Facultatea de
Matematică şi Informatică
Universitatea din Bucureşti

Un optimizator de dimensiune pentru Java

Teză de Licență

Petru-Eric Stăvărache

22 Iunie, 2018

Coordonator Conf. Dr. Traian-Florin Șerbănuță

Facultatea de Matematică și Informatică, UNIBUC

Rezumat

Sistemul de operare Android este cea mai populară platformă pentru telefoanele mobile, iar limbajul utilizat pentru a dezvolta aplicații este Java. O reducere de doar câțiva octeți a dimensiunii pachetului unei aplicații populare, precum Facebook, ar duce la economisirea de câțiva giga-octeți de trafic de internet lunar.

In aceasta teza am proiectat și implementat un compilator care efectuează optimizări de dimensiune a codului asupra fișierelor compilate Java.

Aceste optimizări elimina funcțiile și metodele neutilizate dintr-un proiect dezvoltat in limbajul Java și se bazează pe analiza statică a proiectului. O serie de teste au fost create pentru a testa corectitudinea, cat și eficienta optimizărilor aplicate. Compilatorul lucrează direct cu fișiere compilate, in formatul utilizat și de către Android.

Cuprins

Capitolul 1

Introducere

Eliminarea codului inutil (Eng. "Dead code elimination") [21] este o optimizare clasică. Ea presupune eliminarea dintr-un program a codului care nu afectează rezultatul computației.

Codul poate să fie eliminat dacă, de exemplu, nu există niciun fir de execuție care să conțină instrucțiunile respective, sau dacă acel cod nu are efecte laterale.

Acest tip de optimizare este implementată tradițional în limbajele compilate, precum C, cât și în cele 'compilate-la-timp' (Eng. 'just-in-time') precum Java sau JavaScript. Beneficiile principale aduse de către optimizare sunt:

1. Reducerea dimensiunii programului
2. Creșterea vitezei de execuție

Cu cât un limbaj este mai dinamic, și permite mai multe schimbări ale comportamentului la rulare, cu atât eliminarea de cod nefolosit devine o sarcină mai grea.

De exemplu, echipa care dezvoltă motorul de JavaScript pentru browser-ul Internet Explorer a avut probleme de corectitudine în optimizările realizate, datorită naturii dinamice a limbajului JavaScript [18].

Deși pentru un programator nu este natural să creeze cod nefolosit, acest lucru nu înseamnă ca principiul eliminării acestuia nu poate fi aplicat; compilatoarele în sine, prin modul lor de a trece de mai multe ori prin codul sursă și de a avea mai multe reprezentări intermediare, pot genera cod nefolosit.

Limbajul C, prin includerea de fișiere mot-a-mot, este un exemplu bun pentru acest lucru: un program de câteva linii, care afișează "Hello, world!", ajunge să aibă, înaintea eliminării codului mort, câteva mii de linii și sute

de funcții nefolosite. Acest lucru se datorează nevoii includerii bibliotecii standard.

1.1 Eliminarea metodelor

Eliminarea metodelor și funcțiilor este un una dintre multele posibilități de a scăpa de cod nefolosit.

Procedeul constă în îndepărtarea dintr-un program a funcțiilor și a metodelor care nu sunt niciodată apelate.

Această lucrare va explora această optimizare particulară, atât teoretic, cât și implementat în limbajul Java.

1.2 Muncă anterioară

1.2.1 C si C++

Compilatorul gcc este capabil să elimine funcțiile nefolosite, însă acest comportament nu este implicit.

Pentru exemplificare, vom considera programul

main.c

```
#include <stdio.h>

void foo()
{
    puts("in foo!");
}

void bar()
{
    puts("in bar!");
}

int main()
{
    foo();
    return 0;
}
```

Dacă îl compilăm cu **gcc** folosind opțiunea `-Os` (permite optimizările pentru dimensiune), atunci binarul generat va conține atât funcția `foo`, care este apelată din `main`, cât și funcția `bar`, care nu este niciodată folosită.

```
$ gcc main.c -Os
$ nm a.out | grep "foo"
000000000000064a T foo
$ nm a.out | grep "bar"
0000000000000656 T bar
$
```

Acest lucru se datorează faptului că optimizatorul pentru C lucrează cu simboluri, nu direct cu funcții. Diferența este că un simbol poate reprezenta atât o funcție, cât și o variabilă. Așadar, când optimizatorul decide dacă să elimine un simbol, acesta nu consideră dacă acel simbol este o funcție sau un obiect (o variabilă se mai numește și obiect).

În limbajul C, un obiect global trebuie să fie inițializat cu o valoare constantă (i.e., poate fi evaluată la compilare), ceea ce înseamnă că inițializările nu pot avea efecte laterale [1].

Pe de altă parte, în limbajul C++, inițializările pot conține expresii arbitrare, care necesită evaluarea acestora la timpul rulării [2] și care pot avea efecte laterale.

Optimizarea de a elimina funcțiile nefolosite nu este activată în mod implicit deoarece optimizatorul folosit în gcc lucrează și pentru C, dar și pentru C++.

Această optimizare ar altera comportamentul unui program C++ care folosește efecte laterale la inițializare, întrucât dacă optimizatorul elimină din program un simbol care corespunde unei variabile, acea variabilă nu mai este inițializată, la rularea programului, iar efectul lateral corespunzător inițializării ei nu se mai realizează.

Opțiuni speciale date compilatorului

Optimizarea de a elimina simboluri nefolosite nu este aplicabilă oricărui program. Totodată, dacă programatorul îi garantează compilatorului că programul nu folosește inițializări cu efecte laterale, compilatorul gcc este capabil de a elimina funcții nefolosite.

Vom folosi în continuare programul definit la 1.2.1. Conform [16], este necesar să pasăm opțiuni suplimentare compilatorului, iar acesta va elimina funcția nefolosită:

```
$ gcc main.c -Wl,-static -Wl,--gc-sections -fdata-
   sections -ffunction-sections -Os
$ nm a.out | grep "foo"
0000000000400acd T foo
$ nm a.out | grep "bar"
$
```

În concluzie, pentru limbajele C și C++, este nevoie ca programatorul să garanteze că aplicarea optimizării păstrează corectitudinea programului.

1.2.2 Java

La compilare

Compilatorul standard de Java, `javac`, nu este un compilator optimizator: acesta doar transformă codul Java în bytecode, fără a aplica transformări de optimizre pe acesta.

În limbajul Java compilatorul lucrează cu câte un fișier o data: acesta nu are conceptul de proiect sau executabil, ci doar de fișiere clasă. Compilatorul nu cunoaște ce funcții pot fi apelate, sau din ce locuri, deci nu poate efectua nicio optimizare.

La Rulare

Optimizarea Java se produce la timpul de rulare. Mașina virtuală Java interpretează codul binar, iar secțiunile care sunt executate foarte des (de exemplu, corpul unei bucle, sau o metodă care este apelată la intervale regulate) sunt optimizate direct la rulare.

Fiecare implementare a mașinii virtuale are propriile moduri de a optimiza codul, însă cele mai moderne se bazează pe profilarea acestuia și detectarea posibililor candidați pentru optimizare [10].

Android

Deși codul Java arbitrar este imposibil de optimizat, structuri particulare de proiecte permit eliminarea de cod nefolosit. De-a lungul timpului, au fost create mai multe optimizatoare pentru scopuri specifice, însă majoritatea nu mai sunt întreținute în mod activ [3]. Această lipsă de interes probabil se datorează faptului că cei mai mulți oameni sunt deserviți foarte bine de optimizarea oferită de mașinile virtuale la timpul de rulare.

În zilele noastre, cel mai întâlnit loc unde este JVM-ul nu este satisfăcător este optimizarea de aplicații pentru sistemul de operare Android [14]. Programul recomandat de Google este `Proguard` [15]. Deși Proguard este el capabil de optimizări de dimensiune, principalul său scop este obfuscarea codului, pentru a nu putea fi descifrat (Eng. "reverse engineered")

Soluția acestui program pentru a rezolva problema pe care o întâmpină `javac` este să îi ceară programatorului să specifice în mod explicit modurile în care programul este rulat (e.g., de unde poate începe execuția).

Având această informație, `proguard` poate analiza static proiectul pentru a deduce care metode pot fi eliminate.

1.3 Asumpții

Înainte de a începe descrierea problemei, voi expune asumpțiile pe care le-am făcut în implementarea aplicației pentru Java. Deși aceste asumpții par destul de restrictive, cele mai multe proiecte normale, atât de Android, cât și normale, le vor satisface.

Programul dezvoltat în această lucrare optimizează programele Java la compilare. În urmare, acesta nu poate să trateze invocarea de metode la rulare.

Programul va emite un mesaj de eroare în caz că detectează că asumpția este încălcată,

1.3.1 Reflecția

Reflecția constă în introspecția programului la rulare – spre exemplu, identificarea câmpurilor sau metodelor unei clase.

