Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási nyelvek és Fordítóprogramok Tanszék

JVM bytecode interpreter Javában

Témavezető:
Kozsik Tamás Dr.
egyetemi docens

Szerző:
Balázs Zoltán
programtervező informatikus BSc

Budapest, 2023

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:
Név: Balázs Zoltán
Neptun kód: HV56L5

Képzési adatok:
Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)
Tagozat : Nappali

Belső témavezetővel rendelkezem

Témavezető neve: *Kozsik Tamás Dr.*
munkahelyének neve, tanszéke: ***ELTE IK, Programozási nyelvek és Fordítóprogramok Tanszék***
munkahelyének címe: ***1117, Budapest, Pázmány Péter sétány 1/C.***
beosztás és iskolai végzettsége: ***egyetemi docens, programtervező matematikus***

A szakdolgozat címe: Java bytecode interpreter Javában

A szakdolgozat témája:
(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

A Java nyelvben írt programok fordításuk során nem közvetlenül gépi kódra fordulnak, hanem egy hardver-független nyelvre, amit bytecode-nak neveznek.
Ezt a bytecode-ot az esetek többségében a JVM (Java Virtual Machine) interpreter-e hajtva végre, vagy futási időben fordul le a fordító gép hardverének gépi kódjára.

A szakdolgozat célja egy olyan Java bytecode interpreter fejlesztése, amely képes már előre, valamilyen Java fordító által, elkészített bytecode-ot interpreter-álni, ezt sikeresen (és helyesen) lefuttatni.
A fejlesztett interpreter-nek képesnek kell lennie az ELTE Programtervező Informatikus BSc szakán, különböző, Java-t használó tárgyakon (Programozási nyelvek, Konkurens programozás) elkészített beadandók és házi feladatok generált bytecode-ját interpreter-álni, ezeket helyesen futtatni.

Budapest, 2022. 11. 24.

Tartalomjegyzék

1. fejezet

Bevezetés

A Java nyelvben írt programok fordításukat követően nem egy közletlen futtatható állományra (gépi kódra) fordulnak (a fordítást általában a beépített `javac` program végzi el), hanem egy köztes nyelvre, bytecode-ra, amelyet aztán különböző programokkal az adott architektúrán interpretáljuk. Legtöbb esetben az interpretálást a JVM (Java Virtual Machine) interpretere hajtja végre (ez a beépített `java` program).

A szakdolgozat célja egy kiegészítő program (fantázianevén *Jabyinja* – ***Ja****va* ***by****tecode* ***in****terpreter in* ***Ja****va*) írása, amely ugyan hagyatkozik a `javac` és `java` programokra (az előbbire a fordítás, az utóbbira a futtatás miatt), de a tényleges futtatást a különböző bytecode instrukciók implementálásval végzi el.

A program nincsen Java kód interpretálásához kötve, a Java bytecode a neve ellenére más programozási nyelveknek is az alapja (ezek közül az ismertebbek: Kotlin, Clojure), viszont a tesztelés csak Java kódból generált bytecodera tér ki, ugyanis a szakdolgozat céljaként az ELTE Programtervező Informatikus BSc szakán elkészített Java programok fordításának interpretálását tűztem ki.

A programnak szükséges értelmeznie kell egy adott Classfájlt (többet is ha egy külön fájlra is hivatkozunk), helyesen beolvasnia a benne lévő adatokat, majd a belépési (*main*) metódust lefuttatnia. A program erősen alapszik a Java nyelvbe beépített reflekcióra, ezen felül saját stack implementálása is szükséges. Mivel a Java nyelvre épül a program, ezért saját heap megírására nincsen szükség, ez automatikusan kezelve lesz.

2. fejezet

Felhasználói dokumentáció

A program elsődleges felhasználói fejlesztők, alapszintű tudás szükséges a Java nyelvről (vagy bármilyen olyan nyelvről amely JVM Bytecode-ra fordul), a class fájlokról, illetve Java programok fordításáról.

Mivel az elkészített program csak interpretálni tud, a fordítást egy már elérhető Java fordítóprogrammal szükséges megtenni. Mivel a Java programok class fájlokra fordulnak, ezek futtatásához szükséges egy interpretáló program.

