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. Glasgov Haskel Kompajler (eng. *G*lasgov Haskell Compiler(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. *H*askell).

1.1 Haskel

Haskel je funkcionalni jezik opšte namene, koji sadrži mnoge inovacije u dizajnu programskih jezika. Haskel podržava funkcije višeg reda (eng.*h*igher-order functions), ne-striktnu semantiku, statički polimorfizam, korisnički definisane algebarske tipove, prepoznavanje šablona (eng. *p*attern-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).

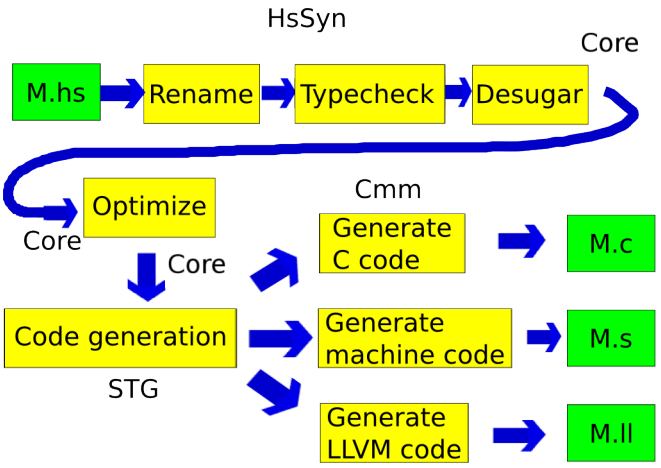
Sastavni deo svakog kompajlera za funkcionalni jezik čini i run-time sistem. Za GHC, run-time sistem je pisan u C-u i sadrži 87,000 linija koda.

Tekuću verziju kompajlera, razvijalo je 23 istraživača (eng. *d*evelopers) sa preko 500 priloga (eng. *c*ommits).

Celokupna struktura kompajlera se sastoji iz tri faze :

1. Prva faza (eng. *F*rontend) čini pretvaranje izvornog koda pisanog u Haskelu, u takozvani Core jezik (eng. *C*ore 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. *M*iddle End) izlazni kod iz prethodne faze – međujezik (eng. *I*ntermediate 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. *B*ackend) obuhvata generisanje koda. Core-ov kod iz prethodne faze postaje ulaz u STG-mašinu (eng. *S*pineless 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
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 sto 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

3 Middle

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

Primer 3.1 *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 3.2 *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

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 3: 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 2: 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.

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.