ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Un optimizator de dimensiune pentru Java

Teza de Licenta

Petru-Eric Stavarache

4 Iunie, 2018

Coordonator Prof. Dr. Traian-Florin Serbanuta

Facultatea de Matematica si Informatica, UNIBUC

Abstract

Sistemul de operare Android este cea mai populara platforma 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 statica 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.

Contents

Chapter 1

Introducere

Eliminarea codului inutil (eng. "Dead code elimination") [11] 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 [9].

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 sursa 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 sa aibă, înaintea eliminării codului mort, câteva mii de linii și sute

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

1.1 Eliminarea metodelor

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

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

Această lucrare va explora aceasta optimizare particulară, atât teoretic, cât și implementat in limbajul Java.

1.2 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 a 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.

Chapter 2

Problema de optimizare

2.1 Formalizarea problemei

Fie programul $\mathcal{P}$ un proiect Java, format dintr-o mulțime de fișiere clasa. Scopul optimizatorului este sa creeze un program $\mathcal{P}'$, care sa se comporte identic cu $\mathcal{P}$, și sa fie mai bun decât $\mathcal{P}$ pentru o anumita 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, in sensul pur matematic, cat 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

In contextul optimizării de programe este nevoie sa definim ce înseamnă ca dintre doua programe echivalente $\mathcal{P}$ și $\mathcal{Q}$, $\mathcal{P}$ sa fie mai performant decât $\mathcal{Q}$. Cele mai folosite doua metrici sunt metrica de viteza 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, pana când acesta își termina 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
```

In 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 $\mathcal{P}$ input-ul $\mathcal{I}$.

Comparare bazata pe timpul de rulare

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

Vom spune ca $\mathcal{P}$ 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 definita relația de ordine, problema optimizării pe baza metricii de viteza este:

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

$$\underset{\mathcal{Q}}{\operatorname{argmin}}\, \mathcal{P} \text{ echivalent cu } \mathcal{Q}$$

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* ocupa 5 octeți, in timp ce instrucțiunea *dmul* ocupa un singur octet.

Comparare bazata 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 viteza:

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

$$\underset{\mathcal{Q}}{\operatorname{argmin}}\, \mathcal{P} \text{ echivalent cu } \mathcal{Q}$$

2.4 Discuție asupra metricilor

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

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

Programul

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

Va fi optimizat pentru viteza in

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

În timp ce aceasta optimizare va creste 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 ocupa de 2 ori mai puțin.

Acest lucru se datorează faptului ca performanta memoriei (prețul per unitate de memorie) a continuat sa scadă in ultimul deceniu, in timp ce performanta procesoarelor (numărul de instrucțiuni executate per secunda) a stagnat.

Asa cum se poate observa in figura 2.1, trendul care urma legea lui Moore [10] a început sa se oprească. Pe de alta parte, eficienta memoriei calculatoarelor și-a continuat trendul de creștere, asa cum se poate observa in continuat sa crească, asa cum se poate observa in figura 2.2.

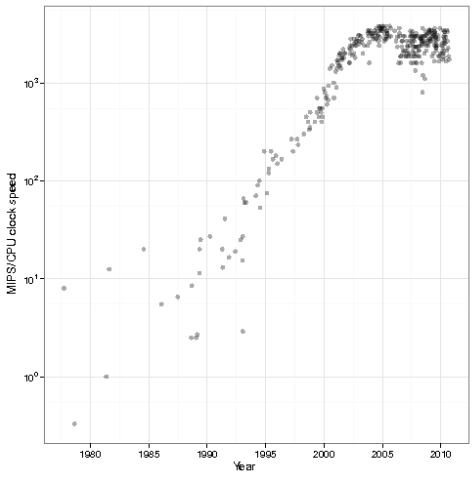

Figure 2.1: Evoluția puterii de procesare[7]

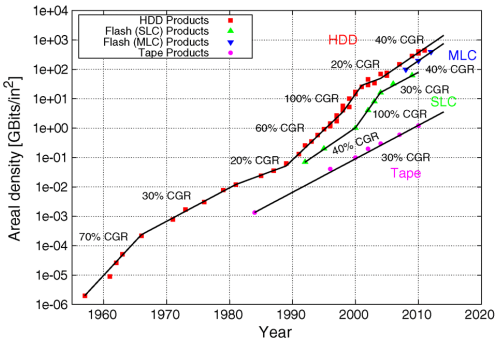

Figure 2.2: Evoluția puterii de procesare[8]

Chapter 3

Optimizari de dimensiune

3.1 Eliminarea functiilor nefolosite

Aceasta lucrare contine o abordare de optimizare a dimensiunii programelor bazata pe eliminarea functiilor nefolosite.

Desi metrica principala este aceea de dimensiune, reducerea numarului de instructiuni ale programului poate avea si beneficii asupra vitezei de executie, din cauza interactiunilor cu cache-ul calculatoarelor. Pe de alta parte, acest beneficiu este destul de minor, si nu preprezinta principala motivatie.

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

3.2 Functie/Metoda

In domeniul limbajelor de programare, o functie $\mathcal{F}$ este formata dintr-o colectie de instructiuni tratate ca un întreg, impreuna cu un protocol pentru executarea acelor instructiuni.

O metoda $\mathcal{M}$ reprezinta o functie asociata unei clase.

In limbajul Java este imposibil sa avem o functie care sa nu apartina unei metode. Desi definitiile nu sunt echivalente, deoarece in Java nu exista functii propriu-zise, ci doar metode, termenii de functie si de metoda sunt considerati interschimbabili.

Protocolul de executare a functiilor variaza de la limbaj la limbaj si nu este important pentru scopurile noastre.

De exemplu, functia f

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

