Kompilacija funkcionalnih jezika - Naslov nije definisan

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Marija Mijailović, Miroslav Mišljenović, Nemanja Antić, Filip Lazić
mijailovicmarija@hotmail.com, drugog (trećeg) autora

april 2018.

Sažetak

U ovom tekstu je ukratko prikazana osnovna forma seminarskog rada. Obratite pažnju da je pored ove .pdf datoteke, u prilogu i odgovarajuća .tex datoteka, kao i .bib datoteka korišćena za generisanje literature. Na prvoj strani seminarskog rada su naslov, apstrakt i sadržaj, i to sve mora da stane na prvu stranu! Kako bi Vaš seminarski zadovoljio standarde i očekivanja, koristite uputstva i materijale sa predavanja na temu pisanja seminarskih radova. Ovo je samo šablon koji se odnosi na fizički izgled seminarskog rada (šablon koji *morate* da ispoštujete!) kao i par tehničkih pomoćnih uputstava. Molim Vas da kada budete predavali seminarski rad, imenujete datoteke tako da sadrže temu seminarskog rada, kao i imena i prezimena članova grupe (ili samo temu i prezimena, ukoliko je sa imenima predugačko). Predaja seminarskih radova biće isključivo preko web forme, a NE slanjem mejla.

Sadržaj

1 Uvod

Mnogi kompajleri obavljaju neki deo svog rada pomoću očuvanja ispravnosti, poboljšanja performansi, programskih transformacija. Radi konkretnosti, fokusirali smo se na Glasgov Haskel Kompajler (eng.*Glasgov Haskell Compiler(GHC)*),ali ističemo da sve što navedemo u ovom radu je primenljivo i za bilo koji kompajler za funkcionalni jezik, a možda i na kompilaciju za druge jezike. GHC preuzima ovu ideju "kompilacije transformacijom" pokušavajući da što više izrazi proces kompilacije u obliku programskih transformacija. U ovom radu prikazaćemo u primeni transformacione tehnike kroz GHC kompajler za funkcionalni jezik Haskel(eng. *Haskell*).

1.1 Haskel

Haskel je funkcionalni jezik opšte namene, koji sadrži mnoge inovacije u dizajnu programskih jezika. Haskel podržava funkcije višeg reda (eng. *higher-order functions*), ne-striktnu semantiku, statički polimorfizam, korisnički definisane algebarske tipove, prepoznavanje šablona (eng. *pattern-matching*), rad sa listama. Razvijen je kroz sistem modula kao proširenje programskog jezika. Poseduje veliki skup primitivnih tipova uključujući liste, nizove, cele brojeve različitih namena i dužina, realne brojeve. Može se reći da je Haskel završetak dugogodišnjeg rada i istraživanja na ne-striktnim funcionalnim jezicima.

1.2 GHC

Uobičajno se tretira kao standardna interpretacija Haskela, na kojoj se baziraju i druge implementacije. Razvijan je počev od 1989. godine. GHC je pisan u Haskelu – kompajler sadrži 227,000 linija koda (uključujući komentare), a biblioteke (moduli) sadrže 242,000 linija koda (uključujući komentare).

Sastavni deo svakog kompajlera za funkcionalni jezik čini i run-time sistem. Za GHC, run-time sistem je pisan u C-u i sadrži 87,000 linija koda.

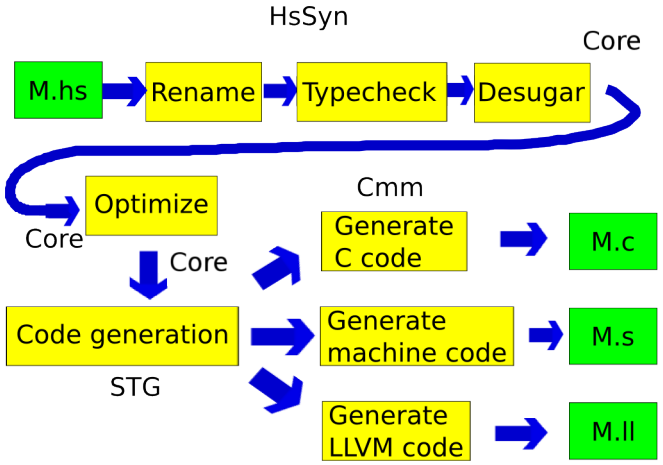
Tekuću verziju kompajlera, razvijalo je 23 istraživača (eng. *developers*) sa preko 500 priloga (eng. *commits*).

