Skripta za vezbe iz Konstrukcije kompilatora

Optimizacije (passovi u LLVM-u)

LLVM optimizacije se nazivaju passovima, zato sto podrazumevaju prolazenje kroz programski kod radi prikupljanja nekih relevantnih informacija ili modifikacije postojeceg programskog koda. Razlikujemo tri glavne vrste optimizacija:

1. `Analysis passes` - prolaze kroz IR radi prikupljanja odgovarajucih informacija koje mogu biti korisne drugim vrstama passova;
2. `Transform passes` - prolaze kroz IR radi njegove modifikacije;
3. `Utility passes` - oni passovi koji nisu ni analysis ni transform.

Kako kreirati i pokrenuti odgovarajuci pass

Koraci pri kreiranju passova su isti, bez obzira kojoj kategoriji sam pass pripada:

1. U okviru kloniranog repozitorijuma `llvm-project`, pozicionirati se u direktorijum `llvm/lib`;
2. U zavisnosti od toga koji tip optimizacije kreiramo, pozicioniramo se u neki od direktorijuma `Analysis` ili `Transforms`;
3. Kreiramo novi direktorijum koji nosi ime nove optimizacije, npr.

   ```
   mkdir OurDeadCodeEliminationPass
   ```

4. U novokreiranom direktorijumu, pisemo `CMakeLists.txt` fajl koji nam je neophodan za build. Neka uobicajena struktura ovog fajla je sledeca:

   ```
   add_llvm_library( LLVMOurDeadCodeEliminationPass MODULE
       OurDeadCodeEliminationPass.cpp

       PLUGIN_TOOL
       opt
   )
   ```

   Napomena: **LLVMPassName** oznacava naziv deljene biblioteke koja ce biti kreirana prilikom kompilacije. U .cpp fajlu, pise se logika konkretne optimizacije.

5. U fajl `CMakeLists.txt` koji se nalazi u `llvm/lib` direktorijumu neophodno je dodati direktorijum u kome se nalazi CMakeLists.txt za odgovarajucu optimizaciju. CMake rekurzivno obilazi fajlove da bi prikupio sve sto mu je neophodno za uspesan build.

   ```
   add_subdirectory(OurDeadCodeEliminationPass)
   ```

6. Pomocu skripte `make_llvm`, pokrenuti build celokupnog projekta.

7. Ako je sve proslo uspesno, pozicionirati se u `build` direktorijum koji se nalazi u `llvm-project`, pa zatim izvrsiti pokretanje optimizacije na sledeci nacin:

   ```
   ./bin/opt -load lib/LLVMOurDeadCodeEliminationPass.so -enable-new-pm=
   0 -our-dead-code-elimination-pass example.ll
   ```

   Ukoliko zelimo da eliminisemo poruke koje nam generise LLVM, a koje nam nisu od nekog znacaja, to radimo preusmeravanjem izlaza prethodne komande u `/dev/null`:

   ```
   ./bin/opt -load lib/LLVMOurDeadCodeEliminationPass.so -enable-new-pm=
   0 -our-dead-code-elimination-pass example.ll > /dev/null
   ```

   Napomena: sve sto ukljucuje dead code elimination, menja se nazivima za odgovarajuci pass koji pokrecemo.

Dodatno: Posto nam je neophodan neki kod koji sluzi za testiranje optimizacije, moramo ga prevesti u LLVM-ov medjukod. Ako je recimo nas program bio example.c, transformacija u medjukod vrsi se na sledeci nacin:

```
./bin/clang -emit-llvm -S example.c
```

Kao rezultat ove komande dobijamo fajl sa ekstenzijom `.ll`.

CALL GRAPH - graf poziva funkcija

Za razlicite tipove transform passova, korisna nam je informacija o tome koje funkcije medjusobno interaguju u programu. Konkretno, zanima nas koja funkcija poziva koju. Ovo se relativno jednostavno moze odrediti uz pomoc koriscenja grafovske strukture podataka, koju u ovom slucaju nazivamo **graf poziva funkcija**. Ova struktura nam je od veceg znacaja pri radu kompajlera, zato sto nam moze omoguciti da recimo detektujemo delove koda koji se nikad ne koriste, ali isto ako i da detektujemo razlicite anomalije koje se u programu javljaju.

