Static Flow Analysis of Programs

Prof. Licia Sbattella

aa 2007-08

Translated and adapted by L. Breveglieri

STATIC FLOW ANALYSIS OF PROGRAMS

Static flow analysis is a technique widely employed by compilers to translate a high level source program into machine code.

COMPILATION:

FIRST STAGE: translates the program into an intermediate form, more similar to the machine language.

NEXT STAGE – is applied to the program in intermediate form and has several possible aims:

- VERIFICATION – refined check of the correctness of the program
- OPTIMIZATION – transformation of the program to improve execution efficiency (e.g. register allocation, etc)
- SCHEDULING – change of the order of the instruction flow, to improve execution efficiency and pipeline utilization.

It is convenient to model the *control flow graph* of the program as an automaton, and to process such an automaton to carry out the above mentioned tasks.

CAUTION: the automaton defines a single program, and not the entire source language ! Static flow analysis is something different form *syntax driven translation*.

IN THE PRESENT CASE: a string that is recognisd by the automaton represents the time sequence of the operations the program can execute, i.e. an execution trace.

STATIC FLOW ANALYSIS: consists of studying some properties of the program, by means of the methods and techniques of automata theory, formal logic or statistics. Here only automata theory is of interest.

THE PROGRAM AS AN AUTOMATON

- consider only the simplest instructions (those in intermediate form):
  - scalar variable and constant
  - assignment to a variable
  - simple unary arithmetic, logic or relational operation
- and consider only INTRAPROCEDURAL ANALYSIS (not INTERPROCEDURAL)

PROGRAM CONTROL GRAPH:
- every node represents an instruction
- if at execution time instruction *p* is immediately followed by instruction *q*, the graph has an arc directed from *p* to *q* (*p* is the *predecessor* of *q*)
- the first instruction of the program is the input node of the graph (*initial* node)
- an instruction without successors is an output node of the graph (*final* node)
- unconditional instructions have a unique successor, conditional ones have two or more successors
- an instruction with two or more predecessors is a *confluence* node (or merge or join node)

THE CONTROL GRAPH is not a complete and faithful representation of the program:

- the logical value of a condition (TRUE or FALSE) to select the appropriate successor of a conditional instruction, is not represented
- the assignment operation is replaced by the following abstractions:
  - assigning or reading a variable, defines that variable
  - referencing a variable in the right member of an assignment (expression or write operation), uses that variable
  - each node *p* of the control flow graph is associated with two sets:

def(*p*) and usa(*p*)

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

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

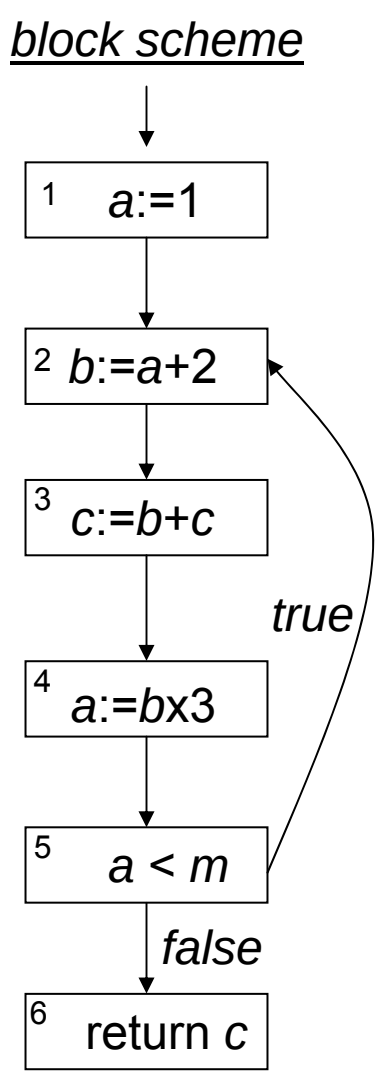
$q: \quad read(a) \qquad def(q) = \{a\}, usa(q) = \varnothing$

