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 aplicatii este Java. O reducere de doar cativa octeti a dimensiunii pachetului unei aplicatii populare, precum Facebook, ar duce la economisirea de cativa giga-octeti de trafic de internet lunar.

In aceasta teza am proiectat si implementat un compilator care efectueaza optimizari de dimensiune a codului asupra fisierelor compilate Java.

Aceste optimizari elimina functiile si metodele neutilizate dintr-un proiect dezvoltat in limbajul Java si se bazeaza pe analiza statica a proiectului. O serie de teste au fost create pentru a testa corectitudinea, cat si eficienta optimizarilor aplicate. Compilatorul lucreaza direct cu fisiere compilate, in formatul utilizat si de catre Android.

# Contents

Chapter 1

# Introducere

Eliminarea codului nefolosit (eng. ”Dead code elimination”) [11] este o optimizare clasica. Ea presupune eliminarea dintr-un program a codului care nu afecteaza rezultatul. Acest lucru poate sa fie datorat faptului ca nu exista niciun fir de executie care sa ajunga la instructiunile respective, sau ca acele instructiuni nu au efecte laterale care sa efecteze restul programului.

Acest tip de optimizare este implementata traditional in limbajele compilate precum C, cat si in cele ‘compilate-la-timp’ (eng. ‘just-in-time’) precum Java sau JavaScript. Beneficiile principale aduse sunt:

1. Reducerea dimensiunii programului
2. Cresterea vitezei executiei

Cu cat un limbaj este mai dinamic, si permite mai multe schimbari ale comportamentului uzual, cu atat eliminarea de cod nefolosit devine mai grea. De exemplu, echipa care dezvolta Internet Explorer a avut probleme de corectitudine in optimizarile realizate, datorita naturii dinamice a limbajului JavaScript [9].

Desi pentru un programator nu este natural sa creeze cod nefolosit, acest lucru nu inseamna ca principiul eliminarii acestuia este ineficient; compilatoarele in sine, prin natura lor de a trece de mai multe ori prin codul sursa si de a avea mai multe reprezentari intermediare pot genera cod nefolosit.

Limbajul C, prin natura sa de a include fisiere mot-a-mot, este o exemplificare foarte buna pentru acest lucru: un program de cateva linii, care afiseaza ”Hello, world!” la ecran, ajunge sa aiba, inaintea eliminarea codului mort, cateva mii de linii si sute de functii nefolosite. Acest lucru se datoreaza nevoii includerii librariei standard.

Eliminarea metodelor si functiilor este un una dintre multele posibilitati de a scapa de cod nefolosit. In limbajul Java, din cauza natura acestuia

care permita incarcarea si executarea de cod arbitrar la rulare, eliminarea functiile intr-un mod general corect este un procedeu imposibil - daca o functie pusa la dispozitie de o librarie este eliminata, nu se poate garanta ca pe viitor clasa care contine functie nu va fi incarcata dinamic intr-un mod neprevazut.

Chapter 2

# Problema de optimizare

## 2.1 Formalizarea problemei

Fie programul $\mathcal{P}$ un proiect Java, format dintr-o multime de fisere clasa. Scopul optimizatorului este sa creeze un program $\mathcal{P}'$, care sa se comporte identic cu $\mathcal{P}$, si sa fie mai bun decat $\mathcal{P}$ pentru o anumita metrica $\mathcal{M}$.

## 2.2 Diferentierea programelor

Doua programe $\mathcal{P}$ si $\mathcal{Q}$ pot fi diferentiate daca exista un input $\mathcal{I}$ pentru care $\mathcal{P}$ rulat pe $\mathcal{I}$ si $\mathcal{Q}$ rulat pe $\mathcal{I}$ dau rezultate diferite.

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

Unde prin rezultat intelegem atat output-ul programului, in sensul pur matematic, cat si efectele laterale generate, care au efect asupra mediului unde ruleaza programul.