Deși reflecția nu este de obicei folosită în acest scop (decât în circumstanțe specifice, de exemplu implementarea altor limbaje), aceasta permite programului să apeleze metode care nu sunt cunoscute decât la timpul rulării.

De exemplu, programul [4]

```
public class Main {
  public int add(int a, int b)
  {
     return a + b;
  }

  public static void main(String args[])
  {
     try {
       Class cls = Class.forName("Main");
       Class partypes[] = new Class[2];
        partypes[0] = Integer.TYPE;
        partypes[1] = Integer.TYPE;
        Method meth = cls.getMethod(
          "add", partypes);
        Main methobj = new Main();
        Object arglist[] = new Object[2];
        arglist[0] = new Integer(37);
        arglist[1] = new Integer(47);
        Object retobj
          = meth.invoke(methobj, arglist);
        Integer retval = (Integer)retobj;
        System.out.println(retval.intValue());
```

```
        }
        catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

apelează metoda add în mod dinamic.

Pentru a putea implementa aplicația, am presupus că în programele pe care le optimizăm reflecția nu este utilizată pentru a apela metode dinamic.

1.3.2 Invocarea dinamică de metode

În Java 7, a fost introdusă o noua instrucțiune de cod mașină menită să faciliteze implementarea de limbaje dinamice în Java [17]. Aceasta este o alternativă mai rapidă la reflecție pentru apelarea de metode care nu sunt cunoscute decât la rulare.

Din aceleași motive ca la reflecție, voi presupune că programele nu vor conține această instrucțiune.

Funcții lambda

Funcțiile lambda (sau funcțiile anonime) sunt o adiție recentă în limbajul Java. Implementarea folosește invocarea dinamică atunci când o astfel de funcție este instanțiată. Așadar, dacă un program folosește funcții anonime, acesta va încălca presupunerile făcute și nu va putea fi optimizat corect.

1.4 Structura lucrării

În prima parte a acestei lucrări voi formaliza în mod teoretic problema de a optimiza un program. În a doua parte voi descrie limbajul Java, și modul de funcționare al acestuia. În a treia parte voi detalia cum putem adapta teoria de optimizare pentru programe dezvoltate în limbajul Java. În ultima parte voi prezenta detaliile de implementare ale optimizatorului de Java.

Capitolul 2

Problema de optimizare

Toate programele date ca exemplu sunt scrise într-un pseudo cod inspirat din limbajul Python.

2.1 Formalizarea problemei

Fie programul $\mathcal{P}$ un proiect Java, format dintr-o mulțime de fișiere clasă. Scopul optimizatorului este să creeze un program $\mathcal{P}'$, care să se comporte identic cu $\mathcal{P}$, și să fie mai bun decât $\mathcal{P}$ pentru o anumită metrica $\mathcal{M}$.

2.2 Diferențierea programelor

Doua programe $\mathcal{P}$ și $\mathcal{Q}$ pot fi diferențiate dacă exista un input $\mathcal{I}$ pentru care $\mathcal{P}$ rulat pe $\mathcal{I}$ și $\mathcal{Q}$ rulat pe $\mathcal{I}$ dau rezultate diferite.

$\exists \mathcal{I}$ pentru care $\mathcal{P}(\mathcal{I}) \neq \mathcal{Q}(\mathcal{I})$

Unde prin rezultat înțelegem atât output-ul programului, în sensul pur matematic, cât și efectele laterale generate, care au efect asupra mediului unde rulează programul.

Dacă doua programe nu pot fi diferențiate (i.e., pentru toate input-urile $\mathcal{I}$, cele doua programe se comporta la fel), vom spune despre ele ca sunt echivalente.

De exemplu, fie $\mathcal{P}$

```
def P(a: int, b: int) -> int:
    for i = 1:b
        a = inc(a)
    return a
```

Și fie $\mathcal{Q}$

```
def Q(a: int, b: int) -> int:
    return a + b
```

Atunci pentru orice a și b din $\mathbb{N}$, $\mathcal{P}$(a, b) va fi egal cu $\mathcal{Q}$(a, b).

2.3 Metrici de optimizare

În contextul optimizării de programe este nevoie să definim ce înseamnă ca dintre doua programe echivalente $\mathcal{P}$ și $\mathcal{Q}$, $\mathcal{P}$ să fie mai performant decât $\mathcal{Q}$. Cele mai folosite doua metrici sunt metrica de viteză de execuție a unui program, și metrica de dimensiune a programului.

2.3.1 Metrica de viteza

Timpul de rulare

Vom defini timpul de rulare al unui program $\mathcal{P}$ pe un input $\mathcal{I}$ ca fiind diferența de timp dintre când programul își începe execuția, până când acesta își termină execuția.

Pe sistemele *nix, un mod ușor a măsura timpul de rulare este folosind comanda *time*:

```
$ time ./build.sh
./build.sh  0.47s user 0.20s system 100% cpu 0.664
   total
```

În acest context, vom spune ca programul *build.sh* a rulat pentru un timp de 0.664 secunde.

Vom defini astfel

$$time(\mathcal{P}, \mathcal{I})$$

ca fiind timpul de rulare al programului $\mathcal{P}$ pe input-ul $\mathcal{I}$.

Comparare bazata pe timpul de rulare

Fie doua programe echivalente $\mathcal{P}$ și $\mathcal{Q}$.

Vom spune că $\mathcal{P}$ este mai rapid decât $\mathcal{Q}$ pe baza timpului de rulare dacă timpul de rulare mediu al lui $\mathcal{P}$ este mai mic decât timpul de rulare mediu al lui $\mathcal{Q}$:

$$\sum_{\mathcal{I}\ input} time(\mathcal{P}, \mathcal{I}) < \sum_{\mathcal{I}\ input} time(\mathcal{Q}, \mathcal{I})$$

Pe baza acestei comparații, putem defini o relație de ordine asupra mulțimii programelor: $\mathcal{P} < \mathcal{Q}$ dacă $\mathcal{P}$ este mai rapid decât $\mathcal{Q}$.

Problema optimizării pe baza metricii de viteza

Având definită relația de ordine, problema optimizării pe baza metricii de viteza este:

Dându-se un program $\mathcal{P}$, să se găsească $\mathcal{Q}$ ca cel mai rapid program echivalent cu $\mathcal{P}$.

2.3.2 Metrica de dimensiune

Dimensiunea unui program

Pentru un program $\mathcal{P}$, vom defini dimensiunea acestuia ca fiind suma dimensiunilor tuturor instrucțiunilor acestui program:

$$size(\mathcal{P}) = \sum_{i \in \mathcal{P}} instruction\_size(i)$$

Unde prin $instruction\_size(i)$ înțelegem numărul de octeți ocupați de instrucțiunea i.

De exemplu, pentru limbajul Java, instrucțiunea *invokedynamic* ocupă 5 octeți, în timp ce instrucțiunea *dmul* ocupă un singur octet.

Comparare bazată pe dimensiunea programelor

Pentru doua programe $\mathcal{P}$ și $\mathcal{Q}$, vom spune ca $\mathcal{P}$ este mai mic decât $\mathcal{Q}$ dacă $size(\mathcal{P}) < size(\mathcal{Q})$.

Similar ca la metrica de viteza, putem defini o relație de ordine pe mulțimea programelor.

Problema optimizării pe baza metricii de dimensiune

Având definită relația de ordine, problema optimizării pe baza metricii de dimensiune este aceeași ca la metrica de viteză:

Dându-se un program $\mathcal{P}$, să se găsească $\mathcal{Q}$ cel mai mic program echivalent cu $\mathcal{P}$:

$$\operatorname*{argmin}_{\mathcal{Q}} \mathcal{P} \text{ echivalent cu } \mathcal{Q}$$

2.4 Discuție asupra metricilor

Pentru cele mai multe cazuri, cele doua metrici sunt corelate – o reducere a dimensiunii programului aduce cu ea și o reducere a timpului de rulare.

Totodată, exista cazuri când cele doua metrici sunt contrare. Un exemplu clasic este tehnica de "derularea buclelor" (Eng. "Loop unrolling"). Aceasta constă în explicitarea unei bucle cu un număr cunoscut de iterații:

Programul

