УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Ана Д. Митровић

ПРИМЕНА СКАЛЕ У ПАРАЛЕЛИЗАЦИЈИ РАСПЛИНУТОГ ТЕСТИРАЊА

мастер рад

Београд, 2018.

Ментор:

др Милена Вујошевић Јаничић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

Наслов мастер рада: Примена Скале у паралелизацији расплинутог тестирања

Резиме:

Кључне речи:

Садржај

Глава 1

Увод

Глава 2

Програмски језик Скала

Скала је језик опште намене настао 2003. године са циљем да превазиђе ограничења програмског језика Јава комбиновањем објектно оријентисане и функционалне програмске парадигме. Мотивација иза овакве идеје је не ограничавати се на једну од ових парадигми и њених предности већ искористити најбоље из оба света. Управо због овакве комбинације парадигми Скала је веома погодна за решавање различитих врста проблема, почевши од малих незахтевних скриптова па све до великих компликованих система. На основу ове прилагодљивости је и добила своје име: реч „скала" означава „скалабилан језик" (енг. *scalable language*), односно језик који ће се прилагођавати и расти заједно са потребама система. [9, 8]

Творац Скале је Мартин Одерски (нем. *Martin Odersky*) (1958-), немачки научник и професор на универзитету EPFL (École Polytechnique Fédérale de Lausanne) у Лозани, Швајцарској. Још док је био студент, имао је жељу да укомбинује објектну и функционалну парадигму, говорећи да су ове две парадигме само две стране истог новчића. Желео је да напише језик који се преводи у Јава бајт код али и да превазиђе ограничења језика Јава. Први резултат оваквог његовог рада је био језик Funnel, који због свог минималистичког дизајна није заживео. Тако је настала Скала, на којој је Мартин радио од 2001. године заједно са својом групом на универзитету EPFL. Познат је и по другим радовима, као што је имплементација GJ (Generic Java) компајлера који је постао основа javac компајлера. [7, 10]

2.1 Опште карактеристике

Компатибилност са Јавом Скала није сама по себи продужење Јаве али је потпуно компатибилна са њом: њен изворни код се преводи у Јава бајт код који се извршава на Јава виртуелној машини. Из кода писаног у Скали је омогућено коришћење Јава библиотека, класа, интерфејса, метода, поља и типова без посебне синтаксе. Омогућено је и обрнуто, позивање Скала кода из Јаве, мада се оно ређе користи. Такође, Скала омогућава лакшу и лепшу употребу Јава типова уз помоћ имплицитне конверзије омогућавајући употребу својих метода за манипулацију типовима. Компатибилност олакшава развијаоцима софтвера лакши прелазак из Јаве у Скалу јер нису принуђени да се одједном одрекну написаног кода у Јави. [9]

Објектно оријентисана парадигма Писање програма и смештање података и њихових својстава у класе и објекте тих класа је веома популарно и интуитивно развијаоцима софтвера. Скала је то задржала притом мало изменивши концепт. У многим језицима укључујући и Јаву су дозвољене вредности које нису објекти или нису у склопу истог. То могу бити примитивне вредности у Јави или статичка поља и методи који не припадају ниједном објекту. Скала то не дозвољава и зато је објектно оријентисан језик у *чистој форми*: „Свака вредност је објекат и свака операција је позив метода" [9]. Елегантан начин коришћења метода налик на операције је један од начина на који се Скала брине да програмерима буде пријатно њено коришћење.

Концизност Скала код има тенденцију да буде краћи од Јава кода, чак се процењује да има барем два пута мање линија кода од Јаве. Постоје и екстремни случајеви где је број линија и десет пута мањи. Ова особина није значајна само због тога што знатно олакшава програмеру у писању кода, већ олакшава читање кода и откривање грешака којих ионако има мање јер има мање простора за њих. Ово је нешто што одликује саму синтаксу језика, а веома помажу и разне библиотеке које имају већ имплементиране многе функције које врше послове са којима се често сусрећемо. Лако се и имплементирају и касније употребљавају библиотеке које сами напишемо. Такође, Скала подржава аутоматско закључивање типова (енг. *type inference*) што омогућава изостављање понављања већ наведених типова, што резултује читљивијим кодом. [9]

