

Metody Realizacji Języków Programowania

Generacja i ulepszanie kodu

Marcin Benke

MIM UW

3 stycznia 2011

Generacja i ulepszanie kodu

- Bloki proste, grafy i analiza przepływu
- Zmienne żywe i docierające definicje
- Generacja kodu
- Alokacja rejestrów
- Ulepszanie (“optymalizacja”) kodu
 - ▶ Zwijanie stałych
 - ▶ Eliminacja wspólnych podwyrażeń
 - ▶ Eliminacja martwego kodu
 - ▶ Optymalizacja pętli

Przykład

```
void quicksort(int m, int n)
{
    int i, j;
    int v, x;
    if (n<=m) return;
// === Początek rozważanego fragmentu ===
    i=m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x =a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
// === K O N I E C ===
    quicksort(m, j); quicksort(i+1, n);
}
```

Kod czwórkowy

```
[ 1]  i := m-1
[ 2]  j := n
[ 3]  t1 := 4*n
[ 4]  v := a[t1]
[ 5]  i := i+1
[ 6]  t2 := 4*i
[ 7]  t3 := a[t2]
[ 8]  if t3 < v goto (5)
[ 9]  j := j-1
[10]  t4 := 4*j
[11]  t5 := a[t4]
[12]  if t5 > v goto (9)
[13]  if i >= j goto (23)
[14]  t6 := 4*i
[15]  x := a[t6]
[16]  t7 := 4*i
[17]  t8 := 4*j
[18]  t9 := a[t8]
[19]  a[t7] := t9
[20]  t10 := 4*j
[21]  a[t10] := x
[22]  goto (5)
[23]  t11 := 4*i
[24]  x := a[t11]
[25]  t12 := 4*i
[26]  t13 := 4*n
[27]  t14 := a[t13]
[28]  a[t12] := t14
[29]  t15 := 4*n
[30]  a[t15] := x
```

Blok prosty

Definicja

Blok prosty jest sekwencją kolejnych instrukcji, do której sterowanie wchodzi wyłącznie na początku i z którego wychodzi wyłącznie na końcu, bez możliwości zatrzymania ani rozgałęzienia wewnątrz.

Rozważmy fragment:

```
[ 1]  i := m-1
[ 2]  j := n
[ 3]  t1 := 4*n
[ 4]  v := a[t1]

[ 5]  i := i+1
[ 6]  t2 := 4*i
[ 7]  t3 := a[t2]
[ 8]  if t3 < v goto (5)
```

Sekwencja 5–8 tworzy blok prosty; sekwencja 1-8 nie.

Blok prosty

Rozważany fragment składa się z dwu bloków prostych:

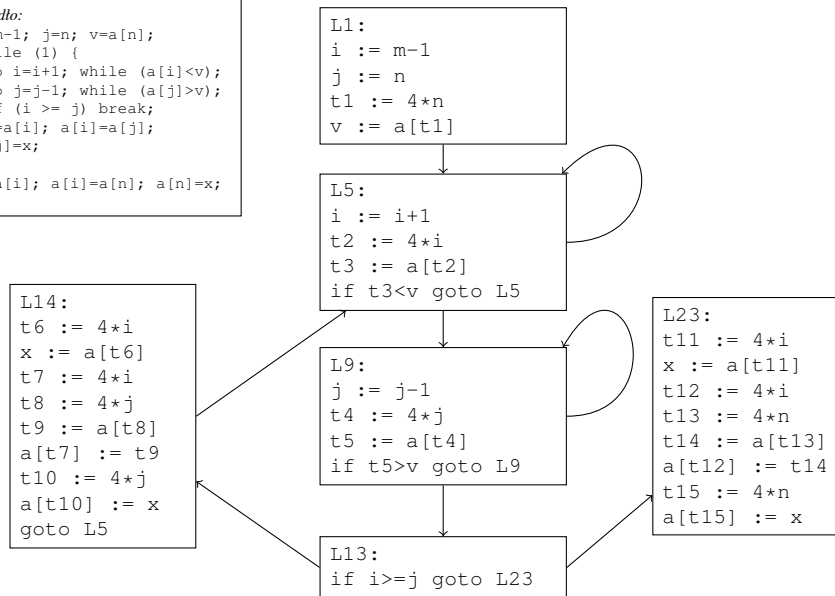
```
L1:  i := m-1  
     j := n  
     t1 := 4*n  
     v := a[t1]  
     goto L5
```

```
L5:  i := i+1  
     t2 := 4*i  
     t3 := a[t2]  
     if t3 < v goto L5
```

Graf bloków prostych (przepływu sterowania, CFG)

Źródło:

```
i=m-1; j=n; v=a[n];  
while (1) {  
  do i=i+1; while (a[i]<v);  
  do j=j-1; while (a[j]>v);  
  if (i >= j) break;  
  x=a[i]; a[i]=a[j];  
  a[j]=x;  
}  
x=a[i]; a[i]=a[n]; a[n]=x;
```



Analiza żywotności

- *Definicja zmiennej* — instrukcja nadająca wartość tej zmiennej
- *Użycie zmiennej* — instrukcja odwołująca się do wartości tej zmiennej
- Instrukcja $x := y + z$ definiuje zmienną x , używa zmiennych z oraz y

Definicja

*Zmienna jest **żywa** w danym punkcie, jeśli jej obecna wartość może być jeszcze użyta, tzn istnieje ścieżka od tego punktu do użycia zmiennej, nie zawierająca po drodze definicji tej zmiennej.*

Przykład

Na początku bloku

```
t := a
```

```
a := b
```

```
b := t
```

żywe są zmienne a, b . Zmienna t nie jest żywa, gdyż jedyna ścieżka od początku bloku do użycia t zawiera jej definicję.

