VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERICKÝ ZPĚTNÝ PŘEKLAD PROGRAMŮ V BAJTKÓDU DO VYŠŠÍ FORMY REPREZENTACE

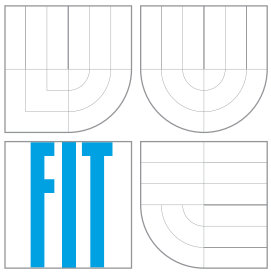
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MRÁZEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERICKÝ ZPĚTNÝ PŘEKLAD PROGRAMŮ V BAJTKÓDU DO VYŠŠÍ FORMY REPREZENTACE

GENERIC DECOMPILATION OF BYTECODE INTO HIGH-LEVEL REPRESENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR MRÁZEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAKUB KŘOUSTEK
SUPERVISOR

BRNO 2013

Abstrakt

Práce popisuje postupy a principy zpětného překladu. Uvádí základní informace o zpětném inženýrství a jeho užití v oboru softwatového inženýrství i inženýrství obecně. Dále představuje zpětný překladač vyvíjený v rámci projektu Lissom na FIT VUT v Brně. Cílem práce je navrhnout a implementovat rekonfigurovatelný zpětný překladač bajtkódu navazující na tento překladač.

Abstract

The work describes methods and principles of decompilation, basic information about reverse engineering and its use in both software engineering and engineering in general. Furthermore, it introduces the decompiler developed within the Lissom project at BUT FIT. The goal of the work is to design and implement a retargetable decompiler for bytecode, which extends the original decompiler.

Klíčová slova

zpětné inženýrství, Java, Dalvik, zpětný překlad, bajtkód, Lissom, LLVM IR

Keywords

reverse engineering, Java, Dalvik, decompilation, bytecode, Lissom, LLVM IR

Citace

Petr Mrázek: Generický zpětný překlad programů
v bajtkódu do vyšší formy reprezentace, diplomová práce, Brno, FIT VUT v Brně, 2013

Generický zpětný překlad programů v bajtkódu do vyšší formy reprezentace

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením pana Ing. Jakuba Křoustka a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Mrázek
29. května 2013

Poděkování

Rád bych poděkoval mému vedoucímu Ing. Jakubu Křoustkovi za konzultace a poskytnuté rady.

Obsah

Kapitola 1

Úvod

Zpětný překlad (angl. *decompilation*) je forma zpětného inženýrství aplikovaná na již přeložený počítačový program. Zpravidla je použita, když je původní zdrojový kód programu ztracen, nebo z nějakého důvodu nedostupný. K dekompilaci se používají zpětné překladače neboli dekompilátory, což jsou programy převádějící nízkoúrovňovou reprezentaci aplikace (např. ve strojovém kódu konkrétního procesoru nebo bajtkódu) zpět do vysokoúrovňové, čitelné a ideálně i okamžitě opět přeložitelné formy.

Bajtkód je forma zápisu počítačového programu účelem podobná strojovému kódu, ale namísto provádění přímo procesorem je buď interpretována, nebo za běhu programu překládána do strojového kódu. Z toho plyne nutná vyšší úroveň abstrakce této reprezentace. Její výhodou je nezávislost na konkrétní architektuře. Nevýhodou pak je zjednodušení zpětného inženýrství, protože lze využít dodatečné informace v bajtkódu obsažené.

Cílem práce je navrhnout a implementovat generický zpětný překladač bajtkódu, v budoucnu snadno rozšiřitelný o podporu dalších platforem používajících bajtkód. Minimálně vznikne sada obecných a znovu použitelných algoritmů pro zpracování bajtkódu, umožňující snadnou implementaci dalších vstupních i cílových jazyků.

Práce vzniká jako součást výzkumného projektu Lissom na Fakultě informačních technologií VUT v Brně [10].

Struktura této práce je následující. Kapitola 2 pojednává o zpětném inženýrství v oboru informačních technologií i mimo něj. První část kapitoly 3 uvádí základy jazyka Java, ze kterého vycházejí obě platformy vybrané pro ověření navrženého konceptu zpětného překladače. V druhé části kapitoly jsou blíže popsána specifika platformy JVM[1]. Primárním zaměřením kapitoly je popis technik zpětného překladu a existujících nástroje pro zpětný překlad bajtkódu. Kapitola 4 podobným způsobem popisuje platformu DVM[2], její rozdíly a podobnosti v porovnání s JVM. V kapitole 5 se lze seznámit se zpětným překladačem vyvíjeným v rámci projektu Lissom, na který bude výsledný zpětný překladač navázán. Kapitola 7 je zaměřena na návrh a započatou implementaci přední části zpětného překladače.

[1] Java Virtual Machine
[2] Dalvik Virtual Machine

Kapitola 2

Zpětné inženýrství

Zpětné inženýrství je definováno jako proces analýzy předmětného systému s cílem identifikovat komponenty systému a jejich vzájemné vazby a/nebo vytvořit reprezentaci systému v jiné formě nebo na vyšší úrovni abstrakce [24]. Je opakem inženýrství, které je normálním postupem od návrhu po realizaci.

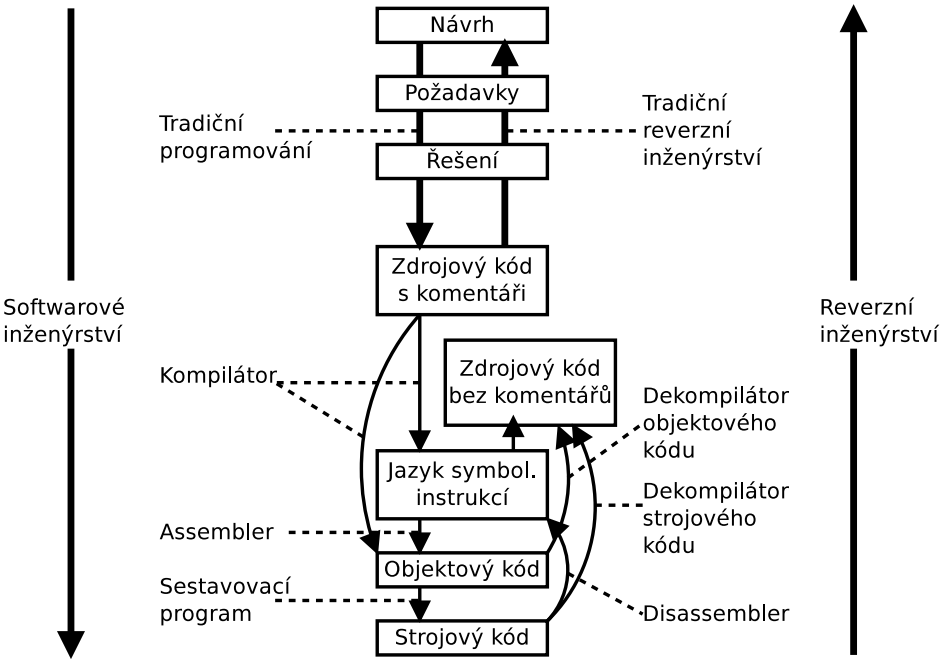
Podle této definice se zpětné inženýrství zabývá pouze zkoumáním daného předmětu. Zde se může jednat jak o reálný, fyzický předmět, tak virtuální systém – software. Postupy jsou v obou případech rozdílné, ale cílem je vždy odkrýt princip fungování zkoumaného předmětu. Informace zde uvedené jsem získal ze zdrojů [9, 8]

Zpětné inženýrství může mít mnoho různých motivací. Jejich úplný výčet by byl zcela nemožný, proto jich uvedu pouze několik:

- Interoperabilita – pokud je zapotřebí, aby nový systém spolupracoval se stávajícím, který ovšem postrádá dokumentaci a/nebo rozumná rozhraní pro integraci s jinými systémy.
- Vojenská a průmyslová špionáž – např. okopírování Amerických bombardérů B-29 sovětským svazem pod názvem Tupolev Tu-4 [2].
- Zestárnutí – zastaralé a vyřazené systémy je často potřeba emulovat nebo nahradit. V takovém případě je nutné porozumět původnímu systému. Viz např. emulátory starých herních konzolí založené na analýze použitých čipů [22].
- Bezpečnostní analýza – zde lze uvažovat např. prolamování šifer, analýzu počítačových virů nebo nezávislou analýzu bezpečnosti konkrétního výrobku.
- Prolomení ochran proti kopírování (tzv. *cracking*).
- Analýza síťových protokolů a jejich reimplementace. Příkladem může být projekt Samba, který implementuje síťové protokoly používané v OS Windows [26].
- Obnova zapomenutého výrobního postupu, v oblasti IT pak např. obnova ztraceného zdrojového kódu.
- Obnova chybějící dokumentace (tzv. redokumentace). tohoto přístupu se může použít i v rámci inženýrství, pokud se dokumentace nevytváří průběžně.

2.1 Softwarové zpětné inženýrství

Vztah softwarového inženýrství a zpětného inženýrství lze nejlépe popsat pomocí diagramu 2.1. Platí, že v každém kroku inženýrství dochází k rozkladu na menší části a ztrátě abstrakce – od návrhu po program přeložený do cílového jazyka. Zpětné inženýrství jde pak opačným směrem a naráží na problém, protože musí nějak doplnit ztracenou abstrakci. Kroky těchto procesů jsou navzájem opačné. Nástroje používané ke zpětnému inženýrství softwaru lze v zásadě rozdělit na ladicí nástroje (angl. *debuggers*), disassemblery a zpětné překladače.



Obrázek 2.1: Vztah zpětného inženýrství a softwarového inženýrství [9] a [8].

Ladicí nástroje

Ladicí nástroje (často kombinované s disassemblerem pro zobrazení kódu právě laděné aplikace) nejsou samozřejmě nijak specifické pro zpětné inženýrství a jsou daleko častěji využity v rámci normálního, dopředného inženýrství. Tyto nástroje ovšem jdou použít k dynamické analýze zkoumaného kódu a způsobu jakým např. manipuluje s pamětí nebo komunikuje s okolím. Ladicí nástroje totiž zpravidla umožňují kdykoliv (i automaticky v závislosti na různých podmínkách) zastavit běh aplikace a zkoumat stav (virtuálního) stroje, na kterém běží.

Disassemblery

Disassemblery, jak název napovídá, „rozkládají něco na části“. V kontextu počítačových programů se jedná o přeložení programu v binární podobě do čitelné formy, ovšem na stejné úrovni abstrakce. Příkladem může být překlad ze strojového kódu pro procesor do ekvivalentního jazyka symbolických instrukcí. Tyto nástroje se používají ke statické analýze, popř. úpravám kódu aplikace.

V případě použití klasického překladu, strojového kódu pro konkrétní procesor a optimalizací vzniká těžko srozumitelný a obtížně zpracovatelný jazyk, který se vůbec nemusí podobat původnímu zápisu zdrojového kódu. Např. překladač z jazyka C++ do konkrétního strojového kódu zahodí téměř veškeré informace o typech, datových strukturách, metodách a jejich názvech. Tyto se sice promítnou do struktury výsledného programu, ale jejich opětovné získání je obtížné.

Úkolem disassembleru je tedy tyto informace od sebe oddělit a doplnit to co chybí – např. názvy proměnných nebo návěští. Jeho výstup je zpravidla zcela nepodobný původnímu programu ve vysokoúrovňovém programovacím jazyce.

Zpětné překladače

Zpětné překladače jdou o krok dále než disassemblery a snaží se z původního strojového kódu nebo bajtkódu opět vytvořit čitelný zdrojový kód. Toto je samozřejmě daleko složitějším úkolem, protože tam, kde disassembler pouze popisuje program v čitelné podobě, zpětný překladač musí rozeznávat a doplňovat původní idiomy vysokoúrovňového jazyka. Dá se říci, že zpětný překladač začíná pracovat tam, kde disassembler skončil. Postup zpětného překladu závisí na vstupu zpětného překladače:

Strojový kód

Vstupem je binární, spustitelná reprezentace programu, cílená na konkrétní typ procesoru. Takováto reprezentace už zpravidla neobsahuje žádná metadata a ani stopu po původně použitých konceptech vyššího programovacího jazyka. Prvním krokem zpětného překladu je tedy aplikace podobného postupu jako u disassembleru. Pokud již disassembler pro strojový kód existuje, je samozřejmě možné použít jeho výstup jako vstup zpětného překladače.

