Metody Realizacji Języków Programowania

Generacja kodu pośredniego

Marcin Benke

MIM UW

5 listopada 2018

Kod pośredni?

- Przydatny poziom abstrakcji przy generacji kodu maszynowego.
- Nie jest niezbędny — zwłaszcza przy generacji kodu na maszynę stosową.
- Przy generacji kodu na różne architektury wspólne transformacje niezależne od architektury.
- Ułatwia niektóre optymalizacje.
- Różne postaci — tu zajmiemy się kodem czwórkowym.

Kod czwórkowy

Czwórka:

$$w := a_1 \oplus a_2$$

- argumenty a_1, a_2
- operacja $\oplus$
- lokalizacja wyniku w

Zwany również kodem trójadresowym, większość instrukcji zawiera bowiem trzy adresy: wyniku i dwu argumentów.

Instrukcje

- Przypisanie postaci `x := y op z`, gdzie `op` to jeden z operatorów `+, -, *, /, and, or, xor`.
- Przypisanie jednoargumentowe postaci `x := op y`, gdzie op to jeden z `-, not`.
- Kopiowanie postaci `x := y`.
- Skoki bezwarunkowe postaci `goto L`, gdzie L to adres w kodzie zazwyczaj reprezentowany przez etykietę;
- przed każdą instrukcją może wystąpić etykieta odnosząca się do pierwszego adresu występującego po niej.
- Skoki warunkowe postaci `if x oprel y goto L`, gdzie oprel to operator relacyjny (`<, >, =, <=, >=, !=`).

Instrukcje c.d.

- Wywołanie funkcji: `param x` i `t := call L, n` — przekazanie x jako kolejnego parametru, oraz wywołanie funkcji pod adresem L z n parametrami, np

```
param x
param y
call f, 2
```

- Powrót z funkcji: `return x`.
- Przypisania indeksowane: `x := y[i]` oraz `x[i] := y`.

Stan podczas generacji kodu

Generacja kodu wymaga przechowywania pewnych informacji. Obejmuje to co najmniej

- kolejny wolny numer dla etykiet i zmiennych/rejestrów
- definicje globalnych nazw dla literałów napisowych
- mapowanie nazw zmiennych na ich adresy
- informacje o typach funkcji

W dalszym ciągu wykładu będziemy się odwoływać do stanu przy pomocy operacji takich jak

- `freshTemp`
- `freshLabel`
- `addLocal name`
- `getLocal`

Generacja kodu dla wyrażeń arytmetycznych

$$e_1 \; op \; e_2$$

Przyjmijmy konwencję, że funkcja tworząca kod dla podwyrażenia daje nazwę zmiennej, na której zapamiętany jest wynik.

```
genBinOp op e1 e2 = do
  a1 <- genExp e1
  a2 <- genExp e2
  t <- freshTemp              -- nowa zmienna
  emitBinOp t a1 op a2        -- t := a1 op a2
  return t
```

Przykład

Dla wyrażenia

$$(a * a - c) + b * b + d$$

wygenerujemy kod

```
t1 := a*a
t2 := t1 - c  // t2 = a*a - c
t3 := b*b
t4 := t2 + t3 // t4 = a*a - c + b*b
t5 := t4 + d
```

Kod czwórkowy a LLVM

LLVM jest nowoczesnym wariantem kodu czwórkowego, przy czym

- Niektóre instrukcje mogą mieć więcej argumentów
- prostsze wywołania: `f(a1,...,an)`
- wyraźnie zaznaczone dostępy do pamięci
- dostęp do elementów tablic wymaga uprzedniego obliczenia adresu

```
%x = getelementptr i8* %a, i32 6
%y = load i8* x
```

- skoki warunkowe z dwoma celami
- pewne ograniczenia
  - nie ma kopiowania: `x := y` i `x := c` zabronione
  - postać SSA

Generacja kodu dla wyrażeń arytmetycznych — LLVM

Przykład generacji kodu LLVM (fragment):

```
genExp,genBinOp :: Exp -> GenM Address
genExp (ExpInt i)  = return $ Immediate i
genExp (ExpAdd e1 e2) = genBinOp "add" e1 e2
genExp (ExpVar (Ident s)) = do
  a <- getAddr s
  t <- freshTemp
  emit $ load t a
  return t
...
genBinOp op e1 e2 = do
  a1 <- genExp e1
  a2 <- genExp e2
  t <- freshTemp
  emit $ iBinOp t op a1 a2
  return t
```

