Překladače (3. přednáška)

Jan Hubička

Katedra aplikované matematiky
Univerzita Karlova
Praha

23. března 2020

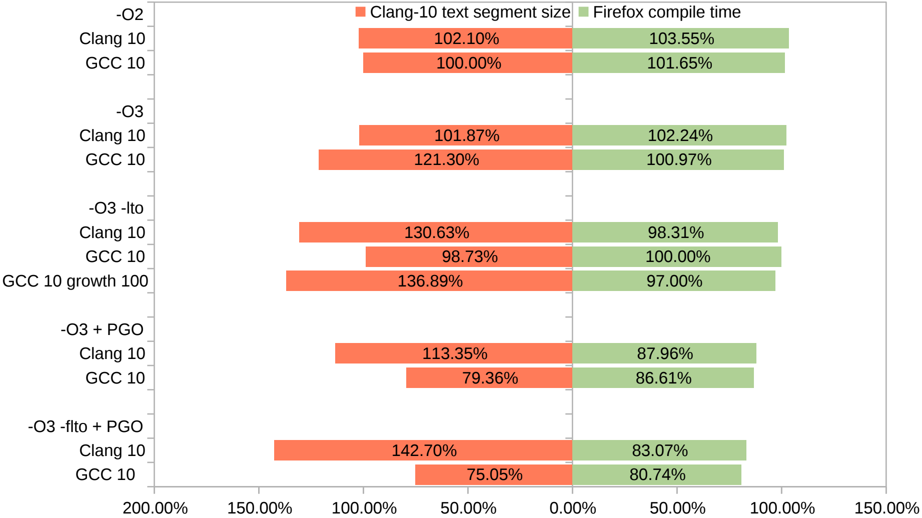
Minulá přednáška

Algoritmus pro jednoduché optimalizace v rámci:

- basic bloku
- super bloku
- extended basic bloku

Rychlostní rozdíl mezi GCC přeloženém s `-O0` a `-O2`: 3.3×

Rychlost clangu

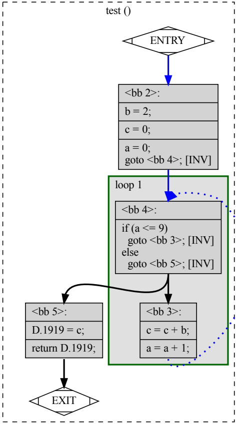


Propagace konstant nad celou funkcí

Chceme pro každý basic block spočítat pole IN. Kde IN[bb][prom] bude obsahovat hodnotu pokud je konstatní jinak $\perp$.

Globální propagace konstant

```
1 IN$[*][*] = \top$
2 OUT$[*][*] = \top$
3 Dokud se hodnoty mění:
4     Pro každý basic block bb
5         Obnov pole IN[bb] sloučením všech hodnot OUT[p],
          kde p je předchůdce bb.
6         Odsimuluj kód v bb a urči OUT[bb].
7 Proveď lokální propagaci konstant
```



Základní vlastnosti

Globální propagace konstant

```
1 $IN[*][*] = \top$
2 $OUT[*][*] = \top$
3 Dokud se hodnoty mění:
4         Pro každý basic block bb
5                 Obnov pole IN[bb] sloučením všech hodnot OUT[p],
                  kde p je předchůdce bb.
6                 Odsimuluj kód v bb a urči OUT[bb].
7 Proveď lokální propagaci konstant
```

Lemma (Konečnost)

Algoritmus skončí po $3 * n * p$ *opakování vnější smyčky, kde* n *je počet basic bloku a* p *je počet proměnných*

Základní vlastnosti

Globální propagace konstant

```
1 $\mathsf{IN}[*][*] = \top$
2 $\mathsf{OUT}[*][*] = \top$
3 Dokud se hodnoty mění:
4         Pro každý basic block bb
5                 Obnov pole IN[bb] sloučením všech hodnot OUT[p],
                  kde p je předchůdce bb.
6                 Odsimuluj kód v bb a urči OUT[bb].
7 Proveď lokální propagaci konstant
```

Lemma (Korektnost)

Výsledné hodnoty v poli IN *jsou korektní: pokud* $\mathsf{IN}[\mathsf{bb}][\mathsf{prom}] = \mathsf{c}$ *kde* c *je nějaká konstanta, potom pro všechny možné běhy algoritmu bude hodnot* prom *na začátku basic bloku* bb *rovna* c*.*

Základní vlastnosti

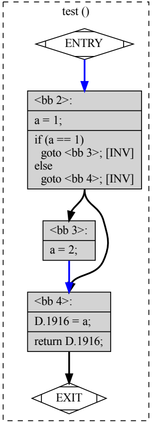
Globální propagace konstant

```
1 $\mathrm{IN}[*][*] = \top$
2 $\mathrm{OUT}[*][*] = \top$
3 Dokud se hodnoty mění:
4         Pro každý basic block bb
5                 Obnov pole IN[bb] sloučením všech hodnot OUT[p],
                  kde p je předchůdce bb.
6                 Odsimuluj kód v bb a urči OUT[bb].
7 Proveď lokální propagaci konstant
```

Lemma (Optimalita)

Výsledné hodnoty v poli IN jsou optimální: pokud $\mathrm{IN}[bb][prom] = \bot$ *potom existují procházky z entry do bb, které nastaví prom na různé hodnoty.*

“Optimalita”



test ()