Celokupna struktura kompajlera se sastoji iz tri faze :

1. Prva faza (eng. *Frontend*) čini pretvaranje izvornog koda pisanog u Haskelu, u takozvani Core jezik (eng. *Core language*). U ovoj fazi, kompajler napisan u Haskelu, vrši analizu izvornog koda, pravi odgovarajuće drvo izvođenja, leksičku i sintaksnu analizu, proveru tipova i kao izlaz daje kod sa veoma redukovanim skupom instrukcija, koji zahteva Core. Više o ovoj fazi može se naći u Sekciji 2.
2. Druga faza (eng. *Middle End*) izlazni kod iz prethodne faze – međujezik (eng. *Intermediate language)*) se dodatno optimizuje, i to kroz niz transformacija. Rezultat rada Core-a se dobija u Core obliku. Na taj način, sledeće etape transformacije koda tačno znaju kakve konstrukcije im se mogu naći na ulazu. Više o ovoj fazi može se naći u Sekciji 3.
3. Treća faza (eng.*Backend*) obuhvata generisanje koda. Core-ov kod iz prethodne faze postaje ulaz u STG-mašinu (eng. *Spineless Tagless G-machine*). Rezultat rada STG-mašine se grana u tri toka. Prvi

tok podrazumeva prevodjenje u izvorni mašinski kod. Drugi tok podrazumeva korišćenje LLVM-a za dobijanje optimizovanog koda pomoću ove virtualne mašine. Treći tok koristi jezik C– za dobijanje izvršnog koda. Više o ovoj fazi može se naći u Sekciji 4.

Razvojne etape pravljenja izvršnog koda mogu se videti na slici 1



Slika 1: Razvojne etape pravljenja izvršnog koda

2 Front end

Kao sto smo rekli, Front end je prva faza u kojoj se izvorni kod pretvara u Core jezik. Sastoji se od :

1. Parsiranja(eng. *Parser*)
2. Promena imena(eng. *Rename*)
3. Provera tipa(eng. *Typecheck*)
4. Prečišćavanja(eng. *Desugaring*)

2.1 Parsiranje

Pravljenje preciznog parsera u konkretnom jeziku je jako teško. GHC-ov parser se služi sledećim principom :

Često se parsira "previše velikodušno", a zatim odbacujemo loše slučajeve.

Paterni su parsirani kao izrazi i transformisani iz *HsExpr.HsExp* u *HsPat.HsPat*. Izraz kao što je

```
[ x | x <- xs]
```

koji ne izgleda kao patern je odbijen.

Ponekad "previše velikodušno" parsiranje izvršava samo "renamer". Na primer: Infkiksni operatori su parsirani kao da su svi levo asocijativni. "Renamer" koristi dekleracije ispravnosti za ponovno povezivanje sintaksnog stabla. Dobra karakteristika ovog pristupa je to da poruke o grešci kasnije tokom kompilacije imaju tendenciju da pruže mnogo korisnije informacije. Greške generisane od strane samog parsera imaju tendenciju samo da kažu da se greška desila na odredjenoj liniji i ne pružaju nikakve dodatne informacije.

2.2 Promena imena

Osnovni zadatak Renamer-a je da zameni RdrNames sa Names. Na primer, imamo:

```
module K where
f x = True

module N where
import K

module M where
import N( f ) as Q
f = (f, M.f, Q.f, \f -> f)
```

U kome su sve promenljive tipa RdrName. Rezultat preimenovanja modula M je :

```
M.f = (M.f, M.f, K.f, \f_22 -> f_22)
```

Gde su sada sve promenljive tipa Name.

- Nekvantifikovani RdrName "f" na najvišem nivou postaje spoljašnji Name M.f.
- Pojavljivanja "f" i "M.f" su zajedno vezane za ovo Name.
- Kvantifikovani "Q.f" postaje Name "K.f" , zato što je funkcija definisana u modulu K.
- Lambda "f" postaje unutrašnji Name, ovde napisan f_22.

