Automatic Program Optimization

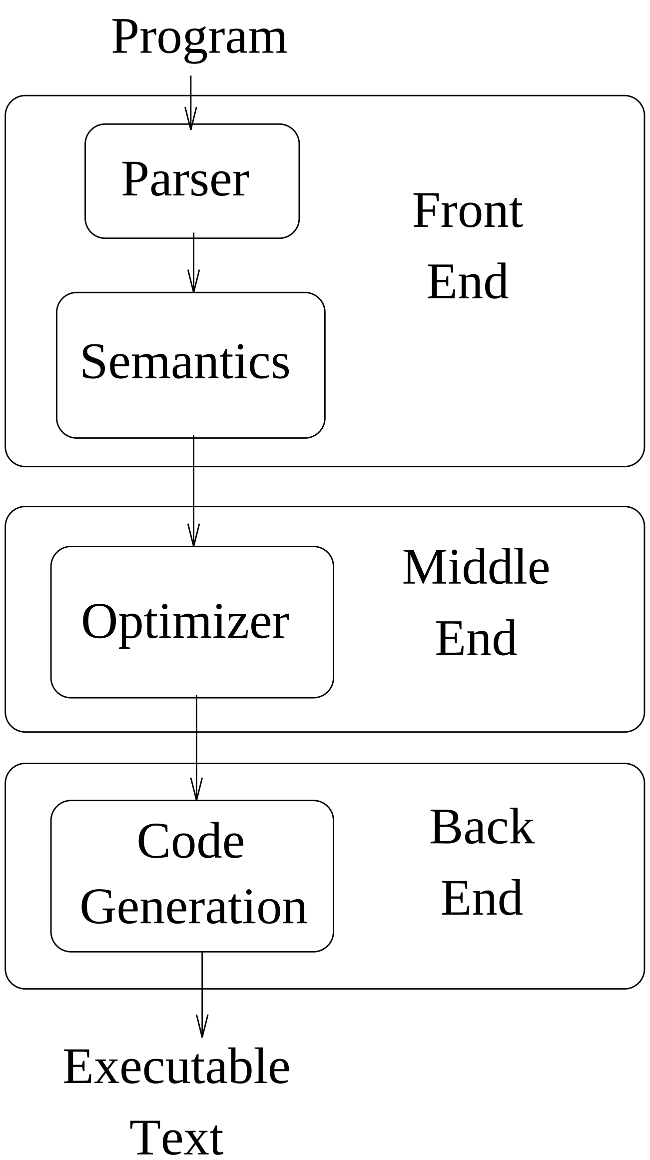
SIGPLAN '93
PLDI Tutorial Notes

Ron Cytron

Washington University
St. Louis, Missouri

IBM T.J. Watson Research Labs
Yorktown Heights, New York

Compiler organizations



Front end: Operator and storage abstractions, alias mechanisms.

Middle end:

- Dead code elimination
- Code motion
- Reduction in strength
- Constant propagation
- Common subexpression elimination
- Fission
- Fusion
- Strip mining
- Jamming
- Splitting
- Collapsing

Back end: Finite resource issues and code generation.

Some thoughts

Misconceptions

Optimization optimizes your program. There's probably a better algorithm or sequence of program transformations. While optimization hopefully improves your program, the result is usually not optimal.

Optimization requires (much) more compilation time. For example, dead code elimination can reduce the size of program text such that overall compile time is also reduced.

A clever programmer is a good substitute for an optimizing compiler. While efficient coding of an algorithm is essential, programs should not be obfuscated by "tricks" that are architecture- (and sometimes compiler-) specific.

All too often. . .

Optimization is disabled by default. Debugging optimized code can be treacherous (71, 42). Optimization is often the primary suspect of program misbehavior—sometimes deservedly so. "No, not the *third* switch!"

Optimization is slow. Transformations are often applied to too much of a program. Optimizations are often textbook recipes, applied without proper thought.

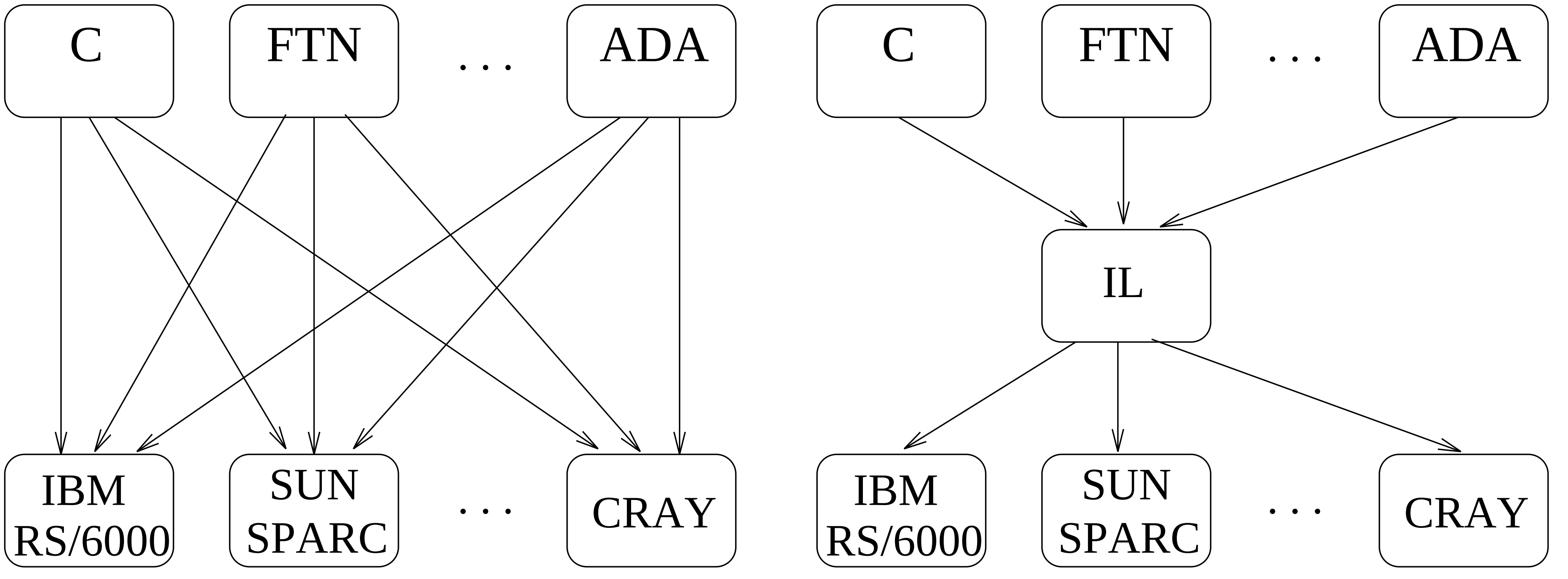
Optimization produces incorrect code. Although recent work is encouraging (67), optimizations are usually developed *ad hoc.*

Programmers are trained by their compilers. A style is inevitably developed that is conducive to optimization.

Optimization is like *sex:*

- Everybody claims to get good results using *exotic* techniques;
- Nobody is willing to provide the details.

Multilingual systems



Architecting an *intermediate language* reduces the incremental cost of accommodating new source languages or target architectures (7). Moreover, many optimizations can be performed directly on the intermediate language text, so that source- and machine-independent optimizations can be performed by a common middle-end.

Intermediate languages

It's very easy to devote much time and effort toward choosing the "right" IL. Below are some guidelines for choosing or developing a useful intermediate language:

- **The IL should be a** *bona fide* **language, and not just an aggregation of data structures.**
- **The semantics of the IL should be cleanly defined and readily apparent.**
- **The IL's representation should not be overly verbose:**
  - **Although some expansion is inevitable, the IL-to-source token ratio should be as low as possible.**
  - **It's desirable for the IL to have a verbose, human-readable form.**
- **The IL should be easily and cleanly extensible.**
- **The IL should be sufficiently general to represent the important aspects of multiple front-end languages.**
- **The IL should be sufficiently general to support efficient code generation for multiple back-end targets.**

A sampling of difficult issues:

- **How should a string operation be represented (intact or as a "loop")?**
- **How much detail of a procedure's behavior is relevant?**

Ideally, an IL has *fractal* **characteristics: optimization can proceed at a given level; the IL can be "lowered"; optimization is then applied to the freshly exposed description.**

Intermediate languages (cont'd)

Components of an IL:

SymbolTable: declares the name space of a procedure.

AliasRelations: declares sets of aliased names.

Semantics: gives a formal specification of the procedure's behavior.

Example program:

Procedure $foo(x, y)$

declare

x, y **integer**

a, b **integer**

$\star p$ **integer**

$p \leftarrow rand()$? &a : &b

$\star p \leftarrow x + y$

end

The procedure randomly assigns the address of a or b to p, and then stores $(x + y)$ into the location dereferenced by p.

This example is intended to illustrate *may-alias* **behavior: $\star p$ may be an alias for a or b, and static analysis certainly can't decide which alias will hold at run-time.**

We'll now look at a possible intermediate representation of this procedure, using a Lisp-like notation that is easy to parse and to extend. The verbose keywords are easily coded and stored efficiently as terminals of the language's grammar.

Symbol table

```
(SymbolTable
   (NumSymbols 5)
   (Symbol
      (SymbolName x)
      (SymbolID   1)
   )
   (Symbol
      (SymbolName y)
      (SymbolID   2)
   )
   (Symbol
      (SymbolName p)
      (SymbolID   3)
   )
   (Symbol
      (SymbolName a)
      (SymbolID   4)
   )
   (Symbol
      (SymbolName b)
      (SymbolID   5)
   )
)
```

Symbol attributes (`SymbolType`, `SymbolSize`, `SymbolVolatile`, etc.) can be added as needed.

Alias relations

In programs with pointers or reference parameters, use of a lexical name could refer to one of multiple, distinct storage names. We therefore specify sets of names that might be treated in this manner. Subsequently, a given reference to a lexical name can be associated with an alias relation entry, to describe the potential effects of that reference.

In this example, we have an alias relation with may-aliases for a and b and no must-aliases. This relation might be appropriate for the assignment through $\star p$.

Also shown are two relations in which x and y are alternately must- and may-aliased. If parameters are passed "by-address", then a use of x may be a use of y if each parameter is supplied the same address.

```
(AliasRelations
   (NumAliasRelations 2)
   (AliasRelation
      (AliasID 1)
      (MayAliases 2 a b)
   )
   (AliasRelation
      (AliasID 2)
      (MustAliases 1 y)
      (MayAliases  1 x)
   )
   (AliasRelation
      (AliasID 3)
      (MustAliases 1 x)
      (MayAliases  1 y)
   )
)
```

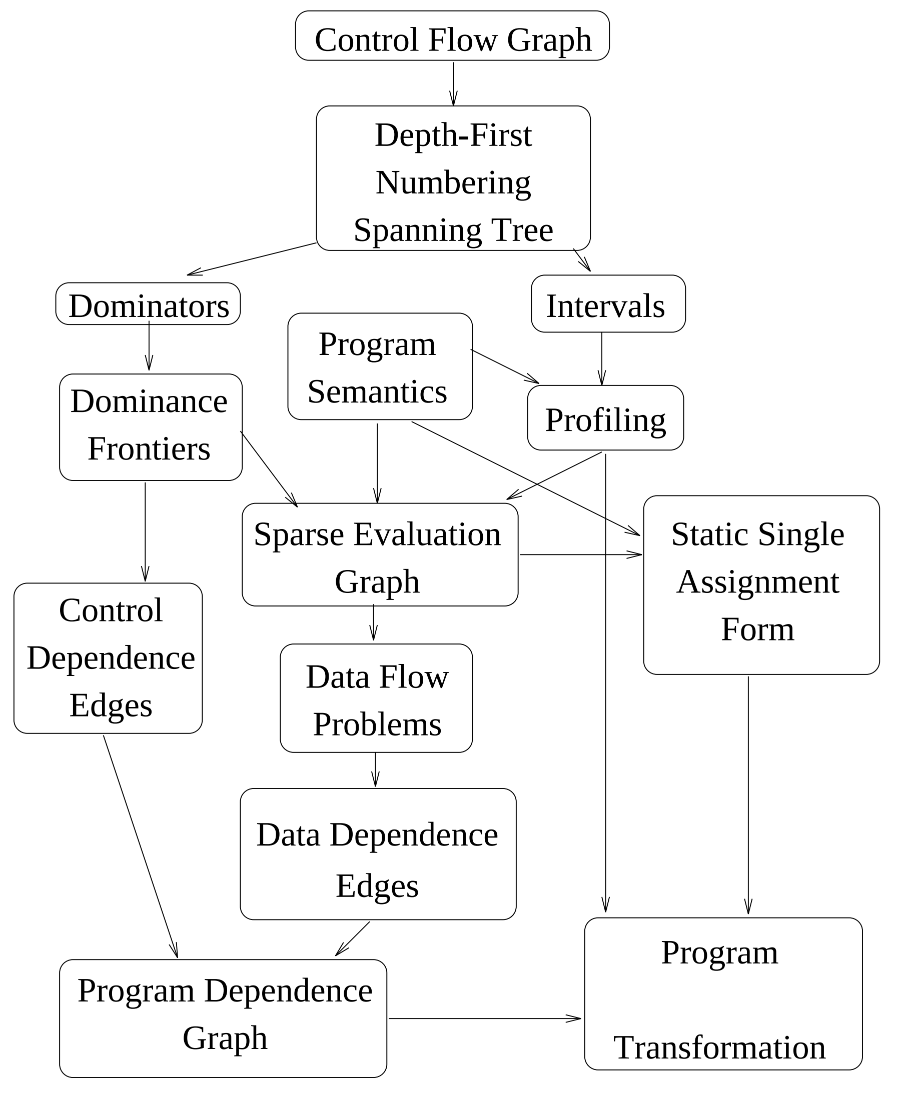
Procedure semantics

```
(ProcSemantics
  (NumNodes 5)

  (NodeSemantics
    (NodeID 1)
    (Def
      (DefID 1)
      (SymbID p)
      (DefValue
        (Choose
          ((= rand() 0)
            (Addr a)
            (Addr b)
          )
        )
      )
    )
    (Jump 2)
  )

  (NodeSemantics
    (NodeID 2)
    (Def
      (DefID 2)
      (SymbID ?)
      (AliasWith 1)
      (DefValue
        (+
          (Use
            (UseID 1)
            (SymbID x)
          )
          (Use
            (UseID 2)
            (SymbID y)
          )
        )
      )
    )
  )

)
```

What happens in the middle end?

Essentially, the program is transformed into an observably equivalent while less resource-consumptive program. Such transformation is often based on:

- **Assertions provided by the program author or benefactor.**
- **The program dependence graph (50, 35, 10).**
- **Static single assignment (SSA) form (26, 5, 69, 28).**
- **Static information gathered by solving data flow problems (44, 51, 52, 53, 41, 54, 55, 48).**
- **Run-time information collected by** *profiling* **(61).**



Let's take a look at an example that benefits greatly from optimization. . .

Unoptimized matrix multiply

for $i = 1$ **to** N **do**
  for $j = 1$ **to** N **do**
    $A[i,j] \leftarrow 0$
    for $k = 1$ **to** N **do**
      $A[i,j] \leftarrow A[i,j] + B[i,k] \times C[k,j]$
    od
  od
od

Note that $A[i,j]$ is really

$$Addr(A) + ((i-1) \times \; K_1 + (j-1)) \times K_2$$

which takes 6 integer operations.

The innermost loop of this “textbook” program takes

24	integer ops
3	loads
1	floating add
1	floating mpy
1	store
30	instructions

Optimizing matrix multiply

for $i = 1$ **to** N **do**
    for $j = 1$ **to** N **do**
        $a \leftarrow \&(A[i,j])$
        for $k = 1$ **to** N **do**
            $\star a \leftarrow \star a + B[i,k] \times C[k,j]$
        od
    od
od

The expression $A[i,j]$ is *loop-invariant* with respect to the k loop. Thus, *code motion* can move the address arithmetic for $A[i,j]$ out of the innermost loop.

The resulting innermost loop contains only 12 integer operations.

for $i = 1$ **to** N **do**
    $b \leftarrow \&(B[i,1])$
    for $j = 1$ **to** N **do**
        $a \leftarrow \&(A[i,j])$
        for $k = 1$ **to** N **do**
            $\star a \leftarrow \star a + \star b \times C[k,j]$
            $b \leftarrow b + K_B$
        od
    od
od

As loop k iterates, addressing arithmetic for B changes from $B[i,k]$ to $B[i,k+1]$. *Induction variable analysis* detects the constant difference between these expressions.

The resulting innermost loop contains only 7 integer operations.

Similar analysis for C yields only 2 integer operations in the innermost loop, for a speedup of nearly 5. We can do better, especially for large arrays.

If optimization is...

so great becase:

A good compiler can sell (even a slow) machine. Optimizing compilers easily provide a factor of two in performance. Moreover, the analysis performed during program optimization can be incorporated into the "programming environment" (50, 25, 68).

New languages and architectures motivate new program optimizations. Although some optimizations are almost universally beneficial, the advent of functional and parallel programming languages has increased the intensity of research into program analysis and transformation.

Programs can be written with attention to clarity, rather than performance. There is no substitute for a good algorithm. However, the expression of an algorithm should be as independent as possible of any specific architecture.

then:

Why does it take so long? Compilation time is usually 2–5 times slower, and programs with large procedures often take longer. Often this is the result of poor engineering: better data structures or algorithms can help in the optimizer.

Why does the resulting program sometimes exhibit unexpected behavior? Sometimes the source program is at fault, and a bug is uncovered when the optimized code is executed; sometimes the optimizing compiler is itself to blame.

Why is "no-opt" the default? Most compilations occur during the software development cycle. Unfortunately, most debuggers cannot provide useful information when the program has been optimized (71, 42). Even more unfortunately, optimizing compilers sometimes produce incorrect code. Often, insufficient time is spent testing the optimizer, and with no-opt the default, bugs in the optimizer may remain hidden.

Outline

1. Examples of data flow problems.
2. Flow graphs and their abstractions.
3. Data flow frameworks.
   (a) Specification.
   (b) Evaluation.
   (c) Properties.
4. Sparse Evaluation Graphs.
5. Static Single Assignment (SSA) form.
6. SSA-based algorithms.

Ingredients in a data flow framework

Data flow graph

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

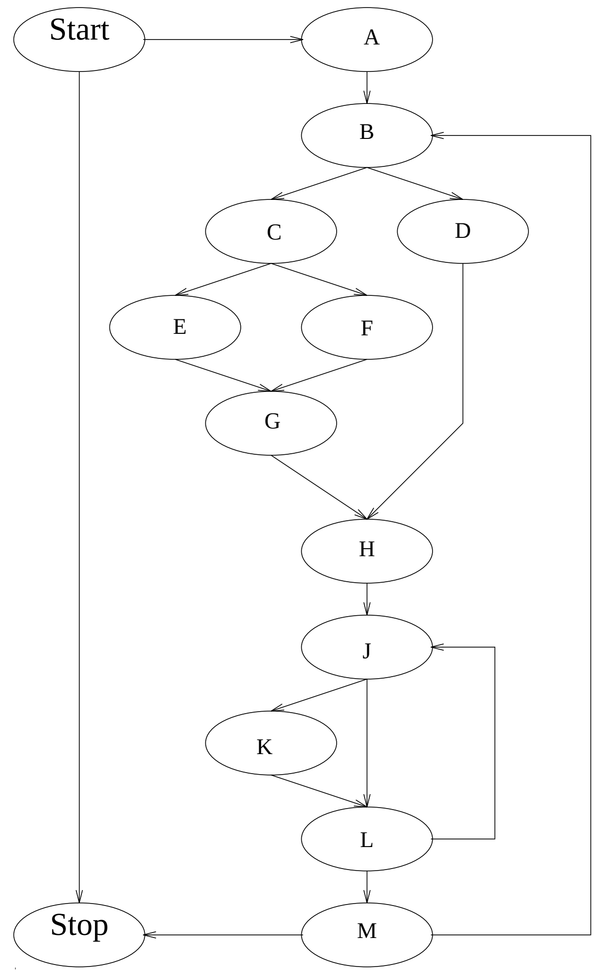
which is based on a directed *flow graph* $\mathcal{G}_f = (\mathcal{N}_f, \mathcal{E}_f)$**, typically the** *control flow graph* **of a procedure.**

A data flow problem is

forward if the solution at a node may depend only on the program's past behavior;

backward if the solution at a node may depend only on a program's future behavior;

bidirectional if both past and future behavior is relevant (31, 32, 33).



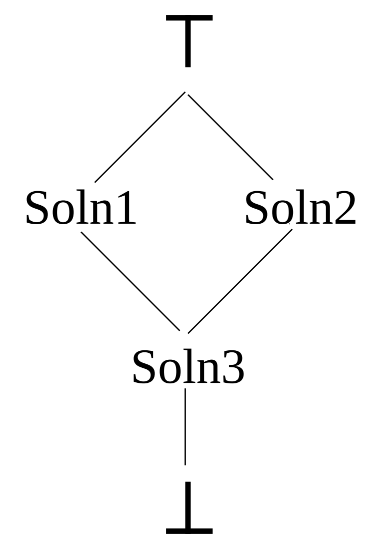
- We'll assume the data flow graph is augmented with a $Start$ and $Stop$ node, and an edge from $Start$ to $Stop$.
- We'll limit our discussion to non-bidirectional problems, and assume that edges in the data flow graph are oriented in the direction of the data flow problem.

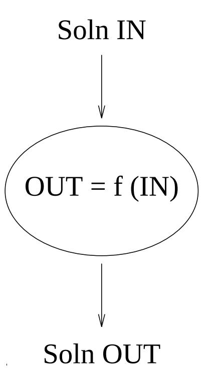
Ingredients in a data flow framework (cont'd)

Meet lattice which determines the outcome when disparate solutions combine. The lattice is specified with distinguished elements

- $\top$ which represents the best possible solution, and
- $\perp$ which represents the worst possible solution.

Transfer Functions which transform one solution into another.





We'll use the meet lattice to summarize the effects of convergent paths in the data flow graph, and transfer functions to model the effects of a data flow graph path on the data flow solution.

We'll begin with the four simple and related *bit-vectoring* data flow problems, classically solved as operations on bit-vectors. For ease of exposition, we'll associate data flow solutions with the edges, rather than the nodes, of the data flow graph.

Available expressions

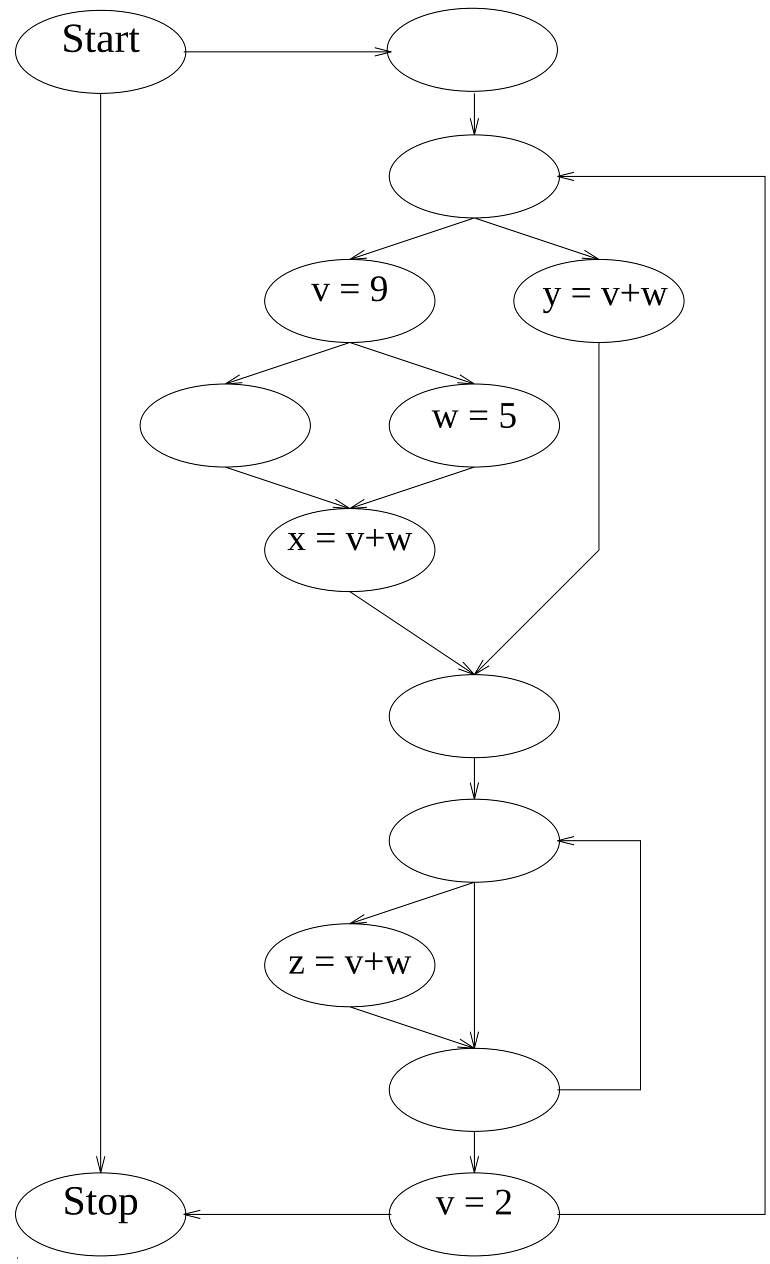
An expression $expr$ is *available* ($Avail$) at flow graph edge e if any past behavior of the program includes a computation of the value of $expr$ at e.

Consider the expression $(v + w)$ in the flow graph shown to the right. If the expression is available at the assignment to z, then it need not be recomputed.

- This is a forward problem, so the data flow graph will have the same edges and $Start$ and $Stop$ nodes as the flow graph.
- The solution for any given $expr$ is either $Avail$ or $\overline{Avail}$.
- The "best" solution for an expression is $Avail$. We thus obtain the two-level lattice:

  $\top$ is $Avail$.

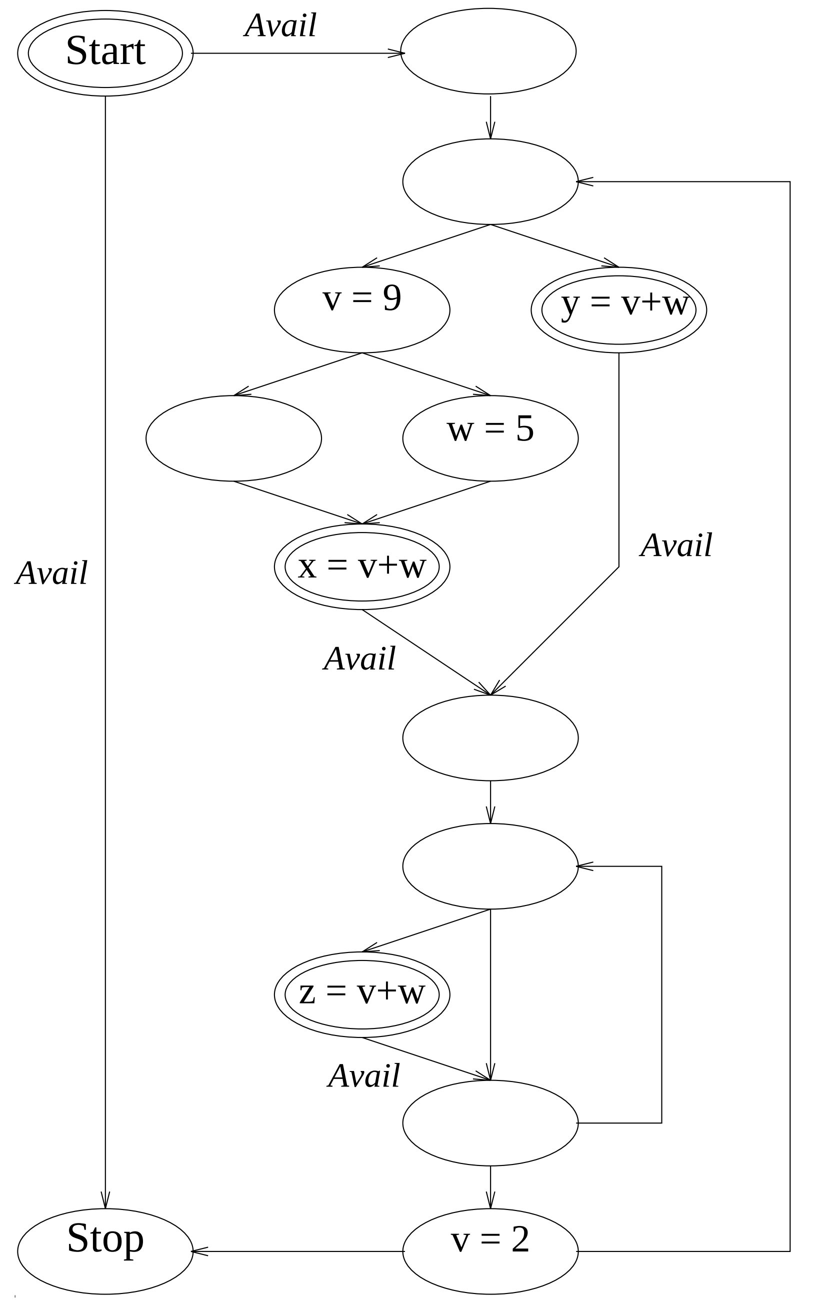
  $\bot$ is $\overline{Avail}$.



Available expressions(cont'd)

Nodes that compute an expression make that expression available. We also assume that every expression is available from $Start$**.**

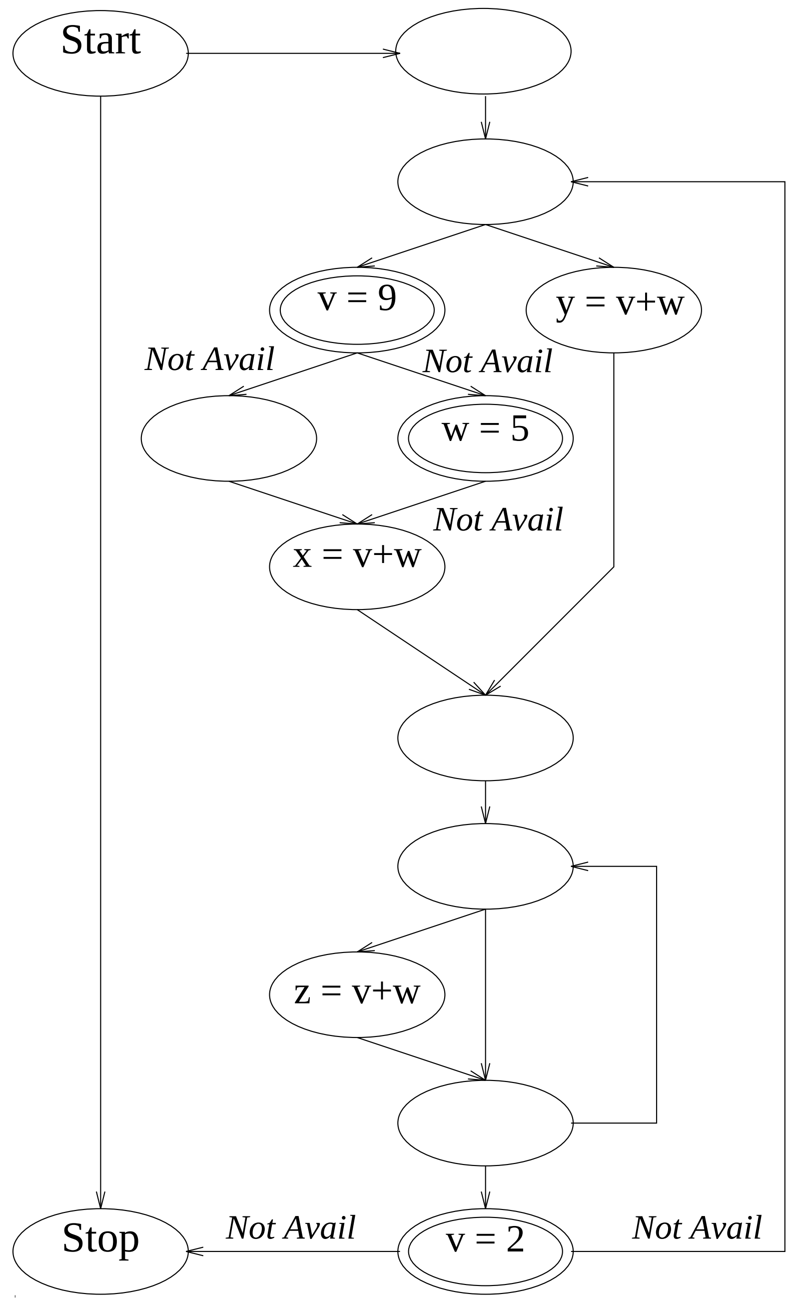
The transfer function for each highlighted node makes the expression $(v + w)$ $Avail$**, regardless of the solution present at the node's input.**



Available expressions(cont'd)

Nodes that assign to any variable in an expression make that expression not available, even if the variable's value is unchanged.

The transfer function for each highlighted node makes the expression $(v + w)$ $\overline{Avail}$, regardless of the solution present at the node's input.



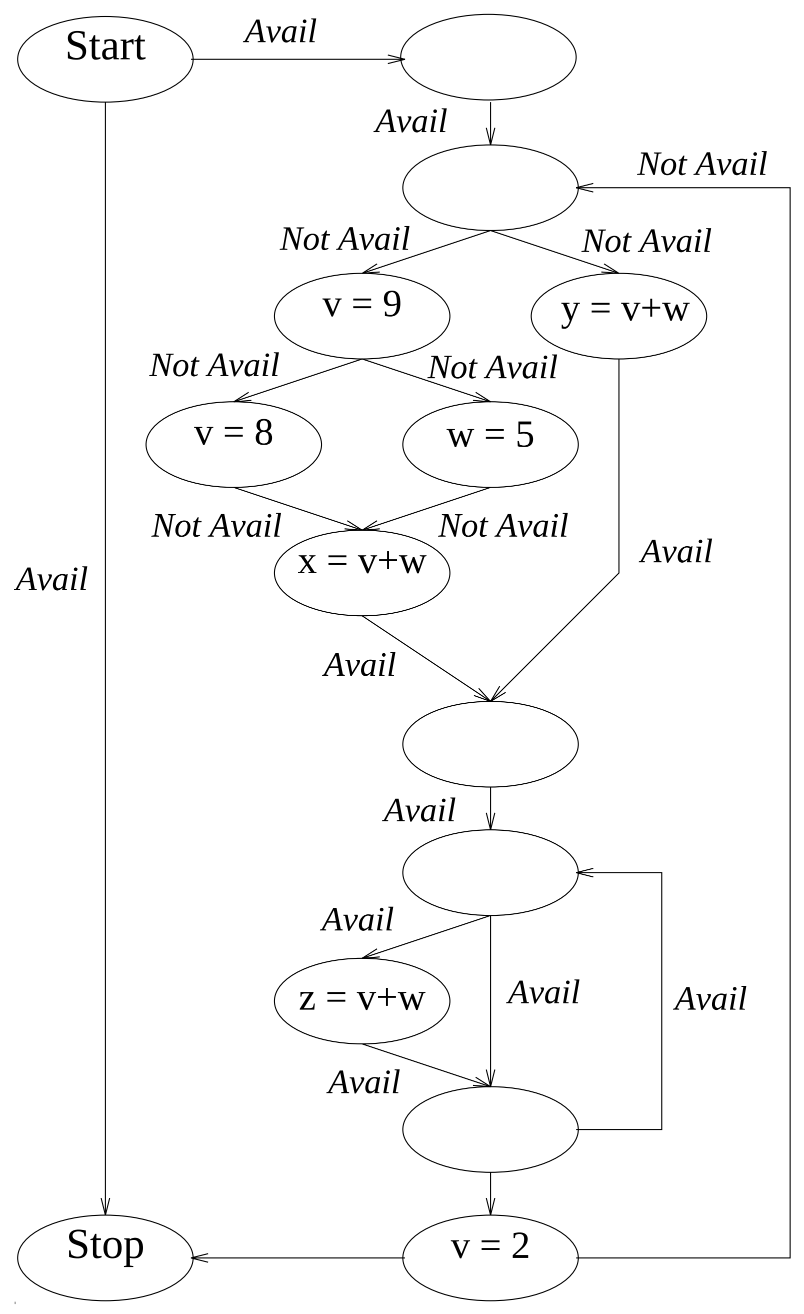
Available expressions (cont'd)

Here we see the global solution for availability of the expression $(v + w)$.

Each of the highlighted nodes shown previously asserts a solution on its output edge(s). It's the job of global data flow analysis to assign the best possible solution to every edge in the data flow graph, consistent with the asserted solutions.

The expression $(v + w)$ need not be computed in the assignment to z. The relevant value is held either in x, or y, depending on program flow.

To solve this problem using bit-vectors, assign each expression a position in the bit-vector. When an expression is available, its associated bit is 1.



Very busy expressions

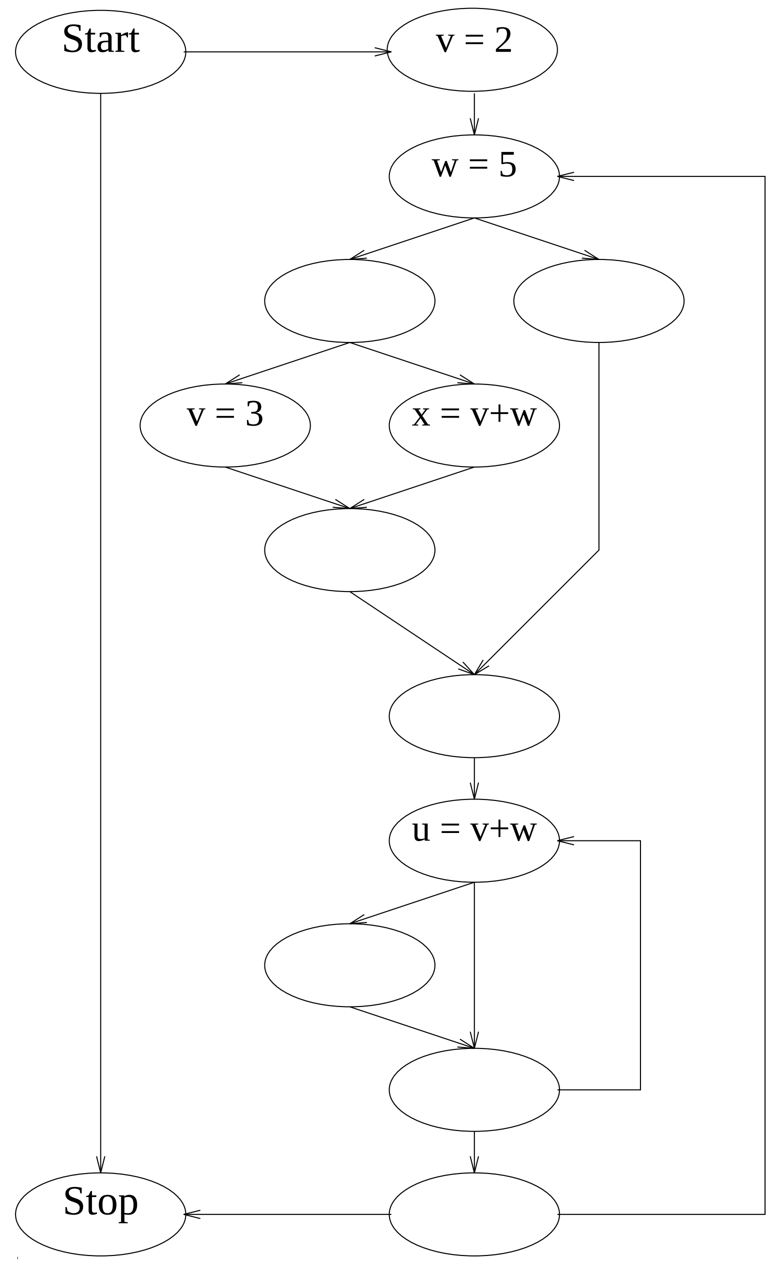
An expression $expr$ is *very busy* (VB) at flow graph edge e if any future behavior of the program references the value of $expr$ at e.

Where an expression is very busy, resources allocated to compute or hold that expression won't be wasted.