Alapvető esetben ez a fordítóprogrammal együtt telepítésre kerül. A szakdolgozat esetében a lefordított class fájl futtatásával képesek lehetünk más, már lefordított Java programot futtatni.

A mellékelt fájlok között elérhető egy jar fájl is, ennek a futtatásához ugyanúgy szükségünk van egy beépített interpretáló programra, amely képes Java programokat futtatni és nem a szakdolgozat maga.

2.1. Kikötések

A program csak Java 7-nél újabb fordítóprogrammal fordított Java programokat képes interpretálni, számos Bytecode instrukciót a Java 7-es verziójában elavulttá tettek (ezek: `ret`, `jsr`, `jsr_w`), nem fordulnak elő class fájlokban. A szakdolgozat ezeket az instrukciókat nem implementálta.

Ezen felül egy másik instrukció is implementálatlan maradt (`invokedynamic`), tehát nem minden program futtatható. Ha a class fájlok egyike tartalmazza ezt az instrukciót, akkor a program jelez a felhasználó számára. Akaratlanul is része lehet

a programunknak ez az instrukció, amikor egy változót szöveggel együtt próbálunk kiírni:

```
String world = "world";
System.out.println("Hello " + world);
```

akkor a legtöbb fordítóprogram egy `invokedynamic` utasítást is elhelyez a programunkban.

Ez viszont elkerülhető, ha megfelelő flagekkel fordítjuk le a programunkat, mégpedig a `-XDstringConcat=inline` flag használatával az `invokedynamic` nem fog szerepelni a string konkatenációnál.

2.2. Fordítástól futásig

2.2.1. Minimum követelmények

A program fordításához legalább a Java 17-es verziója szükséges. Ez alatt a program fordulni sem képes, mivel pár olyan funkciót használ, amely csak a 17-es verzióban lett bevezetve.

A könnyebb fordítás (illetve egyszerűbb jar fájl készítés) érdekében a Maven fordítás automatizálási program telepítése ajánlott, ezen belül is a 3.9.0-ás verzió.

2.2.2. Fordítás

Ha nem akarunk Maven-t használni, akkor a fordítás menete a következő:

- Menjünk a `src/main/java` mappába: `cd src/main/java/`
- Fordítsuk le a `com/zoltanbalazs/Main.java` fájlt: `javac com/zoltanbalazs/Main.java`
- Az elkészült class fájl a `src/main/java/com/zoltanbalazs` mappában lesz

Maven-t használva ez a procedúra egyszerűbb:

- Futtassuk le a csomagoló parancsot: `mvn package`
- Az elkészült class fájl a `target/classes/com/zoltanbalazs` mappában lesz, ezen felül a `target` mappában lesz egy futtatható jar fájl is

2.2.3. Futtattás

Ha a generált class fájl-lal akarjuk futtatni a programot, futtassuk le a `java com.zoltanbalazs.Main` parancsot a `src/main/java` mappában. (ha Maven-nel fordítottunk akkor a `target/classes` mappában futtassuk le az előző fenti parancsot)

A maven által készített jar fájl-lal való futtatáshoz, futtassuk le a `java -jar target/jabyinja-1.0.0.jar` parancsot a főmappában.

Mindkét esetben egy opcionális argumentumot (argumentum sorozatot ha a futtatandó programunk vár parancssori argumentumot) meg tudunk adni, ez a `main` metódust tartalmazó class fájl elérési útvonala. Alapvető esetben a program a futási mappában próbál meg egy `Main.class` fájlt futtatni.

Futásra egy példa: `java -jar target/jabyinja-1.0.0.jar target/test-classes/com/zoltanbalazs/PTI/_01/Greet.class World`

2.2.4. Önfuttatás

Az elkészült interpreter képes saját magát is futtatni, ehhez a futtatáshoz hasonlóan meg kell adni a programnak a saját class fájlának elérési útját, majd opcionálisan a többi paramétert.

Ez jar fájl esetén így néz ki, a főkönyvtárból futtatva: `java -jar target/jabyinja-1.0.0.jar target/classes/com/zoltanbalazs/Main.class target/test-classes/com/zoltanbalazs/PTI/_01/Greet.class World`

2.3. Felmerülő problémák

A futtatandó program futása során nem merül fel probléma (hacsak nincsen `invokedynamic` a generált class fájlban) amelyet a program okoz. Ha a futtatandó programunk hibát dob, akkor ezt az interpretáló program is ugyanúgy megteszi; viszont a hiba kiírása során nem biztos hogy ugyanazt a kimentet kapjuk mint a beépített interpreter-rel.