Docierające definicje

Definicja

*Definicja dociera do danego punktu, jeśli żadna ścieżka pomiędzy nimi nie zawiera **innej** definicji tej samej zmiennej.*

```
[ 2]  j := n
[ 3]  t1 := 4*n
[ 4]  v := a[t1]
[ 5]  i := i+1
[ 6]  t2 := 4*i
[ 7]  t3 := a[t2]
[ 8]  if t3 < v goto (5)
[ 9]  j := j-1
[10]  t4 := 4*j
[11]  t5 := a[t4]
```

Definicja *t1* z (3) dociera do (11); definicja *j* z (2) nie dociera do (10).

Postać SSA (Static Single Assignment)

Powyższe analizy, jak i późniejsze przekształcenia są łatwiejsze jeśli kod jest w szczególnej postaci: każda zmienna ma tylko jedną definicję.

Taką postać nazywamy postacią SSA: Static Single Assignment — **statycznie** na każdą zmienną jest tylko jedno przypisanie (nic nie stoi natomiast na przeszkodzie by wykonało się wiele razy, np. w pętli)

LLVM wymaga kodu w postaci SSA

Przekształcanie do postaci SSA — blok prosty

W obrębie bloku prostego przekształcenie do postaci SSA jest trywialne: każdą definicję zmiennej zastępujemy przez definicję nowej zmiennej, np.

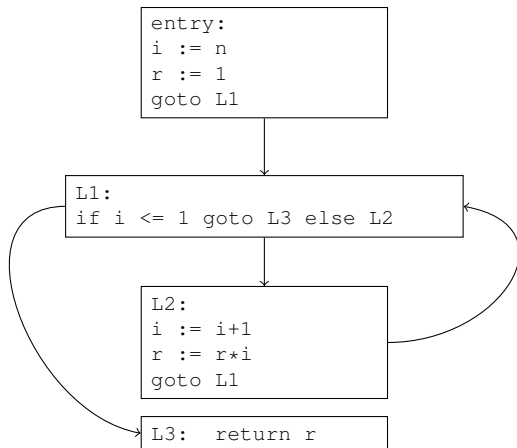
```
i := n
r := 1
r := r * i
i := i - 1
return r
```

Zastępujemy przez

```
i1 := n
r1 := 1
r2 := r1 * i1
i2 := i1 - 1
return r2
```

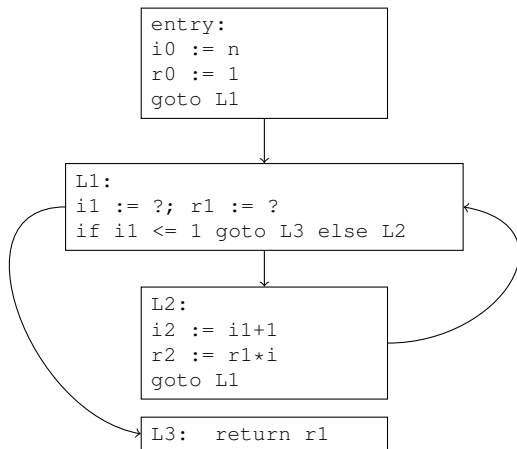
Przekształcanie do postaci SSA — graf sterowania

Problem pojawia się gdy ta sama zmienna jest definiowana na dwóch łączących się ścieżkach w grafie sterowania, np.



Przekształcanie do postaci SSA — graf sterowania

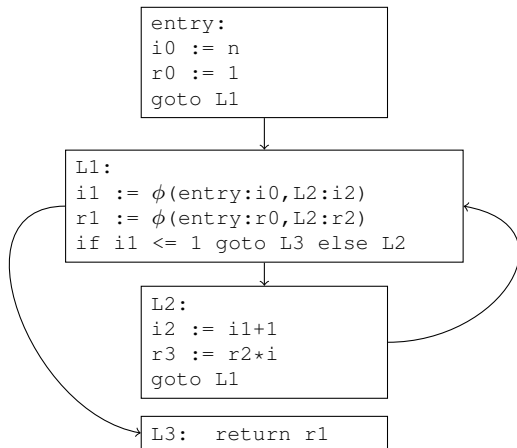
Ponumerowanie zmiennych spowoduje problem



Wartości zmiennych w bloku L1 zależą od tego, którą krawędzią do niego wejdzie sterowanie.

