Konstrukcija kompilatora

— Optimizacije —

Milena Vujošević Janičić

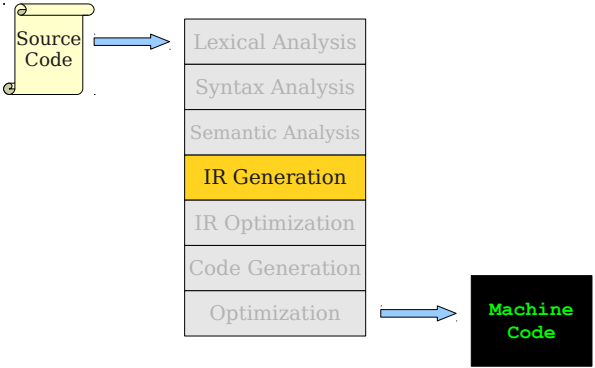
www.matf.bg.ac.rs/~milena

Matematički fakultet, Univerzitet u Beogradu

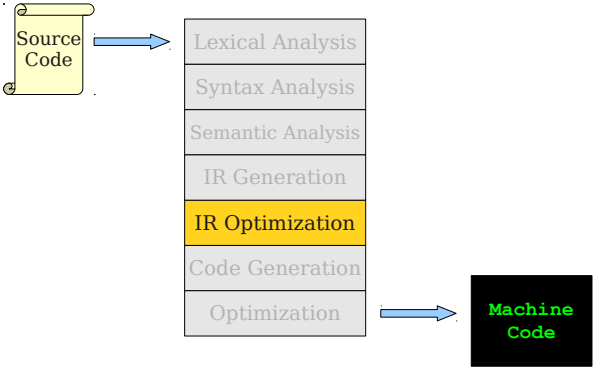
Pregled

Pregled

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

IR Optimizacija

- Cilj optimizacije je da se poboljša kôd generisan u prethodnom koraku
- To je najvažniji i najkompleksniji deo modernog kompajlera
- Veoma aktivna oblast istraživanja
- Razlozi za optimizaciju
  - Prilikom generisanja koda uvodi se redudantnost (jednostavno prevođenje konstrukata višeg nivoa u IR često uvodi dodatna izračunavanja koja mogu da se ubrzaju, podele ili eliminišu)
  - Programeri često ne razmišljaju o efikasnosti koda, na primer, često se kôd koji se izvršava u petlji može *izdići* da se izvršava pre petlje

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

b2 = _t0 == _t1;

b3 = _t0 < _t1;
```

```
while (x < y + z) {
    x = x - y;
}
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Terminologija

- Termin optimizacija označava traženje optimalnog (idealnog) koda za zadati program
- Ovaj problem je, u opštem slučaju, neodlučiv
- Zapravo, cilj je poboljšanje međureprezentacije, a ne njegova optimizacija
- Karakteristike dobrog optimizatora:
  - Ne sme da suštinski promeni ponašanje programa
  - Treba da proizvede što efikasniji kod
  - Ne treba da koristi puno vremena (kao deo kompilacije, ne sme dugo da traje)
- Nažalost:
  - I dobri optimizatori nekada uvedu grešku u kod.
  - Optimizatori nekada promaše da urade neku jednostavnu optimizaciju zbog ograničenja algoritama koje koriste.
  - Većina interesantnih optimizacija je NP-teško ili neodlučivo!

Šta optimizujemo

- Optimizatori mogu da pokušaju da poboljšaju kod u skladu sa različitim željenim osobinama.
- Šta je to što možemo da želimo da optimizujemo?

Šta optimizujemo

- Optimizatori mogu da pokušaju da poboljšaju kod u skladu sa različitim željenim osobinama.
- Šta je to što možemo da želimo da optimizujemo?
- **Vreme izvršavanja** (napraviti program da bude što brži, na račun memorije i energije)
- **Upotreba memorije** (napraviti što je manji program, na račun vremena izvršavanja i energije)
- **Upotreba energije** (napraviti program da troši što manje energije, biranjem jednostavnih instrukcija, na račun brzine izvršavanja i memorije)
- Postoje i razni drugi mogući zahtevi — minimizovati pozive funkcija, redukovati korišćenje jedinice za rad u pokretnom zarezu...

Optimizacija koda vs optimizacija međukoda

- Nije uvek skroz jasna razlika između optimizacije međukoda i optimizacije izvršnog koda
- Obično optimizacija međukoda sprovodi pojednostavljivanja koja su validna za sve arhitekture dok optimizacija koda pokušava da unapredi performanse na osnovu karakteristika ciljne arhitekture računara
- Neke optimizacije su negde između, na primer zamena x*0.5 sa x/2

Pregled

Saglasna analiza programa

- Da bi se optimizovao program, kompajler mora da bude u stanju da rasuđuje o osobinama tog programa
- Analiza je saglasna (engl. *sound*) ako nikada ne proizvodi netačne činjenice o programu
- Sve analize koje ćemo diskutovati su saglasne

Soundness

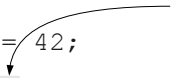
```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

Soundness

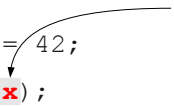
```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```



"At this point in the program, **x** holds some integer value."

Soundness

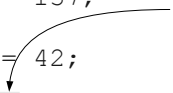
```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```



"At this point in the program, **x** is either 137 or 42"

Soundness

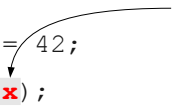
```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```



"At this point in the program, **x** is 137"

Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```



"At this point in the program, x is either 137, 42, or 271"

Optimizacija koja čuva semantiku

- Optimizacija čuva semantiku ako ne menja semantiku originalnog programa
- Na primer, naredne optimizacije čuvaju semantiku programa
  - Izračunavanje vrednosti koje se mogu znati statički u vreme kompilacije umesto u vreme izvršavanja
  - Evaluacija konstantnih izraza van petlje umesto u petlji
  - Eliminisanje nepotrebnih privremenih promenljivih
- Primer optimizacije koja ne čuva semantiku programa je zamena *bubblesort* sortiranja sa *quicksort* sortiranjem
- Optimizacije koje ćemo razmatrati sve čuvaju semantiku programa

Formalizam za IR optimizaciju

- Svaka faza kompilacije koristi neku vrstu apstrakcije
- Skeniranje koristi regularne izraze
- Parsiranje koristi kontekstno slobodne gramatike
- Semantička analiza koristi tabele simbola i sistem tipova
- Generisanje međukoda koristi apstraktno sintaksno stablo
- I za optimizaciju je potreban formalizam koji ima mogućnost da uhvati strukturu programa na način koji omogućava optimizaciju

