Scalar Optimization

Organization

1. What kind of optimizations are useful?
2. Program analysis for determining opportunities for optimization: dataflow analysis:
   - lattice algebra
   - solving equations on lattices
   - applications to dataflow analysis
3. Speeding up dataflow analysis:
   - exploitation of structure
   - sparse representations: control dependence, SSA form, sparse dataflow evaluator graphs

Optimizations performed by most compilers

- Constant propagation: replace constant-valued variables with constants
- Common sub-expression elimination: avoid re-computing value if value has been computed earlier in program
- Loop invariant removal: move computations into less frequently executed portions of program
- Strength reduction: replace expensive operations (like multiplication) with simpler operations (like addition)
- Dead code removal: eliminate unreachable code and code that is irrelevant to output of program

Optimization example:

```
DO J = 1,100
 DO I = 1,100
   A[I,J] = 1
```

If we assume column-major order of storage, and 4 bytes per array element,

Address of `A[I,J] = BaseAddress(A) + (J-1)*100*4 + (I-1)*4`

= BaseAddress(A) + J*400 + I*4 - 404

- Only the term $I * 4$ depends on I => rest of computation is invariant in the inner loop and can be hoisted out of it.
- Further hoisting of invariant code is possible since only the subterm $J * 400$ depends on J.
- Since I and J are incremented by 1 each time through the loop, expressions like $I * 4$ and $J * 400$ can be strength reduced by replacing them with additions.

Pseudo-code for original loop nest:

```
DO   J = 1,100
  DO I = 1,100
     t = BaseAddress(A) + J*400 + I*4 - 404
     STORE(t,1)
```

Pseudo-code for optimized loop nest:

```
t0 = BaseAddress(A) - 404
DO J = 1,100
  t0 = t0 + 400
  t1 = t0
  DO I = 1,100
    t1 = t1 + 4
    STORE(t,1)
```


Terminology

- A definition of a variable is a statement that may assign to that variable. Definitions of x:

```
 (i) x  = 3
(ii) ... F(x,y).. (call by reference)
```

  In second example, invocation of F may write to x, so to be safe, we declare invocation to be a definition of x.

- A use of a variable is a statement that may read the value of that variable. Uses of x:

```
 (i) y = x + 3
(ii) ...F(x,z)...
```


Aliasing: occurs in a program when two or more names refer to the same storage location.

Examples:

(i)

```
procedure f(vax:x,y)
      ....
...f(z,z) ... f(a,b)...
```

Within f, reference parameters x and y may be aliases!

(ii)

```
..
x := 3;
y := @x;
*y := 5;
 ....x...
```

x and *y are aliases for the same location!

Our position:

We will not perform analysis for variables that may be aliased such (reference parameters, local variables whose addresses have been taken, etc.).

This implies we can determine syntactically where all definitions and uses of a variable are.

More refined approach: perform alias analysis.

For the next few slides, we will focus on constant propagation to illustrate the general approach to dataflow analysis.

Examples:

```
(i)...                              ...
   x := 1;                          x:= 1;
   y := x + 2;                      y := 3;
   if x > z then y:= 5; fi;  =>     if 1 > z then y:= 5; fi;
   ...y...                          ...y...

Constant propagation may simplify control flow as well:
(ii)...                              ...
    x := 1;                          x:= 1;
    y := x + 2;                      y := 3;   <-- dead code
    if y > x then y:= 5; fi;  =>     if true then y := 5; <-- simplify
    ...y...                          ...5...
```


Why do opportunities for constant propagation arise in programs?