Przekształcanie do postaci SSA — funkcja ϕ

możemy odsunąć problem przy użyciu (hipotetycznej) funkcji ϕ , która wybiera odpowiedni wariant zmiennej:



Generacja kodu maszynowego

- Stan maszyny: zawartość zasobów pamięciowych (rejestrów, stosu, pamięci).
- Podstawowa technika: **symulacja** zachowania maszyny docelowej (ciągu stanów).
- Korzystamy z wzajemnie powiązanych opisów zasobów (głównie rejestrów) oraz opisów zmiennych i wartości.
- Każda wartość wyliczana przez program i każdy zasób są określone przez opisy.

Opis rejestru

- Stan: wolny, zablokowany, etc
- Co zawiera (być może wiele wartości)

Opis wartości

- typ, rozmiar
- Gdzie jest wartość (być może w wielu miejscach)
- Aliasy (np zmienna może być dostępna zarówno bezpośrednio jak i poprzez wskaźnik)
- Opis wartości jest interesujący tylko dla zmiennych żywych

Założenia co do maszyny docelowej

W trakcie wykładu zakładamy, że maszyna posiada n rejestrów ogólnego przeznaczenia: R_0, \dots, R_{n-1} , oraz instrukcje

- LOAD a, R_i (lub MOV a, R_i) — sprowadź wartość spod adresu a (pamięci) do R_i
- STORE R_i, a — zapisz zawartość R_i (do pamięci) pod adresem a
- Adresy mogą być postaci:
 - ▶ stała (adres zmiennej globalnej),
 - ▶ stała+ R_j (miejsce w tablicy),
 - ▶ stała+FP (zmienna lokalna lub argument, ale przeważnie będziemy używać nazw symbolicznych, nie czyniąc rozróżnienia między zmiennymi lokalnymi a globalnymi).
- op R_i, R_j — o znaczeniu $R_j := R_i$ op R_j , gdzie op — operacja arytmetyczna
- op a, R — analogicznie dla komórki pamięci a
- op $\$c, R_i$ — analogicznie dla stałej c

Dygresja — teoria a praktyka

Opisana powyżej maszyna docelowa jest zbliżona do architektury x86. W praktyce nie możemy poczynić nawet tak skromnych założeń, co do wspólnego mianownika dla różnych procesorów, np:

- SPARC nie ma instrukcji LOAD a, Ri a tylko LOAD [Ri], Ri
- Na i386 możemy wykonać dodawanie dla (niemal) dowolnych rejestrów, ale już dzielenie tylko z dzielną w EDX:EAX i ilorazem w EAX i resztą w EDX (Sparc z kolei nie ma instrukcji dzielenia jako takiej)
- SPARC posiada instrukcje ADD/SUB Ri, Rj, Rk. Na i386 istnieje instrukcja, którą można wykorzystać do podobnego dodawania, ale odejmowania już nie (kto wie jaka to instrukcja?).
- itd itp

Przykład symulacji z opisami rejestrów i wartości

Zakładamy, że na końcu bloku d jest żywe.

Instrukcje	Kod	R0	R1	Opisy wartości
$t := a - b$	MOV a, R0	a	—	a w R0
	SUB b, R0	t	—	t w R0
$u := a - c$	MOV a, R1	t	a	t w R0, a w R1
	SUB c, R1	t	u	t w R0, u w R1
$v := t + u$	ADD R1, R0	v	u	u w R1, v w R0
$d := v + u$	ADD R1, R0	d	u	d w R0
	MOV R0, d	d	u	d w R0 i w pamięci

Generacja kodu dla bloku prostego

- 1 Wyznaczamy bloki proste
- 2 Określamy zmienne żywe na końcu bloku
- 3 Wyznaczamy *następne użycie* dla każdego argumentu i wyniku czwórki
- 4 Generujemy kod dla kolejnych czwórek, w biegu przydzielając im rejestry, odkładając zapis do pamięci, o ile się da
- 5 Na końcu bloku zapisujemy wszystkie żywe, a nie zapisane dotąd wartości.

Generacja kodu dla pojedynczej czwórki

Niech bieżącą czwórką będzie $A := B \text{ op } C$

- 1 Wybieramy instrukcję
- 2 Wybieramy rejestr L do przeprowadzenia operacji
- 3 Korzystając z opisów, badamy gdzie jest B i w razie potrzeby generujemy MOV B, L
- 4 Wybieramy C' – jedno z miejsc zawierających C (najlepiej rejestr)
- 5 Generujemy OP C', L
- 6 Poprawiamy opisy A,L (wartość A jest tylko w L)
- 7 Jeśli C nie jest żywe, to poprawiamy jego opis, zwalniając rejestr

Co zrobić, jeśli nie ma wolnego rejestru (spilling)

Strategia Bellady'ego (oryginalnie dla zwalniania stron pamięci wirtualnej)

- 1 Wybieramy rejestr, przechowujący wartość, której użycie leży najdalej w przyszłości
- 2 Odsyłamy do pamięci (spilling)
- 3 Jeśli rejestr przechowywał więcej niż jedną wartość, musimy odesłać wszystkie
- 4 Warto zatem wybierać najdłużej nieużywany rejestr spośród przechowujących najmniej (przeważnie 1) wartości.

Wiele różnych możliwości, pole do wielorakich optymalizacji i heurystyk.