Visualizing IR

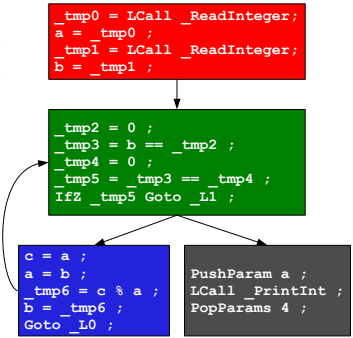
```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

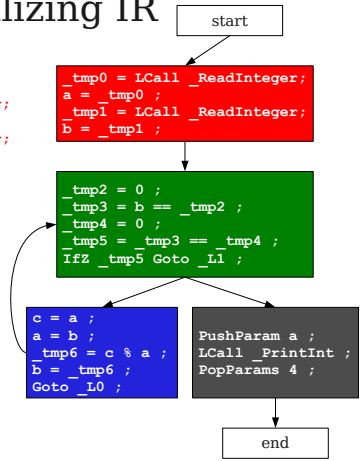
Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



Osnovni blok

- Osnovni blok je niz IR instrukcija gde
  - Postoji samo jedna tačka ulaska u blok, tj mesto na kojem počinje izvršavanje instrukcija bloka. Ulazak u blok mora biti na vrhu samog bloka.
  - Postoji samo jedna tačka izlaska iz bloka, tj mesto na kojem se završava izvršavanje instrukcija bloka. Izlazak iz bloka mora biti na kraju samog bloka.
- Neformalno, niz instrukcija u bloku se uvek izvršava u celini

Graf kontrole toka (engl. *Control Flow Graph*)

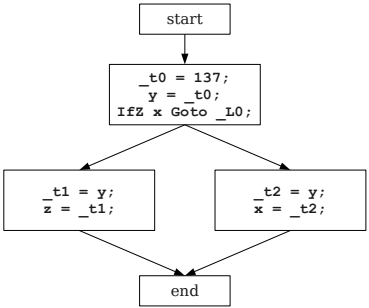
- Graf kontrole toka (engl. *Control Flow Graph*, CFG) je graf koji sadrži osnovne blokove funkcije
- Svaka ivica iz jednog osnovnog bloka do drugog označava da kontrola izvršavanja može da ide od kraja prvog do početka drugog bloka
- Postoje i dva čvora koji označavaju početak i kraj funkcije

Vrste optimizacija

- Postoje tri vrste optimizacija: lokalna, globalna i međuproceduralna
- Lokalna optimizacija radi u okviru jednog osnovnog bloka
- Globalana optimizacija radi na grafu kontrole toka funkcije
- Međuproceduralna optimizacija radi na celokupnom grafu kontrole toka koji prati pozive funkcija i njihovu međusobnu interakciju

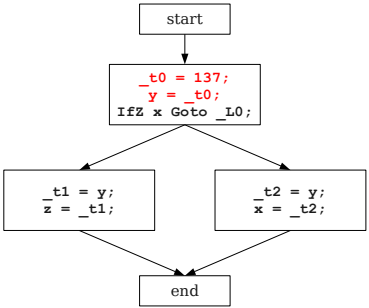
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



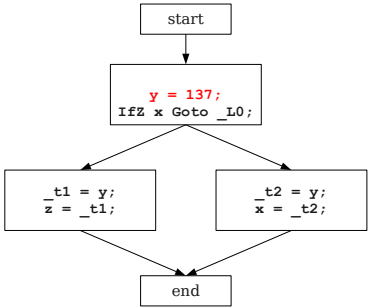
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



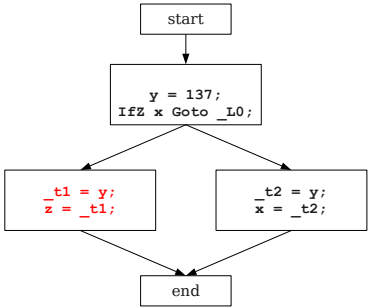
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



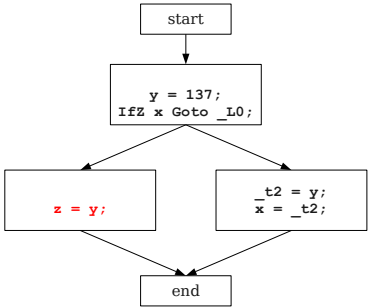
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



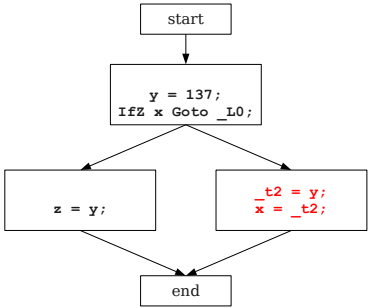
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



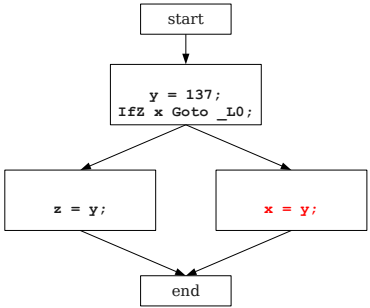
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



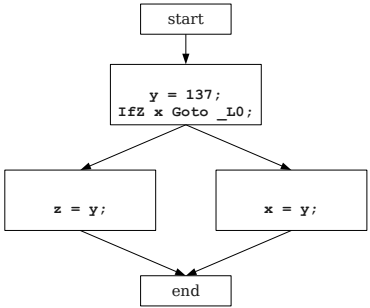
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



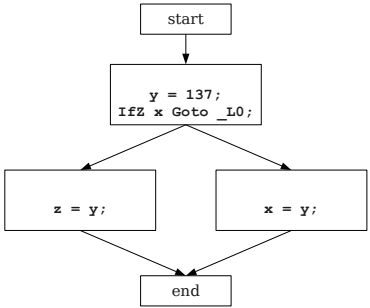
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



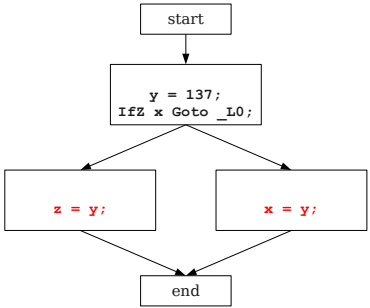
Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



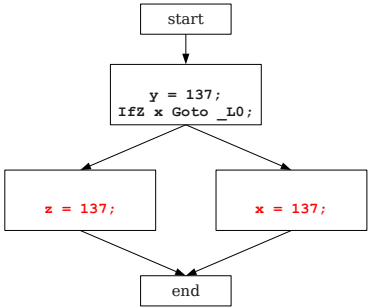
Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



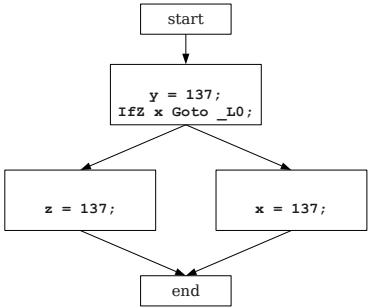
Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



Pregled

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Eliminacija zajedničkih podizraza

- Ako imamo dve dodele promenljivama

```
v1 = a op b
...
v2 = a op b
```

tako da se vrednosti promenljivih v1, a i b ne menjaju između ove dve dodele, onda možemo da prepišemo kod na sledeći način

```
v1 = a op b
...
v2 = v1
```

- Na ovaj način se eliminiše bespotrebno izračunavanje i pravi se prostor za kasnije optimizacije

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Prenos kopiranja

- Ako imamo dodelu

  `v1 = v2`

  onda sve dok v1 i v2 nemaju ponovo dodeljene neke nove vrednosti, možemo da zapišemo izraze oblika

  `a = ... v1 ...`

  kao

  `a = ... v2 ...`

  pod uslovom da je takvo prezapisivanje u redu (u smislu da postoji takva instrukcije, npr ako je nešto konstanta, nekada imamo dostpunu instrukciju a nekada nemamo)
- Ova optimizacija stvara prostor za naredne optimizacije koje ćemo uskoro videti

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;

_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;

_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;


_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;


_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;


_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Eliminacija mrtvog koda

- Dodela promenljivoj v se naziva mrtva dodela ako se vrednost te promenljive nigde kasnije ne koristi
- Eliminacija mrtvog koda uklanja mrtve dodele iz međureprezentacije
- Utvrđivanje da li je dodela mrtva zavisi od toga kojim promenljivama se dodeljuju vrednosti i gde se ta dodela vrši

For Comparison

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
_tmp4 = _tmp0 + b ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Primena lokalnih optimizacija

- Različite optimizacije koje smo do sada videli sve zapravo vode računa o samo malom delu optimizacije
- Eliminacija zajedničkih podizraza eliminiše nepotrebne naredbe
- Prenos kopiranja pomaže u pronalaženju mrtvog koda
- Eliminacija mrtvog koda uklanja naredbe koje više nisu potrebne
- Da bi se izvukao maksimum, može da bude potrebno da se ove optimizacije izvrše više puta

Applying Local Optimizations

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Applying Local Optimizations

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Applying Local Optimizations

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Common Subexpression Elimination

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Common Subexpression Elimination

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Copy Propagation

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Copy Propagation

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Common Subexpression Elimination (Again)

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = d;
```

Common Subexpression Elimination (Again)

Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = d;
```

Drugi tipovi lokalnih optimizacija

- Aritmetička pojednostavljenja (engl. *arithmetic simplification*): zameni teške operacije sa lakšim. Na primer, prepiši `x = 4 * a;` u `x = a << 2;`
- Slaganje konstanti (engl. *constant folding*): evaluiraj izraze u fazi kompilacije ako imaju konstantne vrednosti Na primer, prepiši `x = 4 * 5;` u `x = 20;`

Optimizacija i analiza

- Mnoge optimizacije su moguće samo ako se obezbedi odgovarajuća analiza ponašanja programa
- Da bi optimizacija mogla da se implementira, pričaćemo o odgovarajućim analizama koje je potrebno sprovesti

Analiza dostupnih izraza

- I eliminacija zajedničkih podizraza i prenos kopiranja zavise od analize **dostupnih izraza** u programu
- Izraz se naziva **dostupan** ako neka promenljiva u programu čuva vrednost tog izraza
- U okviru eliminacije zajedničkih izraza, zamenjujemo dostupni izraz sa promenljivom koja nosi tu vrednost
- U prenosu kopiranja, menjamo korišćenje promenljive sa dostupnim izrazom koji ona koristi

Pronalaženje dostupnih izraza

- Inicijalno, ni jedan izraz nije dostupan
- Kada god se izvrši izraz a = b + c
  - Svaki izraz koji sadrži promenljivu a više nije validan
  - Izraz a = b + c postaje dostupan
- Ideja: iteriraj kroz instrukcije osnovnog bloka počevši sa praznim skupom izraza i ažuriraj dostupne izraze nakon svake instrukcije

Available Expressions

```
a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
   { }
  a = b;
{ a = b }
  c = b;

d = a + b;

e = a + b;

  d = b;

f = a + b;
```

Available Expressions

```
        { }
      a = b;
  { a = b }
      c = b;
{ a = b, c = b }
    d = a + b;

    e = a + b;

      d = b;

    f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = e;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
a = b;
c = a;
d = a + b;
e = d;
d = a;
f = e;
```

Analiza živosti (engl. *liveness analysis*)

- Analiza koja odgovara eliminaciji mrtvog koda se naziva analiza živosti
- Promenljiva je živa u nekoj tački programa ako se kasnije u programu njena vrednost čita pre nego što se u nju nešto novo upiše
- Eliminacija mrtvog koda radi tako što sračunava živost svake promenljive i onda eliminiše dodele mrtvim promenljivama

Izračunavanje živih promenljivih

- Da bi znali koja promenljiva će biti korišćena u nekoj tački programa, iteriramo kroz sve naredbe osnovnog bloka u obrnutom redosledu
- Inicijalno, neki mali skup vrednosti se zna da će biti živ: koji tačno, to zavisi od konkretnog programa
- Kada vidimo naredbu oblika `a = b + c`
  - Neposredno pre te naredbe, a nije živa, jer će njena vrednost da bude prepisana
  - Neposredno pre te naredbe, i `b` i `c` su žive, jer će obe biti korišćene
- Primetimo da nam međukod često ne dozvoljava naredbu oblika `a = a + b`. U slučaju da je takva naredba dozvoljena, onda je živost *jača* odnosno, najpre se vrši uklanjanje promenljivih iz skupa, pa onda dodavanje, tako da će a nakon ove analize ipak biti živa promenljiva.

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

{ b, d }

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;
```
{ b, d, e }
```
f = e;
```
{ b, d }

