Constant propagation is example of FORWARD-FLOW/ALL-PATHS problem.

Intuitively, data is propagated forward in CFG, and value is constant at a point p only if it is the same constant for all paths from start to p.

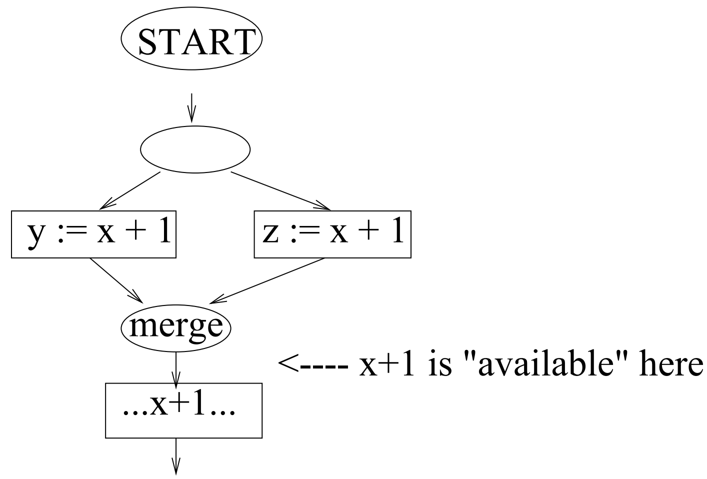
General classification of dataflow problems:

	ALL PATHS	ANY PATH
FORWARD	constant propagation available expressions	reaching definitions
BACKWARD	very busy expressions	live variables

Available expressions:FORWARD FLOW, ALL PATHS

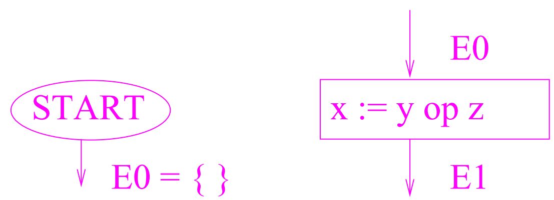
Definition: An expression 'x op y' is available at a point p if every path from START to p contains an evaluation of p after which there are no assignments to x or y.

Lattice: powerset of all expressions in program ordered by containment



Lattice: powerset of all expressions in procedure

EQUATIONS:



E1 = {y op z} U (E0 - Ex)

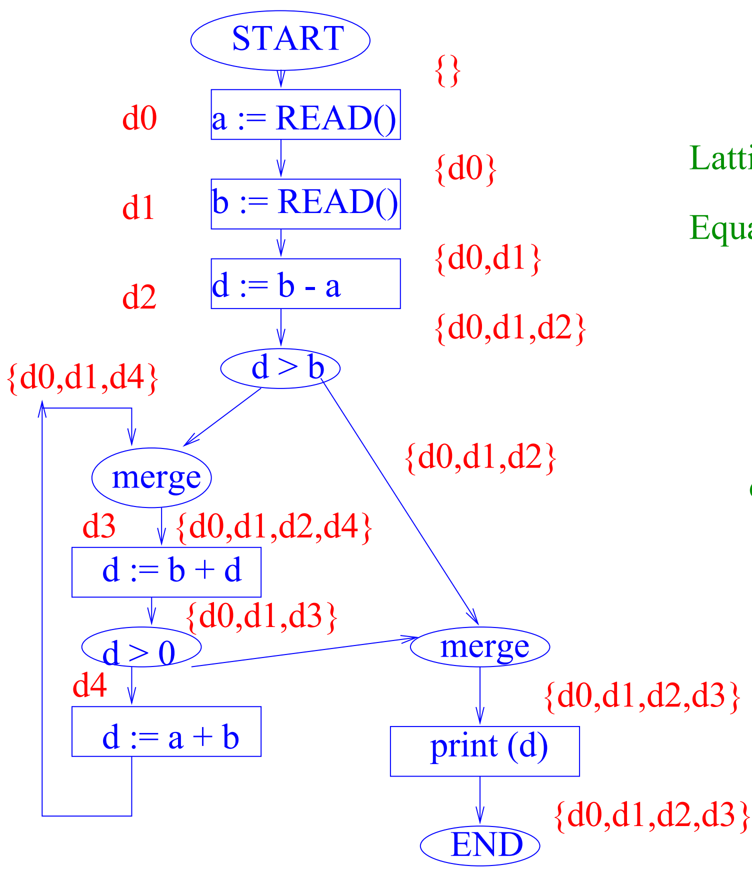
(where Ex is all expressions involving x)

confluence operator: meet (intersection)

compute greatest solution

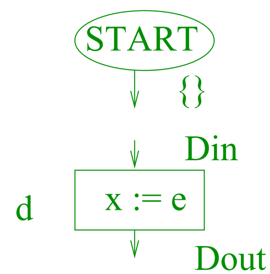
Reaching definitions:FORWARD FLOW, ANY PATH

A definition d of a variable v is said to reach a point p if there is a path from START to p which contains d, and which does not contain any definitions of v after d.