Висок ниво Како је Скала погодна и за велике и комплексне системе она се труди да се прилагоди њиховим захтевима подижући ниво апстракције у

свом коду. Омогућава много једноставнији и краћи начин кодирања разних проблема. Рецимо, уместо да пролазимо кроз ниску карактера карактер по карактер користећи петљу, у Скали се то може урадити у једној линији кода помоћу *предиката* тј. *функцијских литерала* који су детаљније објашњени у поглављу 2.2. Скала код тежи да буде разумљивији и мање комплексан како би олакшао већ комплексан систем који се имплеметира. [9]

Статичка типизираност Скала је статички типизиран језик што значи да се типови података који су коришћени знају у време компилације. Неки сматрају ово маном као и да је навођење типова сувишно поред техника тестирања софтвера као што је нпр. тестирање јединица кода (енг. *unit testing*). Ипак, у Скали је статичка типизираност напреднија јер нам дозвољава да изоставимо тип на местима где би он био поновљен и вероватно би нам само сметао. Због оваквих понављања која су неопходна у неким језицима, многи се одрекну предности статички типизираних језика као што су: детектовање разних грешака у време компилације, лакше рефакторисање кода и коришћење типова као вид документације. [9]

Корен језика Корен Скале није само језик Јава, иако је Јава имала највећи утицај у њеном стварању. Идеје и концепти из разних језика, како објектно оријентисаних тако и функционално оријентисаних, су инспирисали развој Скале. Међу њима су језици: C# од кога је Скала преузела синтаксне конвенције, Erlang чије идеје су сличне идејама конкурентности базиране на моделу Actors и други: C, C++, Smalltalk, Ruby, Haskell, SML, F# итд. [9]

Карактеристика која највише раздваја Скалу од Јаве је њена **функционалност** - Скала је у потпуности функционалан језик. Ова особина је детаљно објашњена у следећем поглављу.

2.2 Функционални део језика

Функцинална парадигма се развија од 1960-тих година. У њеној основи леже *ламбда рачун* (енг. *λ-calculus*) и комбинаторна логика. Ламбда рачун представља математичку апстракцију и формализам за описивање функција и њихово израчунавање. Њега је увео Алонзо Черч (енг. *Alonzo Church*) 1930-тих година а Алан Тјуринг (енг. *Alan Turing*) је 1937. године показао да је експресивност ламбда рачуна еквивалентна експресивности Тјурингових машина. Иако

је ламбда рачун првобитно развијен само за потребе математике, данас се он сматра првим функционалним језиком. Други модел рачунања који је утицао на функционалне програмске језике, комбинаторну логику, створили су Мојсеј Шејнфинкел (рус. *Моисей Шейнфинкель*) и Хаскел Кари (енг. *Haskell Curry*) 1924. године са циљем да елиминишу употребу променљивих у математичкој логици. [4, 3, 1, 5]

Најстарији виши функционални програмски језик је LISP који је пројектовао Џон Макарти (енг. *John McCarthy*) на институту МИТ (Масачусетски технолошки институт) 1958. године. Након тога се полако развијају и други функционални језици као што су ISWIM, FP, Scheme, ML, Miranda, Erlang, SML, Haskell, OCAML, F#, Elixir итд. Поред функционалних језика постоје и језици који нису функционални али у одређеној мери подржавају функционалне концепте или могу да их остваре на други начин, нпр. Java, C, C++, C#, Phyton. [3, 6]

Разлика између императивне и функционалне парадигме се може описати дефинисањем самог програма[2]:

- Императивна парадигма:
  Програм представља формално упутство о томе шта рачунар треба да ради да би урадио неки посао
  Програм представља одговор на питање КАКО се нешто РАДИ

- Функционална парадигма:
  Програм представља формално објашњење онога што рачунар треба да израчуна
  Програм представља одговор на питање ШТА се РАЧУНА

Функционални језици су веома блиски математици због чега се и називају *функционалним* језицима: њихове функције се понашају као функције у математичком смислу. О томе говоре следећа два концепта којима се воде функционални језици[9]:

