

Formalizacija LLVM međureprezentacije za verifikaciju transformacija programa

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Mirko Brkušnin
mirkobrkusanin94@gmail.com

maj 2018.

Sažetak

U ovom radu će ukrato biti predstavljena ideja i rezultati rada pod nazivom "Formalizacija LLVM međureprezentacije za verifikaciju transformacija programa" (engl. *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*) [5]. Prikazuje se alat *vellvm* koji služi za rezonovanje o programima izraženim u LLVM međureprezentaciji kao i transformacijama koje se vrše nad njom. Uvodi se formalna semantika koja opisuje tu međureprezentaciju. Alat se gradi uz pomoć *Coq* interaktivnog dokazivača teorema koji uz pomoć više novouvedenih operacionih semantika dokazuje relacije među njima kako bi se omogućilo rezonovanje različitim stilovima i tehnikama dokazivanja. Zatim se vrši verifikacija i testiranje samog alata kao i praktična upotreba nad novim transformacijama kôda.

Sadržaj

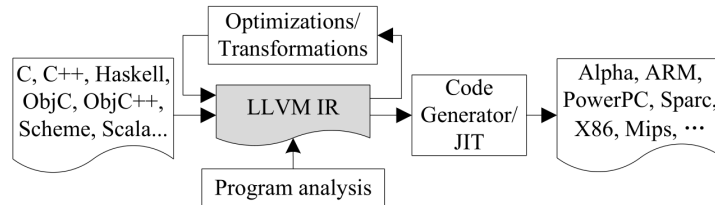
1	Uvod	2
2	Statička svojstva LLVM IR-a	3
2.1	Sintaksa jezika	3
2.2	Statička semantika	3
3	Memorijski model za Vellvm	3
4	Operaciona semantika	4
4.1	Nedeterminizam u operacionoj semantici	4
4.2	Svojstva delimičnosti, održavanja i napretka	4
4.3	Determinističke operacione semantike	5
5	Implementacija	5
6	Verifikacija SoftBound alata	6
7	Zaključak	6
	Literatura	6

1 Uvod

Da li možemo verovati našim kompilatorima? Za kompilirani kôd očekujemo da će se ponašati kako je i opisan semantikom izvornog kôda. Međutim kompilatori, a posebno njihove optimizacije, su kompleksni procesi koji vrše kompleksne simboličke transformacije. I pored intenzivnog testiranja greške se mogu pojaviti. Sa porastom značajnosti softvera raste i naša potreba za rigoroznijim metodama provere. Ovo se odnosi i na ispravnost kompilatora u svim njihovim delovima. Bagovi u kompilatoru koji prevode formalno verifikovan izvorni kôd u izvršni mogu učiniti sve mukotrpno dobijene garancije beznačajnim. [6]

Optimizacije koje kompilatori vrše se najčešće obavljaju na nekoj internoj međureprezentaciji u koju retko imamo uvid. To je slučaj sa GCC-om. Kod LLVM-a međutim imamo precizno definisanu međureprezentaciju poznatu kao LLVM IR (engl. *intermediate representation*). U ovom slučaju optimizacije kompilatora su zapravo prevođenje iz jednog IR kôda u drugi takođe IR kôd. Međutim da bismo mogli vršiti rigorozno dokazivanje svojstava ovih transformacija moramo imati dobro definisanu formalnu semantiku.

LLVM IR je originalno razvijen kao istraživački alat za proučavanje optimizacija kompilatora [2] i čini se prikladnim što su se autori spomenutog rada odlučili baš za ovu međureprezentaciju. Kompilator zasnovan na LLVM-u se sastoji iz prevodioca sa višeg jezika na LLVM IR (slika 1). LLVM zatim nudi komplet transformacija iz IR u IR koje nude optimizacije, programske transformacije i statičke analize. Dobijeni IR kôd se zatim može prevesti na niski kôd ciljne arhitekture kao što su x86, PowerPC, ARM i druge. Iako je prvenstveni fokus LLVM projekta na jezicima C i C++ postoji podrška i za mnoge druge kao što su Haskell, Scala i Objective C što čini potencijalne doprinose još značajnijim nego u slučaju drugih kompilatora.



Slika 1: Infrastruktura LLVM kompilatora

Prvi deo ovog rada se bavi spomenutom formalizacijom IR-a. Predstavljena su sintaksna i statička svojstva koja su značajna u proverama LLVM kôda i njegovih transformacija. Zatim se opisuje memorijski model alata *vellvm* (engl. *verified LLVM*) koji predstavlja proširenu verziju CompCert-ovog modela (jedinog potpuno formalno verifikovanog kompilatora) [3]. Razvijena je operaciona semantika gde je posebna pažnja posvećena rukovanju sa nedeterminizmom koji se nekada javlja u kôdu zbog LLVM-ove eksplicitne **undef** vrednosti. Na kraju će biti predstavljen način implementacije kao i način validacije *vellvm* alata kao i sama efikasnost ove tehlike korišćenjem između ostalog i interaktivni dokazivač teorema Coq.

2 Statička svojstva LLVM IR-a

LLVM IR je tipizirani jezik u SSA formi (engl. *static single assignment*) što ga čini pogodnim za izražavanje velikog broja transformacija i optimizacija kompilatora.

2.1 Sintaksa jezika

Vellvm implementira značajan deo LLVM sintakse ali ne potpun. Neka od svojstava koja nisu prisutna su rukovanje funkcijama sa promenljivim brojem argumenata ili mehanizmi za bacanje i hvatanje grešaka. Opisi velikog broja prisutnih svojstava će ovde biti izostavljeni radi sažetosti (kao što su opis i vrste tipova, postojanje raznih vrsta binarnih operacija i sl.) i pažnja će biti posvećena delovima koji su predstavljali najveće izazove (čitaj: probleme).

Glavni među njima je postojanje **undef** konstante. Ona služi da opiše neinicijalizovane vrednosti kao i da dopusti neke optimizacije. **Undef** semantički predstavlja skup mogućih niza bitova koji se mogu javiti. Drugim rečima **undef** predstavlja skup svih mogućih vrednosti koje može imati ta promenljiva. Biranjem pogodne vrednosti mogu se vršiti neke agresivnije optimizacije. Zbog prisustva takve konstante će operaciona semantika koju vellvm koristi biti nedeterministička.

2.2 Statička semantika

Vodeći se odrednicama LLVM IR-a, vellvm zahteva da svaki LLVM program zadovoljava određene invarijante kako bi se mogao smatrati dobro formiranim. Konkretno, svaka promenljiva u funkciji mora imati dobro određen tip, opseg i može joj biti dodeljena vrednost tačno jednom. Svaka pravilna LLVM transformacija mora održati ove invarijante koje zajedno podrazumevaju da je program u SSA formi.

Svi tipovi osim **void** i funkcijskih pokazivača se smatraju tipovima prve klase. Definicija korisničkog tipa ne sme imati u sebi degenerativne cikluse. Postoje dva sintakсна opsega: globalni i lokalni (koje ne sadrži dodatne ugnježdene opsege). Ovo su samo još neka od svojstava LLVM-a koja je potrebno precizirati radi kasnijeg definisanja operacione semantike.

3 Memorijski model za Vellvm

Važan deo razumevanja LLVM programa je i razumevanje semantike njegovih memorijskih operacija. Postoji dosta svojstava koja se očekuju od razumne implementacije takvih naredbi. Na primer da **load** instrukcija ne utiče na memoriju koju čita ili da **malloc** ne zloupotrebljava zauzete memorijske lokacije. Nažalost LLVM dokumentacija ne nabraja takva svojstva. Donosi se dosta pretpostavki o pravilnom ponašanju takvih naredbi koje se neformalno koriste kao opravdanje ispravnosti transformacija programa. LLVM specifikacija takođe ostavlja neka ponašanja nedefinisanim kao što su čitanje iz nealocirane ili neinicijalizovane memorije.

Izabran je memorijski model koji je zasnovan na CompCert-ovom. Ovo je delom urađeno iz praktičnih razloga jer je i za njegovo dokazivanje ispravnosti takođe korišćen formalni sistem za dokazivanje Coq. Ovaj model dopušta vellvm-u da ispravno implementira LLVM IR kao i da detektuje već spomenute greške. Takođe nasleđuje neke osobine koje potiču iz CompCert-ove implementacije kao što su: da radi u jednoj niti,

pokazivači su 32-bitni i poravnati na četiri bajta kao i da je memorija beskonačna. Zatim je model proširen da podrži dodatna svojstva koja potiču od LLVM-a kao što su mogućnost rukovanja sa celobrojn timer tipovima proizvoljne veličine, dopunjavanjem takvih tipova kao i ograničenjima za poravnanja. Memorijski model takođe definiše ponašanje za sve memorijske instrukcije LLVM-a kao i rukovanje sa osnovnim i proizvoljnim tipovima podataka.

4 Operaciona semantika

Vellvm nudi nekoliko srodnih operacionih semantika za LLVM IR (slika 2). Svaka od njih mora da zna kako da upravlja različitim stanjima greške. Operaciona pravila se mogu opisati kao prelaz između različitih stanja mašine S koja imaju oblik M, \bar{S} gde M predstavlja memoriju, a \bar{S} stek okvira (frejmova). Okviri prate koji skupovi vrednosti su vezani za koje lokalno alocirane promenljive i koje instrukcije se trenutno evaluiraju.

$$\begin{array}{c} \text{LLVM}_{ND} \\ \Downarrow \\ \text{LLVM}_{Interp} \approx \text{LLVM}_D \gtrsim \text{LLVM}_{DFn}^* \gtrsim \text{LLVM}_{DB}^* \end{array}$$

Slika 2: Odnos između različitih operacionih semantika

4.1 Nedeterminizam u operacionoj semantici

Postoji više razloga za pojavljivanje nedeterminizma u LLVM semantici kao što su: postojanje **undef** vrednosti koja označava proizvoljan niz bitova određenog tipa ili različite memorijske greške kao što je čitanje sa neinicijalizovane memorije. Modelujemo ih nedeterministički zato što odgovaraju izborima koji će biti odlučeni prilikom pokretanja programa sa konkretnom implementacijom memorije.

Najosnovnija od svih uvedenih operacionih semantika je LLVM_{ND} koja je nedeterministička relacija evaluacije malog koraka (engl. *small step*) i opisana je pravilima oblika $config \vdash S \rightarrow S'$. U njoj se nedeterminizam pojavljuje na jedan od dva načina. Prvi je zato što stek okviri vezuju lokalne promenljive za neki skup vrednosti V. Drugi zato što relacija \rightarrow može prevesti jedno stanje u više mogućih različitih stanja.

4.2 Svojstva delimičnosti, održavanja i napretka

Pokazano je da nad spomenutom LLVM_{ND} semantikom važe sledeća svojstva. Delmičnost koja označava da program može da zaglavi u jednom od stanja greške kao što su: pozivanje *free* nad memorijom koja nije vraćena od poziva *malloc*, alociranje negativne količine memorije, pokušaj izvršavanja nedostižne naredbe i drugih. Svojstvo održavanja govori o održavanju dobre formiranosti stanja programa kroz njegove transformacije primenom pravila operacione semantike, dok svojstvo napretka kaže da za svako stanje koje nije uspešno završeno ili stiglo u neko stanje greške postoji naredno stanje u koje može preći.

4.3 Determinističke operacione semantike

Vellvm takođe nudi LLVM_D koja je prerađena deterministička semantika LLVM_{ND} semantike koja je takođe malog koraka. Pored nje imamo i dve operacione semantike velikih koraka (engl. *large step*) a to su: LLVM^{*}_{DFn} i LLVM^{*}_{DB}.

Postoji više razloga za uvođenje determinističkih semantika. Kao prvo, pružaju osnovu za testiranje LLVM programa sa konkretnom memorijskom implementacijom. Drugo, pokazujući da je LLVM_D instanca LLVM_{ND} i pokazivanjem odnosa između pravila semantika malih koraka i velikih koraka pruža se validacija svih semantika. Treće, semantike malih i velikih koraka imaju različite primene prilikom rezonovanja o transformacijama LLVM programa.

Kako bi se rešili nedeterminizma koji nastaje od **undef** vrednosti i neispravnih memorijskih operacija korigujemo semantike sa sledećim interpretacijama: **undef** se tretira kao **zeroinitializer** i čitanje neinicijalizovane memorije vraća **zeroinitializer**.

Ostalo je još samo objasniti razliku između semantika malih i velikih koraka. LLVM^{*}_{DFn} jednostavno evaluira funkcijski poziv kao jedan veliki korak dok LLVM^{*}_{DB} evaluira svaki podblok (kôd između dva funkcijska poziva) kao jedan veliki korak. Suprotno njima semantike malih koraka evaluiraju samo jednu naredbu. Semantike velikih koraka su korisne zato što kompilatori često optimizuju transformacijom više instrukcija odjednom u jednom prolazu. Takve transformacije narušavaju svojstvo održavanja u semantikama malih koraka.

Slika 2 takođe ilustruje odnose između operacionih semantika. Vellvm nudi dokaz da LLVM^{*}_{DB} simulira LLVM^{*}_{DFn} kao i da LLVM^{*}_{DFn} simulira LLVM^{*}_D. U tim dokazima pod simuliranjem se podrazumeva da su stanja mašine sintaksički identična u predodređenim tačkama tokom evaluacije.

5 Implementacija

Vellvm kodira ranije opisanu sintaksu u potpunosti u Coq-u koristeći njegove induktivne tipove podataka (uz pomoć dodatnih alata i biblioteka). Coq reprezentacija ipak odstupa od punog LLVM jezika ali samo u par manje značajnih načina. Sintaksna i statička semantika čine oko 2500 linija Coq kôda. Vellvm-ova implementacija memorijskog modela proširuje CompCert-ov sa još 5000 linija koje opisuju novouvedena svojstva. Nakon implementacija svih operacionih semantika dolazimo do 32000 linija Coq kôda.

Coq ima mehanizam izvlačenja kôda (engl. *extraction*) u druge funkcionalne jezike kao što su OCaml, Haskell, Scala. Ova nam nudi mogućnost da izvučemo interpreter pomoću kojeg lakše možemo vršiti testiranja. Kako LLVM_D nije pogodna za izvlačenje kôda uvodi se još jedna semantika LLVM_{Interp} koja je deterministički funkcionalni interpreter implementiran u Coq-u. Ovaj interpreter možemo pokrenuti direktno ali on nije efikasan. Međutim pokazano je da je LLVM_{Interp} srodna LLVM_D semantici, tj. jedna se može svesti na drugu i obratno.

Ova implementacija uspešno ispunjava 134 od 145 testova iz LLVM-ovog paketa regresionih testova koji pokreće LLVM llc (alat za pokretanje LLVM bitkôda). Preostali testovi pokrivaju instrukcije koje vellvm ne implementira.

6 Verifikacija SoftBound alata

Za kraj je predstavljena formalna verifikacija alata SoftBound (koji su potpuno isti autori razvili nešto ranije). Ovo je alat koji dati C kôd prevodi u LLVM IR a zatim vrši instrumentacije kôda koje unose dodatne provere sa ciljem da obezbedi program od narušavanja memorijskog prostora. Primeri grešaka koje se detektuju su: prekoračenje bafera, pogrešno indeksiranje niza, greške prilikom pokazivačke aritmetike i sl. Da bi ceo alat bio formalno verifikovan potrebno je uvesti novu operacionu semantiku koja opisuje sve njegove operacije. Kako SoftBound samo uvodi nove instrukcije i ne menja postojeći kôd ova semantika predstavlja relativno jednostavno proširenje LLVM_{ND} semantike. Autori su uspeli do dokazu korektnost svog alata i usput da pronađu par novih bagova.

7 Zaključak

Vellvm alat služi kao dobar korak prema potpuno verifikovanom LLVM kompilatoru. Fokus ovog rada je formalizacija same LLVM IR semantike koja u tome značajno pomaže pošto najveći deo kompleksnosti potiče upravo od IR u IR transformacija. A zbog svoje višestranosti, LLVM bi mogao ujedno predstavljati verifikovan kompilator za više različitih jezika. Drugi istraživači su u svojim radovima baš to i pokušali ali do sada nemamo nijedno potpuno provereno rešenje [4, 1]. CompCert za sada ostaje kao jedini u potpunosti formalno verifikovani kompilator.

Literatura

- [1] Anna Zaks, Amir Pnueli. Program Analysis for Compiler Validation. 2008.
- [2] LLVM Foundation. LLVM, 2002. on-line at: <http://llvm.org/>.
- [3] INRIA. CompCert, 2007. on-line at: <http://compcert.inria.fr/>.
- [4] Jean-Baptiste Tristan, Paul Govereau, Greg Morrisett. Evaluating Value-Graph Translation Validation for LLVM. 2011.
- [5] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. 2012.
- [6] Xavier Leroy. A formally verified compiler back-end. 2009.