Pored ovoga, renamer radi i sledeće stvari:

- Vrši analizu zahteva za uzajmno rekurzivne grupe deklaracija. Ovo deli dekleracije u snažno povezane komponente.
- Izvršava veliki broj provera grešaka : promenljive van opsega, neiskorišćene biblioteke koje su uključene,..
- Renamer se nalazi izmedju parsera i typechecker-a, ipak, njegov rad je isprepletan sa typechecker-om.

2.3 Provera tipa

Verovatno najvažnija faza u frontendu je kontrolor tipa (eng.*type checker*), koji se nalazi u compiler/typecheck/. GHC proverava programe u njihovoj orginalnoj Haskel formi pre nego što ih desugar konvertuje u Core kod. Ovo umnogome komplikuje type checker ali poboljšava poruke o grešci.

GHC definise apstraktu sintaksu Haskel programa u compiler/hsSyn/hsSyn/HsSyn.hs koristeći strukture koje apstraktuju konkretnu reprezentaciju graničnih pojavljivanja identifikatora i paterna.

Interfejs type checker-a ostatku kompajlera obezbeđuje compiler/typechecj/TcRnDriver.hs. Svi moduli se izvršavaju zvanjem tcRnModule, i GHCI koristi tcRnStmt, tcRnStmt, tcRnExpr i tcRnType za proveru iskaza, izraza i tipova redom.

Funkcije tcRnModule i tcRnModuleTcRnM kontrolišu kompletnu statičku analizu Hasklel modula. Oni razrešavaju sve import iskaze, inicira stvarni postupak preimenovanja i provere tipa i završava sa obradom izvoza(eng. *Export list*).

Reprezentacija tipova je fiksirana u modulu TypeRep i eksportovana kao podatak tipe Type.

2.4 Prečišćavanje

Prečišćavanje prevodi iz masivnog HsSyn tipa u GHC-ov medjujezik CoreSyn. Obicno se prečišćavanje programa izvršava pre faza povere tipa, ili preimenovanja, jer to onda olakšava posao renamer-u i typechecker-u jer imaju mnogo manji jezik za obradu.

2.5 Core jezik

Lepo bi bilo da pre nego što krenemo o optimizaijama napišemo nešto ukratko o Cor-u, ovo treba doterati

Core jezik se sastoji od nekoliko elemenata: variables, literals, let, case, lambda abstraction, application. Uopšteno govoreći, naredba let odgovara alokaciji, naredba case odgovara evaluaciji. Osnovna ideja Core-a je da se napravi jednostavan tipizirani lambda račun, sa najmanjim brojem konstrukcija koju obuhvataju izvorni jezik. Na taj način se jednostavnije analizira kod, razmišlja o njemu, vrši optimizacija itd. Dakle, Core je jednostavan lenji funcionalni jezik; možemo ga smatrati asemblerom funkcionalnog jezika.

Jedna od karakteristika Core-a je parcijalna evaluacija. Zbog mogućnosti lenjog izračunavanja funkcija, često dolazimo u situaciju da nemamo sve argumente na raspolaganju u trenutku izvršavanja funkcije. Tada se prekida izvršavanje funkcije dok se ne dobiju svi potrebni argumenti, nakon čega se izvrši funkcija.

3 Middle end

U ovoj fazi izlazni kod iz prethodne faze – Core jezik se dodatno optimizuje Postoje mnoge metode optimizacije koje implementira Core jezik. Tako, na primer, primenjuje se umetanje koda (eng. *Inlining*), eliminacija zajedničkih podizraza (eng. Common Subexpression Elimination), kao i izbacivanje koda koji se ne koristi (eng. Dead Code Elimination).

Interesantno je napomenuti, da se umetanjem koda, u funkcionalnim jezicima dobija 20 – 40% ubrzanja, dok se kod imperativnih jezika, istom medotom postiže ubrzanje od 10 - 15%. Stoga ćemo u ovom delu detaljije opisati rad Inlinera, kao vodećeg igrača u poboljšanju rada kompajlera. GHC Inliner pokušava učiniti što je više moguće umetanje u jednom prolazu. Pošto umetanje često otkriva nove mogućnosti za dalje transformacije, Inliner je zapravo deo GHC Simplifikatora(eng. *GHC simplifier*), koja obavlja veliki broj lokalnih transformacije na iterativan način (ili dok se ne postigne određen broj ponavljanja).

