Etape prevođenja Haskela do mašinskog jezika

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

Marija Mijailović, Miroslav Mišljenović, Nemanja Antić, Filip Lazić
mijailovicmarija@hotmail.com, mr12260@alas.matf.bg.ac.rs,
antic.cubaka@gmail.com, filipl41@yahoo.com

april 2018.

Sažetak

Funkcionalni jezici ranije nisu bili popularni, pre svega zbog njihove manje upotrebe u industriji, a i zbog razvijenog mišljenja da su teški za učenje. Danas se oni sve više koriste i u neprekidnom su usponu, najviše u kritičnim delovima koda, jer je mnogo lakše dokazati tačnost koda napisanog u funkcionalnom jeziku. U ovom radu, predstavićemo osnovne teorijske koncepte na kojima leže implementacije kompajlera funkcionalnih jezika, sa akcentom na Haskelu. Najviše se koncentrišemo na proces prevođenja iz funkcionalnog koda u međujezik, i detaljno predstavljamo sve korake koji se nalaze u procesu prevođenja.

Sadržaj

1 Uvod

Funkcionalna paradigma danas predstavlja jedan od najznačajnijih stilova jezika i u širokoj je upotrebi kako u akademskim tako i u industrijskim krugovima. Funkcionalna paradigma se zasniva na matematičkim funkcijama, eliminiše bočne efekte i tako umnogome olakšava dokazivanje korektnosti programa. Džon Bakus je 1977. godine dobio Tjuringovu nagradu za doprinos u razvoju jezika FORTRAN, a u govoru, prilikom prijema nagrade, izneo je niz argumenata zbog čega su funkcionalni jezici bolji od imperativnih. Da bi potkrepio svoje tvrdnje predstavio je funkcionalni jezik FP.

Prilikom izučavanja svake programske paradigme, velika pažnja posvećuje se prevodenju izvornog koda do izvršnog koda, to jest kompajliranju(eng. *compile*). Kompajliranje funkcionalnih jezika, zbog njihovog dizajna, ima posebna svojstva. U ovom radu fokus će biti na procesu transformacije od izvornog koda do mašinskog jezika, stoga za bolje razumevanje poželjno je da se čitaoc dobro upozna sa nekim osnovnim elemtima funkcionalnog jezika koje su date u radu [1] . Detaljno ce biti opisane 3 glavne faze : Frontend, Middle end i backend. Radi konkretnosti, fokusirali smo se na Glazgov Haskel Kompajler (eng. Glasgow Haskell Compiler(GHC)) [10], ali ističemo da sve što navedemo u ovom radu se odnosi i na bilo koji kompajler funkcionalnih jezika, a možda i na kompajliranje drugih jezika. GHC preuzima ovu ideju "kompilacije transformacijom" (eng. *compilation by transformation*) pokušavajući da što više izrazi proces kompilacije u obliku programskih transformacija, iz razloga što automatke transformacije znatno poboljšavaju performanse generisanog koda. U ovom radu prikazaćemo primenu transformacione tehnike kroz GHC.

2 Uvod u Haskel i GHC

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 proširenja 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 funkcionalnim jezicima [3]. Baš zbog ovih specijalnih odluka u dizajnu, problem kompilacije kod Haskela, a i funkcionalnih jezika uopšte se smatra veoma teškim.

GHC se uobičajno tretira kao standardna implementacija Haskela, na kojoj se baziraju i druge implementacije. Razvijan je počev od 1989. godine. GHC je pisan u Haskelu – kompajler sadrži 227,000 linija kôda (uključujući komentare), a biblioteke (moduli) sadrže 242,000 linija kôda (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 kôda. Tekuću verziju kompajlera razvijalo je 23 programera (eng. developers) sa preko 500 priloga (eng. commits) [14].