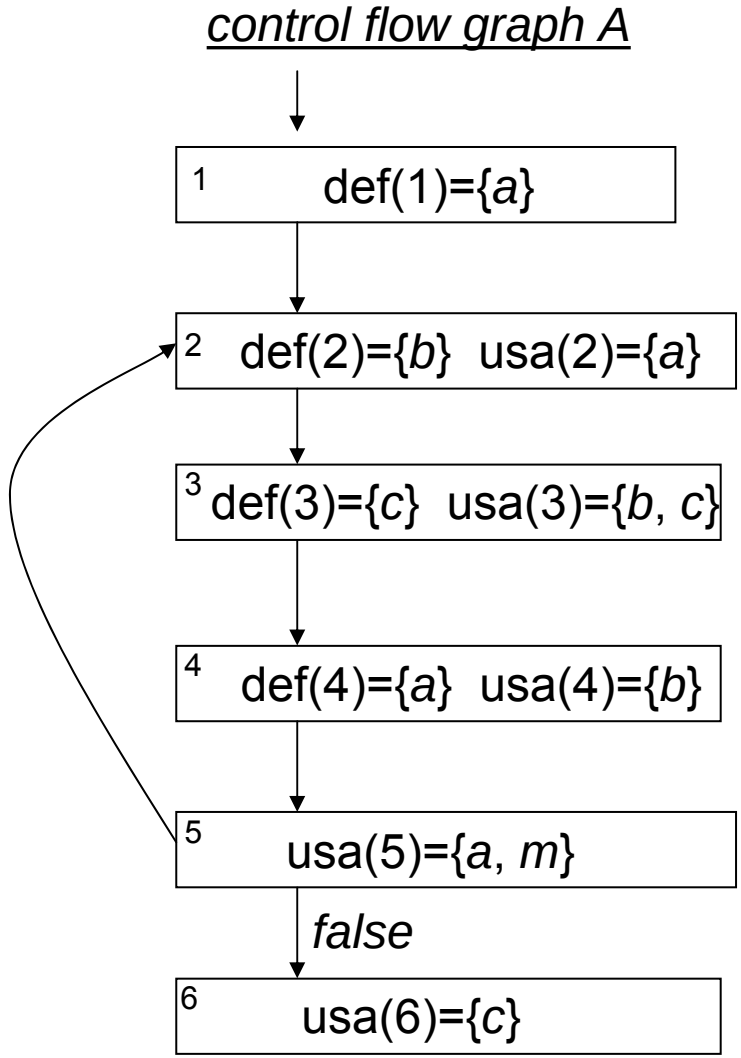
$w: \quad a := 7 \qquad def(w) = \{a\}, usa(w) = \varnothing$

EXAMPLE – block scheme of a program and related control flow graph

program

```
     a := 1
e1:  b := a + 2
     c := b + c
     a := b x 3
     if a < m goto e1
     return c
```





DEFINITION: LANGUAGE OF THE CONTROL GRAPH

The finite state automaton A, represented by the control flow graph, has an instruction set I as *terminal alphabet*, and each instruction is defined as:

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

- *initial state*: the state without predecessors
- *final state*: a state without successors

THE FORMAL LANGUAGE L(A) recognised by the automaton contains the strings of alphabet I labeling a path from the input node to an output node of the graph. SUCH A PATH represents a sequence of instructions the machine can execute when the program is run.

L(A) is a LOCAL language (a subfamily of REG), because each node has a distinguished label, unique to that node.

PREVIOUS EXAMPLE: I = {1 … 6}

A recognised path is 1 →2 → 3→ 4→ 5→ 2→ 3→4 → 5→6 = 1234523456

The set of all the paths forming the language is $1(2345)^{+}6$ (is a regexp)

PRECAUTIONAL APPROXIMATION

It is not necessarily true that every path in the graph is an executable sequence of instructions, as the control flow graph does not model the clauses of the conditional instructions.

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

The formal language accepted by the automaton contains the two paths { 12, 13 }, but path 13 is actually not executable (a square can not be negative).

IN GENERAL IT IS UNDECIDABLE WHETHER a path in the control flow graph is executable or not (as the Turing halting problem is reducible to it).

PRECAUTIONAL DIAGNOSIS: examining all the paths from the input to an output node may lead to diagnose inexistent errors, or to the precautional assignment of unnecessary resources; however effective errors are always diagnosed and in conclusion the method is *error safe*, though sometimes inefficient.

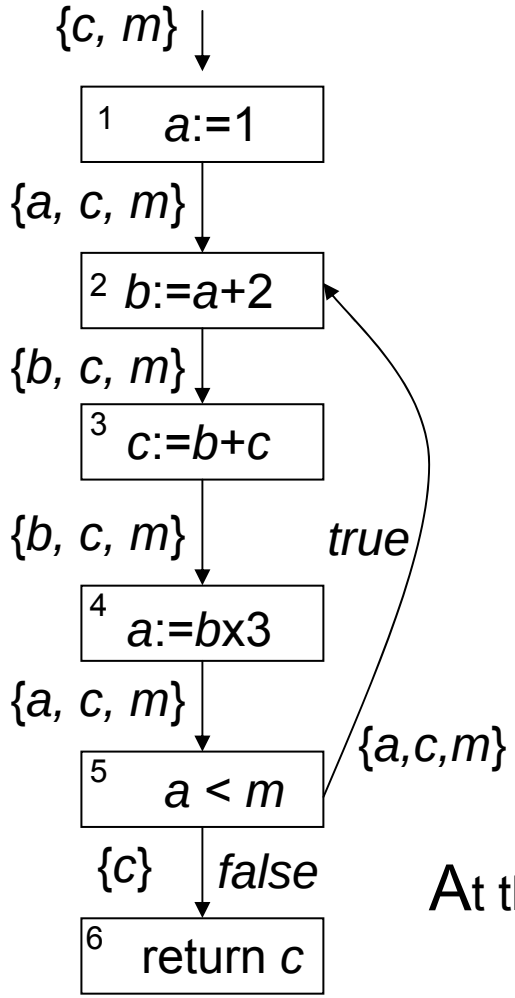
HYPOTHESIS: in static flow analysis the automaton MUST be in reduced form (all nodes must be useful: reachable and defined).