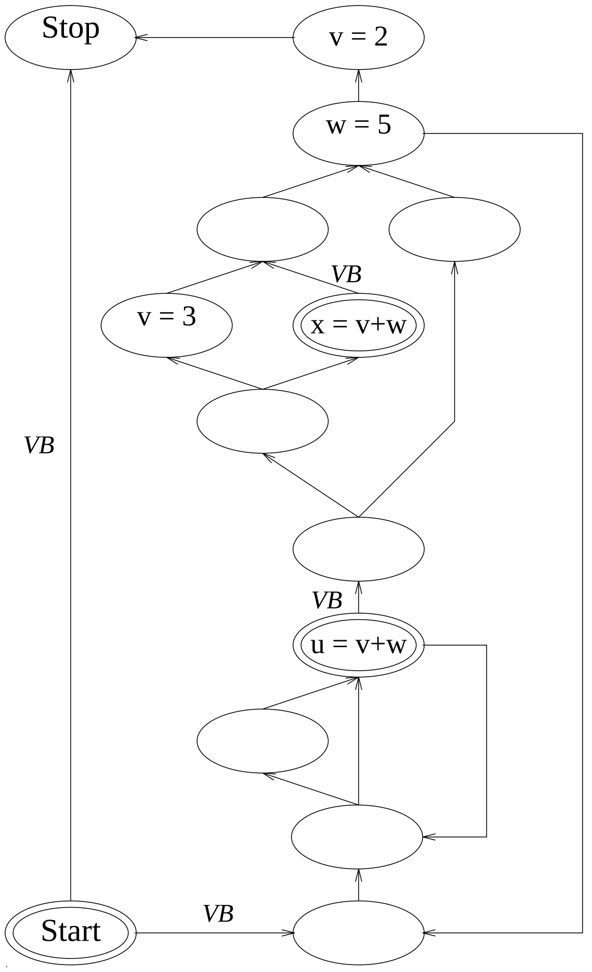
- **This is a backward problem, so the data flow graph will have every edge reversed, and nodes $Start$ and $Stop$ will be interchanged.**
- **The solution for any given expression is either VB or $\overline{VB}$.**
- **The "best" solution for an expression is VB. We thus obtain the two-level lattice:**

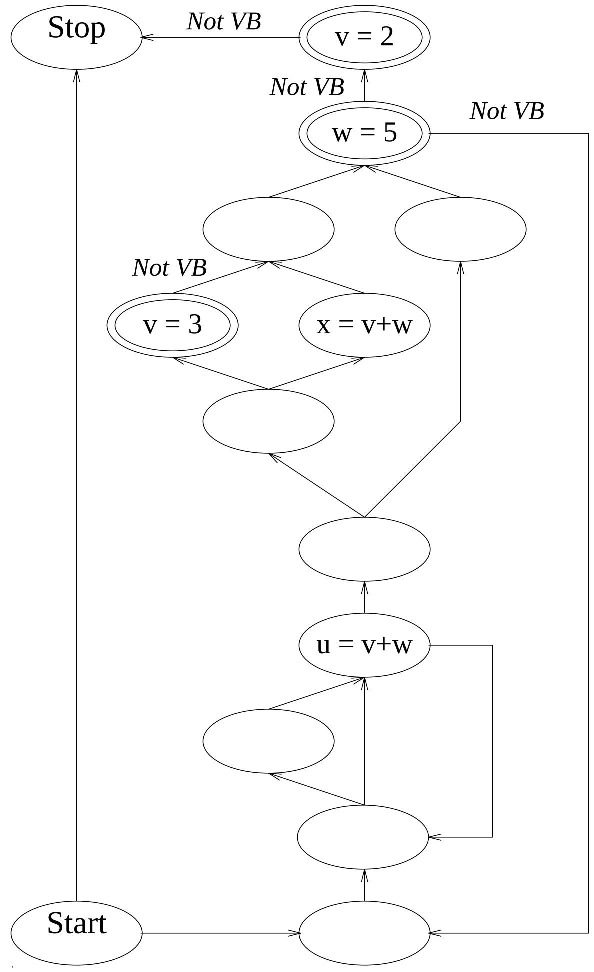
  $\top$ **is** VB.

  $\perp$ **is** $\overline{VB}$.



Very busy expressions (cont'd)





Nodes that compute an expression make that expression very busy. We also assume that every expression is very busy from $Start$.

Nodes that assign to any variable in an expression make that expression not very busy.

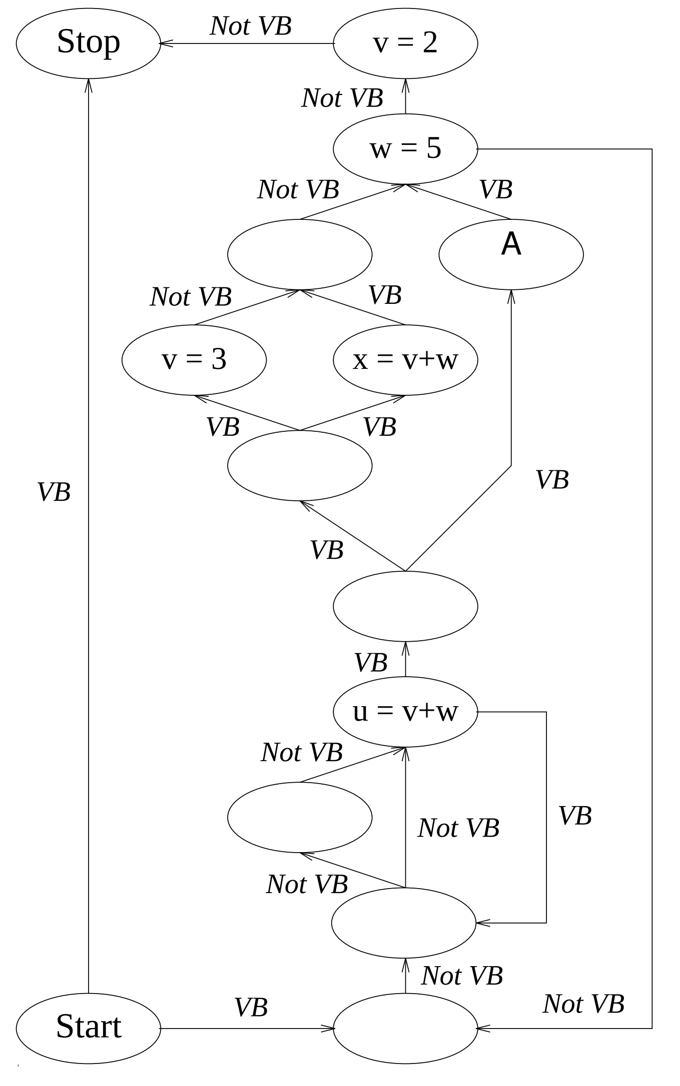
The above graphs are data flow graphs, with their edges oriented in the direction of this backward data flow problem.

Very busy expressions (cont'd)

Here we see the global solution for the expression $(v + w)$.

The original program may call for two evaluations of $(v+w)$: one at the assignment of x and one at the assignment of u. By moving the computation of $(v + w)$ from the assignment of u to node A, we can avoid two computations of the expression in an iteration of the outer loop, even though the expression $(v+w)$ isn't available at the assignment to u.

As with available expressions, each expression is assigned a position in the bit-vector.



A variant of this data flow problem is useful for early transmission of data or messages in a distributed or hierarchical storage system (i.e., message passing architectures or programmable caches) (30).

Reaching defs

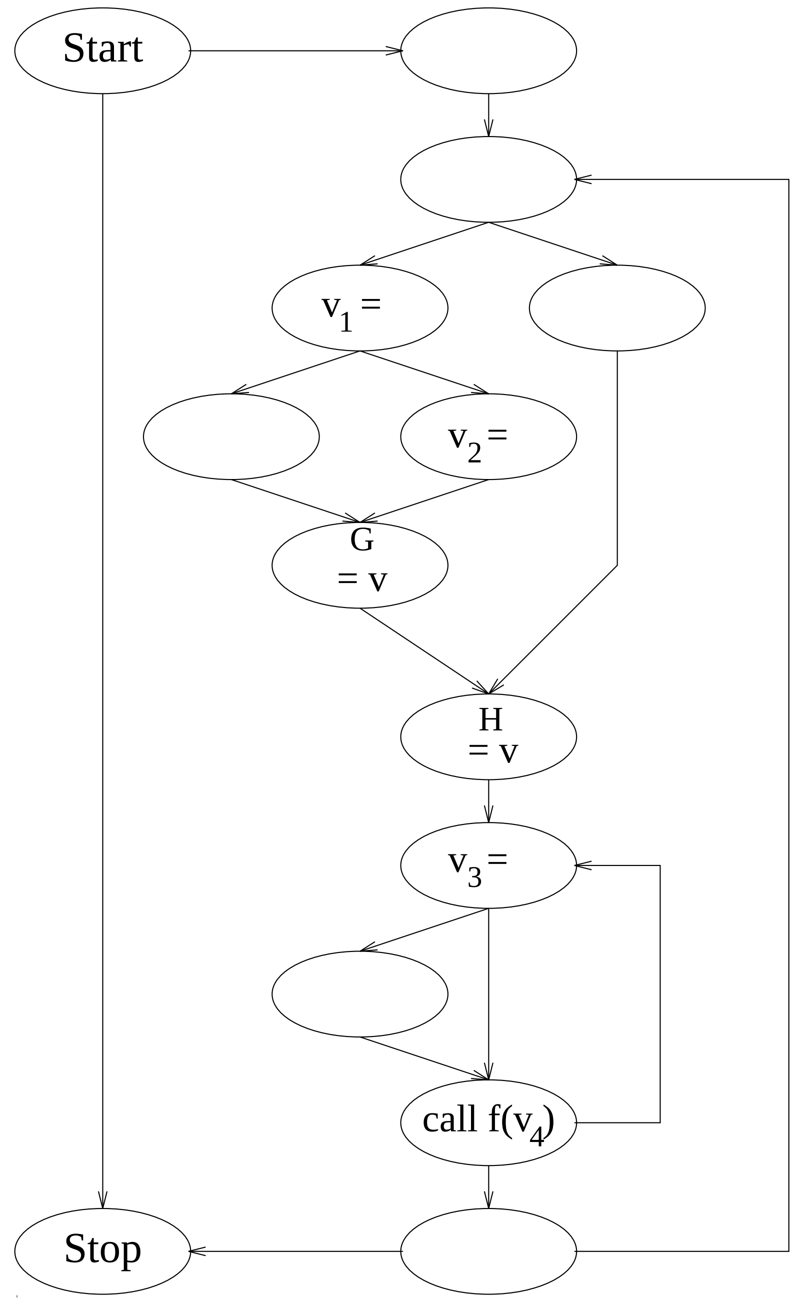
A definition d of variable v *reaches* an edge e if the past behavior of the program may cause d to provide the value for v at e.

Each definition shown in the flow graph is to the variable v; the definitions are subscripted so we may distinguish among the definitions of v.

- This is a forward data flow problem.
- If a definition d does not reach e, then d is irrelevant to program optimization at e. Thus, we obtain a lattice where

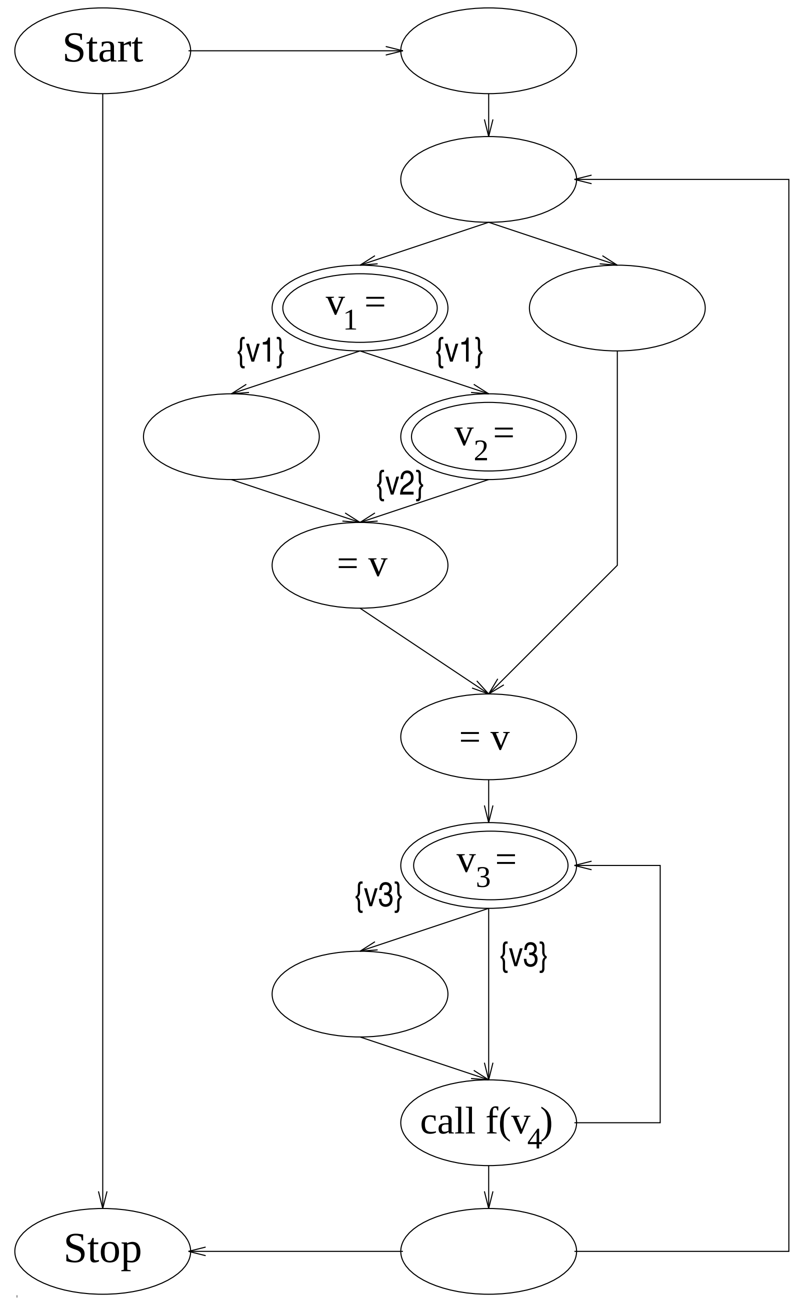
  $\top$ is the empty set.

  $\perp$ is the set of all definitions.
- The meet of two solutions a and b is the union of the definition sites in a and b.



Reaching defs (cont'd)

If a node absolutely changes the value of a variable, then the definition *kills* all other values for that variable. The transfer function for each highlighted node produces only the node's definition of v in the output.

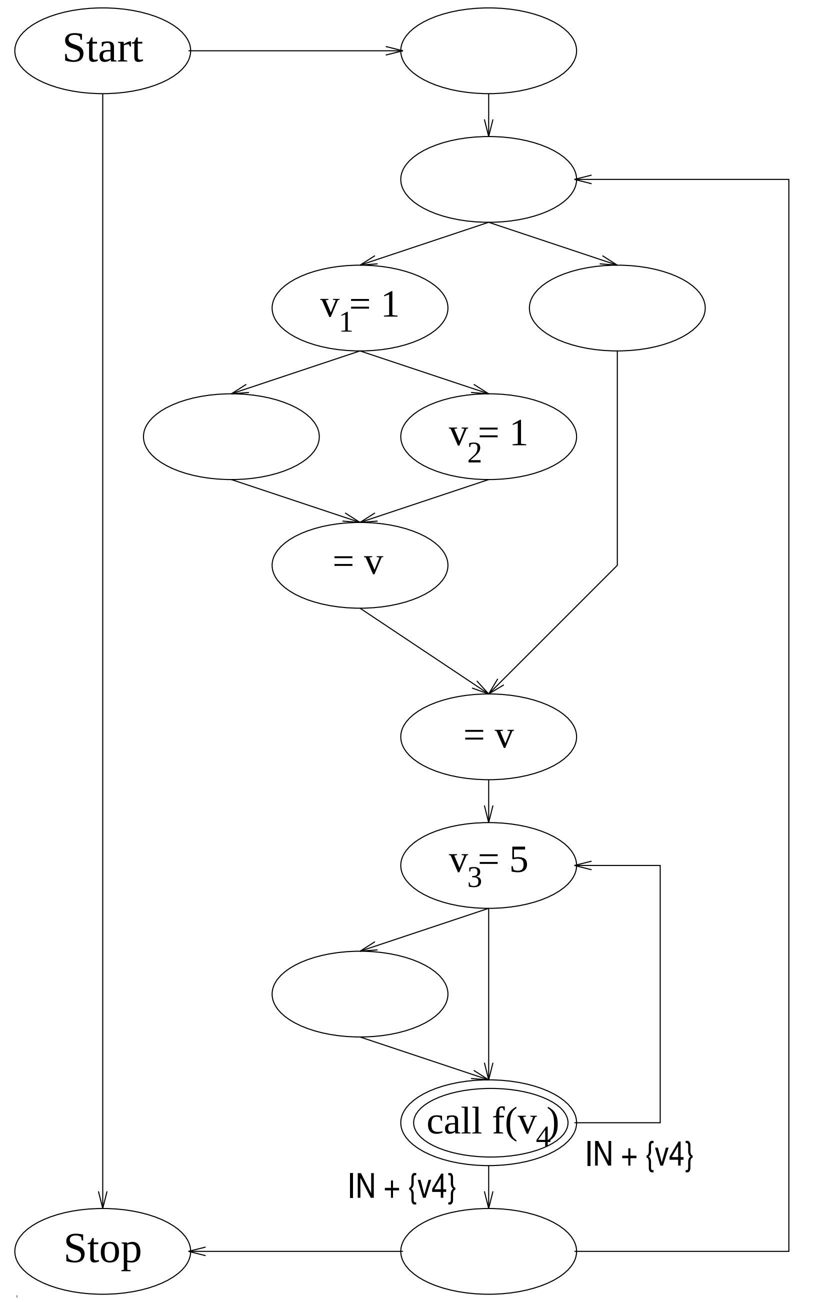


Reaching defs (cont'd)

Otherwise, the definition *preserves* all other reaching defs of the variable, as might be the case with the procedure call highlighted above. The node's transfer function adds the node's definition of v to any that already reach the node.

Examples of preserving defs include

- **potentially modified variables at a call site;**
- **assignments that modify some, but not all, elements of an array;**
- **assignments to may-aliased variables.**



Reaching defs (cont'd)

Here we see the global solution for reaching defs of v.

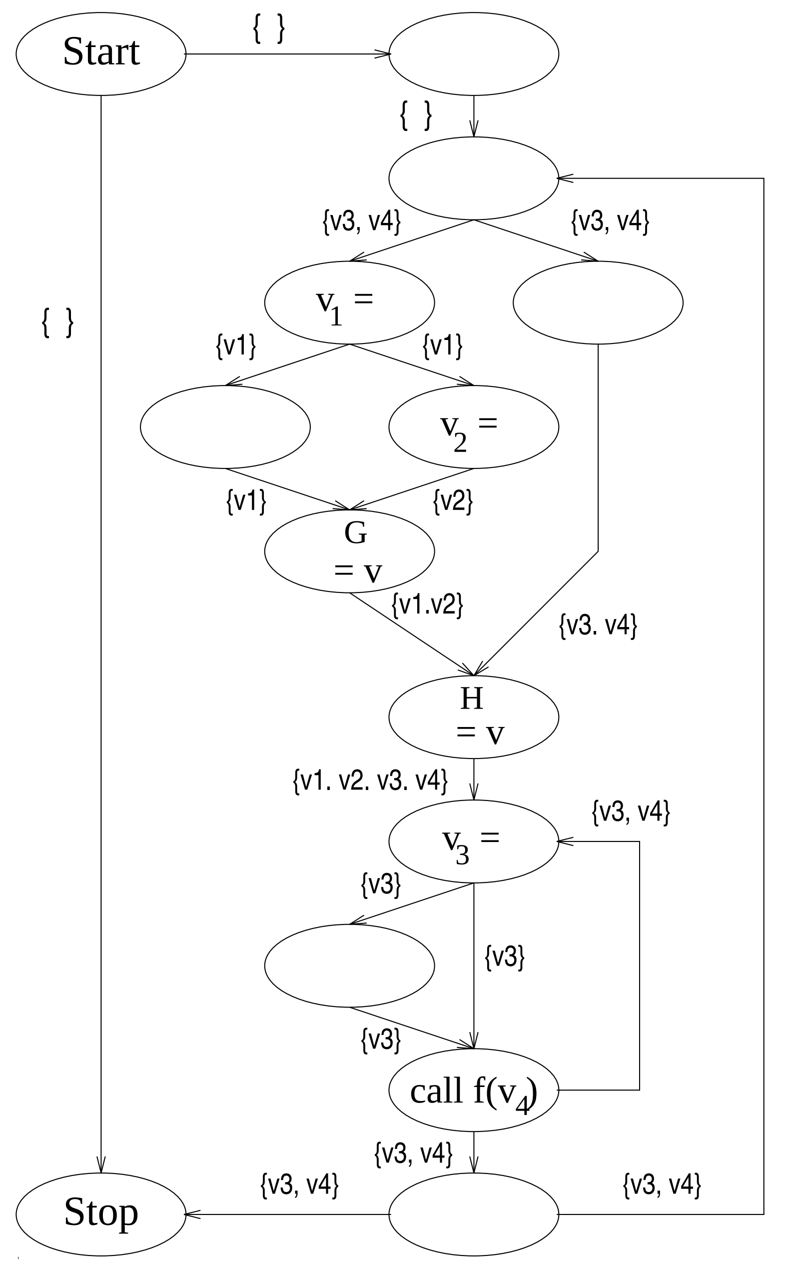
Reaching defs is often solved in support of other optimization problems, such as constant propagation and register allocation.

In its bit-vector setting, each definition is assigned a position in the bit-vector.

- The transfer function at node Y has the form

  $$f_Y(IN) = (IN - KILL_Y) \cup GEN_Y$$

  where $KILL_Y$ and GEN_Y are node-specific bit-vector constants.
- The meet lattice can be formed by set-union, which is easily realized for the bit-vectors.



Live variables

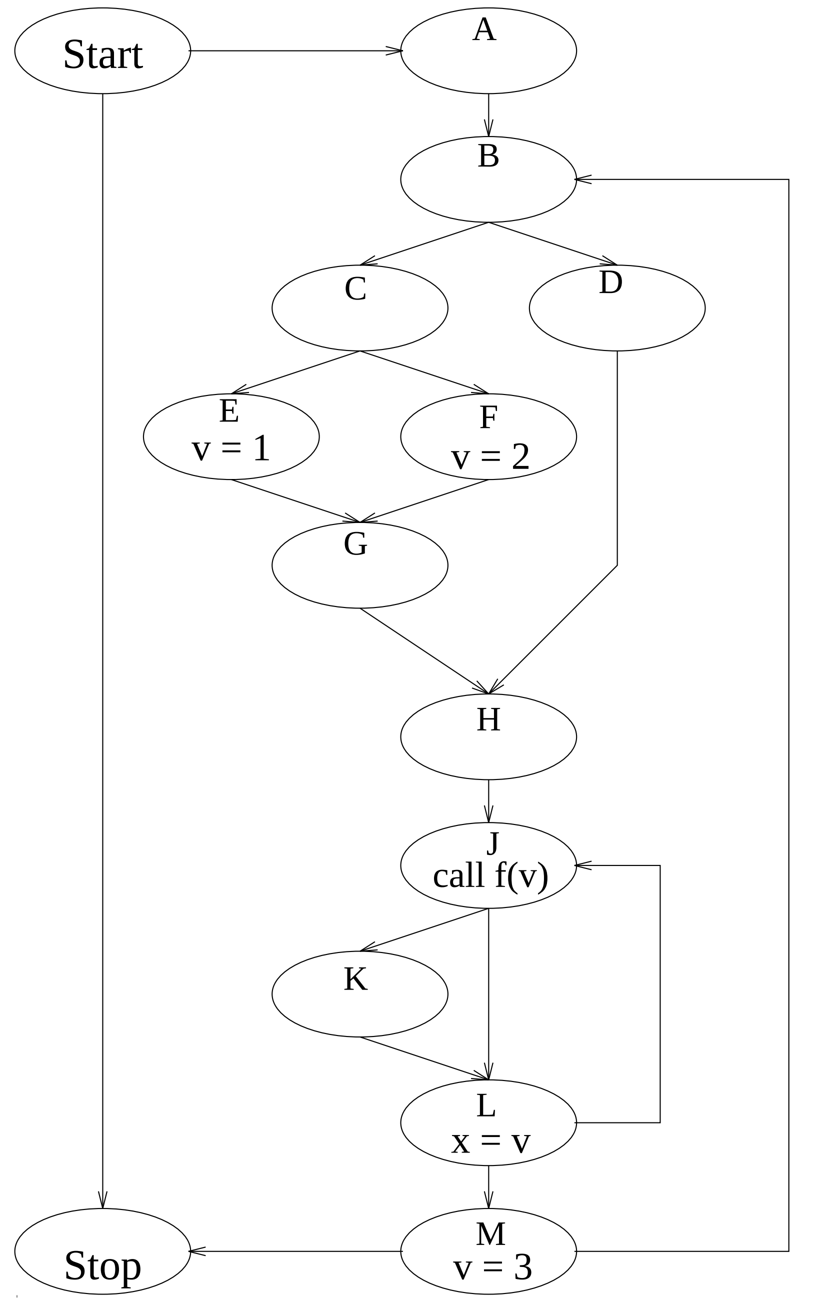
A variable v is *live* at edge e if the future behavior of the program may reference the value of v at e.

If a variable v is not live, then any resources associated with v (registers, storage, etc.) may be reclaimed.

- **This is a backward problem.**
- **In the bit-vector representation, each variable is associated with a bit.**
- **The "best" solution is $\overline{Live}$, so we obtain the two-level lattice:**

  $\top$ **is** $\overline{Live}$.

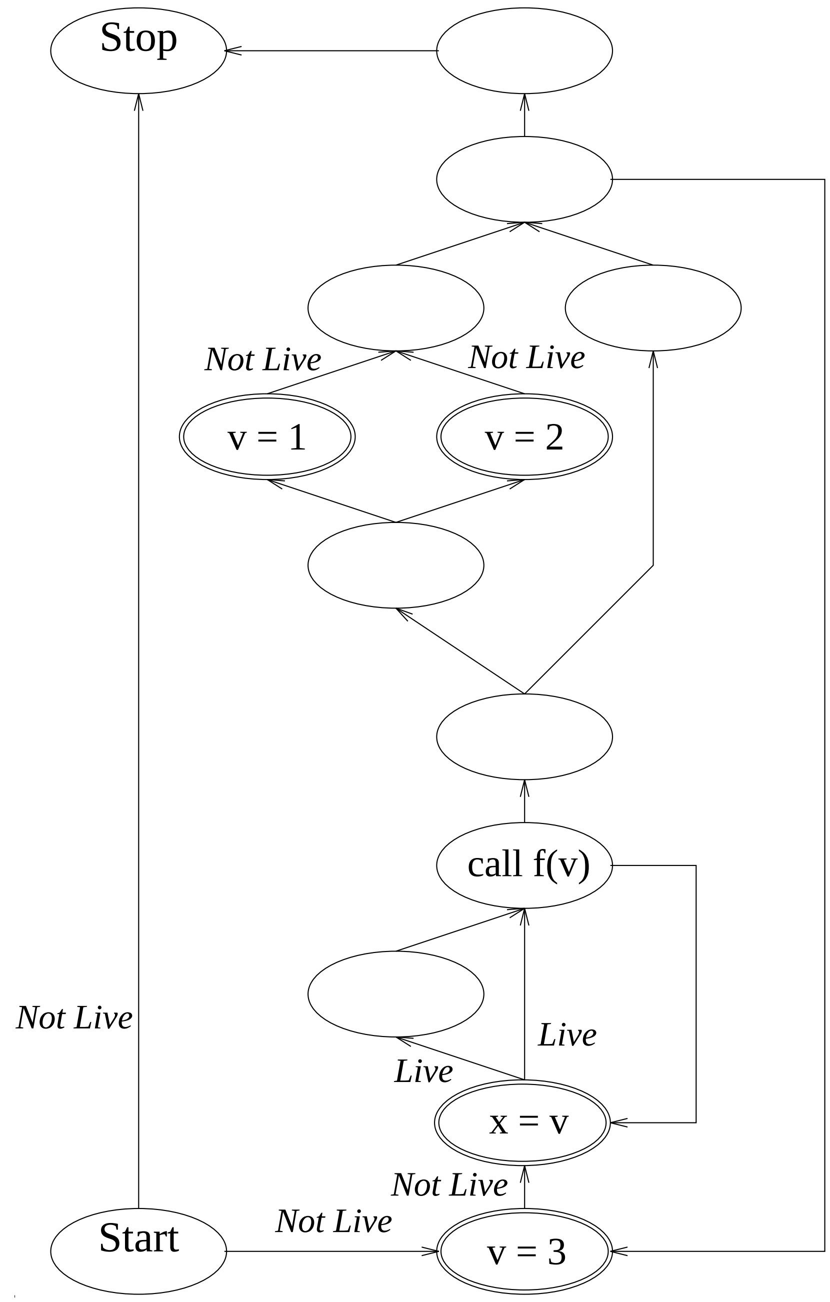
  $\perp$ **is** $Live$.



Live variables (cont'd)

Each of the highlighted nodes affects the data flow solution:

- **If a node uses v, then the node's asserts that v is $Live$.**
- **If a node kills v, then the node's output asserts that v is $\overline{Live}$.**



Live variables (cont'd)

If a node Y preserves v (as might a procedure call), then the node does not affect the solution.

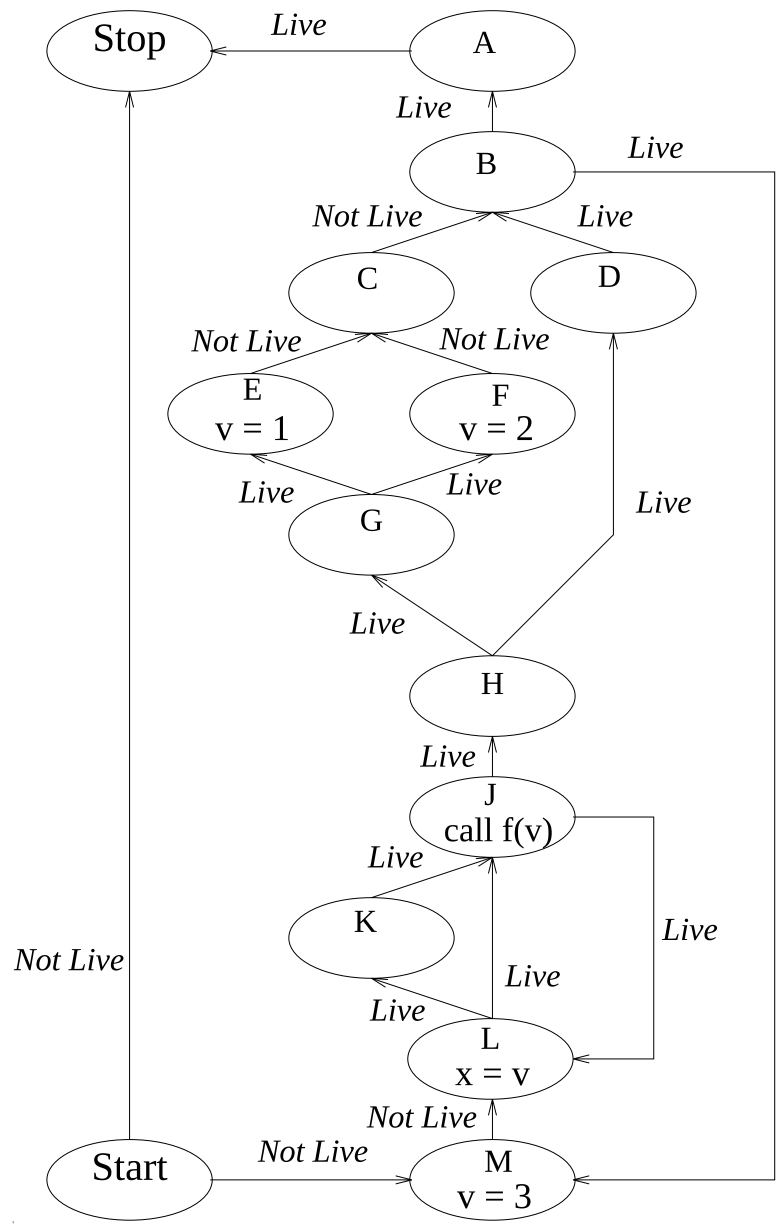
- **If v is $Live$ on "input" to Y, then Y cannot make v $\overline{Live}$.**
- **If v is $\overline{Live}$ on "input" to Y, then Y does not make v $Live$.**

Node Y's transfer function is therefore the *identity* function:

$$f_Y(IN) = IN$$

assuming node Y does not use v.

Global solution: Live variables



The bit-vectoring data flow problems

The four problems we've just covered share the following properties:

- A solution can be represented as a bit-vector.
- The meet operation can be realized as the Boolean operation
  or for reaching defs and live variables;
  and for available expressions and very busy expressions.
- At each node Y we have a transfer function of the form

$$f_Y = (IN - KILL_Y) \cup GEN_Y$$

  where $KILL_Y$ and GEN_Y are node-specific bit-vector constants.

Flow graphs

Intraprocedural: nodes of the *control flow graph* should represent all potentially executable code, and any program path followed during execution should exist as a path in the control flow graph. For languages whose control flow behavior is syntactically apparent (using source text), the associated control flow graph can be constructed easily and precisely (3). Higher-order languages such as Scheme complicate the construction (65), as do languages such as APL, where any statement could be the target of a branch (20).

Interprocedural: nodes of the *procedure call graph* should represent all potentially executable procedures, and any possible sequence of procedure calls during program execution should exist as a path in the call graph. Where procedures cannot be passed as parameters, construction of the call graph is straightforward; otherwise, some preliminary flow analysis is required to construct a precise graph (57, 15).

The *correctness* of static analysis depends on the availability of a suitably approximate flow graph; the *success* of static analysis depends on the degree of approximation.

Flow graphs (cont'd)

Procedure $main$

  if $(cond1)$ **then**

    call foo

  fi

  while $(cond2)$ **do**

    $w \leftarrow 5$

  od

end

Procedure foo

  if $(cond3)$ **then**

    call A

    call B

  else

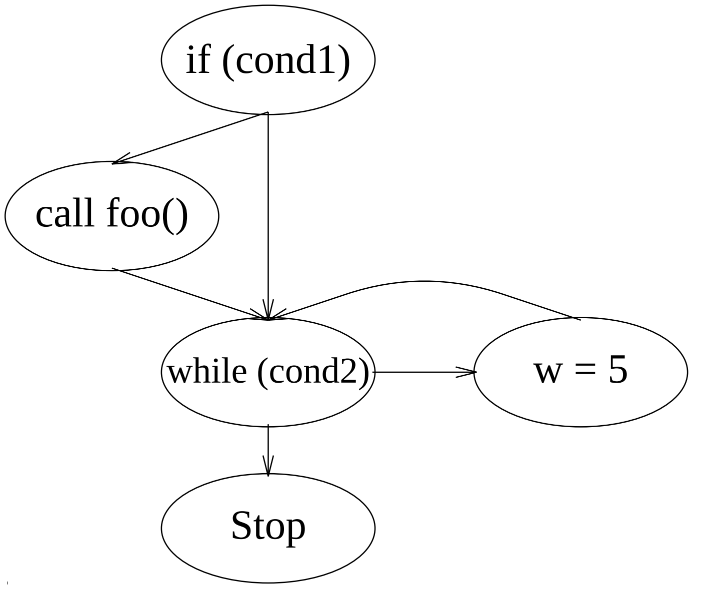
    call C

    call $main$

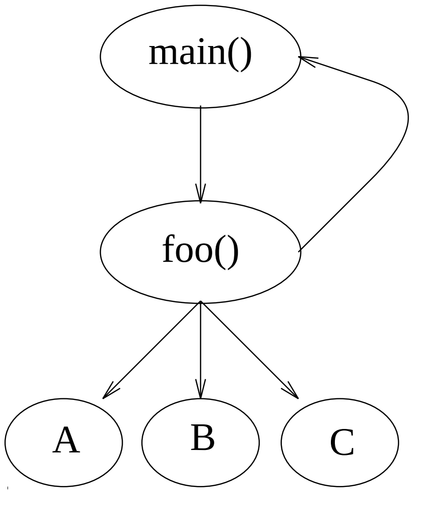
  fi

end

Control Flow Graph for main()



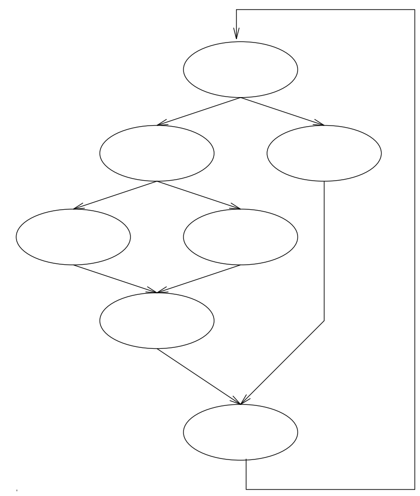
Procedure Call Graph



Flow graphs (cont'd)

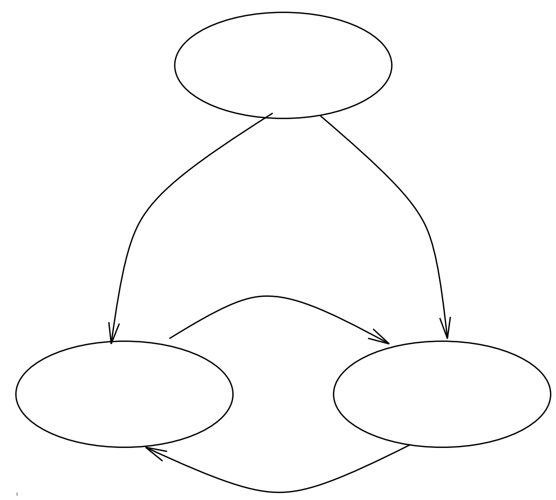
- Implicit exceptional control flow (division-by-zero, storage violations, etc.) can also be precisely modeled, but many analyzers do not account for exceptions that cause a program to halt.
- The procedure call graph is useful in programming environments (43, 14) and for solving certain interprocedural problems (4, 11, 24, 13).
- Notably absent from the procedure call graph are edges that represent *return* from procedure calls, although more sophisticated representations are possible (49, 16).
- An approach that obtains a uniform view of intra- and interprocedural flow is *continuation-passing style*, in which anything executable is modeled as a function, supplied with continuation expressions to accomplish future action (6).

A flow graph is...



structured if its associated procedure consists of the standard compound statements, structured loops, and structured branching constructs. A more precise definition requires graph-grammars (34) or control dependence (35).

reducible if every loop has a single point of entry. By applying a simple graph-grammar, such graphs can be reduced to a single node; moreover, applying that graph-grammar to an irreducible graph always results in the graph shown to the right (36).



Generally, analysis can be faster for structured or reducible flow graphs; however, the best algorithms work for arbitrary graphs while obtaining the performance of specialized algorithms on specialized graphs.

A more detailed classification of graph structure is given by Baker (8) and Sharir (64).

Flow graph data structures

The following data structures simplify and facilitate program optimization:

DFST is the depth-first spanning tree of a flow graph, constructed by depth-first search (1).

Dom is the (immediate) dominator tree of the flow graph, constructed by transitive reduction of a full dominator graph (2) or by a more clever and direct algorithm (47).

PDom is the (immediate) postdominator tree of the flow graph, constructed by finding dominators of the reverse flow graph (35).

DomFron is the dominance frontier graph, constructed by one pass over the dominator tree (28).

CD is the control dependence relation, constructed by finding dominance frontiers of the reverse flow graph (28).

Intervals are the results of partitioning a flow graph's nodes into single-entry (perhaps strongly-connected) loops (3, 66, 63).

