Miloš Savić

YAPCC: prenosivi C cross kompajler

YAPCC je prenosivi C cross kompajler namenjen skupu registarskih arhitektura. Prednji deo kompajlera, koji generiše međukod izvornog programa, jasno je razdvojen od zadnjeg dela kompajlera koji na osnovu tog međukoda generiše ciljni kod. Međukod je tako dizajniran da odgovara najvećem skupu registarskih arhitektura. Osnovna ideja YAPCC kompajlera jeste da se kao deo generatora koda koristi virtuelna mašina koja ima mogućnost interpretiranja semantičkih akcija, na osnovu kojih se generiše ciljni kod za datu arhitekturu, te je stoga kompajler cross kompajler. U generatoru koda uočeni su delovi koji su nezavisni od arhitekture, delovi koji su poluzavisni i potpuno zavisni delovi. Nezavisni delovi i nezavisni delovi poluzavisnih delova su implementirani direktno u generatoru koda, dok se zavisni delovi interpretiraju na virtuelnoj mašini. Na taj način omogućeno je lakše pisanje formalizama za konkretnu arhitekturu. Mašinski orijentisane optimizacije omogućene su korišćenjem specijalnog dela LISP formalizma u kome se definišu zavisnosti među podstablima stabla zavisnosti. Specijalna rutina, scheduler, obilazi stablo zavisnosti na osnovu LISP dela formalizma i startuje procese na virtuelnoj mašini koji generišu ciljni kod.

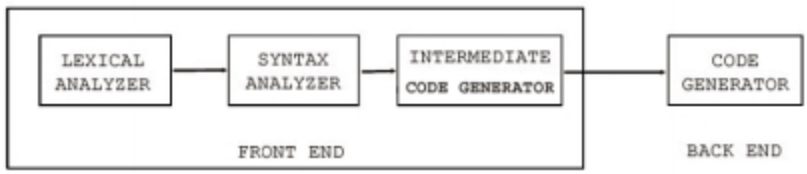
Uvod

Kompajler se sastoji od leksičkog analizatora, sintaksnog analizatora i generatora koda. Posebne vrste kompajlera imaju, između sintaksnog analizatora i generatora koda, i optimizator koda.

Osnovni zadatak *leksičkog analizatora* jeste da sa ulaza čita karaktere izvornog koda koji se prevodi, a na izlazu da daje sekvencu tokena koji će se kasnije koristiti u sintaksnoj analizi. *Sintaksni analizator* analizira sekvencu tokena i osnovni zadatak te analize jeste da se utvrdi da li se ta sekvenca tokena može generisati pomoću *konteksno slobodne gramatike*. Konteksno slobodna gramatika je skup pravila na osnovu kojih se vrši rekonstrukcija jezika. Često se za konteksno slobodnu gramatiku koristi i samo termin gramatika. Zadnji deo kompajlera je generator koda koji prema skupu semantičkih akcija koje su pridružene sintaksnim konstruktima na izlazu daje konačni generisani kod.

Miloš Savić (1984), Valjevo, Maljenska bb, učenik 2. razreda Valjevske gimnazije

Leksički analizator i sintaksni analizator spadaju u mašinski nezavisan deo kompajlera, jer na svim arhitekturama obavljaju identičnu funkciju, dok generator koda spada u mašinski zavisan deo. Generator koda ne može biti isti za sve arhitekture usled njihove različitosti. Razlike postoje u setu instrukcija, formatu instrukcija, načinima adresiranja, tipovima podataka, organizaciji ulazno-izlaznog sistema, i uopšte u pogledu konvencionalnog mašinskog jezika, te stoga je ova faza izričito mašinski zavisna (Miletić 1997). U literaturi mašinski nezavisan deo se označava kao prednji deo kompajlera (front end), dok se mašinski zavisan deo označava kao zadnji deo (back end) (Aho *et al*. 1986). Na slici 1 nalazi se dijagram toka prevođenja.



Slika1.
Dijagram toka prevođenja

Figure 1.
Phases of a compiler

Prva i osnovna ideja svih prenosivih kompajlera jeste da se objedine svi delovi koji su zajednički za sve kompajlere (prednji deo), a da se zatim po potrebi dopisuju mašinski zavisni delovi kompajlera (Miletić 1997).

Do sada je poznato više razrada ove ideje (PCC 1, PCC 2, Gnu C, JAFFA), međutim u svim slučajevima dolazi se do potrebe da se na formalan način opišu arhitekture mašina.

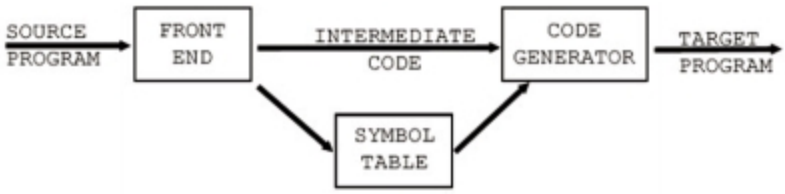
U ovom radu se detaljnije diskutuje novi pristup pisanju prenosivih kompajlera. Cilj rada je dobiti efikasan sistem koji se što lakše proširuje. Detaljnije se diskutuju prednji deo kompajlera, ulaz koji dobija sam generator koda i generator koda kao najvažniji deo prenosivog kompajlera. Kao posebna celina diskutuje se formalizam predložen za proširivanje palete mašina za koje je moguće generisanje koda.

