Skripta za vezbe iz Konstrukcije kompilatora

Sadrzaj

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.

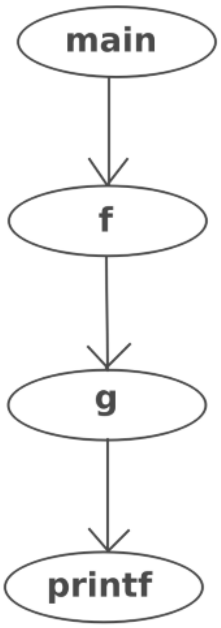
Primer. Razmotrimo naredni segment koda:

```
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:

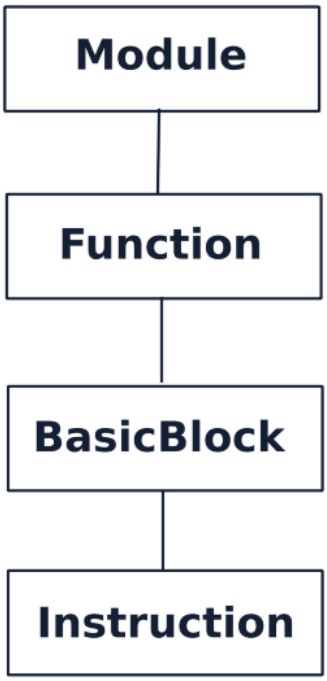
```
void CreateCallGraph(Module &M)
{
    ModuleName = M.getName().str();
    Function* Main = M.getFunction("main");

    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.

CFG - Control Flow Graph

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

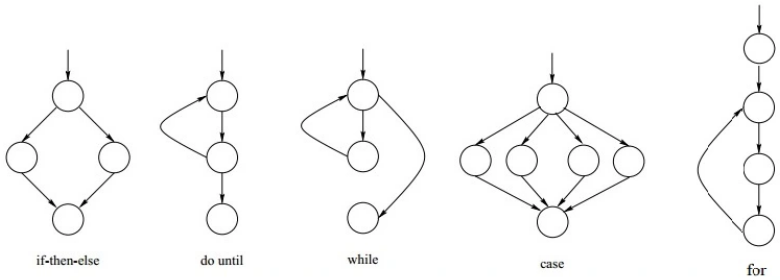
Neke karakteristike CFG-a:

- prikazuje sve putanje koje se mogu obici tokom izvrsavanja 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 sluzi 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 prilicno 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 izvrsavanja, 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 izvrsavanja, pa samim tim moramo da obradimo vise slucajeva:

