část, která slouží k převodu optimalizovaného LLVM IR na reprezentaci ve vysokoúrovňovém jazyce (např. C, Python'). V rámci zadní části zpětného překladače je reprezentace v LLVM IR převedena do interní reprezentace zvané BIR (více v kapitole 5). Dále jsou provedeny optimalizace této interní reprezentace a je vygenerován kód v cílovém vysokoúrovňovém jazyce. Zadní část umožňuje také generování grafu volání funkcí (anglicky *call graph*, zkráceně CG) a grafů toku řízení (anglicky *control flow graph*, zkráceně CFG).

3.2 Předzpracování

Fáze předzpracování slouží k počáteční analýze vstupní aplikace, rozbalení aplikace (v případě, že byla zabalena packerem) a převodu do jednotného formátu COFF. Dále tato fáze generuje konfigurační soubor, který obsahuje zjištění informace o vstupní aplikaci a další nastavení zpětného překladu (např. cílový vysokoúrovňový jazyk nebo zapnutí generování grafů). Tento konfigurační soubor poté využívají všechny další fáze. Schéma fáze předzpracování je znázorněno na obrázku 3.2.

Prvním krokem (1) je detekce překladače nebo packeru, kterým byla spustitelná aplikace vytvořena. Informace o použitém překladači je užitečná např. v přední části zpětného



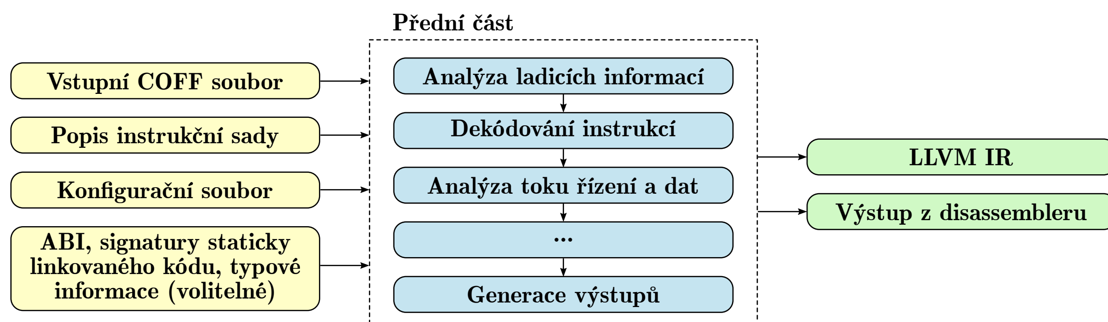
Obrázek 3.2: Schéma fáze předzpracování ve zpětném překladači, převzato z [11] a upraveno

překladače pro detekci funkce `main()` nebo pro detekci staticky linkovaného kódu. Dále se informace o použitém překladači používá v optimalizační části zpětného překladače k analýze idiomů. Detekce použitého překladače může probíhat na základě databáze signatur nebo na základě heuristických analýz.

Druhým krokem (2) je rozbalení v případě, že bylo detekováno použití packeru. Rozbalení funguje na základě obecné knihovny a zásuvných modulů pro rozbalení aplikací, které byly zabaleny konkrétním packerem. Tyto zásuvné moduly obsahují funkce pro detekci vstupního bodu aplikace, funkce pro dekompresi kódu atd. Zásuvný modul může fungovat buď tak, že provede všechny operace, které provádí packer, ale v opačném pořadí nebo tak, že spustí vstupní aplikaci, počká až se do paměti dekomprimuje a uloží obraz paměti. V některých případech je možné, že byla aplikace zabalena více packery, v takovém případě se provádí postup opakovaně.

Třetím krokem (3) je konverze rozbalené vstupní aplikace do jednotného formátu COFF. Konverze probíhá na základě zásuvných modulů, které převádí konkrétní formát objektových souborů do formátu COFF.

3.3 Přední část



Obrázek 3.3: Struktura přední části zpětného překladače, převzato z [11] a upraveno

V přední části zpětného překladače probíhá konverze vstupních instrukcí strojových závislých na platformě na reprezentaci v jazyce LLVM IR, která již není závislá na platformě. Je zde prováděno několik statických analýz – oddělení dat a kódu, dekódování instrukcí, analýza toku řízení (anglicky *control flow analysis*, zkráceně CFA), analýza toku dat (anglicky *data flow analysis*, zkráceně DFA), detekce funkcí, proměnných a datových typů, detekce staticky linkovaného kódu (např. volání funkcí ze standardních knihoven) atd. Výstupem přední části zpětného překladače je kód v jazyce LLVM IR a výstup z disassembleru (ukázka výstupu z disassembleru je znázorněna v příloze A na obrázku A.3). Schéma přední části zpětného překladače je znázorněno na obrázku 3.3.

Na vstupu přední části se nachází COFF soubor a konfigurační soubor, které byly vygenerovány ve fázi předzpracování; dále pak popis instrukční sady, který slouží k dekódování instrukcí. Přední část podporuje také rozšířené vstupy, které nejsou povinné a slouží ke zlepšení kvality výsledného kódu:

- Popis aplikačně binárního rozhraní (ABI), který slouží ke zlepšení analýzy toku dat.
- Signatury staticky linkovaného kódu, které slouží k eliminaci kódu např. ze standardních knihoven programovacích jazyků.
- Typové informace, které slouží ke zlepšení rekonstrukce datových typů při volání staticky linkovaných funkcí.

Jednotlivé analýzy jsou volány v pořadí, které závisí na jejich použití a závislostech mezi nimi. Některé analýzy jsou volány iterativně (např. detekce funkcí je volána tak dlouho, dokud nejsou nalezeny všechny funkce).

3.4 Optimalizační část

V optimalizační části zpětného překladače probíhá optimalizace kódu v LLVM IR a analýza idiomů. Tyto optimalizace slouží ke zmenšení objemu kódu, který vstupuje do zadní části zpětného překladače a ke zlepšení čitelnosti výsledného kódu.

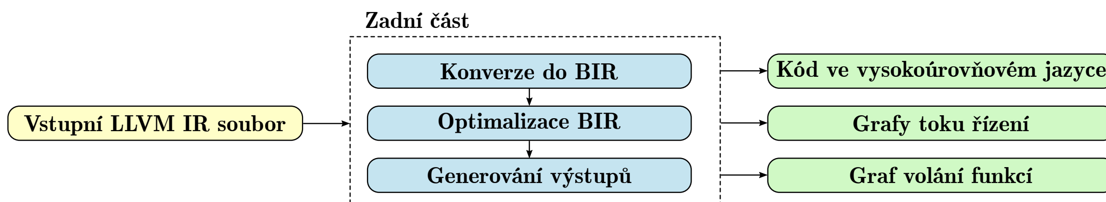
Optimalizace jsou prováděny vestavěným nástrojem platformy LLVM s nastavením optimalizací na nejvyšší stupeň. Některé optimalizace jsou však záměrně vypnuty, protože cílem optimalizace u překladačů je převod vysokoúrovňového kódu na strojový kód, k čemuž jsou optimalizace přizpůsobeny. Mezi použité optimalizace patří:

- Analýza ukazatelů, která je použita pro zjištění, ke kterým paměťovým místům může být přistupováno nepřímo pomocí ukazatelů.
- Odstranění mrtvého kódu – odstranění kódu, který není dosažitelný, nebo který neprovádí smysluplný výpočet.
- Dále propagování konstant, zjednodušení výrazů atd.

V průběhu zpětného překladu je rovněž nutné vrátit zpět některé optimalizace, které při překladu provádí překladače, aby výsledná aplikace byla co nejvíce optimalizována na rychlost nebo paměťovou náročnost. Výsledné výrazy jsou složeny z komplikovaných posloupností bitových operací apod. a jsou velmi obtížně čitelné. Jednoduchým příkladem takového idiomu je vynulování registru, které lze efektivněji provést pomocí operace XOR (např. `xor eax, eax`) místo přiřazení nuly do tohoto registru (např. `mov eax, 0`). V rámci analýzy idiomů jsou tyto idiomy převedeny zpět na čitelné výrazy.

3.5 Zadní část

Poslední fází zpětného překladu je zadní část. V této části probíhá konverze optimalizované reprezentace v LLVM IR do cílové reprezentace ve vysokoúrovňovém jazyce. Podporován je cílový jazyk C a upravený Python (úprava spočívá např. v přidání příkazu `goto`, jelikož v některých případech není možné příkaz `goto` eliminovat). Schéma zadní části zpětného překladače je znázorněno na obrázku 3.4.



Obrázek 3.4: Struktura zadní části zpětného překladače, převzato z [11] a upraveno

Prvním krokem zadní části zpětného překladače je konverze vstupního kódu v LLVM IR do vnitřní reprezentace BIR. Během této konverze jsou nízkoúrovňové konstrukce jazyka LLVM IR převáděny na vysokoúrovňové konstrukce. Rovněž jsou získávány připojené ladicí informace. Následuje provedení optimalizací nad vnitřní reprezentací kódu – na rozdíl od optimalizační části zpětného překladače jsou zde však prováděny optimalizace na vyšší úrovni. Posledním krokem zadní části je přejmenování proměnných a generování výstupů, tedy kódu ve zvoleném vysokoúrovňovém jazyce a generování grafů volání funkcí a grafu toku řízení. Ukázka vygenerovaného kódu a grafů je znázorněna v příloze A.

3.5.1 Konverze do BIR

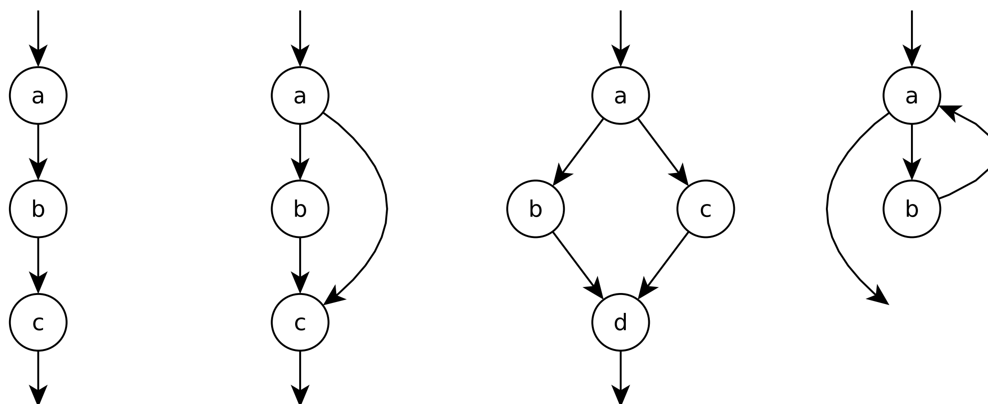
Cílem konverze do BIR je převod podmíněných skoků, které se vyskytují v LLVM IR, na vysokoúrovňové konstrukce (větvení, cykly). Tyto vysokoúrovňové konstrukce jsou čitelnější než kód, který obsahuje jen skoky pomocí příkazu `goto`.

Na obrázku 3.5 jsou znázorněny příklady vzorů vysokoúrovňových konstrukcí. Obrázek 3.5a znázorňuje posloupnost bloků, které jsou vykonány postupně. Obrázky 3.5b a 3.5c znázorňují podmíněný příkaz `if` bez větve `else` a s větví `else`. Cyklus typu `while` je znázorněn na obrázku 3.5d.

3.5.2 Optimalizace BIR

Po konverzi do BIR je potřeba provést další fázi optimalizací, které souvisí s převodem kódu na konstrukce vyšší úrovně. Některé optimalizace jsou však stejné jako v optimalizační části zpětného překladače. Mezi nejdůležitější použité optimalizace patří:

- Zjednodušení aritmetických výrazů.
- *Copy propagation*, tedy zrušení pomocných přiřazení, které pouze kopírují existující proměnné a jejich nahrazení původním výrazem. Příklad takové optimalizace je znázorněn na obrázku 3.6.
- Převod cyklů `while` na cykly `for`, pokud v daném případě je takový cyklus čitelnější (např. obsahuje indukční proměnnou).



(a) Posloupnost příkazů (b) Příkaz `if` (c) Příkaz `if-else` (d) Příkaz `while`

Obrázek 3.5: Příklady vzorů vysokoúrovňových konstrukcí

- Odstranění mrtvého kódu a zbytečných přetypování.

```
x = rand();
y = x;
return y % 100;
```

(a) Výstup před optimalizací

```
return rand() % 100;
```

(b) Výstup po optimalizaci

Obrázek 3.6: Příklad optimalizace – *copy propagation*

V rámci zvýšení čitelnosti kódu je dále prováděno přejmenování proměnných, jelikož původní jméno proměnné je většinou při překladu odstraněno. Jsou použity následující přejmenování proměnných, aby názvy proměnných byly co nejčitelnější v daném kontextu použití:

- V případě dostupnosti ladicích informací jsou použity původní názvy proměnných.
- Argumenty funkcí mají názvy `a1`, `a2` atd., lokální proměnné mají názvy `v1`, `v2` atd. a globální proměnné mají názvy `g1`, `g2` atd.
- Indukční proměnné v cyklech mají obvykle užívané názvy `i`, `j`, `k` atd.
- Proměnná obsahující výsledek funkce je pojmenována `result`.
- Návratové hodnoty volání standardních funkcí jsou v některých případech pojmenovány podle sémantiky navrácené hodnoty (např. návratová hodnota funkce `strlen()` je pojmenována `len`).

Poslední optimalizací, která je zmíněna v této sekci je převod literálů na symbolické konstanty, což slouží k dalšímu zlepšení čitelnosti. Např. optimalizace volání knihovní funkce `socket()` je znázorněna na obrázku 3.7.

```
sock_id = socket(2, 3, 255);
```

```
sock_id = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

(a) Výstup před optimalizací (b) Výstup po optimalizaci

Obrázek 3.7: Příklad optimalizace – převod literálů na symbolické konstanty, převzato z [11]

3.5.3 Generování výstupů

Posledním krokem procesu zpětného překladu je generování výstupů, kde zadní část zpětného překladače generuje cílový kód ve vysokoúrovňovém jazyce, graf volání funkcí a grafy toku řízení pro každou funkci. Jak již bylo zmíněno v sekci 3.3, přední část zpětného překladače generuje výstup z disassembleru.

V rámci generování výstupního kódu jsou zohledněny všechny informace, které byly získány. Dále se generátor výstupního kódu stará o `#include` hlavičkových souborů (v případě jazyka C).

Kapitola 4

Jazyk LLVM IR

Tato kapitola popisuje jazyk LLVM IR. Nevěnuje se však všem jeho konstrukcím, ale pouze těm podstatným pro tuto práci. Vynechán je např. popis metadat, adresových prostorů, systému zachycování výjimek, atomických a vektorových instrukcí nebo zabudovaných (anglicky *intrinsic*) funkcí. Tato kapitola je zpracována na základě [14].

LLVM IR je jazyk typu *Static Single Assignment (SSA)*, což znamená, že do jedné proměnné může být přiřazena hodnota pouze jednou. Hlavním důvodem tohoto přístupu je jednodušší provádění optimalizací [8]. Mezi výhody jazyka LLVM IR patří typová bezpečnost, flexibilita, možnost reprezentovat nízkoúrovňové i vysokoúrovňové operace atd.

4.1 Identifikátory

Identifikátory v jazyce LLVM IR se rozdělují na globální a lokální. Globální identifikátory, které slouží pro jména funkcí a globálních proměnných, vždy začínají znakem `@`. Lokální identifikátory, které slouží pro uložení proměnných a typů, vždy začínají znakem `%`. Pro samotné identifikátory se používají následující tři formáty:

1. Pojmenované hodnoty jsou reprezentovány jako textový řetězec znaků s prefixem. Jsou povoleny malé a velké znaky anglické abecedy, číslice, pomlčka, podtržítko, tečka a znak dolaru. Jako první znak nesmí být použita číslice. Příkladem identifikátoru pojmenované hodnoty může být `%foo` nebo `@DivisionByZero`.

2. Nepojmenované hodnoty jsou reprezentovány jako neznaménková číselná hodnota s prefixem. Příkladem identifikátoru nepojmenované hodnoty může být `%12` nebo `@2`.

3. Konstanty, které jsou blíže popsány v sekci 4.4.

4.2 Vysokoúrovňová struktura

V této sekci jsou popsány jednotlivé části vysokoúrovňové struktury zdrojového kódu v jazyce LLVM IR. Programy napsané v jazyce LLVM IR jsou složeny z modulů, kde každý z těchto modulů je jednotkou překladu programu. Každý modul se skládá z funkcí a globálních proměnných.

Na obrázku 4.1 je znázorněn příklad funkce na výpočet faktoriálu v jazyce LLVM IR. Tato funkce obsahuje tři základní bloky. V bloku `entry` je podmíněný skok na základě výsledku porovnání. V bloku `body` je provedena operace odčítání, rekurzivní zavolání funkce,

operace násobení a poslední instrukcí je nepodmíněný skok na poslední základní blok `after`. Tento poslední blok slouží k vrácení hodnoty výsledku a obsahuje instrukci `phi`, která je podrobně vysvětlena v sekci 4.5.8.

