Konstrukcija kompilatora
- odgovori na ispitna pitanja -

Mina Milošević
mi17081@alas.matf.bg.ac.rs

2020/2021

Materijal je preuzet sa stranice prof. dr. Milene Vujošević-Janičić.

Sadržaj

1 Nastanak i namena programskih prevodioca. Veza kompilatora i programskih jezika. Izazovi u razvoju kompilatora

Programski prevodilac je program za prevođenje programa iz jezika visokog nivoa u jezik hardvera. Postoji veliki broj različitih programskih jezika i za svaki je potrebno da postoji odgovarajući prevodilac. Postoji veliki broj različitih mašina i za svaku je potrebno da postoji odgovarajući prevodilac.
Glavni pristupi implementaciji programskih jezika: *kompajleri* (kompilatori, programski prevodioci) i *interpretatori* (ne procesiraju program pre izvršavanja; karakteriše ih sporije izvršavanje, koriste se često za jezike skript paradigme).
Kompajleri daju mogućnost korišćenja istog koda napisanog na višem programskom jeziku na različitim procesorima. Prvi kompajler nastaje za prvi viši programski jezik, tj. za Fortran (1957). Budući kompajleri pratiće osnovne principe Fortrana.
Kompajler iz jednog programa pravi drugi program, ciljni kod. Ciljni kod može biti asembler, objektni kod, mašinski kod, bajtkod... Izvršni program se nezavisno pokreće nad podacima, kako bi se dobio izlaz. Izvršni program se može pokretati željeni broj puta.
Kompajler treba da: omogući da je pisanje programa jednostavno; omogući da se izbegnu greške i koriste apstrakcije; prati način na koji programeri razmišljaju; ostvari prostor za što bolji izvorni kod; napravi što bolji izvršni kod (da kreira brz, kompaktan, energetski efikasan kod niskog nivoa); prepoznaje i korektno prevodi samo jezički ispravne programe tj. da pronađe greške u neispravnim programima; uvek ispravno završi svoj rad, bez obzira na vrstu i broj grešaka u izvornom programu; bude efikasan (posebno da bude brz); generiše kratak i efikasan objektni program; generiše odgovarajući program (semantički ekvivalentan izvornom kodu).
Dodatni izazovi u razvoju kompajlera. Omogućiti da kompajleri, osim što izvršavaju komande na standardnim mikroporcesorima, mogu kontrolisati fizičku opremu, te automatski generisati hardver. Omogućiti da specifikacija izvornog programa bude još bliža ljudima, njihovim govornim jezicima i iskustvu.

2 Struktura kompajlera

Prednji deo kompajlera (front end) - zavisi od jezika: vrši leksičku, sintaksnu i semantičku analizu. Transformiše ulazni program u međureprezentaciju (engl. intermediate representation (IR)) koja se koristi u srednjem delu kompajlera. Ova međureprezentacija je obično reprezentacija nižeg nivoa programa u odnosu na izvorni kod.
Srednji deo kompajlera (middle end) - nezavisan od jezika i ciljne arhitekture i vrši optimizacije. Vrši optimizacije na međureprezentaciji koda sa ciljem da unapredi performanse i kvalitet mašinskog koda koji će kompajler proizvesti. Izvršava optimizacije na međureprezentaciji koje su nezavisne od CPU arhitekture za koju je krajnji kod namenjen. Srednji deo kompajlera, da bi izvršio kvalitetnu optimizaciju, najpre vrši analizu koda. Koristeći rezultate analize, vrši se odgovarajuća optimizacija. Izbor optimizacija koje će biti izvršene zavisi od želja korisnika i argumenata koji se zadaju prilikom pokretanja kompajlera. Podrazumevan nivo optimizacija obuhvata optimizacije nivoa O2.
Zadnji deo kompajlera (back end) - zavisi od ciljne arhitekture i vrši generisanje koda. Zadnji deo kompajlera je takođe odgovoran za optimizacije koje su arhitekturalno specifične. Osnovne faze zadnjeg dela kompajlera obuhvataju arhitekturalno specifičnu optimizaciju i genereisanje koda.

3 Pretprocesiranje i linkovanje

Osnovne faze prevođenja, osim kompilacije, obuhvataju i pretprocesiranje i linkovanje. Faza *pretprocesiranja* je pripremna faza kompilacije. Faza *linkovanja* je neophodna faza kako bi se na osnovu proizvoda kompilacije napravio izvršivi program.
Pretprocesiranje. Kompilator ne obrađuje tekst programa koji je napisao programer, već tekst programa koji je nastao pretprocesiranjem. Jedan od najvažnijih zadataka pretprocesora je da omogući da se izvorni kod pogodno organizuje u više ulaznih datoteka. Pretprocesor izvorni kod iz različitih datoteka objedinjava u tzv. jedinice prevođenja i prosleđuje ih kompilatoru. Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o samom programskom jeziku. Pretprocesor analizira samo pretprocesorske direktive.
Povezivanje. Povezivanje je proces kreiranja jedinstvene izvršne datoteke od jednog ili više objektnih modula koji su nastali ili kompilacijom izvornog koda programa ili su objektni moduli koji sadrže mašinski kod i podatke standardne ili neke nestandardne biblioteke. Pored statičkog povezivanja, koje se vrši nakon kompilacije, postoji i dinamičko povezivanje, koje se vrši tokom izvršavanja programa (zapravo na njegovom početku).

4 Leksička analiza

Leksika je podoblast sintakse koja se bavi opisivanjem osnovnih gradivnih elemenata jezika. U okviru leksike, definišu se reči i njihove kategorije. U programskom jeziku, reči se nazivaju *lekseme*, a kategorije *tokeni*. Dakle, pojedinačni karakteri se grupišu u nedeljive celine lekseme koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika i pridružuju im se tokeni koji opisuju leksičke kategorije kojima te lekseme pripadaju. U programskom jeziku, tokeni mogu da budu identifikatori, ključne reči, operatori...
Leksička analiza je proces izdvajanja leksema i tokena, osnovnih jezičkih elemenata, iz niza ulaznih karaktera. Leksičku analizu vrše moduli kompilatora koji se nazivaju *leksički analizatori* (lekseri, skeneri). Oni obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom modulu (sintaksičkom analizatoru) koji nastavlja analizu teksta programa. Leksički analizator najčešće radi na zahtev sintaksičkog analizatora tako što se sintaksički analizator obraća leksičkom analizatoru kada god mu zatreba naredni token.
Tokenima mogu biti pridruženi i neki dodatni *atributi*.
Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole... Ti obrasci za opisivanje tokena su obično *regularni izrazi*, a mehanizam za izdvajanje leksema iz ulaznog teksta zasniva se na *konačnim automatima*. *Lex* je program koji generiše leksere na programskom jeziku C, a na osnovu zadatog opisa tokena u obliku regularnih izraza.
Razlikujemo ključne reči i identifikatore: identifikator ne može biti neka od ključnih reči. Postoje ključne reči koje zavise od konteksta, koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima.

5 Sintaksička analiza

Sintaksa definiše formalne relacije između elemenata jezika, time pružajući strukturne opise ispravnih niski jezika. Sintaksa se bavi samo formom i strukturom jezika bez bilo kakvih razmatranja u vezi sa njihovim značenjem. Sintaksom programa obično se bavi deo programskog prevodioca koji se naziva **sintaksički analizator** ili parser. Rezultat njegovog rada se isporučuje daljim fazama u obliku *sintakstičkog stabla*.

Sintaksa jezika se obično opisuje gramatikama. Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom *kontekstno-slobodne gramatike*. Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza.
Kontekstno-slobodne gramatike su određene *skupom pravila*. Svako pravilo ima levu i desnu stranu. Sa leve strane pravila nalaze se tzv. pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo tzv. završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom).
Na osnovu gramatike jezika formira se *potisni automat* na osnovu kojeg se jednostavno implementira program koji vrši sintakstičku analizu. Formiranje automata od gramatike se obično vrši automatski, uz korišćenje tzv. generatora parsera, poput sistema Yacc, Bison ili Antlr.
Za opis sintakse koriste se i određene varijacije kontekstno-slobodnih gramatika. Najpoznatije od njih su BNF (Bakus-Naurova forma), EBNF (proširena Bakus-Naurova forma) i sintaksički dijagrami. BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksički dijagrami predstavljaju slikovni meta jezik za predstavljanje sintakse.

6 Semantička analiza

Semantika pridružuje značenje ispravnim konstrukcijama na nekom programskom jeziku. Semantika programskog jezika određuje značenje jezika. Može da se opiše formalno i neformalno. Uloga ***neformalne semantike*** je da programer može da razume kako se program izvršava pre njegovog pokretanja. ***Formalne semantike*** koriste se za izgradnju alata koji se koriste za naprednu semantičku analizu softvera. Ovi alati se mogu koristiti kao dopuna semantičkoj analizi koju sprovode kompajleri.
Semantičke greške se otkriju nakon leksičke i sintaksne analize. **Primeri**:
upotreba nedefinisanog simbola; simboli definisani više puta u istom dosegu; greške u tipovima; greške u pozivima funkcija, prosleđivanju parametara.
Izazovi semantičke analize - odbaciti što više nekorektnih programa, prihvatiti što više korektnih programa i uraditi to brzo. Rezultati semantičkih provera često se prijavljuju programeru kao upozorenja.
Semantička upozorenja su često nepotpuna. Na izbor semantičkih provera koje kompajler implementira utiče pre svega efikasnost analize - kompajler mora da radi efikasno i kompleksna semantička analiza nije poželjna jer usporava proces kompilacije. Na izbor semantičkih provera koje kompajler sprovodi na nekom projektu utiče pre svega domen projekta. Najčešće se izbor semantičkih provera može kontrolisati opcijama kompajlera "Options to Request or Suppress Warnings". **Primeri:**
neiskorišćena promenljiva; predlog upotrebe zagrada oko izraza; deljenje nulom
Jednostavna semantička analiza zasniva se na ispitivanju karakteristika apstraktnog sintaksnog stabla. Za dodavanje jednostavnih semantičkih provera, u okviru clang-a postoji čak tri interfejsa, dva koja su sastavni deo kompajlera, i treći koji je nezavisan alat ali u sklopu clang projekta. Kompleksnija semantička analiza uključuje i analizu koda i grafa kontrole toka.
Jedan od osnovnih zadataka semantičke analize je provera tipova (engl. typchecking). Tokom provere tipova proverava se da li je svaka operacija primenjena na operande odgovarajućeg tipa. U zavisnosti od jezika se u nekim slučajevima u sintaksičko drvo umeću implicitne konverzije, gde je potrebno ili se prijavljuje greška.

