UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Aleksandar Stefanović

RAZREŠAVANJE REFLEKTIVNIH POZIVA STATIČKOM ANALIZOM JAVA BAJTKODA

master rad

Beograd, 2025.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Mirko Spasić, docent
Univerzitet u Beogradu, Matematički fakultet

Ivan Ristović, asistent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Zahvaljujem se mentoru prof. dr Mileni Vujošević Janičić i kolegi Ivanu Ristoviću na vrednim savetima i pomoći tokom pisanja rada. Zahvalnost dugujem i Vojinu Jovanoviću, ali i svim ostalim kolegama iz kompanije Oracle Labs, na pomoći, strpljenju i motivaciji.

Naslov master rada: Razrešavanje reflektivnih poziva statičkom analizom Java bajtkoda

Rezime: Strategija kompilacije unapred (engl. *Ahead-of-Time*, skraćeno AOT) pruža bolje performanse Java programa u ranim fazama izvršavanja i manje memorijsko zauzeće u odnosu na standardnu strategiju kompilacije tokom izvršavanja (engl. *Just-in-Time*, skraćeno JIT) posredstvom Java virtuelne mašine [41]. Ovo čini AOT kompilaciju posebno pogodnom u kontekstu kratkih programa i izvršavanja u oblaku, gde se izvršavanje programa naplaćuje na osnovu utrošenog procesorskog vremena i memorije.

Trenutno najznačajnije rešenje za AOT kompilaciju Java programa je kompilatorska infrastruktura *GraalVM* [22] koja, radi poboljšanja performansi rezultujuće izvršive datoteke, koristi pretpostavku da je sav kôd koji program može da izvrši dostupan pre početka izvršavanja, tj. tokom njegove kompilacije [41]. Ova pretpostavka uvodi određena ograničenja u dinamičkim mogućnostima jezika, poput refleksije. U cilju prevazilaženja tih ograničenja, jedan od mehanizama koje *GraalVM* koristi je statička analiza programa radi automatskog razrešavanja reflektivnih poziva.

Trenutna implementacija pomenute analize u sistemu *GraalVM* zavisi od kompilatorskih optimizacija i stoga dovodi do nepredvidivog ponašanja, što može prouzrokovati neočekivane rezultate izvršavanja programa. Glavni cilj rada je razvoj nove statičke analize za razrešavanje reflektivnih poziva koja nije podložna uticaju kompilatorskih optimizacija, kao i njena implementacija i evaluacija u okviru sistema *GraalVM*.

Ključne reči: statička analiza, Java, refleksija, GraalVM

Sadržaj

Glava 1

Uvod

Još od najranijih verzija, programski jezik Java pruža podršku za refleksiju — mogućnost programa da vrši upite nad svojom strukturom i modifikuje je, čak i tokom izvršavanja programa. Iako se refleksija pokazala kao moćan alat u određenim situacijama, poput razvoja biblioteka za serijalizaciju i deserijalizaciju podataka, ona predstavlja fundamentalan kamen spoticanja u kontekstu statičke analize programa [12], tj. analize programa bez njegovog izvršavanja. Ovo je najočiglednije u problemu analize dostižnosti koda — ukoliko se proizvoljan kôd može učitavati i izvršavati na osnovu uslova i vrednosti poznatih tek u fazi izvršavanja programa, onda je sav dostupan kôd potencijalno dostižan.

Kompilator *Native Image* u okviru sistema *GraalVM* [22] pruža mogućnost generisanja izvršive binarne datoteke od Java programa. Radi postizanja prihvatljive veličine generisanih izvršivih datoteka, kompilator koristi analizu dostižnosti koda, pri čemu se refleksija, u opštem slučaju, zanemaruje. Međutim, u ograničenim slučajevima, neki reflektivni pozivi se mogu automatski razrešiti specijalizovanom statičkom analizom.

Trenutno implementirana analiza za razrešavanje reflektivnih poziva u okviru sistema *GraalVM* zasnovana je na analizi *Graal* međureprezentacije i zavisi od kompilatorskih optimizacija, što može dovesti do nepredvidivog ponašanja programa. Cilj ovog rada je implementacija statičke analize za razrešavanje reflektivnih poziva na nivou Java bajtkoda, čime se prevazilazi prethodno pomenuti problem zavisnosti od optimizacija.

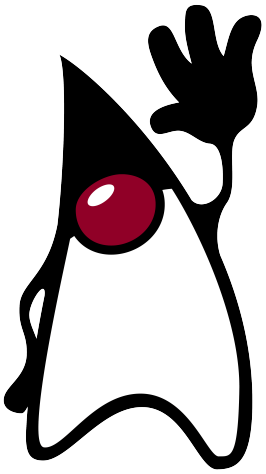
Poglavlje 2 opisuje programski jezik Java, sa posebnim akcentom na standardni proces izvršavanja Java programa i strukturu Java bajtkoda, tj. međureprezentaciju programa pogodnu za izvršavanje na Java virtuelnoj mašini. Poglavlje 3 opisuje

kompilatorsku infrastrukturu *GraalVM* i kompilator *Native Image* u okviru nje, kao i pretpostavke koje se uvode radi postizanja visokih performansi generisanog izvršivog programa. Poglavlje 4 opisuje glavni doprinos ovog rada — statičku analizu za razrešavanje reflektivnih poziva i motivaciju za njenu implementaciju na nivou Java bajtkoda. Konačno, poglavlje 5 predstavlja evaluaciju implementirane analize nad referentnim programom *Spring PetClinic*, veb aplikacijom koja demonstrira mogućnosti radnog okvira *Spring*.

Glava 2

Izvršavanje Java programa

Programski jezik Java je objektno orijentisani jezik višeg nivoa sa automatskim upravljanjem memorijom [10]. Pored toga, Java je statički tipizirana, što znači da se zaključivanje tipova vrši u fazi kompilacije programa, kao i jako tipizirana, tj. mogućnosti za implicitne konverzije između tipova su minimalne. Java je nastala 1995. godine, a njen glavni tvorac je Džejms Gosling (engl. *James Gosling*), koji je tada radio za kompaniju *Sun Microsystems*. Na slici 2.1 je prikazan *Duke*, maskota programskog jezika Java.



Slika 2.1: *Duke* — manje poznata maskota programskog jezika Java

Još od samog nastanka, Java je promovisana pod sloganom „napiši jednom, izvršavaj svuda" (engl. *write once, run anywhere*), što je značilo da se Java programi, bez izmena, mogu izvršavati na mnoštvu različitih uređaja, nezavisno od njihove procesorske arhitekture ili operativnog sistema. Da bi ovo bilo moguće, Java programi

se ne prevode u izvršive, binarne datoteke, već se izvršavaju uz pomoć tzv. Java virtuelne mašine [13].

Poglavlje 2.1 daje detaljniji opis standardnog načina izvršavanja Java programa i Java virtuelne mašine. Poglavlje 2.2 opisuje neke dinamičke odlike programskog jezika Java, poput mogućnosti učitavanja novog koda u fazi izvršavanja. Poglavlje 2.3 opisuje strukturu Java međukoda, tzv. bajtkoda (engl. *bytecode*), a poglavlje 2.4 pruža osvrt na metode analize i modifikacije bajtkoda.

2.1 Java virtuelna mašina

Java virtuelna mašina (engl. *Java Virtual Machine*, skraćeno JVM) je apstraktni računarski model, te programi koje njena implementacija izvršava ne moraju da prave pretpostavke o arhitekturi računara. U svojoj osnovi, JVM definiše skup instrukcija od kojih program može biti sastavljen, kao i način na koji te instrukcije menjaju stanje izvršavanja programa u okviru ovog apstraktnog modela. Zbog ove apstrakcije, za izvršavanje programa na određenoj platformi potrebna je samo implementacija Java virtuelne mašine za tu platformu. Opis Java virtuelne mašine zadat je njenom specifikacijom [13].

Uprkos nazivu, Java virtuelna mašina ne izvršava Java programe direktno, već za izvršavanje koristi binarnu međureprezentaciju koja se često naziva i bajtkod. Prema tome, na Java virtuelnoj mašini se ne moraju nužno izvršavati samo Java programi, već to mogu biti programi pisani u bilo kom programskom jeziku koji se može prevesti na nivo bajtkoda. Osim Jave, neki poznatiji programski jezici koji se oslanjaju na JVM za izvršavanje su *Kotlin*, *Scala* i *Clojure*.

Java izvorni kôd se prevodi u bajtkod pomoću Java kompilatora, i to pre samog izvršavanja, tj. pokretanja Java virtuelne mašine. Program `javac` je jedna implementacija ovog kompilatora [34]. Listinzi 2.1 i 2.2 prikazuju jednostavan Java program i tekstualni prikaz dela njemu odgovarajućeg bajtkoda. Detaljniji opis Java bajtkoda dat je u poglavlju 2.3.

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

Listing 2.1: Izvorni kôd jednostavnog Java programa koji ispisuje *Hello, world!*

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #7  // Field System.out:PrintStream
       3: ldc           #13 // String Hello, world!
       5: invokevirtual #15 // Method PrintStream.println(String)
       8: return
```

Listing 2.2: Tekstualni prikaz bajtkoda metode `main` programa *Hello, world!*

Organizacija memorije u Java virtuelnoj mašini

Memorija dodeljena programu koji se izvršava na Java virtuelnoj mašini je podeljena u više segmenata. Ova podela je slična tradicionalnoj podeli memorije u slučaju programa koji se ne izvršavaju posredstvom virtuelne mašine, već posredstvom operativnog sistema računara. Naime, memorija programa u JVM se deli na naredne segmente:

Stek segment — Svaka nit koja se izvršava ima svoj programski stek (engl. *stack*). Ovaj stek sadrži podatke koji su potrebni za izvršavanje metoda, smeštene u tzv. stek okvire (engl. *stack frame*). Stek okviri se sastoje iz tabele lokalnih promenljivih (engl. *local variable table*) i steka operanada (engl. *operand stack*). Tabela lokalnih promenljivih sadrži vrednosti lokalnih promenljivih metoda koji se trenutno izvršava, kao i argumente sa kojima je taj metod pozvan, a stek operanada sadrži međurezultate izračunavanja tokom izvršavanja metoda.

Hip segment — Hip (engl. *heap*) segment Java virtuelne mašine je deljen između svih niti. On sadrži instance klasa i nizova alociranih tokom izvršavanja programa. Radi sprečavanja curenja memorije, hip segment se periodično čisti pomoću sakupljača otpadaka (engl. *garbage collector*), što programera oslobađa od potrebe za ručnim upravljanjem memorijom.

Segment metoda — Segment metoda (engl. *method area*) sadrži kôd metoda učitanih klasa. Ovaj segment je pandan segmentu koda u tradicionalnoj podeli memorije.

Skup konstanti — Skup konstanti (engl. *constant pool*) je zaseban za svaku učitanu klasu i sadrži vrednosti konstanti (poput konstantnih niski) na koje se referiše iz te klase. U tradicionalnoj podeli memorije, ovaj segment odgovara segmentu podataka.

Tipovi podataka u Java virtuelnoj mašini

Većina bajtkod instrukcija, o kojima će više reči biti u poglavlju 2.3, operiše nad podacima određenog tipa. Ovi tipovi se mogu podeliti na dve osnovne kategorije — na primitivne tipove i na referencne tipove.

Primitivni tipovi obuhvataju osnovne tipove podataka, poput celobrojnih vrednosti ili brojeva u pokretnom zarezu. Njihov tačan skup je dat u tabeli 2.1. Iako specifikacija JVM definiše primitivne tipove `boolean`, `byte`, `short` i `char`, njihove vrednosti se, prilikom izvršavanja, proširuju u vrednosti tipa `int`.

Primitivni tip	Opis
`boolean`	Bulovski tip, gde 1 predstavlja vrednost *true*, a 0 *false*
`byte`	8-bitni označeni ceo broj
`short`	16-bitni označeni ceo broj
`char`	16-bitni neoznačeni ceo broj koji predstavlja UTF-16 karakter
`int`	32-bitni označeni ceo broj
`long`	64-bitni označeni ceo broj
`float`	32-bitni broj u pokretnom zarezu, prema standardu IEEE-754
`double`	64-bitni broj u pokretnom zarezu, prema standardu IEEE-754
`returnAddress`	Predstavlja adresu određene bajtkod instrukcije

Tabela 2.1: Primitivni tipovi u JVM

Referencni tipovi obuhvataju reference na objekte koji su alocirani na JVM hipu, tj. njihove memorijske adrese. Ove reference mogu biti reference na instancu neke klase, reference na nizove, ili reference na objekte koji implementiraju određene interfejse. Jedan specijalan tip reference je `null` referenca, koja ne pokazuje ni na jedan objekat.

JIT kompilacija

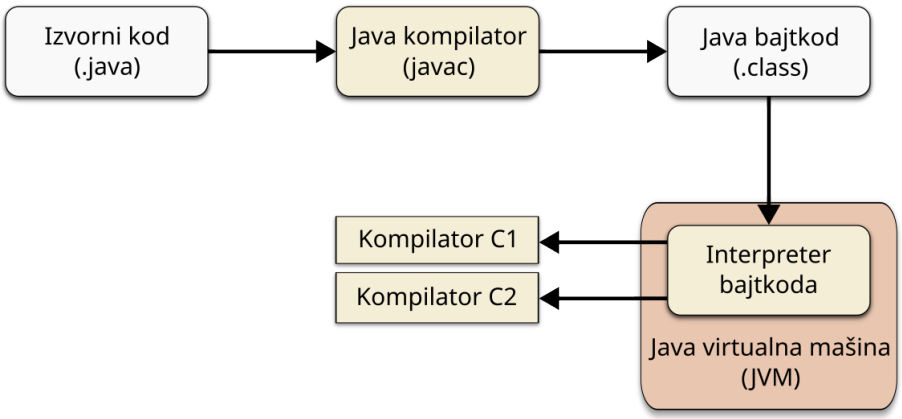
U standardnim okolnostima, izvršavanje Java programa podrazumeva postojanje programske implementacije Java virtuelne mašine. Neke od poznatijih implementacija su *HotSpot* JVM [16] i *OpenJ9* JVM [9].

U osnovnom obliku, Java virtuelna mašina se može implementirati kao interpreter bajtkoda. Međutim, ovakav pristup postaje relativno neefikasan u realističnim slučajevima upotrebe. Iz tog razloga, implementacije poput *HotSpot* Java virtuelne mašine koriste pristup kompilacije u fazi izvršavanja (engl. *Just-in-Time*, skraćeno JIT). Naime, JIT kompilacija podrazumeva prevođenje određenog dela bajtkoda u

optimizovani mašinski kôd za vreme izvršavanja programa. Tada se, umesto spore interpretacije, može izvršiti dobijeni mašinski kôd.

JIT kompilacija se ne vrši za celokupan bajtkod programa, već samo za one delove programa koji se izvršavaju dovoljno često (npr. kôd koji se izvršava u okviru neke frekventne petlje). Određivanje delova programa koji će biti prevedeni u mašinski kôd se vrši na osnovu profila prikupljenih tokom izvršavanja programa. Osnovna metrika koja se može koristiti prilikom odluke da li treba koristiti JIT kompilaciju za neki metod je broj poziva tog metoda.

HotSpot JVM koristi JIT kompilaciju u više nivoa, gde se metodi mogu inicijalno kompilirati JIT kompilatorom *C1*, a onda, po potrebi i sa preciznijim profilima, rekompilirati JIT kompilatorom *C2* koji sprovodi intenzivnije optimizacije koda [17]. Slika 2.2 prikazuje proces izvršavanja Java programa na *HotSpot* JVM — od izvornog Java koda, do bajtkod međureprezentacije, njegove interpretacije i, konačno, JIT kompilacije i daljeg izvršavanja.



Slika 2.2: Proces izvršavanja programa i JIT kompilacije u *HotSpot* JVM

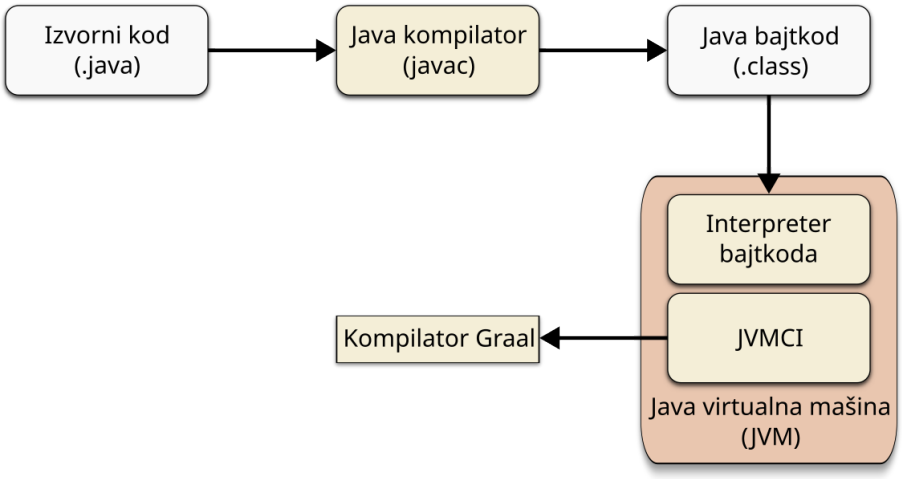
Jedna mana JIT kompilacije je tzv. problem hladnog pokretanja (engl. *cold start*). Naime, izvršavanje programa na Java virtuelnoj mašini je relativno neefikasno u ranim fazama izvršavanja usled toga što se program interpretira sve dok se ne prikupe profili izvršavanja i JIT kompilator ne kompilira često pozivane metode [14]. Ovaj problem je posebno osetan kod programa sa kratkim životnim vekom, u kom slučaju samo pokretanje Java virtuelne mašine može zahtevati veći deo vremena izvršavanja programa. U trenutku pisanja ovog master rada, aktivno se radi na smanjivanju efekata ovog problema [28, 19, 41].

JVM kompilatorski interfejs

JVM kompilatorski interfejs (engl. *JVM Compiler Interface*, skraćeno JVMCI) omogućava direktnu interakciju sa Java virtuelnom mašinom iz Java koda. Neke od mogućnosti koje JVMCI pruža su pristup bajtkodu učitanih klasa ili instalacija kompiliranih metoda u kôd keš (engl. *code cache*) virtuelne mašine[1].

JVMCI je inicijalno razvijen iz potrebe za jednostavnijim procesom razvoja i održavanja JIT kompilatora za Java virtuelnu mašinu [18]. Tradicionalno, JIT kompilatori, kao i veći deo JVM, bili su razvijani u programskim jezicima C ili C++. Primer ovoga su upravo JIT kompilatori *C1* i *C2* koji se koriste u okviru *HotSpot* JVM. Iako ovaj izbor doprinosi velikoj efikasnosti izvršavanja na Java virtuelnoj mašini, on takođe uvodi i nezanemarljiv nivo dodatne složenosti u proces daljeg razvoja i održavanja JIT kompilatora, poput potrebe za ručnim upravljanjem memorijom. Ovaj problem se prevazilazi time što JVMCI omogućava razvoj JIT kompilatora u programskom jeziku Java.

Java kompilatorski interfejs je omogućio i razvoj kompilatorske infrastrukture *GraalVM* [22], koja, između ostalog, pruža alternativu za JIT kompilatore *C1* i *C2* u vidu kompilatora *Graal* [21]. Slika 2.3 prikazuje proces izvršavanja programa i JIT kompilacije posredstvom kompilatora *Graal*. Detaljniji opis projekta *GraalVM* dat je u poglavlju 3.



Slika 2.3: Proces izvršavanja programa i JIT kompilacije posredstvom kompilatora *Graal*

[1]Kôd keš je prostor u kojem Java virtuelna mašina čuva kompilirani mašinski kôd.

2.2 Dinamičke odlike programskog jezika Java

Upotreba Java virtuelne mašine za izvršavanje programa otvara mogućnost programskom jeziku Java da ima različite dinamičke osobine. Naime, JVM standardno sprovodi učitavanje klasa, tj. bajtkoda koji odgovara nekoj klasi, u fazi izvršavanja — tek onda kada su oni potrebni za dalje izvršavanje programa. Ovo znači da je moguće učitavanje novog koda[2], njegova modifikacija i instrumentacija, pa čak i kreiranje potpuno novog koda, sve tokom izvršavanja programa.

Veliki broj popularnih Java radnih okvira (engl. *framework*) i biblioteka interno koriste ove mogućnosti Jave u svojoj implementaciji. Čest primer toga su biblioteke za serijalizaciju i deserijalizaciju objekata, poput biblioteke *Jackson* [6]. Još jedan zanimljiv primer je okvir *Hibernate* [8] koji koristi ove dinamičke odlike zarad realizacije objektno-relacionog preslikavanja[3] (engl. *object-relational mapping*, skraćeno ORM) koje se može podešavati korisnički zadatim anotacijama ili konfiguracionim datotekama.

Refleksija u programskom jeziku Java

Refleksija je mogućnost programa da vrši upite nad svojom strukturom i modifikuje je, u opštem slučaju i u fazi izvršavanja [15]. Programski jezik Java, još od svojih najranijih verzija, većinski implementira podršku za refleksiju [7] kroz klase iz paketa `java.lang.reflect`, kao i klasu `java.lang.Class`. Svaka klasa (ili interfejs) koju učita JVM ima svoj `Class` objekat. Ovi objekti sadrže informacije o učitanoj klasi, pa se shodno tome koriste kao ulazna tačka za upotrebu refleksije. Deo javnog interfejsa klase `Class` čine naredne metode:

- `Class<?> forName(String className)` — vraća, ukoliko postoji, klasu sa zadatim imenom;
- `Field[] getFields()` — vraća sva javna polja date klase;
- `Method[] getMethods()` — vraća sve javne metode date klase;
- `Constructor[] getConstructors()` — vraća sve javne konstruktore date klase;

[2]Kôd u ovom kontekstu se odnosi na bajtkod klasa.
[3]Automatizovano preslikavanje iz Java objekata u SQL tabele i obrnuto.

- `Method getMethod(String name, Class<?>... parameterTypes)` — vraća, ukoliko postoji, javni metod date klase sa zadatim imenom i tipovima parametara;
- `boolean isInterface()` — vraća vrednost `true` ukoliko je data klasa interfejs, a `false` u protivnom.

Jedna od mogućnosti refleksije u Javi je dobijanje reference na metod klase na osnovu njegovog imena i parametara, nakon čega se taj metod može i izvršiti. Ovako nešto bi bilo teško, ili čak i nemoguće, izvesti u programskim jezicima bez ugrađene podrške za refleksiju[4]. Naredni listinzi ilustruju ovu mogućnost — klasa `RemoteHello`, prikazana listingom 2.3, se nalazi na udaljenom računaru, a klasa `RemoteGrabber`, prikazana listingom 2.4, učitava klasu `RemoteHello`, reflektivnim pozivom `getMethod` dobija referencu na njen metod `printGreeting` i izvršava ga. Rezultat izvršavanja je prikazan listingom 2.5.