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;
```
{ a, b, e }
```
d = a;
```
{ b, d, e }
```
f = e;
```
{ b, d }

Liveness Analysis

```
a = b;

c = a;

d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;
{ a, b }

{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```

Dead Code Elimination

```
a = b;
```

```
d = a + b;
e = d;
d = a;
```

Liveness Analysis II

```
a = b;
```

```
d = a + b;
e = d;
d = a;
```

Liveness Analysis II

```
a = b;


d = a + b;

e = d;

d = a;
```
{ b, d }

Liveness Analysis II

```
a = b;
```

```
d = a + b;
```

```
e = d;
{ a, b }
d = a;
{ b, d }
```

Liveness Analysis II

```
a = b;
```

```
d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

Liveness Analysis II

```
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

Liveness Analysis II

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

Dead Code Elimination

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }

{ a, b }
d = a;
{ b, d }
```

Dead Code Elimination

```
a = b;
```

```
d = a + b;
```

```
d = a;
```

Liveness Analysis III

```
a = b;
```

```
d = a + b;
```

```
d = a;
```

Liveness Analysis III

```
a = b;
```

```
d = a + b;
```

```
d = a;
{b, d}
```

Liveness Analysis III

```
a = b;
```

```
d = a + b;
```

{a, b}

```
d = a;
```

{b, d}

Liveness Analysis III

```
a = b;
```

{a, b}

```
d = a + b;
```

{a, b}

```
d = a;
```
{b, d}

Liveness Analysis III

```
        {b}
a = b;

        {a, b}

d = a + b;

        {a, b}

d = a;
        {b, d}
```

Dead Code Elimination

```
{b}
a = b;

{a, b}

d = a + b;

{a, b}

d = a;
{b, d}
```

Dead Code Elimination

```
{b}
a = b;

{a, b}

d = a + b;

{a, b}

d = a;
{b, d}
```

Dead Code Elimination

```
      {b}
a = b;

{a, b}


{a, b}

d = a;
{b, d}
```

Dead Code Elimination

```
a = b;
```

```
d = a;
```

A Combined Algorithm

A Combined Algorithm

```
a = b;
c = a;
d = a + b;
e = d;
d = a;
f = e;
```

A Combined Algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

{b, d}

A Combined Algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
{b, d}
```

A Combined Algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;
```

{b, d}

A Combined Algorithm

```
a = b;

c = a;

d = a + b;

e = d;
{a, b}
d = a;


{b, d}
```

A Combined Algorithm

```
a = b;

c = a;

d = a + b;

e = d;
{a, b}
d = a;

{b, d}
```

A Combined Algorithm

```
a = b;

c = a;

d = a + b;

{a, b}
d = a;

{b, d}
```

A Combined Algorithm

```
a = b;

c = a;

d = a + b;


{a, b}
d = a;


{b, d}
```

A Combined Algorithm

```
a = b;

c = a;



{a, b}
d = a;

{b, d}
```

A Combined Algorithm

```
a = b;

c = a;



{a, b}
d = a;