```
2       return c
```

contine 2 instructiuni, corespunzatoare liniilor 1-2:

(i_1) $a = a + b$

(i_2) *return* c

Desi in acest exemplu functia $\mathcal{F}$ incepe de pe pozitia 0, intr-un program format din mai multe functii locul de inceput al functiei variaza (indicele primei instructiuni).

Vom spune ca doua instructiuni sunt egale daca se afla pe aceeasi linie. i.e., reprezinta aceeasi pozitie in 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
```

instructiunile i_1 si i_5 **nu** sunt egale.

Vom spune ca o instructiune **i** apartine functiei $\mathcal{F}$ ddaca exista o linie a lui $\mathcal{F}$ unde putem gasi instructiunea respectiva.

3.3 Punctul de intrare principal

Fie $\mathcal{P}$ un program, executat in ordine secventiala. Punctul principal de intarare al lui $\mathcal{P}$ reprezinta locul de unde incepe executia programului – prima instructiune executata.

Acest loc va fi notat cu `main`.

3.4 Sirul de executie al unui program

Un sir de executie $\mathcal{S}$ al programului $\mathcal{P}$ reprezinta o inaltuire finita de instructiuni executate secvential, incepand cu `main` si terminandu-se cu ultima instructiune executata de catre $\mathcal{P}$ inainte ca programul sa se termine. Pentru ca un sir de executie $\mathcal{S}$ sa fie valid, trebuie sa existe un input pe care, daca rulam programul, acesta sa execute fix instructiunile lui $\mathcal{S}$.

3.5 Functii nefolosite

O functie a unei clase poate fi eliminata daca nu exista niciun sir de executie valid al programului care sa treaca prin acea functie.

Vom nota cu $elim(\mathcal{F}) = 1$ daca functia $\mathcal{F}$ poate fi eliminata deoarece este nefolosita.

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

Pentru oricare $\mathcal{S}$ sir de executie, nu exista **i** din $\mathcal{S}$ care sa faca parte din corpul functiei $\mathcal{F}$.

3.5.1 Sirurile de executie - in practica

Problema determinarii tuturor sirurilor de executie ale unui program este o problema grea, intrucat aceasta ar implica rularea programului pe fiecare input posibil.

Din cauza ca unele programe pot sa fie definite doar partial (comportamentul lor sa fie nedefinit pe anumite clase de input), aceasta problema poate fi echivalenta cu Problema Opririi (eng. "The Halting Problem") [6]: nu avem cum sa stim daca un input al programului este bun sau nu fara sa rulam programul, insa nu putem determina daca un program se va termina vreodata.

Deoarece problema opririi este intractabila, determinarea tuturor sirurilor de executie posibile ale lui program arbitrar este de asemenea o problema intractabila.