```
s = 0
for i = 1:10:
    s = inc(s)
```

Va fi optimizat pentru viteză în

```
s = 0
s = inc(s))
...
s = inc(s))
```

În timp ce această optimizare va crește numărul de instrucțiuni al programului, deci și dimensiunea acestuia.

Deoarece cele mai multe programe utilizate nu rulează pe medii constrânse de memorie, tipul de optimizare folosit aproape întotdeauna este cel de viteza: este mult mai util dacă un program rulează de 2 ori mai repede, decât dacă acesta ocupă de 2 ori mai puțin spațiu.

Acest lucru se datorează faptului că performanța memoriei (prețul per unitate de memorie) a continuat să scadă în ultimul deceniu, în timp ce performanța procesoarelor (numărul de instrucțiuni executate per secundă) a stagnat.

Asa cum se poate observa în figura 2.1, trendul care urma legea lui Moore [19] a început să se oprească. Pe de altă parte, eficiența memoriei calculatoarelor și-a continuat trendul de creștere, așa cum se poate observa în figura 2.2.

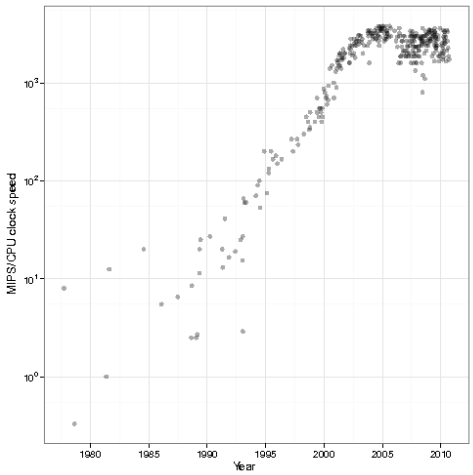

Figura 2.1: Evoluția puterii de procesare[12]

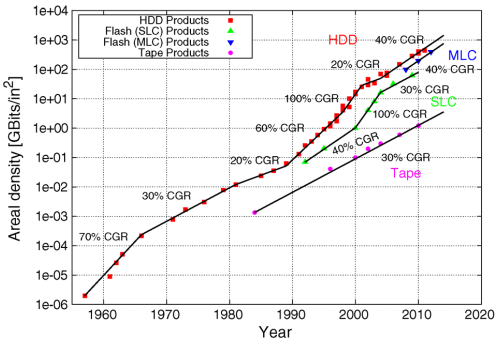

Figura 2.2: Evoluția puterii de procesare[13]

Capitolul 3

Optimizări de dimensiune

3.1 Eliminarea funcțiilor nefolosite

Această lucrare conține o abordare de optimizare a dimensiunii programelor bazată pe eliminarea funcțiilor nefolosite.

Deși metrica principală este aceea de dimensiune, reducerea numărului de instrucțiuni ale programului poate avea și beneficii asupra vitezei de execuție, din cauza interacțiunilor cu cache-ul calculatoarelor. Pe de altă parte, acest beneficiu este destul de minor, și nu reprezintă principala motivație.

În continuare, voi descrie conceptele necesare pentru a înțelege cum putem implementa acest fel de optimizare.

3.2 Funcție/Metoda

În domeniul limbajelor de programare, o funcție $\mathcal{F}$ este formată dintr-o colecție de instrucțiuni tratate ca un întreg, împreună cu un protocol pentru executarea acelor instrucțiuni.

O metoda $\mathcal{M}$ reprezintă o funcție asociată unei clase.

În limbajul Java este imposibil să avem o funcție care să nu aparțină unei metode. Deși definițiile nu sunt echivalente, deoarece în Java nu există funcții propriu-zise, ci doar metode, termenii de funcție și de metodă sunt considerați interschimbabili.

Protocolul de executare a funcțiilor variază de la limbaj la limbaj și nu este important pentru scopurile noastre.

De exemplu, funcția f

```
0 def f(a, b, c):
1     a = a + b
```

```
2       return c
```

conține 2 instrucțiuni, corespunzătoare liniilor 1-2:

(i_1) $a = a + b$

(i_2) *return* c

Deși în acest exemplu funcția $\mathcal{F}$ începe de pe poziția 0, într-un program format din mai multe funcții locul de început al funcției variază (indicele primei instrucțiuni).

Vom spune că două instrucțiuni sunt egale dacă se află pe aceeași linie. i.e., reprezintă aceeași poziție în program.

Pentru programul format din

```
0 def g(a, b, c):
1       a = a + b
2       a = a * b
3       return a
4 def f(a, b, c):
5       a = a + b
6       return c
```

instrucțiunile i_1 și i_5 **nu** sunt egale.

Vom spune ca o instrucțiune **i** aparține funcției $\mathcal{F}$ dacă există o linie a lui $\mathcal{F}$ unde putem găsi instrucțiunea respectivă.

3.3 Punctul de intrare principal

Fie $\mathcal{P}$ un program, executat în ordine secvențială. Punctul principal de intrare al lui $\mathcal{P}$ reprezintă locul de unde începe execuția programului – prima instrucțiune executată.

Acest loc va fi notat cu `main`.

3.4 Șirul de execuție al unui program

Un șir de execuție $\mathcal{S}$ al programului $\mathcal{P}$ reprezintă o înlănțuire finită de instrucțiuni executate secvențial, începând cu `main` și terminându-se cu ultima instrucțiune executată de către $\mathcal{P}$ înainte ca programul să se termine. Pentru ca un șir de execuție $\mathcal{S}$ să fie valid, trebuie să existe un input pe care, dacă rulam programul, acesta să execute fix instrucțiunile lui $\mathcal{S}$.

3.5 Funcții nefolosite

O funcție a unei clase poate fi eliminată dacă nu există niciun șir de execuție valid al programului care să treacă prin acea funcție.

Vom nota cu $elim(\mathcal{F}) = 1$ dacă funcția $\mathcal{F}$ poate fi eliminată deoarece este nefolosită.

Cu alte cuvinte, $elim(\mathcal{F}) = 1$ dacă

Pentru oricare $\mathcal{S}$ șir de execuție, nu există **i** din $\mathcal{S}$ care să facă parte din corpul funcției $\mathcal{F}$.

3.5.1 Sirurile de execuție - în practică

Problema determinării tuturor șirurilor de execuție ale unui program este o problema grea, întrucât aceasta ar implica rularea programului pe fiecare input posibil.

Din cauza că unele programe pot să fie definite doar parțial (comportamentul lor să fie nedefinit pe anumite clase de input), această problemă poate fi echivalentă cu Problema Opririi (Eng. "The Halting Problem") [11]: nu avem cum să știm dacă un input al programului este bun sau nu fără să rulam programul pe acel input. Totodată, nu putem determina dacă un program se va termina vreodată.

Deoarece problema opririi este nedecidabilă, determinarea tuturor șirurilor de execuție posibile ale lui program arbitrar este de asemenea o problema nedecidabilă.

3.5.2 Supersirul de execuție

Această lucrare va construi o aproximare – un "supersir" de execuție format dintr-o superpoziție a tuturor șirurilor de execuție existente, inclusiv pe cele invalide (pentru care nu există un input care să le genereze).

De exemplu, pentru programul

```
1  def main():
2      if False:
3          f()
4
5  def f():
6      if True:
7          return
8      g()
9
10 def g():
```

```
11      pass
12
13  def h():
14      pass
```

Șirul de execuție folosit în problema noastră va conține și secvența corespunzătoare lanțului de apeluri $main() -> f()$, cât și lanțului $main() -> f() -> g()$, chiar dacă acestea sunt invalide.

Totodată, șirul de execuție generat **nu** va conține secvența $main() -> h()$, sau orice secvență care să îl conțină pe h, din cauză că aceste secvențe nu sunt șiruri de execuție (nu există nicio secvență de instrucțiuni secvențiale care să pornească din funcția `main` și să ajungă în funcția h).

3.5.3 Graful apelurilor de funcții

Vom construi un graf $\mathcal{G}$ al metodelor: fiecărui nod îi va corespunde o metodă prezentă în program, iar fiecare metodă va avea un singur nod corespunzător.

Supersirul de execuție consideră toate secvențele de instrucțiuni dintr-o metodă, chiar dacă acestea sunt invalide. Așadar, supersirul va considera toate posibilele căi de apel către alte funcții.

Așadar, putem reduce problema construcției supersirului la problema construirii grafului $\mathcal{G}$.

În acest graf, există o muchie de la metoda F la metoda G dacă în secvența de instrucțiuni a lui F (corpul funcției) există un apel către funcția G.

Având acest graf generat, putem calcula mulțimea $\mathcal{M}$ a metodelor care sunt accesibile pornind din nodul funcției care conține punctul principal de intrare. Pentru a optimiza programul, vom elimina toate metodele care nu aparțin acestei mulțimi.

Demonstrație că acest procedeu este corect:

Lemă 3.1 *Fie m o metodă care nu aparține lui $\mathcal{M}$.*

Acest fapt înseamnă că în supersirul de execuție nu există nicio instrucțiune care să aparțină lui m.

Cum supersirul de execuție include toate șirurile de execuție valide, înseamnă că nu există niciun șir de execuție valid care, la un moment dat, să apeleze funcția m.

Prin urmare, programul obținut prin eliminarea funcției m va fi echivalent cu programul inițial.

3.5.4 Algoritmul pentru eliminarea funcțiilor

Pentru a construi graful, trebuie să putem deduce pentru fiecare metodă care sunt metodele pe care aceasta le apelează. Acest lucru este dependent de limbajul asupra căruia se aplică optimizarea, așadar îl vom considera deja implementat.

Putem considera astfel funcția

```
def direct_calees(m: Method) -> [Method]
    return all methods directly called by m.
```

ca fiind la dispoziția noastră.

Pentru a forma mulțimea metodelor accesibile din `main`, putem utiliza o parcurgere în lățime a grafului.