Przykład

Dla wyrażenia

$$(a * a - c) + b * b + d$$

wygenerujemy kod

```
%i2=load i32* %loc1
%i3=load i32* %loc1
%i4=mul  i32 %i2,%i3
%i6=load i32* %loc5
%i7=sub  i32 %i4,%i6
%i9=load i32* %loc8
%i10=load i32* %loc8
%i11=mul  i32 %i9,%i10
%i12=add  i32 %i7,%i11
%i14=load i32* %loc13
%i15=add  i32 %i12,%i14
```

Widać tu pewne nieefektywności — później pomyślimy jak się ich pozbyć.

Kolejność obliczeń

- Rozważmy wyrażenie $e_1 + e_2$, przy czym obliczenie e_1 wymaga k zmiennych tymczasowych (rejestrów), zaś e_2 — n zmiennych (załóżmy, że $n > k$).
- Jeśli możemy obliczać e_1 i e_2 w dowolnym porządku, to które lepiej obliczyć najpierw, aby zużyć jak najmniej zmiennych tymczasowych?

Kolejność obliczeń

- Rozważmy wyrażenie $e_1 + e_2$, przy czym obliczenie e_1 wymaga k zmiennych tymczasowych (rejestrów), zaś e_2 — n zmiennych (załóżmy, że $n > k$).
- Jeśli możemy obliczać e_1 i e_2 w dowolnym porządku, to które lepiej obliczyć najpierw, aby zużyć jak najmniej zmiennych tymczasowych?
  1. "najpierw łatwiejsze": k rejestrów dla obliczenia e_1, potem 1 przechowujący jego wartość plus n dla obliczenia e_2

$$\max(k, 1 + n) = 1 + n$$

Kolejność obliczeń

- Rozważmy wyrażenie $e_1 + e_2$, przy czym obliczenie e_1 wymaga k zmiennych tymczasowych (rejestrów), zaś e_2 — n zmiennych (załóżmy, że $n > k$).
- Jeśli możemy obliczać e_1 i e_2 w dowolnym porządku, to które lepiej obliczyć najpierw, aby zużyć jak najmniej zmiennych tymczasowych?
  1. "najpierw łatwiejsze": k rejestrów dla obliczenia e_1, potem 1 przechowujący jego wartość plus n dla obliczenia e_2

$$\max(k, 1 + n) = 1 + n$$

  2. "najpierw trudniejsze"

$$\max(n, 1 + k) = n$$

- Kolejność wyliczenia może zadecydować, czy uda nam się obliczyć wyrażenie tylko przy użyciu rejestrów (bez odsyłania wyników pośrednich do pamięci).

Wywołanie funkcji

$$f(e_1, \ldots, e_n)$$

```
genExp (ECall f es) = do
  vs <- mapM genExp es
{- czyli
  v1 <- genExp(e1)
  ...
  vn <- genExp(en) -}
  r <- freshTemp
  mapM_ emitParam vs
{- czyli
    param v1
    ...
    param vn -}
  emitCall f $ length es       --  r := call f, n
  return r
```

Uwaga: czasem lepiej argumenty podawać od ostatniego.

Generacja kodu na maszynę stosową

Kod dla wyrażeń:

$$e_1 \; op \; e_2$$

```
kod e1
kod e2
op
```

$$f(e_1, \ldots, e_n)$$

```
kod e1
...
kod en
call f
```

(zakładając, że instrukcja `call` — jak w JVM — pobiera argumenty ze stosu).

Przykład

Dla wyrażenia

$$(a * a - c) + b * b + d$$

wygenerujemy kod

```
iload_2
iload_2
imul
iload_3
isub
iload 4
iload 4
imul
iadd
iload 5
iadd
```

(przy założeniu, że zmienne a–d są pod adresami 2–5)

Przypisanie

$x := e$

Gdy przypisanie jest instrukcją

```
genStmt (SAssign (Ident s) e) = do
  lhs <- getLocal s
  rhs <- genExp e
  emit $ store rhs lhs
```

Na przykład dla przypisania `j=i+1` można wygenerować kod LLVM

```
  %t1 = load i32* loc_i
  %t2 = add i32 %t1, 1
  store i32 %t2, i32* %loc_j
```