  1. Klasican br koji nema nikakvog grananja - instrukcija se standardno upisuje u fajl.
  2. **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 specificnog nacina definisanja tipa `SwitchInst` u LLVM-u, treba obratiti paznju 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
...
};
```

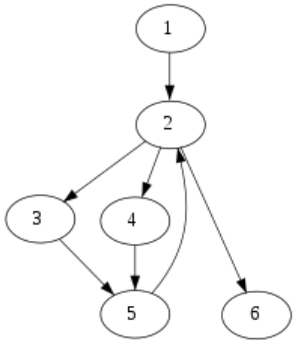
Nama su za .dot fajl neophodne iskljucivo vrednosti koje imamo u svakoj od case naredbi, pa iz tog razloga razmatramo samo **parne** indekse. Takodje, kako nulti i prvi indeks predstavljaju vrednost po kojoj se grana, kao i basic block podrazumevanog slucaja, njih preskacemo.

Dominatorska stabla

> Relaciju dominacije uvodimo na naredni nacin:

Cvor **d** u grafu kontrole toka **dominira** nad cvorom **n** ako svaka putanja od startnog cvora do **n** mora da sadrzi cvor **d**. Najcesce oznacavamo kao: d dom n. Prema definiciji svaki cvor dominira samim sobom.

Primer



Ako posmatramo dati graf mozemo zakljuciti sledece:

- cvor numerisan brojem jedan dominira svim ostalim cvorovima - startni je cvor pa putanja od njega do svih ostalih mora da ga sadrzi;
- cvor numerisan brojem dva dominira samim sobom, kao i cvorovima numerisanim brojevima 3, 4, 5 i 6;
- cvorovi numerisani brojevima 3, 4, 5 i 6 dominiraju jedino samim sobom.

Postoje razliciti koncepti koji imaju veze sa relacijom dominacije:

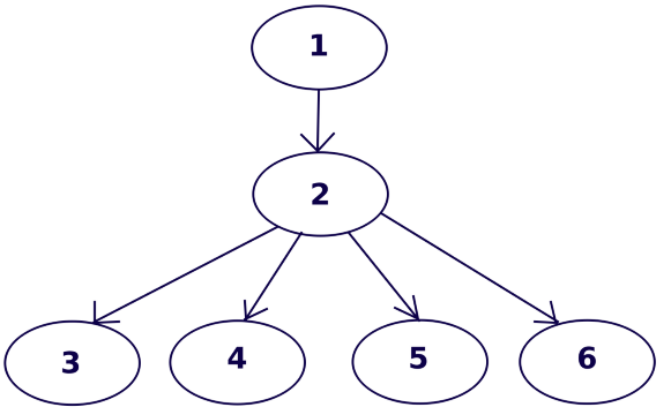
> **Immediate dominator**

Onaj dominator koji je najblizi od svih dominatora. Svaki cvor koji nije startni, ima svog immediate dominatora koji je jedinstven. Oni su nam znacajni prilikom odredjivanja **dominatorskog stabla**.

> **Dominatorsko stablo**

Dominatorsko stablo formiraju ivice oblika: $\{(idom(w), w) | w \in V - \{S\}\}$ gde je S koren r stabla.

Za stablo kazemo da je dominatorsko ukoliko su deca svakog cvora ona kojima on dominira po relaciji immediate dominacije. Za primer naveden iznad, odgovarajuce dominatorsko stablo bi bilo sledece:



Postupak nalazenja dominatorskog stabla (Lengauer-Tarjan algoritam)

Da bismo sproveli algoritam neophodno je da se uvede pojam semidominatora.

> **Semidominator**

$sdom(w) = min\ \{v|$ postoji putanja $v = v_0 \dots v_k = w$ tako da je $v_i > w$ za $i = 1, \dots, k-1\}$

pri cemu relacija $>$ oznacava da cvor ima vecu ulaznu numeraciju.

Semidominator je minimalni cvor w koji se nalazi u uniji naredne dve grupe:

- svi cvorovi v takvi da je (v, w) usmerena grana u grafu i vazi da je ulazna numeracija cvora v manja od ulazne numeracije cvra w;
- semidominatori cvora u pri cemu je (v, w) grana u grafu takva da je u predak cvora v. Ovo je znacajno jer pri uslovu $u > w$ i u je predak cvora v takvog da je (v, w) grana, ako mozemo doci do u bilo kojom putanjom, onda putanja od u do w ne ukljucuje nijedan cvor za koji vazi da je $< w$.

Koraci:

- Sprovesti DFS pretragu grafa dobijenog na ulazu i svakom cvoru dodeliti **ulaznu numeraciju** (marker kojim odredjujemo kada je koji cvor bio posecen). Inicijalizovati sve neophodne strukture:
  - lista povezanosti
  - obrnuta lista povezanosti
  - lista roditeljskih cvorova
  - lista predaka
  - vektor koji pamti sdom (za svaki cvor se postavlja da je semidominator samom sebi)
  - vektor koji pamti idom
  - lista koja cuva redosled obilaska cvorova
- Za svaki cvor odrediti njegovog semidominatora. Semidominatore traziti u redosledu obrnutom od onog u kom su cvorovi inicijalno bili poseceni pri DFS-u.
- Racunamo immediate dominatore za svaki cvor na osnovu prethodno sracunatih semidominatora:
  - ako postoji direktna grana izmedju u i $sdom(u)$ tada imamo da je: $sdom(u) = idom(u)$;
  - ako direktna grana ne postoji: $idom(u) = idom(idom(anc))$.

Constant propagation

Jedna od standardnih optimizacija koje kompajler sprovodi. Podrazumeva zamenu vrednosti promenjivih 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 sto nakon ulaska u if granu dodeljujemo promenljivoj x vrednost 7, a zatim koristimo x u daljim izracunavanjima, mozemo direktno izvrsiti sledecu zamenu: z = 7 + 3 (to ce 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 izvrsavanja:

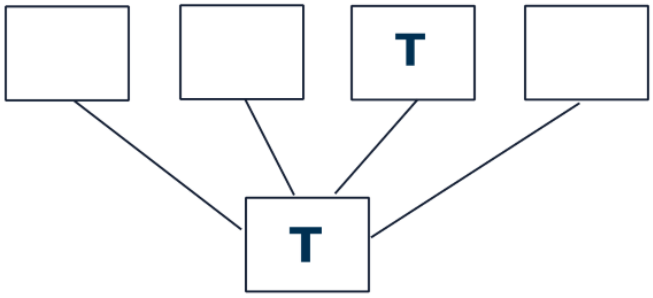
- **C(s, x, in)** - stanje promenljive x pre izvrsavanja naredbe s;
- **C(s, x, out)** - stanje promenljive x nakon izvrsavanja naredbe s.

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

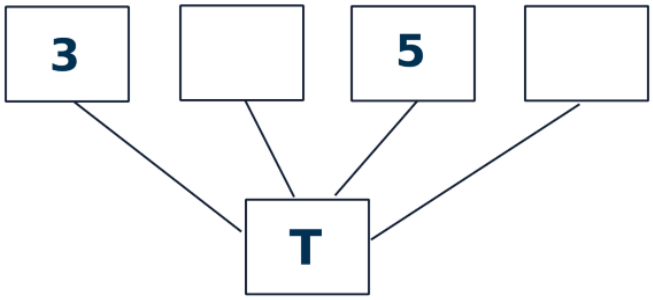
Sprovodjenje optimizacije

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

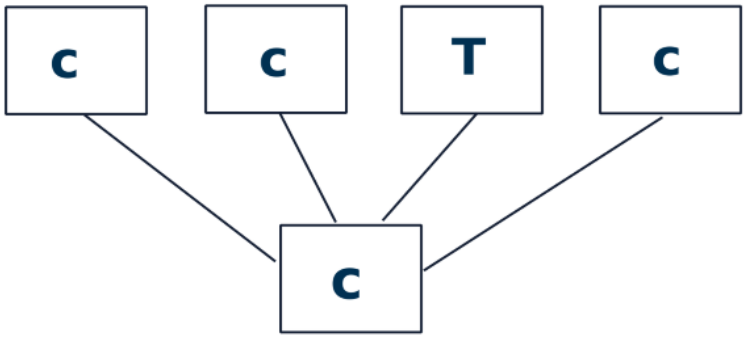
- **if C(pi, x, out) == T for any i then C(s, x, in) = T**
    - ako je vrednost promenljive nakon bilo kod predecessora tekuceg basic blocka nepoznata, onda nece biti poznata ni pre izvrsavanja tekuce naredbe.



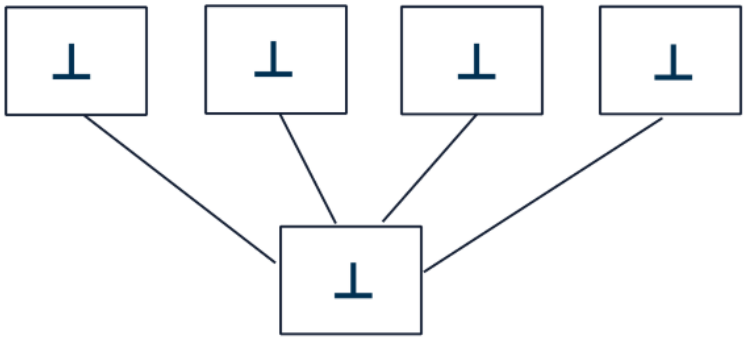
- **if C(pi, x, out) == c && c(pj, x, out) == d && c != d then C(s, x, in) == T**
    - ako je vrednost promenljive x u razlicitim predecessorima razlicita, onda ona nece biti poznata pre izvrsavanja naredbe s.



- **if C(pi, 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(pi, x, out) == ⊥ for all i then C(s, x, in) = ⊥**
    - ako je svaki predecessor nedostizan, ona ce i u naredbi s vrednost promenljive x biti nedostizan.



- **C(s, x, out) = ⊥ if C(s, x, in) == ⊥**
    - ako je pre izvrsavanja naredbe s vrednost promenljive x bila nedostizna, onda ce biti nedostizna i nakon njenog izvrsavanja.
- **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 izvrsavanja 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 izvrsavanja naredbe s.
- **C(y := ..., x, out) = C(y := ..., x, in) if x != y**
    - ako je s naredba dodele koja ne menja vrednost promenljive x, tada ce njena vrednost biti ista i pre i posle izvrsavanja naredbe s.

Algoritam koji vrsi 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

Propagacija za jednu promenljivu

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);
```

