VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

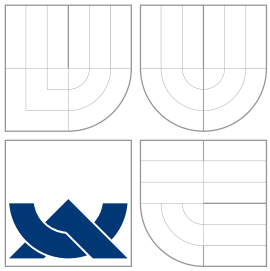
OPTIMALIZACE VELIKOSTI BAJTKÓDU JAVY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

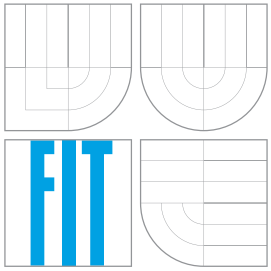
AUTOR PRÁCE
AUTHOR

Bc. VENDULA PONCOVÁ

BRNO 2016



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZACE VELIKOSTI BAJTKÓDU JAVY

JAVA BYTECODE SIZE OPTIMIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VENDULA PONCOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2016

Abstrakt

Tato práce je zaměřená na studium bajtkódu Javy z hlediska jeho velikosti. Popisuje virtuální stroj Javy a formát instrukčního souboru a uvádí přehled některých nástrojů pro manipulaci s bajtkódem. Pomocí těchto nástrojů jsem implementovala nástroj pro analýzu bajtkódu. Ze získaných dat se mi podařilo diagnostikovat místa, která jsou vhodná pro optimalizaci velikosti bajtkódu.

Abstract

This paper deals with the Java bytecode in terms of its size. It describes the Java Virtual Machine and the Java class file format. It also presents some tools for bytecode manipulation. Using these tools, I have designed and implemented a tool for bytecode analysis. From collected data, I have diagnosed locations suitable for size optimization.

Klíčová slova

Java, JVM, bajtkód, optimalizace velikosti, ASM, BCEL, Javassist

Keywords

Java, JVM, bytecode, size optimization, ASM, BCEL, Javassist

Citace

Vendula Poncová: Optimalizace velikosti bajtkódu Javy, diplomová práce, Brno, FIT VUT v Brně, 2016

Optimalizace velikosti bajtkódu Javy

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracovala samostatně pod vedením pana Ing. Radka Kočího, Ph.D. a pana Ing. Pavla Tišnovského, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Vendula Poncová
2. dubna 2016

Obsah

Kapitola 1

Úvod

V této práci se zabývám bajtkódem Javy z hlediska optimalizace jeho velikosti. V kapitole 2 popisuji virtuální stroj Java Virtual Machine a způsob, jakým je bajtkód interpretován. Kapitola 3 je věnovaná formátu, v jakém je bajtkód uložen v instrukčních souborech. Dále uvádím stručný popis nástrojů pro manipulaci s bajtkódem a shrnuji jejich výhody a nevýhody v kapitole 4. Konkrétně zmiňuji nástroje BCEL, ACM a Javassist. Kapitola 5 je věnovaná návrhu a implementaci nástroje pro analýzu bajtkódu. Pomocí tohoto nástroje jsem získala data, která jsem zpracovala a vyhodnotila v kapitole 6. Cílem této práce je seznámení se s bajtkódem a diagnostika míst vhodných k optimalizaci z hlediska velikosti.

Kapitola 2

Java Virtual Machine

Architektura Javy se podle Vennerse [5] skládá z programovacího jazyka Java, formátu instrukčního souboru, aplikačního programového rozhraní Java Application Programming Interface (Java API) a virtuálního stroje Java Virtual Machine (JVM). Pro psaní zdrojových kódů a jejich spouštění je zapotřebí všech těchto částí. Zdrojový kód zapsaný v programovacím jazyce Java je uložený v souboru s příponou `.java` (dále `java` souboru). Tento kód je při kompilaci převeden na mezikód, tzv. bajtkód, a uložen v souborech s příponou `.class` (dále `class` souborech). Bajtkód lze spustit pomocí virtuálního stroje, který má přístup k Java API. O tomto stroji lze hovořit z hlediska abstraktní specifikace nebo konkrétní implementace. Konkrétní implementace je závislá na daném systému a hardwaru, ale jednotná interpretace `class` souborů napříč platformami je zajištěna dodržením specifikace a platformově nezávislým formátem `class` souborů. Vzhledem k zaměření práce se v této kapitole zabývám abstraktní specifikací JVM dle dokumentu [2].

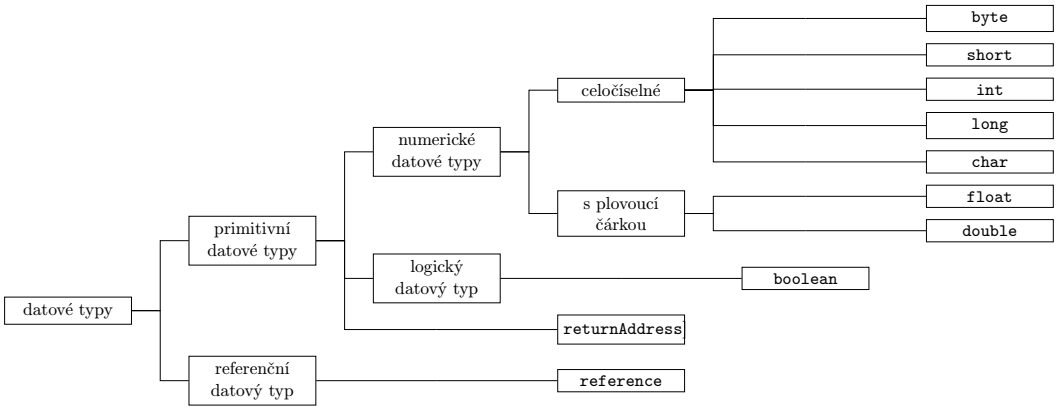
2.1 Datové typy a hodnoty

Datové typy a hodnoty podporované JVM jsou znázorněné na obrázku 2.1. Celá čísla jsou reprezentovaná datovými typy `byte` o délce 8 bitů, `short` o délce 16 bitů, `int` o délce 32 bitů nebo `long` o délce 64 bitů. Pro čísla s plovoucí čárkou jsou definovány typy `float` o 32 bitech a `double` o 64 bitech. Typ `boolean` reprezentuje pravdivostní hodnoty pravda a nepravda. Typ `returnAddress` reprezentuje ukazatel na instrukci v instrukčním souboru.

Znaky a řetězce jsou v Javě kódované podle standardu Unicode v kódování UTF-16, kde jeden znak je kódovaný jednou nebo dvěma kódovými jednotkami. Kódovou jednotku reprezentuje typ `char`. Jedná se o 16-bitové nezáporné číslo. Řetězec je pak reprezentovaný pomocí pole hodnot typu `char`. V instrukčním souboru jsou řetězcové konstanty kódované v modifikovaném kódování UTF-8.

Typ `reference` označuje referenční datový typ a reprezentuje referenci na dynamicky vytvořený objekt. Podle toho, zda objekt je instancí třídy, pole nebo instancí třídy či pole, které implementují nějaké rozhraní, se rozlišuje typ reference. Hodnotou typu `reference` může být též speciální hodnota `null`, tedy reference na žádný objekt.

Instrukční sada JVM je omezená a nenabízí u všech instrukcí podporu pro všechny datové typy. Celá čísla jsou primárně reprezentovaná datovými typy `int` a `long` a případně přetypovaná na jeden z typů `byte`, `short` či `char`. Pro datový typ `boolean` existuje jen podpora pro přetypování a pro vytvoření pole hodnot typu `boolean`. Pro výpočet logických výrazů se používá typ `int` s hodnotami 0 a 1.



Obrázek 2.1: Datové typy a hodnoty podporované JVM.

2.2 Paměťové oblasti

JVM pracuje s několika typy paměťových oblastí. Velikosti těchto oblastí mohou být pevně dané, nebo se mohou měnit dynamicky podle potřeby. Při spuštění JVM vzniká halda a oblast metod. Halda je paměť určená pro alokaci instancí tříd a polí. Alokovanou paměť nelze dealokovat explicitně. Haldu automaticky spravuje tzv. *garbage collector*. Oblast metod slouží k ukládání kompilovaného kódu. Pro každou načtenou třídu se do této paměti ukládají struktury definující tuto třídu. Jednou z těchto struktur je tzv. *run-time constant pool*. Jedná se o tabulku konstant z `class` souboru, kde jsou symbolické reference na třídy, metody a členské proměnné nahrazeny konkrétními referencemi. Více se o tabulce konstant zmiňuje kapitola 3.2.

Každá aplikace je spuštěna v samostatném vlákně a při běhu mohou vznikat a zanikat i další vlákna. Všechna taková vlákna sdílí přístup k haldě a oblasti metod. Navíc má každé vlákno k dispozici vlastní `pc` registr a zásobník rámců. V `pc` registru je uchováván ukazatel na aktuálně vykonávanou instrukci, není-li aktuálně vykonávaná metoda nativní. Zásobník rámců obsahuje data zavolaných metod. Při každém volání metody je vytvořen nový rámec. Rámec má vlastní pole lokálních proměnných a operační zásobník. V poli lokálních proměnných se uchovávají hodnoty parametrů a lokálních proměnných. Hodnoty lze vkládat na operační zásobník, provádět nad nimi výpočty a ukládat zpět do pole. Operační zásobník slouží k předávání operandů instrukcím a k uchovávání mezivýsledků. Podílí se také na předávání parametrů a návratových hodnot.