Gdy przypisanie jest wyrażeniem

"Typem wyrażenia przypisania jest typ lewego argumentu, wartością zaś [. . .] wartość umieszczona w tym argumencie.
[Kernighan, Ritchie s. 213 (WNT 1988)]

```
genExp (EAssign (Ident s) e) = do
  lhs <- getLocal s
  rhs <- genExp e
  emit $ store rhs lhs
  return rhs
```

Na przykład dla instrukcji `return j=i+1` można wygenerować kod

```
%t1 = load i32* loc_i
%t2 = add i32 %t1, 1
store i32 %t2, i32* %loc_j
ret i32 %t2
```

Dygresja — zagadka

Rozważmy program

```
int main() {
  int i,j;
  (i=j=0)=42;
  printf("i=%d j=%d\n",i,j);
  return 0;
}
```

Czy jest on poprawny w C? A w C++? Co zostanie wypisane?

Rozwiązanie

```
  int i,j;
  (i=j=0)=42;
  printf("i=%d j=%d\n",i,j);
```

```
$ make assign-rval
cc     assign-rval.c   -o assign-rval
assign-rval.c:4:10: error:
  lvalue required as left operand of assignment
```

W C wynik przypisania jest r-wartością...

```
$ make assign-lval
g++     assign-lval.cc   -o assign-lval
$ ./assign-lval
i=42 j=0
```

...natomiast w C++ jest l-wartością.

Gdy przypisanie jest l-wartością

Wynikiem operacji przypisania [. . .] jest wartość umieszczona w lewym argumencie; wynik jest l-wartością.
[Stroustrup, s. 559 (WNT 1995)]

```
genLhs (EAssign e1 e2) = do
  lhs <- genLhs e1
  rhs <- genRhs e2
  emit $ store rhs lhs
  return lhs
```

(funkcja `genRhs` odpowiada dawnemu `genExp`)

Na przykład dla przypisania `(i=j=0)=42` możemy wygenerować

```
store i32 0, i32* %loc_j
store i32 0, i32* %loc_i
store i32 42, i32* %loc_i
```

Przypisanie dla JVM

$x := e$

```
kod e
istore adres
```

Jeśli przypisanie jest wyrażeniem

```
kod e
dup
istore adres
```

Instrukcja wyrażenia e;

```
kod e
pop
```

Oczywiście zbędne “`dup`” oraz “`pop`” mozna zoptymalizować.

Pętla while

while(*warunek*) instrukcja

Można wygenerować kod następujący:

```
L1: kod warunku, wynik w t
    if not t goto L2
    kod instrukcji
    goto L1
L2: ...
```

Można też trochę inaczej:

```
    goto L2
L1: kod instrukcji
L2: kod warunku, wynik w t
    if t goto L1
```

W pierwszym wariancie na n obrotów petli wykonujemy $2n + 1$ skoków, w drugim — $n + 2$.

Instrukcja warunkowa

if(*warunek*) instrukcja$_1$ **else** instrukcja$_2$

Można wygenerować kod następujący:

```
        kod warunku, wynik w t
        if not t goto Lfalse
Ltrue:  kod instrukcji1
        goto Lend
Lfalse: kod instrukcji2
Lend:   ...
```

lub

```
        kod warunku, wynik w t
        if t goto Ltrue
Lfalse: kod instrukcji2
        goto Lend
Ltrue: kod instrukcji1
Lend:   ...
```

Skrócone tłumaczenie wyrażeń logicznych

Wyrażenia logiczne można tłumaczyć albo tak jak wyrażenia arytmetyczne, albo przy użyciu tzw. *kodu skaczącego*

$w1 \&\& w2$

```
if not w1 goto Lfalse
if not w2 goto Lfalse
kod Ltrue lub goto Ltrue
```

$w1 || w2$

```
if w1 goto Ltrue
if w2 goto Ltrue
kod Lfalse lub goto Lfalse
```

Przykład

```
if (i>=0) && (i<n) then I1 else I2
```

Można przetłumaczyć jako

```
        if i<0 goto Lfalse
        if i>=n goto Lfalse
Ltrue:  I1
        goto Lend
Lfalse: I2
Lend:   ...
```