```
public class RemoteHello {
    public static void printGreeting() {
        System.out.println("Hello from Poincare!");
    }
}
```

Listing 2.3: Klasa `RemoteHello` koja se nalazi na udaljenom računaru

```
import java.net.*;
import java.lang.reflect.*;

public class RemoteGrabber {
    public static void main(String[] args) throws Throwable {
        String myPoincare =
            "http://poincare.math.rs/~aleksandar.stefanovic/java_examples/";
        // Create a class loader which checks for classes at the specified URL(s).
        URLClassLoader poincareLoader = new URLClassLoader(new URL[]{
            URI.create(myPoincare).toURL() });
        // Try to dynamically load the PoincareHello class.
        Class<?> poincareClass = poincareLoader.loadClass("RemoteHello");
        // Reflectively get a reference to its printGreeting method.
        Method poincareGreeting = poincareClass.getMethod("printGreeting");
        // Invoke the greeting method.
        poincareGreeting.invoke(null);
    }
}
```

Listing 2.4: Program koji dinamički učitava klasu `RemoteHello` sa udaljenog računara i reflektivno poziva njen metod `printGreeting`

[4]Neki programski jezici imaju podršku za statičku refleksiju u kojoj su reflektivni upiti ograničeni na fazu kompilacije [3].

```
$ java PoincareGrabber
Hello from Poincare!
```

Listing 2.5: Rezultat izvršavanja programa `RemoteGrabber`

2.3 Struktura Java bajtkoda

Java bajtkod je binarna međureprezentacija programa za izvršavanje na Java virtuelnoj mašini. U kontekstu programskog jezika Java, bajtkod se definiše na nivou klasa, tj. svaka klasa ima svoju zasebnu bajtkod reprezentaciju, nezavisnu od bajtkoda drugih klasa. Ova međureprezentacija se obično[5] čuva u datotekama sa ekstenzijom `.class` koji se dobijaju kao rezultat kompilacije koda pisanog u izvornom jeziku — u slučaju programskog jezika Java, to se obično postiže kompilatorom `javac`.

Listing 2.6 prikazuje strukturu bajtkoda po specifikaciji JVM [13], zapisane u formatu struktura u programskom jeziku C. Oznake `u2` i `u4` redom predstavljaju dvobajtne i četvorobajtne neoznačene cele brojeve, a ostale oznake, poput `method_info`, predstavljaju složenije strukture. U nastavku će biti dat kratak opis svakog od polja u ovoj strukturi.

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Listing 2.6: Struktura bajtkoda po specifikaciji JVM [13]

`magic` — Četvorobajtni broj kojim počinje bajtkod svake klase. Njegova vrednost, u heksadekadnom zapisu, je `0xCAFEBABE`.

[5] Bajtkod može biti generisan i u fazi izvršavanja direktno od strane JVM.

`minor_version`, `major_version` — Oznake za verziju bajtkod formata koja se koristi. Sve starije verzije bajtkod formata su kompatibilne sa novijim verzijama, te se i dalje mogu izvršavati na JVM.

`constant_pool_count`, `constant_pool[]` — Niz konstanti, poput niski, naziva klasa i polja, na koje se referiše u bajtkodu. Indeksiranje u nizu `constant_pool` počinje od vrednosti 1.

`access_flags` — Bitovska maska koja kodira određena svojstva klase kojoj odgovara dati bajtkod, poput njenog modifikatora pristupa i apstraktnosti.

`this_class` — Indeks na stavku u nizu `constant_pool` koja sadrži naziv klase kojoj odgovara dati bajtkod.

`super_class` — Indeks na stavku u nizu `constant_pool` koja sadrži naziv natklase za klasu kojoj odgovara dati bajtkod. U slučaju da ova klasa nema natklasu, vrednost ovog polja je 0.

`interfaces_count`, `interfaces[]` — Niz interfejsa koje data klasa implementira. Svaki element ovog niza predstavlja indeks na stavku u nizu `constant_pool` koja sadrži naziv interfejsa.

`fields_count`, `fields[]` — Broj polja i niz polja koja su deklarisana u datoj klasi. Nasleđena polja iz natklasa se ne nalaze u ovom nizu.

`methods_count`, `methods[]` — Broj metoda i niz metoda koje su deklarisane u datoj klasi. Nasleđene metode iz natklasa se ne nalaze u ovom nizu. Metode, tj. strukture `method_info`, između ostalog sadrže i sam kôd metode u vidu niza instrukcija.

`attributes_count`, `attributes[]` — Broj atributa i niz atributa date klase. Između ostalog, ovde su sadržani i podaci o anotacijama date klase.

Alat `javap` [35] omogućava tekstualni prikaz sadržaja `.class` datoteka, sa svim prethodno opisanim elementima. U zavisnosti od prosleđenih opcija, taj prikaz može biti više ili manje detaljan. Listing 2.7 sadrži prikaz bajtkoda klase `HelloWorld` date u listingu 2.1, dobijen pokretanjem naredbe `javap -verbose HelloWorld.class`. Neki delovi prikaza su izostavljeni zarad čitljivosti, što je označeno sa tri tačke (. . .).

```
Classfile HelloWorld.class
  Last modified Jun 7, 2025; size 427 bytes
  SHA-256 checksum 370cb6840017ff3d890afa0991323a1965f374c382f0423fe49d28a07672e82d
  Compiled from "HelloWorld.java"
public class HelloWorld
  minor version: 0
  major version: 69
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #21                         // HelloWorld
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   ...
   #7 = Fieldref           #8.#9          //
       java/lang/System.out:Ljava/io/PrintStream;
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // Hello, world!
  #14 = Utf8               Hello, world!
  #15 = Methodref          #16.#17        //
      java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  ...
{
  public HelloWorld();
  ...

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #13 // String Hello, world!
         5: invokevirtual #15 // Method
             java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 3: 0
        line 4: 8
}
SourceFile: "HelloWorld.java"
```

Listing 2.7: Tekstualni prikaz bajtkoda klase `HelloWorld`, dobijen pomoću alata `javap`

Instrukcije Java virtuelne mašine

Kôd metoda je, na nivou bajtkoda, zadat nizom instrukcija čiji su skup i semantika definisani specifikacijom JVM [13]. Jedno svojstvo ovog skupa instrukcija, od kojeg i potiče naziv *bajtkod*, je da se svaka oznaka instrukcije (engl. *opcode*) može kodirati jednim bajtom, tj. ukupno postoji manje od $2^8 = 256$ instrukcija

koje JVM prepoznaje. Pored oznake, neke instrukcije zahtevaju i dodatne operande poput odgovarajućeg indeksa u `constant_pool` strukturi, kao što je slučaj kod instrukcija koje referišu na neki metod, polje ili konstantu. Konkretan primer ovoga je instrukcija `ldc` koja čita vrednost neke konstante iz `constant_pool` strukture, pa kao dodatni operand zahteva indeks širine jednog bajta koji referiše na odgovarajuću konstantu.

Svakoj instrukciji u okviru jednog metoda se može pridružiti celobrojna oznaka u vidu broja bajtova od početka niza instrukcija do početka te instrukcije (engl. *bytecode index*, skraćeno BCI). Zbog dodatnih operanada nekih instrukcija, ove oznake nisu neprekidne. Listing 2.8 prikazuje niz instrukcija metoda `main` *Hello, world!* programa, pri čemu su instrukcije navedene u formatu `BCI: opcode <dodatni operandi>`. Dodatni operandi u listingu koji su formata `#n` označavaju n-ti indeks u strukturi `constant_pool`.

```
0: getstatic     #7
3: ldc           #13
5: invokevirtual #15
8: return
```

Listing 2.8: Niz instrukcija `main` metoda *Hello, world!* programa

Veliki deo bajtkod instrukcija operiše samo nad određenim tipom podataka. Tako, na primer, za sabiranje brojeva postoje četiri instrukcije — `iadd`, `ladd`, `fadd`, `dadd`, gde svaka vrši sabiranje brojeva koji pripadaju, redom, tipu `int`, `long`, `float` i `double`.

Izvršavanje bajtkoda metoda počinje od prve instrukcije u odgovarajućem nizu instrukcija i nastavlja se redom, osim u slučaju instrukcija skoka ili izbacivanja izuzetka (engl. *exception*), kada se dalje izvršavanje može nastaviti od neke druge instrukcije. Instrukcije mogu modifikovati programski stek niti na kojoj se dati metod izvršava. Ovo obuhvata dodavanje, uklanjanje ili modifikaciju stek okvira na programskom steku.

Stek operanada je struktura sa koje instrukcije uzimaju vrednosti potrebne za njihovo izvršavanje, ali i postavljaju vrednosti koje predstavljaju rezultat njihovog izvršavanja. Sa druge strane, tabela lokalnih promenljivih služi za preslikavanje lokalnih promenljivih i argumenata metoda u njihove vrednosti, pri čemu su promenljive i argumenti predstavljeni celobrojnim indeksom u tabeli. Listing 2.9 opisuje proces izvršavanja jednostavnog niza bajtkod instrukcija koji sabira brojeve 1 i 4, čuva ih u

lokalnoj promenljivoj, tu promenljivu inkrementira i, konačno, postavlja njenu novu vrednost na stek.

