AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Postępowanie z synonimami

Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki

Postępowanie z synonimami (aliases) związanymi ze wskaźnikami

Jeśli dwa lub więcej wyrażenia oznaczają te samą lokalizację w pamięci, nazywamy je synonimami. Będziemy rozważać dwa źródła powstawania synonimów: wskaźniki i procedury.

Jeśli nie wiemy nic na temat: gdzie wskazuje pointer, jedynym bezpiecznym założeniem jest, że pośrednie podstawienie poprzez wskaźnik $(*p := x)$ może potencjalnie zmienić (definiować) każdą zmienną, a także że użycie danej wskazywanej przez pointer $(x := *p)$ może potencjalnie dotyczyć każdej zmiennej. Rezultatem takiego założenia jest nierealistycznie duża liczba dostępnych definicji i żywych zmiennych oraz nierealistycznie mała liczba dostępnych wyrażeń.

Postępowanie z synonimami (aliases) związanymi ze wskaźnikami

Dlatego istotne jest uzyskanie wiarygodnej informacji na temat: na co w danym miejscu programu może wskazywać pointer.

Informację tę możemy uzyskać metodą analizy przepływu danych, co będzie zilustrowane na przykładzie prostego języka dopuszczającego wykorzystanie wskaźników. Załóżmy, że język ten operuje na elementarnych danych, z których każda zajmuje jedną komórkę pamięci i na tablicach, których elementami są dane zajmujące pojedyncze komórki pamięci. Pointery mogą wskazywać na dane elementarne lub tablice, ale nie mogą wskazywać na inne pointery. Wystarczającą informacją dla nas będzie, że pointer wskazuje na jakiś element tablicy, bez precyzowania, który konkretny element jest wskazywany przez pointer.

Postępowanie z synonimami (aliases) związanymi ze wskaźnikami

Załóżmy ponadto, że dozwolone jest dodanie lub odjęcie liczby całkowitej tylko do pointera wskazującego na tablicę i że po takiej operacji arytmetycznej pointer w dalszym ciągu skazuje na pewien element tej tablicy. Dozwolone jest także podstawienie do pointera innego wskaźnika i przypisanie pointerowi adresu zmiennej elementarnej lub tablicy, a także wykorzystywanie pointera do odwołań pośrednich do wskazywanych przez niego danych. Zasady określające, co dany pointer wskazuje w danym miejscu programu, zostaną przedstawione na kolejnym slajdzie:

Postępowanie z synonimami (aliases) związanymi ze wskaźnikami

(1) Bezpośrednio po instrukcji $s : p := \&a$ pointer p wskazuje tylko na a. Jeśli a jest tablicą, to pointer p bezpośrednio po instrukcji $s : p := \&a \pm c$ (gdzie c jest całkowitą stałą lub zmienną) wskazuje tylko na tablicę a (na jakiś jej element).

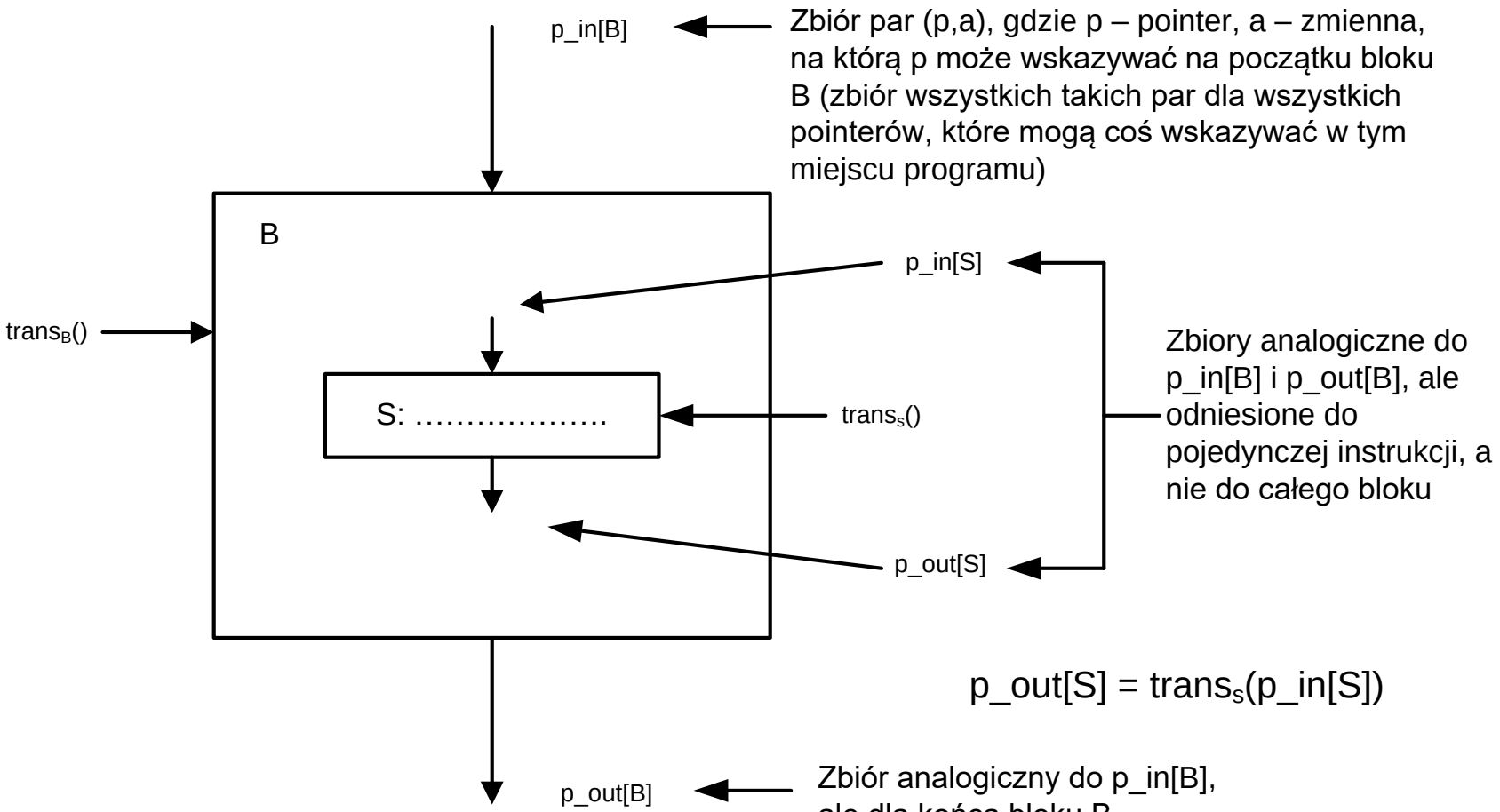
(2) Bezpośrednio po instrukcji $s : p := q \pm c$ (gdzie p i q są pointerami, zaś c jest całkowite i różne od zera) pointer wskazuje na wszystkie tablice, na które wskazywał q przed wykonaniem s i na nic więcej (nie wskazuje na żadną ze zmiennych elementarnych, na którą wskazywał q).