Tehát ha a hibánk nem egy `try, catch` blokk-ban szerepel, akkor a kiírt üzenet nem biztos hogy ugyanaz lesz mint a beépített interpreter-rel, az összes többi kiírt üzenet viszont ugyanaz kell hogy legyen.

3. fejezet

Fejlesztői dokumentáció

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis nibh leo, dapibus in elementum nec, aliquet id sem. Suspendisse potenti. Nullam sit amet consectetur nibh. Donec scelerisque varius turpis at tincidunt.

3.1. Class fájl felépítése

3.1.1. Class fájltól a benne levő metódus futtatásáig

A fő osztály a `ClassFile`, ez felel számos dologért, többek között egy class fájl beolvasáért, a megfelelő adattagok beállításával. A `ClassFile` osztálynak egy konstruktora van, mégpedig:

```
1 public ClassFile(String fileName, String[] mainArgs)
```

Tehát az első paraméter a beolvasandó class fájl neve, a második pedig a `main` metódusnak adott argumentumok.

Az implementáció alapján nem kötött a `main` metódus használata belépési pontként, tehát a 2. argumentum lehet `null` is.

A konstruktor meghívása egyidejüleg meghívja a `readClassFile` függvényt is:

```
1 public void readClassFile(String fileName)
```

Ez a függvény egy adott fájlnévre beolvassa a class fájlban tárolt adatokat megfelelő változókba. (Ezen felül egy `VALID_CLASS_FILE` változót is beállít; feltételezük hogy ha a mágikus szám `(CA FE BA BE)` megtalálható a fájl elején, akkor az adott fájl egy

valid class fájl, ellenkező esetben egy `InvalidClassFileException`-t dob a beolvasó függvény.)

A beolvasás után (tehát az objektum létrehozása után) érdemes a belépési függvényt (általában `main`) megkeresni a `findMethodsByName` metódussal:

```
1 public Method_Info findMethodsByName(String methodName)
```

Ez egy adott függvénynévre a megfelelő nevü metódust visszaadja a beolvasott fájlból (ha nem talál ilyet akkor `null`-t ad vissza). Egy példa a használatára:

```
1 ClassFile CLASS_FILE = new ClassFile("Main.class", null);
2 Method_Info method = CLASS_FILE.findMethodsByName("main");
```

A függvény megtalálása után ajánlott a `Code` attribútumot megtalálni, ebben, többek között, található a futtatandó bytecode is. A segédfüggvény erre a `findAttributesByName`:

```
1 public List<Attribute_Info>
  ↪  findAttributesByName(List<Attribute_Info> attributes, String
  ↪  attributeName)
```

Mivel egy attribútumból több is lehet, egy listát kapunk vissza (a `Code`-ból csak egy lesz), bemeneti paraméterként az attribútumnév mellett a megfelelü függvény attribútumait is át kell adnuk, például:

```
1 List<Attribute_Info> attributes =
  ↪  CLASS_FILE.findAttributesByName(method.attributes, "Code");
```

(Ha nem talál ilyen nevezetű attribútumot akkor üres listát ad vissza.)

A megfelelüen beolvasott attribútum után, a megtalált attribútumok között ajanlott végigmenni, a `List` implementálja az `Iterable`-t, így egy for ciklussal elegánsan megtehetjük ezt:

```
1 for (Attribute_Info attribute : attributes)
```

Mivel `Code` attribútumokról beszélünk, ezért a következő ajánlott dolog hogy ebből az attribútumból olvassuk be az adatokat. Ehhez a `Code_Attribute_Helper` osztály `readCodeAttributes` metódusa megfelelő:

```
public static Code_Attribute readCodeAttributes(Attribute_Info
↪  attribute) throws IOException
```

A függvény egy attribútumot vár (például az előbbi kódrészlet `attribute` változóját), majd pedig beolvassa a specifikációnak megfelelően a `Code_Attribute`-ot, és visszaadja azt, ha valamiért nem sikerült a beolvasás akkor `IOException`-t dob a függvény.

```
Code_Attribute codeAttribute =
↪  Code_Attribute_Helper.readCodeAttributes(attribute);
```