ENTRY

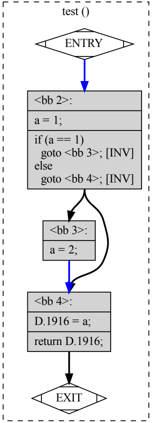
```
<bb 2>:
a = 1;
if (a == 1)
  goto <bb 3>; [INV]
else
  goto <bb 4>; [INV]
```

```
<bb 3>:
a = 2;
```

```
<bb 4>:
D.1916 = a;
return D.1916;
```

EXIT

“Optimalita”



test ()

ENTRY

```
<bb 2>:
a = 1;
if (a == 1)
  goto <bb 3>; [INV]
else
  goto <bb 4>; [INV]
```

```
<bb 3>:
a = 2;
```

```
<bb 4>:
D.1916 = a;
return D.1916;
```

EXIT

Později ukážeme jak algoritmus zlepšit o propagaci podmínek

Dataflow

Obecně řešíme dataflow problém. Ten je zadán pomocí:

- Control flow grafu
- Polosvazu hodnot, které počítáme. Polosvaz definuje top $\top$ a slučovací operaci $\wedge$
- Přechodovými funkcemi jump(bb) pro jednotlivé basic blocky
- Směr propagace (dopředu, dozadu, obousměrně)

Iterativni dopředné dataflow

```
$IN[*][*] = \top$
$OUT[*][*] = \top$
Dokud se hodnoty mění:
    Pro každý basic block bb
        Obnov pole IN[bb] sloučením všech hodnot OUT[p], kde p je předchůdce bb.
        $OUT[bb] = jump(bb)$.
```

Příklady dataflow

1. Propagace konstant

Příklady dataflow

1. Propagace konstant
2. Availability (dostupnost výrazů)

Příklady dataflow

1. Propagace konstant
2. Availability (dostupnost výrazů)
3. Liveness (životnost proměnných)

Příklady dataflow

1. Propagace konstant
2. Availability (dostupnost výrazů)
3. Liveness (životnost proměnných)
4. Reaching definitions

Příklady dataflow

1. Propagace konstant
2. Availability (dostupnost výrazů)
3. Liveness (životnost proměnných)
4. Reaching definitions
5. Value range propagation

Příklady dataflow

1. Propagace konstant
2. Availability (dostupnost výrazů)
3. Liveness (životnost proměnných)
4. Reaching definitions
5. Value range propagation
6. . . .

Dataflow

Iterativni dopředné dataflow

```
$IN[*][*] = \top$
$OUT[*][*] = \top$
Dokud se hodnoty mění:
    Pro každý basic block $bb$
        Obnov pole $IN[bb]$ sloučením všech hodnot $OUT[p]$,
        kde $p$ je předchůdce $bb$.
        $OUT[bb] = jump(bb)$.
```

Lemma (Konečnost)

Algoritmus skončí po $h * n$ *opakování vnější smyčky, kde* n *je počet basic bloků a* h *je výška polosvazu*

Dataflow

Iterativni dopředné dataflow

```
1 IN[*][*] = ⊤
2 OUT[*][*] = ⊤
3 Dokud se hodnoty mění:
4         Pro každý basic block bb  v pořadí reverse postorder
5                 Obnov pole IN[bb] sloučením všech hodnot OUT[p],
                  kde p je předchůdce bb.
6                 OUT[bb] = jump(bb).
```

Lemma (Konečnost)

Algoritmus skončí po $d(\mathrm{CFG}) * h$ *opakování vnější smyčky, kde* $d(\mathrm{CFG})$ *maximální počet vnořených zpětných hran DFS průchodu a h výška polosvazu*

Dataflow

Iterativni dopředné dataflow

```
IN[$*$][$*$] $= \top$
OUT[$*$][$*$] $= \top$
Dokud se hodnoty mění:
        Pro každý basic block bb  v pořadí reverse postorder
                Obnov pole IN[bb] sloučením všech hodnot OUT[p],
                kde p je předchůdce bb.
                OUT[bb] = jump(bb).
```

Lemma (Konečnost)

Algoritmus skončí po $d(\text{CFG}) * h$ opakování vnější smyčky, kde $d(\text{CFG})$ maximální počet vnořených zpětných hran DFS průchodu a h výška polosvazu

Ve ručně psaném kódu lze považovat $d(\text{CFG})$ za malou konstantu (maximální počet vnořených smyček)

Rychlost dataflow

Překladače postavené na dataflow solverech jsou výrazně pomalejší než překladače založené na lokální optimalizaci.

Situaci lze zlepšit:

1. Maximální využití rapid dataflow problémů (kde svaz je $0, 1$)

Rychlost dataflow

Překladače postavené na dataflow solverech jsou výrazně pomalejší než překladače založené na lokální optimalizaci.

Situaci lze zlepšit:

1. Maximální využití rapid dataflow problémů (kde svaz je $0, 1$)
2. Existují složitější solvery, které umí obnovovat řešení po transformaci

Rychlost dataflow

Překladače postavené na dataflow solverech jsou výrazně pomalejší než překladače založené na lokální optimalizaci.

Situaci lze zlepšit:

1. Maximální využití rapid dataflow problémů (kde svaz je 0, 1)
2. Existují složitější solvery, které umí obnovovat řešení po transformaci
3. Řídké reprezentace dataflow (DU/UD chains)