```
iconst_1 // Push int value 1 to the stack
iconst_4 // Push int value 4 to the stack
iadd     // Pop values from the stack (4 and 1) and push their sum (5) to the stack
istore_0 // Pop value from the stack (5) and store it into local variable 0
iinc 0 1 // Increment local variable 0 by 1 (5 + 1 = 6)
iload_0  // Push the value of local variable 0 (6) to the stack
```

Listing 2.9: Primer sabiranja brojeva, čuvanja rezultata i inkrementiranja promenljive

Vrednosti se, kako na steku operanada, tako i u tabeli lokalnih promenljivih, čuvaju u 32-bitnim okvirima. Ovo znači da je za vrednosti koje pripadaju tipovima `long` i `double` potrebno dva okvira, pa bi, na primer, čuvanje lokalne promenljive tipa `double` na indeksu 0 u tabeli lokalnih promenljivih zauzelo i indeks 1. Iz ovog razloga, određene instrukcije mogu imati nešto drugačije značenje u kontekstu manipulacije nad samim vrednostima u zavisnosti od tipa, tj. veličine tih vrednosti. Primer ovoga je instrukcija `POP2` koja uklanja dve ili jednu vrednost sa vrha steka u zavisnosti od toga da li vrednost na vrhu zauzima jedan ili dva okvira.

Detaljan opis semantike svih instrukcija može se naći u poglavlju 6 specifikacije JVM [13], a u nastavku sledi kategorizacija većeg dela instrukcija Java virtuelne mašine prema njihovoj nameni. Radi kraćeg zapisa, u nekim slučajevima će više instrukcija biti istovremeno opisano navođenjem sufiksa i prefiksa u formatu `<oznaka 1 | ... | oznaka n>`. Oznake `I`, `L`, `F` i `D` redom označavaju tipove `int`, `long`, `float` i `double` u instrukcijama, a `A` referencu.

Učitavanje konstanti — Veliki broj JVM instrukcija ima namenu postavljanja neke konstantne vrednosti na stek operanada. Ovde spadaju naredne instrukcije:

- `ICONST_<M1|0|1|2|3|4|5>` — Na vrh steka postavlja odgovarajuću vrednost tipa `int`, pri čemu `M1` označava vrednost -1.
- `LCONST_<0|1>` — Na vrh steka postavlja vrednosti 0 ili 1 tipa `long`.
- `FCONST_<0|1|2>` — Na vrh steka postavlja vrednost 0.0, 1.0 ili 2.0 tipa `float`.
- `DCONST_<0|1>` — Na vrh steka postavlja vrednost 0.0 ili 1.0 tipa `double`.
- `ACONST_NULL` — Na vrh steka postavlja `null` referencu.

- `BIPUSH n, SIPUSH n` — Na vrh steka postavlja `byte` ili `short` vrednost `n` promovisanu u tip `int` po postavljanju na stek.
- `LDC #idx, LDC_W #idx, LCD2_W #idx` — Na vrh steka postavlja vrednost sa indeksa `#idx` strukture `constant_pool`.

Upravljanje lokalnim promenljivama — Upravljanje lokalnim promenljivama se vrši pomoću izvršavanja operacija nad tabelom lokalnih promenljivih. Ovde spadaju naredne instrukcije:

- `<I|L|F|D|A>LOAD_<0|1|2|3>` — Vrednost odgovarajućeg tipa se čita sa indeksa označenog sufiksom instrukcije iz tabele lokalnih promenljivih i postavlja na vrh steka.
- `<I|L|F|D|A>LOAD n` — Vrednost odgovarajućeg tipa se čita sa indeksa `n` iz tabele lokalnih promenljivih i postavlja na vrh steka.
- `<I|L|F|D|A>STORE_<0|1|2|3>` — Sa vrha steka se uzima vrednost odgovarajućeg tipa i skladišti se u tabeli lokalnih promenljivih na indeksu označenom sufiksom instrukcije.
- `<I|L|F|D|A>STORE n` — Sa vrha steka se uzima vrednost odgovarajućeg tipa i skladišti se u tabeli lokalnih promenljivih na indeksu `n`.

Aritmetičke operacije — Aritmetičke operacije nad vrednostima u JVM su omogućene kroz naredne instrukcije (rezultat se postavlja na vrh steka):

- `<I|L|F|D>ADD` — Zbir dve vrednosti sa vrha steka.
- `<I|L|F|D>SUB` — Razlika dve vrednosti sa vrha steka, pri čemu je prva vrednost sa vrha umanjilac, a druga umanjenik.
- `<I|L|F|D>MUL` — Proizvod dve vrednosti sa vrha steka.
- `<I|L|F|D>DIV` — Količnik dve vrednosti sa vrha steka, pri čemu je prva vrednost sa vrha delilac, a druga deljenik.
- `<I|L|F|D>REM` — Ostatak pri deljenju dve vrednosti sa vrha steka, pri čemu je prva vrednost sa vrha delilac, a druga deljenik.
- `<I|L|F|D>NEG` — Promena znaka vrednosti sa vrha steka.
- `<I|L>SHL, <I|L>SHR, <I|L>USHR` — Bitovsko pomeranje ulevo, udesno i neoznačeno pomeranje udesno.

- `<I|L>AND`, `<I|L>OR`, `<I|L>XOR` — Bitovske operacije konjukcije, disjunkcije i ekskluzivne disjunkcije.
- `<F|D>CMPL`, `<F|D>CMPG`, `LCMP` — Poređenje prve i druge vrednosti sa vrha steka. Rezultat je -1, 0 ili 1 u zavisnosti od toga da li je prva vrednost veća, jednaka ili manja od druge vrednosti.
- `IINC n c` — Inkrementiranje lokalne promenljive na indeksu `n` tabele lokalnih promenljivih za vrednost `c`.

Konverzija tipova — Konverzija između vrednosti primitivnog tipa je omogućena kroz unarne instrukcije `I2F`, `I2D`, `L2F`, `L2D`, `F2I`, `F2L`, `F2D`, `D2I`, `D2L`, `D2F`, `L2I`, `I2L`, `I2B`, `I2S` i `I2C`.

Upravljanje objektima — Rad sa objektima, tj. instancama klasa i nizovima, je omogućen kroz naredne instrukcije:

- `NEW #idx` — Alocira novu instancu objekta klase koja je određena stavkom na indeksu `#idx` u `constant_pool` strukturi i postavlja njenu referencu na vrh steka.
- `GETFIELD #idx`, `GETSTATIC #idx` — Učitavaju, redom, vrednost polja instance ili statičkog polja određenog stavkom na indeksu `#idx` u `constant_pool` strukturi i postavljaju je na vrh steka.
- `PUTFIELD #idx`, `PUTSTATIC #idx` — Čuvaju prosleđenu vrednost u, redom, polje instance ili statičko polje određeno stavkom na indeksu `#idx` u `constant_pool` strukturi.
- `NEWARRAY pt`, `ANEWARRAY #idx`, `MULTIANEWARRAY #idx dim` — Alociraju instancu niza određenog tipa i dimenzije i postavljaju njegovu referencu na vrh steka.
- `<B|C|S|I|L|F|D|A>ALOAD` — Sa vrha steka skidaju, redom, indeks i referencu na niz, nakon čega se na vrh steka postavlja odgovarajući element datog niza.
- `<B|C|S|I|L|F|D|A>ASTORE` — Sa vrha steka skidaju, redom, vrednost, indeks i referencu na niz, nakon čega se u nizu na odgovarajućem indeksu čuva data vrednost.
- `ARRAYLENGTH` — Na vrh steka postavlja dužinu datog niza.

- `INSTANCEOF #idx, CHECKCAST #idx` — Proveravaju da li je prosleđeni objekat tipa određenog stavkom na indeksu `#idx` u `constant_pool` strukturi. `INSTANCEOF` instrukcija signalizira rezultat sa vrednošću 0 ili 1, dok `CHECKCAST` instrukcija to čini izbacivanjem `ClassCastException` izuzetka.

Operacije nad stekom — Nekoliko instrukcija je posvećeno upravljanju stekom operanada. Ove instrukcije pružaju mogućnost sklanjanja vrednosti sa vrha steka (`POP`, `POP2`), dupliranja vrednosti na steku (`DUP`, `DUP_X1`, `DUP_X2`, `DUP2`, `DUP2_X1`, `DUP2_X2`) i zamene vrednosti na vrhu steka (`SWAP`).

Kontrola toka — Instrukcije kontrole toka, bezuslovno ili na osnovu nekog uslova, nastavljaju izvršavanje metoda od ciljne instrukcije. Uslovni skokovi, poput instrukcije `IFNULL` (proverava da li je operand `null` referenca), izvršavaju proveru i, ukoliko je uslov zadovoljen, prebacuju izvršavanje metoda na ciljnu instrukciju (na osnovu njene `BCI` oznake), a u suprotnom se izvršavanje nastavlja od naredne instrukcije u nizu. Instrukcije bezuslovnog skoka, poput instrukcije `GOTO`, uvek prebacuju izvršavanje metoda na ciljnu instrukciju. Postoji i posebna podrška za `switch` naredbe u vidu instrukcija `LOOKUPSWITCH` i `TABLESWITCH`.

Na kontrolu toka utiču i instrukcije `RETURN` za završetak metoda bez povratne vrednosti (`void` metoda) i `<I|L|F|D|A>RETURN` za povratak iz metoda koji imaju povratnu vrednost.

Još jedan način na koji se kontrola toka može promeniti je bacanjem izuzetka. U ovom slučaju se prvo konsultuje tabela izuzetaka (engl. *exception table*) pridružena metodu da bi se proverilo da li se izuzetak može uhvatiti i obraditi, a u suprotnom se izvršavanje metoda završava i izuzetak se propagira na pozivaoca trenutnog metoda. Izuzetak se može baciti instrukcijom `ATHROW`.

Pozivanje metoda — JVM omogućava pozivanje metoda kroz naredne četiri instrukcije, pri čemu `#idx` odgovara indeksu u strukturi `constant_pool`:

- `INVOKEVIRTUAL #idx` — Poziv (virtuelnog) metoda nad objektom.
- `INVOKEINTERFACE #idx` — Poziv metoda interfejsa nad objektom koji implementira taj interfejs.
- `INVOKESPECIAL #idx` — Poziv metoda nad objektom, pri čemu se zaobilazi razrešavanje virtuelnih poziva. Ova instrukcija se koristi u pozivima

metoda natklase pomoću ključne reči `super`, pozivima konstruktora i pozivima privatnih metoda.

- `INVOKESTATIC #idx` — Poziv statičkog metoda.
- `INVOKEDYNAMIC #idx` — Poziv metoda sa dinamičkim (u vreme izvršavanja) vezivanjem.

Sinhronizacija — Java virtuelna mašina nudi određenu podršku za sinhronizaciju niti na samom nivou JVM instrukcija. Instrukcija `MONITORENTER` pokušava sa preuzimanjem monitora nad prosleđenim objektom (ili suspenduje izvršavanje dok on ne postane slobodan), dok instrukcija `MONITOREXIT` oslobađa monitor nad prosleđenim objektom.

2.4 Analiza i modifikacija Java bajtkoda

Zarad implementacije određenih alata, poput profajlera, debagera ili alata za praćenje pokrivenosti koda testovima, pristup bajtkodu programa koji se izvršava može biti neophodan. Jedan od načina za dobijanje tog pristupa je upotreba Java kompilatorskog interfejsa. Međutim, pošto je on projektovan sa razvojem kompilatora u vidu, ovo za veliki broj alata nije pogodno rešenje. Umesto toga, ovi alati se često implementiraju pomoću tzv. agenata — posebnih programa koji imaju privilegovan pristup Java virtuelnoj mašini i mogu, između ostalog, da presretnu, analiziraju i modifikuju bajtkod klasa koje JVM učitava. U nastavku će ukratko biti opisane dve vrste agenata — agenti zasnovani na Javinom interfejsu za instrumentaciju[6] (engl. *Instrumentation API*) i agenti zasnovani na JVM interfejsu za alate [29] (engl. *JVM Tool Interface*, skraćeno JVM TI).

Java agenti

Agenti koji se oslanjaju na Javin interfejs za instrumentaciju [26] nazivaju se Java agenti. Slično kao i standardne Java aplikacije, Java agenti moraju da imaju ulaznu tačku za svoje izvršavanje, pri čemu je to, umesto standardnog metoda `main`, specijalni metod `premain` ili `agentmain`. Koji od ova dva metoda je potrebno definisati zavisi od načina na koji se agent pokreće. Naime, ukoliko se agent pokreće preko komandne linije (uz pomoć opcije `-javaagent`), on će biti pokrenut za vreme

[6] Instrumentacija je proces umetanja dodatnih instrukcija u kôd programa radi prikupljanja podataka o njegovom izvršavanju, poput vremena izvršavanja pojedinih metoda.

inicijalizacije JVM (pre pokretanja metoda `main` glavne aplikacije), i u tom slučaju je potrebno definisati metod `premain`. Sa druge strane, za programabilno pokretanje agenta nakon pokretanje glavne aplikacije, potrebno je definisati metod `agentmain`. Agent može definisati oba metoda ukoliko je to potrebno.

Metodi za pokretanje agenta kao parametre imaju nisku `agentArgs`, koja predstavlja opcije sa kojima je agent pokrenut, kao i opcioni parametar `inst` tipa `Instrumentation`. Objekat ove klase, između ostalog, omogućava transformisanje bajtkoda učitanih klasa pomoću metoda `addTransformer(ClassFileTransformer transformer)`.

Interfejs `ClassFileTransformer` zahteva implementaciju metoda `transform` koji, kao jedan od parametara, prima bajtkod učitane klase kao niz bajtova, a kao rezultat izvršavanja vraća transformisani niz bajtova. Sama analiza i transformacija ovog niza bajtova se obično vrši kroz biblioteke specijalno namenjene za to, poput biblioteke ASM [2] ili klasa datih u paketu `java.lang.classfile`. Listing 2.10 prikazuje kôd jednostavnog Java agenta koji, bez transformacije sadržaja bajtkoda, ispisuje nazive svih klasa čije učitavanje presretne.

```
import java.lang.instrument.*;
import java.security.ProtectionDomain;

public class MyAgent {
    public static void premain(String agentArgs, Instrumentation inst) {
        inst.addTransformer(new MyTransformer());
    }
}

class MyTransformer implements ClassFileTransformer {
    @Override
    public byte[] transform(ClassLoader loader, String className, Class<?>
        classBeingRedefined, ProtectionDomain protectionDomain, byte[]
        classFileBuffer) {
        System.out.println("Class " + className + " intercepted");
        return classFileBuffer;
    }
}
```

Listing 2.10: Java agent koji ispisuje nazive svih klasa koje presretne

JVM TI agenti

JVM interfejs za alate, JVM TI, je skup funkcija implementiranih u programskom jeziku C koji omogućava interakciju sa Java virtuelnom mašinom, uključujući i instrumentaciju učitanih klasa. Alati koji koriste JVM TI, tj. JVM TI agenti, obično

su dinamički povezane biblioteke koje se mogu uvezati sa Java virtuelnom mašinom prilikom njenog pokretanja, što se radi pomoću opcija `-agentlib` ili `-agentpath`.

JVM TI agenti se primarno razvijaju implementiranjem odziva (engl. *callback*) na određene događaje (engl. *event*). Sledi par primera događaja na koje JVM TI agent može reagovati:

`jvmtiEventClassFileLoadHook` — Događaj koji odgovara učitavanju neke klase.

`jvmtiEventThreadStart` — Događaj koji odgovara pokretanju nove niti.

`jvmtiEventMethodEntry` — Događaj koji odgovara početku izvršavanja nekog metoda.

U odnosu na Java agente, JVM TI agenti imaju privilegovan pristup Java virtuelnoj mašini, što im pruža napredne mogućnosti poput suspendovanja izvršavanja niti ili njihovog ponovnog započinjanja. Ovo ih čini izuzetno pogodnim za implementaciju alata poput debagera. Primer ovoga je Java debager `jdb` [36] koji je implementiran kao JVM TI agent.

Glava 3

Kompilatorska infrastruktura *GraalVM*

Kompilatorska infrastruktura *GraalVM* [22] predstavlja skup alata za razvoj u programskom jeziku Java (engl. *Java development kit*, skraćeno JDK) koji pored standardnog izvršavanja Java programa [23] pruža i dodatne, napredne mogućnosti. Neke od njih su:

- Bolje performanse izvršavanja programa posredstvom JIT kompilatora *Graal* [21] koji se, pomoću JVM kompilatorskog interfejsa, može koristiti umesto *C1* i *C2 HotSpot* JIT kompilatora.
- Kompilacija Java programa u izvršivu binarnu datoteku uz pomoć kompilatora *Native Image* [30].
- Mogućnost implementacije različitih alata i integracije drugih programskih jezika u *GraalVM* infrastrukturu kroz radni okvir *Truffle* [37].
- Poliglotsko programiranje, tj. pozivanje i izvršavanje koda napisanog u proizvoljnom programskom jeziku podržanom od strane okvira *Truffle* iz drugog, matičnog jezika [33].

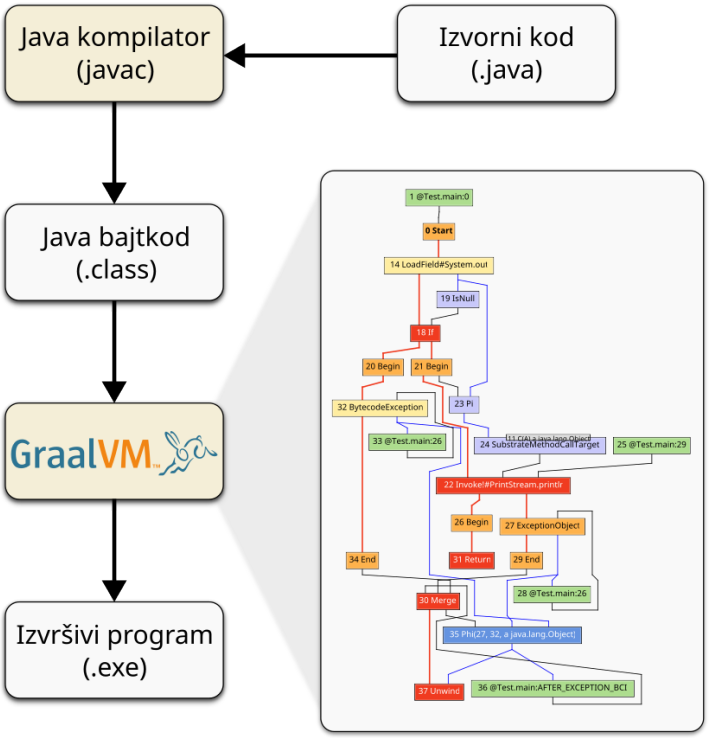
Projekat *GraalVM* je integrisan u zvanični Oracle JDK [27], a njegov veliki deo, *GraalVM Community Edition*, besplatan je za korišćenje i otvorenog je koda [24]. *GraalVM* je implementiran u programskom jeziku Java.

Poglavlje 3.1 detaljno opisuje kompilator *Native Image* i pretpostavke koje se uvode radi postizanja što boljih performansi i manje potrošnje resursa generisanih

izvršivih datoteka. Poglavlje 3.2 je posvećeno opisu restrikcija nad nekim dinamičkim mogućnostima Jave, konkretno refleksije, koje nastaju usled uvedenih pretpostavki.

3.1 AOT kompilacija Java programa

Kompilator *Native Image*, kao važan deo sistema *GraalVM*, pruža mogućnost kompilacije pre vremena izvršavanja (engl. *Ahead-of-Time*, skraćeno AOT) Java programa. Za razliku od JIT kompilacije, gde se prevođenje u mašinski kôd vrši samo za one delove programa koji se izvršavaju često, AOT kompilacija podrazumeva prevođenje celokupnog izvornog Java koda u izvršivu binarnu datoteku. *Native Image*, u procesu optimizacije i generisanja mašinskog koda, koristi grafovsku međureprezentaciju koda. Slika 3.1 prikazuje proces AOT kompilacije.



Slika 3.1: Proces kompilacije od Java izvornog koda do izvršive binarne datoteke — desni deo slike predstavlja grafovsku reprezentaciju programa u jednoj od faza kompilacije

Strategijom AOT kompilacije programa se izbegava problem hladnog pokretanja, tj. slabijih performansi programa u ranim fazama izvršavanja, koji je prisutan prilikom JIT kompilacije. AOT kompilacija omogućava i manju potrošnju memorijskih i procesorskih resursa jer za izvršavanje rezultujuće binarne datoteke nije

potrebna celokupna Java virtuelna mašina. Ove prednosti su od najvećeg značaja u kontekstu izvršavanja u oblaku (engl. *cloud computing*), gde se cena usluge izvršavanja najčešće obračunava prema utrošenom procesorskom vremenu i zauzetim memorijskim resursima [11, 1].

Grafovska međureprezentacija kompilatorske infrastrukture *GraalVM*

Umesto direktnog prevođenja Java bajtkoda u mašinski kôd, *GraalVM* koristi grafovsku međureprezentaciju koja se naziva *more čvorova* (engl. *sea of nodes*) [5] nad kojom se dalje sprovodi najveći deo naprednih optimizacija. Listinzi 3.1 i 3.2 redom prikazuju Java kôd i njemu odgovarajući bajtkod rekurzivnog metoda za računanje faktorijela, a slika 3.2 generisanu grafovsku međureprezentaciju[1] ovog metoda.

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

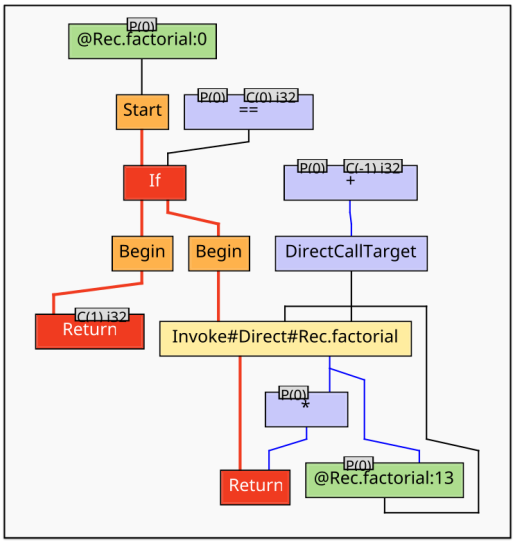
Listing 3.1: Izvorni Java kôd funkcije za računanje faktorijela

