VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

EXTRAKCE GRAFU TOKU ŘÍZENÍ Z BAJTKÓDU JAVA

EXTRACTION OF CONTROL FLOW GRAPH FROM JAVA BYTECODE

BAKALÁŘSKÁ PRÁCE
BACHELOR’S THESIS

AUTOR PRÁCE PETRA SEČKAŘOVÁ
AUTHOR

VEDOUCÍ PRÁCE Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19241/2016/xsecka02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Sečkařová Petra**
Obor: Informační technologie
Téma: **Extrakce grafu toku řízení z bajtkódu Java**
Extraction of Control Flow Graph from Java Bytecode
Kategorie: Analýza a testování softwaru

Pokyny:

1. Nastudujte programovací jazyk Java. Seznamte se se specifikací Java Virtual Machine. Nastudujte bajtkód jazyka Java.
2. Navrhněte extrakci grafu toku řízení z binárních programů napsaných v jazyce Java. Výstupní formát extrahovaných grafů bude odpovídat požadavkům testovací platformy Testos. Základní bloky budou popsány instrukcemi LLVM IR.
3. Implementujte program pro extrakci grafu toku řízení. Velký důraz klaďte na udržovatelnost zdrojových kódů.
4. Ověřte správnou funkcionalitu programu na sadě binárních programů pokrývajících všechny řídicí konstrukce a všechny základní datové typy Java.

Literatura:

- T. Lindholm, F. Yellin, G. Bracha, A. Buckley. The Java Virtual Machine Specification. Java SE 8 Edition. 2015. url: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html
- *LLVM Language Reference Manual*. Dokument dostupný online: http://llvm.org/docs/LangRef.html
- P. Ammann, J. Offutt. *Introduction to Software Testing*, Cambridge University Press, 2008. ISBN 978-0-511-39330-3.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% 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: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Grafy toku řízení (Control Flow Graph – CFG) slouží jako základ pro mnoho analýz vyhodnocujících kvalitu programu. Takovou analýzou je i testování založené na modelech (model-based testing), které na základě analýzy modelu kódu, např. grafu, generuje testovací případy. Aby bylo možné tuto analýzu provádět co nejobecněji, je vhodné, aby instrukce obsažené v CFG patřily do některé z obecných instrukčních sad. Tato práce se zabývá extrakcí grafů toku řízení z bajtkódu jazyka Java a následným překladem jednotlivých instrukcí bajtkódu uvnitř základních bloků do instrukční sady LLVM IR. Výsledný program dokáže spolehlivě získat grafy toku řízení z programů v jazyce Java zadaných v jakékoli z nejběžnějších forem pro šíření tohoto typu software (.jar archiv, .java nebo .class soubory). Grafy na výstupu jsou navíc koncipovány tak, aby nad nimi bylo možné provádět analýzu za účelem generování jednotkových testů.

Abstract

The most of the analyses evaluating the quality of code are derived from Control Flow Graphs – CFG. Model-based testing as one of them uses paths found in CFG for generation of test cases. To ease use of a general analysis of CFG, there is a need for CFG to contain instructions of some general instruction set. This work deals with extraction of control flow graphs from Java bytecode, followed by a translation of the instructions inside basic blocks into LLVM IR set. The resulting program is able to reliably extract control flow graphs from a Java program, given in any of its casual forms (.jar archive, .java or .class file). In addition to that, the graphs on output are assembled so, that they can be analyzed in order to generate unit tests.

Klíčová slova

bajtkód jazyka Java, graf toku řízení, analýza, LLVM IR

Keywords

Java bytecode, control flow graph, analysis, LLVM IR

Citace

SEČKAŘOVÁ, Petra. *Extrakce grafu toku řízení z bajtkódu Java.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrčka Aleš.

Extrakce grafu toku řízení z bajtkódu Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana doktora Aleše Smrčky. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

....
Petra Sečkařová
17. května 2017

Poděkování

Za ochotnou spolupráci, cenné rady a odborný dohled děkuji vedoucímu práce, doktoru Aleši Smrčkovi.

Obsah

1 Úvod **3**

2 Analýza kódu na základě grafů toku řízení **5**
2.1 Úvod do testování 5
2.2 Co je to graf toku řízení a základní blok 6
2.3 Identifikace základních bloků v kódu 7
2.4 Testování na základě cest a kritérií pokrytí 8

3 Bajtkód jazyka Java a jeho provádění **10**
3.1 Java Virtual Machine 10
3.2 Binární zápis programů v jazyce Java 11
3.3 Instrukce specifické pro jazyk Java 14

4 Extrakce grafů toku řízení z bajtkódu jazyka Java **15**
4.1 Porovnání existujících řešení 15
4.2 Identifikace základních bloků 16
4.3 Další extrahovaná data 18

5 Návrh překladu instrukcí z bajtkódu jazyka Java do LLVM IR **19**
5.1 Dělení instrukcí podle možností překladu 20
5.2 Problematické překlady 21

6 Nástroj pro extrakci grafů toku řízení s instrukcemi v LLVM IR z bajtkódu jazyka Java **24**
6.1 Objektový návrh 25
6.2 Vzorový příklad fungování 27

7 Zajímavé implementační detaily **28**
7.1 Bližší vysvětlení principu extrakce základních bloků 28
7.2 Provádění překladu 28
7.3 Převod a správa lokálních proměnných 29
7.4 Tisk výsledných grafů 31

8 Ověření správnosti transformace **32**
8.1 Porovnání výstupu překladu instrukcí z bajtkódu jazyka Java do LLVM IR 32
8.2 Evaluace 33

9 Závěr **35**

Literatura **36**

Přílohy **37**

A Obsah přiloženého paměťového média **38**

B Manuál **40**
B.1 Použití stávajícího programu 40
B.2 Nové sestavení programu 41

C Překladová tabulka instrukcí z bajtkódu do LLVM IR **42**

D Diagramy tříd **62**

Kapitola 1

Úvod

Tato práce vznikla v rámci projektu Testos (Test Tool Set) [7], jehož hlavním cílem je vytvoření sady nástrojů podporující automatizované testování softwaru. Nástroje v platformě Testos (viz obrázek 1.1) kombinují různé úrovně testování a lze je řadit do kategorií:

- testování založené na požadavcích (Requirement-based testing),
- testování založené na datech (Data-based testing),
- dynamická analýza (Execution-based testing),
- testování grafického uživatelského rozhraní (GUI testing) a
- testování založené na modelech (Model-based testing), do kterého spadá nástroj, jehož návrh a implementace je cílem této práce. V rámci této kategorie mají být z testovaného software extrahovány grafy toku řízení (Control Flow Graph – CFG), které pak budou dále analyzovány za účelem generování testovacích případů pro jednotkové testy.

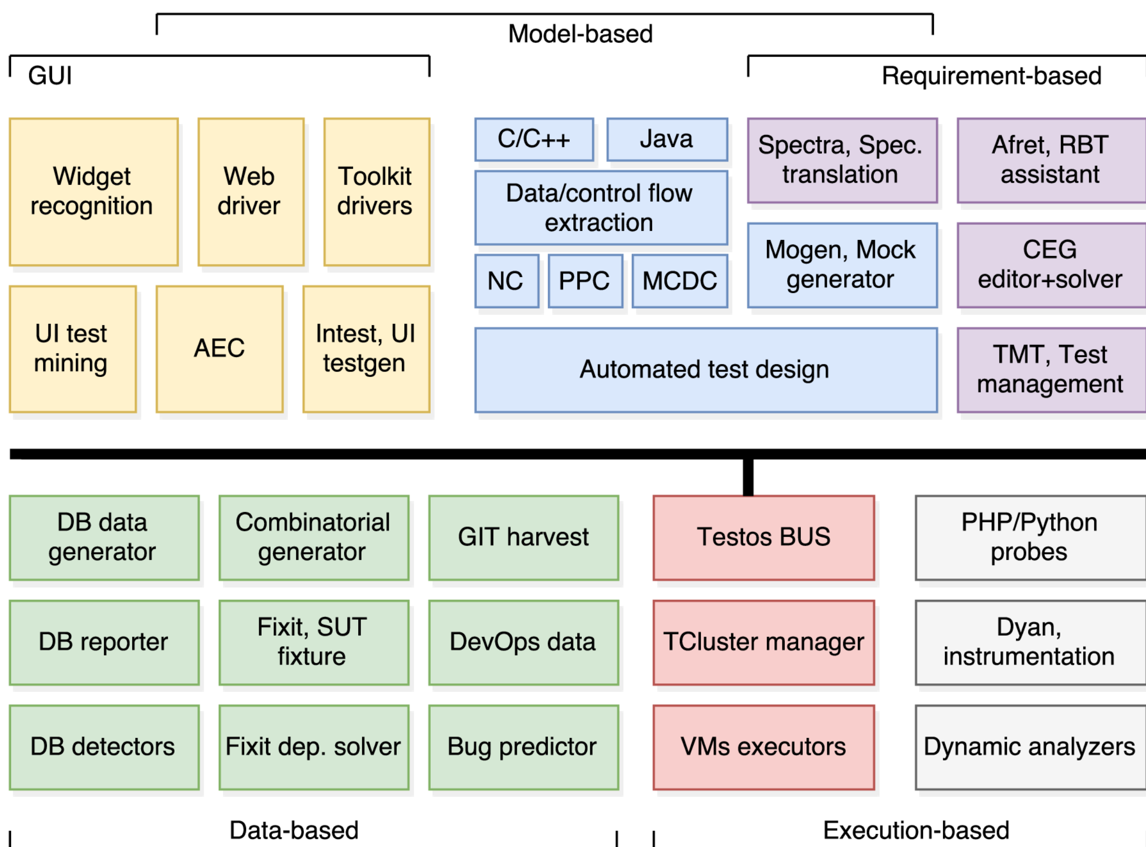
V aktuálním vývoji nástrojů pro testování založené na modelech jsou nástroje pro extrakci grafů toku řízení ze zdrojových kódů jazyka C/C++ [5] a Java (tato bakalářská práce) a nástroj pro hledání požadavků na testy z CFG, tj. cest v CFG [9].

Grafy získané implementovaným nástrojem mají být zpracovávány společně s grafy extrahovanými z ostatních masově používaných programovacích jazyků, a je proto nutné, aby instrukce obsažené ve výsledných grafech patřily do některé z obecných instrukčních sad. Jako taková byla vybrána sada LLVM IR, protože má silnou nástrojovou podporu a díky rostoucí popularitě překladače *clang* také velkou uživatelskou bázi.

Cílem této práce je tedy vytvoření nástroje pro extrakci grafů toku řízeni, jehož výstup bude (i) odpovídat formátu specifikovanému platformou Testos[1] a (ii) obsahovat instrukce sady LLVM IR.

V následující kapitole budou osvětleny principy analýzy kódu na základě grafů toku řízeni. Kapitola 3 poskytuje náhled do struktury a provádění bajtkódu Javy a další kapitoly se poté zabývají použitými principy extrakce grafů a překladu instrukcí, následovány kapitolami 6 a 7, které rozebírají implementaci výsledného programu. Nakonec bude zhodnocena správnost výstupů vytvořeného nástroje.

[1] `https://pajda.fit.vutbr.cz/testos/cfgqe/blob/master/doc/cfglang.md`



Obrázek 1.1: Schéma typů zaměření a propojení nástrojů projektu Testos.

Kapitola 2

Analýza kódu na základě grafů toku řízení

Při revizi kódu by měl každý dobrý tester ověřit nejen vyhovění požadavkům zadání, ale také správné, bezpečné a pokud možno optimální fungování programu za všech okolností. Mnozí z nich proto při tomto nelehkém úkolu používají rozmanité metodické přístupy, kterým je i testování založené na modelech – model-based testing.

Jeho podstatou je vytvoření abstraktního modelu zkoumaného software, nad kterým se provádí analýza s cílem definovat další testovací požadavky. Stejně jako u každého jiného modelování i takto vznikající modely zobrazují jen ty vlastnosti původního programu, které jsou pro daný případ podstatné, a ostatní jsou zanedbány.

Nejběžnější z metod testování založeného na modelech je vytváření grafových reprezentací zkoumaného kódu, které se tester následně snaží pokrýt sadami testů na základě různých kritérií. V této kapitole budou kromě jiného vysvětleny základní principy analýzy kódu na základě *grafů toku řízení*.

2.1 Úvod do testování

Verifikace programů se podle [8] dá obecně rozdělit na čtyři hlavní přístupy.

Statická analýza zkoumá vlastnosti software, aniž by docházelo k jeho spouštění.

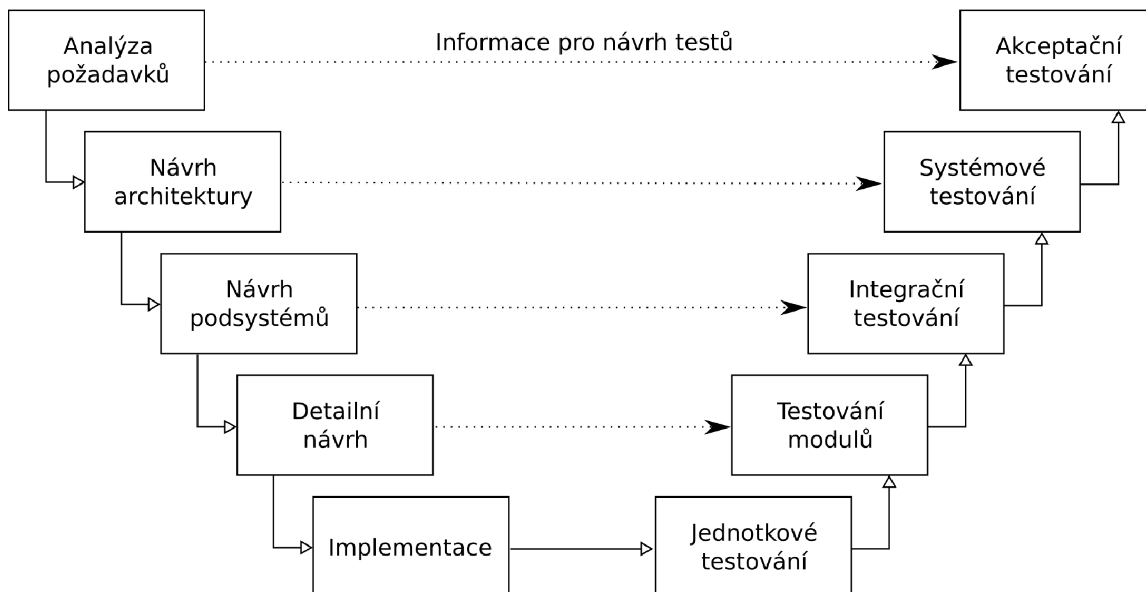
Dynamická analýza se zabývá chováním programu při jeho spouštění.

Formální verifikace ověřuje formálními metodami, že software odpovídá dané specifikaci.

Testování je potom spouštění daného programu za účelem zvýšení jeho kvality. Testovat lze porovnáváním očekávaných a reálných výstupů, podle časových kriterií a mnoha dalšími způsoby.

Testování dále dělíme na několik úrovní. Ty korespondují s fázemi vývoje software jak ukazuje V-model na obrázku 2.1. V tomto modelu explicitně nefiguruje *regresní testování*, což je druh testů vytvářených proto, aby novější verze systému nezaváděly chyby oproti starším verzím. Tento druh testů je vytvářen pro každou úroveň testování z uvedeného modelu.

Zatímco vyšší fáze testování, testy modulů počínaje, vznikají na základě požadavků a předpokladů vytvořených při návrhu systému, jednotkové testy vycházejí ze samotného



Obrázek 2.1: V-model aktivit spojených s vývojem software s vazbami na úrovně testování podle [1].

kódu. Jednotkou jsou zde myšleny metody a datové struktury. Úkolem tohoto druhu testování je prověřit funkcionalitu, správnost návratových hodnot, chování při předání krajních hodnot vstupních parametrů, odhalení případných nežádoucích vedlejších účinků apod.

Provádění jednotkového testu dále stejně jako v [8] dělíme na 4 fáze:

1. nastavení stavu systému,
2. provedení kódu testované jednotky v nachystaném nastavení systému,
3. kontrola výsledků testu a
4. vyčištění vedlejších efektů provedeného testu.

Vytváření požadavků na různá nastavení systému pro tento typ testů může být prováděno na základě nějakého modelu. Graf toku řízení je nejpoužívanější z těchto modelů a testovací požadavky z něj mohou být generovány podle zvoleného kritéria pokrytí kódu, která přibližuje podkapitola 2.4. Motivací pro tento projekt je přispět k automatizaci tohoto procesu.

2.2 Co je to graf toku řízení a základní blok

Formálně definujeme graf toku řízení jako orientovaný graf G podle [1] jako čtveřici (N, N_0, N_f, E) kde:

N je konečná množina všech uzlů grafu G,

N_0 je **neprázdná** množina všech vstupních uzlů grafu G, $N_0 \subseteq N$,

N_f je množina všech koncových uzlů grafu G, $N_f \subseteq N$,

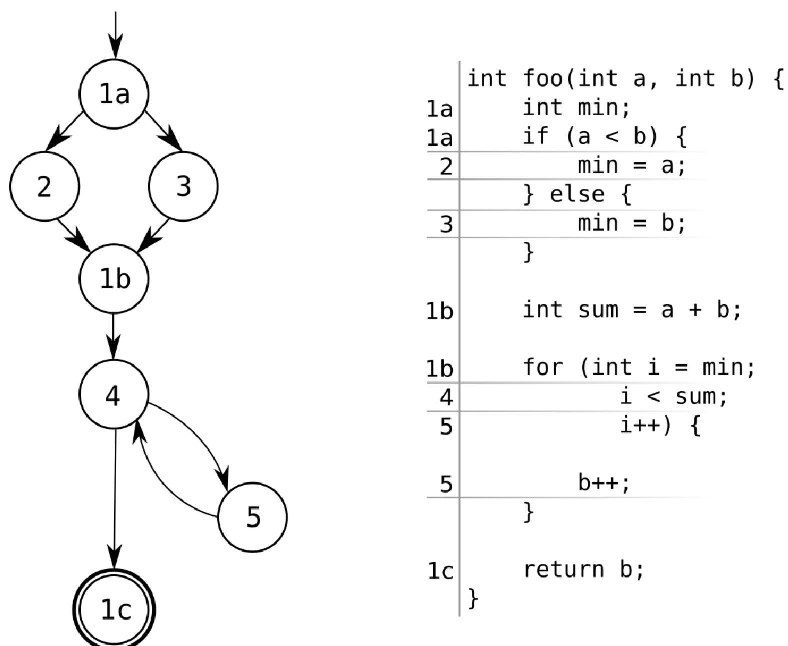
E je množina všech hran, $E \subseteq N \times N$.

Graf toku řízení je potom právě takový graf s orientovanými hranami, kde E obsahuje uspořádané dvojice (n_i, n_j) určující, že uzel n_i je předchůdcem uzlu n_j. Jednotlivými uzly tohoto grafu jsou *základní bloky* kódu. Pod pojmem základní blok rozumíme takovou sekvenci příkazů, z nichž, je-li proveden jeden, musí být provedeny i všechny ostatní.

Způsob zobrazení těchto grafů je možné vidět v levé části obrázku 2.2. Každý uzel je zobrazen jedním kruhem z nichž vstupní uzly doznačíme šipkou směřující do uzlu jako je znázorněno u uzlu *1a*. Koncové uzly jsou zvýrazněny jako *1c* a sousedící uzly jsou propojeny šipkami ve směru orientace hrany mezi nimi.

2.3 Identifikace základních bloků v kódu

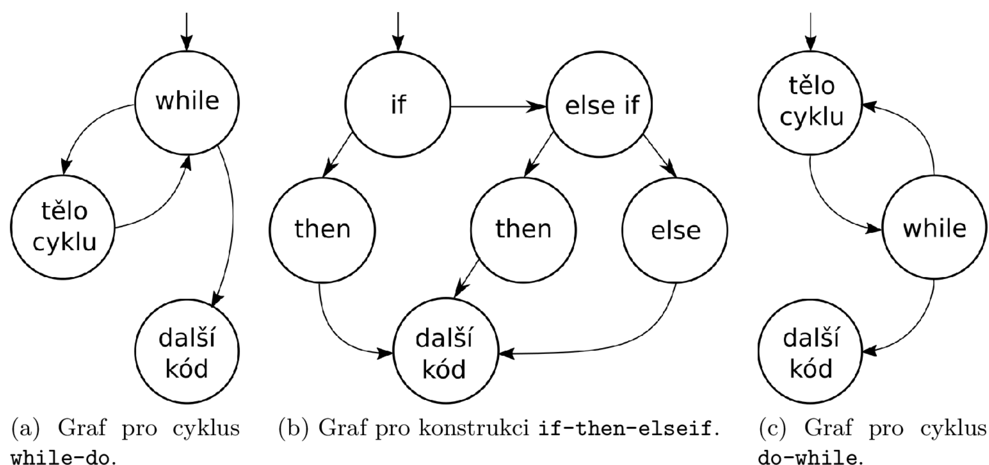
Zásadním úkolem pro tvorbu grafu toku řízeni jako modelu určitého programu je identifikace jednotlivých základních bloků a hran mezi nimi v kódu. Možný výsledek této snahy znázorňuje obrázek 2.2.