Ezt a beolvasott attribútumot a `ClassFile` osztály fel tudja használni az `executeCode` metódusával, mely egy `byte[]` változót vár bemeneti paraméterként, ami a `Code_Attribute` része:

```
public Pair<Class<?>, Object> executeCode(byte[] code)
  throws IOException, ClassNotFoundException, NoSuchFieldException,
  ↪  IllegalAccessException,
  NoSuchMethodException, SecurityException, InstantiationException,
  ↪  IllegalArgumentException,
  InvocationTargetException, Throwable
```

A reflekció miatt számos hibát dob vissza a függvény, ha nem helyes a kód formátuma akkor `IOException`-t dob a függvény, a `Throwable` az `ATHROW` bytecode instrukció miatt szükséges (ekkor egy hibát dob vissza a metódusunk). Visszatérési értéke `Pair<Class<?>, Object>`, a számos `RETURN` utasítás miatt (ezeket a stack-en szükséges elhelyezünk) Példa a használatára:

```
CLASS_FILE.executeCode(codeAttribute.code);
```

Ezzel el is jutottunk egy class fájl beolvasásától, az abban lévő adott függvény bytecodejának futtatásáig, több teendőnk nincsen, a program az adott függvényben levő külön függvényhívásokat automatikusan elvégzi.

A teljes példakód:

```
ClassFile CLASS_FILE = new ClassFile("Main.class", null);
Method_Info method = CLASS_FILE.findMethodsByName("main");
List<Attribute_Info> attributes =
 ↪  CLASS_FILE.findAttributesByName(method.attributes, "Code");

for (Attribute_Info attribute : attributes) {
  Code_Attribute codeAttribute =
   ↪  Code_Attribute_Helper.readCodeAttributes(attribute);
  CLASS_FILE.executeCode(codeAttribute.code);
}
```

3.1.2. Pár minta class fájl felépítése

A legegyszerűbb class fájl ami értelmes, viszont nem futattható:

CA FE BA BE 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00

Java kódban ennek megfelelője az üres fájl:

```

```

Class fájl formátumának magyarázata:

- CA FE BA BE: Mágikus szám, amely minden Class fájl elején megtalálható
- 00 00 00 00: Class fájl `Minor` és `Major` verziószáma, egy táblázatnak megfelelően a fordítóprogram verziója
- 00 00: A Constant Pool mérete (+1, mivel 1-től indexelt, itt nem számít)
- 00 00: Hozzáférési zászlók ()
- 00 00: `This` osztály indexe a Constant Pool-ban
- 00 00: `Super` osztály indexe a Constant Pool-ban
- 00 00: Interfészek száma
- 00 00: Adattagok száma
- 00 00: Függvények száma
- 00 00: Osztály attribútumainak száma

A legegyszerűbb class fájl amit a *Jabyinja* program le tud futtatni (a beépített `java` program nem képes ezt lefuttatni, mivel nincsenek benne osztályok, a JVM specifikáció alapján az osztályok elhanyagolhatóak):

CA	FE	BA	BE	00	00	00	00	00	04	01	00	04	43	6F
64	65	01	00	04	6D	61	69	6E	01	00	03	28	29	56
00	21	00	00	00	00	00	00	00	00	00	01	00	09	00
02	00	03	00	01	00	01	00	00	00	0D	00	00	00	00
00	00	00	01	B1	00	00	00	00	00	00				

Java kód megfelelője:

```
1 public static void main() {
2     return;
3 }
```

Class fájl formátumának magyarázata:

- CA FE BA BE: Mágikus szám, amely minden class fájl elején megtalálható
- 00 00 00 00: Class fájl `Minor` és `Major` verziószáma, egy táblázatnak megfelelően a `javac` fordítóprogram verziója
- 00 04: A Constant Pool mérete (+1, mivel 1-től indexelt)
- 01 00 04 43 6F 64 65 01 00 04 6D 61 69 6E 01 00 03 28 29 56: Constant Pool
  - 01 00 04 43 6F 64 65