Před voláním metody je třeba nejprve vložit parametry na operační zásobník aktuálního rámce. Zavoláním metody se vytvoří nový rámec a umístí se na vrchol zásobníku rámců. Parametry se následně přesunou z operačního zásobníku předchozího rámce do pole lokálních proměnných nového rámce. Registr `pc` se nastaví na první instrukci volané metody a začnou se vykonávat jednotlivé instrukce. Při návratu z metody je návratová hodnota umístěná na vrcholu operačního zásobníku, vrací-li metoda nějakou hodnotu. Tato hodnota je přesunuta na operační zásobník předcházejícího rámce a aktuální rámec je ze zásobníku rámců odstraněn. Dojde k obnovení stavu volající metody. Registr `pc` je nastaven na index instrukce, která bezprostředně následuje za instrukcí volající metodu. Pokud je metoda ukončena vyvoláním nezachycené výjimky, pak k předání hodnoty nedochází. Aktuální rámec je odstraněn a výjimka je znovu vyvolaná ve volající metodě. Zpracování výjimek je více vysvětleno v kapitole 3.5.

Nejmenší jednotka, se kterou pracuje operační zásobník, je 32-bitová hodnota. Hodnoty

většiny datových typů lze vyjádřit pomocí jedné jednotky, ale hodnoty typů `long` a `double` je třeba reprezentovat dvěma jednotkami. S takovou dvojicí je třeba vždy manipulovat jako s celkem. Datový typ hodnoty na zásobníku je daný instrukcí, která ho tam vložila, a na hodnotu nelze nahlížet jinak. Hloubka operačního zásobníku je určená počtem jednotek na zásobníku. Proto se o jednotce dále zmiňuji jako o jednotce hloubky zásobníku.

Lokální proměnná v poli lokálních proměnných je 32-bitová hodnota. Proměnnou lze adresovat pomocí indexu do pole, kde pole je indexováno od nuly. Lokální proměnná může být typu `byte`, `short`, `int`, `char`, `float`, `boolean`, `reference` nebo `returnAddress`. Hodnoty typu `long` a `double` jsou uchovávané pomocí dvojice lokálních proměnných. V tom případě k adresaci slouží nižší z indexů a na větší index se nesmí přistupovat.

2.3 Kontrola instrukčního souboru

Při načítání `class` souboru je ověřeno správné formátování tak, jak je popsáno v kapitole 3. Jsou zkontrolovány první čtyři bajty souboru, předdefinované atributy musí být správné délky, názvy a typy tříd, rozhraní, metod a proměnných musí být validní, indexy do tabulky konstant musí adresovat správný typ položky. Dále jsou kladena jistá omezení na kód metod. Mimo jiné, argumenty instrukcí musí mít správný typ a musí být správného počtu, integrita hodnot typu `long` a `double` nemůže být nikdy narušena, k hodnotě lokální proměnné se nesmí přistupovat před inicializací proměnné, nesmí dojít k načtení hodnoty z prázdného zásobníku a všechny metody musí byt ukončené `return` instrukcí. Verifikace těchto omezení se provádí pomocí typové kontroly nebo typové inference. Kontrola těchto omezení se tak nemusí provádět za běhu programu.

Kapitola 3

Formát instrukčního souboru

Při kompilaci `java` souboru překladač pro každou definovanou třídu a rozhraní vytvoří jeden `class` soubor. Tento soubor obsahuje binární reprezentaci kompilovaného mezikódu, který lze interpretovat prostřednictvím JVM. V této kapitole popisuji formát `class` souboru dle specifikace ve verzi Java SE 8 Edition [2].

Pro popis formátu jsem zvolila rozšířenou Backus-Naurovu formu, která umožňuje zapsat syntaxi formálního jazyka pomocí pravidel a terminálních a nonterminálních symbolů. Nonterminální symboly jsou definovány pomocí definujícího symbolu `:=`, symbolu pro konkatenaci `,`, symbolu pro alternaci |, symbolů pro nula a více opakování {}, ukončujícího symbolu `;` a pomocí graficky odlišených `terminálních` a *nonterminálních* symbolů. Symboly popisují jednotlivé struktury, ze kterých se `class` soubor skládá.

3.1 Základní struktura

Položky `class` souboru tvoří posloupnost osmibitových bajtů. Základní stavební jednotkou je tedy bajt, který je v pravidlech reprezentovaný symbolem *B*. Symbol $\langle n \rangle B$, kde $\langle n \rangle \in \{2, 3, \dots\}$, reprezentuje n bajtů. Terminály začínají prefixem `0x` a jsou hexadecimální reprezentací posloupnosti bajtů.

Základní struktura souboru je popsaná symbolem *classfile*. Soubor obsahuje informace o svém typu a verzi (*version*), disponuje tabulkou všech konstant, které se v souboru vyskytují (*constants*), nese informace o třídě či rozhraní (*class*), které reprezentuje, obsahuje seznam rozhraní (*interface_list*), které reprezentovaná třída implementuje, případně rozhraní rozšiřuje, seznam členských proměnných (*field_list*), seznam metod (*method_list*) a seznam atributů (*attribute_list*).

classfile := *version*, *constants*, *class*, *interface_list*, *field_list*, *method_list*, *attribute_list*;

Typ souboru je definován prvními čtyřmi bajty, které jsou popsané symbolem *magic_number*. Verze souboru je tvořena hodnotou *M* symbolu *major_version* a hodnotou *m* symbolu *minor_version* jako *M.m*.

version := *magic_number*, *minor_version*, *major_version*;
magic_number := `0xCAFEBABE`;
minor_version := *2B*;
major_version := *2B*;

3.2 Konstanty

Tabulka konstant obsahuje některé číselné konstanty, všechny řetězce a symbolické informace o všech třídách, rozhraních, metodách a členských proměnných, které se v souboru, instrukcích i atributech vyskytují. Tato tabulka se nazývá *constant pool* a slouží v podstatě jako databáze dat, do které se pomocí indexů odkazují další položky souboru. Odkazem do tabulky konstant je tedy v dalším textu myšlen platný index do tabulky konstant adresující položku očekávaného typu.

Struktura tabulky konstant je popsána symbolem *constants*. Symbol *constant_pool_count* reprezentuje hodnotu $1 + n$, kde n je počet položek v tabulce konstant. Položky tabulky jsou indexované od jedné, neboť nultý index je vyhrazen pro odkaz na žádnou z položek. Samotná tabulka je reprezentovaná symbolem *constant_pool*.

constants := *constant_pool_count*, *constant_pool*;
constant_pool_count := *2B*;
constant_pool := { *constant_integer*
| *constant_float*
| *constant_long*
| *constant_double*
| *constant_utf8*
| *constant_string*
| *constant_nameAndType*
| *constant_class*
| *constant_fieldref*
| *constant_methodref*
| *constant_interfaceMethodref*
| *constant_methodHandle*
| *constant_methodType*
| *constant_invokeDynamic*
};

Každá položka tabulky je tvořena označením typu a posloupností bajtů s informacemi o položce. Položky mohou mít různou velikost v závislosti na svém typu a obsahu. Stejným způsobem jsou definovány všechny tabulky v `class` souboru. Jestliže se jedná o pole, pak prvky pole jsou stejného typu, a proto označení typu v prvcích chybí.

Číselné konstanty jsou reprezentované symboly *constant_integer*, *constant_float*, *constant_long* a *constant_double* a skládají se jen z typu a číselné hodnoty. Symbol *constant_utf8* reprezentuje řetězec v upraveném kódování `UTF-8` a skládá se z typu, délky pole bajtů a pole bajtů nesoucích reprezentaci řetězce. Znaky řetězce mohou být vhledem ke kódování tvořeny různými počty bajtů. Symbol *constant_string* je reprezentací řetězcové konstanty a kromě typu obsahuje index do tabulky konstant na položku *constant_utf8*. Třídy a rozhraní jsou reprezentované položkami *constant_class* s odkazy na jejich název (*constant_utf8*). Entity jako členské proměnné (i statické), metody třídy a metody rozhraní jsou reprezentované položkami *constant_fieldref*, *constant_methodref* a *constant_interfaceMethodref* obsahujícími odkaz na třídu, případně rozhraní, dané entity (*constant_class*), a odkaz na položku se jménem a typem této entity (*constant_nameAndType*). Jméno a typ v položce *constant_nameAndType* jsou odkazy na řetězce (*constant_utf8*). Položky *constant_methodHandle*, *constant_methodType* a *constant_invokeDynamic* souvisí s podporou dynamických jazyků.

