

# 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 postojećeg programskog koda. Razlikujemo tri glavne vrste optimizacija:

1. **Analysis passes** - prolaze kroz IR radi prikupljanja odgovarajućih 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 odgovarajući 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 odgovarajuću 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 izvršiti 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 služi za testiranje optimizacije, moramo ga prevesti u LLVM-ov medjukod. Ako je recimo nas program bio `example.c`, transformacija u medjukod vrši 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.

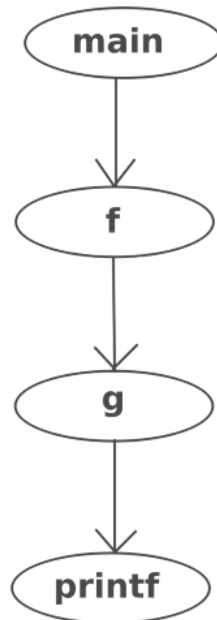
*Primer.* Razmotrimo naredni segment koda:

```
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` vrši konverziju dobijenog grafa u `DOT` format.

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

Kreiranje:

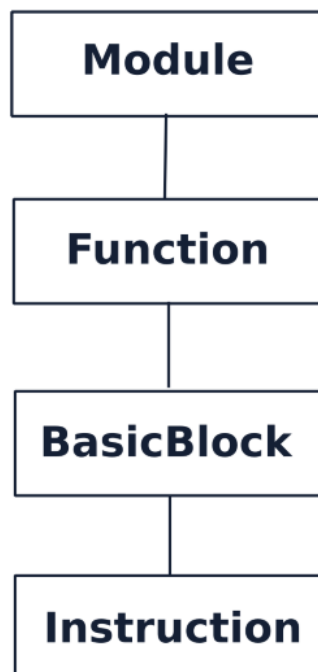
```
void CreateCallGraph(Module &M)
{
    ModuleName = M.getName().str();
    Function* Main = M.getFunction("main");

    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_cast<CallInstr>(&Instr)) {
                Function* Callee = CallInstruction->getCalledFunction();
                AdjacencyList[F].insert(Callee);

                if (AdjacencyList.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.

---

## CFG - Control Flow Graph

**CFG** predstavlja grafovsku reprezentaciju izracunavanja koja se desavaju u toku izvorsavanja samog programa. Cesto se koriste prilikom staticke analize koda, ali imaju i veliku primenu u oblasti kompajlera zato sto mogu precizno da predstavje tok izvorsavanja u okviru neke jedinice koda.

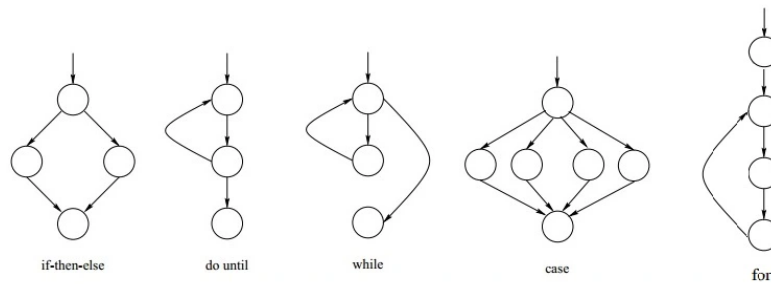
Neke karakteristike CFG-a:

- prikazuje sve putanje koje se mogu obici tokom izvorsavanja samog programa;
- graf je usmeren;
- grane grafa predstavljaju **putanje** u toku kontrole toka, dok su cvorovi **BasicBlock**-ovi.

Obicno postoje dve specijalne vrste blokova u okviru CFG-a:

- **Entry block** - dozvoljava kontroli da udje u CFG;
- **Exit block** - kontrola toka izlazi kroz ovaj blok.