U modernoj verziji GHC kompajlera se koristi LLVM (eng. *low-level virtual machine*) u završnoj fazi generisanja koda. LLVM u sebi ima ugrađene mnogobrojne optimizacije, tako da ako je neka preskočena u Core optimizaciji, LLVM će je obuhvatiti.

3.1 Umetanje i beta redukcija (eng. *Inlining and beta reduction*)

Utvrdili smo da je korisno identifikovati tri različite transformacije povezane sa umetanjem(eng. *inlining*):

1. Umetanje (eng. *Inlining itself*) - zamenjuje pojavu ograničene let (eng.let-bound - nemam predstavu kako bolje od ovog) promenljive definicijom sa njene desne strane. Na primer, umetanje za f izgleda:

```
let { f = \x -> x*3 } in f (a + b) - c
==> [inline f]
let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
```

Treba obratiti pažnju da umetanje nije ograničeno na definiciju funkcije, već svaka ograničena let promenljiva može potencijalno biti umetnuta. (Ipak treba imati na umu da pojava promenljive na poziciji argumenta nije kandidat za umetanje, jer su oni ograničeni tako da budu atomični.)

2. Eliminacija mrtvog koda (eng. *Dead code elimination*) - odbacuje let vezivanje koje se više ne koriste. Ovo se obično javlja kada su sve pojave promeljive umetnute. Ako primenimo eliminaciju na prethodni primer dobijamo :

   ```
   let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
   ==> [dead f]
   (\x -> x*3) (a + b) - c
   ```

3. β redukcija (eng. *Beta reduction*) - jednostavno prezapišemo lamda izraz na sledeći način:

   ```
   (\x->E) A to let {x = A} in E
   ```

   Primenom β redukcije na gornji primer:

   ```
   (\x -> x*3) (a + b) - c
   ==> [beta]
   (let { x = a+b } in x*3) - c
   ```

Dok su eliminacija mrtvog koda i β redukcija su jednostavne, umetanje je jedino nezgodno, pa nam je ono interesantnije.

3.1.1 Jednostavno umetanje (eng. *Simple inlining*)

Korisno je razlikovati dva slučaja umetanja:

1. SGNF(eng. WHNF) - Ako je promeljiva vezana za slabu glavnu normalnu formu (SGNF) (eng. weak head normal form (WHNF)) - to je atom, lambda ili konstruktor - tada se može umetnuti bez rizika od dupliranja poslanja. Možda je jedina negativna strana povećanje veličine koda.
2. Ne-SGNF(eng.Non-WHNFs) - U suprotnom, umetanje nosi rizik od gubitka razmene, a samim tim i dupliranja rada. Na primer,

   ```
   let x = f 100 in ... x ... x ...
   ```

   možda bi bilo neupotrebljivo za umetanje x, jer bi tada f 100 bio ocenjeno dva puta umesto jednom. Neformalno, kažemo da je transformacija S-sigurna(eng. W-safe) ako garantuje da ne duplira posao.

U slučaju SGNF-a(eng WHNFs), kompromis je između veličine koda i koristi od umetanja. Primene atoma i konstruktora su jednostavne: uvek su dovoljno male da bi se umetale. (Konstruktori moraju imati atomične argumente). Funkcije, za razliku od toga, mogu biti velike, tako da efekat neograničenih umetanja u odnosu na veličinu koda može biti znatan. Kao i većina kompajlera, koristi se heuristika za odlučivanje kada se radi umetanje funkcija.

Za Ne-SGNF-u(engl.Non-WHNFs), pažnja se fokusira na to kako se promenljiva koristi. Ako se promenljiva pojavi samo jednom, onda je verovatno sigurno da ćemo je umetnuti. Inače jedan od naivnih pristupa bi

bio da vršimo jednostavnu analizu pojava koja beleži za svaku promeljivu na koliko mesta se koristi, i koristimo ove informacije kako bi naučili simplifikator da izvrši umetanje. Medjutim ovaj način ima svoje komplikacije, pa je drugo rešenje dato u sledećoj sekciji.