Názvy tříd a rozhraní jsou interně uváděné v úplném tvaru, ale z historických důvodů se tečky nahrazují lomítky. Například, třída `Object` má úplný název `java.lang.Object` a interní název `java/lang/Object`. Typ proměnné nebo metody je specifikován řetězcem. Základní datové typy jsou reprezentované písmeny `B` pro `byte`, `C` pro `char`, `D` pro `double`, `F` pro `float`, `I` pro `int`, `J` pro `long`, `S` pro `short`, `Z` pro `boolean`. Typ reference na objekt je reprezentovaný řetězcem `L`$ClassName$`;`, kde $ClassName$ je interní název třídy nebo rozhraní. Typ reference na jednorozměrné pole je reprezentovaný řetězcem `[`$ComponentType$, kde $ComponentType$ je řetězec reprezentující základní datový typ, referenci na objekt nebo referenci na pole. Pomocí zanoření referencí na pole lze definovat referenci na vícerozměrné pole. Například, řetězec `Ljava/lang/Object;` označuje referenci na objekt typu `Object` a `[[[I` označuje referenci na trojrozměrné pole typu `int`. Typ metody je reprezentován řetězcem, který se skládá z výčtu typů formálních parametrů metody a typu její návratové hodnoty. Tedy například, typ metody s hlavičkou `int method(boolean b, Object o)` bude specifikovaný řetězcem `(BLjava/lang/Object;)I`. Nemá-li metoda žádné parametry či nevrací žádnou hodnotu, pak je chybějící typ nahrazen písmenem `V`.

3.3 Třída

Každý `class` soubor reprezentuje jednu třídu nebo rozhraní. Informace o reprezentované entitě jsou definované symbolem *class*. Položka *this_class* je odkazem na tuto entitu v tabulce konstant, *super_class* je odkaz na nadřazenou třídu a *access_flags* je bitové pole příznaků pro přístup k této entitě.

$$
\begin{aligned}
\textit{class} &:= \textit{access\_flags}, \textit{this\_class}, \textit{super\_class};\\
\textit{access\_flags} &:= \textit{2B};\\
\textit{this\_class} &:= \textit{class\_ref};\\
\textit{super\_class} &:= \textit{class\_ref};\\
\textit{class\_ref} &:= \textit{constant\_pool\_index};\\
\textit{constant\_pool\_index} &:= \textit{2B};
\end{aligned}
$$

Seznam rozhraní, které reprezentovaná třída implementuje, případně reprezentované rozhraní rozšiřuje, je definované v poli *interfaces* o *interface_count* prvcích. Prvky jsou odkazy do tabulky konstant na položky *constant_class* reprezentující nějaké rozhraní.

$$
\begin{aligned}
\textit{interface\_list} &:= \textit{interface\_count}, \textit{interfaces};\\
\textit{interfaces} &:= \{\ \textit{class\_ref}\ \};\\
\textit{interface\_count} &:= \textit{2B};
\end{aligned}
$$

3.4 Členské proměnné

Členské proměnné (proměnné třídy i proměnné instance) jsou definované v poli členských proměnných *fields* o *fields_count* prvcích. Každá proměnná má pole příznaků dané symbolem *access_flags*, jméno dané symbolem *name_ref*, typ daný symbolem *descriptor_ref* a seznam atributů daný symbolem *attribute_list*. Jméno a typ jsou reprezentované odkazem do tabulky konstant na položku *utf8_ref*. Atributům se věnuje kapitola 3.6.

$$
\begin{aligned}
\textit{field\_list} &:= \textit{fields\_count}, \textit{fields};\\
\textit{fields} &:= \{\ \textit{field\_info}\ \};\\
\textit{field\_info} &:= \textit{access\_flags}, \textit{name\_ref}, \textit{descriptor\_ref}, \textit{attribute\_list};\\
\textit{fields\_count} &:= \textit{2B};\\
\textit{name\_ref} &:= \textit{utf8\_ref};\\
\textit{descriptor\_ref} &:= \textit{utf8\_ref};\\
\textit{utf8\_ref} &:= \textit{constant\_pool\_index};
\end{aligned}
$$

3.5 Metody

Metody reprezentované třídy či rozhraní jsou definované v poli metod *methods* o *methods_count* prvcích. Stejně jako u členských proměnných jsou metody popsané bitovým polem příznaků, jménem, typem a seznamem atributů.

$$
\begin{aligned}
\textit{method\_list} &:= \textit{methods\_count}, \textit{methods};\\
\textit{methods} &:= \{\ \textit{method\_info}\ \};\\
\textit{method\_info} &:= \textit{access\_flags}, \textit{name\_ref}, \textit{descriptor\_ref}, \textit{attribute\_list};\\
\textit{methods\_count} &:= \textit{2B};
\end{aligned}
$$

Pokud metoda není abstraktní, pak jedním z jejích atributů je `Code` s kódem metody. Tento atribut je definován symbolem *code_attribute*. Položka *name_ref* je odkazem do tabulky konstant na řetězec "Code". Položka *attribute_length* určuje délku atributu v bajtech bez prvních šesti bajtů. Dále položky *max_stack* a *max_locals* označují maximální hloubku operačního zásobníku přepočtenou na jednotku hloubky a maximální počet lokálních proměnných včetně parametrů. Kód metody je reprezentovaným polem bajtů *code* o délce *code_length*. Symbol *attribute_list* je seznamem atributů.

$$
\begin{aligned}
\textit{code\_attribute} &:= \textit{name\_ref}, \textit{attribute\_length}, \textit{code\_info}\\
\textit{code\_info} &:= \textit{max\_stack}, \textit{max\_locals}, \textit{code\_list}, \textit{exception\_list}, \textit{attribute\_list};\\
\textit{code\_list} &:= \textit{code\_length}, \textit{code}\ ;\\
\textit{code} &:= \{\ B\ \};\\
\textit{max\_stack} &:= \textit{2B};\\
\textit{max\_locals} &:= \textit{4B};\\
\textit{code\_length} &:= \textit{4B}\ ;
\end{aligned}
$$

Informace o zpracování výjimek jsou dostupné v tabulce výjimek *exception_table* o délce *exception_table_length*. Každá položka tabulky obsahuje dva indexy *start_pc* a *end_pc* do pole *code*, jež společně definují blok instrukcí, pro který je odchycení dané výjimky aktivní. Dále obsahuje index *handler_pc* do pole *code* odkazující na začátek bloku pro zpracování výjimky a nakonec index *catch_type* do tabulky konstant na položku *constant_class* reprezentující typ odchycené výjimky. Jestliže je tento index nulový, pak jsou odchytávány všechny výjimky. Na pořadí položek v tabulce výjimek se nevztahují žádná omezení, neboť při odchytávání výjimky se postupuje od nejniternějšího bloku.

$$
\begin{aligned}
\textit{exception\_list} &:= \textit{exception\_table\_length}, \textit{exception\_table}\ ; \\
\textit{exception\_table} &:= \{\ \textit{start\_pc, end\_pc, handler\_pc, catch\_type}\ \}; \\
\textit{start\_pc} &:= \textit{code\_index}; \\
\textit{end\_pc} &:= \textit{code\_index}; \\
\textit{handler\_pc} &:= \textit{code\_index}; \\
\textit{catch\_pc} &:= \textit{class\_ref}; \\
\textit{exception\_table\_length} &:= \textit{2B}; \\
\textit{code\_index} &:= \textit{2B};
\end{aligned}
$$

3.6 Atributy

Reprezentovaná třída, případně rozhraní, metody, členské proměnné a i některé atributy mají definovaný seznam atributů. Seznam se skládá z tabulky atributů *attributes* o *attributes_count* položkách. Typ atributu je daný odkazem *name_ref* na název atributu, délka atributu bez prvních šesti bajtů je daná hodnotou *attribute_length*. Další informace, které atribut nese v položce *info*, se liší podle typu atributu.

$$
\begin{aligned}
\textit{attribute\_list} &:= \textit{attributes\_count}, \textit{attributes}; \\
\textit{attributes} &:= \{\ \textit{name\_ref}, \textit{attribute\_length}, \textit{info}\ \}; \\
\textit{info} &:= \{\ B\ \}; \\
\textit{attributes\_count} &:= \textit{2B}; \\
\textit{attribute\_length} &:= \textit{4B};
\end{aligned}
$$

Specifikace [2] definuje 23 atributů. Překladače však mohou definovat a vkládat do `class` souborů i atributy vlastní. Pokud je JVM neumí rozpoznat, pak je ignoruje. Atributy mají různou míru důležitosti vzhledem k interpretaci `class` souboru.