{b, d}
```

A Combined Algorithm

```
a = b;
```

{a, b}

```
d = a;
```

{b, d}

A Combined Algorithm

{b}

```
a = b;
```

{a, b}

```
d = a;
```

{b, d}

A Combined Algorithm

```
a = b;
```

```
d = a;
```

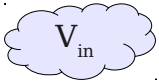
Informacije potrebne za lokalnu analizu

U okviru lokalne analize imamo tri suštinska pitanja:

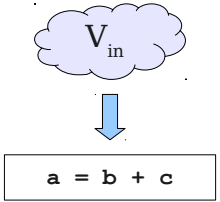
1. U kojem smeru se vrši analiza?
   - Videli smo dve analize sa dva različita smera: unapred i unazad
   - U zavisnosti od lokalne analize, imamo nekada jedan, a nekada drugi smer
2. Na koji način ažuriramo informacije prilikom obrade pojedinačne naredbe?
3. Koje informacije imamo inicijalno?

Another View of Local Analyses

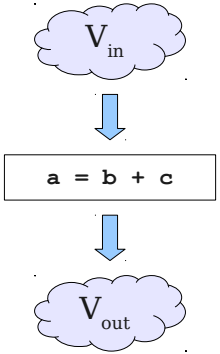
Another View of Local Analyses



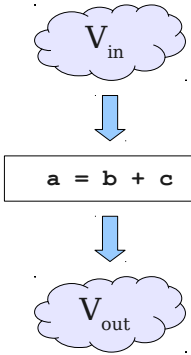
Another View of Local Analyses



Another View of Local Analyses



Another View of Local Analyses



$$V_{out} = f_{a = b+c}(V_{in})$$

Lokalna analiza formalno

- Lokalna analiza osnovnog bloka se definiše kao uređena četvorka (D, V, F, I) gde je
  - D (engl. *direction*) je smer, može biti unapred ili unazad
  - V je skup vrednosti koje program ima u svakoj tački
  - F je familija funkcija prenosa (engl. *transfer functions*) koje difinišu značenje svakog izraza kao funkciju $f : V \to V$
  - I (engl. *initial*) je početna informacija na vrhu (ili dnu) osnovnog bloka

Analiza dostupnih izraza

- Smer: Unapred
- Domen: Skup izraza dodeljenih promenljivama
- Funkcije prenosa: Za dati skup dodela V i naredbu `a = b + c`
  - Ukloni iz V svaki izraz koji koristi a kao podizraz
  - Dodaj u V izraz `a = b + c`
- Početne vrednosti: Prazan skup izraza

Analiza živosti

- Smer: Unazad
- Domen: Skup promenljivih
- Funkcije prenosa: Za dati skup promenljivih i naredbu `a = b + c`
  - Ukloni a iz V (svaka prethodna vrednost od a je sada mrtva)
  - Dodaj u V promenljive b i c (svaka prethodna vrednost od b i c je sada živa)

  Formalno: $f_{a=b+c}(V) = (V - a) \cup \{b, c\}$
- Početne vrednosti: Zavisi od semantike jezika

Algoritam

- Ako je data analiza (D, V, F, I) za osnovni blok
  - Pretpostavimo da je D unapred, analogno bi bilo i unazad
- Najpre, postavi OUT[entry] na I (entry je labela koja označava ulazak u blok)
- Za svaku naredbu s, redom
  - Postavi IN[s] na OUT[prev], gde je prev prethodna naredba
  - Postavi OUT[s] na $f_s(IN[s])$ gde je f_s funkcija prenosa za naredbu s

Pregled

Globalne optimizacije

- Globalna analiza je analiza koja radi na grafu kontrole toka
- Suštinski moćnija od lokalne analize ali i suštinski komplikovanija od lokalne analize
- Mnoge optimizacije zasnovane na lokalnoj analizi mogu da se sprovedu i globalno
- Neke optimizacije ne mogu da se sprovedu lokalno već mogu samo globalno (na primer, pomeranje koda iz jednog u drugi osnovni blok kako bi se izbeglo nepotrebno izračunavanje)