Obrázek 2.2: Příklad identifikace základních bloků v konkrétním kódu.

Ukázka rozkresluje kód uvedené funkce `foo` podle kritéria o provedení všech příkazů bloku, pokud je proveden alespoň jeden z nich. Funkce obsahuje jednu konstrukci `if - then - else` a cyklus `for`.

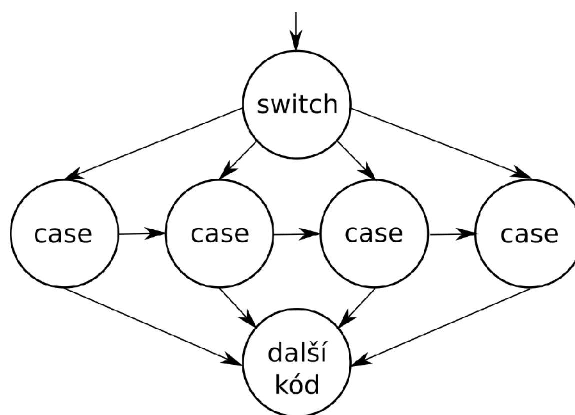
Z uvedeného příkladu identifikace bloků lze odvodit, že za příkazem porovnání je aktuální základní blok přerušen. U cyklu `for` je základní blok přerušen už po iniciačním výrazu, protože následné vyhodnocení podmínky se provede při každém cyklu znovu, a tím se počet provedení tohoto příkazu liší od jak od předchozích, tak od příkazů těla cyklu. Ty jsou zpravidla provedeny $(n-1)$krát, pokud je počet vyhodnocení podmínky roven n. Další příklady grafů toku řízení zobrazující nejběžnější programové konstrukce ukazuje obrázek 2.3



(a) Graf pro cyklus `while-do`.

(b) Graf pro konstrukci `if-then-elseif`.

(c) Graf pro cyklus `do-while`.



(d) Graf pro větvení `switch` – hrany mezi jednotlivými uzly *case* znázorňují možnost serializace těchto případů v některých programovacích jazycích (např. C/C++ nebo Java).

Obrázek 2.3: Ukázky grafů toku řízení pro běžné programové konstrukce.

2.4 Testování na základě cest a kritérií pokrytí

Při testování programu na základě toku řízení je prvním krokem jeho správné rozkreslení do grafové reprezentace. Typicky takto vzniká větší počet grafů, kdy např. každý reprezentuje jednu metodu zkoumaného programu. V těchto grafech jsou následně hledány cesty. Z hlediska testování existuje podle [1] hned několik důležitých definic cest.

Cesta (path) jako taková představuje sekvenci uzlů grafu, které mohou být v daném pořadí procházeny. Formálně definujeme cestu jako $P = [n_i, n_{i+1}, ..., n_k]$, kde $i <= j < k, (n_j, n_{j+1}) \in E$. Platnou cestou pro obrázek 2.2 by byla např. $P_1 = [3, 1b, 4, 5]$ nebo $P_2 = [4, 5, 4, 1c]$. Naopak nevalidní by byla cesta $P_3 = [3, 5]$, protože z uzlu *3* žádná hrana do uzlu *5* nevede. Její délka je dána počtem změn uzlu, a tedy délka cesty P_3 je 1, pro P_2 je to 3 atd.

Jednoduchá cesta obsahuje každý z uzlů grafu maximálně jednou, kromě případu, že cesta začíná a končí ve stejném uzlu. Taková cesta tedy neobsahuje žádné vnitřní cykly, ale sama být cyklem může. Platnou jednoduchou cestou by byla zmiňovaná

cesta P_1, ale P_2 už ne. Cesta $P_4 = [4, 5, 4]$, která je podcestou cesty P_2 však kritéria jednoduché cesty splňuje.

Primární cesta (prime path, v češtině kvůli neustálenému výrazu někdy též hlavní cesta) je jednoduchá cesta s maximální délkou, tzn. taková, která není podcestou žádné jiné jednoduché cesty. Graf z obrázku 2.2 má pět primárních cest, a to již zmiňovanou P_4 a cesty $P_5 = [1a, 3, 1b, 4, 1c]$, $P_6 = [1a, 3, 1b, 4, 5]$, $P_7 = [1a, 2, 1b, 4, 1c]$ a $P_8 = [1a, 2, 1b, 4, 5]$.

Testovací cesta (test path) je taková cesta, která začíná v některém vstupním uzlu grafu a končí v uzlu koncovém. Ten by měl být dosažitelný z každého uzlu grafu, tzn. že musí existovat platná cesta z začínající v libovolném uzlu grafu a končí ve vybraném koncovém uzlu. Testovací cesta může byt potenciálně nekonečná, proto je typicky v praxi vybírána tak, aby byla co nejkratší. Platnou testovací cestou by mohly být cesty P_5 a P_7, ale také $P_9 = [1a, 2, 1b, 4, 5, 4, 5, 4, 1c]$.

Testovací případy jsou mapovány na jednotlivé testovací cesty tak, že jedna testovací cesta je pokryta alespoň jedním, ale často hned několika testy. To, které testovací cesty budou zvoleny jako předlohy k tvorbě testů, závisí na zvoleném kritériu pokrytí, z nichž nejjednodušší jsou pokrytí:

Uzlů (Node Coverage – NC) – navštívení každého dosažitelného uzlu grafu.

Hran (Edge Coverage – EC) – využití každé dosažitelné cesty o délce 1 nebo 0.

Dvojic hran (Edge Pair Coverage – EPC) – využití každé dosažitelné cesty do délky 2 včetně.

Primárních cest (Prime Path Coverage – PPC) – využití všech primárních cest grafu.

Podmínek (Condition Coverage – CC) – každá boolovská proměnná vstupující do podmínky musí musí být alespoň jednou vyhodnocena jako `true` a alespoň jednou jako `false`

Rozhodnutí (Decision Coverage – DC) – **výsledek** libovolné podmínky v kódu musí být alespoň jednou vyhodnocen jako `true` a alespoň jednou jako `false`

Modifiované pokrytí podmínek a rozhodnutí (Modified condition/decision coverage – MCDC) – pokrytí podmínek s tím, že změna elementární hodnoty musí změnit i celkový výsledek rozhodnutí. To znamená že mezi jednotlivými testovacími případy měníme atomickou hodnotu pouze v případě, že to změní výsledek rozhodnutí. Toto kritérium je standardem DO-178C vyžadováno pro praktické nasazení vybraných kritických systémů, viz [2].

Po zvolení vhodného kritéria pokrytí pro daný program je zpravidla hledána minimální kolekce testovacích cest, které dané kritérium dokáží splnit a ze kterých budou později vytvářeny testovací případy.

Kapitola 3

Bajtkód jazyka Java a jeho provádění

Jak se píše v [6], programovací jazyk Java je objektově orientovaný jazyk se syntaxí podobnou C nebo C++. Díky způsobu provádění programů, které jsou v něm napsány, je dobře přenositelný na různé typy zařízení, což je jeden z důvodů, proč je Java jedním z nejpoužívanějších a podle TIOBE indexu[1] také nejpopulárnějších programovacích jazyků na světě.

Při vykonávání programu psaného v jazyce Java se uplatňují dva přístupy:

1. **kompilace** – Každý .java soubor je nejdříve zkompilován do binární podoby. Ta ovšem na rozdíl od binárních souborů vzniklých například kompilací kódu v C nebo C++ není závislá na platformě, která ji vytvořila.

2. **interpretace** – Samotné provádění programu obstarává Java Virtual Machine – JVM, kterému se blíže věnuje podkapitola 3.1.

Právě tento přístup umožňuje, aby binární forma programů jazyka Java nebyla závislá na platformě, protože specifikace instrukcí pro JVM je napříč platformami jednotná. Je proto běžné distribuovat software napsaný v jazyce Java jako archiv binárních souborů, který lze spustit na libovolném operačním systému podporujícím tento jazyk.

3.1 Java Virtual Machine

Java Virtual Machine je interpret bajtkódu jazyka Java. V jeho specifikaci, viz [6], jsou uvedeny detailní popisy jeho funkcionality a ostatních úzce souvisejících faktů (kompilace kódu jazyka Java, instrukční sada mezikódu, atp.).

Jako vstup jsou JVM předkládány `class` soubory – soubory s mezikódem jazyka Java, které budou blíže popsány v následující podkapitole. K provádění instrukcí je v JVM pro každé případné vlákno programu použit jeden zásobník s operandy, který je používán i při tvorbě zásobníkových rámců jednotlivých metod a pro předávání parametrů. Dále má JVM pro libovolný počet vláken jednu společnou hromadu (heap), kam jsou ukládány jednotlivé instance tříd nebo polí. Tento prostor je uvolňován automaticky systémem známým jako *garbage collector*.

[1] `https://www.tiobe.com/tiobe-index/`

Všechna konstantní data, jako jsou názvy tříd a metod, číselné konstanty nebo znakové řetězce, jsou uchovávána ve struktuře nazývané *constant pool*. Ke každé třídě naleží jeden a jeho obsah je vypsán na začátku příslušného `class` souboru.

Dalším údajem, který je v těchto souborech obsažen, jsou záznamy o lokálních proměnných jednotlivých metod. V nich je uvedeno nejen jméno a typ každé proměnné, která bude v dané metodě použita, ale také její rozsah platnosti a číslo *slotu*, což je index do zásobníkového rámce dané metody, kam bude daná proměnná umístěna.

Veškeré ostatní údaje potřebné pro vykonávání programu dohledává JVM dynamicky, nebo je někdy úplně zanedbá. Takový případ nastává například u kompletních definic typů používaných objektů (pole objektů, která jsou v kódu použita, jsou uvedena v *constant poolu*, ostatní jsou ignorována).

3.2 Binární zápis programů v jazyce Java

Kromě údajů pro JVM uvedených v předchozí podkapitole obsahují `class` soubory také samotné instrukce programu. Specifikace jednotlivých instrukcí bajtkódu jazyka Java se nachází jak v [6], tak v online dokumentech korporace Oracle[2].

Stejně jako samotný jazyk Java pracuje bajtkód prostřednictvím JVM se dvěma druhy datových typů – primitivními (celá a desetinná čísla, znaky apod.) a referencemi (objekty a pole). Tento fakt se projevuje i na instrukcích této sady, kde se často vyskytuje stejný operační kód pro několik různých typů. Datový typ operandů, s nimiž má instrukce pracovat, je dán prvním písmenem operačního kódu, jejichž význam je uveden v tabulce 3.1.

Označení	Datový typ	Označení	Datový typ
a	reference	f	float
b	byte	i	integer
c	char	l	long
d	double	s	short

Tabulka 3.1: Označení datového typu operandů v instrukcích bajtkódu jazyka Java.

Binární forma bajtkódu obsažená v `class` souborech je pro člověka nečitelná, ale vzhledem k tomu, že programy i knihovny jazyka Java jsou šířeny právě v této formě, existuje mnoho nástrojů, které dokáží nejen vytisknout textovou reprezentaci obsahu těchto binárních souborů, ale dokonce dekompilovat tuto formu zpět do jazyka Java. Tato funkcionalita je často zabudována do vývojových prostředí pro tento jazyk, aby mohla být využita např. při krokování kódu.

Jedním z nástrojů, které dokáží vytisknout textovou formu bajtkódu z `class` souborů je nástroj *javap*, disassembler `class` souborů od společnosti Oracle. Rozsah tištěných informací lze regulovat přepínači jejichž význam i forma jsou uvedeny v oficiální dokumentaci[3]. Obrázek 3.1 ukazuje výstup tohoto nástroje pro následující kód:

```
public class BpTest {
    public static void main(String args[]) {
        int i = foo(1,3);
    }
```

[2] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

[3] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html

```
    public static int foo(int a, int b) {
        int c;
        if (a > 5)
                c = a;
        else
                c = b;
        return c;
    }
}
```

Při tisku textové formy bajtkódu zkompilovaného z uvedené třídy byly použity tři přepínače:

-c pro vytištění sekce *Code:* s jednotlivými instrukcemi metod,

-l zapne tisk tabulek lokálních proměnných a mapování řádků na instrukce (sekce *LocalVariableTable:* a *LineNumberTable:*) a

-s doplňující signatury datových typů (*descriptor:*).

Signatura je zkrácený zápis datového typu používaný např. pro označení očekávaných parametrů volané metody. Tento případ se vyskytuje na obrázku 3.1 v metodě `main` u instrukce `invokestatic`, kde je volána metoda `foo(II)I`. Tento zápis znamená, že metoda očekává jako parametry dvě proměnné typu `integer` – `(II)` – a stejného datového typu je i návratová hodnota, jak je uvedeno za závorkou s parametry. Signatura může začínat libovolným počtem znaků `[`, které značí, že daný typ je polem typu, jehož signatura následuje za tímto znakem, a tedy například `[[I` by bylo dvojrozměrné pole celých čísel. V tomto formátu jsou vyjadřovány i třídy objektů. Datový typ objektu začíná písmenem L a končí středníkem a mezi těmito znaky je uvedena přesná specifikace třídy, např. `java/lang/String` u parametru metody `main` z obrázku 3.1. Kompletní přehled typů v signaturách uvádí tabulka 3.2.

Označení	Datový typ	Označení	Datový typ
Lnázev/třídy;	objekt	F	float
B	byte	I	integer
C	char	J	long
D	double	S	short

Tabulka 3.2: Označení datového typu operandů v instrukcích bajtkódu jazyka Java.

Nastavení dekompilace z obrázku 3.1 poskytuje veškeré běžně potřebné informace. U každé instrukce je uvedeno číslo znamenající offset její adresy oproti adrese první instrukce. Lze tedy dokonce odvodit délky jednotlivých instrukčních kódů v binární formě. U instrukcí pracujících s *constant pool* je za indexem adresujícím konkrétní konstantní hodnotu uveden komentář, sdělující jaká konstanta se na daném indexu nachází. Hned v konstruktoru na obrázku 3.1 můžeme u druhé instrukce vidět odkaz na metodu `init` třídy `java/lang/Object`.

Mapování instrukcí na řádky původního kódu je uvedeno v sekci *LineNumberTable* ve formátu

$$line\ n:\ offset_instrukce$$

kde n je číslo řádku v původním kódu a $offset_instrukce$ udává první instrukci na daném řádku. Například pro metodu `main` z obrázku 3.1 je vidět, že instrukce s offsety 0 až 5 jsou v kódu na řádku 3, následovány řádkem 4 s jedinou instrukcí – `return`.

```
Compiled from "BpTest.java"
public class BpTest {
  public BpTest();
    descriptor: ()V
    Code:
       0: aload_0
       1: invokespecial #1     // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 1: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   LBpTest;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    Code:
       0: iconst_1
       1: iconst_3
       2: invokestatic  #2     // Method foo:(II)I
       5: istore_1
       6: return
    LineNumberTable:
      line 3: 0
      line 4: 6
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       7     0  args   [Ljava/lang/String;
          6       1     1     i   I

  public static int foo(int, int);
    descriptor: (II)I
    Code:
       0: iload_0
       1: iconst_5
       2: if_icmple     10
       5: iload_0
       6: istore_2
       7: goto          12
      10: iload_1
      11: istore_2
      12: iload_2
      13: ireturn
    LineNumberTable:
      line 8: 0
      line 9: 5
      line 11: 10
      line 12: 12
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          7       3     2     c   I
          0      14     0     a   I
          0      14     1     b   I
         12       2     2     c   I
}
```

Obrázek 3.1: Ukázka výstupu nástroje `javap` s přepínači `-c -l -s`.

V tabulce lokálních proměnných jsou vypsány všechna pojmenovaná místa v paměti, se kterými daná metoda pracuje. U každé proměnné je uvedeno kromě začátku a délky její platnosti v jednotkách offsetu instrukcí, jména a datového typu ve formě signatury také *slot*, v němž je uložena. Jak již bylo zmíněno v podkapitole 3.1, právě číslo slotu slouží pro adresaci lokálních proměnných v zásobníkovém rámci dané metody. Použití tohoto údaje lze sledovat typicky v instrukcích nahrávajících obsah lokální proměnné na zásobník – `Tload_S`

– nebo ukládajících hodnotu z vrcholu zásobníku do lokální proměnné – `Tstore_S`. Písmeno `T` v uvedených příkladech značí datový typ podle tabulky 3.1 a `S` udává číslo slotu[4].

3.3 Instrukce specifické pro jazyk Java

Vzhledem k tomu, že je bajtkód jazyka Java prováděn vlastním, specifickým interpretem, obsahuje jeho instrukční sada mnoho specifických instrukcí. V prvé řadě se jedná o instrukce pro práci se zásobníkem, jakými jsou `pop` nebo `swap`[5].

Další zajímavou skupinou jsou instrukce. které počítají s tím, že JVM uchovává některé metadata interně, nebo je dokáže zjistit dynamicky, bez toho, aby byla uvedena v bajtkódu. Do této skupiny patří `arraylength` – zjistí aktuální délku pole – nebo `instanceof` – odpoví boolovskou hodnotou na dotaz, jestli objekt patří do dané třídy.

JVM také rozumí několika instrukcím, které jsou v obecných sadách často složeny ze sekvence jednodušších. Dobrým příkladem jsou instrukce `Taload`, která dokáže načíst na zásobník hodnotu z určitého indexu pole hodnot typu `T`, nebo `Tastore`, která ji do pole ze zásobníku zase vloží. Podobně jsou zde také instrukce `getfield` a `putfield`, které přistupují k polím objektů podle jména a signatury, nebo `getstatic` a `putstatic`, které pracují se statickými proměnnými[6] předané třídy.

Specifické jsou také instrukce volající nějakou metodu, jejichž název začíná slovem `invoke`. Tyto instrukce lze rozdělit do dvou skupin.

Volané nad určitým objektem:

- `invokeinterface` – volání metody deklarované implementovaným rozhraním,
- `invokespecial` – volání metod otcovské třídy, metod označených jako `private`, nebo konstruktorů,
- `invokevirtual` – nejpoužívanější volání metod, vyhledá nejaktuálnější implementaci pro daný objekt,

Nezávislé na konkrétním objektu:

- `invokedynamic` – dynamické volání metod prováděný pomocí JVM objektu, nebo `MethodHandler`
- `invokestatic` – volání statických metod.

Za zmínku stojí také instrukce pro iniciaci výjimky – `athrow`, kde `a` napovídá, že na vrchol zásobníku bude vložena reference na objekt výjimky, nebo instrukce `monitorenter` a `monitorexit`, které slouží pro synchronizaci vícevláknových programů v jazyce Java.

[4]Číslo slotu nemusí být vždy odděleno znakem '_', ten se uvádí pouze do hodnoty 3, vyšší čísla jsou oddělována mezerou.

[5]Tato instrukce prohodí dvě svrchní hodnoty na zásobníku.

[6]Statické proměnné nahrazují v jazyce Java globální proměnné – jedná se o pole třídy, která nenáleží k jednotlivým objektům, ale samotné definici třídy, a jsou tedy přístupné prakticky odkudkoli.

Kapitola 4

Extrakce grafů toku řízení z bajtkódu jazyka Java

Jak již bylo řečeno dříve, úkolem této práce je návrh a vývoj nástroje pro extrakci univerzálně zpracovatelných grafů toku řízení z bajtkódu jazyka Java. Binární forma programů je pro tento úkol velmi vhodnou předlohou nejen proto, že v ní lze snadno sledovat instrukce identifikující začátky a konce základních bloků, ale také proto, že je v této formě šířena většina programů v jazyce Java.