3.1.2 Koristeći linearnost(eng. *Using linearity*)

Zbog komplikacija, postalo je neophodno pratiti informacije o pojavama koje su najkomplikovanije i koje su podložne bug-u. Štaviše, greške u radu sa dupliranjem se manifestuju samo kao problemi sa performansama i mogu proći neopaženo dugo vremena. To sugeriše da bi sistem linearnog tipa bi bio dobar način da se identifikuju promelnjive koje se mogu bezbedno umetnuti, iako se one javljaju unutar lambde, ili koje se ne mogu bezbedno umetnuti iako se (trenutno) pojavljuju samo jednom. S-sigurne(eng. W-safe) transformacije čuvaju i informacije o linijskom tipu, a time i garancije da se dupliranje neće vršiti.

Nažalost, većini linearnih sistema je neadekvatna jer ne uzimaju u obzir procenu potreba po pozivu (eng. *call-by-need*). Na primer, razmotrite izraz:

```
let x = 3*4
y = x+1 in y + y
```

Po proceni potreba po pozivu , iako se y procenjuje mnogo puta, x će biti procenjen samo jednom. Većna linearnih sistema bi bila previše konzervativna i pripisivala bi nelinearni tip x, kao i y, sprečavajući da x bude umetnut.

Postoji nekoliko pokušaja da se razvije sistem linearnog tipa koji uzima u obzir procenu potreba, ali nijedan nije našao svoju praktičnu prmenu. Jedan od pokušaja se može naći u [1]

3.2 Transformacija uslova (eng. *Transforming conditionals*)

Većina kompajlera ima posebna pravila za optimizaciju uslova. Na primer, razmotrimo izraz:

```
if (not x) then E1 else E2
```

Nijedan pristojan kompajler ne bi ustvari negirao vrednost x u vreme izvršavanja! Da vidimo, šta se onda događa ako jednostavno izvršimo transformacije. Nakon što smo uklonili *if* i umetnuli definiciju za *not*, dobijamo:

```
case (case x of {True -> False; False -> True}) of
True -> E1
False -> E2
```

Ovde, spoljašnji slučaj ispituje vrednost koju vraća unutrašnji slučaj. Ova zapažanja sugerišu da možemo premestiti spoljni slučaj unutar grana unutrašnjeg, tako da:

```
case x of
True -> case False of {True -> E1; False -> E2}
False -> case True of {True -> E1; False -> E2}
```

Obratite pažnju na to da je originalni spoljašnji case izraz dupliran, ali svaka kopija sada ispituje poznatu vrednost, pa je očigledno sledeće pojednostavljenje,i dobije se tačno ono čemu smo se i nadali:

```
case x of
True -> E2
False -> E1
```

Obe ove transformacije su generalno primenljive. Druga transformacija, transformacija "poznatih-konstruktora"(eng. *case-of-known- constructor*) eliminiše niz izraza koji ispituje poznatu vrednost.

3.2.1 Pridruživanje tačaka (eng. *Join points*)

Postavlja se pitanje kako možemo dobiti koristi od transformacija "case-case"(eng. *case-of-case*) bez rizika od dupliranja koda? Jednostavna ideja je napraviti lokalne definicije sa desne strane spoljnjeg slučaja, ovako:

```
case (case S of {True -> R1; False -> R2}) of
True -> E1
False -> E2
=>
let e1 = E1; e2 = E2
in case S of
True -> case R1 of {True -> e1; False -> e2}
False -> case R2 of {True -> e1; False -> e2}
```

Sada E1 i E2 nisu duplirani, iako umesto toga imamo troškove implementacije vezivanja za e1 i e2. U primeru, međutim, dva unutrašnja slučaja se eliminišu, ostavljajući samo jednu pojavu svakog od e1 i e2, tako da će njihove definicije biti umetnute, ostavljajući isti rezultat kao i ranije.

Sigurno ne možemo garantovati da će novo uvedena vezivanja biti eliminisati. Razmotrimo, na primer, izraz:

```
if (x || y) then E1 else E2
```

Ovde, || je operator disjunkcije, definiše se:

```
|| = \ a b -> case a of {True -> True; False -> b}
```

Odstranjivanje(eng. *Desugaring*) uslova i umetanje || daje nam:

```
case (case x of {True -> True; False -> y}) of
True -> E1
False -> E2
```

Sada primenjujući (novu) "case-case" transformaciju:

```
let e1 = E1 ; e2 = E2
in case x of
True -> case True of {True -> e1; False -> e2}
False -> case y of {True -> e1; False -> e2}
```