Daca doua programe nu pot fi diferentiate (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
```

si fie $\mathcal{Q}$

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

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

## 2.3 Metrici de optimizare

In contextul optimizarii de programe este nevoie sa definim ce inseamn ca dintre doua programe echivalente $\mathcal{P}$ si $\mathcal{Q}$, $\mathcal{P}$ sa fie mai performant decat $\mathcal{Q}$. Cele mai folosite doua metrici sunt metrica de viteza de executie a unui program, si 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 diferenta de timp dintre cand programul isi incepe executia, pana cand acesta isi termina executia.

Pe sistemele *nix, un mod usor a masura 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 $\mathcal{P}$ si $\mathcal{Q}$.

Vom spune ca $\mathcal{P}$ este mai rapid decat $\mathcal{Q}$ pe baza timpului de rulare daca timpul de rulare mediu al lui $\mathcal{P}$ este mai mic decat 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 comparatii, putem defini o relatie de ordine asupra multimii programelor: $\mathcal{P} < \mathcal{Q}$ ddaca $\mathcal{P}$ este mai rapid decat $\mathcal{Q}$.

#### Problema optimizarii pe baza metricii de viteza

Avand definita relatia de ordine, problema optimizarii pe baza metricii de viteza este:

Dandu-se un program $\mathcal{P}$, sa se gaseasca $\mathcal{Q}$ 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 instructiunilor acestui program:

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

Unde prin $instruction\_size(i)$ intelegem numarul de octeti ocupati de instructiunea $i$.

De exemplu, pentru limbajul Java, instructiunea *invokedynamic* ocupa 5 octeti, in timp ce instructiunea *dmul* ocupa un singur octet.

#### Comparare bazata pe dimensiunea programelor

Pentru doua programe $\mathcal{P}$ si $\mathcal{Q}$, vom spune ca $\mathcal{P}$ este mai mic decat $\mathcal{Q}$ ddaca $size(\mathcal{P}) < size(\mathcal{Q})$.

Similar ca la metrica de viteza, putem defini o relatie de ordine pe multimea programelor.

#### Problema optimizarii pe baza metricii de dimensiune

Avand definita relatia de ordine, problema optimizarii pe baza metricii de dimensiune este aceeasi ca la viteza:

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

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

## 2.4 Discutie asupra metricilor

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

Totodata, exista cazuri cand cele doua metrici sunt contrare. Un exemplu clasic este tehnica de "derularea buclelor" (eng. loop unrolling). Aceasta consta in explicitarea unei bucle cu un numar cunoscut de iteratii:

Programul

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

va fi optimizat pentru viteza in

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

in timp ce aceasta optimizare va creste numarul de instructiuni al programului, deci si dimensiunea acestuia.

Deoarece cele mai multe programe utilizate nu ruleaza pe medii constranse de memorie, tipul de optimizare folosit aproape intotdeauna este cel de viteza: este mult mai util daca un program ruleaza de 2 ori mai repede, decat daca acesta ocupa de 2 ori mai putin.

Acest lucru se datoreaza faptului ca performanta memoriei (pretul per unitate de memorie) a continuat sa scada in ultimul deceniu, in timp ce performanta procesoarelor (numarul de instructiuni executate per secunda) a stagnat.

Asa cum se poate observa in figura 2.1, trendul care urma legea lui Moore [10] a inceput sa se opreasca. Pe de alta parte, eficienta memoriei calculatoarelor si-a continuat trendul de crestere, asa cum se poate observa in continuat sa creasca, asa cum se poate observa in figura 2.2.

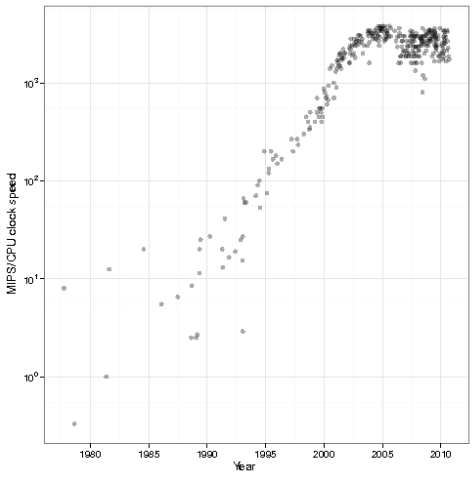


**Figure 2.1:** Evolutia puterii de procesare[7]

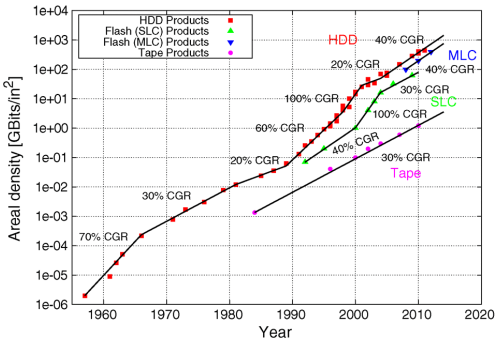


**Figure 2.2:** Evolutia puterii de procesare[8]

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 intreg, 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

# Limbajul Java si optimizarea acestuia

In prima parte, vom detalia limbajul Java si modul de functionare a acestuia. In a doua parte, vom descrie cum putem implementa optimizarea eliminarii de metode nefolosite in limbajul Java.

## 4.1 Limbajul Java

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

