Analisi statica dei programmi

Prof. A. Morzenti

aa 2008-2009

ANALISI STATICA DEI PROGRAMMI

Analisi statica e ottimizzazione dei programmi: tecnica impiegata da tutti i compilatori che traducono un linguaggio di programmazione in un codice macchina.

COMPILAZIONE:

PRIMO STADIO: traduce un programma in una rappresentazione intermedia che si avvicina al linguaggio macchina.

STADIO SUCCESSIVO – si applica al programma intermedio e può avere diverse finalità:

- VERIFICA – ulteriore esame della correttezza del programma
- OTTIMIZZAZIONE – trasformazione del programma per migliorare l'efficienza di esecuzione (allocazione ottimale dei registri alle variabili,…)
- SCHEDULAZIONE – modifica dell'ordine delle istruzioni per migliorare lo sfruttamento delle pipeline e delle unità funzionali del processore.

Conviene rappresentare come un AUTOMA il GRAFO DI CONTROLLO DEL FLUSSO (*control-flow graph*) su cui si basano i compiti precedentemente elencati

ATTENZIONE: l'automa definisce un ben preciso programma e NON l'intero linguaggio sorgente! La prospettiva non è quindi quella utilizzata per la traduzione guidata da sintassi.

IN QUESTO CASO: una stringa riconosciuta dall'automa è la sequenza temporale delle operazioni di calcolo che tale programma può eseguire, ossia una traccia della sua esecuzione.

ANALISI STATICA: lo studio di certe proprietà del grafo di controllo d'un programma, mediante i metodi della teoria degli automi, della logica o della statistica.
Consideremo solo i metodi basati su automi.

IL PROGRAMMA COME AUTOMA

- si considerano istruzioni più semplici (quelle della rappresentazione intermedia):
  - -variabili semplici e costanti
  - -assegnamenti a variabile,
  - -semplici operazioni aritmetiche, relazionali, logiche con un solo operatore
- consideriamo solo analisi INTRAPROCEDURALE (non interprocedurale)

GRAFO DI CONTROLLO DI UN PROGRAMMA:
-ogni nodo è un'istruzione
-se nell'esecuzione l'istruzione *p* può essere immediatamente seguita da dall'istruzione *q,* il grafo ha un arco orientato da *p* a *q* (*p* è il predecessore di *q*)
-la prima istruzione del programma è il nodo d'entrata (*nodo iniziale)*
-un'istruzione senza successori è un nodo d'uscita (*nodo finale)*
-le istruzioni non condizionali hanno un solo successore, quelle condizionali hanno uno o più successori
- un'istruzione con più predecessori è un *nodo di confluenza*

IL GRAFO DI CONTROLLO non è una rappresentazione fedele e completa del programma:

-non è rappresentato il valore VERO o FALSO che fa scegliere il successore di un'istruzione condizionale

-l'operazione di assegnamento, lettura, scrittura, … viene sostituita da una delle seguenti astrazioni:

- l'assegnamento o la lettura di una variabile … ***definisce*** quella variabile
- la comparsa di una variabile nella parte destra di un assegnamento, in una espressione o in una scrittura … ***usa*** quella variabile
- quindi nel grafo ogni nodo (istruzione) *p* associato due insiemi:

def(p) *e usa(p)*

$p: \quad a := a \oplus b$

$def(p) = \{a\}, usa(p) = \{a, b\}$

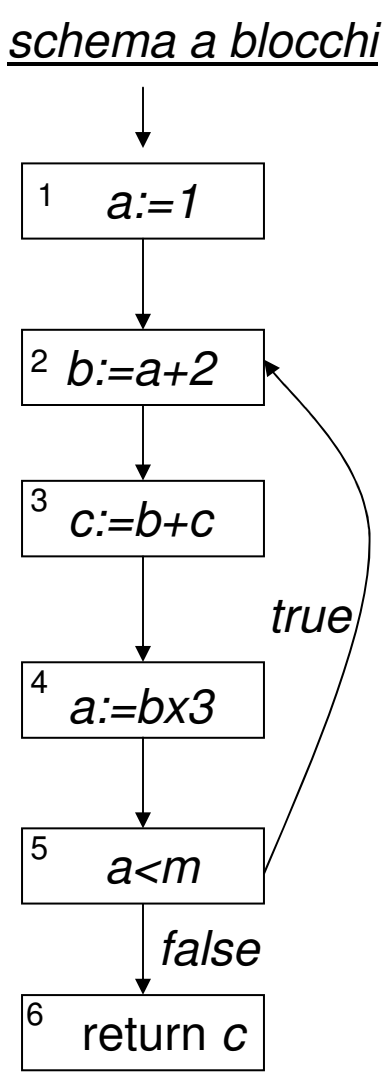
$q: \quad read(a) \qquad def(q) = \{a\}, usa(q) = \varnothing$

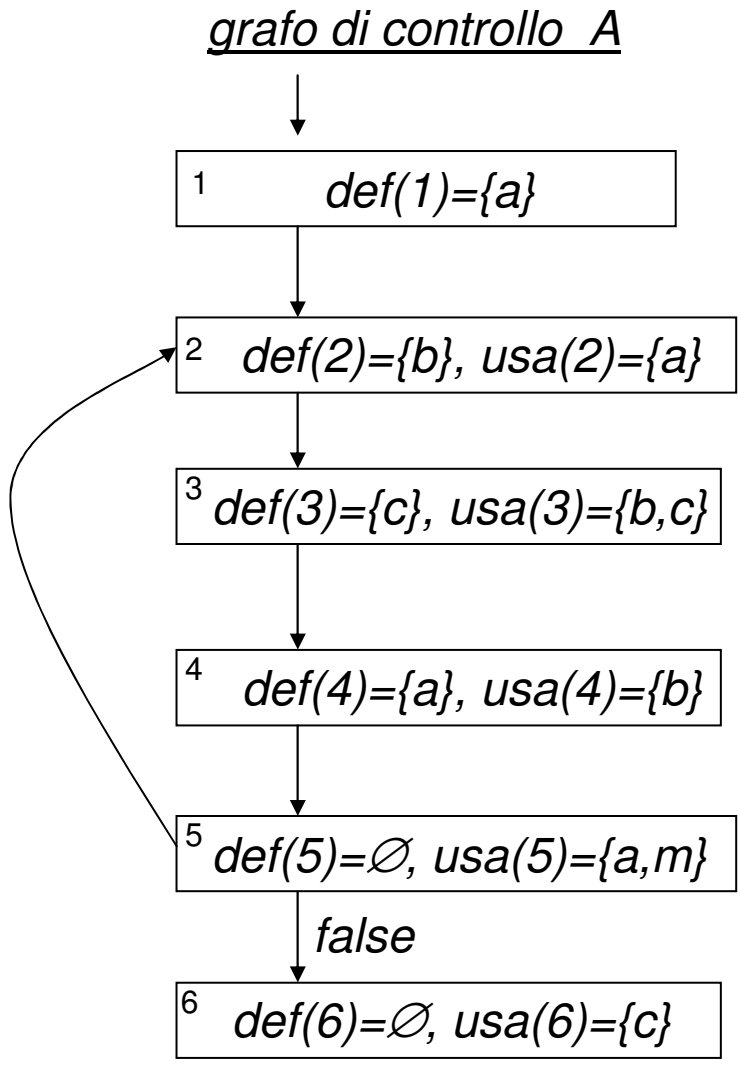
$w: \quad a := 7 \qquad def(w) = \{a\}, usa(w) = \varnothing$

ESEMPIO 1 – schema a blocchi e grafo di controllo

programma

```
    a:=1
e1: b:=a+2
    c:=b+c
    a:=bx3
    if a<m goto e1
    return c
```





DEFINIZIONE: LINGUAGGIO DEL GRAFO DI CONTROLLO
L'automa finito A, rappresentato dal grafo di controllo ha
-*alfabeto terminale*, l'insieme delle istruzioni I, ciascuna schematizzata dalla terna

$$\langle n, def(n) = \{...\}, usa(n) = \{...\} \rangle$$

-*stato iniziale:* quello senza predecessori *(*con freccia entrante)
-*stati finali:* quelli privi di successori

IL LINGUAGGIO FORMALE L(A) riconosciuto dall'automa contiene le stringhe di alfabeto I che etichettano un cammino dall'entrata all'uscita del grafo.
Il CAMMINO rappresenta una SEQUENZA DI ISTRUZIONI che la macchina potrebbe eseguire quando si lancia un programma.

ESEMPIO precedente: I = {1 … 6}
Un cammino riconosciuto è 1 →2 → 3→ 4→ 5→ 2→ 3→4 → 5→6 = 1234523456
L'insieme dei cammini è il linguaggio $1(2345)^{+}6$

APPROSSIMAZIONE CAUTELATIVA

Non è sempre vero che ogni cammino del grafo sia eseguibile dal programma dato che il grafo di controllo trascura le condizioni booleane che determinano la scelta del successore di un nodo.

$$1: \quad \text{if } a**2 \geq 0 \text{ then } istr_2 \text{ else } istr_3$$

Il linguaggio formale accettato dall'automa contiene i due cammini {12,13} ma il cammino 13 non è eseguibile.

E' IN GENERALE INDECIDIBILE se un cammino del grafo di controllo d'un programma possa o meno essere eseguito (questo equivarrebbe a decidere se esistono certi valori delle variabili di ingresso del programma, che causano l'esecuzione di quel cammino, ma il problema è riconducibile a quello dell'alt d'una macchina di Turing.

DIAGNOSI CAUTELATIVA: esaminare tutti i cammini dall'entrata all'uscita può portare alla diagnosi di errori inesistenti o all'assegnazione prudenziale di risorse in realtà non necessarie, ma non trascura mai le condizioni di errore.

IPOTESI: nell'analisi statica si fa l'ipotesi che l'AUTOMA SIA PULITO:
ogni istruzione giace su un cammino che dall'entrata (unica per ipotesi) porta a un'uscita.

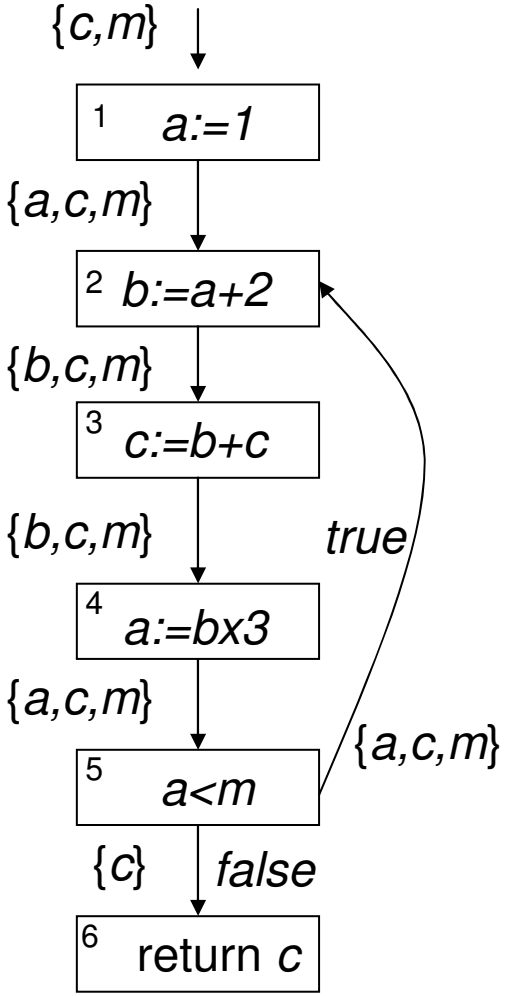
IN CASO CONTRARIO:
-il programma potrebbe non raggiugnere la fine
-il programma potrebbe contenere istruzioni che non possono mai essre eseguite (esiste del *codice irraggiungibile*)

INTERVALLI DI VITA DELLE VARIABILI

Una **variabile è viva in un certo punto** se qualche istruzione che potrebbe essere eseguita successivamente fa uso del valore che la variabile ha in quel punto (i.e., la variabile verrà usata prima di essere assegnata).

DEFINIZIONE: una variabile *a* è viva in un punto *p* del programma (ingresso o uscita di un'istruzione) se nel grafo esiste un cammino da *p* a un altro nodo *q* tale che

- il cammino non passa per un'istruzione *r, r* != *q* che definisce *a*, ossia tale che $a \in def(r)$ AND
- l'istruzione *q* fa uso di *a* ossia $a \in usa(q)$



← ***variabili vive sugli archi***

una variabile è
- ***viva all'uscita di un nodo*** se è viva su uno degli archi uscenti dal nodo
- ***viva all'entrata di un nodo*** se è viva su uno degli archi entranti nel nodo.

ESEMPIO:

c è viva all'ingresso del nodo 1 perché esiste il cammino 123: *c*∈ *usa(3)* e né 1 né 2 definiscono c

a è viva nei cammini 12 e 452,
a non è viva in 234 e 56

All'uscita di 5 sono vive *{a,c,m}*U{c}

CALCOLO DEGLI INTERVALLI DI VITA – Sia I l'insieme delle istruzioni, D(a) e U(a) gli insiemi delle istruzioni che rispettivamente definiscono e usano la variabile *a*

La proprietà che *a* sia viva all'uscita del nodo *p* equivale alla seguente condizione sul linguaggio accettato dall'automa:

Esiste in $L(A)$ una frase $x = upvqw$ tale che

$u \in I^* \wedge p \in I \wedge v \in (I \setminus D(a))^* \wedge q \in U(a) \wedge w \in I^*$

L'insieme di tutte le frasi x che soddisfano la condizione

è un linguaggio regolare

$L_p \subseteq L(A) \quad L_p = L(A) \bigcap R_p \quad R_p = I^* p (I \setminus D(a))^* U(a) I^*$

L'espressione prescrive che il carattere *p* sia seguito da un carattere *q* scelto tra U(a) e che gli eventuali caratteri interposti tra *p* e *q* non appartengano a D(a).
Per sapere se *a* sia viva all'uscita di *p,* basta vedere se il linguaggio L_p non è vuoto.

Per vedere se L_p non è vuoto potremmo costruire il riconoscitore del linguaggio Lp (la macchina prodotto cartesiano che riconosce l'intersezione e guarda se ci sono cammini dallo stato iniziale allo stato finale). Ma il procedimento
rischia di esser molto pesante per le dimensioni dei programmi da analizzare

METODO DELLE EQUAZIONI DI FLUSSO – calcola simultaneamente tutte le variabili vive in ogni punto del grafo. Si esaminano i cammini che dal punto considerato vanno a qualche istruzione che fa uso di una variabile.

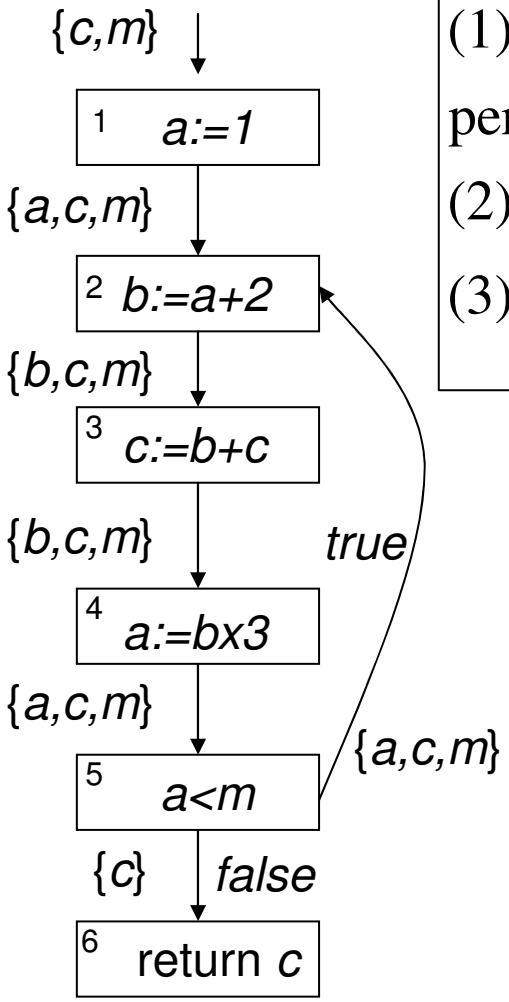
per ogni nodo p finale:

$$vive_{out}(p) = \varnothing$$

per ogni altro nodo p:

$$vive_{in}(p) = usa(p) \cup (vive_{out}(p) \setminus def(p))$$

$$vive_{out}(p) = \coprod_{\forall q \in succ(p)} vive_{in}(q)$$



per ogni nodo p finale:

(1) $vive_{out}(p) = \varnothing$

per ogni altro nodo p:

(2) $vive_{in}(p) = usa(p) \bigcup (vive_{out}(p) \setminus def(p))$

(3) $vive_{out}(p) = \coprod_{\forall q \in succ(p)} vive_{in}(q)$

Per (1): nessuna variabile è viva all'uscita del grafo

Per (2): $vive_{in}(4) = \{b,m,c\} = \{b\} \cup (\{a,c,m\} \setminus \{a\})$

Per (3): $succ(5) = \{2,6\}$

$vive_{out}(5) = vive_{in}(2) \cup vive_{in}(6) =$

$\{a,c,m\} \cup \{c\} = \{a,c,m\}$

SOLUZIONE DELLE EQUAZIONI DI FLUSSO

Per un grafo di |I|=n nodi si ottiene un sistema di 2 x n equazioni nelle 2 x n incognite $vive_{in}(p)$, $vive_{out}(p)$, $p \in I$.

La soluzione del sistema è un vettore di 2 x n insiemi.

Il sistema si risolve iterativamente assegnando l'insieme vuoto come valore iniziale a ogni incognita (iterazione i=0):

$$\forall p : vive_{in}(p) = \varnothing; vive_{out}(p) = \varnothing$$

Si sostituiscono nelle equazioni del sistema i valori dell'iterazione corrente *i* e si ricavano i valori dell'iterazione *i+1.* Se almeno uno di essi differisce dal valore dell'iterazione *i* si continua allo stesso modo, altrimenti si termina e i valori dell'iterazione *i+1* costituiscono la soluzione del sistema.
(Il solito PUNTO FISSO…)

Il calcolo converge dopo un numero finito di iterazioni:

1) ogni insieme $vivei_n(p)$ e $vive_{out}(p)$ ha cardinalità limitata superiormente dal numero di variabili del programma;

2) l'iterazione non toglie elementi da nessun insieme, ma li aggiunge o lo lascia immutato;

3) se l'iterazione lascia immutati tutti gli insiemi, l'algoritmo termina.

ESEMPIO – Calcolo iterativo delle variabili vive

1	$in(1) = out(1) \setminus \{a\}$	$out(1) = in(2)$
2	$in(2) = \{a\} \cup (out(2) \setminus \{b\})$	$out(2) = in(3)$
3	$in(3) = \{b,c\} \cup (out(3) \setminus \{c\})$	$out(3) = in(4)$
4	$in(4) = \{b\} \cup (out(4) \setminus \{a\})$	$out(4) = in(5)$
5	$in(5) = \{a,m\} \cup out(5)$	$out(5) = in(2) \cup in(6)$
6	$in(6) = \{c\}$	$out(6) = \emptyset$

	D	U
a	$1,4$	$2,5$
b	2	$3,4$
c	3	$3,6$
m	$\emptyset$	5

Insiemi delle istruzioni che definiscono e usano le variabili

NB: si suppone a ogni iterazione di calcolare prima gli *in* e poi gli *out*

	$in = out$	in	out	in	out	in	out	in	out	in	out
1	$\emptyset$	$\emptyset$	a	$\emptyset$	a,c	c	a,c	c	a,c,m	c,m	a,c,m
2	$\emptyset$	a	b,c	a,c	b,c	a,c	b,c,m	a,c,m	b,c,m	a,c,m	b,c,m
3	$\emptyset$	b,c	b	b,c	b,m	b,c,m	b,c,m	b,c,m	b,c,m	b,c,m	b,c,m
4	$\emptyset$	b	a,m	b,m	a,c,m	b,c,m	a,c,m	b,c,m	a,c,m	b,c,m	a,c,m
5	$\emptyset$	a,m	a,c	a,c,m	a,c	a,c,m	a,c	a,c,m	a,c,m	a,c,m	a,c,m
6	$\emptyset$	c	$\emptyset$	c	$\emptyset$	c	$\emptyset$	c	$\emptyset$	c	$\emptyset$

Complessità: O(n^2) nel caso pessimo, in pratica poco piu` che lineare

APPLICAZIONE DEL RISULTATO

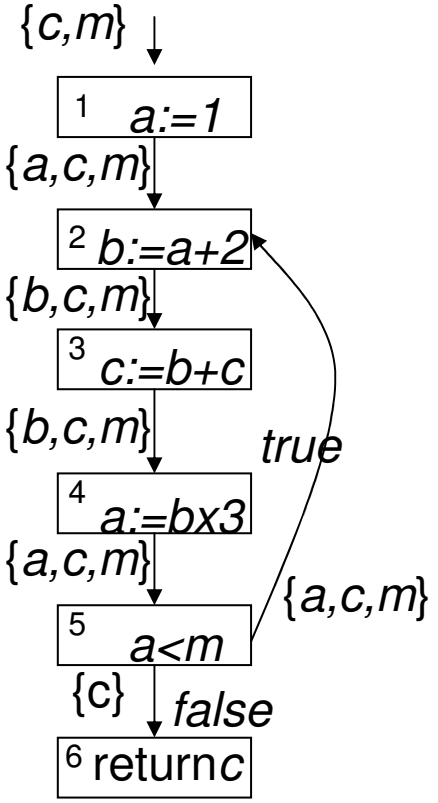
ASSEGNAZIONE DELLA MEMORIA – Se due variabili non sono mai vive contemporaneamente, NON INTERFERISCONO e possono occupare la stessa posizione in memoria o lo stesso registro. (NB: in generale, date *n* variabili ci sono n*(n-1)/2 coppie non ordinate di variabili distinte)

Le coppie (*a,c*) (*c,m*) e (*a,m*) interferiscono (sono presenti in *in(2))*

Le coppie (*b,c*) (*b,m*) e (*c,m*) interferiscono (sono presenti in *in(3))*

a e *b* non interferiscono

Per le quattro variabili a,b,c,m bastano quindi tre 'celle'



Nei moderni compilatori l'asegnazione dei registri alle variabili si fa con metodi euristici che sfruttano la relazione di interferenza.

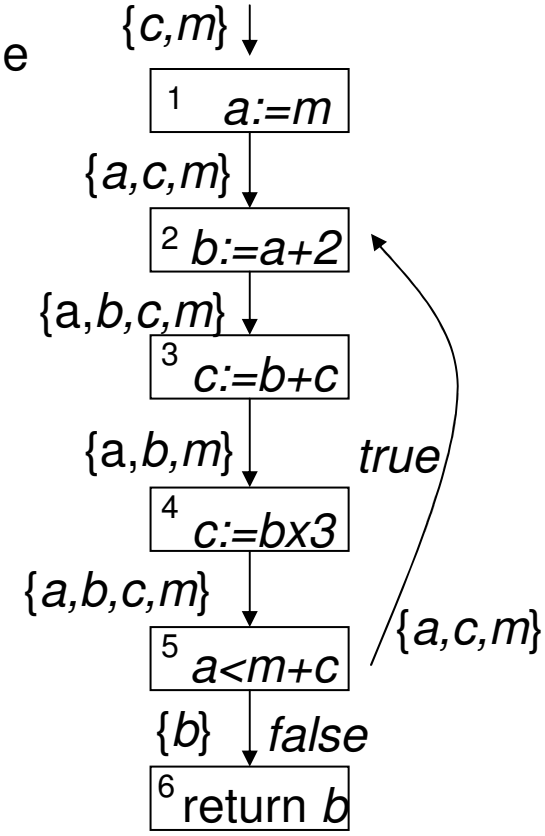
DEFINIZIONI INUTILI – Un'istruzione che definisce una variabile è inutile se la variabile non è viva all'uscita dell'istruzione.
Per cercare le istruzioni inutili si analizza ogni istruzione *p* che definisce una variabile *a* e si controlla che la variabile *a* appartenga all'insieme *out(p)*

Nell'esempio precedente nessuna definizione è inutile

ESEMPIO – Definizioni inutili

La variabile *c* non è viva all'uscita di 3,
l'istruzione 3 è inutile
Eliminando l'istruzione 3 si accorcia il programma
e *c* scompare da *in(1), in(2), in(3)* e *out(5)*

Apportare un miglioramento a un programma
potrebbe consentire una catena di ottimizzazioni



DEFINIZIONI RAGGIUNGENTI

Calcolo delle definizioni che raggiungono i vari punti del programma

DEFINIZIONE: Si dice che la definizione di *a* in *q*, a_q. raggiunge l'ingresso di un'istruzione *p* (non necessariamente distinta da *q*) se esiste un cammino da *q* a *p* che non passa per un nodo, diverso da *q*, che definisce *a.*

In tal caso l'istruzione *p* potrà usare il valore di *a* definito in *q.*

Con riferimento all'automa A (al grafo di controllo del programma), ecco la condizione:
Esiste nel linguaggio L(A) una frase *x* tale che *x=uqvpw* dove:

$$u \in I^*, q \in D(a), v \in (I \setminus D(a))^*, p \in I, w \in I^*$$

p e *q* possono coincidere

ESEMPIO precedente (p.17):
- la definizione di a_1 raggiunge l'ingresso di 2,3,4 ma non 5.
- la definizione di a_4 raggiunge l'ingresso di 5,6,2,3,4

EQUAZIONI DI FLUSSO PER LE DEFINIZIONI RAGGIUNGENTI

Calcolo delle definizioni raggiungenti i vari punti del programma formulato come soluzione di certe equazioni di flusso.

Se il nodo *p* definisce la variabile *a*, si dice che ogni altra definizione a_q, *q!=p*, della stessa variabile è *soppressa* da *p.*

L'insieme delle definizioni soppresse da *p* è:

$$sop(p) = \{a_q \mid q \in I \wedge q \neq p \wedge a \in def(q) \wedge a \in def(p)\}, \text{ se } def(p) \neq \emptyset$$

$$sop(p) = \emptyset, \qquad \text{se } def(p) = \emptyset$$

L'insieme def(p) può contenere più di una variabile nel caso (ad esempio) di istruzioni come la lettura "read(a,b,c)".

EQUAZIONI DI FLUSSO:

Per il nodo 1 iniziale:

(1) $in(1) = \varnothing$

Per ogni altro nodo $p \in I$:

(2) $out(p) = def(p) \bigcup (in(p) \setminus sop(p))$

(3) $in(p) = \coprod_{\forall q \in pred(p)} out(q)$

L'eq (1) ipotizza per semplicità che non ci siano var passate come par di ingresso. Altrimenti *in(1)* contiene le definizioni esterne al sottopr.

L'eq (2) pone nell'uscita di *p* le definizioni proprie di *p* e quelle raggiungenti l'ingresso di *p,* purché non soppresse da *p*.

L'eq. (3) dice che confluiscono all'ingresso di *p* le definizioni raggiungenti le uscite dei nodi predecessori.

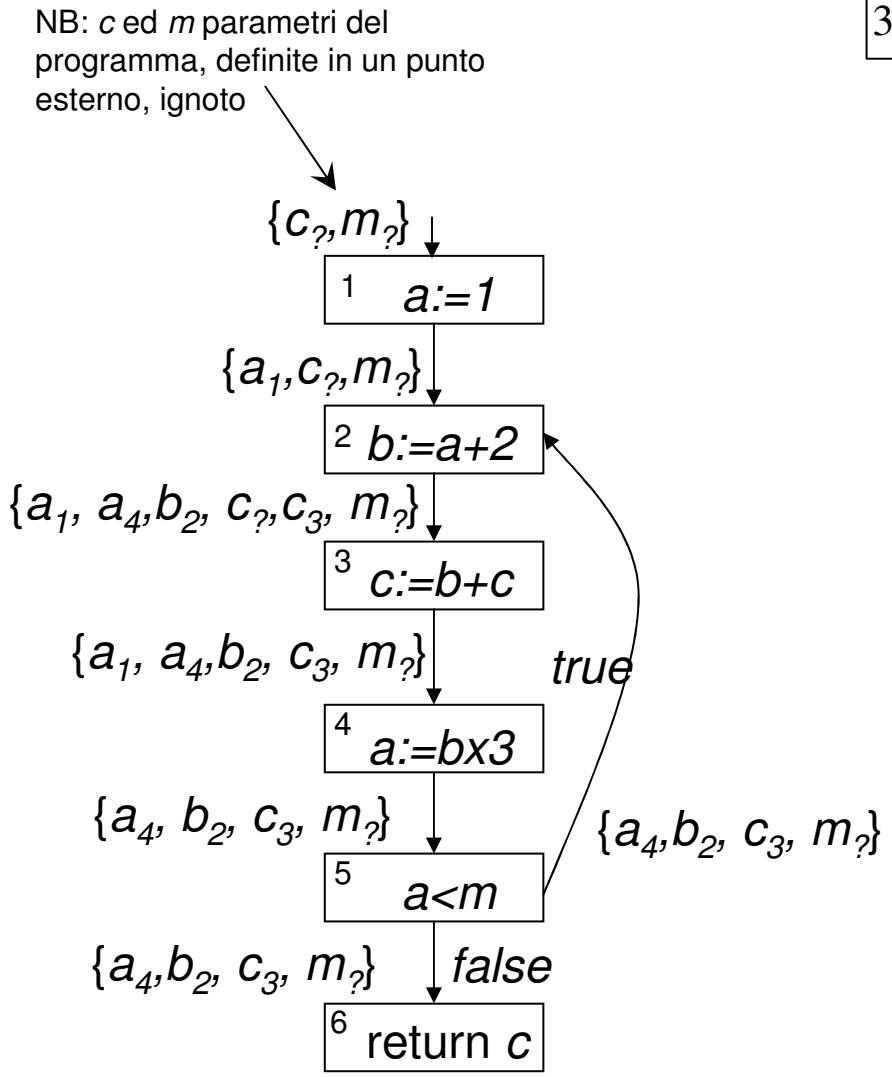
Il sistema di equazioni si risolve mediante successive iterazioni che convergono al primo punto fisso (come con le variabili vive). All'inizio tutti gli insiemi sono vuoti.

ESEMPIO – Definizioni raggiungenti

Termini costanti:

nodo	‖*def* ∣ *sop*
1 $a := 1$	‖ a_1 ∣ a_4
2 $b := a + 2$	‖ b_2 ∣ $\varnothing$
3 $c := b + c$	‖ c_3 ∣ $c_?$

nodo	‖*def* ∣ *sop*
4 $a := b \times 3$	‖ a_4 ∣ a_1
5 $a < m$	‖ $\varnothing$ ∣ $\varnothing$
6 return c	‖ $\varnothing$ ∣ $\varnothing$



$$in(1) = \{c_?, m_?\}$$
$$out(1) = \{a_1\} \cup (in(1) \setminus \{a_4\})$$
$$in(2) = out(1) \cup out(5)$$
$$out(2) = \{b_2\} \cup (in(2) \setminus \varnothing) = \{b_2\} \cup in(2)$$
$$in(3) = out(2)$$
$$out(3) = \{c_3\} \cup (in(3) \setminus \{c_?\})$$
$$in(4) = out(3)$$
$$out(4) = \{a_4\} \cup (in(4) \setminus \{a_1\})$$
$$in(5) = out(4)$$
$$out(5) = \varnothing \cup (in(5) \setminus \varnothing) = in(5)$$
$$in(6) = out(5)$$
$$out(6) = \varnothing \cup (in(6) \setminus \varnothing) = in(6)$$

PROPAGAZIONE DELLE COSTANTI

Continuando l'esempio precedente, si considera la possibilità di sostituire a una variabile un valore costante. Ad esempio non è possibile sostituire alla variabile *a* nella 2 la costante 1, assegnata dall'istruzione 1 (definizione di a_1) perché l'insieme *in(2)* contiene anche un'altra deinizione di *a,* la a_4.

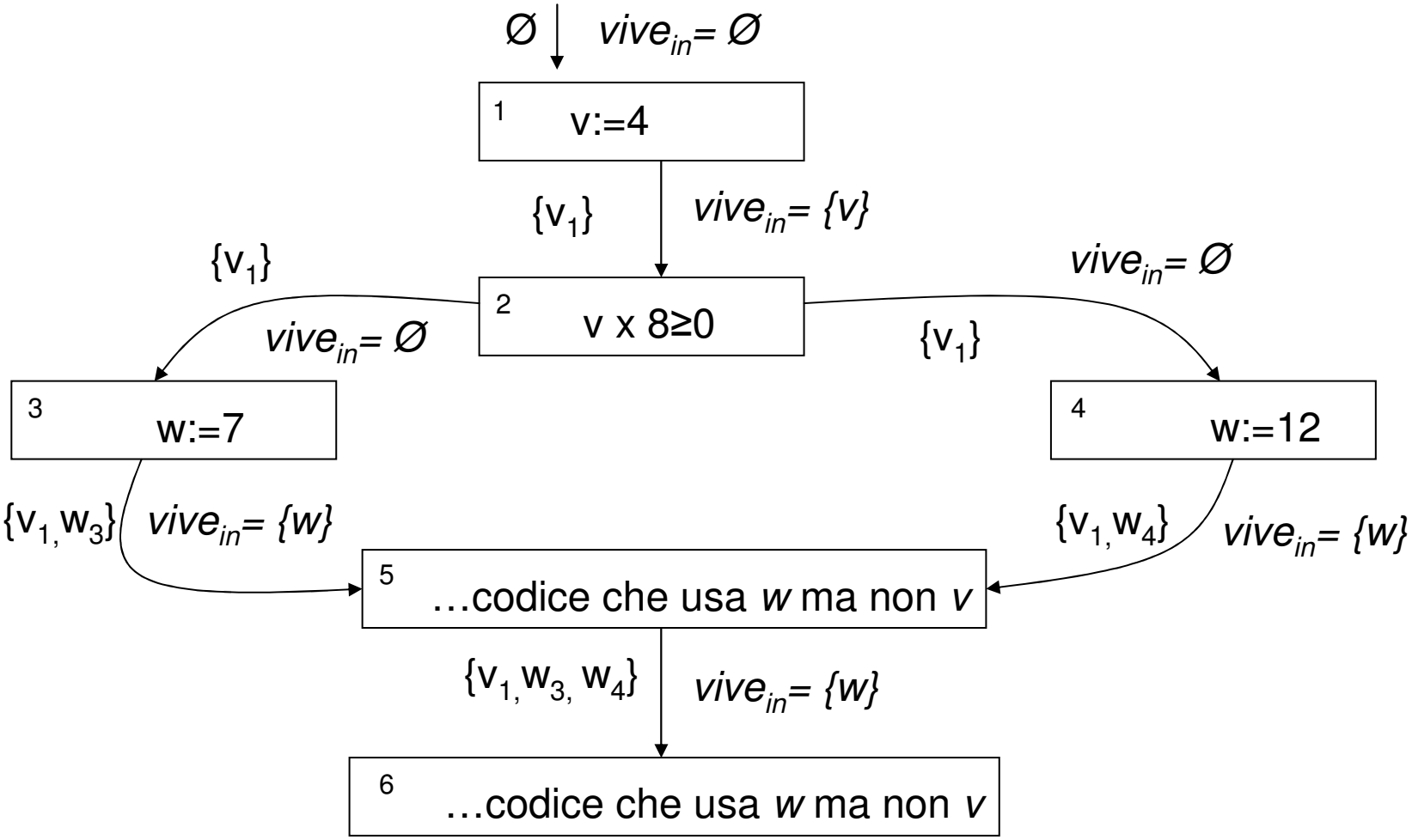
CONDIZIONE GENERALE

E' lecito sostituire nella istruzione *p*, al posto della variabile *a* ivi usata, la costante *k* se:

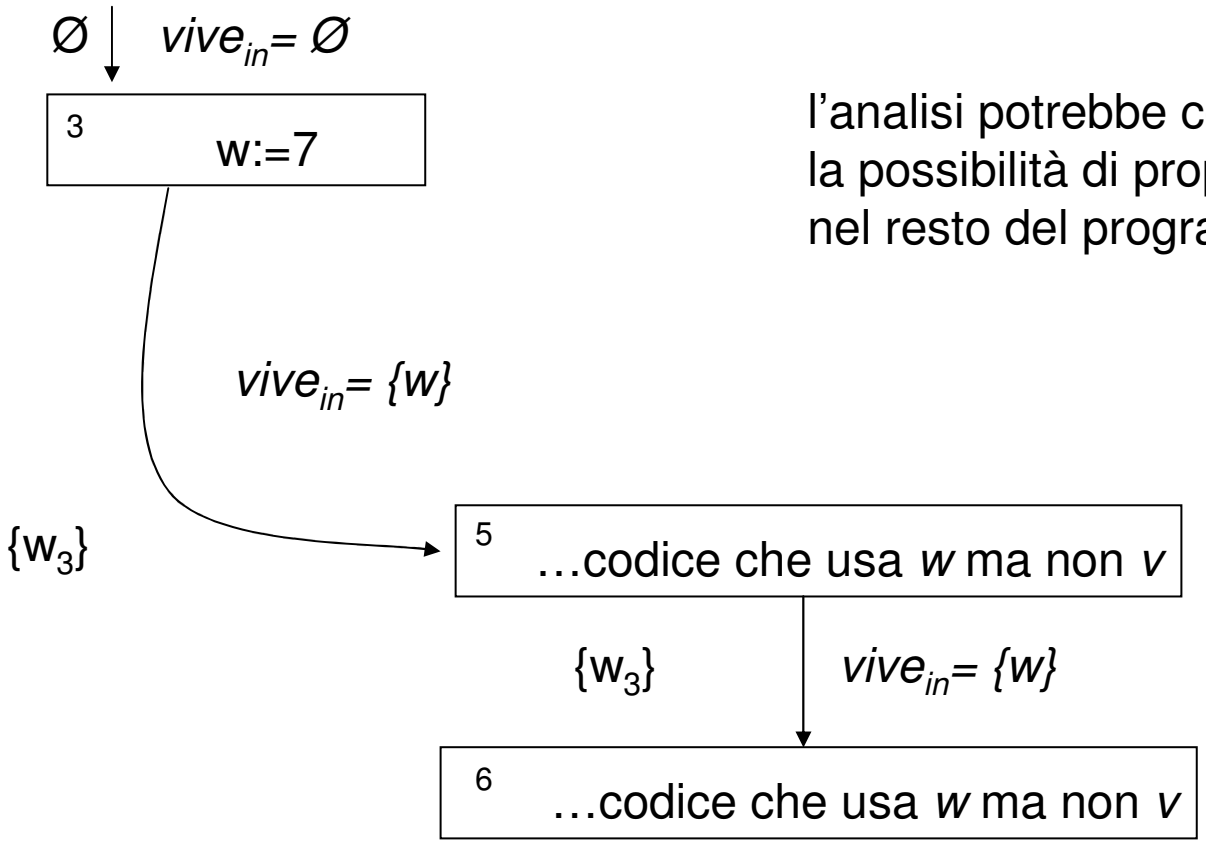
1) esiste un'istruzione *q: a:=k, k* costante, tale che la definizione a_q raggiunge l'ingresso di *p* AND
2) nessun'altra definizione a_r, *r != q* di *a* raggiunge l'ingresso di *p.*

ESEMPIO – propagazione delle costanti (miglioramenti ottenuti con propagazione delle costanti e altre semplificazioni indotte.)

definizioni raggiungenti e variabili vive:



Programma semplificato



l'analisi potrebbe continuare esaminando la possibilità di propagare la costante w=7 nel resto del programma.

DISPONIBILITA' DELLE VARIABILI E INIZIALIZZAZIONI

Il compilatore deve controllare che al momento dell'esecuzione ogni istruzione ogni variabile in essa usata abbia un valore, ottenuto tramite un assegnamento valido o una definizione.
In caso contrario la variabile risulta ***indisponibile*** e si verifica un *errore.*

ESEMPIO (p.22) – Dal grafo di controllo:

3 usa *c* a cui non viene assegnato un valore in 123
c ed *m* potrebbero essere parametri d'ingresso del sottoprogramma
a disponibile all'ingresso di 2 perché ha ricevuto un valore in 1
b disponibile all'ingresso di 3 perché ha ricevuto valore in 2

DEFINIZIONE – Una variabile *a* è disponibile all'ingresso del nodo *p*, cioè subito prima della sua esecuzione, se nel grafo di controllo ogni cammino dal nodo iniziale all'ingresso di *p*, contiene una definizione di *a*.
(trascuriamo parametri di entrata…)

DISPONIBILITA' / DEFINIZIONI RAGGIUNGENTI

Se una definizione a_q di *a* raggiunge l'ingresso di *p*, certamente esiste un cammino da 1 a *p* passante per il punto di definizione *q*. Ma non si può escludere che esista anche un altro cammino da 1 a *p*, non passante per *q* né per un'altra definizione di *a*.

Il concetto di disponibilità è più restrittivo di quello di definizione raggiungente.

Se per ogni nodo *q* predecessore di *p*, l'insieme *out(q)* delle definizioni raggiungenti l'uscita di *q* contiene una definizione della variabile *a* , essa risulta disponibile all'ingresso di *p*: cioè, **qualche definizione di *a* raggiunge sempre il nodo *p*.**

Per un'istruzione *q* indichiamo con *out'(q)* l'insieme delle definizioni raggiungenti l'uscita di *q,* private dei pedici. Se $out(q)=\{a_1,a_4,b_3,c_6\}$, $out'(q)= \{a,b,c\}$

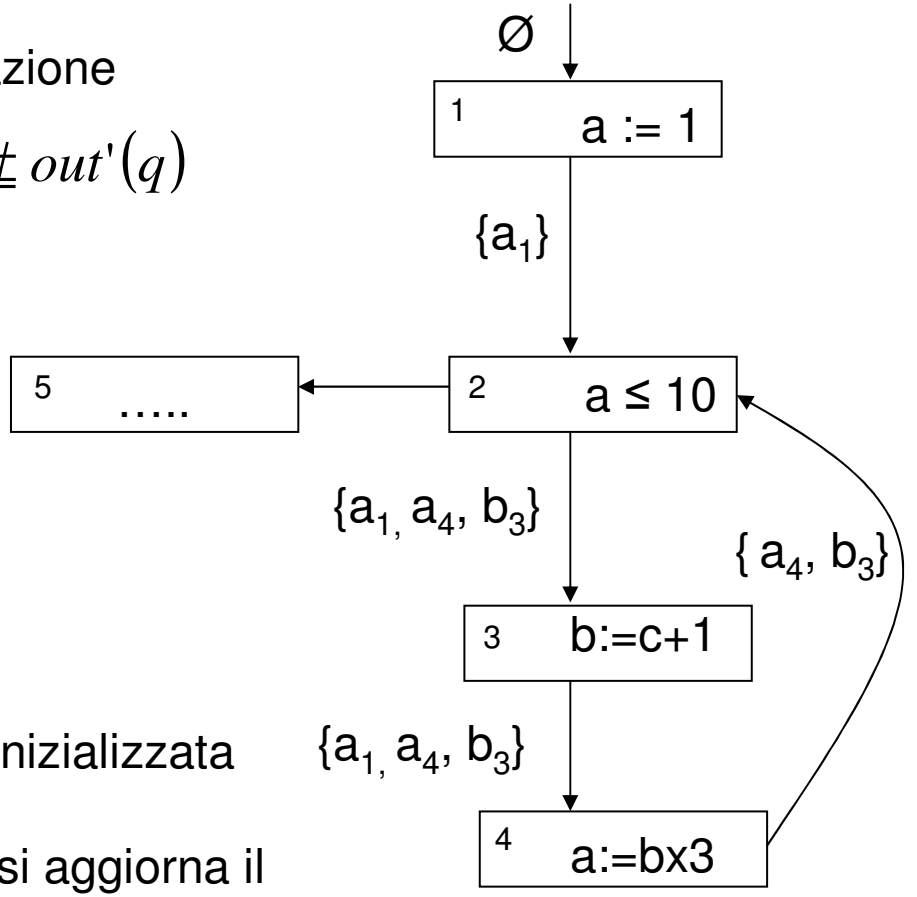
NON BUONA INIZIALIZZAZIONE: Un'istruzione *p* non è ben inizializzata se esiste un predecessore *q* di *p*, le cui definizioni raggiungenti non comprendono tutte le variabili usate in *p* (quando il calcolo segue il cammino che passa per *q* una o più variabili usate in *p* saranno prive di valore). ***p non è ben inizializzata se:***

$$\exists q \in pred(p) \quad tale\,che \quad usa(p) \not\subseteq out'(q)$$

ESEMPIO – Scoperta di variabili non inizializzate

La condizione di non buona inizializzazione

$$\exists q \in pred(p) \quad tale\,che \quad usa(p) \not\subseteq out'(q)$$



E' falsa nel nodo 2:
ogni suo predecessore (1 e 4)
contiene nel proprio insieme *out'*
una definizione di *a,* la sola variabile
usata in 2.

E' vera nel nodo 3, perchè non vi sono
definizioni di *c* raggiungenti l'uscita di 2
→Errore in 3: l'istr. fa uso di una var non inizializzata

Si cancella l'istruzione erronea 3 trovata, si aggiorna il calcolo degli insiemi delle definizioni raggiungenti e si rivaluta la condizione (4 non è ben inizializzata – la def b_3 di out(3) non è disponibile.)

Cancellando 4 e proseguendo nello stesso modo non si trovano altri errori.