Први концепт се односи на **„статус" функција**. У функционалним језицима функције су вредности *првог реда*, тј. *грађани првог реда*. То значи да се оне могу користити као вредности типова као што су нпр. цели, реални бројеви, ниске карактера итд. Статус оваквих типова се не разликује од статуса функција, што у многим језицима није случај. Ово омогућава коришћење функција

на природан, концизан и читљив начин: као аргументе других функција, повратне вредности функција, њихово чување у променљивим, угњеждавање и слично. Другим речима, функције се креирају и прослеђују без икаквих рестрикција на које смо навикли у императивним језицима.

У Скали можемо користити локалне (угњеждене) функције које су често веома мале и лако се комбинују како би заједно заокружиле један већи посао. Ово одговара функционалном стилу који промовише следећу идеју: програм треба изделити на много мањих функција (целина) од којих свака има јасно дефинисан задатак.

Посебно су корисни и важни *функцијски литерали* који представљају функције које могу бити неименоване (анонимне функције) и прослеђиване као обичне вредности. Назив „функцијски литерал" се користи за функције у изворном коду, а у време извршавања оне се називају *„функцијским вредностима"* (разлика слична разлици између класа и објеката у објектно оријентисаним језицима). Функцијске вредности су објекти које можемо чувати у променљивама али су истовремено и функције које можемо позивати. Пример функцијског литерала који инкрементира цео број изгледа овако [9]:

```
var increase = (x: Int) => x + 1
increase(10)
```

Променљива **increase** је објекат који садржи литерал, али је и функција коју позивамо са аргументом 10.

Други концепт се односи на **непостојање стања** које се иначе представља променљивама у програму. То значи да извршавање метода нема *„бочне ефекте"*, односно да функције које позивамо не мењају податке „у месту" већ враћају нове вредности као резултате свог израчунавања. Избегава се (у чисто функционалним језицима се избацује) кад год је то могуће коришћење променљивих које мењају своје стање, а за промену се користе променљиве код којих то није дозвољено. То су променљиве које се иницијализују само једном. Оне се у Скали дефинишу помоћу кључне речи **val** и покушај промене вредности ове променљиве након што је иницијализована резултирао би грешком. Променљиве које су стандардне у императивним језицима и које можемо мењати по потреби се дефинишу кључном речју **var**.

Овакво непостојање стања има за последицу избацивање итеративних конструкција. Овакав резон може испрва деловати чудно, међутим све што се може

остварити кроз итерације се може остварити и *рекурзијом* јер је експресивност функционалних језика еквивалентна експресивности императивних језика. Рекурзија је у функционалним језицима потпуно природна, много примењенија и неизбежна. Некада се решења проблема који захтевају стање у оваквим језицима превише закомпликују, али оно се може направити по потреби на друге начине тако да се ово не сматра недостатком. Предност је у томе што не морамо пратити вредности променљивих и њихове промене што олакшава разумевање програма. Један програм у функционалном језику представља низ дефиниција и позива функција а његово извршавање је евалуација тих функција.[3]

Као пример метода који нема бочне ефекте може послужити метод **replace** који као аргументе добија ниску карактера и два карактера. Његов задатак је да у датој ниски замени сва појављивања једног карактера другим карактером. Он неће променити ниску коју је добио као аргумент, већ ће вратити нову која више нема појављивања датог карактера који смо заменили новим.

Ова особина метода без бочних ефеката назива се *референтна прозирност*: „за било које вредности аргумената позив метода се може заменити својим резултатом без промене семантике програма"[9]. Пример:

```
val p = previous(90)
```

Свако појављивање промењиве p можемо заменити са изразом previous(90).