Objektový kód

Objektový kód je již přeložený zdrojový kód, ale ještě nesestavený do výsledného programu. Stále obsahuje původní (nebo mírně předzpracované) názvy viditelných proměnných a funkcí. Odpadají tedy problémy s optimalizacemi aplikovanými překladačem na celý program. Toto je spíše výjimkou, protože objektový kód se většinou generuje a používá pouze během překladu. V praxi se může jednat o dynamicky načítané knihovny, kde je třeba propojit program a knihovnu za běhu. Viditelné symboly normálně použité pro spojení programu a knihovny lze využít jako vstupní body pro zpětný překlad.

Bajtkód

Při překladu do bajtkódu dochází k podobné ztrátě abstrakce jako v případě překladu do strojového kódu, ale zdaleka ne do takové míry. V bajtkódu se na rozdíl od strojového kódu často používají koncepty bližší původně překládanému jazyku. Skutečný překlad (za běhu) a optimalizace jsou ponechány na běhovém prostředí. Je samozřejmé, že dále nijak neupravený bajtkód je tak jednoduchým cílem pro zpětné inženýrství. Odbourávají se mnohé problémy spojené se zpětným překladem a vznikají jiné – zatímco skutečné procesory pracují podobně, mezi platformami používajícími bajtkód mohou být velké rozdíly. Skutečně generický zpětný překladač tak nutně musí podporovat daleko širší množinu konceptů již při čtení vstupního programu. Některé platformy ani nemají pevnou specifikaci a použitý bajtkód a struktura

jeho uložení je pouze implementačním detailem, který se mezi verzemi jazyka mění – toto je např. případ jazyka Python a jeho mnoha implementací.

2.2 Základní postupy a koncepty zpětného překladu

Zpětný překlad lze rozdělit na tři části:

- Přední část (angl. *front-end*) načítá program na vstupu, dělí jej na data a instrukce, ty dekóduje a dále provádí základní sémantickou analýzu instrukcí. Je závislá na vstupním jazyce – platformě pro kterou je program určen. Jejím výstupem je ekvivalent vstupního programu ve vhodné interní reprezentaci (IR), popř. graf toku řízení (angl. *control flow graph*).
- Prostřední část (angl. *middle-end*) přijímá program od přední části a dále jej analyzuje. Její hlavní úlohou je identifikovat v již platformě nezávislé interní reprezentaci datové typy (analýza toku dat), a vysokoúrovňové řídicí struktury (analýza toku řízení).
- Zadní, nebo také výstupní část generuje výsledný zdrojový kód ve zvoleném programovacím jazyce.

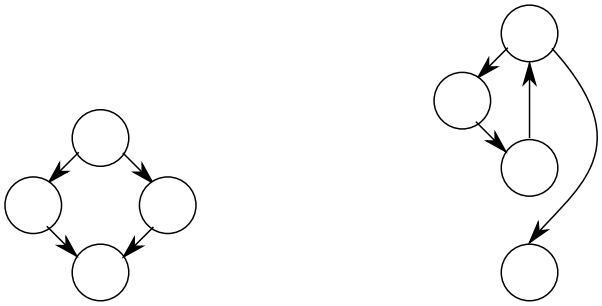
Sdružením jednotlivých úloh zpětného překladače do těchto tří částí vede k lepší rozšiřitelnosti překladače. Přidání více vstupních nebo výstupních jazyků znamená změny pouze v přední, respektive zadní části. Předpokladem ovšem je, že jak vstupní tak výstupní jazyky lze převádět do, respektive ze zvolené interní reprezentace.

Dekódování instrukcí

Vstupní jazyk zpětného překladače je zpravidla nějakou formou strojového kódu, popř. bajtkódu. Taková reprezentace (obzvláště v případě strojového kódu) není vhodná pro další analýzu, protože její instrukce často mají vedlejší účinky. Je tedy potřeba tento vstupní jazyk převést na vhodnou interní reprezentaci, kde je dosah jimi způsobených změn lokalizován a vhodně zakódován pro další zpracování.

Analýza toku řízení

Analýzu toku řízení lze rozdělit na generaci grafu toku řízení a následné analýzy na něm prováděné. Vstupem je nestrukturovaný seznam instrukcí, výstupem pak graf popisující, jakým způsobem na sebe instrukce za běhu programu mohou navazovat. V programu totiž vedle lineárního provádění instukcí může docházet k podmíněným a nepodmíněným skokům. Vstupní program se rozdělí na tzv. základní bloky (angl. *basic blocks*), které reprezentují skoky nepřerušené sekvence instrukcí a tvoří uzly grafu, a skoky, které tvoří hrany grafu. Následným zpracováním tohoto grafu lze eliminovat skoky přidané původním překladačem z technických důvodů (např. architektura nepovoluje skoky delší než nějaký limit) a získat obecné, vysokoúrovňové řídicí struktury jako je např. podmínka, nebo *while* cyklus. Příklad jak může vypadat graf toku řízení pro tyto struktury je na obrázku 2.2.



(a) if/else (b) while

Obrázek 2.2: Graf toku řízení pro vybrané řídicí struktury [9].

Analýza toku dat

Analýza toku dat dále pracuje s grafem toku řízení a obecně slouží k optimalizaci programu. V kontextu zpětného překladače a zvláště v kontextu dekódování původních instrukcí je využití optimalizace takřka nezbytné. Pokud mají instrukce vstupního jazyka mnoho vedlejších účinků, jsou pro ně generovány zbytečné instrukce interní reprezentace. Právě analýzou toku dat je možné odlišit a odstranit kód, který nemá žádný efekt na výsledek programu, nebo zjednodušit interní reprezentaci programu. Dalším efektem analýzy toku dat je typová inference. Pokud např. známe typ konkrétního parametru funkce, je možné prokázat, kde všude je tento parametr použit a odvodit tak typy použité uvnitř funkce.

práce, 2007.

2.3 Právní aspekty zpětneho inženýrství

Názory na zpětné inženýrství se samozřejmě v různých částech světa liší, a právě tyto názory formují, jakým způsobem na tuto problematiku pohlíží právo.

V České republice manipulaci se softwarem upravuje autorský zákon (121/200Sb.), dle kterého program může osoba oprávněná k jeho užívání zkoumat dle libosti a upravovat jej, pokud je to nezbytné pro jeho použití k stanovenému účelu (§ 66). Jako porušení autorských práv je ovšem chápána jakákoli činnost, která vede k odstranění prostředků sloužícího k ochraně díla (§ 43).

Citováno z autorského zákona [1]:

§ 66 Omezení rozsahu práv autora k počítačovému programu

(1) Do práva autorského nezasahuje oprávněný uživatel rozmnoženiny počítačového programu, jestliže

b) jinak rozmnožuje, překládá, zpracovává, upravuje či jinak mění počítačový program, je-li to nezbytné k využití oprávněně nabyté rozmnoženiny počítačového programu v souladu s jeho určením, není-li dohodnuto jinak,

d) zkoumá, studuje nebo zkouší sám nebo jím pověřená osoba funkčnost počítačového programu za účelem zjištění myšlenek a principů, na nichž je založen kterýkoli prvek počítačového programu, činí-li tak při takovém zavedení, uložení počítačového programu do paměti počítače nebo při jeho zobrazení, provozu či přenosu, k němuž je oprávněn,

Kapitola 3

Java a Java Virtual Machine

První část této kapitoly popisuje základy jazyka Java, který je jak původním jazykem většiny aplikací pro platformu JVM, tak zvoleným výstupním jazykem navrhovaného zpětného překladače. Druhá část kapitoly popisuje platformu JVM. Právě bajtkód používaný v JVM jsem zvolil jako první vstupní jazyk pro zpětný překladač.

Platforma JVM je virtuálním zásobníkovým počítačem, doplněným o čítač instrukcí. Nejedná se o jeden konkrétní program nebo o jednu konkrétní implementaci, ale o specifikaci jak má virtuální stroj fungovat. Implementace JVM jsou dostupné pro mnoho různých operačních systémů a hardwarových platforem, ale bajtkód je vždy stejný. Dle firmy Oracle, která v současné době Javu vlastní, běží JVM na řádově miliardách zařízení [17].

Java jako jazyk od svého prvního vydání v roce 1996 prodělala značné změny. V každé verzi došlo k rozšíření základních knihoven dodávaných s JVM. Zajímavější z pohledu zpětného překladače jsou ovšem podporované vlastnosti JVM:

- 1997 – byla vydána verze 1.1, ve které byla přidána podpora vnitřních tříd.
- 1998 – vydána verze J2SE 1.2. První verze označená jako „Java 2“. Byla přidána podpora reflekce a poprvé využita kompilace za běhu (*just in time compilation*).
- 2000 – verze 1.3. Toto je nejčastěji podporovaná verze v existujících zpětných překladačích.
- 2002 – verze 1.4. Byla přidána podpora zřetězování výjimek.
- 2004 – verze 5.0 (došlo ke změně verzování). Uvádí podporu for-each cyklů, odpověď na šablony v C++ (*generics*), automatické přetypování mezi základními a odpovídajícími referenčním typy (*autoboxing*), metody s variabiním počtem parametrů a metadat ve formě anotací.
- 2006 – verze 6. Přidána podpora využití anotací během překladu.
- 2011 – verze 7. Přidána podopora řetězců v příkazu `switch`, `try-with-resources` (lepší podpora mazání objektů, když dojde k výjimce) a podpora dynamicky typovaných jazyků pomocí instrukce `invokedynamic`.

3.1 Základy jazyka Java

Jazyk Java je staticky typovaným, objektově orientovaným, impravitním jazykem. Syntaxe je velmi podobná jazyku C++. S výjimkou primitivních typů jsou všechny typy jazyka

třídami odvozenými od jediného základního typu: `java.lang.Object`. Není možné odděleně deklarovat a definovat třídu. Třídy jsou dále organizovány do balíků (angl. *packages*). Ty spolu s třídami tvoří hierarchickou strukturu, která odpovídá způsobu uložení v souborovém systému a uvnitř JAR archivů (viz dále).

Pokud tedy máme třídu `org.dethware.hello.HelloWorld`, její zdrojový kód musí být uložen v souboru `HelloWorld.java` umístěném na cestě `org/dethware/hello/`. Třída navíc musí být i uvnitř souboru pojmenována `HelloWorld`. Porušení tohoto schématu vede k chybám při překladu.

Java používá koncept „cesty ke třídám“ (angl. ClassPath), což je seznam cest v souborovém systému a uvnitř JAR archivů, který je kořenem pro vyhledávání a načítání tříd. Tento seznam lze specifikovat při spuštění aplikace. Platí, že cesty uvedené v seznamu dříve mají prioritu. Dalším faktorem je použitá třída pro načítání tříd (`ClassLoader`). Tuto třídu je však možné nahradit a zcela způsob načítání tříd změnit (nebo dokonce třídy generovat za běhu programu).

Třídy lze rozdělit na tři základní typy – normální třídy, rozhraní a anotace. Jazyk nepodporuje vícenásobnou dědičnost jako např. C++, ale platí, že třída může implementovat více rozhraní. Rozhraní pak pouze definují prototypy metod. Anotace slouží k definici struktur pro metadata a jejich zápis je odlišný. Metody anotace definují jednotlivé položky struktury a mohou mít přiřazenu výchozí hodnotu.

Třída dále sestává z definic metod a polí (angl. *fields* – členské proměnné). Jazyk podporuje statické metody a pole sdílené mezi všemi instancemi třídy. Je tak možné implementovat obdobu globálních proměnných z jiných jazyků. Vstupním bodem Java aplikace může být statická metoda `main` kterékoliv třídy v aplikaci (mechanismus nastavení konkrétního vstupního bodu je popsán dále). Příkladem minimální definice třídy může být klasický program „Hello World“, uvedený ve výpisu 3.1.

Narozdíl od jazyků jako je C nebo C++ Java nepodporuje ukazatele a přímý přístup do paměti. Není ani možné přímo paměť alokovat a uvolňovat. Objekty jsou vytvářeny na haldě (viz dále) a v kódu se pak nepracuje přímo s objektem, ale referencí na něj. Uvolňování paměti zastává tzv. *garbage collector*. JVM sleduje platnost jednotlivých objektů (počet referencí na ně) a pokud tato vyprší, jsou objekty automaticky zničeny. Zpracování chyb je v Javě řešeno použitím výjimek.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Obrázek 3.1: Ukázka jednoduchého programu `HelloWorld.java`.

3.2 Běhové prostředí

Běhové prostředí JVM[1] lze rozdělit do čtyř částí:

- Halda
- Registry

[1] Java Runtime Environment

- Prostor pro metody
- Zásobník

Každá aplikace má svou vlastní haldu a prostor pro metody. Každé vlákno má svůj vlastní čítač instrukcí a zásobník. Zásobník je dále rozdělen na rámce, kde každá volaná metoda má svůj vlastní rámec. Pro přístup k zásobníku se používají tři další registry, ale ty nejsou pro nijak viditelné v prováděném programu.