- Primeri globalnih optimizacija: globalna eliminacija mrtvog koda (engl. *global dead code elimination*), globalno kopiranje konstanti (engl. *global constant propagation*), parcijalna eliminacija redudantnosti (engl. *partial redundancy elimination*).

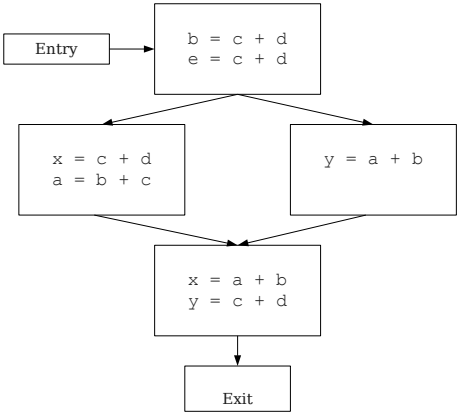
Globalna eliminacija mrtvog koda

- Lokalna eliminacija mrtvog koda mora da zna koje vrednosti su žive na izlasku iz osnovnog bloka
- Ova informacija može da se sračuna samo kao deo globalne analize
- Na koji način modifikovati analizu živosti tako da obuhvati CFG?
- Osnovni problemi: više predaka i petlje

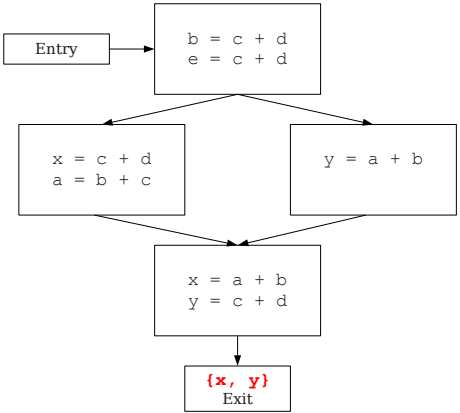
Glavni izazovi, prvi deo

- U okviru lokalne analize, svaka naredba ima tačno jednog prethodnika
- U okviru globalne analize, svaka naredba može imati više prethodnika
- Globalna analiza mora imati neku vrstu kombinovanja informacija od svih prethodnika osnovnog bloka
- U narednom primeru, informacije kombinujemo unijom

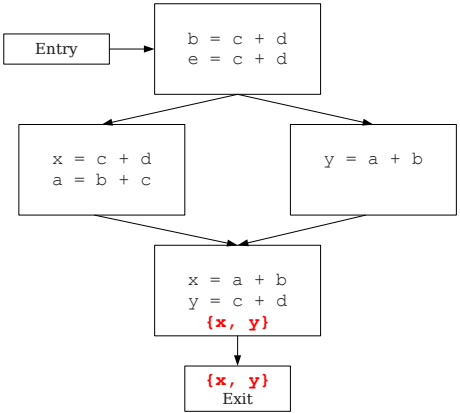
CFGs Without Loops



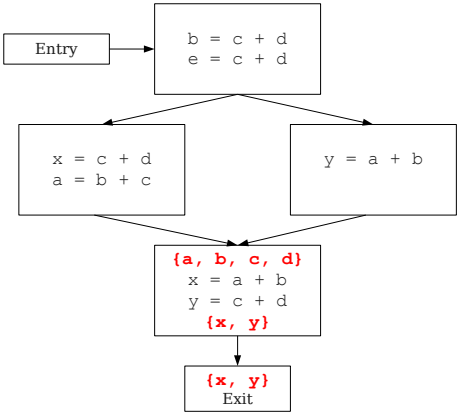
CFGs Without Loops



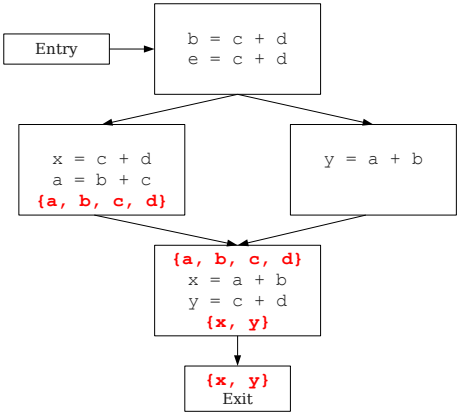
CFGs Without Loops



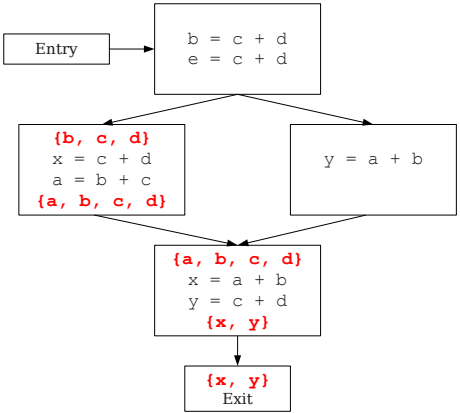
CFGs Without Loops



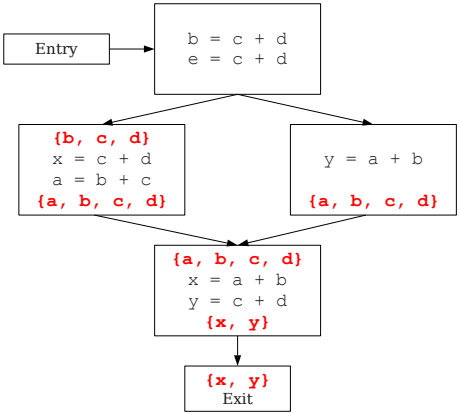
CFGs Without Loops



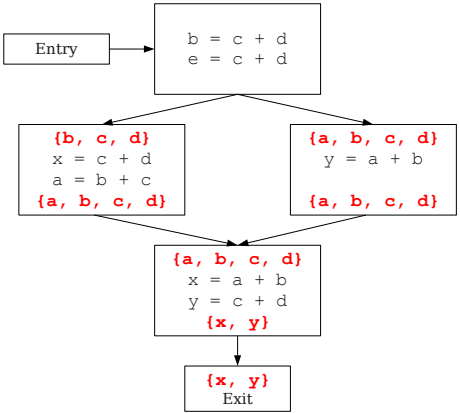
CFGs Without Loops



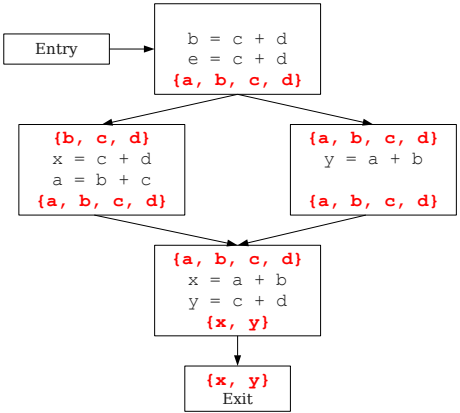
CFGs Without Loops



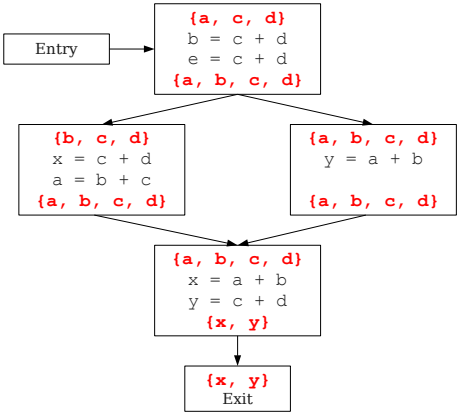
CFGs Without Loops



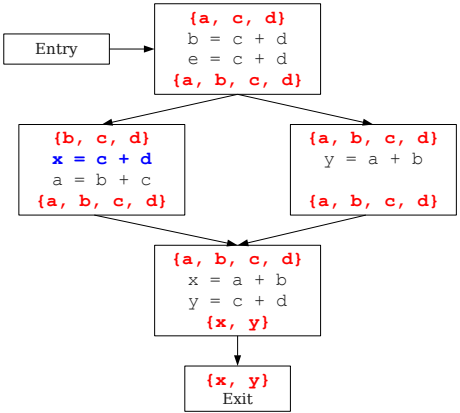
CFGs Without Loops



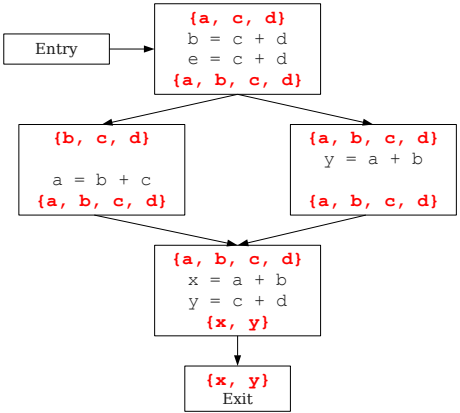
CFGs Without Loops



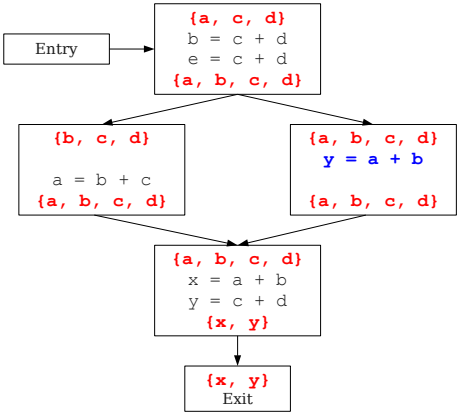
CFGs Without Loops



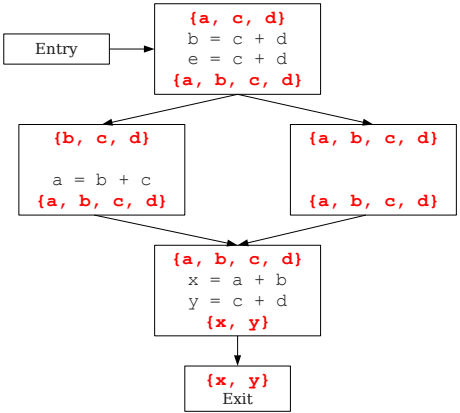
CFGs Without Loops



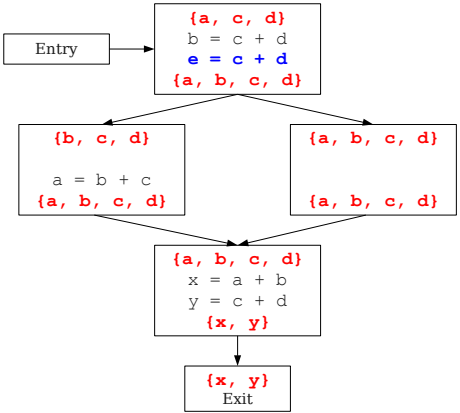
CFGs Without Loops



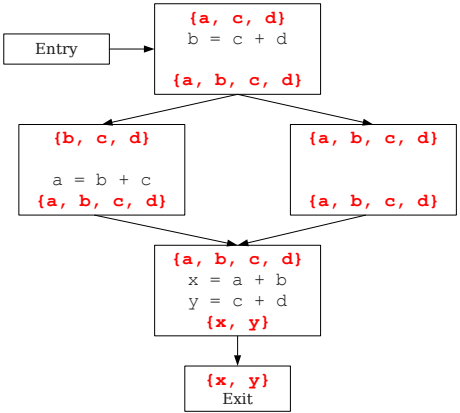
CFGs Without Loops



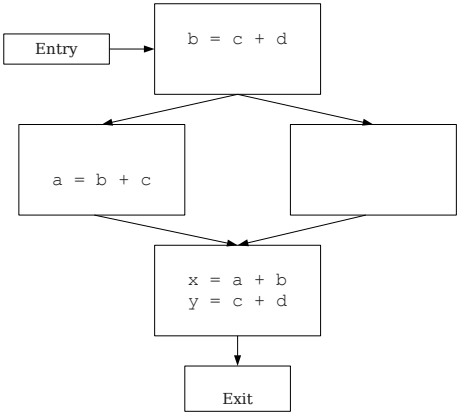
CFGs Without Loops



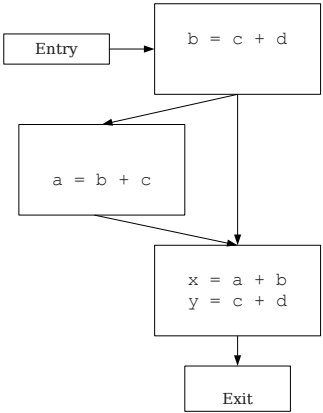
CFGs Without Loops



CFGs Without Loops



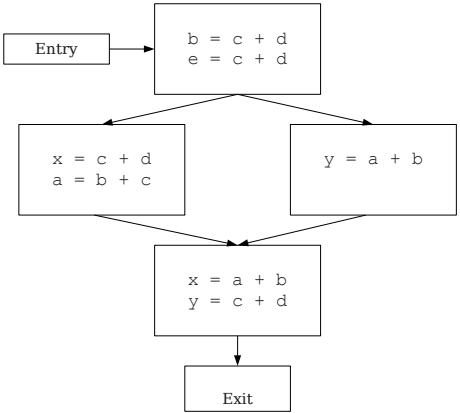
CFGs Without Loops



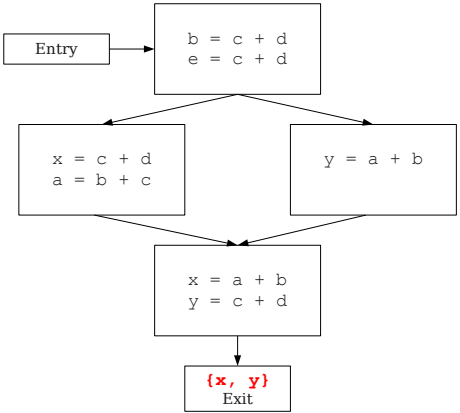
Glavni izazovi, drugi deo

- U okviru lokalne analize, postoji samo jedan mogući put kroz osnovni blok
- U okviru globalne analize, može postojati puno puteva kroz CFG
- Može da bude potrebno da se ponovo izračunaju vrednosti više puta, tj uvek kada nove informacije postanu dostupne
- Treba biti oprezan kako se ne bi desilo da se upadne u beskonačnu petlju

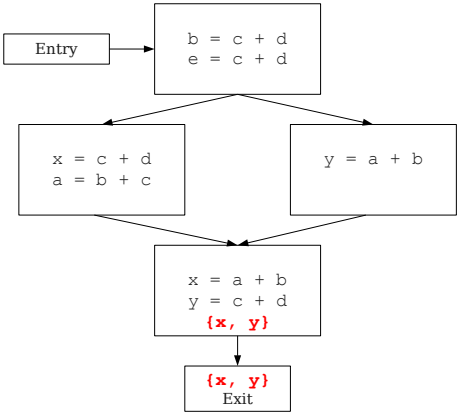
CFGs Without Loops



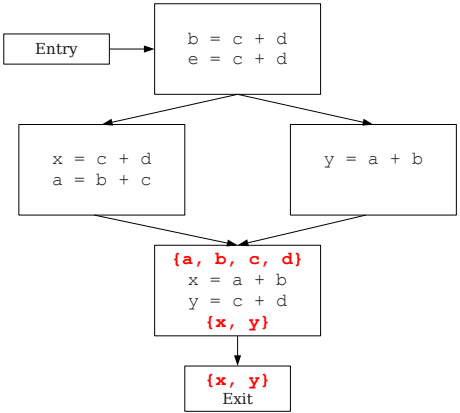
CFGs Without Loops



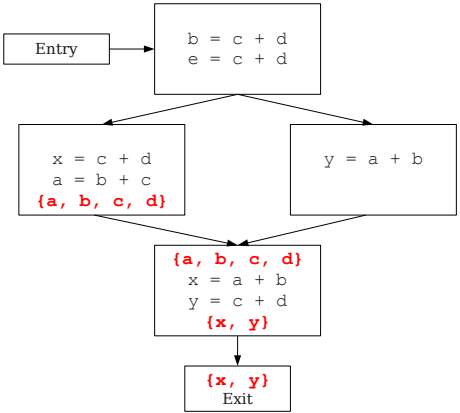
CFGs Without Loops



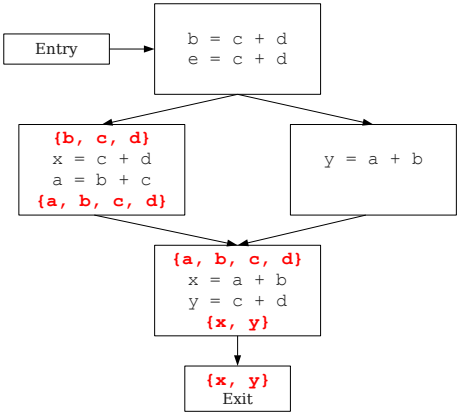
CFGs Without Loops



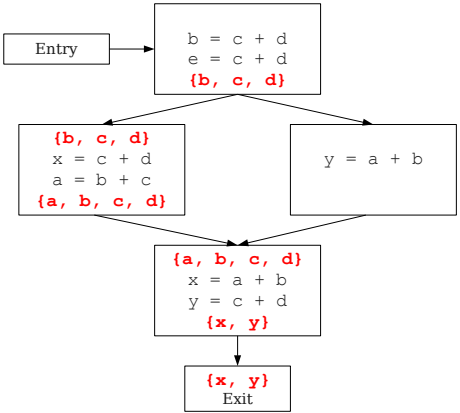
CFGs Without Loops