Lattice: powerset of definitions in procedure

Equations:



Dout = {d} U (Din - dx)

(dx is set of all definitions of x)

Confluence operator: join (union)

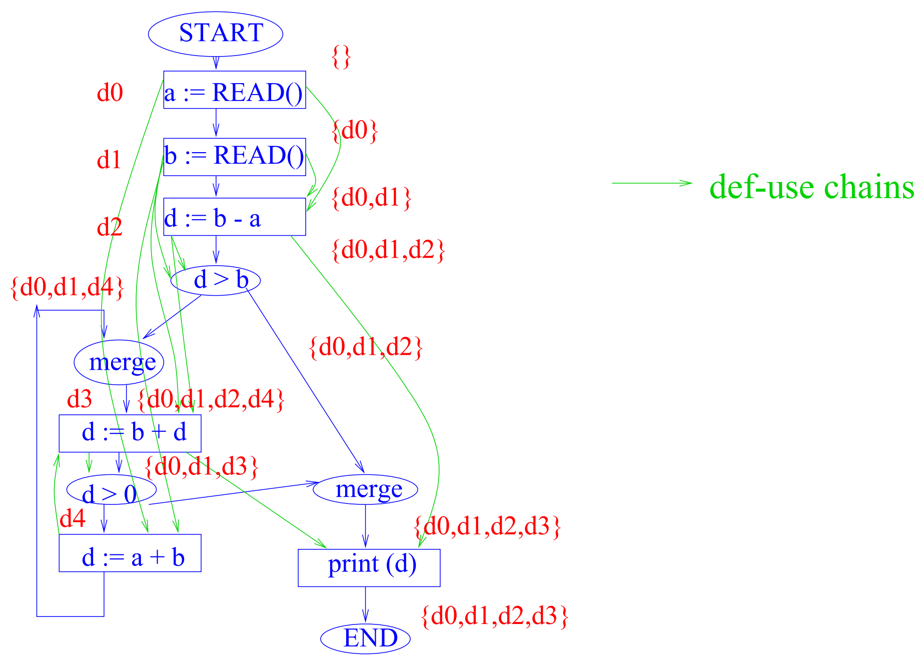
Compute least solution

Complexity: D*E*D (D is number of definitions)

Many intermediate representations record reaching definitions information in graphical form.

def-use chain: edge whose source is a definition of variable v, and whose destination is a use of v reached by that definition

use-def chain: reverse of def-use chain

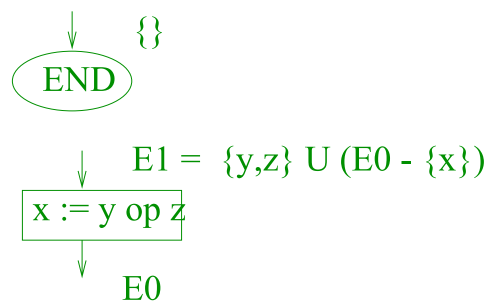


Live variable analysis:BACKWARD FLOW, ANY PATH

A variable x is said to be live at a point p if x is used before being assigned on some path from p to END (used in register allocation).

Lattice: powerset of variables ordered by containment

Equations:



Confluence operator: join (union)

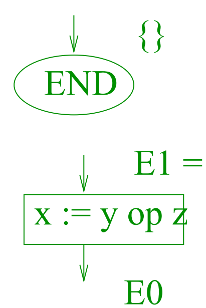
Compute least solution

Very busy expressions: FORWARD FLOW, ALL PATHS

An expression e (= y op z) is said to be very busy at a point p if it is evaluated on every path from p to END before an assignment to y or z.

Lattice: powerset of expressions ordered by containment

Equations:



E1 = {y op z} U (E0 - Ex)

(Ex is set of expressions containing x)

Confluence operator: meet (intersection)

Compute greatest solution

Pragmatics of dataflow analysis:

- Compute and store information at basic block level.
- Use bit vectors to represent sets.

Question: can we speed up dataflow analysis?