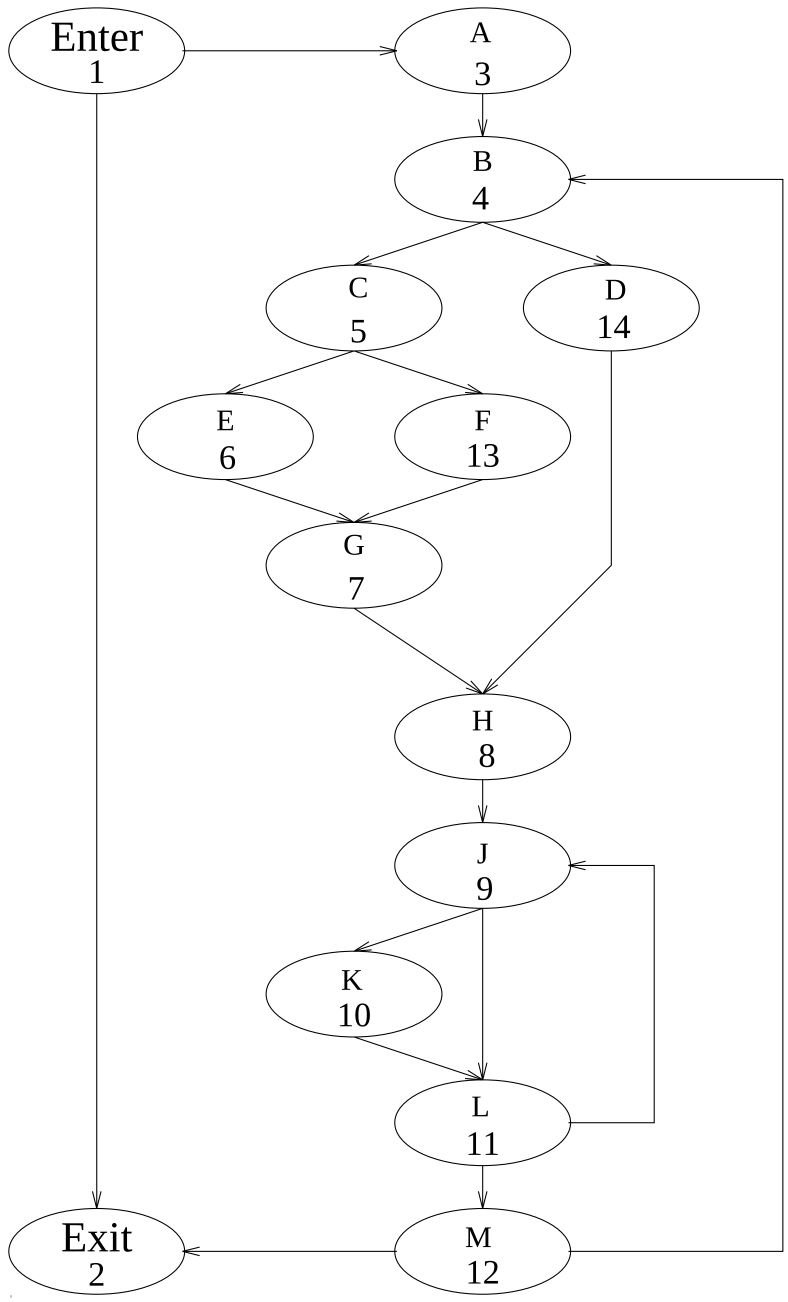
Depth-first spanning trees

```
num ← 0 ; root ← ⊥ ; child(⋆) ← ⊥ ; parent(⋆) ← ⊥ ; dfn(⋆) ← 0
foreach (Z ∈ N_f) do
    if (dfn(Z) = 0) then
        call DFS(Z)
        Sibling(Z) ← root ; root ← Z
    fi
od

Procedure DFS(X)
    num ← num + 1 ; dfn(X) ← num ; vertex(num) ← X
    foreach (Y ∈ Succ(X)) do
        if (dfn(Y) = 0) then
            parent(Y) ← X ; sibling(Y) ← child(X) ; child(X) ← Y
            call DFS(Y)
        fi
    od
    progeny(X) ← num − dfn(X)
end
```

Depth-first spanning trees (cont'd)

Here is shown a possible depth-first numbering of a flow graph. Although unvisited nodes can be considered in any order, the numbering on the right is obtained when:

- $Enter$ **is the first node visited;**
- **Unvisited successors of a branch node are considered "left to right".**



Depth-first spanning trees (cont'd)

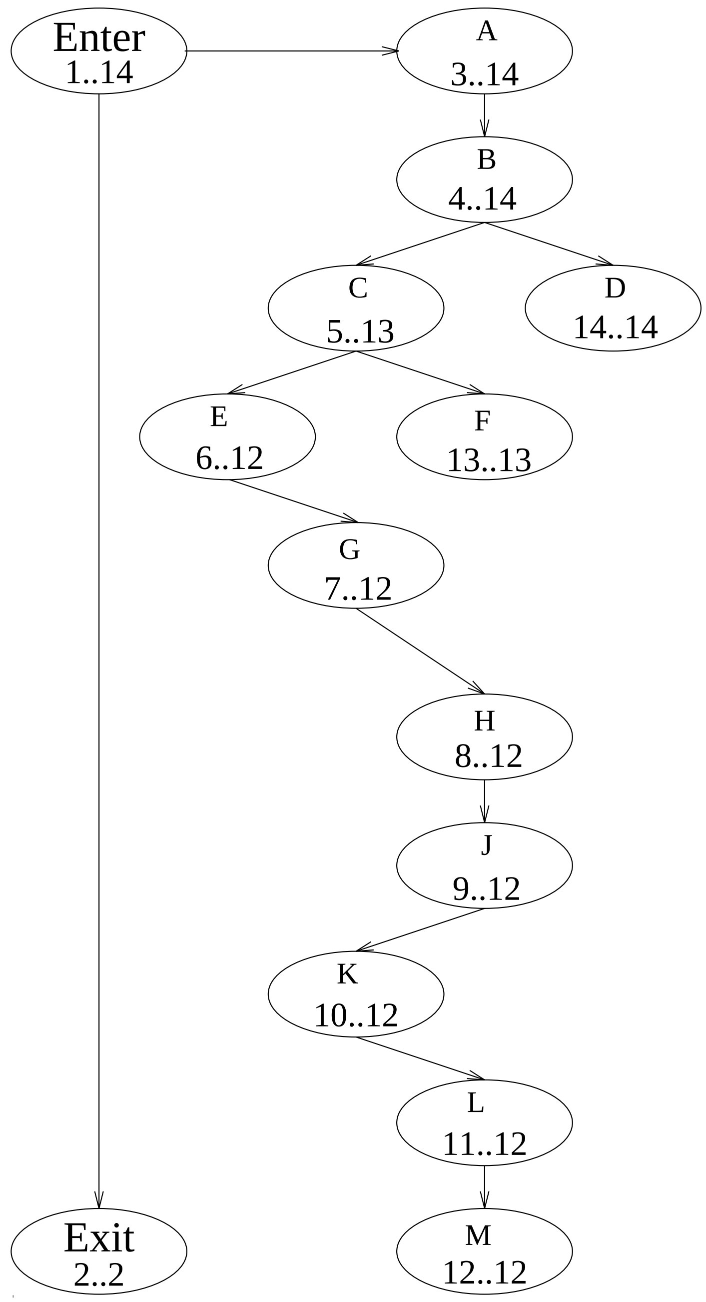
Here is shown the associated depth-first spanning tree. Each node X is labeled with the range of node numbers in its depth-first spanning subtree:

$$dfn(x) \ldots dfn(x) + progeny(X)$$

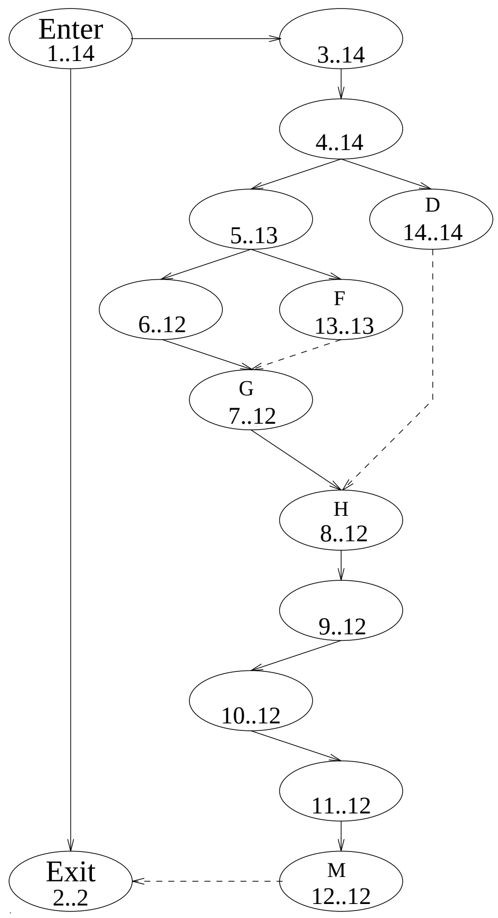
as computed by the algorithm.

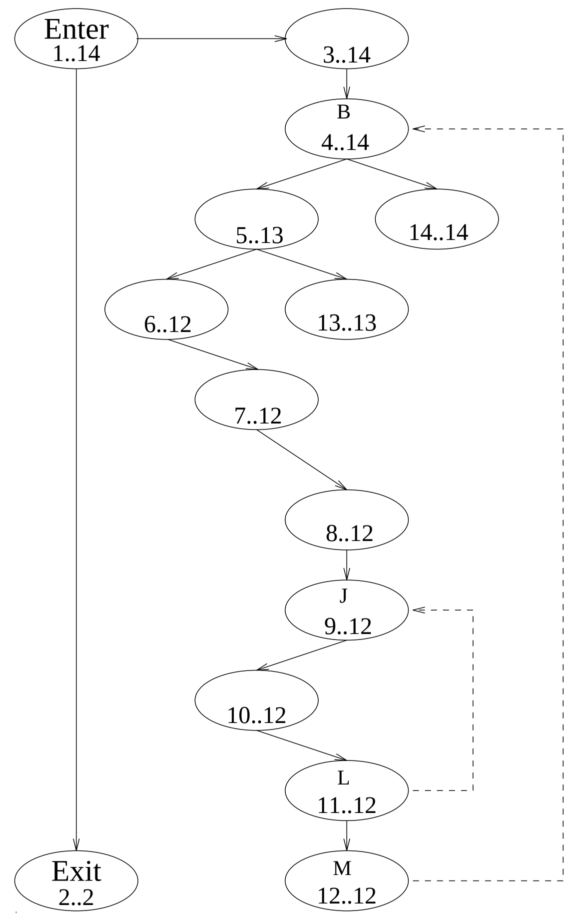
By retaining $progeny(X)$, we obtain an $O(1)$ test for Y's inclusion in X's depth-first spanning subtree: we need only test $dfn(Y)$ for inclusion in the labeled range for X. Where Y is in the depth-first spanning subtree rooted at X, we'll use the notation

$$X \lhd Y$$



Depth-first spanning trees (cont'd)





The dashed edges above are *cross* edges with respect to our DFST. Pictorially, such edges go right-to-left.

The dashed edges above are *back* edges with respect to our DFST; such edges represent loops in the graph.

Both types of edges are from X to Y, $dfn(X) \geq dfn(Y)$; however, $Y \lhd X$ for a back edge and $Y \not\lhd X$ for a cross edge. A *chord* edge goes from X to Y, $X \lhd Y$ (e.g., flow graph edge $J \rightarrow L$, not shown).

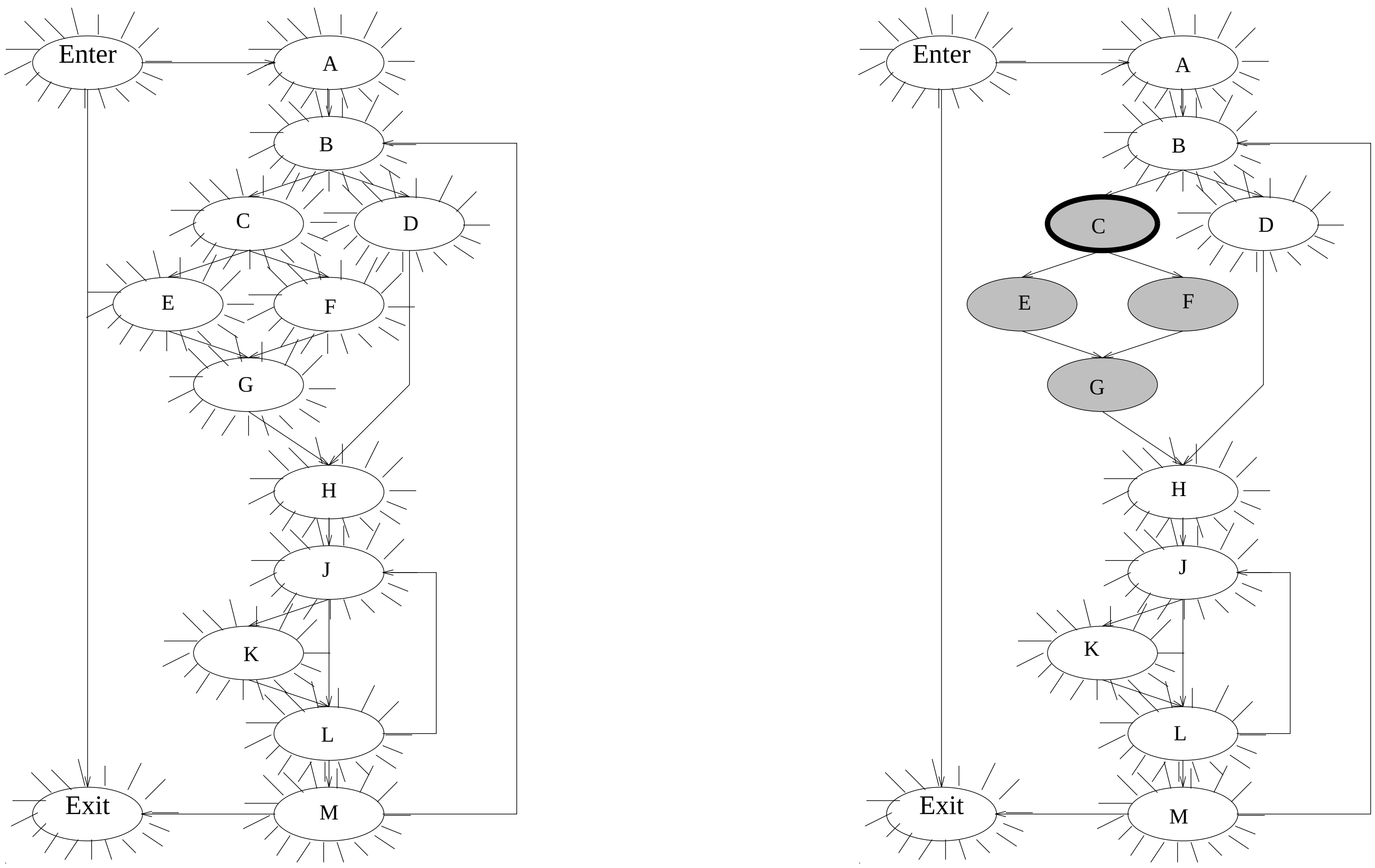
Dominance

- Node Y *dominates* node Z, denoted $Y \geqslant Z$, if every path from the graph's $root$ to Z includes node Y.
  During program execution, if node Z has executed, then so has node Y.
- A node always dominates itself.
- Node Y *strictly dominates* Z, denoted $Y \gg Z$, if $Y \geqslant Z$ and $Y \neq Z$. A node never strictly dominates itself.
- The *immediate dominator* of node Z, denoted $idom(Z)$, is the *closest* strict dominator of Z:

$$Y = idom(Z) \Longleftrightarrow (Y \gg Z \text{ and } \forall\, X \gg Z, X \geqslant Y)$$

- The *dominator tree* for $\mathcal{G}_f$ has nodes $\mathcal{N}_f$; Y is a parent of Z in this tree if and only iff $Y = idom(Z)$.

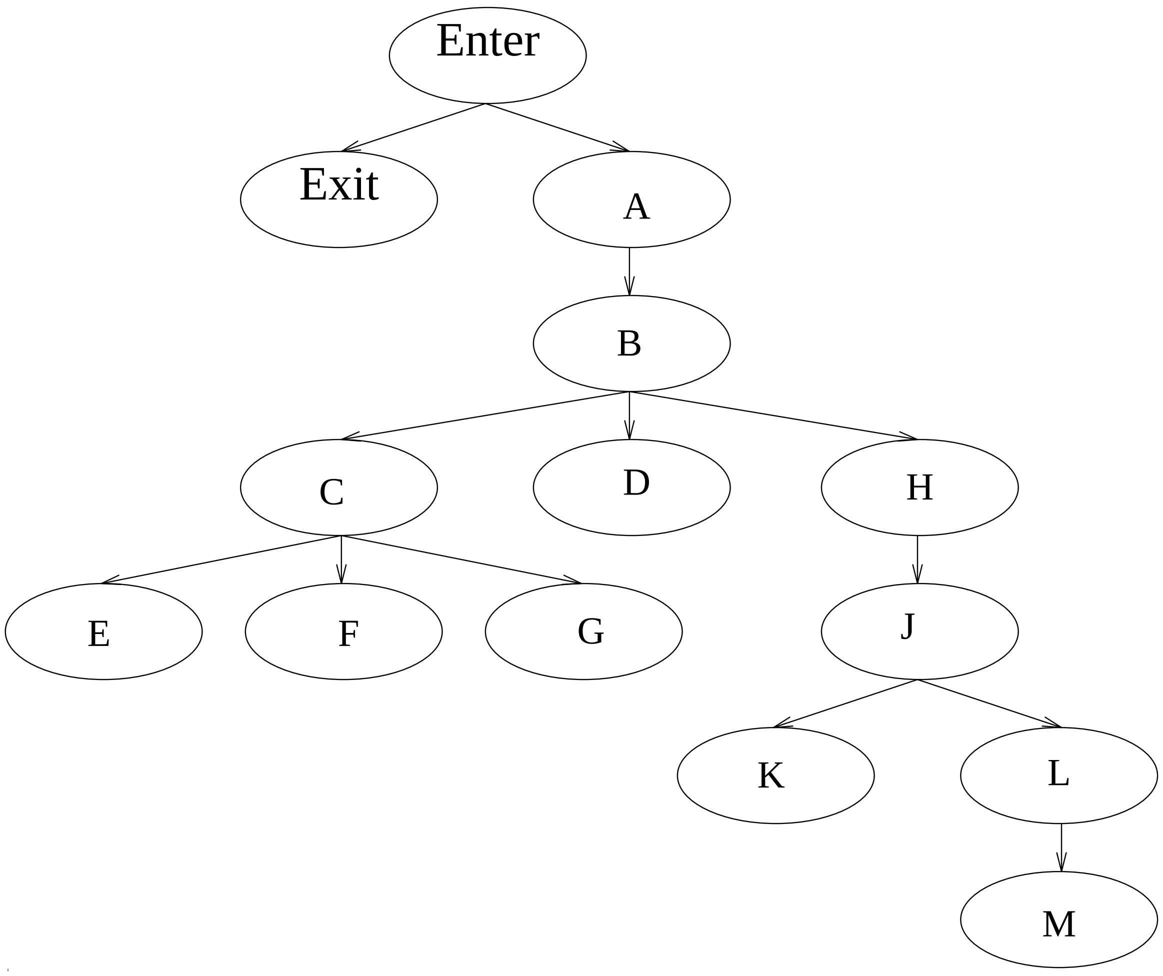
Fiber-optic view of dominance



- $Start$ is a light source;
- each edge conducts light from its source to its target;
- each node normally transmits light received on any in-edge to each out-edge.

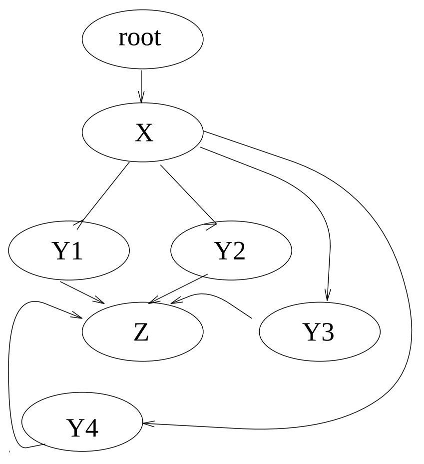
Nodes dominated by X are “cast in shadow” when X is made *opaque* (transmission of light through X is prevented).

Dominator tree



This is the most compact representation of dominance, requiring only two “pointers” per node: one for the node’s leftmost child and one for the node’s right sibling. In the tree representation, a node dominates all descendants, strictly dominates all proper descendants, and immediately dominates all children.

Dominance (cont'd)



If some node X dominates each predecessor Y_i of Z, then X appears on every path from $root$ to Z and so X dominates Z.

Let $dom(Z)$ be the set of nodes that dominate Z. A recursive way of specifying $dom(Z)$ is:

- **Z is in $dom(Z)$;**
- **Any node dominating each predecessor of Z also dominates Z**

We then obtain the following equation at each node Z:

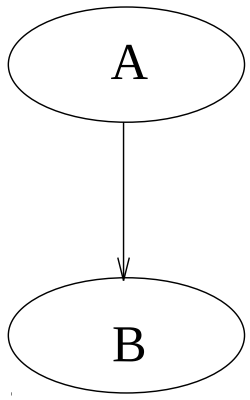
$$dom(Z) = \{ Z \} \cup \bigcap_{Y \in Preds(Z)} dom(Y)$$

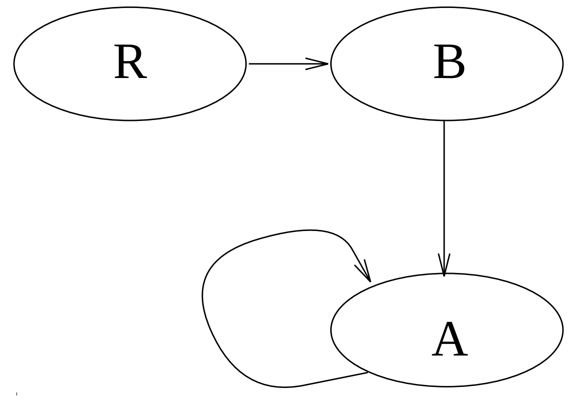
We essentially obtain a simple forward data flow framework, in which

- meet is set intersection;
- the transfer function for each node Z is

$$f_Z(IN) = \{ Z \} \cup IN$$

Dominance as a data flow problem





Writing a suitable equation at each node, we obtain:

$$\begin{aligned} dom(A) &= \{A\} \\ dom(B) &= \{B\} \cup dom(A) \end{aligned}$$

On inspection, it's clear how to evaluate these equations to obtain:

$$\begin{aligned} dom(A) &= \{A\} \\ dom(B) &= \{A, B\} \end{aligned}$$

$$\begin{aligned} dom(R) &= \{R\} \\ dom(B) &= \{B\} \cup dom(R) \\ dom(A) &= \{A\} \cup \\ &\quad (dom(A) \cap dom(B)) \end{aligned}$$

The first two equations are easily evaluated:

$$\begin{aligned} dom(R) &= \{R\} \\ dom(B) &= \{R, B\} \end{aligned}$$

The equation for $dom(A)$ is problematic, because the solution for $dom(A)$ depends on itself.

In general, solving such a system of equations involves an initial approximation for the solution at each node.

$$dom(A) = \{ A \} \cup \left(dom(A) \cap dom(B) \right)$$

Suppose we initially approximate

$$dom(Z) = \emptyset$$

for each node Z.

We then obtain

$$\begin{aligned} dom(A) &= \{ A \} \cup (\emptyset \cap dom(B)) \\ &= \{ A \} \end{aligned}$$

The above equation represents a *fixed point*:

1. Every node in $dom(A)$ truly dominates A;
2. Further evaluation (substitution of $dom(A)$) does not change any solution.

However, we did not obtain the best, or *maximum* fixed point, in which each solution would contain the "largest" possible set of nodes that satisfies the equations.

Suppose our initial approximation were

$$dom(Z) = \mathcal{N}_f$$

We would then obtain

$$\begin{aligned} dom(A) &= \{ A \} \cup (\mathcal{N}_f \cap dom(B)) \\ &= \{ R, A, B \} \end{aligned}$$

which is indeed a correct solution, "larger" than $\{ A \}$.

In any data flow problem, the best answer is obtained by initially assuming each solution is $\top$, and then iterating to a fixed point. We may informally reason now that

$$\top = \mathcal{N}_f$$

for the dominance problem, because dominators allow latitude for code motion. A more formal basis for such reasoning will surface shortly.

Simple dominance algorithm

The loop at $\boxed{2}$ performs the fixed point computation, based on the initialization at $\boxed{1}$. The set of nodes whose solution may have changed is maintained in the work list $Wlist$. The equation for node Y is recomputed at $\boxed{4}$. Step $\boxed{5}$ detects any change in solution, in which case successors of Y are added to the work list at $\boxed{6}$.

When the work list is empty, each equation has stabilized and the algorithm is finished.

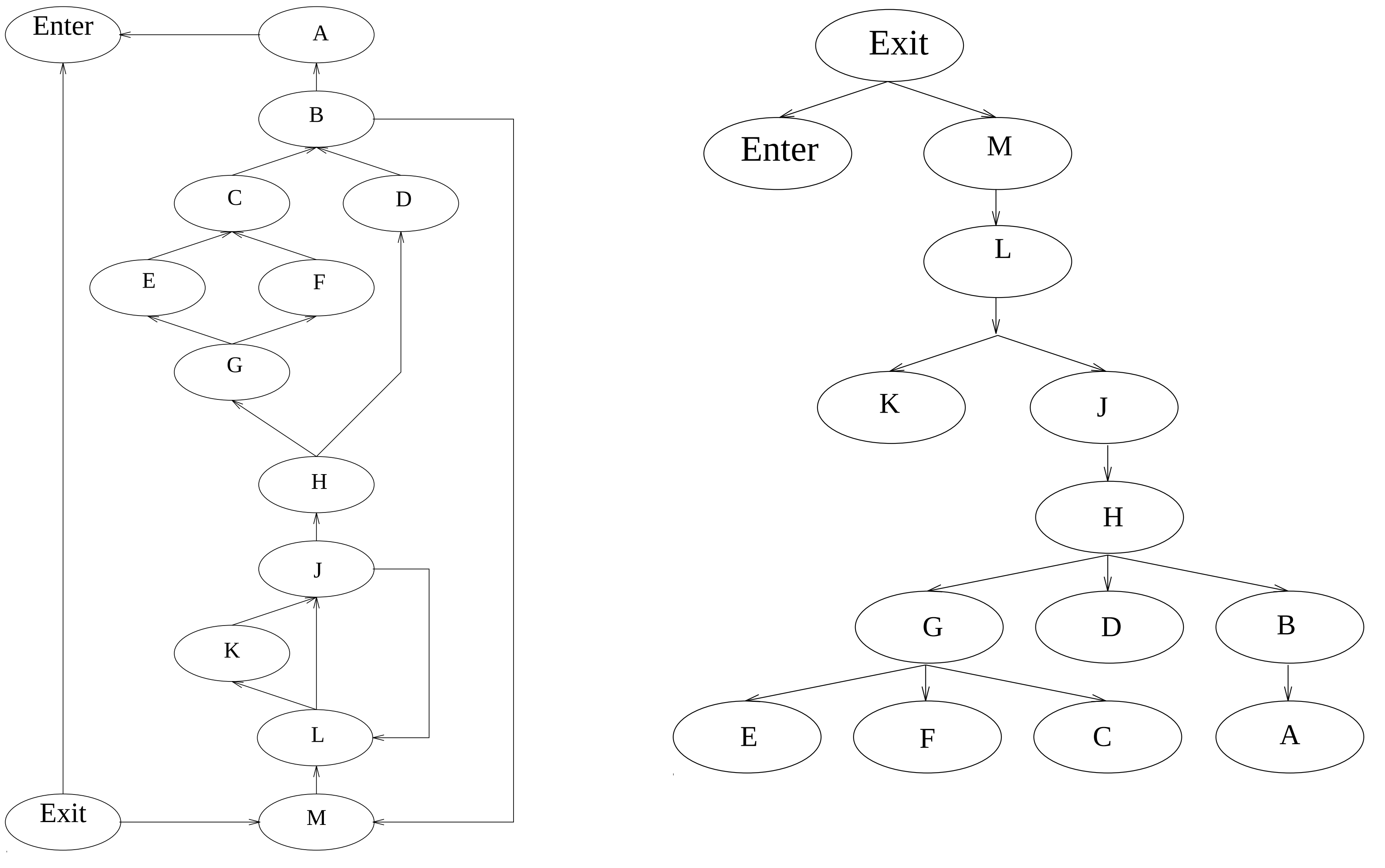
$dom(\star) \leftarrow \mathcal{N}_f$ $\Leftarrow \boxed{1}$
$Wlist \leftarrow$ root
while $(Wlist \neq \emptyset)$ **do** $\Leftarrow \boxed{2}$
    $Y \leftarrow$ element from $Wlist$ $\Leftarrow \boxed{3}$
    $Wlist \leftarrow Wlist - \{ Y \}$
    $ndom \leftarrow \{ Y \} \cup \bigcap_{(X,Y) \in \mathcal{E}_f} dom(X)$ $\Leftarrow \boxed{4}$
    if $(ndom \neq dom(Y))$ **then** $\Leftarrow \boxed{5}$
        $dom(Y) \leftarrow ndom$
        foreach $((Y, Z) \in \mathcal{E}_f)$ **do** $\Leftarrow \boxed{6}$
            $Wlist \leftarrow Wlist \cup \{ Z \}$
        od
    fi
od

The dominators algorithm is written this way to expose its data flow nature.

Fast dominance algorithm

- The dominance relation can take $O(|\mathcal{N}_f|^2)$ space and $O(|\mathcal{N}_f|^2|\mathcal{E}_f|)$ time to compute.
- A faster algorithm computes immediate dominance directly (47):
  1. The graph $\mathcal{G}_f$ is distilled into a dominator-equivalent graph free of cross- and back-edges.
  2. Immediate dominators are found for a graph containing only tree and chord edges.

Postdominance



To compute postdominators, simply compute dominators of the reverse flow graph. Postdominance is useful for code motion (26) and for computing control dependence (35).

Dominance frontiers [28]

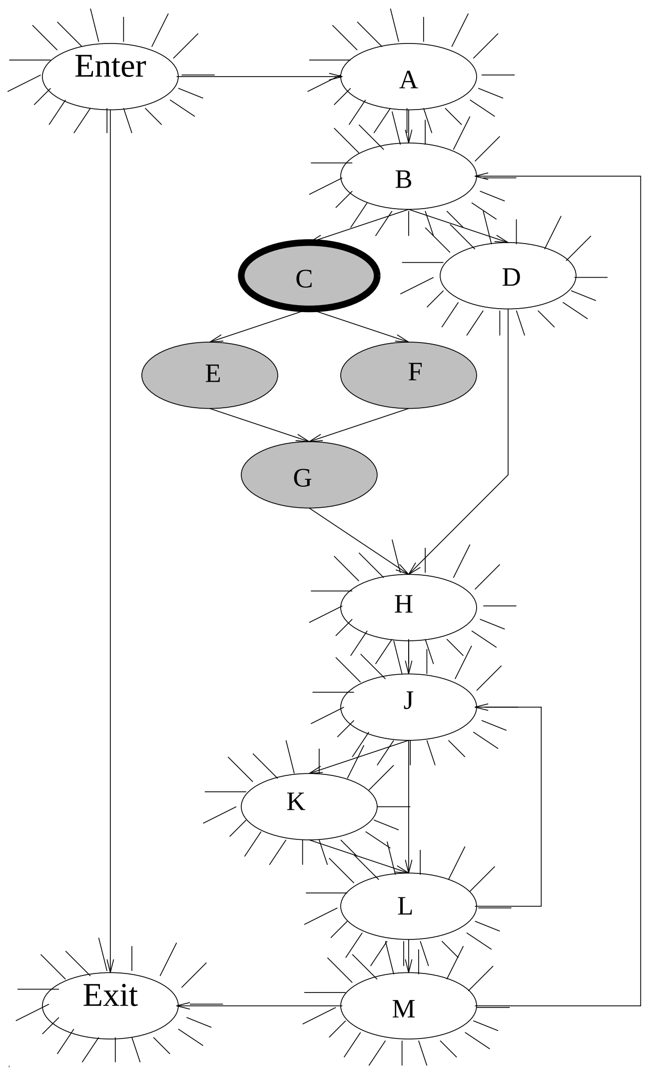
traverse tree $(DomTree)$ **order** (BottomUp) **at node** (X) **do**
    $DF(X) \leftarrow \emptyset$
    foreach $(Y \in Succ(X))$ **do**
        if $(idom(Y) \neq X)$ **then**
            $DF(X) \leftarrow DF(X) \cup \{ Y \}$ $\Leftarrow \boxed{1}$
        fi
    od
    foreach $(Z \in Children(X))$ **do**
        foreach $(Y \in DF(Z))$ **do**
            if $(idom(Y) \neq X)$ **then**
                $DF(X) \leftarrow DF(X) \cup \{ Y \}$ $\Leftarrow \boxed{2}$
            fi
        od
    od
od

Node Z is in the dominance frontier of node X if X dominates a predecessor Y of Z but X does not strictly dominate Z (28):

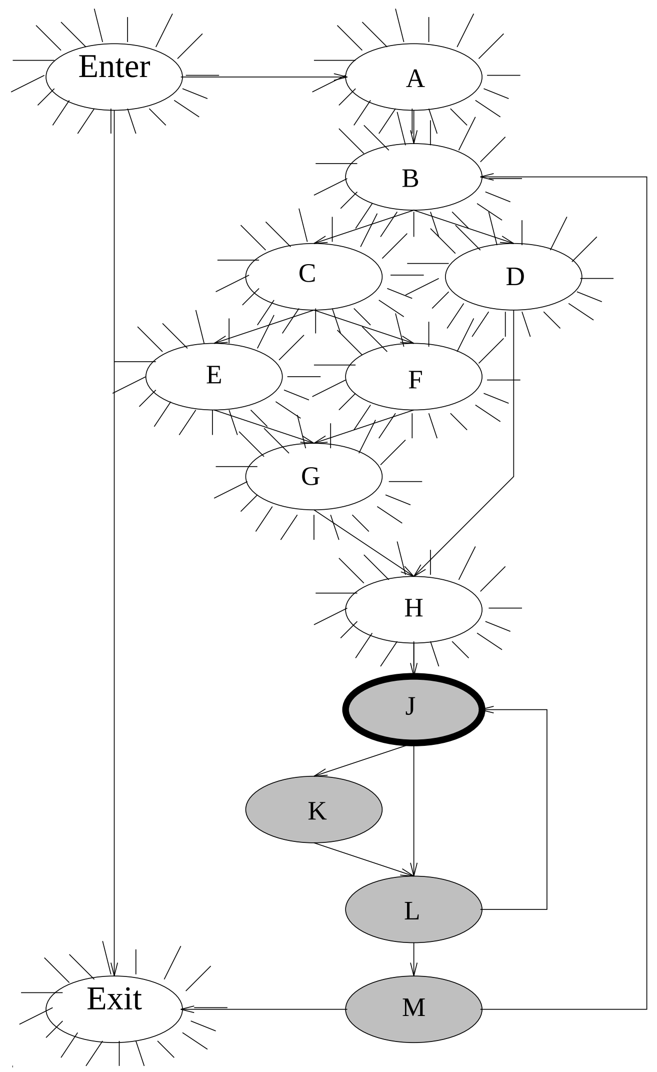
$$DF(X) = \{ Z \mid (\exists\, Y \in Preds(Z))(X \geqslant Y \text{ and } X \not\gg Y) \}$$

Step $\boxed{1}$ finds successors of X in $DF(X)$; step $\boxed{2}$ completes $DF(X)$ with nodes selected from dominance frontiers of nodes immediately dominated by X.

Fiber-optic view of dominance frontiers



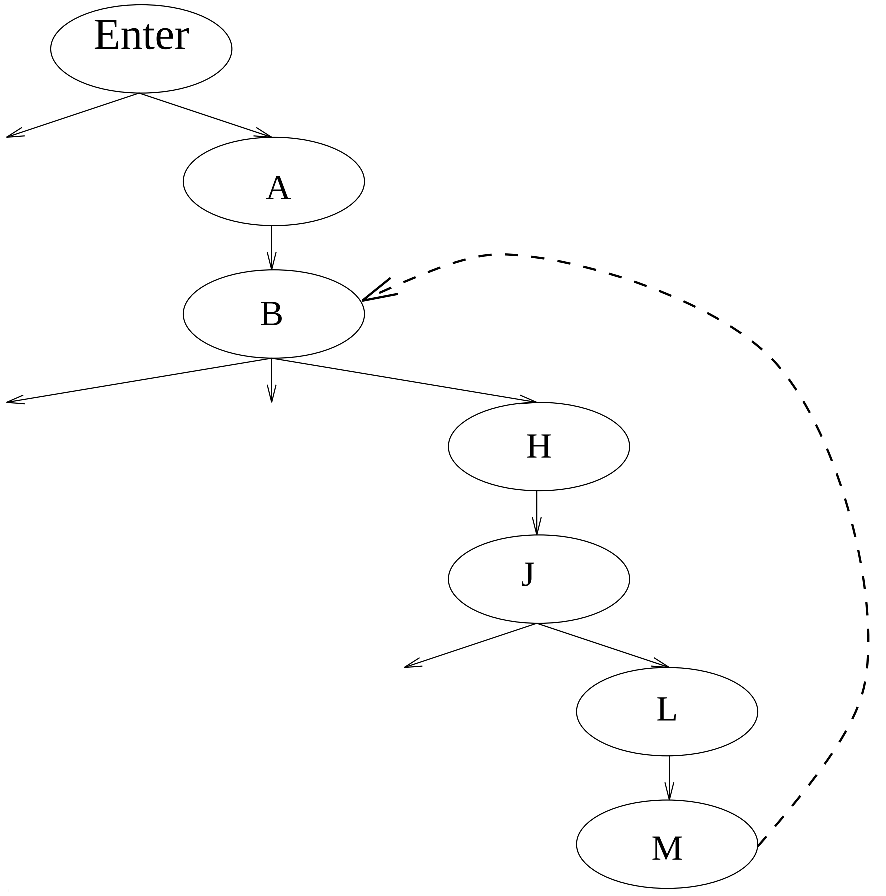
$$DF(C) = \{\, H \,\}$$



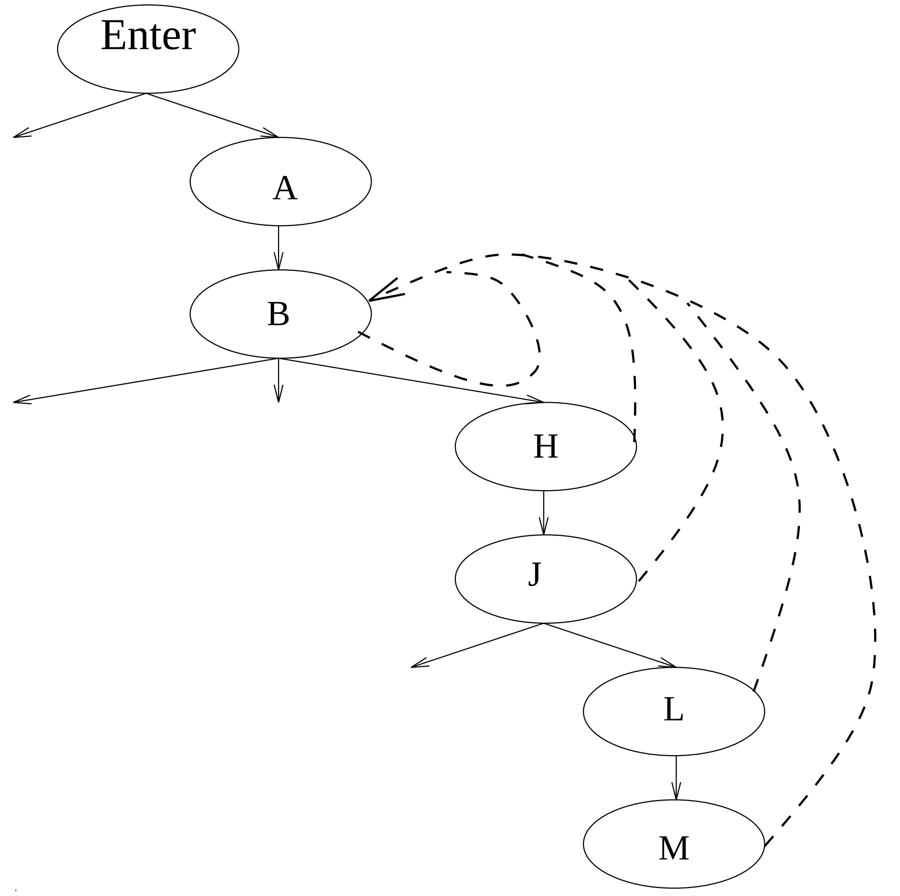
$$DF(J) = \{\, J, Exit \,\}$$

Nodes just outside the dominance "shadow" cast by X are in $DF(X)$.