3.5.2 Supersirul de executie

Aceasta lucrare va construi o aproximare – un "supersir" de executie format dintr-o superpozitie a tuturor sirurilor de executie existente, inclusiv pe cele invalide (pentru care nu exista un input care sa 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
```

sirul de executie folosit in problema noastra va contine si secventa corespunzatoare lantului de apeluri $main() -> f()$, cat si lantului $main() -> f() -> g()$, chiar daca acestea sunt invalide.

Totodata, sirul de executie generat **nu** va contine secventa $main() -> h()$, sau orice secventa care sa il contina pe h, din cauza ca aceste secvente nu sunt siruri de executie (nu exista nicio secventa de instructiuni secventiale care sa porneasca din functia `main` si sa ajunga in functia h).

3.5.3 Graful apelurilor de functii

Vom construi un graf $\mathcal{G}$ al metodelor: fiecarui nod ii va corespunde o metoda prezenta in program, iar fiecare metoda va avea un singur nod corespunzator.

Supersirul de executie considera toate secventele de instructiuni dintr-o metoda, chair daca acestea sunt invalide. Prin urmare, toate posibilele cai de apel vor fi considerate de catre acesta.

Asadar, putem reduce problema constructiei supersirului la problema construirii grafului $\mathcal{G}$.

In acest graf, exista o muchie de la metoda F la metoda G daca in secventa de instructiuni a lui F (corpul functiei) exista un apel catre functia G.

Avand acest graf generat, putem computa multimea $\mathcal{M}$ a metodelor care sunt accesibile pornind din nodul corespunzator functiei care contine punctului principal de intrare. Pentru a optimiza programul, vom elimina toate metodele care nu apartin acestei multimi.

Demonstratie ca acest procedeu este corect:

Lemma 3.1 *Fie m o metoda care nu apartine lui $\mathcal{M}$.*

Acest fapt inseamna ca in supersirul de executie nu exista nicio instructiune care sa apartina lui m.

Cum supersirul de executie include toate sirurile de executie valide, inseamna ca nu exista niciun sir de executie valid care, la un moment dat, sa apeleze functia m.

Prin urmare, programul obtinut prin eliminarea functiei m va fi echivalent cu programul initial.

3.5.4 Algoritmul pentru eliminarea functiilor

Pentru a construi graful, trebuie sa putem deduce pentru fiecare metoda care sunt metodele pe care aceasta le apeleaza. Acest lucru este dependent de limbajul asupra caruia se aplica optimizarea, asadar il vom considera deja implementat.

Putem considera astfel functia

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

ca fiind la dispozitia noastra.

Pentru a forma multimea metodelor accesibile din `main`, putem utiliza o parcugere in latime 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
```

Avand multimea generata, ramane doar sa eliminam functiile 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
```

Chapter 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 platforma țintă, Java a avut avantajul ca 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 platforma este de transforma codul într-o reprezentare intermediara, în loc de direct în cod binary pentru o anumita arhitectura .

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

Acest limbaj este un limbaj 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 implementata ca o "mașină cu stiva" (Eng. Stack machine), unde toate instrucțiunile limbajului bytecode interacționează cu datele de pe o stivă controlată de aplicație.

Mașina virtuală însuși este implementata în C/C++, și este compilata în cod binar direct, dependent de arhitectura. Dezvoltatorii limbajului Java sunt responsabili pentru corectitudinea și siguranța mașinii virtuale, în timp ce dezvoltatorii de aplicații Java au garanția ca 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 [1]. Totuși, Java obține performante mult mai bune decât acestea. Acest fapt se datorează compilării tocmai-la-timp (Eng. just-in-time), în care atunci când interpretorul observa o secventa de cod care este interpretata repetitiv de foarte multe ori, va genera direct cod binary pentru aceasta.

4.4 Fișierele clasa

Fișierele de clasa Java sunt formate din 10 secțiuni[2]:

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

4.5.2 Versiunea

Versiunea unui fișier clasa este data de doua valori, versiunea majora `M` și versiunea minora `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 metoda a unei clase.
  - Referința către o constanta 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 tabela, însa, 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 masca de biți, care reprezintă operațiile permise pe această clasa:

- dacă clasa este publica, 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, înseamna că clasa curentă nu moștenește nimic: singura clasa fără o super clasa 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 interfața 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 abstracta, 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 clasa, 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 camp în tabela de constante, de tipul `CONSTANT_Utf8` — un sir 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 contante. 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 structura identica cu `CONSTANT_Fieldref`, doar ca primul indice arata neapărat spre o clasa, în timp ce al doilea indice arata spre numele și tipul metodei.

CONSTANT_InterfaceMethodref

Corespunde valorii etichetei de 11 și conține o referința spre metoda unei interfețe. Are o structura identică cu `CONSTANT_Methodref`, doar ca primul indice arata spre o interfața.

CONSTANT_String

Corespunde valorii etichetei de 8 și reprezintă un sir de caractere. Conține un indice către o structura 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 virgula 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 virgula 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 camp sau al unei metode, fără informații despre clasa. Conține doi indici, amândoi către structuri de tipul `CONSTANT_Utf8`. Primul reprezinta numele, iar al doilea tipul.

CONSTANT_Utf8

Corespunde valorii etichetei de 1. Reprezintă un sir 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 ca 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 camp, o metoda de clasa, sau o metoda de interfața.

CONSTANT_MethodType

Corespunde valorii etichetei de 16 și conține un indice către o constanta `CONSTANT_Utf8`, ce reprezinta 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:

```
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 structura de tipul `field_info`.

