Универзитет у Београду

МАТЕМАТИЧКИ ФАКУЛТЕТ

Семинарски рад из предмета
МЕТОДОЛОГИЈА СТРУЧНОГ И НАУЧНОГ РАДА

Тема:

ИМПЛИЦИТНО ПРОГРАМИРАЊЕ

Студент:
Стана Обренић
1120/2012

Професор:
Владимир Филиповић

Јун 2013, Београд

Садржај

Увод

Програмирање је сложен процес који од формулације проблема води до програма који га решава. Обухвата разне активности: детаљну анализу проблема, разумевање и решавање чији је резултат неки алгоритам и имплементацију. Алгоритам је прецизно описан поступак како обавити неку радњу или опис решења одређеног проблема. Програмерске активности укључују како се чини неке неизбежне аспекте, као што је писање циља рачунања и главне идеје решавања проблема, већина тога је досадно и ниског нивоа. Претварање неког проблема у облик прихватљив рачунару се састоји од више корака, почев од описа и анализе проблема (**Шта** имам? **Шта** желим?), затим одређивање поступка за решавање проблема (**Како** то урадити?), и потом запис поступка, односно програмирање у неком програмском језику. Захтева се експлицитно навођење и разраде концептуално једноставних корака, и на крају то захтева много више напора него по првој процени. Као резултат, чак и једноставни задаци захтевају професионалца и експерта, чије програмерске вештине су у великом контрасту у ондосу на друге кориснике рачунара. У деценији са билионима корисника рачунара, умањење ове разлике учинило би програмирање лакшим и доступнијим.

Имплицитно програмирање је предлог парадигме за развој софтвера чији је циљ уклањање уских грла у програмирању. Проблем на који се циља да се уклони је тај што је програмирање теже је него што би требало да буде. Имплицитна програмска парадигма има за циљ умањење тих разлика тако што се праве софтвери чија је контструкција далеко лакша на више нивоа, од нових декларативних програмерских језика до нових алата за развој софтвера. У овом раду представљени су неки алати и интерфејси за манипулацију апликацијама који показују примене имплицитног програмирања, развијени на ЕПФЛ (фран. *École polytechnique fédérale de Lausanne*).

Имплицитно програмирање

Имплицитно програмирање спада у програмерски модел високог нивоа. Као додатак традиционалним конструкцијама, имплицитне спецификације дају својства резултата, не и како се долази до њега.

Углавном је тренутна пракса у програмирању да се експлицитно пише програмски код, а имплицитно дизајн програма да се ради (на пример, подешавање својства код објеката). Циљ је да се та пракса обрне, да програмирање буде имплицитно, јер као такво лакше је за разумевање, израженије је и лакше за тумачење исправности. Тиме се не повећава обим посла, чак се и смањује доста у неким случајевима. Добија са на тачности урађеног програма, јер се редовно прати резултат и дизајн.

Погледајмо та два приступа решавању проблема из другог угла. Са једне стране имамо израчунавање $F(x) = y^?$, то су данашњи програмски језици (императивни и функционални језици). Код њих ми задајемо функцију и дате вредности и на основу тога добијемо неки излаз. Са друге стране имамо $F(x^?) = y$, односно од софтвера тражимо програм који задовољава неку

спецификацију, какве особине излаз треба да има, али не и функцију. Такво имплицитно израчунавање је софтвер будућности.

Циљ ове парадигме је помоћ при конструкцији софтвера који ради оно што се очекује. Једна од потешкоћа програмирања је та што се израчунавања дају експлицитно, односно програмирањем се одговара на питање „**КАКО** решити дати проблем**?“.** Имплицитна израчунавања дају одговор на „**ШТА** треба урадити? Који резултат се очекује?“ чиме се програмирање олакшава.

Кључна техника на којој се заснива имплицитно програмирање је **аутоматско резоновање**, затим синтеза процедура, а поред њих веома су битне и енгл. *runtime* библиотеке, управљање развојним окружењем графичким путем...

У **верификацији софтвера** се могу користити предности имплицитног програмирања. Потребно је наћи улаз за који дати програм не ради или докажи да таквог улаза нема. За неке функционалности програма морају се прво имплементирати, па потом верификовати, тренутно и нема другог избора. Потребно је да се направе спецификације у унутрашњем делу програма, што помаже и верификацији и продуктивности.