Halda

JVM neumožňuje přímo alokovat a dealokovat paměť, jak je tomu často jinde. Java má operátor *new*, který alokuje objekty na haldě, a které jsou automaticky uvolněny garbage collectorem, pokud již nejsou programem používány. Jedním z důvodů pro tuto implementaci je bezpečnost. Když není možné přímo přistupovat k paměti, není také možné v ní přepsat něco jiného, než jsou vlastní data programu. Halda dále nebude příliš zajímavá, protože zpětný překladač ji nebude muset emulovat. Zpětný překlad je druhem statické analýzy a se způsobem uložení dat aplikace v paměti nepřichází do kontaktu.

Prostor pro metody

JVM má prostor pro metody, který je sdílen mezi všemi vlákny. Je analogický k uložení kódu v konvenčních spustitelných souborech nebo k sekci `.text`(kódová sekce) v procesu operačního systému. Ukládá struktury Java tříd jako jsou konstanty, data a kód jednotlivých metod a polí - tedy statická data, která se obvykle nemění během výpočtu. Konkrétní implementace se může lišit mezi jednotlivými verzemi JVM a opět není z hlediska zpětného překladu zajímavá (obdobný případ jako halda).

Zásobník

Zásobník je v JVM rozdělen na rámce, kde volání metody vytváří nový rámec. Rámec zásobníku je pak rozdělen na tři části: pole lokálních proměnných, zásobník operandů a reference na pole konstant třídy, ke které metoda patří.

V každém vlákně je aktivní vždy jen jedna metoda a rámec zásobníku. Při návratu z metody je rámec zrušen. Pokud nedošlo k výjimce, je návratová hodnota uložena na zásobník operandů volající metody a výpočet pokračuje další instrukcí v této metodě. Jinak se žádná hodnota nevrací a kontrolu nad výpočtem přebírá vyhodnocení výjimek.

Z pohledu zpětného překladu bude potřeba emulovat zásobník v rámci metod. Tedy, každá metoda třídy začne s výchozím stavem zásobníku a emulací průchodu metodou (tak, jak by jí procházel JVM) se bude dát zjistit typ jednotlivých lokálních proměnných. Zvláštní pozornost pak bude věnována správnému vyhodnocení výjimek.

3.3 Struktura jar souborů

JAR (Java ARchive) je archivním souborem používaným k agregaci mnoha souborů (tříd, metadat, a zdrojů), ze kterých se skládá typická Java aplikace. Tyto soubory jsou uloženy ve stromové struktuře složek, podobně jako v klasickém souborovém systému. Formát je založen na populárním formátu PKWARE ZIP [20]. JAR soubor může navíc obsahovat tzv. manifest, uložený v `META-INF/MANIFEST.MF`. Tento soubor může obsahovat (vedle jiných) záznamy `Classpath` a `Main-Class`.

Classpath určuje, které další soubory mají být načteny při načítání JAR souboru. *Main-Class* pak určuje hlavní třídu, jejíž metoda *main* se použije při přímém spuštění JAR souboru jako vstupní bod. Případným výstupem zpětného překladače by mohl být i seznam možných vstupních bodů, s vyznačeným hlavním vstupním bodem, pokud je specifikován. Vedle JAR souborů může Java používat i jiných, vesměs však velmi podobných formátů.

V rámci zpětného překladače nebude extrakce těchto souborů řešena, protože k tomu lze využít běžně dostupných nástrojů jako *unzip*[2]. Úplnou specifikaci lze nalézt na webových stránkách firmy Oracle [19].

3.4 Struktura souborů se třídami

Základní strukturu souboru se třídou (standardně soubory s příponou „.class“) lze popsat strukturou v jazyce C uvedenou ve výpisu 3.2. Jedná se tedy o seznam jednotlivých částí, kde jejich délka je určena před jejich výskytem.

```
struct Class_File_Format {
    u4 magicke_cislo;
    u2 vedlejsi_verze;
    u2 hlavni_verze;
    u2 pocet_konstant;
    cp_info konstanty[pocet_konstant - 1];
    u2 pristupove_priznaky;
    u2 tato_trida;
    u2 nadrazena_trida;
    u2 pocet_rozhrani;
    u2 rozhrani[pocet_rozhrani];
    u2 pocet_poli;
    field_info pole[pocet_poli];
    u2 pocet_metod;
    method_info metody[pocet_metod];
    u2 pocet_atributu;
    attribute_info attributy[pocet_atributu];
}
```

Obrázek 3.2: Základní struktura souboru se třídou.

Všechny číselné hodnoty v souboru jsou uloženy v big-endian formátu. Magická hodnota je vždy `0xcafebabe`. Hlavní a vedlejší verze pak určují verzi formátu souboru se třídou.

Protože úplná specifikace popisující soubory se třídami má 186 stran a její úplná reprodukce by byla spíše kontraproduktivní, uvedeme zde pouze ty skutečnosti, které budou mít vliv na návrh zpětného překladače.

V této sekci je čerpáno z oficiální specifikace souborů se třídami [18] a knihy Decompiling Java Language [15].

Tabulka konstant

Tabulka konstant obsahuje jednotlivé konstanty použité ve třídě – řetězce v modifikovaném formátu UTF-8, celá čísla se znaménkem, čísla s plovoucí desetinnou čárkou a záznamy popisující typy a názvy použitých polí, tříd a metod. Nultá konstanta je vyhrazena jazyku

[2] http://www.winimage.com/zLibDll/minizip.html

a konstanty se tak indexují od 1. Další zvláštností je, že z historických důvodů zabírají v tabulce konstanty typu `long` a `double` místo jednoho indexu dva.

Přístupové příznaky

Přístupové příznaky určují, jestli se jedná o třídu nebo rozhraní a jestli je třída abstraktní, veřejná, finální (tedy z ní nejdou dále odvozovat další třídy), je výčtovým typem, je anotací (o těch dále), nebo je syntetická.

Syntetické třídy se nevyskytují v původním zdrojovém kódu, nebo jsou tak alespoň označeny. Je otázka, jak by se k nim měl zpětný překladač zachovat a jestli tak nejsou označeny všechny třídy (např. po použití obfuskátoru).

Tato a nadřazená třída

Toto jsou ukazatele do tabulky konstant určující názvy této (*this*) třídy a třídy ze které je odvozená. Ve výsledku řetězce odkazující se na typy.

Rozhraní

Třída dále obsahuje seznam jednotlivých rozhraní implementovaných touto třídou, v podobě odkazů do tabulky konstant.

Pole

Pole třídy (*field*, neplést si s datovým typem *array*) jsou proměnné – buď svázané s vytvářenými instancemi třídy nebo s třídou samotnou – statické. Každé pole lze popsat strukturou field_info, uvedenou ve výpisu 3.3.

```
struct field_info {
  u2 pristupove_priznaky;                    // access_flags
  u2 index_jmena;                            // name_index
  u2 index_popisovace;                       // descriptor_index
  u2 pocet_atributu;                         // attributes_count
  attribute_info atributy[pocet_atributu];   // attributes
}
```

Obrázek 3.3: Struktura field_info.

Podobně jako třída samotná mají i její pole přístupové příznaky. Ty mohou být veřejné, privátní, chráněné, statické, finální (hodnota lze přiřadit jen jednou), volatilní (hodnota se nemůže ukládat do vyrovnávací paměti, význam pro synchronizaci), přechodné (hodnota se nemá načítat a ukládat automaticky), syntetické a dále může být pole označeno jako výčtový typ.

Následuje odkaz do tabulky konstant na název pole a na jeho deskriptor (určující typ).

Dále může každé pole mít atributy (seznam struktur `field_info`) typu `ConstantValue` (určuje konstantní hodnotu pole), `Synthetic` (obvyklý význam), `Signature` a `Deprecated`.

Metody

Metody jsou samozřejmě z hlediska zpětného překladu nejzajímavější. Jsou uloženy jako struktury `method_info`, které jsou stejné jako struktury `field_info` na obrázku 3.3. Rozdíl

je v atributech a možných hodnotách přístupových příznaků.

Metoda může být podobně jako pole veřejná, privátní, chráněná, statická a finální. Na rozdíl od polí však také může být abstraktní, nativní (tedy kód není v jazyce Java a jde o volání nativního kódu), syntetická (je generována překladačem a nenachází se v původním zdrojovém kódu), striktní (deklarována s klíčovým slovem `strictfp`, které upravuje chování operací s plovoucí desetinnou čárkou), označená jako přijímající variabilní počet parametrů a označená jako přemosťovací (tyto metody jsou generovány překladačem pro přemostění mezi normálními a generickými typy).

Možné atributy metody jsou `Code` (obsahuje bajtkód metody), `Exceptions` (obsahuje seznam výjimek, které může metoda produkovat), `Synthetic` (opět, obvyklý význam), `Signature` (obdobný význam jako u polí), `RuntimeVisibleParameterAnnotations` a `RuntimeInvisibleParameterAnnotations`. Anotace jsou popsány v sekci 3.4. Atribut `Code` je blíže popsán v následující sekci.

Atribut `Code`

Tento atribut je prakticky nejdůležitější částí popisu třídy, protože obsahuje bajtkód konkrétní metody a informace o vyhodnocování výjimek. Atribut lze popsat pomocí struktury `Code_attribute` uvedené ve výpisu 3.4.

Na počátku, jako u všech ostatních atributů, je uveden název atributu a jeho úplná délka v paměti. Následně jsou uvedeny hodnoty `max_stack` a `max_locals`, určující největší hloubku zásobníku operandů během vyhodnocování metody a počet lokálních proměnných včetně předávaných parametrů alokovaných při zavolání metody. Poté následuje samotný bajtkód (instrukční sada je v hrubých rysech popsána v sekci 3.6) a tabulka výjimek.