- constant declarations for modularity
- macros
- procedure inlining: small methods in OO languages
- machine-specific values

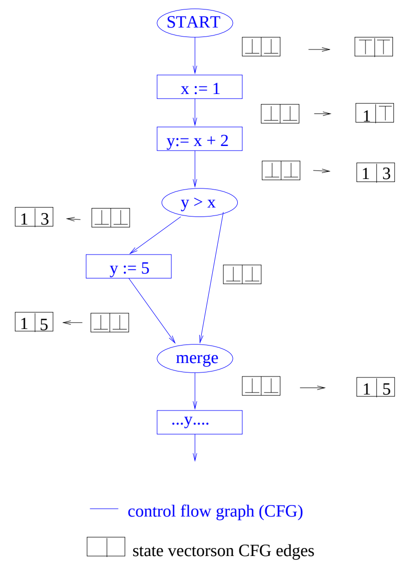
Overview of algorithm:

1. Build the control flow graph of program.
   makes flow of control in program explicit
2. Perform "symbolic evaluation" to determine constants.
3. Replace constant-valued variable uses by their values and simplify expressions and control flow.

Step1: build the control flow graph (CFG).

Example:

```
...
x := 1;
y := x+2;
if (y > x) then y := 5; fi;
...y...
```



Algorithm for building CFG: easy, only complication being `break`'s and `GOTO`'s (need to identify jump targets)

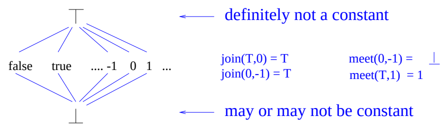
Basic Block: straight-line code without any branches or merging of control flow

Nodes of CFG: statements(or basic blocks)/switches/merges

Edges of CFG: represent possible control flow sequence

Symbolic evaluation of CFG for constant propagation

Propagate values from following lattice:



Two operators:

- join(a,b): lowest value above both a and b (also written as $a \cup b$)
- meet(a,b): highest value below both a and b (also written as $a \cap b$)

Symbolic interpretation of expressions: EVAL(e, Vin): if any argument of e is $\top$ (or $\perp$) in Vin, return $\top$ (or $\perp$ respectively); otherwise, evaluate e normally and return that value

1. Associate one state vector with each edge of the CFG, initializing all entries to $\perp$. Initialize work-list to empty.
2. Set each entry of state vector on edge out of `START to ⊤, and place this edge on the worklist.`

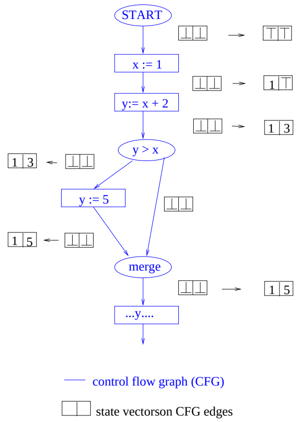
```
3. while worklist is not empty do
        Get edge from worklist;
        Let state vector on edge be Vin;

      //Symbolically evaluate target node of the edge,
      //using the state vectors on its inputs,
      //and propagate result state vector to output edge of node;
        if (target node is assignment statement x:= e)
           Propagate Vin[EVAL(e,Vin)/x] to output edge;
        else if (target node is switch(p))
           {if EVAL(p,Vin) is T, Propagate Vin to all outputs of switch;
            else if EVAL(p,Vin) is true, Propagate Vin to true side of switch;
            else Propagate Vin to false side of switch;
```

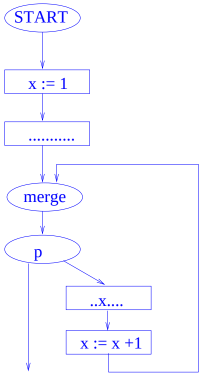
```
      }
    else //target node is merge
      Propagate join of state vectors on all inputs to output;

   If this changes the output state vector, enqueue output edge
        on worklist;
od;
```


Running example:



Algorithm can quite subtle:



First time through loop, use of x in loop is determined to be constant 1. Next time through loop, it reaches final value $\top$.

Complexity of algorithm:

Height of lattice = 2 => each state vector can change value $2 * V$ times.

So *while loop* in algorithm is executed at most $2 * E * V$ times.

Cost of each iteration: $O(V)$.

So overall algorithm takes $O(EV^2)$ time.

Questions:

- Can we use same work-list based algorithm with different lattices to solve other analysis problems?
- Can we improve the efficiency the algorithm?

Need to separate what is being computed from how it is being computed => use algebras once again!!