    01: Constant Pool Info érték `(CONSTANT_Utf8)`

    00 04: 4 hosszú

    43 6F 64 65: A `CONSTANT_Utf8` értéke: `Code`
  - 01 00 04 6D 61 69 6E

    01: Constant Pool Info érték `(CONSTANT_Utf8)`

    00 04: 4 hosszú

    6D 61 69 6E: A `CONSTANT_Utf8` értéke: `main`
  - 01 00 03 28 29 56

    01: Constant Pool Info érték `(CONSTANT_Utf8)`

    00 03: 3 hosszú

    28 29 56: A `CONSTANT_Utf8` értéke: `()V`
- 00 21: Hozzáférési zászlók `(Public, Super)` - elhanyagolhatóak ebben az esetben
- 00 00: `This` osztály indexe a Constant Pool-ban
- 00 00: `Super` osztály indexe a Constant Pool-ban
- 00 00: Interfészek száma

- 00 00: Adattagok száma
- 00 01: Függvények száma
- 00 09 00 02 00 03 00 01 00 01 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00: Függvények
  - 00 09 00 02 00 03 00 01 00 01 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00
    00 09: Hozzáférési zászlók `(Public, Static)`
    00 02: Constant Poolban lévő indexe a függvénynek: `main`
    00 03: Függvény leírása (bemeneti paraméterek, visszatérési érték): `()V`
    00 01: Függvény attribútumainak száma
    00 01 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00: Attribútumok
    - 00 01 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00
      00 01: Constant Pool-ban lévő indexe az attribútumnak: `Code`
      00 00 00 0D: Attribútum hossza (`0D` = 13 bájt)
      00 00 00 00 00 00 00 01 B1 00 00 00 00: Attribútum
      - 00 00 00 00 00 00 00 01 B1 00 00 00 00
        00 00: Stack mérete
        00 00: Lokális változók száma
        00 00 00 01: Kód hossza
        B1: Kód `(B1 = return)`
        00 00: Kivételek száma
        00 00: Attribútum attribútumainak száma
- 00 00: Osztály attribútumainak száma

3.1.3. Adatszerkezetek

A class fájlnak megfelelően a két legfontosabb adattag a `stack` és a `local` (lokális) változók. A különböző instrukciók az ezeken lévő adatokkal dolgoznak, erre/ebbe helyeznek el megfelelő adatokat.

Az egyszerűség kedvéért a `stack` reprezentációjában az osztály típusát is elmentjük, a két adattag Java reprezentációja a `ClassFile` osztályban:

```
1 public List<Pair<Class<?>, Object>> stack = new ArrayList<>();
2 public Object[] local = new Object[65535];
```

(A `Pair` egy egyedi osztály, mely két adattagot tud eltárolni, más nyelvekben `tuple`-ként is ismeretes.)

A lokális változók maximális mennyiségét előre tudjuk, ez nem lehet több mint egy 16-bites előjel nélküli szám ($2^{16} = 65536$), alapból ennek az értéke egy 8-bites előjel nélküli szám ($2^8 = 256$) lenne, mivel a `store` és `load` utasításokat csak egy 8-bites előjel nélküli szám (az `index`) követi, viszont a `wide` utasítással a `store` és `load` utasítások módosíthatóak, hogy 2 db 8-bites előjel nélküli számot olvassanak be, tehát lényegében egy 16-bites előjel nélküli számot.

Gyakorlatban ez a szám csökkenthető lenne, tudhatjuk hogy futási időben mennyi lokális változója (illetve a `stack` nagyságát is tudhatjuk, tehát tömbként is reprezentálhatnánk) van egy metódusnak. Ez bővebben le van írva a továbbfejlesztési lehetőségekben.

Kényelmi szempontból létezik a `CodeIndex` osztály, amely lényegében egy `int` szám absztrakciója:

```
1 class CodeIndex {
2     private int index = 0;
3
4     ...
5 }
```

Az absztrakció oka hogy függvényeknek átadva lehessen módosítani ezt a számot; a szám a jelenlegi index a kódot reprezentáló `byte` tömbben, megmondja hogy a tömbben lévő melyik indexen levő instrukciót kell végrehajtani.

Az absztrakció különösen észrevehető amikor az `if` és `goto` utasításokat hajtjuk végre, a `ClassFile` objektumunk lokális változója módosítható az `Instructions` osztály metódusain keresztül. Mivel a Java érték szerint adja át a paramétereket, ez egy sima `int` számmal nem lehetne megoldani.