Každá položka v tabulce výjimek obsahuje počáteční pozici v kódu, odkud začíná platit, konečnou pozici, pozici kódu provádějícího obsluhu výjimky a typ výjimky v podobě odkazu do tabulky konstant. Překladače obvykle negenerují tabulku výjimek, ve které by se překrývaly výjimky. S tím se však nedá počítat, pokud je kód nějak dále upraven (např. obfuskátorem). Zpětný překladač tedy musí správně reagovat na překryv výjimek. Atribut dále může obsahovat další nepovinné atributy, vesměs používané jako ladicí informace.

```
struct Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  {
    u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  } exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Obrázek 3.4: Struktura popisující atribut `Code`.

Atributy třídy

Třída jako taková samozřejmě může mít své vlastní atributy. Zde je potřeba zmínit atributy `InnerClasses`, `EnclosingMethod`, `RuntimeVisibleAnnotations` a `RuntimeInvisibleAnnotations`. Anotace a jejich atributy jsou popsány v následující sekci.

Pokud se třída odkazuje na další vnitřní nebo anonymní třídy, používá se atribut `InnerClasses` pro kódováni jejich názvů. Protože specifikace neumožňuje libovolně zanořovat datové struktury popisující třídy, je nutné původní definice tříd rozdělit. Zanořeným a anonymním třídám jsou pak přiřazeny názvy automaticky. Pro správný zpětný překlad bude nutné rozeznávat tento atribut a následně zpět začlenit definice zanořených tříd. Symetricky k atributu `InnerClasses` se používá atribut `EnclosingMethod`, který u zanořené třídy určuje, že byla definována v rámci určité metody.

Anotace

Anotace jsou metadata připojená k třídám, metodám, polím (členským proměnným) nebo parametrům metod. Jedná se o speciálně definované třídy a je možné jich specifikovat libovolný počet. Nemohou mít žádné metody, protože se pouze přidávají hodnotou. K viditelným anotacím je možné přistupovat pomocí reflexe za běhu programu. Java navíc povoluje přidávat anotace i k dalším anotacím. Jejich účelem je obohatit jazyk o další sémantickou vrstvu.

Java obsahuje několik předdefinovaných anotací. Např.:

- @Override – Určuje, že metoda je odvozena od metody v nadtřídě. Pokud se metoda v nadtřídě nevyskytuje, hlásí tuto skutečnost překladač během kompilace.
- @Deprecated – Metoda je vyřazena a neměla by se používat. Překladač opět hlásí takováto použití.

Uživatelsky definované anotace pak mohou být použity k jakémukoliv účelu. Příkladem může být specifikace vstupního bodu modulu programu, načítaného dodatečně za běhu. V takovém případě se prohledají třídy modulu a zavolají se jeho zaváděcí metody označené takovouto anotací. Zpětný překladač tedy pro zachování sémantiky programu musí umět anotace zpracovat.

3.5 Typy a názvy typů

Typový systém Javy je poměrně bohatý, ale v zásadě založený na několika málo typech. Popisovače typů (polí a metod) jsou v souborech s třídami uváděny v tabulce konstant v podobě řetězců, které lze popsat jednoduchou gramatikou. Pro stručnost není třeba ji uvádět, ale zpětný překladač tyto řetězce bude muset umět přečíst a získat z nich informace o typech jednotlivých elementů tříd.

3.6 Instrukční sada a základní datové typy

Jak jsem již uvedl, JVM je jednoduchým virtuálním zásobníkovým počítačem. Kód každé instrukce je vždy identifikován jedním bytem, ale instrukce mohou mít variabilní délku. Ne všechny kódy instrukcí jsou využity a určitá část z nich je vyhrazena pro interní použití v JVM (dvě implementačně závislé instrukce a breakpoint). Instrukční sadu lze rozdělit do několika skupin:

- Načítání a ukládání hodnot na/ze zásobníku
- Aritmetické a logické instrukce
- Převody typů
- Vytváření a manipulace s objekty (`new`, `newarray`, `putfield` atp.)
- Operace se zásobníkem (např. instukce `swap`, která zamění dvě horní hodnoty na zásobníku)
- Volání metod
- Instrukce řídicí struktury (`if`, `goto`, `return` atd.)
- Instrukce pro práci s monitory – pro synchronizaci
- Některé další speciální instrukce jako `instanceof`

JVM podporuje tyto základní typy:

- Primitivní typy
  - `boolean`
  - `returnAddress` – typ použitý pro uložení návratové adresy metody
  - číselné typy
    - Celočíselné typy
      - `byte` – 8-bitové celé číslo ve dvojkovém doplňku
      - `short` – 16-bitové celé číslo ve dvojkovém doplňku
      - `int` – 32-bitové celé číslo ve dvojkovém doplňku
      - `long` – 64-bitové celé číslo ve dvojkovém doplňku
      - `char` – 16-bitové číslo bez znaménka, které reprezentuje znak v UTF-16
    - Čísla s plovoucí desetinnou čárkou
      - `float` – IEEE 754 single-precision (32-bit)
      - `double` – IEEE 754 double-precision (64-bit)
- Referenční typy - všechny odvozeny od `java.lang.Object`
  - reference na instanci třídy
  - reference na pole (array)
  - reference na rozhraní

JVM navíc používá při práci se zásobníkem specifické instrukce pro každý primitivní datový typ (s výjimkou boolean). To zjednodušuje zpětnou rekonstrukci typů použitých v původním programu.

3.7 Krátký přehled dostupných nástrojů

Pro JVM existuje velké množství disassemblerů, zpětných překladačů a knihoven pro čtení a zpracování bajtkódu.

Bylo by nemístné nezmínit vůbec první zpětný překladač pro jazyk Java – Mocha [23]. Ten v době svého vydání způsobil poměrně velký rozruch. Jeho autor, Hanpeter van Vliet, ovšem brzy po vydání zpětného překladače tragicky zemřel a vývoj tak ustal.

Aktuální stav zpětných překladačů je dá se říci obdobně neutěšený. Převážná většina funkčních nástrojů je určena pro velmi staré verze JVM a nástroje podporující novější verze jsou buď experimentální, opuštěné, nebo nejdou začlenit do většího projektu ani nijak rozumně automatizovat (JD-gui). V reálném nasazení je možné použít nástroje JadRetro [11], který překládá třídy do formátu podporovaného v JVM 1.3 a následně použít jeden ze starších (kvalitních) nástrojů. Takové řešení však znamená ztrátu informace danou okleštěním o vlastnosti přidané v novějších verzích JVM.

BCEL, ASM a jiné knihovny pro manipulaci s bajtkódem

BCEL je knihovna pro analýzu, vytváření a manipulaci s Java bajtkódem a třídami. Reprezentuje vstupní soubor jako strom objektů, kterým jde procházet a dále s ním manipulovat. Knihovna je napsána v jazyce Java a vydána pod licencí Apache 2.0. ASM je podobný framework, ale zaměřený na jednoduchost použití a optimální výkon.

JAD

Zpětný překladač JAD (Java Decompiler) je v současné době neudržovaný. Poslední verze vyšla v roce 2001 a podporuje třídy generované pomocí JDK 1.3 a starších. Je volně k dispozici pro nekomerční účely, funguje jako aplikace pro příkazovou řádku (tedy je jednoduché jeho použití automatizovat) a je často používaný v kombinaci s různými grafickými nadstavbami.

Původní web projektu již bohužel neexistuje a zpětný překladač je tak dostupný pouze z neoficiálních zdrojů [27]. I tak je stále používán. Pro zpětný překlad novějšího bajtkódu je možné jej kombinovat s použitím nástroje JadRetro [11].

JD (JD-Core, JD-Gui)

V současné době jeden z nejlepších dostupných zpětných překladačů Javy, navíc bezplatně. Je napsán v jazyce C++ (lze to poznat z použitých knihoven), ale zdrojový kód pravděpodobně nikdy nebude zveřejněn. Neposkytuje navíc ani rozhraní pro spuštění z příkazové řádky a tak je jeho využitelnost výrazně omezena. Autorem je Emmanuel Dupuy a binární distribuce je dostupná z webových stránek projektu [4].

JODE – Java Optimize and Decompile Environment

JODE je jedním z volně dostupných zpětných překladačů. Podobně jako JAD podporuje maximálně JDK 1.3 a poslední vydaná verze je z roku 2004. Na rozdíl od něj je však část kódu pod licencí GNU LGPL a je tak možné nahlédnout do jeho fungování.

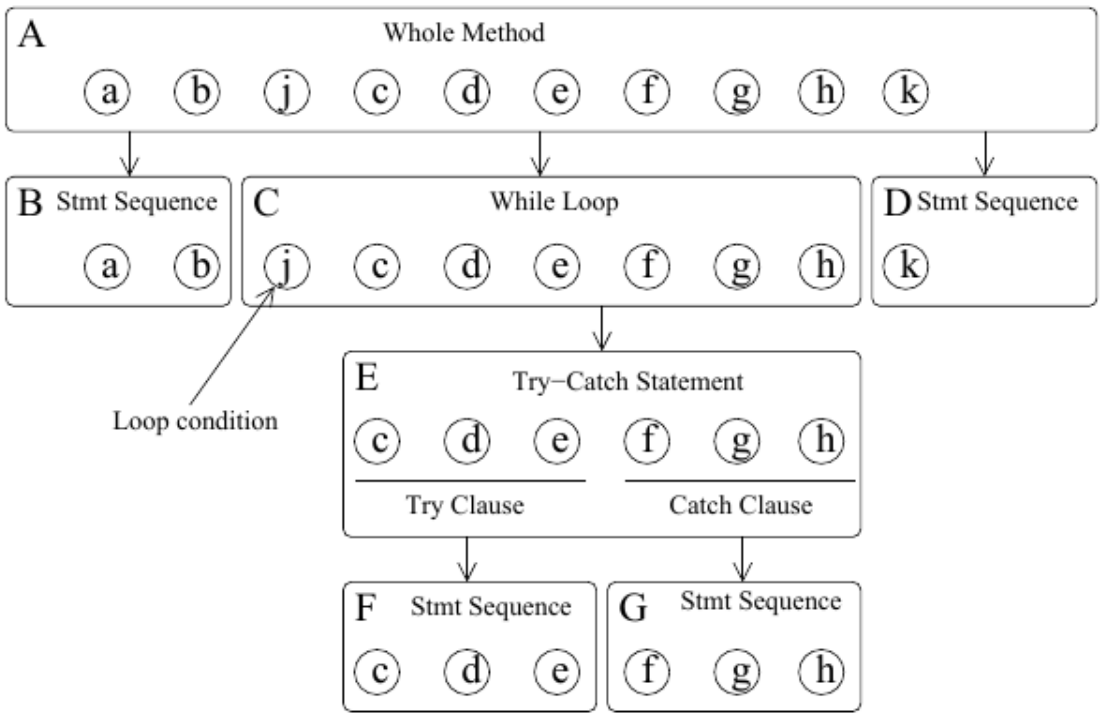
JODE umí nejen číst JVM třídy, ale i je zpět generovat a provádět na kódu transformace a optimalizace. Může tedy být považován vedle zpětného překladače i za obfuskátor.

V zásadě převádí vstupní bajtkód na vlastní interní reprezentaci (základní bloky) a následně nad touto reprezentací provádí analýzu toku řízení. Převádí základní bloky na "flow bloky", které analyzuje a propojí s vyhodnocováním výjimek.

DAVA

DAVA je zpětný překladač vyvíjený výzkumnou skupinou Sable na kanadské univerzitě McGill. Jeho implementací a zlepšováním se zabývá několik volně dostupných technických zpráv a diplomových prací. Projekt je vystavěn nad frameworkem Soot a jeho hlavním přínosem je nezávislost na původně použitém překladači nebo původním jazyce [12].

Vedle základní analýzy toku řízení provádí další transformace založené na strukturu zapouzdřujících stromech (*Structure Encapsulation Tree*). Tato struktura je tvořena jednotlivými uzly grafu toku řízení a zapouzdřuje je tak, aby vznikl strom odpovídající původnímu zdrojovému kódu. Příklad takovéto struktury pro jednoduchou metodu je na diagramu 3.5. Nad tímto pak provádí další transformace pro zpřehlednění a zjednodušení výsledného kódu [14]. Výhodou oproti např. JODE je obecnost a formální přístup k problému. Framework Soot je aktuálně vyvíjen a podporuje nejnovější verzi JVM bajtkódu. Je k dizpozici pod licencí GNU LGPL.



Obrázek 3.5: Příklad strukturu zapouzdřujicího stromu [12].

3.7.1 Shrnutí

Srovnání dostupných zpětných překladačů je uvedeno v tabulce 3.1.

Mimo uvedených nástrojů existuje celá řada jiných, zpravidla zastaralých a opuštěných projektů. Dá se říci, že žádný existující nástroj nelze přímo použít jako základ navrhovaného

Nástroj	Licence	Zdrojový kód	JVM	Kvalita	Jazyk
JAD	proprietární	NE	1.3	Vysoká, ale nástroj zastaral	Java
JD-gui	proprietární	NE	6	Vysoká, má problémy s některými konstrukcemi	C++
JODE	GNU LGPL	ANO	1.3	Vysoká, má ovšem problémy s optimalizovaným kódem	Java
DAVA	GNU LGPL	ANO	7	Vysoká	Java

Tabulka 3.1: Srovnání uvedených nástrojů.

zpětného překladače. Existující nástroje a informace o nich publikované však mohou být brány jako inspirace. Z dostupných nástrojů je jediným perspektivním nástoj DAVA – je aktivně vyvíjen a kvalita výstupu je na vysoké úrovni. Navíc se pracuje na podpoře bajtkódu platformy DVM. Překážkami v jeho použití jsou implementační jazyk (Java) a licence – ta nedovoluje úpravy, které by způsobily nekompatibilitu s původní knihovnou, protože podle licence musí uživatel mít možnost nahradit dodanou verzi knihovny za vlastní. Textová forma interní reprezentace používané ve frameworku SOOT by ovšem mohla být alternativním vstupem pro existující zpětný překladač.

3.8 Příklad reálného využití zpětného překladu Javy

Jako dobrý příklad může posloužit projekt *Minecraft Coder Pack* [25], který je postaven na zpětném překladu a deobfuskaci hry Minecraft. Na třídy aplikace jsou aplikovány některé techniky obfuskace časté u platforem používajících bajtkód. Zejména se jedná o přejmenování (a znehodnocení) všech jmen tříd, metod a polí, kombinovaného s agresivním přetěžováním metod. MCP tento problém řeší mapováním nových jmen na obfuskovaná jména (jedná se o bijektivní zobrazení).

Překlad je řešen použitím nástroje JadRetro na původní JAR soubor, zpětným překladačem JAD a aplikací záplat tam, kde tento překladač neprodukuje přeložitelný kód. Výsledkem je automatizovaný nástroj pro zpětný překlad a následný nový překlad upraveného zdrojového kódu.

Tento projekt by mohl posloužit jako ideální finální test pro vzniklý zpětný překladač – jedná se o reálný kód využívající novější vlastnosti JVM, který zároveň používá rozšíření v podobě nativních knihoven.

Kapitola 4

Dalvik VM