Generacja kodu skaczącego dla JVM (fragment)

```
genCond (CGt e1 e2) lThen lElse = do
  genExp e1
  genExp e2
  emit $ Jif_icmpgt lThen
  emit $ Jgoto lElse
genCond (CAnd c1 c2) lTrue lFalse = do
  lMid <- freshLabel
  genCond c1 lMid lFalse
  emit $ placeLabel lMid
  genCond c2 lTrue lFalse
genCond (COr c1 c2) lTrue lFalse = do
  lMid <- freshLabel
  genCond c1 lTrue lMid
  emit $ placeLabel lMid
  genCond c2 lTrue lFalse
genCond (CNot c) lTrue lFalse =
  genCond c lFalse lTrue
```

Przykład — JVM

```
if((a>0&&b>0)||(a<0&&b<0)) /* L0 */ return 9;
  else /* L1 */ return 1; /* L2 */

genCode ((a>0&&b>0)||(a<0&&b<0)) L0 L1 =>
-- lMid=L3
  genCond (a>0&&b>0) L0 L3;   placeLabel L3
  genCond (a<0&&b<0) L0 L1

genCond (a>0&&b>0) L0 L3 =>  -- lMid = L4
  genCond (a>0) L4 L3 ; placeLabel L4
  genCond (b>0) L0 L3

genCond (a>0) L4 L3 =>
  iload a
  ifgt L4
  goto L3
```

Przykład — JVM

```
if((a>0&&b>0)||(a<0&&b<0)) return 9;
  else return 1;
```

```
 start:
   iload_2  ; a
   ifgt L4
   goto L3
 L4:
   iload_3  ; b
   ifgt L0
   goto L3
 L3:
   iload_2
   iflt L5
   goto L1
```

```
L5:
  iload_3
  iflt L0
  goto L1
L0:
  bipush 9
  ireturn
  goto L2
L1:
  iconst_1
  ireturn
L2: ...
```

Przykład — JVM

```
if((a>0&&b>0)||(a<0&&b<0)) return 9;
  else return 1;
```

Dalej jest przestrzeń dla ulepszeń, np.

```
    iload_2
    ifle L3 ; pierwszy and falszywy - sprawdz drugi
L4: iload_3
    ifgt L0 ; caly warunek prawdziwy
L3: iload_2
    ifge L1 ; caly warunek falszywy
L5: iload_3
    ifge L1 ; caly warunek falszywy
L0: bipush 9
    ireturn
L1:
    iconst_1
    ireturn
```

Generacja kodu skaczącego dla JVM (fragment)

Można to uzyskać pamiętając, która etykieta jest następna i sprytniej generując skoki, np.

```
genCond2 (CGt e1 (ExpInt 0)) lThen lElse lNext = do
  genExp e1
  genCondJumps lThen lElse lNext Jifgt Jifle

genCondJumps lThen lElse lNext posJump negJump
      | lNext == lThen = do
        emit $ negJump lElse
      | lNext == lElse = do
        emit $ posJump lThen
      | otherwise = do
        emit $ posJump lThen
        emit $ Jgoto lElse
```

Skoki po “return” można łatwo usunąć na etapie optymalizacji.

Postać SSA (Static Single Assignment)

Analizy i przekształcenia kodu 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 (może natomiast wykonać się wiele razy np. w pętli)