3.1.4. Interpretálás algoritmusa

Az 1. algoritmus egy általános elágazás és korlátozás algoritmust (*Branch and Bound algorithm*) mutat be. A 3. lépésben egy megfelelő kiválasztási szabályt kell alkalmazni. Példa forrása: Acta Cybernetica (ez egy hiperlink).

1. algoritmus A general interval B&B algorithm

Funct IBB(S, f)

```
1: Set the working list L_W := {S} and the final list L_Q := {}
2: while ( L_W ≠ ∅ ) do
3:     Select an interval X from L_W                          ▷ Selection rule
4:     Compute lbf(X)                                         ▷ Bounding rule
5:     if X cannot be eliminated then                         ▷ Elimination rule
6:         Divide X into X^j, j = 1, ..., p, subintervals     ▷ Division rule
7:         for j = 1, ..., p do
8:             if X^j satisfies the termination criterion then ▷ Termination rule
9:                 Store X^j in L_W
10:            else
11:                Store X^j in L_W
12:            end if
13:        end for
14:    end if
15: end while
16: return L_Q
```

3.2. Erőforrás igények

A `Linux` operációs rendszeren beépített `time` programot (illetve a `hyperfine` programot) használva az erőforrás igények a tesztfájlokra az alábbiak (a tesztelt számítógép releváns specifikációi: Intel Core i7-8700k processzor 4.7 GHz-en, 16 GB DDR4 memória 2133 MT/s sebességgel):

Tesztfájl	**/usr/bin/java**		**Jabyinja**	
	Memória	Futási idő	Memória	Futási idő
Own/Arithmetic.class	37,1 MB	21,4 ms	47,6 MB	92,8 ms
Own/Arrayclass.class	34,9 MB	20,9 ms	54,6 MB	130,8 ms
Own/Arraylist.class	37,2 MB	21,4 ms	49,6 MB	87,1 ms
Own/Athrow.class	34,6 MB	20,4 ms	46,9 MB	61,7 ms
Own/Dup2.class	36,3 MB	20,3 ms	39,2 MB	45,6 ms
Own/Inheritence.class	34,8 MB	20,7 ms	51,9 MB	98,1 ms
Own/Instanceof.class	38,8 MB	20,2 ms	43,1 MB	64,2 ms
Own/Multianewarray.class	34,4 MB	20,6 ms	47,7 MB	62,9 ms
Own/Nested.class	39,3 MB	22,1 ms	47,2 MB	80,5 ms
Own/Ownclass.class	37,5 MB	20,2 ms	61,9 MB	135,5 ms
Own/SwitchAthrow.class	36,8 MB	20,5 ms	40,5 MB	43,9 ms
Own/Template.class	39,2 MB	21,2 ms	51,9 MB	86,5 ms

3.1. táblázat. A beépített `java` és az interpreter közötti erőforrás különbségek

3.3. Továbbfejlesztési lehetőségek

3.3.1. Invokedynamic utasítás

Az egyik legszembetűnőbb hiány a szakdolgozatban az egyik nem implementált utasítás, az `invokedynamic`, hiánya. Ez az utasítás számos helyen előfordul Java programokban, leginkább a lambda kifejezésekben (ezen belül is a Konkurens programozás tárgyon megismert `Executor` osztály paramétereként), illetve a kiírás során szöveg(ek) és változó(k) konkatenációjánál is ez használt.

Az utóbbi egyszerűen kiküszöbölhető a `-XDstringConcat=inline` flag-gel való fordítás során az `invokedynamic` utasítás lecserélődik `StringBuilder`-en keresztül lévő `invokevirtual` és `invokespecial` hívásokra.

Az előző viszont sajnos jelen állapotban nem megoldott, és nem is oldható meg egyszerűen. Ahhoz hogy lambda függvények működjenek, az `invokedynamic`-ot implementálni kell. Ehhez már az alapvető előkészület megvan, a class fájlban lévő `bootstrap` metódosuk egy külön adattag elemeiként el vannak helyezve. A továbbfejlesztés során csak a megfelelő `CallSite` helyet, illetve a class fájlban lévő constant pool általi `index`-eken levő metódusokkal (illetve paraméterekkel) kell meghívni az éppen leírt függvényt.