```
def reachable_methods(main: Method) -> [Method]:
    coada = [main]
    at = 0
    while at < size(coada):
        m = coada[at]
        at = at + 1
        for next in direct_calees(m):
            if next not in coada:
                coada.push(next)
    return coada
```

Având mulțimea generată, rămâne doar să eliminăm funcțiile care nu fac partea din ea:

```
def optimize_for_size(p: Program) -> Program:
    main = p.main_method()
    used_methods = reachable_methods(main)
    for m in p.all_methods():
        if m not in used_methods:
            p.remove_method(m)
    return p
```

Capitolul 4

Detaliile tehnice ale limbajului Java

În acest capitol vom detalia limbajul Java și modul de funcționare a acestuia.

4.1 Limbajul Java

Java este un limbaj de programare orientat pe obiecte. Acesta a fost dezvoltat de către Sun Microsystems (acum Oracle), iar prima versiune a apărut în anul 1995.

Java s-a bazat pe sintaxa limbajului C, și a introdus noțiunea de "scrie o data, rulează peste tot" (Eng. "write once, run everywhere"). Spre deosebire de C și de C++, care trebuiesc compilate pentru fiecare platformă țintă, Java a avut avantajul că trebuie compilat o singura data, și va merge garantat pe toate platformele suportate de limbaj.

4.2 Java Bytecode

Soluția limbajului Java pentru a fi independent de platformă este de a transforma codul într-o reprezentare intermediara, în loc de direct în cod binar pentru o anumită arhitectură .

Compilatorul Java (`javac`), transformă codul Java într-un limbaj intermediar, numit Java Bytecode.

Acest limbaj este un limbaj de nivel scăzut (Eng. "low-level"), destinat în mod exclusiv procesării de către mașini, spre deosebire de codul Java, care este destinat oamenilor.

După ce compilatorul a procesat codul Java, provenit din fișiere .java în format text, acesta salvează rezultatul în fișiere de tip clasă (.class) în format binar.

4.3 JVM

Odată generate fișierele binare, acestea sunt executate pe o mașina virtuală specifica limbajului Java — numită `JVM` (Eng. Java Virtual Machine).

Această mașina virtuală are rolul de a citi fișierele de clasă binare și de a le interpreta.

Mașina virtuală este implementată ca o "mașină cu stivă" (Eng. Stack machine), unde toate instrucțiunile limbajului bytecode interacționează cu datele de pe o stivă controlată de aplicație.

Mașina virtuală însuși este implementată în C/C++, și este compilată în cod binar direct, dependent de arhitectura calculatorului. Dezvoltatorii limbajului Java sunt responsabili pentru corectitudinea și siguranța mașinii virtuale, în timp ce dezvoltatorii de aplicații Java au garanția că dacă codul lor Java este corect, atunci acesta va rula la fel, deterministic, pe orice platforma.

În acestă privință, limbajul Java poate fi văzut ca un limbaj interpretat. Comparând cu alte limbaje populare interpretate, ca de exemplu Python, Ruby, sau Perl, ne-am aștepta ca și Java să fie la fel de încet ca acestea [5]. Totuși, Java obține performanțe mult mai bune decât acestea. Acest fapt se datorează compilării tocmai-la-timp (Eng. just-in-time), în care atunci când interpretorul observă o secventa de cod care este interpretată repetitiv de foarte multe ori, va genera direct cod binari pentru aceasta.

4.4 Fișierele clasa

Fișierele de clasa Java sunt formate din 10 secțiuni [6]:

1. Constanta magică.
2. Versiunea fișierului.
3. Constantele clasei.
4. Permisiunile de acces.
5. Numele clasei din fișier.
6. Numele super clasei.
7. Interfețele pe care clasa le implementează.
8. Câmpurile clasei.
9. Metodele clasei.
10. Atribute ale clasei.

În continuare voi da o scurta descriere a formatului secțiunilor.

4.5 Secțiunile fișierelor clasa

4.5.1 Magic

Toate fișierele clasă trebuiesc să înceapă cu un număr denumit constanta magică. Acesta este folosit pentru a identifica în mod unic că acestea sunt într-adevăr fișiere clasă. Numărul magic are o valoare memorabilă: reprezentarea hexadecimală este `0xCAFEBABE`,

4.5.2 Versiunea

Versiunea unui fișier clasă este dată de doua valori, versiunea majoră `M` și versiunea minoră `m`. Versiunea clasei este atunci reprezentata ca `M.m`. (e.g., `45.1`). Aceasta este folosită pentru a menține compatibilitatea în cazul modificărilor mașinii virtuale care interpretează clasa sau ale compilatorului care o generează.

4.5.3 Constantele clasei

Tabela de constante este locul unde sunt stocate valorile literale constante ale clasei:

- Numere întregi.
- Numere cu virgula mobilă.
- Șiruri de caractere, care pot reprezenta la rândul lor:
  - Nume de clase.
  - Nume de metode.
  - Tipuri ale metodelor.
- Informații compuse din datele anterioare:
  - Referința către o metodă a unei clase.
  - Referința către o constantă a unei clase.

Toate celelalte tipuri de date compuse, cum ar fi metodele sau câmpurile, vor conține indecși în tabela de constante.

Aproape toate tipurile de constante ocupă o singură poziție în tabelă, însă, din motive istorice, unele constante ocupă două poziții. Tot din motive istorice, tabela este indexată de la 1, și nu de la 0, cum sunt celelalte.

4.5.4 Permisiunile de acces

Aceste permisiuni constau într-o mască de biți, care reprezintă operațiile permise pe această clasă:

- dacă clasa este publică, și poate fi accesată din afara pachetului acesteia.
- dacă clasa este finala, și dacă poate fi extinsă.
- dacă invocarea metodelor din super clasă să fie tratata special.
- dacă este de fapt o interfață, și nu o clasă.
- dacă este o clasă abstractă și nu poate fi instanțiată.

4.5.5 Clasa curentă

Reprezintă un indice în tabela de constante, unde sunt stocate informații despre clasa curentă.

4.5.6 Clasa super

Reprezintă un indice în tabela de constante, cu informații despre clasa din care a moștenit clasa curentă. Dacă este 0, clasa curentă nu moștenește nimic: singura clasă fără o super clasă este Object.

E.g. Pentru

```
public class MyClass extends S implements I
```

Indicele corespunde lui `S`.

4.5.7 Interfețele

Reprezintă o colecție de indici în tabela de constante. Fiecare valoare de la acei indici constituie o interfață implementată în mod direct de către clasa curentă. Interfețele apar în ordinea declarată în fișierele Java.

E.g. Pentru

```
class MyClass extends S implements I1, I2
```

Primul indice ar corespunde lui `I1`, iar al doilea lui `I2`.

4.5.8 Câmpurile

Reprezintă informații despre câmpurile (Eng. Fields) clasei:

- Permisiunile de acces: dacă este public sau privat, etc.
- Numele câmpului.
- Tipul câmpului.
- Alte atribute: dacă este depreciat, dacă are o valoare constantă, etc.

4.5.9 Metodele

Reprezintă informații despre toate metodele clasei, inclusiv constructorii:

- Permisiuni de acces: dacă este publică sau privată, dacă este finală , dacă este abstractă.
- Numele metodei.
- Tipul metodei.
- În caz că nu este abstractă, codul metodei.
- Alte atribute:
  - Ce excepții poate arunca.
  - Dacă este depreciată.

Codul metodei este partea cea mai importantă, iar formatul acestuia urmează să fie detaliat ulterior.

4.5.10 Atributele

Reprezintă alte informații despre clasă, cum ar fi:

- Clasele definite în interiorul acesteia.
- În caz că este o clasă anonimă sau definită local, metoda în care este definită.
- Numele fișierul sursă din care a fost compilată clasa.

4.6 Tipuri de date

În continuare, voi descrie din punct de vedere tehnic tipurile de date întâlnite în fișierele de clasă.

4.6.1 Tipuri de baza

În formatul fișierelor clasa există trei tipuri elementare, toate bazate pe întregi. În caz că un întreg are mai mulți octeți, aceștia au ordinea de `big-endian`: cel mai semnificativ octet va fi mereu primul în memorie.

Nume	Semantica	Echivalentul în C
`u1`	întreg pe un octet, fără semn	`unsigned char` sau `uint8_t`
`u2`	întreg pe doi octeți, fără semn	`unsigned short` sau `uint16_t`
`u4`	întreg pe un octet, fără semn	`unsigned int` sau `uint32_t`

În codul sursă al proiectului, acestea sunt tratate astfel:

```
using u1 = uint8_t;
using u2 = uint16_t;
using u4 = uint32_t;
```

4.6.2 Tipuri compuse

4.6.3 Constantele

Fiecare constantă din tabela de constante începe cu o etichetă de 1 octet, care reprezintă datele și tipul structurii. Conținutul acesteia variază în funcție de etichetă, însa indiferent de etichetă, conținutul trebuie să aibă cel puțin 2 octeți.

CONSTANT_Class

Corespunde valorii etichetei de 7 și conține un indice spre un alt câmp în tabela de constante, de tipul `CONSTANT_Utf8` — un șir de caractere. Acel șir de caractere va conține numele clasei.

CONSTANT_Fieldref

Corespunde valorii etichetei de 9 și conține o referință spre câmpul unei clase. Referința constă în doi indici, amândoi care arată spre tabela de constante. Primul indice arată spre o constantă `CONSTANT_Class`, care reprezintă clasa sau interfața căreia aparține metoda. Al doilea indice arată spre o constantă `CONSTANT_NameAndType`, care conține informații despre numele și tipul câmpului.

CONSTANT_Methodref

Corespunde valorii etichetei de 10 și conține o referința spre metoda unei clase. Are o structură identică cu `CONSTANT_Fieldref`, doar că primul indice arată neapărat spre o clasă, în timp ce al doilea indice arată spre numele și tipul metodei.

CONSTANT_InterfaceMethodref

Corespunde valorii etichetei de 11 și conține o referință spre metoda unei interfețe. Are o structura identică cu `CONSTANT_Methodref`, doar că primul indice arată spre o interfață.

CONSTANT_String

Corespunde valorii etichetei de 8 și reprezintă un șir de caractere. Conține un indice către o structură de tipul `CONSTANT_Utf8`.

CONSTANT_Integer

Corespunde valorii etichetei de 3 și conține un întreg pe 4 octeți.

CONSTANT_Float

Corespunde valorii etichetei de 4 și conține un număr cu virgulă mobilă pe 4 octeți.

CONSTANT_Long

Corespunde valorii etichetei de 5 și conține un întreg pe 8 octeți. Din motive istorice, ocupă 2 spatii în tabela de constante.

CONSTANT_Double

Corespunde valorii etichetei de 6 și conține un număr cu virgulă mobilă pe 8 octeți. Din motive istorice, ocupă 2 spatii în tabela de constante.

CONSTANT_NameAndType

Corespunde valorii etichetei de 12. Descrie numele și tipul unui câmp sau al unei metode, fără informații despre clasă. Conține doi indici, amândoi către structuri de tipul `CONSTANT_Utf8`. Primul reprezintă numele, iar al doilea tipul.

CONSTANT_Utf8

Corespunde valorii etichetei de 1. Reprezintă un șir de caractere encodat în formatul UTF-8. Conține un întreg `length`, de tipul `u2`, și apoi `length` octeți care descriu șirul în sine. Din cauza că este encodat ca UTF-8, un singur caracter poate fi format din mai multi octeți.

CONSTANT_MethodHandle

Corespunde valorii etichetei de 15 și conține o referințe către un câmp, o metodă de clasă, sau o metodă de interfață.

CONSTANT_MethodType

Corespunde valorii etichetei de 16 și conține un indice către o constantă `CONSTANT_Utf8`, ce reprezintă tipul unei metode.

CONSTANT_InvokeDynamic

Corespunde valorii etichetei de 18 și este folosit de către `JVM` pentru a invoca o metodă polimorfică.

În cod C++, am reprezentat cp_info astfel:

```
struct cp_info {
    enum class Tag : u1 {
        CONSTANT_Class = 7,
        CONSTANT_Fieldref = 9,
        CONSTANT_Methodref = 10,
        CONSTANT_InterfaceMethodref = 11,
        CONSTANT_String = 8,
        CONSTANT_Integer = 3,
        CONSTANT_Float = 4,
        CONSTANT_Long = 5,
        CONSTANT_Double = 6,
        CONSTANT_NameAndType = 12,
        CONSTANT_Utf8_info = 1,
        CONSTANT_MethodHandle = 15,
        CONSTANT_MethodType = 16,
        CONSTANT_InvokeDynamic = 18,
    };

    Tag tag;
    std::vector<u1> data;
};
```

Iar structurile folosite pentru obiectivul propus au fost reprezentate astfel [6]:

```
struct CONSTANT_Methodref_info {
    cp_info::Tag tag;
    u2 class_index;
    u2 name_and_type_index;
};
struct CONSTANT_Class_info {
    cp_info::Tag tag;
    u2 name_index;
};
struct CONSTANT_NameAndType_info {
    cp_info::Tag tag;
    u2 name_index;
    u2 descriptor_index;
};
```

`field_info` Fiecare câmp din cadrul unei clase este reprezentat printr-o structură de tipul `field_info`.

În cod C++, această structură a fost reprezentată astfel [6]:

```
struct field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    std::vector<attribute_info> attributes;
};
```

Unde:

- `name_index` este o intrare în tabela de constante unde se afla o constantă de tipul `CONSTANT_Utf8`.
- `descriptor_index` arată spre o constantă de tipul `CONSTANT_Utf8` și reprezintă tipul câmpului.

`method_info` Fiecare metodă a unei clase/interfețe este descrisă prin această structură.

În cod C++, am implementat-o astfel [6]:

```
struct method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    std::vector<attribute_info> attributes;
};
```

Unde `name_index` și `descriptor_index` au aceeași interpretare ca și la `field_info`.

Dacă metoda nu este abstractă, atunci în vectorul `attributes` se va găsi un atribut de tipul `Code`, care conține bytecode-ul corespunzător acestei metode.

`attribute_info` În C++, a fost implementată astfel [6]:

```
struct attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    std::vector<u1> info;
};
```

Numele atributului determină modul în care octeții din vectorul `info` sunt interpretați.

Atributul de cod

Pentru intențiile noastre, atributul de interes este cel de cod [6]:

```
struct Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;

    u2 max_stack;
    u2 max_locals;

    u4 code_length;
    std::vector<u1> code;

    u2 exception_table_length;
    struct exception {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    };
    std::vector<exception> exception_table; // of
       length exception_table_length.
    u2 attributes_count;
    std::vector<attribute_info> attributes; // of
       length attributes_count.
};
```

Această structură este esențială pentru implementarea optimizării, întrucât ea ne permite să determinăm metodele apelate din cadrul unei secvente de cod. În continuare, o voi descrie detaliat:

- `max_stack`: Reprezintă adâncimea maximă a stivei mașinii virtuale când această bucată de cod este interpretată.
- `max_locals`: Reprezintă numărul maxim de variabile locale alocate în același timp când această bucată de cod este interpretată.
- `code`: Codul metodei.
- `exception_table`: Excepțiile pe care le poate arunca metoda.

Code

Vectorul `code` din cadrul atributului `Code` reprezintă bytecode-ul propriu-zis al metodei.

Acest vector conține instrucțiunile care sunt executate de către mașina virtuală.

JVM-ul rulează ca o mașina cu stivă, iar toate instrucțiunile operează pe această stivă. Rezultatul rulării unei instrucțiuni este modificarea stivei: scoaterea și adăugarea de elemente în vraful acesteia.

Instrucțiunile au în general formatul [7]

```
nume_instr
operand1
operand2
...
```

Cu un număr variabil de operanzi, prezenți în mod explicit în vectorul de `cod`.

Fiecărei instrucțiuni îi corespunde un octet, denumit opcode. Fiecare operand este fie cunoscut la compilare, fie calculat în mod dinamic la rulare.

Cele mai multe operații nu au niciun operand dat în mod explicit la nivelul instrucțiunii — ele lucrează doar cu valorile din vârful stivei la momentul execuției codului.

De exemplu:

Instrucțiunea `imul` are octetul 104 sau 0x68. Aceasta extrage două valori din vârful stivei: `value1` și `value2`. Ambele valori trebuie să fie de tipul `int`. Rezultatul este înmulțirea celor două valori: `result = value1 * value2`, și este pus în vârful stivei.

Dintre cele peste o sută de instrucțiuni, noi suntem preocupați doar de 5 dintre acestea: cele care au de a face cu invocarea unei metode.

invokedynamic

Format:

```
invokedynamic
index1
index2
0
0
```

Opcode-ul corespunzător acestei instrucțiuni este 186 sau 0xba [7].

Index1 și index2 sunt doi octeți. Aceștia sunt compuși în

```
index = (index1 << 8) | index2
```

Unde << reprezintă shiftare pe biți, iar | reprezintă operația de sau pe biți.

Indicele compus reprezintă o intrare în tabela de constante. La locația respectivă trebuie să se afle o structură de tipul `CONSTANT_MethodHandle`

invokeinterface

Format:

```
invokeinterface
index1
index2
count
0
```

Opcode-ul corespunzător este 185 sau 0xb9 [7]. `index1` și `index2` sunt folosiți, în mod similar ca la `invokedynamic`, pentru a construi un indice în tabela de constante.

La poziția respectivă în tabelă, trebuie să se găsească o structură de tipul `CONSTANT_Methodref`.

`count` trebuie să fie un octet fără semn diferit de 0. Acest operand descrie numărul argumentelor metodei, și este necesar din motive istorice (aceasta informație poate fi dedusa din tipul metodei).

invokespecial

Format:

```
invokespecial
index1
index2
```

Opcode-ul corespunzător este 183 sau 0xb7 [7]. La fel ca la `invokeinterface`, instrucțiunea conține un indice în tabela de constante către o structură `CONSTANT_Methodref`.

Această instrucțiune este folosită pentru a invoca constructorii claselor.

invokestatic

Format:

```
invokestatic
index1
index1
```

Opcode-ul corespunzător este 184 sau 0xb8 [7]. Instrucțiunea invocă o metodă statică a unei clase.

La fel ca la `invokeinterface`, este construit un indice compus, și folosit pentru a indexa tabela de constante.

invokevirtual

Format:

```
invokevirtual
index1
index1
```

Opcode-ul corespunzător este 182 sau 0xb6 [7], iar interpretarea este la fel ca la `invokeinterface`. Aceasta este cea mai comună instrucțiune de invocare de funcții.

În C++, am reprezentat aceste instrucțiuni de interes astfel:

```
enum class Instr {
    invokedynamic = 0xba,
    invokeinterface = 0xb9,
    invokespecial = 0xb7,
    invokestatic = 0xb8,
    invokevirtual = 0xb6,
};
```

4.6.4 ClassFile

Folosind definițiile anterioare, putem descrie un fișier de clasă binar în C++:

```
struct ClassFile {
    u4 magic; // Should be 0xCAFEBABE.

    u2 minor_version;
    u2 major_version;

    u2 constant_pool_count;
    std::vector<cp_info> constant_pool;

    u2 access_flags;

    u2 this_class;
    u2 super_class;
```

```
    u2 interface_count;
    std::vector<interface_info> interfaces;

    u2 field_count;
    std::vector<field_info> fields;

    u2 method_count;
    std::vector<method_info> methods;

    u2 attribute_count;
    std::vector<attribute_info> attributes;
};
```

Pentru a vedea un fișier clasa analizat în detaliu, va puteți uita la appendix-ul Studiu de Caz .1.

Capitolul 5

Limbajul Java și optimizarea acestuia

În acest capitol vom descrie cum putem adapta optimizarea eliminării de metode pentru limbajul Java. Scopul final este să implementăm algoritmul descris la sfârșitul capitolului 'Optimizări de dimensiune' 3.

5.1 Optimizarea limbajului Java

5.1.1 Preliminarii

Java este un limbaj dinamic. Acest lucru înseamnă posibilitatea de a modifica codul în timpul rulării, sau de a apela cod în mod dinamic la rulare: aceasta este partea de 'reflecție' a limbajului Java.

Utilizarea acestor caracteristici face abordarea noastră de analiză statică imposibilă, întrucât optimizatorul ar putea elimina metode care nu sunt apelate în mod explicit în cod, însă care ajung să fie referențiale în mod dinamic la rulare.

Prin urmare, în continuarea lucrării vom presupune că toate proiectele Java cu care lucrăm nu se folosesc de niciun mod de a schimba structura codului în timpul rulării. În caz contrar, optimizatorul nu mai oferă nicio garanție asupra corectitudinii optimizărilor realizate.

Restul lucrării va presupune că această condiție este respectată.

5.1.2 Determinarea punctului de intrare

Un proiect Java este format dintr-o mulțime de clase. Punctul `main` al proiectului este reprezentat de funcția denumită `main` [8], cu antetul

```
public static void main(String [] args)
```

Într-un proiect trebuie să existe o singură astfel de metodă, care să aparțină unei clase a proiectului.

Pentru a găsi metoda `main`, sunt scanate toate fișierele clasa care fac parte din proiect, și sunt analizate listele de metode ale acestora.

Acesta este pseudocodul pentru această operație:

```
def main_method(p: Proiect):
    ret = None
    for classfile in p.classfiles:
        if "main" in p.methods():
            if ret:
                assert False, "Am gasit mai multe
                   metode main!".
            ret = p.method_of_name("main")
    if not ret:
        assert False, "Proiectul trebuie sa aiba o
           metoda main!".
    return ret
```

5.1.3 Găsirea apelurilor de funcții

Pentru a determina ce funcții sunt apelate din cadrul unei metode `m`, vom inspecta atributul de cod al lui `m`. Dintre instrucțiunile conținute vom fi interesați doar de cele ce implică apeluri de funcții - familia invoke*.

Odată găsite instrucțiunile de invocare de funcții, este necesar să rezolvăm referințele conținute de acestea, întrucât în reprezentarea interna a JVM-ului, o metodă este referențială pur simbolic, prin șiruri de caractere.

5.1.4 Rezolvarea metodelor speciale

Acest tip de rezolvare a metodelor este necesar atunci când întâmpinam instrucțiunea **invokespecial**.

Prin metode speciale înțelegem funcții precum constructorul unei clase sau funcții private.

Pentru scopul acestei lucrări, nu vom elimina constructorii, întrucât asta ar înseamna eliminarea unei clase cu totul (i.e., singurul caz în care putem elimina constructorul unei clase este când clasa nu este instanțiată niciodată).

Metodele private, în schimb, vor fi tratate ca niște funcții normale.

5.1.5 Rezolvarea metodelor dinamice

Din versiunea 7 a limbajului Java a fost introdusă instrucțiunea **invokedynamic**. Aceasta instrucțiune este folosită în rezolvarea referințelor de

metode la rulare - similar cu conceptul de ”duck typing” din limbajele dinamice. Deoarece am făcut presupunerea că proiectele cu care lucrăm nu vor utiliza reflecția sau invocarea dinamică a codului, această instrucțiune nu va fi întâlnită.

5.1.6 Rezolvarea metodelor statice

Metodele statice sunt cele mai simple de rezolvat: referința către o astfel de metodă indică numele metodei, tipul metodei, cât și clasa din care face parte și de unde este invocată. Aceste metode sunt rezolvate direct la compilare, deci tuplul (nume, tip, clasă) identifică în mod unic o metodă statică.

5.1.7 Rezolvarea metodelor virtuale

În Java, toate metodele de instanță (non-statice) sunt polimorfice la rulare (Eng. Run time polymorphism). Cu alte cuvinte, la compilare se știu numele, tipul metodei și clasa unde metoda este definită, însă **nu** și clasa de unde este invocată.

Aceasta este cea mai comună operație, și corespunde instrucțiunii **invokevirtual**.

De exemplu, pentru programul

```
1  public class Main {
2      static private class One extends Other {
3          public void foo() {
4              System.out.println("foo() of One");
5          }
6      }
7
8      public static void bar(Other o) {
9          o.foo();
10     }
11
12     public static void main(String[] args) {
13         Other o = new One();
14         bar(o);
15     }
16 }
17
18 public class Other {
19     public void foo() {
20         System.out.println("foo() of Other");
21     }
22 }
```

format din concatenarea fișierelor din testul `fixtures/project4`, apelul de pe linia 9 către metoda foo este un apel polimorfic.

În codul de JVM, instrucțiunea corespunzătoare este:

```
invokevirtual #2 // Method Other.foo:()V
```

Unde poziția 2 din tabela de constante este o referință către o metodă cu numele de `foo` cu tipul `void foo()`, definită în clasa `Other`.

```
#2 = Methodref #22.#23 // Other.foo:()V
```

Deși metoda `foo` este definită inițial în clasa `Other`, la rulare va fi apelată versiunea definită în subclasa `One`.

În implementarea JVM-ului, în vârful stivei mașinii se va afla o referință către instanța obiectului a cărui metodă este apelată. În cazul programului nostru, pe stivă se va afla o referință către variabila `o`, de tipul `Other`, definită pe linia 13 a programului.

Pentru rezolvarea metodei virtuale, JVM-ul se asigură că tipul lui `o`, adică `Other`, este un moștenitor al clasei de care aparține metoda `One`. În caz că tipul `Other` definește chiar el metoda căutată, mașina va folosi definiția respectivă. În caz contrar, mașina virtuală va cauta recursiv metoda în super clasa lui `Other`. În cazul în care căutarea recursivă a ajuns până la o clasa fără super (singura astfel de clasa este `Object`), mașina virtuală va emite o eroare.

5.1.8 Rezolvarea metodelor de interfața

Ultimul tip de invocare a metodelor corespunde instrucțiunii **invokeinterface**, și corespunde apelării unei funcții declarate în cadrul unei interfețe.

De exemplu, pentru programul

```
1  public interface I {
2      public void foo();
3  }
4  public class Main {
5      static private class C implements I{
6          public void foo() {
7              System.out.println("foo() of C");
8          }
9      }
10
11     public static void bar(I i) {
12         i.foo();
13     }
14
```