LLVM wymaga kodu w postaci SSA

Bloki proste

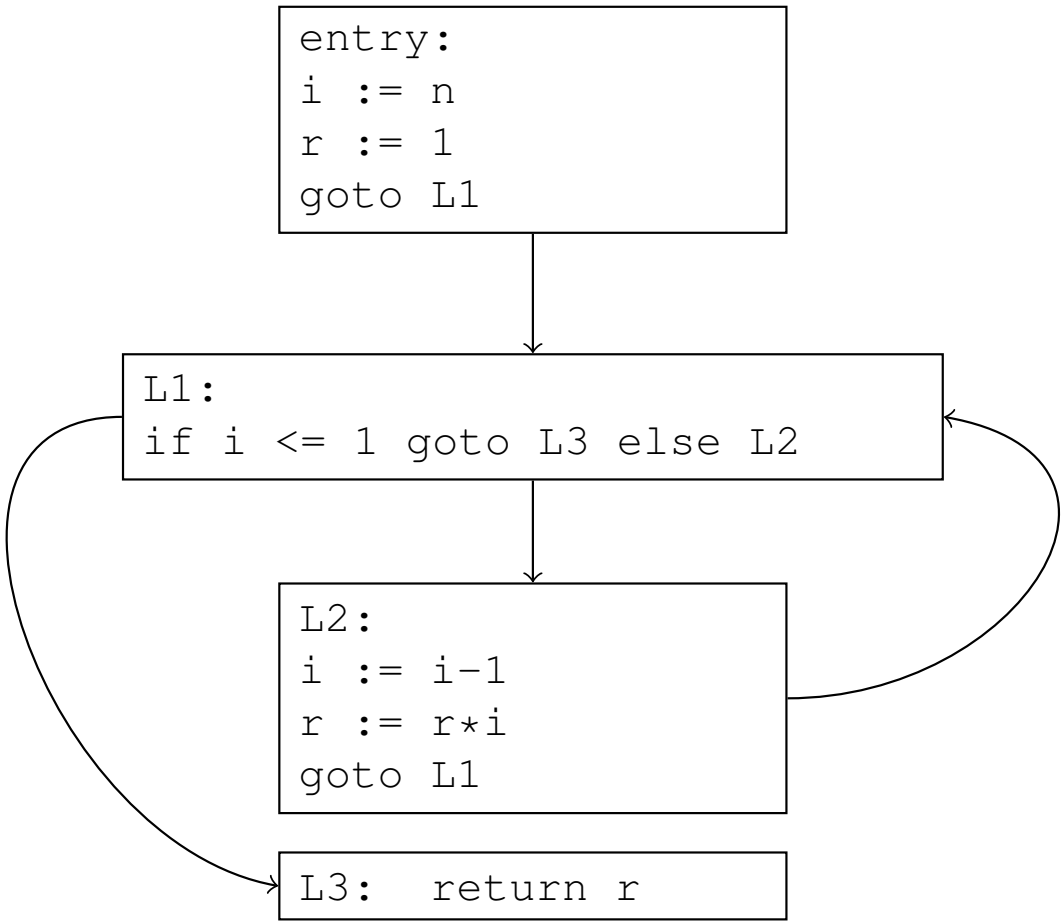
Blok prosty jest sekwencją instrukcji, do której sterowanie wchodzi wyłącznie na początku i z którego wychodzi wyłącznie na końcu, bez możliwości zatrzymania ani rozgałęzienia wewnątrz.

Przykład (abstrakcyjny kod pośredni):

```
E:   i := n
     r := 1
     goto L1

L1:  if i <= 1 goto L3 else L2

L2:  i := i-1
     r := r*i
     goto L1

L3:  return r
```

Graf przepływu sterowania (Control Flow Graph)

Wierzchołkami są bloki proste, krawędziami możliwe przejścia:



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.