```
public static int factorial(int);
  descriptor: (I)I
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
       0: iload_0
       1: ifne          6
       4: iconst_1
       5: ireturn
       6: iload_0
       7: iload_0
       8: iconst_1
       9: isub
      10: invokestatic  #25  // Method factorial:(I)I
      13: imul
      14: ireturn
```

Listing 3.2: Bajtkod funkcije za računanje faktorijela

Čvorovi grafa većinski predstavljaju operacije (poput aritmetičkih operacija, poziva metoda ili naredbi kontrole toka) ili vrednosti (poput konstanti ili argumenata

[1]Za vizualizaciju grafova korišćen je alat *Ideal Graph Visualizer* (*IGV*) [25].



Slika 3.2: Vizualni prikaz grafa funkcije za računanje faktorijela, dobijen alatom *IGV*

metoda), ali mogu imati i druge, pomoćne uloge (npr. oznaka za ulaznu tačku metoda — *Start* čvor). Da bi se predstavio tok podataka (engl. *data flow*) programa, čvorovi koji odgovaraju operacijama mogu imati grane koje ih povezuju sa čvorovima koji predstavljaju njihove operande. Tako, na primer, čvor *If* koji odgovara *If-Then-Else* naredbi kontrole toka ima jednu ulaznu granu koja ga povezuje sa čvorom koji odgovara vrednosti uslova koji se proverava. Na slici 3.2 ulazne grane su prikazane tanjim linijama, ili uopšte nisu prikazane ako povezuju čvor sa nekom konstantom ili vrednošću argumenta (u kom slučaju su odgovarajući čvorovi samo „nalepljeni" na čvor operacije).

Za predstavljanje kontrole toka (engl. *control flow*) programa, čvorovi mogu imati svoje sledbenike — čvorove sa kojima su povezani granama koje predstavljaju promenu u toku kontrole programa. Čvor *If* tada ima dva sledbenika — jedan predstavlja granu izvršavanja u slučaju da je uslov provere ispunjen, a drugi granu koja odgovara slučaju da uslov nije ispunjen. Na slici 3.2 grane kontrole toka su prikazane podebljanim, crvenim linijama. Čvorovi *Begin* označavaju početak bloka kontrole toka.

Ovakva međureprezentacija omogućava jednostavnu i efikasnu implementaciju različitih optimizacija, poput propagacije konstanti [4]. Pored toga što se koristi kao međureprezentacija u AOT kompilatoru *Native Image*, ova grafovska reprezentacija programa se koristi i u JIT kompilaciji u kompilatoru *Graal*. Ovo omogućava deljenje implementacije kompilatorskih optimizacija u ova dva dela sistema *GraalVM*.

Analiza dostižnosti i pretpostavka zatvorenog sveta

Radi postizanja što boljih preformansi generisanog izvršivog programa, kompilator *Native Image* koristi kombinaciju statičke analize[2] programa koja utvrđuje koji njegovi elementi su dostižni, kao i inicijalizacije određenih objekata i klasa za vreme same kompilacije [41]. Ovaj postupak podrazumeva iterativno izvršavanje narednih koraka:

Analiza dostižnosti — statička analiza, počev od svih mogućih ulaznih tačaka programa[3], određuje dostižne klase, metode i polja programa. Element se smatra dostižnim ukoliko je moguće pristupiti mu za vreme izvršavanja programa. Odgovarajući mašinski kôd će biti generisan samo za one metode koji se smatraju dostižnim.

Inicijalizacija klasa — ukoliko se za statički inicijalizator neke klase može dokazati da nema sporednih efekata, ili ukoliko je tako eksplicitno zahtevano prosleđivanjem dodatnih opcija kompilatoru, njegovo izvršavanje se simulira. Statička polja tako inicijalizovane klase dobijaju svoju inicijalnu vrednost još u fazi kompilacije, čime se dodatno umanjuje problem hladnog starta.

Izgradnja hipa slike — analizom dostižnih objekata, počev od određenih *korenih* objekata (poput dostižnih statičkih polja), vrši se izgradnja grafa dostižnih objekata. Ovi objekti se, po završetku cele procedure, čuvaju u zasebnom segmentu generisane izvršive datoteke koji se naziva `svm_heap`. Oni čine tzv. hip slike (engl. *image heap*). Prednost ovog pristupa je inicijalizacija dela objekata još za vreme kompilacije, kao i mogućnost deljenja hipa slike kroz više različitih instanci programa, gde bi se kopiranje vršilo samo u slučaju modifikacije objekata u hipu slike iz neke od instanci programa (engl. *copy-on-write*).

Da bi ovakva analiza bila moguća, *Native Image* koristi pretpostavku zatvorenog sveta (engl. *closed-world assumption*). Ova pretpostavka nalaže da sav kôd koji može biti izvršen mora biti dostupan u fazi kompilacije programa. Ovo je u suprotnosti sa nekim dinamičkim mogućnostima Jave, poput učitavanja klasa za vreme izvršavanja programa i refleksije.

[2] Analiza koja se izvršava bez pokretanja programa.

[3] Ovo je najčešće `main` metod programa, ali mogu biti i metodi registrovani za refleksiju ili metodi anotirani sa `@CEntryPoint` anotacijom, što označava da metod može biti pozvan u slučaju izgradnje deljene biblioteke (engl. *shared library*) [20].

3.2 Refleksija pod pretpostavkom zatvorenog sveta

Refleksija omogućava učitavanje klasa i njihovu introspekciju za vreme izvršavanja programa. Međutim, jedna posledica pretpostavke zatvorenog sveta je da sve klase kojima se može pristupiti u programu moraju biti dostupne u fazi njegove kompilacije. Pošto refleksija omogućava pristup proizvoljnim elementima, koji potencijalno postaju poznati tek u fazi izvršavanja, statičkom analizom se, u opštem slučaju, ne može odrediti njihova dostižnost [12]. Posledica ovoga je moguće neuspešno izvršavanje reflektivnih operacija koje bi bile uspešne u standardnom okruženju za izvršavanje Java programa.

Listing 3.3 prikazuje Java program koji ispisuje informacije o klasi čije se ime učitava preko argumenata komandne linije. Kako su konkretni argumenti komandne linije poznati tek u fazi izvršavanja, analiza dostižnosti kompilatora *Native Image* ne može da zaključi kojoj klasi se pristupa reflektivnim pozivom `Class.forName`, te ona neće biti uključena u rezultujuću izvršivu datoteku. Ovo za posledicu ima izbacivanje izuzetka `ClassNotFoundException` čak i u slučaju da je tražena klasa zapravo definisana u izvornom kodu. Listing 3.4 prikazuje rezultat izvršavanja ovog programa pri pokretanju sa Java virtuelnom mašinom i pri pokretanju izvršive datoteke koju je generisao kompilator *Native Image*.

U cilju ispravnog funkcionisanja refleksije u kompiliranim Java programima, kompilator *Native Image* koristi dva mehanizma. Prvi mehanizam je zadavanje metapodataka o reflektivnim pristupima u programu, a drugi je statička analiza za automatsko razrešavanje nekih jednostavnijih slučajeva upotrebe refleksije.

Metapodaci za reflektivne pristupe

Jedna mogućnost za prevazilaženje problema nastalih usled pretpostavke zatvorenog sveta je upotreba eksternih datoteka za zadavanje podataka o klasama i njihovim elementima kojima će se reflektivno pristupati u programu. Isti sistem se koristi i u svrhe podrške za ugradnju resursa (teksta, slika, zvuka, ...) u izvršivu datoteku, kao i podrške za kôd, pisan u drugim programskim jezicima, koji pristupa Java objektima kroz Java nativni interfejs (engl. *Java Native Interface*, skraćeno JNI) [31].

Ove konfiguracione datoteke, posebno imenovane sa *reachability-metadata.json*, smeštaju se u zasebne direktorijume koji odgovaraju JAR datotekama koje učestvuju u kompilaciji. One se zadaju u `JSON` formatu sa korenom strukturom prikazanom

```
public class ClassProber {
    public static void main(String[] args) {
        try {
            Class<?> clazz = Class.forName(args[0]);
            printInfo(clazz);
        } catch (ClassNotFoundException e) {
            System.out.println("Class " + args[0] + " could not be found!");
        }
    }

    private static void printInfo(Class<?> clazz) {
        System.out.println(clazz + " information");
        System.out.println("----------------------");
        System.out.println("Decl. fields: " + clazz.getDeclaredFields().length);
        System.out.println("Decl. constructors: " +
            clazz.getDeclaredConstructors().length);
        System.out.println("Decl. methods: " + clazz.getDeclaredMethods().length);
    }
}

class A {
    public static String SOME_FIELD = "some_value";

    public int someMethod(String s, int n) {
        return n + s.length();
    }
}
```

Listing 3.3: Program koji ispisuje osobine proizvoljne klase

```
$ java ClassProber A
class A information
----------------------
Decl. fields: 1
Decl. constructors: 1
Decl. methods: 1

$ ./classprober A
Class A could not be found!
```

Listing 3.4: Rezultati izvršavanja programa *ClassProber* na *HotSpot* virtuelnjoj mašini i slike dobijene sa kompilatorom *Native Image*

u listingu 3.5. Svako od polja predstavlja jednu vrstu metapodataka koja se može zadati kao niz stavki koje je potrebno uključiti u finalnu izvršivu datoteku. U nastavku će pažnja biti posvećena metapodacima za reflektivne pristupe, koji se zadaju u sekciji `"reflection"`.

```
{
    "reflection": [],
    "resources": [],
    "bundles": [],
    "jni": []
}
```

Listing 3.5: Korena struktura datoteka *reachability-metadata.json*

Svaka stavka u metapodacima za refleksiju odgovara nekoj klasi i, uz ime klase, može sadržati i podatke o poljima i metodama te klase kojima se reflektivno pristupa. Registrovanje klase `A`, njenog polja `SOME_FIELD` i metoda `someMethod(String, int)` za refleksiju je prikazano listingom 3.6.

```
"reflection": [
    ...
    {
        "type": "A",
        "fields": [{"name": "SOME_FIELD"}],
        "methods": [
            {
                "name": "someMethod",
                "parameterTypes": ["java.lang.String", "int"]
            }
        ]
    },
    ...
]
```

Listing 3.6: Registrovanje klase i njenih elemenata za refleksiju

Radi jednostavnijeg zapisa, moguća je i grupna registracija svih javnih ili deklarisanih polja i metoda neke klase, ali po cenu potencijalno veće finalne izvršive datoteke. Sa druge strane, postoji i mogućnost uslovne registracije za refleksiju, pri čemu odgovarajuća stavka biva registrovana samo u slučaju da je zadati uslov ispunjen. Jedan primer takvog uslova je uslov dostižnosti tipa koji je ispunjen samo ukoliko je zadata klasa, na osnovu analize dostižnosti, dostižna. Ovime se sprečava registracija i uključivanje klasa i njihovih elemenata u izvršivu datoteku ukoliko ona mesta u kodu u kojima im se reflektivno pristupa nisu dostižna. Primer upotrebe ovih mogućnosti prikazan je listingom 3.7.

```
"reflection": [
    ...
    {
        "condition": {
            "typeReachable": "some.package.SomeClassUsingReflectionOnB"
        },
        "type": "B",
        "allDeclaredFields": true,
        "allPublicFields": true,
        "allDeclaredMethods": true,
        "allPublicMethods": true
    },
    ...
]
```

Listing 3.7: Uslovna registracija za refleksiju

U svrhe jednostavnijeg generisanja potrebnih metapodataka, moguća je upotre-

ba JVM TI agenta za praćenje reflektivnih poziva (engl. *Tracing Agent*) [32] koji delimično automatizuje taj proces. Naime, aplikaciju od koje želimo da izgradimo izvršivu datoteku moguće je prethodno pokrenuti na Java virtuelnoj mašini uz agenta naredbom prikazanom u listingu 3.8. Agent zatim detektuje izvršavanje reflektivnih poziva i njihovih argumenata, na osnovu čega je moguće generisati odgovarajuće metapodatke. Pošto agent detektuje samo one reflektivne pozive koji se zapravo izvrše prilikom datog pokretanja, često je potreban veći broj pokretanja aplikacije radi što veće pokrivenosti koda, a time i reflektivnih poziva koji se mogu izvršiti.

```
$ java -agentlib:native-image-agent=config-output-dir=/path/to/config-dir/ ...
```

Listing 3.8: Pokretanje Java aplikacije uz agenta za generisanje metapodataka

Statička analiza za razrešavanje reflektivnih poziva

Kompilator *Native Image* ima i mogućnost automatskog razrešavanja jednostavnijih slučajeva upotrebe refleksije, bez potrebe za zadavanjem dodatnih metapodataka. Ovakvo razrešavanje je moguće ukoliko su argumenti reflektivnog poziva poznati unapred. *Native Image* će u tom slučaju pokušati sa izvršavanjem datog reflektivnog poziva još u fazi prevođenja, nakon čega se u grafovskoj međureprezentaciji čvorovi koji odgovaraju reflektivnom pozivu i njegovim argumentima zamenjuju čvorom koji predstavlja rezultat izvršavanja tog poziva.

Jedan slučaj gde je ovakvo razrešavanje poziva moguće je ukoliko se reflektivnim pozivima zadaju literali kao argumenti. Listing 3.9 prikazuje primer takvog slučaja.