7 Uloga međukoda i generisanje međukoda. Primer gcc

Većina kompilatora prevodi sintaksičko stablo provereno i dopunjeno tokom semantičke analize u određeni ***međukod*** (intermediate code/representation), koji se onda dalje analizira i optimizuje i na osnovu koga se u kasnijim fazama gradi rezultujući asemblerski i mašinski kod.
Čemu služi ova međureprezentacija? Pojednostavljivanje optimizacija; da bi imali više prednjih delova za isti zadnji deo; da bi imali više zadnjih delova iz istog prednjeg dela.
Veoma je teško dizajnirati dobar IR jezik. Potrebno je balansirati potrebe visokog jezika i potrebe jezika niskog nivoa mašine za koju je izvršavanje namenjeno. Previsok nivo: nije moguće optimizovati neke implementacione detalje. Prenizak nivo: nije moguće korisititi znanje visokog nivoa da se izvrše neke gresivne optimizacije. Kompajleri često imaju više nego jednu međureprezentaciju.
Postoje različiti oblici za međureprezentaciju - graphical representations, three-address representation, virtual machine representations, linear representations. Najčešći oblik međureprezentacije je tzv. *troadresni kod* u kome se javljaju dodele promenljivama u kojima se sa desne strane dodele javlja najviše jedan operator. Naziv troadresni: u svakoj instrukciji se navode "adrese"najviše dva operanda i rezultata operacije. Naredbe kontrole tokase uklanjaju i svode na uslovne i bezuslovne skokove (naredba goto). Da bi se postigla troadresnost, u međukodu se vrednost svakog podizraza smešta u novouvedenu privremenu promenljivu. Njihov broj je potencijalno neograničen, a tokom faze generisanja koda (tj. registarske alokacije) svim promenljivama se dodeljuju fizičke lokacije gde se one skladište.
GCC koristi tri najvažnije međureprezentacije da predstavi program prilikom kompilacije - GENERIC (nezavisna od jezika, svaki gcc podržan jezik se može prevesti na ovu međureprezentaciju), GIMPLE (troadresna međureprezentacija nastala od GENERIC tako što se svaki izraz svodi na troadresni ekvivalent; koristi SSA (static single assignment)) i RTL (Register Transfer Language).

8 Optimizacije međukoda. Uloga i primeri

Optimizacija podrazumeva poboljšanje performansi koda, zadržavajući pri tom ekvivalentnost sa polaznim (optimizovani kod za iste ulaze mora da vrati iste izlaze kao i originalni kod i mora da proizvede iste sporedne efekte). Fazi optimizacije prethodi faza analize na osnovu koje se donose zaključci i sprovode optimizacije. Cilj je unaprediti IR generisan prethodnim koracima da bi se bolje iskoristili resursi.
Zašto je potrebna optimizacija? Generisanje IR-a uključuje redundatnost u naš kod; programeri su lenji.
Optimizacija se najčešće odvija na dva nivoa:

- *optimizacija međukoda* se generiše na početku faze sinteze i podrazumeva mašinski nezavisne optimizacije tj. optimizacije koje ne uzimaju u obzir specifičnosti ciljne arhitekture
- *optimizacija ciljnog koda* se izvršava na samom kraju sinteze i zasniva se na detaljnom poznavanju ciljne arhitekture i asemblerskog i mašinskog jezika na kome se izražava ciljni program

Primeri:

- *constant folding* - konstantni izrazi se mogu izračunati
- *constant propagation* - izbegava se upotreba promenljivih čija je vrednost konstantna

- *strength reduction* - operacije se zamenjuju onim za koje se očekuje da će se izvršiti brže
- *common subexpression elimination* - izbegava se vršenje istog izračunavanja više puta
- *copy propagation* - izbegava se uvođenje promenljivih koje samo čuvaju vrednosti nekih postojećih promenljivih
- *dead code elimination* - izračunavanja vrednosti promenljivih koje se dalje ne koriste, se eliminišu
- *optimizacija petlji* - npr. izdvajanje izračunavanja vrednosti promenljivih koje su invarijantne za tu petlju ispred same petlje

9 Generisanje koda. Izazovi. CISC i RISC arhitekture

Tokom generisanja koda optimizovani međukod se prevodi u završni asemblerski tj. mašinski kod. Tri osnovne faze:

1. faza odabira instrukcija - tada se određuje kojim mašinskim instrukcijama se modeluju instrukcije troadresnog koda
2. faza alokacije registara - tada se određuje lokacija na kojoj se svaka od promenljivih skladišti
3. faze raspoređivanja instrukcija - tada se određuje redosled instrukcija koji doprinosi kvalitetnijem iskorišćavanju protočne obrade i paralelizacije na nivou instrukcija

Osnovni problem generisanja koda je što je generisanje optimalnog programa za dati izvorni kod neodlučiv problem. Koriste se razne heurističke tehnike koje generišu dobar ali ne garantuju da će izgenerisati optimalan kod.
Najbitniji kriterijum za generator koda je da on mora da proizvede ispravan kod. Ispravnost ima specijalni značaj posebno zbog velikog broja specijalnih slučajeva sa kojima se generator koda susreće i koje mora adekvatno da obradi.
Ulaz u generator koda je IR izvornog programa koji je proizveo prednji deo kompajlera, zajedno sa tablicama simbola koje se koriste za utvrđivanje run-time adresa objekata koji se koriste po imenu u okviru IRa. Generatori koda razdvajaju IR instrukcije u "basic blocks", koje se sastoje od sekvenci instrukcija koje se uvek izvršavaju zajedno. U okviru faza optimizacije i generisanja koda najčešće postoje višestruki prolazi kroz IR koji se izvršavaju pre finalnog generisanja ciljnog programa.
Arhitektura procesora definiše, pre svega, skup instrukcija i registara. Skup instrukcija ciljne mašine ima značajan uticaj na teškoće u konstruisanju dobrog generatora koda koji je u stanju da proizvede mašinski kod visokog kvaliteta. Broj i uloge registara takođe imaju značajan uticaj. Najčešće arhitekture su RISC i CISC.
CISC arhitekuru procesora karakteriše bogat skup instrukcija. Bogat skup instrukcija ima za cilj da se smanje troškovi memorije za skladištenje programa. Broj instrukcija po programu se smanjuje žrtvovanjem broja ciklusa po instrukciji, tj. ugradnjom više operacija u jednu instrukciju, praveći tako različite kompleksnije instrukcije. Instrukcije mogu biti različitih dužina. CISC procesori se uglavnom koriste na ličnim računarima, radnim stanicama i serverima, a primer ovakvih procesora je arhitektura Intel x86. CISC procesori imaju više različitih načina adresiranja, od kojih su neki veoma kompleksni. CISC procesori obično nemaju veliki broj registara opšte namene.

RISC arhitektura procesora se zasniva na pojednostavljenom i smanjenom skupu instrukcija koji je visoko optimizovan. Zbog jednostavnosti instrukcija, potreban je manji broj tranzistora za proizvodnju procesora, pri čemu procesor instrukcije može brže izvršavati. Međutim, redukovanje skupa instrukcija umanjuje efikasnost pisanja softvera za ove procesore, što ne predstavlja problem u slučaju automatskog generisanja koda kompajlerom. Ne postoje složene instrukcije koje pristupaju memoriji, već se rad sa memorijom svodi na load i store instrukcije. Najveća prednost je protočna obrada, koja se lako može implementirati. Protočna obrada (pipeline) je jedna od jedinstvenih odlika arhitekture RISC, koja je postignuta preklapanjem izvršavanja nekoliko instrukcija. Zbog protočne obrade RISC arhitektura ima veliku prednost u performansama u odnosu na CISC arhitekture. RISC procesori se uglavnom koriste za aplikacije u realnom vremenu.

10 Izbor instrukcija. Izbor registara. Raspoređivanje instrukcija

Izbor instrukcija. Kod generator mora da mapira IR program u sekvencu koda koji može da bude izvršen na ciljanoj arhitekturi. Kompleksnost mapiranja zavisi od:

- *Nivoa apstrakcija/Preciznosti IR-a* - ukoliko je IR visokog nivoa, kod generator može prevoditi svaku IR instrukciju u sekvencu instrukcija, takav kod kasnije verovatno mora da se optimizuje; ukoliko je IR niskog nivoa, onda se očekuje da takav kod bude efikasan
- *Prirode instrukcija arhitekture* - uniformnost i kompletnost skupa instrukcija su bitni faktori; na nekim mašinama recimo floating point se rešava sa odvojenim registrima
- *Željenog kvaliteta generisanog koda* - brzina instrukcija je bitan faktor; ako nas nije briga za efikasnost ciljanog programa, odabir instrukcija je trivijalan. Neophodno je da znamo cene instrukcija kako bi mogli da dizajniramo dobre sekvence koda

Izbor registara. Tokom faze registarske alokacije određuju se lokacije na kojima će biti skladištene vrednosti svih promenljivih koje se javljaju u međukodu. Cilj je da što više promenljivih bude skladišteno u registre procesora, međutim, to je često nemoguće, jer je broj registara ograničen i često prilično mali. Izbor registara je NP kompletan problem. Problem korišćenja registara je obično podeljen u dva podproblema:

1. *Alokacija registara* - tokom koje se biraju skupovi promenljivih koji treba da borave u registrima u svakoj tački programa
2. *Dodela registara* - tokom koje se biraju određeni konkretni registri u kojima će promenljiva boraviti

Faza ***raspoređivanja instrukcija*** pokušava da doprinese brzini izvršavanja programa menjanjem redosleda instrukcija. Naime, nekim instrukcijama je moguće promeniti redosled izvršavanja bez promene semantike programa. Neki rasporedi instrukcija zahtevaju manji broj registara za čuvanje privremenih rezultata. Jedan od ciljeva raspoređivanja je da se upotrebe pojedinačnih promenljivih lokalizuju u kodu, čime se povećava šansa da se registri oslobode dugog čuvanja vrednosti nekih promenljivih i da se upotrebe za čuvanje većeg broja promenljivih. Izbor najboljeg redosleda je u opštem slučaju NP-kompletan problem. Najjednsotavnije rešenje je ne menjati redosled instrukcija u odnosu na ono što je dao generator međukoda. Promena redosleda instrukcija može uticati na to da se protočna obrada bolje iskoristi tj. da se izbegnu čekanja i zastoji u protočnoj obradi nastala zbog zavisnosti između susednih instrukcija.