```
15      public static void main(String[] args) {
16          I i = new C();
17          bar(i);
18      }
19  }
```

format din fișierele testului `fixtures/project5`, apelul de pe linia 11 către metoda foo este un apel polimorfic pe interfețe.

În codul de JVM, instrucțiunea corespunzătoare este:

```
invokeinterface #2, 1 // InterfaceMethod I.foo:()V
```

Unde slotul 2 din tabela de constante este o referința către o metoda de interfața cu numele de `foo` cu tipul `void foo()`, definită în interfața `I`. `#2 = InterfaceMethodref #22.#23 // I.foo:()V`

Deși metoda `foo` este declarată inițial în interfața `I`, la rulare va fi apelată versiunea definită în subclasa `C`.

Pentru rezolvarea referințelor către metode de interfață, se va aplica un algoritm similar cu cel de la rezolvarea referințelor de metode virtuale: căutare recursivă în lanțul de moșteniri al clasei asupra căruia se execută metoda.

5.1.9 Algoritmul de rezolvare al referințelor metodelor

Putem defini în pseudo cod algoritmul de rezolvare al referințelor către metode virtuale și către metode de interfața astfel:

```
def from_symbolic_reference(
    class, method_name, method_type) -> Method:
    if class is Object:
        return None
    for m in class.methods():
        if m.name == method_name and m.type ==
           method_type:
            return m
# Nu am putut sa rezolvam referinta in clasa curenta,
# incercam in super.
    return resolve_reference(
        class.super_class, method_name, method_type)
```

Metoda exactă care este executată de o astfel de instrucțiune nu poate fi știută decât abia la timpul rulării (e.g., pentru programul 5.1.7 nu putem știi în timpul analizei statice dacă va fi apelată metoda `Other::foo()` sau metoda `One::foo()`).

5.1.10 Supersirul de execuție și polimorfismul de execuție

Vom reconsidera supersirul de execuție definit la 3.5.2 și graful apelurilor, definit la 3.5.3.

La momentul analizei statice nu știm tipul exact al unei referințe. O referință către o interfață poate să conțină orice obiect care implementează acea interfață, sau o referință către o clasă $\mathcal{C}$ poate conține fix clasa $\mathcal{C}$, sau orice clasă care o are pe $\mathcal{C}$ ca strămoș. , care determină ce funcții Prin urmare, vom extinde supersirul de execuție ca să reflecte aceste posibilități. Mai precis, pentru fiecare apel de funcții polimorfice (virtuale prin **invokevirtual** sau de interfață prin **invokeinterface**), vom considera toate metodele posibile care pot fi referențiate.

De exemplu, pentru programul 5.1.7, vom considera toate posibilitățile de rezolvare ale apelului către foo(), atât `Other::foo()`, cat și `One::foo()`.

5.1.11 Graful de apeluri și polimorfismul de execuție

Acum că am adaptat supersirul de execuție pentru polimorfismul de execuție, este nevoie să extindem și graful de apeluri.

Pentru acest lucru, în cadrul unei rezoluții vom considera nu numai metoda direct referențială, ci toate metodele pe care supersirul de execuție le-ar putea rezolva ca posibili candidați.

Vom spune ca metoda m este un candidat pentru rezoluția metodei q dacă în rezolvarea unei referințe către q, supersirul de execuție va considera și metoda m.

Proprietatea de a fi candidat este o relație reflexivă și tranzitivă, însa nu și simetrică.

Această relație este echivalentă cu: metoda m este un candidat pentru rezoluția metodei q dacă:

1. q este o metodă a unei interfețe, iar m implementează direct sau tranzitiv interfața respectivă.
2. q este o metodă a unei clase, iar m moștenește direct sau tranzitiv clasa respectivă.

În continuare, vom nota cu $cand(m)$ tot candidații lui m. Din faptul ca este o relație reflexiva, m aparține mulțimii $cand(m)$.

Graful de apeluri trebuie schimbat astfel încât să includă și candidații unei metode: în loc să tragem o singură muchie de la metoda p care face invocarea, la metoda m care este invocată, vom trage mai multe muchii: de la p la toate metodele candidate pentru m — $cand(m)$.

5.1.12 Noul algoritm pentru eliminarea funcțiilor

În această parte, vom adapta algoritmul de eliminare a funcțiilor, definit anterior la 3.5.4. În particular, vom actualiza modul de calculare al funcțiilor accesibile:

```
def reachable_methods_in_java(main: Method) -> [
   Method]:
    coada = [main]
    at = 0
    while at < size(coada):
        m = coada[at]
        at = at + 1
        for next in direct_calees(m):
            for c in cand(next):
                if c not in coada:
                    coada.push(c)
    return coada
```

Iar procedura de optimizare va rămâne identică:

```
def optimize_for_size_in_java(p: Program) -> Program:
    main = p.main_method()
    used_methods = reachable_methods(main)
    for m in p.all_methods():
        if m not in used_methods:
            p.remove_method(m)
    return p
```

5.2 Implementarea în C++

În această secțiune, voi face legătura între algoritmii în pseudocod descriși anterior și implementarea în C++. Codul sursă este disponibil pe GitHub [20] este documentat, cu comentarii, iar programul Doxygen poate fi folosit pentru a genera documentație pentru proiect. În continuare, voi descrie legătura dintre implementarea în C++, și algoritmii în pseudocod descriși anterior.

Partea relevantă a proiectului pentru problema optimizării este conținută în două clase: Method și Project.

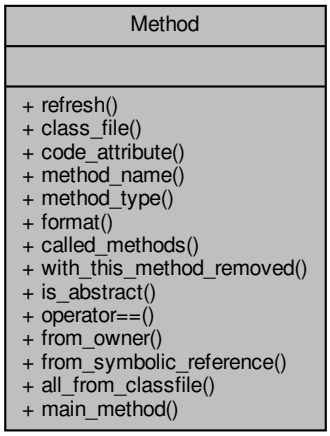

Figura 5.1: Clasa Method

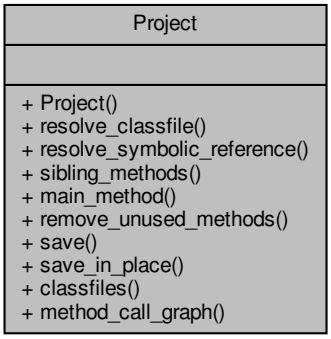

Figura 5.2: Clasa Project

Clasa Project se ocupă de operațiile la nivelul întregului proiect - rezolvarea metodelor simbolice și optimizarea de dimensiune.

Clasa Method corespunde unei singure metode dintr-o clasă.

Metodele care nu sunt descrise în această parte nu au fost considerate suficient de relevante pentru a fi incluse în teză, iar documentația pentru ele se

poate găsi inline în codul sursă, cât și în documentația generată de Doxygen.

Funcția main_method 5.1.2, care determină metoda principală (main entry-point) al proiectului, are antetul

```
Method Project::main_method() const;
```

Funcția cand 5.1.11, care determină ce funcțiile candidat, este reprezentată de funcția

```
std::vector<Method> Project::sibling_methods(const
    Method& m) const;
```

denumită astfel deoarece am considerat că o metodele candidat sunt înrudite între ele.

Procedura reachable_methods_in_java 5.1.12 pentru determinarea funcțiilor accesibile este reprezentată de

```
std::vector<Method> Project::method_call_graph(const
    Method& m) const;
```

denumită astfel pentru a evidenția faptul că mulțimea returnată sunt nodurile unui graf.

Iar procedura principală optimize_for_size_in_java **??** este reprezentată de:

```
void Project::remove_unused_methods();
```

Procedura from_symbolic_reference 5.1.9, folosită pentru a rezolva referințele simbolice către metode, este reprezentată de

```
static std::optional<Method>
from_symbolic_reference(const ClassFile &file, int
    cp_index, cp_info info);
```

Motivul pentru care această funcție întoarce un `optional`, în loc de direct metoda, este că există referințe către funcții terțe, de obicei definite în librării, care nu pot fi rezolvate în clasele proiectului nostru. De exemplu, funcția `println(String s)`, definită în librăria standard `java.io`.

Capitolul 6

Aplicație și concluzii

În acest capitol voi prezenta aplicația și concluziile lucrării.

6.1 Aplicația

Programul este implementat în limbajul C++. Acesta funcționează în mod corect: formează graful de apelări ale metodelor și elimină metodele nefolosite.

6.1.1 Rulare

Programul vine cu opțiunea –help, care specifică modul de utilizare.

```
$ thesis --help
JVM Optimizer
Usage: thesis [OPTIONS] classfiles...

