Seminarski rad

Implementacija G apstraktne mašine

Kurs: Funkcionalno programiranje
Profesor: Nenad Mitić

Isidorović Stefan 1014/2013, Luković Filip 1048/2013,

Kostić Tijana 1155/2014, Branislava Živković 1040/2014

28. avgust 2015.

Sadržaj

1 Uvod

Poslednjih godina raste interesovanje i upotreba funkcionalnih jezika u raznim oblastima. Veliki novac se ulaže u razvoj funkcionalnih jezika koji počinju sve više da se koriste u industriji.

Funkcionalna paradigma se zasniva na apstrakciji matematičkih funkcija. Osnovna ideja je oponašanje matematičkih funkcija, zbog čega se ovaj stil programiranja dosta razlikuje od ostalih. Glavna karakteristika funkcionalnih jezika je nepostojanje implicitnog stanja, samim tim ni naredbe dodele kao ni promenljivih. Programi se izvršavaju evaluacijom izraza, a sve iterativne konstrukcije se ostvaruju pomoću rekurzije.

Matematička teorija koja leži ispod ovih jezika je lambda račun. Svaki funkcionalni jezik je u osnovi lambda račun, malo ulepšan kako bi bio intuitivniji i bliži ljudima. Lambda račun definiše pravila za transformaciju izraza i njihovu evaluaciju, nezavisno od mašine na kojoj se izvršava.

Postoje razni algoritmi koji se koriste prilikom prevođenja funkcionalnih jezika. Mi smo se fokusirali na algoritam koji koristi graf redukciju. Opis samog algoritma redukcije se može pronaći u [1]. Kako je moguće svaki funkcionalni jezik prevesti u lambda račun, a nakon toga i u superkombinatore (lambda izraze koji ne sadrže slobodne promenljive) [2], kompilacija pomoću graf redukcije ne zavisi od jezika koji se kompajlira. G apstraktna mašina, opisana u nastavku, prevodi superkombinatore u međukod *G code* {2} nakon čega se on izvršava pomoću graf redukcije {3}.

2 Prevođenje superkombinatora u *G code*

Superkombinatori se definišu na sledeći način:

```
S x1 x2 ...  xn = E
```

E predstavlja izraz u sledećem obliku

```
E = <konstanta>
| <identifikator>
| <E> <E>
| let <identifikator> = <E> in <E>
| letrec <identifikator> = <E>
...
<identifikator> = <E>
in <E>
```

Program koji se sastoji od superkombinatora sadrži niz ovakvih definicija, uz dodat poslednji konstantni superkombinator koji predstavlja izračunatu vrednost. Na primer:

```
F ...  = ...
G ...  = ...
...
Z ...  = ...
--------
Prog
```

Svaki program, preveden u *G code*, sadrži sledeće delove:

- deo koji se odnosi na inicijalizaciju
- deo koji izvšava superkombinator `Prog` i ispisuje njegovu vrednost
- deo koji sadrži definicije superkombinatora označene labelama
- deo koji sadrži označene delove koda koji se odnose na već ugrađene funkcije, na taj način se konstruiše biblioteka ugrađenih funkcija koja je ista za sve programe

Naredni segment koda se odnosi na prve dve stavke i on je isti za svaki program.

```
BEGIN; {inicijalizuje program}
PUSHGLOBAL Prog; {postavlja labelu Prog na stek}
EVAL; {evaluira}
PRINT; {ispisuje rezultat}
END;
```

Sledeća stavka se odnosi na prevođenje superkombinatora o čemu govorimo u nastavku.

Prevođenje definicije superkombinatora

Kao što smo već videli, definicija superkombinatora izgleda ovako:

```
S x1 x2 ...  xn = E
```

Definisaćemo funkciju `F` koja će prihvatati definiciju superkombinatora kao argument i vraćaće njegov prevod u *G code*, zvaćemo je kompilaciona šema .

```
F[ S x1 x2 ...  xn = E ] = ...  G code za F ...
```

S obzirom na to da se za izvršavanje *G code*-a koriste stekovi, potrebno je za svaki superkombinator čuvati informacije o trenutnom kontekstu. Čuvamo relativne pozicije svakog argumenta u odnosu na vrh steka, i naravno informaciju o korenom čvoru, superkombinatoru koji se redukuje, tzv. redex.
Ove informacije će se čuvati na sledeći način:

`p` - funkcija koja prihvata identifikator i vraća njegovu udaljenost od korena trenutnog konteksta
`d` - dubina trenutnog konteksta minus 1

Na primer, udaljenost promenljive `x` od korena čvora se računa kao `d - p(x)`.