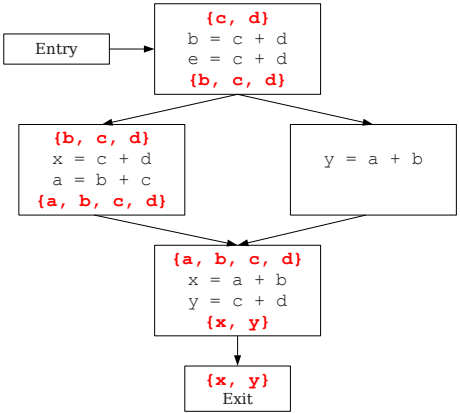
CFGs Without Loops



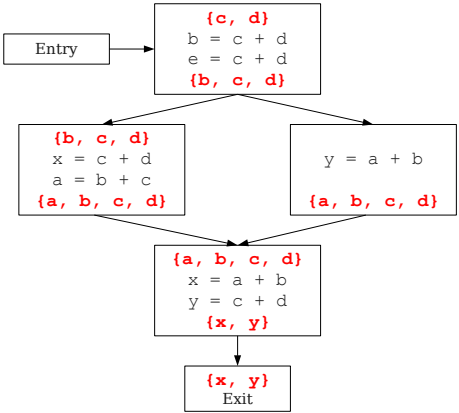
CFGs Without Loops



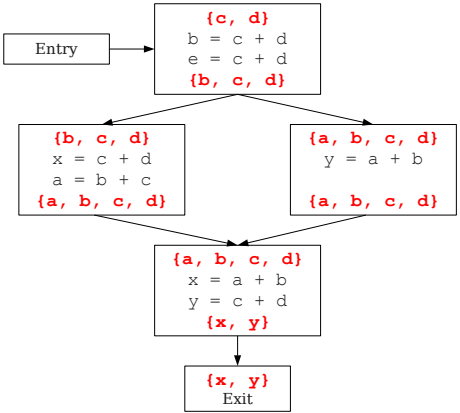
CFGs Without Loops



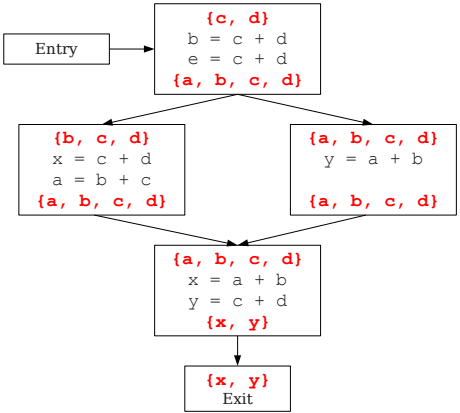
CFGs Without Loops



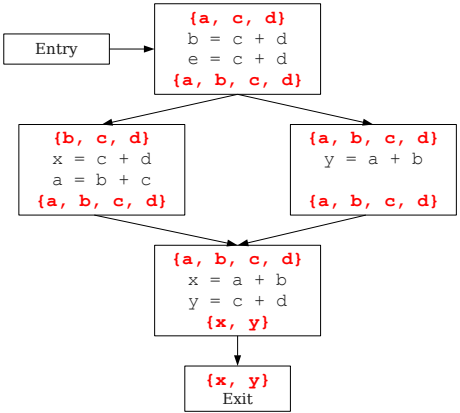
CFGs Without Loops



CFGs Without Loops

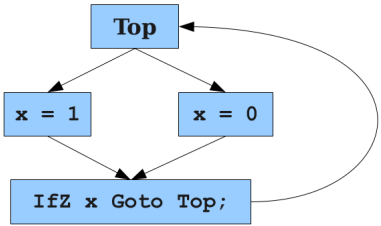


CFGs Without Loops



CFG sa petljom

- Do sada, razmatrali smo samo CFGove bez petlji koji imaju konačno mnogo mogućih putanja
- Kada dodamo petlje, ovo više nije istina
- Primetimo da nisu sve mogućnosti CFGa ostvarive u izvršavanjima programa

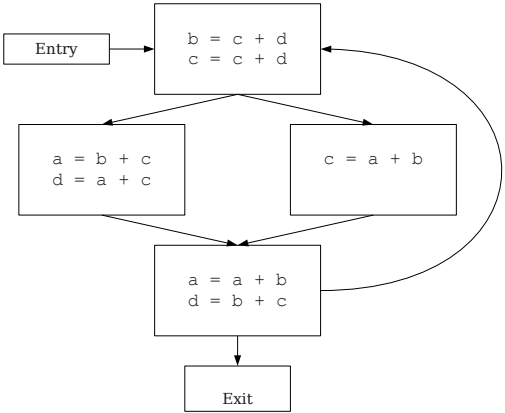


CFG sa petljom

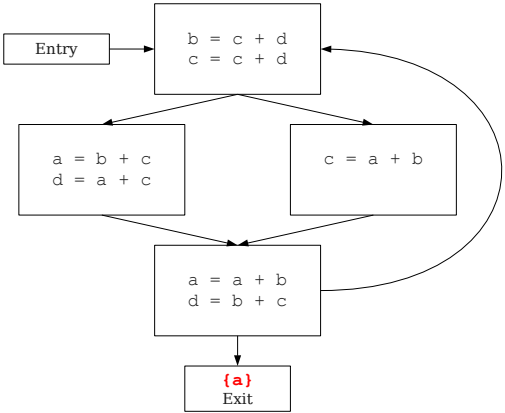
- Saglasna aproksimacija: pretpostvimo da svaka moguća putanja kroz CFG odgovara validnom izvršavanju
- Na ovaj način, uključujemo sve putanje koje se mogu realizovati, ali i neke dodatne putanje
- To može da učini našu analizu manje preciznom, ali i dalje saglasnom
- Zahvaljujući ovome, možemo i da ostvarimo analizu

CFGs With Loops

CFGs With Loops



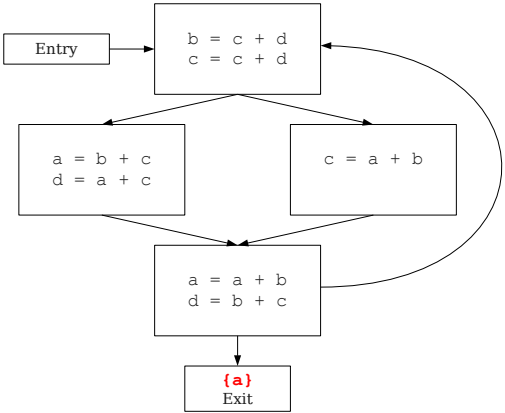
CFGs With Loops



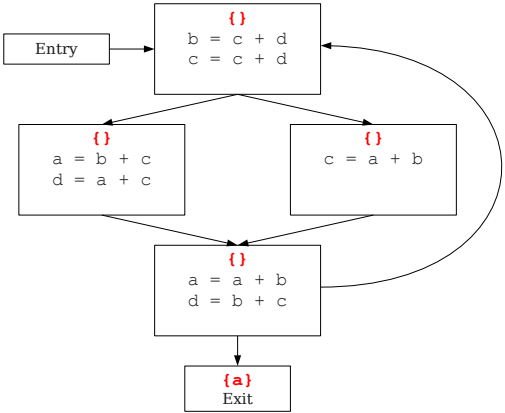
Glavni izazovi, treći deo

- U okviru lokalne analize, uvek je definisana prva naredba od koje se počinje
- U okviru globalne analize sa petljama, svaki blok može da zavisi od svakog drugog bloka
- Da bismo ovo rešili, moramo da dodelimo nekakve početne vrednosti svim blokovima CFGa

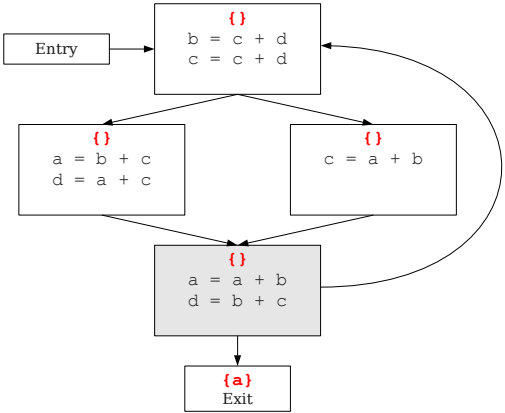
CFGs With Loops



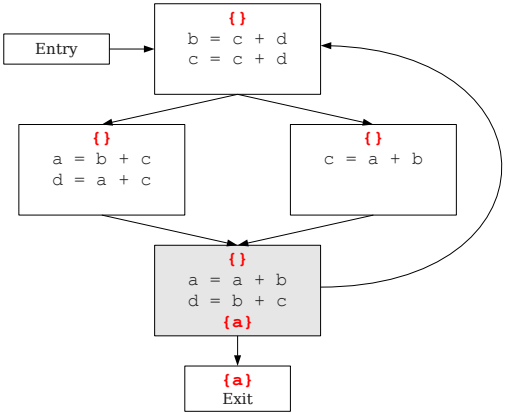
CFGs With Loops



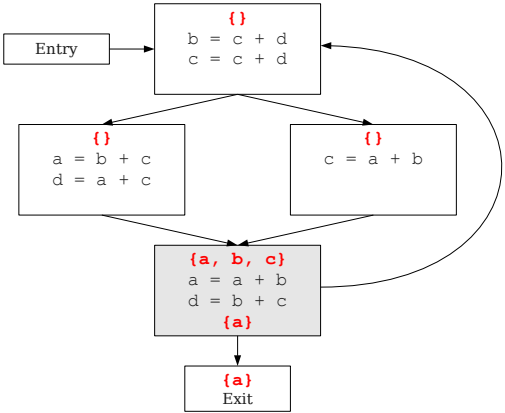
CFGs With Loops



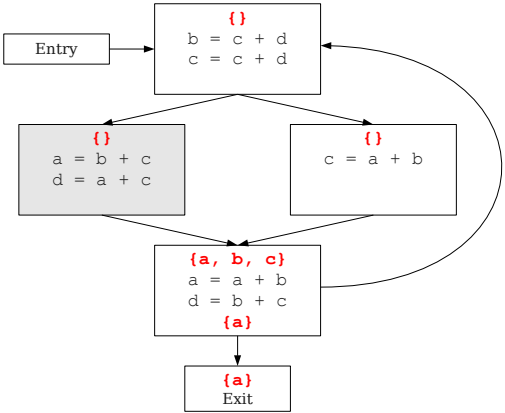
CFGs With Loops



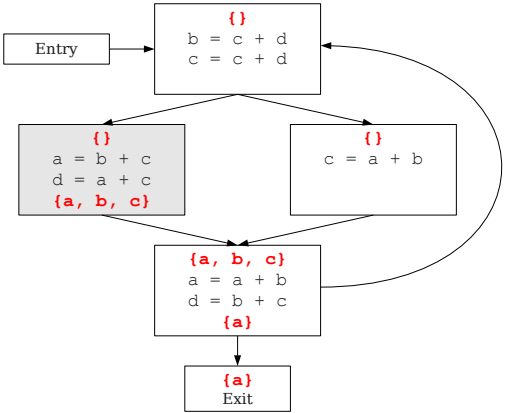
CFGs With Loops



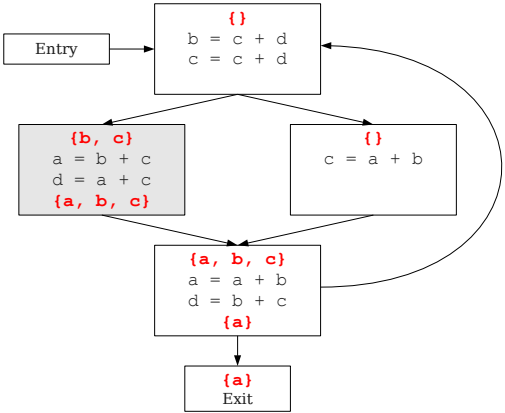
CFGs With Loops



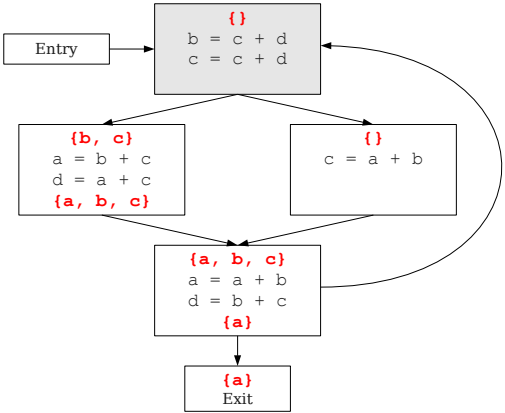
CFGs With Loops



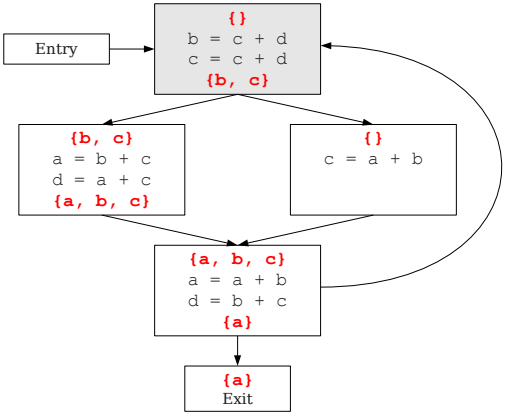
CFGs With Loops



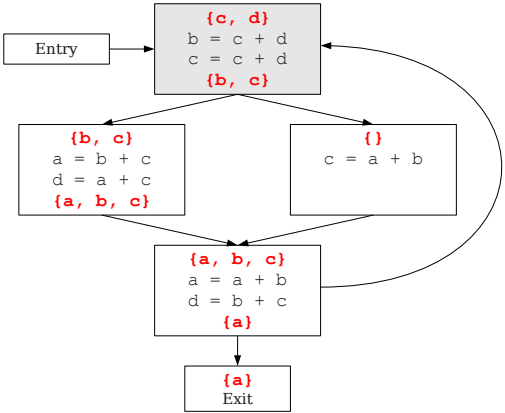
CFGs With Loops



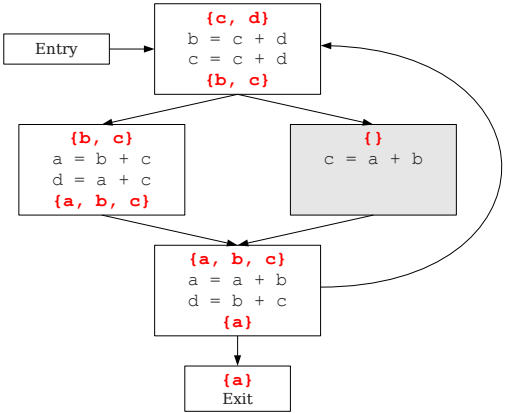
CFGs With Loops



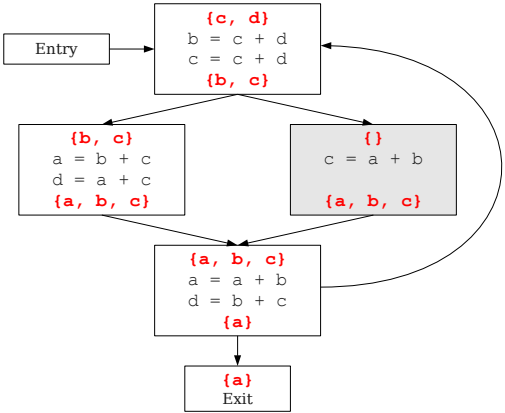
CFGs With Loops



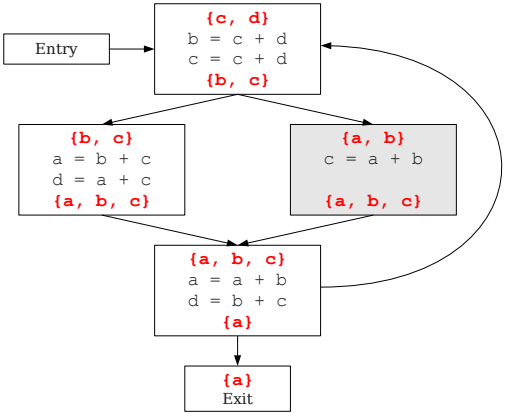
CFGs With Loops



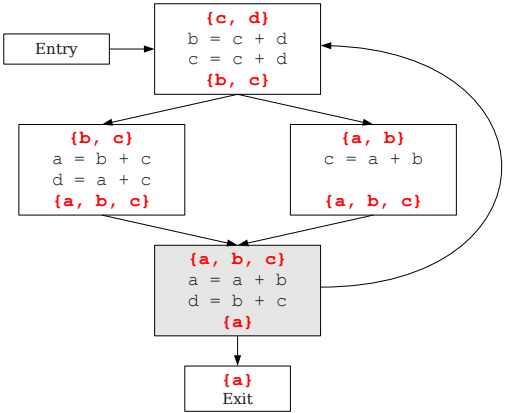
CFGs With Loops



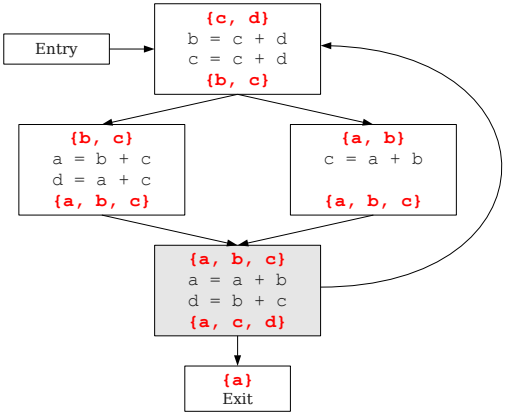
CFGs With Loops



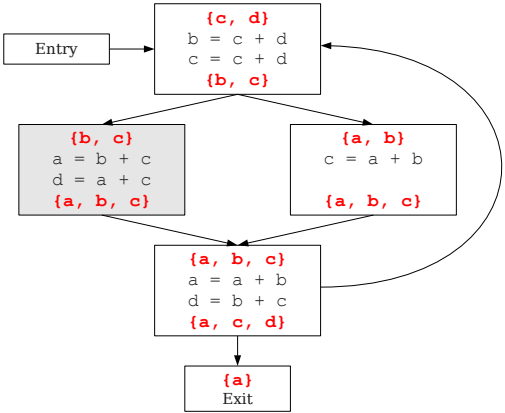
CFGs With Loops



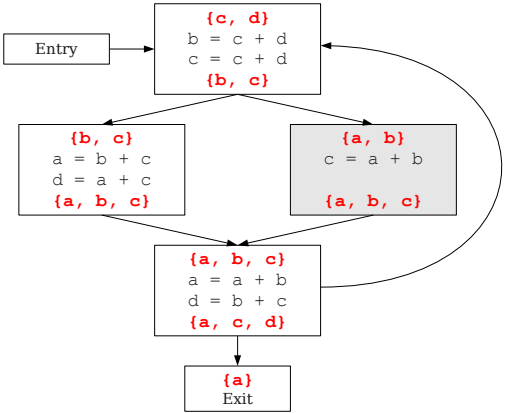
CFGs With Loops



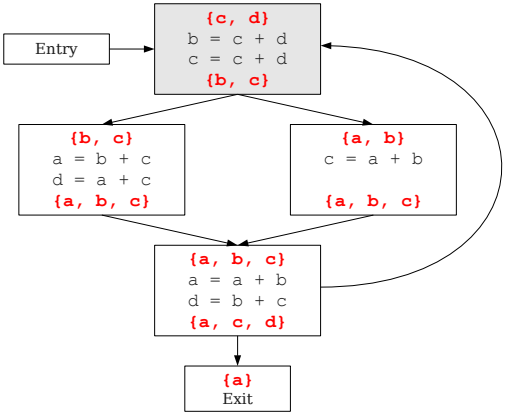
CFGs With Loops



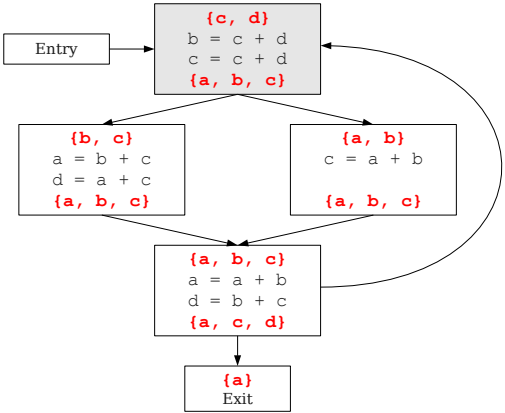
CFGs With Loops



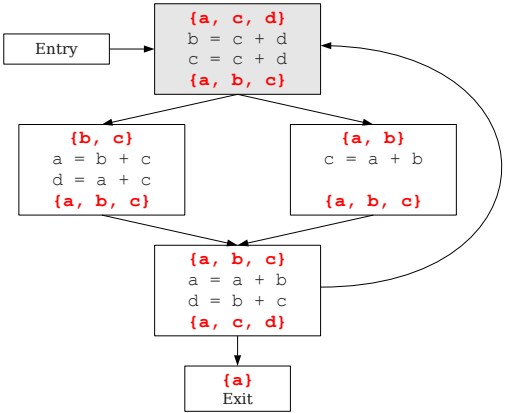
CFGs With Loops



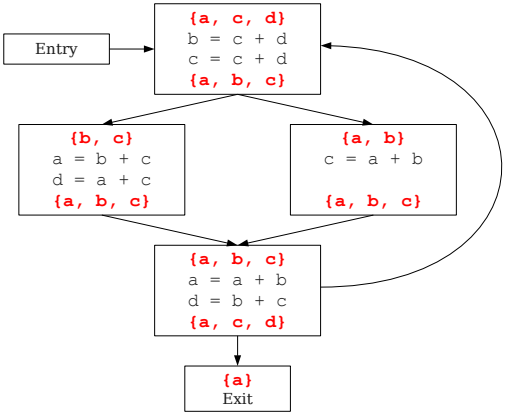
CFGs With Loops



CFGs With Loops



CFGs With Loops



Sumiranje razlika

- Potrebno je da možemo da obradimo više prethodnika/sledbenika osnovnog bloka