Pro správnou interpretaci virtuálním strojem je důležitých následujících pět atributů. Atribut `Code` s instrukcemi metody byl představen v kapitole 3.5. Jedním z atributů `Code` může být `StackMapTable`. Tento atribut je důležitý pro typovou kontrolu při verifikaci `class` souborů. Pro každý základní blok instrukcí jsou specifikovány typy lokálních proměnných a hodnot na operačním zásobníku. U starších verzí `class` souboru tento atribut chybí, a proto se provádí typová inference pomocí analýzy datového toku. `Exceptions` je atribut metody. Obsahuje odkazy do tabulky konstant na typy kontrolovaných výjimek, které metoda může vyhodit. Jedním z atributů členské proměnné může být `ConstantValue`, který v sobě nese index do tabulky konstant na položku s číselnou nebo řetězcovou konstantou. Jestliže je daná proměnná statická, pak je jí při inicializaci třídy přiřazena právě tato hodnota. Atribut `BootstrapMethods` souvisí s dynamickými jazyky.

Následujících dvanáct atributů je podstatných pro správnou interpretaci knihovnami Java API. Vnitřní třídy třídy reprezentované `class` souborem jsou vyjmenované v atributu `InnerClasses`. Pro každou vnitřní třídu atribut uchovává bitové pole příznaků, ukazatel na název vnitřní třídy, ukazatel na vnější třídu a ukazatel na vnitřní třídu. Každá lokální nebo anonymní třída pak musí mít atribut `EnclosingMethod` obsahující ukazatel na vnější třídu a ukazatel na metodu, která definici třídy uzavírá. Atribut `Synthetic` reprezentuje příznak, že daný člen třídy se nevyskytuje ve zdrojovém kódu a zároveň není standardním členem. Atribut `Signature` nese deklaraci třídy, rozhraní, členské proměnné nebo metody, v níž se vyskytují typové proměnné nebo parametrizované typy. Jména a přístupové příznaky formálních parametrů metody mohou být dostupné v atributu `MethodParameters`. Anotace jsou dostupné v atributech `Runtime`$Visibility Annotations$, kde $Visibility$ určuje, zda je

anotace viditelná (`Visible`) či neviditelná (`Invisible`) za běhu programu, a *Type* označuje jeden z typů anotací `Annotations`, `TypeAnnotations`, `ParameterAnnotations`. Atribut `AnnotationDefault` reprezentuje výchozí hodnotu elementu, který patří typu anotace.

Další atributy jsou pouze informativní a mohou sloužit například k ladění chyb ve zdrojovém souboru. Atribut `SourceFile` obsahuje odkaz na název zdrojového kódu. Atribut `SourceDebugExtension` v sobě nese řetězec s ladícími informace. `LineNumberTable` reprezentuje mapování indexů do pole instrukcí na čísla řádků zdrojového kódu. Informace o lokálních proměnných metody mohou být dostupné v atributu `LocalVariableTable`, kde je obsažen rozsah instrukcí, ve kterém proměnná nese hodnotu, odkaz na název proměnné, odkaz na typ proměnné a index do pole lokálních proměnných. Stejné informace nese atribut `LocalVariableTypeTable`, ale pouze pro proměnné, jejichž typy používají typované proměnné nebo parametrizované typy. Atribut `Deprecated` slouží k indikaci toho, že třída, metoda, členská proměnná či rozhraní jsou zastaralé.

3.7 Instrukce

Každá instrukce se skládá z jednobajtového operačního kódu *opcode* a nula a více operandů. Instrukce může dále pracovat s obsahem operačního zásobníku a má přístup do pole lokálních proměnných a tabulky konstant. Pro snazší orientaci je každému operačnímu kódu přiřazen jednoznačný název *mnemonic*. Součástí *mnemonic* může být označení datového typu, s jehož hodnotami instrukce pracuje. Typ je specifikován písmeny `i` pro `int`, `l` pro `long`, `s` pro `short`, `b` pro `byte`, `c` pro `char`, `f` pro `float`, `d` pro `double` a `a` pro `reference`. V následujícím textu jsou instrukce uvedené ve tvaru: *mnemonic* $operand_1$ $operand_2 \ldots operand_n$. Pokud nebude řečeno jinak, pak každý operand má velikost jednoho bajtu.

Konstantní hodnoty

Konstantní hodnotu lze dle jejího typu, velikosti a hodnoty vložit na zásobník několika způsoby. Instrukce `aconst_null` vloží na zásobník referenci na `null`. Instrukce `iconst_`*value*, kde $value \in \{\texttt{m1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{5}\}$ a `m1` označuje hodnotu -1, vloží na zásobník hodnotu *value* typu `int`. Obdobně lze instrukcí `fconst_`*value*, kde $value \in \{\texttt{0}, \texttt{1}, \texttt{2}\}$, a instrukcemi `lconst_`*value* a `dconst_`*value*, kde $value \in \{\texttt{0}, \texttt{1}\}$, vložit konstantní hodnoty typu `float`, `long` a `double`. Větší celočíselnou hodnotu typu `int` umožňují vložit instrukce `bipush` *value* a `sipush` *value*, kde *value* je jednobajtová, respektive dvoubajtová, celočíselná hodnota.

Ve všech ostatních případech, je nutné vložit hodnotu z tabulky konstant. Pomocí instrukce `ldc` *index*, kde *index* je jednobajtový index do tabulky konstant, lze vložit hodnotu typu `int` či `float` nebo referenci na objekt. Instrukce `ldc_w` *index* umožňuje použít dvoubajtový index. Instrukce `ldc2_w` *index* vloží na zásobník hodnotu typu `long` nebo `double` danou dvoubajtovým indexem *index*.

Práce s lokálními proměnnými

Do lokální proměnné lze přiřadit hodnotu instrukcí *t*`store` *index*, kde $t \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}, \texttt{a}\}$ a *index* je index do pole lokálních proměnných. Dané proměnné se přiřadí hodnota, která se odebere z vrcholu zásobníku. Na druhou stranu, instrukce *t*`load` *index*, načte hodnotu z dané lokální proměnné na zásobník. Pro lokální proměnné s indexy $index \in \{0, 1, 2, 3\}$ lze použít jednobajtové instrukce *t*`store_`*index* a *t*`load_`*index*. Celočíselné lokální proměnné

lze inkrementovat instrukcí `iinc` *index value*, která k hodnotě proměnné na indexu *index* přičte jednobajtovou celočíselnou hodnotu *value*.

Práce s polem

Pole lze vytvořit instrukcí `newarray` *type*, kde *type* označuje typ pole. Instrukce načte ze zásobníku hodnotu *count* typu `int`, vytvoří pole typu *type* o délce *count* a referenci na toto pole vloží na zásobník. Pole objektů umožňuje vytvořit instrukce `anewarray` *index*, kde *index* je dvoubajtový index do tabulky konstant na typ vytvářeného pole. Typem zde může být třída, rozhraní nebo pole. Obdobně lze vytvořit vícerozměrné pole objektů instrukcí `multianewarray` *index dimension*, kde *index* je opět index do tabulky konstant a *dimension* udává počet dimenzí vytvářeného pole. Ze zásobníku jsou načteny délky pro jednotlivé dimenze a je vložena reference na vícerozměrné pole objektů daného typu.

Instrukce *t*`astore`, kde $t \in \{\texttt{b}, \texttt{c}, \texttt{s}, \texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}, \texttt{a}\}$ určuje typ pole, umožňuje vložit hodnotu do pole. Ze zásobníku odebere hodnotu *value*, index *index* a referenci na pole *array* typu *t* a provede operaci $array[index] := value$. Instrukcí *t*`aload` lze hodnotu z pole načíst na zásobník. Ze zásobníku se odebere *index* a *array* a vloží se na něj hodnota $array[index]$.

Délku pole lze zjistit instrukcí `arraylength`, která ze zásobníku odebere referenci na pole a vloží na něj délku tohoto pole.

Metody a objekty

Nový objekt lze vytvořit instrukcí `new` *index*, kde *index* je dvoubajtový index do tabulky konstant na třídu nebo rozhraní. Instance třídy nebo rozhraní se inicializuje a její reference se vloží na zásobník.

Přístup k proměnným instance umožňují instrukce `getfield` *index* a `putfield` *index*, kde *index* je dvoubajtový index do tabulky konstant na členskou proměnnou. Instrukce `getfield` odebere ze zásobníku referenci na objekt, získá hodnotu dané členské proměnné a vloží ji na zásobník. Instrukce `putfield` odebere ze zásobníku hodnotu *value* a referenci na objekt a dané členské proměnné tohoto objektu přiřadí hodnotu *value*. Obdobně lze přistupovat k proměnným třídy pomocí instrukcí `getstatic` *index* a `putstatic` *index*.

Metodu lze zavolat instrukcí `invokevirtual` *index*, kde *index* je dvoubajtový index do tabulky konstant na metodu. Instrukce ze zásobníku odebere parametry metody včetně reference na objekt, který bude sloužit jako parametr `this`, a zavolá příslušnou metodu. Obdobně instrukce `invokestatic` volá statickou metodu, `invokeinterface` volá metodu rozhraní a `invokedynamic` volá dynamickou metodu. Pro ostatní metody je třeba použít instrukci `invokespecial`.