Generator koda

Kao što je već rečeno zadnja faza u modelu jednog kompajlera jeste generator koda. On na ulazu dobija međukodnu reprezentaciju izvornog koda (koja je proizvod prednjeg dela kompajlera) i tabelu simbola, dok na izlazu daje konačan oblik generisanog koda. Šema generatora koda i njegova pozicija u modelu kompajlera data je na slici 2.

Informacije u tabeli simbola se koriste za određivanje run-time adrese objekta podataka datog njegovim imenom u međukodnoj reprezentaciji.

Postoji mnogo izbora za međukodni jezik, uključujući linearne reprezentacije (postfix notacija), troadresni međukod, kod stek mašina, ili grafičke reprezentacije (sintaksna stabla i *dag*ovi). YAPCC kao međukod koristi troadresni kod specijano projektovan za njega.



Slika 2.
Pozicija generatora koda u modelu kompajlera

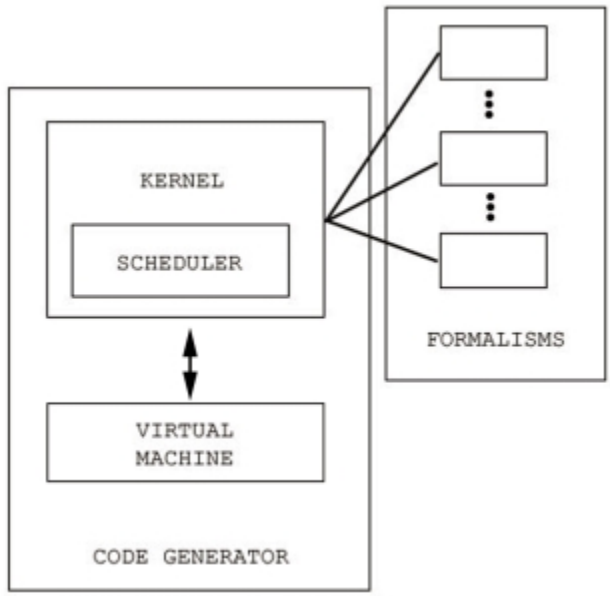
Figure 2.
Position of code generator in the compiler design

Izlaz generatora koda jeste ciljni program. Ovaj izlaz može biti u različitim formama: apsolutni mašinski jezik, relokatibilni mašinski jezik ili asemblerski jezik. YAPCC nije ograničen time koja se od ovih formi generiše na izlazu jer se to definiše u okviru formalizma.

Uopšteno, problemi koje rešava generator koda su: izbor instrukcije (engl. instruction selection), rukovanje memorijom (engl. memory management), alokacija registara (engl. register allocation) i biranja najboljeg rasporeda instrukcija u generisanom kodu (engl. evaluation order).

Ideja

Iz izlaganja u prethodnom poglavlju možemo zaključiti sledeće: generator koda G je skup uređenih dvojki koje čine instrukcije međukoda M i semantičke akcije, koja će se izvršiti kada se zatraži generisanje koda za M i načina na koji se obilazi stablo zavisnosti i u skladu sa time kako se pozivaju semantičke akcije. Osnovna ideja generatora koda YAPCC kompajlera jeste da se ustanovi tačan skup semantičkih akcija koje su iste za sve arhitekture i da se realizuje virtuelna mašina koja će biti sposobna interpretirati te instrukcije. Na taj način postižemo i da je YAPCC kros (engl. cross) kompajler, jer je moguće na jednoj mašini generisati kod za drugu mašinu. U procesu generisanja koda uočeni su delovi koji su isti za sve arhitekture, delovi koji su poluzavisni od arhitekture i potpuno zavisni delovi. Delovi nezavisni za sve arhitekture obično rade transformacije na međukodu. Poluzavisni delovi su delovi koji su idejom isti na svim mašinama ali su u fazama implementacije i primene različiti, dok su potpuno zavisni delovi oni koji su osobenost date arhitekture, koja nije prethodno generalizovana. Ideja je da se nezavisni i nezavisni delovi poluzavisnih delova implementiraju direktno u generator koda, dok bi se zavisni delovi kodirali u okviru formalizma i kasnije interpretirali na virtuelnoj mašini. Naravno, zavisni i nezavisni delovi poluzavisnih delova bi morali komunicirati na određeni način da bi sinhronizovali svoj rad i da bi mogli razmenjivati podatke. Jedan međuzavisan deo u generatoru koda je alokacija registara. Idejno ona je ista na svim mašinama ali ne i u implementaciji (primer je IBM 370 system) (Aho *et al*. 1986). Ovakav način projektovanja generatora koda ima više posledica: nije potrebna rekompilacija kompajlera kada se želi proširivanje skupa mašina za koje je moguće generisanje koda, formalizmi postaju manji i lakši za pisanje, kompajler dobija osobine kros-kompajlera.