11 Jednoprolazni i višeprolazni kompilatori

Analiza se može obaviti u jednom ili više prolaza. Skeniranje i parsiranje se može odraditi u jednom prolazu. Neki kompajleri kombinuju skeniranje, pasrsiranje, semantičku analizu i generisanje koda u jednom prolazu. Takvi kompajleri se nazivaju ***jednoprolazni kompajleri***. Većina kompajlera ipak prolazi kroz kod više puta i to su ***višeprolazni komapjleri***. Neki jezici su dizajnirani tako da podrže jednoprolazne kompajlere (C i C++). Neki jezici zahtevaju višeprolazne komajlere (Java). Većina modernih komapjlera koristi veoma veliki broj prolaza kroz kod. Pravila dosega u višeprolaznim kompajlerima:

- *Prvi prolaz* - kompletno parsiranje ulaznog koda i kreiranje ASTa
- *Drugi prolaz* - prolazak kroz AST i skupljanje informacija o klasama
- *Treći prolaz* - prolazak kroz AST i provere raznih osobina

12 LLVM osnovne informacije. Značaj i mogućnosti

Projekat ***LLVM*** sastoji se iz biblioteka i alata koji zajedno čine veliku kompajlersku infrastrukturu. Započet je kao istraživački projekat na Univerzitetu Ilinois, 2000. godine, kao istraživački rad sa ciljem proučavanja tehnika kompajliranja i kompajlerskih optimizacija. Ideja: skup modularnih i ponovno iskoristivih kompajlerskih tehnologija, čiji je cilj podršska statičkoj i dinamičkoj kompilaciji proizvoljnih programskih jezika. Osnovna filozofija LLVM-a je da je „svaki deo neka biblioteka" i veliki deo koda je ponovno upotrebljiv. Inicijatori projekta su Kris Latner i Vikram Adve.
LLVM je sveobuhvatni naziv za više projekata koji zajedno čine potpun kompajler: prednji deo, središnji deo, zadnji deo, optimizatore, asemblere, linkere, libc++ i druge komponente. Projekat je napisan u programskom jeziku C++, koristeći prednosti objektno-orijentisane paradigme, generičkog programiranja, a takođe sadrži i svoje implementacije raznih struktura podataka koje se javljaju u standardnim bibliotekama programskog jezika C/C++. *Clang/Clang++* se često koristi kao sinonim za LLVM kompajler. Clang/Clang++ ima odlične karakteristike u poređenju sa kompajlerima kao što su gcc i icc. Kada se vrše poređenja, ona se vrše na odabranim primerima. U zavisnosti od primera (benchmarks), poređenja mogu da daju različite rezultate. Obično Clang/Clang++ ima brže vreme kompilacije u odnosu na pomenute kompajlere.
Licenca koda u okviru LLVM projekta je "Apache 2.0 License with LLVM exceptions". Licenciranje se menjalo tokom razvoja projekta, ali je uvek bila licenca otvorenog koda.
LLVM implementira kompletan tok kompilacije:

- *Front-end* - leksička, sintaksička i semantička analiza (alat Clang)
- *Middle-end* - analize i optimizacije (alat opt)
- *Back-end* - različite arhitekture (alat llc)

Na primer, korisnik može da kreira svoj front-end, i da ga poveže na LLVM, koji će mu dodati middle-end i back-end za izabran postojeći front-end/back-end; da isprobava promene na nivou middle-end-a; za novu arhitekturu obezbedi back-end, i da koristi Clang i postojeći middle-end.

13 LLVM projekti

LLVM - kompajlerska infrastruktura sa velikim brojem pomoćnih alata, podržanih arhitektura, sa interfejsima ka ostvarivanju različitih analiza i optimizacija.
Dva osnovna podprojekta LLVM-a su:

- *LLVM Core libraries* - obezbeđuju moderni optimizator koji ne zavisi od izvornog koda niti od ciljne arhitekture, kao i podršku za generisanje koda za puno popularnih CPU-a. Ove biblioteke koriste posebnu reprezentaciju koda koja se naziva LLVM međureprezentacija.
- *Clang* - LLVM-ov C/C++/Objective-C kompilator, koji ima za cilj efikasnu kompilaciju, veoma detaljnu dijagnostiku, obezbeđivanje platforme za izgradnju različitih alata koji rade na nivou izvornog koda. Clang Static Analyzer i Clang-tidy su alati koji automatski pronalaze greške u kodu.

Ostali LLVM projekti:

- *LLDB* - LLVM debager (nove generacije). Izgrađen je kao skup ponovno upotrebljivih komponenti koje koriste postojeće biblioteke LLVM projekta. Veoma je vremenski i memorijski efikasan, efikasniji od GDB-a po nekim kriterijumima.
- *libc++ i libc++ ABI* - standardna biblioteka. Pružaju efikasnu i u skladu sa standardom implementaciju C++ standardnih biblioteka, uključujući punu podršku za C++11 i C++14.
- *compiler-rt* - podrška za različite run-time izazove. Sastoji se od nekoliko manjih podprojekata, uključujući builtins, sanitizer runtimes, profile.
- *MLIR* - nova međureprezentacija. Ima za cilj da smanji cenu izgradnje novih kompajlera i da pomogne u povezivanju postojećih kompajlera, kao i da unapredi kompilaciju koja ima za cilj heterogen hardver. Omogućava dizajn i implementaciju generatora koda, translatora i optimizatora različitih nivoa apstrakcije.
- *OpenMP* - podrška za Open Multi-Processing kod. Obezbeđuje runtime podršku za korišćenje OpenMP konstrukta u okviru Clang-a.
- *polly* - optimizator petlji i upotrebe podataka, kao i infrastruktura za optimizaciju u okviru LLVM-a. U projektu se koristi apstraktna matematička reprezentacija za analizu i optimizaciju pristupa memoriji programa. Trenutno izvodi klasične transformacije petlji, posebno razdvajanje (tiling) i spajanje (fusion) petlji.
- *libclc* - standardna biblioteka za OpenCL. OpenCL je okvir za pisanje paralelnih programa koji se mogu izvršavati na heterogenim platformama. OpenCL pruža standardni interfejs za paralelno izračunavanje korišćenjem paralelizma zadataka i paralelizma podataka.
- *klee* - simboličko izvršavanje koda. Implementira simboličku virtuelnu mašinu koja ima za cilj automatsko generisanje testova i pronalaženje grešaka u kodu. Koristi dokazivač teorema za evaluaciju putanji i uslova ispravnosti.
- *lld* - LLVM linker. Zamena za sistemske linkere koja bi trebalo da radi brže.

14 LLVM prednji deo

Prednji deo (front-end) - prevođenje i analiza izvornog koda programa napisanih u višim programskim jezicima u LLVM-ovu međureprezentaciju. Prevođenje obuhvata leksičku, sintaksičku i semantičku analizu, a završava se fazom generisanja LLVM-ovog međukoda.
Clang je prednji deo kompajlera koji prevodi C-olike jezike u srednju reprezentaciju. Deo je paketnih sistema za skidanje programa na većini BSD i GNU/Linux distribucija. Često se Clang spominje kao kompajler ne uzimajući u obzir da je on samo prednji deo kompajlera i da se ispod njega nalazi moćan optimizator i generator koda. Clang je dizajniran tako da bude memorijski efikasan, da ispisuje jasne, pouzdane i korisne poruke o greškama i upozorenjima. Takođe on pruža čist API tako da se on može koristiti u drugim projektima.

- *LibClang* - ceo sistem LLVM-a je organizovan kao skup biblioteka, pa tako imamo i biblioteku libClang koja treba da se uključi ako želimo da korisimo API kompajlera Clang.
- *Clang Plugins* - omogućavaju pokretanje dodatnih akcija na ASTu kao deo kompilacije. Plugins su dinamičke biblioteke koje se učitavaju u fazi izvršavanja i lako se integrišu u okruženje za izgradnju koda.
- *LibTooling* - C++ interfejs koji ima za cilj da omogući pisanje samostalnih alata.

15 LLVM srednji deo. LLVM-ov međukod

Središnji deo (middle-end) obuhvata skup mašinski nezavisnih optimizacija koje se izvršavaju nad LLVM-ovom međureprezentacijom. LLVM-ova međureprezentacija predstavlja sponu između prednjeg i zadnjeg dela kompilatora. LLVM-ov međukod može biti predstavljen pomoću narednih, međusobno ekvivalentnih formi:

- memorijska reprezentacija (in-memory) - implicitna forma međukoda, smeštena u radnoj memoriji računara
- bitkod reprezentacija (bitcode) - prostorno efikasna reprezentacija, smeštena u datotekama ekstenzije bc
- pseudo-asemblerska reprezentacija (pseudo-assembly) - lako čitljiva, smeštena u datotekama ekstenzije ll

LLVM-ov međukod organizovan je u hijerarhijsku strukturu koja se sastoji od nekoliko važnih entiteta. Sadržaj datoteke u kojoj se nalazi međukod definiše *modul*, koji predstavlja najviši entitet hijerarhije. Modul odgovara jednoj jedinici prevođenja i sačinjen je od proizvoljnog broja funkcija i drugih entiteta. Funkcije konceptualno odgovaraju funkcijama programskog jezika C. Naredni entitet u hijerarhiji predstavljaju *osnovni blokovi*, od kojih je sačinjena svaka funkcija. Osnovni blok predstavlja najduži niz troadresnih instrukcija koje se izvršavaju sekvencijalno. Važnu karakteristiku osnovnog bloka predstavlja činjenica da se u njega može uskočiti jedino sa početka, a iskočiti jedino sa kraja istog. Hijerarhijski najniže entitete predstavljaju same *instrukcije*. Instrukcije su slične RISC instrukcijama, ali sa ključnim dodatnim informacijama koje omogućavaju efektivne analize.
Najvažnije jezičke osobine srednje reprezentacije su: poseduje svojstvo jednog statičkog dodeljivanja koje omogućava lakšu i efikasniju optimizaciju koda; instrukcije za obradu podataka su troadresne; poseduje neograničen broj registara.

16 LLVM srednji deo. Alat opt i LLVM prolazi

Alat opt vrši analizu i optimizaciju koda. Kao ulazni fajl, očekuje LLVM-IR kod (.llformat ili .bc format) i onda pokreće specifičnu optimizaciju ili vrši analizu koda, u zavnosti od ulaznih opcija (ukoliko je zadata ulazna opcija -analyze vrši se analiza koda). Ukoliko opcija -analyze nije zadata, opt pokušava da proizvede optimizovan izlazni kod. Svaka optimizacija je zadata u vidu prolaza kroz kod i moguće je korišćenjem alata opt pokrenuti izabrane optimizacije u odabranom redosledu. Da bi se pokrenula neka konkretna optimizacija, navodi se ime odgovarajućeg prolaza $-\{passname\}$. Redosled pojavljivanja opcija u komandnoj liniji određuje redosled kojim će prolazi biti izvršeni. Neke transformacije zahtevaju da su pre njih izvršene odgovarajuće analize i da su prikupljeni potrebni podaci.
Prolazi mogu da rade analizu *Analysis Passes* ili optimizaciju *Transformation Passes*. Postoje i prolazi koji omogućavaju dobijanje raznih važnih informacija, npr. informacija koje su bitne za razvoj drugih prolaza ili prolazi koji upsuju modul u odgovarajući bitcode. Ovi prolazi se zovu *Utility Passes*. Da biste napisali neku optimizaciju/prolaz potrebno je da - znate šta želite da vaš prolaz uradi; da znate na koji način zavisite od analiza programa koje vrše neki drugi prolazi; da znate na koji način utičete na analize programa koje su već izvršene. Svi prolazi su podklase apstraktne klase *Pass* - definiše virtuelne metode koje svi prolazi treba da implementiraju. Svi prolazi imaju neka svoja pravila šta mogu/ne mogu da rade i kako moraju da se ponašaju. *PassManager* ima ulogu da obezbedi ispravno pokretanje prolaza, tj. da obezbedi da su ispoštovane njihove međusobne zavisnosti, kao i da optimizuje redosled pozivanja prolaza. Zbog toga svaki prolaz mora da obezbedi informacije koji su prolazi pre njega neophodni i šta je od postojećih informacija narušeno nakon datog prolaza. PassManager obezbeđuje deljenje rezultata analize - pokušava da izbegne ponovno računanje analiza uvek kada je to moguće. Bitno je da nakon svakog prolaza mora da se ostavi IR u validnom stanju.