Positionals:
  classfiles [File] ... (REQUIRED)
                                All of the class files
                                   from the project

Options:
  -h,--help                     Print this help message
      and exit
  --classfiles [File] ... (REQUIRED)
                                All of the class files
                                   from the project
  --out Dir Excludes: --in-place
```

```
                                  Where to save the
                                     modified class files
  --in-place Excludes: --out      Whether to save the
     class files in-place
```

Utilizarea normală ar arăta așa:

```
$ thesis *.class --in-place
Done parsing constants.
...
Found the main file and method.
...
Trying to resolve method from symbolic reference: (
   bar) and type (LMain$One;)V.
Trying to resolve_classfile for Main
Successfully resolved bar :: (LMain$One;)V
...
Main/bar :: (LMain$One;)V and Other/foo :: ()V are
   not sibling methods
Found the following sibling methods for Main/bar :: (
   LMain$One;)V: Main/bar :: (LMain$One;)V;
...
Successfully resolved foo :: ()V
...
Found 1 method(s) to remove from class Other
Removing Other/foo :: ()V...
Done putting constants.
...
Done putting attributs.
```

6.2 Testare

Testarea aplicației este realizată automat, cu teste scrise de mână. Aceste teste simulează atât cazuri particulare, cât și funcționarea uzuală a programului.

Teste pot fi rulate prin programul `test.py`, din rădăcina proiectului:

```
$ ./test.py
Building...
Running on test/fixtures/project2
...
Running on test/fixtures/project3
Testing on /home/ericpts/work/ug-thesis/test/fixtures
   /TAP/tema4/var33...
```

```
...
Testing on /home/ericpts/work/ug-thesis/test/fixtures
   /TAP/tema3/Var3Munte...
```

6.3 Concluzii

Scopul lucrării a fost de a prezenta o optimizare pentru dimensiunea programelor - eliminarea funcțiilor nefolosite. Prima parte a ei a implicat formalizarea problemei, definirea unui algoritm generic pentru optimizări de dimensiune și demonstrația corectitudinii acestuia.

În a doua parte, am prezentat cum algoritmul poate fi implementat în limbajul Java, și am analizat condițiile necesare pentru a putea afirma faptul că algoritmul poate fi aplicat în mod corect.

În concluzie, implementarea acestui fel de optimizări este un procedeu complex, cu multe greutăți de implementare. De asemenea, optimizările prezintă și o sporită dificultate teoretică, în a identifica asumpțiile făcute de către optimizator, a demonstra suficiența acestora, și a le prezenta utilizatorilor programului într-un mod ușor de înțeles.

6.4 Îmbunătățiri propuse

6.4.1 Testarea

Testarea programului ar putea fi îmbunătățită extinzând suita de teste, cu mai multe cazuri și situații.

Deși motivația inițială a constat în proiectele de Android, acestea au o structură complexă și nu au putut fi testate (întrucât efortul necesar pentru a le parsa programatic este ridicat). O adiție bună ar fi adaptarea proiectului pentru a trata și proiectele Android.

De asemenea, programul ar putea fi pus să optimizeze proiecte mai complexe, pentru verificarea corectitudinii.

6.4.2 Optimizarea implementării

Programul a fost implementat pe principiul de a fi funcțional: (aproape) toate funcțiile și metodele întorc obiecte noi, în loc să le modifice pe cele existente.

Acest lucru face implementarea mai clară și mai ușor de urmărit, însă aduce penalizări destul de mare la timpul de rulare, întrucât limbajul C++ nu este gândit pentru a fi utilizat într-un mod funcțional. De asemenea, mulți algo-

ritmi implementați în program au complexitatea *O* sub optimă, deoarece au fost implementați în cel mai simplu mod, nu neapărat cel mai rapid.

.1 Appendix - Studiu de caz

In continuare, voi exemplica structura unui fisier clasa cu un exemplu.

Codul Java este urmatorul:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("project1 - hello world");
        foo();
    }

    public static void foo() {
        System.out.println("project1 - foo()");
    }
}
```

Compilatorul folosit este `openjdk-11`. Clasa a fost utilizata folosind utilitarul `javap` [9], care este de asemenea inclus in pachetul `openjdk-11`.

In primul rand, tabela de constante:

```
Constant pool:
   #1 = Methodref          #8.#18         // java/
      lang/Object."<init>":()V
   #2 = Fieldref           #19.#20        // java/
      lang/System.out:Ljava/io/PrintStream;
   #3 = String             #21            // project1
       - hello world
   #4 = Methodref          #22.#23        // java/io/
      PrintStream.println:(Ljava/lang/String;)V
   #5 = Methodref          #7.#24         // Main.foo
      :()V
   #6 = String             #25            // project1
       - foo()
   #7 = Class              #26            // Main
   #8 = Class              #27            // java/
      lang/Object
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Utf8               LineNumberTable
  #13 = Utf8               main
```

```
  #14 = Utf8               ([Ljava/lang/String;)V
  #15 = Utf8               foo
  #16 = Utf8               SourceFile
  #17 = Utf8               Main.java
  #18 = NameAndType        #9:#10          // "<init
     >":()V
  #19 = Class              #28             // java/
     lang/System
  #20 = NameAndType        #29:#30         // out:
     Ljava/io/PrintStream;
  #21 = Utf8               project1 - hello world
  #22 = Class              #31             // java/io/
     PrintStream
  #23 = NameAndType        #32:#33         // println
     :(Ljava/lang/String;)V
  #24 = NameAndType        #15:#10         // foo:()V
  #25 = Utf8               project1 - foo()
  #26 = Utf8               Main
  #27 = Utf8               java/lang/Object
  #28 = Utf8               java/lang/System
  #29 = Utf8               out
  #30 = Utf8               Ljava/io/PrintStream;
  #31 = Utf8               java/io/PrintStream
  #32 = Utf8               println
  #33 = Utf8               (Ljava/lang/String;)V
```

In acest format, namespace-urile imbricate sunt reprezentate prin /.

Informatii despre clasa:

```
Classfile Main.class
  Last modified May 28, 2018; size 520 bytes
  MD5 checksum 248b729dfe4b4bc8da895944d30fdc28
  Compiled from "Main.java"
public class Main
  minor version: 0
  major version: 55
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #7                          // Main
  super_class: #8                         // java/
     lang/Object
  interfaces: 0, fields: 0, methods: 3, attributes: 1
```

Constructorul clasei:

```
{
```

```
  public Main();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  //
            Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0
```

Metoda `main(String[] args)`:

```
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     #2                  // Field
        java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #3                  // String
        project1 - hello world
     5: invokevirtual #4                  // Method
        java/io/PrintStream.println:(Ljava/lang/
        String;)V
     8: invokestatic  #5                  // Method
        foo:()V
    11: return
  LineNumberTable:
    line 3: 0
    line 4: 8
    line 5: 11
```

Metoda `foo()`:

```
public static void foo();
descriptor: ()V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=0, args_size=0
     0: getstatic     #2                  // Field
        java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #6                  // String
        project1 - foo()
```

```
    5: invokevirtual #4                  // Method
       java/io/PrintStream.println:(Ljava/lang/
       String;)V
    8: return
LineNumberTable:
  line 8: 0
  line 9: 8
```

Bibliografie

[1] https://en.cppreference.com/w/c/language/initialization.

[2] https://en.cppreference.com/w/cpp/language/initialization.

[3] http://www.comscigate.com/tutorial/ajay/Part%202/Development%20for%20Mobile%20Devices/Software/proguard2.1/proguard2.1/docs/alternatives.html.

[4] http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html.

[5] https://github.com/trizen/language-benchmarks.

[6] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html.

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

[8] https://docs.oracle.com/javase/tutorial/getStarted/application/index.html#MAIN.

[9] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html.

[10] Matthew Arnold, Stephen Fink, David Grove, and Michael Hind. Dynamic compilation and adaptive optimization in virtual machines. 2006.

[11] L Burkholder. The halting problem. *SIGACT News*, 18(3):48–60, April 1987.

[12] Colin Gillespie. CPU and GPU trends over time. https://csgillespie.wordpress.com/2011/01/25/cpu-and-gpu-trends-over-time/.

[13] K. Goda and M. Kitsuregawa. The history of storage systems. *Proceedings of the IEEE*, 100(Special Centennial Issue):1433–1440, May 2012.

[14] Goole. Deadcode elimination. https://developers.google.com/j2objc/guides/dead-code-elimination. [Online; accessed 4-June-2018].

[15] Guard Square. ProGuard. https://www.guardsquare.com/en/proguard, 2016. [Online; accessed 4-June-2018].

[16] Eljay Love-Jensen. Removing unused functions/dead code. https://gcc.gnu.org/ml/gcc-help/2003-08/msg00128.html.

[17] Oracle. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/multiple-language-support.html.

[18] Rob Sayre. Dead code elimination for beginners. http://chris.improbable.org/2010/11/17/dead-code-elimination-for-beginners/, 2010. [Online; accessed 4-June-2018].

[19] Robert R. Schaller. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[20] Eric Stavarache. http://github.com/ericpts/ug-thesis/.

[21] Wikipedia contributors. Dead code elimination — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Dead_code_elimination&oldid=841070703, 2018. [Online; accessed 4-June-2018].