Slika 3.
Šematski prikaz YAPCC kod generatora

Figure 3.
YAPCC code generator

Generator koda YAPCC kompajlera sastoji se od virtuelne mašine i kernela. Semantičke akcije se posmatraju kao zasebni procesi koji se mogu startovati na virtuelnoj mašini. Kernel je deo generatora koda koji kontroliše izvršavanje procesa na virtuelnoj mašini i u njemu su implementirani nezavisni delovi generatora koda i nezavisni delovi poluzavisnih delova. Pored toga, u kernelu se nalazi i osnovni algoritam za generisanje koda. To je dinamički algoritam koji generisanje koda vrši na osnovu stabla zavinosti i realizovan je kao modifikacija algoritama koji koriste kompajleri PCC 1 i PCC 2 (Aaho *et al*. 1986). Specijalna rutina u njemu koja na osnovu stabla zavisnosti startuje procese na virtuelnoj mašini, naziva se *scheduler*. Zbog optimizacija koda koji se dobija na izlazu, a koje su zavisne od arhitekture (tzv. post optimizacije), veoma bitna karakteristika schedulera jeste način na koji se biraju podstabla za koje će se kasnije startovati procesi. Pošto je ovo čisto mašinski zavisan deo, on se takođe mora realizovati u okviru formalizma. To je specijalni deo formalizma koji se piše u nešto pojednostavljenoj varijanti LISP jezika. Na slici 3 šematski je prikazan generator koda YAPCC kompajlera.

U daljem tekstu rada biće detaljnije diskutovan generator koda i formalizam za opis ciljne arhitekture.

Implementacija

Kompajler koji je realizovan u ovom projektu zove se YAPCC, što je skraćenica od *yet another portable C compiler* (još jedan prenosivi C kompajler). Implementiran je u programskom jeziku C, uz korišćenje pro-

gramskih alata FLEX (za generisanje leksičkog analizatora C jezika i jezika za opis formalizama) i YACC (za generisanje sintaksnog analizatora C jezika i jezika za opis formalizama).

Prva verzija prevodioca rađena je na Linux sistemu a prevođena Gnu C prevodiocem. Druga verzija prevodioca kompajlirana je sama sobom i tako je kao krajnji proizvod dobijen YAPCC prevodilac za sve sisteme za koje je napisan formalizam.

Prednji deo kompajlera

Leksička analiza je prva faza u prevođenju jednog programa. Zadatak leksičkog analizatora jeste da sa ulaza čita karaktere izvornog programa, a da na izlazu daje sekvencu tokena koju parser koristi pri sintaksnoj analizi. Leksički analizator takođe možemo posmatrati kao korutinu sintaksnog analizatora. Leksički analizator za programski jezik C u YAPCC kompajleru implementiran je u programskoj formi Flex. Flex koristi specijalan jezik koji se zove Flex language, kao ulaznu specifikaciju. Kao proizvod Flex prevodioca dobijamo C datoteku u kojoj je implementirana reprezentacija tranzicionih dijagrama koji su konstruisani od regularnih izraza u Flex specifikaciji.

Sledeća faza u modelu jednog prevodioca jeste sintaksna analiza. Zadatak sintaksne analize jeste da tumači sekvencu tokena koja je dobijena od leksičkog analizatora, da proveri sintaksnu ispravnost programa i, ukoliko je program sintaksno ispravan, da startuje semantičke akcije koje će generisati međukod izvornog programa.

Gramatika jednog jezika opisuje se pomoću jedne specijalne notacije koja se zove kontekstno slobodna gramatika. Ovakva notacija se često koristi za opisivanje hijerarhijskih struktura u mnogim programskim jezicima. Na primer while iskaz C jezika (while (expression) statement) možemo pomoću kontekstno slobodne gramatike zapisati kao:

```
stmt → while (expr) stmt
```

u kojem se strelica (→) može čitati kao "ima oblik". Ovakvo pravilo se naziva produkcija. U produkciji, leksički elementi se nazivaju tokenima. Izrazi kao *expr* i *stmt* predstavljaju sekvencu tokena koja je opisana nekom drugom produkcijom koja sledi i oni se nazivaju neterminalni simboli.

Kontekstno slobodna gramatika ima 4 komponente:

1. Skup tokena (terminalnih simbola)
2. Skup neterminalnih simbola
3. Skup produkcija, pri čemu se svaka produkcija sastoji od neterminalnog simbola (leva strana produkcije) i sekvenca terminalnih i neterminalnih simbola (desna strana produkcije)
4. Jedan neterminalni simbol, označen kao startni

Jedan neterminalni simbol sa leve strane produkcije može imati više opisa i oni se obično razdvajaju znakom pipe (|).

Sintaksna analiza kod YAPCC-a poverena je parser generatoru YACC. YACC je skraćenica od "yet another compiler – compiler". Datoteka u kojoj se nalazi gramatika programskog jezika naziva se YACC specifikacija. Proizvod YACC prevodioca jeste reprezentacija LALR parsera koji je napisan u programskom jeziku C.

Tabela simbola je specijalna struktura podataka koju koristi kompajler za čuvanje informacija (atributa) o svim identifikatorima koje se koriste u programu. Svako polje u tabeli simbola se dobija iz deklaracije promenljive. Tabela simbola je u YAPCC kompajleru implementirana kao hash tabela, i to tako da omogućuje kontrolu nad leksičkim opsegom života identifikatora.