Dominance frontiers (cont'd)



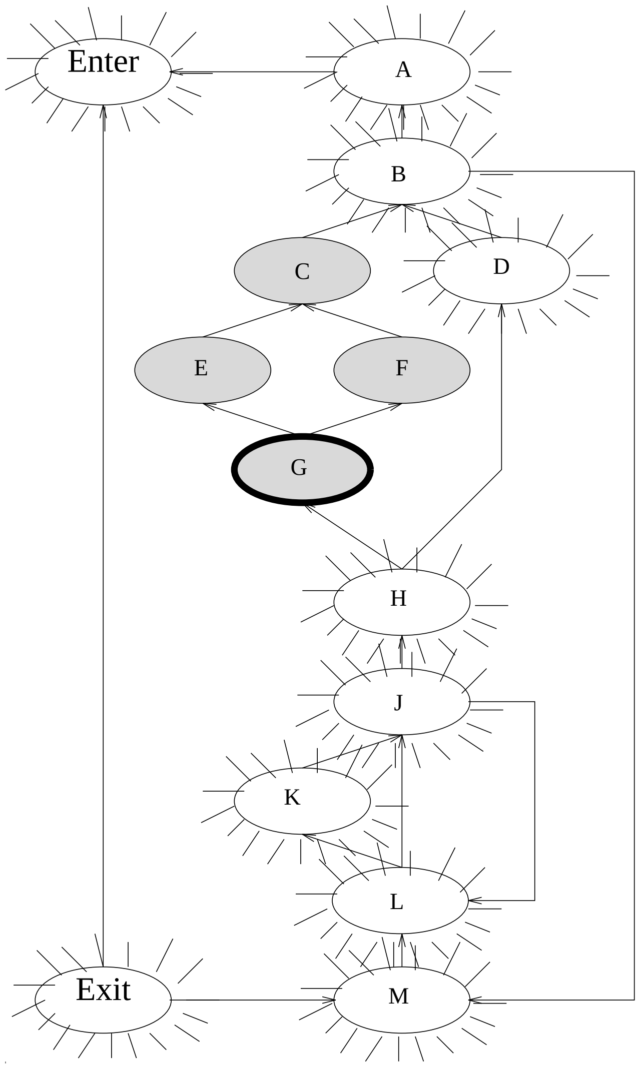
Another way to visualize dominance frontiers is to start with a flow graph edge superimposed on the dominator tree, such as the edge (M, B)...



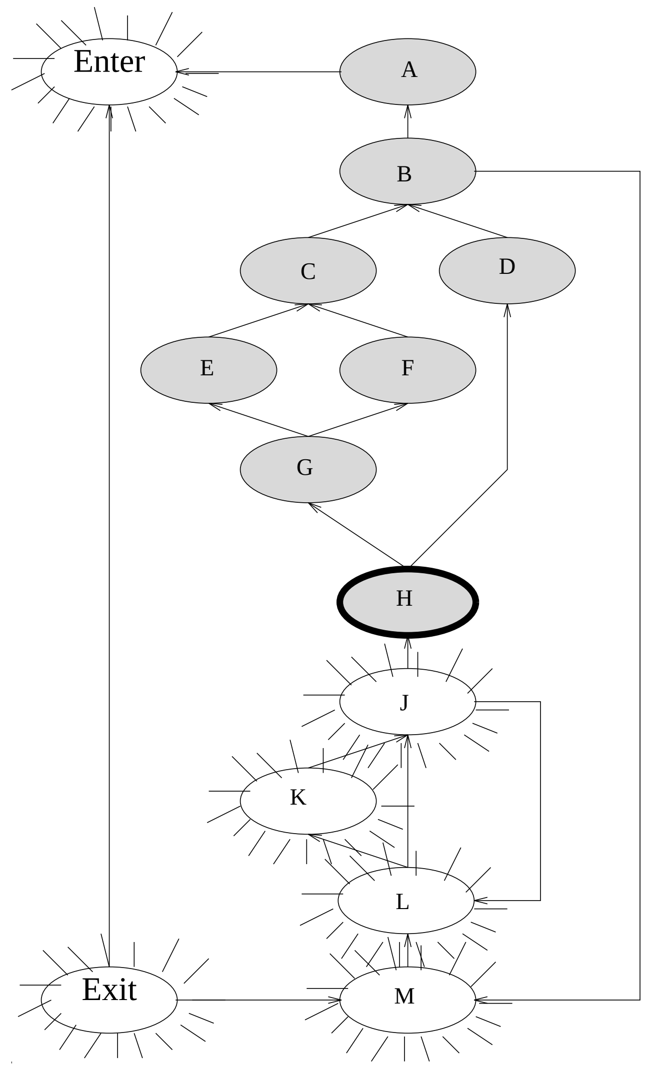
and keep sliding the source of the edge up the tree. For each position where the tail is not above the head, the target of the edge is in the dominance frontier of the source.

So, B is in $DF(M)$, $DF(L)$, $DF(J)$, $DF(H)$, and $DF(B)$.

Control dependence



Node B controls node G by edge $B \rightarrow C$.



Node $Enter$ controls H by edge $Enter \rightarrow A$; node M controls H by edge $M \rightarrow B$.

To compute control dependence, find the inverse of dominance frontiers of the reverse flow graph (28), but qualify the relation with the in-edges at ϕ-functions.

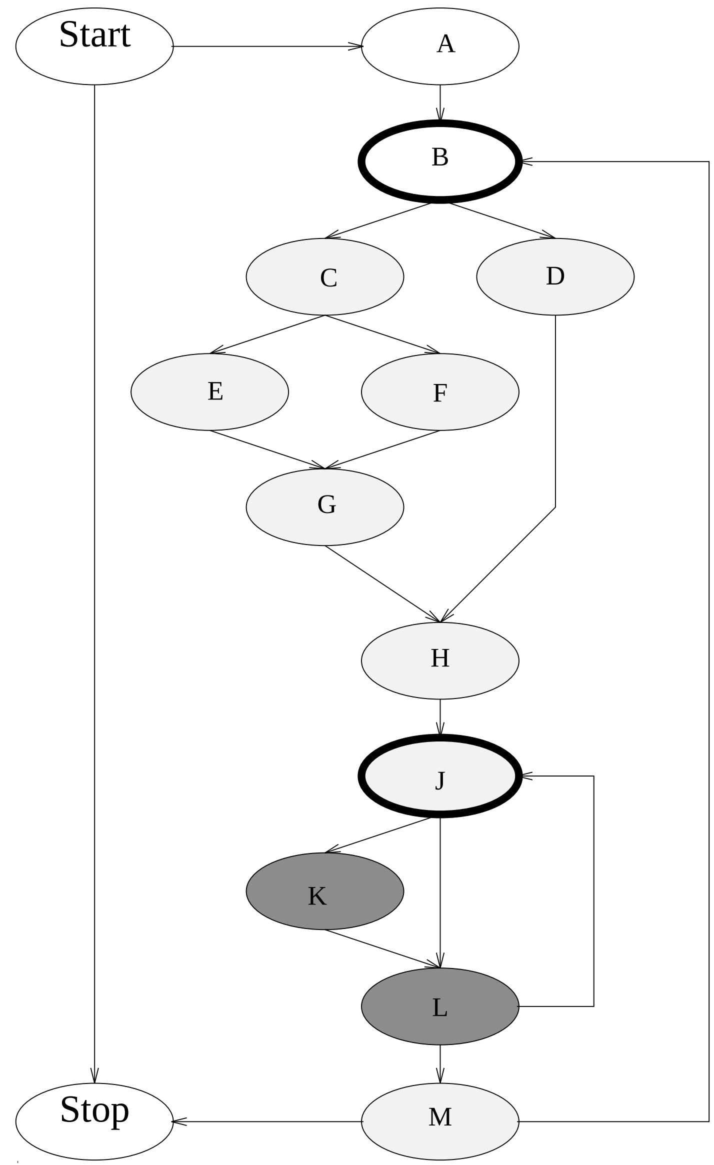
Intervals [63]

Algorithm summary:

1. Find back edges of the flow graph. Each target of a back edge is an interval header;
2. Find and collapse members of intervals, inside-out.
3. Path-compression helps speed the search across intervals that have already been identified.

In the example to the right,

- Highlighted nodes are interval headers.
- Dark-shaded nodes are in the innermost loop.
- Light-shaded nodes are in the outermost loop.
- The remaining nodes are not in a loop, but may be regarded as belonging to the outermost interval.



Formal specification of a data flow framework

The *data flow graph*

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

has been described previously:

- **its edges are oriented in the direction of the data flow problem;**
- $\mathcal{G}_{df}$ **is augmented with nodes** $Start$ **and** $Stop$ **and an edge** $(Start, Stop)$**, suitably inserted with respect to the direction of the data flow problem.**

Successors and predecessors are also defined with respect to the direction of the data flow problem:

$$\begin{aligned} Succs(Y) &= \{ Z \mid (Y, Z) \in \mathcal{E}_{df} \} \\ Preds(Y) &= \{ X \mid (X, Y) \in \mathcal{E}_{df} \} \end{aligned}$$

The *meet semilattice* **is**

$$L = (A, \top, \bot, \preceq, \wedge)$$

A **is a set (usually a powerset), whose elements form the domain of the data flow problem,**

$\top$ and $\bot$ **are distinguished elements of** A**, usually called "top" and "bottom", respectively,**

$\preceq$ **is a reflexive partial order, and**

$\wedge$ **is the associative and commutative** *meet* **operator, such that for any** $a, b \in A$**,**

$$\begin{array}{ccc} a \preceq b & \Longleftrightarrow & a \wedge b = a \\ a \wedge a & = & a \\ a \wedge b & \preceq & a \\ a \wedge b & \preceq & b \\ a \wedge \top & = & a \\ a \wedge \bot & = & \bot \end{array}$$

These rules allow formal reasoning about $\top$ **and** $\bot$ **in a framework.**

Formal specification (cont'd)

The set F of *transfer functions*

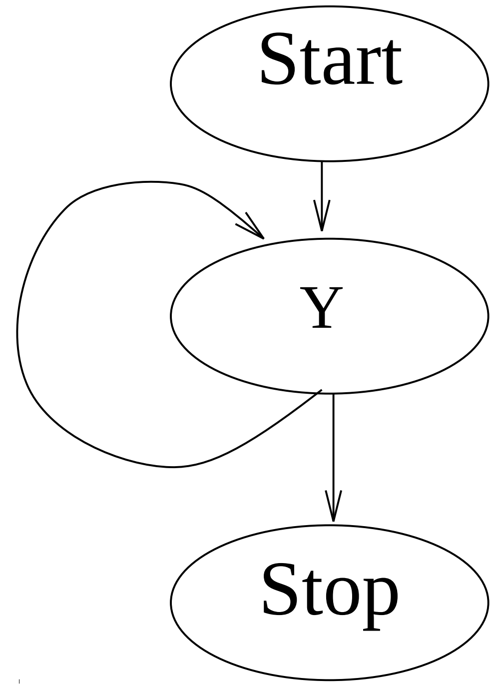
$$\mathcal{F} \subseteq \{f : L \mapsto L\}$$

has elements for describing the behavior of any flow graph node with respect to the data flow problem.

To obtain a stable solution, we'll require the functions in $\mathcal{F}$ to be *monotone***:**

$$(\forall f \in \mathcal{F})(\forall x, y \in L)$$
$$x \preceq y \rightarrow f(x) \preceq f(y)$$

In other words, a node cannot produce a "better" solution when given "worse" input. Given a two-level lattice, evaluation of the data flow graph shown to the right oscillates between solutions and never reaches a fixed point.



$$f_Y(IN) = \begin{cases} \top & \textbf{if } IN = \bot \\ \bot & \textbf{if } IN = \top \end{cases}$$

Examples of data flow frameworks: Dominators

Data flow graph is the forward control flow graph of a procedure.

Lattice

A is $2^{\mathcal{N}_f}$: the powerset of the flow graph's nodes. That is, any solution is a subset of $\mathcal{N}_f$;

$\top$ and $\perp$ are $\mathcal{N}_f$ and $\emptyset$, respectively;

$\preceq$ defined as

$$(a \preceq b) \Longleftrightarrow (a \subseteq b)$$

For example, $\{\,A, B, C\,\} \preceq \{\,A, B, C, D\,\}$, but solutions $\{\,A, B\,\}$ and $\{\,B, C\,\}$ are incomparable, and thus unrelated by $\preceq$.

$\wedge$ is set intersection, which legitimizes $\top$ and $\perp$ since

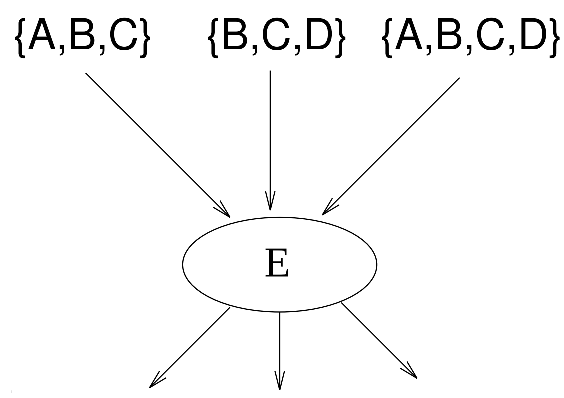
$$\begin{aligned} a \wedge \top &= a \wedge \mathcal{N}_f \\ &= a \\ a \wedge \perp &= a \wedge \emptyset \\ &= \emptyset \end{aligned}$$

Transfer function at each node Z is

$$f_Z(IN) = IN \cup \{\,Z\,\}$$

Examples: dominators (cont'd)

Consider evaluation of the transfer function at node E below:



Let's compute the *meet-over-all-paths* **(MOP) solution:**

1. **Let $AllPaths(E)$ be the set of all paths in the graph that terminate with node E, and let IN_p be the solution input to E on path p;**
2. $OUT = \bigwedge_{p \in AllPaths(E)} f_E(IN_p)$

In our example, we would compute:

Edge j	IN	$f_E(IN_j)$
1	$\{A, B, C\}$	$\{A, B, C, E\}$
2	$\{B, C, D\}$	$\{B, C, D, E\}$
3	$\{A, B, C, D\}$	$\{A, B, C, D, E\}$
OUT		$\{B, C, E\}$

Now, suppose that prior to applying $f_E()$, we took the meet of all input values. We would then compute

$$\begin{aligned} IN &= \{A, B, C\} \wedge \{B, C, D\} \wedge \{A, B, C, D\} \\ &= \{B, C\} \end{aligned}$$

$$\begin{aligned} OUT &= IN \cup \{E\} \\ &= \{B, C, E\} \end{aligned}$$

These two approaches compute the same solution for this framework; unfortunately there are some frameworks for which the MOP evaluation produces a more precise solution.

Examples of data flow frameworks: live variables

Data flow graph is the reverse control flow graph of a procedure.

Lattice

- A **is** 2^V**, where** V **is the set of the program's variables. Each element of** A **represents a set of** *live* **variables;**
- $\top$ and $\perp$ **are** $\emptyset$ **and** V**, respectively;**
- $\preceq$ **defined as**

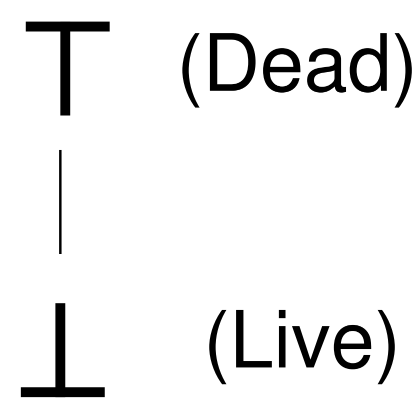
$$(a \preceq b) \Longleftrightarrow (a \supseteq b)$$

- $\wedge$ **is set union, which again legitimizes** $\top$ **and** $\perp$**.**

Transfer functions

$$f_Z(IN) = \begin{cases} IN \cup \{v\} & \text{if node } Z \text{ has an } \textit{upwards-exposed} \text{ use of } v \\ IN - \{v\} & \text{if node } Z \text{ kills } v \\ IN & \text{otherwise} \end{cases}$$

The lattice shown to the right occurs when the live variables data flow framework is specified only for a single variable v**. It turns out that live variables is** *partitionable* **(70, 48), and so the solution for a set of variables** V **can be easily constructed from the separate solutions of each** $v \in V$**.**



Examples of data flow frameworks: constant propagation

The object of constant propagation is to determine compile-time constants, to be substituted for run-time computations.

Data flow graph is the forward control flow graph of a procedure.

Lattice

A is the set of all total functions

$$t : V \mapsto (\mathcal{Z} \cup \{\top, \bot\})$$

Each element of A associates a value from $\mathcal{Z} \cup \{\top, \bot\}$ with each $v \in V$.

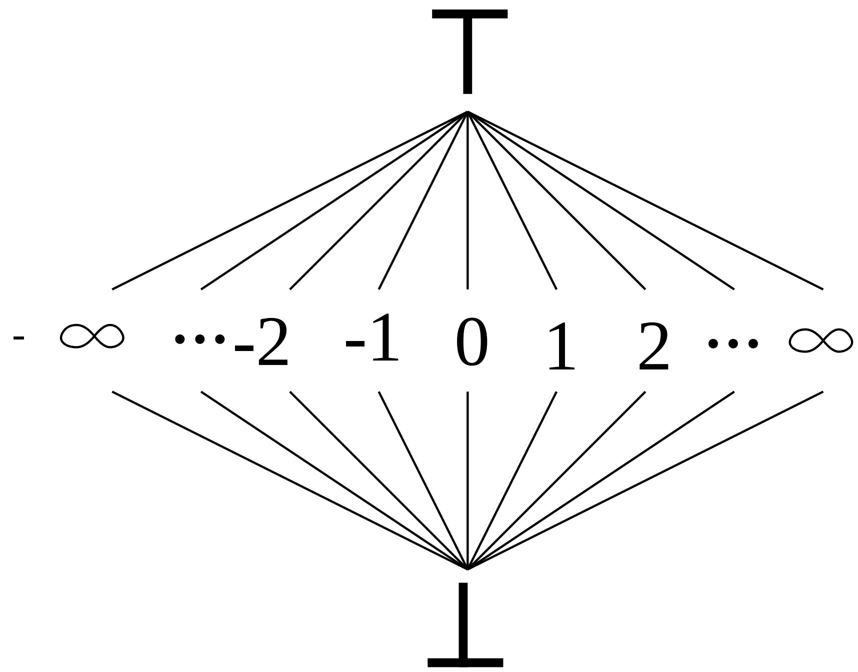
$\top$ and $\bot$ are $t(v) = \top$ and $t(v) = \bot$;

$\wedge$ is performed by-variable, using the lattice shown to the right.

$\preceq$ is defined as

$$(t_a \preceq t_b) \iff (\forall\ v\ t_a(v) \preceq t_b(v))$$

also using the lattice.



- $\top$ represents “any constant you like”;
- $\bot$ represents “not a constant value”;
- The meet of two disparate values for the same variable results in $\bot$.
- Since the (per-variable) lattice has depth greater than two, solutions cannot be represented by simple bit-vectors. Most implementations use a list of tuples (69).

Examples: constant propagation (cont'd)

Transfer functions: Recall that the domain of our lattice is a set of functions. Thus, a transfer function is a mapping from one function to another. When we write

$$OUT = f_Y(IN)$$

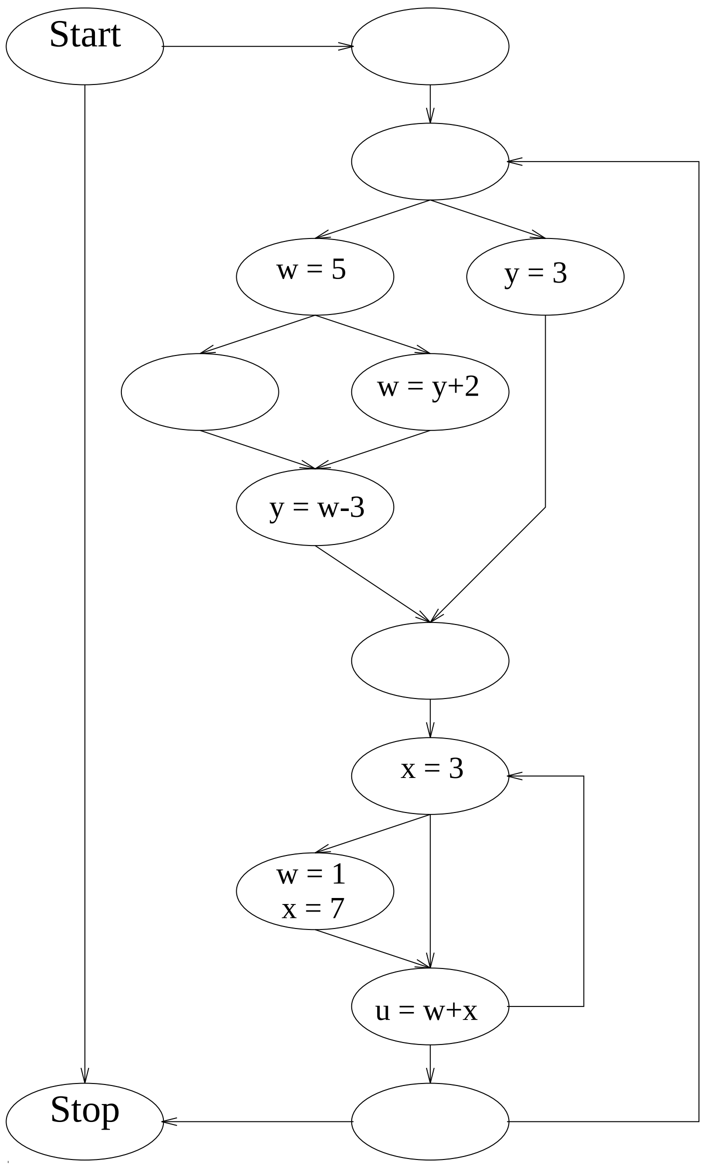
for the transfer function at node Y, IN and OUT are themselves functions from V to $\mathcal{Z} \cup \{\top, \bot\}$. The transfer function for a node Y is then formulated as follows:

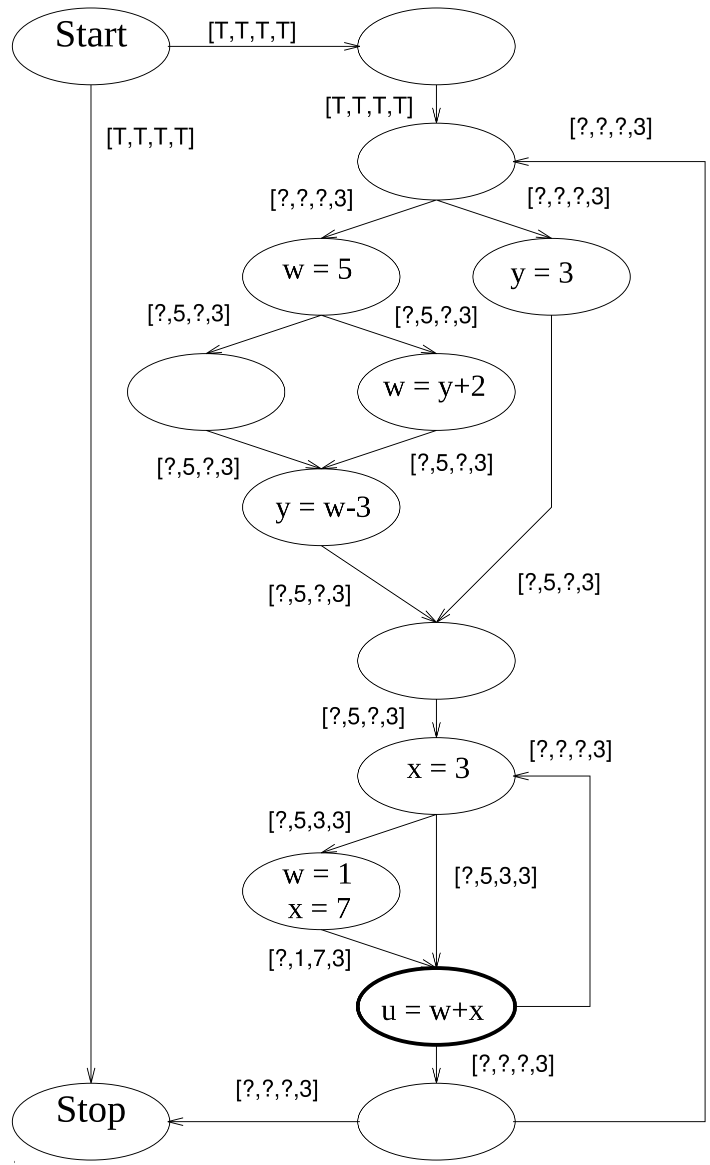
- **For each variable v whose value is not affected by Y, $OUT(v) = IN(v)$;**
- **For each variable v whose value is changed by the expression ψ, $OUT(v)$ is computed as follows:**
  - **If any variable $y \in \psi$ has $IN(y) = \bot$, then $OUT(v) = \bot$;**
  - **Otherwise, if any variable $y \in \psi$ has $IN(y) = \top$, then $OUT(v) = \top$;**
  - **Otherwise, each $y \in \psi$ has a constant value, and $OUT(v)$ is set to that value.**

Examples of transfer functions:

Statement	Transfer function $f(IN) =$
`w = 5`	$\begin{cases} IN(v) & \textbf{if } v \neq w \\ 5 & \textbf{if } v = w \end{cases}$
`w = y+2`	$\begin{cases} IN(v) & \textbf{if } v \neq w \\ \top & \textbf{if } v = w \textbf{ and } IN(y) = \top \\ \bot & \textbf{if } v = w \textbf{ and } IN(y) = \bot \\ IN(y) + 2 & \textbf{otherwise} \end{cases}$

Examples: constant propagation (cont'd)





Solutions in this example are given as $[\,u, w, x, y\,]$. Here, MOP gives a better result after the highlighted node: Taking the meet of $[\,\perp, 1, 7, 3\,]$ and $[\,\perp, 5, 3, 3\,]$ yields $[\,\perp, \perp, \perp, 3\,]$ on input, so u is computed as $\perp$. However, $(w + x)$ is 8 on each input edge.

Evaluation of a data flow framework

Although more specialized methods exist (46, 17, 48, 70), data flow frameworks are usually evaluated by one of the following methods:

Iteration (37, 45)

1. **The flow graph is preprocessed to obtain a clever node ordering NL.**
2. **Within each iteration, nodes are visited in order NL:**
   (a) **The input to a node Y is evaluated as the** *meet* **of the solutions at Y's predecessors.**
   (b) **The transfer function across Y is evaluated on that input.**
   (c) **The solution is made available at each of Y's successors.**
3. **Iteration continues until** *convergence* **is reached, by predetermining the number of iterations that suffice for convergence, or by determining dynamically that subsequent iterations are unnecessary.**

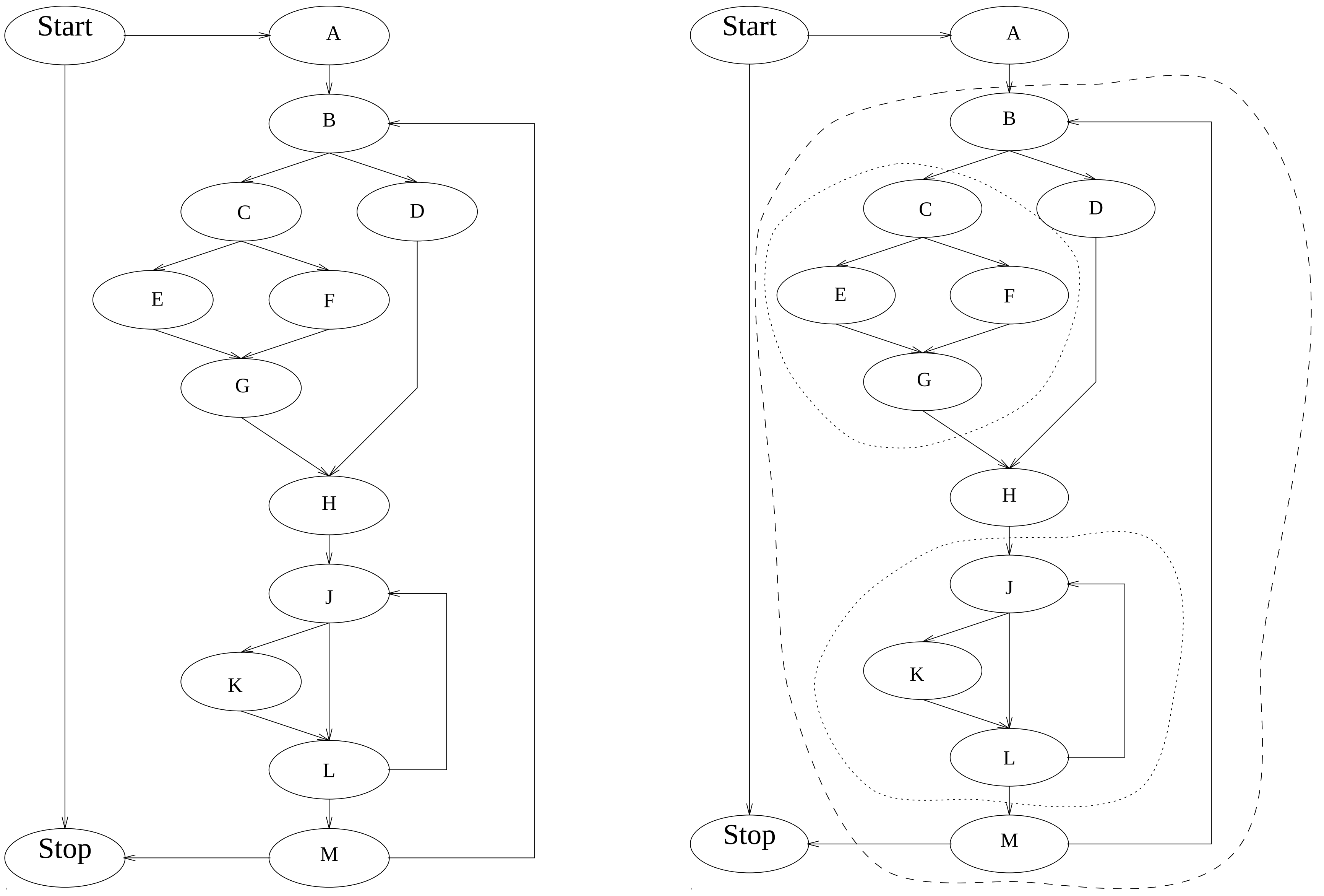
Elimination (58)

1. **The flow graph is partitioned into a set of (typically single-entry) regions R. With interval analysis (3, 13), each region is a loop, or** *interval* **(66, 63), of the flow graph.**
2. **The flow graph is** *reduced* **into an interconnection of members of R.**
3. **Each $r \in R$ is assigned a transfer function, typically computed by composition and meet within r (64).**
4. **Steps 1– 3 are repeated on the reduced flow graph, until no further reduction can occur.**
5. **Solutions are computed in the outermost region, using the summarizing transfer functions of any imbedded regions. Inner regions are then recursively solved.**

Evaluation of a data flow framework (cont'd)

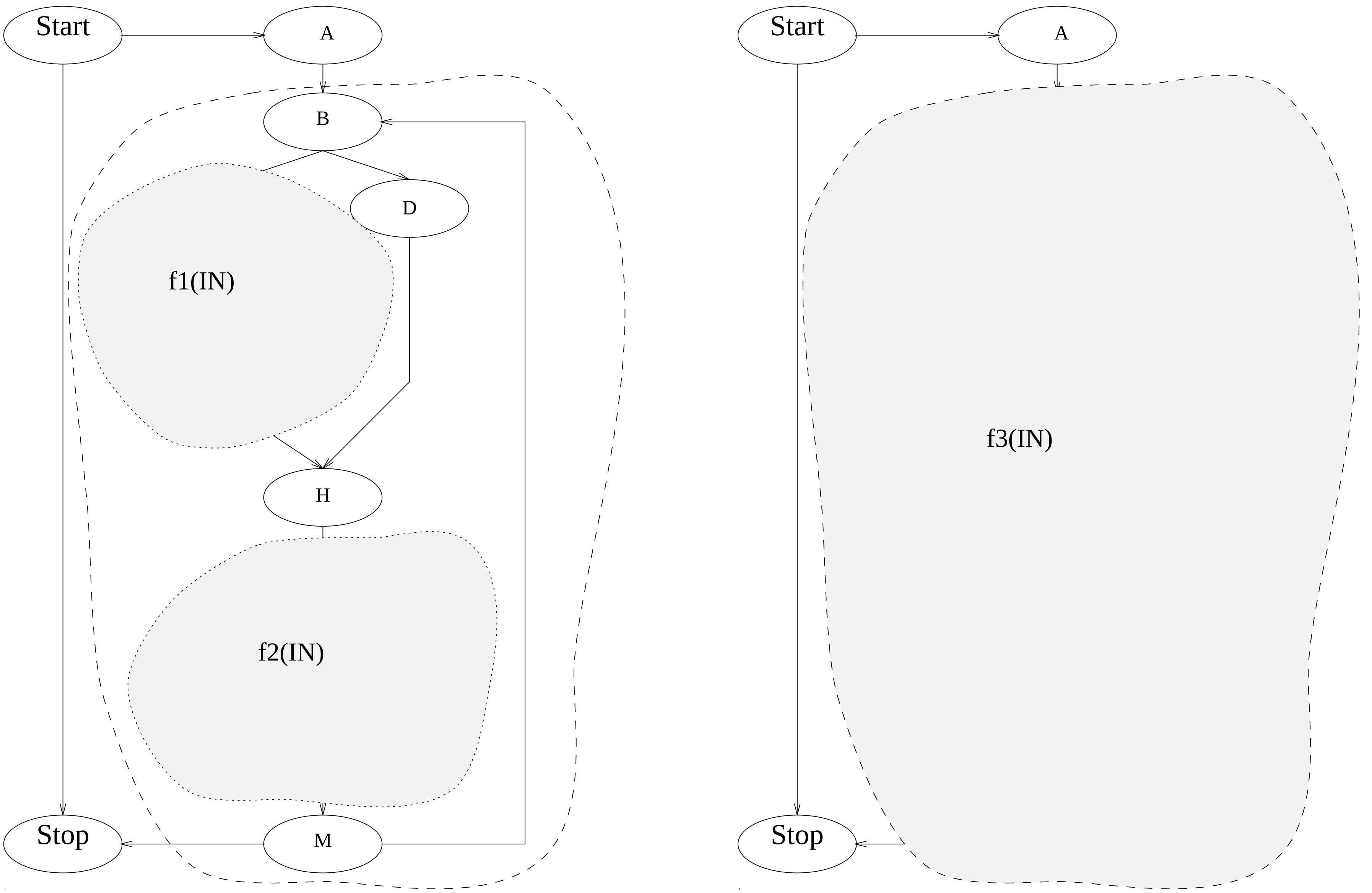
- **Iteration essentially utilizes single-node regions, and elimination may require multiple evaluations of a region to achieve fixed point. Thus, the two methods have much in common.**
- **It's possible to combine aspects from both methods to achieve a** *hybrid* **evaluation method (48).**
- **Although each method can be incrementalized, there are difficulties and tradeoffs to be considered (23, 60, 59, 12).**

Evaluation by elimination



Suppose regions are discovered as shown above, with the two “dotted” regions nested inside the “dashed” region.

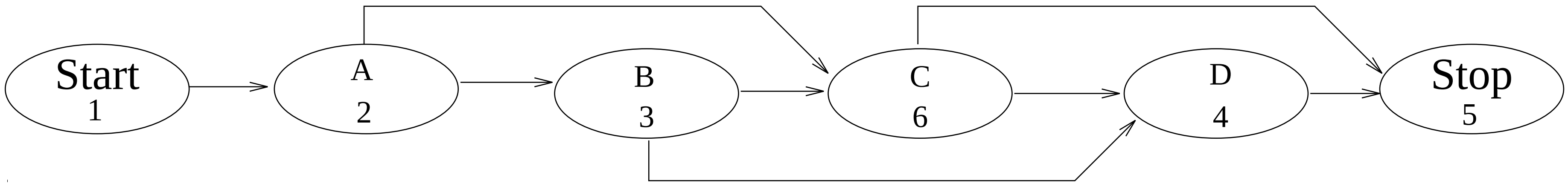
Evaluation by elimination (cont'd)



Each region is replaced by a suitable transfer function. The solution into and across the outermost region is computed first. That solution is pushed into the inner regions, so that they can be computed.

Iterative evaluation

Consider the following flow graph, annotated with a depth-first numbering of the nodes:



The transfer function at each node Y is

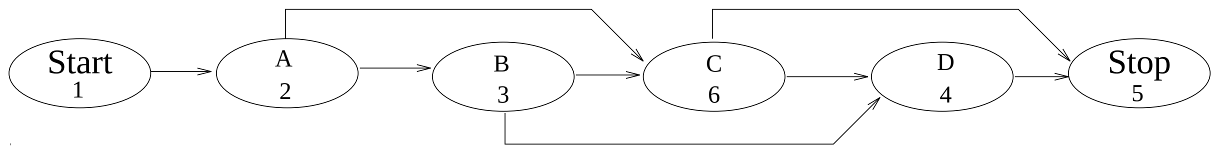
$$f_Y(IN) = \{\, Y \,\} \cup IN$$

Meet is set intersection and $\top$ is $\{\, Start, A, B, C, D, Stop \,\}$.

The following table illustrates evaluation using a poor node ordering:

Round	$Stop$	D	C	B	A	$Start$
1	$\top$	$\top$	$\top$	$\top$	$\top$	$\{\, Start \,\}$
2	$\top$	$\top$	$\top$	$\top$	$\{\, Start, A \,\}$	$\{\, Start \,\}$
3	$\top$	$\top$	$\top$	$\{\, Start, A, B \,\}$	$\{\, Start, A \,\}$	$\{\, Start \,\}$
4	$\top$	$\{\, Start, A, B, D \,\}$	$\{\, Start, A, C \,\}$	$\{\, Start, A, B \,\}$	$\{\, Start, A \,\}$	$\{\, Start \,\}$
5	$\{\, Start, A, Stop \,\}$	$\{\, Start, A, D \,\}$	$\{\, Start, A, C \,\}$	$\{\, Start, A, B \,\}$	$\{\, Start, A \,\}$	$\{\, Start \,\}$
6	.... No change					

Iteration (cont'd)



Now let's try ordering the nodes by their depth-first numbering:

Round	$Start$	A	B	D	$Stop$	C
1	$\{Start\}$	$\{Start, A\}$	$\{Start, A, B\}$	$\{Start, A, B, D\}$	$\{Start, A, B, D, Stop\}$	$\{Start, A, C\}$
2	$\{Start\}$	$\{Start, A\}$	$\{Start, A, B\}$	$\{Start, A, D\}$	$\{Start, A, Stop\}$	$\{Start, A, C\}$
3	.... No change					

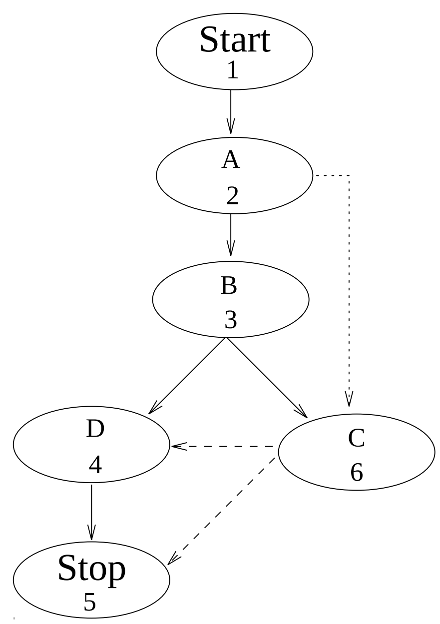
Convergence is achieved in fewer iterations because information is is propagated in the direction of the data flow problem. Can we do better?

Recall edge classification with respect to a DFST:

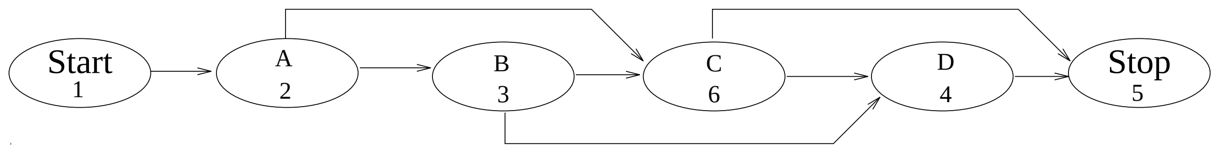
tree and chord: these are satisfied by preorder depth-first ordering;

cross: these require visiting children right-to-left, in reverse of the order in which they were visited during depth-first numbering;

back: Well, you can't win them all.