Globalna alokacja rejestrów

- 1 Wydzielamy pewną pulę r rejestrów; w pewnym fragmencie programu wybrane wartości będziemy przechowywać na stałe w rejestrach.
- 2 Tworzymy graf kolizji: wierzchołkami są zmienne, jeśli przy definicja a , zmienna b jest żywa, to dodajemy krawędź (a, b)
- 3 Kolorujemy graf r kolorami (uwaga: NP-trudne)

Optymalizacja “przez dziurkę od klucza” (peep-hole)

- Definiujemy zbiór wzorców krótkich sekwencji kodu, które łatwo ulepszyć, np. w sekwencji

```
MOV Ri, a
```

```
MOV a, Ri
```

druga instrukcja jest zbędna.

- Przesuwamy się wzdłuż wygenerowanego kodu małym “okienkiem” (zwykle 2–3 instrukcje), jeśli kod w okienku pasuje do któregoś z wzorców — ulepszymy.

Przykład

Dla instrukcji $x = x + 7$ może zostać wygenerowany kod

```
iload x  
bipush 7  
iadd  
istore x
```

(gdzie x — liczba odpowiednia dla położenia x)

Optymalizator może potem rozpoznać taki fragment kodu i zastąpić go

```
iinc x 7
```

Można unikać generowania takiego kodu, ale to niepotrzebnie komplikuje generator. Optymalizacja peephole jest szybka i prosta w implementacji (zwykle wyszukiwanie wzorców).

Optymalizacje niezależne od maszyny docelowej

- Zwijanie stałych
- Eliminacja wspólnych podwyrażeń
- Eliminacja martwego kodu
- Optymalizacja pętli

Zwijanie stałych

Jeśli na zmienną przypisywana jest stała, możemy wszystkie użycia tej zmiennej w zasięgu definicji zastąpić wystąpieniami tej stałej, ewentualnie obliczając wyrażenia w czasie kompilacji.

Na przykład sekwencję

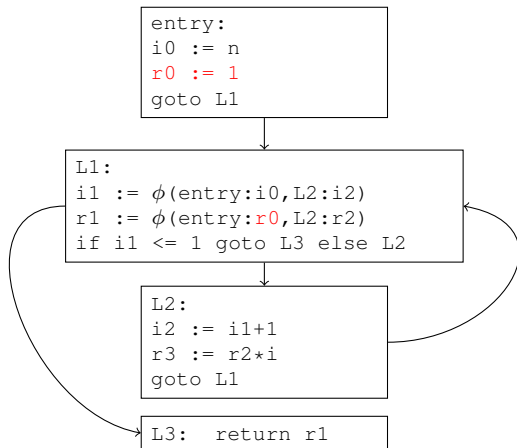
```
t1 := 7
t2 := t1 - 1
t3 := t2 * t2
a := b + t3
```

Możemy zastąpić przez

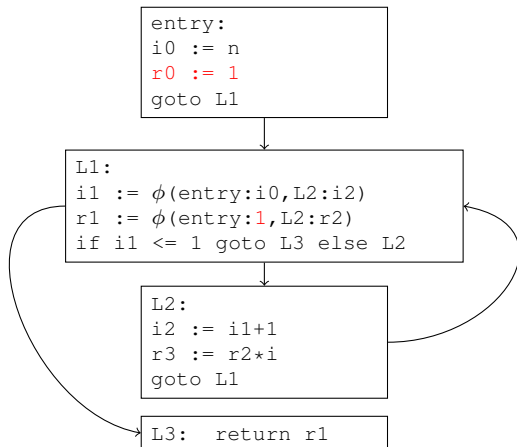
```
a := b + 36
```

NB dla maszyny stosowej można to zrobić na etapie peephole.

Zwijanie stałych

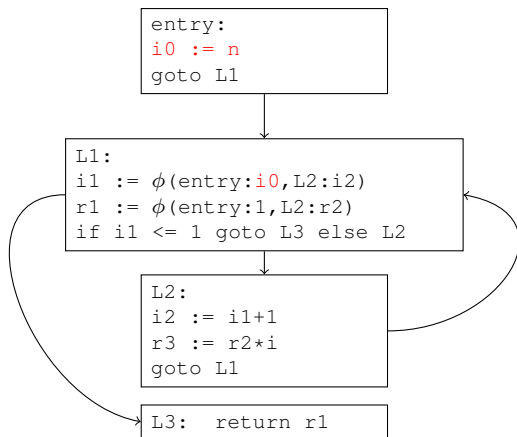


Zwijanie stałych

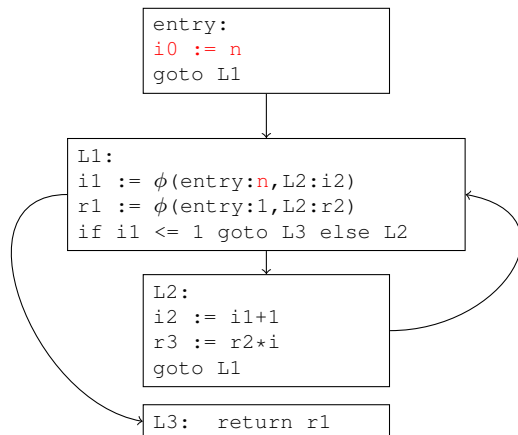


Propagacja kopii

Popodobnie jeśli występuje kopiowanie $x = y$ wszystkie użycia x do których dociera ta definicja można zastąpić przez y (SSA pomaga)