Međukod

U modelu jednog savremenog kompajlera, prednji deo (front end) prevodi izvorni program u međukodnu reprezentaciju, iz koje zadnji deo generiše ciljni kod. Prednosti korišćenja međukoda su:

1. Retargeting – korišćenjem međukoda se postiže da jedan kompajler poseduje jedan prednji deo koji je nezavisan od arhitekture a da se pri proširivanju skupa mašina koje kompajler podržava kreiraju samo zadnji delovi prevodioca.
2. Mašinski nezavisne optimizacije se mogu primeniti direktno na međukodu, te se ne moraju kasnije primenjivati na generisanom ciljnom kodu za datu arhitekturu.

Za potrebe YAPCC kompajlera kreiran je specijalni međukod koji odgovara najvećem broju registarskih mašina. To su skupovi instrukcija zaduženih za rad sa memorijom, registrima, Bulovom, celobrojnom i binarnom aritmetikom. YAPCC kao način zapisivanja međukoda koristi troadresni kod. Troadresni kod je sekvenca instrukcija opšteg oblika $c = a$ **op** b, gde su a i b imena, konstante ili kompajlerski generisane privremene varijable, a **op** je operator. Shodno tome, a i b su operandi operatora **op**, dok je c lokacija u koju se smešta rezultat. Troadresni kod svaki izraz predstavlja koristeći tri adrese kao operande. Otuda potiče i njegov naziv. Neki kodovi troadresnog međukoda ne koriste sve tri adrese i tada se umesto njih umeće simbol *. Argumenti u međukodu mogu biti memorijske adrese, konstante ili registri.

Virtuelna mašina

Virtuelna mašina je deo generatora koda koja interpretira instrukcije iz kojih se obično grade semantičke akcije. Kao posledica interpretacije, na izlazu dobijamo ciljni kod datog izvornog koda, za datu arhitekturu.

Interpreter je najvažniji deo virtuelne mašine koji može interpretirati bilo koju instrukciju iz seta instrukcija virtualne mašine. Memorija virtuelne mašine je memorija u kojoj se čuvaju instrukcije i podaci koje koristi interpreter virtualne mašine. Registri virtuelne mašine predstavljaju primarnu memorija virtualne mašine. Koriste se jer se instrukcije nad memorijskim operandima sporije izvršavaju od onih instrukcija koje za operande imaju registre.

Memorija virtuelne mašine je podeljena na ćelije koje imaju istu veličinu kao registri virtualne mašine. U jednoj memorijskoj ćeliji može se nalaziti jedan podatak, dok se jedna instrukcija kodira u dve memorijske ćelije i to tako što gornji bajt instrukcije ima manju fizičku adresu virtualne mašine od donjeg bajta virtualne mašine.

Sve instrukcije virtualne mašine prema nameni možemo klasifikovati u nekoliko grupa:

1. Instrukcije za rad sa Bulovom aritmetikom: and, or, xor i not.
2. Instrukcije za rad sa celobrojnom aritmetikom: binarne: plus, minus, mul, div i mod i unarna: uminus.
3. Instrukcija dodele: move.
4. Instrukcije skoka: goto, goto_t i goto_f. Goto je nekondicionalni skok, dok su goto_f i goto_t kondicionalni skokovi.
5. Instrukcije poređenja: eq, no, lt, gt, le i ge.
6. Instrukcija sir. Ova instrukcija omogućuje relativna adresiranja. U zavisnosti od operanda setuje registre virtuelne mašine ir1, ir2 ili irres.
7. Instrukcije za generisanje ciljnog koda: igcb, fgcb, acgcb i gencode.
8. Instrukcije za pozivanje procesa koje se izvršavaju na virtuelnoj mašini: param, call i pexit. Instrukcije generišu prekide virtualne mašine i kontrola se prepušta rutinama u kernelu fork(), exec() i exit(), respektivno.
9. Instrukcije za pozivanje procesa implementiranih u kernelu: kparam i kcall. Instrukcije generišu prekid virtualne mašine, koji kontrolu prepušta rutinama implementiranim direktno u kernelu, a koje pomažu generisanje koda (nezavisni delovi poluzavisnih delova).
10. Instrukcije za dinamičku alokaciju i dealokaciju memorije virtualne mašine: memalloc i memfree. Ove instrukcije mogu dinamički alocirati ili dealocirati po jednu stranicu iz segmenta podataka.
11. Instrukcije za rad sa mehanizmom za komunikaciju među procesima, koji je baziran na porukama (engl. messages): ims, fms, gcms, acms, sndmsg i recvmsg. Instrukcije sndmsg i recvmsg generišu prekid virtualne mašine koji poziva rutine send_msg() i recv_msg() iz kernela koje služe za slanje i primanje poruka.