Instrukce `instanceof` *index* umožňuje ověřit, zda je objekt daný referencí z vrcholu zásobníku instancí třídy dané dvoubajtovým indexem do tabulky konstant *index*, případně zda objekt implementuje rozhraní dané tímto indexem. Výsledek ověření je vložen na zásobník v podobě hodnoty typu `int` (1 úspěch, 0 neúspěch). Obdobně se chová instrukce `checkcast` *index*, ale v případě úspěchu vloží referenci na objekt zpět na zásobník, v případě neúspěchu vyhodí výjimku `ClassCastException`.

Výjimku lze vyhodit instrukcí `athrow`. Ze zásobníku se odebere reference na instanci třídy `Throwable` nebo její podtřídu a v tabulce výjimek se vyhledá blok pro zpracování této instance. Pokud pro danou výjimku takový blok neexistuje, vykonávání aktuální metody *method* se okamžitě bez předání návratové hodnoty ukončí a výjimka se znovu vyvolá v metodě, která metodu *method* zavolala.

Vstupu do synchronizovaného bloku instrukcí předchází vstup do monitoru daného objektu. Opuštění takového bloku znamená uvolnění tohoto monitoru. Toto chování zajišťují instrukce `monitorenter` a `monitorexit`. Referenci na objekt, s jehož monitorem budou pracovat, získávají z operačního zásobníku.

Konverze hodnot

Hodnotu z vrcholu zásobníku lze konvertovat na jiný datový typ instrukcí typu $t_1 2 t_2$, kde $t_1, t_2 \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}\}$ a pro t_1 rovno `i` navíc $t_2 \in \{\texttt{b}, \texttt{c}, \texttt{s}\}$. Hodnota je pak konvertovaná z typu t_1 na typ t_2.

Matematické a bitové operace

Instrukce pro matematické operace jsou ve tvaru *toperation*, kde $t \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}\}$ specifikuje typ operandů a *operation* $\in \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem}, \texttt{neg}\}$ určuje jednu z matematických operací pro součet, rozdíl, násobení, dělení, zbytek po dělení a negaci. Instrukce pro bitové operace jsou ve tvaru *toperation*, kde $t \in \{\texttt{i}, \texttt{l}\}$ a *operation* $\in \{\texttt{shl}, \texttt{shr}, \texttt{ushr}, \texttt{and}, \texttt{or}, \texttt{xor}\}$ označuje bitový posuv doleva, aritmetický posuv doprava, logický posuv doprava, logický součin, logický součet nebo exkluzivní logický součet. Uvedené instrukce odeberou ze zásobníku příslušný počet operandů a vrátí na zásobník výsledek operace.

Podmíněné skoky

Instrukce `if`*condition next*, kde *condition* $\in \{\texttt{eq}, \texttt{ne}, \texttt{lt}, \texttt{le}, \texttt{ge}, \texttt{gt}\}$ a *next* je dvoubajtová znaménková hodnota, umožňuje provést podmíněný skok na jinou instrukci. Ze zásobníku odebere hodnotu typu `int` a porovná ji s nulou na rovnost, nerovnost, menší než, menší nebo rovno, větší než či větší nebo rovno dle *condition*. Pokud je podmínka pro skok splněna, pokračuje se instrukcí ve vzdálenosti *next* od pozice aktuální instrukce. Jinak se pokračuje následující instrukcí. Instrukce `if_icmp`*condition* umožňuje vzájemně porovnat dvě hodnoty typu `int`. Pro hodnoty typu `long`, `float` a `double` je nutné nejprve provést jednu z instrukcí `lcmp`, `fcmp`x a `dcmp`x, kde $x \in \{\texttt{l}, \texttt{g}\}$. Instrukce ze zásobníku odebere dvě hodnoty, porovná je a výsledek porovnání vloží na zásobník (1 pro větší než, 0 pro rovnost, -1 pro menší než). Podmíněný skok lze následně vykonat instrukcí `if`*condition*. Pro porovnání objektu s `null` jsou k dispozici instrukce `ifnull` a `ifnonnull`. Pro porovnání dvou objektů lze použít instrukce `if_acmpeq` a `if_acmpne`.

Příkaz `switch` se převádí na jednu z instrukcí `tableswitch` a `lookupswitch`. První instrukce pracuje s tabulkou relativních adres, kde vstupní hodnota lze přímo převést na index do tabulky. Druhá instrukce pracuje s tabulkou dvojic klíč-adresa, kde pro vstupní hodnotu je třeba nalézt dvojici s odpovídajícím klíčem. Tabulka adres je vhodnější, nejsou-li jednotlivé případy příkazu `switch` navzájem příliš rozptýlené, jinak je lepší použít tabulku dvojic.

První instrukce je definovaná jako `tableswitch` *pad default low high table*, kde *pad* je výplň o délce nula až tři bajty, která slouží ke správnému zarovnání dalších položek, *default*, *low* a *high* jsou čtyřbajtové hodnoty a *table* je sekvence čtyřbajtových hodnot o délce *high* - *low* + 1. Instrukce načte ze zásobníku hodnotu *value* typu `int` a ověří zda leží v rozsahu hodnot *low* a *high*. Pokud ne, pak pro skok použije relativní adresu *default*, pokud ano, pak použije adresu z tabulky relativních adres *table* na pozici *value* - *low*. Následně se provede skok.

Podoba druhé instrukce je `lookupswitch` *pad default count pairs*, kde *count* je čtyřbajtová hodnota označující počet dvojic v tabulce *pairs* a *pairs* je sekvence dvojic čtyřbajtových hodnot *key* a *next*. Dvojice jsou v tabulce seřazené podle hodnoty *key*. Instrukce načte ze zásobníku hodnotu *value* typu `int`, vyhledá v tabulce *pairs* dvojici, kde *key* je rovno *value*, a odpovídající hodnotu *next* použije jako relativní adresu skoku. Pokud takovou dvojici nenajde, skočí na relativní adresu *default*.

Nepodmíněné skoky

Návrat z metody umožňuje instrukce `return`. Jestliže metoda vrací hodnotu, pak je třeba tuto hodnotu vložit na zásobník a zavolat instrukci t`return`, kde $t \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}, \texttt{a}\}$ označuje typ návratové hodnoty. Instrukce `goto` *next*, kde *next* kde dvoubajtová relativní adresa, provede skok na instrukci na dané adrese. Podobně instrukce `goto_w` *next* umožňuje skočit na čtyřbajtovou relativní adresu. Instrukce `jsr`, `jsr_w` a `ret` slouží k obsluze podprogramu. Používají se k implementaci bloku *finally*.

Práce se zásobníkem

Níže zmíněné instrukce umožňují manipulovat se zásobníkem. Způsob manipulace je popsaný pomocí tzv. jednotek délky zásobníku, přičemž některé hodnoty na zásobníku se mohou skládat ze dvou jednotek. Po provedení instrukce musí být vždy zachována integrita těchto hodnot. Porušení integrity by bylo odhaleno při verifikaci `class` souboru.

K odstranění hodnot z vrcholu zásobníku slouží instrukce `pop` a `pop2`, které odstraní jednu, respektive dvě, jednotky. Instrukce `dup` duplikuje jednotku na vrcholu zásobníku, zatímco instrukce `dup_x1` a `dup_x2` duplikovanou jednotku navíc přesunou o dvě, respektive tři, jednotky níže. Obdobně instrukce `dup2` duplikuje dvojici jednotek na vrcholu zásobníku a instrukce `dup2_x1` a `dup2_x2` dvojici navíc přesunou o tři, respektive čtyři, jednotky níže. Instrukce `swap` prohodí pořadí dvou jednotek na vrcholu zásobníku.

Další instrukce

Instrukce `nop` nic nedělá. Instrukce `breakpoint`, `impdep1` a `impdep1` jsou rezervované pro použití v jiných programech, ale ve validním `class` souboru se objevit nemohou.

Kapitola 4

Nástroje pro manipulaci s bajtkódem

Tato kapitola je věnovaná existujícím nástrojům pro manipulaci s bajtkódem. Vybrala jsem tři populární knihovny BCEL, ASM a Javassist implementované v jazyce Java. Knihovny se navzájem liší mírou abstrakce a způsobem manipulace s bajtkódem.

4.1 BCEL

BCEL nebo-li Byte Code Engineering Library [4] je knihovna, která je součástí projektu Apache Commons. Je poskytovaná pod licencí Apache License 2.0. Poslední verze BCEL 5.2 nepodporuje Javu 8, ale z repozitáře je dostupná verze 6.0, kde je podpora z větší části implementovaná. Vývoj knihovny však v posledních letech není příliš aktivní.