Constant folding

Constant folding je optimizacija koja je dosta povezana sa propagacijom konstanti i njih dve se najcesce upotrebljavaju jedna za drugom. Ona podrazumeva pronalaznje i racunanje konstantnih izraza u fazi kompilacije.

Primer

```
int main ()
{
    int x = 10;
    int y = 10 + x;
    printf("%d\n", y);

    return 0;
}
```

Nakon izvrsavanje Constant propagation optimizacije dobili bismo naredni segment koda:

```
int main ()
{
```

```
    int x = 10;
    int y = 10 + 10;
    printf("%d\n", y);

    return 0;
}
```

Zadatak Constant foldinga jeste da izraz koji se dodeljuje promenljivoj y odmah sracuna, odnosno:

```
int main ()
{
    int x = 10;
    int y = 20;
    printf("%d\n", y);
    return 0;
}
```

U prethodnom primeru mozemo primetiti da dobijamo *dead code* nakon izvrsavanja ovih optimizacija (promenljiva x nece nigde biti koriscena). Naprednija forma propagacije konstanti, koja je poznata kao **sparse conditional constant folding** bi mogla paralelno sa propagacijom da izvrsi i eliminaciju mrtvog koda.

Implementacija constant foldinga se moze sprovesti dosta jednostavno: prolazimo kroz instrukcije u nasem IR-u i vrsimo provere da li su operandi koji ucestvuju u operacijama konstante. Ukoliko jesu, menjamo ih rezultatom koji konkretna operacija daje. Neophodno je posebno izvrsiti postupak za binarne operatore, branch instrukciju, kao i operatore poredjenja.

Dead-code elimination

Dead-code elimination (dead-code removal, dead-code strip) je kompajlerska optimizacija koja za cilj ima uklanjanje onih segmenata programskog koda koji nece imati nikakav uticaj na rezultat rada programa. Kao dead-code mozemo posmatrati:

- segmente koda koji nikad nece biti izvrseni - **unreachable code**;

  ```
  int foo()
  {
      int a = 42;
      return a * 7;
      printf("Hello!\n"); // unreachable code
  }
  ```

- promenljive koje su deklarisane, ali nikad u kodu nisu koriscene.

  ```
  int foo()
  {
  ```

```
    int a = 42;
    int b = 7;  // promenljva koju nigde ne koristimo
    return a * 7;
}
```

Neki od benefita ove optimizacije:

- smanjuje velicinu programa sto u nekim kontekstima moze biti od znacaja;
- daje mogucnost programu da izbegne izvrsavanje nebitnih operacija sto doprinosi njegovim performasama;
- moze modifikovati IR tako da bude pogodan za primenu nekih drugih optimizacija.

Primer

IR pre

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@.str.1 = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32 noundef %0, ptr noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca ptr, align 8
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  store i32 0, ptr %3, align 4
  store i32 %0, ptr %4, align 4
  store ptr %1, ptr %5, align 8
  store i32 5, ptr %6, align 4
  store i32 10, ptr %7, align 4
  %8 = load i32, ptr %7, align 4
  %9 = add nsw i32 %8, 6
  %10 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %9)
  %11 = call i32 (ptr, ...) @printf(ptr noundef @.str.1)
  ret i32 0
}

declare i32 @printf(ptr noundef, ...) #1
```

i nakon izvrsavanja DCE:

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@.str.1 = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32 noundef %0, ptr noundef %1) #0 {
  %3 = alloca i32, align 4
  store i32 10, ptr %3, align 4
  %4 = load i32, ptr %3, align 4
  %5 = add nsw i32 %4, 6
  %6 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %5)
  %7 = call i32 (ptr, ...) @printf(ptr noundef @.str.1)
  ret i32 0
}

declare i32 @printf(ptr noundef, ...) #1
```