```
public class MainClass {
    public static void main(String[] args) throws Throwable {
        Class<?> clazz = Class.forName("SomeClass");
        Field f = clazz.getField("SOME_FIELD");
        Method m = clazz.getMethod("someMethod", String.class, int.class);
        System.out.println("Class " + c + " has field " + f + " and method " + m);
    }
}

class SomeClass {
    public static String SOME_FIELD = "some_value";

    public int someMethod(String s, int n) {
        return n + s.length();
    }
}
```

Listing 3.9: Reflektivni pozivi koji se mogu razrešiti u fazi prevođenja

Glava 4

Analiza za razrešavanje reflektivnih poziva

Glavni doprinos ovog rada čini analiza za razrešavanje reflektivnih poziva na nivou Java bajtkoda i njena implementacija u okviru kompilatorske infrastrukture *GraalVM*. Poglavlje 4.1 opisuje probleme nastale usled implementacije analize na nivou *Graal* međureprezentacije, kao i motivaciju za njeno podizanje na nivo Java bajtkoda. Poglavlje 4.2 opisuje bajtkod analizu reflektivnih poziva, a poglavlje 4.3 opisuje njenu implementaciju u okviru kompilatorske infrastrukture *GraalVM*.

4.1 Motivacija za podizanje analize na nivo bajtkoda

U okviru *GraalVM*-a je implementirana statička analiza za pronalaženje i razrešavanje reflektivnih poziva. Međutim, činjenica da je ova analiza implementirana na nivou *Graal* grafova dovodi do dva značajna problema:

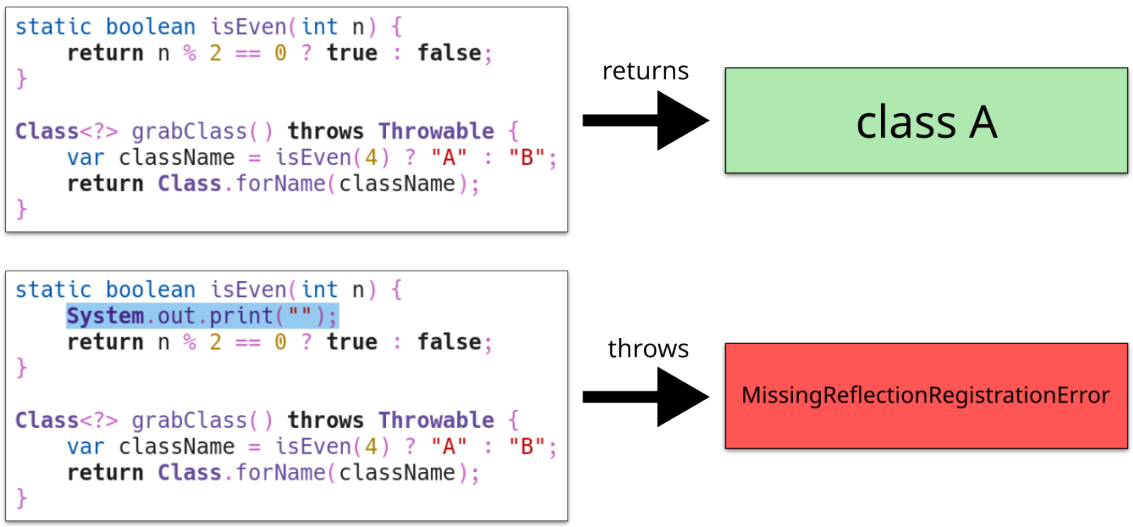
Zavisnost analize od optimizacija — Analiza za razrešavanje reflektivnih poziva zavisi od kompilatorskih optimizacija, što može dovesti do nepredvidivog ponašanja u fazi izvršavanja.

Gubitak informacija — Prevođenje bajtkoda u *Graal* međureprezentaciju dovodi do gubitka informacija, što onemogućava preciznu specifikaciju analize.

Zavisnost analize od optimizacija

Graal grafovi su, još u fazi njihove konstrukcije, podložni agresivnim optimizacijama. Jedna od takvih optimizacija je faza izgradnje grafa koja vrši umetanje (engl. *inlining*) metoda radi poboljšanja preciznosti analize dostižnosti (engl. *reachability analysis*). Ova faza se naziva `InlineBeforeAnalysis`. Pošto se analiza za razrešavanje reflektivnih poziva izvršava u kasnijoj fazi, ona zavisi od rezultata optimizacija koje joj prethode. Posledica ovoga je to da se ponašanje kompiliranog programa pri upotrebi refleksije može promeniti usled manjih, naizgled beznačajnih promena koda, kao i u slučaju uključivanja ili isključivanja određenih optimizacija.

Na slici 4.1 prikazan je metod `grabClass` čije ponašanje pri pokretanju AOT kompiliranog programa zavisi od toga da li je metod `isEven` umetnut. U prvom slučaju, metod `isEven` će biti umetnut u fazi `InlineBeforeAnalysis`, te analiza za razrešavanje reflektivnih poziva može da zaključi vrednost argumenta `Class.forName` poziva, pa time i njegov rezultat. U drugom slučaju, metod `isEven` više ne može biti umetnut usled dodatog poziva metoda `print` i analiza za razrešavanje reflektivnih poziva više ne može da zaključi vrednost argumenta poziva `Class.forName`, pa on prilikom izvršavanja izbacuje grešku `MissingReflectionRegistrationError`.

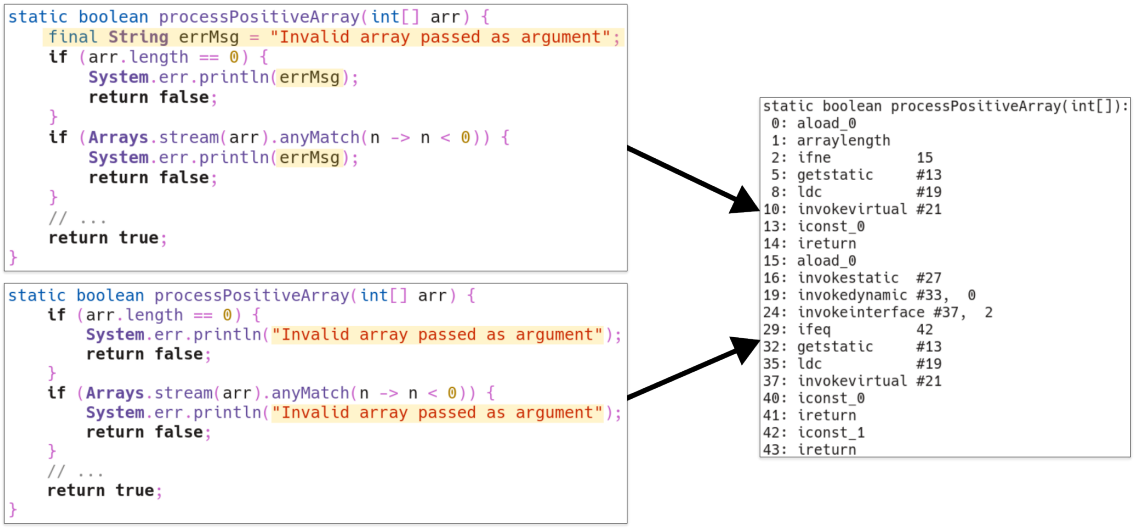


Slika 4.1: Primer programa čije ponašanje zavisi od izvršenih optimizacija

Da bi se izbegla zavisnost analize od izvršenih optimizacija, potrebno je pomeriti njeno izvršavanje u fazu pre izvršavanja bilo kakvih optimizacija. Jedan od načina da se to postigne je izvršavanjem analize direktno nad ulazom u kompilator *Native Image*, tj. nad Java bajtkodom.

Gubitak informacija

Jedna od posledica upotrebe međureprezentacije za analizu programa je mogućnost gubitka informacija sadržanih u izvornom kodu. Naime, prilikom prevođenja Java izvornog koda u Java bajtkod, moguće je dobiti isti rezultujući bajtkod za dva različita izvorna koda. Slika 4.2 prikazuje takav primer, pri čemu se gube informacije o tome da li su kao argumenti `println` metoda korišćeni literali ili promenljiva. Slično važi i za prevođenje Java bajtkoda u *Graal* grafovsku međureprezentaciju.



Slika 4.2: Primer dva metoda koja daju isti rezultujući bajtkod

Pošto analiza za razrešavanje reflektivnih poziva direktno utiče na izvršavanje kompiliranog programa (kao što je opisano u poglavlju 3.2), važno je da ona bude relativno jednostavna za specifikaciju i objašnjenje korisniku. Međutim, gubitak informacija kroz svaki korak prevođenja programa čini specifikaciju u kontekstu Java izvornog koda teškom ili nemogućom. Podizanje analize na nivo Java bajtkoda sa nivoa *Graal* međureprezentacije delimično prevazilazi ovaj problem time što se izbegava jedan nivo gubitka informacija, te omogućava formalnu specifikaciju analize u terminima bajtkoda, kao i jednostavnije objašnjenje analize u terminima Java koda.

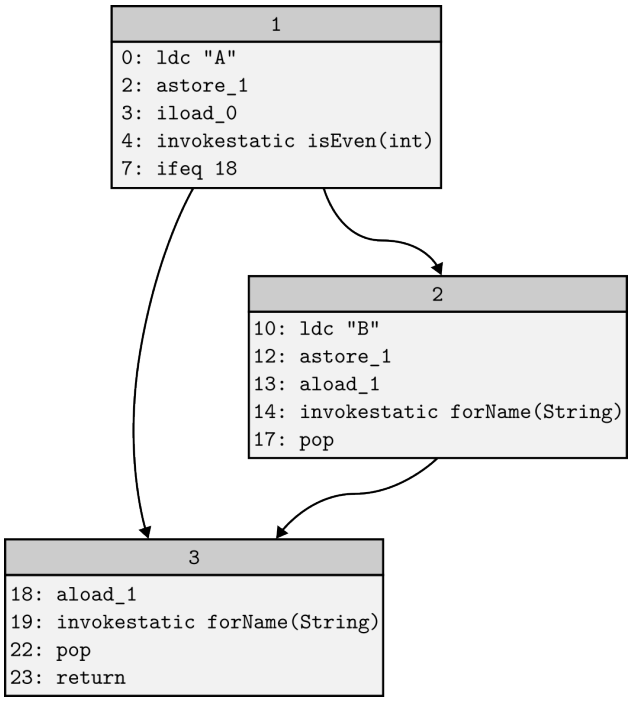
4.2 Algoritam analize

Za automatsko razrešavanje reflektivnih poziva potrebno je odrediti vrednosti argumenata tih poziva u fazi kompilacije. Jedan od načina za određivanje vrednosti

argumenata je upotreba sledećeg pravila — ukoliko se za argumente poziva može utvrditi da su poznati i isti kroz sve putanje izvršavanja metoda, taj poziv se može razrešiti. Pogodna struktura za proveru ovakvog pravila je tzv. graf kontrole toka[1] (engl. *control-flow graph*, skraćeno CFG) bajtkoda metoda koji se analizira. Slika 4.3 prikazuje graf kontrole toka bajtkoda metoda datog u listingu 4.1.

```
public static void tryToLoadClasses(int n) throws Throwable {
    String className = "A";
    if (isEven(n)) {
        className = "B";
        Class.forName(className);
    }
    Class.forName(className);
}
```

Listing 4.1: Jednostavan metod koji, u zavisnosti od uslova, vrši reflektivno učitavanje klasa



Slika 4.3: Graf kontrole toka bajtkoda metoda prikazanog u listingu 4.1

Graf kontrole toka se može konstruisati jednostavnim algoritmom u kojem se

[1]Graf kontrole toka metoda je usmereni graf u kojem čvorove čine osnovni blokovi (engl. *basic block*) metoda, tj. neprekidni nizovi instrukcija sa samo jednom ulaznom i jednom izlaznom tačkom izvršavanja, a veza između dva osnovna bloka A i B postoji ukoliko izvršavanje bloka B može započeti neposredno po završetku izvršavanja bloka A.

bajtkod particioniše na osnovne blokove pronalaženjem tzv. vodećih naredbi. Naredba je vodeća u sledećim slučajevima:

- naredba je prva u nizu instrukcija;
- naredba je meta neke naredbe skoka (uslovnog ili bezuslovnog);
- naredba se nalazi odmah iza neke naredbe skoka u nizu instrukcija.

Svaki osnovni blok onda počinje nekom od vodećih naredbi i sadrži sve naredbe do prve sledeće vodeće. Sledbenici, tj. veze između osnovnih blokova se onda mogu odrediti prema poslednjoj naredbi svakog od osnovnih blokova. Ovaj pristup konstrukciji grafa kontrole toka prikazan je algoritmom 1.

Algoritam 1 Konstrukcija grafa kontrole toka

```
1:  function CONSTRUCT_CFG(List<Instruction> bytecode)
2:      Set<Int> leaders = {0}
3:      for Int i from 1 to bytecode.size() do
4:          if bytecode[i] is a jump instruction then
5:              leaders.add(bytecode[i].jumpTarget())
6:              leaders.add(i + 1)
7:          end if
8:      end for
9:      Map<Int, BasicBlock> cfg = {}
10:     for Int l in leaders do
11:         BasicBlock bb = BasicBlock.create(bytecode[l])
12:         Int i = l + 1
13:         while i < bytecode.size() and i not in leaders do
14:             bb.addInstruction(bytecode[i++])
15:         end while
16:         Instruction last = bb.lastInstruction()
17:         if last is a jump instruction then
18:             bb.addSuccessor(last.jumpTarget())
19:         end if
20:         if last is not an unconditional jump instruction then
21:             bb.addSuccessor(last.index() + 1)
22:         end if
23:         cfg[l] = bb
24:     end for
25:     return cfg
26: end function
```

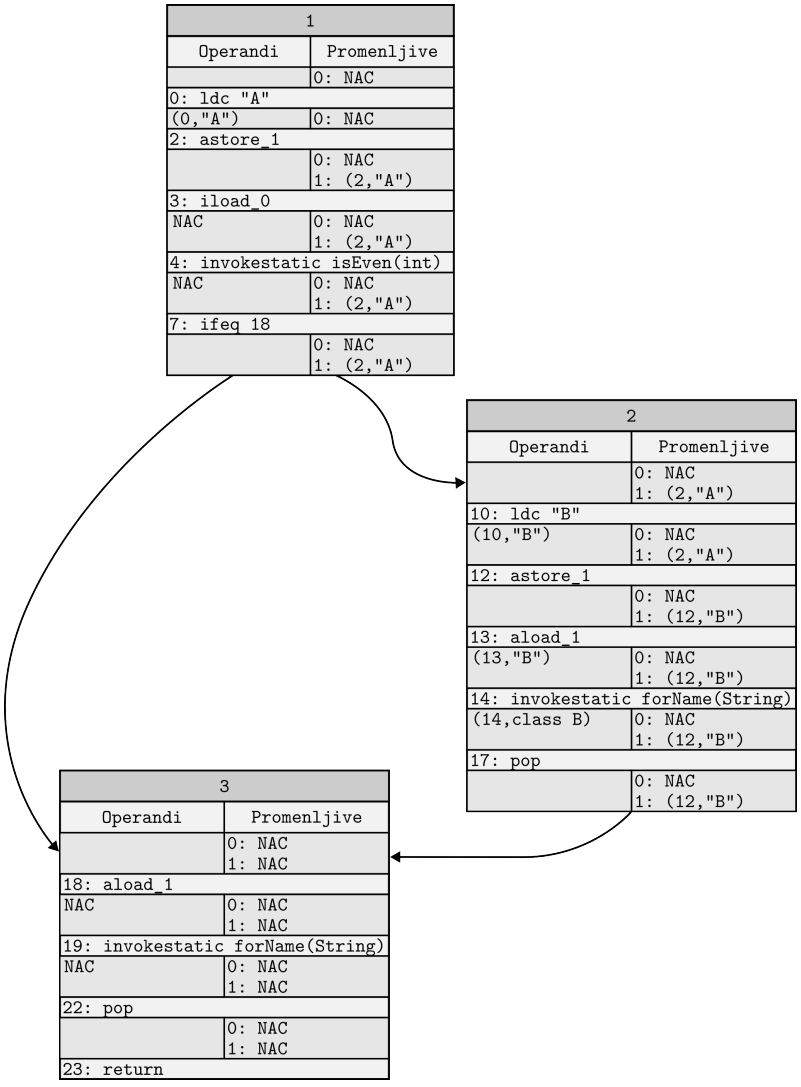
Analiza toka podataka unapred (engl. *forward data flow analysis*), na osnovu grafa kontrole toka i počev od nekog inicijalnog stanja (koje odgovara početku izvršavanja metoda), za svaku instrukciju metoda računa apstraktno stanje programa pre izvršavanja te instrukcije. Ukoliko se do neke instrukcije može doći preko različitih grana izvršavanja, tj. iz različitih osnovnih blokova, stanje koje odgovara takvoj instrukciji se dobija *spajanjem* stanja dobijenih na izlazu tih osnovnih blokova. Računanje stanja je iterativno i izvršava se sve do postizanja fiksne tačke, tj. iteracije u kojoj više nema promene u stanjima. Uopšteni algoritam analize toka podataka unapred dat je algoritmom 2, pri čemu je za konkretnu analizu potrebno definisati apstraktnu reprezentaciju stanja `State`, inicijalno stanje dobijeno funkcijom `initialState`, funkciju prelaska u novo stanje na osnovu instrukcije `processInstruction` i funkciju spajanja stanja `mergeStates`.

Algoritam 2 Analiza toka podataka unapred