Lattice algebraic approach to dataflow analysis

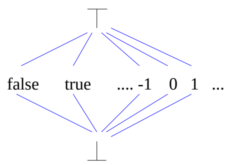
Abstractly, our work-list algorithm can be viewed as one solution procedure for solving a set of lattice algebraic equations.

Dataflow lattices:

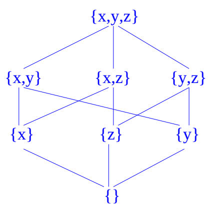
- partially order set of finite height
- meet and join operations with appropriate algebraic properties are defined for all pairs of values from po-set.

These properties imply that the lattice has a least and a greatest element.

Examples:



lattice for constant propagation



power set of variables {x,y,z}

Monotonic function: If D is a partially ordered set and $f : D \rightarrow D$, f is said to be monotonic if $x \subseteq y => f(x) \subseteq f(y)$.

Intuitively, if the input of a monotonic function is increased, the output either stays the same or increases as well.

Examples of monotonic functions on CP lattice:

- identity: f(x) = x
- bottom function: f(x) = ⊥
- constant function: f(x) = 2

Examples of non-monotonic functions on CP lattice:
f(x) = if (x == 2) then 1 else ⊥

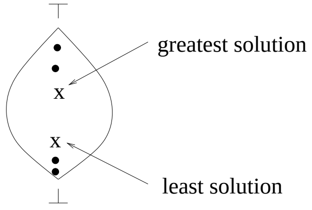
Theorem: Let D be a lattice of finite height and $f : D \to D$ be monotonic. Then, the equations $x = f(x)$ has a least and a greatest solution given by the limits of the chains $\bot, f(\bot), f^2(\bot), ...$ and $\top, f(\top), f^2(\top), ...$

Proof:

$$\bot \subseteq f(\bot)(definitionof\ \bot)$$
$$f(\bot) \subseteq f^2(\bot)(monotonicityof f)$$
$$....$$
$$=> \bot \subseteq f(\bot) \subseteq (f^2(\bot))...$$

Since the lattice has finite height, this chain has some largest element l, and $f(l) = l$. So l solves the equation. It is also easy to show that l is the least solution to the equations.

A similar argument shows that a greatest solution exists.



Examples:

- f(x) $= \bot$
  $limit(\bot, f(\bot) = \bot, ...) = \bot$
  $limit(\top, f(\top) = \bot, f(\bot) = \bot, ...) = \bot$
- f(x) = x
  $limit(\bot, f(\bot) = \bot, ...) = \bot$
  $limit(\top, f(\top) = \top, ...) = \top$

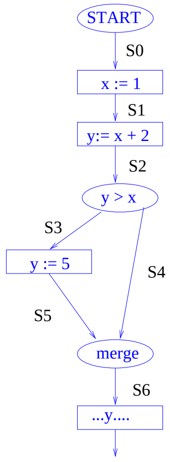
Corollary:

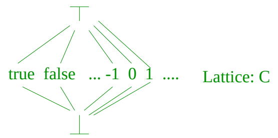
If f, g, h etc are monotonic functions, the system of equations

$$x = f(x, y, z...)$$
$$y = g(x, y, z, ..)$$
$$z = h(x, y, z, ..)....$$

has least and greatest solutions given by the limits of the obvious chains (eg, least solution is obtained by starting with $\bot$ for all variables, substituting into right hand sides to get new values for all variables, and iterating till convergence occurs).

Connection between constant propagation and lattice equations:
iterative procedure is just a method to solve lattice equations!