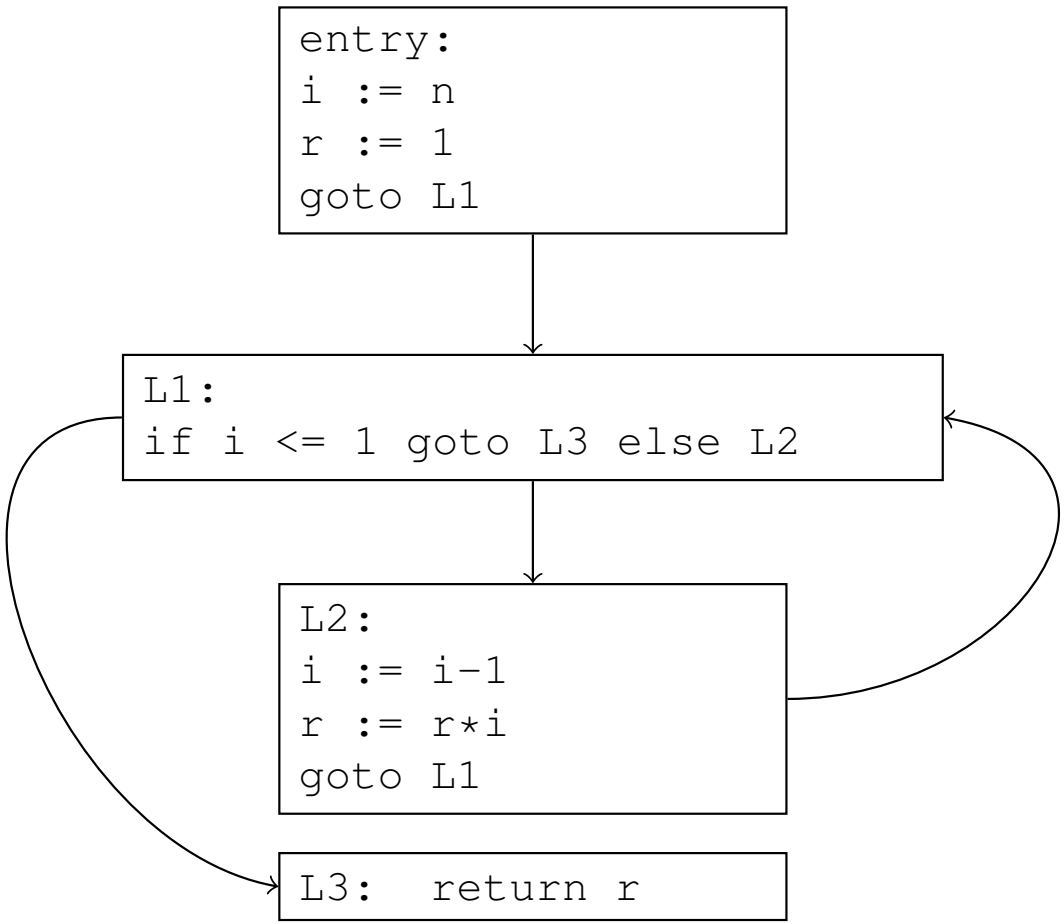
```
i := n
r := 1
r := r * i
i := i - 1
return r
```

Zastępujemy przez

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

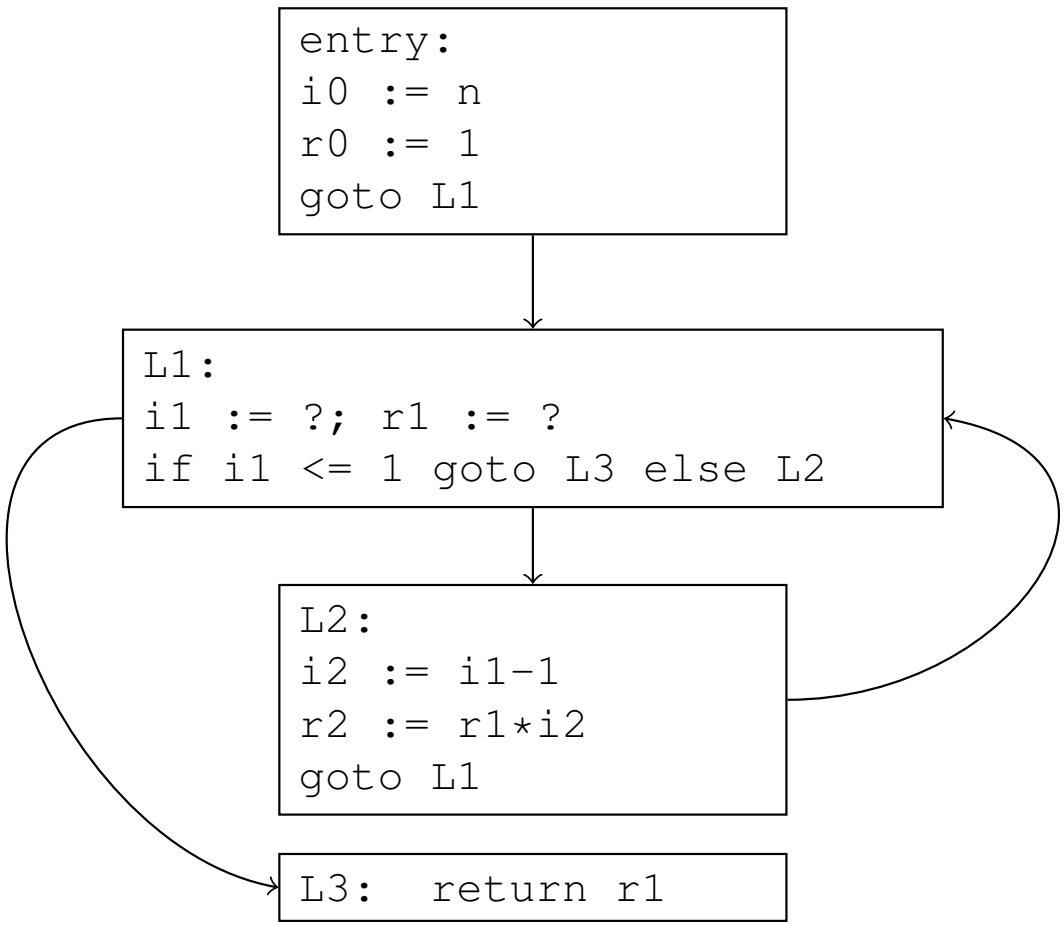
Przekształcanie do postaci SSA — graf sterowania