În cod C++, aceasta structura a fost reprezentata astfel:

```
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 constanta de tipul `CONSTANT_Utf8`.
- `descriptor_index` arata spre o constanta de tipul `CONSTANT_Utf8` și reprezinta tipul câmpului.

`method_info` Fiecare metoda a unei clase/interfețe este descrisa prin aceasta structura.

În cod `C++`, am implementat-o asa:

```
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 abstracta, 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 implementata astfel:

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

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

Atributul de cod

Pentru intențiile noastre, atributul de interes este cel de cod:

```
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.
};
```

Aceasta structura este esențiala pentru implementarea optimizării, intricat ea ne permită să determinam 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 aceasta bucata de cod este interpretata.
- `code`: Codul metodei.
- `exception_table`: Excepțiile pe care le poate arunca metoda.

Code

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

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

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

Instrucțiunile au în general formatul [3]

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

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

Fiecărei instrucțiuni ii 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. Acestea da pop la doua valori din vârful stivei: `value1` și `value2`. Ambele valori trebuie să fie de tipul `int`. Rezultatul este înmulțirea celor doua valori: `result = value1 * value2`, și este pus în vârful stivei.

Dintre cele peste o suta 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.

Index1 și index2 sunt doi octeți. Aceștia sunt compuși în

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

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

Indicele compus reprezinta o intrare în tabela de constante. La locația respectiva trebuie să se afle o structura de tipul `CONSTANT_MethodHandle`

invokeinterface

Format:

```
invokeinterface
index1
index2
count
0
```

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

La poziția respectivă în tabela, trebuie să se găsească o structura 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-u corespunzător este 183 sau 0xb7. La fel ca la `invokeinterface`, este format un indice în tabela de constante, către o structura `CONSTANT_Methodref`.

Aceasta instrucțiune este folosită pentru a invoca constructorii claselor.

invokestatic

Format:

```
invokestatic
index1
index1
```

Opcode-ul corespunzător este 184 sau 0xb8. Instrucțiunea este invocată pentru a invoke o metoda statica 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, iar interpretarea este la fel ca la `invokeinterface`. Aceasta este cea mai comuna 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.

Chapter 5

Limbajul Java și optimizarea acestuia

În acest capitol vom descrie cum putem implementa optimizarea eliminării de metode nefolosite în limbajul Java.

5.1 Optimizarea limbajului Java

În aceasta parte vom urmări implementarea algoritmului descris la sfârșitul capitolului 'Optimizări de dimensiune' 3.

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 analiza statica imposibila, intricat optimiza torul ar putea elimina metode care nu sunt apelate în mod explicit în cod, însa care ajung să fie referențiale în mod dinamic la rulare.

Prin urmare, în continuarea lucrării vom presupune ca toate proiectele Java cu care lucram 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 ca aceasta condiție este respectata.

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 denumita `main` [4], 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. În implementarea lucrării, aceasta operație este realizata de către clasa Project:

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

Pentru a găsi metoda `main`, sunt scanate toate fișierele clasa care fac parte din proiect, și sunt analizatele 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 Fals, "Am gasit mai multe
                   metode main!".
            ret = p.method_of_name("main")
    if not ret:
        assert Fals, "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 implica apeluri de funcții - familia invoke*.

Odată găsite instrucțiunile de invocare de funcții, este necesar să rezolvam referințele conținute de acestea, intricat în reprezentarea interna a JVM-ului, o metoda este referențiala 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 instantiate 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 introdusa 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 ca proiectele cu care lucram nu vor utiliza reflecția sau invocarea întâlnita dinamica a codului, aceasta instrucțiune nu va fi întâlnita.

5.1.6 Rezolvarea metodelor statice

Metodele statice sunt cele mai simple de rezolvat: referința către o astfel de metoda indica numele metodei, tipul metodei, cat și clasa din care face parte și de unde este invocata. Aceste metode sunt dezvolta direct la compilare, deci cuplul (nume, tip, clasa) identifica în mod unic o metoda statica.

5.1.7 Rezolvarea metodelor virtuale

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

Aceasta este cea mai comuna 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 slotul 2 din tabela de constante este o referința către o metoda cu numele de `foo` cu tipul `void foo()`, definita în clasa `Other`.

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

Deși metoda `foo` este definita initial în clasa `Other`, la rulare va fi apelata versiunea definita în subclasa `One`.