Ideální představou tedy je předložit nástroji zdrojový soubor, `class` soubor nebo `jar` archiv a získat na výstupu kolekci všech extrahovaných CFG. Tento výstup má sloužit nově vznikající platformě Testos, kde z něj postupně budou čerpat další nástroje mající za úkol automatizovat jednotkové testování předloženého programu.

Platforma Testos má být schopna automatického testování software psaného nejen v jazyce Java, ale také v ostatních masově používaných programovacích jazycích. Aby bylo možné testovací požadavky z výsledných grafů generovat univerzálním způsobem, je potřeba, aby grafy na výstupu tohoto nástroje obsahovaly místo instrukcí bajtkódu instrukce některé obecné sady. Tomuto problému se dále podrobněji věnuje kapitola 5, avšak tento fakt je důležitý už při návrhu extrakce dat z bajtkódu, protože k následnému překladu je potřeba větší množství informací, než k pouhému rozdělení kódu do základních bloků.

4.1 Porovnání existujících řešení

Extrakci grafů toku řízení z programů jazyka Java umí provádět hned několik existujících nástrojů, které jsou však často jednoúčelově zaměřené na něco jiného, případně jsou neudržované, nebo je jejich výstup nepoužitelný pro další univerzální analýzu s cílem generovat testovací požadavky.

Mezi takovéto nástroje patří **ConFlEx**[1], popsaný v [3], který využívá knihovnu Sawja-/Javalib[2] k načtení `class` souborů do struktur v jazyce OCaml a transformuje bajtkód do jiného ne zásobníkového mezikódu. Tento nástroj již není udržovaný, a navíc využití jazyka OCaml je pro zapojení do rozsáhlejšího projektu náročnější, než samotná extrakce grafů z bajtkódu.

[1] `http://www.csc.kth.se/~pedrodcg/conflex/`
[2] `http://sawja.inria.fr/`

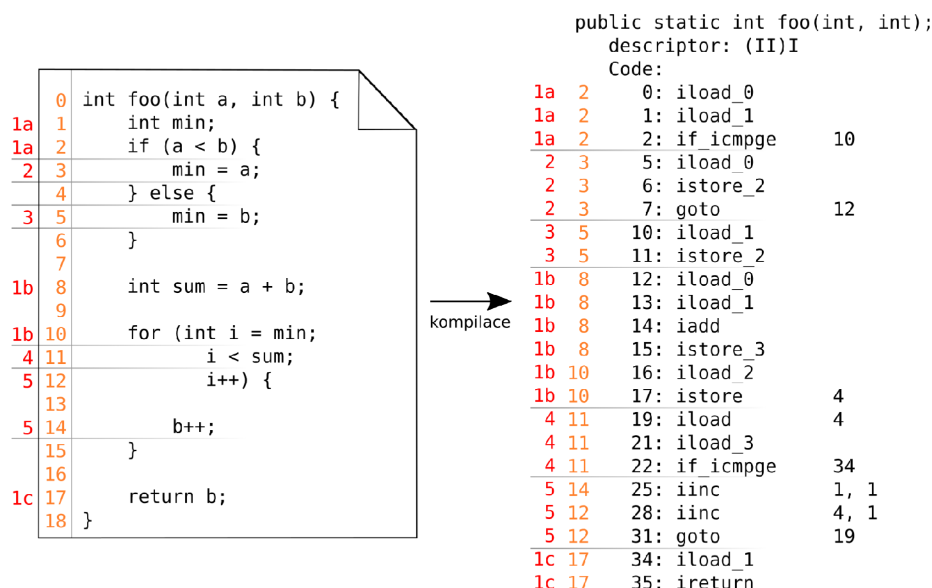
Dalším zajímavým nástrojem je **Soot**[3]. Jedná se o framework zaměřený na optimalizaci programů v jazyce Java, který ke svému fungování používá grafy toku řízení, které jsou ovšem upravené pro daný účel. Jejich modifikace a překlad zahrnutých instrukcí do požadované formy by byl opět zbytečně náročný.

V praxi se k nahlížení do grafů toku řízení používají také pluginy do vývojových prostředí, jakými jsou **Dr. Garbage Tools**[4] pro Eclipse. Jejich využití pro vytvářený nástroj by s sebou ovšem podobně jako u ostatních zmiňovaných neslo více problémů než užitku.

Takto by se jistě dalo pokračovat ještě dlouho. Obecně je nejčastějším problémem existujících řešení kromě mnohdy jednoúčelového zaměření a případné neudržovanosti především to, že výsledek není zpracovatelný univerzálně s výsledky podobných nástrojů pro jiné programovací jazyky.

4.2 Identifikace základních bloků

Pro vysvětlení způsobu identifikace základních bloků v sekvencích instrukcí bajtkódu využijeme metodu `foo` jejíž kód mapovaný na bajtkód ukazuje obrázek 4.1. Vzhledem k tomu, že se jedná o stejnou metodu, jaká je na obrázku 2.2, je zde zakresleno i rozdělení do základních bloků.



Obrázek 4.1: Kód metody a její překlad do bajtkódu jazyka Java s mapováním řádků (oranžově) a rozdělením do základních bloků (červeně) podle obrázku 2.2.

Při pozornějším sledování vyznačených hranic jednotlivých bloků si lze všimnout, že se vždy pohybují v okolí skokových instrukcí, nebo cílů těchto skoků. Není nutno zdůrazňovat, že podmíněné skoky značí vždy rozvětvení kódu, a tím pádem také konec aktuálního bloku. Vzhledem k tomu, že jazyk Java nepodporuje příkaz `goto` přímo v kódu[5], nemůže ani nepodmíněný skok znamenat setrvání v jedné linii programu. Cílové instrukce těchto skoků

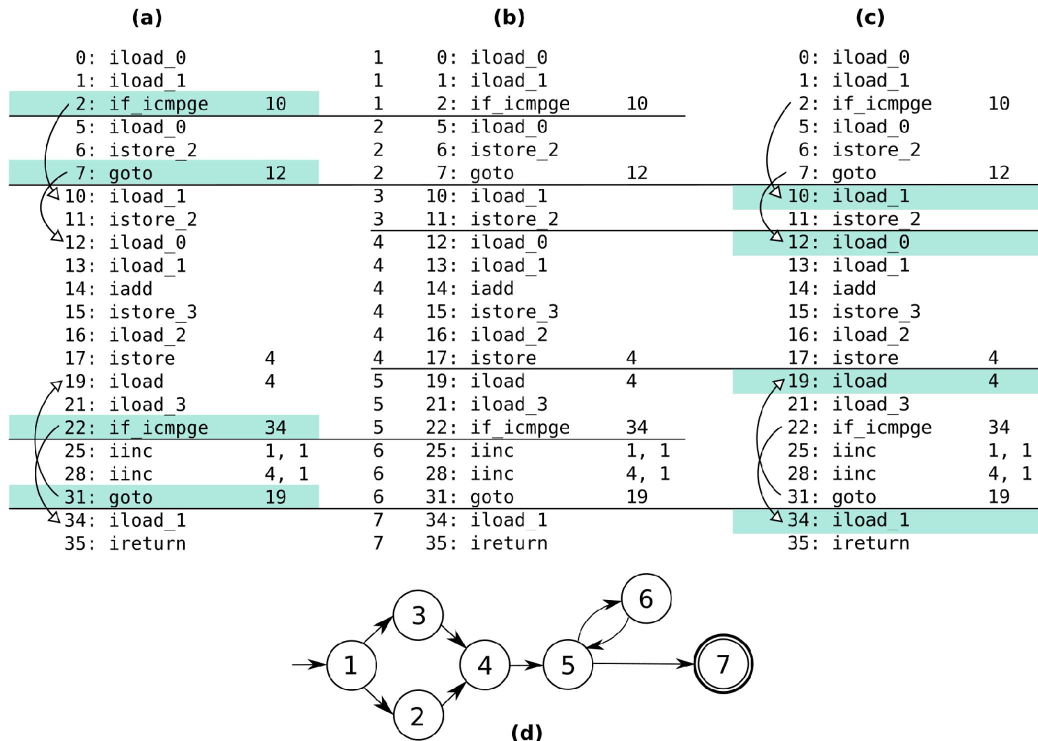
[3]`https://github.com/Sable/soot`

[4]`http://drgarbagetools.sourceforge.net/`

[5]Slovo `goto` je však označeno jako klíčové, protože jeho použití jako identifikátoru by dokázalo způsobit JVM mnoho problémů.

pak označují začátek nového bloku, protože jsou často přístupné jak sekvenčně, tak tímto skokem, a tím pádem počet provedení těchto instrukcí, cílem skoku počínaje, může být různý od okolních. Příkladem toho, kdy cíl skokové instrukce značí začátek nového základního bloku, který je přístupný i sekvenčně, je instrukce s offsetem 19 v obrázku 4.1, která je první instrukcí podmínky cyklu `for`.

Je nutno podotknout, že základní bloky vzniklé oddělením sekvencí instrukcí podle skoků, mohou být pouze částmi bloku, který by dokázal identifikovat člověk na základě pravidla o provedení všech instrukcí z podkapitoly 2.2, avšak výsledné grafy budou bijektivní, jak si lze všimnout porovnáním výsledných grafů na obrázcích 2.2 a 4.2.



Obrázek 4.2: Ukázka metody automatické identifikace základních bloků a jejich složení do grafu založené na identifikaci skokových instrukcí a cílů těchto skoků. V části (a) jsou vyznačeny všechny skokové instrukce daného bajtkódu, za kterými následuje předěl základních bloků. Část (c) zvýrazňuje cíle skoků, které jsou vždy první instrukcí nového základního bloku a nakonec část (b) ukazuje kód rozdělený do 7 základních bloků podle předchozích dvou pravidel, které jsou zakresleny do grafu toku řízení v části (d).

Propojení základních bloků ze sekce (b) do grafu v (d) na obrázku 4.2 je provedeno ve dvou krocích.

- Na základě skokových instrukcí z kódu vzniknou uspořádané dvojice základních bloků: $(1, 3)$, $(2, 4)$, $(6, 5)$ a $(5, 7)$.
- Propojením sekvenčně po sobě jdoucích bloků, které nekončí instrukcí `goto`, dostáváme dvojice $(1, 2)$, $(3, 4)$, $(4, 5)$ a $(5, 6)$.

4.3 Další extrahovaná data

Vzhledem k tomu, že výstup nástroje, který je cílem této práce, má obsahovat instrukce některé z obecných instrukčních sad, nestačí pouze rozdělit sekvence instrukcí do základních bloků. Kvůli pozdějšímu překladu je potřeba rozlišit jejich operační kódy a operandy, a také vytvořit si záznamy o lokálních proměnných. Dále je nutné jednotlivým instrukcím přiřadit původní programovou lokaci, aby bylo později možné hlásit chyby nalezené automatickými testy platformy Testos přímo s umístěním nalezeného problému.

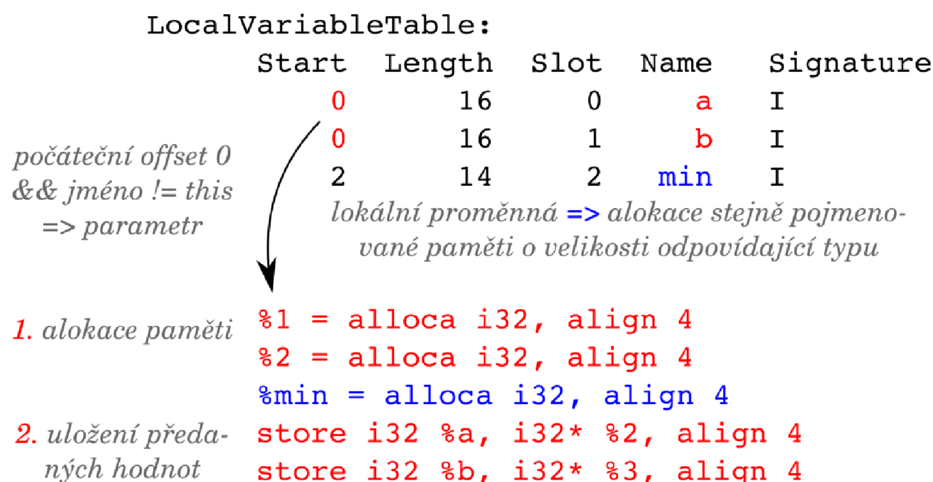
Kapitola 5

Návrh překladu instrukcí z bajtkódu jazyka Java do LLVM IR

Jako obecná instrukční sada, která bude používána v rámci platformy Testos, byla vybrána LLVM IR, protože má silnou nástrojovou podporu a díky rostoucí popularitě překladače *clang* také velkou uživatelskou bázi. Sada LLVM IR je z hlediska zápisu programu na srovnatelné úrovni jako bajtkód jazyka Java, ale jejich provádění se výrazně liší.

Zatímco bajtkód operuje nad zásobníkem, LLVM používá velké množství pomocných proměnných. Instrukce LLVM IR obsahují prakticky veškerá data potřebná k jejich provádění explicitně – včetně délek polí, struktury uložení objektů v paměti, alokace prostoru pro lokální proměnné a mnoha jiných informací, které jsou z bajtkódu prakticky nezjistitelné.

Přes všechny překážky, které se v tomto překladu vyskytují, je jeho cílem získat ke každé sekvenci instrukcí bajtkódu takovou sekvenci v LLVM IR, aby jejich sémantika byla shodná. Hned první úkon překladu spočívá v převodu záznamů o lokálních proměnných z bajtkódu na jejich alokaci v LLVM IR, jak je znázorněno na obrázku 5.1. Dalším o dost složitějším úkolem je poté překlad jednotlivých sekvencí instrukcí.



Obrázek 5.1: Převod tabulky lokálních proměnných bajtkódu jazyka Java na alokaci prostoru pro tyto proměnné v instrukční sadě LLVM IR.

Instrukční sada LLVM IR používá jiné vyjádření datových typů než bajtkód jazyka Java. Celočíselné typy značí jako `iN`, kde písmeno `N` vyjadřuje počet bitů, které bude hodnota

tohoto typu zabírat. Hodnoty s plovoucí desetinou čárkou jsou ukládány jako `float`(32 bit) nebo `double`(64 bit). Reference se značí * za označením typu, na který reference ukazuje. Největší rozdíl je však v uchopení datového typu objektů. V souboru s LLVM IR jsou všechny používané třídy nejdříve strukturálně deklarovány, tzn. že pro pro třídu

```
class testClass {
    public :
        double b[10];
        int a;
};
```

existuje v LLVM IR souboru řádek

```
%class.testClass = type { [10 x double], i32 }
```

deklarující nový typ `%class.testClass`. Tato deklarace je jednou z důležitých součástí metadat uvedených u každého výstupního CFG. V rámci překladu instrukcí potom může být původní zápis `Lidentifikace/třídy;` převeden na `%Lidentifikace/třídy`.

5.1 Dělení instrukcí podle možností překladu

Jednotlivé instrukce bajtkódu lze podobně, jak je zmíněno v [4], rozdělit do několika skupin podle použitého způsobu překladu.

Pro **instrukce bajtkódu modifikující obsah zásobníku s operandy** nejsou generovány žádné nové instrukce sady LLVM IR. Patří mezi ně například `dup` (duplikace hodnoty na vrcholu zásobníku), `swap` (prohození dvou hodnot na vrcholu zásobníku), `pop`, nebo načítání konstant na vrchol zásobníku. Ty se v LLVM neukládají do nových pomocných proměnných, ale jsou používány přímo, a tak jejich vložení na zásobník nemá přímý překlad. Sémantika této operace se projeví až při použití dané hodnoty. Jak tato problematika napovídá, je pro překlad instrukcí bajtkódu do LLVM IR potřeba mít vlastní simulaci zásobníku s operandy.

Instrukce přeložitelné 1 ku 1 patří k nejjednodušším bodům tohoto překladu. Jedná se typicky o aritmetické operace, základní načítání a ukládání lokálních proměnných, nebo jednoduché volání metod. Pro tyto instrukce existují přímé ekvivalenty v obou sadách a jediným úkolem překladače je tedy správně převést operandy ze bajtkódového zásobníku do pomocných proměnných LLVM. Například se jedná o instrukci `Tload_S`, jejíž význam pro JVM byl popsán v podkapitole 3.2. V rámci překladu je (i) vytvořena nová pomocná proměnná pro LLVM, do níž je načtena hodnota lokální proměnné v LLVM, která byla alokována jako ekvivalent proměnné bajtkódu typu `T` ze slotu `S` a (ii) vloženo jméno této nové pomocné proměnné na vrchol simulovaného zásobníku s operandy, pro zpřístupnění této hodnoty jako operandu. Následovat pak může další instrukce z této skupiny – např. `imul`, která se v bajtkódu uvádí bez operandů. Jejím ekvivalentem v LLVM IR je instrukce `mul`, která očekává operandy dva. Podobně jako JVM musí překladač dosadit za operandy pro násobení první dvě hodnoty z vrcholu zásobníku. V těch jsou načtené buď konstanty, nebo jména pomocných proměnných, jak bylo popsáno dříve.

Instrukce ekvivalentní sekvencím operací, jakou je například práce s poli, nebo vyhodnocování podmínek nad složitějšími datovými typy, jsou o něco komplikovanější právě kvůli zodpovědnosti za zvolení správné sekvence instrukcí v LLVM IR, která kompletně zachová původní sémantiku. Do této skupiny patří instrukce `baload`, která má za úkol načíst na zásobník jeden byte z pole, z prvku na indexu uloženého na vrcholu zásobníku, z pole daného referencí na druhém místě zásobníku. Při překladu je generována nejprve

instrukce `getelementptr`, která získá adresu požadovaného prvku, následována instrukcí `load`. Ta z dané adresy přečte hodnotu a uloží ji do pomocné proměnné a zároveň vloží jméno této proměnné na vrchol zásobníku s operandy pro další použití.

Specifické instrukce, které provádějí operace nad objekty JVM, mezi které řadíme např. `arraylength`, nebo `instanceof`, jsou v mnohých případech přeložitelné buď obtížně, nebo vůbec, kvůli již zmiňovaným odlišnostem těchto dvou sad. V kódu jazyka Java však nejsou tyto instrukce ničím neobvyklým, a proto i ty, pro které vhodný ekvivalent neexistuje, musí být alespoň symbolicky přeloženy. Jako řešení tohoto problému bylo zvoleno nahrazení daných operací voláním imaginárních funkcí, které si zachovávají jména původních instrukcí. Při analýze se tento úsek kódu bude tvářit jako volání knihovní funkce, u které může případný čtenář předpokládat chování podle významu dané instrukce v JVM.

V případech, kdy překladač sahá k této možnosti, jde často o možnost poslední, avšak o nic méněcennější než ostatní způsoby překladu. Pokud je instrukce simulována jako knihovní volání, bude při analýze akceptována jako úsek kódu, za jehož správnost autor analyzovaného kódu nezodpovídá, což vyhovuje i pro získávání výsledků těchto specifických instrukcí. Je ovšem potřeba s tímto řešením při důkladnější analýze výsledných CFG počítat.

Kompletní překladová tabulka mezi instrukcemi bajtkódu a sady LLVM IR společně s původními i novými významy je uvedena v příloze C.

5.2 Problematické překlady

Hledání ekvivalentů pro jednotlivé instrukce není jediným úkolem tohoto překladu. Pokud má být kompletně zachována sémantika celých sekvencí instrukcí, musí se překladač vypořádat s celou řadou problémů. V první řadě jde o nedostatky překladu v případě, kdy by v JVM proběhlo vyhodnocení výrazu, který by zabránil tomu, aby po vykonání všech příkazů patřících do jednoho základního bloku zůstaly na zásobníku hodnoty, které v budoucnu nemusejí být přečteny.

5.2.1 Selektivní datová závislost

Praktickou ukázkou tohoto problému je kód na obrázku 5.2. Z navrženého způsobu překladu instrukcí vyplývá, že před vykonáním instrukce s offsetem 25 bude na vrcholu simulovaného zásobníku uložena hodnota proměnné `x` a pod ní proměnné `y`. Jakmile tedy překladač dospěje k instrukci 25, přeloží ji jednoduše jako uložení hodnoty z vrcholu zásobníku do proměnné `z`. Ať tedy bude výsledek předchozího porovnání jakýkoli, do proměnné `z` bude vždy uložena hodnota proměnné `x`. Hodnota proměnné `y` zůstane nepřečtená na zásobníku a z instrukcí 17 až 24 budou vytvořeny dva základní bloky, z nichž první bude obsahovat pouze jeden skok, a druhý zůstane úplně bez instrukcí. Řešení tohoto problému spočívá v symbolické exekuci bajtkódu. Ta je ovšem nad rámec této bakalářské práce a bude úkolem pro další vývoj.