Propagacja kopii



Kod dla LLVM

Stąd wziął się kod dla LLVM z poprzedniego wykładu:

```
define i32 @fact(i32 %n) {  
entry: br label %L1
```

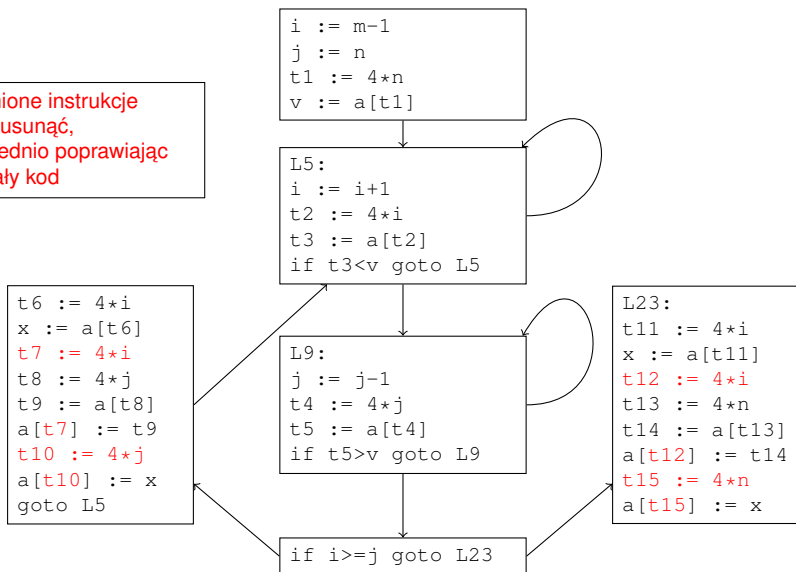
```
L1:  
    %i.1 = phi i32 [%n, %entry], [%i.2, %L2]  
    %r.1 = phi i32 [1, %entry], [%r.2, %L2]  
    %c0 = icmp sle i32 %i.1, 1  
    br i1 %c0, label %L3, label %L2
```

```
L2:  
    %r.2 = mul i32 %r.1, %i.1  
    %i.2 = sub i32 %i.1, 1  
    br label %L1
```