Optimizacije petlji

Uvodna terminologija vezana za petlje

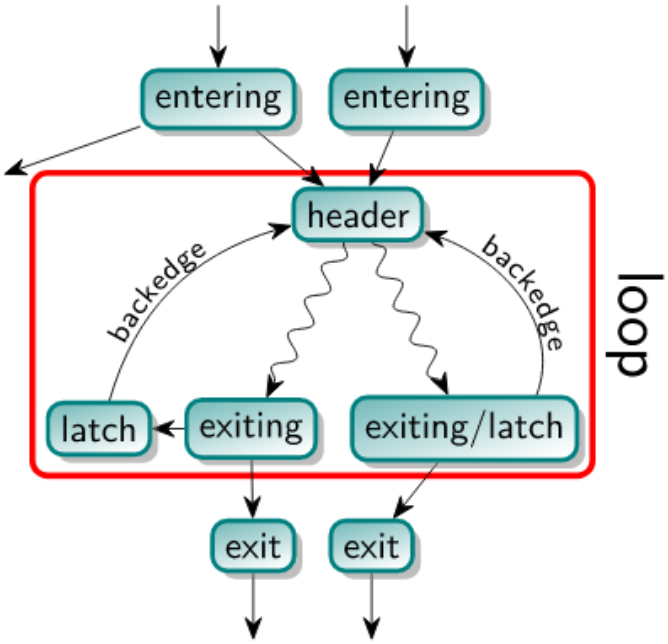
U LLVM-u, detekcija petlji u grafu kontrole toka je realizovana od strane klase `LoopInfo`. Ona je zasnovana na sledecoj definiciji: **petlja** je podskup cvorova grafa kontrole toka sa narednim karakteristikama:

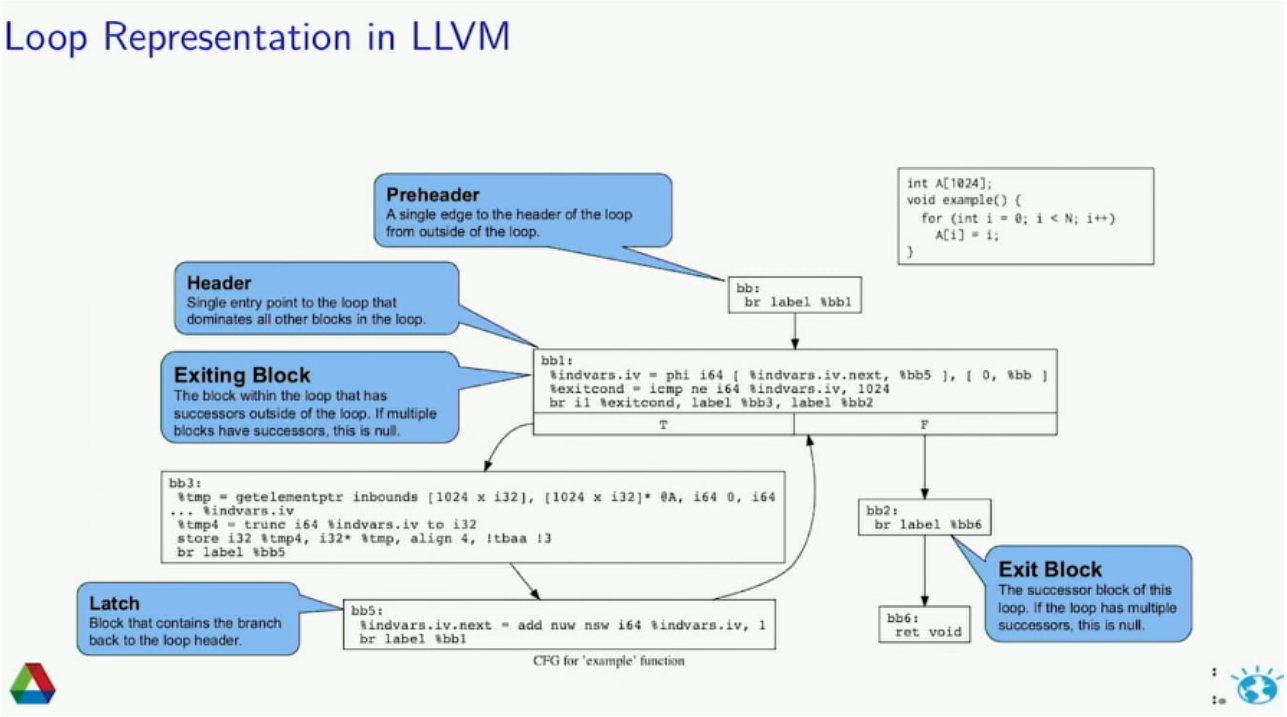
- Podgraf koji sadrzi sve grane CFG-a unutar petlje je jako povezan (svaki cvor je dostizan iz svih ostalih);

- Sve grane van tog podskupa koje idu ka cvorovima tog podskupa pokazuju na isti cvor koji se naziva **header**. Ovaj cvor dominira svim cvorovima unutar petlje;
- Petlja je maksimalan podskup sa ovim karakteristikama.