Iteration (cont'd)

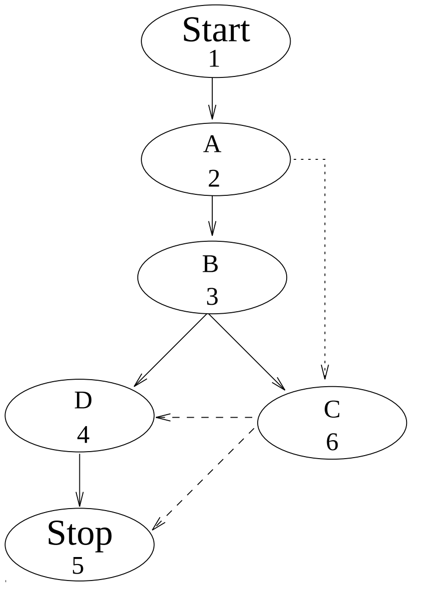


Right-to-left preorder can evaluate a DAG (directed acyclic graph) in a single pass. Now let's try this DAG-optimal node ordering:

Round	$Start$	A	B	C	D	$Stop$
1	$\{Start\}$	$\{Start, A\}$	$\{Start, A, B\}$	$\{Start, A, C\}$	$\{Start, A, D\}$	$\{Start, A, Stop\}$
2	. . . . No change					

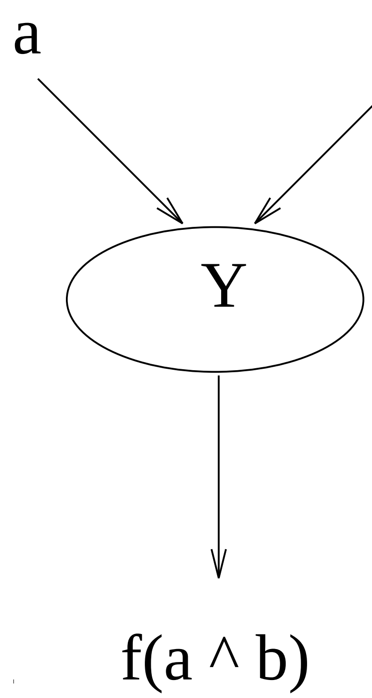
Convergence is achieved in a single iteration, because nodes are visited topologically.

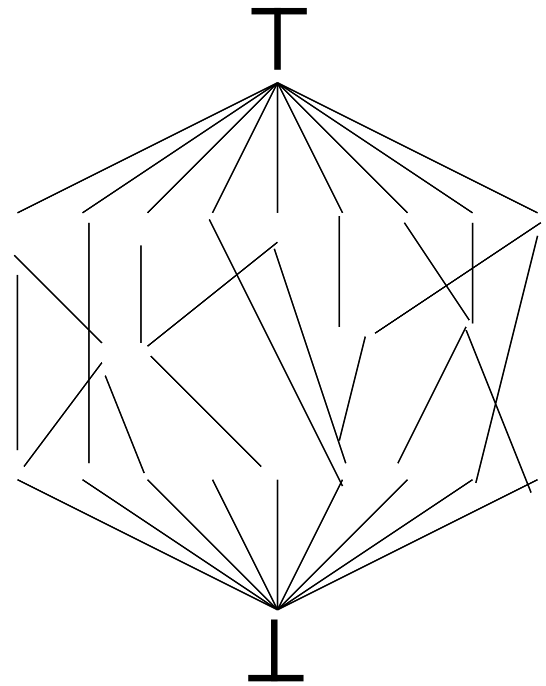
Note that right-to-left preorder is equivalent to reverse postorder (try a structurally inductive proof).



Thus, a DAG can be evaluated in a single pass. What about back edges?

Termination of iterative evaluation





- **If we require transfer functions to be *monotonic*, then**

$$c_2 \preceq c_1 \Longleftrightarrow f_Y(c_2) \preceq f_Y(c_1)$$

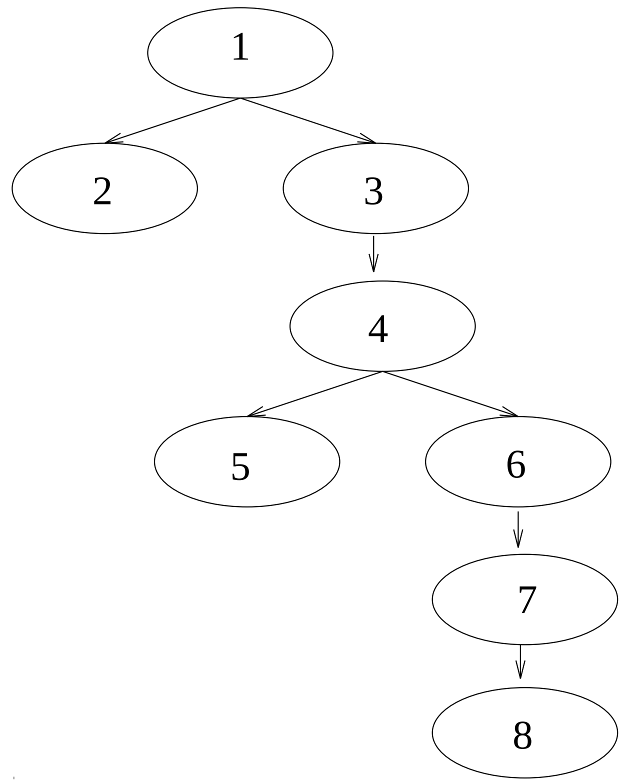
- **When meets are taken, we obtain**

$$a \wedge b \preceq a$$
$$a \wedge b \preceq b$$

We can then expect that if we monitor the solution at any given flow graph edge, we will see a nonincreasing sequence: the solution never gets "better" as iteration proceeds.

If the lattice has *finite descending-chains*, then iteration must eventually converge, and so the iterative algorithm terminates.

Rate of convergence



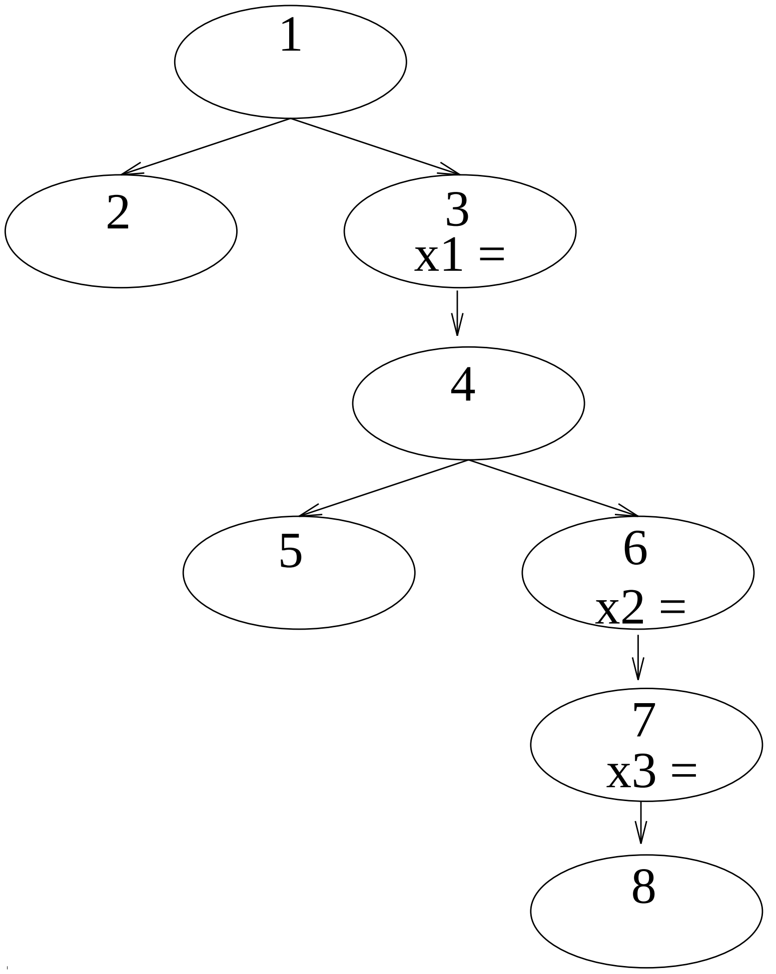
Reverse postorder for this flow graph is

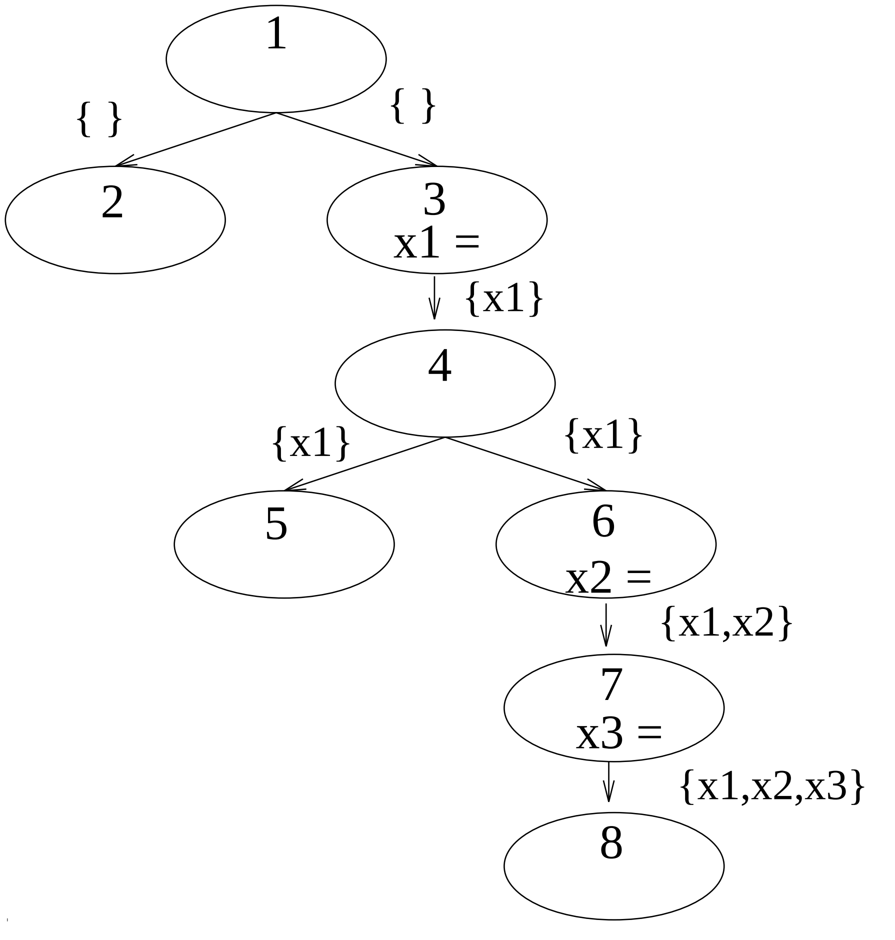
$$[1, 3, 4, 6, 7, 8, 5, 2]$$

We'll now look at evaluating two frameworks on this graph.

Convergence for reaching definitions

Consider the framework of reaching definitions, applied to the flow graph shown on the left, where each definition of `x` **is** ***preserving*****.**

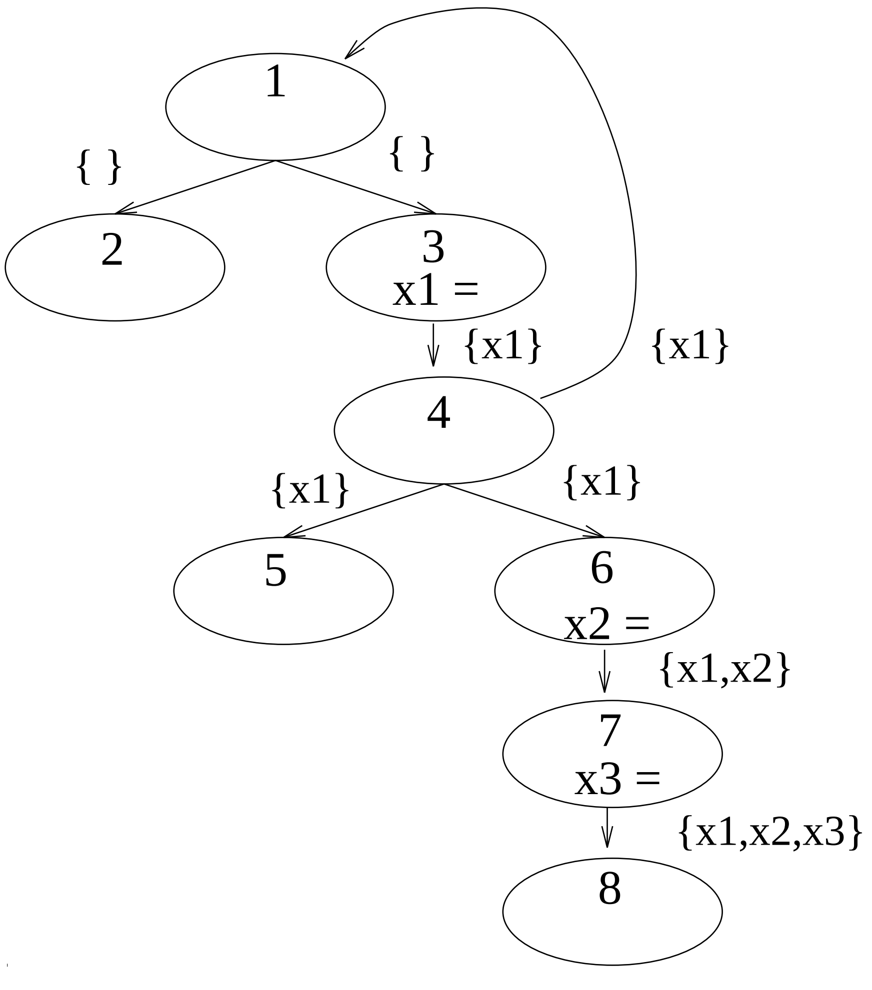




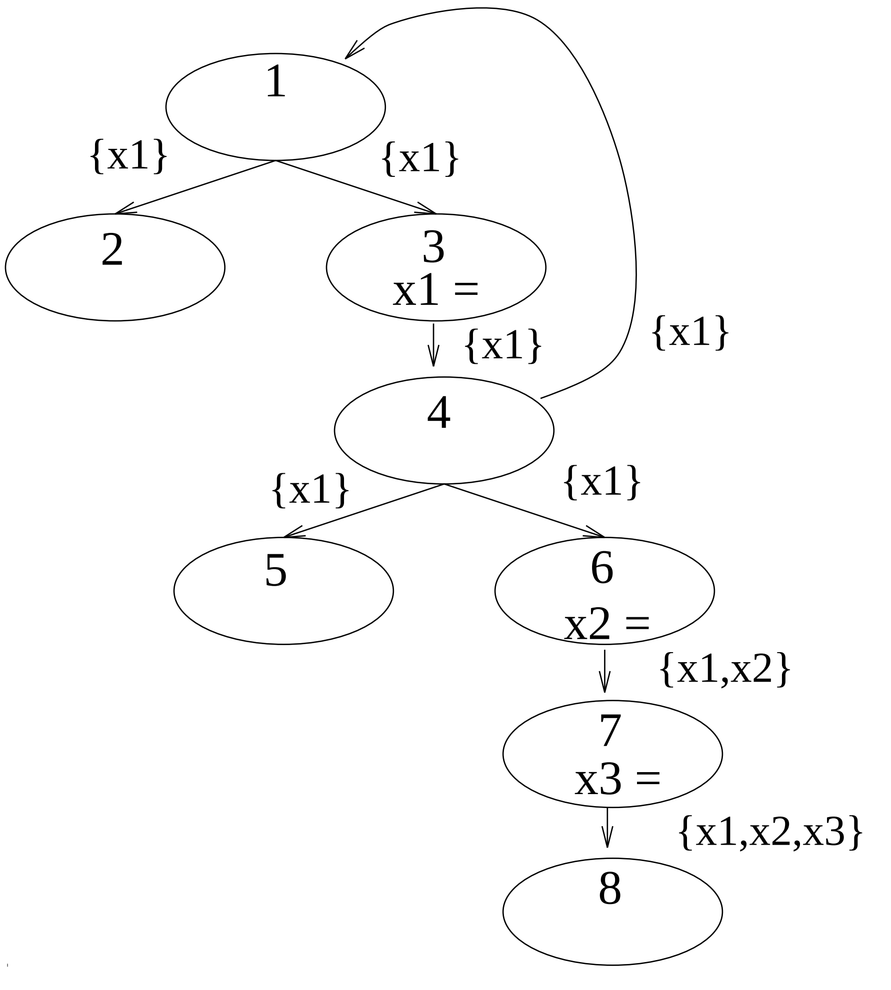
The solution stabilizes in a single pass to the solution shown on the right.

Convergence for reaching definitions (cont'd)

First pass



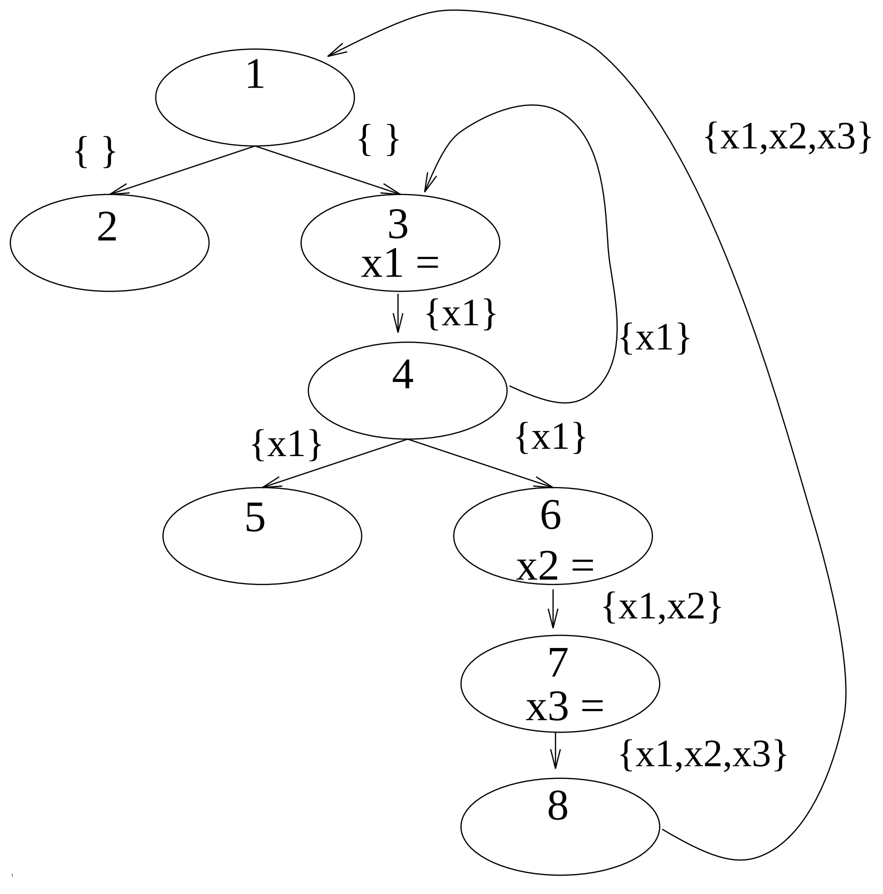
Second pass



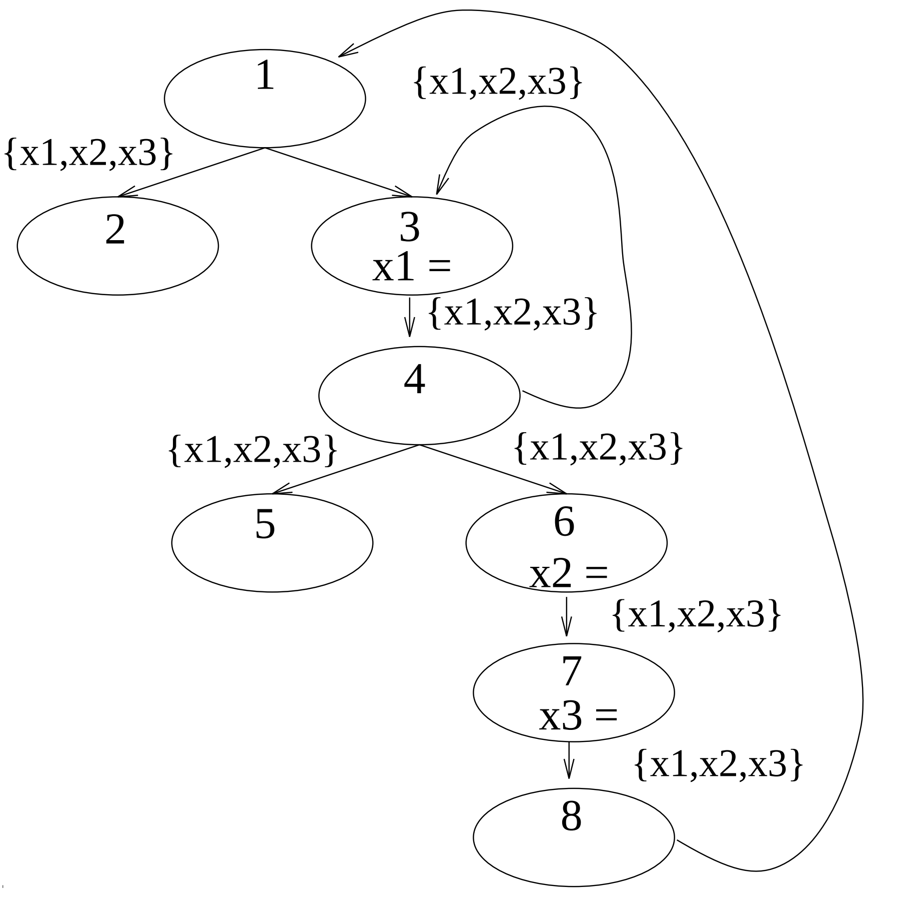
With the back edge added, two passes are required to reach convergence as shown above.

Convergence for reaching definitions (cont'd)

First pass



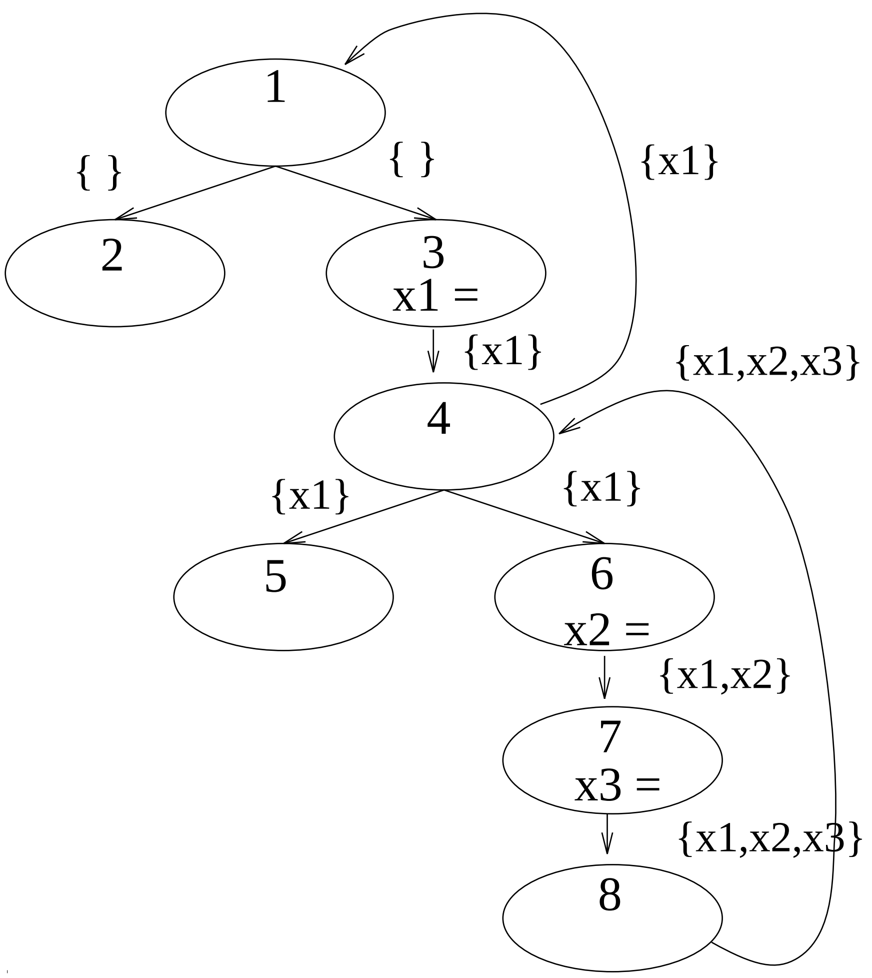
Second pass



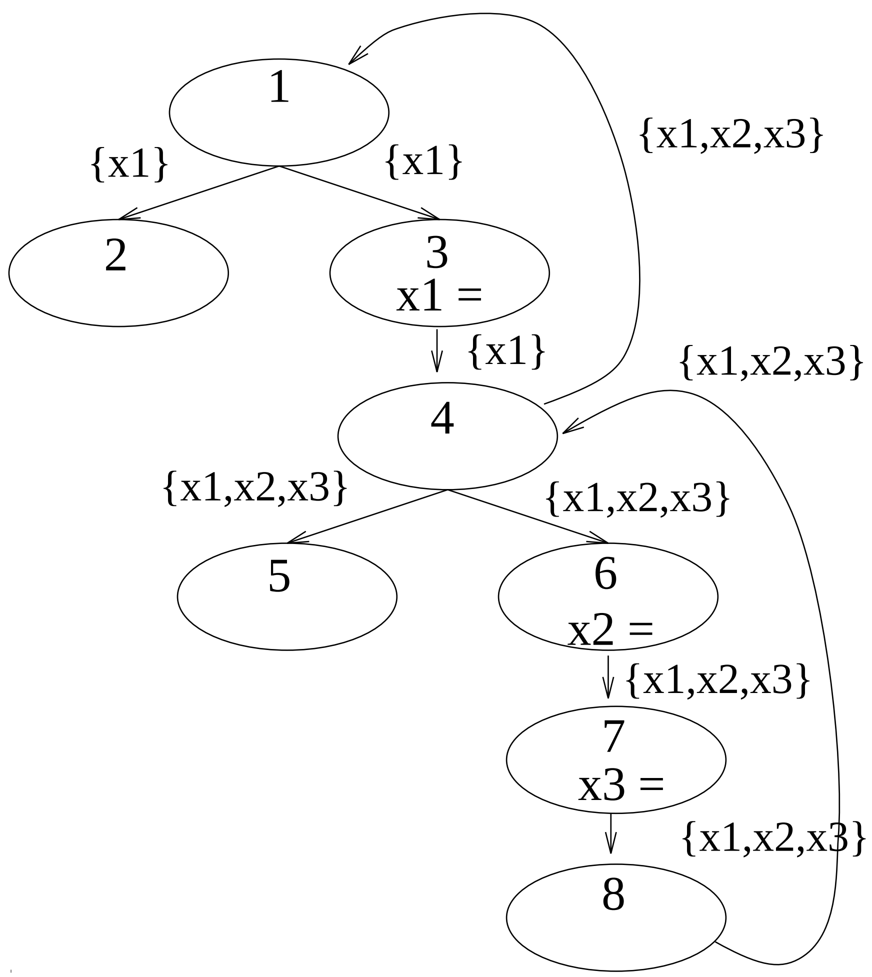
With two nested back edges, two passes still suffice for convergence.

Convergence for reaching definitions (cont'd)

First pass



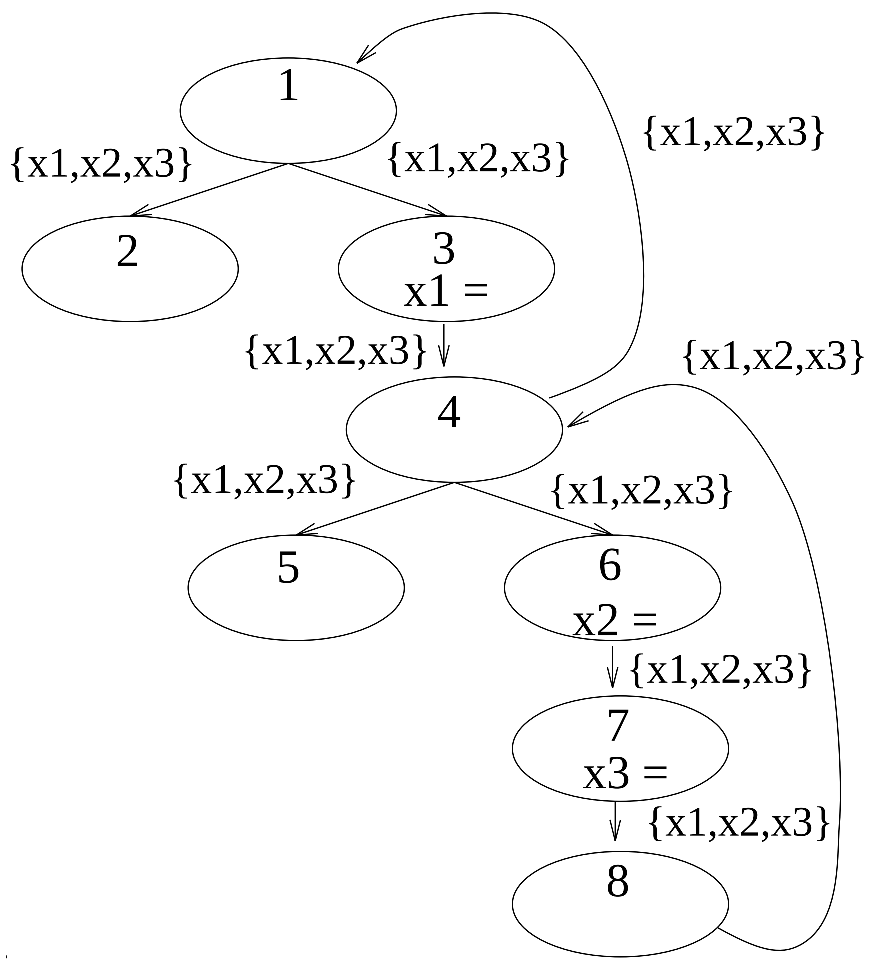
Second pass



Convergence isn't reached in the second pass, because node 4 does not have its final solution when node 1 is evaluated.

Convergence for reaching definitions (cont'd)

Third pass



Suppose information from node 6 must reach node 2. The information must first use the back edge $8 \to 4$ (after the first pass), and then use the back edge $4 \to 1$ (after the second pass). In the third pass, information from node 1 can flow to node 2.

Let w be the maximum number of consecutive back edges through which information must flow to "communicate" a solution from nodes X to Y. Then at least $w + 1$ passes are required to reach convergence, and w is called the *width* of the flow graph (33).

We'll now examine the conditions under which $w + 1$ iterations suffice for convergence.

In the third pass, node 4 has stabilized, and so all nodes receive their final solution.

Rapid data flow frameworks

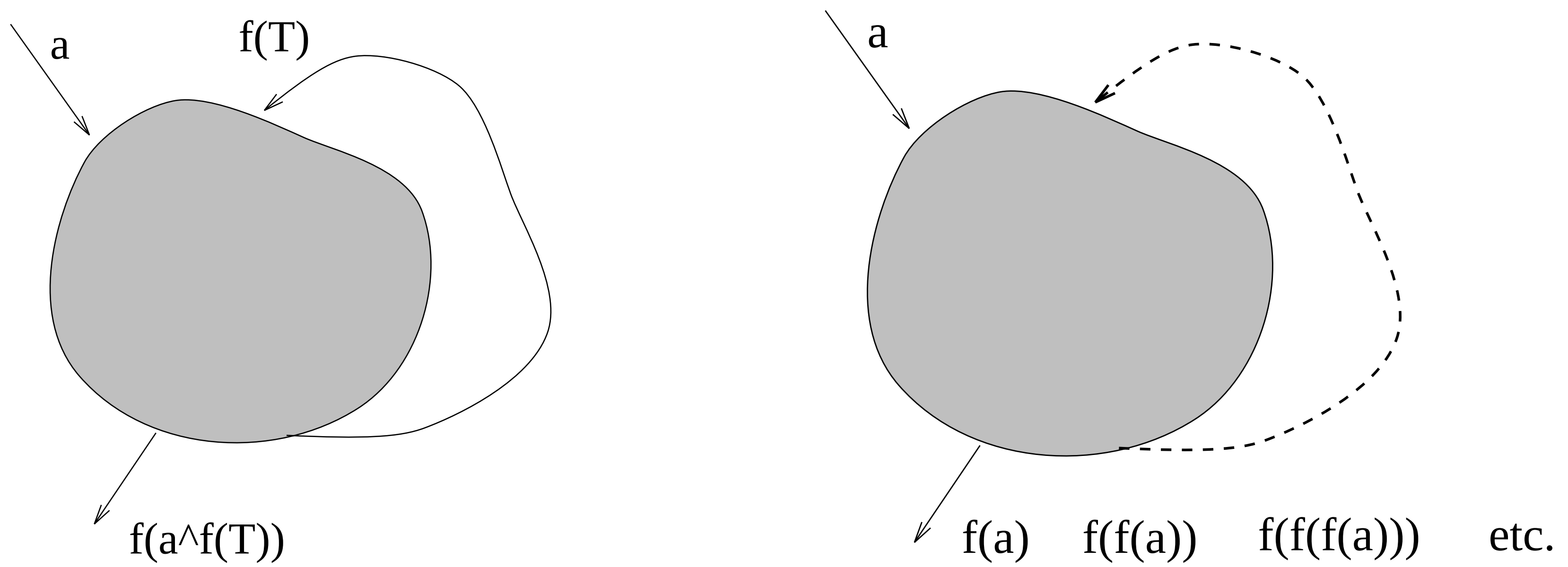
A data flow framework is *rapid* if

$$(\forall\ a \in A)(\forall\ f \in \mathcal{F})\ \ a \wedge f(\top) \preceq f(a)$$

When presented with some real input a, $a \wedge f(\top)$ takes us just as far toward convergence (down the lattice) as would application of f on the real input a. More formally, for monotone f

$$\begin{aligned} a \wedge f(\top) &\preceq f(a) \\ f(a \wedge f(\top)) &\preceq f(f(a)) \end{aligned}$$

and $f(f(a))$ represents the solution after two iterations.



With a rapid framework, one iteration gathers all the information necessary to reach convergence; the solution is not affected by subsequent iteration.

Reaching definitions is rapid

Recall the framework of reaching definitions:

$$\begin{aligned} \top &= \emptyset \\ \wedge &= \textbf{set union} \\ a \preceq b &\iff a \supseteq b \end{aligned}$$

Any transfer function in this framework can be characterized by

$$f_p(IN) = (IN - KILL_p) \cup GEN_p$$

where $KILL_p$ and GEN_p are constants for path p.

And so:

$$\begin{aligned} a \wedge f_p(\top) &= a \cup (\emptyset - KILL_p) \cup GEN_p \\ &= a \cup GEN_p \\ &\preceq (a \cup GEN_p) - (KILL_p - (a \cup GEN_p)) \\ &\preceq (a - KILL_p) \cup GEN_p \\ &\preceq f(a) \end{aligned}$$

□

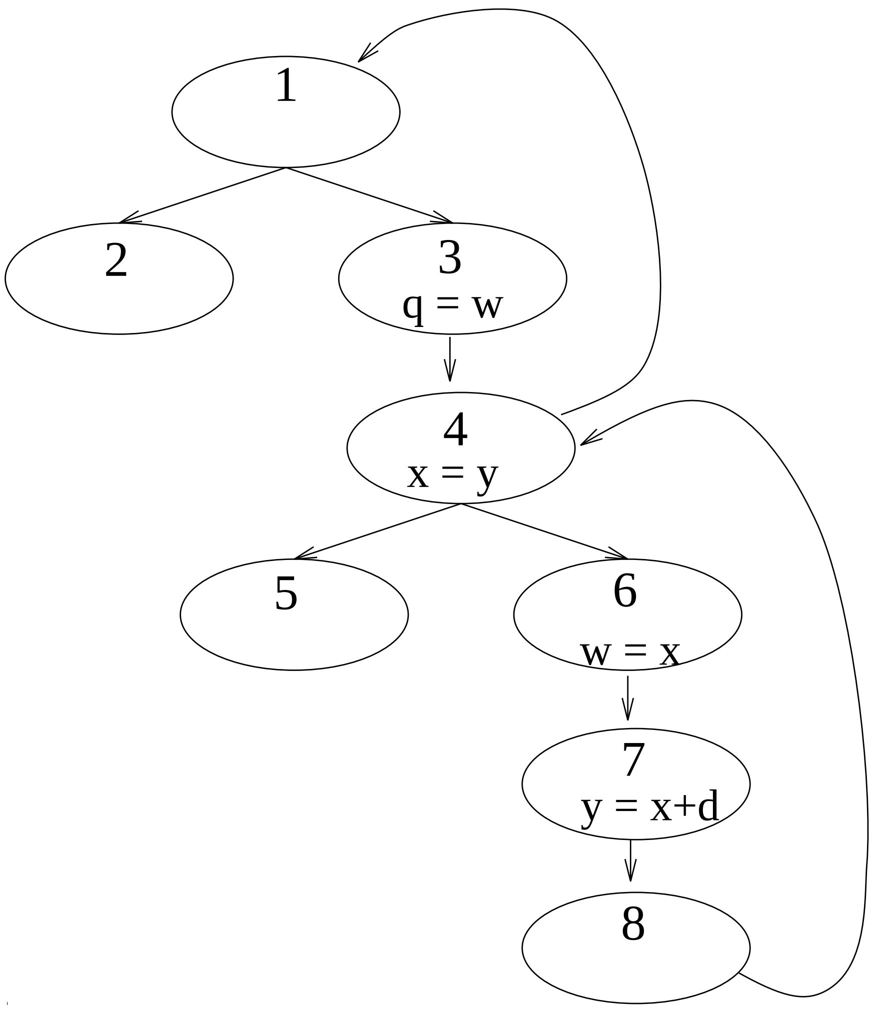
A nonrapid problem

Consider the problem of determining the number of *bits* that should be associated with a variable. In the program shown to the right, assume that node 1 forces the solution:

Variable	# bits
x,y,q,w	4
d	7

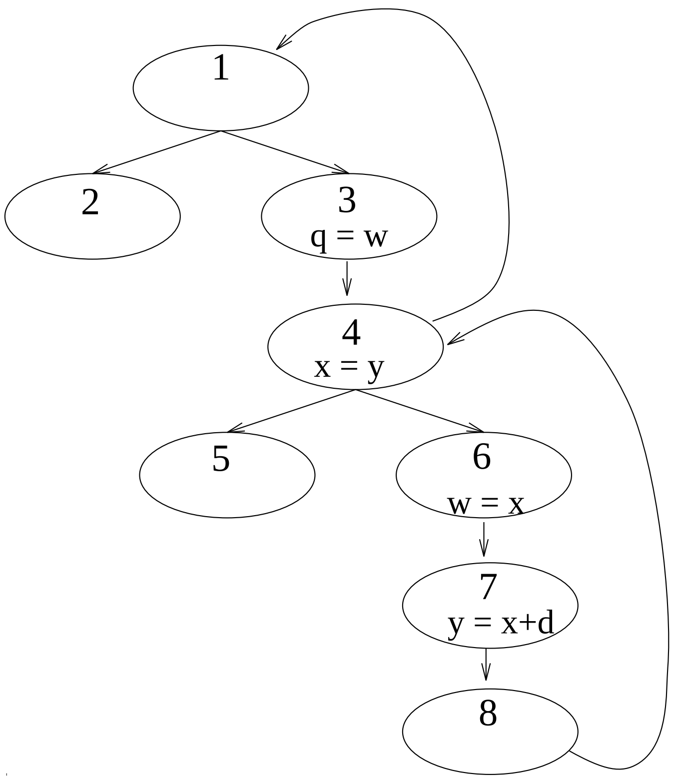
Let $NB(v)$ denote the number of bits associated with v; let solution a be denoted as the 5-tuple $[NB(x), NB(y), NB(q), NB(w), NB(d)]$; let the value for x in solution a be denoted a_x. For this framework

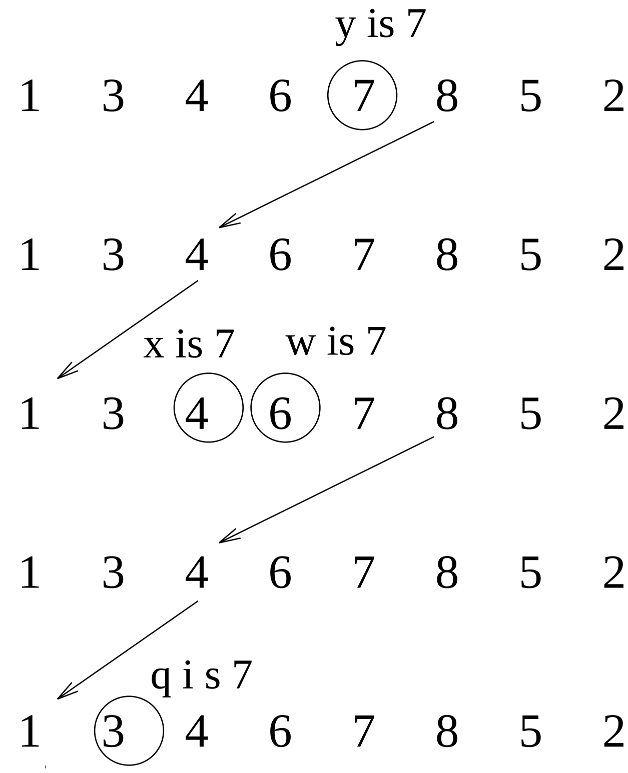
$$\begin{aligned} \top &= [0,0,0,0,0] \\ a \wedge b &= [max(a_x, b_x), \ldots, max(a_d, b_d)] \\ a \preceq b &\iff (a \wedge b = a) \textbf{ or } (a \wedge b = b) \end{aligned}$$



For transfer functions, let the number of bits of an assigned variable be the maximum number of bits associated with any variable used in the assignment.