Virtuelna mašina takođe poseduje mogućnosti različitih načina adresiranja operanda i rezultata instrukcije. Modovi adresiranja koje ona podržava su: apsolutno, registarsko, indeksno, indirektno registarsko i indirektno indeksno. Memorija virtuelne mašine se deli na stranice iste veli-

čine. U jednoj stranici mogu biti ili instrukcije ili podaci. Virtuelna mašina ima podršku za *paging*. Naime, u procesima koji se izvršavaju na virtuelnoj mašini koriste se virtuelne adrese koje se prevode u fizičke pomoću tabele stranica. Tabela stranica je lista koja sadrži fizičke adrese svih stranica datog procesa od nižih ka višim virtuelnim adresama virtualne slike procesa koji se trenutno izvršava na virtuelnoj mašini. Kernel za rukovanje memorijom virtuelne mašine koristi algoritam poznat kao statičko straničenje.

Virtuelna mašina takođe poseduje sistem prekida. Prekid virtuelne mašine je događaj koji se ne može predvideti, a čija je posledica prebacivanje u kernel režim generatora koda i startovanje rutina koje su implementirane u kernelu.

Kernel

Kernel je deo generatora koda koji kontroliše kreiranje, održavanje i raspoređivanje, uopšte rukovanje procesima na virtuelnoj mašini. Pored toga, on kontroliše alokaciju i dealokaciju memorije virtuelne mašine. U kernelu su implementirane rutine koje su zajedničke za generatore koda za registarsku arhitekturu (nezavisni delovi i nezavisni delovi poluzavisnih delova generatora koda). Da bi se omogućila komunikacija između nezavisnih i zavisnih delova poluzavisnih delova generatora koda u kernelu su implementirane specijalne rutine za međuprocesnu komunikaciju.

Procesi se raspoređuju za izvršavanje na virtuelnoj mašini, kao što je već pomenuto, od strane specijalne rutine implementirane u kernelu koja se zove scheduler. Scheduler rutina će biti detaljno diskutovana u sledećem poglavlju. Najvažnije rutine vezane za rukovanje procesima, pored schedulera, su fork(), exec() i exit(). Kreiranje i pozivanje procesa može se obaviti ili od strane schedulera ili od strane procesa koji se trenutno izvršava na virtuelnoj mašini. Za rukovanje procesima kernel koristi specijalnu tabelu koja se naziva tabela procesa i poseban stek koji se zove stek virtuelne mašine. Na početku svi slotovi u tabeli procesa su prazni. Kako se traži izvršavanje nekog procesa koji se nije pre toga izvršavao na virtuelnoj mašini, tako se odvajaju slotovi tabele procesa i to upravo predstavlja zadatak fork() rutine. Exec() rutina u slobodne stranice memorije učitava sliku procesa kome predstoji izvršavanje na virtuelnoj mašini, iz spoljne memorije, iz fajlova koji su posebno formatirani. Exec() rutina poziva rutine za rukovanje memorijom iz kernela, koje će alocirati memoriju potrebnu za instrukcije i podatke datog procesa i setuje tabele stranica virtuelne mašine koje odgovaraju datom procesu. Kada proces završi sa svojim izvršavanjem poziva se exit() rutina koja će vratiti stanje virtuelne mašine na početno, što znači da će resetovati tabele stranica virtuelne mašine i inicijalizovati ih na nulte vrednosti; registre virtuelne ma-

šine će inicijalizovati takođe na nulte vrednosti i osloboditi sve stranice u memoriji gde se nalaze stranice podataka datog procesa (stranice sa instrukcijama se ne oslobađaju dok to ne bude potrebno, jer se kasnije može startovati isti proces i onda se stranice ne moraju učitavati iz binarnog fajla gde se nalazi slika datog procesa).

Svaki proces koji se izvršava na virtuelnoj mašini sastoji se iz dva segmenta: segmenta podataka i segmenta instrukcija. Pored ova dva segmenta postoji i globalni segment podatataka i deljeni segment podataka i ta četiri segmenta čine virtuelni adresni prostor procesa koji se izvršava. Virtuelna mašina podržava straničenje a kernel za rukovanje memorijom koristi statičke stranice. Na taj način postignuto je maksimalno iskorišćenje memorije virtuelne mašine.

IPC, ili Inter-Process Communication, omogućava komunikaciju između procesa koji se izvršavaju na virtuelnoj mašini i procesa koji su implementirani u kernelu. Komunikacija među ovim procesima je potrebna jer se oni međusobno dopunjuju u svom radu (zavisni i nezavisni delovi poluzavisnih delova). Virtuelna mašina podržava IPC zasnovan na porukama i deljeni segment podataka, i to su dva mehanizma za komunikaciju među procesima koji se koriste u generatoru koda. Poruke se, na primer, koriste da se rutine obaveste gde se u deljenom segmentu nalaze argumenti i povratna vrednost koju će rutina koristiti. U kernelu su implementirane specijalne rutine za slanje i primanje poruka: send_msg() i recv_msg().

U kernelu su implementirane i rutine koje pomažu generisanje koda. To su nezavisni delovi generatora koda, koji rade transformacije nad međukodom i međuzavisni delovi koji vrše alokaciju registara i memorije.