Jaza s-a bazat pe sintaxa limbajului C, si a introdus notiunea de "scrie o data, ruleaza peste tot" (eng. "write once, run everywhere"). Spre deosebire de C si de C++, care trebuiesc compilate pentru fiecare platforma tinta, Java a avut avantajul ca trebuie compilat o singura data, si va merge garantat pe toate platformele suportate de limbaj.

## 4.2 Java Bytecode

Solutia limbajului Java pentru a fi independent de platforma este de transforma codul intr-o reprezentare intermediara, in loc de direct in cod binary pentru o anumita arhitectura .

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

Acest limbaj este un limbaj low-level, destinat in mod exclusiv procesarii de catre masini, spre deosebire de codul Java, care este destinat oamenilor.

Dupa ce compilatorul a procesat codul Java, provenit din fisere .java in format text, acesta salveaza rezultatul in fisiere de tip clasa (.class) in format binar.

## 4.3 JVM

Odata generate fisierele binare, acestea sunt executate pe o masina virtuala specifica limbajului Java — numita `JVM` (eng. Java Virtual Machine).

Aceasta masina virtuala are rolul de a citi fisierele de clasa binare si de a le interpreta.

Masina virtuala este implementata ca o "masina cu stiva" (eng. stack machine), unde toate instructiunile limbajului bytecode interactioneaza cu datele de pe o stiva controlata de aplicatie.

Masina virtuala insusi este implementata in C/C++, si este compilata in cod binar direct, dependent de arhitectura. Dezvoltatorii limbajului Java sunt responsabili pentru corectitudinea si siguranta masinii virtuale, in timp ce dezvoltatorii de aplicatii Java au garantia ca daca codul lor Java este corect, atunci acesta va rula la fel, deterministic, pe orice platforma.

In acest regard, limbajul Java poate fi vazut ca un limbaj interpretat. Comparand cu alte limbaje populare interpretate, ca de exemplu Python, Ruby, sau Perl, ne-am astepta ca si Java sa fie la fel de incet ca acestea [1]. Totusi, Java obtine performante mult mai bune decat aceastea. Acest fapt se datoreaza compilarii tocmai-la-timp (eng. just-in-time), in care atunci cand interpretorul observa o secventa de cod care este interpretata repetitiv de foarte multe ori, va genera direct cod binary pentru aceasta.

## 4.4 Fisierele clasa

Fisierele de clasa Java sunt formate din 10 sectiuni[2]:

1. Constanta magica.
2. Versiunea fisierului.
3. Constantele clasei.
4. Permisiunile de acces.
5. Numele clasei din fisier.
6. Numele superclasei.
7. Interfetele pe care clasa le implementeaza.
8. Campurile clasei.
9. Metodele clasei.
10. Atribute ale clasei.

In continuare voi da o scurta descriere a formatului sectiunilor.

## 4.5 Sectiunile fiserelor clasa

### 4.5.1 Magic

Toate fiserele clasa trebuiesc sa inceapa cu un numar denumit constanta magica. Acesta este folosit pentru a identifica in mod unic ca acestea sunt intradevar fisiere clasa. Numarul magic are o valoare memorabila: reprezentarea hexadecimala este `0xCAFEBABE`,

### 4.5.2 Versiunea

Versiunea unui fisier clasa este data de doua valori, versiunea majora `M` si versiunea minora `m`. Versiunea clasei este atunci reprezentata ca `M.m`. (e.g., `45.1`). Aceasta este folosita pentru a mentine compatibilitatea in cazul modificarilor masinii virtuale care interpreteaza clasa sau ale compilatorului care o genereaza.

### 4.5.3 Constantele clasei

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

- Numere intregi.
- Numere cu virgula mobula.
- Siruri de caractere, care pot reprezenta la randul lor:
  - Nume de clase.
  - Nume de metode.
  - Tipuri ale metodelor.
- Informatii compuse din datele anterioare:
  - Referinta la o metoda a unei clase.
  - Referinta la o constanta a unei clase.

Toate celelalte tipuri de date compuse, cum ar fi metodele sau campurile, vor contine indecsi in tabela de constante.

Aproape toate tipurile de constante ocupa un singur slot in tabela, insa, din motive istorice, unele constante ocupa doua sloturi. Tot din motive istorice, tabela este indexata de la 1, si nu de la 0, cum sunt celelalte.

### 4.5.4 Permisiunile de acces

Aceste permisiuni constau intr-o masca de bitsi, care reprezeinta operatiile permise pe aceasta clasa:

- daca clasa este publica, si poate fi accesta din afara pachetului acesteia.