За овакве методе кажемо да су референтно прозирни или транспарентни (енг. *referentially transparent*) и они су концизнији, читљивији и производе мање грешака јер немају пропратне ефекте. Ипак, у пракси су нам углавном неопходни пропратни ефекти због чега они могу бити присутни на контролисан начин. У том случају, програмски језик више није чисто функционалан. Још једна особина која одликује језике са референтном прозирношћу је да редослед операција тј. наредби није важан. У императивним језицима ситуација се веома разликује.[3]

Чисто функционални језици (нпр. Haskel, Miranda) захтевају коришћење имутабилних структура података, референтно прозирних метода и рекурзије. Други језици, као што су Скала, Python, Ruby итд. охрабрују овакво програмирање али не условљавају програмера. Скала омогућава бирање начина за решавање проблема и нуди функционалне алтернативе за све императивне конструкције. На овај начин програмер се полако и без притиска навикава на другачији начин размишљања.[3, 9]

Неке од карактеристика које Скала подржава а које су заједничке и другим функционалним језицима су[9]:

- **Функције вишег реда** Функције вишег реда су оне које имају друге функције као своје аргументе. Оне поједностављују код и смањују његово понављање. Ово је пример функције која као аргументе има ниску карактера **query** и другу функцију **matcher**:

```
def filesMatching(query: String,
                  matcher: (String, String) => Boolean) = {
       for (file <- filesHere; if matcher(file.getName, query))
                  yield file
}
```

  Функција **filesMatching** треба да међу фајловима **filesHere** пронађе фајлове који испуњавају неки критеријум. Да се код не би понављао за сваки критеријум појединачно, он се прослеђује функцији као аргумент у виду функције **matcher**. Сам критеријум представља функцију која има две ниске карактера као аргумент, и враћа логички тип - тачно ако је критеријум испуњен и нетачно у супротном.

  Следећи пример демонстрира коришћење функције вишег реда из Скала библиотеке:

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

  Метод **exists** проверава постојање елемента листе **nums** који задовољава наведени предикат који му је прослеђен као аргумент (дељивост са 2). Постоји доста метода сличних методу exists, попут: **find**, **filter**, **foreach**, **forall**, итд.

- **Каријеве функције** Каријеве функције уместо једне листе аргумената узимају аргументе један по један, имајући више листи које садрже по један аргумент. На овај начин омогућују дефинисање функција помоћу већ дефинисаних на следећи начин:

```
def curriedSum(x: Int)(y: Int) = x + y
```

  Ово је функција која сабира два цела броја. Када је позовемо на овај начин:

```
curriedSum(1)(2)
```

десиће се два позива функције. Први позив ће узети први параметар **x** и вратити другу функцију која узима параметар **y**, и враћа збир ова два параметра. Прва и друга функција би редом изгледале овако:

```
def first(x: Int) = (y: Int) => x + y
val second = first(1) /* poziv prve funkcije */
second(2) /* poziv druge funkcije */
```

Каријев поступак има пуно примена. Рецимо, можемо дефинисати функцију која додаје број 1 неком целом броју:

```
val onePlus = curriedSum(1)_
onePlus(2) /* --> vraca 3 */
```

Позивамо Каријеву функцију **curriedSum** користећи _ као знак да на том месту може бити било која вредност (енг. *wildcard*). Суштина је да је функција **onePlus** дефинисана помоћу Каријеве функције која сабира било која два цела броја, а код које је први аргумент везан за број 1.

- **Упаривање образаца** Често је потребно препознати да ли неки израз има одређену форму тј. да ли одговара одређеном обрасцу. То је корисно уколико треба да се имплементира функција која ради различите ствари у зависности од типова аргумената. Уобичајени примери су аритметичке операције или секвенце одређеног типа. Ово је пример са листама:

```
expr match {
       case List(0, _, _) => println("found it")
       case _ =>
}
```

Помоћу кључне речи **match** проверавамо да ли израз **expr** одговара некој листи од три елемента којој је први елемент 0 а друга два било који бројеви. Пошто се израз **expr** пореди редом са случајевима како су наведени, након неуспешног поређења са трочланом листом успешно ће се упарити са _ који служи као образац који одговара свему (енг. *wildcard pattern*).