CFG za neke ceste konstrukcije jezika:



## Kako kreirati CFG?

- Kreiramo klasu OurCFG (ili samo CFG) koja ima sledecu strukturu

```
class OurCFG {
private:
    std::unordered_map<BasicBlock*, std::vector<BasicBlock*>>
AdjacencyList;
    std::string FunctionName;

    void DumpBasicBlock(BasicBlock*, raw_fd_stream &File, bool
only = true);

public:
    void AddEdge(BasicBlock*, BasicBlock*);
    void CreateCFG(Function&);
    void DumpToFile();
};
```

- neuredjena mapa cuva BasicBlock-ove, kao i prelaze koji vode od jednog BasicBlock-a ka njegovim **successorima**.
- FunctionName nam služi za kreiranje .dot fajla
- metode za dump takodje imaju ulogu da adekvatno formatiraju graf kako bi bio upisan u .dot fajl
- Funkcija koja kreira CFG za datu funkciju radi prilično jednostavno: prolazi kroz svaki BasicBlock, gleda koji su mu **successori** i zatim ih dodaje u listu povezanosti. Successore nekog BasicBlock-a dobijamo jednostavno uz koriscenje funkcije **successors(BasicBlock\*)**.
- Kreiranje reprezentacije za .dot fajl je u ovom slucaju malo kompleksnije. Kao sto je od ranije poznato, da bismo presli u neki drugi tok izvršavanja, neophodno je da postoji **branch** instrukcija, odnosno ono sto je u LLVM-ovom IR-u predstavljeno preko instrukcije **br**. Medjutim, ona moze biti takva da potencijalno vodi u vise od jednog toka izvršavanja, pa samim tim moramo da obradimo vise slucajeva:
  - Klasican br koji nema nikakvog grananja - instrukcija se standardno upisuje u fajl.
  - br** koji predstavlja **if-then-else** konstrukciju - proveru da li se radi o ovakvom tipu instrukcije vrsimo uz pomoc metode: **isConditional()**. U tom slucaju zapis u .dot fajlu je sledeci:

```
| {<s0>T|<s1>F}
```

3. Instrukcija koju razmatramo je **switch** instrukcija - zbog specifičnog načina definisanja tipa **SwitchInst** u LLVM-u, treba obratiti pažnju na njenu obradu.

```
class SwitchInst : public Instruction {
    unsigned ReservedSpace;

    // Operand[0]      = Value to switch on
    // Operand[1]      = Default basic block destination
    // Operand[2n  ]   = Value to match
    // Operand[2n+1]   = BasicBlock to go to on match
    ...
};
```

Nama su za .dot fajl neophodne isključivo vrednosti koje imamo u svakoj od case naredbi, pa iz tog razloga razmatramo samo **parne** indekse. Takođe, kako nulti i prvi indeks predstavljaju vrednost po kojoj se grana, kao i basic block podrazumevanog slučaja, njih preskacemo.

---

## Dominatorska stabla

Dodati objašnjenje...

---

## Constant propagation

Jedna od standardnih optimizacija koje kompajler sprovodi. Podrazumeva zamenu vrednosti promenljivih sa konstantnim vrednostima u izrazima u kojima se koriste. Konstantne vrednosti promenljivih se mogu propagirati kroz graf kontrole toka.

Primer.

```
int main ()
{
    int x = 3;
    int y, z, w;

    if (y > 0) {
        x = 7;
        z = x + 3;
    } else {
        w = 4;
    }

    return 0;
}
```

Umesto što nakon ulaska u if granu dodeljujemo promenljivoj x vrednost 7, a zatim koristimo x u daljim izracunavanjima, mozemo direktno izvršiti sledeću zamenu:  $z = 7 + 3$  (to će kompajler i uraditi).

Ova optimizacija funkcionise zajedno sa constant foldingom.

## Uvodni termini

Da bi se optimizacija sprovela neophodno je uvesti nekoliko termina. Prvi od njih podrazumeva **stanje promenljive**. Svaka promenljiva se moze naci u 3 stanja:

- **⊥ : bottom** - naredba nije dostizna, pa samim tim ni promenljiva;
- **T : top** - naredba jeste dostizna, ali nam je vrednost promenljive u toj naredbi nepoznata;
- **c : const** - vrednost promenljive u datoj naredbi je konstantna.

Takodje, za svaku naredbu pratimo kakvo je stanje promenljive *pre* i *posle* njenog izvršavanja:

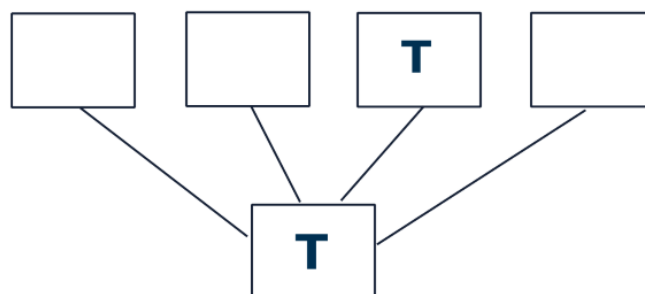
- **C(s, x, in)** - stanje promenljive x pre izvršavanja naredbe s;
- **C(s, x, out)** - stanje promenljive x nakon izvršavanja naredbe s.