17 LLVM zadnji deo

Zadnji deo (backend) - generisanje mašinskog koda za navedenu ciljnu arhitekturu, na osnovu prosleđenog LLVM-ovog međukoda, vrši se u zadnjem delu infrastrukture. Takođe, u ovom delu se vrše mnoge mašinski zavisne optimizacije. Neke od trenutno podržanih arhitektura su: MIPS, ARM, x86, SPARC. Osnovni alat zadnjeg dela je llc - kompajler koji prevodi LLVM bitcode u asemblerski fajl. Da bi se ostvarila podrška za novi hardver, potrebno je:

- Poznavati LLVM-IR, jer zadnji deo počinje obradu nad IR-om
- Poznavati tehnike pisanja prolaza
- Poznavati *td* format - recizira način definisanja karakteristika ciljne arhitekture sa ciljem bržeg i jednostavnijeg automatskog generisanja odgovarajućih C++ klasa i metoda. Sintaksa liči na generičko programiranje u C++-u.
- Poznavati algoritme koji se koriste u fazi generisanja koda: izbor instrukcija, redosled instrukcija, SSA-zasnovane optimizacije, alokacija registara, ubacivanje prologa/epiloga funkcija, kasne optimizacije na mašinskom kodu i emitovanje koda (za svaku od prethodnih faza postoji skup klasa koje treba poznavati i predefinisati).

18 Semantička analiza. Ime, doseg i tabela simbola. Operacije nad tabelom simbola

Semantičkom analizom proverava se, na primer:

- Da li su svi identifikatori deklarisani na mestima na kojima se upotrebljavaju?
- Da li se poštuju navedeni tipovi podataka?
- Da li su odnosi nasleđivanja u objektno orijetnisanim jezicima korektni?
- Da li se klase definišu samo jednom?
- Da li se metode sa istim potpisom u klasama definišu samo jednom?

Izazovi semantičke analize: odbaciti što više nekorektnih programa; prihvatiti što više korektnih programa; uraditi to brzo; prikupiti i druge korisne informacije o programu koje su potrebne za kasnije faze.
Dve vrste semantičke analize: *Scope-Checking* (provera dosega) i *Type-Checking* (provera tipova).
Isto *ime* u programu može da referiše na više različitih stvari.
Doseg nekog objekta je skup lokacija u programu gde se uvedeno ime može koristiti za imenovanje datog objekta. Uvođenje nove promenljive u doseg može da sakrije ime neke prethodne promenljive. Na koji način pratimo vidljivost promenljivih? Za to postoji *tabela simbola* (symbol table).
Tabela simbola je preslikavanje koje za svako ime određuje čemu to ime konkretno odgovara. Kako se izvršava semantička analiza, tabela simbola se osvežava i menja.
Operacije nad tabelom simbola. Tabela simbola je tipično implementirana kao stek kataloga (mapa). Svaka mapa odgovara jednom konkretnom dosegu. Osnovne operacije su: *push scope* (ulazak u novi doseg), *pop scope* (napuštanje dosega, izbacivanje svih deklaracija koje je doseg sadržao), *insert symbol* (ubacivanje novog unosa u tabelu tekućeg dosega) i *lookup symbol* (traženje čemu neko konkretno ime odgovara).
Da bi se obradio deo programa koji kreira neki doseg potrebno je ući u doseg, dodati sve deklarisane promenljive u tabelu simbola, obraditi telo bloka/funkcije/klase i izaći iz dosega.

19 Doseg i tabela simbola u OOP. Određivanje dosega kod nasleđivanja. Razrešavanje višeznačnosti

Doseg izvedene klase obično čuva link na doseg njene bazne klase. Traženje polja klase prolazi kroz lanac dosega i zaustavlja se kada se pronađe odgovarajući identifikator ili kada se pojavi semantička greška.
Kod nasleđivanja je potrebno održavati još jednu tabelu pokazivača koja pokazuje na stek dosega. Kada se traži vrednost u okviru specifičnog dosega, počinje se pretraga od tog konkretnog dosega. Neki jezici omogućavaju skakanje do proizvoljne bazne klase (npr. C++).
Pojednostavljena pravila dosega za C++. U okviru klase, pretraži celu hijejrhiju da pronađeš koji skupovi imena se tu mogu naći (koristeći standardnu pretragu dosega). Ako se pronađe samo jedno odgovarajuće ime, onda je pretraga završena bez dvosmislenosti. Ako se pronađe više nego jedno odgovarajuće ime, onda je pretraga dvosmislena i mora se zahtevati razrešavanje pretrage. U suprotnom, počni ponovo pretragu ali van klase.

20 Određivanje dosega. Dinamički dosezi

Neki jezici koriste dinamičko određivanje dosega, koje se sprovodi u fazi izvršavanja: ime odgovara varijabli sa tim imenom koja je najbliže ugnježdena u fazi izvršavanja.
Primeri jezika sa dinamičkim dosezima: Perl, Common Lisp. Implementacija dinamičkih dosega uključuje čuvanje tabele simbola u fazi izvršavanja. Obično je to manje efikasno od statičkog određivanja dosega jer kompajleri ne mogu da hardkoduju lokacije promenljivih već imena moraju da razrešavaju u fazi izvršavanja.

21 Pravila za određivanje tipova u izrazima

Notacija tipova zavisi od programskog jezika ali uključuje: skup vrednosti i skup operacija nad tim vrednostima. Greška u radu sa tipovima se javlja onda kada se primenjuje operacija nad vrednostima koje ne podržavaju tu operaciju. Vrste provere tipova:

- Statička provera tipova - analizira program u fazi kompilacije kako bi pokazao da nema grešaka u tipovima. Nikada ne dozvoliti da se nešto loše desi u fazi izvršavanja
- Dinamička provera tipova - proveriti operacije u fazi izvršavanja, neposredno pre konkretnog izvršavanja. Preciznije od statičkog određivanja tipova, ali manje efikasno
- Bez analize tipova

Pravila koja definišu šta je dozvoljena operacija nad tipovima formiraju *sistem tipova*. U jako tipiziranim jezici svi tipovi moraju da se poklapaju. U slabo tipiziranim jezicima mogu se desiti greške u tipovima u fazi izvršavanja. Jako tipizirani jezici su robusni, ali slabo tipizirani jezici su obično brži.
Dva osnovna koraka statičkog zaključivanja tipova: zaključivanje tipa za svaki izraz na osnovu tipova komponenti izraza i potvrđivanje da se tipovi izraza u određenom kontekstu poklapaju sa očekivanim. Ovi koraci se često kombinuju u jednom prolazu.
Kako odrediti tip izraza? Ovaj proces odgovara logičkom zaključivanju: počinjemo od skupa aksioma i onda primenjujemo pravila zaključivanja da bi odredili tip izraza. Mnogi sistemi tipova se mogu posmatrati kao sistemi za dokazivanje.
Jednostavna pravila zaključivanja:

1. Ako je x promenljiva koja ima tip t, izraz x ima tip t
2. Ako je e celobrojna konstanta, onda e ima tip int
3. Ako operandi e1 i e2 izraza e1+e2 imaju tip int i int, onda i izraz e1+e2 ima tip int

Ovakav zapis treba da se formalizuje i skrati. Aksiome i pravila zaključivanja mogu se zapisati na sledeći način: $\frac{Preduslov}{Postuslov}$. Ako je preduslov tačan, možemo da zaključimo postuslov.
Početne aksiome: $\frac{}{\vdash true : bool}$ i $\frac{}{\vdash false : bool}$.
Jednostavnija pravila: $\frac{i \text{ is an integer constant}}{\vdash i : int}$, $\frac{s \text{ is a string constant}}{\vdash s : string}$, $\frac{d \text{ is a double constant}}{\vdash d : double}$.
Složenija pravila: $\frac{\vdash e1 : int, \vdash e2 : int}{\vdash e1+e2 : int}$, $\frac{\vdash e1 : double, \vdash e2 : double}{\vdash e1+e2 : double}$.
Ovakve definicije su precizne i ne zavise od bilo koje konkretne implementacije. Daju maksimalnu fleksibilnost za implementaciju: provera tipova se može implementirati bilo kako sve dok prati zadata pravila. Ovako zadata pravila omogućavaju formalno dokazivanje svojstva ispravnosti programa.
Potrebno je da ojačamo naša pravila zaključivanja sa kontekstom tako da se pamti u kojim situacijama su rezultati validni. Dodavanje dosega: $S \vdash e : T$. U dosegu S, izraz e ima tip T. Tipove sada dokazujemo relativno za doseg u kojem se nalazimo.

22 Tipovi i nasleđivanje. Tip null

Potrebno je ubaciti nasleđivanje u sistem tipova. Važno je uzeti u obzir oblik hijerarhije klasa. Osobine nasleđivanja i konvertovanja:

- Svaki tip nasleđuje samog sebe (refleksivnost)
- Ako A nasleđuje B i ako B nasleđuje C, onda i A nasleđuje C (tranzitivnost)
- Ako A nasleđuje B i ako B nasleđuje A, onda su A i B istog tipa (antisimetričnost)
- Ako A nasleđuje B, onda se objekat klase A može pretvoriti (konvertovati) u objekat klase B (A $\leq$ B)
- Svaki tip se može pretvoriti (konvertovati) u samog sebe (refleksivnost) - A $\leq$ A
- Ako se A može pretvoriti u B i ako se B može pretvoriti u C, onda se i A može pretvoriti u C (tranzitivnost) - A $\leq$ B i B $\leq$ C povlači A $\leq$ C
- Ako se A može pretvoriti u B i ako se B može pretvoriti u A, onda su A i B istog tipa (antisimetričnost) - A $\leq$ B i B $\leq$ A povlači A = B
- Ako je A osnovni tip ili niz, A se može pretvoriti samo u samog sebe

Tip null. Definišemo novi tip koji odgovara literalu *null*. Definišemo da važi da je null tip $\leq$ A za svaki klasni tip A. Ovaj tip obično nije dostupan programerima, već se koristi samo interno u okviru sistema tipova.