- **Comprehensions** Конструкција **for expression** или **for comprehension** се често користи у Скали не само за стандардно итерирање кроз колекције већ и на другачије начине, на пример:

```
val forLineLengths =
```

```
for {
        file <- filesHere  /* iteriranje kroz listu fajlova */
        if file.getName.endsWith(".scala")  /* prvi uslov */
        line <- fileLines(file)  /* iteriranje kroz linije fajla */
        trimmed = line.trim  /* dodela promenljivoj trimmed */
        if trimmed.matches(".*for.*")  /* drugi uslov */
} yield trimmed.length   /* povratna vrednost */
```

Овај пример обухвата пуно могућности које пружају **for** изрази. Прво, итерирање кроз листу **filesHere** користећи <- метод. Друго, филтрирање тих фајлова помоћу услова задатог у **if** изразу. Онда следи једна угњеждена петља где се врши итерација кроз линије тренутног фајла (који је испунио услов, а ако није, прелази се на следећи фајл). Након тога, чување тренутне тримоване линије у променљивој **trimmed** и још један **if** израз. Најзад, помоћу **yield** наредбе враћа се дужина тримоване линије чиме се попуњава листа **forLineLengths**. Дакле, можемо користити угњеждене петље, филтрирати резултате, чувати податке у променљивама и попуњавати колекције помоћу **yield** наредбе. Све ове опције нам омогућавају конструкције које подсећају на дефинисање елемената скупова у математици.

Ово су неке од најважнијих функционалних особина Скале. Управо њена функционална својства су кључни разлози због којих је Скала веома погодна за паралелизацију израчунавања.

2.3 Паралелизам

Глава 3

Закључак

Библиографија

[1] Ламбда рачун. https://sr.wikipedia.org/wiki/%D0%9B%D0%B0%D0%BC%D0%B1%D0%B4%D0%B0_%D1%80%D0%B0%D1%87%D1%83%D0%BD.

[2] Саша Малков. Појам функционалних програмиских језика. http://poincare.matf.bg.ac.rs/~smalkov/files/fp.r344.2017/public/predavanja/FP.cas.2017.01.Uvod.p2.pdf.

[3] Милена Вујошевић Јаничић. Programske paradigme: Funkcionalna paradigma. http://www.programskijezici.matf.bg.ac.rs/ppR/2018/predavanja/fp/funkcionalno_programiranje.pdf.

[4] Funkcionalno programiranje. https://sr.wikipedia.org/sr/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D0%BD%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8%D1%80%D0%B0%D1%9A%D0%B5.

[5] Kombinatorna logika. https://sh.wikipedia.org/wiki/Kombinatorna_logika.

[6] Lisp (programski jezik). https://sr.wikipedia.org/wiki/Lisp_(programski_jezik).

[7] Martin odersky: Biography and current work. https://people.epfl.ch/martin.odersky/bio?lang=en&cvlang=en.

[8] Skala(programski jezik). https://sr.wikipedia.org/sr-el/%D0%A1%D0%BA%D0%B0%D0%BB%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%81%D0%BA%D0%B8_%D1%98%D0%B5%D0%B7%D0%B8%D0%BA). Accessed: 2018-06-01.

[9] Bill Venners Martin Odersky, Lex Spoon. *Programming in Scala*. Artima Press, Walnut Creek, California, 2 edition, 2011.

[10] Martin Odersky. Scala's prehistory. https://www.scala-lang.org/old/node/239.html, 2008.

Биографија аутора

Вук Стефановић Караџић (*Тршић, 26. октобар/6. новембар 1787. — Беч, 7. фебруар 1864.*) био је српски филолог, реформатор српског језика, сакупљач народних умотворина и писац првог речника српског језика. Вук је најзначајнија личност српске књижевности прве половине XIX века. Стекао је и неколико почасних доктората. Учествовао је у Првом српском устанку као писар и чиновник у Неготинској крајини, а након слома устанка преселио се у Беч, 1813. године. Ту је упознао Јернеја Копитара, цензора словенских књига, на чији је подстицај кренуо у прикупљање српских народних песама, реформу ћирилице и борбу за увођење народног језика у српску књижевност. Вуковим реформама у српски језик је уведен фонетски правопис, а српски језик је потиснуо славеносрпски језик који је у то време био језик образованих људи. Тако се као најважније године Вукове реформе истичу 1818., 1836., 1839., 1847. и 1852.