- Potrebno je da možemo da obradimo više putanja kroz CFG i može biti potrebe da se itereira više puta da bi se sračunala konačna vrednost (ali važno je da analiza mora da se završi u nekom trenutku!)
- Potrebno je da možemo da dodelimo svakom osnovnom bloku smislenu osnovnu početnu vrednost pre nego što ga analiziramo

Globalna analiza živosti

- Na početku, postavi IN[s] = { } za svaku naredbu s
- Postavi IN[exit] na skup promenljivih za koje se zna da su žive na izlazu (znanje specifično za programski jezik)
- **Ponavljaj sve dok ima promena** (zaustavi se kada nema promena)
  - Za svaku naredbu s oblika a = b + c, u bilo kom redosledu obilaska:
    - Postavi OUT[s] na skup unije od IN[p] za svaki sledbenik p od naredbe s
    - Postavi IN[s] na $(OUT[s] - a) \cup \{b, c\}$

Da li je ovakav algoritam dobar?

- Da bi pokazali korektnost, potrebno je da pokažemo naredno:
  - Algoritam se zaustavlja
  - Kada se algoritam zaustavi, daje saglasno rešenje
- Argumenti za zaustavljanje:
  - Kada se pronađe da je neka promenljiva živa u nekom delu programa, to uvek važi
  - Postoji konačno mnogo promenljivih i konačno mnogo mesta na kojima promenljive mogu da postanu žive
- Argument o saglasnosti
  - Svako pojedinačno pravilo, primenjeno na neki skup, korektno ažurira živost skupa
  - Kada se računa unija dva skupa živih promenljivih, promenljiva je živa samo ako je bila živa na nekoj putanji koja vodi do naredbe

Formalizam

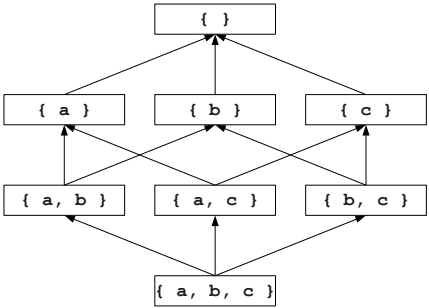
- Građenje formalizma koji može da obuhvati i dizajnira ovakvu analizu je zahtevno
- Osnovne ideje, ipak su nezavisne od same analize
  - Potrebno je da možemo da računamo funkcije koje opisuju ponašanje svake naredbe
  - Potrebno je da možemo da skupimo više izračunavanja zajedno
  - Potrebna nam je početna vrednost za svaki osnovni blok
- Postoji formalizam koji može da uhvati većinu ovih osobina

Polumreža sa opreatorom spajanja (engl. meet semilattice)

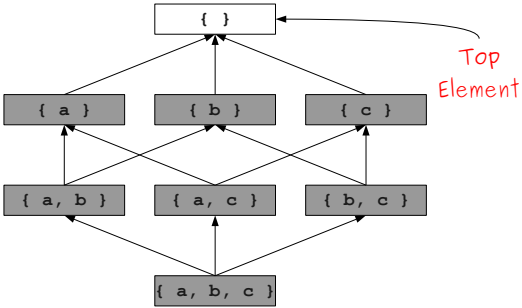
- Polumreža sa operatorom spajanja je uređenje definisano na skupu elemenata
- Svaka dva elementa imaju spajanje koje je najveći element koji je manji od oba elementa
- Intuitivno, spajanje dva elementa predstavlja kombinovanje informacija od ta dva elementa i to je nekakva uopštenija informacija od informacija koje su bile početne (uopštenija informacija znači da je manje precizna)
- Postoji jedinstven element na vrhu koji se naziva i *top*, koji je *veći* od svih drugih elemenata
- Element na vrhu predstavlja „nema informacija" (veoma precizna informacija)

Meet Semilattices for Liveness

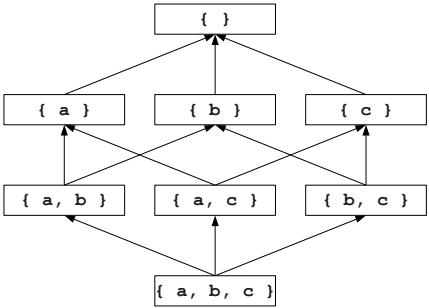
Meet Semilattices for Liveness



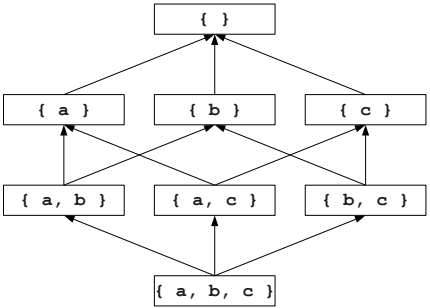
Meet Semilattices for Liveness



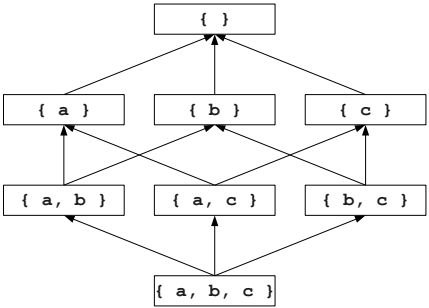
Meet Semilattices for Liveness

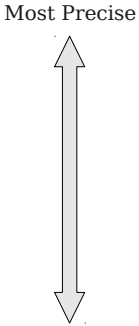