IF THIS IS NOT TRUE:
- the program may not terminate
- the program may contain unreachable instructions (so-called dead code)

LIVENESS INTERVAL OF A VARIABLE

DEFINITION: a variable *a* is live at a point *p* of the program if in the control graph there exists a path from *p* to another point *q*, such that both conditions below hold:
- the path does not pass through any instruction *r,* with *r* != *q,* that defines variable *a*, i.e. an instruction *r* such that $a \in$ def(*r*)
- instruction *q* uses variable *a,* i.e. it holds $a \in$ usa(*q*)

Informally, a **variabile is live at a point** if some instruction that could be executed starting from that point, uses (= is influenced) by the value of the variable in that point.



← ***variables live on the arcs***

A variable is:
- ***live at the ouput of a node*** if it is live on at least one of the arcs outgoing from the node
- ***live at the input of a node*** if it is live on at least one of the arcs ingoing into the node

EXAMPLE*:*

c is live at the input of node 1 because there exists path 123: $c \in$ usa(3) and neither 1 nor 2 define *c*

a is live on both paths 12 and 452,
a is not live on either path 234 or 56

At the output of node 5, var.s {*a, c, m*} U {c} are all live.

HOW TO COMPUTE LIVENESS INTERVALS – let I be the set of instructions, and let D(*a*) and U(*a*) be the sets of the instructions that define and use variable *a*.

The property that *a* is live at the output of node *p* is equivalent to the following condition for the language accepted by the automaton.

there exists in L(A) a phrase *x* = *upvqw* such that

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

the set of all the phrases *x* that satisfy the condition above is a regular language

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

The expression above prescribes that character *p* is followed by a charecter *q* chosen from U(*a*), and that the characters occurring between *p* and *q* do not belong to D(*a*).

To check whether *a* is live at the ouput of *p*, just check whether language L_p is empty.

To check whether L_p is empty, one can build the recognizer of L_p (as a cartesian product machine of A and the recogniser of R_p) and check whether there are paths connecting the input node to some final node.

But this method may be time consuming for large programs.

FLOW EQUATION METHOD – simultaneously determine all the live variables.

This method examines the existence of some paths connecting the definition of a variable to the point where it is used.

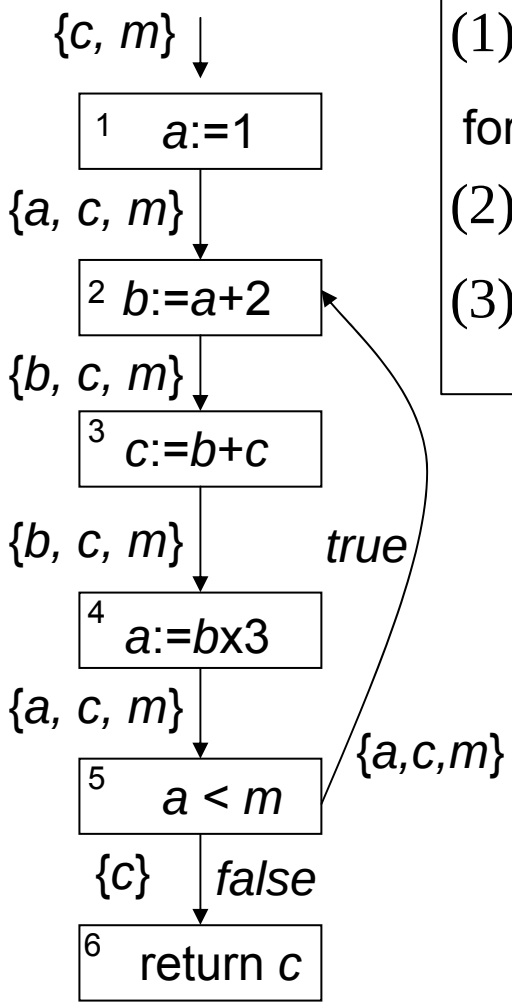
for every final node *p*:

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

for every other node *p*:

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

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



for every final node *p*:

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

for every other node *p*:

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

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

For (1): no variable is live at the graph output

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

For (3): succ(5) = {*2, 6*}

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

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

HOW TO SOLVE FLOW EQUATIONS

For a graph of $|I| = n$ nodes, one has a set of $2 \times n$ equations in $2 \times n$ unknowns, i.e. $vive_{in}(p)$, $vive_{out}(p)$, $p \in I$

The solution of the equation set is a family of $2 \times n$ sets of variables.