Kompilaciona šema `F` se definiše na sledeći način:

```
F[ S x1 x2 ...  xn = E ] = GLOBSTART S, n;  {početak funkcije S koja
                                               ima n argumenata}
                           R[E, p, d] {poziv R kompilacione šeme
                             koja prihvata izraz E, p i d redom}
```

`R` kompilaciona šema pravi instancu tela superkombinatora, koristeći parametre sa steka, ažurira koreni čvor, sklanja parametre sa steka i inicijalizuje sledeću redukciju.

```
R [E, p, d] = C [E, p, d] {poziv C kompilacione šeme koja prihvata
raz E, p i d redom}
              UPDATE (n+1); {ažurira poziciju korenog čvora, koji se
                             sada nalazi na udaljenosti n+1 od vrha steka}
              POP n;    {skida n argumenata sa steka}
              UNWIND;    {inicijalizuje sledeću redukciju}
```

`C` kompilaciona šema pravi instancu izraza. Rezultat funkcije `C` zavisi od tipa izraza `E`, i u nastavku ćemo navesti definicije ove funkcije za svaki od tipova `E`.

`E` je konstanta Razlikovaćemo dva slučaja: kada je `E` celobrojna vrednost i kada je superkombinator.
Ukoliko je `E` konstanta, `C` kompilaciona šema se definiše na sledeći način:

```
C [i, p, d] = PUSHINT i; {postavlja vrednost}
```

Ukoliko je `E` superkombinator:

```
C [f, p, d] = PUSHGLOBAL f; {postavlja pokazivač na funkciju f}
```

`E` je promenljiva Vrednost promenljive se nalazi na udaljenosti `d - p(x)` od vrha steka. Definicija `C` kompilacione šeme je sledeća:

```
C [x, p, d] = PUSH d - p(x); {postavlja udaljenost promenljive x}
```

E je primena `(E1 E2)` prestavlja primenu `E1` na `E2`, gde su `E1` i `E2` proizvoljni izrazi. Potrebno je prvo konstruisati instancu `E2`, nakon toga `E1` i napraviti aplikacioni čvor.

```
C [E1 E2, p, d] = C [E2, p, d]
                  C [E1, p, d+1]
                  MKAP; {uzima prva dva elementa sa steka, pravi
                         aplikacioni čvor, i postavlja pokazivač na njega}
```

E je let izraz (`let x = Ex in Eb`) x je promenljiva, `Ex` i `Eb` su izrazi. Potrebno je napraviti instancu izraza `Ex`, promeniti vrednost funkcije `p` za argument `x` i konstruisati instancu `Eb` sa novim parametrima, nakon čega `Eb` sadrži informaciju o `Ex` , tako da `Ex` možemo obrisati sa steka.

```
C [let x = Ex in Eb, p, d] = C [Ex, p, d]
                             C [Eb, p', d+1] {p'(x) = d+1 ,
                                              p'(y) = p(y), y != x}
                             SLIDE 1; {skida jedan element sa steka}
```

E je letrec izraz (`letrec D in Eb`) `D` je skup uzajamno rekurzivnih definicija a `Eb` izraz. `Letrec` izraz predstavlja jedan cikličan graf. Konstruišemo ga tako što:

- alociramo prazne ćelije za svaku od definicija unutar `D`
- promenimo kontekst (`p` i `d`) i označimo da se vrednosti promenljivih koje su vezane `letrec` izrazom mogu pronaći na alociranim pozicijama
- za svaku definiciju superkombinatora iz `D` napravimo njenu instancu i postavimo pokazivač na stek, nakon čega ažuriramo alociranu ćeliju napravljenom instancom, a sva pojavljivanja vezanih promenljivih u `Eb` zamenimo pokazivačima na alocirane ćelije
- napravimo instancu `Eb` i postavimo pokazivač na stek
- skinemo sa steka pokazivače na definicije iz `D`

```
C [letrec D in Eb, p, d] = CLetrec [D, p', d']
                           C [Eb, p', d']
                           SLIDE (d'-d);
                            where
                           (p', d') = Xr [D, p, d]

CLetrec [x1 = E1, ...  xn = En, p, d] =
                  ALLOC n;  {alocira n ćelija}
                  C [E1, p, d] UPDATE n;
                  C [E2, p, d] UPDATE n-1;
                   ...
                  C [En, p, d] UPDATE 1;

Xr [x1 = E1, ...  xn = En, p, d] = ( p [x1 = d+1] , d + n )
                                     p [x2 = d+2]
                                       ...
                                     p [xn = d+n]
```

3 Izvršavanje *G code*-a

Program preveden u *G code* treba dalje prevesti na asembler konkretne mašine na kojoj se izvršava. Svakoj instrukciji *G code*-a odgovara niz instrukcija ciljne mašine. Potrebno je znati tačno šta svaka instrukcija *G code*-a radi.

G mašina sadrži sledeće 4 komponente:

- S (stek)
- G (graf)
- C (niz *G code* instrukcija)
- D (stek za odlaganje)

Jedno stanje G mašine je predstavljeno uređenom četvorkom <G, S, C, D>. Unutar grafa G mogu se nalaziti čvorovi sledećih tipova:

- INT i (ceo broj)
- CONS n1 n2 (konstantni čvor)
- AP n1 n2 (aplikacioni čvor)

- FUN k C (funkcija od k argumenata sa telom C)

- HOLE (prazan čvor koji će biti popunjen kasnije)

Graf je prestavljen hipom. Čvor grafa ne mora nužno biti jedna ćelija hipa. U zavisnosti od tipa čvora on može zaužimati proizvoljan broj ćelija. Stek S sadrži vrednosti koje trenutno koristimo. Svaka vrednost na steku je pokazivač na čvor grafa. Stek za odlaganje D se koristi za čuvanje nekih delova *G code*-a koji će biti korišćeni kasnije. Jedan primer korišćenja steka D je poziv funkcije. Deo koji se odnosi na glavni program se prebacuje na stek D, dok se na stek S prebacuje deo kode koji se odnosi na poziv funkcije. Nakon završetka funkcije, sa steka D se prebacuje kod na stek S.
G mašina funkcioniše po principu prelaska iz jednog stanja u drugo. Svakim prelazom menjamo neko od stanja četvorke <G, S, C, D>. Jedna instrukcija G mašine se odnosi na jedan prelaz. Skup instrukcija i njihovih prelaza je prikazan na slici 1.

```
EVAL <v:S, G[v = AP v' n], EVAL:C, D> => <v:[], G, UNWIND:[], (S,C):D>
EVAL <n:S, G[n = FUN 0 C'], EVAL:C, D> => <n:[], G, C':[], (S,C):D>
EVAL <n:S, G[n = INT i], EVAL:C, D> => <n:S, G, C, D>
EVAL (slično kao prethodan prelaz definišu se i EVAL za CONS i FUN k, k > 0, prelazi)
UNWIND <n:[], G[n = INT i], UNWIND:[], (S,C):D> => <n:S, G, C, D>
UNWIND (slično kao prethodan prelaz definiše se i UNWIND CONS prelaz)
UNWIND <v:S, G[v = AP v' n], UNWIND:[], D> => <v':v:S, G, UNWIND:[], D>
UNWIND <v0:...:vk:S, G[v0 = FUN k C, vi = AP v(i-1) ni, UNWIND:[], D> => <n1:...:nk:vk:S, G, C, D>
UNWIND <v0:...:va:[], G[v0 = FUN k C'], UNWIND:[], (S,C):D> => <va:S, G, C, D> (za a < k)
RETURN <v0:...:vk:[], G, RETURN:[], (S,C):D> => <vk:S, G, C, D)
JUMP <S, G, JUMP L:...:LABEL L:C, D> => <S, G, C, D>
JFALSE <n:S, G[n = BOOL false], JFALSE L:...:LABEL L:C, D> => <S, G, C, D>
JFALSE <n:S, G[n = BOOL false], JFALSE L:C, D> => <S, G, C, D>
PUSH <n0:...:nk:S, G, PUSH k:C, D> => <nk:n0:...:nk:S, G, PUSH k:C, D>
PUSHINT <S, G, PUSHINT i:C, D> => <n:S, G[n = INT i], C, D>
PUSHGLOBAL (slično kao prerhodni prelaz)
POP <n1:...:nk:S, G, POP k:C, D> => <S, G, C, D>
SLIDE <n0:...:nk:S, G, SLIDE k:C, D> => <n0:S, G, C, D>
UPDATE <n0:...:nk:S, G, UPDATE k:C, D> => <n1:...:nk:S, G[nk = G n0], C, D>
ALLOC <S, G, ALLOC k:C, D> => <n1:...:nk:S, G[n1 = HOLE, ..., nk = HOLE], C, D>
HEAD <n:S, G[CONS n1 n2], HEAD:C, D> => <n1:S, G, C, D>
NEG <n:S, G[n = INT i], NEG:C, D> => <n1:S, G[n' = INT (-i)], C, D>
ADD <n1:n2:S, G[n1 = INT i1, n2 = INT i2], ADD:C, D> => <n:S, G[n = INT (i1 + i2)], C, D>
SUB, MUL, DIV, MIN, MAX (slično kao prerhodni prelaz)
MKAP <n1:n2:S, G, MKAP:C, D> => <n:S, G[n = AP n1 n2], C, D>
CONS (slično kao prerhodni prelaz)
BEGIN, END, GLOBSTART, PRINT (trivijalno)
```