```
define i32 @factorial(i32 %x) {
  entry:
      %cond = icmp ult i32 %x, 2
      br i1 %cond, label %after, label %body

  body:
      %0 = sub i32 %x, 1
      %1 = call i32 @factorial(i32 %0)
      %2 = mul i32 %x, %1
      br label %after

  after:
      %ret = phi i32 [ 1, %entry ], [ %2, %body ]
      ret i32 %ret
}
```

Obrázek 4.1: Příklad rekurzivního výpočtu faktoriálu v jazyce LLVM IR

4.2.1 Globální proměnné

Globální proměnné jsou paměťová místa, která jsou alokována v době překladu programu. Definice globální proměnné musí vždy obsahovat inicializační hodnotu, ale mohou být definovány i globální proměnné, které se nachází v jiných modulech, a ty inicializační hodnotu nemají. Globální proměnná může být označena jako konstanta, což znamená, že její hodnota nemůže být za běhu programu změněna. Označení jako konstanta umožňuje např. lepší optimalizaci.

Definice globální proměnné se provádí pomocí klíčového slova `global`, definice globální konstanty se provádí pomocí klíčového slova `constant`. Při definici lze zadat typ linkování, viditelnost atd.

4.2.2 Funkce

K definici funkce slouží klíčové slovo `define`. K deklaraci funkce slouží klíčové slovo `declare`. U funkce se zadává návratový typ a seznam parametrů. Podobně jako u globálních proměnných lze u definicí i deklarací funkcí zadat typ linkování a viditelnost. Dále však lze zadat i další atributy. Příklad definice funkce je na obrázku 4.1.

Definice těla funkce se skládá ze seznamu základních bloků, které vytváří graf toku řízení pro definovanou funkci. Každý základní blok kromě prvního musí začínat návěštím a skládá se ze seznamu instrukcí, kde poslední instrukcí je instrukce způsobující změnu řízení. První základní blok ve funkci má tyto specifické vlastnosti:

1. Je vykonán ihned při vstupu do funkce.

2. Nemůže mít žádné předchůdce, z čehož vyplývá, že nemůže být cílem žádného skoku, a že nemůže obsahovat žádné `phi` instrukce.

4.3 Typy

Jazyk LLVM IR používá silný typový systém. Díky tomu může být provedeno velké množství optimalizací bez použití speciálních analýz. V následujícím textu je popsán typ funkce a dále dle názvosloví LLVM IR tzv. *first-class types* (bez překladu), což jsou jediné typy, které mohou být výsledky instrukcí. Tyto typy jsou dále rozděleny na jednoduché a složené. V jazyce LLVM IR se dále vyskytuje bezrozměrný typ `void`, který nereprezentuje žádnou hodnotu.

4.3.1 Jednoduché typy

Mezi jednoduché typy patří typ celého čísla, typ čísla s plovoucí desetinnou čárkou, typ ukazatele, typ návěští (anglicky *label*) a další, které nejsou předmětem této práce (např. vektorový typ).

U typu pro celé číslo se zadává bitová šířka, která může být od 1 až do $2^{32}-1$. Syntakticky se zapisuje jako `iN`, kde `N` je počet bitů. Příkladem může být celočíselný typ o šířce 32 bitů, který se v jazyce LLVM IR zapíše jako `i32`.

V jazyce LLVM IR existuje celkem šest typů pro čísla s plovoucí desetinnou čárkou, konkrétně typy `half` (16-bitová šířka), `float` (32-bitová šířka), `double` (64-bitová šířka), `fp128` (128-bitová šířka se 112-bitovou mantisou), `x86_fp80` (80-bitová šířka) a `ppc_fp128` (dvě 64-bitové hodnoty jako jedno číslo).

Typ ukazatel může uvnitř obsahovat libovolný typ kromě typu `void` a typu návěští. Příkladem může být ukazatel na pole čtyř hodnot typu `i32`, který se v LLVM IR zapíše jako `[4 x i32]*`.

4.3.2 Složené typy

Složené typy jsou typy, které obsahují více prvků. Jedná se o pole a struktury. Pole je jednoduchý odvozený typ, který obsahuje jednotlivé prvky za sebou v paměti. Pro zápis typu pole je potřeba zadat počet prvků a tento počet musí být konstantní. Příkladem zápisu pole v jazyce LLVM IR je `[40 x i32]`, což je pole, které obsahuje 40 prvků typu `i32`. Pole může být rovněž vícedimenzionální. Příkladem je `[12 x [10 x float]]`.

Struktura je typ použitý k reprezentaci kolekce dat dohromady v paměti. Prvkem struktury může být libovolný typ, který má rozměr. Struktury mohou být pojmenované (dle názvosloví LLVM IR *identified*) nebo anonymní (dle názvosloví LLVM IR *literal*). Anonymní struktury jsou definovány jako ostatní typy přímo při použití, zatímco pojmenované struktury jsou definovány na nejvyšší úrovni a mají jméno. Pojmenované struktury mohou být rovněž rekurzivní. Příkladem struktury, která obsahuje trojici hodnot typu `i32` je `{ i32, i32, i32 }`.

4.3.3 Typ funkce

Typ funkce lze považovat za signaturu funkce a skládá se z návratového typu a seznamu typů jednotlivých parametrů. Návratový typ může být `void` nebo nějaký typ první třídy kromě typu návěští. Typ funkce může mít i variabilní počet parametrů. Příkladem typu funkce je `i32 (i32, i32)`, což je funkce, která přijímá dva parametry jako celá čísla o šířce 32 bitů (`i32`) a vrací rovněž typ `i32`.

4.4 Konstanty

V jazyce LLVM IR je několik základních typů konstant, které jsou popsány v následujícím textu s rozdělením na jednoduché a složené konstanty.

4.4.1 Jednoduché konstanty

Mezi jednoduché konstanty patří:

- Konstanty logických hodnot (typu `i1`), což jsou dvě konstanty `true` a `false`.
- Konstanty celých čísel (typu `iN`) mohou být libovolná čísla (i záporná), tedy např. číslo `4` nebo `-100`.
- Konstanty čísel s plovoucí desetinnou čárkou mohou být zapsány ve standardní desítkové notaci (např. `22.48`), exponenciální notaci (např. `22.48e+10`) nebo v přesnější hexadecimální notaci. Typy `fp128`, `x86_fp80` a `ppc_fp128` však vyžadují zápis v hexadecimální notaci, jiná notace není podporována. Hexadecimální notace je podrobně popsána v [14].
- Typ ukazatel má jedinou podporovanou konstantu, a to nulový ukazatel `null`.

4.4.2 Složené konstanty

Složené konstanty jsou kombinací jednoduchých konstant a dílčích složených konstant. Mohou být i rekurzivní. Mezi složené konstanty patří:

- Konstanty struktur, které mají podobnou reprezentaci jako samotné definice struktur, tedy seznam prvků oddělených čárkou a obalený ve složených závorkách. Příkladem konstantní struktury je `{ i32 4, float 17.0 }`.
- Konstanty polí jsou reprezentovány jako seznamy prvků oddělených čárkou a obalených v hranatých závorkách. Příkladem konstantního pole je `[ i32 42, i32 74 ]`. Druhou možnou reprezentací, která platí pro typ `i8*` (textový řetězec), je text v uvozovkách, kterému bezprostředně předchází znak `c`. Příkladem takového textového řetězce je `c"Hello World\0A\00"`.
- Nulová inicializace libovolného typu může být použita k inicializaci libovolného typu na nulu. Vhodné použití je pro inicializaci velkých polí nebo struktur, aby nemusely být vypisovány dlouhé seznamy nul. Konstanta nulové inicializace je reprezentována jako `zeroinitializer`.

4.5 Instrukce

Instrukční sada jazyka LLVM IR se skládá z několika kategorií instrukcí: instrukce způsobující změnu řízení, binární operátory, operátory pro práci se složenými typy, operátory přístupu k paměti, instrukce pro porovnávání a další. Tyto instrukce jsou popsány v následujících sekcích.

4.5.1 Instrukce způsobující změnu řízení

Každý základní blok musí končit jednou z instrukcí, která způsobuje změnu řízení. V názvosloví LLVM IR se takovému typu instrukce říká *terminator instruction*. Tato instrukce označuje, který základní blok má být vykonán jako následující. Tyto instrukce nevrací žádné hodnoty, pouze se starají o změnu toku řízení. Mezi tyto instrukce patří:

- Instrukce `ret`, která slouží k návratu z funkce.
- Instrukce `br`, která slouží k provedení nepodmíněného skoku na zadané návěští (analogie příkazu `goto` v jazyce C).
- Instrukce `switch`, která slouží k rozvětvení toku řízení na více míst v závislosti na řídící proměnné.

4.5.2 Binární operátory

Binární operátory provádí většinu výpočtů v programech [14]. Vyžadují dva operandy stejného typu, provedou s nimi výpočetní operaci a vrátí jednu hodnotu, která je stejného typu jako operandy. V jazyce LLVM IR jsou k dispozici tyto binární operátory:

- Instrukce `add` a `fadd` slouží k sečtení dvou operandů.
- Instrukce `sub` a `fsub` slouží k odečtení dvou operandů.
- Instrukce `mul` a `fmul` slouží k vynásobení dvou operandů.
- Instrukce `udiv`, `sdiv` a `fdiv` slouží k vydělení dvou operandů.
- Instrukce `urem`, `srem` a `frem` reprezentují zbytek po celočíselném dělení.

4.5.3 Bitové binární operátory

Pro různé bitové operace se používají bitové operátory, což jsou většinou velmi efektivní instrukce. Bitové binární operátory vyžadují dva operandy stejného typu, provedou s nimi výpočetní operaci a vrátí jednu hodnotu stejného typu jako operandy. V jazyce LLVM IR jsou k dispozici tyto bitové binární operátory:

- Instrukce `shl` reprezentuje bitový posun vlevo (logický i aritmetický).
- Instrukce `lshr` a `ashr` reprezentují bitový posun vpravo, konkrétně `lshr` reprezentuje logický bitový posun vpravo a `ashr` aritmetický.
- Instrukce `and` reprezentuje logický součin.
- Instrukce `or` reprezentuje logický součet.
- Instrukce `xor` reprezentuje exkluzivní logický součet.

4.5.4 Operátory pro práci se složenými typy

Pro práci se složenými typy jsou v jazyce LLVM IR následující dvě instrukce:

- Instrukce `extractvalue` slouží k přečtení hodnoty ze struktury.
- Instrukce `insertvalue` slouží k uložení hodnoty do struktury.

4.5.5 Operátory přístupu k paměti

Pro práci s pamětí v jazyce LLVM IR existují následující instrukce:

- Instrukce `alloca` slouží k alokaci místa v paměti.
- Instrukce `load` slouží k načtení hodnoty z paměti.
- Instrukce `store` slouží k uložení hodnoty do paměti.
- Instrukce `getelementptr` slouží k výpočtu adresy v paměti, která odpovídá konkrétnímu prvku v poli nebo ve struktuře, přičemž tento prvek může být libovolně zanořen.

4.5.6 Instrukce pro konverzi datových typů

- Instrukce `trunc ... to` slouží k ořezání celého čísla z většího na menší počet bitů.
- Instrukce `zext ... to` slouží k rozšíření celého čísla z menšího na větší počet bitů, přičemž přidané bity jsou vyplněny nulami.
- Instrukce `sext ... to` slouží k rozšíření celého čísla z menšího na větší počet bitů, přičemž přidané bity reflektují znaménko.
- Instrukce `fptrunc ... to` slouží k ořezání čísla s plovoucí desetinnou čárkou z většího na menší počet bitů.
- Instrukce `fpext ... to` slouží k rozšíření čísla s plovoucí desetinnou čárkou z menšího na větší počet bitů.
- Instrukce `fptoui ... to` slouží k převodu čísla s plovoucí desetinnou čárkou na neznaménkové celé číslo.
- Instrukce `fptosi ... to` slouží k převodu čísla s plovoucí desetinnou čárkou na znaménkové celé číslo.
- Instrukce `uitofp ... to` slouží k převodu neznaménkového celého čísla na čísla s plovoucí desetinnou čárkou.
- Instrukce `sitofp ... to` slouží k převodu znaménkového celého čísla na čísla s plovoucí desetinnou čárkou.
- Instrukce `ptrtoint ... to` slouží k převodu ukazatele na celé číslo.
- Instrukce `inttoptr ... to` slouží k převodu celého čísla na ukazatel.
- Instrukce `bitcast ... to` slouží k ostatním převodům mezi typy.

4.5.7 Instrukce porovnávání

Pro porovnání existují v LLVM IR dvě instrukce, které přijímají dva číselné argumenty stejného typu a vrací logickou hodnotu:

- Instrukce `icmp` slouží k porovnání celých čísel.
- Instrukce `fcmp` slouží k porovnání čísel s plovoucí desetinnou čárkou.

4.5.8 Instrukce phi

Instrukce `phi` slouží k přiřazení hodnoty do proměnné na základě toho, který základní blok byl předchozí v toku řízení. Tyto instrukce se smí nacházet pouze na začátku každého základního bloku a musí obsahovat hodnotu pro všechny předchůdce daného základního bloku. Instrukce `phi` jsou v kódu potřebné, protože do každé proměnné lze přiřadit pouze jednou, takže pokud se v kódu nachází větvení nebo cyklus, je nutné to řešit pomocí instrukce `phi`.

Na Obrázku 4.2a je znázorněn jednoduchý příklad, kdy funkce `func` vrací buď parametr funkce `x` vynásobený dvěma nebo vydělený dvěma, podle toho, jak je vyhodnocena podmínka. Tedy hodnota proměnné `%ret` bude rovna proměnné `%y1`, pokud byl vykonán základní blok `iftrue` nebo proměnné `%y2`, pokud byl vykonán základní blok `iffalse`. Ekvivalent k tomuto příkladu je na obrázku 4.2b, kde lze vidět, že při možnosti vícenásobného přiřazení do proměnné není instrukce `phi` potřeba.

```
define i32 @func(i32 %x) {
     %cond = icmp slt i32 %x, 10
     br i1 %cond, label %iftrue, label %iffalse

  iftrue:
     %y1 = mul i32 %x, 2
     br label %after

  iffalse:
     %y2 = sdiv i32 %x, 2
     br label %after

  after:
     %ret = phi i32 [ %y1, %iftrue ], [ %y2, %iffalse ]
     ret i32 %ret
}
```

(a) Jednoduchý příklad využití instrukce `phi` v jazyce LLVM IR

```
int func(int x) {
    int ret;