One can solve the equation set iteratively, assigning initially the empty set to each unknown (this is iteration number $i = 0$):

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

Replace in the current solution i the values obtained by the equations, and number the new solution as $i + 1$. If the two solutions differ, go on again. Otherwise the procedure has converged to the final solution.

The computation must converge after a finite number of iteration steps:

1) every set $vive_{in}(p)$ and $vive_{out}(p)$ has a cardinality that is upper bounded by the total number of variables in the program

2) every iteration step either increases the cardinality of the above sets, or at least leaves it unchanged (is a monotonic algorithm)

3) when the solution does not change any longer, the algorithm terminates

EXAMPLE – iterative computation of live variables

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	$in(6) = \{c\}$	$out(6) = \varnothing$

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

Sets of instructions that define and use the variables.

	$in = out$	in	out	in	out	in	out	in	out	in	out
1	$\varnothing$	$\varnothing$	a	$\varnothing$	a,c	c	a,c	c	a,c,m	c,m	a,c,m
2	$\varnothing$	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	$\varnothing$	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	$\varnothing$	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	$\varnothing$	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	$\varnothing$	c	$\varnothing$	c	$\varnothing$	c	$\varnothing$	c	$\varnothing$	c	$\varnothing$

OTHER POSSIBLE APPLICATIONS OF THE FLOW EQUATION METHOD

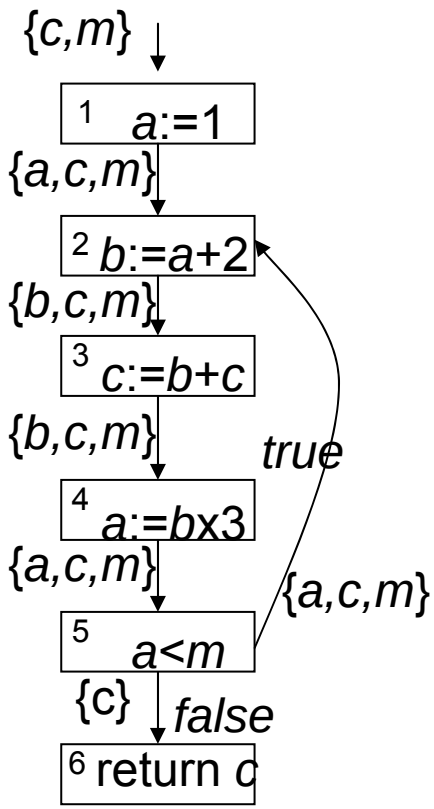
MEMORY ALLOCATION – if two variables are never simultaneously live, they do not interfere, and therefore can be allocated in the same register or memory location.

Pairs *a, c* and *c, m* interfere (both are in in(2)).

Pairs *b, c* and *b, m* interfere (both are in in(3)).

a and *b* do not interfere.

For the four variables *a*, *b*, *c* and *m*, three locations only are sufficient.



Modern compilers allocate variables by using the above relationship and heuristic methods.

UNUSEFUL DEFINITION – an instruction that defines a variable is *unuseful* if the variable is not live on at least one outgoing arc of the instruction.

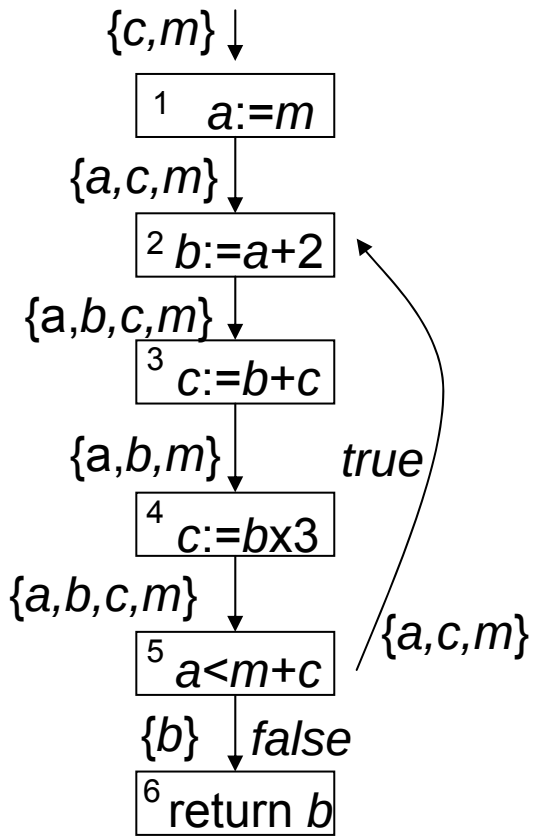
To identify unuseful instructions, search an instruction *p* that defines a variable *a*, and then check whether *a* belongs to the set out(*p*).