Convergence takes 5 passes





Consider a two-variable program that assigns $y = d$, so that

$$f(IN) = [\,IN_d, IN_d\,]$$

Taking f as above and $a = [\,2,7\,]$, we obtain

$$\begin{aligned} a \wedge f(\top) &= [\,2,7\,] \wedge [\,0,0\,] \\ &= [\,2,7\,] \\ &\not\preceq [\,7,7\,] \\ &\not\preceq f(a) \end{aligned}$$

Distributive frameworks

Monotonicity and finite lattice depth guarantee termination, but how good is the answer obtained at convergence?

The meet properties give us

$$\begin{aligned} a \wedge b &\preceq a \\ a \wedge b &\preceq b \end{aligned}$$

and monotonicity gives us

$$\begin{aligned} f(a \wedge b) &\preceq f(a) \\ f(a \wedge b) &\preceq f(b) \end{aligned}$$

and so we obtain

$$\begin{aligned} f(a \wedge b) \wedge f(a \wedge b) &\preceq f(a) \wedge f(b) \\ f(a \wedge b) &\preceq f(a) \wedge f(b) \end{aligned}$$

When

$$(\forall\ a, b \in A)(\forall\ f \in \mathcal{F})\ \ f(a \wedge b) = f(a) \wedge f(b)$$

then we call the data flow framework *distributive*. Evaluation of such a framework produces the best possible (static) solution.

Examples

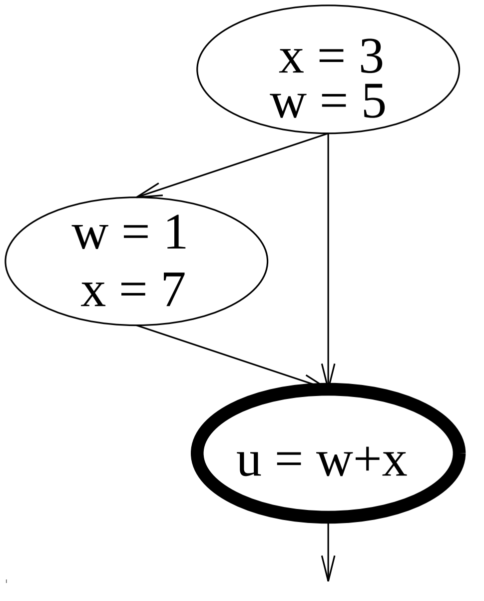
Each of the bit-vectoring data flow frameworks is distributive, since

$$\begin{aligned} f(a \wedge b) &= ((a \wedge b) - KILL) \cup GEN \\ &= ((a - KILL) \wedge (b - KILL)) \cup GEN \\ &= ((a - KILL) \cup GEN) \wedge ((b - KILL) \cup GEN) \\ &= f(a) \wedge f(b) \end{aligned}$$

However, constant propagation is not distributive:

Consider the solution for u after the highlighted node shown on the right. If the nodes' transfer function is applied separately to each in-edge, then we obtain $w+x=8$ on each path, and so the meet determines that $u=8$.

On the other hand, if we take the meet of the solutions entering the node, we find $w=\perp$ and $x=\perp$ so that $u=w+x$ is also $\perp$.



Thus, our iterative technique does not compute the meet-over-all-paths (MOP) solution. For a DAG, we could compute MOP in nondistributive frameworks, but such computation could take exponential time in the size of the flow graph; for acyclic graphs, such computation may not terminate.

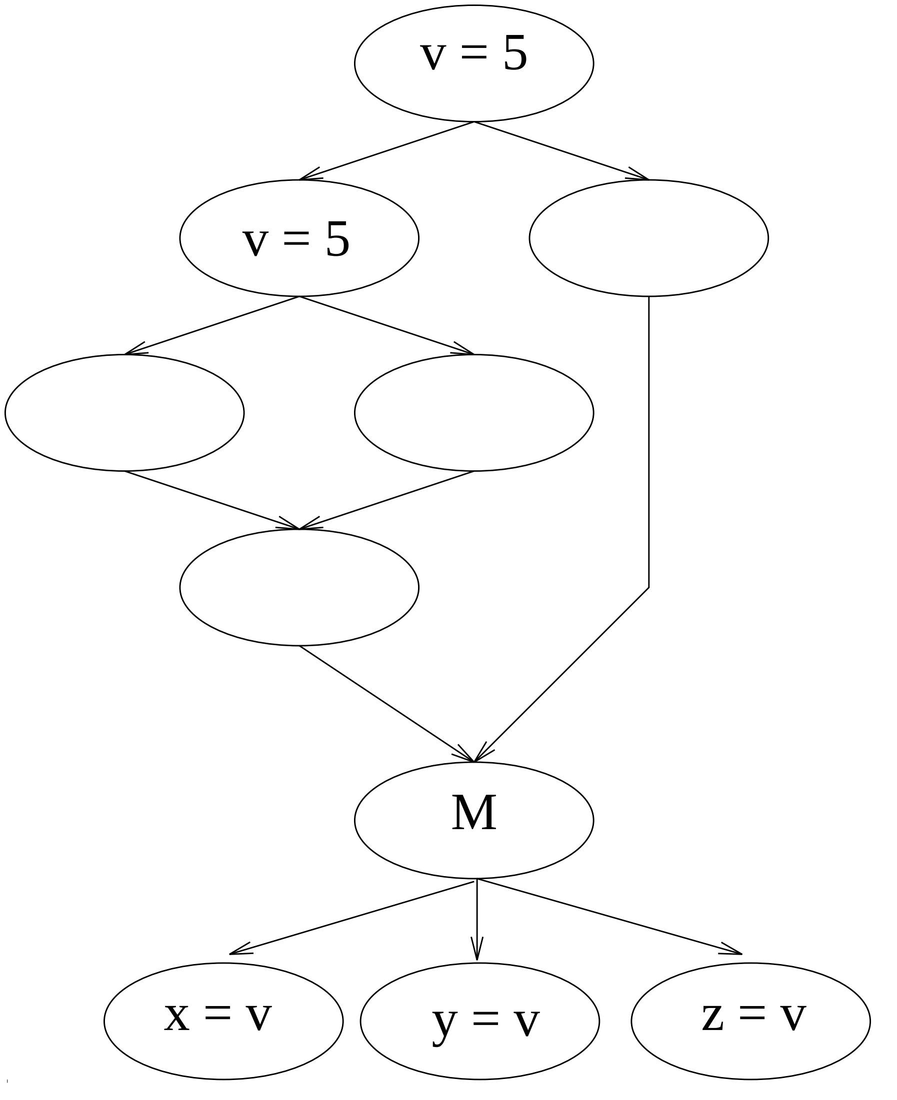
Data flow evaluation graphs–traditional

Consider the data flow problem of constant propagation, applied to the flow graph shown to the right.

As discussed previously, solutions are traditionally propagated through the flow graph:

- **At nodes with multiple in-edges, a meet is performed of the input solutions;**
- **Transfer functions are evaluated across nodes;**
- **The output solution is made available.**

Solutions are iteratively computed until convergence is achieved.



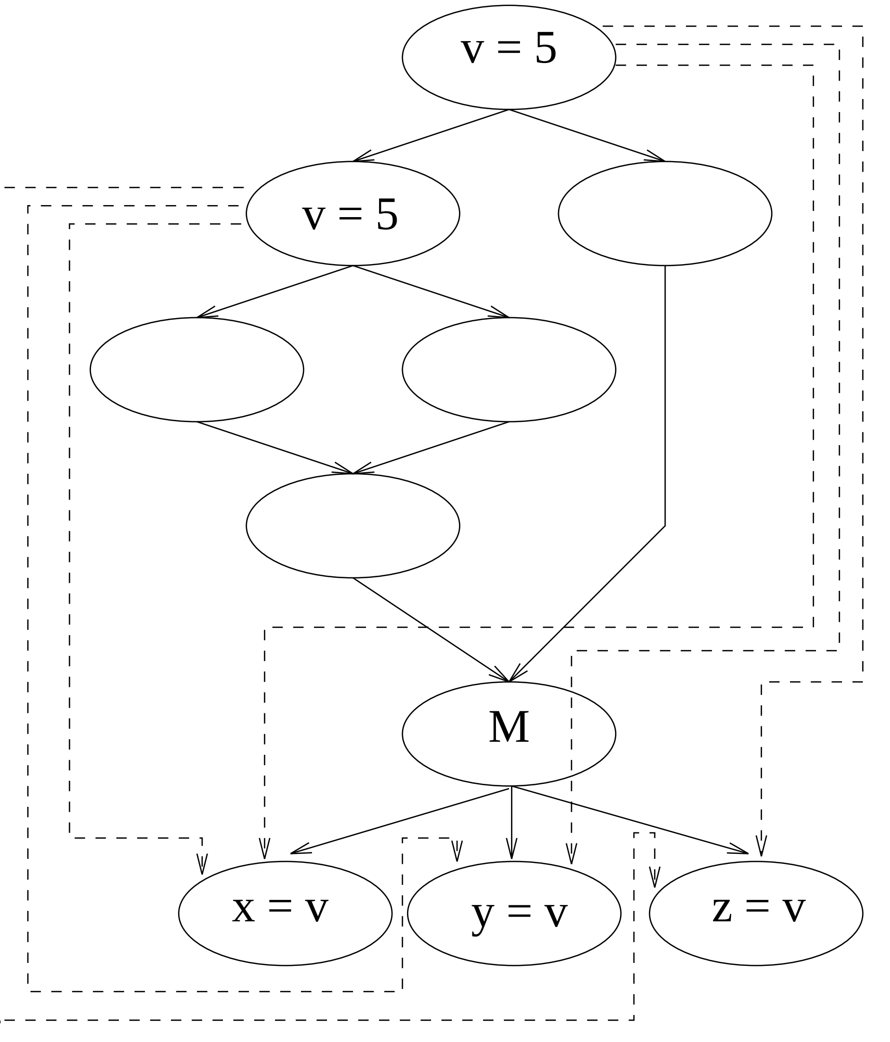
Notice how solutions for v are propagated through portions of the flow graph that neither affect nor are concerned with solutions for v.

Data flow evaluation graphs–direct connection

The dashed lines in the flow graph shown to the right represent *direct connections* **between nodes of the flow graph.**

- **Data flow information is propagated along the direct connections rather than along flow graph edges;**
- **Meets occur where multiple direct connections are incident on a node;**
- **Iteration occurs until convergence is achieved.**

These types of connections are often called *def-use chains*. **Some data flow problems (e.g., constant propagation) are often specified using def-use chains, which assumes that reaching definitions has already been solved.**

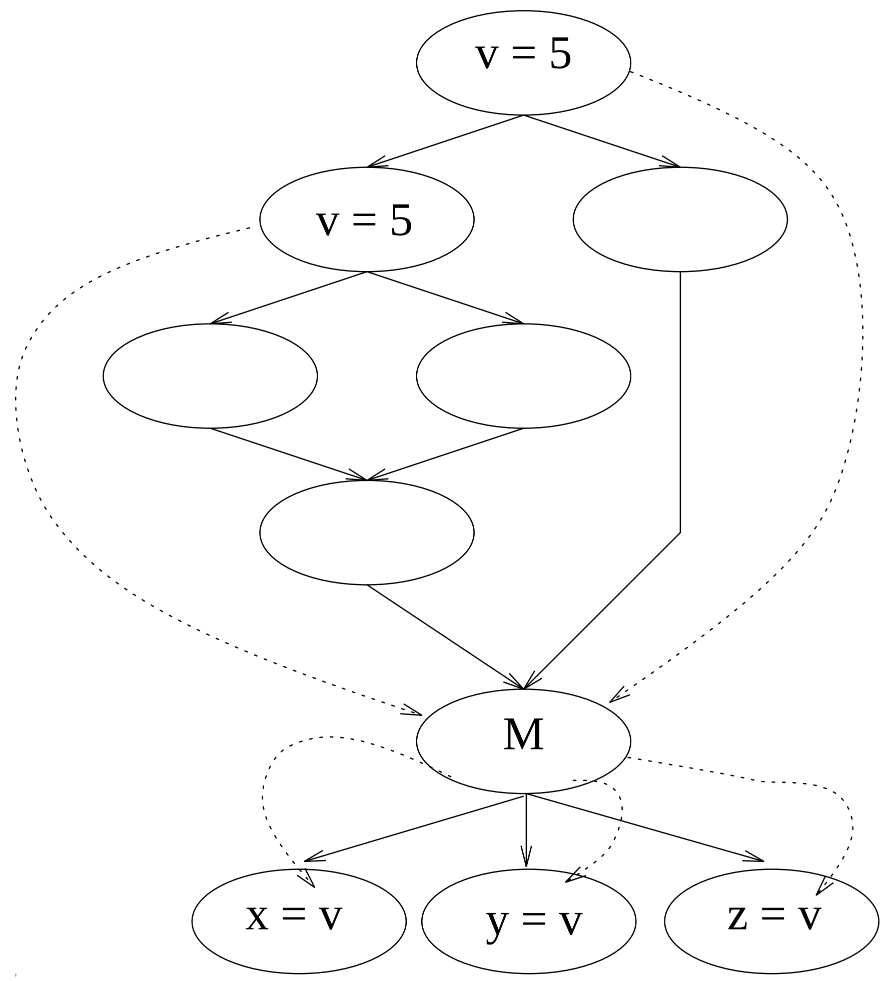


Although uninteresting portions of the graph are avoided during evaluation, notice how the meet of the same information is computed three times. The traditional approach computed that meet only once.

Sparse data flow evaluation graphs

Here we see the best of both worlds:

- **Information does not propagate through uninteresting regions of the flow graph;**
- **Only one meet operation is performed;**
- **Information flows directly to meet and useful sites.**



We'll now study an algorithm for constructing sparse evaluation graphs (21). That algorithm, applied to a special framework, will construct Static Single Assignment (SSA) form (28).

Sparse evaluation graphs (cont'd)

Let's define two special kinds of transfer functions:

identity: A node Y has identity transference, denoted

$$f_Y = \iota$$

if $\forall\ a \in A,\ f_Y(a) = a$. Generally, such nodes will not be useful in evaluating a data flow framework unless information is gainfully combined there.

constant: A node Y has constant transference, denoted

$$f(IN) = K_Y$$

if $\forall\ a \in A,\ f_Y(a) = K_Y$. Because these nodes produce the same solution for any input, we can improve on the efficiency of propagating information through such nodes.

Steps for constructing a sparse evaluation graph $\mathcal{G}_{sg}$ (21):

1. **Let $\mathcal{N}_{sg}$ be the set of nodes with nonidentity transference. The node $Start$ is always included in this set.**
2. **Add to $\mathcal{N}_{sg}$ those nodes where meets occur. These so-called ϕ-nodes are found by iterative closure using dominance frontiers.**
3. **Construct $\mathcal{E}_{sg}$ while traversing the original flow graph's dominator tree.**

Although an almost-linear algorithm has been recently developed for placing ϕ-nodes (27), we'll examine the method more commonly in use (28).

The resulting sparse graph is ready for evaluation using any method (e.g., iteration). Also produced is a function that maps edges in the original flow graph to nodes in the sparse graph, so that a solution is available for every original flow graph edge after the sparse graph has been evaluated.

Sparse evaluation graph construction

$\mathcal{N}_{sg} \leftarrow \{\, start \,\} \cup \{\, Y \mid f_Y \neq \iota \,\}$ $\Leftarrow \boxed{1}$

$MeetNodes \leftarrow \cup_{X \in \mathcal{N}_{sg}} DF^+(X)$ $\Leftarrow \boxed{2}$

$\mathcal{N}_{sg} \leftarrow \mathcal{N}_{sg} \cup MeetNodes$ $\Leftarrow \boxed{3}$

$\forall\, Y \in \mathcal{N}_{sg},\ IN_Y \leftarrow \top$ $\Leftarrow \boxed{4}$

$Stack \leftarrow$ `Empty` $\Leftarrow \boxed{5}$

call $Search(start)$

Procedure $Link(Z)$

  if $(Z \in \mathcal{N}_{sg} - \{\, start \,\})$ **then** $\Leftarrow \boxed{6}$

    if ($f_Z \neq K_Z$) **then** $\Leftarrow \boxed{7}$

      if ($f_{TOS} = K_{TOS}$) **then** $\Leftarrow \boxed{8}$

        $IN_Z \leftarrow IN_Z \wedge K_{TOS}$

      else

        $\mathcal{E}_{sg} \leftarrow \mathcal{E}_{sg} \cup (TOS, Z)$ $\Leftarrow \boxed{9}$

      fi

    fi

  fi

end

- **Step $\boxed{1}$ initializes $\mathcal{N}_{sg}$ as described previously.**
- **Step $\boxed{2}$ actually represents the iterative dominance frontier closure of nodes in $\mathcal{N}_{sg}$.**
- **The set $MeetNodes$ can be maintained as an attribute of elements in $\mathcal{N}_{sg}$, but for clarity the set is shown explicitly.**
- **The sparse nodes are complete at step $\boxed{3}$.**
- **Step $\boxed{5}$ initializes the auxiliary stack, onto which $Start$ will be pushed in the first call to $Search$.**
- **For now, ignore step $\boxed{4}$ and pretend that $Link(Z)$ is just steps $\boxed{6}$ and $\boxed{9}$.**

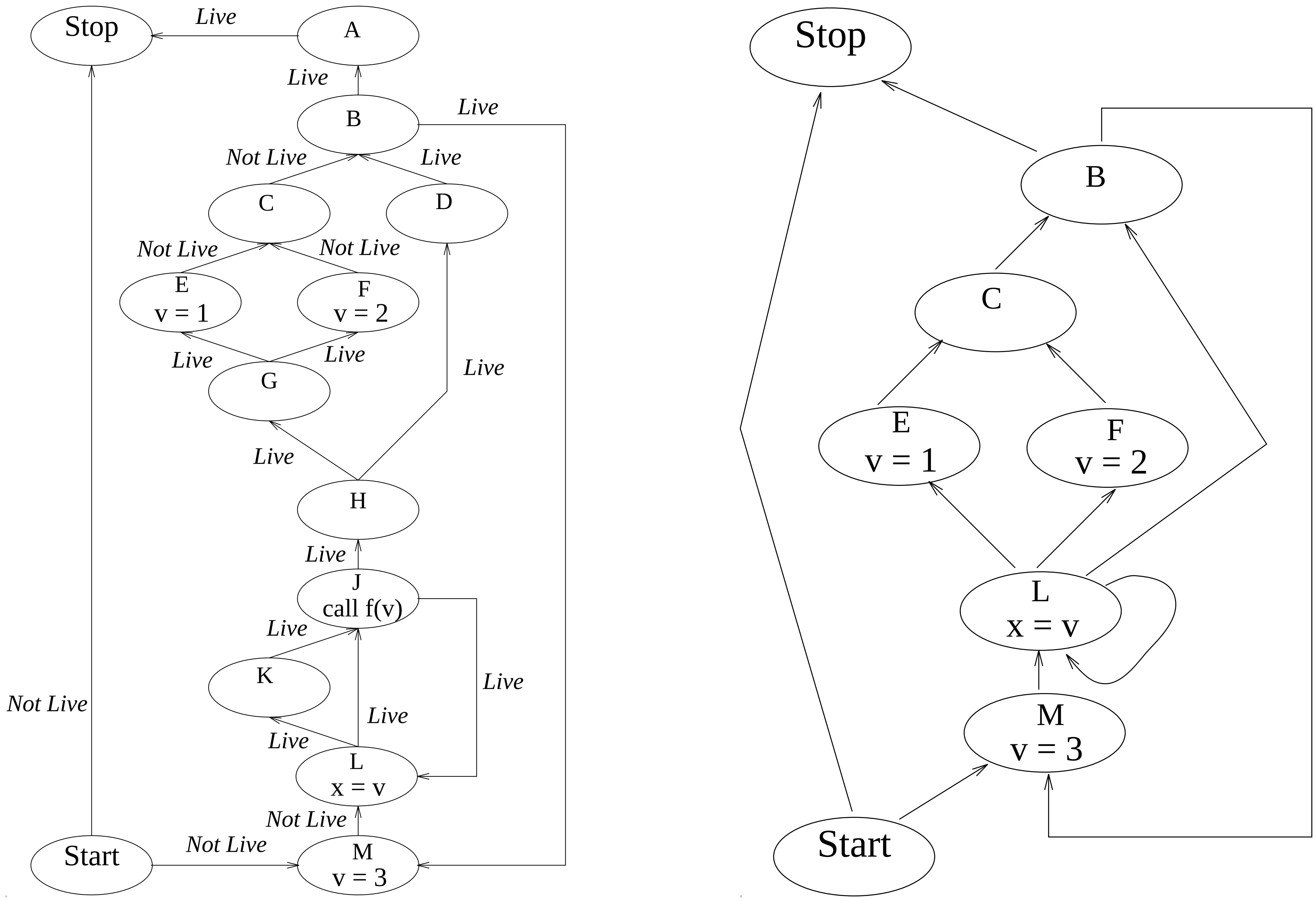
Sparse evaluation graph construction (cont'd)

- **If Y is not a meet node, then its input arrives from whatever is currently on top of stack. Step $\boxed{10}$ attempts to install the appropriate edge.**
- **Meet nodes receive their inputs as directed by step $\boxed{13}$.**
- **If Y is a sparse graph node, then it becomes the relevant solution node at step $\boxed{11}$.**
- **Edges are mapped by step $\boxed{12}$.**
- **The construction proceeds recursively down the dominator tree. Upon leaving node Y, $Search()$ pops Y if it had been pushed earlier, exposing the node stacked below Y as the relevant sparse graph node.**

Procedure $Search(Y)$
    if ($Y \notin MeetNodes$) **then** $\Leftarrow \boxed{10}$
        call $Link(Y)$
    fi
    if $(Y \in \mathcal{N}_{sg})$ **then** $\Leftarrow \boxed{11}$
        call $push(Y)$
    fi
    foreach $(Z \in Succ(Y))$ **do**
        $M(Y, Z) \leftarrow TOS$ $\Leftarrow \boxed{12}$
        if ($Z \in MeetNodes$) **then** $\Leftarrow \boxed{13}$
            call $Link(Z)$
        fi
    od
    foreach (**child** C **of** Y **in** DT) **do**
        call $Search(C)$
    od
    if $(Y \in \mathcal{N}_{sg})$ **then**
        call $pop(Y)$
    fi

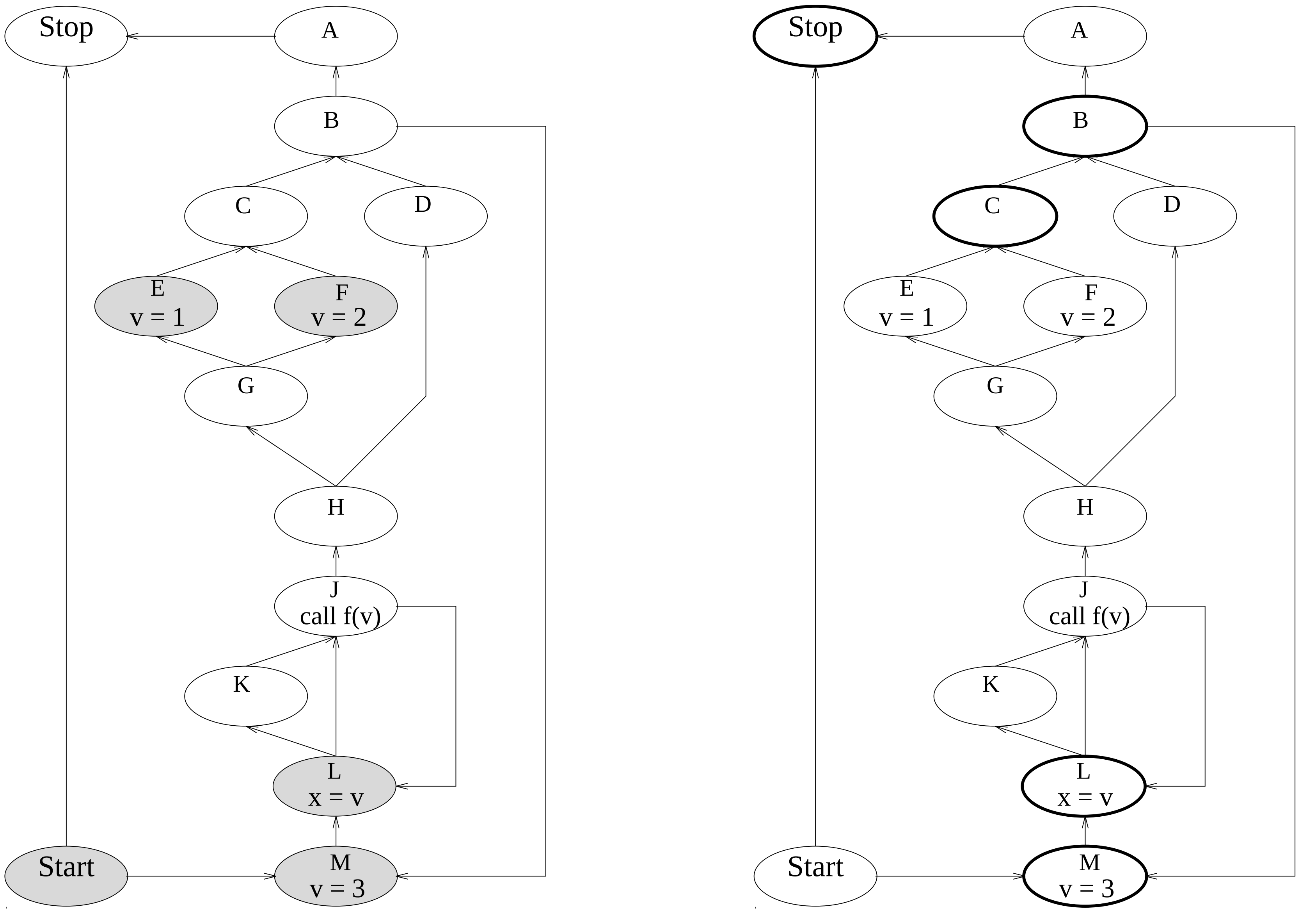
end

Sparse evaluation graph for live variables



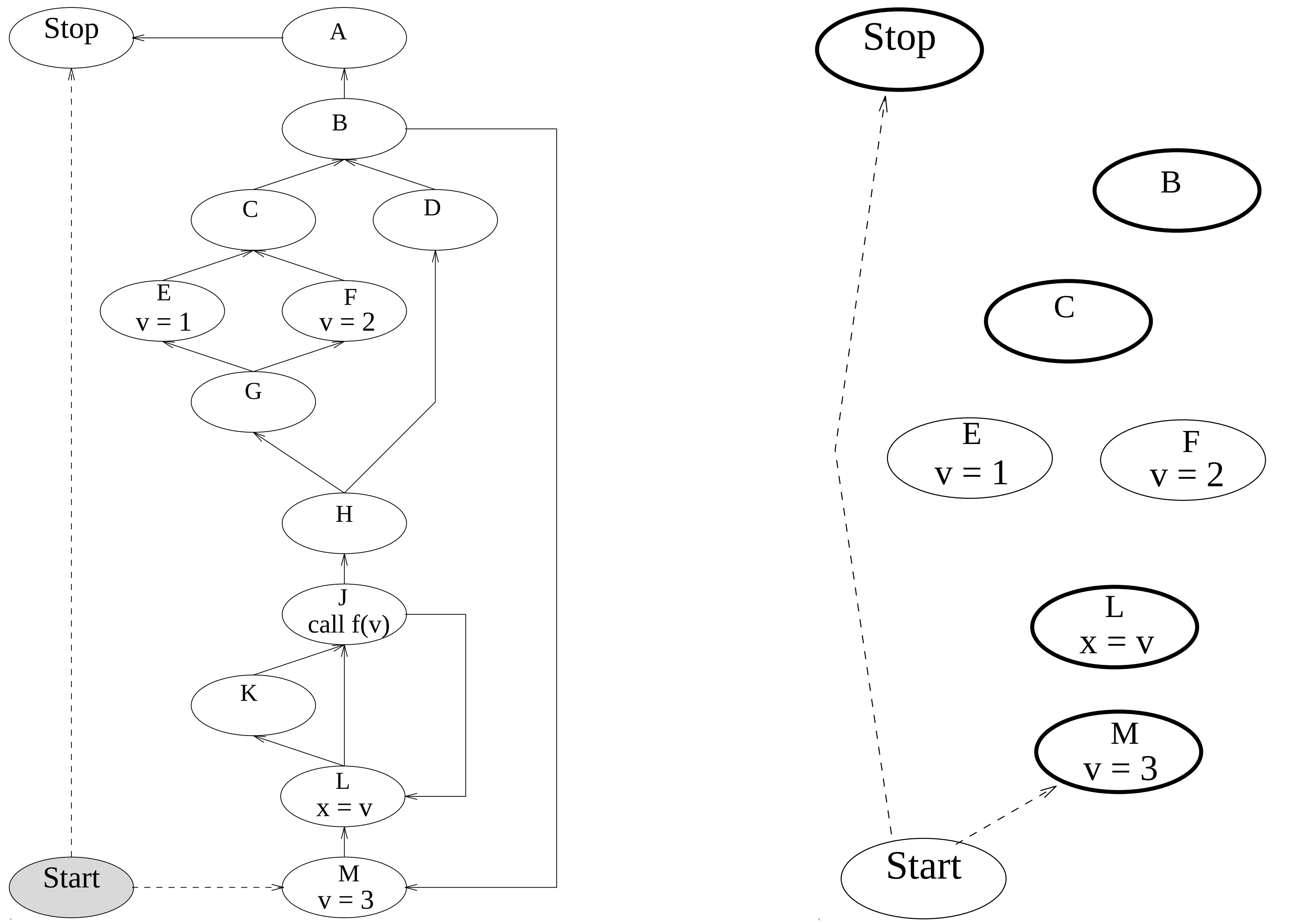
Even though the graph shown on the right is quite sparse, we'll see how to eliminate most of its edges, obtaining an acyclic evaluation graph.

Nodes in the sparse graph



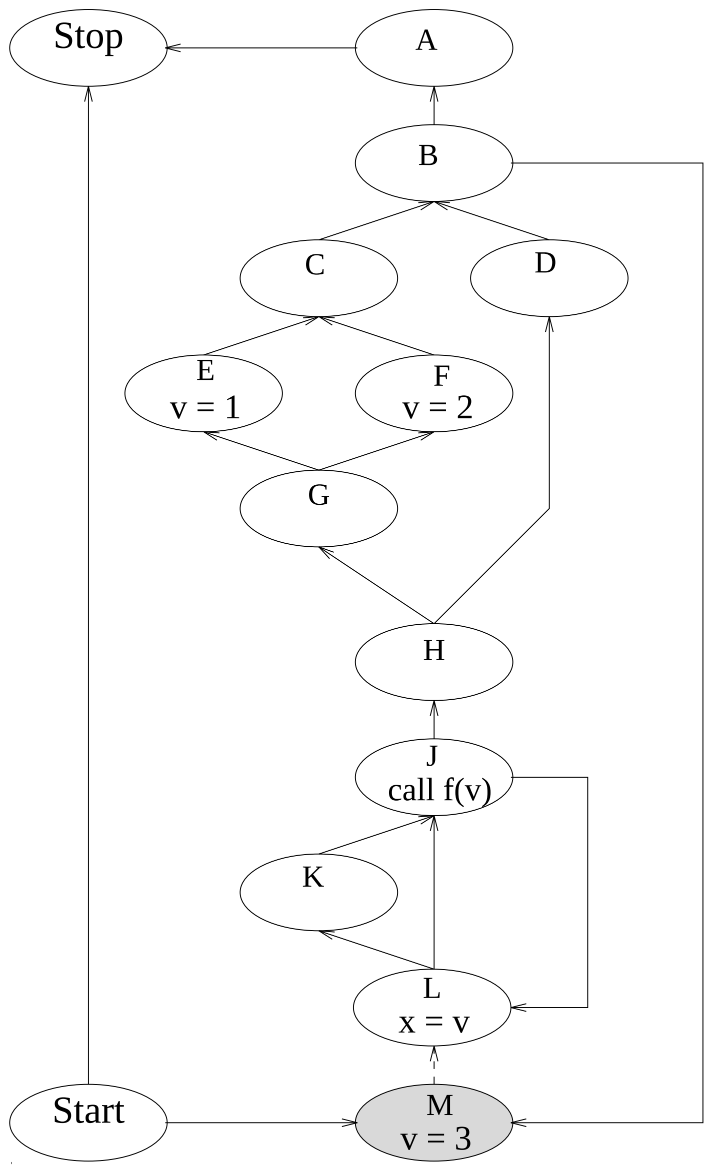
Shaded nodes in the graph on the left are nonidentity transference nodes. Highlighted nodes in the graph to the right are meet nodes: $C \in DF(E)$, $B \in DF(C)$, $M \in DF(B)$, $Stop \in DF(B)$, $L \in DF(L)$.

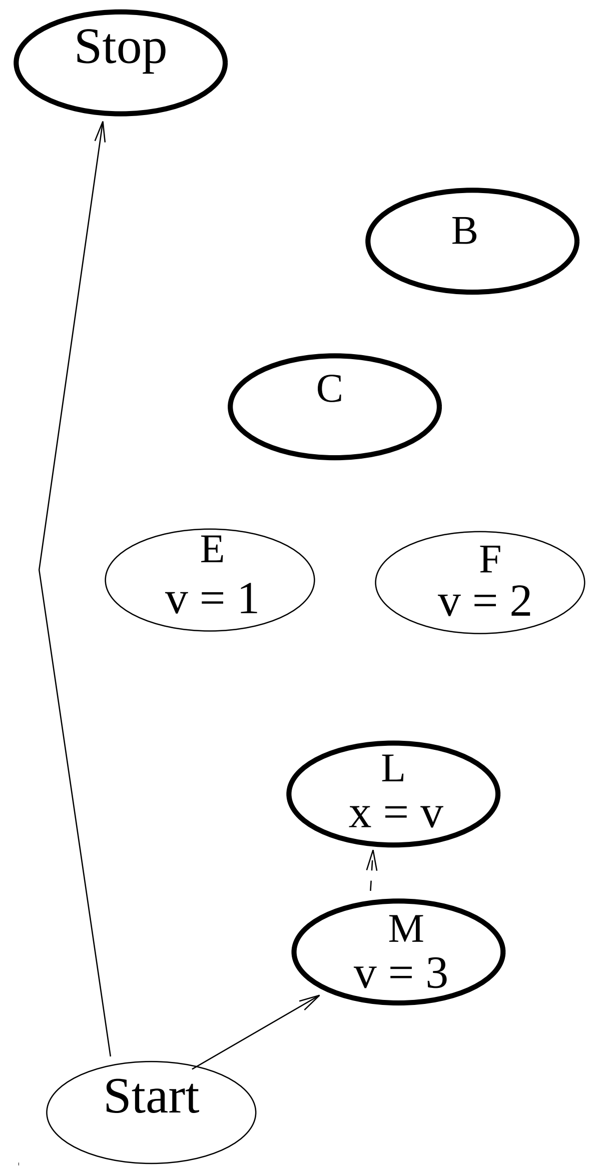
Edges in the sparse graph $\boxed{1}$



In this example, the node currently on top of stack will be shaded; dashed edges are mapped in the flow graph (left) or newly installed in the sparse graph (right).

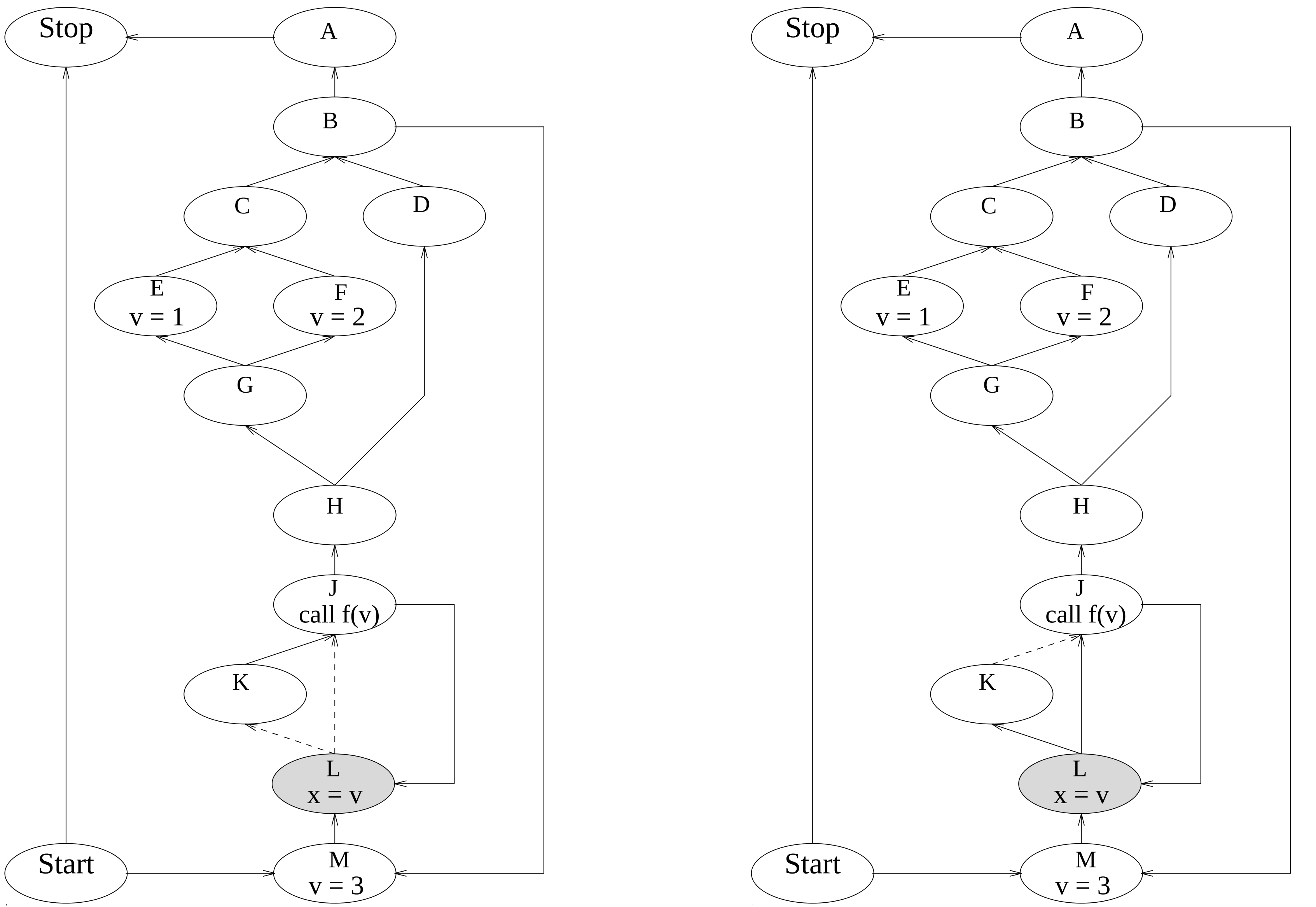
Edges in the sparse graph $\boxed{2}$





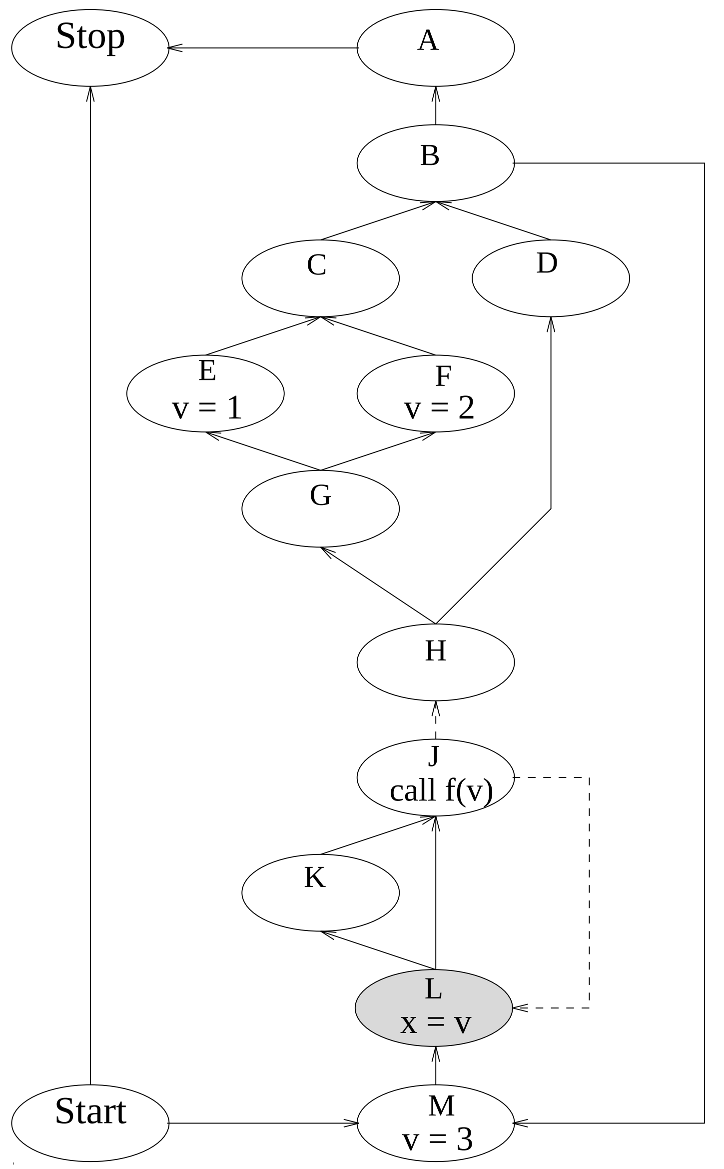
Node L receives one of its meet edges from M.