- daca clasa este finala, si daca poate fi extinsa.
- daca invocarea metodelor din superclasa sa fie tratata special.
- daca este de fapt o interfata, si nu o clasa.
- daca este o clasa abstracta si nu poate fi instatiata.

### 4.5.5 Clasa curenta

Reprezinta un indice in tabela de constante, unde sunt stocate informatii despre clasa curenta.

### 4.5.6 Clasa super

Reprezinta un indice in tabela de consatante, cu informatii despre clasa din care a mostenit clasa curenta. Daca este 0, inseamna ca clasa curenta nu mosteneste nimic: singura clasa fara o superclasa este clasa Object.

E.g. pentru

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

Indicele corespunde lui `S`.

### 4.5.7 Interfetele

Reprezinta o colectie de indici in tabela de constante. Fiecare valoare de la acei indici reprezinta o interfata implementata in mod direct de catre clasa curenta. Interfetele apar in ordinea declarata in fisierele 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 Campurile

Reprezinta informatii despre campurile (eng. fields) clasei:

- Permisiunile de acces: daca este public sau privat, etc.
- Numele campului.
- Tipul campului.
- Alte atribute: daca este deprecat, daca are o valoare constanta, etc.

### 4.5.9 Metodele

Reprezinta informatii despre toate metodele clasei, inclusiv constructorii:

- Permisiuni de acces: daca este public sau privat, daca este finala, daca este abstracta.
- Numele metodei.
- Tipul metodei.
- In caz ca nu este abstracta, byte codul metodei.
- Alte atribute:
  - Ce exceptii poate arunca.
  - Daca este deprecata.

Codul metodei este partea cea mai importanta, iar formatul acestuia urmeaza sa fie detaliat ulterior.

### 4.5.10 Atributele

Reprezinta alte informatii despre clasa, cum ar fi:

- Clasele definite in interiorul acesteia.
- In caz ca este o clasa anonima sau definita local, metoda in care este definita.
- Numele fisierul sursa din care a fost compilata clasa.

## 4.6 Tipuri de date

In continuare, voi descrie din punct de vedere tehnic tipurile de date intalnite in fisierele de clasa.

### 4.6.1 Tipuri de baza

In formatul fisierelor clasa exista trei tipuri de baza, toate bazate pe intregi. In caz ca un intreg are mai multi octeti, acestia au ordinea de `big-endian`: cel mai semnificativ octet va fi mereu primul in memorie.

| Nume | Semantica | Echivalentul in C |
|---|---|---|
| `u1` | intreg pe un octet, fara semn | `unsigned char` sau `uint8_t` |
| `u2` | intreg pe doi octeti, fara semn | `unsigned short` sau `uint16_t` |
| `u4` | intreg pe un octet, fara semn | `unsigned int` sau `uint32_t` |

In codul sursa 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 Tipurile de constante

Fiecare constanta din tabela de constante incepe cu o eticheta de 1 octet, care reprezinta datele si tipul structurii. Continutul acesteia variaza in functie de eticheta, insa indiferent de eticheta, continutul trebuie sa aiba cel putin 2 octeti.

#### CONSTANT_Class

Corespunde valorii etichetei de 7 si contine un indice spre un alt camp in tabela de constante, de tipul `CONSTANT_Utf8` — un sir de caractere. Acel sir de caractere va contine numele clasei.

#### CONSTANT_Fieldref

Corespunde valorii etichetei de 9 si contine o referinta spre campul unei clase. Referinta conta in doi indici, amandoi care arata spre tabela de contante. Primul indice arata spre o constanta `CONSTANT_Class`, care reprezinta clasa sau interfata careia apartine metoda. Al doilea indice arata spre o constanta `CONSTANT_NameAndType`, care contine informatii despre numele si tipul campului.

#### CONSTANT_Methodref

Corespunde valorii etichetei de 10 si contine o referinta spre metoda unei clase. Are o structura identica cu `CONSTANT_Fieldref`, doar ca primul indice arata neaparat spre o clasa, in timp ce al doilea indice arata spre numele si tipul metodei.

#### CONSTANT_InterfaceMethodref

Corespunde valorii etichetei de 11 si contine o refereinta spre metoda unei interfete. Are o structura identica cu `CONSTANT_Methodref`, doar ca primul indice arata spre o interfata.

#### CONSTANT_String

Corespunde valorii etichetei de 8 si reprezinta un sir de caractere. Contine un indice, catre o structura de tipul `CONSTANT_Utf8`.

#### CONSTANT_Integer

Corespunde valorii etichetei de 3 si contine un intreg pe 4 octeti.