23 Određivanje tipova kod ternarnog operatora

Ternarni uslovni operator ? izračunava izraz i vraća jednu od dve vrednosti. Može se koristiti sa osnovnim tipovim, a može se koristiti i sa nasleđivanjem.
Gornja granica za dva tipa A i B je tip C takav da važi: $A \leq C$ i $B \leq C$. Najmanja gornja granica za tipove A i B je tip C takav da važi: C je gornja granica za A i B; ako je C' gornja granica za A i B, onda je $C \leq C'$. Kada postoji najmanja gornja granica za A i B, to označavamo sa $A \vee B$. Minimalno gornje ograničenje tipova A i B je tip C takav da važi: C je gornje ograničenje za A i B; ako je C' gornje ograničenje od A i B, onda ne važi da je C' $<$ C (dakle, C' ako je uporedivo, ne sme da bude manje, ali ne mora da bude uporedivo sa C). Minimalno gornje ograničenje ne mora da bude jedinstveno. Najmanje gornje ograničenje mora da bude minimalno gornje ograničenje, ali ne važi obratno.
Hijerarhija tipova u višestrukom nasleđivanju više nema drvoliku strukturu. Dve klase mogu da nemaju najmanje gornje ograničenje.

24 Pravila za određivanje tipova u naredbama

Ideja: proširiti sistem izvođenja zaključaka tako da modeluje naredbe. Kažemo da je $S \vdash WF(stmt)$ ako je naredba *stmt* dobro formirana u dosegu S. Sistem tipova je zadovoljen ako za svaku funkciju f sa telom B koji je u dosegu S, možemo da pokažemo da važi $S \vdash WF(B)$.

- Pravilo za break: $\frac{S \text{ is in a for or while loop}}{S \vdash WF(break;)}$
- Pravilo za petlje: $\frac{S \vdash expr : bool,\ S' \text{ is the scope inside the loop},\ S' \vdash WF(stmt)}{S \vdash WF(while\ (expr)\ stmt)}$

- Pravilo za blokovske naredbe: $\frac{S' \text{ is the scope formed by adding decls to } S,\ S' \vdash WF(stmt)}{S \vdash WF(\{decls\ stmt\})}$
- Pravilo za return: $\frac{S \text{ is in a function returning } T,\ S \vdash expr : T',\ T' leq T}{S \vdash WF(return\ expr;)}$ ili $\frac{S \text{ is in a function returning void}}{S \vdash WF(return;)}$

25 Tipovi i propagiranje greške

Kako proveriti da li je program ispravno formiran? Proći rekurzivno kroz AST stablo, za svaku naredbu proveri tip svakog podizraza koji sadrži - prijavi grešku ako ne može tip da se dodeli izrazu, prijavi grešku ako je pogrešan tip dodeljen izrazu.
Statička greška tipova se dešava kada ne možemo da dokažemo da izraz ima odgovarajući tip. Greške u tipovima se lako propagiraju. U sistem tipova uvodimo novi tip koji predstavlja grešku. Ovaj tip je manji od svih drugih tipova i označava se sa $\perp$. Nekada se zove i bottom tip. Po definiciji, važi $\perp \leq A$, za svako A. Kada otkriješ tip Error, pretvaraj se kao da je dokazan izraz tipa $\perp$. Potrebno je samo unaprediti pravila zaključivanja tako da sadrže $\perp$.
U semantičkom analizatoru neophodno je da postoji neka vrsta oporavka od greške, kako bi mogla da se nastavi dalje analiza. Jedan vid oporavljanja od grešaka je tip Error koji smo uveli, ali postoje i drugi slučajevi koje treba rešiti, npr. poziv nepostojeće funkcije, deklaracija promenljive nekog neispravnog tipa... Ne postoje ispravni i pogrešni odgovori na ova pitanja, već samo bolji i lošiji izbor.

26 Preopterećivanje funkcija

Preopterećivanje funkcija - više funkcija sa istim imenom ali različitim argumentima. U fazi kompilacije, analizom tipova argumenata, potrebno je odrediti koju funkciju treba pozvati. U slučaju da ne može da se odredi funkcija koja se najbolje uklapa, treba prijaviti grešku.
Jednostavno preopterećivanje. Počinjemo od skupa preopterećenih funkcija. Najpre izfiltriramo funkcije koje se ne poklapaju i tako dobijamo skup kandidata ili skup potencijalno primenljivih metoda. Ako je skup prazan, prijavi grešku. Ako skup sadrži samo jednu funkciju, izaberi nju. Ako skup sadrži više funkcija izaberi najbolju.
Kako odrediti najbolje poklapanje? Ako postoji više od jednog izbora, izaberi funkciju koja najspecifičnije odgovara datom pozivu. Ako imamo dve funkcije koje su kandidati, A i B, sa argumentima $A_1, A_2, ..., A_n$ i $B_1, B_2, ..., B_n$, kažemo da je $A <: B$ ako važi $A_i \leq B_i$ za svako $i \in \{1, 2, ..., n\}$. Relacija $<:$ je parcijalno uređenje. Funkcija A je najbolji izbor ako za svaku drugu funkciju B koja je kandidat za izbor važi $A <: B$ (tj. ona je bar onoliko dobra koliko i svaki drugi izbor). Ako postoji najbolji izbor (ili najbolje poklapanje), onda se bira ta funkcija. U suprotnom, poziv je višesmislen i to mora nekako da se razreši.
Preopterećivanje u kontekstu variadic funkcija. Prva opcija: smatraj poziv višesmislenim. Druga opcija: smatraj boljom funkciju koja nije variadic - funkcija koja je specifično dizajnirana da podrži neki konkretan skup argumenata je verovatno bolji izbor od one koja je dizajnirana da može da podrži proizvoljno mnogo parametara.
Hijerarhijsko preopterećivanje funkcija. Ideja: Napraviti hijerarhiju funkcija kandidata. Konceptualno to je vrlo slično dosezima. Počni sa najnižim nivoom hijerarhije, i traži u okviru njega poklapanje. Ako pronađeš jedinstveno poklapanje, izaberi ga. Ako pronađeš višestruka poklapanja, prijavi višesmislenost. Ako ne pronađeš poklapanje, idi na naredni nivo u hijerarhiji.

27 Kompletnost i saglasnost sistema tipova. Kovarijanta povratnog tipa. Kovarijanta po argumentu funkcije. Kontravarijanta po argumentu funkcije.

Idealan sistem tipova je kompletan (potpun) i saglasan. *Kompletnost* (potpunost) - ako se program može ispravno izvršiti (sistem tipova kaže da je on ispravan u smislu tipova). *Saglasnost* - ako sistem tipova kaže da je program ispravan (onda važi da se program može ispravno izvršiti). Za većinu programskih jezika, nemoguće je ostvariti i potpunost i saglasnost.
Teško je napraviti dobar sistem tipova, tj. lako je napraviti pravila koja su nesaglasna, a često je nemoguće prihvatiti sve ispravne programe. Da bi se izgradio dobar statički sistem za proveru tipova obično je cilj da se napravi jezik koji je što "kompletniji" a da pritom ima saglasan sistem pravila za proveru tipova. Saglasnost se može obezbediti dokazivanjem narednog svojstva za svaki izraz E: $DynamicType(E) \leq StaticType(E)$. Statički sistemi tipova mogu nekada da odbiju program koji bi se mogao ispravno izvršiti (zato što nisu u mogućnosti da dokažu odsustvo greške u tipovima). Takav sistem tipova naziva se nekompletan (nepotpun).
Kovarijanta po povratnom tipu. Neka funkcija A predefiniše funkciju B, ali neka se ove funkcije razlikuju po povratnom tipu: funkcije A i B su kovarijante po povratnom tipu ukoliko se tipovi argumenata poklapaju, a povratni tip funkcije A se može pretvoriti (konvertovati) u povratni tip funkcije B. Kovarijante povratnog tipa su bezbedne, tj. pravilo koje definiše tipove je i dalje saglasno jer je sa kovarijantom povratnog tipa obezbeđeno važenje uslova $DynamicType(E) \leq StaticType(E)$.
Kovarijanta po argumentu. Neka funkcija A predefiniše funkciju B, ali neka se ove funkcije razlikuju po bar jednom argumentu: funkcija A je kovarijanta po argumentu funkciji B ukoliko se argumenti funkcija poklapaju ili se neki argumenti funkcije A mogu pretvoriti u odgovarajuće argumente funkcije B. Dozvoliti potklasi da napravi restrikciju tipa svog parametra funkcije je suštinski nebezbedno! Pozivi kroz baznu klasu mogu da pošalju objekat pogrešnog tipa.
Kontravarijanta po argumentu. Neka važi $C_a \leq C_b \leq C_c$, tj. neka je C_c bazna klasa za klasu C_b, a C_b bazna klasa za klasu C_a. Neka funkcija A (klase C_a) predefiniše funkciju B (klase C_b), ali neka se razlikuju tipovi arugmenata, tj. funkcija A je kontravarijanta po argumentu funkciji B ukoliko A ima za argument tip superklase C_c umesto tip C_b. Kontravarijante po argumentima su bezbedne. Intuitivno, kada zovemo tu funkciju kroz baznu klasu, funkcija će prihvatiti bilo šta što bi bazna klasa već prihvatila. Ipak, većina jezika ne podržava kontravarijante po argumentima jer povećavaju kompleksnost kompajlera i specifikacije jezika i povećavaju kompleksnost proveravanja predefinisanih metoda.

28 Izvršno okruženje i podaci. Enkodiranje osnovnih tipova, nizova i višedimenzionih nizova

Izvršno okruženje predstavlja skup struktura podataka koje se održavaju u fazi izvršavanja sa ciljem omogućavanja koncepata jezika visokog nivoa. Zavisi od osobina izvornog i ciljnog jezika.
Enkodiranje osnovnih tipova. Osnovni celobrojni tipovi (byte, char, short, int, long, unsigned i sl.) se obično preslikavaju direktno u odgovarajuće mašinske tipove. Osnovni realni tipovi (float, double, long double) se takođe obično preslikavaju u odgovarajuće mašinske tipove. Pokazivači se obično implementiraju kao celobrojni tip koji čuva memorijske adrese.

Veličina celobrojnih tipova zavisi od arhitekture računara.
Nizovi. C stil - elementi su jedan za drugim u memoriji. Java stil - elementi su jedan za drugim, s tim što je prvi element broj elemenata niza. D stil - elementi su jedan za drugim, s tim što svaki element čuva pokazivač na prvi element i na element posle poslednjeg elementa.
Višedimenzioni nizovi. Obično se predstavljaju kao nizovi nizova. Oblik zavisi od nizova koji se koriste.