```
1:  function FORWARD_DATAFLOW(Map<Int, BasicBlock> cfg)
2:      Map<Int, State> states = {}
3:      states[0] = initialState()
4:      Set<BasicBlock> workList = {cfg[0]}
5:      while not worklist.empty() do
6:          BasicBlock currentBlock = workList.removeFirst()
7:          State currentState = states[currentBlock.firstInstruction().index()]
8:          for Instruction insn in currentBlock do
9:              states[insn.index()] = currentState
10:             currentState = processInstruction(insn, currentState)
11:         end for
12:         for BasicBlock successor in currentBlock.successors() do
13:             Int startOfSuccessor = successor.firstInstruction().index()
14:             successorState = states[startOfSuccessor]
15:             if successorState is null then
16:                 states[startOfSuccessor] = currentState
17:             else
18:                 State mergedState = mergeStates(currentState, successorState)
19:                 if mergedState is not equal to successorState then
20:                     states[startOfSuccessor] = mergedState
21:                     workList.add(successor)
22:                 end if
23:             end if
24:         end for
25:     end while
26:     return states
27: end function
```

Opis elemenata analize za razrešavanje reflektivnih poziva

Pažljivim izborom reprezentacije stanja programa, inicijalnog stanja i funkcija prelaska i spajanja, moguće je definisati analizu koja proverava prethodno pomenuto pravilo da se reflektivni poziv može razrešiti ukoliko su njegovi argumenti poznati i isti kroz sve putanje izvršavanja. U nastavku će ovi elementi biti opisani i ilustrovani kroz sprovođenje analize na metodu iz listinga 4.1, čiji rezultati su prikazani slikom 4.4.



Slika 4.4: Rezultat analize za razrešavanje reflektivnih poziva metoda prikazanog u listingu 4.1

Reprezentacija stanja

Pošto nas zanimaju argumenti metoda, a oni se, tokom izvršavanja, uzimaju sa steka operanada stek okvira, prirodno je da stanje programa u analizi odgovara upravo stek okvirima. Predložena bajtkod analiza za razrešavanje reflektivnih poziva određuje apstraktnu reprezentaciju stek okvira neposredno pre izvršavanja svake instrukcije metoda, pri čemu, umesto pravih vrednosti, ovi apstraktni stek okviri sadrže vrednosti koje mogu biti:

Nezaključive vrednosti — Njihova prava vrednost se ne može zaključiti analizom. Ovim vrednostima je pridružena oznaka `NAC`.

Zaključive vrednosti — Predstavljene su kao par *(BCI, vrednost)*, pri čemu *BCI* odgovara indeksu instrukcije koja je tu vrednost modifikovala i/ili postavila na stek okvir, a *vrednost* predstavlja vrednost koja bi se nalazila na toj poziciji u stek okviru tokom izvršavanja. Ukoliko su svi operandi nekog reflektivnog poziva u njemu pridruženom stek okviru zaključive vrednosti, on se može razrešiti na osnovu njih.

Apstraktni okviri dobijeni analizom su na slici 4.4 prikazani pre i nakon svake instrukcije.

Inicijalno stanje

Inicijalno stanje stek okvira pre izvršavanja prve instrukcije metoda sadrži prazan stek operanada i tabelu lokalnih promenljivih popunjenu samo sa vrednostima koje odgovaraju argumentima metoda. Pošto predložena analiza nije interproceduralna, vrednosti koje odgovaraju argumentima su uvek nezaključive.

Tabela lokalnih promenljivih u inicijalnom stek okviru prikazanom na slici 4.4 sadrži jednu vrednost, koja nije zaključiva, na indeksu 0 u tabeli. Ova vrednost odgovara argumentu `n` metoda 4.1.

Funkcija prelaska

U kontekstu predložene analize, svaka instrukcija kao ulaz prihvata ulazno stanje okvira i, pomoću određene transformacije, generiše izlazno stanje. Ove transformacije menjaju strukturu okvira, tj. broj vrednosti na steku operanada i u tabeli lokalnih promenljivih, u skladu sa specifikacijom instrukcija Java virtuelne mašine [13]. U nastavku su prikazani motivacioni primeri Java koda u kojima bi analiza trebalo da

razreši reflektivne pozive, a uz njih i transformacije, na nivou bajtkoda i u okviru funkcije prelaska, kojima se to i postiže:

Učitavanje konstanti — U osnovnom slučaju, predložena analiza ima mogućnost razrešavanja reflektivnih poziva čiji su argumenti prosleđeni kao literali, ili nešto opštije, kao konstantni izrazi definisani specifikacijom Java programskog jezika [10] (uz dodatak klasnih literala). Naredni isečak Java koda prikazuje nekoliko primera takvih reflektivnih poziva:

```
private static final String CLASS_NAME = "SomeClass";
...
Class.forName(CLASS_NAME);
Class.forName("SomeClass");
SomeClass.class.getField("SomeField");
```

Način na koji se ovo može postići je propagacijom vrednosti koje potiču od instrukcija za učitavanje konstanti, opisanih u poglavlju 2.3, kao zaključivih vrednosti. Tipovi pravih vrednosti koje se mogu učitati ovim instrukcijama su imutabilni (engl. *immutable*) — nakon kreacije, vrednosti imutabilnog tipa se ne mogu promeniti. Ovo, između ostalog, obuhvata primitivne tipove, tip `String`, kao i tip `Class`. Imutabilnost ovih vrednosti olakšava analizu, jer čak i ukoliko neka od njih bude prosleđena u drugi metod, tj. van opsega analize, možemo biti sigurni da ona neće biti izmenjena. U primeru 4.4, prva instrukcija metoda, instrukcija `LDC` na indeksu 0, učitava nisku `"A"` i postavlja je na stek operanada. U apstraktnom okviru analize, ova instrukcija će se simulirati postavljanjem zaključive vrednosti `(0, "A")` na vrh steka.

Pristup poljima — Predložena analiza, u opštem slučaju, ne prati vrednosti polja, te instrukcije za pristup poljima na stek postavljaju nezaključive vrednosti. Međutim, u kontekstu Java izvornog koda, upotreba klasnih literala nad primitivnim tipovima, kao što je slučaj kod literala `int.class`, na nivo bajtkoda se prevodi kao pristup statičkom polju `TYPE` odgovarajućeg klasnog omotača nad primitivnim tipom (u prethodnom slučaju, to bi bilo polje `Integer.TYPE`). Iz ovog razloga, upotreba `GETSTATIC` instrukcije koja referiše na neko od ovih polja specijalno na vrh steka postavlja odgovarajuću zaključivu vrednost.

Propagacija kroz promenljive — Pored poziva čiji su argumenti literali (i drugi konstantni izrazi), predložena analiza pod određenim uslovima (koji će biti opisani uz funkciju spajanja) može razrešiti i reflektivne pozive u kojima su

argumenti prosleđeni u vidu promenljivih. Tako bi, na primer, poziv metoda `forName` bio automatski razrešen u narednom isečku koda:

```
String className = "SomeClass";
Class.forName(className);
```

Iz ovog razloga, zaključive vrednosti koje odgovaraju imutabilnim tipovima se mogu propagirati kroz promenljive. Ovo se postiže time što instrukcije za upravljanje lokalnim promenljivama prenose zaključivu vrednost ukoliko je njihov operand (u slučaju čuvanja promenljive) ili vrednost promenljive na koju referišu (u slučaju njenog učitavanja) zaključiva vrednost. Prilikom prenošenja ovih vrednosti, njihova `BCI` komponenta se menja u skladu sa indeksom instrukcije koja ih prenosi. U primeru 4.4, instrukcija `ASTORE_1` na indeksu 12 prenosi svoj zaključivi operand `(10, "B")` u tabelu lokalnih promenljivih kao zaključivu vrednost `(12, "B")`, a odmah iza nje, instrukcija `ALOAD_1` na indeksu 13 na vrh steka postavlja zaključivu vrednost `(13, "B")`.

Operacije nad nizovima — Određeni deo reflektivnih metoda, poput metoda `Class.getMethod(String name, Class<?>... parameterTypes)`, imaju promenljiv broj argumenata (engl. *variadic arguments*). Na nivou bajtkoda, ovi argumenti se prevode u nizove odgovarajućeg tipa, te je u bajtkodu tip parametra `parameterTypes` prethodnog metoda zapravo `Class[]`. Prema tome, dva poziva metoda `getMethod` u narednom isečku generišu istovetan bajtkod:

```
SomeClass.getMethod("someMethod", String.class, Integer.class);
SomeClass.getMethod("someMethod", new Class<?>[] { String.class,
    Integer.class });
```

Propagacija nizova kao zaključivih vrednosti kroz analizu započinje instrukcijom `ANEWARRAY` — ukoliko je operand ove instrukcije, koji odgovara veličini niza, zaključiva vrednost, onda se na vrh steka postavlja zaključiva vrednost `(anewarray_bci, [null, null, ..., null])` koja sadrži niz date veličine, popunjen sa `null` vrednostima. Dalje ažuriranje vrednosti ovakvog niza se sprovodi `AASTORE` instrukcijama, i to na sledeći način:

- Ukoliko su, pored reference na niz, operandi indeksa i elementa `AASTORE` instrukcije zaključive vrednosti, onda se sve zaključive vrednosti u stek okviru čija `BCI` komponenta odgovara ciljanom nizu transformišu ažuriranjem njihove `BCI` komponente na indeks `AASTORE` instrukcije, kao i

izmenom prave vrednosti niza u skladu sa pravim vrednostima indeksa i elementa `AASTORE` instrukcije.

- U suprotnom, tj. ukoliko jedan od operanda indeksa i operanda elementa nije zaključiva vrednost, sve zaključive vrednosti u stek okviru čija `BCI` komponenta odgovara ciljanom nizu postaju nezaključive.

Određivanje vrednosti nizova uvodi komplikacije u predloženoj analizi, jer, za razliku od prethodno pomenutih imutablinih vrednosti, vrednost niza se uvek može izmeniti, što može dovesti do izmene njihovog sadržaja van okvira analize. Iz ovog razloga, analiza uvodi određena ograničenja prilikom zaključivanja vrednosti nizova. Naime, u slučaju da je zaključiva vrednost koja odgovara nizu operand neke instrukcije za poziv metoda (`INVOKEVIRTUAL`, `INVOKEINTERFACE`, `INVOKESPECIAL`, `INVOKESTATIC`, `INVOKEDYNAMIC`), instrukcije za izmenu vrednosti polja (`PUTFIELD`, `PUTSTATIC`), `AASTORE` instrukcije u ulozi elementa koji se čuva u nizu, ili neke od instrukcija za čuvanje u lokalnoj promenljivoj, sve zaključive vrednosti u stek okviru čija `BCI` komponenta odgovara datom nizu prestaju da budu zaključive. Ova ograničenja su potrebna da bi se osiguralo da se nizu označenom kao zaključiva vrednost neće pristupati van metoda koji se analizira.

Propagacija kroz pozive metoda — Ne toliko redak slučaj je da se, prilikom reflektivnog pristupa nekom polju ili metodu klase, i samoj klasi pristupa reflektivno. Naredni isečak koda prikazuje ovakav slučaj:

```
Class.forName("SomeClass").getField("SomeField");
```

Za pokrivanje ovakvog slučaja upotrebe refleksije, analiza mora propagirati i rezultate izvršavanja određenih metoda kao zaključive vrednosti. Ovo se može postići ukoliko su svi argumenti tih metoda zaključive vrednosti. Primer ovoga je metod `forName(String className)` — u primeru 4.4, instrukcija `INVOKESTATIC` na indeksu 14 kao operand uzima zaključivu vrednost `(13, "B")`, a kao rezultat na vrh steka postavlja zaključivu vrednost `(14, class B)`.

Konverzija tipa nad `null` vrednostima — Radi izbegavanja alokacije dodatnog niza, šablon upotrebe reflektivnih poziva koji se nekada pojavljuje podrazumeva prosleđivanje `null` literala (konvertovanog u tip `Class[]`) umesto prazne liste varijadičkih argumenata:

```
SomeClass.class.getConstructor((Class[]) null);
```

Zbog ovoga, u specijalnom slučaju da je operand `CHECKCAST` instrukcije zaključiva vrednost `null`, ona će se dalje propagirati kao zaključiva.

Ostale instrukcije — Instrukcije koje nisu obrađene u prethodnim slučajevima, a koje kao rezultat izvršavanja postavljaju neku vrednost na stek operanada ili u tabelu lokalnih promenljivih, to rade postavljanjem nezaključivih vrednosti. Iako bi u nekim slučajevima bilo moguće dalje propagirati zaključive vrednosti, npr. u slučaju `IADD` instrukcije sa zaključivim operandima, ovo nije urađeno radi što jednostavnijeg opisa analize na nivou Java koda.

Funkcija spajanja

U slučaju da neka instrukcija ima dva ili više prethodnika u grafu kontrole toka, njeno ulazno stanje se dobija *spajanjem* izlaznih stanja svih njenih prethodnika. Ovo se postiže spajanjem vrednosti na istim pozicijama u steku operanada[2] i tabele lokalnih promenljivih sa sledećim pravilom: Ukoliko su dve vrednosti koje se spajaju zaključive i jednake, rezultujuća vrednost je takođe zaključiva i nepromenjena je u odnosu na vrednosti koje se spajaju. U suprotnom, rezultujuća vrednost nije zaključiva.

U kontekstu Java koda i propagiranja zaključivih vrednosti kroz promenljive, ovo dovodi do narednog pravila — referisane promenljive nad čijom upotrebom dominira dodela zaključive vrednosti toj promenljivoj su zaključive vrednosti. Dodela dominira nad upotrebom promenljive ukoliko sve putanje izvršavanja programa prolaze kroz tu dodelu, bez prolaženja kroz drugu operaciju dodele nad tom promenljivom pre njene upotrebe. Ovo pravilo je ilustrovano narednim isečkom koda:

```
String className = "A";
if (someCondition()) {
    className = "B";
    Class.forName(className); // Uspesno zakljucivanje - rezultat je Class B
}
Class.forName(className); // Neuspesno zakljucivanje
```

U primeru 4.4, instrukcija `ALOAD_1` na indeksu 18 ima dva prethodnika. Stek operanada je u oba slučaja prazan, ali pri spajanju tabela lokalnih promenljivih,

[2]Specifikacija JVM [13] garantuje da stek operanada u jednoj tački programa uvek ima jednak broj elemenata, nezavisno od toga kojom putanjom izvršavanja se do te tačke i stiglo.

konkretno promenljivih na indeksu 1, dolazi do neslaganja u njihovim vrednostima — vrednost promenljive po izlasku iz osnovnog bloka 1 je `(2, "A")`, a po izlasku iz osnovnog bloka 2 je `(12, "B")`, te rezultujuća vrednost nije zaključiva.

Jedno dodatno ograničenje nad propagacijom nizova kao zaključivih vrednosti je da se, ukoliko je spajanje neuspešno (rezultuje vrednošću koja nije zaključiva), a u njemu je učestvovao neki niz koji je zaključiva vrednost, onda se sve zaključive vrednosti čija `BCI` komponenta odgovara tom nizu u rezultujućem stek okviru označavaju kao nezaključive. Ovo ograničenje je potrebno jer, po neuspešnom spajanju niza, nije više moguće pratiti da li je njegova referenca prosleđena u drugi metod.

4.3 Opis implementacije

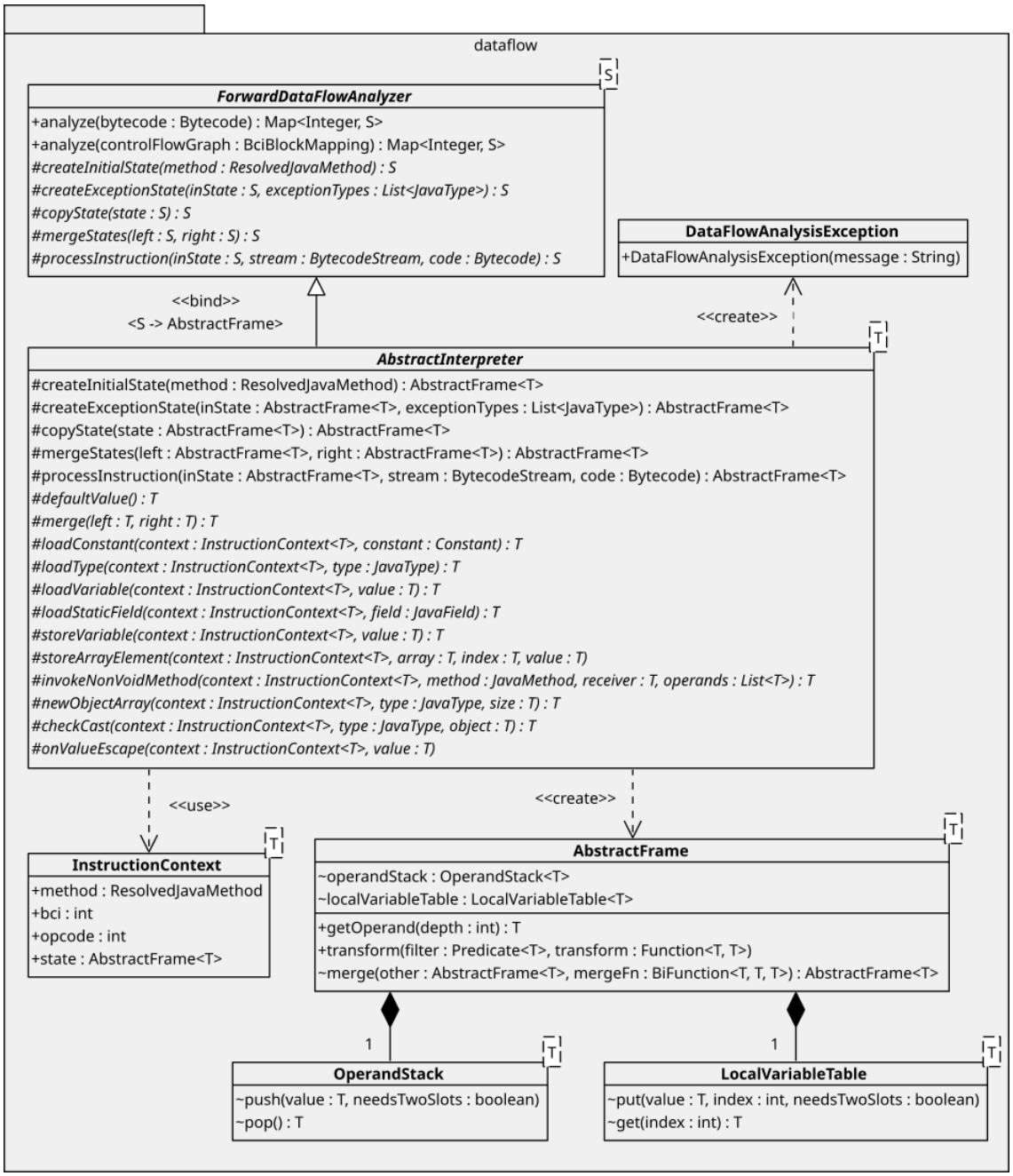
Predložena analiza za razrešavanje reflektivnih poziva je implementirana u okviru kompilatora *Native Image* sistema *GraalVM* i može se pronaći na narednoj lokaciji: `https://github.com/oracle/graal/pull/11079`. Implementacija se grubo može podeliti na dva glavna dela — generički okvir za analizu toka podataka bajtkod programa i implementaciju analize za razrešavanje reflektivnih poziva koja koristi pomenuti okvir, uz propratne klase.

Radni okvir za analizu bajtkoda

Radni okvir za analizu bajtkoda olakšava implementaciju proizvoljne analize toka podataka unapred. Za pristup bajtkodu metoda koristi se Java kompilatorski interfejs, što omogućava laku integraciju sa ostatkom sistema *GraalVM*. Struktura radnog okvira je prikazana klasnim dijagramom 4.5, a u nastavku sledi opis njegovih glavnih klasa:

`ForwardDataFlowAnalyzer<S>` — Apstraktna i generička klasa koja implementira osnovni algoritam analize toka podataka unapred. Generički tip `S` predstavlja tip apstraktnog stanja programa koje će se propagirati kroz analizu.

Javni metod `analyze(Bytecode)` kao argument prihvata bajtkod metoda za analizu, a kao rezultat izvršavanja vraća preslikavanje iz `BCI` indeksa instrukcija metoda u izračunata apstraktna stanja programa pre izvršavanja date instrukcije. Ovaj metod interno poziva naredne apstraktne metode koje je potrebno implementirati za neku konkretnu analizu:



Slika 4.5: UML klasni dijagram radnog okvira za analizu bajtkoda

- `createInitialState` — kreira početno stanje analize;
- `createExceptionState` — kreira stanje analize u koje odgovara ulasku u blok za obradu izuzetaka (`catch` blok);
- `copyState` — pravi duboku kopiju prosleđenog stanja;
- `mergeStates` — vraća rezultat spajanja dva stanja;

- `processInstruction` — implementira funkciju prelaska analize prema instrukciji koju analiza trenutno obrađuje (sadržanu u argumentu `stream`).

`AbstractFrame<T>` — Generička reprezentacija stek okvira, pri čemu generički tip `T` označava tip vrednosti koje će se čuvati u okviru. Objekti ove klase sadrže po instancu ugnježdenih klasa `OperandStack<T>` i `LocalVariableTable<T>` koje predstavljaju, redom, stek operanada i tabelu lokalnih promenljivih.

`AbstractInterpreter<T>` — Apstraktna i generička klasa koja nasleđuje klasu `ForwardDataFlowAnalyzer<AbstractFrame<T>>`, tj. implementira algoritam analize toka podataka unapred, pri čemu apstraktno stanje programa odgovara stek okvirima. Isto kao i u slučaju klase `AbstractFrame<T>`, generički tip `T` označava tip vrednosti koje će se čuvati u stek okvirima izračunatim od strane analize.

Svrha ove klase je uprošćavanje implementaciju bilo kakve analize toka podataka koja koristi stek okvire kao apstraktno stanje programa. Ovo se postiže time što je cela logika za manipulaciju i spajanje stek okvira već implementirana. Za implementaciju konkretne analize je onda potrebno samo definisati reprezentaciju vrednosti `T`, način na koji određene instrukcije utiču na te vrednosti, kao i pravilo spajanja vrednosti pri spajanju stek okvira. Jedan deo apstraktnih metoda koje je potrebno definisati za neku konkretnu analizu su:

- `defaultValue` — vraća podrazumevanu vrednost u analizi — obično označava vrednost o kojoj analiza ne može da donosi zaključke;
- `merge` — definiše proceduru za spajanje dve vrednosti;
- `loadConstant` — vraća vrednost koja odgovara učitanoj konstanti;
- `loadVariable` — vraća vrednost koja će, na osnovu ciljne vrednosti u tabeli lokalnih promenljivih, biti postavljena na stek operanada;
- `storeVariable` — vraća vrednost koja će, na osnovu operanda sa steka, biti postavljena na određenu poziciju u tabeli lokalnih promenljivih.

Postojeće apstraktne metode su dovoljne za implementaciju predložene analize za razrešavanje reflektivnih poziva, pri čemu je klasa `AbstractInterpreter` lako proširiva da podrži i druge analize, ukoliko to u budućnosti bude bilo potrebno.

Implementacija analize za razrešavanje reflektivnih poziva

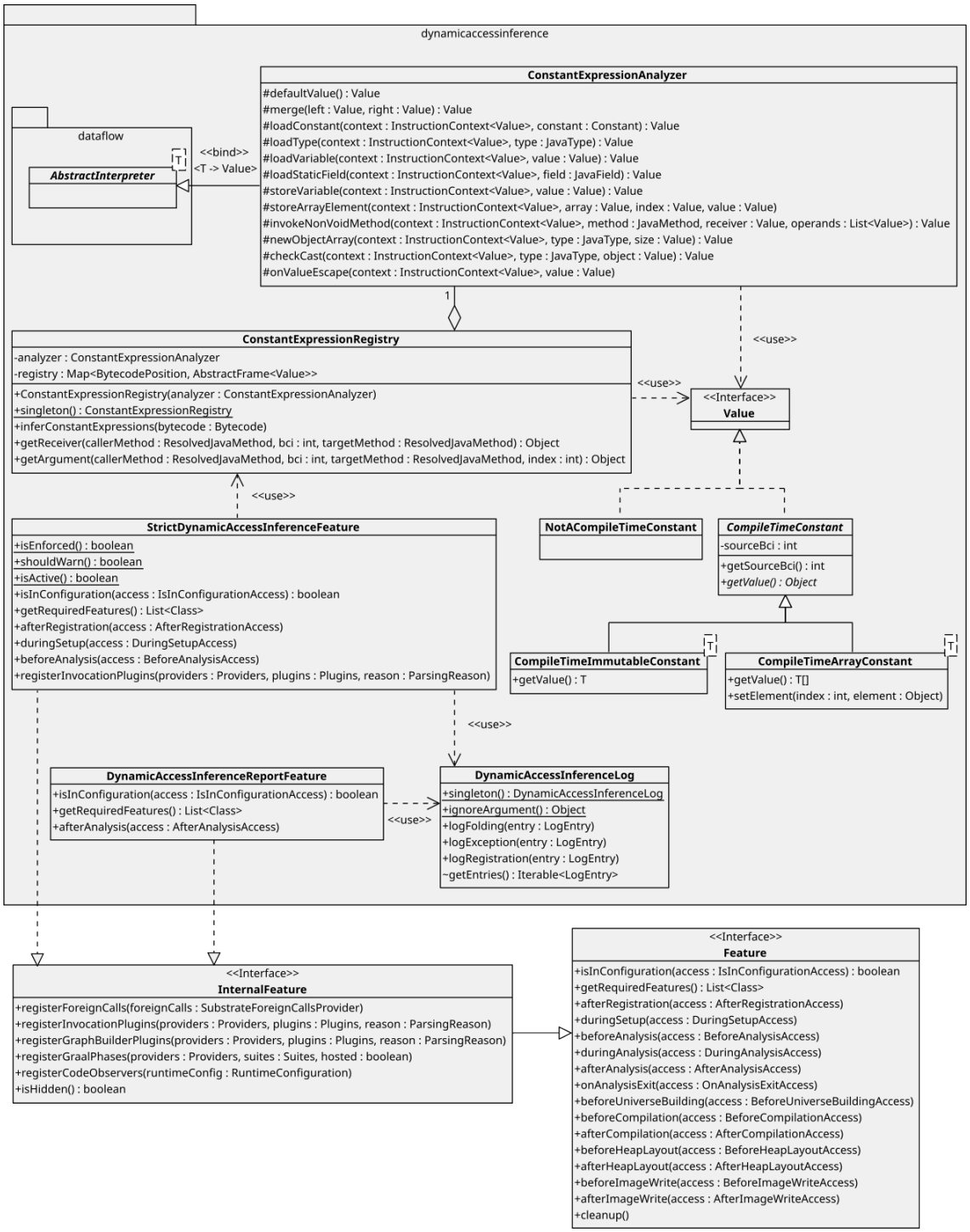
Implementacija predložene analize koristi prethodno opisani okvir za analizu bajtkoda. Struktura implementacije je prikazana klasnim dijagramom 4.6, a u nastavku sledi opis glavnih klasa:

`ConstantExpressionAnalyzer` — Klasa koja, pomoću okvira za analizu bajtkoda, implementira analizu za zaključive vrednosti prema ranije opisanim pravilima. U okviru nje, kao ugnježdene klase, su definisani i tipovi vrednosti koji se propagiraju kroz stek okvire tokom analize, i to kao:

- `Value` — interfejs koji obuhvata sve vrednosti u analizi;
- `NotACompileTimeConstant` — klasa koja implementira interfejs `Value` i predstavlja vrednosti koje nisu zaključive;
- `CompileTimeConstant` — apstraktna klasa koja implementira interfejs `Value` i predstavlja proizvoljnu zaključivu vrednost kao par *(BCI, vrednost)* — njen jedini apstraktni metod je metod `getValue()` koji vraća pravu vrednost ove zaključive vrednosti;
- `CompileTimeImmutableConstant<T>` — klasa koja predstavlja imutabilnu zaključivu vrednost tipa `T` (nasleđuje `CompileTimeConstant`);
- `CompileTimeArrayConstant<T>` — klasa koja predstavlja zaključivu vrednost tipa `T[]` (nasleđuje `CompileTimeConstant`).

`ConstantExpressionRegistry` — Klasa unikat (engl. *singleton*) koja skladišti rezultate dobijene pomoću `ConstantExpressionAnalyzer` analize za dalju upotrebu. Po početku parsiranja bajtkoda metoda zarad generisanja njegove grafovske međureprezentacije, poziva se metod `inferConstantExpressions` ove klase koji pokreće `ConstantExpressionAnalyzer` analizu nad prosleđenim bajtkodom i čuva njene rezultate. Rezultati se čuvaju u mapi koja preslikava poziciju u bajtkodu (metod i `BCI` indeks) u apstraktno stanje programa (`AbstractFrame`) pre izvršavanja instrukcije na toj poziciji.

`StrictDynamicAccessInferenceFeature` — Klasa koja kontroliše izvršavanje predložene analize za razrešavanje reflektivnih poziva. U svrhe toga, ona implementira interfejs `InternalFeature` koji omogućava upravljanje procesom izgradnje izvršive datoteke.



Slika 4.6: Uprošćeni UML klasni dijagram implementacije analize za razrešavanje reflektivnih poziva

`Feature` i `InternalFeature` interfejsi deklarišu skup metoda koji se pozivaju u različitim fazama izgradnje izvršive datoteke. Jedan od njih je metod

`registerInvocationPlugins` iz kojeg se mogu registrovati procedure za modifikaciju procesa generisanja poziva metoda u grafovskoj međureprezentaciji. Upravo ovo omogućava razrešavanje reflektivnih poziva na osnovu rezultata analize. Naime, za reflektivne metode se mogu registrovati procedure koje će, ukoliko `ConstantExpressionRegistry` skladište sadrži zaključive vrednosti koje odgovaraju argumentima njihovih poziva, simulirati taj reflektivni poziv i, umesto generisanja čvora za poziv metoda u grafovskoj međureprezentaciji, generisati čvor koji predstavlja rezultat njegovog izvršavanja.

`DynamicAccessInferenceLog` — Klasa unikat koja čuva podatke o lokacijama (metodu i `BCI` indeksu) razrešenih reflektivnih poziva.

`DynamicAccessInferenceReportFeature` — Klasa koja omogućava generisanje izveštaja o razrešenim reflektivnim pozivima. Izveštaj se generiše u `JSON` formatu.

Upotreba analize za razrešavanje reflektivnih poziva

Postoje tri režima rada za upravljanje bajtkod analizom za razrešavanje reflektivnih poziva:

Disable — Analiza na nivou bajtkoda je isključena i umesto nje se koristi optimizaciono-zavisna analiza na nivou grafovske međureprezentacije. Ovo je trenutno podrazumevani režim rada analize.

Warn — Za razrešavanje reflektivnih poziva se koristi analiza na nivou grafovske međureprezentacije, ali se, prilikom izgradnje izvršive datoteke, na standardnom izlazu ispisuju upozorenja o reflektivnim pozivima koji su razrešeni van pravila zadatih na nivou bajtkoda.

Enforce — Za razrešavanje reflektivnih poziva se koristi implementirana analiza na nivou bajtkoda.

Izbor režima rada prilikom razrešavanja reflektivnih poziva se podešava opcijom `StrictDynamicInference`. Listing 4.2 ilustruje pokretanje *Native Image* kompilatora u *Enforce* režimu rada analize na nivou batjkoda.

Dodatno, implementirana je i mogućnost generisanja izveštaja o razrešenim reflektivnim pozivima (u oba režima analize) pomoću istinitosne (engl. *boolean*) opcije izgradnje `ReportDynamicAccessInference`. Uključivanjem ove opcije, u toku