Dodatna terminologija:

- **entering block (loop predecessor)**: cvor koji nije deo petlje, ali ima granu koja je usmerena ka header cvoru petlje. Ako postoji samo jedan ulazni blok i njegova jedina grana ide ka header-u, onda se on naziva **preheader**. On dominira petljom iako sam nije njen sastavni deo;





Loop unrolling

Loop unrolling je jedna od standardnih i cesto zastupljenih optimizacija u okviru kompilatora koja za zadatak ima da eliminise petlju (do one mere u kojoj je to moguce) i kod iz njenog tela ponovi veci broj puta. Pass koji ovo radi spada u kategoriju *transform* passova zato sto je neophodno izvrsiti promenu IR-a.

Primer Umesto narednog koda:

```
int main ()
{
    for (int i = 0; i < 5; ++i)
        printf("Hello!\n");
    return 0;
}
```

ovom optimizacijom dobijamo kod:

```
int main ()
{
    print("Hello!\n");
    print("Hello!\n");
    print("Hello!\n");
    print("Hello!\n");
    print("Hello!\n");

    return 0;
}
```

Izdvajamo dva tipa loop unrolling-a:

1. **Full unrolling** - razmotava petlju u potpunosti i naredbe iz njenog tela ponavlja onoliko puta kolika je bila gornja granica iteracije. Ovo je moguce iskljucivo ukoliko je granica u petlji imala konstantnu vrednost.
2. **Partial unroll** - ukoliko granica nije konstantna vrednost, moze se delimicno izvrsiti odmotavanje petlje. U ovom slucaju neophodno je navesti **faktor odmotavanja (unrolling factor)** koji podrazumeva broj ponavljanja.

Prednosti:

1. Povecavamo efikasnost programa;
2. Smanjujemo loop overhead;
3. Ako naredbe u telu petlje nisu zavisne jedna od druge mogu se izvrsavati paralelno.

Mane:

1. Povecava se velicina programskog koda sto moze biti nesto sto ne zelimo;
2. Odmotane petlje koje sadrze skokove su potencijalno sporije nego rekurzivne funkcije;
3. Moguca je povecana iskoriscenost registara prilikom jedne iteracije zbog cuvanja privremenih promenljivih sto moze smanjiti performanse.

Implementacija

Od stvari koje su nam neophodne da bi optimizacija bila izvrsena cuvamo sledece:

- vektor BasicBlock-ova koji pripadaju konkretnoj petlji;

- mapu kojom pamtimo sve promenljive od znacaja;
- indikator da li je granica petlje konstantna;
- vrednost granice;
- brojac petlje - kao **Value ***

Za implementaciju neophodno nam je da imamo funkciju koja ce telo petlje (prethodno izdvojeno), ponavljati veci broj puta. Kada se svi BB uklone iz petlje i ostane samo ono sto je neophodno ponavljati, imamo naredni IR:

```
@.str = private unnamed_addr constant [14 x i8] c"Hello world!\0A\00",
align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32 noundef %0, ptr noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca ptr, align 8
  %6 = alloca i32, align 4
  store i32 0, ptr %3, align 4
  store i32 %0, ptr %4, align 4
  store ptr %1, ptr %5, align 8
  store i32 0, ptr %6, align 4
  br label %7

7:                                                ; preds = %2
  %8 = call i32 (ptr, ...) @printf(ptr noundef @.str)
  br label %9

9:                                                ; preds = %7
  ret i32 0
}
```

Jedino sto nam je ostalo je ono sto se nalazi u telu petlje i sto je neophodno ponoviti onoliki broj puta kolika je bila gornja granica iteracije petlje.

Loop inversion

Loop inversion je optimizacija petlje koja tranformise strukturu **while** petlje tako sto je menja **if** naredbom koja u svom telu sadrzi **do-while** petlju. Ako se koristi na adekvatan nacin moze da dovede do poboljsanja performansi.