29 Izvršno okruženje i funkcije. Aktivaciono stablo. Zatvorenja i korutine. Stek izvršavanja

Pozivi funkcija se obično implementiraju korišćenjem steka aktivacionih slogova. Poziv funkcije gura novi aktivacioni slog na stek. Povratak iz funkije skida aktivacioni slog sa vrha steka.
Aktivaciono stablo je stablo struktura koje predstavljaju sve pozive funkcija prilikom nekog konkretnog izvršavanja programa. Zavisi od ponašanja programa, ne može se uvek odrediti u fazi kompilacije. Statički ekvivalent je graf poziva funkcija. Svaki aktivacioni slog čuva kontrolni link prema aktivacionom slogu koji ga je inicirao. Aktivaciono stablo je špageti stek.
Da li možemo uvek da optimizujemo špageti stek? Pretpostavke: jednom kada se funkcija vrati, njen aktivacioni slog ne može da bude ponovo referenciran; svaki aktivacioni slog je ili završio izvršavanje ili je predak tekućeg aktivionog sloga. Ove pretpostavke ne moraju uvek da budu tačne.
Jednom kada se funkcija vrati, njen aktivacioni slog ne može da bude ponovo referenciran. Ova pretpostavka ne važi za zatvorenja.
Zatvorenje Z je ugneždena funkcija u funkciju F koja ima mogućnost pristupa slobodnim promenljivama iz funkcije F, pri čemu je funkcija F završila sa svojim izvršavanjem. Osnovne karakteristike zatvorenja su: to je ugneždena funkcija, ona ima pristup slobodnim promenljivama iz spoljašnjeg dosega i ona je vraćena kao povratna vrednost funkcije u koju je ugneždena. Jezici koji podržavaju zatvorenja obično nemaju stek izvršavanja.
Kontrolni link funkcije je pokazivač na funkciju koja ju je pozvala, koristi se da odredi gde se izvršavanje nastavlja kada funkcija završi sa radom.
Pristupni link funkcije je pokazivač prema aktivacionom slogu u kojem je funkcija kreirana. Koriste ga ugneždene funkcije da odrede lokaciju promenljivih u spoljašnjem dosegu.
Korutine su funkcije koje, kada se pozovu, urade deo posla, i onda vraćaju kontrolu pozivajućoj funkciji, ali mogu da kasnije ponovo nastave sa radom od tog istog mesta. Pretpostavka 2 ne važi za korutine. Korutine često ne mogu da budu implementirane preko steka izvršavanja.
Stek izvršavanja. Svaki aktivacioni slog mora da čuva: sve svoje parametre, sve svoje lokalne promenljive i sve privremene promenljive uvedene sa IR generatorom. Logički izgled stek okvira obično kreira IR generator. Ovaj izgled obično ignoriše detalje o mašinski specifičnim konvencijama poziva funkcija. Fizički izgled stek okvira kreira generator koda. On je zasnovan na logičkom izgledu koji postavlja IR generator i uključuje fram pointer-e, caller-saved registre i slične detalje.
IR pozivna konvencija. Pozivaoc je odgovoran za stavljanje na stek i skidanje sa steka prostora za argumente pozivajuće funkcije. Pozivajuća funkcija je odgovorna za stavljanje i skidanje sa steka svojih lokalnih promenljivih i privremenih promenljivih.
Prenos prametara. Dva česta pristupa - call-by-value i call-by-reference.

30 Izvršno okruženje i objekti. Strukture, objekti i nasleđivanje

Implementacija objekata je veoma složena - veoma je teško napraviti izražajan i efikasan objektno-orijentisan jezik. Koncepti koje je teško implementirati efikasno su: dinamičko raspoređivanje, interfejs i višestruko nasleđivanje i dinamička provera tipova.
Struktura - tip koji sadrži kolekciju imenovanih vrednosti. Najčešći pristup, postaviti svako polje da leži redom kojim su polja deklarisana. Jednom kada se objekat postavi u memoriju, on je samo serija bajtova. Potrebno je znati gde tražiti neko konkretno polje. Ideja: čuvati internu tabelu u okviru kompajlera koja sadrži pomeraje za svako polje. Da bi se pronašlo željeno polje, počni od osnovne adrese objekta i pomeri je unapred za odgovarajući pomeraj.
Jednostruko nasleđivanje. Pojednostavljeno, izgled memorije za izvedenu klasu D je zadat kao izgled memorije za baznu klasu B za kojom sledi izgled memorije za preostale članove klase D tj. pokazivač na baznu klasu koji pokazuje na D idalje vidi objekat B na početku memorije. Operacije koje se izvode na objektu D kroz pokazivač na objekat B su garantovano ispravne, nema potrebe da se proverava na šta tačno B dinamički ukazuje.

31 Izvršno okruženje i funkcije članice klasa. Pokazivač this i dinamičko određivanje poziva

Funkcije članice klasa su kao i obične funkcije, ali sadrže dve komplikacije: kako znati za koji objekat je funkcija vezana i kako znati koju funkciju pozvati u fazi izvršavanja? U okviru funkcije članice klase, ime *this* se koristi za tekući objekat. Ova informacija treba da se iskomunicira funkciji. Ideja: tretiraj *this* kao implicitni prvi parametar funkcije. Svaka funkcija koja ima n-argumenata će u stvarnosti imati n+1 argument pri čemu je prvi parametar pokazivač *this*.
Pokazivač this. Kada se generiše kod, za poziv funkcije članice prosleđuje se još jedan parametar *this* koji predstavlja odgovarajući objekat. U okviru funkcije članice, *this* je samo još jedan parametar funkcije. Kada se implicitno referiše na neko polje od *this*, koristi se ovaj dodatni parametar kao objekat u kojem treba da se traži odgovarajuće polje.
Dinamičko određivanje poziva - znači da odabir poziva funkcije u fazi izvršavanja u zavisnosti od tekućeg dinamičkog tipa objekta. Ideja: u fazi kompilacije napravi listu svih klasa i generiši kod. Ova ideja ima nekoliko ozbiljnih problema: veoma je spora i nije uvek ostvarivo rešenje. Umesto toga, ideja je da se za funkcije napravi sličan pristup kao za podatke.

32 Tabela virtulenih funkcija i tabela metoda. Višestruko nasleđivanje i interfejsi

Tabela virtuelnih funkcija (vtable) je niz pokazivača na implementacije funkcija članica neke klase. Da bi se pozvala funkcija članica klase: odredi statički indeks u virtuelnoj tabeli, prati pokazivač koji se nalazi na tom indeksu u okviru vtable objekta da bi došao do koda funkcije, pozovi tu funkciju. Prednosti: vreme da se odredi funkcija koja će biti pozvana je O(1). Mane: objekti su veći jer svaki objekat mora da ima prostor da skladišti O(M) pokazivača, M je broj funkcija članica. Zbog toga kreiranje postaje sporije.
Dinamička rešavanja u O(1). Kreira se jedinstvena instanca vtable za svaku klasu. Svaki objekat čuva pokazivač na svoj vtable. Svaki objekat može da prati taj pokazivač u O(1) i može da prati index u tabeli u vremenu O(1). Povećava se veličina objekta za O(1). Ovo

rešenje se koristi u većini C++ i Java implementacija. Za prethodno rešenje napravljene su implicitne pretpostavke o jeziku koje dozvoljavaju da vtable radi korektno: svi metodi su poznati statički i jednostruko nasleđivanje. Neki programski jezici ne mogu da rade sa virtuelnim tabelama (PHP).
Opšti okvir nasleđivanja: svaki objekat čuva pokazivač na deskriptor svoje klase; svaki deskriptor klase čuva pokazivač na tabelu metoda i pokazivač na deskriptor bazne klase; da bi se pozvao metod; prati pokazivač do tabele metoda, ako metod postoji pozovi ga, inače navigiraj na baznu klasu i ponovi postupak.
Višestruko nasleđivanje i interfejsi - komplikuju izgled virtuelne tabele jer zahtevaju da metodi imaju konzistentne pozicije kroz sve virtuelne tabele: može da se desi da virtuelna tabela ima neiskorišćene unose. Zbog toga, prilikom višestrukog nasleđivanja ili interfejsa, obično se ne koriste čiste virtuelne tabele. Postoje različiti načini da se ovo efikasno implementira, jedan način je hibridni pristup: korišćenje virtuelnih tabela za standardno nasleđivanje, a za interfejse koristiti opisani metod zasnovan na poređenju stringova.

33 Implementiranje dinamičkih provera tipova

Mnogi jezici imaju potrebu za nekom vrstom provere dinamičkih tipova. Ono što možemo da želimo da odredimo je da li je dinamički tip pretvoriv u neki drugi tip. Kako to implementirati?
Implementirati da svaka virtuelna tabela čuva i pokazivač na svoju osnovnu klasu. Provera da li se objekat može pretvoriti u neki tip S u fazi izvršavanja se svodi na praćenje pokazivača na roditelja u okviru virtuelne tabele sve dok se ne naiđe na tip S ili dok se ne dođe do tipa koji nema roditelja. Vreme izvršavanja ove provere je O(d) gde je d dubina hijerarhije klasa. Da li se ovo može uraditi brže?
Postoji ideja provere pretvorivosti objekata u vremenu O(1), pod pretpostavkom da postoji konstantan i ne previše veliki broj klasa u hijerarhiji, kao i da su svi tipovi poznati statički. Ideja je zasnovana na činjenici: objekat koji je statički tipa A je u fazi izvršavanja tipa B samo ako je A pretvorivo u B tj. $A \leq B$.
Dinamičko određivanje tipova kroz proste brojeve. Dodeli svakoj klasi jedinstven prost broj. Postavi ključ svake klase da bude proizvod njenog prostog broja i svih prostih brojeva njenih nadklasa. Da bi se proverilo u fazi izvršavanja da li se objekat može pretvoriti u tip T: pogledaj ključ objekta; ako je ključ od T deljiv sa ključem objekta, onda se objekat može konvertovati u T; inače, ne može. Ako se proizvod dva prosta broja može smestiti u integer, ova provera se može uraditi u vremenu O(1).

34 Troadresni kod. Aritmetičke i bulovske operacije. Kontrola toka

Troadresni kod - međureprezentacija visokog nivao u kojoj svaka operacija ima najviše tri operanda. U trenutku generisanja troadresnog koda nije bitno misliti na njegovu optimizaciju. Evaluacija izraza sa više od tri podizraza zahteva uvođenje privremenih promenljivih.
Aritmetičke i bulovkse operacije. Dozvoljeni operatori su +, -, *, /, %. Bulovske promenljive se predstavljaju kao celobrojne vrednosti koje mogu imati vrednost nula ili vrednost različitu od nule. Pored aritmetičkih operatora, podržavamo i <, ==, || i &&.
Kontrola toka. Imenovane labele označavaju određene delove koda na koje se može skočiti. Mogu postojati razne instrukcije za kontrolu toka. Goto label; - bezuslovni skok, ifZ value goto label; - uslovni skok.