    if (x < 10) {
        ret = x * 2;
    } else {
        ret = x / 2;
    }

    return ret;
}
```

(b) Ekvivalentní příklad v jazyce C

Obrázek 4.2: Ukázka použití instrukce `phi`

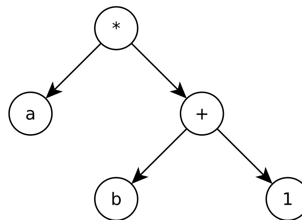
Kapitola 5

Vnitřní reprezentace použitá v zadní části zpětného překladače

V této kapitole je popsána vnitřní reprezentace, která je použita v zadní části zpětného překladače (anglicky *back-end intermediate representation*, zkráceně BIR). Tato reprezentace je pouze interní a je vždy uložena v paměti, takže nemá vlastní textovou reprezentaci. Reprezentace je schopna uložit všechny požadované konstrukce z LLVM IR a navíc i konstrukce na vyšší úrovni. Tato kapitola je zpracována na základě [17].

5.1 Modul, funkce a bázové třídy

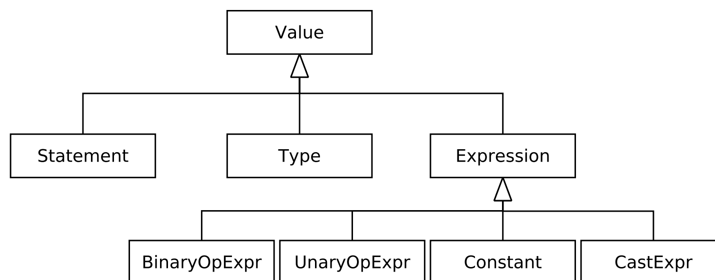
LLVM modul je v BIR reprezentován třídou `Module`. Tato třída umožňuje uložení globálních proměnných, deklarací funkcí a definicí funkcí. Funkce jsou reprezentovány třídou `Function`. Těla funkcí jsou modelována jako abstraktní syntaktické stromy. Příklad reprezentace výrazu `a * (b + 1)` je znázorněn na obrázku 5.1.



Obrázek 5.1: Reprezentace výrazu `a * (b + 1)` v BIR, převzato z [17]

Každá položka, která se může objevit v tomto abstraktním syntaktickém stromu (příkaz nebo výraz) je reprezentována jako podtřída abstraktní třídy `Value`. Na obrázku 5.2 jsou znázorněny všechny abstraktní bázové třídy. Příkazy jsou podtřídy abstraktní třídy `Statement`, typy jsou podtřídy abstraktní třídy `Type` a výrazy jsou podtřídy abstraktní třídy `Expression`.

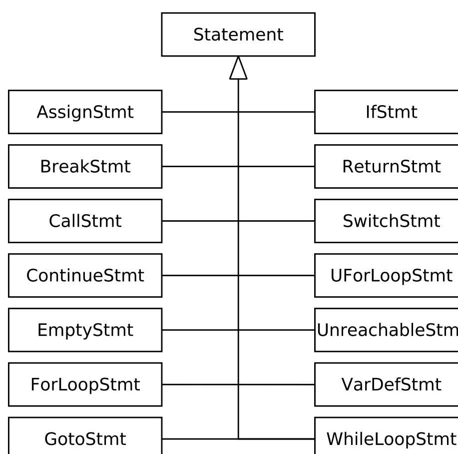
Pro výrazy zde existuje několik dalších bázových tříd; konkrétně `BinaryOpExpr` pro binární operace (např. sčítání, násobení), `UnaryOpExpr` pro unární operace (např. negace, operátor adresy), `Constant` pro konstantní hodnoty (např. konstantní celé číslo, konstantní textový řetězec) a `CastExpr` pro operace přetypování (např. ořezání na nižší počet bitů, přetypování z celého čísla na ukazatel).



Obrázek 5.2: Abstraktní bázové třídy v BIR, převzato z [17]

5.2 Příkazy

V BIR je čtrnáct typů příkazů. Tyto příkazy jsou reprezentovány podtřídami bázové třídy `Statement` a jsou znázorněny na obrázku 5.3.



Obrázek 5.3: Třídy reprezentující příkazy v BIR, převzato z [17] a upraveno

Význam jednotlivých tříd (tedy typů příkazů) je popsán v následujících bodech:

- `AssignStmt` reprezentuje přiřazení jednoho výrazu do druhého ve formě `lhs = rhs`, kde `lhs` je levá strana přiřazení a `rhs` je pravá strana přiřazení.
- `VarDefStmt` reprezentuje definici lokální proměnné ve formě `var = init`, kde `var` je právě definovaná proměnná a `init` je volitelná inicializační hodnota proměnné.
- `ForLoopStmt` a `UForLoopStmt` reprezentují cyklus `for`. `ForLoopStmt` je varianta cyklu `for` ve tvaru `for (indVar = startValue; endCond; indVar += step) { body }`, kde `indVar` je indukční proměnná, `startValue` je počáteční hodnota indukční proměnné, `endCond` je ukončující podmínka, `step` je krok indukční proměnné a `body` je tělo funkce. U této varianty musí být zadány všechny části. `UForLoopStmt` je obecná varianta cyklu `for` ve tvaru `for (init; cond; step) { body }`, kde `init` je inicializační část, `cond` je ukončující podmínka, `step` je část pro aktualizaci indukční proměnné a `body` je tělo funkce. U této varianty však není povinná žádná část.
- `WhileLoopStmt` reprezentuje cyklus `while` a oproti cyklu `for` se skládá pouze ze dvou částí – z části pro podmínku a těla cyklu.

- `BreakStmt` a `ContinueStmt` jsou příkazy pro řízení cyklu. Třída `BreakStmt` reprezentuje příkaz `break`, který slouží k ukončení vykonávání těla nejbližšího cyklu nebo příkazu `switch`, který tento příkaz obsahuje ve svém těle. Třída `ContinueStmt` reprezentuje příkaz `continue`, který slouží k přeskočení zbytku těla nejbližšího cyklu.
- `IfStmt` reprezentuje příkaz větvení a podporuje větve `else-if` i větev `else`.
- `SwitchStmt` reprezentuje druhou variantu příkazu větvení, konkrétně příkaz `switch`. Skládá se z řídícího výrazu a seznamu větví `case`, podporována je rovněž část `default`.
- `ReturnStmt` reprezentuje příkaz `return`, tedy návrat z funkce.
- `GotoStmt` reprezentuje příkaz nepodmíněného skoku `goto`.
- `CallStmt` reprezentuje volání funkce jako příkaz. Jedná se pouze o obálku nad výrazem typu `CallExpr`, který je popsán v sekci 5.3.
- `UnreachableStmt` reprezentuje příkaz, který nemá být z pohledu vykonávání programu dosažitelný. Tento příkaz se používá např. po volání funkce, která má ukončit program.
- `EmptyStmt` reprezentuje prázdný příkaz, který neprovádí žádnou akci. Tento příkaz je použitelný v místech, kde není potřeba provádět žádnou akci, ale syntakticky zde nějaký příkaz musí být vložen.

5.3 Výrazy

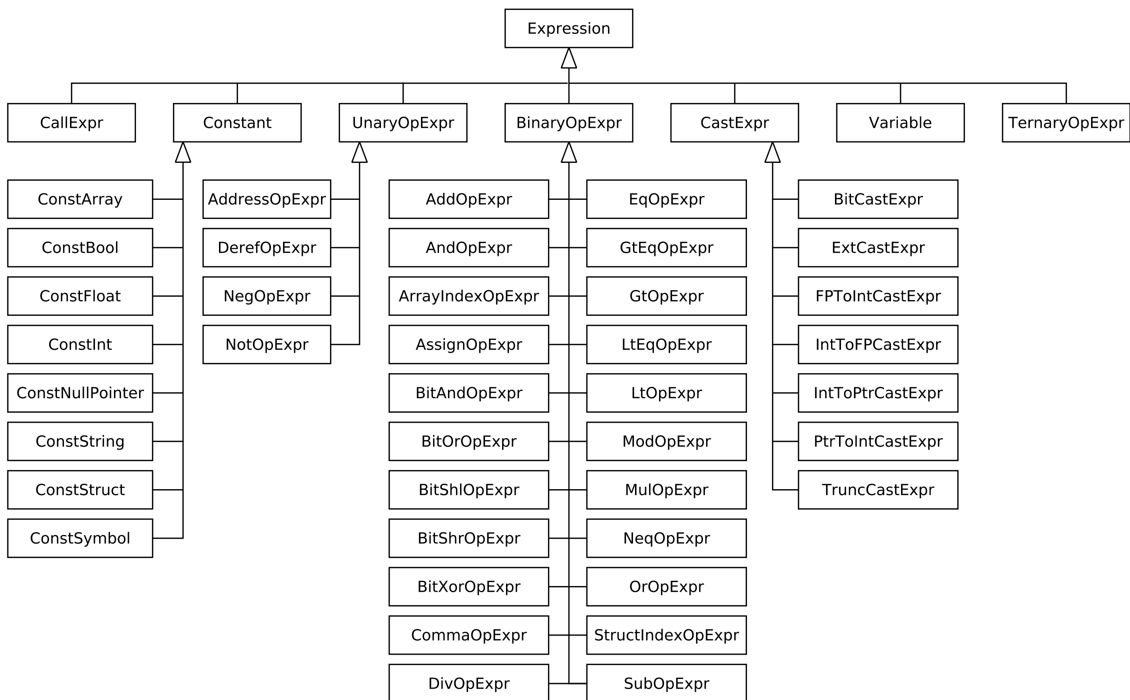
V BIR je mnoho typů výrazů, kde bázové třídy pro jednotlivé typy výrazů jsou popsány v sekci 5.1. Dostupné třídy pro reprezentaci výrazů jsou znázorněny na obrázku 5.4 a jedná se o následující typy výrazů:

- **Konstanty** – konstanta může být vytvořena pro jakýkoliv typ kromě typu `VoidType` a `UnknownType`. Konkrétně třída `ConstBool` reprezentuje konstantní logickou hodnotu, třída `ConstInt` konstantní celočíselnou hodnotu, třída `ConstFloat` konstantní číselnou hodnotu s plovoucí desetinnou čárkou, třída `ConstArray` konstantní pole, třída `ConstStruct` konstantní strukturu, třída `ConstString` konstantní textový řetězec, třída `ConstNullPointer` konstantní nulový ukazatel a třída `ConstSymbol` pojmenovanou symbolickou konstantu.
- **Unární operátory** – existují zde čtyři unární operátory. Třída `AddrOpExpr` reprezentuje operátor získání ukazatele z operandu, třída `DerefOpExpr` reprezentuje operátor dereference, třída `NegOpExpr` reprezentuje operátor negace číselné hodnoty a třída `NotOpExpr` reprezentuje operátor negace logické hodnoty.
- **Binární operátory** – běžné aritmetické operátory jsou reprezentovány následujícími třídami: `AddOpExpr` pro sčítání, `SubOpExpr` pro odčítání, `MulOpExpr` pro násobení, `DivOpExpr` pro dělení a `ModOpExpr` pro operaci modulo (zbytek po celočíselném dělení). Dále operátory pro porovnávání: `EqOpExpr` pro rovnost, `NeqOpExpr` pro nerovnost, `GtOpExpr` pro operaci větší než, `GtEqOpExpr` pro operaci větší než nebo rovno, `LtOpExpr` pro operací menší než a `LtEqOpExpr` pro operaci menší než nebo rovno. Bitové operátory jsou tyto: `BitAndOpExpr` pro operaci logického součinu,

BitOrOpExpr pro operaci logického součtu, BitXorOpExpr pro operaci exkluzivního logického součtu, BitShlOpExpr pro bitový posun vlevo, BitShrOpExpr pro bitový posun vpravo (jsou podporovány aritmetické i logické posuny). Logické operace and a or jsou reprezentovány třídami AndOpExpr a OrOpExpr. Další dva binární operátory jsou AssignOpExpr, což je operátor přiřazení jednoho výrazu druhému a CommaOpExpr, což je operátor čárky jako v jazyce C. Poslední dva binární operátory jsou operátory pro přístup k prvku pole (ArrayIndexOpExpr) a pro přístup k prvku struktury (StructIndexOpExpr).

- **Přetypování** – pro přetypování mezi různými typy jsou zde k dispozici následující třídy: BitCastExpr pro změnu typu beze změny bitové reprezentace, ExtCastExpr pro změnu menšího typu na větší (u znaménkových typů je kopírováno i znaménko), FPToIntCastExpr pro převod celočíselného typu na typ s plovoucí desetinnou čárkou, IntToFPCastExpr pro převod typu s plovoucí desetinnou čárkou na celočíselný typ, IntToPtrCastExpr pro převod celočíselného typu na ukazatel, PtrToIntCastExpr pro převod ukazatele na celočíselný typ a TruncCastExpr pro změnu většího typu na menší, kdy dochází k ořezu hodnoty.

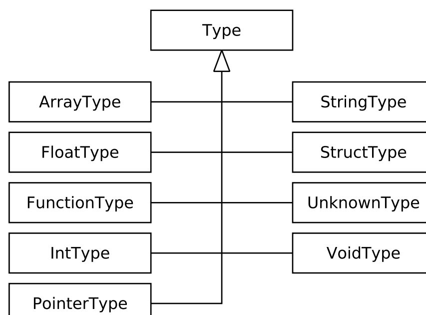
- **Ostatní výrazy** – další typy výrazů jsou třídami, které dědí přímo z bázové třídy Expression, jedná se o tyto třídy: Variable pro reprezentaci proměnné, CallExpr pro reprezentaci volání funkce a TernaryOpExpr pro reprezentaci ternárního operátoru. Ternární operátor se skládá ze tří částí – podmínky, hodnoty, která je vrácena při splnění podmínky a hodnoty, která je vrácena při nesplnění podmínky.



Obrázek 5.4: Třídy reprezentující výrazy v BIR, převzato z [17] a upraveno

5.4 Typy

V BIR je devět podporovaných typů, které jsou reprezentovány podtřídami bázové třídy `Type`. Tyto třídy jsou znázorněny na obrázku 5.5.



Obrázek 5.5: Třídy reprezentující typy v BIR, převzato z [17] a upraveno

Celočíselný typ je reprezentován třídou `IntType`, která podporuje libovolnou bitovou šířku i rozlišení znaménkových a neznaménkových typů. Logické hodnoty jsou také reprezentovány tímto typem (pokud je bitová šířka rovna jedné). Typ čísel s plovoucí desetinnou čárkou je reprezentován třídou `FloatType`, která také podporuje různé bitové šířky.

Dále třída `ArrayType` reprezentuje pole, třída `StructType` reprezentuje strukturu, třída `PointerType` reprezentuje ukazatel, třída `StringType` reprezentuje textový řetězec, třída `FunctionType` reprezentuje funkci, třída `VoidType` reprezentuje prázdný typ `void` a poslední třída `UnknownType` reprezentuje neznámý typ.

Kapitola 6

Rozbor současného stavu a stanovení cílů práce

Tato kapitola popisuje konvertor z LLVM IR do BIR, který je aktuálně používán ve zpětném překladači společnosti AVG. V sekci 6.1 je stručně popsán princip stávajícího konvertoru se zaměřením na jeho nedostatky. Cílem této práce je vyvinout nový konvertor, který se bude snažit zjištěné nedostatky snažit odstranit. Tyto cíle jsou stanoveny v sekci 6.2.

6.1 Popis současného konvertoru

Současný konvertor používaný ve zpětném překladači není založený na žádném formálním algoritmu a strukturování je tedy prováděno pomocí vlastního algoritmu, který je však nedokonalý. Strukturování probíhá tak, že je postupně iterováno přes jednotlivé instrukce a přímo v místě aktuální pozice se provádí převod na reprezentaci v BIR. Konvertor pracuje pouze s malým kontextem a v některých případech je kód procházen zbytečně vícekrát.

Zdrojový kód tohoto řešení se nachází ve dvou souborech a několika dalších podpůrných souborech. Toto rozdělení není dostatečné a v jednom souboru se nachází nesouvisející analýzy. Celé řešení je tedy nepřehledné. Řešení není otestováno jednotkovými testy a výsledný kód obsahuje chyby z hlediska strukturování. Na obrázku 6.1 je ukázka vygenerovaného kódu, který však nebyl strukturován kompletně, ale obsahuje pouze komentář, že byla detekována možná rekurze.

```
if (v7 < v28) {
  lab_0x11bd0_18:
    // 0x11bd0
    v15 = (int32_t *)(g10 + 8);
    v44 = *v15;
    *v15 = v44 - 1;
    if (v44 > 0) {
        goto lab_0x11bc4_10;
    }
    // Detected a possible infinite recursion (goto support failed); quitting...
} else {
    v29 = v28;
}
```

Obrázek 6.1: Příklad výstupu, kdy je do kódu vložen komentář o možném zacyklení, a proto část těla chybí (výňatek z kódu)

Ukázka výstupu, kdy původní konvertor generuje zbytečný duplicitní kód, je na obrázcích 6.2b a 6.2c. Zde je vidět, že návěští `case` s konstantami `0`, `1` a `2` obsahují naprosto stejný kód. Že se jedná o shodný kód lze poznat z generovaných komentářů s adresami základních bloků v původním binárním programu. Kód původního programu je na obrázku 6.2a.

```
int ch;
scanf("%d", &ch);

switch (ch) {
    case 0:
    case 1:
    case 2:
    case 3:
        printf("d\n");

    case 4:
        printf("e\n");

    case 5:
        printf("f\n");

    case 6:
        printf("g\n");
        break;

    default:
        printf("default\n");
        break;
}

printf("after switch");
return 0;
```

(a) Původní kód

```
int32_t v1;
scanf("%d", &v1);
switch (v1) {
    default: {
        // 0x80485f7
        puts("default");
        // branch -> 0x8048603
        break;
    }
    case 0: {
        // 0x804857f
        puts("d");
        // branch -> 0x804858b
        // 0x804858b
        puts("e");
        // branch -> 0x8048597
        // 0x8048597
        puts("f");
        // branch -> 0x80485a3
        // 0x80485a3
        puts("g");
        // branch -> 0x8048603
        break;
    }
    case 1: {
        // 0x804857f
        puts("d");
        // branch -> 0x804858b
        // 0x804858b
        puts("e");
        // branch -> 0x8048597
        // 0x8048597
        puts("f");
        // branch -> 0x80485a3
        // 0x80485a3
        puts("g");
        // branch -> 0x8048603
        break;
    }
```

(b) Výstup zpětného překladače (výňatek kódu)