Celokupni rad kompajlera se sastoji iz tri faze:

1. Prva faza (eng. *Frontend*) čini pretvaranje izvornog kôda pisanog u Haskelu, u takozvani Core jezik (eng. *Core language*). U ovoj

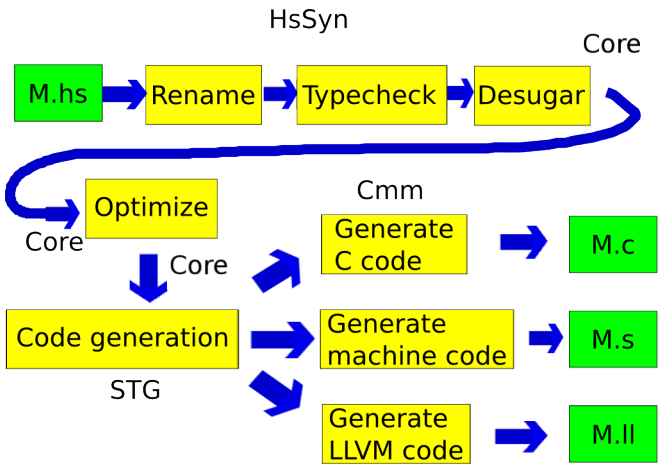
fazi, kompajler napisan u Haskelu, vrši analizu izvornog kôda, pravi odgovarajuće drvo izvođenja, realizuje leksičku i sintaksnu analizu, proveru tipova i kao izlaz daje kôd sa veoma redukovanim skupom instrukcija, koji zahteva Core. Više o ovoj fazi može se naći u poglavlju 3.

2. Druga faza (eng. *Middle End*) – izlazni kôd 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 kôda tačno znaju kakve konstrukcije im se mogu naći na ulazu. Više o ovoj fazi može se naći u poglavlju 4.
3. Treća faza (eng. *Backend*) obuhvata generisanje kôda. Core-ov kôd 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 prevođenje u izvorni mašinski kôd.

   Drugi tok podrazumeva korišćenje LLVM-a (eng. *Low-Level Virtual Machine*) za dobijanje optimizovanog kôda pomoću ove virtualne mašine.

   Treći tok koristi jezik C-- za dobijanje izvršnog kôda. Više o ovoj fazi može se naći u poglavlju 5.

Razvojne etape pravljenja izvršnog kôda mogu se videti na slici 1 preuzetoj sa [14].



Slika 1: Razvojne etape pravljenja izvršnog kôda

3 Frontend

Kao što smo rekli, frontend je prva faza u kojoj se izvorni kôd pretvara u Core jezik. Sastoji se od:

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

3.1 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.

Šabloni (eng. *Pattern*) su parsirani kao izrazi i transformisani iz *HsExpr.HsExp* u *HsPat.HsPat*. Izraz kao što je

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

koji ne izgleda kao patern se odbacuje.

Ponekad "previše velikodušno" parsiranje izvršava samo mehanizam za preimenovanje (eng. *renamer*).

Na primer: Infiksni operatori su parsirani kao da su svi levo asocijativni. Renamer koristi deklaracije ispravnosti za ponovno povezivanje sintaksnog stabla. Dobra karakteristika ovog pristupa je ta da poruke o greškama kasnije tokom kompilacije imaju tendenciju da pruže mnogo korisnije informacije. Greške generisane od strane samog parsera imaju tendenciju samo da ukažu na to da se greška desila na određenoj liniji i ne pružaju nikakve dodatne informacije.

3.2 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 vezani 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 deklaracije u snažno povezane komponente.
- Izvršava veliki broj provera grešaka: promenljive van opsega, neiskorišćene biblioteke koje su uključene...

Renamer se nalazi između parsera i typechecker-a; ipak, njegov rad je isprepletan sa typechecker-om.

3.3 Provera tipa

Verovatno najvažnija faza u frontend-u je mehanizam za proveru tipa (eng. *typechecker*), koji se nalazi u compiler/typecheck/. GHC proverava programe u njihovoj originalnoj Haskel formi pre nego što ih desugar 3.4 konvertuje u Core kôd. Ovo umnogome komplikuje typechecker, ali poboljšava poruke o grešci.

GHC definiše apstraktnu sintaksu Haskel programa u compiler/hsSyn/hsSyn/HsSyn.hs koristeći strukture koje apstrahuju konkretnu reprezentaciju graničnih pojavljivanja identifikatora i paterna.