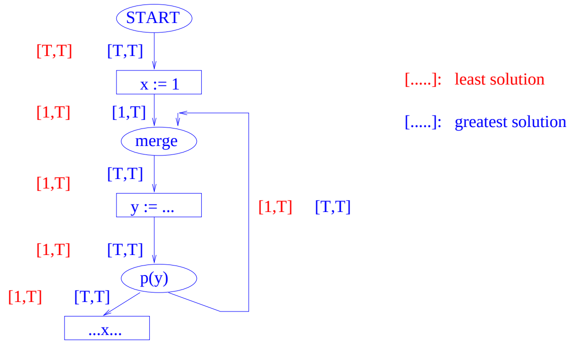


S0, S1, S2,..... : CxC (2 variables)

S0 = [T,T]
S1 = S0{1 / x}
S2 = S1{2 / y }
S3 = if (S2[y] < S2[x]) or (S2[y]=T) or (S2[x] = T)
then S2
else S3
.....
S6 = S4 U S5
......

Question: since equations have many solutions in general, which one should we compute?

For CP, least solution gives more accurate information than other solutions.



In general, if confluence operator is join, compute least solution; otherwise compute greatest solution.

General specification of dataflow problem:

- Lattice: finite height
- Rules for writing down equations from CFG
- Confluence operator

No special arguments about termination or complexity are needed.

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.

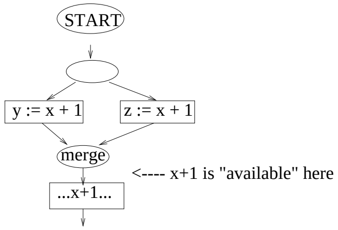
General classification of dataflow problems:

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

Available expressions:FORWARD FLOW, ALL PATHS

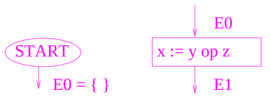
Definition: An expression 'x op y' is available at a point p if every path from START to p contains an evaluation of p after which there are no assignments to x or y.

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)

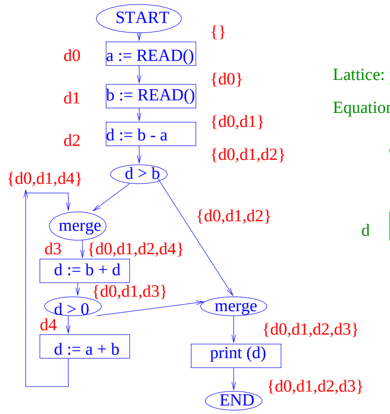
(where Ex is all expressions involving x)

confluence operator: meet (intersection)

compute greatest solution

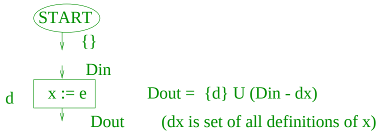
Reaching definitions:FORWARD FLOW, ANY PATH

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:



Confluence operator: join (union)

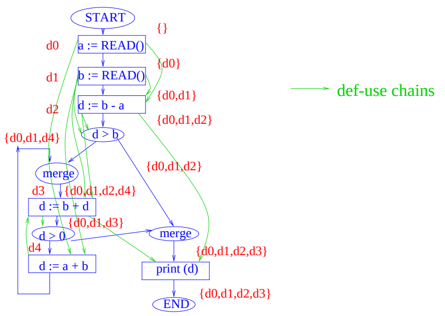
Compute least solution

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

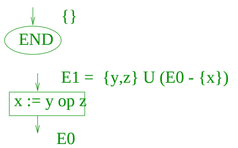


Live variable analysis:BACKWARD FLOW, ANY PATH

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)

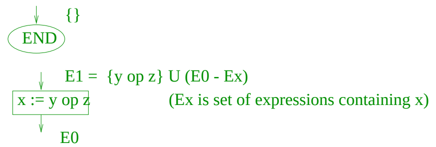
Compute least solution

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:



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 Analysis

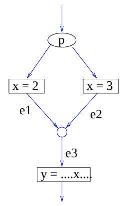
Constant propagation on CFG: $O(EV^2)$
Reaching definitions on CFG: $O(EN^2)$
Available expressions on CFG: $O(EA^2)$

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?

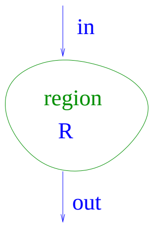
- basic-blocks
- basic-blocks, if-then-else, loops
- intervals
- ....

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.