Generisanje konačnog koda

Generisanje koda je proces u kome se od međukoda generiše ciljni kod. Pored međukoda, ulazni podaci za ovaj proces su tabela simbola i identifikator na osnovu kojega se određuje za koju arhitekturu će biti generisan ciljni kod (pošto je YAPCC kros-kompajler). Naime, u datoteci u kojoj se nalaze parametri konfiguracije kompajlera, nalaze se podaci koji ukazuju koje su sve arhitekture podržane, tj. koje sve formalizme generator koda može koristiti u procesu generisanja koda. Pri startovanju kompajlera, svi ovi podaci smeštaju se u jednu specijalnu listu u kojoj se nalazi identifikator arhitekture i ime datoteke u kojoj se nalazi formalizam za tu arhitekturu. Pod formalizmom se podrazumevaju LISP deo formalizma i binarni fajlovi u kojima se nalaze slike procesa koji se izvršavaju na virtuelnoj mašini, što će biti objašnjeno kasnije. Deo generatora koji se zove *modul handler* daje osnovni interfejs koji povezuje generator koda i fajlove formalizma.

Prva faza u procesu generisanja koda jesu transformacije na međukodu i mašinski nezavisne optimizacije. Kao proizvod ove faze dobijamo graf, koji se još naziva i graf zavisnosti. Pošto je ovde reč o iskazima koji imaju dva operanda, ovaj graf predstavlja binarno stablo, koje se još naziva i stablo zavisnosti. Stablo zavisnosti je specijalna struktura podataka koja pokazuje povezanost elemenata međukoda.

Na osnovu stabla zavisnosti generiše se ciljni kod. Generisanje koda se prvo vrši nad nezavisnim podstablima. Nezavisno podstablo je ono podstablo kod kojeg su operandi lišća stabla zavisnosti. Generator koda startuje semantičku akciju čijim izvršavanjem se generiše ciljni kod i potom za dato nezavisno podstablo nodovi operanda se dealociraju, a u nodu gde je bio kod operacije nalaziće se rezultat izvršene operacije. Sada taj nod predstavlja list stabla i operand nekoj drugoj instrukciji. Postupak se ponavlja sve dok se ne dealocira celo stablo zavisnosti. YAPCC generator koda koristi specijalni deo formalizma koji se piše u nešto pojednostavljenoj varijanti LISP jezika (u daljem tekstu LISP formalizam) i služi za to da se definišu odnosi među nezavisnim podstablima. Pomoću tog dela formalizma, generator koda YAPCC kompajlera, odnosno scheduler rutina iz kernela, na tačno određen način prolazi kroz stablo i traži po tačno određenim kriterijumima nezavisna podstabla i startuje procese na virtuelnoj mašini koji generišu krajnji ciljni kod za neku arhitekturu u zavisnosti od međukoda.

LISP formalizam sastoji se iz sekvence iskaza koje imaju uprošćenu LISP sintaksu:

```
lisp_form → head_list relation_list process_list
head_list → '( NULL )' | '(' heads ')'
heads-STRING | STRING, heads
relation_list → '(NULL)' | '('relations')'
relations → '('STRING STRING STRING')'
process_list → '('NULL')' | '('processes')'
processes → '(' STRING '('args')' ')'
            | ('STRING '('args')', processes')'
args → NULL | STRING | STRING, args
```

Sva LISP pravila se čuvaju u hash tabeli, sortirana po prioritetu. U head listi se nalaze kodovi instrukcija međukoda koji određuju koja će se nezavisna podstabla izabrati za generisajne ciljnog koda. Scheduler rutina prolazi kroz hash tabelu i pokušava da nađe u stablu zavisnosti sva podstabla koja odgovaraju elementima u head listi. Ukoliko su sva stabla pronađena scheduler poziva LISP interpreter koji proverava da li su relacije u listi relacija tačne i ukoliko su tačne, startuje procese date u process listi sa argumentima datim u listi argumenata, jednog po jednog u redosledu

kakvom su navedeni u process listi. Relacije se odnose na podatke iz podstabla koji se mogu porediti sledećim relacijama:

1. Relacijama poređenja adresa operanda i rezultata: =, , , =, , !=
2. Relacijama poređenja modova adresiranja operanda i rezultata. Sve relacije poređenja imaju po jedan argument. Drugi argument koji je specificiran sintaksom navodi se kao *.

Ukoliko je posle prolaza kroz stablo ono ostalo prazno onda se završilo generisanje koda, a u suprotnom scheduler će za prvo nezavisno stablo startovati "podrazumevane" procese na virtuelnoj mašini, i proveriti da li se onda stablo može obići na osnovu LISP pravila. Postupak se ponavlja sve dok stablo ne postane prazno. Pod podrazumevanim procesom se smatra onaj koji se piše za svaku instrukciju međukoda nezavisno od LISP dela formalizma.

Iz izloženog se vidi da formalizam za određenu arhitekturu predstavlja u stvari uniju LISP dela formalizma i instrukcija procesa koji se izvršavaju na virtuelnoj mašini.

U prilogu 1 se nalazi pseudo-kod algoritma scheduler rutine generatora koda, koja čini okosnicu procesa generisanja koda YAPCC prevodioca. Za obilazak stabla koristi se lista kreirana u postupku kreiranja stabla zavisnosti u prvoj fazi generisanja koda. Ona sadrži pointere na nezavisna podstabla stabla zavisnosti (u prilogu referisana kao insubt_list). Kada se briše nezavisno podstablo iz stabla zavisnosti, briše se i pointer iz ove liste. Stablo zavisnosti je prazno kada ova lista sadrži samo 1 elemenat.