In the previous example there are not any unuseful instructions.

EXAMPLE – unuseful instructions

Variable *c* is not live at the output of node 3, hence instruction 3 is unuseful.
If instruction 3 is eliminated, the program is shortened and *c* vanishes from in(1), in(2), in(3) and out(5).

Sometimes after optimizing a program, new optimizations become possible.



REACHING DEFINITION

Analysis of the definitions that reach the points of the program.

DEFINITION: the definition of *a* in *q*, i.e. a_q, *reaches* the input of an instruction *p* (not necessarily different from *q*) if there exists a path from *q* to *p* that does not pass through a node, different from *q*, that defines *a* again.

Instruction *p* will be able to use the value of *a* defined in *q.*

With reference to automaton A (control flow graph), here follows the condition:
In L(A) there is a phrase *x* such that *x* = *uqvpw*, and:

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

p and *q* may coincide

previous EXAMPLE:
- the definition of a_1 reaches the input of 2, 3, 4 but not of 5.
- the definition of a_4 reaches the input of 5, 6, 2, 3, 4

FLOW EQUATIONS FOR REACHING DEFINITIONS

The computation of reaching definitions in a program can be reformulated in terms of a set of flow equations.

If node *p* defines variable *a*, every other definition a_q, with *q != p*, of the same variable *a*, is said to be *suppressed* by *p.*

The set of the definitions suppressed by *p* is the following:

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

$$sop(p) = \emptyset, \quad \text{if } def(p) = \emptyset$$

The set def(*p*) may contain two or more variable names, for instance in the case of a read instruction like “read (*a*, *b*, *c*)”.

FLOW EQUATIONS:

for the initial node 1:

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

for every other node $p \in$ I:

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

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

Equation (1) supposes that there are not any variables passed as input parameters.

Otherwise in(1) contains also definitions external to the subprogram.

Equation (2) inserts into the output of *p* the definitions of *p* and those that reach the input of *p,* provided they are not suppressed by *p*.

Equation (3) states that the definitions that reach the outputs of the predecessor nodes of *p*, are merged into *p* itself.

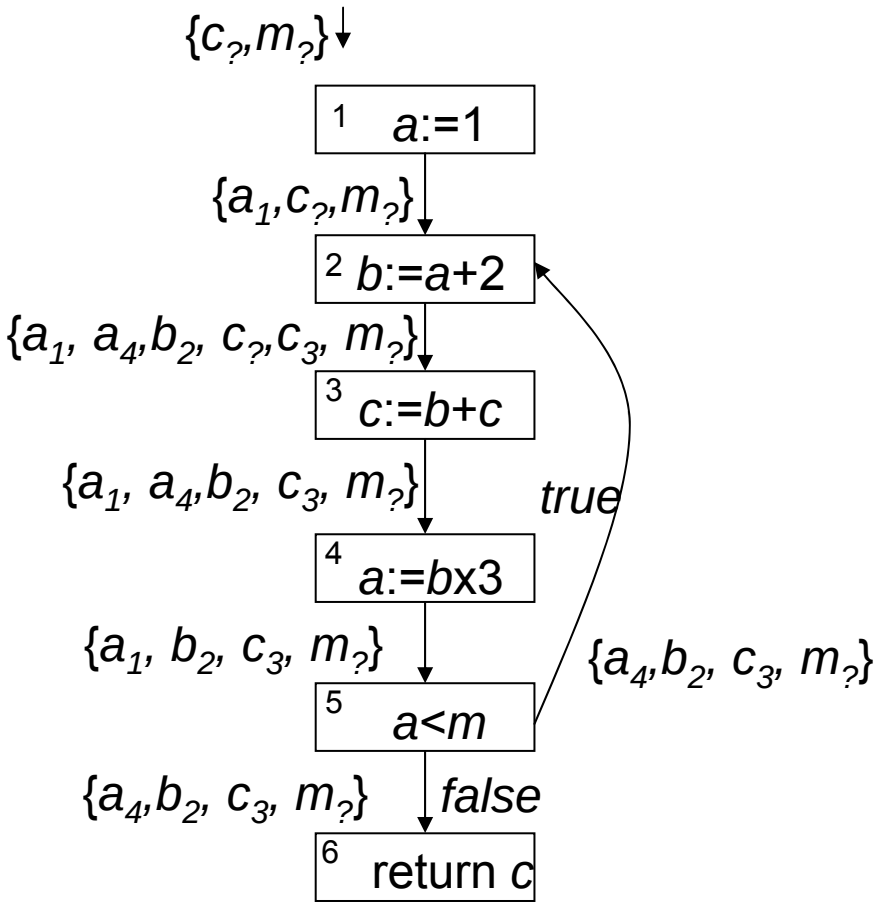
Solve the above system of equations iteratively, similarly to live variables.