Имплицитно програмирање може да се примени и на следећим активностима:

- **Развој са неким интегрисаним развојним окружењем - ИДЕ** (енгл. *Integrated Development Environment*)(Eclipse, Visual Studio, emacs, vim): *InSynh*
- **Компилација и статично проверавање** (оптимизовани компајлери за језике, статични анализатори): *Comfusy* и *RegSy*
- **Извршавање на (виртуалној) машини**: *ScalaZ3* и *UDITA* (језик који се користи за генерисање тестова и тражење багова у *Sun javac, Eclipse, netBeans, JPF)*

У наставку су укратко описани неки од ових алата, као и погодности које они нуде.

Scala Z3

Z3 је *Microsoft*-ов ефикасан СМТ (енгл. *Satisfiability Modulo Theories*) решавач. Доказује теорије коришћењем *DPLL(T)* и Нелсон-Опен процедуре. *Scala* је програмски језик који комбинује парадигме функционалног и објектно-оријентисаног програмирања. Настао је да би се програмирање учинило ефикасније, елегантније и безбедније. Покреће на Јава виртуалној машини.

Scala Z3 је систем који интегрише СМТ решавач *Z3* и програмски језик *Scala*, нуди ефикасно решавање проблема.

Нека на пример имамо функцију *sekundeUVreme* која дати број секунди претвара у сате, минуте и секунде.

```
def sekundeUVreme(ukupnoSekundi: Int) : (Int, Int, Int) =
        choose((h: Var[Int], m: Var[Int], s: Var[Int]) => (
                h*3600 + m*60 + s == ukupnoSekundi
                &&  0<=h && 0<=m && m<60 && 0<=s && s<60 ))
За пример 3787 секунди → exec(Z3 ”h*360+…”)
                                        Sat
                                        model: h=1, m=3, s=7
```

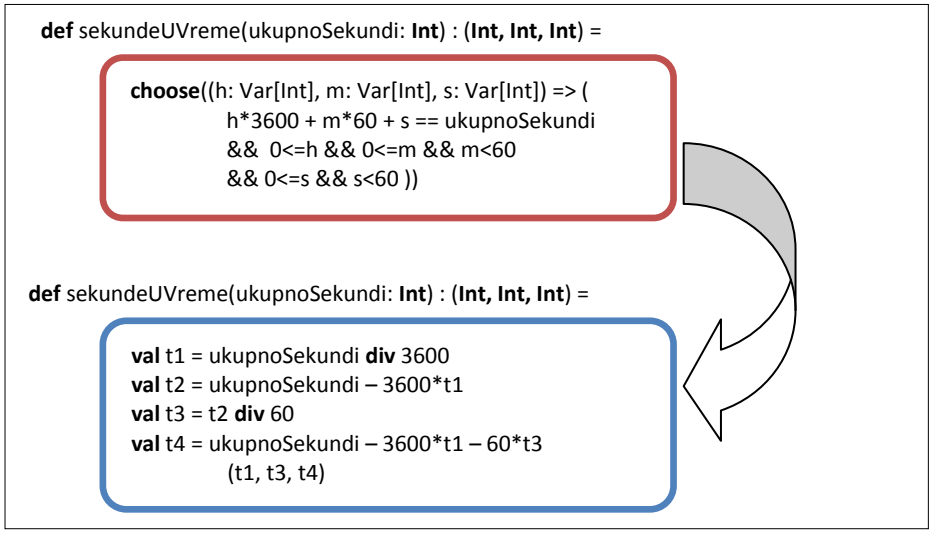
```
def choose (C: T=> boolean) : T = find(C) match{
 case Some(v) => v
 case None() => throw new Exception(“no solution”)
}

На пример: println(choose(x=>(x*x==a)))    // штампаће x такво да је x*x=a
```

choose је део програма и он покушава да нађе један елемент који испуњава захтев и враћа изузетак ако не успе. Поред њега постоје уграђени и *find* и *findAll* и све три методе су у уској вези са *Z3* решавачем. Позивом *sekundeUVreme,* позива се СМТ решавач који враћа модел за дати проблем ако постоји.

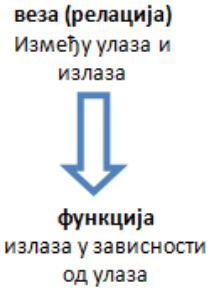
Овакав приступ ради за ограничења за које је СМТ решавач комплетан и обезбеђује генерисање модела.