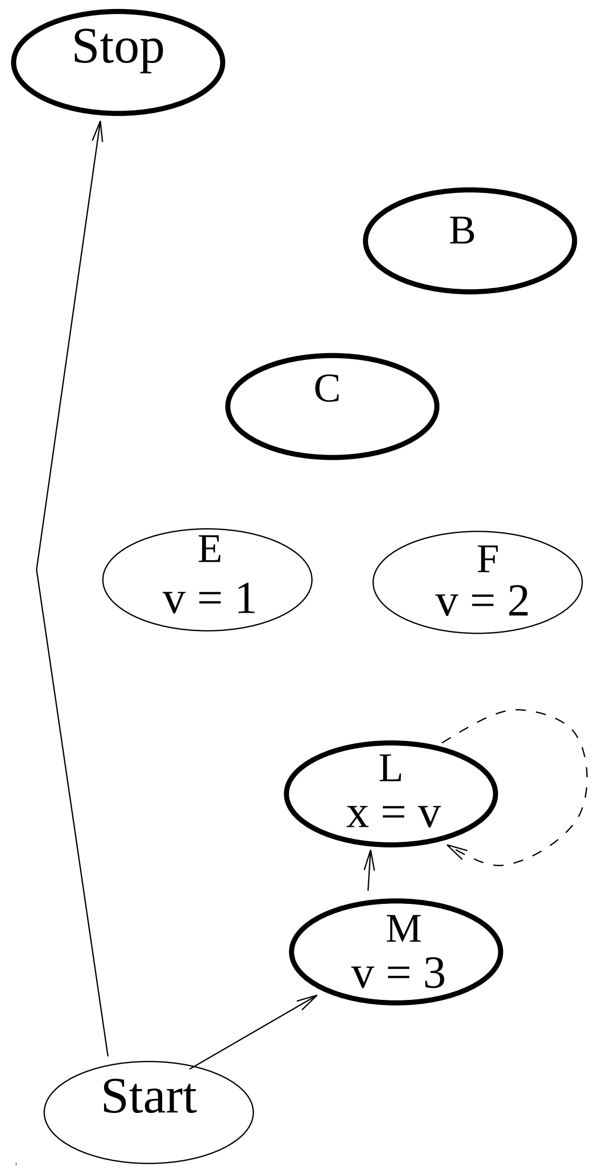
Edges in the sparse graph $\boxed{3}$



No edges are added to the sparse graph when nodes L and K are visited; however, the edges are mapped to the relevant sparse graph node (shaded).

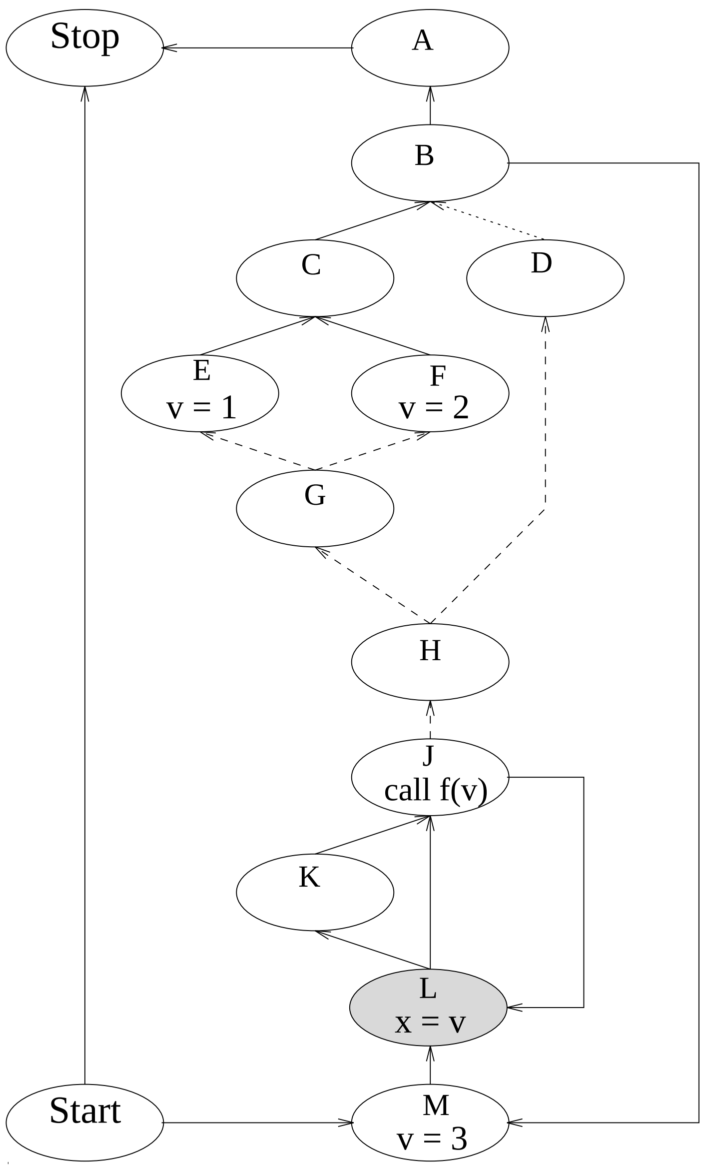
Edges in the sparse graph $\boxed{4}$

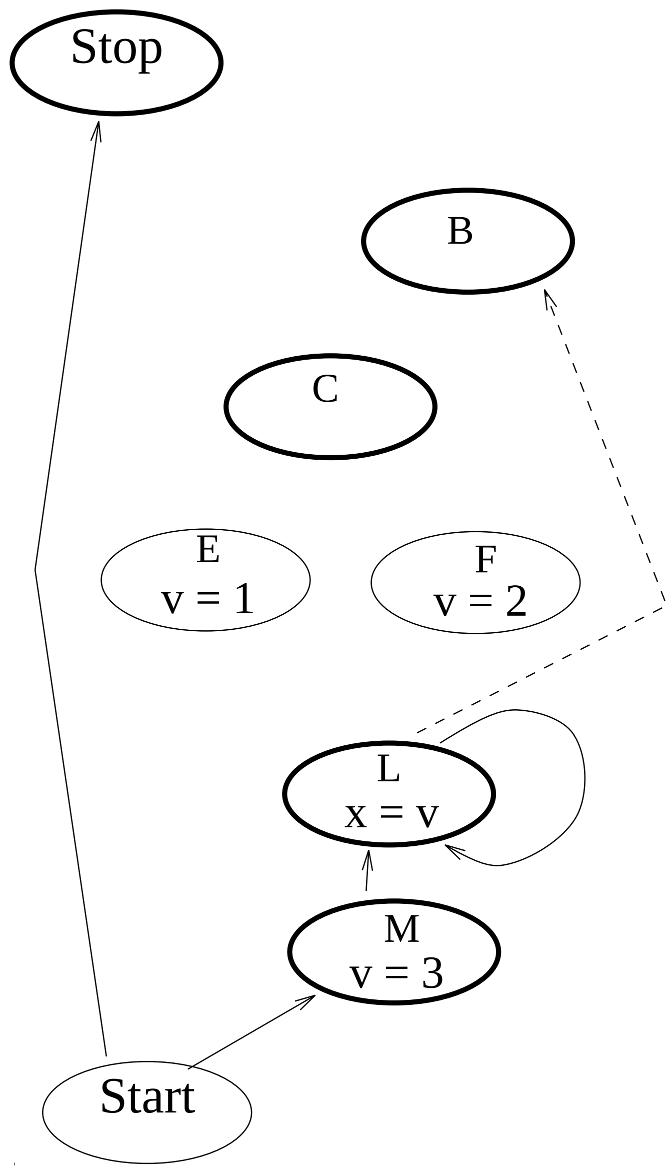




Edges are mapped, and the sparse edge to meet node L is installed.

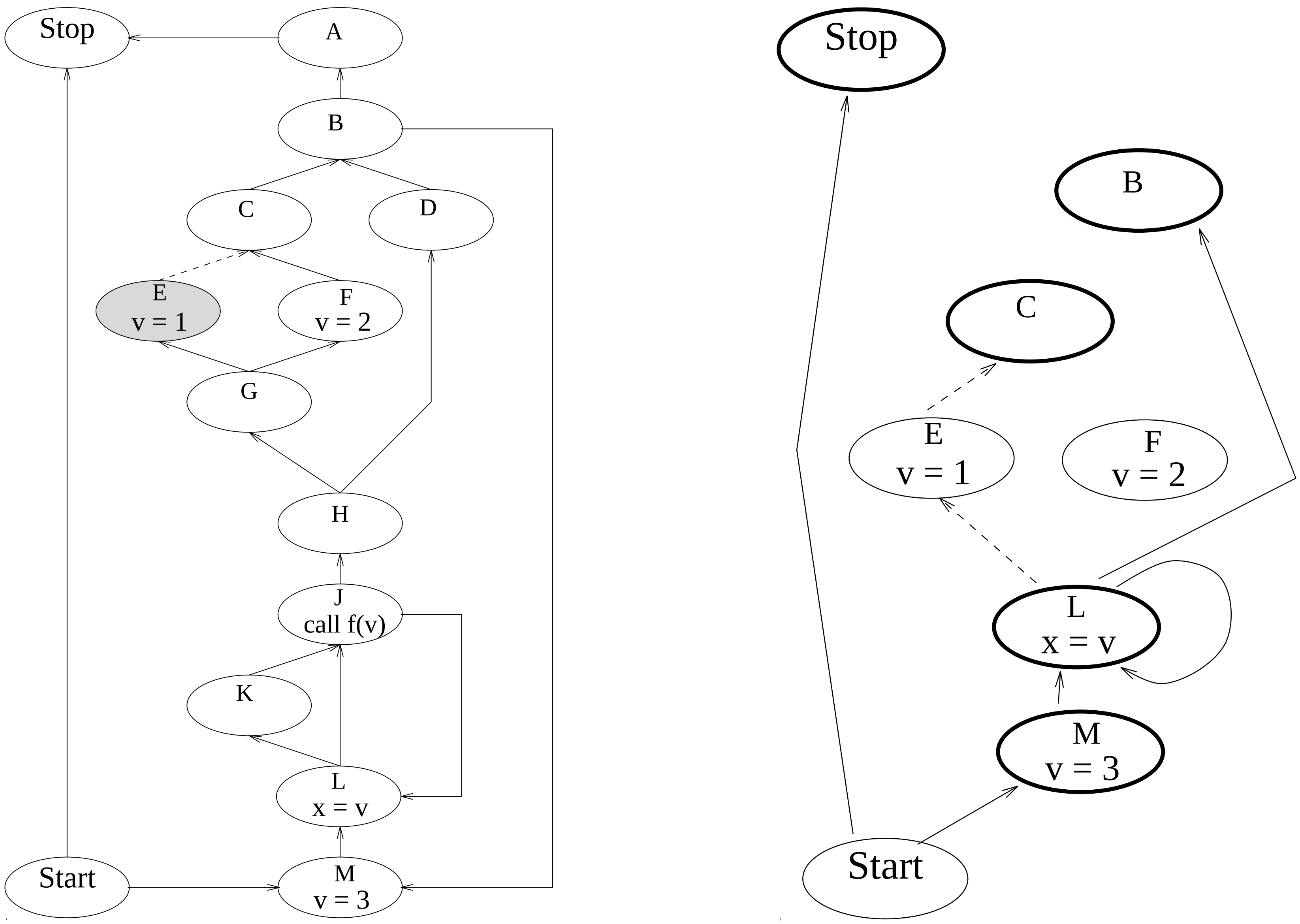
Edges in the sparse graph $\boxed{5}$





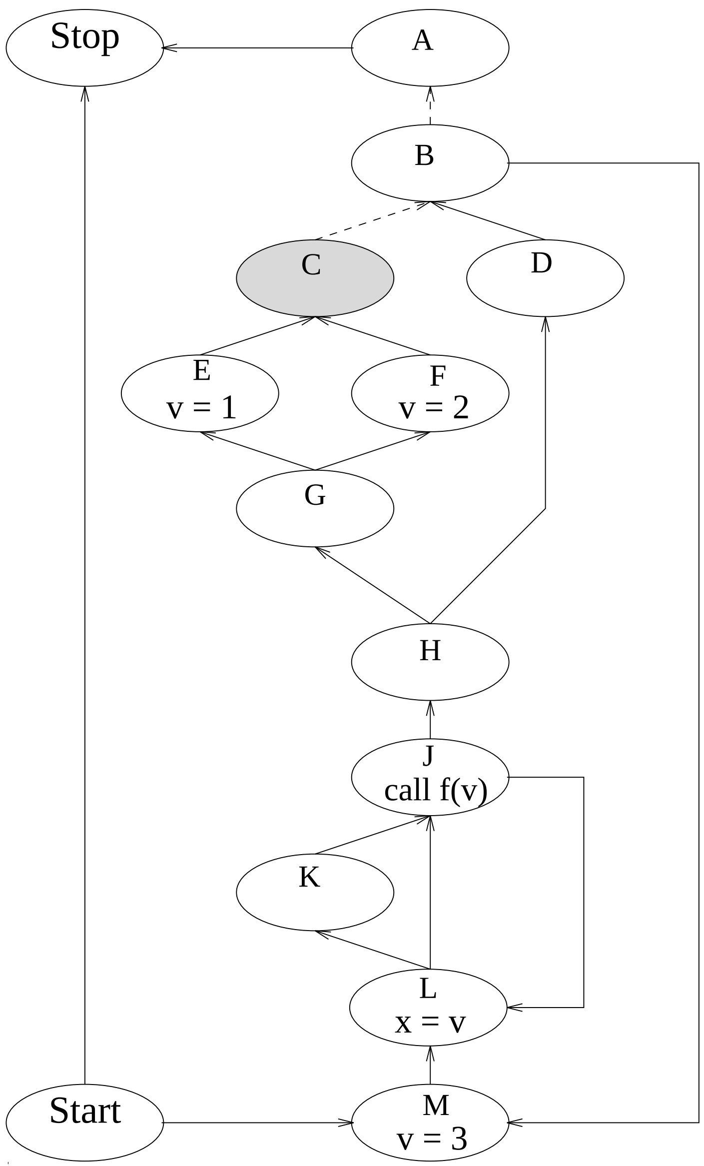
Here are shown the results of visiting nodes J, H, G, and D. For D, the meet edge is installed from L to B. Otherwise, only edge-mapping takes place.

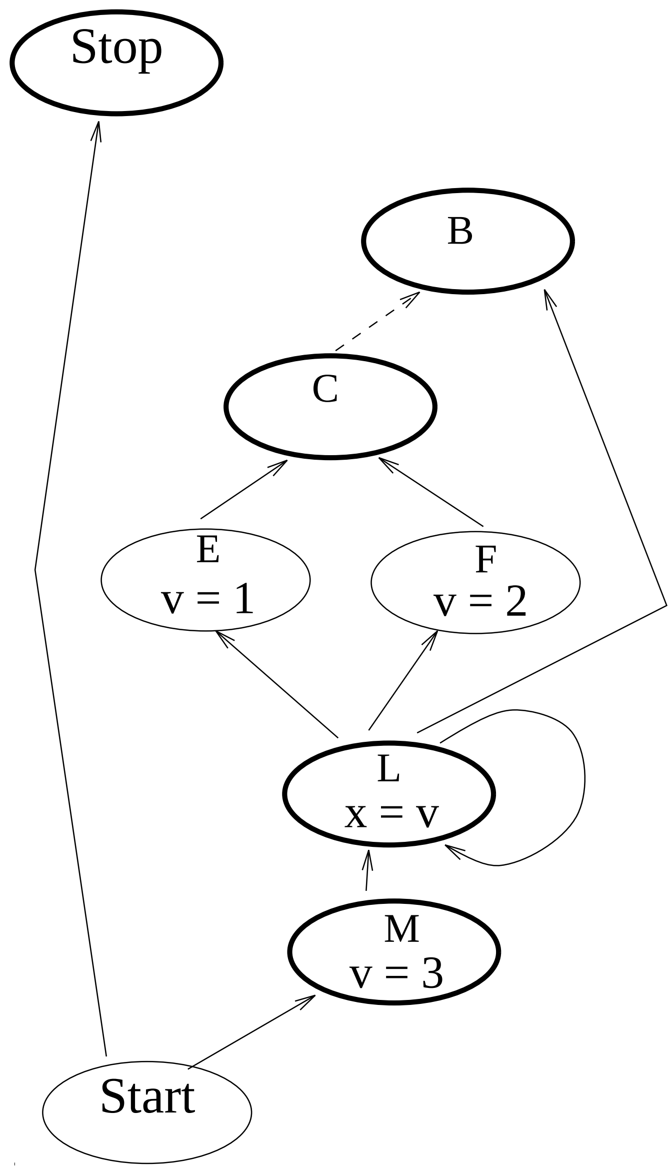
Edges in the sparse graph $\boxed{6}$



Prior to making E the relevant node, a link edge is installed to E from the former relevant node L. Edges are mapped and a meet edge is installed in the sparse graph from E to C. Similar behavior occurs when visiting F (elided).

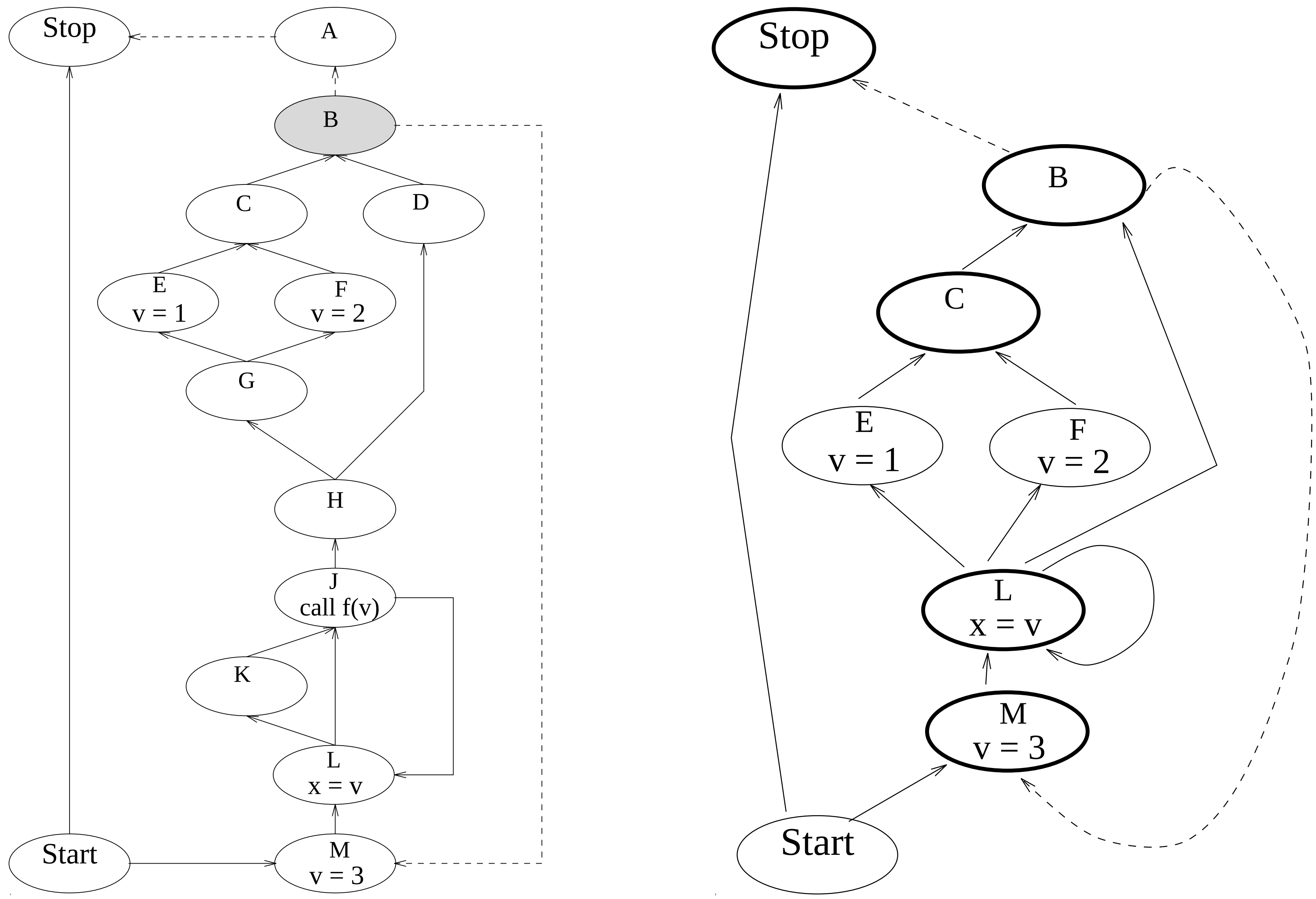
Edges in the sparse graph $\boxed{7}$





Here are shown the results of visiting node C.

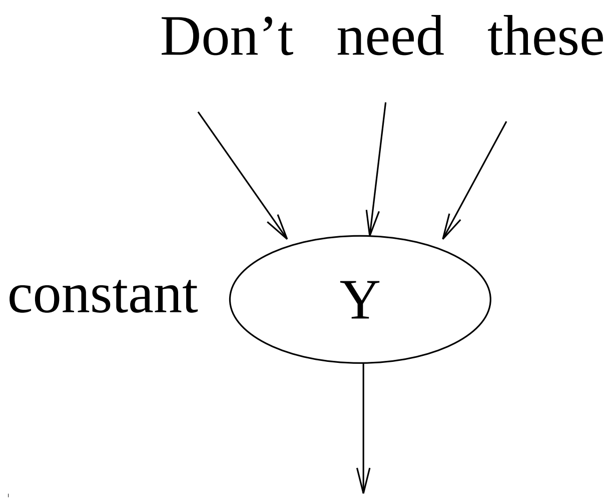
Edges in the sparse graph $\boxed{8}$



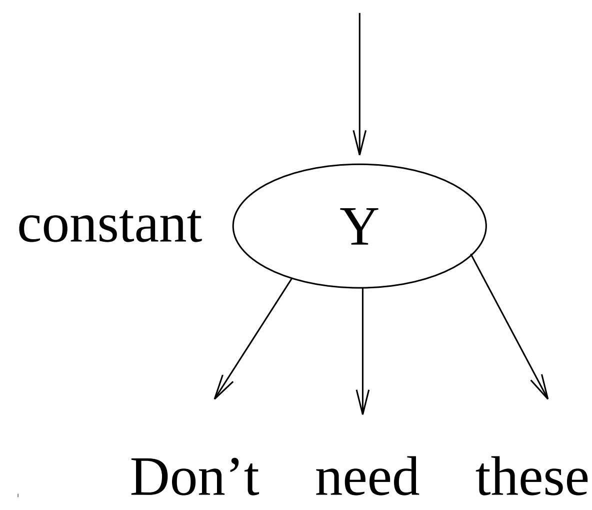
Here are shown the results of visiting nodes B and A, and the process is complete.

Edge optimization

If a node Y has constant transference ($f_Y = K_Y$), then the following optimizations are possible:



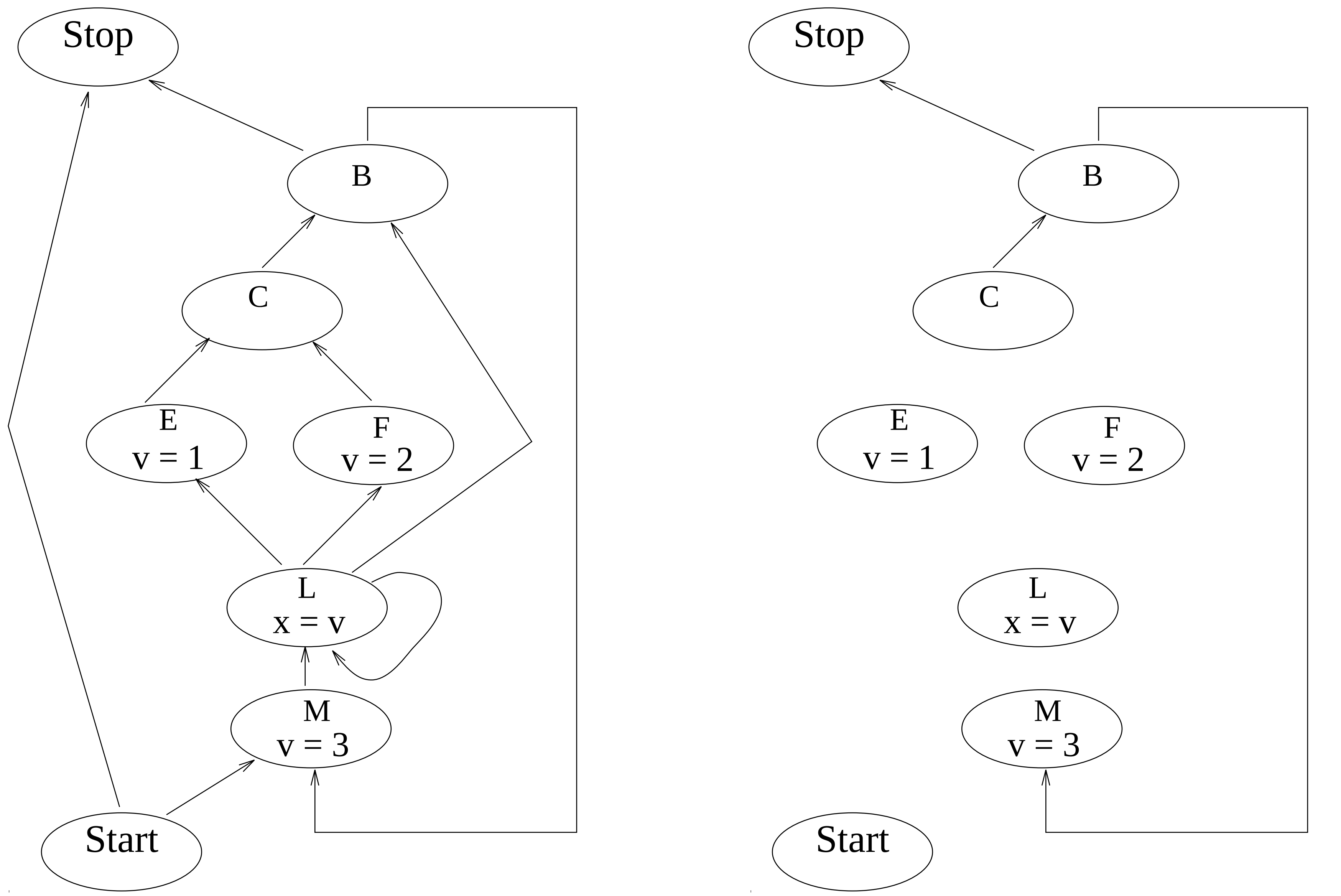
If the transfer function of node Y does not reference its input solution, then no information need be transmitted into the node at evaluation-time.



If the output of node Y is always the same, then node Y's solution can be forwarded to successors of Y prior to evaluation.

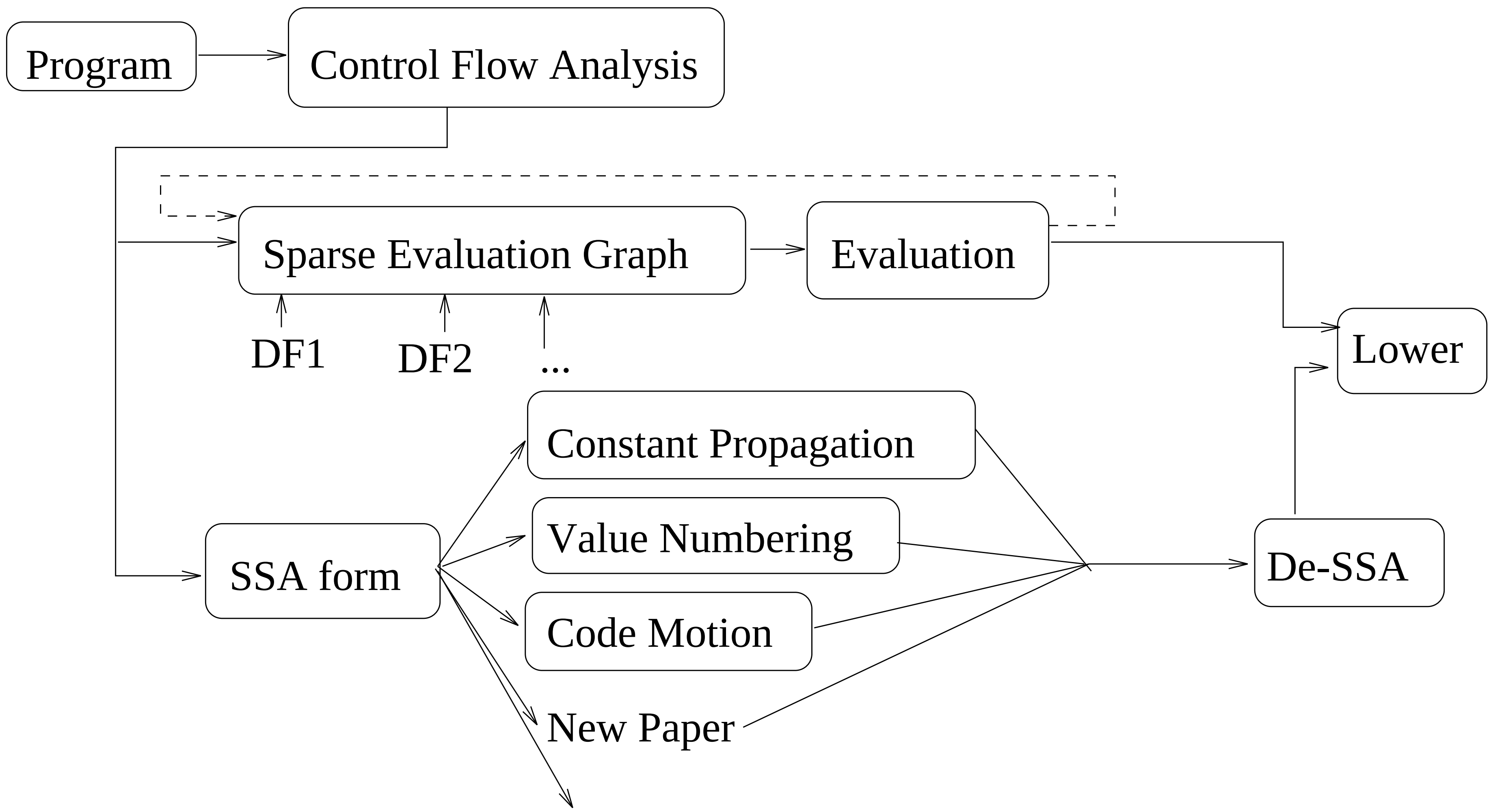
The left optimization is safe for any evaluation method; the right optimization is appropriate only where path-specific information is not utilized. Since our iterative evaluator performs meets prior to node evaluation, both optimizations are appropriate for that method.

Effects of optimization



The resulting graph is acyclic, so a single pass of the ordinary iterative evaluator suffices for convergence.

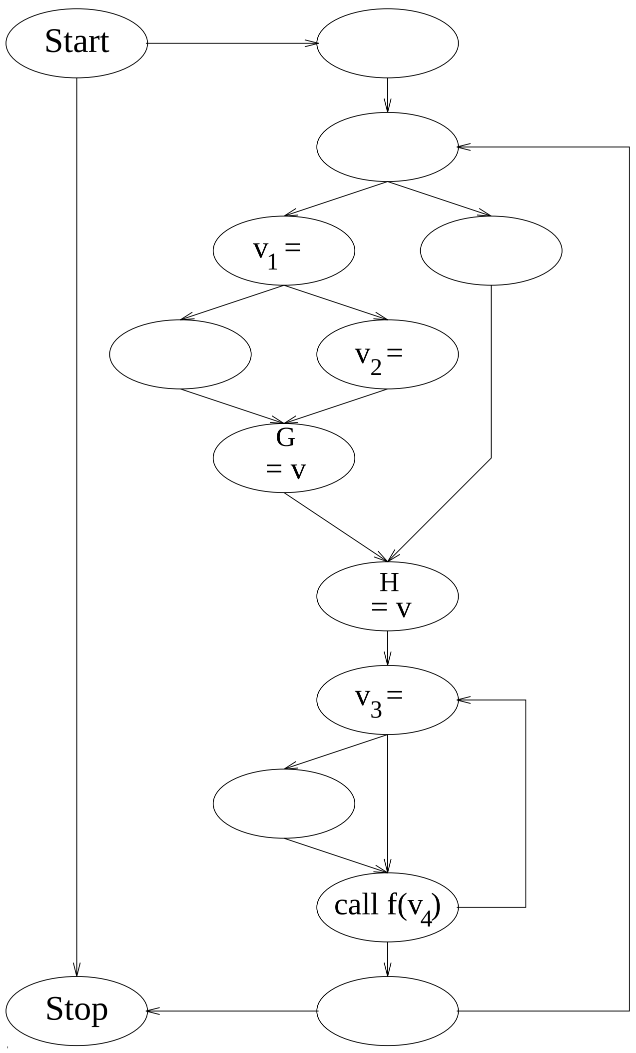
Big picture

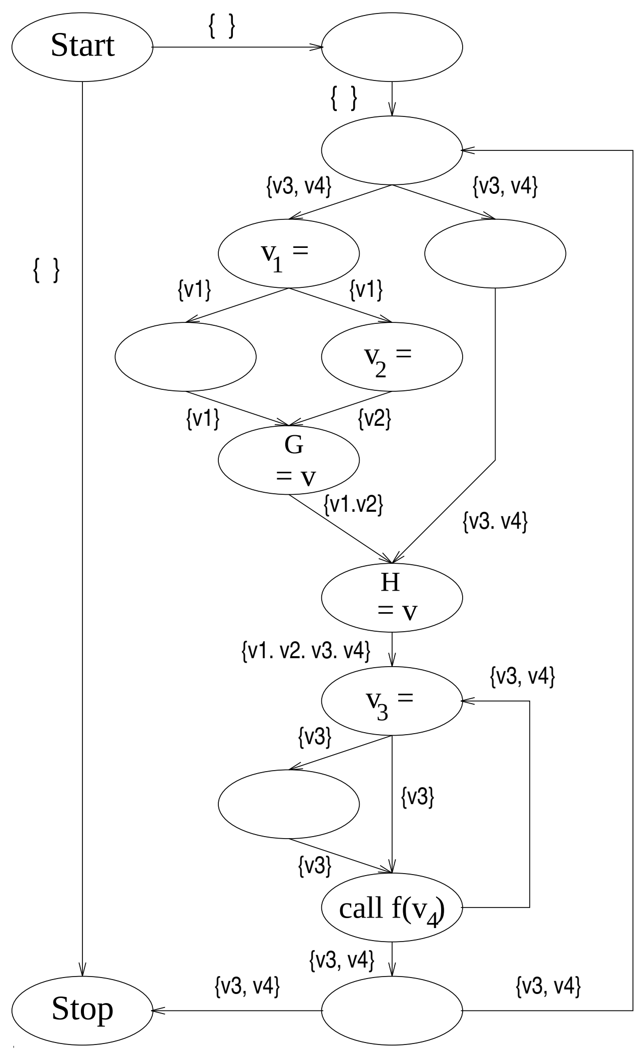


We'll now examine some special algorithms for optimization, based on a single assignment representation.

Static Single Assignment (SSA) form

Recall our reaching definitions example, shown on the left, and its solution, shown on the right:



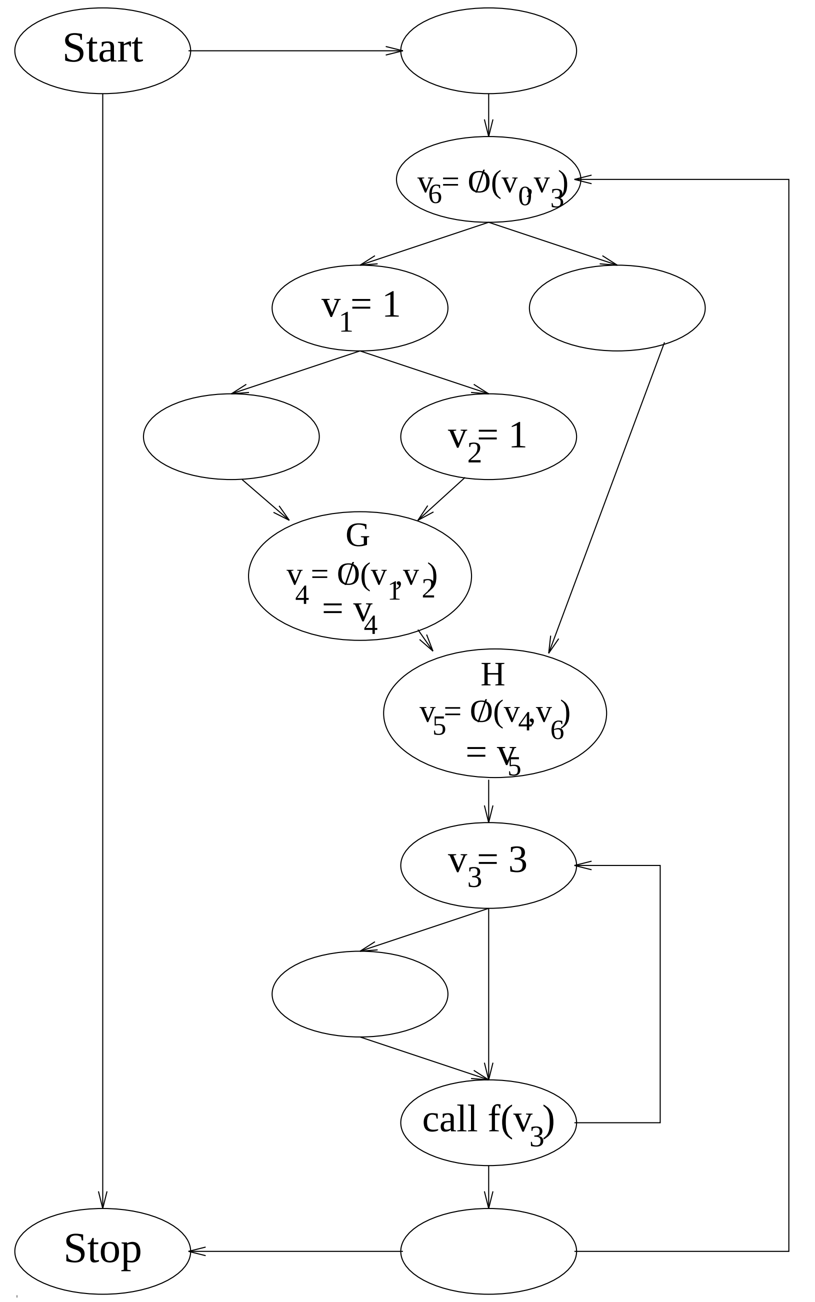


Notice how the use of `v` at `G` is reached by two definitions, and the use at `H` is reached by four definitions. If each use were reached by just a single definition, then a direct connection graph would yield a sparse representation of def-use chains.

