AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Analiza przepływu

Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki

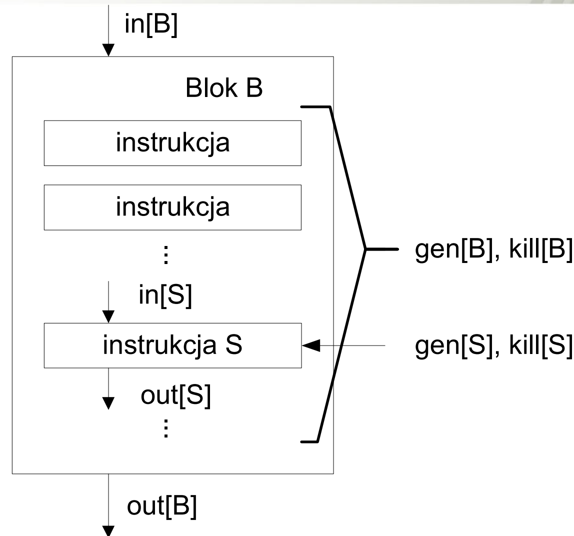
Równania przepływu

Globalna analiza przepływu danych zajmuje się zbieraniem informacji o pewnych aspektach działania programu i kojarzenia ich ze stosownymi miejscami (punktami) w grafie przepływu. Informacje są uzyskiwane w wyniku konstruowania i rozwiązywania tzw. równań przepływu danych. Typowe równanie ma postać:

$$out[S] = get[S] \cup (in[S] - kill[S])$$

co może być przeczytane jako: „informacja na końcu instrukcji *S* jest albo generowana wewnątrz tej instrukcji, albo jest informacją „przychodzącą” do instrukcji *S* i nie zabitą (zniszczoną) w trakcie wykonywania instrukcji *S*”. Informacja dotyczy instrukcji lub bloku bazowego.

in, out, gen i *kill*



Problem zasięgu definicji

Definicją zmiennej *x* będziemy nazywać taką instrukcję, która nadaje (lub może to zrobić) wartość zmiennej *x*.

Mamy definicje jednoznaczne:

- instrukcje przypisania (podstawienia)
- instrukcja czytania „read”

oraz definicje niejednoznaczne:

- wywołanie procedury ze zmienną *x* jako parametrem (nie przekazywanym przez wartość)
- podstawienie przez wskazanie, które może odwoływać się do zmiennej *x*, np.: **q=y*, jeśli jest możliwość, żeby *q* wskazywało na *x*.

Problem zasięgu definicji

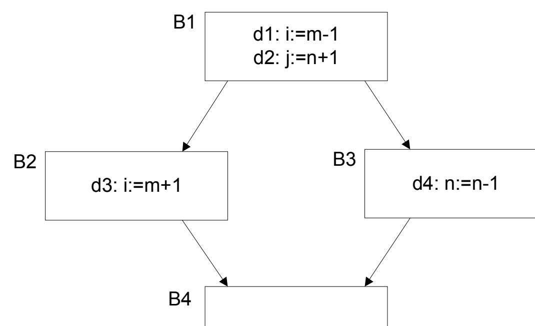
Definicja *d* zmiennej *a* osiąga punkt *p*, jeżeli istnieje ścieżka od punktu następującego po *d* do *p*, na której nie ma redefinicji *a*.

Mówimy, że definicja d_2 zmiennej *a* zabija definicję d_1 zmiennej *a*, jeżeli d_1 osiąga punkt poprzedzający d_2.

Problem zasięgu definicji

Definicja *d* zmiennej *a* osiąga punkt *p*, jeżeli istnieje ścieżka od punktu następującego po *d* do *p*, na której nie ma redefinicji *a*. Mówimy, że definicja d_2 zmiennej a zabija definicję d_1 zmiennej a, jeżeli d_1 osiąga punkt poprzedzający d_2.

Przykład:



Obie definicje d_1 i d_2 z bloku B_1 osiągają blok B_4 chociaż na jednej ze ścieżek definicja d_1 zmiennej *i* zostaje zabita poprzez definicję d_3 z bloku B_2. Istnieje jednak ścieżka z B_1 do B_4 przez B_3 bez redefinicji zmiennej *i*.