Comfusy

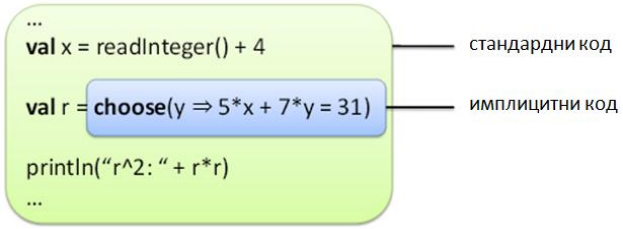
Претходном проблему може да се приступи и током компилације.



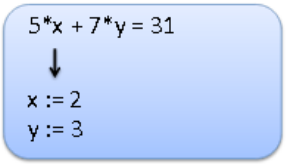
Comfusy програм је пример комплетне функционалне синтезе, за процедуру која почиње од неке имплицитне спецификације рачуна функцију која задовољава дату везу између улаза и излаза.

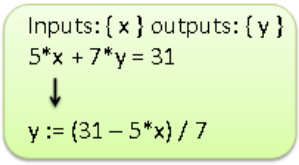


Односно, ако имамо ограничење (везу) између улаза и излаза, из тога може да се добије функција која у зависности од улаза даје излаз.



Scala Z3 добија формулу и враћа модел те формуле, а *Comfusy*, тј. процедура синтезе добија формулу са улазом (параметри) и излазом (променљиве), долази до формула за израчунавање вредности променљивих у зависности од параметара. У *Scala Z3* за пример процедуре *sekundeUVreme* проблем настаје када треба много пута да се позива, СМТ решавач ће сваки пут тражити нови модел. Код процедура синтезе време за израчунавање се троши у току компилације, решава се једном и добија се одлучујућа процедура. Погледајмо на примеру разлику:





Лево је пример како *Scala Z3* помоћу решавача долази до модела, десно је на који начин *Comfusy* долази до истог решења

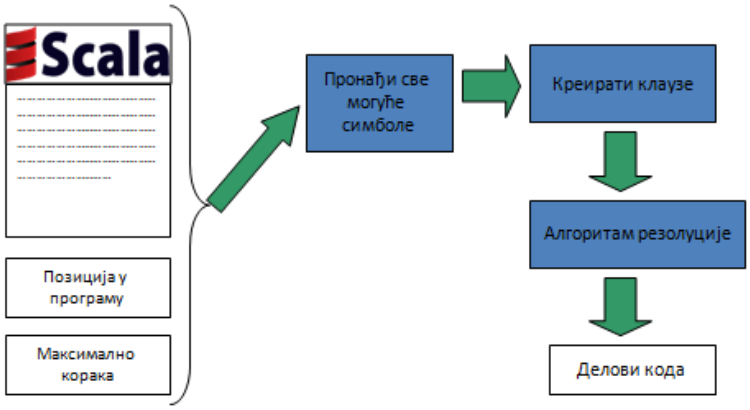
```
val vreme =
    for (sek ← timeStats)
        yield sekundeUVreme(sek)
```

Пример када може бити неефикасно позивање функције у току извршавања у *Scala Z3*

InSynth - интерактивна синтеза делова кода

Као што је познато, у интегрисаним развојним окружењима има опција **аутоматског допуњавања**, односно предвиђање израза које корисник жели да откуца. *InSynth* је алат који нам нуди бољу искоришћеност те опције тако што прецизније предвиђа шта је даље програмер наумио да куца.

InSynth за одговарајући улаз који се састоји од дела програма са видљивим декларацијама и позицијом где жели да се допуни код, даје излаз у облику израза са одговарајућим типом. До могућих израза се долази алгоритмом синтезе заснованом на логици првог реда.