5.2.2 Kompletní definice datového typu

Další překážkou pro tento překlad je získání potřebných údajů k definici datových typů jednotlivých objektů nebo polí. Z bajtkódu je možné vyčíst typy pouze těch polí objektu, která jsou v dané třídě použita. Navíc zde není uvedeno pořadí uložení v paměti, nebo

```
int min = x > y ? y : x

Instrukce bajtkódu                     Symbolický  Význam
odpovídající výrazu                stav zásobníku  operací
                                      po provedení
                                     instrukce (bez
                                      vyhodnocení)
  ...                                          → |
  15: iload_1                               → x  |
  16: iload_2                            → y, x  |
  17: if_icmple      24                        → |  if y <= x goto 24
  20: iload_2                               → y  |
  21: goto           25                     → y  |
  24: iload_1                            → x, y  |
  25: istore_3                              → y  |  min = x
  ...
LocalVariableTable:
  Start  Length  Slot  Name   Signature
    13      17     1      x   I
    15      15     2      y   I
    26       4     3    min   I
```

Obrázek 5.2: Ukázka bajtkódu zkompilovaného pro ternární operátor v jazyce Java. Modře je pro každou instrukci vypsán symbolický stav zásobníku (vrchol je naznačen šipkou) po jejím provedení bez vyhodnocování skoků a podmínek (tzn. tak, jak funguje překlad).

případné délky polí, pokud nebyla vytvořena v rámci dané metody (v tom případě musí být instrukci pro vytvoření nového pole jeho požadovaná délka předána).

V tomto ohledu je možné považovat pole, která jsou zpracovávaným metodám pouze předávána, za ukazatele. Tato praxe je známá např. z jazyka C, se kterým je sada LLVM IR kompatibilní. Způsobí jeden rozdíl v chování programu, a to ten, že v případech, kdy by JVM ukončilo provádění programu s výjimkou znamenající přístup mimo hranice pole, způsobí tato situace v LLVM pád programu kvůli výpadku stránky.

Kompletní definici datového typu objektu, jakou LLVM vyžaduje, nemusí být možné z informací uvedených v bajtkódu sestavit. Jedinou možností tedy je typ objektů definovat pouze z údajů, které jsou díky jejich použití známy. Vzhledem k tomu, že pro zamýšlenou analýzu nezáleží na pořadí umístění v paměti, mohou být jednotlivá pole uvedena v libovolném pořadí – zde bylo zvoleno pořadí podle jejich použití v kódu.

Tento nedostatek by bylo možné do budoucna řešit následnou analýzou metadat uvedených u extrahovaných grafů. Odtud by v rámci jednoho projektu mělo být zřejmé, že se podle názvu jedná o tentýž objekt. Jednotlivé záznamy by se takto daly sloučit za účelem získání kompletnější definice.

5.2.3 Zarovnání paměti

Neposlední komplikací překladu je explicitně uváděné zarovnávání paměti uváděné v jednotkách byte – např. typ `integer`, tedy v LLVM `i32`, se zarovnává na 4B apod. Toto zarovnání je uváděno u instrukcí pracujících s pamětí, jakou je také instrukce `load`, která z pojmenovaného místa v paměti – ukazatel na hodnotu určitého typu – nahraje do nové pomocné proměnné danou hodnotu.

```
%1 = load i32* %x, align 4
```

U jednoduchých datových typů je generování zarovnání v rámci překladu jednoduché, problémy se však objevují u položek objektů. Zde LLVM předpokládá kompletní znalost

datového typu, a to především maximální délku jedné položky, podle které musí být zarovnávány i všechny ostatní. Jak již bylo popsáno dříve, informace o všech jednotlivých položkách nejsou z bajtkódu dopředu zjistitelné. Rozsah čísla pro indexaci jednotlivých prvků i jejich zarovnáváni v paměti je proto voleno shodné, jako pro největší datové typy – `long` a `double`.

Pokud se mezi parametry nebo položkami objektu vyskytne pole, je v kódu považováno za ukazatel. Pokud je později do takto alokovaného prostoru ukládána reference na nové pole, jehož rozměry jsou známé, je původní ukazatel na datový typ přetypován na odpovídající pole, a do jeho hodnoty je vložena reference na první prvek nového pole.

Už dříve bylo zmíněno, že pro požadovanou analýzu na uložení v paměti tolik nezáleží. Ačkoli by tyto nepřesnosti mohli být zdrojem problémů, pokud by měl přeložený kód být prováděn. Cílem prováděného překladu je získat sémanticky ekvivalentní program z hlediska operací daných kódem. Nejdůležitější tedy je, aby tato chyba nenarušovala sémantiku jednotlivých operací, což v tomto případě nehrozí.

5.2.4 Práce s poli objektů a elementy polí

Podstatný rozdíl mezi převáděnými platformami tkví také v práci s elementy datových struktur. Zatímco JVM získává tyto elementy přímo z daných struktur načtených na zásobníku, LLVM takto pracuje s jejich referencemi. Tento problém lze vyřešit připojením jména původní proměnné, která je v LLVM referencí na daný typ, k datům pomocné proměnné vytvořené na základě instrukce `aload`. Pokud má být z takto načtené struktury vybrán některý její element, nahradí překladač tuto pomocnou proměnnou jménem a typem původní lokální proměnné. V kódu by se takto vyskytnou pomocné proměnné, které nebudou využity, což ovšem nepředstavuje z hlediska porozumění, ani případného provádění přeloženého kódu, žádný problém.

5.2.5 Výjimky

Neopomenutelnou součástí programů v jazyce Java jsou výjimky. Mohou být inicializovány jak z vlastního kódu, tak z velkého množství instrukcí bajtkódu – tedy z JVM – což je ten častější případ. Mnoho výjimek vyvolaných instrukcemi bajtkódu jen předchází tomu, aby bylo vykonávání programu ukončeno operačním systémem např. kvůli výpadku stránky apod. Mohlo by se tedy stát, že výjimku, kterou měl v úmyslu programátor později zachytit, překladač nepřeloží, a pokud by byl přeložený program takto prováděn, bude místo ošetření vzniklé výjimky násilně ukončen. Překlad má ovšem sloužit k následné analýze toho, co se v jednotlivých základních blocích děje, aby na jejím základě mohly být generovány jednotkové testy. Uvedený problém proto reálně nenastává a ošetření případů, které by vedly k výjimce ve vykonávání původního programu, zůstává úkolem pro následnou analýzu grafů za účelem získání testovacích případů.

Kapitola 6

Nástroj pro extrakci grafů toku řízení s instrukcemi v LLVM IR z bajtkódu jazyka Java

Výsledný nástroj kombinuje činnosti z kapitol 4 a 5. Jak již bylo nastíněno na začátku první z nich, dokáže zpracovat vstup ve formě:

- přeložitelného[1] zdrojového souboru v jazyce Java (přípona `.java`),
- binárního souboru ve formátu `class`, nebo
- archivu programu jazyka Java s příponou `.jar`.

Výstup je ukládán do souboru ve formátu JSON, jehož objekty vytvořené pro tento účel jsou specifikovány[2] platformou Testos.

Objekt **Module** je kořenovým prvkem výstupního JSON souboru, který obsahuje kolekci všech extrahovaných grafů toku řízení a případné globální proměnné.

Jeden **graf** je reprezentován objektem s kolekcí všech obsažených základních bloků, identifikátorem tvořeným zpravidla ze jména původní metody, z níž graf vznikl, seznamem identifikátorů vstupních a koncových bloků a vyjmenovanými zdrojovými soubory.

Každý **základní blok** obsahuje kromě kolekce obsažených instrukcí, vlastního identifikátoru a původní programové lokace také seznamy identifikátoru svých předchůdců a následovníků.

Instrukce musí obsahovat operační kód, případně pokud se jedná o instrukci bajtkódu bývá tento kód dále rozdělen na jméno instrukce a operandy. Tento objekt může obsahovat také údaj o původní programové lokaci, který se vynechává pouze v případě, že základní blok, ke kterému tato instrukce náleží, se celý nachází na jediném řádku a není tedy potřeba blíže specifikovat umístění konkrétní instrukce.

Objekt reprezentující **programovou lokaci** obsahuje minimálně jméno zdrojového souboru a nejnižší číslo řádku z původního kódu, na němž se daný objekt nacházel. Dále

[1] Pokud nebude možné předaný soubor přeložit, bude uživateli tento problém sdělen společně s hláškami překladače.

[2] `https://pajda.fit.vutbr.cz/testos/cfgqe/blob/master/doc/cfglang.md`

může udávat také počáteční a koncový sloupec, který je ovšem v našem případě vždy vynechán, protože tento údaj nelze z bajtkódu vyčíst. Rozmezí řádků původního kódu je zde uváděno jako minimum a maximum např. kvůli cyklům `for`, kde instrukce vykonávané v těle cyklu jako poslední vznikly z části hlavičky s inkrementací, a proto by určování rozsahu takového základního bloku podle čísel řádků první a poslední instrukce tohoto bloku vedlo k úkazům, kdy blok končí o řádek dříve, než začíná, a jeho opravdový rozsah zůstane neznámý.

Kromě specifického výstupu jsou na implementovaný nástroj kladeny i další požadavky plynoucí z toho, že (i) platforma Testos má být projektem, do nějž bude postupně zapojováno větší množství studentů, a (ii) jakožto nástroj pro testování musí vykazovat velice spolehlivé chování. Mezi tyto požadavky patří:

- deterministické chování,
- snadná modifikovatelnost a udržovatelnost,
- jednoduché použití a
- kvalitní testování.

6.1 Objektový návrh

Výsledný nástroj je implementován v jazyce Java a veškerá funkcionalita je tedy formulována do objektových modulů. Jejich kompletní schéma je uvedeno v příloze D. Mezi použité třídy patří:

`SourceCompiler` – kompilace zdrojových souborů jazyka Java daných cestou k nim do `class` souborů uložených v dočasné složce v adresáři nástroje, která bude po dokončení běhu programu smazána.

`JBCParser`[3] – načítání textové formy `class` souborů, získávané pomocí nástroje *javap* způsobem popsaným v podkapitole 3.2.

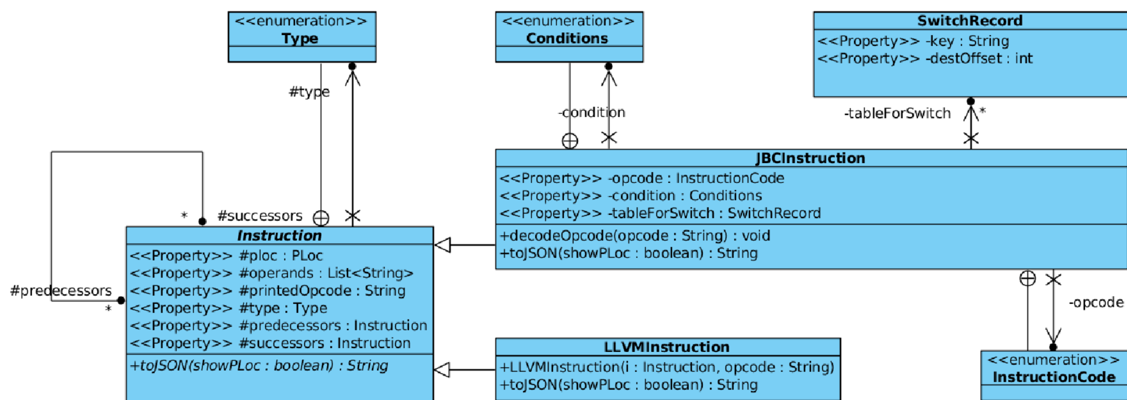
`Method` – reprezentace jednotlivých metod, obsahuje seznam všech instrukcí dané metody a kolekci lokálních proměnných.

`PLoc` – reprezentace programové lokace. Nese záznam o minimálním a maximálním čísle řádku původního kódu včetně jména zdrojového souboru.

`Instruction` – reprezentace jedné instrukce, obsahuje všechny údaje potřebné pro tisk do výstupního JSON formátu. Pro účely tohoto nástroje jsou potřebné dva typy instrukcí, kterými jsou (i) instrukce bajtkódu a (ii) instrukce LLVM IR. Aby bylo bylo možné s oběma typy instrukcí pracovat na úrovni základních bloků uniformně, bylo pro tuto část nástroje navrženo schéma z obrázku 6.1.

`JBCInstruction` – obsahuje množství konkrétnějších údajů, než její nadtřída. Nejdůležitějším z nich je výčtový typ reprezentující kód instrukce, kterým je později řízen překlad do LLVM IR.

`LLVMInstruction` – má kromě funkcionality, kterou dědí z nadřazené třídy, ještě specifickou implementaci metody, která vytvoří její záznam pro JSON formát.



Obrázek 6.1: Třídní schéma modulu instrukcí pro navržený nástroj.

`BasicBlock` – reprezentace základního bloku, obsahuje sekvenci příslušných instrukcí, objekt programové lokace s rozsahem řádků kódu, kde se tento základní blok nachází, číselný identifikátor a seznamy identifikátorů předchozích a následujících základních bloků.

`CFG` – reprezentace jednoho grafu. Obsahuje identifikátor ve formě pojmenování dané metody odpovídající instrukční sadě, seznamy lokálních proměnných, identifikátorů počátečních a koncových základních bloků a také kolekci všech obsažených základních bloků ve formě objektů `BasicBlock`.

`CFGExtractor` – vytvoří pro každou instanci třídy `Method` nový objekt `CFG` a její instrukce rozdělí na základě identifikace skokových instrukcí a cílů těchto skoků do sekvencí jednotlivých základních bloků. Tyto sekvence jsou přiřazeny odpovídajícím instancím třídy `BasicBlock`.

`JBC2LLVMTranslator` – překládá instrukce z bajtkódu do sady LLVM IR po jedné v rámci kontextu jednoho objektu `CFG`, simuluje zásobník s operandy.

`FieldRegister` – uchovává a spravuje záznamy o jednotlivých používaných polích objektů, s nimiž analyzovaný program pracuje.

`Slot` – reprezentace jednoho slotu pro jednu metodu. Obsahuje záznamy o jednotlivých proměnných, které zde budou ukládány, spolu s jejich rozsahy platnosti.

`LocalVarsRegister` – spravuje všechny lokální proměnné metody, která je aktuálně zpracovávána. Po dobu zpracovávání jedné metody udržuje seznam objektů třídy `Slot`. Také uchovává pro jednu metodu registr jmen všech lokálních proměnných, kterým se řídí alokace jejich ekvivalentů pro LLVM, kde by duplicitní jména na rozdíl od bajtkódu, kde jsou rozlišeny pozicí a platností ve slotech, znamenala nerozlišitelnost daných proměnných. Duplicitní jména jsou tedy modifikována přidáním sekvenčního indexu použití daného jména.

6.2 Vzorový příklad fungování

Implementovaný nástroj dostane na vstupu

- zdrojový `java` soubor. Pomocí nástroje *javac* se ho pokusí zkompilovat do složky s názvem jbc, která bude pro tento účel vytvořena v aktuálním adresáři. Pokud se toto podaří, je cesta ke složce jbc interně uložena jako cesta ke `class` souborům. V případě, že se kompilace nezdaří, je na výstup vytištěna odpovídající hláška společně s výstupem kompilátoru a vykonávání programu je ukončeno.
- `jar` archiv, který rozbalí pomocí nástroje *jar* do stejné složky jako v předchozím případě. Cesta k této složce je opět uložena jako cesta ke `class` souborům.
- `class` soubor. Cesta k němu je uložena jako cesta ke zpracovávaným `class` souborům.

Proměnná obsahující cestu ke `class` souborům je předána modulu pro načítání bajtkódu. Ten, pokud dostal na vstupu název složky, získá jména všech `class` souborů, které tato složka obsahuje. Ta jsou seřazena podle abecedy a následně jsou tyto souboru jednotlivě načítány. Pokud byl na vstupu jediný soubor, je zpracován pouze ten.

Modul pro načítání bajtkódu používá regulární výrazy k načtení a následnému uložení veškerých potřebných informací do objektových reprezentací metod a instrukcí. U skoků jsou doplněny odkazy ze skokové instrukce na cíl a stejně tak i do cílové instrukce je vložen odkaz na skokovou, pro účely pozdější identifikace hranic základních bloků. Výsledná kolekce metod je předána modulu pro extrakci grafů toku řízení.

`CFGExtractor` prochází postupně kolekci všech načtených metod z veškerých zpracovávaných `class` souborů. Podle nich vytváří kolekci grafů, jeden pro každou metodu. Do grafů vkládá základní bloky s částmi sekvence instrukcí původní metody, rozdělené právě podle hranic jednotlivých bloků.

Výsledná kolekce grafů je buď vytištěna do souboru s nepřeloženými instrukcemi bajtkódu, nebo je předána třídě `JBC2LLVMTranslator`. Ta prochází po jednom všechny grafy, pro které si vytváří a udržuje záznamy o lokálních proměnných a v jejichž základních blocích nahrazuje původní sekvence instrukcí bajtkódu instrukcemi sady LLVM IR. Po tomto kroku jsou výsledné grafy vytištěny do souboru a vykonávání programu je ukončeno.

Kapitola 7

Zajímavé implementační detaily

Při vývoji nástroje byl kladen důraz jak na snadnou rozšiřitelnost, tak na deterministické chování, které je třeba zajišťovat hned na několika místech programu. Pokud je nástroji předán zdrojový soubor, nebo `jar` archiv, je potřeba analyzované `class` soubory nejdříve seřadit podle jména. Na různých platformách totiž mohou platit jiná pravidla pro řazení souborů ve složce a což by se mohlo projevit rozdílným pořadím extrahovaných grafů ve výstupním souboru. Další nedeterminismus by mohlo vnést také používání některých datových typů modelujících neuspořádané kolekce, jakým je v jazyce Java např. objekt `HashMap`.

Dalším zajímavým úkolem implementovaného nástroje je případný překlad zdrojového souboru, nebo extrakce `class` souborů z `jar` archivu. K tomuto jsou používány nástroje spouštěné z příkazové řádky – *javac* a *jar* – a navíc později ještě nástroj *javap* potřebný pro samotné čtení bajtkódu.

7.1 Bližší vysvětlení principu extrakce základních bloků

Poté, co je zajištěna dostupnost `class` souborů a použitelnost potřebných nástrojů příkazové řádky, přichází na řadu načítání bajtkódu. Jak bylo zmíněno v předchozí kapitole, tuto funkcionalitu zajišťuje objekt třídy `JBCParser`. Ten pracuje na principu stavového automatu a používá regulární výrazy pro rozdělování jednotlivých řádků bajtkódu do logických celků.

Při načítání instrukcí je provedeno dekódování instrukčního kódu a vyplněny údaje o tom, zda má instrukce nějakého nestandardního předchůdce, nebo následovníka – tj. jestli je cílem nějakého skoku, nebo je sama skokovou instrukcí. Po zpracování všech instrukcí dané metody jsou vzniklým objektům `JBCInstruction` doplněny údaje o příslušných číslech řádků.

Objekt třídy `CFGExtractor` následně rozdělí sekvenci instrukcí příslušejících k jedné metodě do podsekvencí podle záznamů o předchůdcích a následovnících. Jsou zde také vytvořeny seznamy počátečních a koncových bloků.

7.2 Provádění překladu

Překlad bajtkódu do LLVM IR začíná modifikací samotných identifikátorů grafů, které by měly souhlasit se zápisem, jakým budou z instrukcí volány metody, z nichž grafy vycházejí. Přepis tohoto vyjádření z bajtkódu do LLVM IR vyžaduje krom jiného také doplnění jmen parametrů, která jsou čerpána z tabulky lokálních proměnných, kde jsou identifikována podle začátku platnosti, který je v tomto případě roven první instrukci. Mezi para-

metry se mísí také proměnná `this`, kterou je potřeba z doplňování jmen vynechat. Tato práce je zajištěna metodou, která z předaného jména funkce a seznamu proměnných, jejichž jména mohou být do hlavičky doplněna, dosazuje tyto jména za každý typ parametru získaný překladem signatury z bajtkódu. Ta samá funkce je použita také pro doplnění hodnot proměnných předávaných volaným funkcím jako parametry v samotných instrukcích zpracovávaných metod.