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

Funkcije tcRnModule i tcRnModuleTcRnM kontrolišu kompletnu statičku analizu Haskel modula. Oni razrešavaju sve import iskaze, iniciraju stvarni postupak preimenovanja i provere tipa i završavaju sa obradom izvoza (eng. *export list*).

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

3.4 Prečišćavanje

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

3.5 Core jezik

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 koje obuhvataju izvorni jezik. Na taj način se jednostavnije analizira kôd, razmišlja o njemu, vrši optimizacija itd. Dakle, Core je jednostavan lenji funkcionalni 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 [5].

4 Middle end

U ovoj fazi izlazni kôd iz prethodne faze – Core jezik - se dodatno optimizuje. Postoje mnoge metode optimizacije koje implementira Core jezik. Tako, na primer, neke od optimizacija koje se primenjuju su umetanje kôda (eng. *inlining*) [7], kao i eliminacija zajedničkih podizraza (eng. *common subexpression elimination*).

U modernoj verziji GHC kompajlera se koristi LLVM u završnoj fazi generisanja kôda. LLVM u sebi ima ugrađene mnogobrojne optimizacije, tako da ako je neka preskočena u Core optimizaciji, LLVM će je obuhvatiti.

Kako ne bismo detaljno opisivali svaku metodu optimizacije, u ovom delu rada detaljno će biti opisane transformacije koje su povezane sa umetanjem, i transformacije uslova.

4.1 Umetanje

Interesantno je napomenuti da se umetanjem kôda, u funkcionalnim jezicima dobija 20 – 40% ubrzanja, dok se kod imperativnih jezika, istom metodom postiže ubrzanje od 10 - 15% [4]. Stoga sledi detaljniji opis rada mehanizma za umetanje (eng. *inliner*), kao vodećeg igrača u poboljšanju rada kompajlera. GHC Inliner pokušava učiniti što je više moguće umetanja u jednom prolazu. Pošto umetanje često otkriva nove mogućnosti za dalje transformacije, Inliner je zapravo deo GHC Simplifikatora (eng. *GHC simplifier*), koji obavlja veliki broj lokalnih transformacija na iterativan način (ili dok se ne postigne određen broj ponavljanja). Utvrdili smo da je korisno identifikovati tri različite transformacije povezane sa umetanjem:

1. Samoumetanje (eng. *inlining itself*) - zamenjuje pojavu ograničene let-povezane (eng. *let-bound*) promenljive definicijom sa njene desne strane. Na primer, umetanje za f izgleda, gde su a,b i c konstante:

   ```
   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 argumenti osmišljeni tako da budu atomični.)

2. Eliminacija mrtvog kôda (eng. *dead code elimination*) - odbacuje let vezivanja koja se više ne koriste. Ovo se obično javlja kada su sve pojave promenljive 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 kôda i β redukcija jednostavne, samoumetanje je jedino nezgodno, pa nam je ono interesantnije.

Korisno je razlikovati dva slučaja samoumetanja:

1. SGNF (eng. *Weak Head Normal Form (WHNF)*) - Ako je promenljiva vezana za slabu glavnu normalnu formu (SGNF) - to je atom,

lambda ili konstruktor - tada se može umetnuti bez rizika od dupliranja posla. Možda je jedina negativna strana povećanje veličine kôda.

2. Ne-SGNF (eng. *Non-Weak Head Normal Form (Non-WHNF)*) - 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, kompromis je između veličine kôda 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 kôda može biti znatan. Kod većine kompaljera koristi se heuristika za odlučivanje kada se radi umetanje funkcija.

Za Ne-SGNF-u, 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 koja beleži za svaku promenljivu na koliko se mesta koristi, i koristimo ove informacije kako bi naučili simplifikator da izvrši umetanje. O ovom načinu se može više pronaći u [15]. Međutim, ovaj način ima svoje komplikacije, drugo predloženo rešenje bi bio *sistem linearnog tipa* [2] pomoću kojeg bi se pratile informacije o pojavama koje su najkomplikovanije i koje su podložne greškama, tako da bi se identifikovale promenljive koje se mogu bezbedno umetnuti, ili koje se ne mogu bezbedno umetnuti. Nažalost, većina linearnih sistema je neadekvatna. Postoji nekoliko pokušaja da se razvije sistem linearnog tipa, ali nijedan nije našao svoju praktičnu primenu. Jedan od pokušaja se može naći u [9].

4.2 Transformacija uslova

Većina kompajlera ima posebna pravila za optimizaciju i transformaciju uslova (eng. *transforming conditionals*). Na primer, razmotrimo izraz:

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

Nijedan moderan kompajler ne bi 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 spoljašnji 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 dobija 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 ispituju poznatu vrednost.

4.2.1 Pridruživanje tačaka

Jedna od tehnika za redukciju kôda je pridruživanje tačaka (eng. *join points*). Postavlja se pitanje kako možemo imati koristi od transformacija "case-case" (eng. *case-of-case*) bez rizika od dupliranja kôda? Jednostavna ideja je napraviti lokalne definicije sa desne strane spoljašnjeg 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 novouvedena vezivanja biti moguće 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}
```