Поступак добијања могућих израза од *InSynth*

На слици испод видимо пример које изразе InSynth предлаже за неку линију у коду.

```
import java.io._

object Main {
  def main(args:Array[String]) = {

    var body = "email.txt"
    var sig = "signature.txt"

    var inStream:SequenceInputStream =

    var eof:Boolean = false;
    var byteCount:Int = 0;
    while (!eof) {
      var c:Int = inStream.read()
      if (c == -1)
        eof = true;
      else {
        System.out.print(c.toChar);
        byteCount+=1;
      }
    }
    System.out.println(byteCount + " bytes were read");
    inStream.close();
  }
```

```
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(sig))
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(body))
new SequenceInputStream(new FileInputStream(body), new FileInputStream(sig))
new SequenceInputStream(new FileInputStream(body), new FileInputStream(body))
new SequenceInputStream(new FileInputStream(sig), System.in)

Press 'Ctrl+Space' to show Default Proposals
```

Конкретан пример могућих израза које је понудио *InSynth* у неком делу кода

Тренутна верзија програма је ***InSynth2,*** развијена за ***Eclipse Scala IDE***.

Pong designer: програмирање игрице показивањем

Једна од апликација која користи предности имплицитног програмирања је и *Pong designer.*

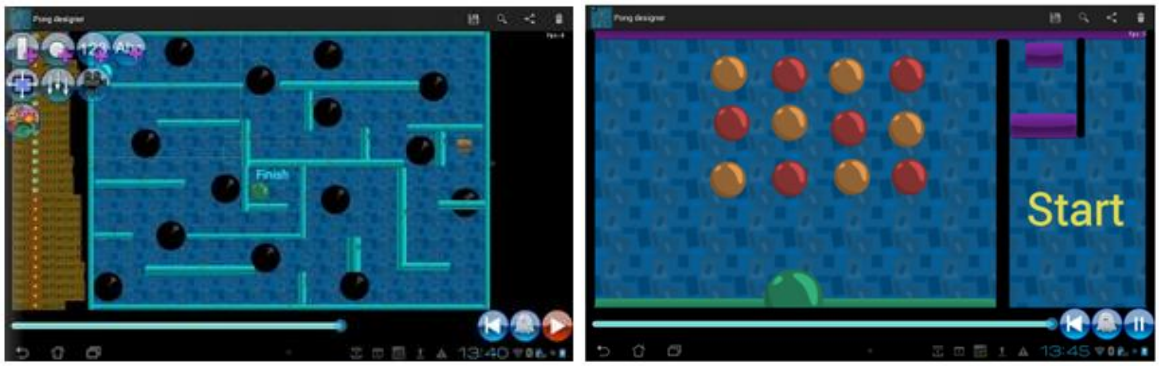
Pong designer је настао као помоћ крајњим развијаоцима софтвера. То је окружење за креирање *2D* игрица. Идеја је да се програмира показивањем, односно директним управљањем објектима, као и њиховим понашањима.

Имплементирана је у *Scala* програмском језику и може се инсталирати на таблетима са *Android* платформом који имају енгл. *multi touch* екран.

Ова игра може да се преузме са *Google store*, где и у опису апликације пише *„хиљаду игара у једном*“ баш због те могућности да се без много муке може променити игра додавањем нових објеката, а да при томе није потребно посебно знање из програмирања.

Пример додавања новог објекта у *PongDesigner*

Нажалост, нисам успела да испробам ову апликацију из техничких разлога, тако да Вам не могу пренети искуство из прве руке. Можете видети на сликама како изгледа окружење.



Примери игара које су креиране у *PongDesigner* апликацији, спремне за почетак

Уз *Pong designer* је изванредно лако креирати игрице као што су понг, пекмен и многе сличне њима.

Закључак

У данашње време имамо десетине милиона корисника који користе паметне телефоне и таблете, односно крај себе имају јака средства за рачунање, комуникацију, итд. Нажалост, иако се број корисника јаких уређаја драстично повећава, број развијаоца је још увек недовољан, што представља уско грло у развоју софтвера. Једна од баријера потенцијалним развијаоцима су постојећа програмска окружења која захтевају детаљан поступак решавања неког проблема. У овом раду показане су неке од могућности примене имплицитног програмирања и начин на које такво програмирање може да олакша процес развоја неког софтвера. Можда ова парадигма помогне у порасту броја развијаоца програма и постане заиста основа софтвера за развој софтвера у будућности.

Референце

[1] http://mir.cs.illinois.edu/udita
[2] http://www.epfl.ch/
[3] http://lara.epfl.ch/w/impro
[4] http://lara.epfl.ch/w/comfusy
[5] http://lara.epfl.ch/w/scalaz3
[6] http://lara.epfl.ch/w/insynth
[7] http://lara.epfl.ch/w/pong
[8] https://play.google.com/store/apps/details?id=ch.epfl.lara.synthesis.kingpong