Překlad jednotlivých instrukcí je prováděn v kontextu celého grafu toku řízení, kvůli lokálním proměnným. Pro ty musí být do vstupního bloku vygenerovány instrukce pro jejich alokaci, včetně alokace prostoru pro parametry funkce, z níž daný graf vznikl. Jeden graf také sdílí stejné konstanty z *constant pool*, nebo pomocné proměnné pro LLVM, v nichž jsou uloženy jak reference na lokální proměnné, tak všechny operandy a mezivýsledky.

Pro ulehčení práce s těmito sdílenými hodnotami byly vytvořeny třídy pro jejich správu. Třída `FieldRegister` uchovává ve dvourozměrném seznamu záznamy o známých polích objektů. Pokud některá s instrukcí vyžaduje použití nějakého pole objektu, `FieldRegister` prohledá své záznamy o polích pro danou třídu, pokud není požadované pole ještě zaregistrováno, tak jej do záznamů přidá, a sdělí zbytku programu index daného pole v rámci objektu. Ten je potřebný pro explicitně uváděnou indexaci těchto polí v LLVM IR. Záznamy této třídy jsou také využity k doplnění definic datových typů do metadat jednotlivých grafů.

Další podobnou třídou je `LocalVarsRegister`, která spravuje záznamy o lokálních proměnných aktuálně zpracovávané metody – grafu a udržuje informaci o počtu pomocných proměnných, která je klíčová pro další deklarace těchto proměnných.

V rámci překladu je potřeba často převádět různé formy datových typů mezi sebou. Tuto a další podobné funkcionality zajišťuje třída `JBC2LLVM`, která je knihovnou statických funkcí právě pro tyto účely.

7.3 Převod a správa lokálních proměnných

Správa lokálních proměnných je komplikovaná, protože jsou tyto proměnné z bajtkódu adresovány podle přidělených slotů. Ten může být pro více proměnných, jejichž platnost se nekryje, využit vícekrát. Navíc, údaje o začátku a délce platnosti lokální proměnné nesouvisí s její deklarací, ale instrukcemi, pro které se v daném slotu nachází platná hodnota této proměnné. Pokud tedy máme funkci

```
public static int foo(int a, int b) {
    int min;
    if (a < b)
        min = a;
    else
        min = b;

    return min;
}
```

dostaneme v bajtkódu toto

```
public static int foo(int, int);
    descriptor: (II)I
    Code:
       0: iload_0
       1: iload_1
       2: if_icmpge     10
       5: iload_0
```

```
     6: istore_2
     7: goto          12
    10: iload_1
    11: istore_2
    12: iload_2
    13: ireturn
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        7       3     2   min   I
        0      14     0     a   I
        0      14     1     b   I
       12       2     2   min   I
```

V tabulce lokálních proměnných je vidět zdánlivé znovupoužití slotu 2 pro proměnnou `min`. Ze zdrojového kódu ovšem vidíme, že se jedná o tutéž proměnnou. Neznamená to ovšem, že sloty lze takto přidělit více lokálním proměnným pouze pokud se doopravdy jedná o tu samou, dokonce nemusí jít ani o proměnnou stejného datového typu.

Překážkou pro přímý překlad jen na základě jednoho záznamu z tabulky *LocalVariableTable* je v této problematice také skutečnost, že uvedená délka platnosti hodnoty dané proměnné se vztahuje na **prováděné** instrukce. Tedy v uvedeném případě platí hodnota proměnné `min`, která je vložena instrukcí s offsetem 6, pro instrukce 7, **12** a **13**.

Prakticky je tedy nutné namísto sledování těchto údajů o platnosti v tabulce lokálních proměnných vycházet z předpokladu, že daný slot je rezervován pro nějakou proměnnou do té doby, dokud není nahrazena jinou. Při vytváření záznamů o lokálních proměnných je uplatňována tato sekvence akcí:

1. Lokální proměnné z tabulky jsou seřazeny podle indexu první instrukce, pro niž daná proměnná obsahuje platnou hodnotu.

2. Seřazený seznam záznamů je po jednom procházen a jednotlivé proměnné jsou přiřazovány do slotů, přičemž jsou porovnávány s poslední proměnnou, která byla v tomto slotu uložena.

   (a) Pokud se jedná o tu samou proměnnou, není provedena žádná akce. Ve výsledném kódu budou tyto záznamy sloučeny do jediné proměnné s platností přes oba záznamy.

   (b) Pokud se jedná o jinou proměnnou jsou provedeny následující akce:

      i. Pokud existuje nějaká předchozí proměnná v tomto slotu, je její rozsah platnosti zkrácen po offset instrukce o 2 menší než kde začíná platnost nové proměnné. Předpokládá se totiž, že platná hodnota se v daném slotu bude vyskytovat o instrukci později, než je do něj uložena. Při ukládání instrukce (offset - 1) už však musí být za platnou označena ona nová proměnná a proto je platnost předchozí ukončena instrukcí s offsetem instrukce uvedeného v tabulce ve sloupci *Start* - 2.

      ii. Je zaregistrováno nové jméno proměnné – pokud takto pojmenovaná proměnná již existuje, je za její jméno přidán index podle toho, kolikátá takto pojmenovaná proměnná v pořadí to je.

      iii. Proběhne vložení nové proměnné do slotu s platností od offsetu instrukce uvedeného v tabulce ve sloupci *Start* - 1 až po maximální offset instrukce vyskytující se v dané metodě.

7.4 Tisk výsledných grafů

Výsledné grafy, ať už před překladem nebo po něm, jsou tištěny pomocí metody `toJSON`, kterou v různé formě implementují všechny objekty, jejichž reprezentace se má objevit na výstupu. V rámci základních bloků mohou být tištěny buďto instrukce bajtkódu rozdělené na operační kód a operandy, nebo instrukce LLVM IR, které jsou reprezentovány jedním souvislým řetězcem. Tato skutečnost vyžaduje rozdílný způsob tisku pro různé typy instrukcí.

Jednotlivé objekty reprezentující základní bloky obsahují seznamy objektů třídy `Instruction`. Díky tomu, že v Javě jsou všechny metody virtuální, a tedy se vždy hledá jejich nejaktuálnější implementace vzhledem k hierarchii dědičnosti, budou i zde volány metody konkrétních typů instrukcí. To umožňuje pokrýt rozdíly ve výstupu bez nutnosti předávání nějakých příznaků pro způsob požadovaného tisku. Metoda `toJSON` pak může být ve třídě `Instruction` označena jako abstraktní.

Kapitola 8

Ověření správnosti transformace

Fungování implementovaného nástroje bylo ověřováno na dvou úrovních – jednotkovými a akceptačními testy. Pro každou metodu, s výjimkou těch nejjednodušších `get` a `set`, byl vytvořen test ověřující její chování pro různé skupiny vstupů.

Pro akceptační testy byly použity zdrojové soubory se všemi základními konstrukcemi jazyka Java, přejatými z unit testů projektu llvm-project[1]. Pro ty byly manuálně vytvořeny soubory se vzorovým výstupem jak pro ověření správné extrakce grafů obsahujících instrukce bajtkódu, tak pro kontrolu provedeného překladu.

8.1 Porovnání výstupu překladu instrukcí z bajtkódu jazyka Java do LLVM IR

Pro jednoduché konstrukce dosahuje implementovaný překlad velmi dobrých výsledků, jak je možné vidět na porovnání překladu metody

```
int Ack(int m, int n) {
    if (m == 0)
        return (n + 1);
    else if (n == 0)
        return Ack(m-1, 1);
    else
        return Ack(m-1, Ack(m, n - 1));
}
```

pro výpočet Ackermannovy funkce s výstupem nástroje *clang* pro tutéž metodu, která byla se shodným zápisem zkompilována v jazyce C++, na obrázku 8.1. Jsou zde vyznačeny i rozdíly, které ovšem výslednou sémantiku programu nemění.

Nástroj *clang* si pro návratovou hodnotu alokuje prostor v paměti, což ovšem není nutné a v případě překladu ani vhodné, protože by se tak změnil i výsledný graf toku řízení, a to není cílem překladu instrukcí uvnitř základních bloků. Tento prostor je označen jako `%1` a proměnné `m` a `n` jsou tedy posunuty do paměti označené jako `%2` a `%3`.

Dalším rozdílem je formulace podmínek, která závisí na implementaci kompilátorů. Ačkoli JVM použité ke kompilaci funkce `Ack` používá opačnou podmínku, než nástroj *clang*, je tento rozdíl smazán hned následující instrukcí, kde jsou adekvátně vyměněny také cíle podmíněného skoku.

[1] `https://llvm.org/svn/llvm-project/java/trunk/test/Programs/SingleSource/UnitTests/`

```
i32 @ackermann.Ack(i32 %m, i32 %n)

  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %m, i32* %1, align 4
  store i32 %n, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = icmp ne i32 0, %3
  br i1 %4, label %8, label %5

  <label>:5
  %6 = load i32* %2, align 4
  %7 = add nsw i32 %6, 1
  ret i32 %7


  <label>:8
  %9 = load i32* %2, align 4
  %10 = icmp ne i32 0, %9
  br i1 %10, label %15, label %11

  <label>:11
  %12 = load i32* %1, align 4
  %13 = sub nsw i32 %12, 1
  %14 = call i32 @Ack(i32 %13, i32 1)
  ret i32 %14


  <label>:15
  %16 = load i32* %1, align 4
  %17 = sub nsw i32 %16, 1
  %18 = load i32* %1, align 4
  %19 = load i32* %2, align 4
  %20 = sub nsw i32 %19, 1
  %21 = call i32 @Ack(i32 %18, i32 %20)
  %22 = call i32 @Ack(i32 %17, i32 %21)
  ret i32 %22
```

(a) Kód LLVM IR vytvořený implementovaným nástrojem.

```
i32 @_Z3Ackii(i32 %m, i32 %n)
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %m, i32* %2, align 4
  store i32 %n, i32* %3, align 4
  %4 = load i32* %2, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %9

  <label>:6
  %7 = load i32* %3, align 4
  %8 = add nsw i32 %7, 1
  store i32 %8, i32* %1
  br label %24

  <label>:9
  %10 = load i32* %3, align 4
  %11 = icmp eq i32 %10, 0
  br i1 %11, label %12, label %16

  <label>:12
  %13 = load i32* %2, align 4
  %14 = sub nsw i32 %13, 1
  %15 = call i32 @_Z3Ackii(i32 %14, i32 1)
  store i32 %15, i32* %1
  br label %24

  <label>:16
  %17 = load i32* %2, align 4
  %18 = sub nsw i32 %17, 1
  %19 = load i32* %2, align 4
  %20 = load i32* %3, align 4
  %21 = sub nsw i32 %20, 1
  %22 = call i32 @_Z3Ackii(i32 %19, i32 %21)
  %23 = call i32 @_Z3Ackii(i32 %18, i32 %22)
  store i32 %23, i32* %1
  br label %24

  <label>:24
  %25 = load i32* %1
  ret i32 %25
```

(b) Kód vytvořený nástrojem *clang* pro sémanticky ekvivalentní program v jazyce C++.

Obrázek 8.1: Porovnání výstupu překladu provedeného implementovaným nástrojem a výstupu nástroje *clang* s vyznačenými částmi instrukcí s rozdílným významem.

8.2 Evaluace

Při posuzování správnosti celkové funkcionality implementovaného nástroje je důležité mít na paměti jeho účel, kterým je pomoc při automatizaci tvorby jednotkových testů pro software napsaný v jazyce Java. Jeho nejdůležitější funkcí je správné rozdělení kódu do jednotlivých grafů toků řízení, což se na testovací sadě se všemi základními konstrukcemi

jazyka Java daří spolehlivě. Dalším důležitým, ovšem ne primárním, úkolem tohoto nástroje je překlad instrukcí do sady LLVM IR, kvůli možnosti následné univerzální analýzy.

Prováděný překlad má vzhledem k zásadním rozdílům mezi jednotlivými platformami mnoho problémů a z nich vyplývajících limitů.

Ternární operátory v kódu jsou v aktuální verzi řešeny způsobem, který narušuje sémantikou ekvivalenci přeloženého programu s tím původním. Řešení tohoto problému by vyžadovalo symbolickou exekuci překládaného programu, která však byla zatím vynechána, protože implementovaný nástroj se zaměřuje především na základní programové konstrukce, a řešení tohoto problému tedy přesahuje rozsah této práce.

Správa lokálních proměnných je oproti předchozímu problému tímto nástrojem vyřešena, a to pomocí simulace jednotlivých slotů a rozsahů platnosti jednotlivých hodnot, které zde mohou být v průběhu programu uloženy, což vytváří simulaci očekávaného chování JVM. Může se ovšem stát, že v rámci optimalizace při kompilaci bude vynechán záznam o některém použitém slotu proto, že hodnota do něj uložená nebude nikdy použita. Tuto situaci řeší implementovaný nástroj vynecháním práce s takovýmto slotem, protože pro něj neexistuje alokovaná proměnná pro LLVM. Ztratí se tímto ovšem chyba programátora, který přiřadil hodnotu do proměnné, kterou nemá v úmyslu použít.

Úplná rekonstrukce datových typů je jedním z úkolů, jejichž řešení rozdílnost převáděných platforem téměř znemožňuje. Je prováděna pouze částečně, pro každý graf je typ rekonstruován pouze z používaných polí, což by se mohlo projevit komplikacemi s jejich zpracováním později. tyto datové typy však budou vždy pojmenovány stejně a je proto eventuálně možné tyto definice později propojovat, pokud by někdy bylo potřeba je znát co nejpodrobněji.

Použití imaginárních knihovních volání jako simulace téměř, nebo vůbec nepřeložitelných instrukcí je pro následnou analýzou stabilním, jednoduchým a vhodným řešením. Sémantika zůstane pro analyzátor plně zachována, je ovšem nutné při jeho tvorbě s tímto řešením počítat.

Implicitní konverze datových typů, které JVM běžně provádí jak u návratových hodnot, tak u parametrů volaných funkcí, je prováděna pouze u návratových hodnot, protože pro každý graf – metodu je návratový typ daný a známý. Typová konverze proměnných předávaných volaným funkcím jako parametry je však více komplikovaná a běh nástroje by jejím prováděním byl násobně pomalejší. Její vynechání navíc nebrání porozumění kódu a proto ji vzhledem k aktuálním požadavkům není nutné implementovat.

Kapitola 9

Závěr

Cílem této práce bylo navrhnout a implementovat nástroj pro extrakci grafů toku řízení z programů jazyka Java, za účelem pomoci automatizovat tvorbu jednotkových testů pro software napsaný v tomto jazyce. Tento nástroj vzniká na bázi požadavků platformy Testos, která si klade za cíl automatizovat testování obecně pro všechny masově používané platformy (Java, jazyky C a C++,...). Výsledné grafy tedy v rámci integrace do této platformy musí (i) obsahovat instrukce obecné sady LLVM IR a (ii) být tištěny v požadovaném specifickém formátu JSON.

Vzniklý nástroj dokáže spolehlivě extrahovat grafy toku řízení z programů v jazyce Java zadaných kteroukoli z nejběžnějších forem. Jeho výstup může obsahovat buď instrukce bajtkódu, nebo sady LLVM IR. Sekvence instrukcí přeložené do LLVM IR jsou pro základní konstrukce jazyka Java sémanticky ekvivalentní a po nastudování specifik tohoto překladu by mělo být bez problémů možné na základě přeložených instrukcí provádět analýzu za účelem tvorby testů.

Návrh a implementace výsledného nástroje řeší dva hlavní problémy – (i) extrakci grafů toku řízení a (ii) překlad instrukcí bajtkódu do sady LLVM IR. Extrakce grafů je na úrovni bajtkódu úkol řešitelný poměrně jednoduchým algoritmem. Na druhé straně překlad instrukcí je možný jen do určité míry. Vytvoření takového programu v LLVM IR pro obecný program v jazyce Java, který by bylo možné spouštět a jejich chování bylo doopravdy ekvivalentní není z důvodu odlišnosti těchto platforem jednoduše proveditelné. Pro účely platformy Testos ovšem stačí řešení, ze kterého je jasně vidět, co se v původním kódu na kterém místě přesně děje a taková úroveň překladu proveditelná je a implementovaný nástroj ji do velké míry naplňuje.

Navržený překlad má ještě drobné funkční rezervy, jejichž řešení je ovšem nad rámec této bakalářské práce. Proto také, jelikož je prozatím nutná funkcionalita pouze na základních konstrukcích, jsou tyto rezervy ponechány k vyřešení při dalším rozvoji, kterého se tomuto nástroji v rámci platformy Testos dostane.

Již nyní existuje v této platformě nástroj pro vyhledávání cest ve výstupních grafech požadovaného formátu a podobných nástrojů navazujících na tuto práci bude přibývat. Implementovaný nástroj je tedy v rámci Testos počátkem automatizace jednotkových testů, která bude nad touto prací v dohledné době vystavěna.

Literatura

[1] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, 2008, ISBN 978-0-521-88038-1.

[2] Brosgol, B.; Comar, C.: DO-178C: A New Standard for Software Safety Certification. online prezentace, Duben 2010.
URL `http://www.ieee-stc.org/proceedings/2010/pdfs/bmb2623.pdf`

[3] de Carvalho Gomes, P.: *Sound Modular Extraction of Control Flow Graphs from Java Bytecode*. Diplomová práce, KTH Royal Institute of Technology, Listopad 2012.

[4] Geoffray, N.: The VMKit project: Java and .Net on top of LLVM. pdf, Srpen 2008.
URL `http://llvm.org/devmtg/2008-08/Geoffray_VMKitProject.pdf`

[5] Kondula, V.: C/C++ to CFG. Gitlab repozitář, 2017.
URL `https://pajda.fit.vutbr.cz/testos/cpp2cfg`

[6] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: *The Java® Virtual Machine Specification*. Oracle, Březen 2015.
URL `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`

[7] Skupina Testos: Domovská stránka projektu Testos. online, 2017, Vysoké učení technické v Brně, Fakulta informačních technologií.
URL `http://testos.org`

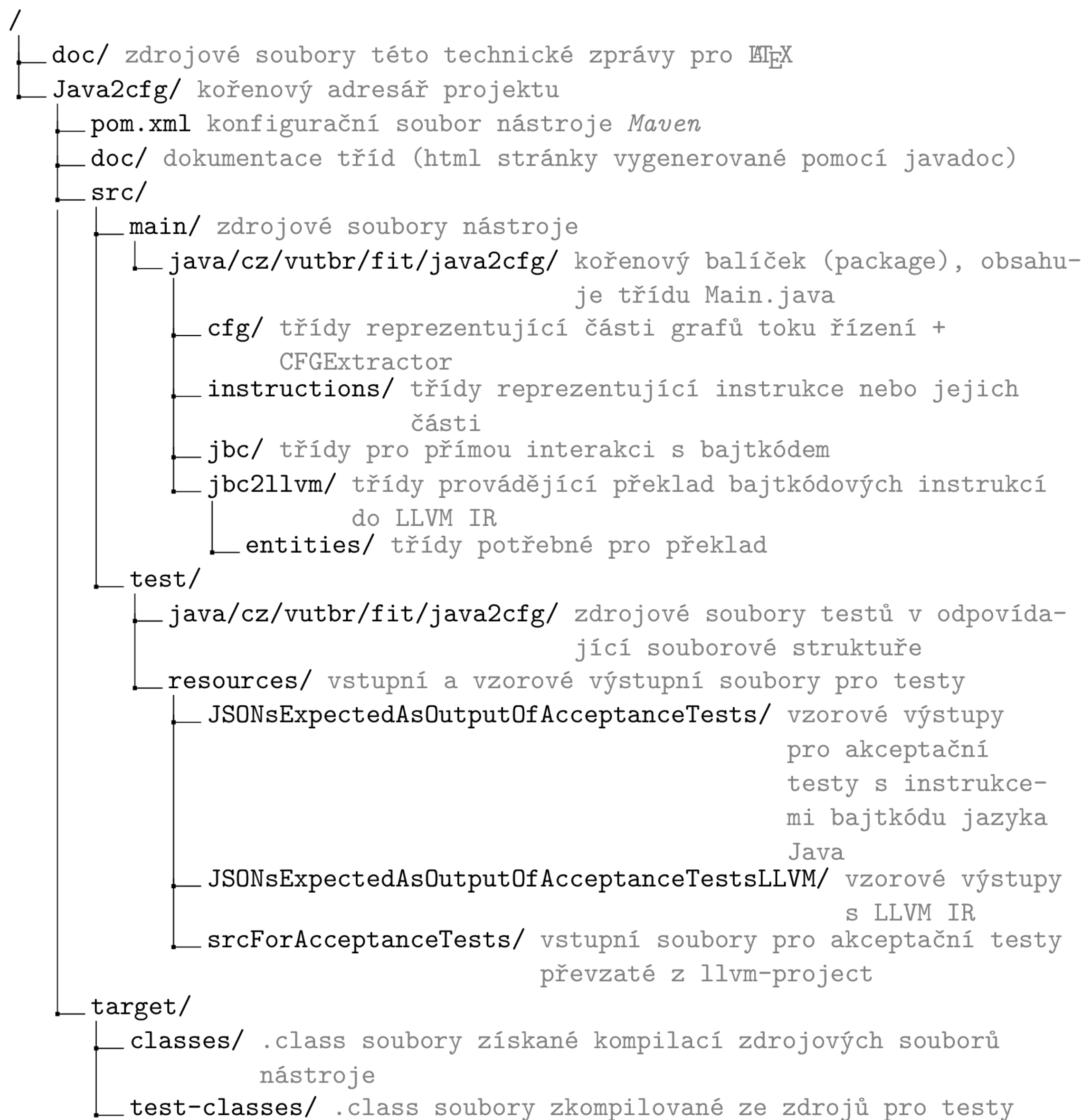
[8] Smrčka, A.: Testování a dynamicka analýza - Úvod a pojmy v testování. online prezentace, 2017.
URL `https://wis.fit.vutbr.cz/FIT/st/course-files-st.php?file=%2Fcourse%2FITS-IT%2Flectures%2F1-uvod.pdf&cid=10349`

[9] Vít, R.: Control Flow Graph Query Engine. Gitlab repozitář, 2017.
URL `https://pajda.fit.vutbr.cz/testos/cfgqe`

Přílohy

Příloha A

Obsah přiloženého paměťového média



- Java2cfg-1.0.jar .jar archiv nástroje bez knihoven, na nichž je závislý
- Java2cfg-1.0-jar-with-dependencies.jar .jar archiv se všemi závislostmi
- Java2cfg-1.0-javadoc.jar .jar archiv s vygenerovanou dokumentací tříd
- ... další složky používaných pluginů nástroje *Maven*

Příloha B

Manuál

Pro správné fungování nástroje *Java2cfg* je potřeba mít nainstalovaný balíček *Java Development Kit (JDK)*, kvůli použití nástrojů *javac*, *javap* a *jar*. V případě problémů je vhodné vyzkoušet, zda některý z těchto nástrojů funguje, např. příkazem

```
$ javap -c NejakyClassSoubor.class
```

Pokud tento příkaz vypíše textovou formu daného `.class` souboru, měl by nástroj bez problémů fungovat.