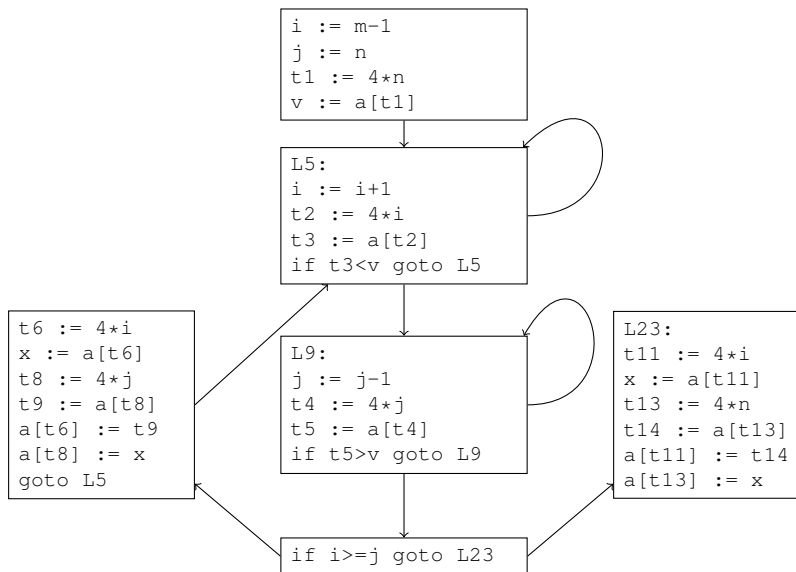
```
L3:  
    ret i32 %r.1  
}
```

Wspólne podwyrażenia lokalnie

Wyróżnione instrukcje
można usunąć,
odpowiednio poprawiając
pozostały kod

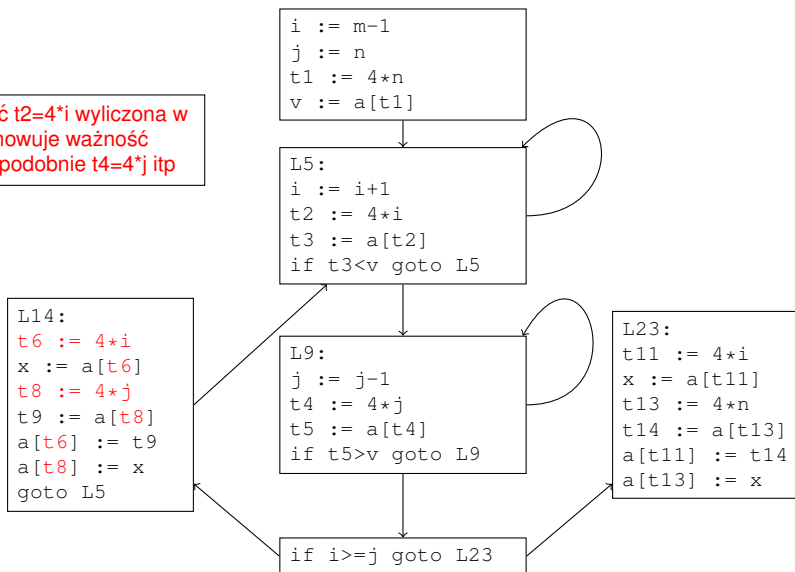


Wspólne podwyrażenia lokalnie

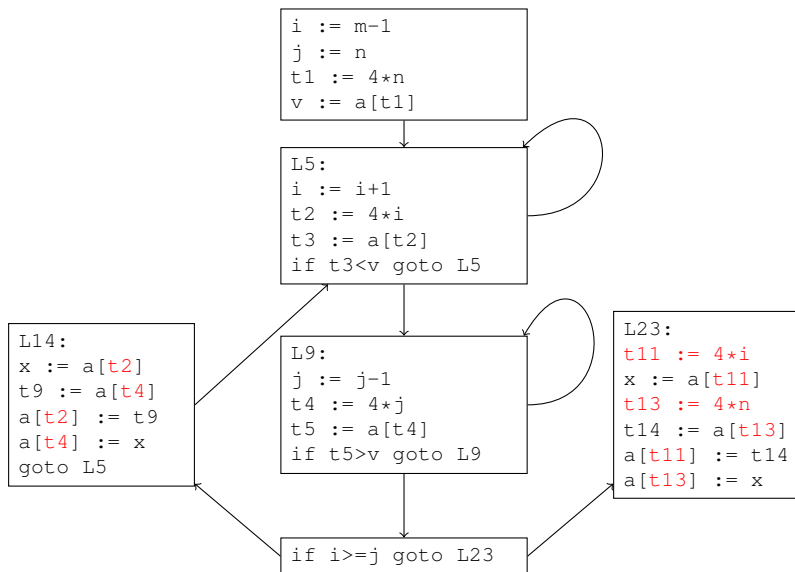


Wspólne podwyrażenia globalnie

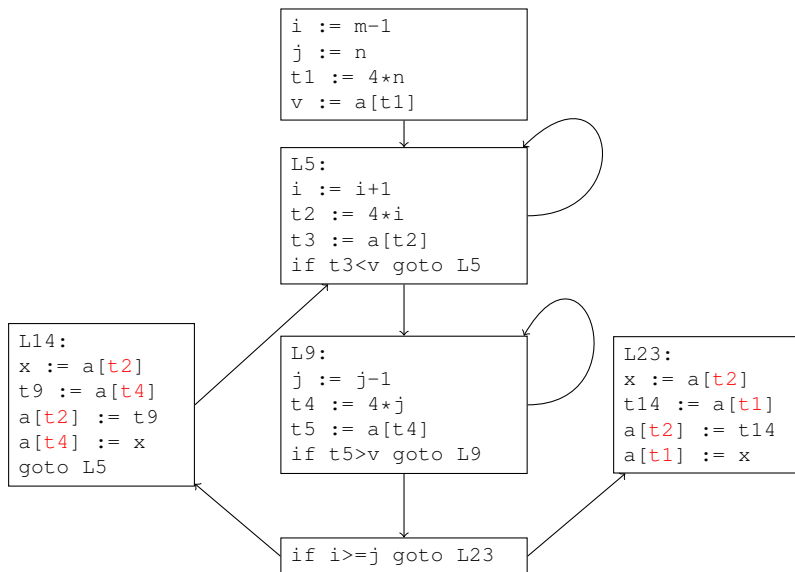
Wartość $t2=4*i$ wyliczona w L5 zachowuje ważność w L14; podobnie $t4=4*j$ itp