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



Przekształcanie do postaci SSA — graf sterowania

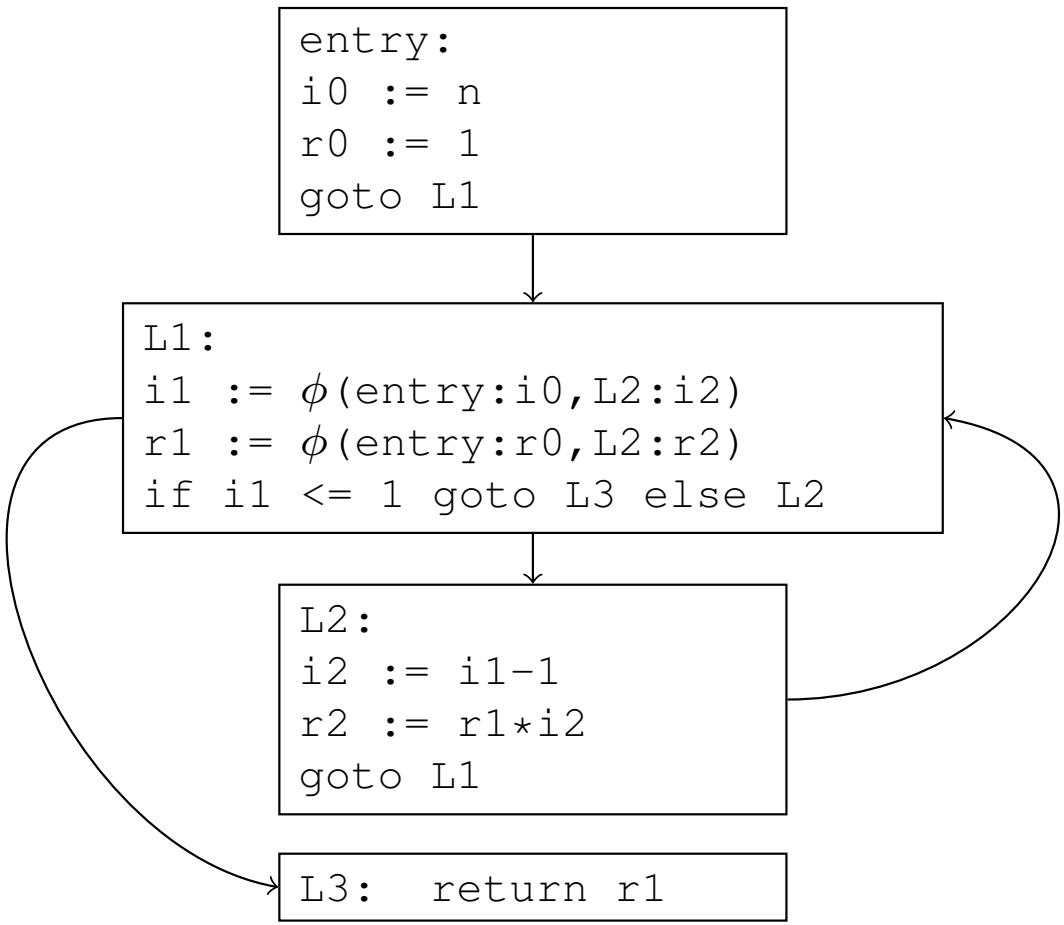
Ponumerowanie zmiennych spowoduje problem



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:



Kod dla LLVM

Eliminując przypisania postaci $a := b$, otrzymamy 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
}
```

Eliminacja funkcji ϕ

Podejście klasyczne: eliminacja ϕ przed alokacją rejestrów

Zaletą tego podejścia jest jego prostota, powszechnie stosowane.

Podejście alternatywne: eliminacja ϕ po alokacji rejestrów

Bardziej skomplikowane, ale umożliwia łatwiejszą alokację rejestrów dla kodu w postaci SSA. Jak dotąd rzadko używane.

Eliminacja funkcji ϕ — podejście klasyczne

Eliminacja ϕ przed alokacją rejestrów

Uzycie ϕ na początku bloku B

$$x = \phi(B_1 : a_1, B_2 : a_2)$$

zastępujemy przez odpowiednie przypisania na końcu poprzedników B: $x = a_1$ na końcu B_1, $x = a_2$ na końcu B_2,

Eliminacja funkcji ϕ — podejście alternatywne

Eliminacja ϕ po alokacji rejestrów

$$\begin{aligned} R_1 &= \phi(\quad \ldots, \quad B_i : R_{i1}, \quad \ldots) \\ &\ldots \\ R_n &= \phi(\quad \ldots, \quad B_i : R_{in}, \quad \ldots) \end{aligned}$$

na końcu bloku B_i wstawiamy kod realizujący permutację

$$(R_1, \ldots, R_n) = (R_{i1}, \ldots, R_{in})$$

Może się tu przydać instrukcja XCHG Ri, Rj

Warto generować kod taki, aby permutacje były jak najmniejsze (niestety, NP-trudne).

Nowoczesny algorytm transformacji do SSA

[Braun, Buchwald, Hack, Leißa, Mallon, Zwinkau 2013]
Dla bloku prostego: budujemy mapowanie zmienne $\mapsto$ wartości, np.

```
a := 42
b := a
a := a + b
```

$v_0 : 42$

$a \mapsto v_0$

Nowoczesny algorytm transformacji do SSA

[Braun, Buchwald, Hack, Leißa, Mallon, Zwinkau 2013]
Dla bloku prostego: budujemy mapowanie zmienne $\mapsto$ wartości, np.

`a := 42`	$v_0 : 42$	$a \mapsto v_0$
`b := a`		$b \mapsto v_0$
`a := a + b`		

Nowoczesny algorytm transformacji do SSA

[Braun, Buchwald, Hack, Leißa, Mallon, Zwinkau 2013]
Dla bloku prostego: budujemy mapowanie zmienne $\mapsto$ wartości, np.

`a := 42`	$v_0 : 42$	$a \mapsto v_0$
`b := a`		$b \mapsto v_0$
`a := a + b`	$v_1 : v_0 + v_0$	
`c := a + d`		

Nowoczesny algorytm transformacji do SSA

[Braun, Buchwald, Hack, Leißa, Mallon, Zwinkau 2013]
Dla bloku prostego: budujemy mapowanie zmienne $\mapsto$ wartości, np.

`a := 42`	$v_0 : 42$	
`b := a`		$b \mapsto v_0$
`a := a + b`	$v_1 : v_0 + v_0$	$a \mapsto v_1$
`c := a + d`		

Nowoczesny algorytm transformacji do SSA

[Braun, Buchwald, Hack, Leißa, Mallon, Zwinkau 2013]
Dla bloku prostego: budujemy mapowanie zmienne $\mapsto$ wartości, np.

`a := 42`	$v_0 : 42$	
`b := a`		$b \mapsto v_0$
`a := a + b`	$v_1 : v_0 + v_0$	$a \mapsto v_1$
`c := a + d`	$v_2 : v_1 + v_?$	$c \mapsto v_2$

Wartości niezdefiniowanych szukamy rekurencyjnie wśród poprzedników i dodajemy do ϕ; potem trywialne ϕ eliminujemy.

Nowoczesny algorytm transformacji do SSA

[Braun, Buchwald, Hack, Leißa, Mallon, Zwinkau 2013]
Dla bloku prostego: budujemy mapowanie zmienne $\mapsto$ wartości, np.

	$v_? : \phi()$	$d \mapsto v_?$
`a := 42`	$v_0 : 42$	
`b := a`		$b \mapsto v_0$
`a := a + b`	$v_1 : v_0 + v_0$	$a \mapsto v_1$
`c := a + d`	$v_2 : v_1 + v_?$	$c \mapsto v_2$

Wartości niezdefiniowanych szukamy rekurencyjnie wśród poprzedników i dodajemy do ϕ; potem trywialne ϕ eliminujemy.