B.1 Použití stávajícího programu

Program se spouští z příkazové řádky ve formátu:

```
$ java -jar [cesta/k/archivu/]Java2cfg-1.0-jar-with-dependencies.jar cesta/
k/analyzovanemu/souboru [--output=cesta/k/vystupnimu/souboru] [--jbc]
```

Cestu k archivu je je nutné uvádět v případě, že se nachází jinde než ve složce nalistované v příkazové řádce. Zadávání dalších parametrů se řídí následujícími pravidly.

- `Cesta/k/analyzovanému/souboru` je povinný parametr, který musí být uveden na prvním místě. Může být zadána absolutně nebo relativně vůči aktuální pozici v souborovém systému nalistované v příkazové řádce.
- Parametr `--output=`... je nepovinný, specifikuje soubor, do kterého má být zapsán výstupní JSON. I tento parametr může obsahovat absolutní nebo relativní cestu. Pokud není tento parametr specifikován, bude výstup uložen do souboru `output.json`, který se bude nacházet ve složce, odkud byl nástroj spuštěn. Obsah výstupního souboru odpovídá specifikaci[1] platformy Testos[2].
- Volitelný parametr `--jbc` způsobuje, že výsledné grafy toku řízení budou obsahovat instrukce bajtkódu a ne sady LLVM IR.

Nápověda je tištěna v každém případě, kdy jsou zadané parametry pro nástroj nějakým způsobem nezpracovatelné, spolu s odpovídající chybovou hláškou.

[1] `https://pajda.fit.vutbr.cz/testos/cfgqe/blob/master/doc/cfglang.md`
[2] `www.testos.org`

B.2 Nové sestavení programu

K sestavení nástroje se používá nástroj *Maven* ve verzi 4.0.0. Je tedy nutné mít tento nástroj nainstalovaný buď přímo v této verzi, nebo ve verzi pro daný účel kompatibilní. Příkazy tohoto nástroje se spouští v kořenovém adresáři projektu, kde se nachází soubor `pom.xml` (konfigurační soubor nástroje *Maven*). Pro nové sestavení programu se všemi závislostmi se používá příkaz

```
$ mvn clean package
```

který zkompiluje zdrojové soubory nástroje a zdrojové soubory testů. Poté provede všechny testy a v případě, že proběhnou úspěšně, sestaví archivy `Java2cfg-1.0-javadoc.jar`, `Java2cfg-1.0-jar-with-dependencies.jar` a `Java2cfg-1.0.jar`.

Java2cfg-1.0.jar je archiv obsahující pouze zkompilované zdrojové soubory tohoto nástroje, bez použitých knihoven.

Java2cfg-1.0-jar-with-dependencies.jar obsahuje kromě `.class` souborů tohoto nástroje také všechny potřebné knihovny. Pro spouštění ve složce projektu stačí i verze bez knihoven, není ovšem na rozdíl od této libovolně přenosná.

Java2cfg-1.0-javadoc.jar je archiv s vygenerovanou dokumentací ve formátu html z *javadoc* komentářů v kódu nástroje.

Příloha C

Překladová tabulka instrukcí z bajtkódu do LLVM IR

Vysvětlivky: DT – datový typ, **H** – hodnota, **I** – index, **S** – slot, **P** – počet, **D** – délka, **PODM** – podmínka

BAJTKÓD		**LLVM IR**	
instrukce	**popis**	**instrukce**	**popis**
aaload	Načte hodnotu z pole, na které ukazuje reference z vrcholu zásobníku s operandy. Z pole je čtena hodnota nacházející se na indexu určeném hodnotou uloženou na zásobníku pod odkazem na pole. Čtená hodnota je vložena na vrchol zásobníku.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** %x+1 = load **DT*** %x, align *N*	Získá ukazatel na požadovanou hodnotu z pole, která je uložena do proměnné %x.
aastore	Do pole daného referencí na vrcholu zásobníku s operandy je na index daný hodnotou umístěnou pod referencí na pole uložena hodnota ze druhé pozice pod vrcholem zásobníku.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 I store **DT H**, **DT*** %x, align *N*	Získá adresu prvku pole a uloží do něj danou hodnotu.
aconst_null	Nahraje na vrchol zásobníku hodnotu `null`.	-	Nahraje na vrchol interního zásobníku s operandy hodnotu `null`
aload **S** aload_0 aload_1 aload_2 aload_3	Nahraje na vrchol zásobníku referenci uloženou v lokální proměnné z daného slotu (_číslo ≙ _slot).	-	Nepřekládáme, tato reference je pouze interně vložena na zásobník s operandy k pozdějšímu použití.

anewarray #index	Vytvoří nové pole referencí na komponenty typu daného #indexem do tabulky konstant o počtu prvkù zadaném hodnotou na vrcholu zásobníku s operandy.	%x = alloca [**P** x **DT***], align *N*	Alokuje prostor pro nové pole a referenci na něj uloží do lokální proměnné.
areturn	Vrátí vykonávání programu zpět z funkce a jako návratovou hodnotu předá referenci z vrcholu zásobníku s operandy.	ret **DT*** **H**	Vrátí vykonávání programu zpět z funkce, jako návratovou hodnotu předá danou referenci.
arraylength	Uloží na vrchol zásobníku délku pole.	%x = call i32 @arraylength()	Simuluje získání délky pole imaginární knihovní funkcí.
astore **S**	Uloží hodnotu z vrcholu zásobníku do lokální proměnné uložené v daném slotu (_číslo ≙ _slot).	store **DT** **H**, **DT*** %jmeno_promenne, align *N*	Uloží danou hodnotu do lokální proměnné odpovídající číslu slotu
astore_0			
astore_1			
astore_2			
astore_3			
athrow	Produkuje výjimku, obsah zásobníku vymazán, na jeho vrchol je poté vložena reference na objekt výjimky.	call void @athrow()	Pro účely tohoto projektu stačí dát najevo co se zde děje voláním imaginární knihovní funkce.
baload	Načte hodnotu typu `byte` z pole jehož reference se nachází na vrcholu zásobníku. Ta je společně s indexem, který je uložený pod ní, nahrazena hodnotou prvku na pole na daném indexu.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** %(x+1) = load i8* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.
bastore	Uloží hodnotu typu `byte` z vrcholu zásobníku do pole jehož reference se nachází na zásobníku pod hodnotou, do prvku na indexu, který je uložený pod referencí na pole.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** store i8 **H**, i8* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu uloží danou hodnotu.
bipush <byte>	Vloží na vrchol zásobníku zadaný byte jako typ `integer`.	-	Nepřekládá se, hodnota je interně vložena na zásobník s operandy pro další použití.
caload	Načte hodnotu typu `char` z pole jehož reference se nachází na vrcholu zásobníku. Ta je společně s indexem, který je uložený pod ní, nahrazena hodnotou prvku na pole na daném indexu.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** %(x+1) = load i8* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.

castore	Uloží hodnotu typu `char` z vrcholu zásobníku do pole jehož reference se nachází na zásobníku pod hodnotou, do prvku na indexu, který je uložený pod referencí na pole.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** store i8 **H**, i8* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu uloží danou hodnotu.
checkcast	Provede přetypování dané hodnoty a požadovaný typ za předpokladu, že je to možné. Pokud ne, je vyhozena výjimka.	call **DT2** @checkcast(**DT1** %jmeno_promenne)	Imaginární knihovní funkce ověří možnost přetypování a v případě, že je to možné vrátí hodnotu dané proměnné jako požadovaný typ.
d2f	Převede hodnotu typu `double` z vrcholu zásobníku na typ `float`.	%x = fptrunc double %n to float	Uloží do pomocné proměnné %x oříznutou `float` hodnotu původního %n typu `double`.
d2i	Převede hodnotu typu `double` z vrcholu zásobníku na typ `int`.	%x = fptosi double %n to i32	Uloží do pomocné proměnné %x oříznutou celočíselnou hodnotu typu `int` původního `double` %n.
d2l	Převede hodnotu typu `double` z vrcholu zásobníku na typ `long`.	%x = fptosi double %n to i64	Uloží do pomocné proměnné `long` %x celočíselnou část hodnoty původního `double` %n.
dadd	Sečte dvě hodnoty typu `double` z vrcholu zásobníku a nahradí je výslednou hodnotou.	%x = add nsw double %y, %z	Uloží do pomocné proměnné %x výsledek součtu proměnných %y a %z.
daload	Načte hodnotu typu `double` z pole, jehož adresa je uložena na vrcholu zásobníku. Hodnota je čtena z indexu, který je uložený pod referencí na pole.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** %(x+1) = load double* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.
dastore	Uloží hodnotu typu `double` z vrcholu zásobníku do pole jehož reference se nachází na zásobníku pod hodnotou, do prvku na indexu, který je uložený pod referencí na pole.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** store double **H**, double* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu uloží danou hodnotu.
dcmpg	Porovná dvě hodnoty typu `double` z vrcholu zásobníku a nahradí je celočíselným výsledkem porovnání. Ten je roven 1 pokud je první hodnota vetší, 0 pokud jsou shodné, nebo -1 pokud je větší druhá z hodnot.	%x = fsub double %y, %z	Uloží do pomocné proměnné %x hodnotu rozdílu mezi danými proměnnými – typicky následuje porovnání této hodnoty proti nule, které se bude pro takový výsledek chovat stejně, jako pro výsledek instrukce *dcmpg*.

dcmpl	Porovná dvě hodnoty typu `double` z vrcholu zásobníku a nahradí je celočíselným výsledkem porovnání. Ten je roven 1 pokud je první hodnota menší, 0 pokud jsou shodné, nebo -1 pokud je menší druhá z hodnot.	%x = fsub double %y, %z %(x+1) = fmul %x, -1	Uloží do pomocné proměnné %x opačnou hodnotu rozdílu mezi danými proměnnými, tedy kladnou hodnotu pokud je první hodnota menší než druhá atd.
dconst_0	Vloží na vrchol zásobníku konstantu 0.0.	-	Nepřekládáme, konstanta 0.0 je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
dconst_1	Vloží na vrchol zásobníku konstantu 1.0.	-	Nepřekládáme, konstanta 1.0 je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
ddiv	Vybere z vrcholu zásobníku dvě hodnoty typu `double` a nahradí je jejich podílem	%x = fdiv double %y, %z	Uloží do pomocné proměnné %x podíl %y / %z.
dload **S** dload_0 dload_1 dload_2 dload_3	Nahraje na vrchol zásobníku hodnotu typu `double` uloženou v lokální proměnné z daného slotu (_číslo ≙ _slot).	-	Nepřekládáme, tato hodnota je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
dmul	Vybere z vrcholu zásobníku dvě hodnoty typu `double` a nahradí je výsledkem jejich násobení	%x = fmul double %y, %z	Uloží do pomocné proměnné %x výsledek násobení %y * %z.
dneg	Zneguje hodnotu typu `double` na vrcholu zásobníku.	%x = fsub double -0.0, %n	Uloží do pomocné proměnné %x negovanou hodnotu `double` %n.
drem	Vybere z vrcholu zásobníku dvě hodnoty typu `double` a nahradí je zbytkem po jejich vydělení. Znaménko výsledku je shodné se znaménkem dělitele.	%x = frem double %y, %z	Uloží do pomocné proměnné %x zbytek po dělení %y / %z. Znaménko výsledku je shodné se znaménkem dělitele.
dreturn	Vrátí `double` hodnotu z vrcholu zásobníku jako návratovou hodnotu a ukončí provádění probíhající funkce.	ret double **H**	Vrátí předanou `double` hodnotu jako návratovou a ukončí provádění probíhající funkce.
dstore **S** dstore_0 dstore_1 dstore_2 dstore_3	Nahraje hodnotu typu `double` z vrcholu zásobníku do lokální proměnné uložené v daném slotu (_číslo ≙ _slot).	store double %x, double* **S**, align *N*	Nahraje do lokální proměnné odpovídající danému slotu hodnotu typu `double` z pomocné proměnné %x.

<table>
<tr><td>dsub</td><td>Vybere z vrcholu zásobníku dvě hodnoty typu <code>double</code> a nahradí je jejich rozdílem.</td><td>%x = fsub double %y, %z</td><td>Uloží do pomocné proměnné %x rozdíl %y - %z.</td></tr>
<tr><td>dup</td><td>Duplikuje horní bajt programového zásobníku.</td><td rowspan="7">-</td><td rowspan="7">Duplikace bajtů zásobníku je simulována pouze interně, nepřekládáme.</td></tr>
<tr><td>dup_x1</td><td>Duplikuje horní bajt programového zásobníku 2 bajty pod vrchol zásobníku.</td></tr>
<tr><td>dup_x2</td><td>Duplikuje horní bajt programového zásobníku 3 bajty pod vrchol zásobníku.</td></tr>
<tr><td>dup2</td><td>Duplikuje horní 2 bajty programového zásobníku.</td></tr>
<tr><td>dup2_x1</td><td>Duplikuje horní 2 bajty programového zásobníku 3 bajty pod vrchol zásobníku.</td></tr>
<tr><td>dup2_x2</td><td>Duplikuje horní bajt programového zásobníku do 4. bajtu pod vrchol zásobníku.</td></tr>
<tr><td>f2d</td><td>Převede <code>float</code> hodnotu z vrcholu zásobníku na typ <code>double</code></td><td>%x = fpext float %y to double</td><td>Uloží do pomocné proměnné %x na 64b rozšířenou <code>double</code> hodnotu původního %y typu <code>float</code>.</td></tr>
<tr><td>f2i</td><td>Převede <code>float</code> hodnotu z vrcholu zásobníku na typ <code>int</code></td><td>%x = fptosi float %y to i32</td><td>Uloží do pomocné proměnné %x celočíselnou část hodnoty původního %y typu <code>float</code>.</td></tr>
<tr><td>f2l</td><td>Převede <code>float</code> hodnotu z vrcholu zásobníku na typ <code>long</code></td><td>%x = fptosi float %y to i64</td><td>Uloží do pomocné proměnné %x celočíselnou část hodnoty původního %y typu <code>float</code>, rozšířenou na 64b.</td></tr>
<tr><td>fadd</td><td>Sečte dvě hodnoty typu <code>float</code> z vrcholu zásobníku a nahradí je výslednou hodnotou.</td><td>%x = add nsw float %y, %z</td><td>Uloží do pomocné proměnné %x výsledek součtu proměnných %y a %z.</td></tr>
<tr><td>faload</td><td>Načte na vrchol zásobníku hodnotu typu <code>float</code> z pole</td><td>%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 I
%(x+1) = load float* %x, align N</td><td>První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.</td></tr>
</table>

fastore	Uloží hodnotu typu `float` z vrcholu zásobníku do pole	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** store float **H**, float* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu uloží danou hodnotu.
fcmpg	Porovná dvě hodnoty typu `float` z vrcholu zásobníku a nahradí je celočíselným výsledkem porovnání. Ten je roven 1 pokud je první hodnota vetší, 0 pokud jsou shodné, nebo -1 pokud je větší druhá z hodnot.	%x = fsub float %y, %z	Uloží do pomocné proměnné %x hodnotu rozdílu mezi danými proměnnými – typicky následuje porovnání této hodnoty proti nule, které se bude pro takový výsledek chovat stejně, jako pro výsledek instrukce *fcmpg*.
fcmpl	Porovná dvě hodnoty typu `float` z vrcholu zásobníku a nahradí je celočíselným výsledkem porovnání. Ten je roven 1 pokud je první hodnota menší, 0 pokud jsou shodné, nebo -1 pokud je menší druhá z hodnot.	%x = fsub float %y, %z %(x+1) = fmul %x, -1	Uloží do pomocné proměnné %x opačnou hodnotu rozdílu mezi danými proměnnými, tedy kladnou hodnotu pokud je první hodnota menší než druhá atd.
fconst_0	Vloží na vrchol zásobníku konstantu 0.0.	-	Nepřekládáme, konstanta 0.0 je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
fconst_1	Vloží na vrchol zásobníku konstantu 1.0.	-	Nepřekládáme, konstanta 1.0 je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
fconst_2	Vloží na vrchol zásobníku konstantu 2.0.	-	Nepřekládáme, konstanta 2.0 je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
fdiv	Vybere z vrcholu zásobníku dvě hodnoty typu `float` a nahradí je jejich podílem	%x = fdiv float %y, %z	Uloží do pomocné proměnné %x podíl %y / %z.
fload **S** fload_0 fload_1 fload_2 fload_3	Nahraje na vrchol zásobníku hodnotu typu `float` uloženou v lokální proměnné z daného slotu (_číslo ≙ _slot).	-	Nepřekládáme, tato hodnota je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
fmul	Vybere z vrcholu zásobníku dvě hodnoty typu `float` a nahradí je výsledkem jejich násobení	%x = fmul float %y, %z	Uloží do pomocné proměnné %x výsledek násobení %y * %z.