Dalvik VM (dále jen DVM) se používá v operačním systému Android, který je velmi rozšířený v dnešních chytrých telefonech. DVM je podobně jako JVM virtuálním počítačem a podobně jako JVM je primárně zaměřen na jazyk Java. Od toho se odvíjí i jeho typový systém a struktury formátu uložení tříd, které jsou podobné (implementují stejné vlastnosti). Na rozdíl od JVM jsou ve virtuálním stroji použity registry a instrukční sada se výrazně liší. Každá aplikace také běží jako samostatný proces se svou vlastní instancí DVM. Je tedy kladen důraz na minimalizaci velikosti kódu aplikací.

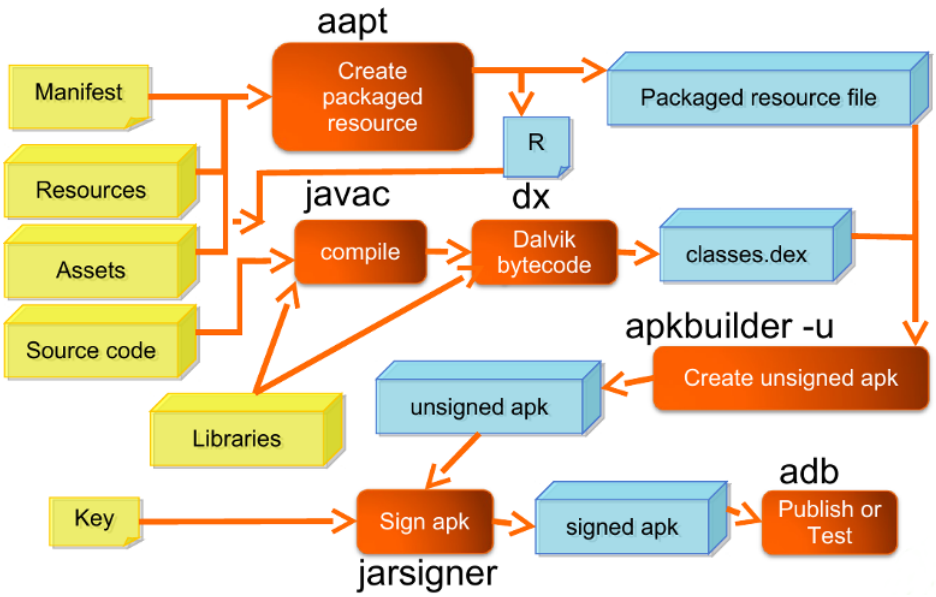
4.1 Struktura APK souborů

Formát Android application package (APK) je používán pro distribuci a instalaci softwaru v systému Google Android a většina aplikací pro DVM je k dispozici právě v tomto formátu. Základem je jako v případě JAR ZIP archiv. Obsah archivu je podobný, ale na rozdíl od JAR souborů se třídy neukládají odděleně, ale jako jeden `classes.dex` soubor. Data aplikace mohou být předkompilována ve formě souboru `resources.arsc`. Jakékoliv použité XML dokumenty mohou navíc být uloženy jako binární data. Cílem je minimalizovat dobu načítání aplikace, protože se předpokládá použití v zařízeních jako jsou mobilní telefony.

Překlad aplikace (znázorněný na obrázku 4.1) do tohoto formátu probíhá v několika krocích. Data aplikace se převedou na `resources.arsc` a syntetickou třídu `R`, která slouží k propojení bajtkódu s daty aplikace. Kód se zpracuje překladačem `javac` do formy JVM bajtkódu, který je následně zpracován překladačem `dx` do formátu `dex`.

Typicky je obsah APK archivu následující:

- Složka `META-INF`
  - `MANIFEST.MF` – Soubor s manifestem, podobého obsahu jako u formátu JAR
  - `CERT.RSA` – Digitální podpis pro následující soubor
  - `CERT.SF` – Obsahuje seznam použitých datových souborů a jejich SHA-1 kontrolní součty
- Složka `lib`, která může obsahovat nativní knihovny pro konkrétní architektury
- Složka `res`, obsahující datové soubory nezkompilované do souboru resources.asrc
- `AndroidManifest.xml` – Dodatečná metadata, která popisují obsah archivu a jeho závislosti. Může být v binární podobě.



Obrázek 4.1: Postup překladu aplikace pro Android [7].

- `classes.dex` – Všechny třídy ve formátu DEX
- `resources.arsc` – Soubor s předkompilovanými daty.

Zpětný překladač by měl ideálně umět získat zpět nejen čitelný a korektní zdrojový kód, ale také i původní data aplikace. Použití překladače `javac` navíc zaručuje, že v bajtkódu DVM se nevyskytuje nic, co by nešlo vyjádřit v bajtkódu JVM – lze mezi nimi převádět.

4.2 Formát DEX

V DEX souboru jsou uloženy všechny třídy aplikace [21]. Princip uložení je podobný jako u .class souborů používaných v JVM – tabulky s náhodným přístupem pomocí indexů. Základní struktura je následující:

- Hlavička – obsahuje informace o verzi souboru a odkazuje se na počátky jednotlivých sekcí.
- Tabulka identifikátorů řetězců – všechny v souboru použité řetězcové konstanty i typové identifikátory.
- Tabulka použitých typů – identifikátory veškerých tříd, polí a primitivních typů, nezávisle na tom jestli jsou nebo nejsou definovány v rámci stejného .dex souboru. Řazena podle názvu typů.
- Tabulka prototypů metod – identifikátory použitých metod. Je řazena podle návratového typu a typu argumentů.
- Tabulka identifikátorů polí (fields, členské proměnné)
- Tabulka identifikátorů metod – sdružuje typ, název a prototyp metod.
- Definice tříd

- Data – sekce pro uložení dat všech předchozích tabulek.
- Link data – standardem nedefinovaná sekce pro uložení staticky linkovaných souborů.

Tabulky na počátku souboru se buď odkazují do jiných tabulek pomocí indexů nebo absolutní adresou od počátku souboru na samotná data. Důsledkem tohoto rozložení jsou mnohem menší nároky na paměť, protože konstanty a identifikátory jsou sdíleny mezi třídami. V JVM je tomu naopak – každá třída má svoji vlastní tabulku konstant a dochází tak k duplikaci hodnot mezi třídami.

4.2.1 Optimalizované soubory se třídami (ODEX)

Odex soubory jsou optimalizovanou verzí DEX souborů. K jejich úspěšnému přečtení je potřeba mít k dispozici všechny knihovny, na kterých jsou závislé. ODEX má totiž rozšířenou instrukční sadu, a je optimalizován pro konkrétní nasazení – např. obsahuje virtuální tabulky metod a předpokládá konkrétní rozložení aplikace v paměti. Pro převod ODEX souborů zpět na DEX existují nástroje.

4.3 Instrukční sada a datové typy

Datové typy použité v DVM jsou podobné jako u JVM. Rozdíl je v instrukční sadě, která používá *virtuální* registry. Většina instrukcí podporuje přímé použit prvních 16 registrů, ale v rámci metody je možné použít registrů více. V instrukční sadě není žádný prostředek pro přístup k zásobníku. Interně jsou virtuální registry mapovány buď na zásobník, nebo na skutečné registry procesoru. Ve výsledku má pak DVM méně práce při načítání tříd, protože mapování proměnných na registry je částečně provedeno již překladačem. Dá se říci, že tyto registry jsou obdobou lokálních proměnných v zásobníkových rámcích JVM.

4.4 Přehled dostupných nástrojů

Dalvik VM je novější než JVM a je pro něj dostupných méně nástrojů. Zatím neexistuje jediný zpětný překladač cílený přímo na bajtkód použitý v DVM, který by nebyl pouhým experimentem. Obvyklý postup je buď použití disassembleru, nebo překladačů z DEX formátu do JVM tříd. Následně lze použít víceméně kterýkoliv zpětný překladač cílený na JVM bajtkód.

- dex2jar zahrnuje více nástrojů pro práci s APK archivy, včetně konverze do JVM JAR formátu. Kód je vydán pod licencí Apache 2.0 a nic nebrání jeho použití.
- Dare, podobné jako dex2jar umožňuje překlad z DVM bajtkódu do JVM. Je pokračováním předchozího projektu ded. Oba jsou vydány pod GNU GPL. Dare podle dostupné dokumentace dosahuje lepších výsledků než dex2jar a úspěšně přeložil *do verifikovatelné podoby* 99% z 1100 testovaných aplikací, dex2jar korektně přeložil pouze 60% aplikací. Verifikovatelnou podobou se rozumí podoba ve které cílový virtuální stroj načte aplikaci bez chyb [16]. Dare právě kvůli rozšířené podpoře verifikace jak původního tak cílového bajtkódu vyžaduje integraci s platformou Android a jejím zdrojovým kódem. Ten nelze přeložit pod OS Windows.

- android-apktool je nástroj pro manipulaci s APK soubory a může vedle rozbalení komprimovaných datových souborů (resources.arsc) také pracovat jako disassembler. Na pozadí využívá (dis)assembleru (bak)smali. Je dostupný pod licencí Apache 2.0 [28].

Kapitola 5

Zpětný překladač projektu Lissom

Tato kapitola popisuje výzkumný projekt Lissom probíhající na Fakultě informačních technologií v Brně a v jeho rámci vyvíjený generický zpětný překladač. Kapitola vychází ze zdrojů [30], [9], [8] a čtení zdrojového kódu zpětného překladače.

Cílem projektu je vytvořit integrované vývojové prostředí pro návrh nových procesorů schopné generovat všechny potřebné nástroje – assembler, disassembler, linker, debugger, simulátor navrženého procesoru, překladač z jazyka C a rekonfigurovatelný zpětný překladač. Architektura procesoru a jeho instrukční sada jsou popsány v jazyce ISAC[1].

Zpětný překladač je strukturován podobně jako jiné překladače – skládá se z přední části (front-end), optimalizační části (middle-end) a výstupní části (back-end). Přední a prostřední část jsou ve zdrojovém kódu ovšem propojeny. Schematicky lze popsat strukturu zpětného překladače obrázkem 5.1. Pouze přední část je závislá na platformě a je generována na základě popisu architektury. Zpětný překladač je vystavěn na systému LLVM[2] a používá jeho interní reprezentaci (LLVM IR) jako svou vlastní.

V současné době je možné generovat zpětný překladač pro architektury MIPS, ARM a Intel x86.

5.1 LLVM IR

LLVM je infrastruktura pro překladače jejíž hlavními rysy jsou univerzálnost, jazykově nezávislý soubor instrukcí, typový systém, interní reprezentace, mnoho vestavěných optimalizačních algoritmů a aplikačně programové rozhraní pro několik programovacích jazyků.

LLVM IR má tři formy – v paměti uloženou formu určenou pro zpracování překladačem, serializovanou binární formu pro uložení na disku a čitelnou formu určenou pro zobrazení. Instrukční sada je založena na konceptu Static Single Assignment (SSA), což znamená, že každé přiřazení se provádí do nové, unikátní proměnné [5].

Formát reprezentace lze nejlépe ukázat na příkladu. Velmi jednoduchému programu v jazyce C uvedenému ve výpisu 5.2 odpovídá ekvivalentní zápis v LLVM IR ve výpisu 5.3. Z výsledného zápisu je zřejmé, že instrukce může mít na levé straně přiřazení a na pravé straně je vždy operační kód instrukce spolu s operandy a jejich datovými typy. Reprezentace je velmi nízkoúrovňová.

Více informací je dostupných v dokumentaci projektu LLVM: [3].

[1] Instruction Set Architecture C – C pro popis instrukční sady architektury

[2] Low Level Virtual Machine – nízkoúrovňový virtuální stroj

Přední část (Front-end)

Na počátku zpětného překladu je převeden vstupní binární soubor do interního formátu LOFF (Lissom obj. file format). Tím se zajistí další uniformní zpracování programu. Dalším krokem je samotné spuštění front-endu. Ten přebírá na vstupu nový binární soubor a popis architektury v jazyce ISAC spolu se signaturami knihoven a popisem typů známých funkcí.

Po případném rozeznání knihovních funkcí a zpracování dostupných ladicích informací je dalším úkolem přední části zpětného překladače vygenerování vnitřní reprezentace z architektonicky specifického strojového kódu pomocí instrukčního dekodéru. Tato část pracuje podobně jako disassembler. Výstupem ovšem není konkrétní jazyk symbolických instrukcí, ale sémantický popis instrukcí programu.

V praxi je přední část spojena s prostřední (optimalizační) částí v jeden celek. Následuje tedy celá řada analýz zaměřených na odstranění přebytečného kódu, vylepšení vlastností interní reprezentace a její připravení pro konečnou fázi generování výsledného kódu – tedy optimalizace tak jak by je prováděl oddělený middle-end, ovšem zpětný překladač se vyhne zbytečnému ukládání a načítání LLVM IR, ke kterému by jinak došlo. V této části také dochází k detekci funkcí, vyhledání všech funkcí dostupných ze vstupního bodu (obvykle funkce main) a detekce typů. Finálním výstupem je pak LLVM IR připravená ke zpracování back-endem do vysokoúrovňového jazyka.

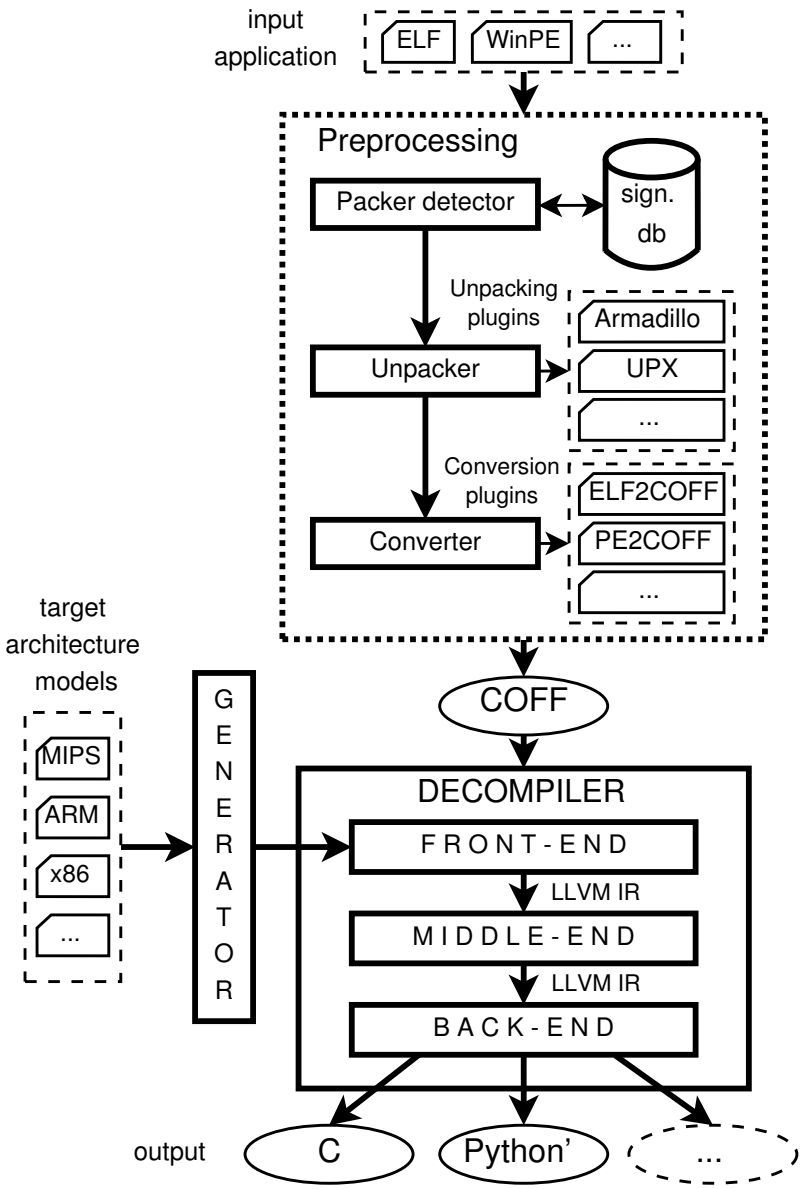
Výstupní část (Back-end)

Druhou (pokud uvažujeme rozdělení přední a prostření části, třetí) fází zpětného překladu je zpracování optimalizovaného LLVM IR z výstupu přední části překladače a emise vysokoúrovňového kódu. Zadní část překladače je implementována v rámci původního zdrojového kódu LLVM přidáním nového cílového jazyka. V současné době je podporován výstup do jazyka C a jazyka Python, doplněného o některé koncepty z jazyka C, jako jsou např. ukazatele.

BIR

Zadní část zpětného překladače používá vlastní interní reprezentaci, do které na počátku převede LLVM IR z přední části. Backend IR (zkráceně BIR) nemá textovou podobu, a je používána jako vstup pro převod do výstupního, vysokoúrovňového jazyka. Objekty BIR jsou odvozeny od některé z abstraktních bázových tříd uvedených na diagramu 5.4.

Struktura BIR je podobná abstraktnímu suntaktickému stromu (AST) – objekty reprezentující funkce dále obsahují strukturované příkazy (odvozené od třídy Statement) a výrazy (odvozené od třídy Expression). Strukturované proto, že při převodu z LLVM IR dochází k rozeznání obvyklých řídicích struktur vysokoúrovňových jazyků – v praxi tak BIR zahrnuje příkazy pro podmíky a cykly, které jsou kořeny podstromů obsahujících další příkazy. BIR dále má vlastní reprezentaci pro typy a konstanty. Kompletnější dokumentace je dostupná v podobě interní technické zprávy [29], případně ve formě zdrojového kódu.



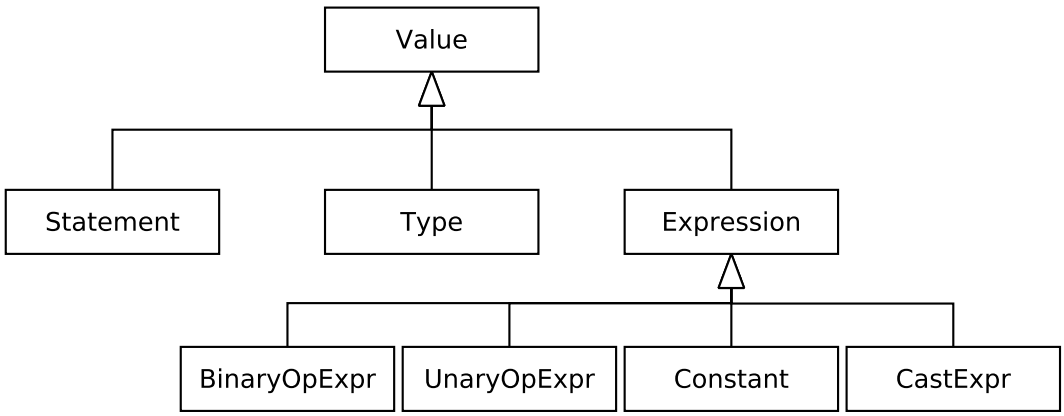
Obrázek 5.1: Zpětný překladač projektu Lissom [8].