```
    case 2: {
        // 0x804857f
        puts("d");
        // branch -> 0x804858b
        // 0x804858b
        puts("e");
        // branch -> 0x8048597
        // 0x8048597
        puts("f");
        // branch -> 0x80485a3
        // 0x80485a3
        puts("g");
        // branch -> 0x8048603
        break;
    }
    case 3: {
        // 0x804857f
        puts("d");
        // branch -> 0x804858b
    }
    case 4: {
        // 0x804858b
        puts("e");
        // branch -> 0x8048597
    }
    case 5: {
        // 0x8048597
        puts("f");
        // branch -> 0x80485a3
    }
    case 6: {
        // 0x80485a3
        puts("g");
        // branch -> 0x8048603
        break;
    }
}
// 0x8048603
printf("after switch");
```

(c) Výstup zpětného překladače (pokračování)

Obrázek 6.2: Příklad výstupu, kdy je zbytečně generován duplicitní kód

6.2 Stanovení cílů práce

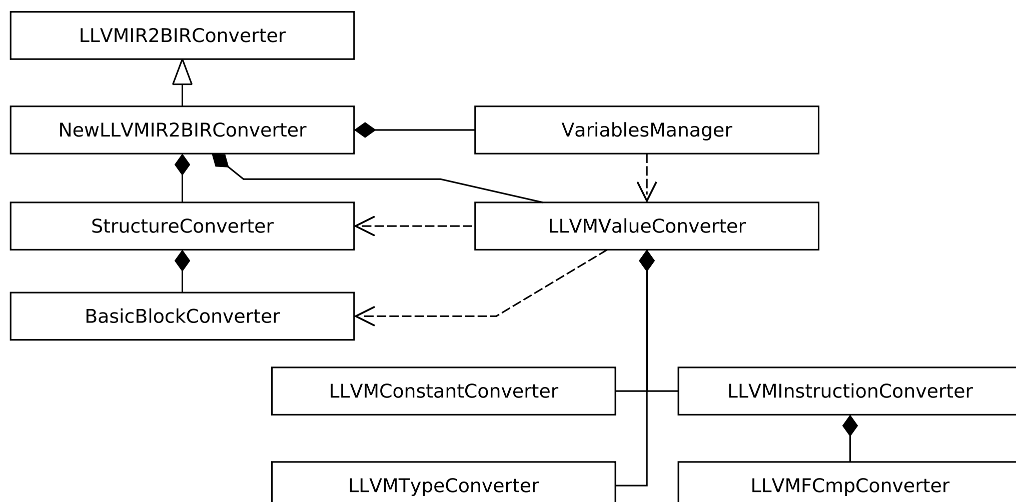
Cílem této práce je navržení a implementace nového konvertoru, který se bude snažit odstranit nedostatky původního řešení. Nový konvertor by tedy měl být založen na nějakém formálním algoritmu, dále by měl produkovat lépe strukturovaný kód. Z hlediska zdrojového kódu by měl být rozdělen do menších modulů, které budou plnit právě jednu funkci. V neposlední řadě by měl být důkladně otestován jednotkovými testy.

Kapitola 7

Návrh konvertoru

Tato kapitola popisuje návrh konvertoru zdrojového kódu z reprezentace v jazyce LLVM IR do reprezentace v BIR. Konverze se skládá z několika kroků: nejprve jsou konvertovány hlavičky funkcí, následně globální proměnné a posledním a nejdůležitějším krokem je převod samotných těl funkcí. Na převod těl funkcí se lze dívat ze dvou úrovní abstrakce. Na nižší úrovni abstrakce je potřeba řešit konverzi základního bloku, tedy posloupnosti instrukcí, na posloupnost příkazů v BIR. Na vyšší úrovni abstrakce je potřeba řešit samotné strukturování kódu, tedy konverzi skoků na vysokoúrovňové podmíněné příkazy a cykly.

Na obrázku 7.1 je znázorněn zjednodušený diagram tříd, které slouží ke konverzi funkce. Řídící třídou konverze je třída `NewLLVMIR2BIRConverter`, která zajišťuje konverzi globálních proměnných, hlaviček funkcí a koordinaci konverze těla funkce. Návrh této třídy je popsán v následující sekci. Pro správu proměnných slouží třída `VariablesManager`, jejíž návrh je popsán v sekci 7.2. Ke konverzi výrazů slouží třída `LLVMValueConverter` a další závislé třídy. Popis návrhu těchto tříd je uveden v sekci 7.3. Ke konverzi jednotlivých příkazů v základních blocích slouží třída `BasicBlockConverter`, jejíž návrh je popsán v sekci 7.4. Třída `StructureConverter` slouží ke konverzi funkce na vyšší úrovni abstrakce a její návrh je popsán v sekci 7.5.



Obrázek 7.1: Závislosti tříd zajišťujících konverzi LLVM IR do BIR

7.1 Konvertor z LLVM IR do BIR

Třída `NewLLVMIR2BIRConverter` je řídící třídou konverze a zajišťuje konverzi globálních proměnných, hlaviček funkcí, těl funkcí a zajištění validity identifikátorů. V následujících sekcích jsou popsány jednotlivé kroky, které jsou zde vykonávány.

7.1.1 Konverze hlaviček funkcí

Konverze hlaviček funkcí musí být prováděna zvlášť a musí předcházet konverzi globálních proměnných i konverzi těl funkcí. Konverzi globálních proměnných musí předcházet, protože globální proměnná může obsahovat ukazatel na funkci, tedy je důležité, aby všechny funkce již byly ve výsledném modulu v BIR deklarovány. Konverzi těl funkcí musí předcházet z důvodu příkazu volání funkce, který se může nacházet uvnitř těla konvertované funkce, přičemž volaná funkce může být definovaná až za funkcí, ze které je volána. Dále je to důležité pro řešení rekurzivních volání, protože funkce v průběhu konverze těla by ještě nebyla definována. Jedná se tedy o první průchod funkcemi a uložení jejich hlaviček do výsledného modulu v BIR, aby bylo možné se na tyto funkce odkazovat.

7.1.2 Konverze globálních proměnných

Konverze globálních proměnných z LLVM IR do BIR probíhá tak, že se prochází všechny globální proměnné z LLVM IR a vkládají se do výsledného modulu v BIR. Pokud má globální proměnná inicializátor, je tento inicializátor konvertován jako výraz a přiřazen ke konvertované globální proměnné. Konverze výrazu je popsána v sekci 7.3.

7.1.3 Konverze těla funkce

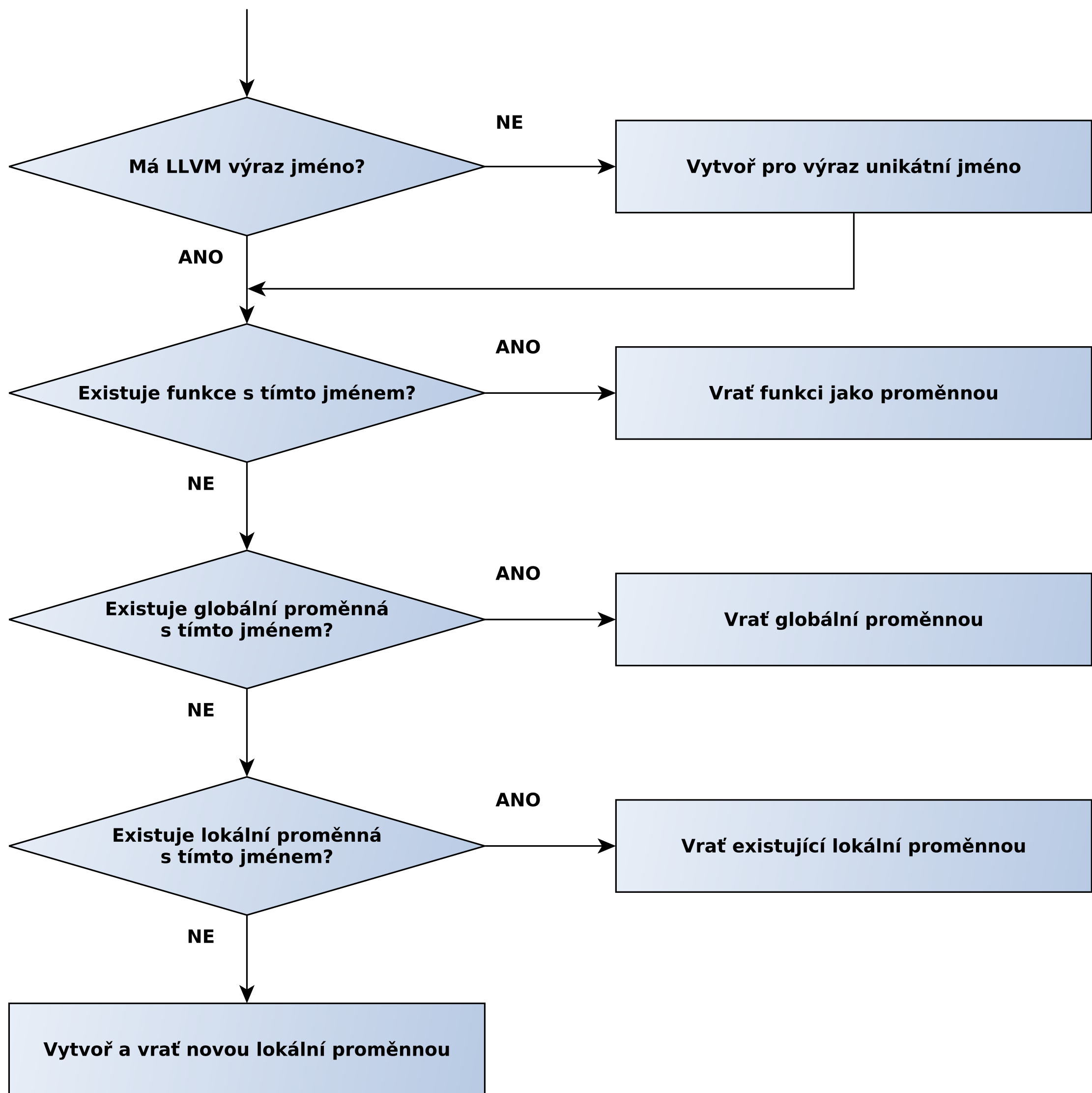
V rámci konverze těla funkce je potřeba udržovat seznam aktuálně použitých lokálních proměnných, jelikož některé výrazy je potřeba ukládat do proměnných, zatímco jiné jsou použity přímo. Na začátku konverze těla funkce je potřeba tento seznam vyprázdnit. V následujících sekcích je popsána konverze těla funkce z již zmiňovaných dvou úrovní abstrakce – konverze základního bloku a řešení strukturování.

7.1.4 Zajištění validity identifikátorů

Posledním krokem, který je nutné v rámci této třídy řešit, je zajištění validity identifikátorů. Jedná se o vypuštění znaků, které jsou v identifikátoru LLVM validní, ale v cílovém vysokoúrovňovém jazyce validní nejsou. Dále je potřeba zajistit jedinečnost identifikátorů, jelikož při vypuštění některých znaků by mohlo docházet ke kolizím.

7.2 Správce proměnných

Jelikož je na více místech potřebné přistupovat k existujícím proměnným a případně vytvářet další, je vhodné oddělení správy proměnných do zvláštní třídy. Správce proměnných je reprezentován třídou `VariablesManager`. Tato třída zajišťuje vrácení stejné instance proměnné v BIR pro stejný výraz v LLVM a dále zajišťuje unikátní pojmenování výrazů, které v LLVM nemají své jméno. Rovněž tento správce umožňuje kontrolovat již existující funkce a globální proměnné, které musí mít přednost před vytvářením lokální proměnné. Postup získávání proměnné je naznačen vývojovým diagramem na obrázku 7.2.



Obrázek 7.2: Vývojový diagram postupu pro získání proměnné

7.3 Konverze výrazů

Konverzí výrazů se souhrnně myslí konverze konstant a instrukcí na výrazy. Je zde potřeba rozlišovat několik typů výrazů – globální proměnná, konstanta, funkce a instrukce. Instrukce se rozlišují na ty, které jsou použity jako výraz a na ty, které jsou použity jako proměnné (tzn. tyto instrukce již byly dříve konvertovány a uloženy do proměnné). Globální proměnné, lokální proměnné a funkce jsou vráceny jako objekt v BIR, který již byl vytvořen v rámci předchozích konverzí. V následujících sekcích jsou popsány konverze jednotlivých typů výrazů.

7.3.1 Konverze konstant

Konstanty jsou konvertovány do BIR v rámci třídy `LLVMConstantConverter` dle jejich typu podle následujících pravidel:

- Celočíselná konstanta bitové šířky 1 je konvertována na `ConstBool`.
- Celočíselná konstanta bitové šířky větší než 1 je konvertována na `ConstInt`.
- Číselná konstanta s plovoucí desetinnou čárkou je konvertována na `ConstFloat`.
- Konstantní pole je konvertováno na `ConstArray`.
- Konstantní struktura je konvertována na `ConstStruct`.
- Konstantní nulový ukazatel je konvertován na `ConstNullPointer`.
- Konstanta `zeroinitializer` je konvertována na konstantu odpovídajícího typu, která je inicializována na nulovou hodnotu. Konkrétně celé číslo má nulovou hodnotu 0, číslo s plovoucí desetinnou čárkou má hodnotu 0.0, logická hodnota má hodnotu `false`, ukazatel má hodnotu `null` a složené výrazy se skládají z nulových hodnot těchto jednoduchých typů.
- Konstantní výrazy jsou převedeny podobně jako instrukce, které jsou popsány níže.

7.3.2 Konverze instrukcí

Instrukce jsou konvertovány do BIR v rámci třídy `LLVMInstructionConverter` podle následujících pravidel:

- Instrukce `bitcast` je konvertována na `BitCastExpr`.
- Instrukce `fpext`, `sext` a `zext` jsou konvertovány na `ExtCastExpr`, kde je nastavena odpovídající varianta přetypování (tedy rozšíření čísla s plovoucí desetinnou čárkou, znaménkového nebo neznaménkového čísla).
- Instrukce `fptosi` a `fptoui` jsou konvertovány na `FPToIntCastExpr`, kde se nerozlišuje, zda původní instrukce byla konverzí na znaménkové nebo neznaménkové číslo.
- Instrukce `trunc` a `fptrunc` jsou konvertovány na `TruncCastExpr`, kde se nerozlišuje, zda původní instrukce byla ořezem celého čísla nebo čísla s plovoucí desetinnou čárkou.
- Instrukce `inttoptr` je konvertována na `IntToPtrCastExpr`.
- Instrukce `ptrtoint` je konvertována na `PtrToIntCastExpr`.
- Instrukce `sitofp` a `uitofp` jsou konvertovány na `IntToFPCastExpr`, kde je nastavena odpovídající varianta (tedy zda se jedná o přetypování znaménkového nebo neznaménkového čísla).
- Instrukce `icmp` a `fcmp` jsou konvertovány dle přítomného porovnávacího predikátu na `EqOpExpr`, `NeqOpExpr`, `GtOpExpr`, `GtEqOpExpr`, `LtOpExpr`, `LtEqOpExpr` nebo `EqOpExpr`. Porovnání čísel s plovoucí desetinnou čárkou je řešeno zvlášť, protože ve zpětném překladači lze nastavit zjednodušování takových porovnání.

- Instrukce `select` je konvertována na `TernaryOpExpr`.
- Instrukce `call` je konvertována na `CallExpr`.
- Instrukce `getelementptr` je konvertována na posloupnost výrazů `ArrayIndexOpExpr` nebo `StructIndexOpExpr`, kdy tyto výrazy mohou být libovolně zanořeny.
- Instrukce `extractvalue` je konvertována obdobně jako instrukce `getelementptr`, tedy jako posloupnost výrazů `ArrayIndexOpExpr` nebo `StructIndexOpExpr`.

7.3.3 Konverze typů

V rámci konverze většiny výrazů je potřeba konvertovat i datový typ. Typy jsou konvertovány do BIR v rámci třídy `LLVMTypeConverter` podle následujících pravidel:

- Celočíselný typ je konvertován na `IntType`.
- Typ čísla s plovoucí desetinnou čárkou je konvertován na typ `FloatType`.
- Typ pole je konvertován na typ `ArrayType`.
- Typ struktury je konvertován na typ `StructType`.
- Typ ukazatele je konvertován na typ `PointerType`.
- Typ funkce je konvertován na typ `FunctionType`.
- Bezrozměrný typ `void` je konvertován na typ `VoidType`.

7.4 Konverze základního bloku

Za konverzi základního bloku je zodpovědná třída `BasicBlockConverter`, která pracuje tak, že postupně prochází všechny instrukce daného základního bloku a konvertuje je na odpovídající příkaz v BIR. Některé instrukce však není žádoucí konvertovat na příkaz, jelikož by vznikalo velké množství proměnných. Takové instrukce jsou v této fázi přeskočeny a jsou konvertovány až později jako operandy jiné instrukce. Instrukce, které nejsou přeskočeny jsou konvertovány následujícím způsobem:

- Instrukce `call` je konvertována jako volání funkce, tedy příkaz typu `CallStmt`.
- Instrukce `load` je konvertována jako načtení hodnoty z paměti, tedy uložení výrazu do proměnné. Výsledný příkaz v BIR je typu `AssignStmt`, kde výraz na levé straně je proměnná a výraz na pravé straně přiřazení dereferencí výrazu.
- Instrukce `store` je konvertována jako uložení hodnoty do paměti. Výsledný příkaz v BIR je typu `AssignStmt`, kde výraz na levé straně je dereferencí proměnné a na pravé straně přiřazení je libovolný výraz.
- Instrukce `insertvalue` je konvertována jako posloupnost dvou příkazů, a to přiřazení původního pole do nové proměnné a poté jako posloupnost výrazů `ArrayIndexOpExpr` nebo `StructIndexOpExpr`, kdy je do konkrétního prvku struktury nebo pole vložena hodnota.

V běžných případech se však instrukce `load` a `store` používají pro přístup k jednoduchým proměnným (např. celé číslo). V takových případech je v rámci zvýšení čitelnosti kódu provedena změna z ukazatele na samotnou proměnnou a odpovídajícím způsobem upraveny výrazy v BIR. Konkrétně je dereference zrušena a místo ní použita samotná proměnná.

7.5 Strukturování funkce

Třída `StructureConverter` zajišťuje vytvoření správné vysokoúrovňové struktury funkce. Pracuje na úrovni základních bloků, ze kterých postupným spojováním vytváří podmíněné příkazy, cykly a posloupnosti příkazů. V následující sekci je popsán princip tohoto strukturování a následně jsou podrobně popsány jednotlivé kroky algoritmu.

7.5.1 Princip

Na obrázku 7.3 je znázorněn zjednodušený vývojový diagram pro strukturování funkce. Nejprve se ze vstupního LLVM IR vytvoří reprezentace grafu toku řízení. Následně se detekují zpětné hrany. Poté se postupně provádí redukce tohoto grafu dle shody s předdefinovanými tvary. Pokud se nepodaří kompletně strukturovat tělo funkce, tak se zbývající uzly propojí příkazy `goto`.

7.5.2 Vytvoření grafu toku řízení dané funkce

Pro vytvoření grafu toku řízení dané funkce je použito prohledávání do šířky přes jednotlivé základní bloky v LLVM IR. Při průchodu každého základního bloku je konvertováno jeho tělo, k čemuž je využita třída `BasicBlockConverter`. Na obrázku 7.4a je znázorněn příklad vstupního kódu v LLVM IR a na obrázku 7.4b je znázorněn graf toku řízení, který je ze vstupního kódu vytvořen.

V následujícím kroku jsou v grafu toku řízení detekovány zpětné hrany, čehož je následně využito v analýzách grafu. Detekce zpětných hran probíhá pomocí prohledávání do hloubky. Na obrázku 7.4c jsou červeně znázorněny zpětné hrany, které byly detekovány v grafu toku řízení z obrázku 7.4b.

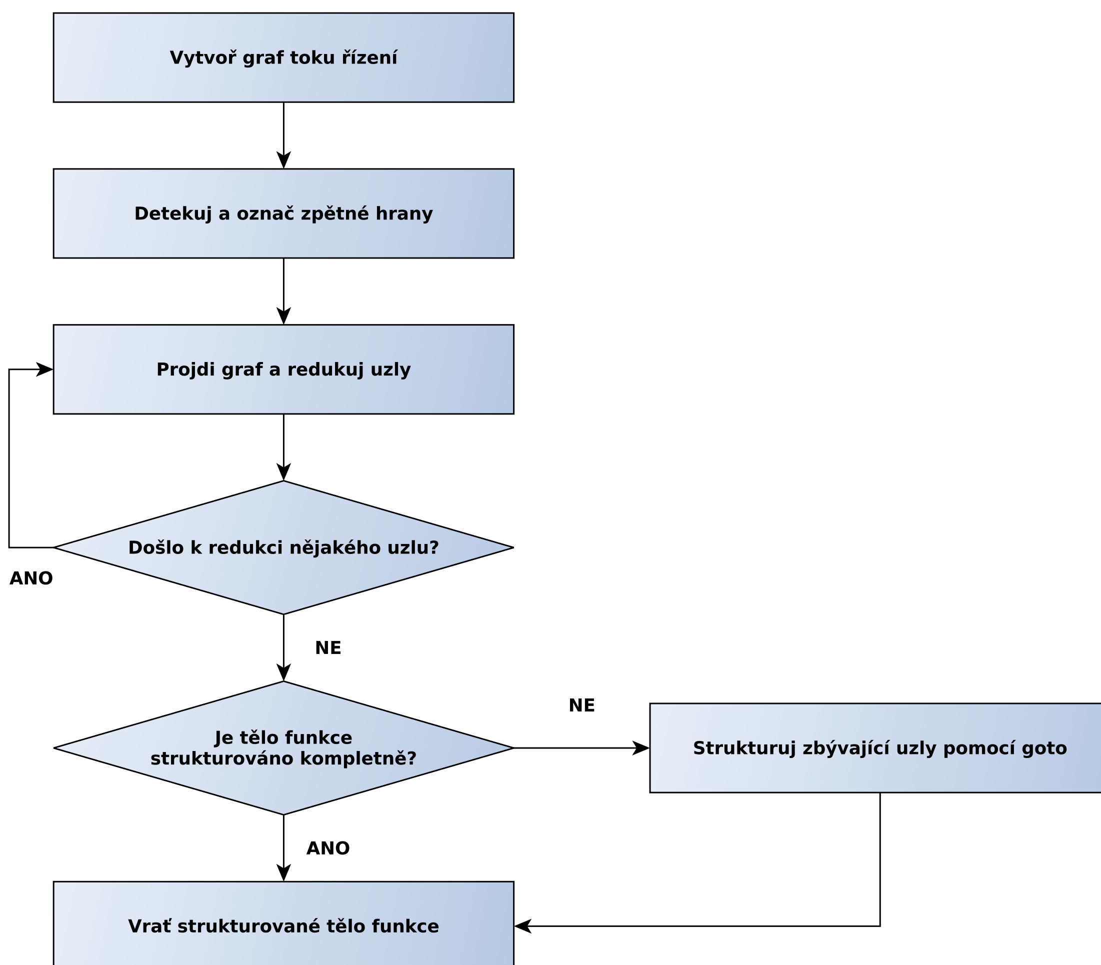
7.5.3 Postupná redukce grafu toku řízení

Redukce grafu probíhá iterativně tak dlouho, dokud není graf kompletně redukován. K zastavení redukce dojde také v případě, že již není možné graf dále redukovat. V každé iteraci se prochází celý graf toku řízení funkce metodou prohledávání do šířky a u každého uzlu probíhá kontrola, jestli ho lze spolu s jeho okolím redukovat. Pokud lze uzel redukovat, tak dojde k jeho redukci a k nastavení příznaku, že má být provedena další iterace. Detekce jednotlivých konstrukcí probíhá na základě shody s předdefinovanými vzory, které jsou podrobněji popsány v následujícím textu.

Tento princip je založen na článku *Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring* [7]. Ukázka postupu redukce funkce je naznačena v příloze B na obrázku B.1.

Detekce a redukce sekvencí příkazů

Sekvence příkazů je nejjednodušším tvarem, který lze v grafu detekovat. Jedná se o uzel, který je následován právě jedním dalším uzlem, přičemž následník má pouze jediného před-



Obrázek 7.3: Vývojový diagram redukce funkce

chůdce. Graficky je tento tvar znázorněn na obrázku 7.5a. V případě shody probíhá redukce tak, že je tělo následníka připojeno na konec těla prvního uzlu a tyto dva uzly jsou sloučeny.

Detekce a redukce podmínek

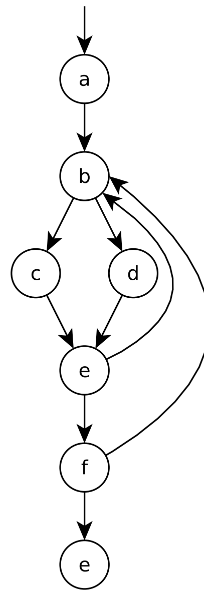
Podmíněné příkazy `if` mohou být tří základních tvarů, a to příkaz `if` bez větve `else` (ukázka je na obrázku 7.5b), příkaz `if`, kdy větev končí příkazem `return` (ukázka je na obrázku 7.5c) a příkaz `if` s větví `else` (ukázka je na obrázku 7.5d).

Redukce příkazu `if` bez větve `else` probíhá tak, že na konec uzlu `a` (z obrázků 7.5b a 7.5c) je připojen nově vytvořený příkaz `IfStmt` v BIR a do jeho těla je vloženo tělo uzlu `b`, který je následně z grafu smazán.

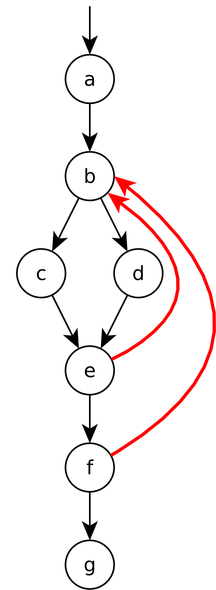
Obdobně probíhá také redukce příkazu `if` s větví `else`. Zde je na konec uzlu `a` (z obrázku 7.5d) připojen nově vytvořený příkaz `IfStmt` v BIR a do jeho těla je vloženo tělo uzlu `b`. Dále je k tomuto příkazu `IfStmt` přidána větev `else`, do které je vloženo tělo uzlu `c`. Uzly `b` a `c` jsou následně smazány a k uzlu `a` je připojen nový následník, a to uzel `d`.