fneg	Zneguje hodnotu typu `float` na vrcholu zásobníku.	%nx = fsub float -0.0, %x	Uloží do pomocné proměnné %nx negovanou hodnotu `float` %x.
frem	Vybere z vrcholu zásobníku dvě hodnoty typu `float` a nahradí je zbytkem po jejich vydělení. Znaménko výsledku je shodné se znaménkem dělitele.	%x = frem float %y, %z	Uloží do pomocné proměnné %x zbytek po dělení %y / %z. Znaménko výsledku je shodné se znaménkem dělitele.
freturn	Vrátí `float` hodnotu z vrcholu zásobníku jako návratovou hodnotu a ukončí provádění probíhající funkce.	ret float **H**	Vrátí předanou `float` hodnotu jako návratovou a ukončí provádění probíhající funkce.
fstore **S** fstore_0 fstore_1 fstore_2 fstore_3	Nahraje hodnotu typu `float` z vrcholu zásobníku do lokální proměnné uložené v daném slotu (_číslo ≙ _slot).	store double %x, double* **S**, align *N*	Nahraje do lokální proměnné odpovídající danému slotu hodnotu typu `float` z pomocné proměnné %x.
fsub	Vybere z vrcholu zásobníku dvě hodnoty typu `float` a nahradí je jejich rozdílem.	%x = fsub float %y, %z	Uloží do pomocné proměnné %x rozdíl %y - %z.
getfield #index	Vymění referenci na objekt na vrcholu zásobníku za hodnotu jeho pole daného #indexem do tabulky konstant. Pokud index není jedno číslo, ale dvě, složí se za sebe jako (index1 « 8) + index2.	%x = getelementptr inbounds <typ_objektu> %jmeno_objektu, i32 0, i32 **I** %(x+1) = load **DT*** %x, align *N*	Získá do proměnné %(x+1) hodnotu pole předaného objektu.
getstatic #index	Vloží na vrchol zásobníku hodnotu statického pole třídy dané #indexem do tabulky konstant. Pokud index není jedno číslo, ale dvě, složí se za sebe jako (index1 « 8) + index2.	%x = load **DT*** @jmeno_pole, align *N*	Načte do proměnné %x hodnotu globální proměnné odpovídající požadovanému statickému poli (statické proměnné tříd LLVM nezná, bere je jako struktury).
goto	Přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2.	br label %y	Je proveden skok na zadané návěští.
i2b	Ořízne celočíselnou 32b hodnotu z vrcholu zásobníku na 8b	%x = trunc i32 %y to i8	Uloží do pomocné proměnné %x na 8 b oříznutou hodnotu původního 32b celého čísla.

i2c	Ořízne celočíselnou 32b hodnotu z vrcholu zásobníku na 8b	%x = trunc i32 %y to i8	Uloží do pomocné proměnné %x na 8 b oříznutou hodnotu původního 32b celého čísla.
i2d	Převede celočíselnou 32b hodnotu z vrcholu zásobníku na hodnotu typu `double`	%x = sitofp i32 %y to double	Uloží do pomocné proměnné %x na `double` převedenou hodnotu původního 32b celého čísla.
i2f	Převede celočíselnou 32b hodnotu z vrcholu zásobníku na hodnotu typu `float`	%x = sitofp i32 %y to float	Uloží do pomocné proměnné %x na `float` převedenou hodnotu původního 32b celého čísla.
i2l	Rozšíří celočíselnou 32b hodnotu z vrcholu zásobníku na 64b	%x = sext i32 %y to i64	Uloží do pomocné proměnné %x na 64 b rozšířenou hodnotu původního 32b celého čísla.
i2s	Ořízne celočíselnou 32b hodnotu z vrcholu zásobníku na 16b	%x = trunc i32 %y to i16	Uloží do pomocné proměnné %x na 16 b oříznutou hodnotu původního 32b celého čísla.
iadd	Sečte dvě celočíselné hodnoty z vrcholu zásobníku a nahradí je výslednou hodnotou.	%x = add nsw i32 %y, %z	Uloží do pomocné proměnné %x výsledek součtu proměnných %y a %z.
iaload	Načte na vrchol zásobníku hodnotu typu `int` z pole.	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** %(x+1) = load i32* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.
iand	Provede bitový součin dvou celočíselných hodnot z vrcholu zásobníku a nahradí je výslednou hodnotou.	%x = and i32 %y, %z	Uloží do pomocné proměnné %x výsledek bitového součinu proměnných %y a %z.
iastore	Uloží hodnotu typu `int` z vrcholu zásobníku do pole	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** store i32 **H** i32* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu vloží předanou hodnotu.
iconst_m1	Vloží na vrchol zásobníku konstantu -1.	-	Nepřekládáme, hodnota je pouze interně vložena na zásobník s operandy pro další použití
iconst_0	Vloží na vrchol zásobníku konstantu 0.		
iconst_1	Vloží na vrchol zásobníku konstantu 1.		
iconst_2	Vloží na vrchol zásobníku konstantu 2.		

iconst_3	Vloží na vrchol zásobníku konstantu 3.	-	Nepřekládáme, hodnota je pouze interně vložena na zásobník s operandy pro další použití
iconst_4	Vloží na vrchol zásobníku konstantu 4.		
iconst_5	Vloží na vrchol zásobníku konstantu 5.		
idiv	Vybere z vrcholu zásobníku dvě hodnoty typu `int` a nahradí je jejich podílem	%x = sdiv i32 %y, %z	Uloží do pomocné proměnné %x podíl %y / %z.
if_acmpeq	Pokud jsou dvě reference na vrcholu zásobníku shodné, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp eq **DT** **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud se reference rovnají, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
if_acmpne	Pokud jsou dvě reference na vrcholu zásobníku různé, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp ne **DT** **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud se reference nerovnají, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
if_icmpeq	Pokud jsou dvě celá čísla z vrcholu zásobníku shodná, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp eq i32 **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud se čísla rovnají, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)

if_icmpge	Pokud je první ze dvou celých čísel na vrcholu zásobníku větší nebo rovno tomu druhému, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp sge i32 **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel větší nebo rovno tomu druhému, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
if_icmpgt	Pokud je první ze dvou celých čísel na vrcholu zásobníku větší než to druhé, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp sgt i32 **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel větší než to druhé, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
if_icmple	Pokud je první ze dvou celých čísel na vrcholu zásobníku menší nebo rovno tomu druhému, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp sle i32 **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel menší nebo rovno tomu druhému, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
if_icmplt	Pokud je první ze dvou celých čísel na vrcholu zásobníku menší než to druhé, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp slt i32 **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel menší než to druhé, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)

if_icmpne	Pokud jsou dvě celá čísla z vrcholu zásobníku různá, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp ne i32 **H1**, **H2** br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud se čísla nerovnají, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
ifeq	Pokud je celé číslo z vrcholu zásobníku rovno 0, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp eq i32 **H1**, 0 br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud se **H1** rovná 0, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
ifge	Pokud je celé číslo z vrcholu zásobníku větší nebo rovno 0, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp sge i32 **H1**, 0 br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel větší nebo rovno 0, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
ifgt	Pokud je celé číslo z vrcholu zásobníku větší než 0, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp sgt i32 **H1**, 0 br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel větší než 0, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)

ifle	Pokud je celé číslo z vrcholu zásobníku menší nebo rovno 0, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp sle i32 **H1**, 0 br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel menší nebo rovno 0, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
iflt	Pokud je celé číslo z vrcholu zásobníku menší než 0, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp slt i32 **H1**, 0 br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je první z čísel menší než 0, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
ifne	Pokud je celé číslo z vrcholu zásobníku různé od 0, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp ne i32 **H1**, 0 br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je hodnota1 různá od 0, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
ifnonnull	Pokud je reference z vrcholu zásobníku různá od `null`, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp ne **DT H1**, null br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je hodnota1 různá od `null`, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)

ifnull	Pokud je reference z vrcholu zásobníku rovna hodnotě `null`, přejde se na instrukci na adrese určené parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Pokud podmínka neplatí, pokračuje se následující instrukcí.	%x = icmp eq **DT H1**, null br i1 %x, label %y, label %(x+1) <label>:(x+1) ... <label>:y	Do hodnoty %x je uložena boolovská hodnota `true`, pokud je hodnota1 rovna hodnotě `null`, a následně podle ní provedena instrukce skoku na zadané návěští, nebo, pokud podmínka neplatí, je provedena následující instrukce (skok na návěští umístěné přímo za instrukcí podmíněného skoku)
iinc **S**, **H**	Přičte k lokální proměnné v daném slotu znaménkovou hodnotu **H**, jejíž rozsah je omezen na 8b.	%x = load i32 %jmeno_promenne %(x+1) = add nsw i32 %x, **H** store i32 %(x+1), i32* %jmeno_promenne	Inkrementuje lokální proměnnou odpovídající danému slotu o **H**.
iload **S** iload_0 iload_1 iload_2 iload_3	Nahraje na vrchol zásobníku hodnotu typu `int` uloženou v lokální proměnné z daného slotu (_číslo ≙ _slot).	-	Nepřekládáme, tato hodnota je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
imul	Vybere z vrcholu zásobníku dvě hodnoty typu `int` a nahradí je výsledkem jejich násobení	%x = mul nsw i32 %y, %z	Uloží do pomocné proměnné %x výsledek násobení %y * %z.
ineg	Zneguje hodnotu typu `int` na vrcholu zásobníku.	%nx = sub i32 0, %x	Uloží do pomocné proměnné %nx negovanou hodnotu `int` %x.
instanceof	Určí jestli předaná hodnota (objekt) je instancí třídy identifikované indexem do tabulky konstant (indexbyte1 « 8 + indexbyte2)	%x = call i32 @instanceof(**H**, **DT**)	Je volána imaginární knihovní funkce, která vloží do nové pomocné proměnné hodnotu 0, pokud daná hodnota náleží k předanému datovému typu.
invokedyna-mic	Zavolá dynamickou metodu a výsledek vloží na zásobník (pokud není návratového typu void). Metoda je identifikována indexem do tabulky konstant. (indexbyte1 « 8 + indexbyte2)	[%x =] call **DT** @jmeno_metody ([**DT H**,?] *)	Zavolá danou metodu.

invokeinter-face	Zavolá metodu daného rozhraní a výsledek vloží na zásobník (pokud není návratového typu void). Metoda je identifikována indexem do tabulky konstant. (indexbyte1 « 8 + indexbyte2	[%x =] call **DT** @jmeno_metody ([**DT** **H**,?] *)	Zavolá danou metodu.
invokestatic	Zavolá statickou metodu a výsledek vloží na zásobník (pokud není návratového typu void). Metoda je identifikována indexem do tabulky konstant. (indexbyte1 « 8 + indexbyte2		
invokespecial	Zavolá metodu nad objektem ze zásobníku a výsledek vloží na zásobník (pokud není návratového typu void). Metoda je identifikována indexem do tabulky konstant. (indexbyte1 « 8 + indexbyte2)		
invokevirtual			
ior	Provede bitový součet dvou celočíselných hodnot z vrcholu zásobníku a nahradí je výslednou hodnotou.	%x = or i32 %y, %z	Uloží do pomocné proměnné %x výsledek bitového součtu proměnných %y a %z.
irem	Vybere z vrcholu zásobníku dvě hodnoty typu `int` a nahradí je zbytkem po jejich vydělení. Znaménko výsledku je shodné se znaménkem dělitele.	%x = srem i32 %y, %z	Uloží do pomocné proměnné %x zbytek po dělení %y / %z. Znaménko výsledku je shodné se znaménkem dělitele.
ireturn	Vrátí `int` hodnotu z vrcholu zásobníku jako návratovou hodnotu a ukončí provádění probíhající funkce.	ret i32 **H**	Vrátí předanou celočíselnou hodnotu jako návratovou a ukončí provádění probíhající funkce.
ishl	Provede bitový posun vlevo celočíselné hodnoty z vrcholu zásobníku o počet bitů uložený na zásobníku pod posouvaným číslem.	%x = shl nsw i32 %y, **P**	Uloží do pomocné proměnné %x výsledek bitového posunu proměnné %y vlevo o **P** bitů.
ishr	Provede aritmetický bitový posun vpravo celočíselné hodnoty z vrcholu zásobníku o počet bitů uložený na zásobníku pod posouvaným číslem.	%x = ashr i32 %y, **P**	Uloží do pomocné proměnné %x výsledek aritmetického bitového posunu proměnné %y vpravo o **P** bitů.

istore **S**	Nahraje hodnotu typu `int` z vrcholu zásobníku do lokální proměnné uložené v daném slotu (_číslo ≙ _slot).	store i32 %x, i32* **S**, align *N*	Nahraje do lokální proměnné odpovídající danému slotu hodnotu typu `int` z pomocné proměnné %x.
istore_0			
istore_1			
istore_2			
istore_3			
isub	Vybere z vrcholu zásobníku dvě hodnoty typu `int` a nahradí je jejich rozdílem.	%x = sub nsw i32 %y, %z	Uloží do pomocné proměnné %x rozdíl %y - %z.
iushr	Provede logický bitový posun vpravo celočíselné hodnoty z vrcholu zásobníku o počet bitů uložený na zásobníku pod posouvaným číslem.	%x = lshr i32 %y, **P**	Uloží do pomocné proměnné %x výsledek logického bitového posunu proměnné %y vpravo o **P** bitů.
ixor	Provede exkluzivní bitový součin dvou celočíselných hodnot z vrcholu zásobníku a nahradí je výslednou hodnotou.	%x = xor i32 %y, %z	Uloží do pomocné proměnné %x výsledek exkluzivního bitového součinu proměnných %y a %z.
jsr	Provede skok na adresu danou určenou parametry. Pokud je parametr jeden, skáče se přímo na dané číslo instrukce, pokud jsou dva složí se za sebe jako (adresa1 « 8) + adresa2. Na zásobník je umístěna návratová adresa.	call void @jsr() br label %x	Imaginární funkce uloží návratovou adresu a následně je proveden skok na požadovanou adresu.
jsr_w	Provede skok na adresu danou určenou parametry. Předpokládají se 4 parametry, které se případně zepředu doplní nulami. Pro určení cíle skoku se složí za sebe jako (adresa1 « 24) + (adresa2 « 16) + (adresa3 « 8) + adresa4. Na zásobník je umístěna návratová adresa.		
l2d	Převede celočíselnou 64b hodnotu z vrcholu zásobníku na hodnotu typu `double`	%x = sitofp i64 %y to double	Uloží do pomocné proměnné %x na `double` převedenou hodnotu původního 64b celého čísla.
l2f	Převede celočíselnou 64b hodnotu z vrcholu zásobníku na hodnotu typu `float`	%x = sitofp i64 %y to float	Uloží do pomocné proměnné %x na `float` převedenou hodnotu původního 64b celého čísla.

<table>
<tr><td>l2i</td><td>Převede ořízne celočíselnou 64b hodnotu z vrcholu zásobníku na 32b.</td><td>%x = trunc i64 %y to i32</td><td>Uloží do pomocné proměnné %x na 32b oříznutou hodnotu původního 64b celého čísla.</td></tr>
<tr><td>ladd</td><td>Sečte dvě celočíselné hodnoty z vrcholu zásobníku a nahradí je výslednou hodnotou.</td><td>%x = add nsw i32 %y, %z</td><td>Uloží do pomocné proměnné %x výsledek součtu proměnných %y a %z.</td></tr>
<tr><td>laload</td><td>Uloží na vrchol zásobníku hodnotu typu <code>long</code> z pole</td><td>%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 I
%(x+1) = load i64* %x, align N</td><td>První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.</td></tr>
<tr><td>land</td><td>Provede bitový součin dvou celočíselných hodnot z vrcholu zásobníku a nahradí je výslednou hodnotou.</td><td>%x = and i64 %y, %z</td><td>Uloží do pomocné proměnné %x výsledek bitového součinu proměnných %y a %z.</td></tr>
<tr><td>lastore</td><td>Uloží hodnotu typu <code>long</code> z vrcholu zásobníku do pole</td><td>%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 I
store i64 H, i64* %x, align N</td><td>První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu vloží danou hodnotu.</td></tr>
<tr><td>lcmp</td><td>Porovná dvě hodnoty typu <code>long</code> z vrcholu zásobníku a nahradí je celočíselným výsledkem porovnání. Ten je roven 1 pokud je první hodnota větší, 0 pokud jsou shodné, nebo -1 pokud je větší druhá z hodnot.</td><td>%x = sub nsw i64 %y, %z</td><td>Uloží do pomocné proměnné %x hodnotu rozdílu mezi danými proměnnými – typicky následuje porovnání této hodnoty proti nule, které se bude pro takový výsledek chovat stejně, jako pro výsledek instrukce lcmp.</td></tr>
<tr><td>lconst_0</td><td>Vloží na vrchol zásobníku konstantu 0L.</td><td rowspan="2">-</td><td rowspan="2">Nepřekládáme, hodnota je pouze interně vložena na zásobník s operandy pro další použití</td></tr>
<tr><td>lconst_1</td><td>Vloží na vrchol zásobníku konstantu 1L.</td></tr>
<tr><td>ldc #index</td><td>Nahraje na zásobník konstantu z daného #indexu v tabulce konstant.</td><td rowspan="2">-</td><td rowspan="2">Nepřekládáme, hodnota je pouze interně vložena na zásobník s operandy pro další použití</td></tr>
<tr><td>ldc_w #index</td><td>Nahraje na zásobník konstantu z daného #indexu v tabulce konstant. Pokud index není jedno číslo, ale dvě, složí se za sebe jako (index1 « 8) + index2.</td></tr>
</table>

ldc2_w #index	Nahraje na zásobník konstantu typu `double`, nebo `long` z daného #indexu v tabulce konstant. Pokud index není jedno číslo, ale dvě, složí se za sebe jako (index1 « 8) + index2.	-	Nepřekládáme, hodnota je pouze interně vložena na zásobník s operandy pro další použití
ldiv	Vybere z vrcholu zásobníku dvě hodnoty typu `long` a nahradí je jejich podílem	%x = sdiv i64 %y, %z	Uloží do pomocné proměnné %x podíl %y / %z.
lload **S**	Nahraje na vrchol zásobníku hodnotu typu `long` uloženou v lokální proměnné z daného slotu (_číslo ≙ _slot).	-	Nepřekládáme, tato hodnota je pouze interně vložena na zásobník s operandy k pozdějšímu použití.
lload_0			
lload_1			
lload_2			
lload_3			
lmul	Vybere z vrcholu zásobníku dvě hodnoty typu `long` a nahradí je výsledkem jejich násobení	%x = mul nsw i64 %y, %z	Uloží do pomocné proměnné %x výsledek násobení %y * %z.
lneg	Zneguje hodnotu typu `long` na vrcholu zásobníku.	%nx = sub i64 0, %x	Uloží do pomocné proměnné %nx negovanou hodnotu `int` %x.
lookupswitch	Provede skok na adresu z tabulky podle hodnoty z vrcholu zásobníku.	switch **DT H**, label %default [**DT H1**, label %case1 **DT H2**, label %case2 ...]	Provede skok na adresu z tabulky podle předané hodnoty.
lor	Provede bitový součet dvou celočíselných hodnot z vrcholu zásobníku a nahradí je výslednou hodnotou.	%x = or i64 %y, %z	Uloží do pomocné proměnné %x výsledek bitového součtu proměnných %y a %z.
lrem	Vybere z vrcholu zásobníku dvě hodnoty typu `long` a nahradí je zbytkem po jejich vydělení. Znaménko výsledku je shodné se znaménkem dělitele.	%x = srem i64 %y, %z	Uloží do pomocné proměnné %x zbytek po dělení %y / %z. Znaménko výsledku je shodné se znaménkem dělitele.
lreturn	Vrátí `long` hodnotu z vrcholu zásobníku jako návratovou hodnotu a ukončí provádění probíhající funkce.	ret i64 **H**	Vrátí předanou celočíselnou hodnotu jako návratovou a ukončí provádění probíhající funkce.