Iteracyjny algorytm dla zasięgu definicji

Wejście: Graf przepływu *G,* dla każdego bloku *B* obliczono zbiory *gen[B]* i *kill[B]*

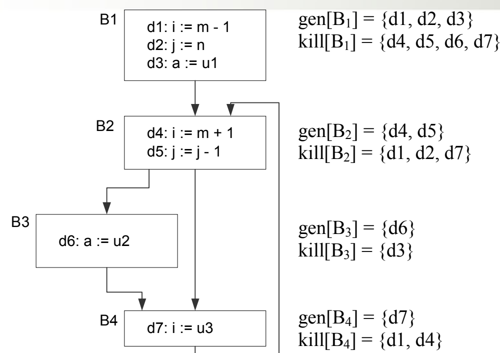
Wyjście: Zbiory *in[B]* i *out[B]* obliczone dla każdego bloku B

```
for each block B do out[B]:=gen[B];
change:=true;
while change do
   begin
      change:=false;
      for each block B do
         begin
            in[B]:=  ⋃ (P a predecessor of B)  out[P];
            old_out:=out[B];
            out[B]:= gen[B]  ∪ (in[B] - kill[B]);
            if out[B] ≠ old_out then change:=true;
         end
   end
```

$$in[B] := \bigcup_{P \text{ a predecessor of } B} out[P]$$

Algorytm pracuje dopóki występują zmiany. Górną granicą iteracji jest liczba węzłów

Iteracyjny algorytm dla zasięgu definicji – przykład



$gen[B_1] = \{d1, d2, d3\}$
$kill[B_1] = \{d4, d5, d6, d7\}$

$gen[B_2] = \{d4, d5\}$
$kill[B_2] = \{d1, d2, d7\}$

$gen[B_3] = \{d6\}$
$kill[B_3] = \{d3\}$

$gen[B_4] = \{d7\}$
$kill[B_4] = \{d1, d4\}$

Blok		Początkowo	Po 1-szym przejściu		Po 2-gim przejściu	
		out[B]	in[B]	out[B]	in[B]	out[B]
B1		111 0000	000 0000	111 0000	000 0000	111 0000
B2		000 1100	111 0001	001 1100	111 0111	001 1110
B3		000 0010	001 1100	000 1110	001 1110	000 1110
B4		000 0001	001 1110	001 0111	001 1110	001 0111

Problem dostępności wyrażeń

Wyrażenie *s: x* <u>*op*</u> *y* jest dostępne w punkcie *p*, gdy na każdej ścieżce z punktu początkowego do punktu *p* jest ono wyliczane i od ostatniego obliczenia do osiągnięcia punktu *p* nie ma definicji ani *x*, ani *y*.

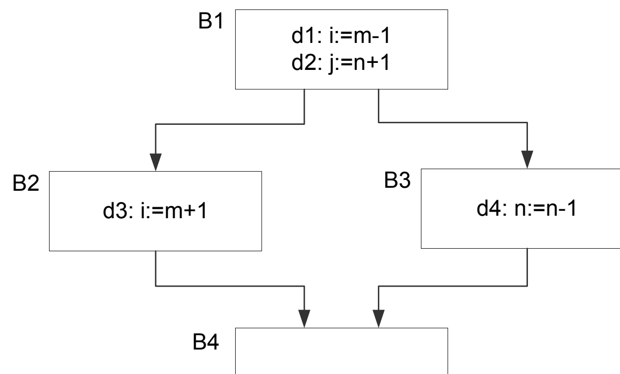
Mówimy że blok <u>zabija</u> wyrażenie *x* <u>*op*</u> *y* jeżeli redefiniuje *x* lub *y* i nie oblicza dalej *x* <u>*op*</u> *y*.

Blok <u>generuje</u> wyrażenie *x* <u>*op*</u> *y* jeżeli oblicza jego wartość i później nie redefiniuje ani *x* ani *y*.

Mimo że w problemie dostępności wyrażenia inaczej określamy generowanie i zabijanie to ogólne prawa analizy przepływu danych oczywiście są tutaj takie same.

Problem dostępności wyrażeń – przykład

Wyrażenie *s: x* <u>*op*</u> *y* jest dostępne w punkcie *p*, gdy na każdej ścieżce z punktu początkowego do punktu *p* jest ono wyliczane i od ostatniego obliczenia do osiągnięcia punktu *p* nie ma definicji ani *x*, ani *y*.



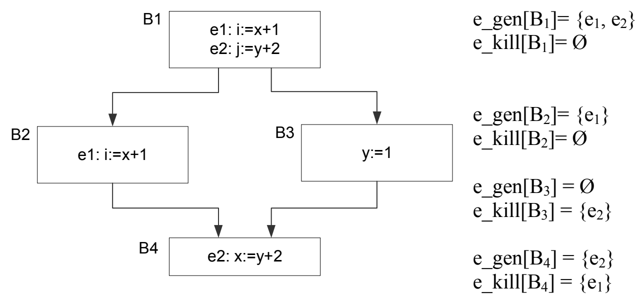
W bloku B_4 dostępne mamy wyrażenia *m-1* (d_1) oraz *m+1* (d_3). Dostępność wyrażeń *n+1* (d_2) i *n-1* (d_4) zostaje zabita przez definicję d_4 zmiennej *n*.