Primer Neka je dat sledeci segment koda:

```
int i, a[100];
i = 0;
while (i < 100) {
    a[i] = 0;
    i++;
}
```

Ukoliko primenimo datu optimizaciju dobijamo sledece:

```
int i, a[100];
i = 0;
if (i < 100) {
    do {
      a[i] = 0;
      i++;
    } while (i < 100);
}
```

Neke od standardnih operacija

Mapiranje promenljivih

Prolaskom kroz IR zelimo da vezemo instrukciju `load` za promenljivu koju koristi. Zbog SSA forme, desava se da originalnu promenljivu koristimo na drugim mestima preko novouvedenih promenljivih. Npr. ako posmatramo naredni segment koda:

```
7:                                                ; preds = %12, %2
  %8 = load i32, ptr %6, align 4
  %9 = icmp slt i32 %8, 10
  br i1 %9, label %10, label %15
```

vidimo da promenljiva %8 zapravo samo cuva vrednost koja se nalazila u promenljivoj %6. Da bismo kasnije izmene IR-a odradili kako treba, nama je bitno da znamo koja je originalna promenljiva. To pamcenje realizujemo na sledeci nacin:

```
void MapVariables(Function &F)
{
    for (BasicBlock &BB : F) {
        for (Instruction &Instr : BB) {
            if (isa<LoadInst>(&Instr))
                VariablesMap[&Instr] = Instr.getOperand(0);
        }
    }
}
```

pri cemu nam je `VariablesMap` mapa definisana na sledeci nacin:

```
std::unordered_map<Value *, Value *> VariablesMap = {};
```

Primer Kada budemo trazili brojac petlje, zbog ovog mapiranja mi cemo dobiti da je to promenljiva %6, a ne %8 koja samo privremeno cuva njenu vrednost.

Trazenje brojaca i gornje granice petlje

Za optimizacije petlji cesto nam je neophodno da znamo koliko puta bi se maksimalno petlja izvrsavala, kao i koja promenljiva je njen brojac. Ako posmatramo naredni segment IR-a:

```
7:                                                ; preds = %12, %2
  %8 = load i32, ptr %6, align 4
  %9 = icmp slt i32 %8, 10
  br i1 %9, label %10, label %15

10:                                               ; preds = %7
  %11 = call i32 (ptr, ...) @printf(ptr noundef @.str)
  br label %12

12:                                               ; preds = %10
  %13 = load i32, ptr %6, align 4
  %14 = add nsw i32 %13, 1
  store i32 %14, ptr %6, align 4
  br label %7, !llvm.loop !6
```

koji predstavlja BB koji ulaze u sastav petlje, mozemo primetiti da informacije o brojacu, kao i o gornjoj granici petlje mozemo dobiti iz nultog bloka i to iz instrukcije koja radi poredjenje:

```
  %9 = icmp slt i32 %8, 10
```

Do tih informacija dolazimo na sledeci nacin:

```
std::vector<BasicBlock *> LoopBasicBlocks = {}; // u nekom momentu
inicijalizovan
// ...
Value *LoopCounter = nullptr;
int LoopBound = -1;
bool IsLoopBoundConstant = false;

void FindLoopBoundAndCounter()
{
    for (Instruction &Instr : *LoopBasicBlocks.front()) {
        if (isa<ICmpInstruction>(&Instr)) {
            // ne zelimo %8 vec ono u sta je on mapiran
            LoopCounter = VariablesMap[Instr.getOperand(0)];
            if (ConstantInt *ConstInt = dyn_cast<ConstantInt>
(Instr.getOperand(1))) {
                IsLoopBoundConstant = true;
                LoopBound = ConstInt->getSExtValue();
            }
        }
```

```
    }
}
```

Kopiranje instrukcija iz jednog BB u drugi

Nekad nam je od znacaja da mozemo instrukcije iz jednog BasicBlock-a da prekopiramo u drugi. Za svaku instrukciju iz BB iz kog vrsimo kopiranje neophodno je napraviti odgovarajucu kopiju i ubaciti je u drugi BB.

```
void CopyInstructions(BasicBlock *From, BasicBlock *To)
{
    Instruction *InstrCopy = nullptr;
    std::unordered_map<Value *, Value *> CopiesMapping = {};

    for (Instruction &Instr : *From) {
        InstrCopy = Instr.clone();
        InstrCopy.insertBefore(To->getTerminator());
        CopiesMapping[&Instr] = InstrCopy;

        // Dodatno je neophodno lepo mapirati operande
        for (size_t i = 0; i < Instr.getNumOperands(); ++i) {
            if (CopiesMapping.find(Instr.getOperand(i)) !=
CopiesMapping.end()) {
                InstrCopy->setOperand(i, Instr.getOperand(i));
            }
        }
    }
}
```