```
define void @function(i32 %x, i32 %y) {
    a:
        br label %b
    b:
        %0 = phi i32 [ 0, %a ], [ %5, %d ]
        call void @test(i32 1)
        %1 = icmp slt i32 %0, %x
        br i1 %1, label %c, label %d
    c:
        %2 = phi i32 [ 0, %b ], [ %3, %c ]
        call void @test(i32 2)
        %3 = add i32 %2, 1
        %4 = icmp slt i32 %2, %i
        br i1 %4, label %c, label %d
    d:
        call void @test(i32 3)
        %5 = add i32 %i, 1
        %6 = icmp eq i32 %i, %y
        br i1 %6, label %e, label %b
    e:
        call void @test(i32 4)
        ret void
}
```

(a) Jednoduchý příklad funkce v jazyce LLVM IR

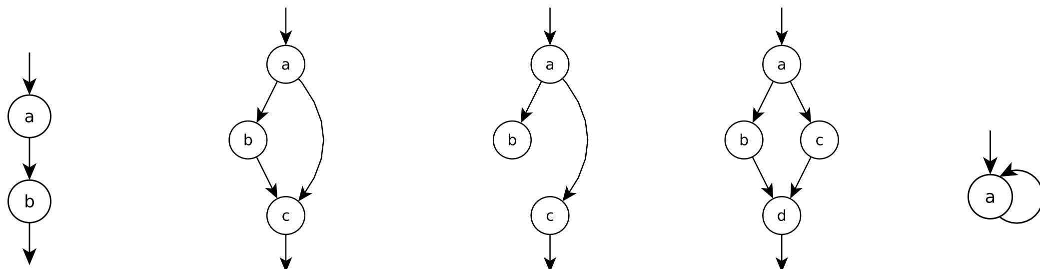


(b) Graf toku řízení funkce



(c) Graf toku řízení funkce s detekovanými zpětnými hranami

Obrázek 7.4: Ukázka převodu LLVM na graf toku řízení



(a) Sekvence příkazů

(b) Příkaz `if`

(c) Příkaz `if` s ukončující větví

(d) Příkaz `if-else`

(e) Cyklus

Obrázek 7.5: Ukázka tvarů sekvence, podmíněného příkazu `if` a cyklů

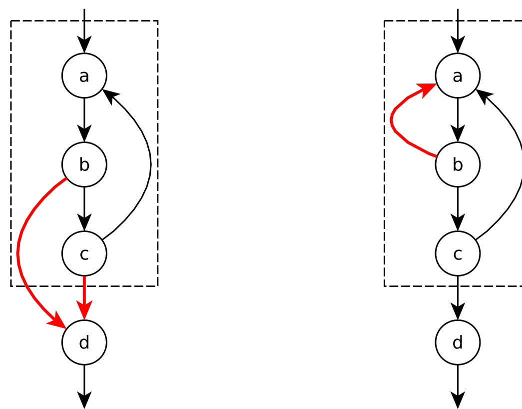
Detekce a redukce cyklů

K detekci cyklu je použita analýza vestavěná v systému LLVM [13]. Je zde však potřeba řešit příkazy `break` a `continue` uvnitř cyklů. Při vstupu do každého cyklu dojde k uložení hlavičky tohoto cyklu a dále k nalezení následníka cyklu. Jako následník cyklu je označen první uzel, který se nachází mimo cyklus. Ke hledání následníka cyklu je použito prohledávání grafu do šířky.

Detekce příkazu `break` probíhá tak, že hrana grafu směřuje zevnitř cyklu na následníka

cyklu. Takovéto hrany jsou graficky znázorněny na obrázku 7.6a červenou barvou (cyklus je ohraničen čárkovanou čarou). Příkaz `break` je řešen v rámci podmíněného příkazu i u uzlů s jediným následníkem. Stejným způsobem může být nalezen také příkaz `break`, který ukončuje nadřazený cyklus, ale takové příkazy je nutné strukturovat pomocí příkazu `goto`. Důvodem je, že cílový jazyk může být např. jazyk C, který podporuje pouze příkaz `break` pro nejbližší nadřazený cyklus.

Podobným způsobem probíhá i nahrazování příkazů `continue` – jedná se o hrany směřující zevnitř cyklu na hlavičku cyklu. Grafické znázornění je na obrázku 7.6b, kde jsou hrany znamenající příkaz `continue` zvýrazněny červeně (cyklus je ohraničen čárkovanou čarou). Jako `continue` však nebude strukturována hrana směřující z posledního uzlu uvnitř cyklu, jelikož by tím došlo k odstranění této hrany a později by nemohl být detekován cyklus (na obrázku 7.6b se jedná o hranu směřující z uzlu `c` do uzlu `a`).



(a) Příkaz `break`   (b) Příkaz `continue`

Obrázek 7.6: Ukázka detekce příkazů `break` a `continue` v cyklech

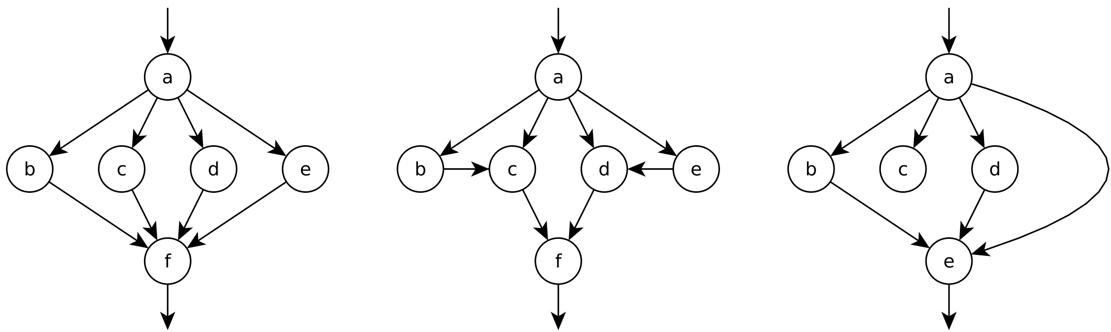
Detekce a redukce příkazů `switch`

Příkaz `switch` je v LLVM IR reprezentován stejnojmennou instrukcí `switch`. Není však možné provést redukci uzlu ihned. Tuto redukci je možné provést až ve chvíli, kdy jsou těla všech větví `case` kompletně redukována. Na obrázku 7.7 jsou znázorněny různé případy příkazu `switch`.

Redukce následně probíhá tak, že je nalezen následník příkazu `switch`, což je uzel, který se nachází za všemi uzly reprezentujícími jednotlivá těla `case`. Současně s tímto krokem také probíhá detekce, zda příkaz `switch` má větev `default`. Příklady na obrázcích 7.7a a 7.7b obsahují větev `default`, ale příklad na obrázku 7.7c větev `default` nemá.

Dále dojde k seřazení větví `case` tak, aby mohly být vygenerovány jako souvislý kód s ohledem na větve, které nekončí příkazem `break`, ale pokračují tzv. „propadem“ (anglicky fall-through) ve vykonávání těla následujícího `case`. V příkladu na obrázku 7.7b je při řazení potřeba provést prohození uzlů `d` a `e`, jelikož uzel `e` pokračuje dále do uzlu `d`. Ostatní uzly jsou seřazeny správně.

V uvedených příkladech je na konec těla uzlu `a` vygenerován nový příkaz `SwitchStmt` a ve správném pořadí jsou vložena těla jednotlivých `case`, případně i větve `default`, pokud ji příkaz `switch` obsahuje. Následně jsou všechny uzly reprezentující větve smazány.



(a) Příkaz `switch`

(b) Příkaz `switch` s propady do následující větve `case`

(c) Příkaz `switch` s větví `case` ukončenou příkazem `return` a bez větve `default`

Obrázek 7.7: Ukázka různých tvarů příkazu `switch`

7.5.4 Strukturování pomocí příkazu goto

Pokud se graf toku řízení dané funkce dostane do stavu, že jej není možné již dále strukturovat dle předdefinovaných tvarů, je nutné zbytek grafu strukturovat pomocí příkazu `goto`. Princip je takový, že se vytvoří příkaz `IfStmt` nebo `SwitchStmt` (dle typu uzlu) a do těla všech větví se umístí příkaz `GotoStmt` s konkrétním cílem. Poté se odstraní všichni následníci tohoto uzlu a pokračuje se dalším uzlem v pořadí.

Aby se předešlo tomu, že by některé cíle příkazu `goto` nebyly vygenerovány do cílového kódu, tak je potřeba všechny cíle `goto` ukládat do množiny. Na konec kódu jedné funkce pak dojde k vygenerování k vygenerování všech cílů `goto`, které se ještě v kódu nenachází.

Kapitola 8

Implementace konvertoru

Tato kapitola se věnuje implementaci konvertoru, jehož návrh je v kapitole 7. Implementace je provedena v programovacím jazyce C++ dle standardu ISO C++14 [10]. Pro automatizaci překladu je využit systém *CMake* [1]. Všechny komentáře ve zdrojových kódech jsou psány ve formátu, který je podporován systémem *doxygen* [2] a je z nich vygenerována aplikační dokumentace. Dále je využívám distribuovaný systém pro správu verzí *git* [3].

Na přiloženém CD se zdrojové kódy konvertoru implementovaného v rámci této práce nachází v adresáři `src/decompiler/decdev/backend/bir/llvm/llvmir2bir_converters`.

8.1 Třída `NewLLVMIR2BIRConverter`

Řídící třídou konvertoru z LLVM IR do BIR je třída `NewLLVMIR2BIRConverter`. Tato třída obsahuje hlavní metodu `convert()`, která slouží pro provedení konverze vstupního LLVM modulu do výstupního modulu v BIR.

Jak již bylo zmíněno v kapitole 7, proces konverze se skládá ze tří kroků – konverze hlaviček funkcí, konverze globálních proměnných a konverze těl funkcí. Tyto kroky zajišťují metody `convertAndAddFuncsDeclarations()`, `convertAndAddGlobalVariables()` a `convertFuncsBodies()`.

8.2 Třída `VariablesManager`

Třída `VariablesManager` slouží ke správě proměnných a je implementována dle popisu v sekci 7.2. Pro uložení již existujících proměnných je využit kontejner `std::unordered_map`. Pro generování nových jmen proměnných je využit již existující generátor `NumVarNameGen`, který vrací proměnné pojmenované jako `var1`, `var2` atd.

8.3 Třída `LLVMValueConverter`

Při vytvoření instance třídy `LLVMValueConverter` dojde k vytvoření instancí všech závislých tříd – `LLVMTypeConverter`, `LLVMInstructionConverter` a `LLVMConstantConverter`. Dále je k běhu potřebná instance třídy `VariablesManager` a výsledný modul v BIR. Hlavními metodami, které tato třída poskytuje, jsou:

- Metoda `convertValueToDerefExpression()` – tato metoda slouží pro konverzi LLVM výrazu na výraz v BIR, kdy je požadována, aby výsledkem byla dereference výrazu.

Pokud je předaný výraz ukazatelem, ale ve skutečnosti nemá být považován za ukazatel (což nastává u většiny instrukcí `alloca`), tak není vytvářena dereference, ale je vrácen pouze výraz převedený do BIR. Dále je zde drobná optimalizace pro výrazy, které jsou operátory pro získání adresy nějakého výrazu. V takových případech není nutné dereferencovat tento složitý výraz, ale stačí vrátit výraz, který je operandem operátoru získání adresy.

- Metoda `convertValueToExpression()` – tato metoda je nejčastěji využívanou metodou a slouží pro konverzi LLVM výrazu na výraz v BIR. Uvnitř se pouze volá metoda `convertValueToExpressionDirectly()`. Pokud je jejím výsledkem řetězec nebo výraz, který již je ukazatelem, je vrácen samotný výraz. V opačném případě je vrácen výraz pro získání adresy tohoto výrazu.

- Metoda `convertValueToExpressionDirectly()` – tato metoda slouží k samotnému rozhodování, jak má být výraz převeden do BIR. Pokud se jedná o konstantu, tak je provedena konverze konstanty pomocí třídy `LLVMConstantConverter`. Pokud se jedná o instrukci, tak je potřeba rozhodnout, zda má být konvertována na výraz a nebo na proměnnou. K tomuto rozhodnutí je použita již existující statická metoda `LLVMSupport::isInlinableInst()`, která je převzatá z původního řešení. Instrukce konvertovatelné na výraz jsou tedy rovnou konvertovány na výraz a ze všech ostatních LLVM výrazů je vytvořena proměnná.

- Metoda `convertValueToVariable()` – tato metoda slouží k vrácení proměnné, která odpovídá konkrétnímu LLVM výrazu. Cílem je, aby byla vrácena stejná proměnná pro stejný výraz, a aby tato proměnná měla správný typ. Ke správě proměnných je použit `VariablesManager`, který byl popsán výše.

8.4 Třída `LLVMTypeConverter`

Třída `LLVMTypeConverter` poskytuje přetíženou metodu `convert()`, která slouží pro konverzi různých LLVM typů na typ v BIR. Samotná konverze probíhá dle pravidel popsaných v sekci 7.3.3. V případě složených typů jsou typy jednotlivých položek konvertovány rekurzivně.

V rámci implementace bylo potřeba řešit rekurzivní typy, jelikož např. definice struktury může obsahovat položku, kde obsahuje ukazatel sama na sebe. Z tohoto důvodu je vždy na začátku konverze provedena kontrola, zda již stejný typ nebyl konvertován dříve. V případě shody je vrácen již existující typ v BIR. Pro uložení existujících typů je využito kontejneru `std::unordered_map`.

8.5 Třída `LLVMInstructionConverter`

Třída `LLVMInstructionConverter` provádí konverze LLVM instrukcí na výrazy v BIR. Pro konverzi je využit návrhový vzor „návštěvník“ [15], který je podporován samotným systémem LLVM. Konverze instrukcí je poměrně přímočará, pouze dochází k vytváření nových instancí tříd v BIR dle popisu v sekci 7.3.2. Podobně jako u konverze typů jsou vnořené výrazy konvertovány rekurzivně.

Konverze LLVM instrukce `fcmp` je z důvodu přehlednosti oddělena do samostatné třídy `LLVMFCmpConverter`, jelikož je zde potřeba rozlišovat dvě možná nastavení zpětného pře-

kladu. Jedná se o nastavení, zda má být použita striktní sémantika porovnání čísel s plovoucí řádovou čárkou. V případě striktního režimu se vytváří výraz, který je složen ze tří logických součtů kde prvním je samotný výraz porovnání. Druhý a třetí slouží ke kontrole, zda je první nebo druhý operand nedefinované číslo. Implementace kontroly, zda je číslo nedefinované je výraz `a != a`.

8.6 Třída `LLVMConstantConverter`

Třída `LLVMConstantConverter` provádí konverze LLVM konstant na konstanty v BIR. Konverze konstant je taktéž přímočará. Dle pravidel popsaných v sekci 7.3.1 se provádí vytváření instancí tříd reprezentujících konstanty v BIR. Konstanty složených typů jsou opět konvertovány rekurzivně.

8.7 Třída `BasicBlockConverter`

Třída `BasicBlockConverter` slouží ke konverzi posloupnosti instrukcí jednoho základního bloku. Nejsou však konvertovány všechny instrukce, ale pouze ty, které mají tvořit samostatný příkaz ve výsledném kódu. K tomuto rozhodnutí je použita již existující statická metoda `LLVMSupport::isInlinableInst()`, která je převzatá z původního řešení. Dále nejsou konvertovány následující instrukce:

- Instrukce `alloca`, protože tato instrukce slouží pouze k alokaci proměnné a v konvertoru je proměnná vytvořena automaticky při prvním použití.
- Instrukce `br`, `switch` a `phi`, protože tyto instrukce jsou konvertovány v rámci třídy `StructureConverter`.

Pro konverzi instrukcí je využit návrhový vzor „návštěvník“, stejně jako u konverze instrukcí ve třídě `LLVMInstructionConverter`.

8.8 Třída `StructureConverter`

Třída `StructureConverter` je nejsložitější třídou celé implementace a představuje jádro samotného strukturování. Implementace je provedena na základě návrhu v kapitole 7.5.

8.8.1 Vytvoření grafu toku řízení

K vytvoření grafu toku řízení je využita metoda procházení do šířky, která je implementována za pomoci fronty `std::queue`. Do této fronty se přidávají základní bloky, které ještě nebyly navštíveny. Dále pak za pomoci asociativního pole `std::unordered_map`, do kterého se ukládají již navštívené základní bloky a k nim odpovídající vytvořené uzly v grafu toku řízení. Pro reprezentaci jednoho uzlu v grafu toku řízení je využita třída `CFGNode`.

8.8.2 Detekce zpětných hran

Pro detekci zpětných hran je využita metoda procházení do hloubky. Tato metoda je implementována za pomoci zásobníku `std::stack` a asociativního pole `std::unordered_map`, do kterého se ukládají stavy uzlů (nenavštívený, navštívený, navštívený a uzavřený). Zpětná hrana je detekována tak, že vede z aktuálního uzlu do jiného uzlu, jehož stav je navštívený.

8.8.3 Iterace redukce grafu toku řízení

Jedna iterace grafu toku řízení spočívá v kompletním průchodu grafu pomocí metody prohledávání do šířky. Tato metoda je při následných analýzách potřebná několikrát, proto je implementována v metodě `BFSTraverse()`. Pro každý navštívený uzel je volána metoda `inspectCFGNode()`, kde probíhá v daném pořadí kontrola na shodu s konkrétními tvary, tak jak bylo popsáno v kapitole s návrhem. Pokud dojde ke shodě, tak je provedena redukce uzlu a vytvořena příslušná konstrukce v BIR.

V průběhu implementace je použito několik pomocných kontejnerů, do kterých se ukládají tyto informace

- Hlavičky cyklů.
- Dvojice základních bloků, pro které již byly vygenerovány přiřazení do instrukcí `phi`.
- Množina již redukovaných cyklů.
- Množina již redukovaných příkazů `switch`.
- Zásobník aktuálního zanoření cyklů.
- Množina uzlů grafu, které jsou již vygenerovány do cílového kódu.
- Množina uzlů grafu, které jsou cílem nějakého příkazu `goto`.

8.8.4 Strukturování pomocí příkazu `goto`

Jak již bylo popsáno v kapitole s návrhem, v některých případech dojde k tomu, že nelze tělo funkce strukturovat pomocí předdefinovaných tvarů. V takovém případě tedy dojde k průchodu grafem metodou prohledávání do šířky a všem uzlům jsou na konec přidávány konstrukce `IfStmt` nebo `SwitchStmt`, které obsahují příkazy `GotoStmt`. Pak jsou těla všech uzlů připojena na konec těla funkce.

Na závěr strukturování se prochází rozdíl množin uzlů, které jsou cílem `goto` a uzlů, které již byly vygenerovány do cílového kódu. Tedy uzly, které ještě nebyly nikde vygenerovány, se vygenerují na konec těla funkce.

Kapitola 9

Testování

Tato kapitola popisuje, jakým způsobem byla implementace konvertoru testována. Jednotlivé části kódu nezávisle na jiných byly testovány pomocí jednotkových testů, které jsou popsány v sekci 9.1. Pro otestování správného výstupu byly kromě jednotkových testů prováděny také ruční testy kvality. Tyto testy jsou popsány v sekci 9.2. Pro otestování funkčnosti celého zpětného překladače s novým konvertorem byly použity tzv. regresní testy, které jsou popsány v sekci 9.3. Kromě samotné funkčnosti bylo potřebné otestovat také robustnost a rychlost na velkém množství vstupních vzorků. K tomu byly využity tzv. noční testy popisované v sekci 9.4.

9.1 Jednotkové testy

Jednotkové testy slouží k otestování určité části kódu bez závislosti na jiných částech, tedy například pro otestování konkrétní metody třídy na určitou sadu vstupů. Pro psaní jednotkových testů je využit testovací framework *Google Test* [4], který je jednotně využíván ve všech částech zpětného překladače. Každá implementovaná třída obsahuje svou sadu testů, které testují její funkčnost. Pro otestování řešení bylo vytvořeno celkem 325 jednotkových testů, které pokrývají přes 90 % kódu.