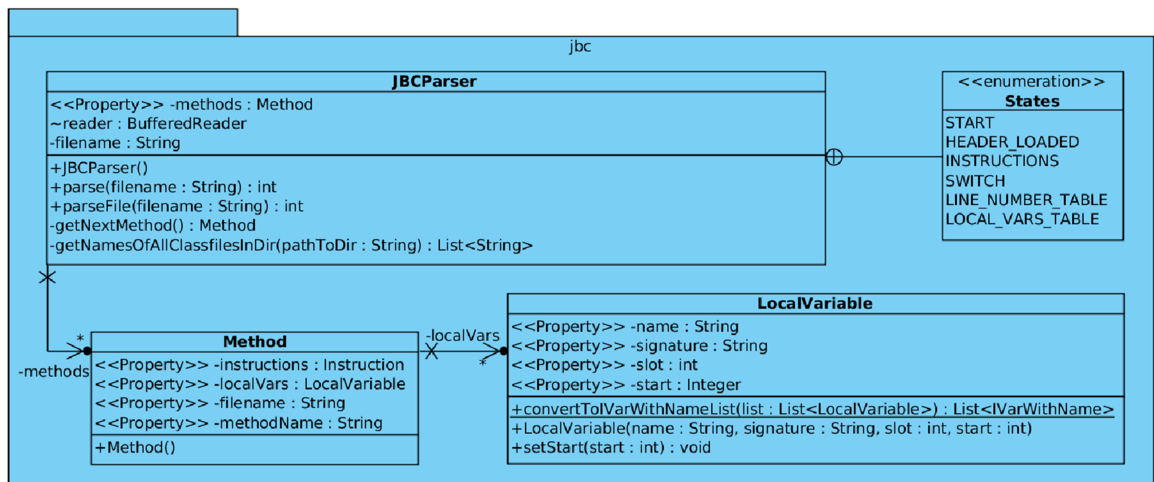
<table>
<tr><td>lshl</td><td>Provede bitový posun vlevo celočíselné hodnoty z vrcholu zásobníku o počet bitů uložený na zásobníku pod posouvaným číslem.</td><td>%x = shl nsw i64 %y, P</td><td>Uloží do pomocné proměnné %x výsledek bitového posunu proměnné %y vlevo o P bitů.</td></tr>
<tr><td>lshr</td><td>Provede aritmetický bitový posun vpravo celočíselné hodnoty z vrcholu zásobníku o počet bitů uložený na zásobníku pod posouvaným číslem.</td><td>%x = ashr i64 %y, P</td><td>Uloží do pomocné proměnné %x výsledek aritmetického bitového posunu proměnné %y vpravo o P bitů.</td></tr>
<tr><td>lstore S</td><td rowspan="5">Nahraje hodnotu typu <code>long</code> z vrcholu zásobníku do lokální proměnné uložené v daném slotu (_číslo ≙ _slot).</td><td rowspan="5">store i64 %x, i64* S, align N</td><td rowspan="5">Nahraje do lokální proměnné odpovídající danému slotu hodnotu typu <code>long</code> z pomocné proměnné %x.</td></tr>
<tr><td>lstore_0</td></tr>
<tr><td>lstore_1</td></tr>
<tr><td>lstore_2</td></tr>
<tr><td>lstore_3</td></tr>
<tr><td>lsub</td><td>Vybere z vrcholu zásobníku dvě hodnoty typu <code>long</code> a nahradí je jejich rozdílem.</td><td>%x = sub nsw i64 %y, %z</td><td>Uloží do pomocné proměnné %x rozdíl %y - %z.</td></tr>
<tr><td>lushr</td><td>Provede logický bitový posun vpravo celočíselné hodnoty z vrcholu zásobníku o počet bitů uložený na zásobníku pod posouvaným číslem.</td><td>%x = lshr i64 %y, P</td><td>Uloží do pomocné proměnné %x výsledek logického bitového posunu proměnné %y vpravo o P bitů.</td></tr>
<tr><td>lxor</td><td>Provede exkluzivní bitový součin dvou celočíselných hodnot z vrcholu zásobníku a nahradí je výslednou hodnotou.</td><td>%x = xor i64 %y, %z</td><td>Uloží do pomocné proměnné %x výsledek exkluzivního bitového součinu proměnných %y a %z.</td></tr>
<tr><td>monitorenter</td><td>Zabere zámek nad objektem, jehož reference je uložena n vrcholu zásobníku.</td><td>call void @monitorenter()</td><td>Imaginární funkce simulující instrukci monitorenter.</td></tr>
<tr><td>monitorexit</td><td>Uvolní zámek nad objektem, jehož reference je uložena n vrcholu zásobníku.</td><td>call void @monitorexit()</td><td>Imaginární funkce simulující instrukci monitorexit.</td></tr>
<tr><td>multianew-array P</td><td>Vytvoří nové pole o počtu dimenzí P komponent typu daného indexem do tabulky konstant o počtech prvků zadaném P hodnotami z vrcholu zásobníku s operandy.</td><td>%x = alloca ..[P2 x [P1 x DT*]].., align 16</td><td>Alokuje prostor pro nové pole a referenci na něj uloží do lokální proměnné.</td></tr>
<tr><td>new</td><td>Vytvoří nový objekt třídy dané indexem do tabulky konstant (indexbyte1 « 8 + indexbyte2)</td><td>%x = alloca DT, align 8</td><td>Alokuje prostor pro daný objekt.</td></tr>
</table>

newarray type	Vytvoří nové pole prvků typu `type` jejichž počet je dán hodnotou na vrcholu zásobníku.	%x = alloca [**P** x **DT**], align 16	Alokuje prostor pro nové pole o **P** prvcích typu **DT**
nop	Žádná operace.	-	Pouze případná modifikace interního zásobníku.
pop	Odstraní horní položku (1B) z programového zásobníku		
pop2	Odstraní horní dva bajty z programového zásobníku		
putfield #index	Nastaví položku třídy, danou #indexem do tabulky konstant. Pokud index není jedno číslo, ale dvě, složí se za sebe jako (index1 « 8) + index2.	%x = getelementptr inbounds <typ_objektu> %jmeno_objektu, i32 0, i32 **I** store **DT H**, **DT** %x, align *N*	Uloží do pole předaného objektu hodnotu **H**.
putstatic #index	Vloží na vrchol zásobníku hodnotu statického pole třídy dané #indexem do tabulky konstant. Pokud index není jedno číslo, ale dvě, složí se za sebe jako (index1 « 8) + index2.	store **DT H**, **DT*** @jmeno_pole, align *N*	Uloží do globální proměnné odpovídající požadovanému statickému poli hodnotu **H** (statické proměnné tříd LLVM nezná, bere je jako struktury).
ret **S**	Vykonávání programu bude pokračovat adresou uloženou v lokální proměnné v daném slotu.	%x = load %jmeno_promene, align *N* br label %x	Provede se skok na adresu uloženou v lokální proměnné odpovídající původnímu slotu.
return	Ukončí provádění probíhající funkce a vrátí vykonávání programu do volajícího bloku.	ret void	Ukončí provádění probíhající funkce a vrátí vykonávání programu do volajícího bloku.
saload	Vloží na vrchol zásobníku hodnotu typu `short` z pole	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** %(x+1) = load i16* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak jeho hodnotu uloží do pomocné proměnné pro další použití.
sastore	Uloží hodnotu typu `short` z vrcholu zásobníku do pole	%x = getelementptr inbounds <typ_pole> %jmeno_pole, i32 0, i32 **I** store i16 **H**, i16* %x, align *N*	První instrukce získá ukazatel na hledaný prvek pole, druhá pak na získanou adresu vloží danou hodnotu.
sipush <byte>	Vloží na vrchol zásobníku zadaný byte jako typ `short`.	-	Nepřekládáme, hodnota je interně načtena na simulovaný zásobník s operandy pro pozdější použití.

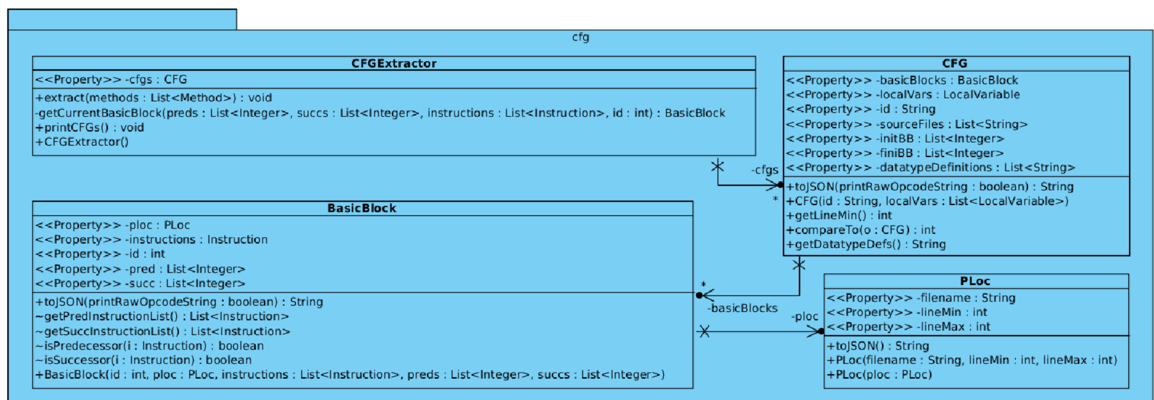
swap	Vymění dva horní bajty programového zásobníku, ani jedna z nich však nesmí být typu `double` nebo `long`	-	Nepřekládáme na instrukce, je pouze provedena odpovídající modifikace simulovaného zásobníku s operandy
tableswitch `min` to `max`: **H1** : **OI**	Provede skok na adresu z tabulky podle hodnoty na vrcholu zásobníku, která patří do uvedeného rozmezí.	switch **DT H**, label %default [**DT H1**, label %case1 **DT H2**, label %case2 ...]	Provede skok na adresu z tabulky podle předané hodnoty.
wide	Předpona k ostatním instrukcím, značící, že parametry nejsou dlouhé 8b, ale 16b.	-	Není potřeba překládat.

Příloha D

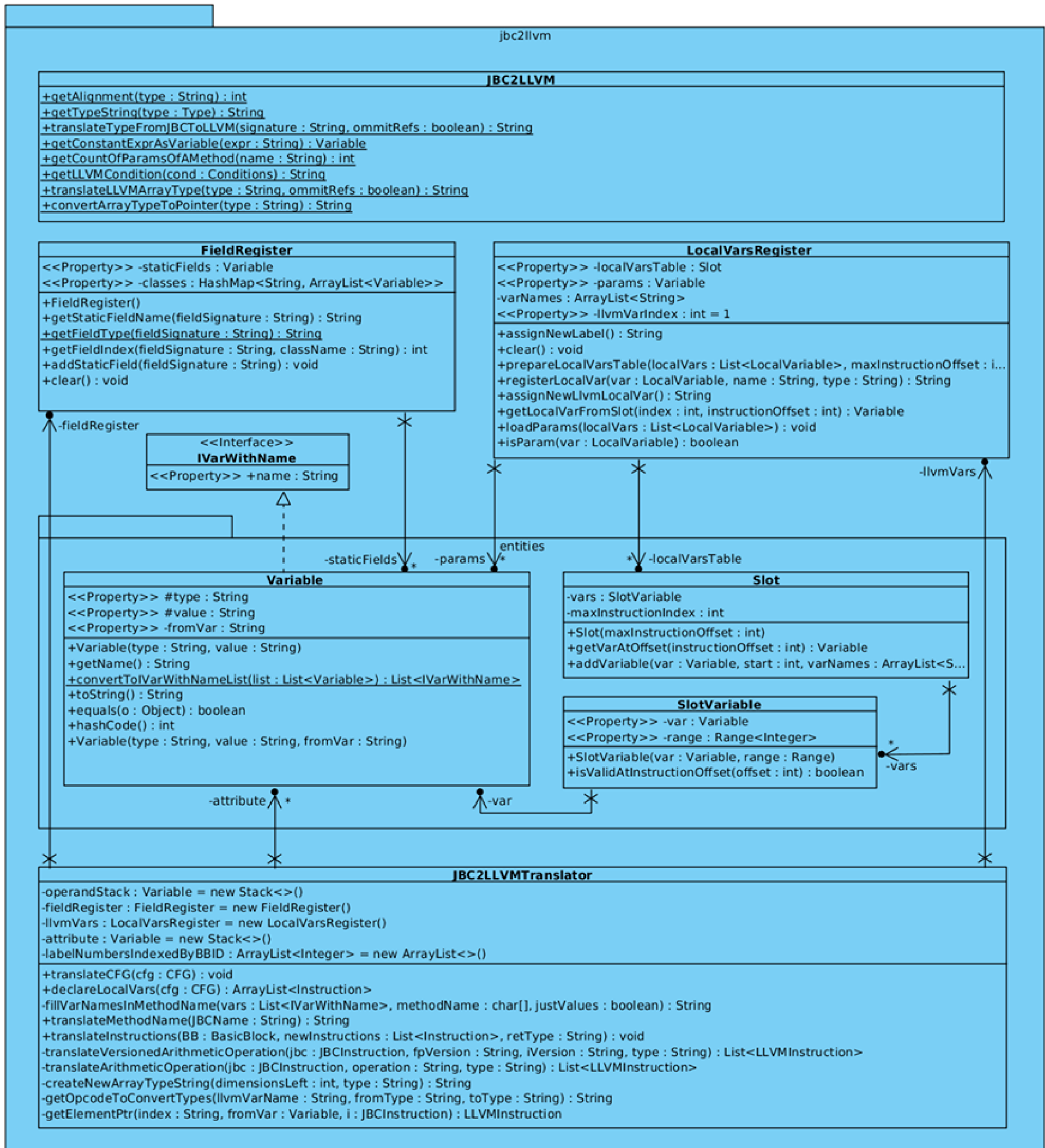
Diagramy tříd



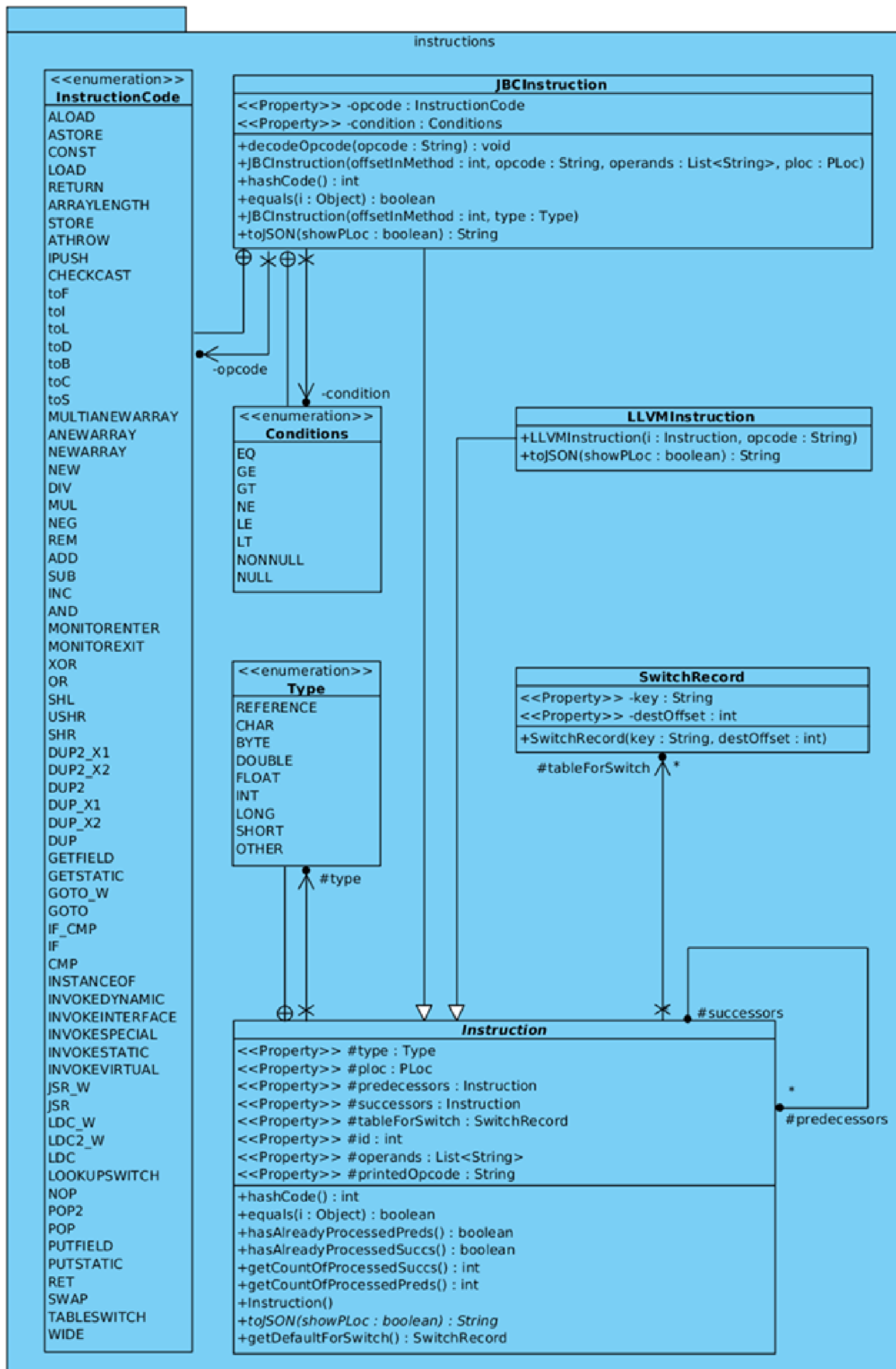
Obrázek D.1: Podrobné třídní schéma balíčku `jbc`.



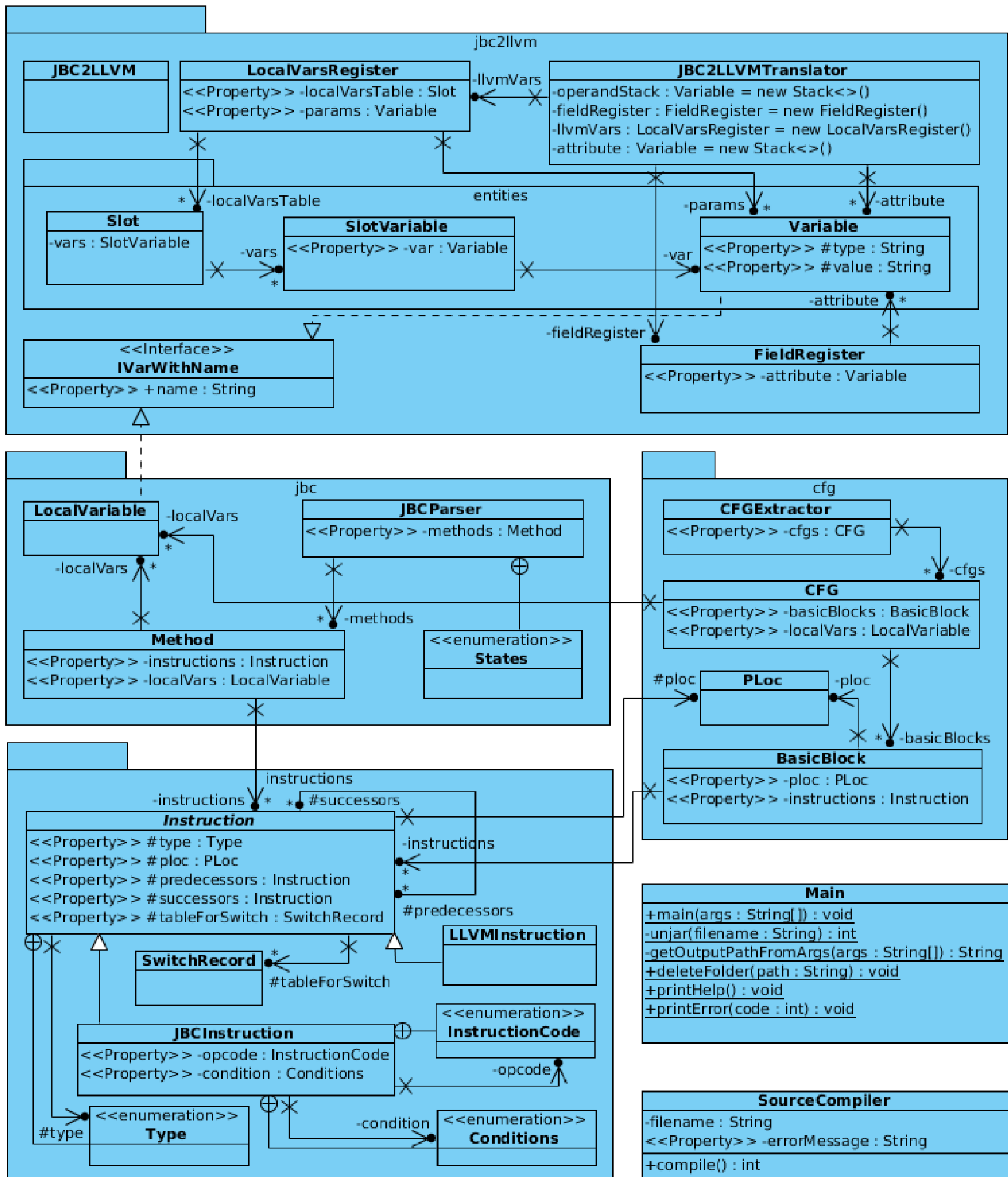
Obrázek D.2: Podrobné třídní schéma balíčku `cfg`.



Obrázek D.3: Podrobné třídní schéma balíčku `jbc2llvm`.



Obrázek D.4: Podrobné třídní schéma balíčku `instructions`.



Obrázek D.5: Schéma propojení jednotlivých tříd a balíčků.