(3) Bezpośrednio po instrukcji $s : p := q$ pointer p wskazuje na wszystko, na co wskazywał q przed wykonaniem s.

(4) Po każdym innym podstawieniu do p – nie ma żadnego obiektu, na który p mógłby wskazywać.

(5) Po każdym podstawieniu do zmiennej innej niż p, pointer p wskazuje na ten sam zbiór obiektów, na który wskazywał przed takim podstawieniem (przypomnienie: pointer nie może wskazywać na inny pointer).

Analiza przepływu danych dla problemu wskaźników



Analiza przepływu danych dla problemu wskaźników

Określanie funkcji przejścia $trans_S(p\_in[S])$:

(1) Jeśli S ma postać $p := \&a$ lub $p := \&a \pm c$ to:

$$trans_S(p\_in[S]) = p\_in[S] - \{(p,b) \mid \text{dla każdego } b\} \cup \{(p,a)\}$$

(2) Jeśli S ma postać $p := q \pm c$, gdzie q – pointer, c – całkowite $\neq 0$, to:

$$trans_S(p\_in[S]) = p\_in[S] - \{(p,b) \mid \text{dla każdego } b\}$$
$$\cup \{(p,d) \mid (q,d) \in p\_in[S] \wedge d \text{ jest tablicą}\}$$

(3) Jeśli S ma postać $p := q$, to:

$$trans_S(p\_in[S]) = p\_in[S] - \{(p,b) \mid \text{dla każdego } b\}$$
$$\cup \{(p,d) \mid (q,d) \in p\_in[S]\}$$

(4) Jeśli w S podstawia się do pointera p coś innego, to:

$$trans_S(p\_in[S]) = p\_in[S] - \{(p,b) \mid \text{dla każdego } b\}$$

(5) Jeśli S nie jest podstawieniem do pointera, to:

$$trans_S(p\_in[S]) = p\_in[S]$$

Analiza przepływu danych dla problemu wskaźników

Funkcję przejścia dla bloku B, składającego się z instrukcji $S_1, S_2, ..., S_k$ określa się następująco:

$$trans_B(p\_in[B]) =$$

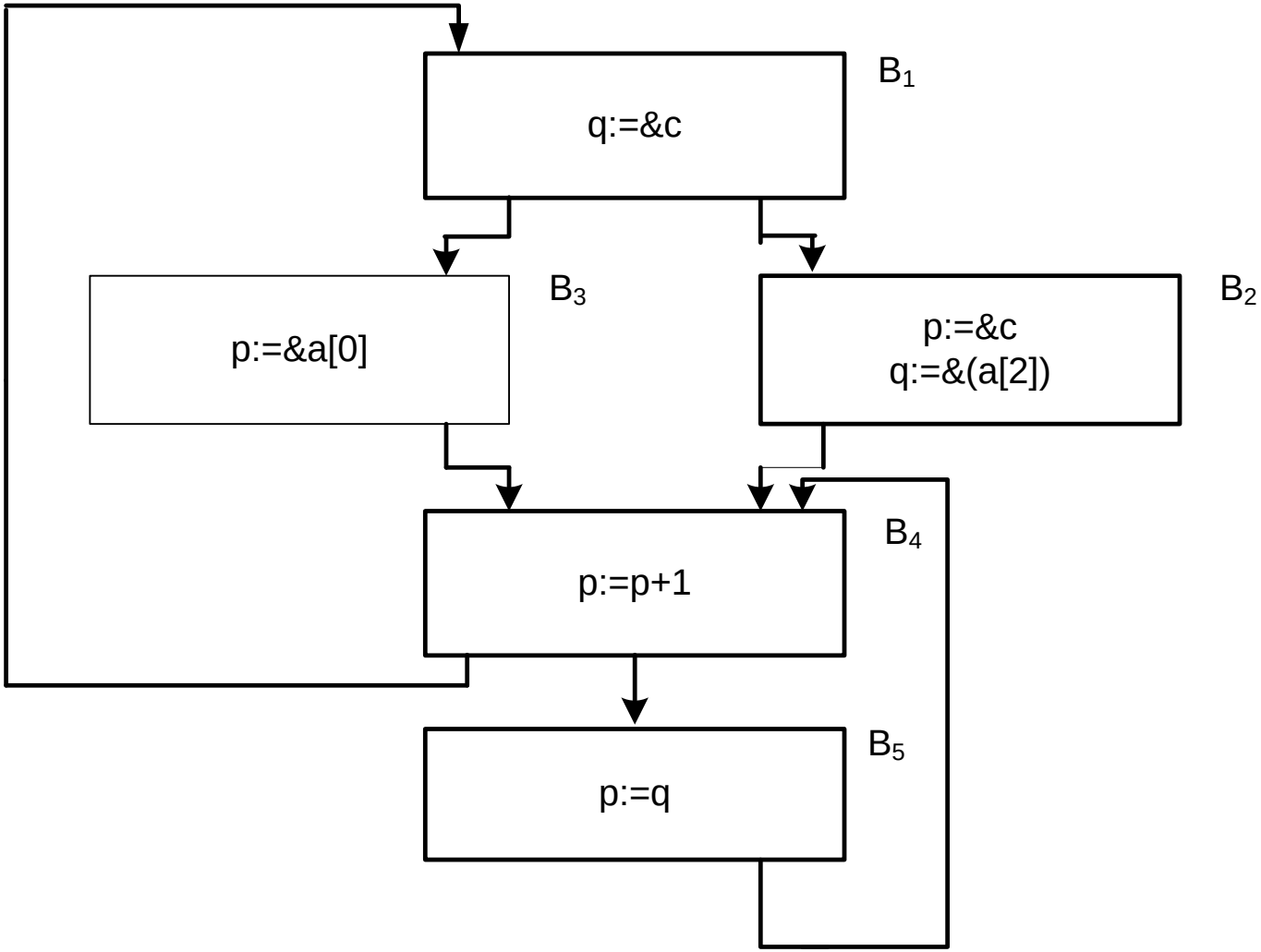
$$trans_{S_k}(trans_{S_{k-1}}(...(trans_{S_2}(trans_{S_1}(p\_in[B])))...))$$