Wspólne podwyrażenia globalnie 2

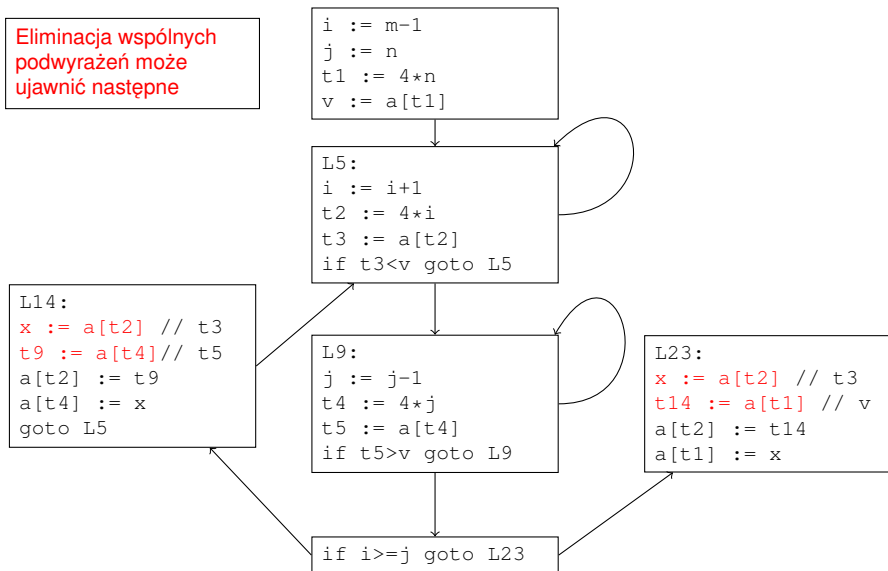


Wspólne podwyrażenia globalnie 2



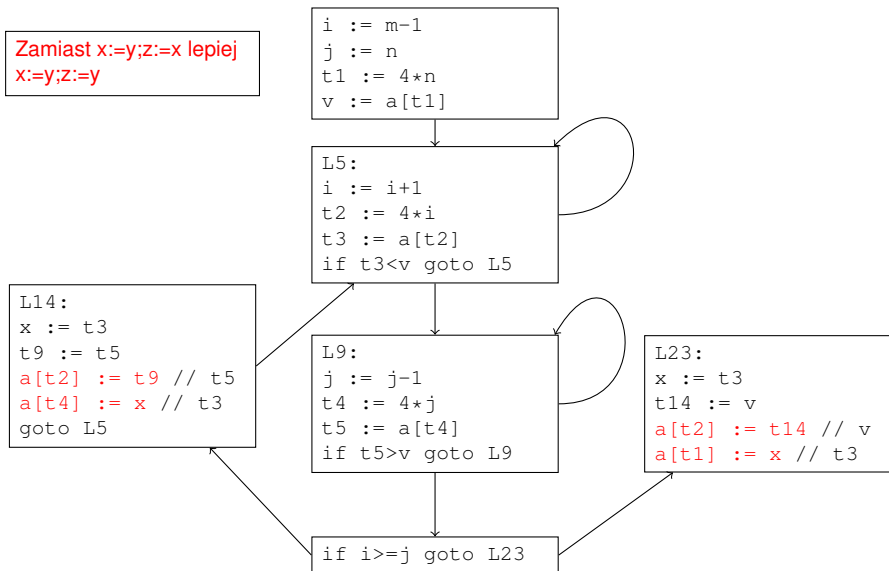
Wspólne podwyrażenia globalnie 3

Eliminacja wspólnych podwyrażeń może ujawnić następane



Propagacja kopii

Zamiast `x:=y;z:=x` lepiej
`x:=y;z:=y`



Eliminacja martwego kodu

x,t9,t14 są martwe,
możemy usunąć
przypisania na nie

```
i := m-1  
j := n  
t1 := 4*n  
v := a[t1]
```

```
L5:  
i := i+1  
t2 := 4*i  
t3 := a[t2]  
if t3 < v goto L5
```

```
L9:  
j := j-1  
t4 := 4*j  
t5 := a[t4]  
if t5 > v goto L9
```

```
if i >= j goto L23
```

```
L14:  
x := t3  
t9 := t5  
a[t2] := t5  
a[t4] := t3  
goto L5
```

```
L23:  
x := t3  
t14 := v  
a[t2] := v  
a[t1] := t3
```

Wysunięcie kodu przed pętlę

Obliczenia wyrażeń, które nie zmieniają swej wartości w trakcie pętli możemy wysunąć przed pętlę.

```
while (i<=n-3) {  
    s += a[i];  
    i++;  
}
```

Możemy zastąpić przez

```
t = n-3;  
while (i<=t) {  
    s += a[i];  
    i++;  
}
```

Redukcja mocy i zmienne indukcyjne

- Redukcja mocy (strength reduction) polega na zamianie droższej operacji (np. mnożenie) przez tańszą (np. dodawanie).
- Jest to możliwe i pożyteczne w stosunku do tzw. *zmiennych indukcyjnych*, czyli takich które są zwiększane (ew. zmniejszane) o stałą (zwykle 1) za każdym obrotem pętli.

Przykład

Zamiast

```
    i := 0;
    goto L2
L1: i := i+1
    t2 := 4*i
    t3 := a[t2]
    s := s + t3
L2: if(a[t2]<=k) goto L2
```

Można

```
    t2 := 0;
    goto L2
L1: t2 := t2 + 4
    t3 := a[t2]
    s := s + t3
L2: if(a[t2]<=k) goto L2
```

Redukcja mocy i zmienne indukcyjne

Niezmiennik pętli:

$t2 = 4 * i$

$t4 = 4 * j$

Dodajemy **zielone**
instrukcje i **usuwamy**
czerwone

```
i := m-1
j := n
t1 := 4*n
v := a[t1]
t2 := 4*i
t4 := 4*n
```

```
L5:
i := i+1
t2 := t2+4
t3 := a[t2]
if t3 < v goto L5
```

```
L9:
j := j-1
t4 := t4-4
t5 := a[t4]
if t5 > v goto L9
```

```
L14:
a[t2] := t5
a[t4] := t3
goto L5
```

```
L23:
a[t2] := v
a[t1] := t3
```

```
if t2 >= t4 goto L23
```

Konkluzja

- Zaczynaliśmy od 30 czwórek, po optymalizacjach 20 i to tańszych.
- Po generacji kodu maszynowego możemy jeszcze wykonać peephole.
- Uzyskujemy mniejszy i szybszy kod.
- Cena: większy i dłużej działający kompilator.
- Łatwo popełnić trudny do wykrycia błąd.