```
67  TEST_F(VariablesManagerTests,
68  IdenticalVariableIsReturnedForIdenticalLLVMValue) {
69      auto type = llvm::Type::getInt32Ty(context);
70      auto llvmVal = std::make_unique<llvm::Argument>(type, "var");
71
72      auto var1 = variablesManager->getVarByValue(llvmVal.get());
73      auto var2 = variablesManager->getVarByValue(llvmVal.get());
74
75      ASSERT_TRUE(var1);
76      ASSERT_TRUE(var2);
77      ASSERT_BIR_EQ(var1, var2);
78  }
```

Obrázek 9.1: Příklad jednotkového testu

Ukázku jednotkového testu lze nalézt na obrázku 9.1. Tento test slouží k otestování třídy `VariablesManager`. Konkrétně testuje, že pro stejný LLVM objekt (zde argument funkce) je vrácena stejná proměnná v BIR. Na řádku 67 je začátek definice nového jednotkového testu. Na řádku 68 je jméno tohoto testu. Na řádcích 69 a 70 probíhá vytvoření argumentu funkce v LLVM. Řádky 72 a 73 obsahují provedení akce, kterou test má testovat. V tomto

testu se jedná o dvě volání metody pro získání proměnné v BIR. Na zbývajících řádcích je poslední část jednotkového testu, a to kontrola správnosti výstupů. Nejprve je testováno, že ukazatele obou proměnných nejsou nulové (na řádcích 75 a 76). Poslední řádek slouží ke kontrole, že jsou oba dva ukazatele na proměnnou shodné.

Na obrázku 9.2 je ukázán výstup testovacího frameworku pro procházející test a na obrázku 9.3 je ukázán stav, kdy je ve zdrojovém kódu chyba a test tedy neprojde. U neprocházejícího testu framework vypisuje, která kontrola neprošla. Zde nastal problém na posledním řádku 77, tedy že proměnné `var1` a `var2` nejsou shodné. Framework zde vypisuje, že proměnná `var2` obsahuje ukazatel, který se neshoduje s očekávaným ukazatelem (proměnnou `var1`). Text „`expected ‘var‘, got ‘var‘`“ znamená, že byla očekávána proměnná s názvem `"var"` a ve skutečnosti má proměnná název `"var"`. V daném kontextu je tedy spíše matoucí, jelikož test netestuje shodnost názvů proměnných, ale shodnost instancí objektu `Variable` v BIR.

```
[ RUN      ] VariablesManagerTests.IdenticalVariableIsReturnedForIdenticalLLVMValue
[       OK ] VariablesManagerTests.IdenticalVariableIsReturnedForIdenticalLLVMValue (65 ms)
```

Obrázek 9.2: Ukázka výstupu testovacího frameworku pro jednotkový test, který prošel

```
[ RUN      ] VariablesManagerTests.IdenticalVariableIsReturnedForIdenticalLLVMValue
[...]/new_llvmir2bir_converter/tests/variables_manager_tests.cpp:77: Failure
Value of: var2
  Actual: 16-byte object <30-D5 07-03 00-00 00-00 70-A1 05-03 00-00 00-00>
Expected: var1
Which is: 16-byte object <50-D4 07-03 00-00 00-00 30-53 05-03 00-00 00-00>
 -> expected ‘var‘, got ‘var‘
[  FAILED  ] VariablesManagerTests.IdenticalVariableIsReturnedForIdenticalLLVMValue (0 ms)
```

Obrázek 9.3: Ukázka výstupu testovacího frameworku pro jednotkový test, který neprošel

9.2 Testy kvality

Pro otestování korektního výstupu pro určité konstrukce byly ručně prováděny testy kvality. Tyto testy probíhaly tak, že byl vytvořen testovací soubor v jazyce LLVM IR a byla spuštěna zadní část zpětného překladače pro vygenerování výstupu. Tento výstup byl následně prověřen, zda se v něm nachází požadovaná vysokoúrovňová konstrukce. Některé z těchto testů byly následně přesunuty do jednotkových testů, aby bylo zajištěno, že výstup bude stále stejný.

Na obrázku 9.4 je ukázka testu kvality pro jednoduchou konstrukci `if` s větví `else`. V ukázce výstupu lze vidět, že konverze byla úspěšná a skutečně byla vytvořena konstrukce `if` s větví `else`. Na obrázku 9.5 je ukázka testu kvality pro jednoduchý cyklus `while(true)`, který uvnitř obsahuje podmíněný příkaz `if` obsahující příkaz `break` pro ukončení provádění těla cyklu. Podobně jako u předchozího příkladu lze v ukázce výstupu vidět, že konverze byla úspěšná a byl vytvořen cyklus `while(true)`.

```
declare void @test(i32)
define void @function(i32 %val) {
entry:
    %cond = icmp eq i32 %val, 1
    br i1 %cond, label %iftrue, label %iffalse
iftrue:
    call void @test(i32 1)
    br label %after
iffalse:
    call void @test(i32 2)
    br label %after
after:
    call void @test(i32 3)
    ret void
}
```

(a) Vstup v jazyce LLVM IR

```
void function(int32_t val) {
    // entry
    if (val == 1) {
        // iftrue
        test(1);
    } else {
        // iffalse
        test(2);
    }
    // after
    test(3);
    return;
}
```

(b) Výstup v jazyce C

Obrázek 9.4: Jednoduchý příklad podmíněného příkazu `if` s větví `else`

```
declare void @test(i32)

define void @function(i32 %val) {
entry:
    call void @test(i32 1)
    br label %loop
loop:
    %x = phi i32 [ %y, %loop ], [ 0, %entry ]
    call void @test(i32 %x)
    %y = add i32 %x, 1
    %cond = icmp eq i32 %y, %val
    br i1 %cond, label %after, label %loop
after:
    call void @test(i32 2)
    ret void
}
```

(a) Vstup v jazyce LLVM IR

```
void function(int32_t val) {
    int32_t x;
    int32_t y;
    // entry
    test(1);
    x = 0;
    while (true) {
        // loop
        test(x);
        y = x + 1;
        if (y == val) {
            // break -> after
            break;
        }
        x = y;
        // continue -> loop
    }
    // after
    test(2);
    return;
}
```

(b) Výstup v jazyce C

Obrázek 9.5: Jednoduchý příklad cyklu `while(true)`

9.3 Regresní testy

Regresní testy slouží pro otestování zpětného překladače jako celku a umožňují otestovat výstupní kód vygenerovaný zpětným překladačem. Lze testovat například, zda výstupní kód obsahuje všechny funkce, volání funkcí se správným počtem parametrů nebo textové řetězce. Při implementaci nového konvertoru byly tyto testy využívány především k ověření, zda vygenerovaný kód obsahuje všechny základní bloky funkce a tedy, zda byl kód strukturován tak, že nebyla nějaká část funkce vypuštěna. V rámci této diplomové práce nebyly vytvořeny žádné regresní testy, jelikož již po zpětný překladač existovaly v dostatečném množství.

Dalším možným použitím regresních testů je porovnat výstupy, které dává vstupní program, a které dává program, ke kterému byl získán zdrojový kód, který byl znovu přeložen. Na oba tyto programy se pak spustí sada testovacích vstupů a porovnají se výsledky. Tato

varianta testů je vhodná k otestování, zda byla konverze LLVM IR na BIR korektní, nebo že se všude používají správné proměnné atd. Celkový počet regresních testů je přibližně 5500.

Ukázka regresního testu je na obrázku 9.6 (test je psán v jazyce Python). V ukázce lze vidět vytvoření objektu `TestSettings`, který slouží pro vymezení spuštěných kombinací. Zde je požadováno spouštět testy pro soubor `factorial.c`, na všech architekturách, ve všech formátech souborů, pro všechny podporované překladače, s vypnutými optimalizacemi překladu a s ponechanými ladícími informacemi. Dále je zde jeden testovací případ, který je pojmenován `test_c_produce_same_output_when_run`. Tento test provede spuštění původního programu a programu, který byl výstupem zpětného překladače a porovná jejich výstupy se vstupy 5, 1, 0, 12 a 9. Aby test úspěšně prošel, tak musí být výstupy shodné. Ukázka výstupu testovacího frameworku pro běh tohoto testu je na obrázku 9.7.

```python
from regression_tests import *

class Test(Test):
    settings = TestSettings(
        input='factorial.c',
        arch=ALL,
        format=ALL,
        compiler=ALL,
        compiler_opts='-O0',
        debug_info=True
    )

    def test_c_produce_same_output_when_run(self):
        self.assert_c_produce_same_output_when_run('5')
        self.assert_c_produce_same_output_when_run('1')
        self.assert_c_produce_same_output_when_run('0')
        self.assert_c_produce_same_output_when_run('12')
        self.assert_c_produce_same_output_when_run('9')
```

Obrázek 9.6: Příklad regresního testu pro testování shodného výstupu

```
Running 14 test cases in [...]/factorial for commit 97a9b278...

[...].factorial.Test (factorial.c -a mips -f elf -c clang -C -O0 -g)        [  OK  ]  (14.73s)
[...].factorial.Test (factorial.c -a mips -f elf -c gcc -C -O0 -g)          [  OK  ]  (5.07s)
[...].factorial.Test (factorial.c -a arm -f elf -c clang -C -O0 -g)         [  OK  ]  (22.68s)
[...].factorial.Test (factorial.c -a arm -f elf -c gcc -C -O0 -g)           [  OK  ]  (22.69s)
[...].factorial.Test (factorial.c -a arm -f pe -c gcc -C -O0 -g)            [  OK  ]  (22.69s)
[...].factorial.Test (factorial.c -a pic32 -f elf -c gcc -C -O0 -g)         [  OK  ]  (3.34s)
[...].factorial.Test (factorial.c -a powerpc -f elf -c gcc -C -O0 -g)       [  OK  ]  (6.88s)
[...].factorial.Test (factorial.c -a powerpc -f elf -c clang -C -O0 -g)     [  OK  ]  (7.56s)
[...].factorial.Test (factorial.c -a thumb -f elf -c gcc -C -O0 -g)         [  OK  ]  (7.32s)
[...].factorial.Test (factorial.c -a thumb -f elf -c clang -C -O0 -g)       [  OK  ]  (9.89s)
[...].factorial.Test (factorial.c -a x86 -f elf -c clang -C -O0 -g)         [  OK  ]  (10.06s)
[...].factorial.Test (factorial.c -a x86 -f elf -c gcc -C -O0 -g)           [  OK  ]  (9.35s)
[...].factorial.Test (factorial.c -a x86 -f pe -c gcc -C -O0 -g)            [  OK  ]  (7.31s)
[...].factorial.Test (factorial.c -a x86 -f pe -c clang -C -O0 -g)          [  OK  ]  (9.72s)