35 Troadresni kod. Funkcije i stek okviri

Funkcije se sastoje iz četiri dela:

- labela koja označava početak funkcije
- instrukcija *BeginFunc N;* koja rezerviše N bajtova za lokalne i privremene promenljive
- telo funkcije
- instrukcija *EndFunc;* koja obeležava kraj funkcije, kada se do nje dođe ona je zadužena da počisti stek okvir i da vrati kontrolu na odgovarajuće mesto (vraća prostor koji je rezervisan sa *BeginFunc N;*)

Funkcija pozivaoc je odgovorna za smeštanje argumenata funkcije na stek. Pozvana funkcija je odgovorna za smeštanje svojih lokalnih i privremenih promenljivih. Jedan parametar je gurnut na stek od strane pozivaoca korišćenjem instrukcije *PushParam var;*. Prester je oslobođen od strane pozivaoca korišćenjem instrukcije *PopParams N;*.
Da bi se implementirao stek, potrebno je znati gde se nalaze lokalne promenljive, parametri i privremene promenljive. One se čuvaju u odnosu na *frame pointer fp*.
Interno, procesor ima specijalni registar koji se naziva brojač instrukcija (PC) koji čuva adresu naredne instrukcije koja treba da se izvrši. Kad kod funkcije završi sa radom, potrebno je da se PC podesi tako da se nastavi izvršavanje funkcije tamo gde je ono bilo prekinuto. Parametri počinju od adrese fp+4 i rastu naviše. Lokalne i privremene promenljive počinju od adrese fp-8 i rastu naniže. Globalne promenljive počinju od adrese gp+0 i rastu naviše.

36 Troadresni kod za objekte. Dinamičko razrešavanje poziva

Adresa virtuelne tabele objekta može da se referencira kroz ime dodeljeno virtuelnoj tabeli, obično je to ime isto kao ime objekta npr $\_t0 = Base;$. Kada se kreira objekat, mora prvo da se postavi pokazivač na virtuelnu tabelu. Instrukcija *ACall* može da se koristi za poziv metoda kada je za njega dat pokazivač na prvu instrukciju.

37 Algoritam generisanja troadresnog koda

U ovom stadijumu kompilacije, imamo na raspolaganju AST koji je anotiran sa informacijama o dosegu i informacijama o tipovima. Da bi se generisao TAC, potrebno je još jednom obići rekurzivno AST - generiši TAC za svaki podizraz ili podnaredbu; koristeći generisani TAC za podizraze/naredbe, generiši TAC za kompletne izraze i naredbe.
Generisanje TAC-a za izraze. Definiši funkciju $cgen(expr)$ koja generiše TAC koji računa izraz, čuva njenu vrednost u privremenoj promenljivoj i vraća ime te promenljive. Definiši $cgen(expr)$ direktno za atomičke izraze. Definiši $cgen(expr)$ rekurzivno za složene izraze.
Generisanje TAC-a za naredbe. Možemo proširiti funkciju *cgen* tako da radi sa naredbama. Za razliku od *cgen* za izraze, *cgen* za naredbe ne vraća ime privremene promenljive koja čuva vrednost.

38 Optimizacije međukoda. Graf kontrole toka

Cilj optimizacije je da se poboljša kod generisan u prethodnom koraku. To je najvažniji i najkompleksniji deo modernog kompajlera. Razlozi za optimizaciju:

- prilikom generisanja koda uvodi se redundantnost
- programeri često ne razmišljaju o efikasnosti koda

Termin optimizacija označava traženje optimalnog koda za zadati program. Ovaj problem je, u opštem slučaju, neodlučiv. Zapravo, cilj je poboljšanje međureprezentacije, a ne njegova optimizacija. Karakteristike dobrog optimizatora: ne sme da suštinski promeni ponašanje programa; treba da proizvede što efikasniji kod; ne treba da koristi puno vremena. I dobri optimizatori nekada uvedu grešku u kod. Optimizatori nekada promaše da urade neku jednostavnu optimizaciju.
Optimizatori mogu da pokušaju da poboljšaju kod u skladu sa različitim željenim osobinama. Želimo da optimizujemo:

- *vreme izvršavanja* - napraviti program da bude što brži, na račun memorije i energije
- *upotreba memorije* - napraviti što manji program, na račun vremena izvršavanja i energije
- *upotreba energije* - napraviti program da troši što manje energije, biranjem jednostavnih instrukcija, na račun brzine izvršavanja i memorije

Optimizacija međukoda sprovodi pojednostavljivanja koja su validna za sve arhitekture, dok optimizacija koda pokušava da unapredi performanse na osnovu karakteristika ciljne arhitekture računara.
Graf kontrole toka je graf koji sadrži osnovne blokove funkcije. Svaka ivica iz jednog osnovnog bloka do drugog označava da kontrola izvršavanja može da ide od kraja prvog do početka drugog bloka. Postoje i dva čvora koji označavaju početak i kraj funkcije. Lokalna optimizacija radi u okviru jednog osnovnog bloka. Globalna optimizacija radi na grafu kontrole toka funkcije. Međuproceduralna optimizacija radi na celokupnom grafu kontrole toka koji prati pozive funkcija i njihovu međusobnu interakciju.

39 Lokalne optimizacije međukoda. Eliminacija zajedničkih podizraza. Prenos kopiranja. Eliminacija mrtvog koda

Eliminacija zajedničkih podizraza. Ako imamo dve dodele promenljivama: $v1 = a\ op\ b$, ..., $v2 = a\ op\ b$, tako da se vrednosti promenljivih $v1, a\ i\ b$ ne menjaju između ove dve dodele, onda možemo da prepišemo kod na sledeći način: $v1 = a\ op\ b, ..., v2 = v1$. Na ovaj način se eliminiše bespotrebno izračunavanje i pravi se prostor za kasnije optimizacije.
Prenos kopiranja. Ako imamo dodelu $v1 = v2$, onda sve dok $v1$ i $v2$ nemaju ponovo dodeljene neke nove vrednosti, možemo da zapišemo izraze oblika: $a = ...v1...$ kao $a = ...v2...$, pod uslovom da je takvo prezapisivanje u redu.
Eliminacija mrtvog koda. Dodela promenljivoj v se naziva mrtva dodela ako se vrednost te promenljive nigde kasnije ne koristi. Eliminacija mrtvog koda uklanja mrtve dodele iz međureprezentacije. Utvrđivanje da li je dodela mrtva zavisi od toga kojim promenljivama se dodeljuju vrednosti i gde se ta dodela vrši.

40 Implementacija lokalnih optimizacija. Analiza dostupnih izraza. Analiza živosti

Mnoge optimizacije su moguće samo ako se obezbedi odgovarajuća analiza ponašanja programa.
Analiza dostupnih izraza. I eliminacija zajedničkih podizraza i prenos kopiranja zavise od analize dostupnih izraza u programu. Izraz se naziva dostupan ako neka promenljiva u programu čuva vrednost tog izraza. U okviru eliminacije zajedničkih izraza, zamenjujemo dostupni izraz sa promenljivom koja nosi tu vrednost. U prenosu kopiranja, menjamo korišćenje promenljive sa dostupnim izrazom koji ona koristi.
→ smer: unapred; domen: skup izraza dodeljenih promenljivama; funckije prenosa: za dati skup dodela V i naredbu a=b+c, ukloni iz V svaki izraz koji koristi a kao podizraz i dodaj u V izraz a=b+c; početne vrednosti: prazan skup izraza.
Analiza živosti. Analiza koja odgovara eliminaciji mrtvog koda se naziva analiza živosti. Promenljiva je živa u nekoj tački programa ako se kasnije u programu njena vrednost čita pre nego što se u nju nešto novo upiše. Eliminacija mrtvog koda radi tako što sračunava živost svake promenljive i onda eliminiše dodele mrtvim promenljivama. Da bi znali koja promenljiva će biti korišćena u nekoj tački programa, iteriramo kroz sve naredbe osnovnog bloka u obrnutom redosledu. Inicijalno, neki mali skup vrednosti se zna da će biti živ.
→ smer: unazad; domen: skup promenljivih; funckije prenosa: za dati skup promenljivih i naredbu a=b+c, ukloni a iz V i dodaj u V promenljive b i c; početne vrednosti: zavisi od semantike jezika.

41 Lokalna analiza - formalno

U okviru lokalne analize imamo tri suštinska pitanja:

1. U kom smeru se vrši analiza?
2. Na koji način ažuriramo informacije prilikom obrade pojedinačne naredbe?
3. Koje informacije imamo inicijalno?

Lokalna analiza osnovnog bloka se definiše kao uređena četvorka (D, V, F, I) gde je: D smer (unapred ili unazad), V skup vrednosti koje program ima u svakoj tački, F familija funkcija prenosa koje definišu značenje svakog izraza kao funkciju $f : V \to V$, I početna informacija na vrhu (ili dnu) osnovnog bloka.

42 Globalne optimizacije. Glavni izazovi

Globalna analiza je analiza koja radi na grafu kontrole toka. Mnoge optimizacija zasnovane na lokalnoj analizi mogu da se sprovedu i globalno. Neke optimizacije ne mogu da se sprovedu lokalno već mogu samo globalno. Primeri: globalna eliminacija mrtvog koda, globalno kopiranje konstanti, parcijalna eliminacija redudantnosti.
Glavni izazovi. U okviru globalne analize, svaka naredba može imati više prethodnika. Globalna analiza mora imati neku vrstu kombinovanja informacija od svih prethodnika osnovnog bloka. U okviru globalne analize, može postojati puno puteva kroz CFG. Može da bude potrebno da se ponovo izračunaju vrednosti više puta. Treba biti oprezan kako se ne bi desilo da se upadne u beskonačnu petlju. U okviru globalne analize sa petljama, svaki blok može da zavisi od svakog drugog bloka. Da bismo ovo rešili, moramo da dodelimo nekakve početne vrednosti svim blokovima CFGa.

43 Globalna analiza živosti

Na početku, postavi IN[s]={} za svaku naredbu s. Postavi IN[exit] na skup promenljivih za koje se zna da su žive na izlazu. Ponavljaj sve dok ima promena: za svaku naredbu s oblika $a = b + c$, u bilo kom redosledu obilaska - postavi OUT[s] na skup unije od IN[p] za svaki sledbenik p od naredbe s, i postavi IN[s] na $(OUT[s] - a) \cup \{b, c\}$.
Da li je ovakav algoritam dobar? Da bi pokazali korektnost, potrebno je da pokažemo da se algoritam zaustavlja i da kada se zaustavi daje saglasno rešenje:

1. Kada se pronađe da je neka promenljiva živa u nekom delu programa, to uvek važi. Postoji konačno mnogo promenljivih i konačno mnogo mesta na kojima promenljive mogu da postanu žive.

2. Svako pojedinačno pravila, primenjeno na neki skup, korektno ažurira živost skupa. Kada se računa unija dva skupa živih promenljivih, promenljiva je živa samo ako je bila živa na nekoj putanji koja vodi do naredbe.