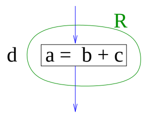
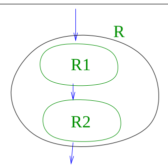
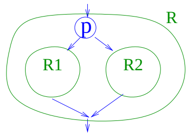
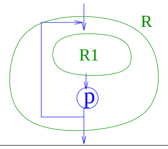
Dataflow equation for region can be written using gen and kill:



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:

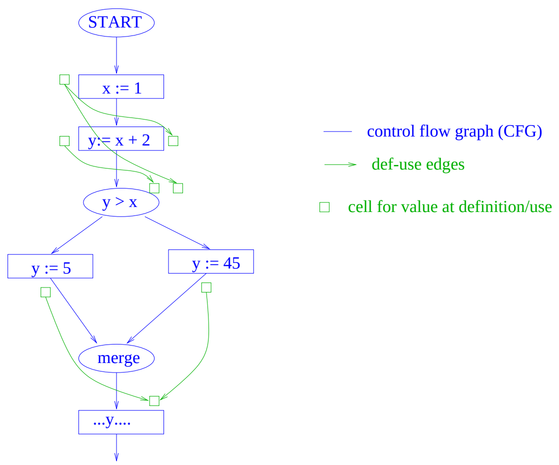
- do constant propagation for each variable separately
- propagate information directly from definitions to uses, skipping over irrelevant portions of control flow graph

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

Constant propagation using def-use chains

- Associate cell with each lhs and rhs occurence of all variables, initialize to $\perp$.
- Propagate $\top$ along each def-use edge out of START, and enqueue target statements of def-use edges onto worklist.
- Enqueue all definitions with constant RHS onto worklist.
-
```
while (worklist is not empty) do
  dequeue definition d from work-list;
  evaluate RHS of d using cell values for RHS variables
    and update LHS cell;
  if this changes LHS cell value,
    propagate new value along def-use chains to each use
      (take join of cell value at use and LHS cell value);
    if cell value at use changes and target statement is a definition,
       enqueue target statement onto worklist;
od;
```


Example:



Complexity: $O(sizeofdef - usechains)$
This can be as large $O(N^2V)$ where N is size of set of CFG nodes.
However, with SSA form, can be reduced to $O(EV)$.

Problem with algorithm: loss of accuracy.

Propagation along def-use chains cannot determine directly that *y:= 45* is dead code, so last use of y is not marked constant.

One possibility: repeated cycles of reaching definition computation, constant propagation and dead code elimination.

Is there a better way?

Key idea:

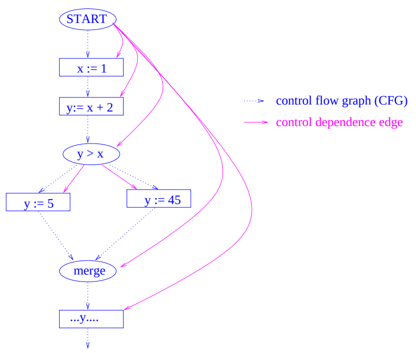
- find unreachable statements during constant propagation
- do not propagate values out of unreachable definitions

One approach: use control dependence and def-use chains

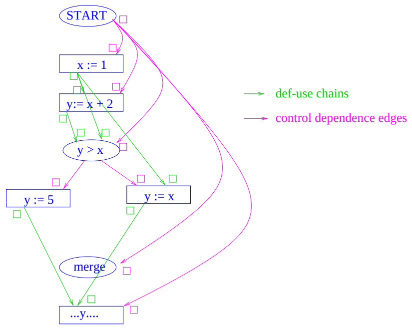
Intuitive idea of control dependence: Node n is control dependent on predicate p if p determines whether n is executed.

Convention: assume START is a predicate so unconditionally executed statements are control dependent on START.

CDG: Control dependence graph



Algorithm: Propagate "liveness" along control dependence edges while propagating constants along def-use chains.