Slika 1: Instrukcije i njihovi prelazi

3.1 Prevođenje *G code*-a u VAX asembler

Imajući u vidu to da je *G code* apstraktan međukod, potrebno je svaku njegovu instrukciju prevesti u odgovarajući niz asemblerskih instrukcija konkretnog asemblera (u našem slučaju koristili smo *VAX* asembler). *VAX* asembler koristi četiri pokazivača: *sp* (vrh steka za odlaganje), *ep* (vrh steka), *hp* (kraj hipa) i *op* (mesto gde se ispisuju vrednosti prilikom poziva funkcije PRINT). Vrednosti pokazivača se inicijalizuju na početku. Koriste se registri *r0* i *r1*, kao i neposredan (npr. #1), direktan i indirektan (npr. $(@ep)) način adresiranja. Instrukcije kao i njihova objašnjenja:

movl - premešta vrednost prvog operanda iz jednog registra u drugi
movlal - premešta adresu prvog operanda iz jednog registra u drugi
addl2 - vrši sabiranje i rezultat smešta u prvi operand
subl2 - vrši oduzimanje i rezultat smešta u prvi operand
mull2 - vrši množenje i rezultat smešta u prvi operand
divl2 - vrši deljenje i rezultat smešta u prvi operand
maxu - računa maksimum dva broja
minu - računa minimum dva broja
cmpl - vrši poređenje dva broja
jlss - izvršava se ukoliko su dva operanda ista
jmp - skače na zadatu lokaciju
jsb - poziva potprogram
rsb - izlazi iz potprograma
incl - uvećava operand za 1
decl - smanjuje operand za 1.

Prevođenje pojedinačnih *G code* instrukcija u *VAX* asembler se nalaze u datoteci GCode-VAX.txt.

4 *Salira* implementacija

Salira implementacija G apstraktne mašine obuhvata podskup *Haskell*-a koji je opisan gramatikom u nastavku.

```
PROGRAM : PROGRAM LINE ;
        | LINE ;
LINE : IDF ID = EXP
     | IDF INTNUM
EXP : EXP + EXP
    | EXP - EXP
    | EXP * EXP
    | EXP / EXP
    | INTNUM
    | ID
    | ( EXP )
    | IDF ( EXP )

INTNUM: (0-9)+
IDF: (A-Z)+
ID: (a-z)+
```

5 Uputstvo za korišćenje softvera

Salira simulator prevodi *Haskell* kod u *G code*, kod apstraktne G mašine. Nakon toga *G code* prevodi na jezik *VAX* asemblera i izvršava ga.