Zaključak

U ovom radu opisali smo ključne aspekte u razradi jednog prenosivog kros-kompajlera. Mogućnosti koje nudi YAPCC generator koda jesu promenljive semantičke akcije, koje su omogućene korišćenjem virtuelne mašine za interpretiranje istih i samim tim je omogućen i *crossing*, lakše pisanje formalizama korišćenjem GL jezika i rutina implementiranih u kernelu (a koje su posledica korišćenja virtualne mašine) i mogućnosti definisanja mašinskih optimizacija koristeći LISP deo formalizma.

Literatura

Aho A.V., Sethi R., Ullman J.D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley

Maurice B. 1986. *The Design of the Unix Operating System*. Prentice Hall

Miletić F. 1997. JAFFA: prenosivi kompajler. *Petničke sveske*, 45 (ur. B. Savić). Valjevo: Istraživačka stanica Petnica

Miloš Savić

YAPCC – Portable C Cross Compiler

YAPCC is a portable cross compiler for C programming language. The compiler's portability represents a possibility of generating the same code on different architectures, while crossing represents the compiler's ability to perform compilation on a single machine and simultaneously generate the code for another machine. The primary aim of this project was to construct a portable cross compiler, which can be easily expanded to new architectures.

This paper deals with the new method of creating and implementation of portable compilers. It especially deals with code generator as one of the compiler's most important parts.

YAPCC compiler consists of two distinct parts, called front and back end. The front end represents machine independent part of compiler and consists of lexical and syntax analyser. Lexical analyser for C programming language is constructed by programming tool flex. Flex uses regular expressions for specifying token patterns. Also, regular expressions can be combined with actions which are written in C language. YACC is used to generate the syntax analyser for C programming language. YACC output is a representation of an LALR parser written in C. The front end accepts correct C subset sorce code, transforming it into an intermediate code representation. During this phase, the symbol, scope and type information tables are built and prepared for further use.

The main idea of all portable compilers is that front end generates intermediate code which can be easily translated to a variety of machine languages. Code generator as input gets the internal representation of the source code. YAPCC uses a specialy projected three-address code, which is common to all types of machines which use registers.

YAPCC code generator consists of two layers: the virtual machine and the kernel. The virtual machine represents the interpreter of the instructions which are assembled in semantic actions. The set of instructions which represents a semantic action and is performed (interpreted) by the virtual machine is called a process. Primary task of the kernel is to control executing processes on the virtual machine.

The most important part of the virtual machine is a Virtual Machine Interpreter, which is able to interpret any instruction which belongs to a set of virtual machine instructions. One instruction is a statement of the general form (op x y z), where x, y and z are virtual machine memory addresses, or constants. Op stands for any operator. All virtual machine instructions are classified in several groups: assignment binary statements

(statements of the form $x = y$ **op** z, where **op** is a binary arithmetic or logical operation), assignment unary instructions (instructions of the form $x =$ **op** y, where op is a unary operation, such as unary minus, logical negation, shift operators, etc.), copy statements (statements of the form $x = y$, where value of y is assigned to x), unconditional and conditional jumps, indexed assignments (statements of the form $x = y[i]$ and $x[i] = y$), address and pointer assignments, instructions for handling procces calls and instructions for generating target code. The virtual machine has its own memory for holding data and instructions, which belong to the process running on the virtual machine. Besides that, the virtual machine has registers as the primary memory. Instructions which use registers are faster than instructions which use virtual memory addresses as operands. One instruction is coded in 2 memory cells. The operands of an instruction are specified by combining virtual machine registers and virtual memory locations with address modes. The virtual machine can support several address modes: absolute, register, indexed, indirect register and indirect indexed. Virtual machine memory is divided into pages, which have the same size, and virtual machine has a support for paging. Conversion between virtual addresses and physical addresses in the virtual machine is done using page tables, which are managed by the kernel.

The kernel is a part of the code generator which controls process creating, executing, terminating and scheduling. Also, it controls allocation and deallocation of virtual machine memory. Routines, which are common to all types of architectures (called kernel processes) and which can be activated by a process performed on the virtual machine, are implemented in the kernel. Because of that, IPC routines are implemented in the kernel, which enable inter-process communication between processes and kernel processes. A special routine in the kernel, called scheduler, is responsibile for the process scheduling. New processes can be created by the scheduler, or by a process currently running on the virtual machine. For the process handling, the kernel uses a special table called process table, which holds information about processes (state, memory maps) and a special stack, called virtual machine stack, which holds calling parameters for every process performed by the virtual machine at that moment and return address, and shows in which order the processes called each other. The memory image of every process contains two segments: code and data. There is, also, a global code and data segment and a shared data segment.