Prečišćavanje uslova i umetanje || daje nam:

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

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

```
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 pojednostavljen, 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}
```

4.2.2 Objedinjavanje pridruženih tačaka

Jedna od korisnih tehnika optimizacije je objedinjavanje pridruženih tačaka (eng. *generalising join points*). Da li se sve ovo generalizuje na tipove podataka koji nisu boolean? 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 promenljivu. Na primer, razmotrite izraz:

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

gde f definišemo:

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

Prečišćavanje if uslova i umetanje f daje nam:

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

Sada, pošto E2 može uključiti 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 proizilazi da celokupna podešavanja funkcionišu za proizvoljne tipove podataka, a ne samo za boolean.

4.2.3 Objedinjavanje case eliminacija

Ranije smo raspravljali o slučaju transformacije "poznatih-konstruktora" koja eliminiše niz izraza.

Postoji korisna varijanta ove transformacije koja takođe eliminiše niz izraza - objedinjavanje case eliminacija (eng. *generalising case elimination*). 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
```

5 Backend

Kod prevođenja jezika Core u neki imperativni međujezik kao što je C-- postoji više etapa. Prvo se *CoreSyn* prevodi u *StgSyn (GHC's intermediate language)*, i to u dve faze:

1. **CorePrep** - Core-to-Core proces konvertuje program u ANF (eng. *A-normal form*) [11]. A-normalnu formu su osmislili Sabry i Fellisen 1992. god. U ANF formi svi argumenti moraju biti trivijalni, odnosno vrednost svih argumenata se mora izračunati odmah. ANF se bavi osnovnim definicijama zasnovanim na λ računu sa slabom redukcijom i let izrazima uz ograničenja:
   - dozvoljene su samo konstante, λ termovi i promenljive kao argumenti funkcije
   - zahtev da rezultat netrivijalnih izraza pripada let-povezanoj promenljivoj ili vraćen iz funkcije.

   Primer ANF:
   f(g(x),h(y))

   let v0 g(x) in
   let v1 h1(y) in
   f(v0, v1)

2. **CoreToStg** - Rezultat prve faze u velikoj meri odgovara krajnjem StgSyn, zato u ovoj fazi nema preterano mnogo posla. Kako god, StgSyn je dekorisan sa mnogo redudantih informacija(slobodnih promenljivih, let-no-escape indikatora...) koji su generisani u letu od strane CoreToStg faze.[12]
   STG program se pomoću generatora kôda (eng. *code generator*) pretvara u neki niži jezik kao što je C-- [8].

5.1 GHC kôd generator

Glazgov Haskel Kompajler je u početku prevodio kôd za STG-mašine (eng. *Spineless Tagless G-machine*) na C jezik. Ideja je bila da se iskoriste C kompajleri koji su portabilni i imaju određene dobre optimizacije. Međutim, pokazalo se da C ima mnogo mana u kontekstu jezika srednjeg nivoa (eng. *intermediate language*), posebno za kompilatore lenjih funkcionalnih jezika sa nestandardnom kontrolom toka. Takođe, C nema efikasan način za repnu rekurziju, pristup steku radi čišćenja memorije (eng. *garbage collection*) i još mnogo drugih stvari. To nije iznenađujuće, jer C nije dizajniran za to. Pisci kompilatora višeg nivoa kao što je GHC su odlučili da ublaže prethodno navedene nedostatke.

Problem je trebalo da reše razne ekstenzije GNU C-a. Međutim, i ovo rešenje ima svoje mane kao što je velika zavisnost od GNU C verzije

kompilatora. Optimizacija za C često nema efekta na te jezike višeg nivoa - mnogo statičkih informacija bi moglo biti izgubljeno.

Kao odgovor na to, GHC je ubacio podršku za kôd generatore niskog nivoa (eng. *native code generators*), koji direktno prevode u mašinski kôd, ali samo za određene sisteme, konkretno x86, SPARC, PowerPC.