Programové rozhraní knihovny je dostupné v balíčku `org.apache.bcel`. Knihovna obsahuje třídy pro statický popis `class` souborů, třídy pro dynamické úpravy a vytváření bajtkódu a třídy s užitečnými nástroji. Syntaktickou analýzu `class` souboru a vytvoření reprezentace jeho obsahu v podobě instance třídy `JavaClass` umožňuje třída `ClassParser` z balíčku `org.apache.bcel.classfile`. Součástí balíčku jsou současně všechny třídy podílející se na popisu obsahu souboru. Pro každou položku souboru je tedy vytvořen nový objekt. Takový přístup může být velmi neefektivní, pokud je třeba zpracovat velké množství souborů. Na druhou stranu třída `JavaClass` velmi přesně kopíruje formát `class` souboru tak, jak byl popsán v kapitole 3, včetně tabulky konstant. Pro dynamické vytváření a úpravu bajtkódu je třeba vyšší míra abstrakce. Tu poskytují třídy z balíčku `org.apache.bcel.generic`. Pomocí těchto tříd je třeba sestavit celý obsah `class` souboru včetně tabulky konstant. Korektnost výsledného bajtkódu lze zkontrolovat třídou `Verifier`.

Knihovna BCEL poskytuje pro bajtkód velmi nízkou úroveň abstrakce. Je třeba být seznámen s formátem `class` souborů a pracovat s tabulkou konstant. Bajtkód je navíc reprezentovaný velkým množstvím objektů a neexistuje efektivní způsob, jak zpracovat jen ty informace, které jsou pro danou aplikaci potřeba. Vhodnou alternativou je proto knihovna ASM.

4.2 ASM

ASM [3] je knihovna od OW2 Consortium poskytovaná pod licencí BSD. Na rozdíl od BCEL se jedná o aktivní projekt a Java 8 je oficiálně plně podporovaná. ASM si zakládá

na snadné použitelnosti, výkonnosti a malé velikosti. Knihovna je založena na návrhovém vzoru Návštěvník (Visitor). Místo reprezentace `class` souboru pomocí objektů jsou při syntaktické analýze volány pro jednotlivé položky metody návštěvníka. Návštěvník může položky zpracovat a předat je dalšímu návštěvníkovi. Pomocí takového zřetězení lze jedním až dvěma průchody `class` souboru dosáhnout požadovaného zpracování bajtkódu. Pokud je třeba provést větší počet průchodů, může být vhodnější použít objektovou reprezentaci pomocí stromu. ASM umožňuje oba přístupy libovolně kombinovat.

Základní rozhraní je dostupné v balíčku `org.objectweb.asm`. Třída `ClassReader` analyzuje daný `class` soubor a volá metody návštěvníka, instance třídy rozšiřující abstraktní třídu `ClassVisitor`. Třída `ClassVisitor` umožňuje vytvořit sekvenci návštěvníků. Jedním z těchto návštěvníků může být i instance třídy `ClassWriter`, která z parametrů volaných metod vytvoří opět binární reprezentaci bajtkódu. Tato třída může být použita i samostatně pro dynamické generování bajtkódu. Při průchodu souborem i při jeho vytváření je třeba pamatovat na pořadí, ve kterém jsou jednotlivé položky navštíveny. Programové rozhraní pro objektovou reprezentaci pomocí stromu je v balíčku `org.objectweb.asm.tree`. Obsah `class` souboru je reprezentovaný třídou `ClassNode`, která tvoří kořen stromu. Jednotlivé položky tvoří uzly. S takto vytvořeným stromem lze libovolně manipulovat i vytvořit zcela nový. Jednotlivé uzly jsou současně návštěvníky daných položek. Díky tomu je možné libovolně přecházet mezi oběma přístupy k bajtkódu. Balíčky `org.objectweb.asm.util`, `org.objectweb.asm.commons` a `org.objectweb.asm.tree.analysis` obsahují některé zajímavé nástroje pro zpracování a analýzu bajtkódu.

ASM zaujme svým návrhem a možností výběru mezi dvěma způsoby práce s bajtkódem. Nabízí vyšší úroveň abstrakce než BCEL, neboť přístup k tabulce konstant je uživateli zcela odepřen. Na druhou stranu je práce s bajtkódem stále na úrovni blízké formátu `class` souboru. Z popisovaných nástrojů je ASM považován za nejrychlejší.

4.3 Javassist

Javassist nebo-li Java Programming Assistant [1] je knihovna poskytovaná pod trojitou licencí MPL, LGPL a Apache License. Je vhodná zejména pro úpravu bajtkódu za běhu programu. Knihovna umožňuje pracovat s `class` soubory na dvou úrovních. Úroveň zdrojového kódu nevyžaduje znalost bajtkódu, ale umožňuje s bajtkódem manipulovat pomocí slovníku programovacího jazyka Java. Úroveň bajtkódu umožňuje přístup k reprezentaci blízké formátu `class` souboru. Java 8 je podporovaná.

V balíčku `javassist` je dostupné základní rozhraní knihovny. Třída `CtClass` je reprezentací `class` souboru. Instanci této třídy je třeba získat z úložiště reprezentovaného třídou `ClassPool`. V tomto úložišti jsou k dispozici všechny takto načtené třídy. Získanou reprezentaci třídy lze modifikovat a uložit do souboru či pole bajtů, nebo lze vytvořit reprezentovanou třídu. Těla metod lze modifikovat pomocí tříd z balíčku `javassist.expr`. Manipulace na úrovni zdrojového kódu má však jistá omezení a nejsou podporovány všechny jazykové konstrukce. Proto balíček `javassist.bytecode` poskytuje rozhraní pro přímou editaci bajtkódu. Instrukční soubor je zde reprezentovaný třídou `ClassFile`. K dispozici je i tabulka konstant reprezentovaná třídou `ConstPool`.

S `class` souborem se v Javassist opět manipuluje prostřednictvím objektové reprezentace. Zajímavá je však možnost pracovat s bajtkódem jako s konstrukcemi programovacího jazyka Java. Javassist tak nabízí mnohem vyšší úroveň abstrakce než BCEL a ASM. Navíc má propracovanější podporu editace bajtkódu za běhu.

Kapitola 5

Nástroj pro analýzu bajtkódu

V této kapitole popisuji návrh a implementaci nástroje `jbyco`. Nástroj je určen pro zpracování velkého množství souborů a získání dat, která jsou vhodná pro analýzu bajtkódu.

5.1 Analýza problému a návrh řešení

Pro analýzu bajtkódu jsem potřebovala získat data, která by vhodným způsobem reprezentovala analyzovaný bajtkód. Zajímaly mne celkové součty položek v `class` souboru, využití lokálních proměnných a parametrů a typické sekvence instrukcí. Dále jsem potřebovala vyřešit načítání velkého množství vstupních dat.

U hledání typických sekvencí jsem zvažovala jednotlivé sekvence a jejich četnosti zaznamenávat jednoduše pomocí tabulky klíč-hodnota. Takový přístup mi nepřipadal vhodný z hlediska paměťové složitosti, neboť by to znamenalo udržovat v paměti všechny podsekvence sekvencí instrukcí. Rozhodla jsem se proto pro reprezentaci sekvencí pomocí orientovaného acyklického grafu, tzv. grafu sufixů. Na instrukce se lze dívat jako na prvky abecedy a na sekvence instrukcí jako na řetězce. Pak definované cesty v grafu sufixů tvoří sufixy reprezentovaných řetězců. Prefixy těchto sufixů tvoří všechny podřetězce reprezentovaných řetězců.

Graf sufixů se skládá z kořene a uzlů. Všechny uzly kromě kořene reprezentují prvky abecedy a pro každou hranu je definovaná množina cest, které danou hranou prochází. Každá taková cesta v grafu má vlastní čítač, který určuje, kolik stejných cest daná cesta reprezentuje. Když se do grafu přidává další sufix, postupuje se grafem směrem od kořenu. Pokud je aktuální prvek sufixu stejný jako prvek některého ze sousedů aktuálního uzlu, pak se daný uzel stane aktuálním a začne se zpracovávat následující prvek sufixu. Pokud takový soused neexistuje, vybere se uzel se stejným prvkem, který je nedosažitelný z aktuálního uzlu. K tomuto uzlu se vytvoří z aktuálního uzlu hrana, vytvoří se nová cesta a uzel se označí za aktuální. Jestliže takový uzel neexistuje, pak se vytvoří nový uzel, hrana i cesta a nový uzel se stane aktuálním. Po vložení posledního prvku sufixu se inkrementuje čítač cesty, kterou sufix duplikuje, nebo se vytvoří cesta nová, pokud už vytvořena nebyla. Cílem je v podstatě minimalizovat duplicitu podřetězců v grafu.

V takto vytvořeném grafu sufixů lze následně zjednodušit každou hranu, kde je součet čítačů všech cest menší než daná hodnota. Prvek uzlu, do kterého vede taková hrana, lze nahradit tzv. divokou kartou a sekvenčně i paralelně sousedící uzly s divokou kartou lze sloučit do jednoho uzlu. V takto zjednodušeném grafu lze nalézt typické vzory sekvencí instrukcí.