44 Polumreže sa operatorom spajanja

Polumreža sa operatorom spajanja je uređenje definisano na skupu elemenata. Svaka dva elementa imaju spajanje koje je najveći element koji je manji od oba elementa. Intuitivno, spajanje dva elementa predstavlja kombinovanje informacija od ta dva elementa i to je nekakva uopštenija informacija od informacija koje su bile početne. Postoji jedinstven element na vrhu koji se naziva *top*, koji je veći od svih drugih elemenata. Element na vrhu predstavlja "nema informacija".
Polumreža sa operatorom spajanja je uređeni par $(D, \wedge)$ gde je: D skup koji označava domen elemenata, a $\wedge$ operator spajanja koji je idempotentan $(x \wedge x = x)$, komutativan $(x \wedge y = y \wedge x)$ i asocijativan $((x \wedge y) \wedge z = x \wedge (y \wedge z))$. Ako važi $x \wedge y = z$, kažemo da je z spajanje (ili najveća donja granica) za x i y. Svaka polumreža sa operatorom spajanja ima element na vrhu koji se označava sa $\top$, takav da je $\top \wedge x = x, \forall x$.
Polumreže prirodno rešavaju veliki broj problema na koje nailazimo u globalnoj analizi. Kako kombinujemo informacije iz različitih osnovnih blokova? Koristimo operator spajanja. Koju vrednost dajemo kao početnu vrednost svakom bloku? Vrednost *top*. Kako znamo da će se algoritam završiti? Zapravo, to ne možemo još uvek da garantujemo: potrebno je da postoje odgovarajuće osobine polumreže (konačna visina) kao i funkcije spajanja (monotona funkcija).

45 Algoritmi globalne analize međukoda

Globalna analiza je petorka $(D, V, \wedge, F, I)$ pri čemu je: D smer (napred, nazad), V skup vrednosti, $\wedge$ operator spajanja na ovim vrednostima, F skup funkcija prenosa $f : V \to V$, a I početno stanje.
Algoritam globalne analize: Pretpostavimo da je $(D, V, \wedge, F, I)$ analiza unapred. Postavi OUT[s]=$\top$, za svaku naredbu s. Postavi OUT[begin]=I. Ponavljaj sve dok ima izmena: za svaku naredbu s sa prethodnicima $p_1, ..., p_n$ - postavi $IN[s] = OUT[p_1] \wedge ... \wedge OUT[p_n]$ i postavi $OUT[s] = f_s(IN[s])$; redosled iteracija nije važan.
Ova vrsta analize naziva se okvir toka podataka.

46 Generisanje koda. Izazovi alokacije registara. Naivni algoritam

Generisanje koda. Ciljevi: izabrati odgovarajuće mašinske instrukcije za svaku IR instrukciju; podeliti reusrse mažine; implementirati detalje niskog nivoa izvršnog okruženja. Postoji velika razlika u brzini i veličini memorije. RAM je brza ali veoma skupa, hard disk je jeftin ali veoma spor. Osnovna ideja je dobiti najbolje iz svih svetova korišćenjem različitih vrsta memorije.
Izazovi generisanja koda je da se objekti postave na takav način koji maksimizuje prednosti memorijske hijerarhije. Dodatno, to je potrebno uraditi potpuno automatski tj. bez pomoći programera.
Većina mašina ima skup registara koji su predviđeni za memorijske lokacije. Alokacija registara je proces dodele varijable registru i upravljanje transferom podataka iz i u registre.
Izazovi alokacije registara:

- *Mali broj registara* - obično je neuporedivo manji broj registara nego broj promenljivih koje se koriste u IRu. Potrebno je naći način da se registri ponovo koriste uvek kada je to moguće.
- *Registri su komplikovani* - kod x86 svaki registar se sastoji od više manjih registara i oni se ne mogu koristiti istovremeno; neke instrukcije moraju svoje rezultate da upišu u neke konkretne registre pa to utiče da je upotreba registara ograničena i izborom instrukcija. MIPS: neki registri su rezervisani za neke specijalne namene.

Naivni algoritam za alokaciju registara. Ideja: sačuvaj svaku vrednost u memoriji, učitaj je samo onda kada je potrebna. Generiši kod na sledeći način:

1. generiši instrukciju učitavanja iz memorije u registar
2. generiši kod da izvršiš izračunavanje nad registrima
3. generiši instrukciju upisivanja rezultata nazad u memoriju

Mane: jako je neefikasan (ogroman broj učitavanja i upisivanja u memoriju; gubi se vreme i prostor za vrednosti koje bi mogle da budu samo u registrima; neprihvatljiv pristup za kompajlere).
Prednosti: jednostavnost (može da prevede svaki deo IR koda direktno u asembler u jednom prolazu; ne mora da brine da će ostati bez dovoljno registara; ne mora da brine o pozivima funkcija ili o registrima specijalne namene; dobar je samo ako nam je potreban prototip kompajlera koji radi).

47 Alokacija registara. Linearno skeniranje. Razlivanje registara

Cilj: Pokušaj da držiš u registrima koliko god promenljivih možeš. Ta ideja bi trebala da smanji broj čitanja/pisanja u memoriju i uopšte ukupno korišćenje memorije.
U svakoj tački programa svaka varijabla mora da bude na istoj lokaciji. U svakoj tački programa, svaki registar sadrži najviše jednu živu promenljivu.
Promenljiva je živa u nekoj tački u programu ako se njena vrednost ćita pre nego što se u nju ponovo nešto upiše. Živost promenljivih se može pronaći korišćenjem globalne analize živosti. Živi opseg promenljive je skup tačaka u programu u kojim je promenljiva živa. Živ

interval promenljive je najmanji interval IR koda koji sadrži sve žove opsege promenljive. Živ interval je osobina IR koda.
Ako su dati živi intervali promenljivih u programu, možemo alocirati registre koristeći jednostavan pohlepni algoritam. Ideja: prati koji registri su slobodni u svakoj tački. Kada živi interval počne, daj promenljivoj slobodan registar. Kada se živi interval završi, oslobodi odgovarajući registar.
Ako za promenljivu v ne postoji slobodan registar, onda je potrebno njenu vrednost *prosuti*. Kada je promenljiva prosuta, ona se čuva u memoriji umesto u registru. Kada nam treba registar za promenljivu koja je bila prosuta potrebno je da:

1. iselimo neki postojeći registar u memoriju
2. u taj registar učitamo vrednost prosute promenljive
3. kada završimo, sadržaj registra vraćamo u memoriju i u taj registar vraćamo originalnu vrednost koja je tu bila

Razlivanje je sporo, ali ponekad neophodno.
Prednosti: veoma je efikasan (izvršava se u linearnom vremenu); daje dosta kvalitetan kod; alokacija radi u jednom prolazu; često se koristi za JIT kompajlere.
Mane: neprecizan je zbog toga što koristi intervale, a ne opsege; postoje bolje tehnike.

48 Alokacija registara. Bojenje grafova. Čajtinov algoritam

Graf međusobne zavisnosti registara je neusmeren graf gde važi:

- svaki čbor je jedna promenljiva
- postoji ivica između dve promenljive koje su žive istovremeno u nekoj tački programa
- alokacija registara se svodi na dodelu svakoj promenljivoj različitog registra u odnosu na njegove susede

Opisani problem je ekvivalentan problemu bojenja grafova, koji je NP-težak ako ima bar tri registra. Ne postoji algoritam u polinomnom vremenu koji rešava ovaj problem.
Čajtinov algoritam. Pretpostavimo da pokušavamo da obojimo graf sa k boja. Pronađi čvor sa manje od k ivica. Ako obrišemo ovaj čvor iz grafa i obojimo ono što preostane, možemo da nađemo bojenje i za ovaj čvor kada ga dodamo nazad (sa manje od k suseda, neka boja je sigurno iskorišćena).
Ako ne možemo da pronađemo čvor sa manje od k suseda, ona biramo proizvoljni čvor i beležimo ga kao "problematičan". Kada vrćamo taj čvor, možda će biti moguće da mu se dodeli ispravna boja. U suprotnom, taj čvor će biti prosut u memoriju.
Prednosti: za mnoge CFG-ove, pronalazi odlične dodele promenljivih registrima; pošto se promenljive razlikuju po korišćenju, proizvede se precizan graf međusobne zavisnosti registara; često se koristi u kompajlerima.
Mane: Osnovni pristup se zasniva na NP teškom problemu bojenja grafova; heuristika može da dovede do patološke dodele najgoreg slučaja.

49 Raspoređivanje instrukcija. Graf zavisnosti podataka

Zbog procesorskog pipelining-a, redosled u kojem se instrukcije izvršavaju može da utiče na performanse. ***Raspoređivanje instrukcija*** je pravljenje rasporeda instrukcija sa ciljem da se poboljšaju performanse. Svi dobri kompajleri imaju neku vrstu podrške za raspoređivanje instrukcija.
Zavisnost među podacima u mašinskom kodu je skup instrukcija čije ponašanje zavisi jedno od druge. Intuitivno, skup instrukcija koje ne mogu da se poređaju na drugačiji način. Postoje tri vrste zavisnosti: čitanje nakon pisanja, pisanje nakon čitanja, pisanje nakon pisanja.
Graf zavisnosti podataka prikazuje zavisnosti podataka u okviru osnovnog bloka. To je direktan aciklični graf. Direktan, jer uvek jedna instrukcija zavisi od druge. Acikličan, jer nisu moguće ciklične zavisnosti. Mogu se rasporediti instrukcije u okviru osnovnog bloka u bilo kom redosledu sve dok se ne raspodele instrukcije tako da neka instrukcija prethodi svom roditelju. Ideja: napravi topološko sortiranje zavisnosti podataka i poređaj instrukcije u tom redosledu.
Problem: može postojati puno ispravnih topoloških uređenja grafa zavisnosti podataka. U opštem slučaju, pronalaženje najboljeg rasporeda instrukcija je NP-težak problem.

50 Optimizacije koda zasnovane na upotrebi keša

Upotreba keša se zasniva na dve vrste lokalnosti: vremenska i prostorna. Vremenska: ako je nekoj memoriji skoro pristupano, verovatno će joj biti ponovo pristupano uskoro. Prostorna: ako je nekoj memoriji skoro pristupano, verovatno će i njeni susedni objekti biti takođe uskoro korišćeni. Većina keš memorija je dizajnirano da iskoristi ove lokalnosti tako što se u kešu drže skoro adresirani objekti i tako što se u keš ubacuje i sadržaj memorije u blizini.
Programeri obično pišu kod bez razumevanja o posledicama lokalnosti jer jezici ne prikazuju detalje memorije. Neki kompajleri su sposobni da prepišu kod tako da se lokalnost iskoristi. Primer takve optimizacije je preraspoređivanje petlji.