Napomena: prva instrukcija u programu **ne sme** imati i before i after status, dok sve ostale **moraju** imati oba.

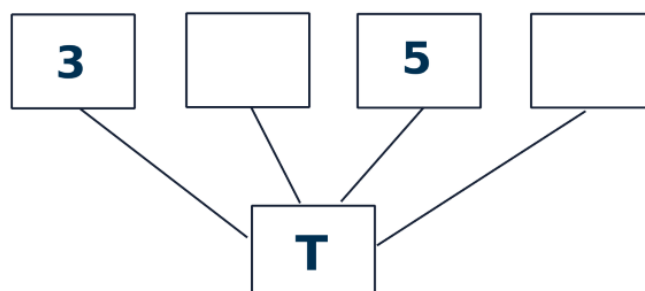
## Sprovođenje optimizacije

Da bismo sproveli optimizaciju, neophodno je da znamo pravila koja za nju vazе:

- **if  $C(p_i, x, out) == T$  for any i then  $C(s, x, in) = T$** 
  - ako je vrednost promenljive nakon bilo kod predecesora tekuceg basic blocka nepoznata, onda neće biti poznata ni pre izvršavanja tekuce naredbe.



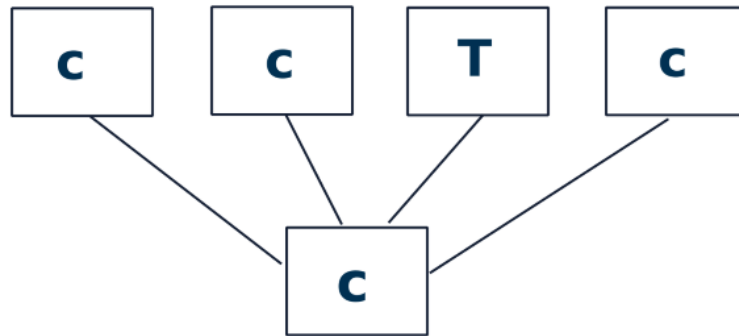
- **if  $C(p_i, x, out) == c \ \&\& \ C(p_j, x, out) == d \ \&\& \ c \neq d$  then  $C(s, x, in) == T$** 
  - ako je vrednost promenljive x u razlicitim predecesorima razlicita, onda ona neće biti poznata pre izvršavanja naredbe s.



- **if  $C(p_i, x, out) == c$  or T for all i then  $C(s, x, in) = c$**

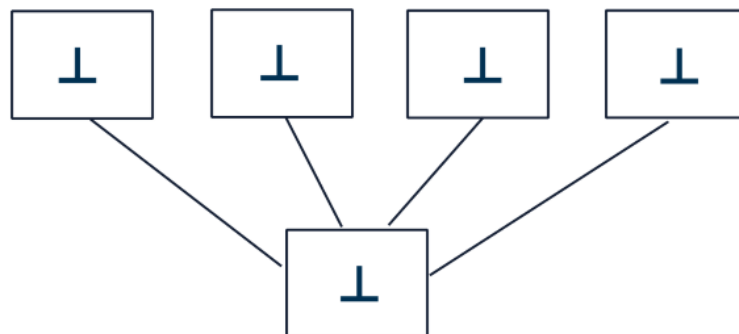


- ako svi predecessori dodeljuju istu vrednost promenljivoj  $x$ , onda ce ta ista vrednost biti i pre ulaska u naredbu  $s$ .



- **if  $C(p_i, x, out) == \perp$  for all  $i$  then  $C(s, x, in) = \perp$**

- ako je svaki predecessor nedostizan, ona ce i u naredbi  $s$  vrednost promenljive  $x$  biti nedostizan.



- **$C(s, x, out) = \perp$  if  $C(s, x, in) == \perp$**

- ako je pre izvršavanja naredbe  $s$  vrednost promenljive  $x$  bila nedostizna, onda ce biti nedostizna i nakon njenog izvršavanja.

- **$C(x := c, x, out) = c$  if  $c$  is constant**

- ako je  $s$  naredba dodele i promenljivoj  $x$  dodeljuje konstantnu vrednost  $c$ , onda  $x$  ima istu tu vrednost i nakon izvršavanja te naredbe.