Meet Semilattices and Orderings



Meet Semilattices and Orderings





Formalna definicija

- Polumreža sa operatorom spajanja je uređeni par $(D, \wedge)$ gde je
  - D je skup koji označava domen elemenata
  - $\wedge$ je operator spajanja koji je
    - idempotentan $x \wedge x = x$
    - komutativan $x \wedge y = y \wedge x$
    - asocijativan $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
  - Ako važi $x \wedge y = z$ kažemo da je z spajanje (ili najveća donja granica) za x i y.
- Svaka polumreža sa operatorom spajanja ima element na vrhu koji se označava sa $\top$ takav da je $\top \wedge x = x$ za svako x.

Primer 1

- Skup prirodnih brojeva i funkcija max
- Idempotencija max{a,a} = a
- Komutativnost max{a, b} = max{b, a}
- Asocijativnost max{a, max{b, c}} = max{max{a, b}, c}
- Top element je 0: max{0, a} = a

Primer 2 — Polumreža za analizu živosti

- Skup živih promenljivih i operacija unije
- Idempotencija $a \cup a = a$
- Komutativnost $a \cup b = b \cup a$
- Asocijativnost $(a \cup b) \cup c = a \cup (b \cup c)$
- Top element je $\emptyset$: $\emptyset \cup a = a$

Polumreže i analiza programa

- Polumreže prirodno rešavaju veliki broj problema na koje nailazimo u globalnoj analizi
- Kako kombinujemo informacije iz različitih osnovnih blokova?
  — Koristimo operator spajanja.
- Koju vrednost dajemo kao početnu vrednost svakom bloku?
  — Vrednost top.
- Kako znamo da će se algoritam završiti?
  — Zapravo, to ne možemo još uvek da garantujemo: potrebno je da postoje odgovarajuće osobine polumreže (konačna visina) kao i funkcije spajanja (monotona funkcija)

Opšti okvir

- Globalna analiza je petorka $(D, V, \wedge, F, I)$ pri čemu je
  - D smer - napred, nazad
  - V je skup vrednosti
  - $\wedge$ je operator spajanja na ovim vrednostima
  - F je skup funkcija prenosa $f : V \to V$
  - I je početno stanje
- Jedina razlika u odnosu na lokalnu analizu je uvođenje operatora spajanja

Algoritam globalne analize

- Pretpostavimo da je $(D, V, \wedge, F, I)$ analiza unapred
- Postavi $OUT[s] = \top$ za svaku naredbu s
- Postavi $OUT[begin] = I$
- Ponavljaj sve dok ima izmena (zaustavi se kada ne bude izmena)
  - Za svaku naredbu s sa predthodnicima $p_1, p_2, ...p_n$
    - Postavi $IN[s] = OUT[p1] \wedge OUT[p2] \wedge ... \wedge OUT[p_n]$
    - Postavi $OUT[s] = f_s(IN[s])$
  - Redosled iteracija nije važan
- Ova vrsta analize naziva se okvir toka podataka (engl. *dataflow framework*). Uz dodatne pretpostavke, može se koristiti da se dokaže svojstvo završavanja analize
- Postoje i druge interesantne globalne analize

Pregled

Literatura

- (The Dragon Book) Compilers: Principles, Techniques, and Tools Alfred V. Aho,Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- Kompajleri Stanford
  `https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/`