```c
#include <stdio.h>
int main( )
{
    printf("Hello World!\n");
}
```

Obrázek 5.2: Vstup v jazyce C.

```
@.str = private constant [13 x i8] c"Hello World!\00", align 1 ;
define i32 @main() ssp
{
    entry:
    %retval = alloca i32
    %0 = alloca i32
    %"alloca point" = bitcast i32 0 to i32
    %1 = call i32 @puts(i8* getelementptr inbounds ([13 x i8]* @.str
        , i64 0, i64 0))
    store i32 0, i32* %0, align 4
    %2 = load i32* %0, align 4
    store i32 %2, i32* %retval, align 4
    br label %return
    return:
    %retval1 = load i32* %retval
    ret i32 %retval1
}
declare i32 @puts(i8*)
```

Obrázek 5.3: Hello World v LLVM IR.



Obrázek 5.4: Abstraktní bázové třídy BIR [29].

Kapitola 6

Návrh řešení

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 7

Implementace

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 8

Výsledky a testy

Produktem stávajícího zpětného překladu je Javě podobný kód doplněný o příkaz `goto`, přiřazení identity a bez strukturovaného toku řízení. Vytvoření bloků kódu pro cykly a podmínky je i v původním zpětném překladači řešeno až při překladu z LLVM IR do BIR. Obdobná strategie bude použita i zde.

8.1 Testování

Pro testování zpětného překladu je využit vcelku jednoduchý, ale účinný postup. Po instalaci (`make install`) jsou ve složce `src` zdrojové kódy jednotlivých testů v jazyce Java. Ve složce `compare` jsou očekávané výstupy zpětného překladače. K překladu testovacích vstupů slouží skript `compile_test.sh`, který vytvoří složku `bin` a umístí do ní přeložené třídy (`.class` soubory). Skript `decompile_test.sh` pak spustí zpětný překladač pro každou třídu, uloží výstup do složky `decomp` a porovná jej s existujícím výstupem ve složce `compare`. Pokud je mezi novým a očekávaným výstupem rozdíl, nebo během překladu dojde k chybě, je toto oznámeno ve výstupu. Testy jsem přidával podle potřeby, za účelem pokrytí instrukční sady JVM a řešení konkrétních problémů zjištěných při použití zpětného překladače na větší objemy bajtkódu (různé Java aplikace, včetně např. hry Minecraft [13]).

Je jednoduché nahradit obsah složky `bin` jiným kódem – například převedeným pomocí nástroje dex2jar z bajtkódu pro DVM, nebo rozbaleným `.jar` archivem.

Pro každý druh testu z následujících si ukážeme a analyzujeme výstup zpětného překladače. Ukázka původního a zpětně přeloženého kódu je v příloze A.

- Jednoduché třídy a základy
  - SimpleClass. Jednoduchá třída s jedinou statickou metodou `main`, která nedělá nic.
  - SimpleEnum. Jednoduchá enumerace barev.
  - HelloWorld
  - Constructors. Test na více konstruktorů v jedné třídě.
  - StaticConstField. Test na konstantní hodnoty členských proměnných.
  - MathOps. Různé výrazy pracující s čísly.
- Testy řízení toku

  - SimpleIf, IfTest, IfThenElse, IfThenElseInt. Podmíněné skoky v několika obměnách.
  - Switches. Test na příkazy switch (instrukce lookupswitch a tableswitch).
  - WhileLoop, DoWhile, ForLoop. Testy na různé druhy cyklů.
  - FlowTest. Komplexní řízení toku zahrnující výjimky.
- Volání metod
  - Basics. Jednoduché volání metody.
  - Recurses. Test na rekurzi.
- Testy polí
  - ArrayInit. Test na různé způsoby inicializace polí (statické, v konstruktoru). Vedle základní práce s poli dobře ilustruje problémy s inicializací proměnných v konstruktorech. Zpětně přeložený kód je vzhledem k původnímu neúměrně rozsáhlý.
  - ArrayTest. Test na zanořená pole (a tedy hlavně správné typy polí).
- Anonymní a zanořené třídy – AnonymousClassTest, AttributeTestClassEM01, AttributeTestClassEM02
- Testy anotací
  - MarkedType, MarkerAnnotation, MarkerAnnotationInvisible. Základní test virtuálním strojem viditelných a neviditelných anotací.
  - AnnotatedFields. Anotace aplikované na proměnné tříd.
  - AnnotatedWithCombinedAnnotation, CombinedAnnotation. Příklad na komplexní anotace.
  - AnnotatedWithEnumClass, AnnotationEnumElement. Enumerace použité v anotacích.
  - SimpleAnnotatedClass, SimpleAnnotation, ComplexAnnotatedClass, ComplexAnnotation. Jednoduché a komplexní anotace třídy – příklady ilustrují funkčnost čtení a použití anotací.

Jednoduché třídy a základy – `HelloWorld`

Z příkladu je zřejmé, že na zpracování výstupu by šlo dále pracovat. Propagace konstant a kopií by velmi prospěla čitelnosti kódu – daly by se odstranit všechny příkazy přiřazení a také příkaz identity pro první parametr metody. Specificky pro Javu by šel odstranit prefix jmenného prostoru `java.lang`, který je vždy implicitně k dispozici. I tak je výsledný kód definitivně na vyšší úrovni, než kód produkovaný disassemblerem. Zmizely instrukce, neviditelný zásobník a s ním i nutnost znát chování jednotlivých instrukcí.

Testy řízení toku – `IfTest`

Podobně jako `HelloWorld` by i zde prospěly optimalizace. Zpětný překladač korektně značí návěští pro skoky a také je vidět správná práce se zásobníkem – typ *double* korektně zabírá dvě pozice na zásobníku.

Volání metod – `Recurses`

Vedle rekurzivního volání je v tomto příkladu ilustrován jeden nedostatek, který bude potřeba odstranit – při překladu z jazyka Java se ztratil syntaktický cukr ve formě konkatenace řetězců pomocí operátoru +. Ten je nahrazen použitím třídy `StringBuilder` a jejích metod `append` a `toString`. Opět je zde prostor pro vylepšení – podporu rozeznání často používaných vzorů v kódu a jejich nahrazením za něco podobnějšího původnímu kódu.

Anonymní a zanořené třídy – `AnonymousClassTest`

Java umožňuje zanořovat třídy do jiných tříd a metod. V tomto příkladu je použita anonymní třída uvnitř metody `foo` a dvě zanořené třídy. V bajtkódu je tato konstrukce rozdělena na čtyři samostatné třídy, pojmenované na základě té původní.

Anotace – `ComplexAnnotatedClass` a `ComplexAnnotatedClass`

Zde je výstup nejblíže původnímu vstupu. Je to dáno tím, že anotace procházejí celým řetězem od zdrojového kódu až po výstup zpětného překladače téměř nezměněny.

Použití na bajtkód DVM v kombinaci s dex2jar

Jako součást testů jsem začlenil i příklad `Hello World` pro platformu DVM. Jedná se o jednoduchou aplikaci, která po stisknutí tlačítka zobrazí neintruzivní hlášení „Hello World“. Zpětným překladačem získaný kód je umístěn v příloze B. Aplikace sestává z několika objektů:

- Hlavní třída odvozená od `android.app.Activity`, která implementuje vytváří pohled na aplikaci a objekt pro zachycení událostí (stisknutí tlačítka).
- Objekt pro zachycení událostí, implementovaný jako anonymní podtřída. Při stisknutí tlačítka reaguje zobrazením hlášení.
- Třída `R` a její podtřídy, přidané při překladu pro propojení se zdroji (aplikace má uživatelské rozhaní, ikonu, apod.).

Zdrojový kód je v rámci možností čitelný a umožňuje analýzu. Podobně jako u ostatních příkladů, absence optimalizací by činila analýzu rozsáhlejších tříd obtížnou. Chybí zpracování zdrojů aplikace, ale toto je mimo rozsah projektu. Na přiloženém CD je jak aplikace `dex2jar`, tak tento příklad ve formátu APK a jako jednolivé již do JVM bajtkódu převedené třídy.

8.2 Výkon

Výkon testuji na stroji s těmito parametry:

- Intel(R) Core(TM)2 Duo CPU E8600 @ 3.33GHz
- 8GB RAM
- Arch Linux x86_64 (aktuální k datu 28.5.2013)
- Použitý překladač: GCC v4.8.0, optimalizace -O3

Zpětný překlad jednotlivých tříd lze v současné podobě projektu považovat téměř za okamžitý – pro všechny testy používané v projektu je čas strávený zpětným překladem téměř neměřitelný. Výpis 8.1 je obvyklým jevem.