Za razliku od *not* primera, samo jedan od dva unutrašnja slučaja je simplifikovan, tako da će samo e2 sigurno biti umetnut, jer e1 se i dalje pominje dvaput:

```
let e1 = E1
in case x of
True -> e1
False -> case y of {True -> e1; False -> E2}
```

3.2.2 Objedinjavanje pridruženih tačaka(eng. *Generalising join points*)

Da li se sve ovo delo generalizuje na tipove podataka koji nisu booleani? Na prvom mestu može se pomisliti da je odgovor "da, naravno", ali ustvari izmenjena "case-case" transformacija je jednostavno besmislena ako izvorno spoljašnji case izraz povezuje bilo koju promeljivu. Na primer, razmotrite izraz:

```
f (if b then B1 else B2)
```

gde f definišemo:

```
f = \ as -> case as of {[] -> E1; (b:bs) -> E2}
```

Prečišćavanje(eng. Desugaring) if uslova i umetanje f daje nam:

```
case (case b of {True -> B1; False -> B2}) of
[] -> E1
(b:bs) -> E2
```

Sada, pošto E2 može ukljuciti b i bs, ne možemo vezati novu promenljivu e2 kao što smo ranije radili! Rešenje je jednostavno vezati funkciju e2 koja uzima b ili bs kao svoje argumente. Pretpostavimo, na primer, da E2 pominje bs, ali ne b.Zatim možemo izvršiti "case-case" transformaciju tako da:

```
let e1 = E1; e2 = \ bs -> E2
in case b of
True -> case B1 of {[] -> e1; (b:bs) -> e2 bs}
False -> case B2 of {[] -> e1; (b:bs) -> e2 bs}
```

Odavde prevazilazi da celokupna podešavanja funkcionišu za proizvoljne tipove podataka od strane korisnika, a ne samo za booleane.

3.2.3 Objedinjavanje case eliminacija (eng. *Generalising case elimination*)

Ranije smo raspravljali o slučaju transformacije "poznatih-konstruktora"(eng. *case-of-known-constructor*) koja eliminiše niz izraza.

Postoji korisna varijanta ove transformacije koja takođe eliminiše niz izraza. Razmotrimo izraz:

```
if null xs then r else tail xs
```

gde su null i tail definisani na sledeći način:

```
null = \ as -> case as of {[] -> True; (b:bs) -> False}
tail = \ cs -> case cs of {[] -> error "tail"; (d:ds) -> ds}
```

Nakon uobičajenog umetanja, dobijamo:

```
case (case xs of {[] -> True; (b:bs) -> False}) of
True -> r
False -> case xs of
[] -> error "tail"
(d:ds) -> ds
```

Sada možemo da odradimo "case-case" transformaciju, i dobijamo:

```
case xs of
[]-> r
(b:bs) -> case xs of
[] -> error "tail"
(d:ds) -> ds
```

Sada je očito jasno da je unutrašnja procena xs redundantna, jer u (b:bs) grani spoljašnjeg case slučaja znamo da xs svakako ima oblik (b:bs)! Stoga možemo eliminisati unutrašnji slučaj, biramo alternativu (d:ds), ali supstitucijom b za d i bs za ds:

```
case xs of
[]-> r
(b:bs) -> bs
```

4 Back end

Slike i tabele treba da budu u svom okruženju, sa odgovarajućim naslovima, obeležene labelom da koje omogućava referenciranje.

Primer 4.1 *Ovako se ubacuje slika. Obratiti pažnju da je dodato i*

`\usepackage{graphicx}`

Slika 2: Pande

Na svaku sliku neophodno je referisati se negde u tekstu. Na primer, na

Primer 4.2 *I tabele treba da budu u svom okruženju, i na njih je neophodno referisati se u tekstu. Na primer, u tabeli*

Tabela 1: Razlčita poravnanja u okviru iste tabele ne treba koristiti jer su nepregledna.

centralno poravnanje	levo poravnanje	desno poravnanje
a	b	c
d	e	f

5 Zaključak

Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak.

Literatura

[1] Andre L M Santos Simon L Peyton Jones. On Computable Numbers, with an application to the Entscheidungsproblem. *Science of Computer Programing*, 32/1:3–47, 1997.

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe.