

Kako vršiti statičku analizu u okviru LLVM projekta

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Ognjen Plavšić, 1014/2019
plavsic.ognjen95@gmail.com

15. januar 2020

Sažetak

U ovom tekstu biće opisane 3 metode statičke analize koje su dostupne za upotrebu u okviru LLVM projekta. Prve dve metode, koje koriste interfejse koje pružaju ASTVizitori i LibASTMatcher-i, se baziraju na obilasku clang-ovog apstraktnog sintaksnog stabla. Treća metoda je kompleksnija, i zasniva se na simboličkom izvršavanju koje je omogućeno Clang-ovim statičkim analizatorom.

Sadržaj

1	Apstraktno Sintakšno Stablo (AST)	2
2	ASTVizitori	2
3	LibASTMatcher-i	3
3.1	Kako kreirati matcher	3
3.2	Vezivanje čvorova u match izrazima	4
4	Clang-ov Statički Analizator	4
4.1	Vrste analize koje pruža Clang-ov Statički Analizator	4
4.2	Analiza grafa kontrole toka	5
4.3	Analiza eksplozivajućeg grafa	6
5	Zaključak	7
	Literatura	7

1 Apstraknto Sintaksno Stablo (AST)

U računarstvu, **apstraktno sintaksno stablo** (eng. abstract syntax tree), ili samo **sintaksno stablo**, je drvoidna reprezentacija apstraktne sintaktičke strukture izvornog koda napisanog u programskom jeziku. Svaki čvor stabla predstavlja konstrukt koji se pojavljuje u izvornom kodu. Sintaksa je apstraktna u smislu da ne predstavlja svaki detalj koji se pojavljuje u stvarnoj sintaksi, već samo strukturne ili detalje vezane za sadržaj [2]. Na slici 1 prikazan je izgled AST stabla generisanog Clang kompajlerom u terminalu.

```
FunctionDecl @x10388cd0 <line:3:1, line:3:1> line:3:6 'void (struct S *)'
| ParamVarDecl @x10388cd0 <col:10, col:13> col:13 used 'struct S *'
| CompoundStmt @x10388cd0 <col:16, line:5:1>
| CallExpr @x10388cf70 <line:4:3, col:25> 'void *'
| ImplicitCastExpr @x10388cf58 <col:3> 'void (*)(void *, int, size_t)' <FunctionToPointerDecay>
| DeclRefExpr @x10388ced8 <col:3> 'void (*)(void *, int, size_t)' lvalue Function @x10387a260 'memset' 'void (void *, int, size_t)'
| ImplicitCastExpr @x10388cf58 <col:10> 'void *' <BitCast>
| ImplicitCastExpr @x10388cf50 <col:10> 'struct S *' <ValueToRValue>
| DeclRefExpr @x10388ce20 <col:10> 'struct S *' lvalue ParamVar @x10388cc60 's' 'struct S *'
| IntegerLiteral @x10388ce50 <col:13> 'int' 0
| UnaryExprOrTypeTraitExpr @x10388ceb8 <col:16, col:24> 'unsigned long' sizeof
| ParenExpr @x10388ce98 <col:22, col:24> 'struct S *' lvalue
| DeclRefExpr @x10388ce70 <col:23> 'struct S *' lvalue ParamVar @x10388cc60 's' 'struct S *'
```

Slika 1: Grafička reprezentacija AST stabla generisanog clang kompajlerom

2 ASTVizitori

ASTVizitor je svaka klasa koja nasleđuje šablonsku klasu `RecursiveASTVisitor<Derived>`. Ovo je klasa koja posećuje svaki čvor Clang-ovog AST stabla obilaskom u dubinu. Ova klasa radi tri odvojene stvari:

1. Obilazi AST (npr. posećuje svaki čvor)
2. Za dati čvor, ide uz klasnu hijerarhiju, počevši od dinamičkog tipa čvora, do klase na vrhu hijerarhije (npr. Stmt, Decl ili Type).
3. Za datu kombinaciju (čvor, klasa), gde je klasa neka od baznih klasa dinamičkog tipa čvora, zove funkcije koje korisnik može predefinisati kako bi posetio čvor.

Ove tri stvari rade tri grupe metoda, redom:

1. `TraverseDecl(Decl *x)` radi zadatak 1. Ovo je ulazna tačka za obilazak AST-a sa korenom u čvoru x. Ovaj metod jednostavno poziva metod `TraverseFoo(Foo *x)`, gde je Foo dinamički tip od *x, koji zove metod `WalkUpFromFoo(x)`, a zatim rekurzivno posećuje decu čvora x. `TraverseStmt(Stmt *x)` i `TraverseType(QualType x)` rade na sličan način.
2. `WalkUpFromFoo(Foo *x)` izvršava zadatak 2. Ne pokušava da odmah poseti decu čvora x, umesto toga prvo zove `WalkUpFromBar(x)` gde je Bar direktna nadklasa klase Foo (sem ukoliko Foo nema roditelja), i tek onda zove `VisitFoo(x)`.
3. `VisitFoo(Foo *x)` izvršava zadatak 3.

Ove tri grupe metoda slede naredni poredak: `Traverse > WalkUpFrom > Visit`. Metoda (npr. `Traverse`) može pozvati samo metode iz svoje grupe metoda ili iz grupe metoda direktno ispod nje. Ne može pozvati metode iz grupe iznad [4].

Primer 2.1 *Primer vizitora*

```

1000 class FindNamedClassVisitor
      : public RecursiveASTVisitor<FindNamedClassVisitor> {
1002 public:
      explicit FindNamedClassVisitor(ASTContext *Context)
1004       : Context(Context) {}

1006     bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
      if (Declaration->getQualifiedNameAsString() == "n::m::C") {
1008         FullSourceLoc FullLocation = Context->getFullLoc(Declaration
->getBeginLoc());
        if (FullLocation.isValid())
1010             llvm::outs() << "Found declaration at "
                           << FullLocation.getSpellingLineNumber() << " "
1012                           << FullLocation.getSpellingColumnNumber() << " "
                           << "\n";
        }
1014         return true;
      }

1016 private:
1018     ASTContext *Context;
    };

```

Listing 1: Primer Vizitora

Da bi se izvršila neka analiza izvornog koda pomoću vizitora dovoljno je naslediti klasu `RecursiveASTVisitor<Derived>` i predefinisati željene `Visit` metode u okviru nje. Ukoliko je `Visit` metodama nađena konstrukcija izvornog koda koju iz nekog razloga smatramo pogrešnom, prijavljujemo upozorenje.

3 LibASTMatcher-i

Druga metoda pronalaženja grešaka u izvornom kodu je analiziranje AST stabla pomoću `LibASTMatcher`-a. `LibASTMatcher`-i pružaju oblasno specifičan jezik (eng. domain specific language) za kreiranje predikata nad Clang-ovim AST stablom. Ovaj oblasno specifičan jezik je napisan, i može se koristiti u C++-u, omogućavajući korisnicima da u istom programu izvuku željeni deo stabla i da nad tim čvorovima koriste C++ interfejs izvlačeći razne attribute, lokacije, i ostale informacije dostupne na AST nivou.

Na primer, za kreiranje matcher-a koji izvlači sve deklaracije klasa ili unija u AST stablu neke jedinice prevođenja, može se koristiti poziv `recordDecl()`. Za sužavanje pretrage, na primer za nalaženje deklaracija svih klasa ili unija sa imenom "Foo" treba ubaciti `hasName` matcher: Poziv `recordDecl(hasName("Foo"))` vraća matcher koji izvlači klase i unije sa imenom "Foo" u bilo kom prostoru imena (eng. namespace). Podrazumevano, matcher-i koji prihvataju više drugih matcher-a koriste implicitno `allOf()`. Ovo omogućava dalje sužavanje pretrage. Na primer za izvlačenje klasa koje nasleđuju "Bar" matcher bi izgledao ovako: `recordDecl(hasName("Foo"), isDerivedFrom("Bar"))`.

3.1 Kako kreirati matcher

Uopšteno, strategija kreiranja matcher-a se svodi da sledeće korake:

1. Nađi najspoljašnju klasu u Clang-ovom AST-u koju želiš da izvučeš.
2. Potraži u `AST Matcher Reference` dokumentu matcher koji ili izvlači čvor za koji si zainteresovan ili sužava pretragu.

3. Kreiraj spoljašnji match izraz i proveri da li radi kao što si očekivao.
4. Pregledaj matcher-e koji bi mogli izvući neki unutrašnji čvor iz željenog dela stabla.
5. Ponavlja ovo dok izvlačenje željenog dela stabla nije završeno.

3.2 Vezivanje čvorova u match izrazima

Match izrazi dozvoljavaju izvlačenje delova AST-stabla kako biste uradili određeni zadatak. Često želimo da nesto uradimo sa izvučenim delom stabla, recimo pravljenje transformacija izvornog koda (eng. source code transformations).

Zbog toga, matcher-i koji izvlače određene čvorove AST stabla se mogu vezati. Na primer, `recordDecl(hasName("MyClass")).bind("id")` će vezati izvučeni `recordDecl` čvor za string "id", kako bi se kasnije mogao koristiti u povratnom pozivu matcher-a (eng. match call-back) [1].

4 Clang-ov Statički Analizator

Clang-ov Statički Analizator je alat za traženje grešaka u programu. Analizirajući izvorni kod, ovaj alat pokušava da simulira izvršavanje delova programa bez njihovog prevođenja i prijavljuje greške koje bi se zapravo desile u toku izvršavanja (eng. run-time errors). Kako ponašanje izvršnih programa zavisi od spoljašnjih faktora kao što su ulazne vrednosti, slučajni brojevi i ostale stvari, mehanizam analizatora (eng. analyzer engine) dodeljuje vrednosti sa algebarskim simbolima, i izvršava simbolička izračunavanja koristeći ove simbole. Takođe otkriva uslove nad simboličkim vrednostima i postavlja ograničenja nad njima.

Kao rezultat, Clang-ov Statički Analizator može da nađe greške koje se pojavljuju na retkim putanjama tokom izvršavanja programa. Ipak, analizator može naći samo greške koje je specifično programiran da nađe. Inače, kad naiđe na neki problem, analiza nastavlja dalje i ne prijavljuje nikakvu grešku. Za svaku grešku koju analizator nađe, kao što je na primer dereferenciranje null pokazivača, postoji poseban modul, **checker**, koji reaguje na te greške tokom analize.

Dakle, jezgro analizatora je odgovorno za simboličko izvršavanje programa, a checker-i se prijavljuju za događaje za koje su zainteresovani, proveravaju razne pretpostavke nad simboličkim vrednostima, i prijavljuju upozorenja ukoliko se ove pretpostavke ne mogu postaviti na određenom putu.

4.1 Vrste analize koje pruža Clang-ov Statički Analizator

Prva odluka koju obično morate doneti kad pravite checker je da li vam je potrebna analiza puteva (eng. path-sensitive analysis) tj. simboličko izvršavanje programa. Analiza puteva je mnogo sporija od kompilacije. Većinu zahtevnih izračunavanja izvršava jezgro analizatora, dok su checker-i uglavnom prilično brzi.

O sintaksoj analizi u okviru Clang-ovog Statičkog Analizatora ovde neće biti reči jer su AST checker-i zapravo jako slični AST vizitorima i po implementaciji i po mogućnostima koje pružaju.

Druga vrsta analize je analiza zasnovana na analiziranju grafa kontrole toka (eng. control flow graph).

4.2 Analiza grafa kontrole toka

Graf kontrole toka je grafovska reprezentacija svih puteva kojima se može proći kroz program tokom njegovog izvršavanja. Ovaj graf se pravi odvojeno za svaku funkciju. Svaki čvor grafa kontrole toka predstavlja osnovni blok naredbi koje ne zadrže nikakvo grananje, pa se stoga izvršavaju sekvencijano. Svaki osnovni blok se završava završnom naredbom (eng. terminator statement), koja je granajuća naredba ili naredba **return**.

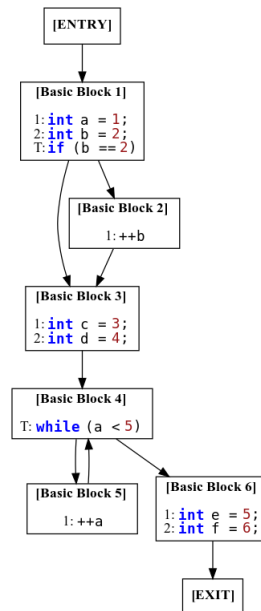
Pogledajmo naredni kod:

```

1000 function() {
1001     int a = 1;
1002     int b = 2;
1003
1004     if (b == 2) {
1005         ++b;
1006     }
1007
1008     int c = 3;
1009     int d = 4;
1010
1011     while (a < 5) {
1012         ++a;
1013     }
1014
1015     int e = 5;
1016     int f = 6;
1017 }

```

Graf kontrole toka koji odgovara prikazanom kodu prikazan je na slici 2:



Slika 2: Grafička reprezentacija grafa kontrole toka

Analiza grafa kontrole toka je korisna za pravljenje sigurnih prover,

za koje je nepotrebno proći kroz sve moguće puteve programa. Na primer, ukoliko želimo da osiguramo da je neki uslov uvek false, pa je onda odgovarajući kod "mrtav", morali bismo da dosegemo sve definicije svih promenljivih koje se koriste u izrazu pomenutog uslova, tako da bi ovde najkorisnija bila analiza grafa kontrole toka.

Međutim, pogledajmo naredni kod:

```

1000 void function(){
      int y, z;
1002  if (x == 0) y = 5;
      if (!x) z = 6;
1004 }

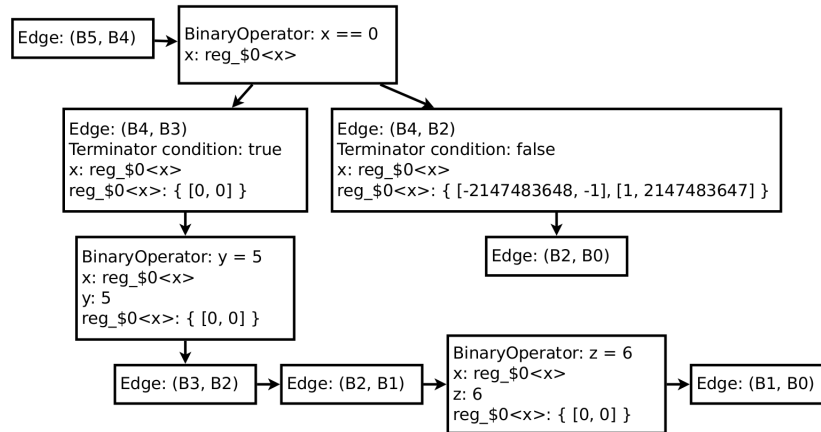
```

Na osnovu analize grafa kontrole toka ne bismo mogli da utvrdimo da ukoliko se izvrši put koji dolazi do bloka tela prvog if-a, sigurno se neće izvršiti put koji dolazi do tela drugog if-a. Analizom grafa kontrole toka bismo morali da ispitamo sve četiri putanje, a zapravo nas interesuju samo dve. Ovo čini analizu dosta manje korisnom.

4.3 Analiza eksplodirajućeg grafa

Eksplodirajući graf Clang-ovog Statičkog Analizatora je osnovna struktura podataka za analizu puteva koje izvršava jezgro statičkog analizatora. Jezgro pokušava da interpretira kod i pravi različite putanje čak i ako te putanje uključuju iste osnovne blokove grafa kontrole toka. Eksplodirajući graf se sadrži od svih puteva kroz graf kontrole toka koji se istražuju od strane jezgra analizatora, i nosi informacije o stanju programa na svakom putu i u svakoj naredbi. Čvorovi grafa, eksplodirajući čvorovi, su parovi koji se sastoje od stanja programa i lokacije u programu koji se trenutno analizira.

Eksplodirajući graf za prethodni kod prikazan je na slici 3:



Slika 3: Grafička reprezentacija eksplodirajućeg grafa

Hajde da vidimo kako analiza puteva radi na ovom primeru. Analizator počinje da simulira prvu operaciju, poređenje vrednosti promenljive x sa nulom. Pošto je x nepoznata u ovom trenutku u analizi, ova vrednost će biti reprezentovana simbolom `reg<x>`.

kada je ovo poređenje simulirano, dostižemo završnu naredbu u grafu kontrole toka, if- naredbu. U zavisnosti od uslova, analizator razvija dva nova puta, jedan u kom je vrednost promenljive x jednaka nuli, i drugi u kom je različita od nule. Na osnovu toga prave se ograničenja nad simbolom `reg<x>`, u prvom slučaju će ograničenje biti 0, dok će u drugom ograničenje biti unija intervala $[-21417483648, -1]$ i $[1, 21417483647]$. U sličnom maniru se dalje nastavlja analiza [3].

5 Zaključak

LLVM infrastruktura nam pruža razne mogućnosti za statičku analizu koda. Ove analize svode se na analize AST stabla i na simboličko izvršavanje. U izboru između ove dve vrste analize treba biti jako pažljiv, s obzirom da je razlika u brzini velika. Stoga analizu zasnovanu na simboličkom izvršavanju treba koristiti samo u slučajevima kada se željeni tip greške ne može otkriti pomoću drugih metoda statičke analize.

Pomoću navedenih alata i interfejsa mogu se praviti novi alati za statičku analizu koji mogu imati razne uloge kao što je provera saglasnosti koda sa nekim standardom ili jednostavno stroža dijagnostika od one koje nam podrazumevano pružaju kompajleri.

Literatura

- [1] Clang 11 documentation Matching the Clang AST . author = Free Software Foundation, year = 2007-2020.
- [2] Abstract syntax tree, Wikipedia. note = on-line at: https://en.wikipedia.org/wiki/Abstract_syntax_tree.
- [3] CLANG STATIC ANALYZER, A Checker Developer's Guide. author = Artem Dergachev, year = 2016.
- [4] clang::RecursiveASTVisitor< Derived > Class Template Reference. author = Free Software Foundation, year = 2007-2020.