EXAMPLE – reaching definitions

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	$\|\| \varnothing$	$\| \varnothing$

Constant terms:



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

CONSTANT PROPAGATION

Going on with the previous example, consider the possibility of replacing a constant value to a variable. For instance, in instruction 2 it is not possible to replace to the variable *a* the constant 1, which is assigned to *a* in instruction 1 (definition a_1), because set in(2) contains also another definition of the same variable *a* (definition a_4).

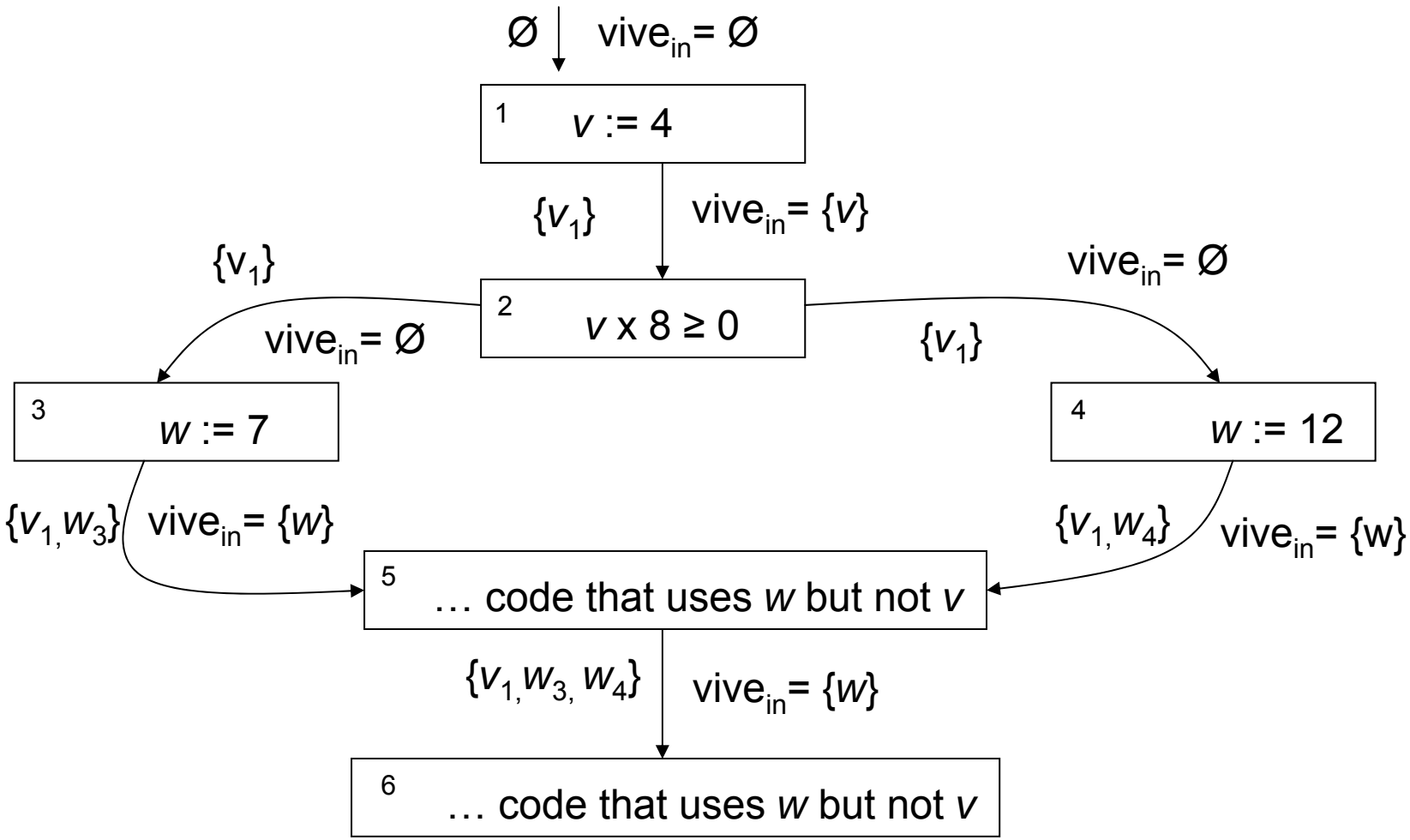
GENERAL CONDITION

In instruction *p* it is possible to replace the constant *k* to the variable *a*, which is used in *p*, if:

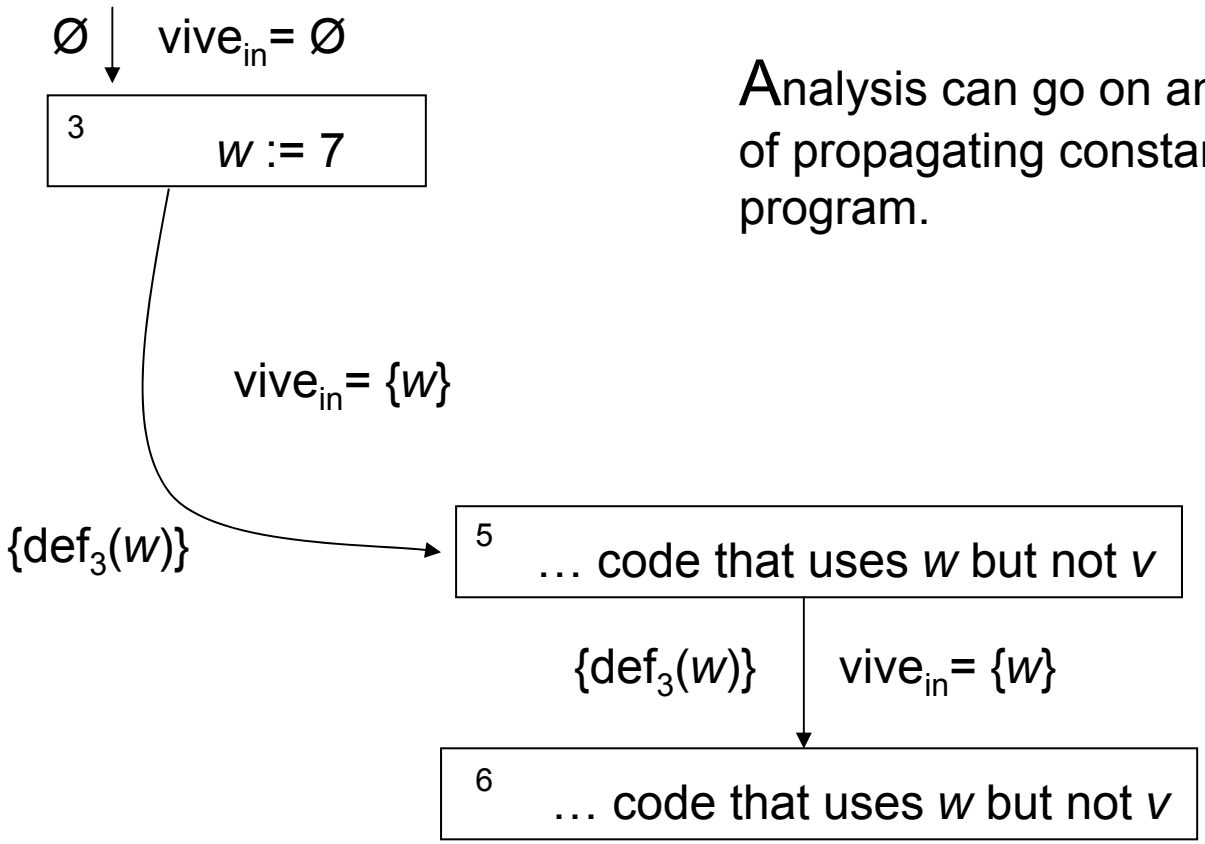
1. there exists an instruction *q*, where *a* := *k* with *k* constant, such that definition a_q reaches the input of *p*
2. there is not any other instruction *r*, with *r* != *q*, containing a definition a_r of *a*, such that a_r reaches the input of *p*

EXAMPLE – constant propagation (optimizations obtained by means of constant propagation and other induced optimizations)

reaching definitions and live variables:



Simplified progran



Analysis can go on and examine the possibility of propagating constant $w = 7$ in the rest of the program.

VARIABLE AVAILABILITY AND INIZIALIZATION

Compilers need check whether every variable that is used in each instruction has a value, when the instruction is executed. Such a value must come from a previous assignment, or from a definition. Otherwise the variable is unavailable and an error occurs.

EXAMPLE 1 – in the control flow graph:

3 uses *c*, and *c* is not assigned any value in 1 2 3
c and *m* are input parameters to the subprogram
a is available at the input of 2 because it is assigned in 1
b is available at the input of 3 because it is assigned in 2

DEFINITION – a variable *a* is *available* at the input of node *p*, that is soon before being executed, if in the control graph every path from the initial node to the input of *p*, contains a definition of *a*.

AVAILABILITY / REACHING DEFINITION

If a definition a_q of a reaches the input of p, there must exist a path from 1 to p, which passes through the point where q is defined. But one can not exclude the existence of another path from 1 to p, not passing through either q or any other definition of a.

The notion of availability is more restrictive than that of reaching definition.

If through any node q that is predecessor to p, the set out(q) of the definitions that reach the output of q contains a definition of variable a, such a variable is available at the input of p → some definitions of a reach node p

For an instruction q, call out'(q) the set of the definitions that reach the output of q, and cancel the pedices. If out(q) = { a_1, a_4, b_3, c_6 }, then out'(q) = {a, b, c}.