- **$C(x := f(...), x, out) = T$**

- ako je naredba  $s$  naredba dodele koja promenljivoj  $x$  dodeljuje vrednost koja nije konstantna, onda necemo znati nista o vrednosti promenljive  $x$  nakon izvršavanja naredbe  $s$ .

- **$C(y := ..., x, out) = C(y := ..., x, in)$  if  $x \neq y$**

- ako je  $s$  naredba dodele koja ne menja vrednost promenljive  $x$ , tada ce njena vrednost biti ista i pre i posle izvršavanja naredbe  $s$ .

Algoritam koji vrši constant propagation radi sledece:

1. Proverava da li postoji instrukcija za koju neko od prethodno navedenih pravila nije ispunjeno;

2. Ukoliko postoji takvo pravilo, zadovoljavamo ga.

## Implementacija

Kreiramo klasu koja ce biti omotac za llvm-ov tip `llvm::Instruction`, koja ima narednu formu:

```
class ConstantPropagationInstruction
{
private:
    Instruction *Instr;
    Value *Variable;
    std::pair<Status, int> StatusBefore;
    std::pair<Status, int> StatusAfter;
    std::vector<ConstantPropagationInstruction *> Predecessors;
public:
    ConstantPropagationInstruction(Instruction *);
    // getters, setters
};
```

**Value:** natklasa za sve bitnije tipove u llvm-u (Instruction, Function,...). Bazna klasa svih vrednosti sracunatih u programu koje mogu biti koriscene kao operandi za druge vrednosti. Svaka instanca klase Value ima svoj **Type**.

Cuvamo **StatusBefore** i **StatusAfter** kao par `{Status, int}`. Ovde nam Status predstavlja jednu od 3 opcije za promenljivu i predstavljen je na sledeci nacin:

```
enum class Status
{
    Top,
    Bottom,
    Const
};
```

Drugi element para je vrednost konkretne promenljive. U ovom slucaju to je celobrojna vrednost jer je optimizacija napisana tako da podrzava samo rad sa celim brojevima. U opstem slucaju, to moze biti bilo koji tip, kako primitivni tako i slozeni.

Za svaki BasicBlock u programu, a potom i za svaku instrukciju, neophodno je kreirati nasu instrukciju koja je prilagodjena za optimizaciju. Prolazom kroz funkciju vrsimo prikupljanje instrukcija, a takodje za svaku odredjujemo njene predecessore.

```
void IterateThroughFunction(Function &F)
{
    ConstantPropagationInstruction *Current, *PreviousCPI;
    Instruction *Previous;

    for (BasicBlock &BB : F) {
        for (Instruction &Instr : BB) {
```

```

        Current = new ConstantPropagationInstruction(&Instr);
        Instructions.push_back(Current);

        // Vraca pokazivac na prethodnu instrukciju koja nije debug ako
        postoji u istom BB.
        Previous = Instr.getPrevNonDebugInstruction();
        if (Previous == nullptr) {
            // Proveravamo da li je poslednja instrukcija iz nekog od
            prethodnih BB jednaka instrukciji u
            // nasem BB.
            for (BasicBlock* Predecessor : predecessors(&BB)) {
                PreviousCPI = *std::find_if(Instructions.begin(),
                Instructions.end(),
                [Predecessor]
                (ConstantPropagationInstruction *CPI) {
                    return CPI->GetInstruction() ==
                    &Predecessor->back();
                });
                Current->AddPredecessor(PreviousCPI);
            }
        } else {
            // Dodajemo pretposlednju, zato sto je tekuca instrukcija vec
            dodata u vektor na pocetku
            Current->AddPredecessor(Instructions[Instructions.size() - 2]);
        }
    }
}

```

```

Instruction * Instruction::getPrevNonDebugInstruction(bool SkipPseudoOp =
false) const

```

- Vraca pokazivac na prethodnu instrukciju koja nije debug, a nalazi se u istom basic block-u kao tekuca ili nullptr ako takva instrukcija ne postoji.

Pravila koja je neophodno proveriti, a zatim i primeniti ukoliko neka instrukcija pravilo ne zadovoljava imaju narednu formu u kodu:

```

bool CheckRuleNum(ConstantPropagationInstruction *CPI);
void ApplyRuleNum(ConstantPropagationInstruction *CPI);

```