Informację dotyczącą problemu wskaźników wyznacza się poprzez wykonanie iteracyjnego algorytmu analizy przepływu danych, analogicznego do algorytmu zasięgu definicji, wykorzystując równania:

$$p\_out[B] = trans_B(p\_in[B])$$

$$P\_in[B] = \bigcup_{P-poprzednik\ \ blokuB} P\_out[P]$$

Analiza przepływu danych dla problemu wskaźników – przykład



Analiza przepływu danych dla problemu wskaźników – przykład

Pierwszy przebieg:

Zakł. początkowo: $p\_in[B_1] = \phi$

$$p\_out[B_1] = transB_1(p\_in[B_1]) = transB_1(\phi) = \{(q,c)\}$$

$$p\_in[B_2] = p\_out[B_1] = \{(q,c)\}$$

$$p\_out[B_2] = transB_2(p\_in[B_2]) = transB_2(\{q,c\}) = \{(p,c),(q,a)\}$$

$$p\_in[B_3] = p\_out[B_1] = \{q,c\}$$

$$p\_out[B_3] = transB_3(p\_in[B_3]) = transB_3(\{q,c\}) = \{(p,a),(q,c)\}$$

$$p\_in[B_4] = p\_out[B_2] \cup p\_out[B_3] \cup p\_out[B_5];$$

zakł. $p\_out[B_5] = \varnothing$

$$p\_in[B_4] = \{(p,a),(p,c),(q,a),(q,c)\}$$

$$p\_out[B_4] = trans_{B_4}(p\_in[B_4]) = \{(p,a),(q,a),(q,c)\};$$

c nie jest tablicą

$$p\_in[B_5] = p\_out[B_4] = \{(p,a),(q,a),(q,c)\}$$

$$p\_out[B_5] = trans_{B_5}(p\_in[B_5]) = \{(p,a),(p,c),(q,a),(q,c)\}$$

Analiza przepływu danych dla problemu wskaźników – przykład

Drugi przebieg:

$$p\_in[B_1] = p\_out[B_4] = \{(p,a),(q,a),(q,c)\}$$
$$p\_out[B_1] = trans_{B_1}(p\_in[B_1]) = \{(p,a),(q,c)\}$$
$$p\_in[B_2] = p\_out[B_1] = \{(p,a),(q,c)\}$$
$$p\_out[B_2] = trans_{B_2}(p\_in[B_2]) = \{(p,c),(q,a)\}$$
$$p\_in[B_3] = p\_out[B_1] = \{(p,a),(q,c)\}$$
$$p\_out[B_3] = trans_{B_3}(p\_in[B_3]) = \{(p,a),(q,c)\}$$
$$p\_in[B_4] = p\_out[B_2] \cup p\_out[B_3] \cup p\_out[B_4] = \{(p,a),(p,c),(q,a),(q,c)\}$$

…dalej bez zmian.

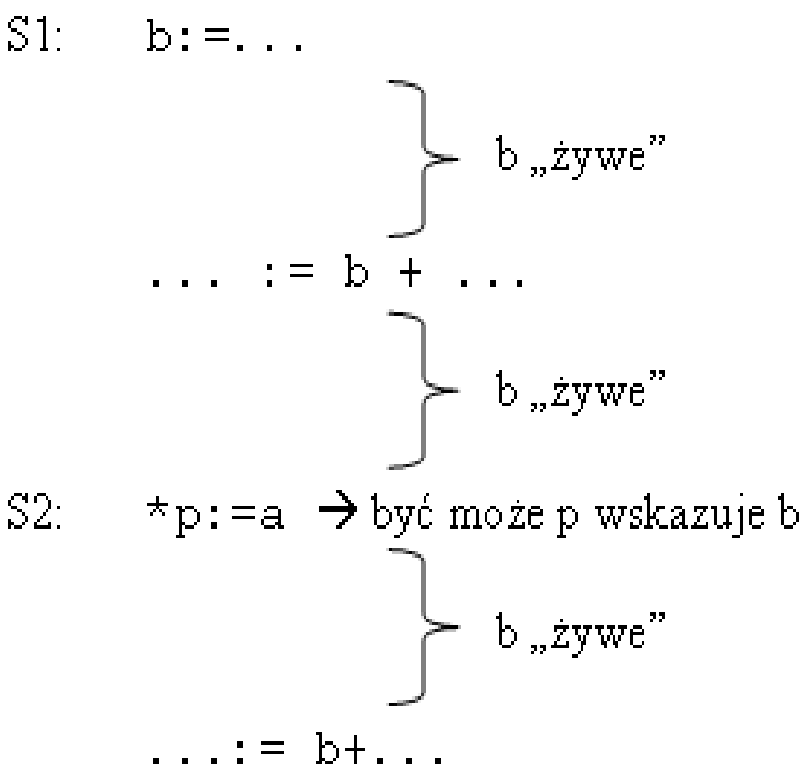
Mając $p\_in[B]$ i $p\_out[B]$ dla wszystkich bloków możemy wyznaczyć $p\_in[S]$ i $p\_out[S]$ dla wszystkich instrukcji wewnątrz bloków.