Sa leve strane se tekstualno polje za unos *Haskell* koda, koji se može učitati i iz datoteke pritiskom na dugme sa direktorijumom.

Dugme sa četkicom pored briše *Haskell* kod.

Pritiskom na dugme sa knjižcom *Haskell* kod se prevodi u *G code* koji se ispisuje u tekstualnom polju pored.

Pritiskom na dugme sa strelicama polje sa *G code*-om se osvežava.

Dugme sa srelicom udesno služi za izvršavanje jedne instrukcije *G code*-a, dok dugme sa trouglom izvšava ceo *G code* i ispisuje rezultat u polje za izlaz.

Iznad polja za izlaz se nalaze tri polja za stanje steka, grafa i steka za odlaganje tokom izvršavanja programa. Na taj način se može pratiti tok izvršavanja korak po korak.

Kartica **File** sadrži:

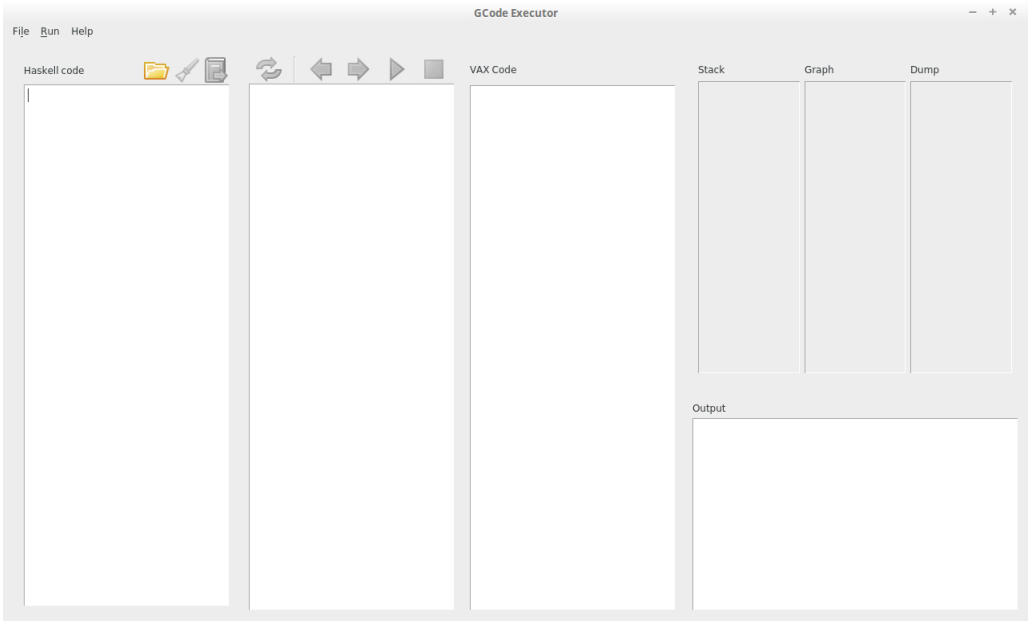
Open -> Učitavanje datoteke sa *Haskell* kodom
Translate to *G code* -> Prevođenje u *G code*
Clear -> Čišćenje kartica
Save *G code* -> Sačuvati *G code* u datoteku
Save *VAX* code -> Sačuvati *VAX* kod u datoteku
Close Program -> Isključiti program

Kartica **Run** sadrži:

Evaluate -> Izvrši
Next command -> Izvrši narednu komandu
Previouse Command -> Vrati prethodnu komandu
Run command -> Izvrši trenutnu komandu
Stop Execution -> Zaustavi izvršavanje

Pokretanje programa se vrši uz `sudo` privilegije iz terminala komandom **sudo ./run.sh** ili pokretanjem izvršnog fajla **Salira** unutar **build** foldera. Kompajliranje se vrši komandom **sudo ./install.sh**.

Izgled grafičkog interfejsa je prikazan na slici 2.



Slika 2: Grafički korisnički interfejs

Literatura

[1] Thomas Jonsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN*, 1984.

[2] Simon L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall International, 1987.