Iteracyjny algorytm dla dostępności wyrażeń

Wejście: Graf przepływu *G*, dla każdego bloku *B* wyznaczono zbiory *e_gen[B]* i *e_kill[B]*
Wyjście: Zbiory *e_in[B]* i *e_out[B]* dla każdego bloku *B*
Uwaga: *U* – zbiór wszystkich wyrażeń typu *x op y* we wszystkich instrukcjach w programie;
B_1 – blok początkowy

```
e_in[B1]:= Ø
e_out[B1]:=e_gen[B1]; /* in and out never change for B1 */
for B ≠ B1 do
  e_out[B]:= U – e_kill[B];
change := true;
while change do
  begin
    change:=false;
    for B ≠ B1 do
      begin   /* por. def.: na każdej ścieżce z B1 do
                 danego B jest wyliczane i … */
        e_in[B]:=        ∩         e_out[P];
                 P a predecessor of B
        oldout:=e_out[B]
        e_out[B]:=e_gen[B] ∪ (e_in[B]– e_kill[B])
        if e_out[B] ≠ oldout then change:=true;
      end;
  end;
```

Iteracyjny algorytm dla dostępności wyrażeń - przykład



$e_gen[B_1]=\{e_1, e_2\}$
$e_kill[B_1]=\emptyset$

$e_gen[B_2]=\{e_1\}$
$e_kill[B_2]=\emptyset$

$e_gen[B_3]=\emptyset$
$e_kill[B_3]=\{e_2\}$

$e_gen[B_4]=\{e_2\}$
$e_kill[B_4]=\{e_1\}$

$U=\{e_1, e_2\}$

Blok	Początkowo		Po pierwszym przejściu	
	e_in[B]	e_out[B]	e_in[B]	e_out[B]
B_1	00	11	00	11
B_2	00	11	11	11
B_3	00	10	11	10
B_4	00	01	10	01

Problem zasięgu instrukcji kopiowania

Instrukcja kopiowania *s: x:=y* osiąga punkt *p*, gdy na każdej ścieżce z punktu początkowego do punktu *p* instrukcja ta się pojawia i po ostatnim pojawieniu się *s* nie ma późniejszej redefinicji *y*.

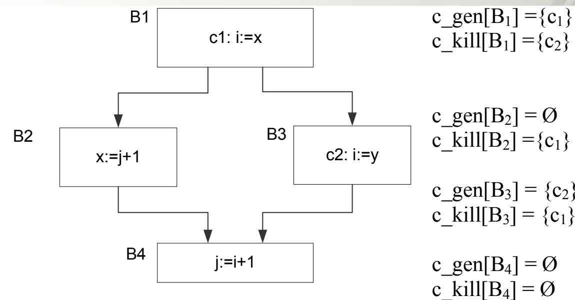
Mówimy, że instrukcja kopiowania *s: x:=y* jest generowana w bloku *B* jeżeli *s* pojawia się w *B* i później wewnątrz bloku *B* nie ma redefinicji zmiennej *y*. Mówimy, że *s: x:=y* jest zabijana w bloku *B*, jeśli w bloku *B* jest definicja zmiennej *x* lub *y* oraz *s* nie występuje w *B*.

Iteracyjny algorytm dla zasięgu instrukcji kopiowania

Wejście: Graf przepływu *G*, w którym wyznaczono zbiory *c_gen[B]* i *c_kill[B]*
Wyjście: Zbiory *c_in[B]* i *c_out[B]* dla każdego bloku *B*
Uwaga: *U* – zbiór wszystkich instrukcji kopiowania *x:=y* w całym programie.
B_1 – blok początkowy

```
c_in[B1]:= Ø
c_out[B1]:=c_gen[B1];
for B ≠ B1 do
   c_out[B]:=U – c_kill[B];
change:=true;
while change do
   begin
      change:=false;
      for B ≠ B1 do
         begin
            c_in[B]:=  ∩ (P a predecessor of B)  c_out[P];
            oldout:=c_out[B]
            c_out[B]:=c_gen[B] ∪ (c_in[B] – c_kill[B])
            if c_out[B] ≠ oldout then change := true;
         end;
   end;
```

Iteracyjny algorytm dla zasięgu instrukcji kopiowania – przykład



$c_gen[B_1] = \{c_1\}$
$c_kill[B_1] = \{c_2\}$

$c_gen[B_2] = \emptyset$
$c_kill[B_2] = \{c_1\}$

$c_gen[B_3] = \{c_2\}$
$c_kill[B_3] = \{c_1\}$

$c_gen[B_4] = \emptyset$
$c_kill[B_4] = \emptyset$

$U=\{c_1, c_2\}$

Blok	Początkowo		Po pierwszym przejściu	
	c_in[B]	c_out[B]	c_in[B]	c_out[B]
B_1	00	10	00	10
B_2	00	01	10	00
B_3	00	01	10	01
B_4	00	11	00	00

Problem życia zmiennych

Mówimy, że zmienna *x* w punkcie *p* jest żywa, jeżeli jej wartość jest używana na jakiejś ścieżce rozpoczynającej się w punkcie *p*. W przeciwnym przypadku mówimy, że zmienna jest martwa.