BAD INIZIALIZATION: an instruction p is *badly initialized* if there exists a predecessor q of p, whose reaching definitions do not include all the variables used in p (in a computation passing through q one or more variables used in p will not have any value). *p is badly initialized if:*

$$\exists q \in pred(p) \quad \text{such that} \quad usa(p) \not\subset out'(q)$$

EXAMPLE – discovery of the variables with bad inizialization

Condition of bad initialization:

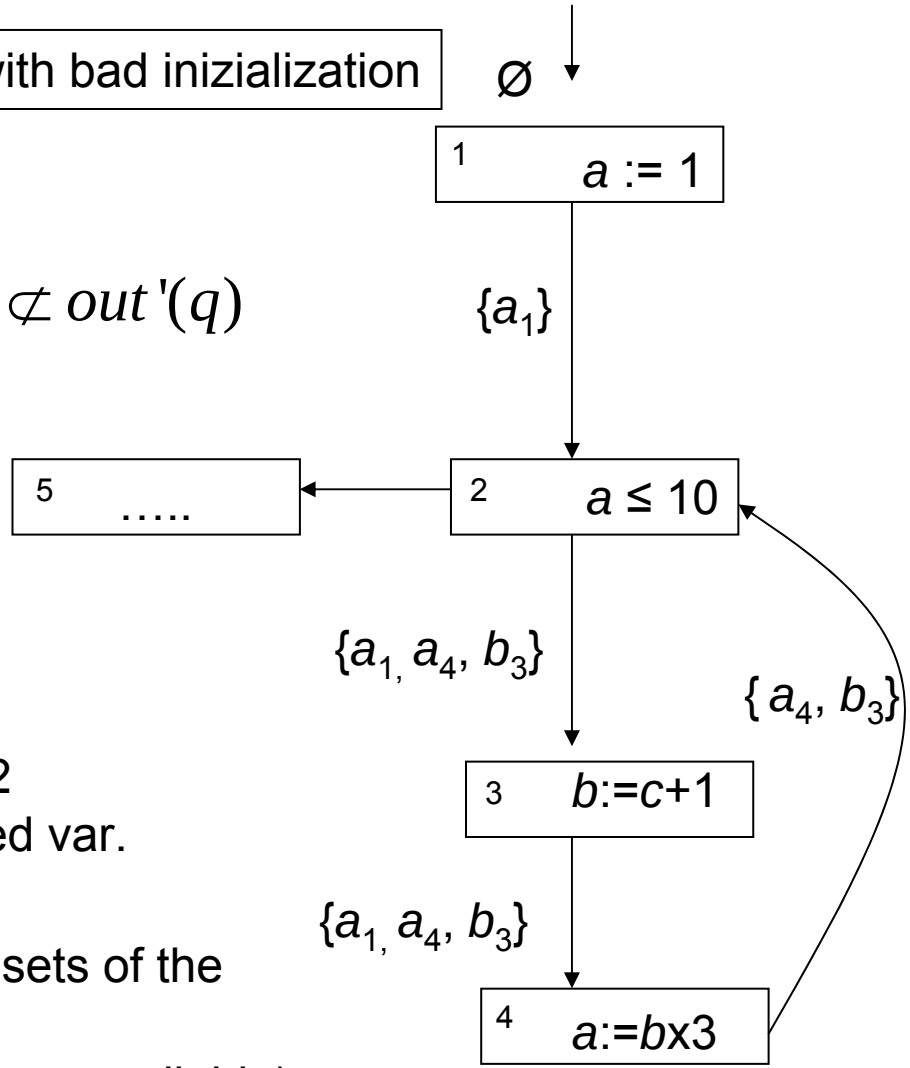
$\exists q \in pred(p)$ such that $usa(p) \not\subset out'(q)$

False in node 2:
every predecessor (1 and 4)
contains in out' a definition of *a,*
the only variable used in 2.

True in node 3, as there are not any
definitions of *c* that reach the output of 2
→ error in 3: instr. uses a badly initialized var.

Cancel wrong instruction 3, update the sets of the
reaching definitions and check again
(4 is badly initialized – def. b_3 of out(3) is unavailable).

Remove 4 and check again, but no other errors are found.



Bibliography

- S. Crespi Reghizzi, *Linguaggi Formali e Compilazione*, Pitagora Editrice Bologna, 2006
- Hopcroft, Ullman, *Formal Languages and their Relation to Automata*, Addison Wesley, 1969
- A. Salomaa – *Formal Languages*, Academic Press, 1973
- D. Mandrioli, C. Ghezzi – *Theoretical Foundations of Computer Science*, John Wiley & Sons, 1987
- L. Breveglieri, S. Crespi Reghizzi, *Linguaggi Formali e Compilatori: Temi d'Esame Risolti,* web site (eng + ita)