În implementarea JVM-ului, pe stiva mașinii se va afla o instanta a obiectului a care metoda este apelata. În cazul programului nostru, pe stiva se va afla o referința către variabila `o`, de tipul `Other`, definita pe linia 13 a programului.

Pentru rezolvarea metodei virtuale, JVM-ul se asigura ca tipul lui `o`, adică `Other`, este un moștenitor al clasei de care aparține metoda `One`. În clasa ca tipul `Other` definește chiar el metoda căutata, mașina va folosi definiția respectiva. În caz contrar, mașina virtuală va cauta recursiv metoda în super clasa lui `Other`. În cazul în care căutarea recursiva a ajuns pana 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()`, definita în interfața `I`. `#2 = InterfaceMethodref #22.#23 // I.foo:()V`

Deși metoda `foo` este declarata initial în interfața `I`, la rulare va fi apelata versiunea definita în subclasa `C`.

Pentru rezolvarea referințelor către metode de interfețe, se va aplica un algoritm similar cu cel de la rezolvarea referințelor de metode virtuale: căutare recursiva în lanțul de moșteniri al clasei asupra căruia se executa 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 resolve_reference(
    class, method_name, method_type):
    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)
```

Acest algoritm este declarat în implementarea în C++ sub forma de

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

Motivul pentru care aceasta funcție întoarce un `optional`, în loc de direct o metoda, este pentru ca pot exista referințe către funcții terțe, de obicei definite în librarii. De exemplu, funcția `println(String s)`, definita în librăria standard `java.io`.

Metoda exacta care este executata de o astfel de instrucțiune nu poate fi știuta decât abia la run time (e.g., pentru programul 5.1.7 nu putem știi în timpul analizei statice dacă va fi apelata 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 referința către o interfața poate să conțină orice obiect care implementează acea interfața, sau o referința către o clasa $\mathcal{C}$ poate conține fix clasa $\mathcal{C}$, sau orice clasa care o are pe $\mathcal{C}$ ca strămoș.

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 interfața prin **invokeinterface**), vom considera toate metodele posibile care pot referențiale.

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 ca 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 referențiala, 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 reflexiva și tranzitiva, însa nu și simetrica.

Aceasta relație este echivalenta cu: metoda m este un candidat pentru rezoluția metodei q dacă:

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

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

Graful de apeluri trebuie schimbat astfel încet să includă și candidații unei metode: în loc să tragem o singura muchie de la metoda p care face invocarea, la metoda m care este invocata, 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 aceasta 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 identica:

```
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
```

În implementarea în C++, aceste au următorii corespondenți:

Funcția $cand(m)$ este reprezentata de

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

Procedura *reachable_methods_in_java* pentru determinarea funcțiilor accesibile este reprezentata de:

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

Iar procedura de optimizare *optimize_for_size_in_java* este reprezentata de:

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

Chapter 6

Implementarea optimizarii

6.1 Deserializare

Prima problema intalnita in construirea optimizatorului este serializarea si deserializarea fisierelor clasa. Problema aceasta a fost rezolvata folosind clasa `ClassReader`:

```
#pragma once

#include <cassert>
#include <cstring>
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>

#include "bytesparser.h"
#include "classfile.h"
#include "types.h"

/// This class handles the parsation (deserialization
    and serialization) of
/// Java's .class files.
/// Normal usage should be:
/// 1. Reading the binary data (for example, from a
   file on disk)
/// 2. Instantiating this ClassReader.
/// 3. Parsing the actual file.
struct ClassReader {
  private:
```

```
    /// The binary representation of the class being
       parser.
    BytesParser m_bparser;

    /// The class file that is being populated as the
        parsing progresses.
    ClassFileImpl m_cf;

  public:
    /// Initialize the reader, with the binary `data`
        of the class file.
    ClassReader(std::vector<uint8_t> data);

    /// Parse an entire class file.
    /// This is the method that you most likely want
       to use.
    ClassFileImpl deserialize();

  private:
    /// Parses a constant from the data buffer, and
       returns the data
    /// and how many slots it takes up in the
       constant table.
    cp_info parse_cp_info();

    /// Parses a field_info struct from the data
       buffer.
    field_info parse_field_info();

    /// Parses a method_info struct from the data
       buffer.
    method_info parse_method_info();

    /// Asserts that `idx` is an index into the
       constant pool, tagged with
    /// `tag`.
    void expect_cpool_entry(int idx, cp_info::Tag tag
       ) const;
};
```

.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` [5], 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
```

Bibliography

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

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

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

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

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

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

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

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

[9] 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].

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

[11] 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].