```
Decompiling minecraft//zz
Running phase: Parsing class  ( 0s )
Running phase: Producing simple IR  ( 0s )
Running phase: Running optimizations  ( 0s )
Running phase: Writing output  ( 0s )
*** Decompiled successfully ***
```

Obrázek 8.1: Zpětný překlad samostatné třídy – výstup do terminálu.

Pro testování výkonu je tedy potřeba použít většího objemu vstupního bajtkódu. Ideálním cílem je dříve již zmínený Minecraft [13]. Jako druhý cíl jsem zvolil Java knihovnu guava [6]. Výsledky jsou zprůměrovány z pěti měření, kde první nebylo započítáno.

Guava je knihovnu vyvíjenou firmou Google - obsahuje kolekce, podporu pro vyrovnávací paměť, funkce pro práci s řetězci a celou řadu dalších podobných vlastností. Po rozbalení JAR archivu je přibližně 4,8 MB velká. Zpětný překladač ji zpracuje během 12.9s bez jakýchkoliv chyb.

Minecraft je kombinace hry a voxelového kreslení. Po rozbalení JAR archivu a odstranění zvuků, textur a jiných částí, které nejsou bajtkód je přibližně 3,5 MB velká. Na rozdíl of knihovny Guava používa extenzivně obfuskaci – všechny názvy jsou převedeny na jedno až tří-písmenné zkratky. Zpětný překladač ji zpracuje průměrně za 16.5s, opět bez chyb.

Nepochybně by se dalo dále optimalizovat – např. zpětný překladač je spouštěn pro každý soubor se třídou znova. Navíc při zachování nazávislosti tříd je možné povádět zpětný překlad více tříd paralelně.

Kapitola 9

Závěr

Cílem práce bylo v prvé řadě seznámit se s technikami a prostředky pro zpětný překlad bajtkódu. Druhým cílem bylo seznámit se zpětným překladačem vyvíjeným v rámci projektu Lissom a s jím používanou interní reprezentací LLVM IR. Třetí cíl bylo seznámit se alespoň se dvěma architekurami používajícími bajtkód pro uložení spustitelného kódu – práce obsahuje analýzu platforem JVM a DVM a v rámci čtvrtého úkolu návrh a implementaci zpětného překladače, který podporuje tyto platformy (JVM přímo, DVM převodem na bajtkód JVM běžně dostupným nástrojem `dex2jar`)

Pátým a posledním úkolem bylo implementované řešení otestovat, zhodnotit dosažené výsledky a diskutovat budoucí vývoj. Testování je diskutováno v předchozí kapitole. Budoucí vývoj je tématem následující sekce.

9.1 Náměty na další práci a otevřené problémy

Po převodu vstupního bajtkódu na SIR bude vhodné provést další transformace a optimalizace. Výstup je prozatím velmi zdlouhavý, protože instrukce vstupního bajtkódu jsou často převedeny na více příkazů SIR. Bylo by možné provést propagaci kopií a konstant a odstanit tak velké množství přiřazení, která převodem vznikají. Toto je zejména případ JVM bajtkódu, kde téměř každá operace zahrnuje práci se zásobníkem.

Dalším námětem na transformace SIR je typová inference – bez té se sice dá obejít a i tak mít poměrně čitelný výstupní kód (podobně jako je možné mít čitelný kód např. v javascriptu), ale bylo by vhodné znát přesné typy i pro další možné analýzy. Ve spojení s typovou inferencí je nutné i rozdělit pojmenované lokální proměnné, pokud jsou proměnné v původním bajtkódu znovupoužívány pro více typů.

Rozšiřitelnost SIR je zjevná – v základu se jedná o tříadresný kód. Pokud by vstupní architektura používala zásobník, je možné provést podobné transformace jako ty použité pro bajtkód JVM. U architektur používajících registry je konverze na SIR samozřejmě mnohem jednodušší. V obou případech je jisté, že by se interní reprezentace musela doplnit o podporu dalších typů, výrazů a vysokoúrovňových objektů – např. moduly, jmenné prostory, funkce, globální proměnné, nové operátory apod.

Podprogramy a výjimky

V současném stavu zpětný překladač nijak nepracuje s výjimkami. Tyto jsou zachovány v SIR v téměř nezměněné podobě a mají samozřejmě i vliv během převodu z bajtkódu.

Implementace výjimek v JVM se během času změnila – starší verze používají pro implementaci bloků *finally* podprogramy (v bajtkódu volané instrukcí `jsr` s návratem pomocí `ret`), zatímco novější verze tento problém řeší přímým vložením kódu těchto bloků na místa, kde jsou potřeba. V novějších verzích jsou navíc podprogramy označeny jako zastaralé (deprecated). Pro zpětný překladač tak vzniká problém: jak se s tímto vyrovnat? Pravděpodobně nejschůdnějším řešením by bylo odstranit volání podprogramů a získat tak podobný výsledek, jako nové verze překladačů. Při zpětném překladu ovšem chceme získat zpět (pokud možno) původní zápis výjimek a podprogramy toto zjednodušují. Blok *finally* odpovídá jednomu podprogramu (v ideálních podmínkách, pokud někdo nepoužil podprogramy za účelem zmatení zpětného překladu). Je však jisté, že podprogramy se v novém (slušném) bajtkódu nevyskytují. Rozeznávání výjimek tak musí počítat s rozkopírovanými bloky *finally*.

Druhou zvláštností výjimek v bajtkódu je, že na rozdíl od jazyka Java se mohou překrývat a nemusí striktně kopírovat normální řízení toku. Jediné, co virtuální stroj vidí je, že na určitém rozsahu adres v bajtkódu jsou zachycovány výjimky určitého typu. Jediná nutná podmínka, kterou musí obsluha výjimek splňovat je, že kód výjimku obsluhující musí být kompatibilní co se týče obsahu zásobníku. Obsluha výjimky implikuje vyprázdnění zásobníku a uložení reference na výjimku na jeho vrchol. Obecné řešení tak vůbec není triviální a jakékoliv předpoklady založené na znalosti existujících překladačů jsou *obecně* mylné.

Explicitní vstup a výstup z monitorů

Obdobný problém jako u výjimek je i u instrukcí (a příkazů SIR) pro explicitní vstup a výstup z monitorů. Zatímco v kódu jazyka Java se jedná o klasické závorky, nekřížící se s normálním tokem řízení, na úrovni bajtkódu toto není nijak vyžadováno. Opět by se dalo řešení založit na znalosti existujících překladačů a jejich chování, to by ovšem způsobilo zranitelnost ze strany různých obfuskátorů. Z tohoto důvodu zatím zpětný překladač podobně jako výjimky explicitní synchronizaci nijak neřeší a pouze převádí instrukce `monitorenter` a `monitorexit` na obdobné příkazy v SIR.

Zpracování konstruktorů

Při překladu z Javy (překladačem `javac`) se konstantní hodnoty členských proměnných zpracovávají tak, že statické proměnné jsou inicializovány ve statickém konstruktoru – v bajtkódu má název `<clinit>`, v Javě je to blok `static` umístěný v definici třídy. Normální členské proměnné jsou inicializovány v každém konstruktoru (`<init>`). Výsledkem je méně přehledný a duplikovaný kód. Pro jeho odstranění je potřeba do určité míry interpretovat kód konstruktorů a zjistit jeho efekt na členské proměnné. Tam, kde jsou inicializovány ve všech konstruktorech pomocí stejných konstantních hodnot, je možné tyto doplnit k definicím členských proměnných a kód odstranit. Tato optimalizace by citelně zlepšila kvalitu výstupu u tříd, které obsahují např. inicializaci velkých polí.

Podpora zanořených a anonymních tříd, hierarchie typů

Zde se jedná o načítání více souborů s třídami zároveň. Zanořené a anonymní třídy jsou propojeny s nadřazenou metodou nebo třídou a k analýze tohoto vztahu je nutné opustit zpracovávání bajtkódu po jednotlivých třídách. Tato část není příliš náročná – je ale potřeba jít dále a rekonstruovat pokud možno celou hierarchii tříd které jsou v rámci analyzované

třídy použité. Toto vyžaduje daleko více úsilí – načítání tříd z knihoven běhového prostředí, apod.

Literatura

[1] Zákon č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

[2] Soviet Union Impounds and Copies B-29 [online]. 2006-02-04 [cit. 2012-12-19]. URL http://www.nationalmuseum.af.mil/factsheets/factsheet.asp?id=1852

[3] Dokumentace projektu LLVM. URL http://llvm.org/docs/

[4] Dupuy, E.: Web JD-Gui [online]. 2012 [cit. 2012-12-20]. URL http://java.decompiler.free.fr/

[5] Emmerik, M. J. V.: *Static Single Assignment for Decompilation*. Dizertační práce, University of Queensland, Brisbane, QLD, AU, 2007.

[6] Google: Aktuální verze knihovny Guava [online]. 2013. URL https://code.google.com/p/guava-libraries/

[7] Huang, J.: Understanding the Dalvik Virtual Machine. GTUG Taipei, 2012.

[8] Kábele, B.: *Metody detekce funkcí při zpětném překladu kódu*. Diplomová práce, FIT VUT v Brně, 2012.

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

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

[11] Maidanski, I.: Nástroj JadRetro [online]. 2012 [cit. 2012-12-21]. URL http://jadretro.sourceforge.net/

[12] Miecznikowski, J.: *New algorithms for a Java decompiler and their implementation in Soot*. Diplomová práce, McGill University, April 2003.

[13] Mojang: Aktuální verze hry Minecraft [online]. 2013. URL http://assets.minecraft.net/1_5_2/minecraft.jar

[14] Naeem, N. A.: *Programmer-friendly decompiled Java*. Diplomová práce, McGill University, August 2006.

[15] Nolan, G.: *Decompiling Java Language*. Apress, 2004, ISBN 978-1-59059-265-6.

[16] Octeau, D.; Jha, S.; McDaniel, P.: Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, November 2012.
URL http://siis.cse.psu.edu/dare/papers/octeau-fse12.pdf

[17] Oracle: Learn about Java Technology [online]. 2012 [cit. 2012-01-07].
URL http://www.java.com/en/about/

[18] Oracle: The class File Format [online]. 2012 [cit. 2012-12-20].
URL http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[19] Oracle: JAR File Specification [online]. 2012 [cit. 2012-12-20].
URL http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html

[20] PKWARE: .ZIP File Format Specification [online]. 2012-09-01 [cit. 2012-12-20].
URL http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[21] Project, T. A. O. S.: .dex – Dalvik Executable Format [online]. `http://source.android.com/tech/dalvik/dex-format.html`, 2007 [cit. 2012-12-26].

[22] Quietust: Chip Images [online]. 2012-10-28 [cit. 2012-12-19].
URL http://www.qmtpro.com/~nes/chipimages/

[23] Smith, E.: Mocha, the Java Decompiler [online]. 2007 [cit. 2012-12-19].
URL http://www.brouhaha.com/~eric/software/mocha/

[24] Sochor, J.: Údržba softwaru. *Zpravodaj ÚVT MU*, ročník VI, č. 3, 1996: s. 15–20, ISSN 1212-0901.

[25] Team, T. M.: Projekt Minecraft Coder Pack [online]. 29.11.2012 [cit. 2012-01-08].
URL http://mcp.ocean-labs.de/index.php/Main_Page

[26] Tridgell, A.; the Samba Team: A bit of history and a bit of fun [online]. 1997-6-27 [cit. 2012-12-19].
URL http://www.rxn.com/services/faq/smb/samba.history.txt

[27] Varaneckas, T.: Mirror dekompilátoru JAD [online]. 2012 [cit. 2012-12-20].
URL http://www.varaneckas.com/jad/

[28] Wiśniewski, R.; Tumbleson, C.: android-apktool [online]. 28.12.2012 [cit. 2012-01-08].
URL http://code.google.com/p/android-apktool/

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

[30] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Design of an Automatically Generated Retargetable Decompiler. In *2nd European Conference of COMPUTER SCIENCE (ECCS'11)*, North Atlantic University Union, 2011, ISBN 978-1-61804-056-5, s. 199–204.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9640

Příloha A

Ukázky použitých testů a výstupu

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Obrázek A.1: HelloWorld.java - původní.