SUCCESS (14/14)
```

Obrázek 9.7: Ukázka výstupu testovacího frameworku pro regresní test, který prošel

9.4 Noční testy

Noční testy jsou svým rozsahem největší, co do počtu vzorků. V každém běhu testů se testuje přibližně 90 000 vzorků, u kterých se testuje úspěšnost zpětného překladu a kvalita výsledného kódu. Cílem těchto testů je odhalit chyby, které se mohou vyskytnout ve velmi malém počtu případů, a tím otestovat robustnost řešení. Noční testy jsou spouštěny pravidelně každou noc na serveru, kde lze k běhu testů využít až 48 jader procesoru. Celková doba běhu testů se pohybuje kolem 6 hodin. V případě potřeby lze spustit rovněž experimentální testy, čehož bylo využíváno při tvorbě této práce.

Mezi chyby, které lze použitím nočních testů odhalit, patří pády aplikace (např. při porušení ochrany paměti), syntaktické chyby ve výstupním kódu (nevalidní kód), zvýšení paměťové a časové náročnosti nebo vznik varování při překladu. Výsledky nočních testů lze mezi sebou porovnávat a pozorovat tak zlepšení či zhoršení oproti předchozím běhům. V rámci této diplomové práce nebyly přidány žádné vzorky pro noční testy, jelikož pro otestování funkcionality již existovaly v dostatečném množství.

Výsledky lze prohlížet prostřednictvím webového rozhraní, kde se nachází několik záložek s konkrétními metrikami. První záložka slouží ke shrnutí, co bylo oproti předchozímu běhu zhoršeno, a co naopak zlepšeno. Ukázka tohoto výstupu je na obrázku 9.8, kde je na levé straně vidět především zvětšení počtu syntaktických chyb ve výstupním souboru v jazyce C a zhoršení paměťové náročnosti. Na pravé straně tohoto obrázku je vidět snížení počtu syntaktických chyb ve výstupním souboru v jazyce C na určitých platformách a snížení časové náročnosti.

Significant Worsenings

Success

- **x86/elf - clang/O3 - C syntax result**: from 91.1% to 70.0% (-23.1%)
- **x86/pe - clang/O3 - C syntax result**: from 93.7% to 67.4% (-28.1%)
- **mips/elf (binary) - C syntax result**: from 40.9% to 28.3% (-30.8%)
- **arm/elf (binary) - O0 - C syntax result**: from 31.3% to 13.9% (-55.6%)
- **arm/elf (binary) - O1 - C syntax result**: from 23.7% to 15.8% (-33.3%)
- **arm/elf (binary) - O2 - C syntax result**: from 34.3% to 10.2% (-70.2%)
- **arm/elf (binary) - O3 - C syntax result**: from 28.9% to 15.7% (-45.7%)
- **arm/pe (binary) - C syntax result**: from 36.9% to 23.3% (-36.9%)
- **x86/elf (binary) - O0 - C syntax result**: from 32.1% to 10.1% (-68.5%)
- **x86/elf (binary) - O1 - C syntax result**: from 14.3% to 8.6% (-40.0%)
- **x86/elf (binary) - O2 - C syntax result**: from 17.7% to 10.7% (-39.3%)
- **x86/elf (binary) - O3 - C syntax result**: from 31.3% to 17.2% (-45.0%)
- **x86/pe (binary) - O0 - C syntax result**: from 20.8% to 14.3% (-31.4%)
- **x86/pe (binary) - O1 - C syntax result**: from 20.9% to 17.7% (-15.4%)
- **x86/pe (binary) - O2 - C syntax result**: from 29.7% to 22.4% (-24.4%)
- **x86/pe (binary) - O3 - C syntax result**: from 16.6% to 12.1% (-27.3%)
- **malware (binary) - mips/elf - C syntax result**: from 46.3% to 28.4% (-38.7%)
- **malware (binary) - arm/elf - C syntax result**: from 65.0% to 48.3% (-25.6%)
- **malware (binary) - arm/pe - C syntax result**: from 50.0% to 0.0% (-100.0%)
- **malware (binary) - x86/elf - C syntax result**: from 20.6% to 14.0% (-32.1%)
- **web-service (binary) - mips/elf - C syntax result**: from 51.5% to 40.1% (-22.0%)
- **web-service (binary) - arm/elf - C syntax result**: from 55.0% to 31.8% (-42.0%)
- **web-service (binary) - arm/pe - C syntax result**: from 32.2% to 21.7% (-32.4%)
- **web-service (binary) - x86/pe - C syntax result**: from 22.3% to 18.7% (-16.2%)
- **web-service (binary) - x86/macho - C syntax result**: from 52.9% to 44.1% (-16.7%)
- **web-service (binary) - powerpc/elf - C syntax result**: from 48.0% to 37.1% (-22.6%)

Memory Usage

- **Overall memory usage**: from 198 MB to 232 MB (+17.2%)

Program Exits

- **C Syntax - RC 1 (syntax error)**: from 14560 to 20830 (+43.1%)
- **C Syntax - RC 4**: from 3 to 4 (+33.3%)

Syntax Errors

- **Generated C**: from 95543 to 228989 (+139.7%)

Significant Improvements

Success

- **x86/pe/delphi7 (binary) - O0-g0 - C syntax result**: from 19.0% to 57.0% (+200.0%)
- **x86/pe/delphi7 (binary) - O0-g - C syntax result**: from 19.0% to 57.0% (+200.0%)
- **x86/pe/delphi7 (binary) - O2-g0 - C syntax result**: from 19.0% to 57.0% (+200.0%)
- **x86/pe/delphi7 (binary) - O2-g - C syntax result**: from 18.0% to 57.0% (+216.7%)
- **malware (binary) - x86/pe - C generation result**: from 82.2% to 98.6% (+20.0%)
- **web-service (binary) - arm/macho - C generation result**: from 48.3% to 62.1% (+28.6%)
- **web-service (binary) - arm/macho - C syntax result**: from 44.8% to 55.2% (+23.1%)

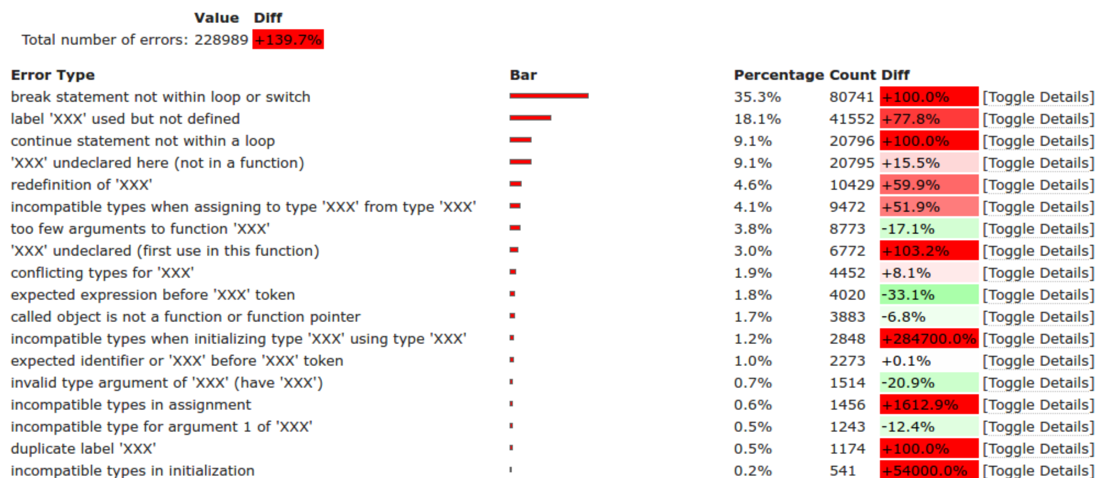
Running Time

- **mips/elf - gcc/O3**: from 01h 17m 41s to 00h 31m 56s (-58.9%)
- **mips/elf - clang/O2**: from 00h 32m 55s to 00h 18m 07s (-45.0%)
- **mips/elf - clang/O3**: from 00h 53m 10s to 00h 19m 09s (-64.0%)
- **mips/ihex - gcc/O3**: from 01h 40m 18s to 00h 45m 28s (-54.7%)
- **mips/ihex - clang/O2**: from 00h 34m 04s to 00h 20m 37s (-39.5%)
- **mips/ihex - clang/O3**: from 00h 50m 03s to 00h 21m 17s (-57.5%)
- **arm/pe - gcc/O0**: from 04h 01m 44s to 02h 05m 22s (-48.1%)
- **arm/pe - gcc/O1**: from 04h 03m 17s to 02h 05m 16s (-48.5%)
- **arm/pe - gcc/O2**: from 04h 04m 36s to 02h 05m 55s (-48.5%)
- **arm/pe - gcc/O3**: from 04h 25m 28s to 02h 07m 42s (-51.9%)
- **arm/macho - clang/O0**: from 04h 18m 38s to 01h 29m 21s (-65.5%)
- **arm/macho - clang/O1**: from 01h 59m 54s to 01h 19m 19s (-33.8%)
- **arm/macho - clang/O2**: from 02h 16m 49s to 01h 19m 59s (-41.5%)
- **arm/macho - clang/O3**: from 02h 44m 18s to 01h 21m 37s (-50.3%)
- **x86/elf - gcc/O3**: from 01h 04m 43s to 00h 34m 47s (-46.3%)
- **x86/elf - clang/O3**: from 03h 48m 07s to 00h 48m 35s (-78.7%)
- **x86/pe - gcc/O3**: from 01h 16m 54s to 00h 48m 15s (-37.3%)
- **x86/pe - clang/O3**: from 02h 13m 04s to 00h 57m 13s (-57.0%)
- **x86/macho - clang/O0**: from 01h 37m 44s to 00h 34m 36s (-64.6%)
- **x86/macho - clang/O3**: from 01h 31m 45s to 00h 34m 21s (-62.6%)
- **powerpc/elf - gcc/O3**: from 01h 25m 43s to 00h 31m 29s (-63.3%)
- **powerpc/elf - clang/O0**: from 00h 31m 57s to 00h 21m 42s (-32.1%)
- **mips/elf (binary)**: from 00h 36m 31s to 00h 11m 25s (-68.7%)
- **arm/elf (binary) - O0**: from 01h 27m 08s to 01h 04m 38s (-25.8%)
- **arm/elf (binary) - O1**: from 01h 41m 29s to 00h 54m 10s (-46.6%)
- **arm/elf (binary) - O2**: from 01h 48m 31s to 01h 02m 25s (-42.5%)
- **arm/elf (binary) - O3**: from 01h 51m 49s to 01h 05m 55s (-41.0%)
- **x86/elf (binary) - O0**: from 06h 42m 08s to 02h 52m 07s (-57.2%)
- **x86/elf (binary) - O1**: from 07h 53m 58s to 02h 54m 08s (-63.3%)

Obrázek 9.8: Ukázka výsledků nočních testů – shrnutí výsledků

Další záložky pak nabízí možnost prohlédnout si podrobnosti o paměťových a časových náročnostech nebo o konkrétních nalezených chybách. Ke všem chybám si lze přímo z webo-

vého rozhraní stáhnout vstupní soubor a zobrazit příkaz, kterým lze zpětný překladač spustit. Tato funkcionalita je velmi nápomocná programátorovi, který tak může jednoduchým způsobem chybu reprodukovat a pracovat na její opravě. Ukázka podrobnějšího zobrazení syntaktických chyb ve výstupních souborech je na obrázku 9.9. Na tomto obrázku lze vidět, že nejčastější chybou je generování příkazu `break` mimo cyklus a mimo příkaz `switch`.



Generated C

	Value	Diff
Total number of errors:	228989	+139.7%

Error Type	Bar	Percentage	Count	Diff	
break statement not within loop or switch		35.3%	80741	+100.0%	[Toggle Details]
label 'XXX' used but not defined		18.1%	41552	+77.8%	[Toggle Details]
continue statement not within a loop		9.1%	20796	+100.0%	[Toggle Details]
'XXX' undeclared here (not in a function)		9.1%	20795	+15.5%	[Toggle Details]
redefinition of 'XXX'		4.6%	10429	+59.9%	[Toggle Details]
incompatible types when assigning to type 'XXX' from type 'XXX'		4.1%	9472	+51.9%	[Toggle Details]
too few arguments to function 'XXX'		3.8%	8773	-17.1%	[Toggle Details]
'XXX' undeclared (first use in this function)		3.0%	6772	+103.2%	[Toggle Details]
conflicting types for 'XXX'		1.9%	4452	+8.1%	[Toggle Details]
expected expression before 'XXX' token		1.8%	4020	-33.1%	[Toggle Details]
called object is not a function or function pointer		1.7%	3883	-6.8%	[Toggle Details]
incompatible types when initializing type 'XXX' using type 'XXX'		1.2%	2848	+284700.0%	[Toggle Details]
expected identifier or 'XXX' before 'XXX' token		1.0%	2273	+0.1%	[Toggle Details]
invalid type argument of 'XXX' (have 'XXX')		0.7%	1514	-20.9%	[Toggle Details]
incompatible types in assignment		0.6%	1456	+1612.9%	[Toggle Details]
incompatible type for argument 1 of 'XXX'		0.5%	1243	-12.4%	[Toggle Details]
duplicate label 'XXX'		0.5%	1174	+100.0%	[Toggle Details]
incompatible types in initialization		0.2%	541	+54000.0%	[Toggle Details]

Obrázek 9.9: Ukázka výsledků nočních testů – podrobnější zobrazení syntaktických chyb ve výstupních souborech

Kapitola 10

Zhodnocení

Tato kapitola zhodnocuje implementovaný konvertor. V následující sekci je popsáno, jak byly splněny požadavky, které byly stanoveny jako cíle práce v sekci 6.2. V sekci 10.2 je uvedeno porovnání výstupů nového konvertoru s původním konvertorem. Sekce 10.3 uvádí problémy, které byly nalezeny v jiných částech zpětného překladače a komplikovaly vývoj nového konvertoru.

10.1 Zhodnocení splnění cílů

Konvertor implementovaný v rámci této diplomové práce je založen na formálním algoritmu strukturování. Návrh tohoto konvertoru je rozdělen do několika menších modulů, které jsou určeny k vykonávání jedné funkce při konverzi. Tyto moduly mezi sebou komunikují podle potřeby. Otestování všech modulů bylo provedeno důkladně pomocí jednotkových testů, které pokrývají přes 90 % zdrojového kódu. Všechny jednotkové testy úspěšně prochází.

Dále bylo provedeno testování pomocí regresních testů. Zde se však nepovedlo zprovoznit 2 regresní testy z celkového počtu 5404. Procentuální úspěšnost regresních testů je $99,96$ %.

Testovaná metrika	Původní konvertor	Nový konvertor	Porovnání
Doba běhu konverze	2 hodiny 28 minut	1 hodina 31 minut	$+38,5$ %
Celková paměťová náročnost	198 MB	232 MB	$-17,2$ %
Počet syntaktických chyb ve výstupu	95 543	228 989	$-139,7$ %
Počet zacyklení zadní části	365	15	$+95,9$ %
Počet pádů zadní části	9	3	$+66,7$ %

Tabulka 10.1: Porovnání výsledků nočních testů při běhu na 89 130 vzorcích

Nejvýznamnější výsledky z nočních testů v porovnání s původním konvertorem jsou uvedeny v tabulce 10.1. V tabulce lze vidět, že rychlost konverze a stabilita programu se zlepšila. Dále však lze vidět, že se zhoršila paměťová náročnost a zhoršil se počet syntaktických chyb ve vygenerovaném výstupním kódu v jazyce C. Mezi nejčastější syntaktické chyby v generovaném výstupu patří:

- Generování příkazu `break` mimo cyklus nebo příkaz `switch` (35,3 % případů).
- Generování příkazu `goto` s cílem, jehož tělo však není vygenerováno (18,1 % případů).
- Generování případu `continue` mimo cyklus (9,1 % případů).

10.2 Srovnání výstupů nového a původního konvertoru

V této sekci jsou uvedeny ukázky výstupů nového konvertoru v porovnání s původním konvertorem. Pro ukázkový příklad, kdy docházelo ke zbytečné duplikaci kódu, který byl uveden v sekci 6.1 na obrázku 6.2, došlo ke zlepšení generovaného výstupu. Výstup nového konvertoru je na obrázku 10.1b. V následující ukázce (na obrázku 10.2) je ukázán vstupní soubor, který byl strukturován shodně novým i původním konvertorem. Veškerý kód je ve funkci `main`, protože již při překladu byla provedena optimalizace.

```
int ch;
scanf("%d", &ch);
switch (ch) {
    case 0:
    case 1:
    case 2:
    case 3:
        printf("d\n");
    case 4:
        printf("e\n");
    case 5:
        printf("f\n");
    case 6:
        printf("g\n");
        break;
    default:
        printf("default\n");
        break;
}

printf("after switch");
return 0;
```

(a) Původní kód

```
int32_t v1;
scanf("%d", &v1);
switch (v1) {
    case 0: {
    }
    case 1: {
    }
    case 2: {
    }
    case 3: {
        puts("d");
    }
    case 4: {
        puts("e");
    }
    default: {
        puts("default");
        break;
    }
}
printf("after switch");
return 0;
```

(b) Výstup zpětného překladače (výňatek kódu)

Obrázek 10.1: Příklad výstupu, kdy došlo ve výstupu ke zlepšení strukturování

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

int main(int argc,char *argv[]) {
    int x = rand();
    int res = factorial(x);
    printf("factorial(%d) = %d\n", x, res);
    return res;
}
```

(a) Původní kód

```
int main(int argc, char ** argv) {
    int32_t v1 = rand();
    int32_t result;
    if (v1 == 0) {
        result = 1;
        printf("factorial(%d) = %d\n", v1, result);
        return result;
    }
    int32_t v2 = v1;
    int32_t v3 = v2;
    result = v3;
    while (v2 != 1) {
        v2--;
        v3 *= v2;
        result = v3;
    }
    printf("factorial(%d) = %d\n", v1, result);
    return result;
}
```

(b) Shodný výstup původního i nového řešení

Obrázek 10.2: Příklad výstupu, kdy jsou výstupy konverze stejné

Nový konvertor však také v mnoha případech dává horší výsledky. Jak již bylo zmíněno v předchozí sekci, často dochází k vygenerování příkazu `break` nebo `continue` mimo cyklus. K tomu dochází typicky při strukturování těla funkce pomocí příkazu `goto`, když algoritmus nezvládne tělo strukturovat jinak. Dojde k vygenerování cílů skoků `goto` na konec těla funkce, ale tyto cíle již obsahují příkazy `break` nebo `continue`, které byly vytvořeny dříve.

V některých případech pak vůbec není cíl skoku `goto` vygenerován. Lze narazit také na případy, kdy nový konvertor sice generuje validní kód, ale tento kód obsahuje zbytečně velké množství skoků `goto` oproti původnímu konvertoru, který obsahuje těchto skoků obsahuje jen minimum. Velmi zjednodušenou ukázku tohoto případu lze vidět na obrázku 10.3. V této ukázce je ponechána pouze vysokoúrovňová struktura, těla jednotlivých základních bloků jsou vynechána.