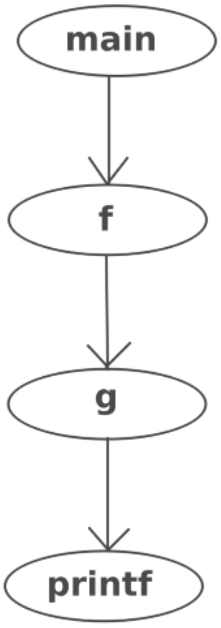
Primer. Razmotrimo naredni segment koda:

```
void f()
{
    g();
}

int main()
{
    f();
}
```

```
void g()
{
    printf(...);
}
```

Graf poziva funkcija koji bi odgovarao prethodno napisanom kodu bi imao naredni oblik:



Iz prethodnog primecujemo da ce graf biti *usmeren* i da grane idu od funkcije pozivaoca ka funkciji koja se poziva.

Kako konstruisati graf poziva funkcija uz koriscenje LLVM infrastrukture?

Da bismo uspesno konstruisali citav graf, neophodno je prvo dohvatiti funkciju `main` od koje i krece izvsavanje citavog programa. Konstrukciju zapocinjemo na nivou modula, a zatim se krecemo dalje kroz funkcije, basic blockove i instrukcije, tako da sustinski prolazimo citavu hijerarhiju IR-a.

Graf predstavljamo u vidu klase koja ce imati sledecu formu:

```
class OurCallGraph {
private:
    std::unordered_map<Function*, std::unordered_set<Function*>>
AdjacencyList;
    std::string ModuleName;
public:
    void CreateCallGraph(Module &M);
    void DFS(Function* F);
    void dumpGraphToFile();
};
```

Napomena: dumpGraphToFile vrsi konverziju dobijenog grafa u `DOT` format.

Graf cemo ispisati u okviru jednog `ModulePass`-a.

Kreiranje:

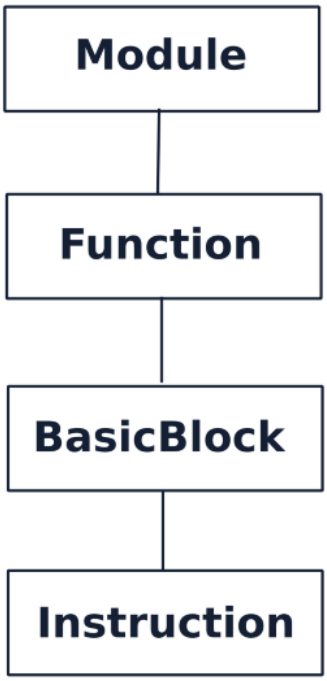
```
void CreateCallGraph(Module &M)
{
    ModuleName = M.getName().str();
    Function* Main = M.getFunction("main");

    if (!Main) {
        errs() << "Main function doesn't exist!\n";
        exit(0);
    }

    DFS(Main);
}
```

- Pronalazimo `main` funkciju iz koje bi trebalo da se zapocne pretraga.
  - Ukoliko ona ne postoji, prijavljujemo gresku i prekidamo dalju obradu.
  - Ukoliko postoji, pozivamo `DFS` kako bismo obisli ceo graf.

Hijerarhija u IR-u



Svaka od prethodno navedenih celina ima odgovarajuci tip u LLVM-u. Ti tipovi nalaze se u zaglavlju `llvm/IR/type` gde mozemo imati:

- **llvm/IR/Module**
- **llvm/IR/Function**
- **llvm/IR/BasicBlock**
- **llvm/IR/Instruction**

Tipovi su dosta korisni jer sadrze razlicite funkcionalnosti koje nama olaksavaju pri radu.

```
void DFS(Function* F)
{
    for (BasicBlock &BB : *F) {
        for (Instruction &Instr : BB) {
            if (auto CallInstruction = dyn_cas<CallInstr>(&Instr)) {
                Function* Callee = CallInstruction->getCalledFunction();
                AdjacencyList[F].insert(Callee);


                if (AjdacencyList.find(Callee) == AdjacencyList.end()) {
                    AdjacencyList[Callee] = {};
                    DFS(Callee);
                }
            }
        }
    }
}
```

- Krecemo se prvo kroz sve `BasicBlock`-ove, a zatim i `Instrukcije`. Kada se u IR-u vrsi poziv neke funkcije, koristi se instrukcija `call` pa je to upravo ono sto proveravamo. Ukoliko jeste, izvlacimo koju funkciju pozivamo datom instrukcijom i ubacujemo je u listu susedstva trenutne funkcije. Proveravamo dalje da li ta funkcija postoji u nasoj mapi i ukoliko ne, vrsimo dalje pretragu za pozvanu funkciju.