Code generating is dynamic. Before starting functions of the code generator, optimizations on three-address code are done and three-address code is converted in the form of dependency tree. The dependency tree is a special data structure, which shows interconnections between elements of intermediate code. Scheduler is based on a special part of formalism,

which is called LISP, because it is in a form of a simple list processing language. LISP part of formalism consists of LISP rules. One LISP rule has three lists (head, relation and process list) and describes dependencies between sub-trees of the dependency tree. All LISP rules are held in hash array and they are ordered by priority. The scheduler browses through the list and tries to find in the dependecy tree all sub-trees which match elements in head list. If all sub-trees are found, scheduler checks if relations in the relation list are correct, and then starts all processes on the virtual machine which are given in the process list, one by one. Those processes generate the target code and replace sub-trees with their equivalents. When the last process finishes its execution, it leaves control to the scheduler. The whole process is repeated until the dependency tree becomes empty. If there are no sub-trees which match one of the LISP rules, and the dependency tree is not empty, the scheduler browses through the dependecy tree using postorder algorithm and starts "considered" processes for every sub-tree and replaces a sub-tree with an equivalent one.

Thus, the formalism of a particular architecture consists of the LISP part of the formalism which describes dependencies between sub-trees of the dependency tree, and a set of routines, which themselves consist of the instructions that the virtual machine is able to interpret. Using virtual machine in the code generator enables portability and crossing, and the LISP part of formalism enables optimizations which depend on the target machine.

To enable an easier way of writing a formalism, there are routines which are implemented in kernel and common to all types of architectures, and which can be activated by a process performed on the virtual machine. To achieve an easier way of writing formalisms, the GL language has been created. Formalisms are written in this language in the manner of higher programming languages.

The whole project has been done in C programming language, using Flex and Yacc. The virtual machine and the kernel have been implemented in C while syntax analyser and lexical analyser for C and GL language have been implemented in Flex and YACC, respectively. The first version of YAPCC was compiled by the GCC compiler on the i386/Linux machine and the second version was compiled by itself.

Prilog: Algoritam za *scheduler* rutinu u pseudo jeziku sličnom programskom jeziku C

```
void schedule(deptree, lisp_hash, insubt_list, num_hash)
    input : dependency tree(deptree),
            hash table with LISP rules(lisp_hash),
            number of hash rules(num_hash),
            list of pointers to independet subtrees(insubt_list)
    output : none
{

    while (1) {
        crule = lisphash;
search:
        /* browse through the hash table with LISP rules */
        for (; crule < lisp_hash + num_hash; crule++) {
            head = crule->head_list;
            init_buffer_list(&buffer_list);

            /* browse through the head and insubt list and find
               all sub-trees which match to the head list */
            while (head != NULL) {
                exsist = FALSE;
                hlp_insubt_list = insubt_list;

                while (hlp_insubt_list != NULL) {
                    exsist = (head->opcode ==
                              hlp_insubt_list->tree->opcode);
                    if (!exist)
                        break;
                    else
                        alloc_buffer(buffer_list,
                                     hlp_insubt_list->tree);

                    hlp_insubt_list = hlp_insubt_list->next;
                }

                head = head->next;
            }
            if (exist)
                break;
        }

        if (!exist)
            break;
        else {
            /* call lisp interpreter */
            in = lisp_interpreter(crule->relation_list,
                                  buffer_list);
            if (in == OK) {
                /* if relations are OK */
                hlp_proc_list = crule->process_list;

                /* start considered processes one by one */
                while (hlp_proc_list != NULL) {
                    module_handler(hlp_proc_list->process,
```

```
                                &process);
                        kernel_exec(process->name, process->params);
                        hlp_proc_list = hlp_proc_list->next;
                    }

                    /* deallocate node from the dependency tree */
                    while (buffer_list != NULL) {
                        if (is_indepst(
                              buffer_list->insubt_list->tree->prev))

                            alloc_insubt(
                               buffer_list->insubt_list->tree->prev);

                        dealloc_insubt(buffer_list->insubt_list);
                        buffer->insubt_list->tree->type = OPCODE;
                        free(buffer_list->insubt_list->tree->left);
                        free(buffer_list->insubt_list->tree->right);

                        buffer_list = buffer_list->next;
                    }

                    if (insubt_list == NULL)
                        return;
                }
            }
            else
                goto search;
        }
    }

    /* start considered processes */
    if (insubt_list == NULL)
        return;
    else {
        module_handler(hlp_instub_list->tree->opcode, &process);
        kernel_exec(process->name, process->params);

        if (is_indepst(instub_list->tree->prev))
            alloc_instub(instub_list->tree->prev);

        dealloc_instub(instub_list);
    }
}
```

U algoritmu je korišćena buffer lista, u kojoj se nalaze pointeri na nezavisna podstabla za koje še se pozvati LISP interpreter. Pored toga korišćene su sledeće funckije: init_buffer_list(), za inicijalizaciju buffer liste, alloc_buffer(), za dodavanje novog elementa u buffer listu, lisp_interpreter(), koja predstavlja implementaciju pojednostavljenog LISP interpreter, module_handler() koja predstavlja implementaciju modul handler-a, kernel_exec(), koji predstavlja exec rutinu u kernelu, is_indepst(), koja proverava da li je dato podstablo nezavisno, alloc_insubt(), koja dodaje novi pointer na podstablo u insubt listu i dealloc_insubt() koja briše pointer iz insubt liste. Sve promenljive koje imaju prefiks hlp_ predstavljaju kopije orginalnih promenljivih i odnose se na pointer promenljive.