#### CONSTANT_Float

Corespunde valorii etichetei de 4 si contine un numar cu virgula mobila pe 4 octeti.

#### CONSTANT_Long

Corespunde valorii etichetei de 5 si contine un intreg pe 8 octeti. Din motive istorice, ocupa 2 spatii in tabela de constante.

#### CONSTANT_Double

Corespunde valorii etichetei de 6 si contine un numar cu virgula mobila pe 8 octeti. Din motive istorice, ocupa 2 spatii in tabela de constante.

#### CONSTANT_NameAndType

Corespunde valorii etichetei de 12. Descrie numele si tipul unui camp sau al unei metode, fara informatii despre clasa. Contine doi indici, amandoi catre structuri de tipul `CONSTANT_Utf8`. Primul reprezinta numele, iar al doilea tipul.

#### CONSTANT_Utf8

Corespunde valorii etichetei de 1. Reprezinta un sir de caractere encodat in formatul UTF-8. Contine un intreg `length`, de tipul `u2`, si apoi `length` octeti care descriu sirul in sine. Din cauza ca este encodat ca UTF-8, un singur caracter poate fi format din mai multi octeti.

#### CONSTANT_MethodHandle

Corespunde valorii etichetei de 15 si contine o referinte catre un camp, o metoda de clasa, sau o metoda de interfata.

#### CONSTANT_MethodType

Corespunde valorii etichetei de 16 si contine un indice catre o constanta `CONSTANT_UTf8`, ce reprezinta tipul unei metode.

#### CONSTANT_InvokeDynamic

Corespunde valorii etichetei de 18 si este folosit de catre `JVM` pentru a invoka o metoda polimorfica.

In 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 camp din cadrul unei clase este reprezentat printr-o structura de tipul `field_info`.

In 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 in tabela de constante unde se afla o constanta de tipul `CONSTANT_Utf8`.
- `descriptor_index` arata spre o constanta de tipul `CONSTANT_Utf8` si reprezinta tipul campului.

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

In 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` si `descriptor_index` au aceeasi interpretare ca si la `field_info`.

Daca metoda nu este abstracta, atunci in vectorul `attributes` se va gasi un attribut de tipul `Code`, care contine bytecode-ul corespunzator acestei metode.

`attribute_info` In `C++`, a fost implementata astfel:

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

Numele atributului determina modul in care octetii din vectorul `info` sunt interpretati.

### Atributul de cod

Pentru intentiile 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 esentiala pentru implementarea optimizarii, intrucat ea ne permita sa determinam metodele apelate din cadrul unei secvente de cod. In continuare, o voi descrie detaliat:

- `max_stack`: Reprezinta adancimea maxima a stivei masinii virtuale cand aceasta bucata de cod este interpretata.
- `max_locals`: Reprezinta numarul maxim de variabile locale alocate in acelasi timp cand aceasta bucata de cod este interpretata.
- `code`: Codul metodei.
- `exception_table`: Exceptiile pe care le poate arunca metoda.

### Code

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

Acest vector contine instructiunile care sunt executate de catre masina virtuala.

JVM-ul ruleaza ca o masina cu stiva, iar toate instructiunile opereaza pe aceasta stiva. Reultatul rularii unei instructiuni este modificarea stivei: scoaterea si adaugarea de elemente in varful acesteia.

Instructiunile au in general formatul [3]

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

cu un numar variabil de operanzi, prezenti in mod explicit in vectorul de `cod`.

Fiecarei instructiuni ii corespunde un octet, denumit opcode. Fiecare operand este fie cunoscut la compilare, fie calculat in mod dinamic la rulare.

Cele mai multe operatii nu au niciun operand dat in mod explicit la nivelul instructiunii — ele lucreaza doar cu valorile din varful stivei la momentul executarii codului.

De exemplu:

Instructiunea `imul` are octetul 104 sau 0x68. Acestea da pop la doua valori din varful stivei: `value1` si `value2`. Amandoua valorile trebuie sa fie de tipul `int`. Rezultatul este inmultirea celor doua valori: `result = value1 * value2`, si este pus in varful stivei.

Dintre cele peste o suta de instructiuni, noi suntem preocupati doar de 5 dintre acestea: cele care au de a face cu invocarea unei metode.

**invokedynamic**

Format:

```
invokedynamic
index1
index2
0
0
```

Opcode-ul corespunzator acestei instructiuni este 186 sau 0xba.

Index1 si index2 sunt doi octeti sunt compusi in

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

Unde << reprezinta shiftare pe bitsi, iar | reprezinta operatia de sau pe bitsi.

Indicele compus reprezinta o intrare in tabela de constante. La locatia respectiva trebuie sa se afle o structura de tipul `CONSTANT_MethodHandle`

**invokeinterface**

Format:

```
invokeinterface
index1
index2
count
0
```

Opcode-ul corespunzator este 185 sau 0xb9. `index1` si `index2` sunt folositi, in mod similar ca la `invokedynamic`, pentru a construi un indice in tabela de constante.

La pozitia respectiva in tabela, trebuie sa se regaseasca o strutura de tipul `CONSTANT_Methodref`.

`count` trebuie sa fie un octet fara semn diferit de 0. Acest operand descrie numarul argumentelor metodei, si este necesar din motive istorice: aceasta informatie poate fi dedusa din tipul metodei.

**invokespecial**

Format:

```
invokespecial
index1
index2
```

Opcode-u corespunzator este 183 sau 0xb7. La fel ca la `invokeinterface`, este format un indice in tabela de constante, catre o structura `CONSTANT_Methodref`.

Aceasta instructiune este folosita pentru a invoca constructorii claselor.

**invokestatic**

Format:

```
invokestatic
index1
index1
```

Opcode-ul corespunzator este 184 sau 0xb8. Instructiunea este invocata pentru a invoke o metoda statica a unei clase.

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

**invokevirtual**

Format:

```
invokevirtual
index1
index1
```

Opcode-ul corespunzator este 182 sau 0xb6, iar interpretarea este la fel ca la `invokeinterface`. Aceasta este cea mai comuna instructiune de invocare de functii.

In `C++`, am reprezentat aceste instructiuni de interes astfel:

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

### 4.6.4 ClassFile

Folosind definitiile anterioare, putem descrie un fisier de clasa binar in 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 fisier clasa analizat in detaliu, va puteti uita la appendix-ul studiu de caz.

## 4.7 Optimizarea limbajului Java

In aceasta parte vom urmari implementarea algoritmului descris la sfarsitul capitolului 'Optimizari de dimensiune' 3.

### 4.7.1 Preliminarii

Java este un limbaj dinamic. Acest lucru inseamna posibilitatea de a modifica codul in timpul rularii, sau de a apela cod in mod dinamic la rulare: aceasta este partea de 'reflectie' a limbajului Java.

Utilizarea acestor caracteristici face abordarea noastra de analiza statica imposibila, intrucat optimizatorul ar putea elimina metode care nu sunt apelate in mod explicit in cod, insa care ajung sa fie referentiate in mod dinamic la rulare.

Prin urmare, in continuarea lucrarii vom presupune ca toate proiectele Java cu care lucram nu se folosesc de niciun mod de a schimba structura codului in timpul rularii. In caz contrar, optimizatorul nu mai ofera nicio garantie asupra corectitudinii optimizarilor realizate.

Restul lucrarii va presupune ca aceasta conditie este respectata.

### 4.7.2 Determinarea punctului de intrare

Un proiect Java este format dintr-o multime de clase. Punctul `main` al proiectului este reprezentat de functia denumita `main` [4], cu antetul

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

Intr-un proiect trebuie sa exista o singura astfel de metoda, care sa apartina unei clase a proiectului. In implementarea lucrarii, aceasta operatie este realizata de catre clasa Project:

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

Pentru a gasi metoda `main`, sunt scanate toate fiserele clasa care fac parte din proiect, si sunt analizatele listele de metode ale acestora.

Acesta este pseudocodul pentru aceasta operatie:

```
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
```

### 4.7.3 Gasirea apelurilor de functii

Pentru a determina ce functii sunt apelate din cadrul unei metode `m`, vom inspecta atributul de cod al lui `m`. Dintre instructiunile continute vom fi interesati doar de cele ce implica apeluri de functii - familia invoke*.

Odata gasite instructiunile de invocare de functii, este necesar sa rezolvam referintele continute de acestea, intrucat in reprezentarea interna a JVM-ului, o metoda este referentiata pur simbolic, prin siruri de caractere.

### 4.7.4 Rezolvarea metodelor speciale

Acest tip de rezolvare a metodelor este necesar atunci cand intampinam instructiunea **invokespecial**.

Prin metode speciale intelegem functii precum constructorul unei clase sau functii private.

Pentru scopul acestei lucrari, nu vom elimina constructorii, intrucat asta ar inseamna eliminarea unei clase cu totul (i.e., singurul caz in care putem elimina constructorul unei clase este cand clasa nu este instantiata niciodata).

Metodele private, in schimb, vor fi tratate ca niste functii normale.

### 4.7.5 Rezolvarea metodelor dinamice

Din versiunea 7 a limbajului Java a fost introdusa instructiunea **invokedynamic**. Aceasta instructiune este folosita in rezolvarea referintelor de metode la rulare - similar cu conceptul de "duck typing" din limbajele dinamice. Deoarece am facut presupunerea ca proiectele cu care lucram nu vor utiliza reflectia sau invocarea intalnita dinamica a codului, aceasta instructiune nu va fi intalnita.

### 4.7.6 Rezolvarea metodelor statice

Metodele statice sunt cele mai simple de rezolvat: referinta catre o astfel de metoda indica numele metodei, tipul metodei, cat si clasa din care face parte si de unde este invocata. Aceste metode sunt rezvolta direct la compilare, deci tuplul (nume, tip, clasa) identifica in mod unic o metoda statica.

### 4.7.7 Rezolvarea metodelor virtuale

In Java, toate metodele de instanta (non-statice) sunt polimorfice la rulare (eng. runtime polymorphism). Cu alte cuvinte, la compilare se stiu numele si tipul metodei si clasa unde metoda este definita, insa **nu** si clasa de unde este invocata.

Aceasta este cea mai comuna operatie, si corespunde instructiunii **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 fisierelor din testul `fixtures/project4`, apelul de pe linia 9 catre metoda foo este un apel polimorfic.

In codul de JVM, instructiunea corespunzatoare este:

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

unde slotul 2 din tabela de constante este o referinta catre o metoda cu numele de `foo` cu tipul `void foo()`, definita in clasa `Other`.

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

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

In implementarea JVM-ului, pe stiva masinii se va afla o instanta a obiectului a carei metoda este apelata. In cazul programului nostru, pe stiva se va afla o referinta catre variabila `o`, de tipul `Other`, definita pe linia 13 a prorgamului.

Pentru rezolvarea metodei virtuale, JVM-ul se asigura ca tipul lui `o`, adica `Other`, este un mostenitor al clasei de care apartine metoda `One`. In clasa ca tipul `Other` defineste chiar el metoda cautata, masina va folosi definitia respectiva. In caz contrar, masina virtuala va cauta recursiv metoda in superclasa lui `Other`. In cazul in care cautarea recursiva a ajuns pana la o clasa fara super (singura astfel de clasa este `Object`), masina virtuala va emite o eroare.

### 4.7.8 Rezolvarea metodelor de interfata

Ultimul tip de invocare a metodelor corespunde instructiunii **invokeinterface**, si corespunde apelarii unei functii declarate in cadrul unei interfete.

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 fisierele testului `fixtures/project5`, apelul de pe linia 11 catre metoda foo este un apel polimorfic pe interfete.

In codul de JVM, instructiunea corespunzatoare este:

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

unde slotul 2 din tabela de constante este o referinta catre o metoda de interfata cu numele de `foo` cu tipul `void foo()`, definita in interfata `I`. `#2 = InterfaceMethodref #22.#23 // I.foo:()V`

Desi metoda `foo` este declarata initial in interfata `I`, la rulare va fi apelata versiunea definita in subclasa `C`.

Pentru rezolvarea referintelor catre metode de interfete, se va aplica un algoritm similar cu cel de la rezolvarea referintelor de metode virtuale: cautare recursiva in lantul de mosteniri al clasei asupra caruia se executa metoda.

### 4.7.9 Algoritmul de rezolvare al referintelor metodelor

Putem defini in pseudocod algoritmul de rezolvare al referintelor catre metode virtuale si catre metode de interfata 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 in implementarea in C++ sub forma de

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

Motivul pentru care aceasta functie intoarce un `optional`, in loc de direct o metoda, este pentru ca pot exista referinte catre functii terte, de obicei definite in librarii. De exemplu, functia `println(String s)`, definita in libraria standard `java.io`.

Metoda exacta care este executata de o astfel de instructiune nu poate fi stiuta decat abia la runtime (e.g., pentru programul 4.7.7 nu putem stii in timpul analizei statice daca va fi apelata metoda `Other::foo()` sau metoda `One::foo()`).

### 4.7.10 Supersirul de executie si polimorfismul de executie

Vom reconsidera supersirul de executie definit la 3.5.2 si graful apelurilor, definit la 3.5.3.

La momentul analizei statice nu stim tipul exact al unei referinte. O referinta catre o interfata poate sa contina orice obiect care implementeaza acea interfata, sau o referinta catre o clasa $\mathcal{C}$ poate contine fix clasa $\mathcal{C}$, sau orice clasa care o are pe $\mathcal{C}$ ca stramos.

Prin urmare, vom extinde supersirul de executie ca sa reflecte aceste posibilitati. Mai precis, pentru fiecare apel de functii polimorfice (virtuale prin

**invokevirtual** sau de interfata prin **invokeinterface**), vom considera toate metodele posibile care pot referentiate.

De exemplu, pentru programul 4.7.7, vom considera toate posibilitatile de rezolare ale apelului catre foo(), atat `Other::foo()`, cat si `One::foo()`.

### 4.7.11 Graful de apeluri si polimorfismul de executie

Acum ca am adaptat supersirul de executie pentru polimorfismul de executie, este nevoie sa extindem si graful de apeluri.

Pentru acest lucru, in cadrul unei rezolutii vom considera nu numai metoda direct referentiata, ci toate metodele pe care supersirul de executie le-ar putea rezolva ca posibili candidati.

Vom spune ca metoda m este un candidat pentru rezolutia metodei q daca in rezolvarea unei referinte catre q, supersirul de executie va considera si metoda m.

Proprietatea de a fi candidat este o relatie reflexiva si tranzitiva, insa nu si simetrica.

Aceasta relatie este echivalenta cu: metoda m este un candidat pentru rezolutia metodei q daca:

1. q este o metoda a unei interfete, iar m implementeaza direct sau tranzitiv interfata respectiva.
2. q este o metoda a unei clase, iar m mosteneste direct sau tranzitiv clasa respectiva.

In continuare, vom nota cu $cand(m)$ toti candidatii lui m. Din faptul ca este o relatie reflexiva, m apartine mutlimii $cand(m)$.

Graful de apeluri trebuie schimbat astfel incat sa includa si candidatii unei metode: in loc sa 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 — $cand(m)$.

### 4.7.12 Noul algoritm pentru eliminarea functiilor

In aceasta parte, vom adapta algoritmul de eliminare a functiilor, definit anterior la 3.5.4. In particular, vom actualiza modul de calculare al functiilor 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 ramane 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
```

In implementarea in C++, aceste au urmatorii corespondenti:

functia $cand(m)$ este reprezentata de

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

procedura *reachable_methods_in_java* pentru determinarea functiilor 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 5

# Implementarea optimizarii

## 5.1 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].