3.3.2. Java 7 előtti verziók támogatása

Viszonylag egyszerűen továbbfejleszthető a program hogy Java 7 előtti verzióval fordított class fájlokat is támogasson.

A hiányzó utasítások a `ret`, `jsr`, `jsr_w`, ezek mindegyikéhez csak az szükséges, hogy a megfelelő index-re ugorjunk, a `jsr` és `jsr_w` utasítások során a visszatérési címet pedig a `stack`-re helyezzük.

Természetesen mindegyik utasítás során a megfelelő index-et is be kell olvasnunk a class fájlból, amely a lokális változó megfelelő indexére (`ret`), vagy egy adott számot (`jsr`, `jsr_w`) határoz meg, amely a visszatérési cím, illetve az ugrási cím.

3.3.3. Erőforrás igény

A futási idő táblázata alapján látható hogy a program exponenciálisan lassabb, mint a beépített `java` interpretáló program. A program több memóriát is igényel mint szükséges lenne. Ezeknek számos oka is van, ezek közül pár:

- A nem beépített osztályok megfelelő konstruktorait minden egyes alkalommal a program egyesével keresi ki a program. Ez a limitáció nagyon szembetűnő ha sok saját osztállyal dolgozunk. Ilyenkor a futási idő mégjobban lassul
- A `stack`-nek maximális értéket (65536) foglal a program minden nem beépített függvény meghívása során, viszont a class fájlban ennek a maximális értéke le van írva a megfelelő függvény attribútumaként.
- A különböző class fájlok beolvasásának eredménye nincs elmentve, ha egy fájlt be kell olvasnunk, akkor azt minden egyes alkalommal külön-külön megteszünk, ha az eredményt elmentenénk akkor drasztikusan lehetne a sebességen gyorsítani.

3.3.4. További tesztelés

A szakdolgozat írása során megpróbáltam az alapos tesztelésre figyelni, ezért is vannak az alapvető instrukciót egyesével tesztelve (minden tesztfájl-ban külön-külön instrukciók szerepelnek). Viszont a tökéletes program nem létezik, elképzelhető hogy valahol nincs megfelelően a `stack` törölve, vagy valamely instrukció mégsem helyes. Tehát a programban elképzelhető a probléma. Ezt a még alaposabb teszteléssel minél inkább meg lehetne cáfolni.

Ehhez egy példa még több tesztfájl mellékelése. A tesztelő környezetbe (Python szkript) viszonylag egyszerűen be lehet helyezni új teszt fájlokat, amely leellenőrzi hogy megfelelő-e a program futása. Továbbfejlesztésként lehet Java programokat írni, majd ezeket a tesztelő környezethez hozzáadni, és ellenőrizni hogy jól fut-e le a program.

3.4. Érdekességek a JVM specifikációból

4. fejezet

Összegzés

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In eu egestas mauris. Quisque nisl elit, varius in erat eu, dictum commodo lorem. Sed commodo libero et sem laoreet consectetur. Fusce ligula arcu, vestibulum et sodales vel, venenatis at velit. Aliquam erat volutpat. Proin condimentum accumsan velit id hendrerit. Cras egestas arcu quis felis placerat, ut sodales velit malesuada. Maecenas et turpis eu turpis placerat euismod. Maecenas a urna viverra, scelerisque nibh ut, malesuada ex.

Aliquam suscipit dignissim tempor. Praesent tortor libero, feugiat et tellus porttitor, malesuada eleifend felis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam eleifend imperdiet lorem, sit amet imperdiet metus pellentesque vitae. Donec nec ligula urna. Aliquam bibendum tempor diam, sed lacinia eros dapibus id. Donec sed vehicula turpis. Aliquam hendrerit sed nulla vitae convallis. Etiam libero quam, pharetra ac est nec, sodales placerat augue. Praesent eu consequat purus.

Köszönyetnyilvánítás

Petes Márton (ELTE IK PTI BSc): Az elekadásaim során elképesztően sok segítséget nyújtott, nélküle nem tudom hogy meglett volna-e a szakdolgozat

Ábrák jegyzéke

Táblázatok jegyzéke

Algoritmusjegyzék

Forráskódjegyzék