Będziemy określać *l_in[B]* jako zbiór zmiennych żywych w punkcie rozpoczynającym blok *B* (usytuowanym przed pierwszą instrukcją bloku *B*). Podobnie określamy *l_out[B]* jako zbiór zmiennych żywych w punkcie kończącym blok *B* (za ostatnią instrukcją bloku *B*).

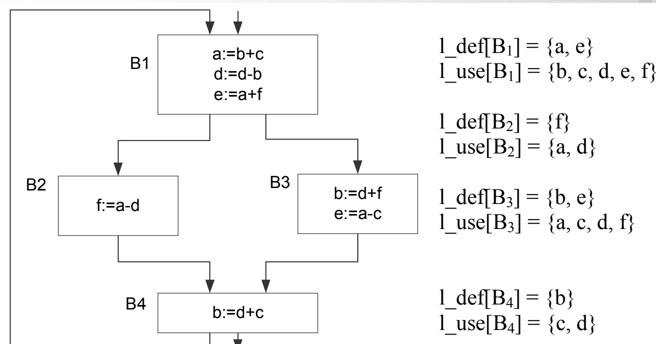
Niech *l_def[B]* będzie zbiorem zmiennych, którym w bloku *B* przypisywane są wartości wcześniej, niż zmienne te są ewentualnie używane w *B*. Niech dalej *l_use[B]* będzie zbiorem takich zmiennych za bloku *B,* które są używane w *B* wcześniej niż następuje ewentualne przypisanie wartości tym zmiennym w bloku *B*. Zbiór *def[]* jest odpowiednikiem *kill[],* zaś *use[]* jest odpowiednikiem *gen[]*.

Iteracyjny algorytm dla życia zmiennych

Wejście: Graf przepływu *G*, dla każdego bloku wyznaczono zbiory *l_use[B]* i *l_def[B]*
Wyjście: Zbiory *l_in[B]* i *l_out[B]* dla każdego bloku *B*
Uwaga: W odróżnieniu od poprzednich algorytm oblicza *in[]* w zależności od *out[]* oraz analizuje graf przepływu "wstecz"

```
for each block B do l_in[B]:=Ø
change:=true;
while change do
   begin
      change:=false;
      for each block B do
         begin
            l_out[B]:=        ⋃             l_in[S];
                     P a predecessor of B
            oldin:=l_in[B]
            l_in[B]:=c_use[B] ∪ (l_out[B] – l_def[B])
            if l_in[B]<>oldin then change:=true;
         end;
   end;
```

Iteracyjny algorytm dla życia zmiennych - przykład



$l_def[B_1] = \{a, e\}$
$l_use[B_1] = \{b, c, d, e, f\}$

$l_def[B_2] = \{f\}$
$l_use[B_2] = \{a, d\}$

$l_def[B_3] = \{b, e\}$
$l_use[B_3] = \{a, c, d, f\}$

$l_def[B_4] = \{b\}$
$l_use[B_4] = \{c, d\}$

Blok	$l_in^1[B]$	$l_out^2[B]$	$l_in^2[B]$	$l_out^3[B]$	$l_in^3[B]$
B_1	b, c, d, f	a, c, d, f	b, c, d, f	a, c, d, f	b, c, d, f
B_2	a, d	c, d	a, c, d	c, d, f	a, c, d
B_3	a, c, d, f	c, d	a, c, d, f	c, d, f	a, c, d, f
B_4	c, d	b, c, d, f	c, d, f	b, c, d, f	c, d, f

Listy Definition-Use Chains i Use-Definition Chains

Lista *DUC(p,x)* – łańcuch użyć dla definicji (Definition-Use Chains) to obliczana dla punktu *p*, poprzedzającego definicję *d* zmiennej *x*, lista tych wszystkich użyć tej zmiennej, które są osiągane przez definicję *d*. Zatem dla każdej definicji mamy zebraną listę instrukcji, które mogą ją potencjalnie wykorzystać

Lista *UDC(p,x)* – łańcuch definicji dla użycia (Use-Definition Chains) to obliczana dla punktu i, poprzedzającego użycie *s* zmiennej *x*, lista tych wszystkich definicji *i*, które osiągają punkt *p*. Lista *UDC* jest więc „odwrotnością" listy *DUC* – dla każdego użycia mamy zebrane informacje o definicjach, które mogą osiągać to użycie.

Listy Definition-Use Chains i Use-Definition Chains - przykład