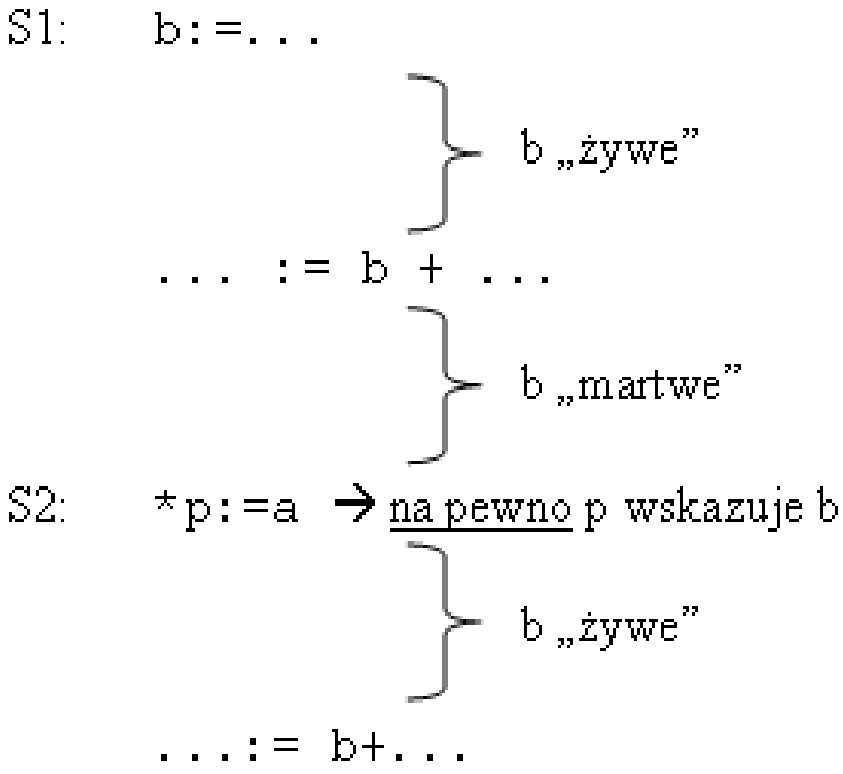
Analiza przepływu danych dla problemu wskaźników – przykład

Przykład:

Problem życia zmiennych rozwiązywany jest przez wcześniej podany algorytm; należy jedynie rozważyć sposób wyznaczania zbiorów *def* i *use* dla instrukcji typu $*p := a$ oraz $a := *p$. Instrukcja podstawienia pośredniego $*p := a$ używa tylko p i a. Zakładamy natomiast, że $*p := a$ definiuje b tylko wtedy, jeśli b jest jedyną zmienną, na którą może wskazywać pointer p, czyli gdy jesteśmy pewnie, że b rzeczywiście jest definiowalne. W ten sposób nigdy nie dopuścimy do sytuacji, że w pewnym punkcie programu przyjmujemy, że zmienna jest martwa, podczas gdy faktycznie jest ona żywa.

Analiza przepływu danych dla problemu wskaźników – przykład





Analiza przepływu danych dla problemu wskaźników – przykład

Instrukcja $a := *p$ definiuje tylko a, zaś używa p i każdej zmiennej b, na którą pointer p może wskazywać. W ten sposób mamy być może zbyt dużo zmiennych żywych, ale zmienne martwe można w trakcie np. generacji kodu odkładać do pamięci, a zmiennych żywych nie należy, powinny być w rejestrze tak długo, jak tylko można.

Wykorzystanie informacji o wskaźnikach

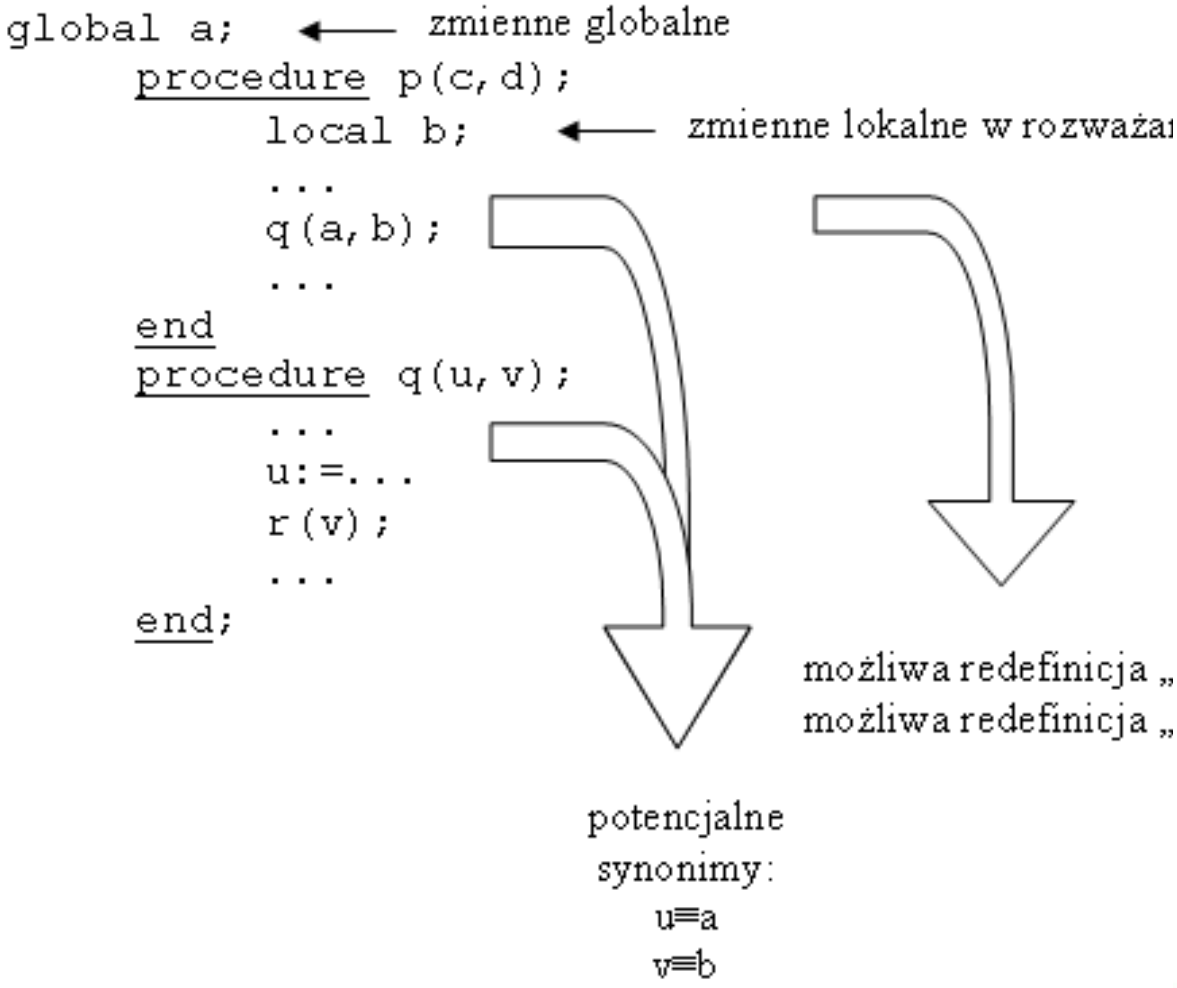
Algorytm poprzedni daje informacje typu: „co może wskazywać każdy pointer w miejscach, gdzie jest używany w pośrednich odwołaniach rodzaju $*p := a$, $a := *p$, itd.”. Te informacje można wykorzystać w poprzednio omawianych problemach analizy przepływu danych. Zawsze należy postępować tak, aby ewentualne błędy miały charakter zachowawczy (aby nic nie popsuły).

Wykorzystanie informacji o wskaźnikach – przykład

Problem zasięgu definicji jest rozwiązywany przy wykorzystaniu wcześniejszego algorytmu; należy tylko zmodyfikować sposób wyznaczenia zbiorów *gen* i *kill*. Przy wyznaczaniu zbioru *gen* przyjmuje się, że pośrednie podstawienie $*p := a$ generuje definicje każdej zmiennej *b*, na którą pointer *p* może wskazywać. Jest to podejście zachowawcze, gdyż lepiej przyjąć, że pewna definicja osiąga dany punkt, podczas gdy rzeczywiście tak nie jest, niż na odwrót – uznać, że pewna definicja nie osiąga danego punktu, kiedy jednak naprawdę go osiąga. Podczas wyznaczania zbioru *kill* zakładamy, że podstawienie pośrednie $*p := a$ zabija definicję zmiennej *b* tylko wtedy, gdy *b* nie jest tablicą i równocześnie *b* jest jedyną zmienną, na którą pointer *p* może wskazywać w tym miejscu programu. Czyli $*p := a$ zabija definicję *b* wskazywanej przez *p* tylko wówczas, gdy jesteśmy tego absolutnie pewni.

Synonimy związane z procedurami

Przykład: (przekazywanie parametrów przez referencje)



Synonimy związane z procedurami

Założenia dotyczące przykładowego języka:

(1) Dopuszczalne rekurencyjne wywołania procedur,
(2) Procedury mogą operować na zmiennych globalnych oraz swoich własnych zmiennych lokalnych (nie ma blokowej struktury ≡ zagłębiania procedur),
(3) Parametry są przekazywane przez referencje (adres),
(4) Każda procedura ma pojedynczy punkt wejścia i pojedynczy punkt wyjścia.

Prosty algorytm znajdowania synonimów

Wejście: zbiór procedur i zmiennych globalnych

Wyjście: relacja równoważności $\equiv$ o tej własności, że zawsze jeśli w programie x oraz y są wzajemnie synonimami, to $x \equiv y$.

Zależność odwrotna nie musi być prawdziwa. Relację $\equiv$ można nazwać: „może być synonimem”.

Prosty algorytm znajdowania synonimów

Metoda:

(1) Jeśli jest to konieczne, zmień nazwy zmiennych tak, aby żadne dwie procedury nie używały tych samych identyfikatorów zmiennych lokalnych i parametrów formalnych oraz aby identyfikatory zmiennych globalnych były różne od identyfikatorów zmiennych lokalnych lub parametrów formalnych procedur.

(2) Jeśli jest procedura $p(x_1, x_2, ..., x_n)$ oraz wywołanie $p(y_1, y_2, ..., y_n)$ tej procedury, dołącz do zbioru relacji $\equiv$ pary $x_i \equiv y_i$ dla wszystkich i, gdyż każdy parametr formalny może być synonimem odpowiadającego mu parametru aktualnego.

(3) Stwórz przechodnie i zwrotne domknięcie relacji $\equiv$, dodając:

(a) $x \equiv x$ dla wszystkich parametrów aktualnych i formalnych,

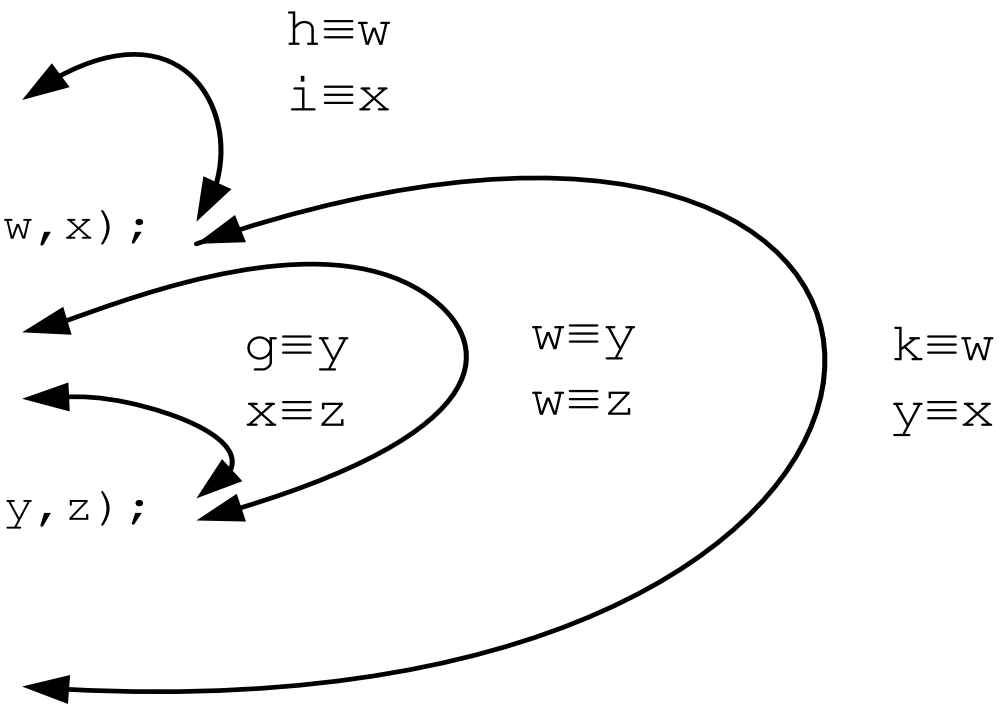
(b) $x \equiv y$ gdy $y \equiv x$,

(c) $x \equiv z$ gdy $x \equiv y$ oraz $y \equiv z$ dla pewnego y.

Prosty algorytm znajdowania synonimów

Przykład

```
global g,h;
      procedure main( );
            local i;
            g:=...;
            one(h,i);
      end;
      procedure one(w,x);
            x:=...;
            two(w,w);
            two(g,x);
      end;
      procedure two(y,z);
            local k;
            h:=...;
            one(k,y)
      end;
```



Prosty algorytm znajdowania synonimów

	g	h	i	w	x	y	z	k
g						1		
h				1				
i					1			
w						1	1	
x							1	
y							1	
z								
k				1				

Relacja ≡ po kroku (2)

	g	h	i	w	x	y	z	k
g	1					1		
h		1		1				
i			1		1			
w		1		1		1	1	1
x			1		1		1	
y	1			1		1	1	
z				1	1	1	1	
k				1				1

Relacja ≡ po kroku (3)(b)

Po wykonaniu kroku (3)(c) okaże się, że każda z rozważanych zmiennych może być synonimem każdej innej. Tak uzyskana relacja jest „zbyt duża” (np. *g* i *h* na pewno zajmują, jako zmienne globalne, odrębne miejsce w pamięci.

Synonimy związane z procedurami

Problem synonimów wnoszonych przez wywołania procedur zilustrujemy na przykładzie modyfikacji procesu eliminacji wspólnych podwyrażeń.

Dla każdej procedury *p* definiujemy zbiór *change[p]*, którego elementami są zmienne globalne i formalne parametry procedury *p*, które mogą być zmieniane w trakcie wykonania procedury *p*. Niech *def[p]* będzie zbiorem zmiennych globalnych i parametrów formalnych procedury *p*, które mają w tej procedurze bezpośrednie definicje (poprzez instrukcje podstawienia, a nie poprzez parametry aktualne).

Wtedy:

$$change[p] = def[p] \cup A \cup G$$

gdzie:

- *A* jest zbiorem takich zmiennych globalnych i formalnych parametrów procedury *p*, które są aktualnymi parametrami wywołania procedur *q* z wnętrza *p*, a odpowiadające im parametry formalne procedury *q* należą do *change[q]*.
- *G* jest zbiorem zmiennych globalnych należących do *change[q]*, przy czym *q* są procedurami wywoływanymi z wnętrza *p*.

Synonimy związane z procedurami

Zależy nam na rozwiązaniu równania

$$change[p] = def[p] \cup A \cup G$$

dla wszystkich procedur (metodą iteracyjną), przy czym chcemy uzyskać rozwiązanie „najmniejsze”. Tworzymy graf wywołań, którego węzły są procedurami, zaś krawędź od p do q jest obecna wtedy i tylko wtedy, gdy p wywołuje q. O ile to możliwe, w algorytmie rozpoczynamy przeszukiwanie grafu wywołań od węzła, który odpowiada procedurze nie wywołującej żadnej innej procedury.

Algorytm wyznaczania zbiorów *change[p]* dla procedur

Wejście: zbiór procedur $p_1, p_2, ..., p_n$. Jeżeli graf wywołań jest acykliczny, zakładamy, że p_i wywołuje p_j tylko gdy $j < i$. W przeciwnym wypadku nie stawiamy takiego wymagania.

Wyjście: dla każdej procedury *p* tworzony jest zbiór *change[p]* zawierający zmienne globalne i parametry formalne, które mogą być zmienione w procedurze *p*.

Metoda: wyznaczamy dla każdej procedury *p* zbiór *def[p]* zgodnie z definicją. Następnie:

Synonimy związane z procedurami

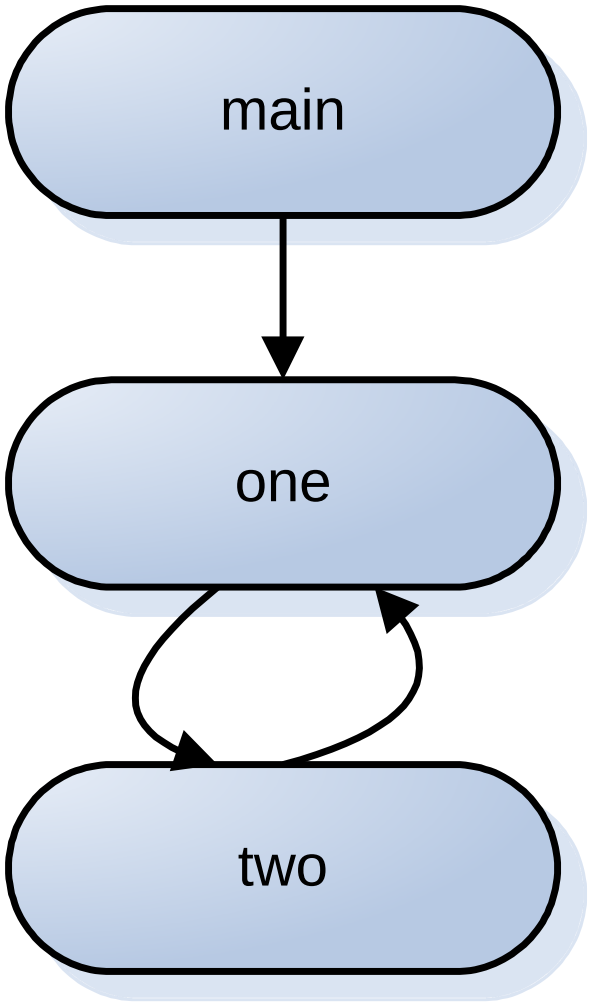
```
(1)     for każda procedura p do change[p]:=def[p];
(2)     while zmiany występują w jakimkolwiek change[p] do
(3)        for i:=1 to n do
(4)          for każda procedura q wywoływana przez [pi] do
              begin
(5)              dołącz każdą zmienną globalną z change[q] do
                      change [pi];
(6)              for każdy parametr formalny x (j-ty) procedury
                          q do
(7)                 if x należy do change[q] then
(8)                    for każde wywołanie q w pi do
(9)                       if a - j-ty aktualny parametr
                             wywołania jest zmienną globalną lub
                             parametrem formalnym w pi
(10)                         then dołącz a do zbioru change[pi]
(11)         end
```

Powyższy algorytm działa niezależnie od poprzedniego i nie korzysta z tamtych wyników.

Synonimy związane z procedurami

Przykład:

```
global g,h;
     procedure main ();
          local i;
          g:=...;
          one (h,i);
     end;
     procedure one (w,x);
          x:=...;
          two (w,w);
          two (g,x);
     end;
     procedure two (y,z);
          local k;
          h:=...;
          one (k,y);
     end;
```



Synonimy związane z procedurami

Ustalamy kolejność:

two, *one*, *main*

Początkowo:

$def[two] = \{h\}$

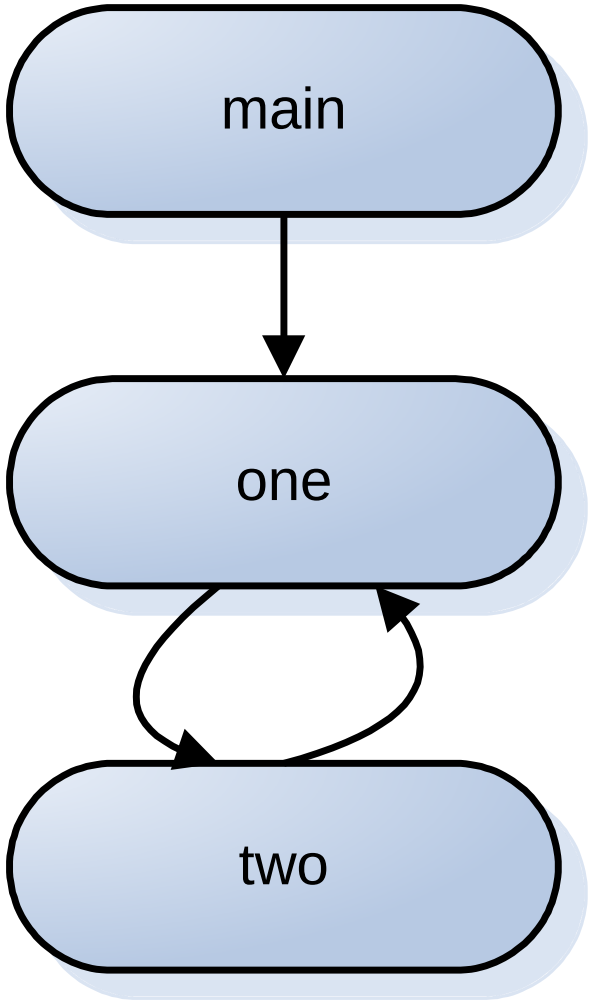
$def[one] = \{x\}$

$def[main] = \{g\}$

$change[two] = \{h\}$

$change[one] = \{x\}$

$chane[main] = \{g\}$



Synonimy związane z procedurami

Przebieg pierwszy:

(a) $p_1 = two$

(4) $q = one$

(5) nic nowego

(6),(7) drugi formalny parametr procedury *one*

(10) $change[two] = \{h, y\}$

(b) $p_2 = one$

(4) $q = two$

(5) $change[one] = \{h, y\}$

(6),(7) pierwszy formalny parametr procedury *one*

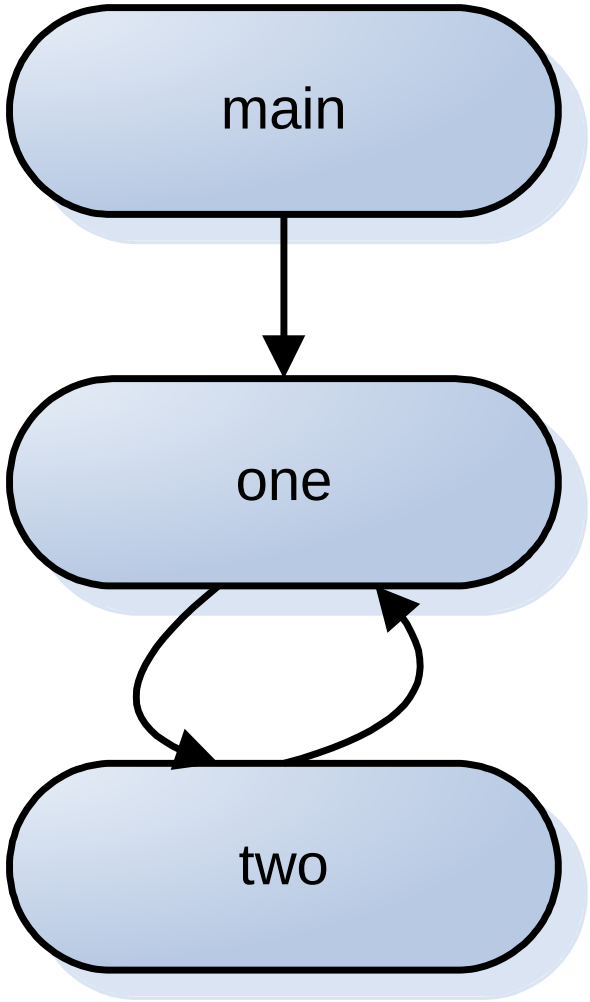
(10) $change[one] = \{h, y, w, g\}$

(c) $p_3 = main$

(4) $q = one$

(5) $change[main] = \{g, h\}$

dalej bez zmian



Synonimy związane z procedurami

Przebieg drugi:

(a) $p_1 = two$

(4) $q = one$

(5) $change[two] = \{h, y, g\}$

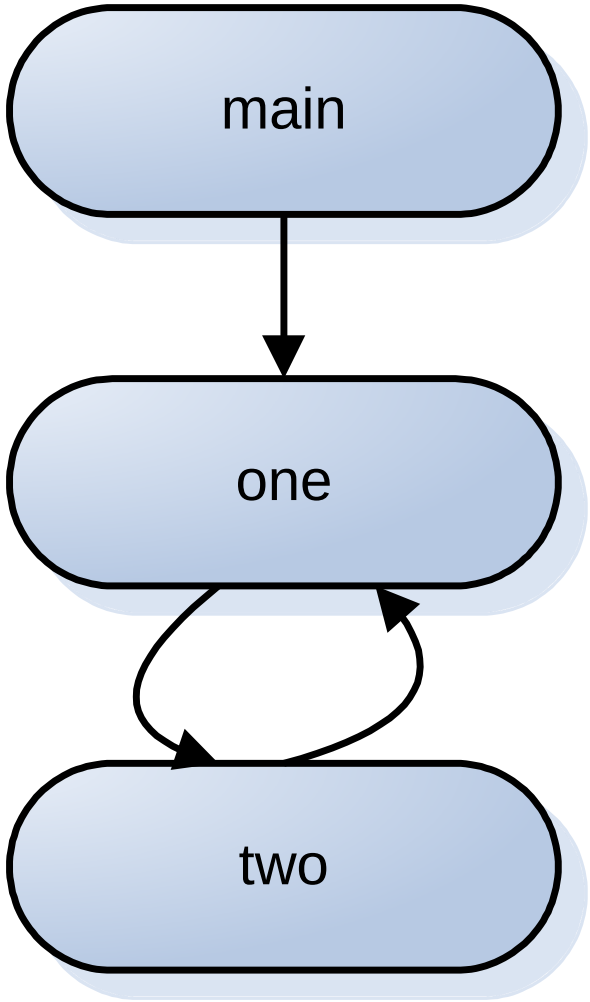
dalej bez zmian

Ostatecznie:

$change[two] = \{g, h, y\}$

$change[one] = \{g, h, w, x\}$

$change[main] = \{g, h\}$



Wykorzystanie zebranych informacji w problemie dostępności wyrażeń i eliminacji wspólnych podwyrażeń

Załóżmy, że mamy wyznaczyć zbiór *e_kill[B]* dla pewnego bloku *B* należącego do procedury *p*. Definicja zmiennej *a* musi być uznana za zabijającą każde wyrażenie odwołujące się do *a* lub odwołujące się do pewnego *x* które może być synonimem *a*. Jednak wywołanie w bloku *B* procedury *q* nie może zabić wyrażenia zawierającego *a*, chyba, że *a* jest synonimem (zauważ, że *a* jest synonimem samego siebie) pewnej zmiennej ze zbioru *change[q]*. Wyznaczenie zbioru *e_kill[B]* wymaga więc znajomości relacji ≡ oraz zbiorów *change[]* dla wszystkich procedur.

Wykorzystanie zebranych informacji w problemie dostępności wyrażeń i eliminacji wspólnych podwyrażeń

Drugim problemem jest kwestia rozstrzygnięcia, kiedy wywołanie procedury może generować wyrażenie *a op b*. Powinniśmy założyć, że *a op b* jest generowane przez wywołanie *q* wtedy i tylko wtedy, gdy na każdej ścieżce od punktu wejściowego procedury *q* do jej punktu wyjściowego *a op b* jest wyliczane i nie ma później żadnej redefinicji ani *a* ani *b*. Poszukując wystąpienia a op b nie możemy zaakceptować *x op y* jako takiego wystąpienia dopóki nie jesteśmy pewni, że w każdym wywołaniu *q x* jest (co nie znaczy „może być”!) synonimem *a* oraz *y* synonimem *b*. Najprościej więc przyjąć, że wywołanie *q* nie generuje niczego.

Wykorzystanie zebranych informacji w problemie dostępności wyrażeń i eliminacji wspólnych podwyrażeń

Bardziej skomplikowanym, lecz bliższym rzeczywistości rozwiązaniem jest iteracyjne wyznaczenie zbioru *gen[p]* (zbioru generowanych dostępnych wyrażeń) dla każdej procedury *p*. W algorytmie podobnym do tego, który wyznaczał zbiory *change[p]* możemy zainicjować zbiory *gen[p]* tak, aby zawierały wszystkie wyrażenia dostępne na końcu bloku wyjściowego z danej procedury *p*. Musimy jednak pamiętać, że obliczając zbiory *gen[p]* nie wolno nam uwzględniać informacji o synonimach; *a* $\underline{op}$ *b* reprezentuje tutaj tylko samego siebie, nawet gdyby inne zmienne mogły być synonimami *a* lub *b*. Wyznaczanie zbiorów *gen[p]* prowadzi do iteracyjnego przeglądania wszystkich bloków z wszystkich procedur i znajdowania wyrażeń dostępnych. Wywołania *q(a,b)* procedury *q* w procedurze *p* generuje te wyrażenia, które zawarte są w *gen[q]* (z uwzględnieniem substytucji odpowiednich parametrów formalnych). Zbiory *e_kill[B]* pozostają bez zmiany. Nowa zawartość *gen[p]* dla każdej procedury *p* może być określona, jeśli znajdziemy wyrażenia dostępne w węźle wyjściowym procedury *p*. Iterację prowadzimy tak długo, jak długo pojawiają się jakiekolwiek zmiany zbiorów wyrażeń dostępnych w poszczególnych węzłach.