5.2 Popis implementace

Nástroj jsem implementovala v jazyce Java s pomocí knihovny BCEL. Hlavní metoda `main` je součástí třídy `App` v balíčku `jbyco`. V této metodě se zpracují parametry, vytvoří se iterátor vstupních souborů a spustí daná analýza. Iterátor vstupních souborů je reprezentovaný třídou `BytecodeFiles` z balíčku `jbyco.io`. Balíček obsahuje třídy pro různé vstupně-výstupní operace. Třída `BytecodeFiles` rekurzivně prochází všechny soubory, složky a `jar` soubory a vrací instance třídy `BytecodeFile` reprezentující `class` soubory. Rozhraní `Analyzer` z balíčku `jbyco.analyze` popisuje rozhraní tříd, které provádí analýzu. V balíčku `jbyco.analyze.size` jsou třídy potřebné k analýze velikosti, v balíčku `jbyco.analyze.locals` třídy určené k analýze využití lokálních proměnných a parametrů a v balíčku `jbyco.analyze.patterns` třídy pro nalezení typických sekvencí instrukcí. Výstupem každé analýzy je vytištěná tabulka se zjištěnými daty. Třídy pro práci s grafem sufixů jsou obsaženy v balíčku `jbyco.analyze.patterns.graph`. Překlad nástroje lze provést příkazem `gradle build`. Aplikaci lze spustit příkazem `gradle run -Dmyargs="`$args$`"`. Argument `help` vypíše nápovědu k programu.

Analýza velikosti a analýza využití proměnných běží i pro velké množství souborů velmi rychle, zatímco vyhledávání typických vzorů je velmi pomalé. Při bližším zkoumání jsem odhalila, že nejvíce času program stráví zajišťováním acykličnosti grafu. To, že graf není strom, je zajímavé z hlediska úspory paměti i zjednodušování hran. Cenou je však čas běhu programu. Budu muset zvážit volbu jiné reprezentace. Algoritmus pro zjednodušování hran jsem implementovala, ale není zcela odladěný, proto jsem jej pro analýzu nepoužila.

Kapitola 6

Analýza bajtkódu

V této kapitole popisuji analýzu bajtkódu a její výsledky. Pomocí nástroje `jbyco` jsem získala data reprezentující velký vzorek testovacích souborů a tato data následně analyzovala a vyhodnotila. Zkoumala jsem velikosti položek v `class` souborech, využití lokálních proměnných a parametrů a typické sekvence bajtkódu.

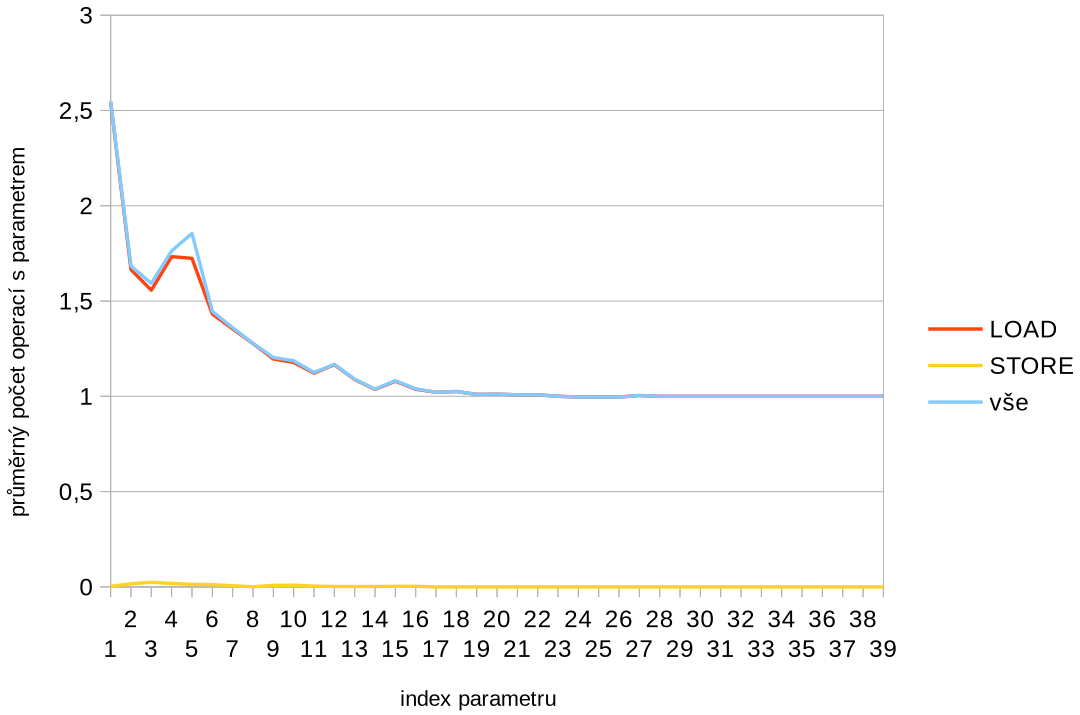
Testovací vzorek jsem vytvořila z `jar` souborů stažených z `http://mvnrepository.com`. Z populárních kategorií jsem vybrala nejčastěji používané `jar` soubory. Po smazání příliš velkých souborů jsem získala testovací vzorek o velikosti 92 souborů a 102,4 MB. Pro analýzu typických sekvencí jsem musela zvolit menší vzorek o velikosti 6 souborů a 1,1 MB, neboť pro více dat mi nestačila paměť.

6.1 Výsledky analýzy

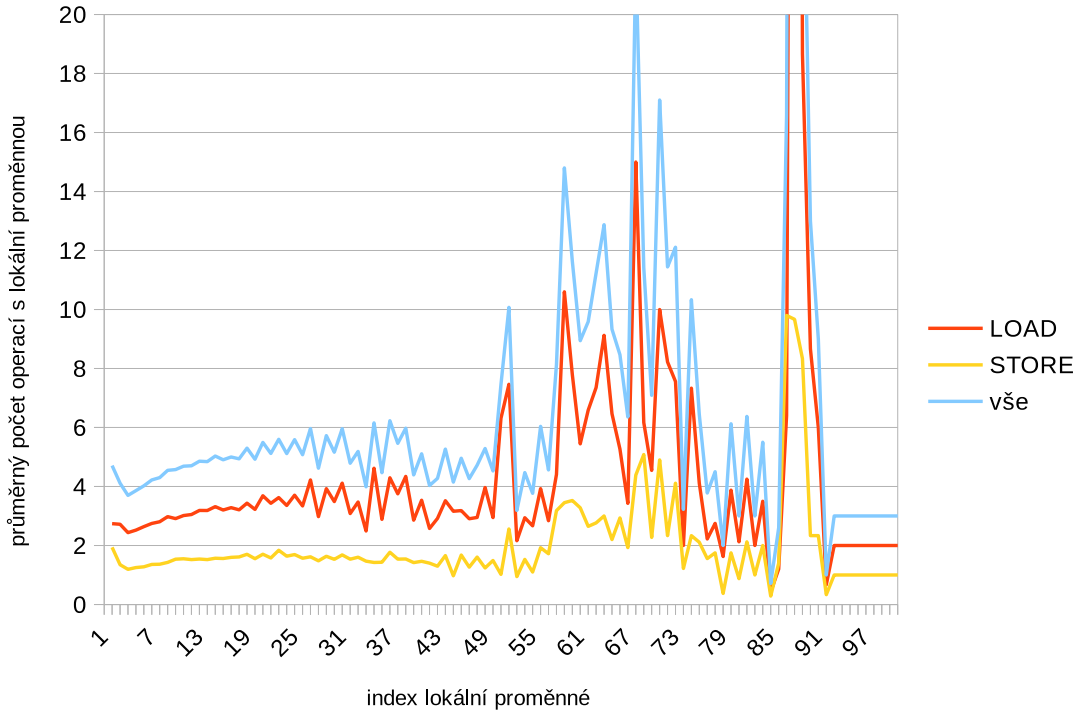
Využití parametrů a lokálních proměnných je znázorněné na grafech 6.1 a 6.2. Zajímalo mne, kolikrát se z daného parametru či lokální proměnné načítá hodnota a vkládá hodnota. Počet operací je zprůměrován celkovým počtem výskytů daného parametru či lokální proměnné. Z grafu 6.1 lze vyčíst, že z parametrů se především načítají hodnoty ale už se do nich nic neukládá. Jejich celkové využití je velmi malé. S lokálními proměnnými se pracuje častěji, ale jejich využití mohlo být pořád lepší.