```
/*
 * Class decompiled with bc-decomp.
 * Original file: bin/HelloWorld.class
 */
public synchronized class HelloWorld
{
    // Methods
    public static void main(java.lang.String[] param_0)
    {
        var local0, stack1, stack2;
        local0 := param_0;
        stack1 = java.lang.System.out;
        stack2 = "Hello World";
        stack1.println(stack2);
        return;
    }
}
```

Obrázek A.2: HelloWorld.java - po zpětném překladu.

```
public class IfTest {
    public static void main(String[] args) {
        double test = -3.14159;
        if(test > 100.0 && test < 200.0)
            test = test - 100.0;
        return;
    }
}
```

Obrázek A.3: IfTest.java - původní.

```
/*
 * Class decompiled with bc-decomp.
 * Original file: bin/IfTest.class
 */
public synchronized class IfTest
{
    // Methods
    public static void main(java.lang.String[] param_0)
    {
        var local0, local1, stack1, stack3;
        local0 := param_0;
        stack1 = -3.14159;
        local1 = stack1;
        stack1 = local1;
        stack3 = 100;
        stack1 = stack1 cmpl stack3;
        if (stack1 <= 0) goto label_0;
        stack1 = local1;
        stack3 = 200;
        stack1 = stack1 cmpg stack3;
        if (stack1 >= 0) goto label_0;
        stack1 = local1;
        stack3 = 100;
        stack1 = stack1 - stack3;
        local1 = stack1;
label_0:
        return;
    }
}
```

Obrázek A.4: IfTest.java - po zpětném překladu.

```
public class Recurses {
    public static void main(String[] args) {
        System.out.println(recurse(25));
        return;
    }
    public static String recurse(int num) {
        if(num != 0)
            return "crap! " + recurse(num -1);
        else
            return "dammit!";
    }
}
```

Obrázek A.5: Recurses.java - původní.

```
/*
 * Class decompiled with bc-decomp.
 * Original file: bin/Recurses.class
 */
public synchronized class Recurses
{
    // Methods
    public static void main(java.lang.String[] param_0)
    {
        var local0, stack1, stack2;
        local0 := param_0;
        stack1 = java.lang.System.out;
        stack2 = 25;
        stack2 = Recurses.recurse(stack2);
        stack1.println(stack2);
        return;
    }
    public static java.lang.String recurse(int param_0)
    {
        var local0, stack1, stack2, stack3;
        local0 := param_0;
        stack1 = local0;
        if (stack1 == 0) goto label_0;
        stack1 = new java.lang.StringBuilder;
        stack2 = stack1;
        stack2.StringBuilder();
        stack2 = "crap! ";
        stack1 = stack1.append(stack2);
        stack2 = local0;
        stack3 = 1;
        stack2 = stack2 - stack3;
        stack2 = Recurses.recurse(stack2);
        stack1 = stack1.append(stack2);
        stack1 = stack1.toString();
        return stack1;
label_0:
        stack1 = "dammit!";
        return stack1;
    }
}
```

Obrázek A.6: Recurses.java - po zpětném překladu.

```
public class AnonymousClassTest
{
    public void foo()
    {
        new Runnable() {
            public void run() {
            }
        } .run();
    }

    class X
    {
    }

    static class Y
    {
    }
}
```

Obrázek A.7: AnonymousClassTest.java - původní.

```
// AnonymousClassTest.java
public synchronized class AnonymousClassTest
{
    // Methods
    public void foo()
    {
        var local0, stack1, stack2, stack3;
        local0 := this;
        stack1 = new AnonymousClassTest$1;
        stack2 = stack1;
        stack3 = local0;
        stack2.AnonymousClassTest$1(stack3);
        stack1.run();
        return;
    }
}

// AnonymousClassTest$1.java
synchronized class AnonymousClassTest$1 implements java.lang.
   Runnable
{
    // Fields
    final synthetic AnonymousClassTest this$0;
    // Methods
    AnonymousClassTest$1(AnonymousClassTest param_0)
    {
        var local0, local1, stack1, stack2;
        local0 := this;
        local1 := param_0;
        stack1 = local0;
        stack2 = local1;
        stack1.this$0 = stack2;
        stack1 = local0;
        stack1.Object();
        return;
    }
    public void run()
    {
        var local0;
        local0 := this;
        return;
    }

}

 ...
```

Obrázek A.8: AnonymousClassTest (více tříd) - po zpětném překladu.

```
@ComplexAnnotation ( ival = 4, bval = 2, cval = '5', fval = 3.0f,
    dval = 33.4, zval = false, jval = 56, sval = 99 )
public class ComplexAnnotatedClass
{
}

@Retention ( RetentionPolicy.RUNTIME )
public @interface ComplexAnnotation
{
    int ival();

    byte bval();

    char cval();

    long jval();

    double dval();

    boolean zval();

    short sval();

    float fval();
}
```

Obrázek A.9: ComplexAnnotation - původní.

```
@ComplexAnnotation(ival = 4, bval = 2, cval = '5', fval = 3, dval =
    33.4, zval = 0, jval = 56, sval = 99)
public synchronized class ComplexAnnotatedClass
{
}

@java.lang.annotation.Retention(value = java.lang.annotation.
    RetentionPolicy.RUNTIME)
public abstract interface ComplexAnnotation implements java.lang.
    annotation.Annotation
{
    // Methods
    public abstract int ival();
    public abstract byte bval();
    public abstract char cval();
    public abstract long jval();
    public abstract double dval();
    public abstract boolean zval();
    public abstract short sval();
    public abstract float fval();
}
```

Obrázek A.10: ComplexAnnotation - po zpětném překladu.

Příloha B

Dekompilovaný kód platformy Android/DVM

```
/*
 * Class decompiled with bc-decomp.
 * Original file: bin-android/HelloWorld.class
 */
public synchronized class com.android.test.HelloWorld extends
    android.app.Activity
{
    // Methods
    public void onCreate(android.os.Bundle param_0)
    {
        var local0, local1, stack1, stack2, stack3, stack4;
        local0 := this;
        local1 := param_0;
        stack1 = local0;
        stack2 = local1;
        stack1.onCreate(stack2);
        stack1 = local0;
        stack2 = 2130903040; // R$layout.main
        stack1.setContentView(stack2);
        stack1 = local0;
        stack2 = 2131034114; // R$id.Button01
        stack1 = stack1.findViewById(stack2);
        stack1 = (android.widget.Button) stack1;
        stack2 = new com.android.test.HelloWorld$1;
        stack3 = stack2;
        stack4 = local0;
        stack3.HelloWorld$1(stack4);
        stack1.setOnClickListener(stack2);
        return;
    }

}
```

Obrázek B.1: Hello World - hlavní třída.

```
/*
 * Class decompiled with bc-decomp.
 * Original file: bin-android/HelloWorld$1.class
 */
synchronized class com.android.test.HelloWorld$1 implements android.
   view.View$OnClickListener
{
    // Fields
    final synthetic com.android.test.HelloWorld this$0;


    // Methods
    .HelloWorld$1(com.android.test.HelloWorld param_0)
    {
        var local0, local1, stack1, stack2;
        local0 := this;
        local1 := param_0;
        stack1 = local0;
        stack2 = local1;
        stack1.this$0 = stack2;
        stack1 = local0;
        stack1.Object();
        return;
    }
    public void onClick(android.view.View param_0)
    {
        var local0, local1, stack1, stack2, stack3;
        local0 := this;
        local1 := param_0;
        stack1 = local0;
        stack1 = stack1.this$0;
        stack2 = "Hello World";
        stack3 = 0;
        stack1 = android.widget.Toast.makeText(stack1, stack2,
           stack3);
        stack1.show();
        return;
    }

}
```

Obrázek B.2: Hello World - reakce na události.

```
// Original file: bin-android/R.class
public final synchronized class com.android.test.R
{
}

// Original file: bin-android/R$string.class
public final synchronized class com.android.test.R$string
{
    // Fields
    public static final int app_name = 2130968577;
    public static final int hello = 2130968576;
}

// Original file: bin-android/R$layout.class
public final synchronized class com.android.test.R$layout
{
    // Fields
    public static final int main = 2130903040;
}

// Original file: bin-android/R$id.class
public final synchronized class com.android.test.R$id
{
    // Fields
    public static final int Button01 = 2131034114;
    public static final int radioGroup1 = 2131034113;
    public static final int tableRow1 = 2131034112;
}

public final synchronized class com.android.test.R$drawable
{
    // Fields
    public static final int icon = 2130837504;
}

// Original file: bin-android/R$attr.class
public final synchronized class com.android.test.R$attr
{
}
```

Obrázek B.3: Hello World - třída R a její podtřídy pro provázání se zdroji.

Příloha C

Obsah CD a použití programu

Na CD se nachází zdrojové kódy vytvořené v rámci diplomové práce a testovací vstupy a výstupy výsledného programu bc-decomp. Program byl testován na platformě linux64, ale je přenositelný. K sestavení byl použit překladač GCC ve verzi 4.8.0 a projekt vyžaduje podporu standardu C++11.

- Kopie externího nástroje dex2jar je umístěna ve složce stejného jména.
- Projekt včetně testovacích dat je v `bc-decomp/`. Zde je možné použít program „make“.
  - Překlad programů proběhne použitím příkazu „make“.
  - Program bc-decfront se spustí s použitím testovacích vstupů příkazem „make test“.
  - Všechny produkty překladu a spustitelné soubory lze vymazat příkazem „make clean“.
  - Se sestaveným zpětným překladačem je možné použít skript „decompile.sh“, který má dva parametry – vstupní a výstupní složku – a slouží ke zpětnému překladu libovolného JVM bajtkódu. Skript pracuje rekurzivně (sestupuje do podsložek).
- V `bc-decomp/test/src` jsou testovací programy ve formě původního zdrojového kódu.
- V `bc-decomp/test/bin` jsou testovací programy v bajtkódu.
- `bc-decomp/test/decomp` je složka, kam budou umístěny testy po zpětném překladu.
- V `bc-decomp/test/compare` jsou předlohy pro porovnání výstupu zpětného překladače.
- `bc-decomp/test/bin-android` obsahuje příklad Hello World pro Android/DVM.

CD dále obsahuje zdrojový kód této zprávy a její znění ve formátu PDF:

- Tato zpráva je uložena v souboru `projekt.pdf`.
- Zdrojový kód zprávy je uložen ve složce `latex/`.