Želja da se zadrže dobre osobine svođenja i kompiliranja na C, a da se opet prevaziđu problemi, inspirisala je razvoj jezika srednje-niskog nivoa (eng. *low-level intermediate languages*). Od interesa je jezik C--, jer je dizajniran pod uticajem GHC-a. Iako je upotreba jezika C-- kao srednjeg jezika tehnički veoma perspektivan pristup, dolazi sa velikim praktičnim problemima: razvoj portabilnog kompilatorskog bekenda je isplativ, ako ga koristi više kompilatora, a pisci kompilatora ne žele da rade na razvoju nečega što neće biti u širokoj upotrebi. Kao posledica toga, neka varijanta C-- jezika se koristi kao srednje-niski jezik u GHC-u, ali generalno nema razvijenog bekenda u jeziku C-- koja bi mogla biti podržana od strane većeg broja kompilatora kao što je GHC [6].

Navodimo tri najznačajnija programa za generisanje izvršnog kôda:

1. NGC (eng. *Native Generated Code*): kao prvo rešenje odgovara generisanju asemblerskog kôda. To se pokazalo kao brzo i efikasno rešenje, ali je imalo i veliki nedostatak, jer se za svaku platformu računara (x86, SPARC, PowerPC, ...) morao održavati pripadni asemblerski kôd. NCG sadrži 20,570 linija kôda, što je približno 4 puta više od veličine kôda C-kompajlera. Pritom je kompilacija bila dvostruko brža. Jedna od prednosti NCG u odnosu na C je jednostavnost; iako je NCG kôd mnogo veći, sigurno je i mnogo jednostavniji.

2. C-- : drugo rešenje se našlo u programskom jeziku C--, podskupu jezika C, specijalno prilagođenom zadacima kompilacije. Jedna od apstrakcija koju C-- nudi su virtualni registri. Oni omogućavaju mapiranje mašinskih registara na optimalan način. Delimično u tome učestvuje i run-time verzija GHC-a. C-- verzija kompajlera se znatno razlikuje od zvaničnog C-- standarda. Osnovna razlika je u implementaciji sopstvenog GC-a (eng. Garbage Collector), jer Haskel zahteva GC koji je generacijski i koji može da radi po blokovima hipa odnosno steka. Generacijski GC polazi od pretpostavke da će najnoviji objekti biti najbrže skidani sa hipa odnosno steka, stoga se GC primenjuje samo na najmlađu generaciju objekata, a tek kada to ne daje zadovoljavajući rezultat, onda se analiziraju i starije generacije. Na slici 2 je prikazan kôd u C-- koji, očigledno, u velikoj meri liči na C kôd.

3. LLVM : trenutno je najperspektivniji bekend frejmvork-okvir (eng. *framework*), koji dolazi sa just-in-time kompilacijom (kojom se kompajliraju samo delovi kôda koji su ispravljani nakon poslednjeg kompajliranja) kao i life-long analizom (dubinskom analizom za dobijanje optimalnog kôda).

   Postoje četiri osnovna razloga za njegovu implementaciju [13]:

   - Pravljenje kompajlera visokih performansi za generisanje kôda zahteva mnogo uloženog truda i znanja. Tako, razvoj LLVM-a je počeo pre oko 15 godina. Postojanje ovakvog sistema će zahtevati mnogo manje posla na održavanju i proširenju, nego što prethodna dva rešenja zahtevaju. Takođe, pošto je LLVM nezavisan projekat, buduća poboljšanja će biti automatski raspoloživa.

- GHC proizvodi Haskel programe koji se brzo izračunavaju. Međutim, mnoge optimizacije na nižem nivou (naročito one koje podrazumevaju poznavanje specifičnosti arhitekture računara) nisu do kraja implementirane. Korišćenjem LLVM-a, koji koristi mnoge specifičnosti arhitektura računara, ćemo ih automatski dobiti.
- Bitna osobina LLVM-a je ta što je od početka dizajniran kao sredina za razvoj kompajlera. Tako je moguće, u kratkom vremenskom roku i sa relativno malo truda, realizovati prilagođavanje svake nove verzije LLVM-a postojećoj aplikaciji GHC-a.
- LLVM je projekat koji uključuje celokupan lanac alata sa C/C++ kompajlerom, asemblerskim alatima, linkerom, debagerom i alatima za statičku analizu. Koristi se za više programskih jezika, pa mu je baza korisnika garancija da će se i u buduće kvalitetno razvijati.