Další ukázky lze nalézt na CD, které je přiloženo k této diplomové práci. Na těchto ukázkách lze také vidět, že v některých případech nový konvertor generuje kód s menší mírou zanoření, což lze považovat za zlepšení čitelnosti.

```
// ...
if (v4 >= 8) {
    // ...
    return result;
}
// ...
if (v8 != 1) {
    // ...
}
int32_t v12 = v8;
if ((v8 & 2) != 0) {
    // ...
}
// ...
if ((v10 & 2) == 0) {
    if (v9 % 2 != 0) {
        // ...
    }
} else {
    // ...
    if (v9 % 2 != 0) {
        // ...
    }
}
// ...
return result;
```

(a) Výstup původního konvertoru

```
  // ...
  if (v4 < 8) {
      goto lab_0x401de0;
  } else {
      goto lab_0x401d79;
  }
lab_0x401de0:;
  // ...
  if (v7 != 1) { // ...
  } else { // ...
  }
  // ...
  if ((v9 & 2) != 0) { // ...
  } else { // ...
  }
  // ...
  if ((v12 & 2) == 0) {
      goto lab_0x401e28;
  } else {
      goto lab_0x401e40;
  }
lab_0x401d79: // ...
  goto lab_0x401d97_4;
lab_0x401e28:
  if (v8 % 2 == 0) {
      goto lab_0x401d97_4;
  } else {
      goto lab_0x401e31_2;
  }
lab_0x401e40: // ...
  if (v8 % 2 == 0) {
      goto lab_0x401d97_4;
  } else {
      // ...
      goto lab_0x401e31_2;
  }
lab_0x401d97_4: // ...
  return result;
lab_0x401e31_2: // ...
  goto lab_0x401d97_4;
```

(b) Výstup nového konvertoru

Obrázek 10.3: Příklad výstupu, kdy jsou výstupy konverze stejné

10.3 Nalezené problémy ve zpětném překladači

V průběhu implementace nového konvertoru byly odhaleny některé chyby v jiných částech zpětného překladače, které byly překážkou pro implementaci. Tyto chyby byly nahlášeny a odstraněny vývojovým týmem zpětného překladače. Jednalo se o tyto problémy:

- **Nedeterministický výstup optimalizační části zpětného překladače** – ve specifických případech, což způsobovalo, že při testování nového řešení byl na vstupu zadní části různý kód v LLVM IR. Tento problém komplikoval hledání chyby.

- **Nekorektní duplikování výrazu v BIR** – při strukturování je někdy vhodnější některou část kódu duplikovat, protože je to čitelnější než použití `goto`. Ve zadní části již existovala metoda pro duplikování výrazu, ale tato metoda nepracovala korektně, což způsobovalo problém v následných analýzách zadní části. Tyto analýzy totiž předpokládají, že všechny výrazy (kromě proměnných a funkcí) jsou ve výsledném kódu unikátní.

- **Problémy související s generováním příkazu `goto`** – ve více případech bylo tělo cíle `goto` zahozeno dalšími analýzami v zadní části zpětného překladače.

- **Chybějící datový typ při vytváření neinicializovaného pole** – tento nedostatek mohl způsobovat vygenerování pole s nekorektním datovým typem.

- Chyba v porušení izolovanosti jednotlivých jednotkových testů pro testování konstrukcí prostřednictvím LLVM IR, což nemělo dopad na správnou funkčnost testů, avšak z hlediska formální korektnosti byla chyba opravena.

- Několik chyb v aplikační dokumentaci, kdy dokumentace nekorespondovala se zdrojovým kódem.

Kapitola 11

Závěr

Cílem této diplomové práce bylo nastudovat problematiku zpětného inženýrství se zaměřením na zpětný překlad binárního kódu do vyšší formy reprezentace. Dále se seznámit se zpětným překladačem společnosti AVG, s jazykem LLVM IR a s vnitřní reprezentací použitou v zadní části tohoto zpětného překladače (dále jen BIR). Následně měla být navržena metoda strukturování vnitřní reprezentace, která bude převádět skoky na podmíněné příkazy a cykly. Tato metoda měla být implementována a otestována.

V rámci této práce byl navržen a implementován nový konvertor kódu z reprezentace v jazyce LLVM IR do BIR. Nový konvertor byl vyvíjen s důrazem na kvalitní a otestovaný kód. Kód je rozdělen do menších částí, kdy každá vykonává pouze jednu činnost a tato činnost je důkladně otestována pomocí jednotkových testů.

Nový konvertor je založen na formálním algoritmu a tento algoritmus pracuje na principu vyhledávání předdefinovaných vzorů vysokoúrovňových konstrukcí. Algoritmus nejprve vytvoří graf toku řízení pro danou funkci a poté tento graf iterativně redukuje. Pokud algoritmus skončí ve fázi, že již nelze dále graf redukovat, ale ještě v něm zbyly nějaké uzly, tak musí použít strukturování pomocí příkazu `goto`.

Z hlediska strukturování nelze říci, že by byl nový konvertor jasně lepší nebo jasně horší než původní. V některých případech totiž dosahuje lepších výsledků a v některých případech horších výsledků. Dle výsledků z nočních testů je nový konvertor rychlejší, ale ve větším počtu testovacích vstupů generuje syntakticky chybný kód. Dále je nový konvertor stabilnější z hlediska pádů programu nebo zacyklení, ale paměťová náročnost je mírně větší.

Mezi nejčastější problémy generování nevalidního kódu patří generování příkazů `break` nebo `continue` mimo cyklus. Dále pak vygenerování skoku pomocí příkazu `goto`, kdy není vygenerováno cílové návěští a jeho kód. Častým problémem u složitějších vstupů je generování velkého množství příkazů `goto`, i když by v mnohých případech mohly být eliminovány.

Z hlediska budoucího vývoje by bylo potřebné se zaměřit na další rozvoj strukturovacího algoritmu, jelikož v některých případech výstupy nejsou ideální. Ve složitějších případech je generováno velké množství příkazů `goto`, avšak mnohé z nich by mohly být eliminovány. Dále se lze zaměřit na nejčastější syntaktické chyby v generovaných výstupech.

Možným řešením tohoto problému by bylo, že by neprobíhala generace příkazů `break` a `continue` rovnou při strukturování. Místo toho by bylo možné si značit místa, kam by měl být takový příkaz vygenerován. Na konci strukturování dané funkce by se taková místa prošla a u každého by se prověřilo, zda se nalézá uvnitř cyklu nebo mimo něj. Na místech, která se nalézají uvnitř cyklu, by se vygenerovaly původní příkazy `break` nebo `continue`, avšak na místech mimo cyklus by se vygenerovaly příkazy `goto`.

Literatura

[1] CMake. [online].
URL https://cmake.org/

[2] Doxygen. [online].
URL http://www.doxygen.org/

[3] Git. [online].
URL https://git-scm.com/

[4] Google Test. [online].
URL https://github.com/google/googletest/

[5] Projekt Lissom. [online].
URL http://www.fit.vutbr.cz/research/groups/lissom/

[6] AVG Technologies: Retargetable Decompiler. [online].
URL https://retdec.com/

[7] Brumley, D.; Lee, J.; Schwartz, E. J.; aj.: Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX, 2013, ISBN 978-1-931971-03-4, s. 353–368.

[8] Cytron, R.; Ferrante, J.; Rosen, B. K.; aj.: An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, New York, NY, USA: ACM, 1989, ISBN 0-89791-294-2, s. 25–35.

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

[10] International Organization for Standardization: ISO/IEC 14882:2014.

[11] Křoustek, J.: *Retargetable Analysis of Machine Code*. Dizertační práce, 2014.

[12] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 2004.

[13] LLVM Project: The LLVM Compiler Infrastructure Project. [online].
URL http://llvm.org/

[14] LLVM Project: The LLVM Intermediate Representation. [online].
URL http://llvm.org/docs/LangRef.html

[15] Pecinovský, R.: *Návrhové vzory*. Computer Press, 2007, ISBN 978-80-251-1582-4.

[16] Piper, F.; Murphy, S.; Mondschein, P.: *Kryptografie: průvodce pro každého*. Dokořán, 2006, ISBN 80-7363-074-5.

[17] Zemek, P.: Design of a Language for Unified Code Representation. Interní technická zpráva společnosti AVG, 2012.

[18] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. In *The 5th International Conference on Information Security and Assurance*, Communications in Computer and Information Science, Volume 200, Springer Verlag, 2011, ISBN 978-3-642-23140-7, s. 72–86.

Přílohy

Příloha A

Ukázka zpětného překladu

Tato příloha obsahuje ukázku zpětného překladu funkce pro výpočet faktoriálu na platformě Intel x86. Obrázek A.1 znázorňuje vstupní soubor v jazyce C, který byl přeložen a po následném zpětném překladu byl vygenerován výstup, který je znázorněn na obrázku A.2. Dále byl vygenerován výstup z disassembleru (na obrázku A.3), graf volání funkcí (na obrázku A.4) a grafy toku řízení pro jednotlivé funkce (na obrázcích A.5 a A.6).

```
#include <stdio.h>
#include <stdlib.h>
int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
int main(int argc, char **argv) {
    int x = rand();
    int res = factorial(x);
    printf("factorial(%d) = %d\n", x, res);
    return res;
}
```

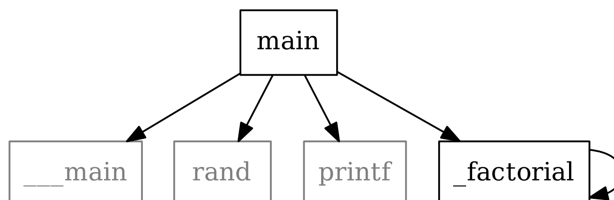
Obrázek A.1: Vstupní program v jazyce C

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
int32_t _factorial(int32_t a1) {
    int32_t result;
    if (a1 != 0)
        result = _factorial(a1 - 1) * a1;
    else
        result = 1;
    return result;
}
int main(int argc, char ** argv) {
    ___main();
    int32_t v1 = rand(); // 0x401595
    int32_t result = _factorial(v1); // 0x4015a5
    printf("factorial(%d) = %d\n", v1, result);
    return result;
}
```

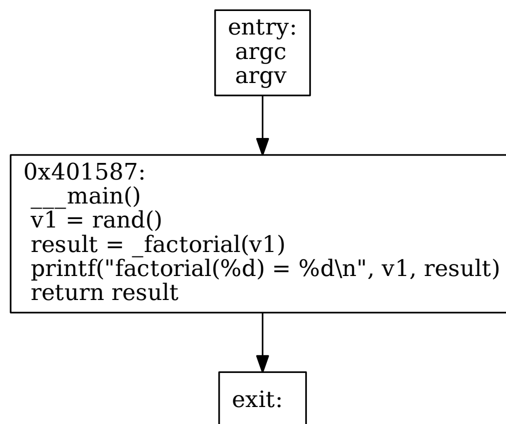
Obrázek A.2: Výstupní program v jazyce C, zjednodušeno

```
;  function: _factorial at 0x401560 -- 0x401586
0x401560:  55                          push ebp
0x401561:  89 e5                       mov ebp, esp
0x401563:  83 ec 18                    sub esp, 0x18
0x401566:  83 7d 08 00                 cmp dword [ ebp + 0x8 ], 0x0
0x40156a:  75 07                       jnz  0x401573 <_factorial+0x13>
0x40156c:  b8 01 00 00 00              mov eax, 0x1
0x401571:  eb 12                       jmp 0x401585 <_factorial+0x25>
0x401573:  8b 45 08                    mov eax, dword [ ebp + 0x8 ]
0x401576:  83 e8 01                    sub eax, 0x1
0x401579:  89 04 24                    mov dword [ esp ], eax
0x40157c:  e8 df ff ff ff              call 0x401560 <_factorial>
0x401581:  0f af 45 08                 imul eax, dword [ ebp + 0x8 ]
0x401585:  c9                          leave
0x401586:  c3                          ret
;  function: main at 0x401587 -- 0x4015cf
0x401587:  55                          push ebp
0x401588:  89 e5                       mov ebp, esp
0x40158a:  83 e4 f0                    and esp, 0xfffffffffffffff0
0x40158d:  83 ec 20                    sub esp, 0x20
0x401590:  e8 db 0d 00 00              call 0x402370 <___main>
0x401595:  e8 2e 5f 00 00              call 0x4074c8 <function_4074c8>
0x40159a:  89 44 24 1c                 mov dword [ esp + 0x1c ], eax
0x40159e:  8b 44 24 1c                 mov eax, dword [ esp + 0x1c ]
0x4015a2:  89 04 24                    mov dword [ esp ], eax
0x4015a5:  e8 b6 ff ff ff              call 0x401560 <_factorial>
0x4015aa:  89 44 24 18                 mov dword [ esp + 0x18 ], eax
0x4015ae:  8b 44 24 18                 mov eax, dword [ esp + 0x18 ]
0x4015b2:  89 44 24 08                 mov dword [ esp + 0x8 ], eax
0x4015b6:  8b 44 24 1c                 mov eax, dword [ esp + 0x1c ]
0x4015ba:  89 44 24 04                 mov dword [ esp + 0x4 ], eax
0x4015be:  c7 04 24 44 90 40 00        mov dword [ esp ], 0x409044 ; "factorial(%d) = %d\n\x00"
0x4015c5:  e8 06 5f 00 00              call 0x4074d0 <function_4074d0>
0x4015ca:  8b 44 24 18                 mov eax, dword [ esp + 0x18 ]
0x4015ce:  c9                          leave
0x4015cf:  c3                          ret
;  section: .rdata
0x409044:  66 61 63 74 6f 72 69 61 6c 28 25 64 29 20 3d 20 |factorial(%d) = |
0x409054:  25 64 0a 00 20                                  |%d..            |
```

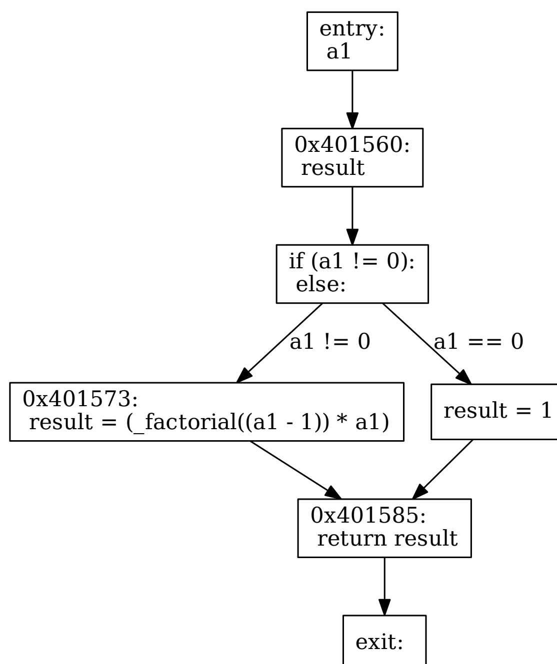
Obrázek A.3: Výstup z disassembleru, zjednodušeno



Obrázek A.4: Graf volání funkcí



Obrázek A.5: Graf toku řízení funkce `main`

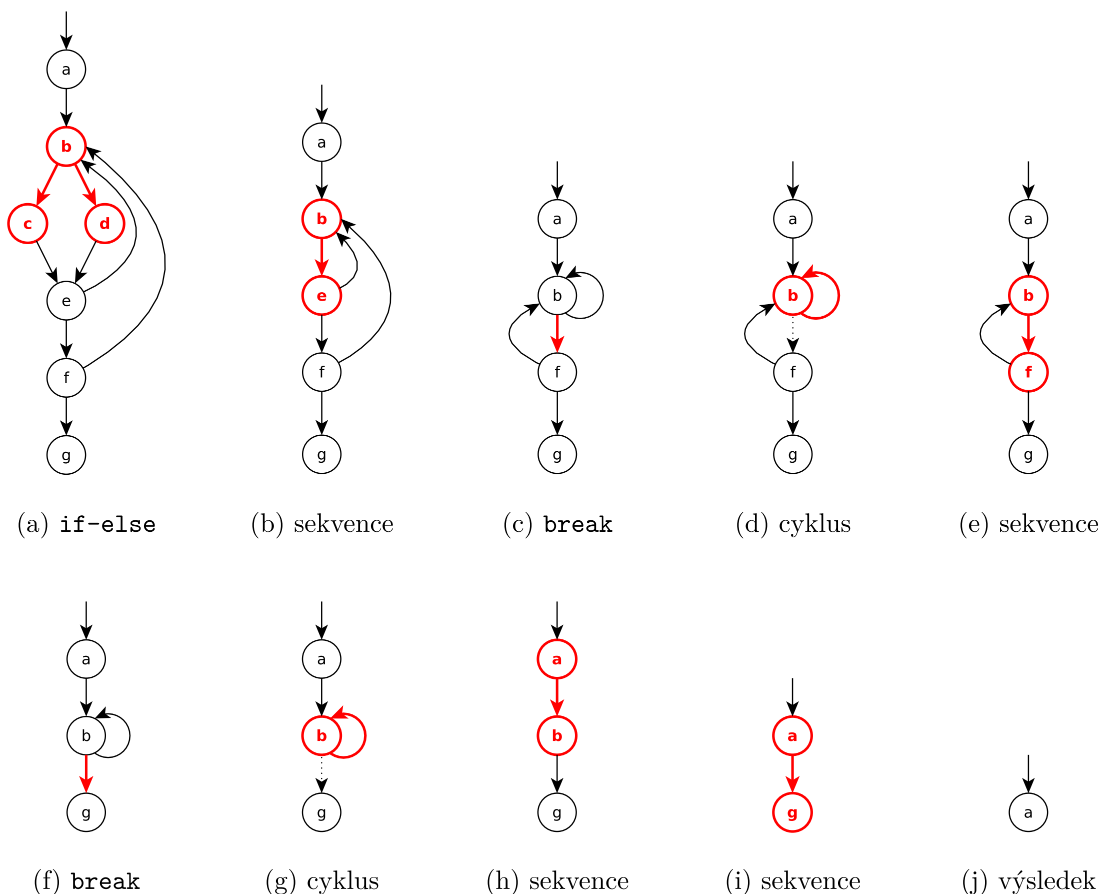


Obrázek A.6: Graf toku řízení funkce `factorial`

Příloha B

Ukázka postupné redukce grafu toku řízení funkce

Na obrázku B.1 je znázorněna postupná redukce grafu toku řízení. V každém kroku je zvýrazněna část grafu, která byla aktuálně detekována pro redukci a v následujícím kroku je již tato část redukována. Tečkovanou čarou je naznačena vazba mezi cyklem a jeho následníkem, která je po redukci cyklu obnovena.



(a) `if-else` (b) sekvence (c) `break` (d) cyklus (e) sekvence

(f) `break` (g) cyklus (h) sekvence (i) sekvence (j) výsledek

Obrázek B.1: Ukázka postupné redukce grafu toku řízení funkce