SSA form (cont'd)

Here we see the SSA form of the program.

- Each definition of v is with respect to a distinct symbol: v_1 is as different from v_2 as x would be from y.
- Where multiple definitions reach a node, a ϕ-function is inserted, with arguments sufficient to receive a different "name" for v on each in-edge.
- Each use is appropriately renamed to the distinct definition that reaches it.
- Although ϕ-functions could have been placed at every node, the program shown has exactly the right number and placement of ϕ-functions to combine multiple defs from the original program.
- Our example assumes that procedure f does not modify v.



SSA form (cont'd)

Each def is now regarded as a "killing" def, even those usually regarded as preserving defs. For example, if v is *potentially* modified by the call site, then the old value for v must be passed into the called procedure, so that its value can be assigned to the name for v that *always* emerges from the procedure.

Procedure $foo(v)$

    if (c) **then**
        $v \leftarrow 7$
    else
        /* *Do nothing* */
    fi
end

Procedure $foo(v_{out}, v_{in})$

    $v_0 \leftarrow v_{in}$
    if (c) **then**
        $v_1 \leftarrow 7$
    else
        /* *Do nothing* */
    fi
    $v_2 \leftarrow \phi(v_0, v_1)$
    $v_{out} \leftarrow v_2$
end

SSA form can be computed by a data flow framework, in which the transfer function for a node with multiple reaching defs of v generates its own def of v. Uses are then named by the solution in effect at the associated node.

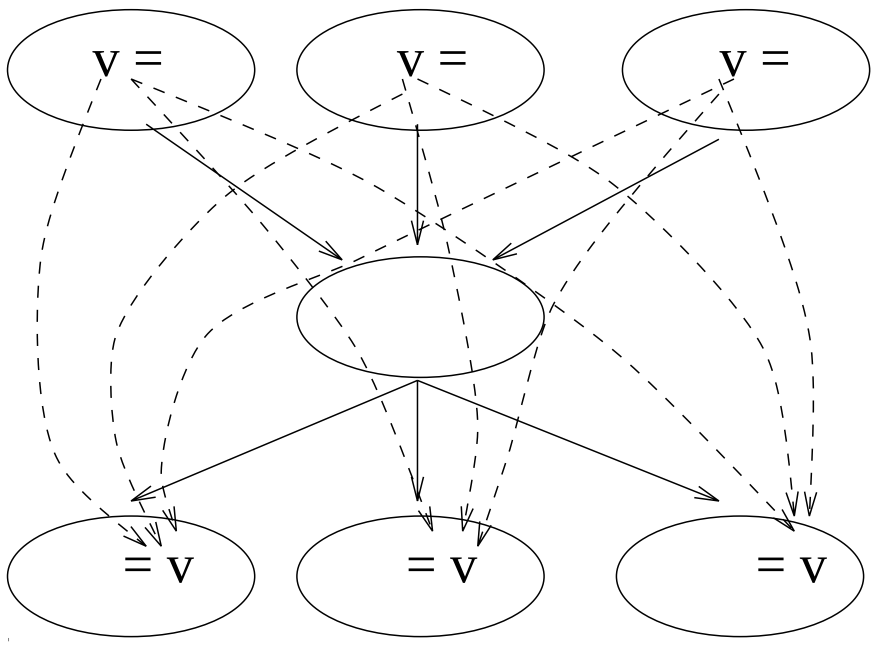
SSA form construction [28]

1. Turn every preserving def into a killing def, by copying potentially unmodified values (at subscripted defs, call sites, aliased defs, etc.).
2. Consider each node that defines v as having nonidentity transference.
3. ϕ-functions then go at the *meet* nodes, as determined by the sparse graph builder.
4. Uses are renamed using the solution propagated along the associated in-edge. For ordinary uses, nodes have only a single in-edge; for ϕ-uses, the use must be distinguished by in-edge;

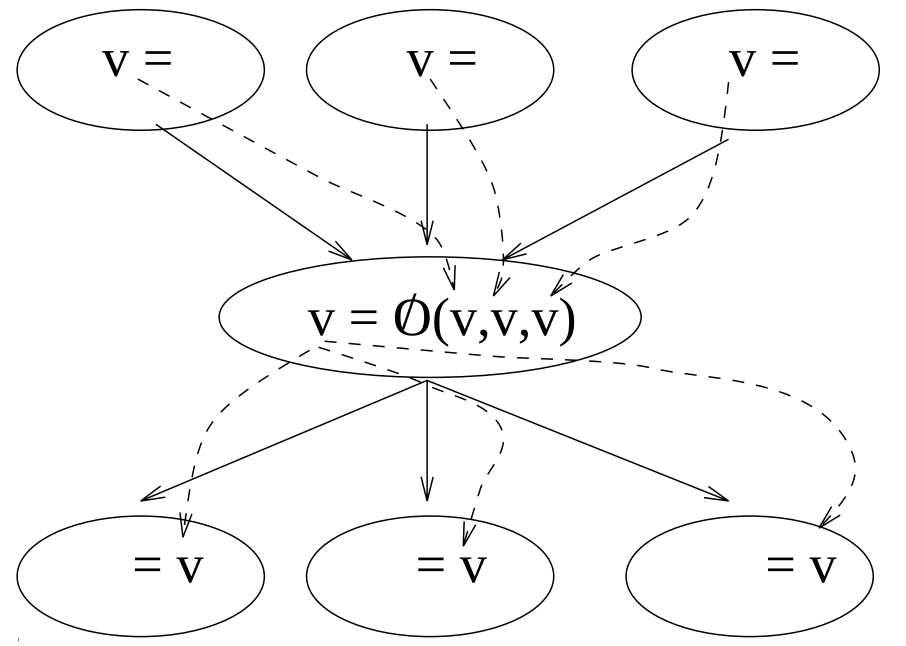
Actually, the SSA work predates the sparse evaluation graph work, but it's easier to describe the SSA algorithm this way if one already understand sparse evaluation graphs.

Why is SSA good?

Data flow algorithms built on def-use chains gain asymptotic efficiency as shown below:



Quadratic def-use chains



Linear def-use chains

With each use reached by a unique def, program transformations such as code motion are simplified: motion of a use depends primarily on motion of its unique reaching def. Intuitively, the program has been transformed to represent directly the flow of values. We'll now look at some optimizations that are simplified by SSA form.

SSA constant propagator [69]

Original Program	**SSA form**
$i \leftarrow 6$	$i_1 \leftarrow 6$
$j \leftarrow 1$	$j_1 \leftarrow 1$
$k \leftarrow 1$	$k_1 \leftarrow 1$
repeat	**repeat**
	$i_2 \leftarrow \phi(i_1, i_5)$
	$j_2 \leftarrow \phi(j_1, j_3)$
	$k_2 \leftarrow \phi(k_1, k_4)$
if $(i = 6)$ **then**	**if** $(i_2 = 6)$ **then**
$k \leftarrow 0$	$k_3 \leftarrow 0$
else	**else**
$i \leftarrow i + 1$	$i_3 \leftarrow i_2 + 1$
fi	**fi**
	$i_4 \leftarrow \phi(i_2, i_3)$
	$k_4 \leftarrow \phi(k_3, k_2)$
$i \leftarrow i + k$	$i_5 \leftarrow i_4 + k_4$
$j \leftarrow j + 1$	$j_3 \leftarrow j_2 + 1$
until $(i = j)$	**until** $(i_5 = j_3)$

Each name is initialized to the lattice value $\top$. Propagation proceeds only along edges marked *executable*. Such marking takes place when the associated condition reaches a non-$\top$ value. The value $\top$ propagates along unexecutable edges.

SSA constant propagator (cont'd)

SSA Form

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5)$
    $j_2 \leftarrow \phi(j_1, j_3)$
    $k_2 \leftarrow \phi(k_1, k_4)$
    if $(i_2 = 6)$ **then**
        $k_3 \leftarrow 0$
    else
        $i_3 \leftarrow i_2 + 1$
    fi
    $i_4 \leftarrow \phi(i_2, i_3)$
    $k_4 \leftarrow \phi(k_3, k_2)$
    $i_5 \leftarrow i_4 + k_4$
    $j_3 \leftarrow j_2 + 1$
until $(i_5 = j_3)$

Pass 1

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
    $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
    $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
    if $(i_2 = 6)$ **then**
        $k_3 \leftarrow 0$
    else
        /* *Not executed* ***/**
    fi
    $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
    $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
    $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
    $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) =$ **false**$)$

SSA constant propagator (cont'd)

Pass 1

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
$i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
$j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
$k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
if $(i_2 = 6)$ **then**
$k_3 \leftarrow 0$
else
/* *Not executed* */
fi
$i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
$k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
$i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
$j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) = \textbf{false})$

Pass 2

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
$i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge 6) = 6$
$j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge 2) = \perp$
$k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = \perp$
if $(i_2 = 6)$ **then**
$k_3 \leftarrow 0$
else
/* *Not executed* */
fi
$i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
$k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
$i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
$j_3 \leftarrow j_2 + 1 \Rightarrow (\perp + 1) = \perp$
until $(i_5 = j_3 \Rightarrow (6 = \perp) = \perp)$

Our solution has stabilized. Even though k_2 is $\perp$, that value is never transmitted along the unexecuted edge to the ϕ for k_4.

SSA value numbering [5, 56]

$a \leftarrow read()$
$v \leftarrow a + 2$
$c \leftarrow a$
$w \leftarrow c + 2$
$t \leftarrow a + 2$
$x \leftarrow t - 1$

For the above program, constant propagation will fail to determine a compile-time value for v and w, because the behavior of the $read()$ function must be captured as $\perp$ at compile-time.

Nonetheless, we can see that v and w will hold the same *value*, even though we cannot determine at compile-time exactly what that value will be. Such knowledge helps us replace the computation of $(c+2)$ by a simple copy from v.

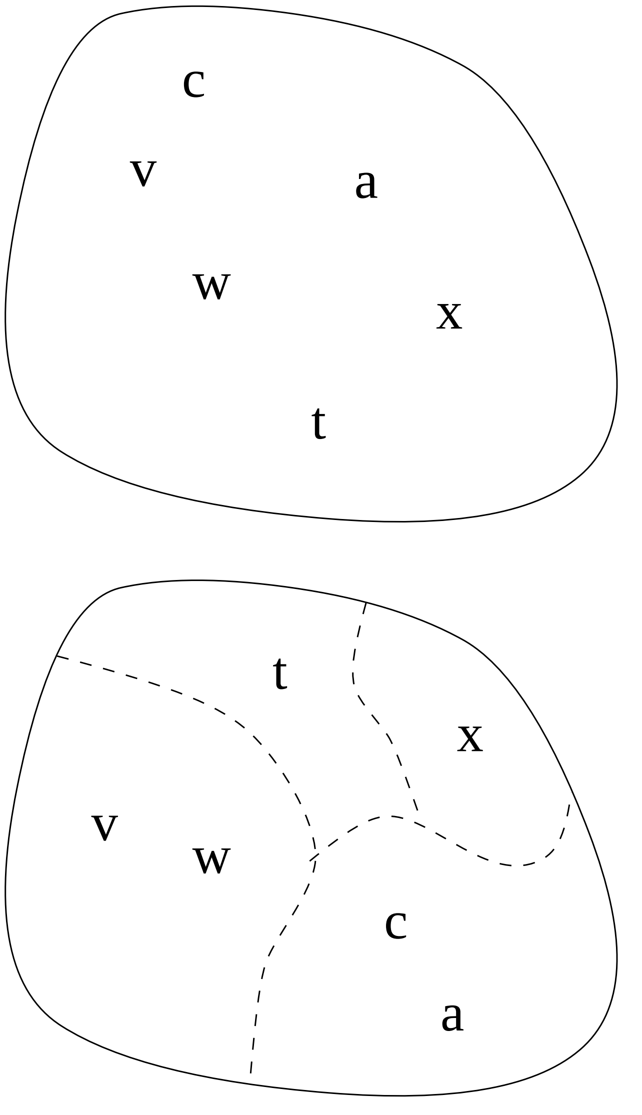
Value numbering attempts to label each computation of the program with a number, such that identical computations are identically labeled.

- **Prior to SSA form, value numbering algorithms were applied only within basic blocks (i.e., no branching) (2).**
- **Early value numbering algorithms relied on textual equivalence to determine value equivalence. The text of each expression (and perhaps subexpression) was *hashed* to a value number. Intervening defs of variables contained in an expression would kill the expression. This approach could not detect equivalence of v and w in the example to the left, since $(a + 2)$ is not textually equivalent to $(c + 2)$.**

It seems that x ought to have the same value as v and w, but our algorithm won't discover this, because the "function" that computes x ($\lambda n.n - 1$) differs from the "function" that computes v and w ($\lambda n.n + 2$).

SSA value numbering (cont'd)

- **We essentially seek a** *partition* **of SSA names by value equivalence, since value equivalence is reflexive, symmetric, and transitive.**
- **We'll initially assume that all SSA names have the same value.**
- **When evidence surfaces that a given block may contain disparate values (names), we'll talk about** *splitting* **the block. Generally, the algorithm will only split a block in two. However, the first split is more severe, in that names are split by the functional form of the expressions that compute their value.**

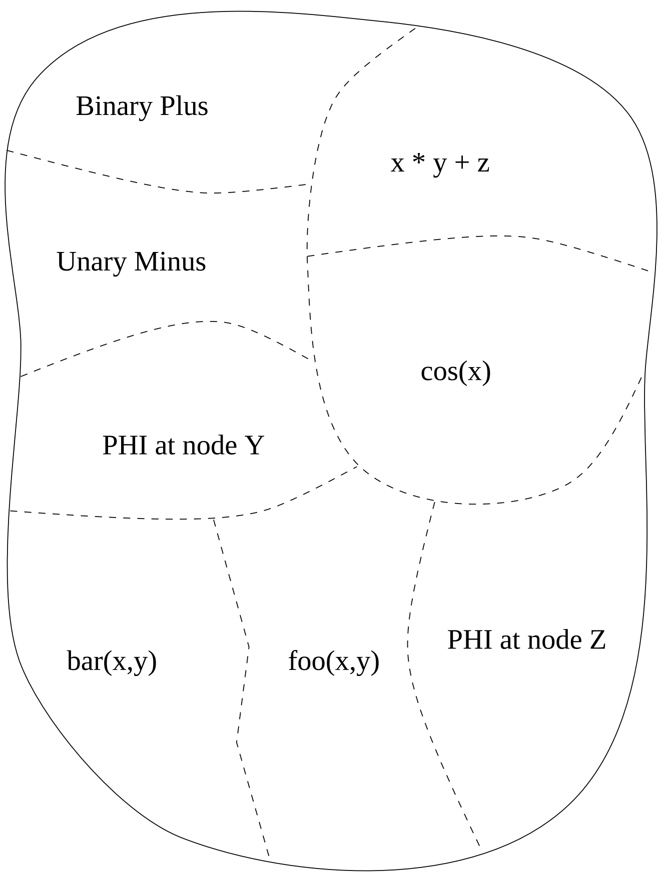


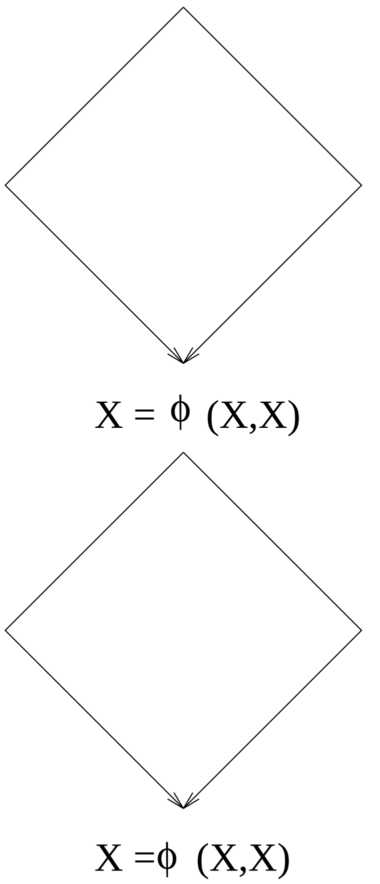
Above are shown the initial and final partitions for the example on the previous page.

SSA value numbering (cont'd)

After construction of SSA form, we split by the function name that computes values for the assigned variables. We thus distinguish between binary addition, multiplication, etc.

One further point is that ϕ-functions at different nodes must also be distinguished, even though their function form appears the same. This is necessary because a branch taken into one ϕ-function is not necessarily the same branch taken into another, unless the two functions reside in the same node.





SSA value numbering example

if $(condA)$ **then**

$a_1 \leftarrow \alpha$

if $(condB)$ **then**

$b_1 \leftarrow \alpha$

else

$a_2 \leftarrow \beta$

$b_2 \leftarrow \beta$

fi

$a_3 \leftarrow \phi(a_1, a_2)$

$b_3 \leftarrow \phi(b_1, b_2)$

$c_2 \leftarrow \star a_3$

$d_2 \leftarrow \star b_3$

else

$b_4 \leftarrow \gamma$

fi

$a_5 \leftarrow \phi(a_1, a_0)$

$b_5 \leftarrow \phi(b_0, b_4)$

$c_3 \leftarrow \star a_5$

$d_3 \leftarrow \star b_5$

$e_3 \leftarrow \star a_5$

For brevity, symbols α, β, and γ represent syntactically distinct function classes in the program shown to the left.

In the figures that follow, we'll see that c_2 and d_2 have the same value, while c_3 and d_3 do not. Thus, program optimization will save a memory fetch by using the value of c_2 for d_2.

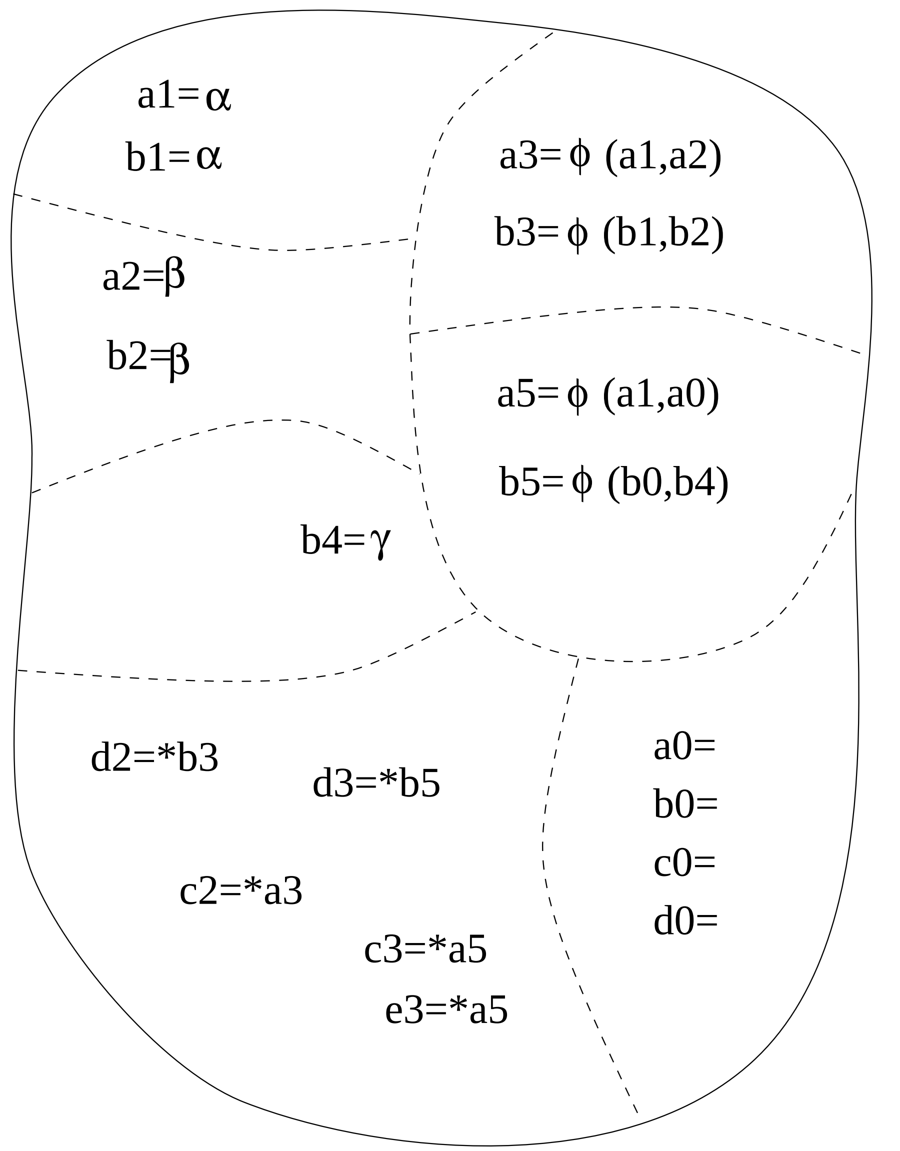
Note that if b is declared *volatile* in the language C, then this optimization would be incorrect, because each reference to b should be realized. How can one account for volatility in this optimization? Perhaps by assuming that volatile variables cannot have the same value.

It would be difficult and expensive to express all possible defs of a volatile variable in SSA form.

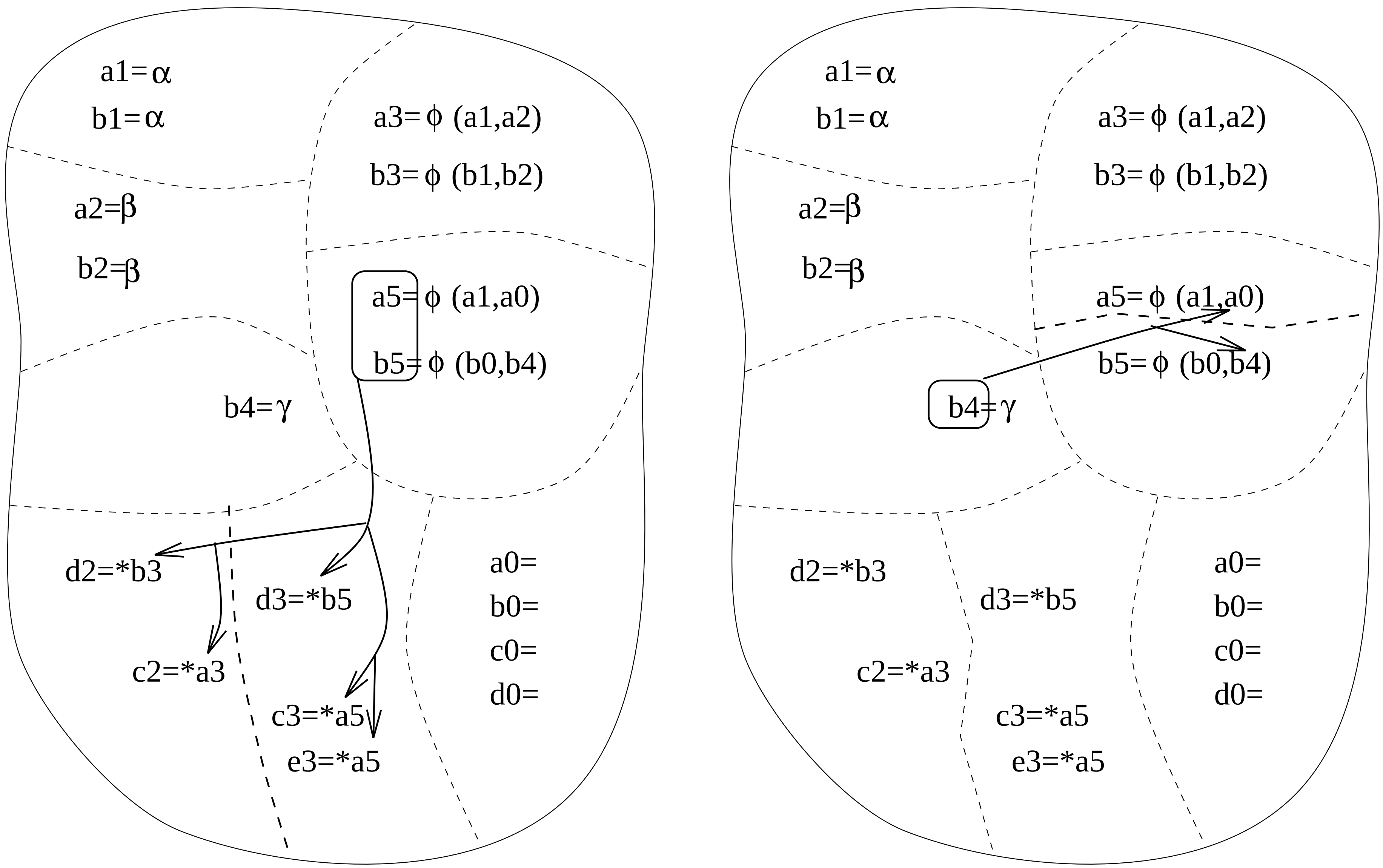
SSA value numbering example (cont'd)

Here we see the initial partition of SSA names:

- **The syntactic classes α, β, and γ are distinguished;**
- **ϕ-functions at different nodes are distinguished;**
- **The initial value of each variable v_0 is considered identical;**
- **Within each syntactic class, values are considered identical.**

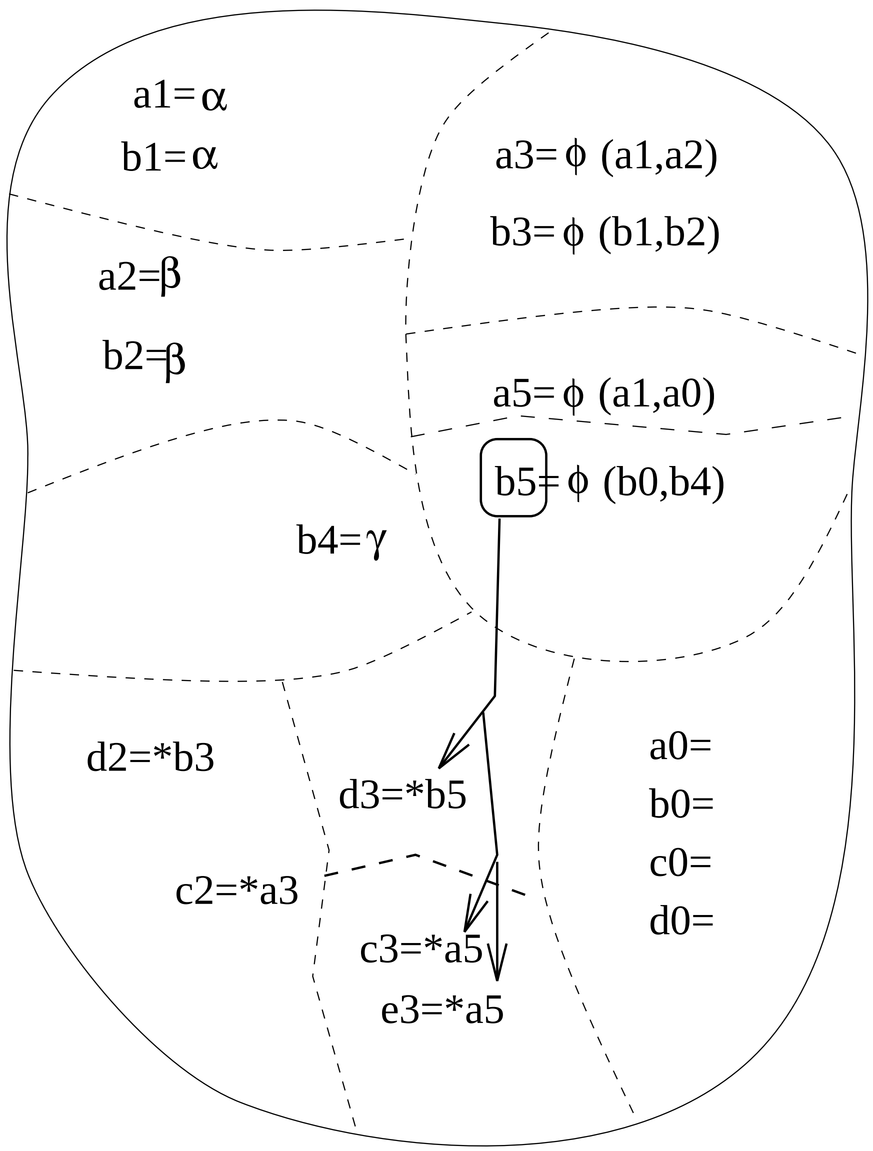


SSA value numbering example (cont'd)



On the left, the block with a_5 splits the five names shown into two subblocks; on the right, b_4 splits a_5 from b_5.

SSA value numbering example (cont'd)

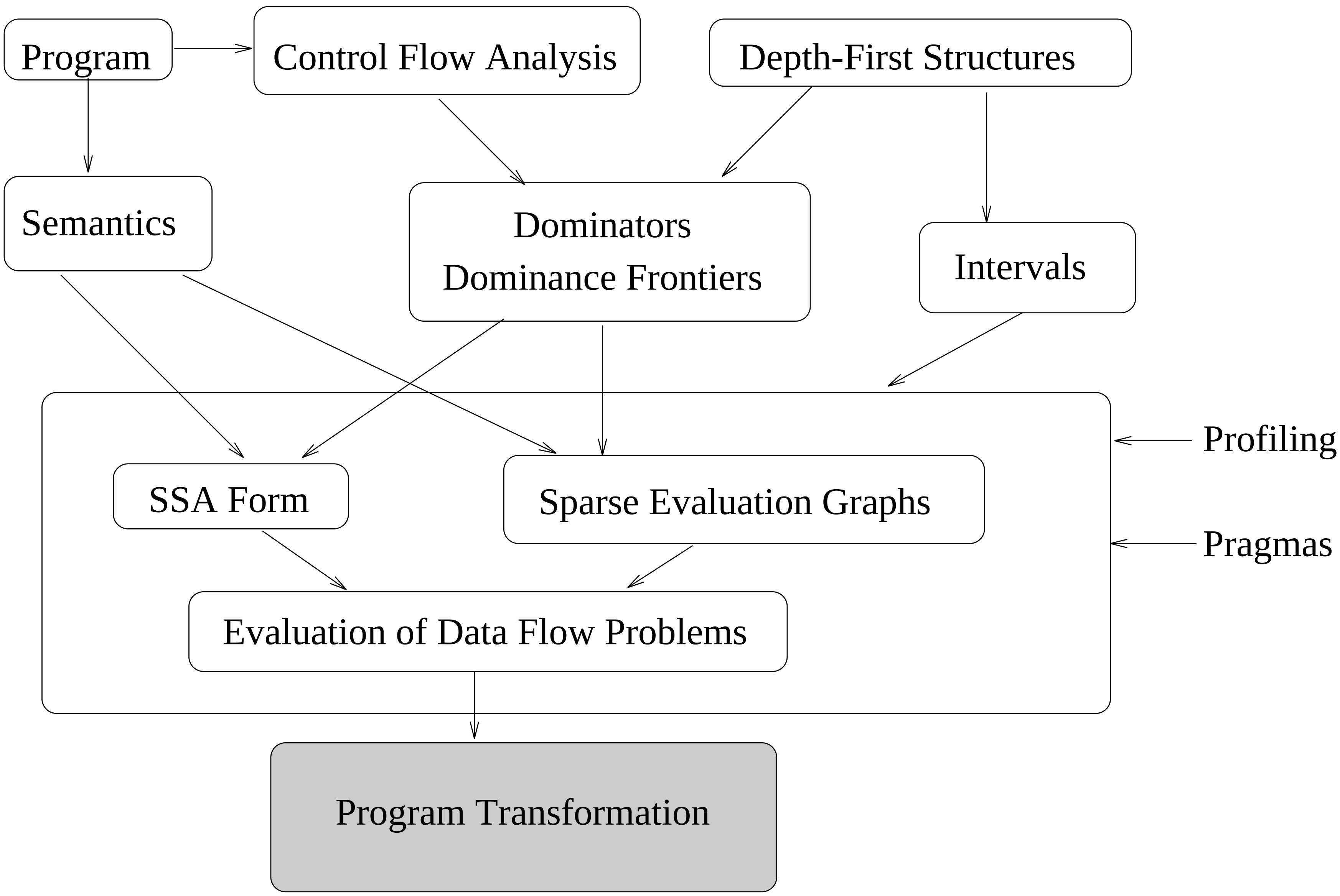


Finally, b_5 splits c_3 from d_3. Here, note that we could have used either a_5 or b_5 to do the job. Asymptotic efficiency is gained by choosing b_5, because there are fewer uses of that name than of a_5.

In summary, the algorithm is as follows:

1. **Let W be a worklist of blocks to be used for further splitting.**
2. **Pick and remove (arbitrary) block D from W.**
3. **For each block C properly split by D,**
   - **(a) If C is on W, then remove C and enqueue its splits by D;**
   - **(b) Otherwise, enqueue the split with the fewest uses.**
4. **Loop to step 2 until W is empty.**

Conclusion



An important open problem is the efficient integration of program transformation and program analysis.

Conclusion (cont'd)

1. **The cost of optimization should be proportional to the *benefits* attributable to optimization.**
   (a) **Construction of an intermediate form should not dominate optimization time (29).**
   (b) **Where little benefit is expected, expensive analysis should be computed *on demand* rather than exhaustively (62, 22).**
2. **The expected asymptotic complexity of optimization applied throughout a program must be linear or almost-linear.**
   (a) **Expected and worst cases must be considered (27).**
   (b) **Profiling and control flow analysis can determine needy portions of a large program (61, 9).**
   (c) **Small procedures must compile quickly, even under extensive optimization.**
3. **Extensions and revisions of older languages are often targeted for improved program optimization (39, 38, 40).**
4. **New languages and language paradigms offer challenging optimization problems (19, 18).**
5. **New and useful program optimizations can be developed most effectively by in-the-trench source and object code evaluation.**

References

(1) Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

(2) A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

(3) F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 19(6):13–24, 1970.

(4) Frances E. Allen. Interprocedural data flow analysis. *Proc. IFIP Congress 74*, pages 398–402, 1974.

(5) B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 1–11, January 1988.

(6) Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

(7) M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, pages 22–31, June 1982. Published as *SIGPLAN Notices* Vol. 17, No. 6.

(8) B. S. Baker. An algorithm for structuring flowgraphs. *J. ACM*, pages 98–120, January 1977.

(9) Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1992.

(10) R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand driven interpretation of languages. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 257–271, June 1990. Published as *SIGPLAN Notices* Vol. 25, No. 6.

(11) Jeffrey Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

(12) M.G. Burke and B.G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7):723–728, July 1990.

(13) Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

(14) Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Trans. on Programming Languages and Systems*, July 1993.

(15) D. Callahan, A. Carle, M. Hall, and K. Kennedy. Constructing the call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, 1990.

(16) David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proceedings of the Sigplan '88 Conference on Programming Language Design and Implementation*, 23(7):47–56, July 1988. Atlanta, Georgia.

(17) M.D. Carroll. *Data Flow Update via Dominator and Attribute Updates*. PhD thesis, Rutgers University, May 1988.

(18) Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992. Available as technical report STAN-CS-92-1420.

(19) Craig Chambers and David Ungar. Making pure object-oriented languages practical. *SIGPLAN Notices*, 26(10), 1991. OOPSLA Conference Proceedings.

(20) W.-M. Ching. Program analysis and code generation in an apl/370 compiler. *IBM Journal of Research and Development*, 30(6):594–602, 1986.

(21) Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991.

(22) Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *Submitted to IEEE TSE*, 1993.

(23) Keith Cooper. *Interprocedural Data Flow Analysis in a Programming Environment*. PhD thesis, Rice University, 1983.

(24) Keith D. Cooper and Ken Kennedy. Complexity of interprocedural side-effect analysis. Technical report, Department of Computer Science, Rice University, October 1987. TR87-61.

(25) Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the rn programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

(26) R. Cytron, A. Lowry, and F. K. Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. Thirteenth ACM Symp. on Principles of Programming Languages*, pages 70–85, January 1986.

(27) Ron Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. *Proceedings of the Workshop on Languages and Compiling for Parallelism*, 1993. submitted.

(28) Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991.

(29) Ron Cytron and Reid Gershbein. Efficiently accommodating may-alias information in ssa form. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1993.

(30) Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches (extended abstract). *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988. Also available as CSRD Rpt. No. 728 from U. of Ill.-Center for Supercomputing Research and Development.

(31) D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of morel and renvoise. *ACM Trans. on Programming Languages and Systems*, 13(2):291–294, April 1991.

(32) D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyse large programs efficiently and informatively. *Proc. SIGPLAN'92 Symp. on Compiler Construction*, June 1992. Published as *SIGPLAN Notices* Vol. 27, No. 6.

(33) Dhananjay M. Dhamdhere and Uday P. Khedker. Complexity of bi-directional data flow analysis. *Conf. Rec. Twentieth ACM Symp. on Principles of Programming Languages*, 1993.

(34) R.K. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program flow analysis. *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, 1975.

(35) J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

(36) M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974.

(37) M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Computing*, 4(4):519–532, December 1975.

(38) S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

(39) S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

(40) S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

(41) L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Trans. on Software Engineering*, SE-7(1):60–78, January 1981.

(42) Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Notices*, 27(7), 1992. SIGPLAN PLDI Conference Proceedings.

(43) Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12,13:26–60, January 1990.

(44) J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

(45) John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *JACM*, 23,1:158–171, January 1976.

(46) Ken Kennedy. A survey of data flow analysis techniques. In Stephen S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, chapter 1. Prentice–Hall, 1981.

(47) T. Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, July 1979.

(48) T. J. Marlowe. *Data Flow Analysis and Incremental Iteration*. PhD thesis, Dept. of Computer Science, Rutgers U., October 1989.

(49) Eugene W. Myers. A precise inter-procedural data flow algorithm. *Conference Record Eighth ACM Symposium on Principles of Programming Languages*, 1981.

(50) Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *Software Engineering Notes*, 9(3), 1984. Also appears in Proceedings of the ACM Symposium on Practical Programming Development Environments, Pittsburgh, PA, April, 1984, and in SIGPLAN Notices, Vol. 19, No 5, May, 1984.

(51) B. K. Rosen. High level data flow analysis. *Comm. ACM*, 20(10):712–724, October 1977.

(52) B. K. Rosen. Data flow analysis for procedural languages. *J. ACM*, 26(2):322–344, April 1979.

(53) B. K. Rosen. Monoids for rapid data flow analysis. *SIAM J. Computing*, 9(1):159–196, February 1980.

(54) B. K. Rosen. Degrees of availability as an introduction to the general theory of data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 2, pages 55–76. Prentice Hall, 1981.

(55) B. K. Rosen. A lubricant for data flow analysis. *SIAM J. Computing*, 11(3):493–511, August 1982.

(56) B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 12–27, January 1988.

(57) Barbara Ryder. Constructing the call graph of a program. *IEEE Software Engineering*, May 1979.

(58) Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

(59) Barbara G. Ryder and Marvin C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

(60) B.G. Ryder, T.G. Marlowe, and M.C. Paull. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11(1):1–15, October 1988.

(61) Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.

(62) Vivek Sarkar. The ptran parallel programming system. *Parallel Functional Programming Languages and Compilers*, pages 309–391, 1991.

(63) J. T. Schwartz and M. Sharir. Tarjan's fast interval finding algorithm. Technical report, Courant Institute, New York University, 1978. SETL Newsletter Number 204.

(64) M. Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

(65) Olin Shivers. Control flow analysis in scheme. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, 1988. Published as *SIGPLAN Notices* Vol. 23, No. 7.

(66) Robert Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, December 1974.

(67) G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. *Proc. SIGPLAN'89 Symp. on Compiler Construction*, pages 1–12, 1989. Published as *SIGPLAN Notices* Vol. 24, No. 7.

(68) Jr. Vincent A. Guarna, Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur. Faust: An integrated environment for the development of parallel programs. Technical report, U. of Il.-Center for Supercomputing Research and Development, November 1988. CSRD Rpt. No. 825 to appear in IEEE Software, 1989.

(69) M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.

(70) F. K. Zadeck. Incremental data flow analysis in a structure program editor. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pages 132–143, June 1984. Published as *SIGPLAN Notices* Vol. 19, No. 6.

(71) P. T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Xerox Palo Alto Research Center, Palo Alto, Ca. 94304, May 1984.