Two approaches:

- exploit structure in control flow graph
- exploit sparsity

Optimizing Dataflow Analaysis

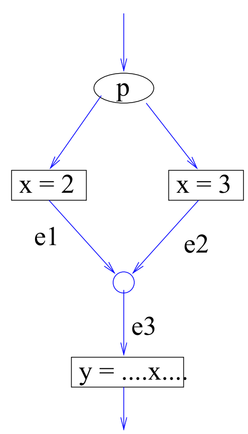
Constant propagation on CFG: $O(EV^2)$
Reaching definitions on CFG: $O(EN^2)$
Available expressions on CFG: $O(EA^2)$

Two approaches to speeding up dataflow analysis:

- exploit structure in the program
- exploit sparsity in the dataflow equations: usually, a dataflow equation involves only a small number of dataflow variables

Exploiting program structure

- Work-list algorithm did not enforce any particular order for processing equations
- Should exploit program structure to avoid revisiting equations unnecessarily



- we should schedule e3 after we have processed e1 and e2; otherwise e3 will have to be done twice

- if this is within a loop nest, can be a big win

General approach to exploiting structure: elimination

- Identify regions of CFG that can be *preprocessed* by collapsing region into a single node with the same input-output behavior as region
- Solve dataflow equations iteratively on the collapsed graph.
- Interpolate dataflow solution into collapsed regions.

What should be a region?

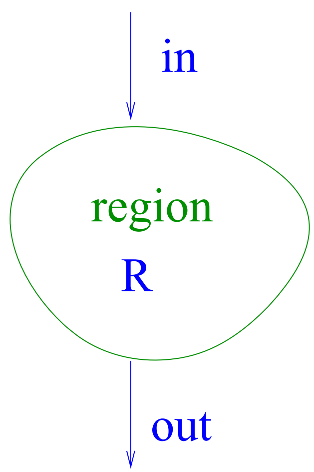
- basic-blocks
- basic-blocks, if-then-else, loops
- intervals
- ....

Structured programs: limit in which no iteration is required

Example: reaching definitions in structured language

To summarize the effect of a region, compute gen and kill for region.

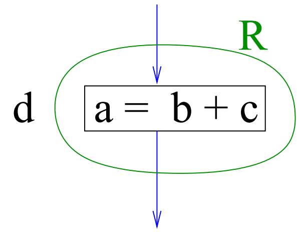
Dataflow equation for region can be written using gen and kill:



gen[R]: set of definitions in R from which there is a path to exit free of other definitions of the same variable

kill[R]: set of definitions in program that do not reach exit of R even if they reach the beginning of R

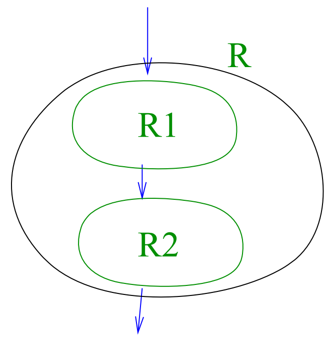
out = gen[R] U (in - kill[R])



gen[R] = {d}

kill [R] = Da (all definitions of a)

out[R] = gen[R] U (in[R] - kill[R])

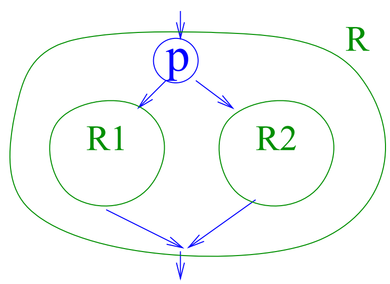


gen[R] = gen[R2] U (gen[R1] - kill[S2])

kill[R] = kill[R2] U kill[R2]

in[R1] = in[R]

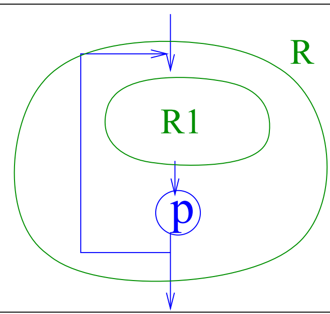
in[R2] = gen[R1] U (in[R] - kill[R1])



gen[R] = gen[R1] U gen[R2]

kill[R] = kill[R1] ∩ kill[R2]

in[R1] = in[R2] = in[R]



gen[R] = gen[R1]

kill[R] = kill[R1]

in[R1] = in[R] U gen[R]

Observations:

- For structured programs, we can solve dataflow problems like reaching definitions purely by elimination (without any iteration) (complexity: $O(EV)$).
- For structured programs, we can even solve the dataflow problem directly on the abstract syntax tree (no need to build the control flow graph).
- For less structured programs (like reducible programs), we must build the control flow graph to identify regions like intervals, but there is still no need to iterate.

Exploiting sparsity to speed up dataflow analysis

Example: constant propagation

- CFG algorithm for constant propagation used control flow graph to propagate state vectors.
- Propagating information for all variables in lock-step forces a lot of useless copying information from one vector to another (consider a variable that is defined at top of procedure and used only at bottom).

Solution:

- do constant propagation for each variable separately
- propagate information directly from definitions to uses, skipping over irrelevant portions of control flow graph

Subtle point: in what order should we process variables??