```
/* Ordinary recursion */
export sp1;
sp1( bits32 n ) {
  bits32 s, p;
  if n == 1 {
      return( 1, 1 );
  } else {
      s, p = sp1( n-1 );
      return( s+n, p*n );
  }
}


/* Tail recursion */
export sp2;
sp2( bits32 n ) {
  jump sp2_help( n, 1, 1 );
}

sp2_help( bits32 n, bits32 s, bits32 p ) {
  if n==1 {
      return( s, p );
  } else {
      jump sp2_help( n-1, s+n, p*n );
  }
}
```

```
/* Loops */
export sp3;
sp3( bits32 n ) {
    bits32 s, p;
    s = 1; p = 1;

  loop:
    if n==1 {
      return( s, p );
    } else {
      s = s+n;
      p = p*n;
      n = n-1;
      goto loop;
}   }
```

Slika 2: Tri funkcije za izračunavanje sume i proizvoda prvih n brojeva u programskom jeziku C--

Prikažimo rezultate korišćenja LLVM-a na jednostavnom primeru izračunavanja funkcije zadate na sledeći način:

- ako je n parno, sledeći broj je n/2
- ako je n neparno, sledeći broj je 3*n+1
- ako je n jedan, stop

Cilj je da nađemo najdužu sekvencu za početne brojeve od jedan do milion. Sekvencu čini broj koraka dok ne stignemo do stopa. Kod napisan u Haskelu je:

```
import Data.Word

collatzLen :: Int -> Word32 -> Int
collatzLen c 1 = c
collatzLen c n | n `mod` 2 == 0 = collatzLen (c+1) $ n `div` 2
| otherwise      = collatzLen (c+1) $ 3*n+1
```

```
pmax x n = x `max` (collatzLen 1 n, n)

main = print . solve $ 1000000
where solve xs = foldl pmax (1,1) [2..xs-1]
```

Kompilacijom ovog kôda različitim vrstama generisanja izvršnog kôda, dobijaju se vremena prikazana u tabeli 1.

Tabela 1: Različita vremena generisanja izvršnih kôdova

GHC-6.13 (NCG)	GHC-6.13 (C)	GHC-6.13 (LLVM)	GCC-4.4.3
2.876s	0.576s	0.516s	0.335s

Iako se očekuje da NCG ima najkraće vreme izvršavanja, jer se direktno prevodi na mašinski kôd, vidimo da u ovom jednostavnom primeru to nije slučaj.

Prethodni primer se može jednostavno paralelizovati. Odgovarajući Haskel kôd je:

```
import Control.Parallel
import Data.Word

collatzLen :: Int -> Word32 -> Int
collatzLen c 1 = c
collatzLen c n | n `mod` 2 == 0 = collatzLen (c+1) $ n `div` 2
| otherwise      = collatzLen (c+1) $ 3*n+1

pmax x n = x `max` (collatzLen 1 n, n)
main = print soln
where
solve xs = foldl pmax (1,1) xs
s1 = solve [2..500000]
s2 = solve [500001..999999]
soln = s2 `par` (s1 `pseq` max s1 s2)
```

U programu je jednostavno ostvarena podela na dva dela i kombinovanje korišćenjem Haskelovih 'par' i 'pseq' funkcija, koje ukazuju kompajleru da paralelno realizuje dva dela s1 i s2. U ovom slučaju, vreme izvršavanja pomoću LLVM-a je:

GHC-6.13 (Parallel, LLVM): 0.312

6 Zaključak

Implementacija funkcionalnih jezika predstavlja veoma težak posao i obimnu temu. U ovom radu fokus je na kompilaciji funkcionalnih jezika, najviše na procesu transformacije funkcionalnog koda u Haskel-u. Čitalac bi trebalo da, nakon čitanja ovog rada, bude upoznat sa teorijskim osnovama kompiliranja funkcionalnih jezika, razume transformaciju izvornog koda i bude u stanju da samostalno istražuje na ovu temu. U našem radu smo se fokusirali na prevodenje izvnornog koda do mašinskog jezika. Taj proces se sastoji od 3 faze : frontend, middle end i backend. U radu su detaljno opisane kljucne faze za efikasnost prevodenja i jedan od najvažnijih razloga za, danas široku upotrebu Haskel jezika. Te faze

su transformacija koda i LLVM framework. GHC kompajler je u stalnom razvoju, a trenutno programeri se najviše fokusiraju na razvoj strategija izvođenja, razne strukture koje bi omogućile efikasniji rad skupljača djubreta(eng. *garbage collector*) i na unapređenje izuzetaka pri prekorečenju heap memorije(dosadašnje verzije izbacuju nedostatke samo pri određenim okolnostima).

Literatura

[1] Stanković Una Stanković Vojislav Ajzenhamer Nikola, Bukurov Anja. Neki elemrnti kompiliranja funkcionalnih jezika. Matematički fakultet, Univerzitet u Beogradu. Srbija, 2017.

[2] C Mossin DM Turner, PL Wadler. Once upon a type. *La Jolla*, 12, 1995.

[3] Simon Marlow (editor). *Haskell 2010 Language Report*. Haskell community, April 2010.

[4] AM Holler JM Davidson. A study of c function inliner. *Software - Practice and Experience*, 1988.

[5] John Launchbury Simon L Peyton Jones. *LISP and Symbolic Computation*. Kluwer Academic Publishers, 1995. on-line at https://doi.org/10.1007/BF01018827.

[6] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, Septembar 1992.

[7] Simon L Peyton Jones. Secrets of the Glasgow Haskell Compiler inliner . *Journal of Functional Programming*, 12:393–434, 2002.

[8] Norman Ramsey, Simon Peyton Jones, and Christian Lindig. The c– language specification, version 2.0, (cvs revision 1.128). Technical report, Cminusminus, Februar 2005. on-line at: https://www.st.cs.uni-saarland.de/publications/files/ramsey-cmm-2005.pdf.

[9] 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.

[10] Edgewall Software. The Glasgow Haskell, Current Status, 2018. online at https://ghc.haskell.org/trac/ghc.

[11] Edgewall Software. The Glasgow Haskell, Current Status, 2018. on-line at https://ghc.haskell.org/trac/ghc/browser/ghc/compiler/coreSyn/CorePrep.hs.

[12] Edgewall Software. The Glasgow Haskell, Current Status, 2018. on-line at https://ghc.haskell.org/trac/ghc/browser/ghc/compiler/stgSyn/CoreToStg.hs.

[13] David A. Terei and Manuel M. T. Chakravarty. An LLVM Backend for GHC. *University of New South Wales*, Septembar 2010.

[14] Stanford University. Why is haskell difficult, 2018. on-line at http://www.scs.stanford.edu/11au-cs240h/notes/ghc.html.

[15] Andrew W.Appel. *Compiling with Continuations*. Princeton University, New Jersey, 2007.