Constant propagation

- Associate cell with each lhs and rhs occurence of all variables, and with each statement, initialized to $\perp$
- Propagate $\top$ along each def-use edge and control dependence edge out of START. If value in any target cell changes, enqueue target statement onto worklist.
-

```
while (worklist is not empty) do
  dequeue statement d from work-list;
  if control dependence cell of statement is $\top$
   switch (type of d) :
    case (definition):
      {Evaluate RHS of d using cell values for RHS variables
           and update LHS cell;
       If this changes LHS cell value,
           propagate new value along def-use chains to each use
             (take join of cell value at use and LHS cell value);
```



```
         If cell value at use changes, enqueue target statement onto worklist;
        }
      case (switch) :
        {Evaluate predicate and propagate along appropriate control dependence
          edges out of predicate;
         If cell value at target changes,
          enqueue target statement onto worklist;
        }
    fi; od;
```

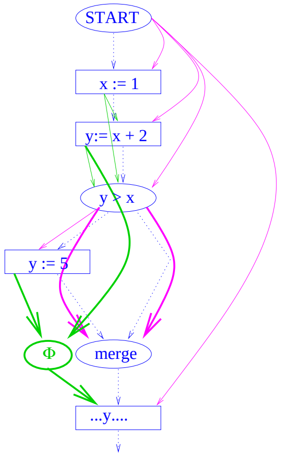

Observations:

- We do not propagate information out of dead (unreachable) statements.
- However, precision of information is still not as good as CFG algorithm: we still propagate information out of statements that are executed but are irrelevant to output (other sort of dead statements, as in Slide 18).
- Need an algorithm to compute control dependence in general graphs.
- Size of CDG: $O(EN)$ (can be reduced)

Solutions:

- Require that a variable assigned on one side of a conditional be assigned on both sides of conditional (by inserting dummy assignments of form $x := x$). Programmers don't want to do this.
- Make compiler insert dummy assignments. Hard to figure out in presence of unstructured control flow.
- Use SSA form: ensure that every use is reached by exactly one definition by inserting ϕ-functions at merges to combine multiple reaching definitions.

SSA algorithm for Constant Propagation



- phi-function combines different reaching definitions at a merge into a single one at output of merge
- phi-function is like pseudo-assignment
- control dependence at merge: compute for each side of the merge separately

Constant propagation:
similar to previous algorithm, but at merge, propagate join of inputs only from live sides of merge

Where should phi-functions be placed?

One possibility: one phi-function for every variable at every merge in CFG.

Minimal SSA form: permit def-use chains to bypass a merge if same definition reaches all sides of merge

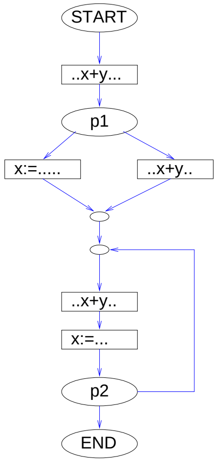
(eg. variable x in example)

Computing minimal SSA form: O(|E|) per variable (Pingali and Bilardi PLDI 96)

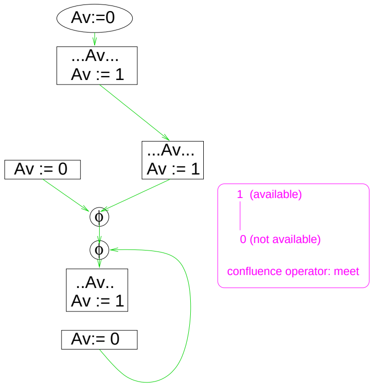
Same idea can be applied to other dataflow analysis problems

- perform dataflow analysis for each sub-problem separately (eg. for each expression separately in available expressions problem)
- build a sparse graph in which only statements that modify or use dataflow information for sub-problem are present, and solve in that

Sparse dataflow evaluator graph can be built in $O(|E|)$ time per problem (Pingali and Bilardi PLDI'96)



Control Flow Graph



Sparse Dataflow Evaluator Graph for availability of x+y

Advantage: sparse graph is usually small and acyclic

Disadvantage: need to solve each sub-problem separately

Many optimizing compilers now use sparse dataflow evaluation graphs (eg: Intel's compilers for Merced).