Při analýze velikosti jsem zkoumala konstanty, atributy a instrukce z hlediska celkového počtu a celkové velikosti. Z tabulek A.1 a A.2 je zřejmé, že konstanta `CONSTANT_Utf8` je nejfrekventovanější a zároveň zabírá ze všech konstant největší procento z celkové velikosti souboru. Následují konstanty pro popis metod, tříd, rozhraní a členských proměnných. A poslední v pořadí jsou číselné konstanty. Nejfrekventovanějším a nejobjemnějším atributem je podle tabulek A.3 a A.4 pochopitelně `Code`. Zajímavé však je, že velké procento atributů i celkové velikosti tvoří atributy sloužící výhradně pro ladící účely. V tabulkách A.5 a A.6 jsou zjištěné hodnoty pro instrukce. Prvenství v četnosti instrukce `aload_0` lze vysvětlit častým používáním reference `this`. Následují instrukce pro volání metody a načtení hodnoty z proměnné instance. Podle dalších konstant v tabulce je zřejmé, že se nejčastěji pracuje s referencí na objekt. Z hlediska celkové velikosti jsou nejobjemnější instrukce pro volání metod a získávání hodnoty členské proměnné.

Přehled typických sekvencí operačních kódů je dostupný v tabulce A.7. Nejčastější sekvence `aload_0;getfield;` poukazuje na častý přístup k proměnným aktuální instance. Sekvence `new;dup;` reprezentuje optimalizaci. Dále se často pracuje s prvními dvěma lokálními proměnnými. Nejčastěji se spolu v jedné sekvenci objevuje načítání reference z lokální



Obrázek 6.1: Průměrné počty operací s danými parametry.



Obrázek 6.2: Průměrné počty operací s danými lokálními proměnnými.

proměnné a volání speciálních metod.

6.2 Vyhodnocení

Studium využití parametrů a lokálních proměnných poukazuje na plýtvání s tímto paměťovým prostorem. Pole lokálních proměnných by zřejmě mohlo být menší, kdyby se proměnné více recyklovaly. Použití parametrů jako lokálních proměnných zase umožní používat instrukce s kratšími kódy.

Z analýzy velikosti konstant plyne, že řetězce tvoří podstatnou část instrukčního souboru. Řetězcové konstanty a řetězce s typy nijak nahradit nelze, ale názvy tříd, rozhraní, metod a proměnných by bylo možné nahradit kratšími řetězci. Analýza velikosti atributů poukázala na to, že atributy, které nemají vliv na interpretaci souboru, mají poměrně velkou velikost. Tyto atributy by za jistých předpokladů šlo odstranit. Z analýzy velikosti instrukcí vyplynulo, že se nejčastěji používají objemné instrukce. To stejné se potvrdilo analýzou typických sekvencí. Bylo by vhodné se zamyslet nad tím, jak zamezit zbytečnému volání metod a načítání hodnot z členských proměnných.

Kapitola 7

Závěr

Popsala jsem virtuální stroj Java Virtual Machine a formát jeho instrukčního souboru. Vycházela jsem ze specifikace [2] s cílem podat ji trochu jiným způsobem. Seznámila jsem se s nástroji pro manipulaci s bajtkódem BCEL, ASM, a Javassist, uvedla jejich stručný popis a vzájemně je porovnala. Dále jsem navrhla a implementovala vlastní nástroj `jbyco` pro analýzu bajtkódu. Stáhla jsem si velké množství `jar` souborů a pomocí `jbyco` jsem získala data, která jsem dále analyzovala. Při analýze jsem se zabývala celkovou velikostí položek v instrukčním souboru, sekvencemi instrukcí s častým výskytem a využitím lokálních proměnných.

Budu pokračovat dokončením implementace nástroje `jbyco` a další analýzou bajtkódu. Konkrétně bych chtěla nalézt sekvence instrukcí pro různé stupně abstrakce operačních kódů a operandů. Tuto část mám rozpracovanou, ale nestihla jsem ji dokončit. Dále navrhnu, implementuji a otestuji nástroj pro optimalizaci velikosti bajtkódu. Vycházet budu z poznatků analýzy.

Literatura

[1] Chiba, S.: Javassist [online]. 2015 [cit. 2016-01-11].
Dostupné z: `<http://jboss-javassist.github.io/javassist/>`

[2] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014, ISBN 013390590X, 600 s.

[3] OW2 Consortium: ASM [online]. 2015 [cit. 2016-01-11].
Dostupné z: `<http://asm.ow2.org/>`

[4] The Apache Software Foundation: BCEL [online]. 2014 [cit. 2016-01-11].
Dostupné z: `<http://commons.apache.org/proper/commons-bcel/>`

[5] Venners, B.: *Inside the Java Virtual Machine*. McGraw-Hill Companies, druhé vydání, 2000, ISBN 0-07-135093-4, 703 s.

Příloha A

Data pro analýzu

A.1 Počet výskytů konstant

CONSTANT_Utf8	3931719
CONSTANT_NameAndType	980941
CONSTANT_Methodref	739577
CONSTANT_Class	624623
CONSTANT_String	205548
CONSTANT_Fieldref	203785
CONSTANT_InterfaceMethodref	129907
CONSTANT_Integer	51430
CONSTANT_Long	12885
CONSTANT_Double	10755
CONSTANT_Float	427

A.2 Celková velikost konstant v bajtech

CONSTANT_Utf8	108921468
CONSTANT_NameAndType	4904705
CONSTANT_Methodref	3697885
CONSTANT_Class	1873869
CONSTANT_Fieldref	1018925
CONSTANT_InterfaceMethodref	649535
CONSTANT_String	616644
CONSTANT_Integer	257150
CONSTANT_Long	115965
CONSTANT_Double	96795
CONSTANT_Float	2135

A.3 Počet výskytů atributů.

Code	465419
LineNumberTable	420096
LocalVariableTable	394358
Signature	105321
StackMapTable	93683
Exceptions	66122
LocalVariableTypeTable	58816
SourceFile	53981
InnerClasses	33003
ConstantValue	32800
RuntimeVisibleAnnotations	13572
EnclosingMethod	10339
Deprecated	5263
Synthetic	3566
RuntimeInvisibleAnnotations	3414
BootstrapMethods	6

A.4 Celková velikost atributů v bajtech.

Code	54509134
LocalVariableTable	11564766
LineNumberTable	7917940
StackMapTable	2272068
LocalVariableTypeTable	1057772
InnerClasses	682574
Exceptions	282092
Signature	210642
RuntimeVisibleAnnotations	148322
SourceFile	107962
ConstantValue	65600
EnclosingMethod	41356
RuntimeInvisibleAnnotations	36844
Deprecated	31578
Synthetic	21396
BootstrapMethods	292

A.5 Počet výskytů nejfrekventovanějších instrukcí.

aload_0	1135548
invokevirtual	918798
getfield	510855
aload	440476
aload_1	436626
dup	397204
invokespecial	364943
ldc	270551
invokeinterface	265836
aload_2	244628
areturn	240651
invokestatic	234834
new	221495
goto	198261
putfield	191594
return	189986
astore	177260
iload	165663
bipush	165586
aload_3	162497
iconst_0	161359
iconst_1	140853
ifeq	137145
getstatic	130984
aastore	108201
checkcast	107941

A.6 Celková velikost nejobjemnějších instrukcí v bajtech.

invokevirtual	2756394
getfield	1532565
invokeinterface	1329180
aload_0	1135548
invokespecial	1094829
aload	880952
invokestatic	704502
new	664485
goto	594783
putfield	574782
ldc	541102
aload_1	436626
ifeq	411435
dup	397204
getstatic	392952
astore	354520
tableswitch	352111
iload	331326
bipush	331172
checkcast	323823
ldc_w	263508
aload_2	244628
areturn	240651
return	189986
sipush	183228
ifnull	177831

A.7 Frekventované sekvence instrukcí

4517 aload_0;getfield;
3124 new;dup;
2108 invokevirtual;invokevirtual;
1862 aload_0;aload_1;
1634 aload_1;invokevirtual;
1561 aload_0;invokevirtual;
1477 ldc;invokevirtual;
1433 aload;invokevirtual;
1187 new;dup;invokespecial;
1187 dup;invokespecial;
1162 putfield;aload_0;
1159 aload_0;invokespecial;
1006 astore;aload;
1004 dup;sipush;
874 aload;invokeinterface;
842 invokevirtual;aload_1;
829 aload_2;invokevirtual;
807 invokevirtual;aload_0;
807 getfield;invokevirtual;
750 aload_0;getfield;invokevirtual;
739 invokespecial;athrow;
716 invokespecial;return;
708 invokevirtual;aload;
696 iastore;dup;
674 invokevirtual;pop;
666 putfield;return;
623 bastore;dup;
621 invokevirtual;areturn;
612 aload_1;aload_2;
599 aload;aload;
598 invokevirtual;ldc;
580 invokespecial;aload_0;
580 invokevirtual;ldc;invokevirtual;
567 aload_1;invokeinterface;
565 aload_0;invokevirtual;invokevirtual;
558 invokevirtual;ifeq;
531 invokevirtual;return;
494 iastore;dup;sipush;
494 bastore;dup;sipush;
481 getfield;aload_0;
471 invokeinterface;ifeq;