```
native-image -H:StrictDynamicAccessInference=Enforce SomeJavaApp
```

Listing 4.2: Naredba za pokretanje *Native Image* kompilatora uz bajtkod analizu za razrešavanje reflektivnih poziva

izgradnje izvršive datoteke generisaće se `JSON` izveštaj o lokacijama i argumentima reflektivnih poziva koji su razrešeni.

Glava 5

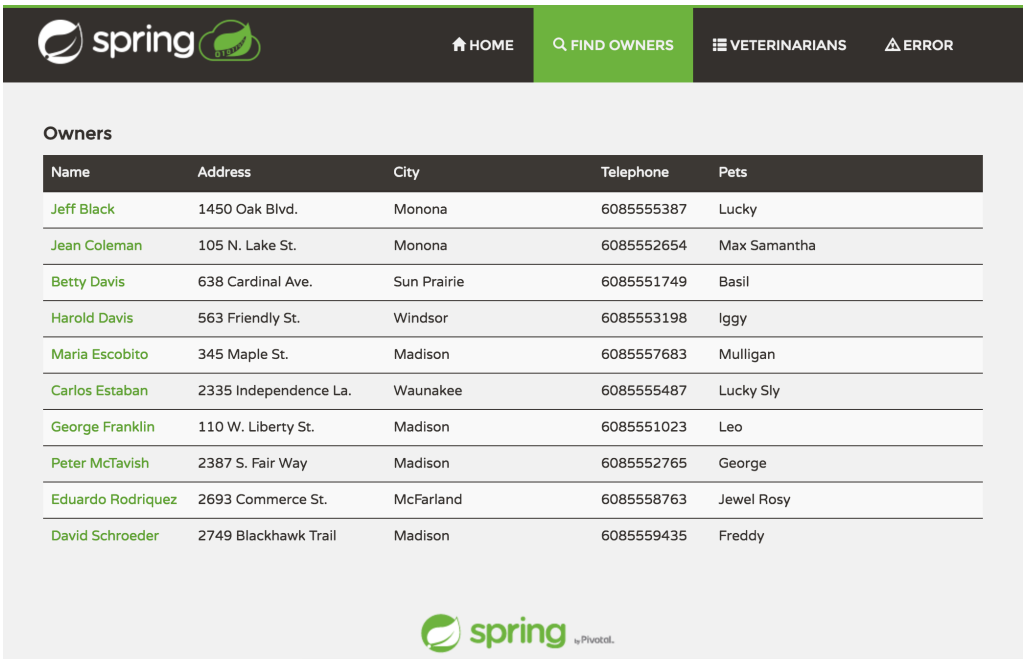
Evaluacija rešenja

Analiza za razrešavanje reflektivnih poziva na nivou bajtkoda je nezavisna od kompilatorskih optimizacija, što uključuje i optimizaciju umetanja metoda. Kako analiza na nivou bajtkoda nije interproceduralna, od nje se ne može očekivati jednaka ili veća mogućnost razrešavanja u odnosu na analizu na nivou *Graal* grafova, koja se, usled umetanja metoda, može smatrati interproceduralnom. Uprkos tome, na primeru jedne *Spring* [38] aplikacije se pokazuje da je gubitak u broju razrešenih reflektivnih poziva relativno mali. Poglavlje 5.1 ukratko opisuje referentni program koji je korišćen za evaluaciju rešenja, a poglavlje 5.2 predstavlja rezultate evaluacije. Poglavlje 5.3 opisuje upotrebu režima analize `Warn` u svrhu lakšeg prihvatanja analize na nivou bajtkoda od strane korisnika sistema *GraalVM*.

5.1 Referentni program *Spring PetClinic*

Spring [38] je Java radni okvir otvorenog koda koji svoju primarnu upotrebu pronalazi u izradi veb aplikacija. Radi omogućavanja jednostavne konfiguracije aplikacija, *Spring* u svojoj implementaciji intenzivno koristi Javine mogućnosti refleksije [39]. Ovo čini *Spring* aplikacije posebno interesantnim u kontekstu kompatibilnosti sa kompilatorom *Native Image* i ispitivanja mogućnosti automatskog razrešavanja reflektivnih poziva.

Spring PetClinic [40] je veb aplikacija koja demonstrira mogućnosti radnog okvira *Spring*. Ona predstavlja aplikaciju za podršku jednoj veterinarskoj ambulanti, uz mogućnosti vođenja evidencije o ljubimcima, njihovim vlasnicima i veterinarima. Slika 5.1 prikazuje izgled aplikacije iz veb pregledača. Evaluacija analize za razrešavanje reflektivnih poziva je sprovedena upravo nad ovom aplikacijom.



Slika 5.1: Izgled veb aplikacije *Spring PetClinic*

5.2 Rezultati evaluacije

Evaluacija predložene analize nad aplikacijom *Spring PetClinic* je sprovedena poređenjem generisanih izveštaja o razrešenim reflektivnim pozivima dobijenih analizom na nivou bajtkoda (opcija `-H:StrictDynamicInference=Enforce`) i analizom na nivou *Graal* grafova (opcija `-H:StrictDynamicInference=Disable`). Svaka stavka `JSON` izveštaja sadrži i niz umetnutih metoda pri izgradnji grafa, što je neophodno za dobijanje pravog broja razrešenih poziva u slučaju analize na nivou grafova. Primer jedne stavke izveštaja prikazan je listingom 5.1.

```
{
    "inliningContext": [
        "org.apache.tomcat.util.compat.Jre19Compat.<clinit>(Jre19Compat.java:37)"
    ],
    "targetMethod": "java.lang.Class.forName(String)",
    "targetArguments": [
        "java.lang.WrongThreadException"
    ],
    "constantValue": "class java.lang.WrongThreadException"
}
```

Listing 5.1: Stavka izveštaja generisanog sa opcijom `-H:+ReportDynamicAccessInference`

Tabela 5.1 prikazuje odnos razrešenih reflektivnih poziva u slučaju upotrebe

analize nad *Graal* grafovima i predložene analize na nivou bajtkoda. Analiza bajtkoda razrešava oko 5% manje reflektivnih poziva nego analiza grafova, ali zato ima osobinu da je nezavisna od kompilatorskih optimizacija, što obezbeđuje predvidivo ponašanje programa.

Metod	**Graf #**	**Bajtkod #**
`java.lang.Class`		
`forName(String)`	77	76
`forName(String, boolean, ClassLoader)`	18	12
`getField(String)`	0	0
`getDeclaredField(String)`	16	16
`getConstructor(Class[])`	5	5
`getDeclaredConstructor(Class[])`	5	5
`getMethod(String, Class[])`	40	37
`getDeclaredMethod(String, Class[])`	20	20
`getFields()`	2	2
`getDeclaredFields()`	6	6
`getConstructors()`	0	0
`getDeclaredConstructors()`	0	0
`getMethods()`	1	1
`getDeclaredMethods()`	6	6
`getClasses()`	0	0
`getDeclaredClasses()`	0	0
`getNestMembers()`	0	0
`getPermittedSubclasses()`	0	0
`getRecordComponents()`	0	0
`getSigners()`	0	0
`java.lang.invoke.MethodHandles.Lookup`		
`findClass(String)`	0	0
`findVirtual(Class, String, MethodType)`	2	2
`findStatic(Class, String, MethodType)`	6	6
`findConstructor(Class, MethodType)`	0	0
`findGetter(Class, String, Class)`	0	0
`findStaticGetter(Class, String, Class)`	0	0
`findSetter(Class, String, Class)`	0	0
`findStaticSetter(Class, String, Class)`	0	0
`findVarHandle(Class, String, Class)`	18	17
`findStaticVarHandle(Class, String, Class)`	0	0
Ukupno	**222**	**211**

Tabela 5.1: Broj razrešenih reflektivnih poziva tokom kompilacije aplikacije *Spring PetClinic* — brojevi su redom prikazani za analizu nad *Graal* grafovima i za analizu nad bajtkodom

5.3 Migracija na novi režim razrešavanja reflektivnih poziva

Kako analiza za razrešavanje reflektivnih poziva utiče na ponašanje generisane izvršive datoteke, neophodno je obezbediti proces za migraciju korisnika sistema *GraalVM* na novi režim analize. U ovu svrhu je pružen režim `Warn` opcije `StrictDynamicAccessInference` u kojem se, prilikom izgradnje izvršive datoteke, na standardni izlaz ispisuju upozorenja o reflektivnim pozivima koji su razrešeni van pravila zadatih na nivou bajtkoda. Korisnici tada, na osnovu dobijenih upozorenja, mogu registrovati odgovarajuće elemente za refleksiju pomoću konfiguracionih datoteka. Listing 5.2 prikazuje ispisana upozorenja pri kompilaciji jednog programa sa opcijom `-H:StrictDynamicAccessInference=Warn`.

```
$ native-image -H:StrictDynamicAccessInference=Warn Test
...
Warning: The following dynamic access method invocations have been inferred
    outside of the strict inference mode:
1. Call to java.lang.Class.getMethod(String, Class[]) reachable in
    Test.cfgTest(Test.java:24) with receiver class A and arguments (otherMethod,
    null) was inferred as the constant public static int A.otherMethod()
2. Call to java.lang.Class.getField(String) reachable in
    Test.cfgTest(Test.java:17) with receiver class A and arguments (SOME_FIELD)
    was inferred as the constant public static java.lang.String A.SOME_FIELD
3. Call to java.lang.Class.forName(String, boolean, ClassLoader) reachable in
    Test.cfgTest(Test.java:14) with arguments (A, false, <ignored>) was inferred
    as the constant class A
...
```

Listing 5.2: Primer upozorenja dobijenih pomoću opcije `-H:StrictDynamicAccessInference=Warn`

Glava 6

Zaključak

U okviru master rada opisan je standardni način izvršavanja Java programa — posredstvom Java virtuelne mašine i JIT kompilacije. Posebna pažnja je posvećena opisu strukture Java bajtkoda, međureprezentacije programa za izvršavanje na Java virtuelnoj mašini, kao i tehnikama za njenu analizu. Takođe, detaljno je opisan i sistem *GraalVM* i njegove mogućnosti AOT kompilacije Java programa, kao i ograničenja u dinamičkim odlikama (posebno refleksije) programskog jezika Java koja se pritom uvode.

Glavni doprinos ovog rada je implementacija radnog okvira za analizu Java bajtkoda u okviru sistema *GraalVM*, kao i analize za automatsko razrešavanje reflektivnih poziva. Ovako implementirana analiza, uz minimalan gubitak u mogućnostima razrešavanja u odnosu na postojeće rešenje zasnovano na analizi *Graal* grafova, dobija veoma važnu osobinu nezavisnosti od kompilatorskih optimizacija. Kako razrešavanje reflektivnih poziva, usled ograničenja u refleksiji nametnutih pretpostavkom zatvorenog sveta, utiče na ponašanje generisane izvršive datoteke, nezavisnost analize od kompilatorskih optimizacija dovodi do predvidljivijeg ponašanja tokom izvršavanja programa.

Mogućnosti implementirane analize na nivou bajtkoda evaluirane su nad referentnim programom *Spring PetClinic*, veb aplikacijom koja koristi radni okvir *Spring*, a samim tim i Javine mogućnosti refleksije. U odnosu na analizu *Graal* grafova, izgubljeno je samo 11 od 222 razrešena poziva, tj. 5% od svih razrešenih reflektivnih poziva.

U kontekstu daljeg razvoja, jedan od pravaca koji se može posmatrati je proširivanje mogućnosti analize uz pomoć korisnički zadatih anotacija. Naime, kako je implementirana analiza intraproceduralna, korisnik bi, kroz anotiranje metoda, u

analizu mogao da uključi i metode koji u svojoj implementaciji koriste reflektivne pozive sa vrednostima koje zavise od argumenata tog metoda.

Bibliografija

[1] Fernando Pires Barbosa and Andrea Schwertner Charão. Impact of pay-as-you-go cloud platforms on software pricing and development: A review and case study. In Beniamino Murgante, Osvaldo Gervasi, Sanjay Misra, Nadia Nedjah, Ana Maria A. C. Rocha, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2012*, pages 404–417, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Eric Bruneton. *ASM 4.0, A Java bytecode engineering library.* on-line at: `https://asm.ow2.io/asm4-guide.pdf`.

[3] Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Andrew Sutton, and Daveed Vandevoorde. Reflection for C++26. on-line at: `https://isocpp.org/files/papers/P2996R4.html`.

[4] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, March 1995.

[5] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Moessenboeck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[6] FasterXML. The Jackson Project. on-line at: `https://github.com/FasterXML/jackson`.

[7] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series).* Manning Publications Co., USA, 2004.

[8] Commonhaus Foundation. Hibernate. on-line at: `https://hibernate.org/`.

[9] Eclipse Foundation. Eclipse OpenJ9. on-line at: `https://eclipse.dev/openj9/`.

[10] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java Language Specification, Java SE 24 Edition. on-line at: `https://docs.oracle.com/javase/specs/jls/se24/jls24.pdf`.

[11] Shadi Ibrahim, Bingsheng He, and Hai Jin. Towards pay-as-you-consume cloud computing. In *2011 IEEE International Conference on Services Computing*, pages 370–377, 2011.

[12] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518, May 2017.

[13] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java Virtual Machine Specification, Java SE 24 Edition. on-line at: `https://docs.oracle.com/javase/specs/jvms/se24/jvms24.pdf`.

[14] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your jvm: understand and eliminate jvm warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 383–400, USA, 2016. USENIX Association.

[15] Jacques Malenfant, Muyenzi Jacques, and F. Demers. A tutorial on behavioral reflection and its implementation. *Proceedings of the Reflection*, 96:1–20, 01 1996.

[16] Sun Microsystems. The Java HotSpot Performance Engine Architecture. on-line at: `https://www.oracle.com/java/technologies/whitepaper.html`.

[17] OpenJDK. HotSpot Compilation Policy. on-line at: `https://github.com/openjdk/jdk/blob/master/src/hotspot/share/compiler/compilationPolicy.hpp`.

[18] OpenJDK. JEP 243: Java-Level JVM Compiler Interface. on-line at: `https://openjdk.org/jeps/243`.

[19] OpenJDK. Project Leyden. on-line at: `https://openjdk.org/projects/leyden/`.

[20] Oracle. Build a Native Shared Library. on-line at: https://www.graalvm.org/latest/reference-manual/native-image/guides/build-native-shared-library/.

[21] Oracle. Graal Compiler. on-line at: https://www.graalvm.org/latest/reference-manual/java/compiler/.

[22] Oracle. GraalVM. on-line at: https://www.graalvm.org/.

[23] Oracle. GraalVM as a Java Virtual Machine. on-line at: https://www.graalvm.org/latest/reference-manual/java/.

[24] Oracle. GraalVM GitHub Repository. on-line at: https://github.com/oracle/graal.

[25] Oracle. Ideal Graph Visualizer. on-line at: https://www.graalvm.org/latest/tools/igv/.

[26] Oracle. Java Instrumentation API. on-line at: https://docs.oracle.com/en/java/javase/23/docs/api/java.instrument/java/lang/instrument/package-summary.html.

[27] Oracle. JDK 23 Release Notes. on-line at: https://www.oracle.com/java/technologies/javase/23-relnote-issues.html.

[28] Oracle. JVM Class Data Sharing. on-line at: https://docs.oracle.com/en/java/javase/23/vm/class-data-sharing.html.

[29] Oracle. JVM Tool Interface. on-line at: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[30] Oracle. Native Image. on-line at: https://www.graalvm.org/latest/reference-manual/native-image/.

[31] Oracle. Native Image Reachability Metadata. on-line at: https://www.graalvm.org/latest/reference-manual/native-image/metadata/.

[32] Oracle. Native Image Tracing Agent. on-line at: https://www.graalvm.org/latest/reference-manual/native-image/metadata/AutomaticMetadataCollection/.

[33] Oracle. Polyglot Programming. on-line at: `https://www.graalvm.org/latest/reference-manual/polyglot-programming/`.

[34] Oracle. The javac Command. on-line at: `https://docs.oracle.com/en/java/javase/23/docs/specs/man/javac.html`.

[35] Oracle. The javap Command. on-line at: `https://docs.oracle.com/en/java/javase/23/docs/specs/man/javap.html`.

[36] Oracle. The jdb Command. on-line at: `https://docs.oracle.com/en/java/javase/23/docs/specs/man/jdb.html`.

[37] Oracle. Truffle Language Implementation Framework. on-line at: `https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/`.

[38] VMware. Spring Framework. on-line at: `https://spring.io/projects/spring-framework`.

[39] VMware. Spring GraalVM Native Image Support. on-line at: `https://docs.spring.io/spring-boot/docs/3.2.3/reference/html/native-image.html`.

[40] VMware. Spring PetClinic. on-line at: `https://spring-petclinic.github.io/`.

[41] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

Biografija autora

Aleksandar Stefanović rođen je 11. septembra 1999. godine u Beogradu. Godine 2014. upisuje ETŠ „Nikola Tesla" u Beogradu, smer „Elektrotehničar informacionih tehnologija" i završava je 2018. godine sa odličnim uspehom. Iste godine upisuje Matematički fakultet Univerziteta u Beogradu, smer Informatika, na kojem je diplomirao 2023. godine sa prosečnom ocenom 9.08. Odmah po diplomiranju upisuje i master studije na istom fakultetu, smer Informatika.

U školskim godinama 2023/2024 i 2024/2025 biran je za saradnika u nastavi na Matematičkom fakultetu gde drži vežbe iz predmeta „Razvoj softvera", „Projektovanje baza podataka" i „Programiranje 2". U martu 2025. godine započinje istraživačku praksu u kompaniji *Oracle Labs*, gde radi na razvoju kompilatorske infrastrukture *GraalVM*.