Wywołania końcowe (tail calls)

```
int factorial(int n) {
    return _factorial(n, 1);
}
int _factorial(int n, int result) {
    if (n <= 0)
        return result;
    else
        return _factorial(n - 1, n * result);
}
```

Jeśli ostatnią instrukcją jest wywołanie funkcji, możemy je zastąpić skokiem.

Jeśli skok jest do tej samej funkcji (nie musi być!), jest to tzw. rekursja ogonowa.

gcc -O1

`_factorial:`

```
    pushl    %ebp
    movl    %esp, %ebp
    subl    $8, %esp

    movl    8(%ebp), %edx ; edx = n
    movl    12(%ebp), %eax ; eax = result
    testl   %edx, %edx
    jle    .L2           ; if n <= 0
    imull   %edx, %eax   ; eax = n * result
    movl    %eax, 4(%esp) ; na stos
    leal   -1(%edx), %eax ; eax = n-1
    movl    %eax, (%esp) ; na stos
    call   _factorial
```

`.L2:`

```
    leave   ; przywroc wskaznik ramki
    ret
```


gcc -O1 -foptimize-sibling-calls

```
    pushl    %ebp
    movl    %esp, %ebp

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    testl   %edx, %edx
    jle    .L3

.L6:
    imull   %edx, %eax
    subl   $1, %edx
    jne    .L6

.L3:
    popl    %ebp
    ret
```

Jeszcze jeden przykład wywołań końcowych

```
int even(int n)
{
    if(!n) return 1; else return odd(n-1);
}
```

```
int odd(int n)
{
    if(n==1) return 1; else return even(n-1);
}
```

W tym wypadku mamy do czynienia z wywołaniami końcowymi, które trudno zoptymalizować na JVM.

gcc -O1

even:

```
    pushl    %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    8(%ebp), %edx
    movl    $1, %eax
    testl   %edx, %edx
    je     .L9
    leal   -1(%edx), %eax
    movl   %eax, (%esp)
    call  odd
```

.L9:

```
    leave
    ret
```

gcc -O1 -foptimize-sibling-calls

even:

```
pushl    %ebp
movl     %esp, %ebp
movl     8(%ebp), %eax
testl   %eax, %eax
je       .L12
subl     $1, %eax
movl     %eax, 8(%ebp)
popl     %ebp
jmp      odd
```

.L12:

```
movl     $1, %eax
popl     %ebp
ret
```

■ ■ ■

-falign-functions[=n] -falign-jumps[=n] -falign-labels[=n] -falign-loops[=n]
-fassociative-math -fauto-inc-dec -fbranch-probabilities -fbranch-target-load-optimize
-fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcheck-data-deps
-fcprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules
-fcx-limited-range -fdata-sections -fdce -fdce -fdelayed-branch -fearly-inlining
-fdelete-null-pointer-checks -fdse -fexpensive-optimizations -ffast-math
-ffinite-math-only -ffloat-store -fforward-propagate -ffunction-sections
-fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fif-conversion -fif-conversion2
-finline-functions -finline-functions-called-once -finline-limit=n -finline-small-functions
-fipa-cp -fipa-marix-reorg -fipa-pta -fipa-pure-const -fipa-reference -fipa-struct-reorg
-fipa-type-escape -fivopts -fkeep-inline-functions -fkeep-static-consts
-fmerge-all-constants -fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves
-fmove-loop-invariants -fmudflap -fmudflapir -fmudflapth -fno-branch-count-reg
-fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls
-fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -freciprocal-math
-fregmove -frename-registers -freorder-blocks -freorder-blocks-and-partition
-freorder-functions -frerun-cse-after-loop -freschedule-modulo-scheduled-loops
-frounding-math -frtl-abstract-sequences -fsched2-use-superblocks -fsched2-use-traces
-fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns-dep[=n]
-fsched-stalled-insns[=n] -fschedule-insns -fschedule-insns2 -fsection-anchors -fsee
-fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller
-fsplit-wide-types -fstack-protector -fstack-protector-all
-fstrict-aliasing -fstrict-overflow -fthread-jumps -ftracer -ftree-ccp
-ftree-ch -ftree-copy-prop -ftree-copyrename -ftree-dce
-ftree-dominator-opts -ftree-dse -ftree-fre -ftree-loop-im -ftree-loop-distribution
-ftree-loop-ivcanon -ftree-loop-linear -ftree-loop-optimize
-ftree-parallelize-loops=n -ftree-pre -ftree-reassoc -ftree-sink -ftree-sra
-ftree-store-ccp -ftree-ter -ftree-vect-loop-version -ftree-vectorize -ftree-vrp
-funit-at-a-time -funroll-all-loops -funroll-loops -funsafe-loop-optimizations
-funsafe-math-optimizations -funswitch-loops -fvariable-expansion-in-unroller
-fvect-cost-model -fvpt -fweb -fwhole-program
-O -O0 -O1 -O2 -O3 -Os