[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., 1977.

[Ken81] Ken Kennedy. A survey of data flow analysis techniques. *Program Flow Analysis: Theory and Applications*, 1981.

[Kil73] G. Kildall. A unified approach to global program optimization. *Conference Record of First ACM Symposium on Principles of Programming Languages*, 194–206, January 1973.

[LT79] T. Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, July 1979.

[Mar89] Thomas J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, October 1989.

[RL77] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. *Conf. Rec. Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[Ros79] Barry K. Rosen. Data flow analysis for procedural languages. *JACM*, 26(2):322–344, April 1979.

[Ros81] Barry K. Rosen. Degrees of availability as an introduction to the general theory of data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[RT82] J. H. Reif and Robert Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, February 1982.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. *Fifteenth ACM Principles of Programming Languages Symposium*, 12–27, January 1988. San Diego, CA.

[Tar81] Robert Tarjan. Fast algorithms for solving path problems. *Journal of the Association for Computing Machinery*, 28(3):594–614, 1981.

[WZ85] Mark Wegman and Ken Zadeck. Constant propagation with conditional branches. *Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages*, 291–299, January 1985.

[YHR89] Wuu Yang, Susan Horwitz, and Thomas Reps. *Detecting Program Components With Equivalent Behaviors.* Technical Report, University of Wisconsin, Madison, April 1989. Computer Sciences Technical Report Number 840.

[Zad84] Frank Kenneth Zadeck. Incremental data flow analysis in a structured program editor. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, 19(6):132–143, June 1984.

Node visited	Link from	New *TOS*	Edges mapped	Link to
`Entry`		`Entry`	`Entry` → 1 `Entry` → `Exit`	`Exit`
1			1 → 2	2
2		2	2 → 3 2 → 7	
3			3 → 4 3 → 5	
4	2	4	4 → 6	6
5	2	5	5 → 6	6
6		6	6 → 8	8
7	2	7	7 → 8	8
8		8	8 → 9	9
9		9	9 → 10 9 → 11	
10			10 → 11	
11	9	11	11 → 9 11 → 12	9
12	11	12	12 → 2 12 → `Exit`	2 `Exit`
`Exit`		`Exit`		

Table 4. Reaching Uses for `v`

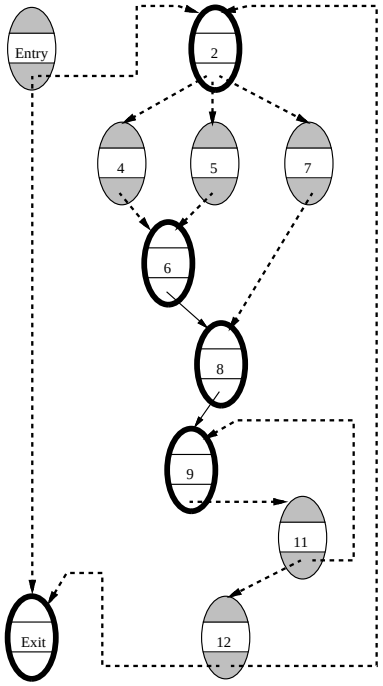


Figure 11. Sparse evaluation graph for Reaching Uses (v).

7 Conclusions and Future Work

We have shown in this paper a sparse evaluation graph can be used to solve forward and backward data flow problems, affords the same potential efficiency as SSA-based solutions, and does not require bit vectors at any step. Moreover, our technique applies equally well to irreducible [ASU86] programs. Although general, our approach is especially attractive for

1. partitionable [Zad84] data flow problems, since for a given partition, there are likely to be many nodes with the identity transfer function, and
2. structured flow graphs, since there are likely to be few meet nodes.

Many open questions are raised by this work. Can our method handle simultaneous solution of coupled data flow problems, like the constant propagation and branch deletion of [WZ85]? For what class of problems can a given evaluation method achieve a time bound *linear* in the size of our evaluation graph? Are there interesting forward or backward problems that are *not* well-suited to this method? We expect these questions to be the subject of future work.

Acknowledgements

We thank Barry Rosen and Ken Zadeck for several helpful technical conversations. We thank Fran Allen, Michael Burke, and Larry Carter for their helpful comments.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. *Fifteenth ACM Principles of Programming Languages Symposium*, 1–11, January 1988. San Diego, CA.

[CCF] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient treatment of preserving definitions. In preparation.

[CFR*89a] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 25–35, January 1989.

[CFR*89b] 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. *Submitted to ACM Transactions on Programming Languages and Systems*, June 1989.

[CLZ86] Ron Cytron, Andy Lowry, and Ken Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. of the ACM Symp. on Principles of Compiler Construction*, 1986.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 319–349, July 1987.

1. Compute $Live(V)$ as described in Section 4.
2. Define the predicate

$$LiveNode(V,Y) \equiv \left(\bigwedge_{Z \in Succs(Y)} OUT_{M((Y,Z))}\right) = \bot$$

3. Place ϕ-functions as described in Figure 4 of [CFR*89a], but change line 12 to read:

   `if` $DomFronPlus(Y) = 0$
   `and` $LiveNode(V,Y)$

4. Complete SSA construction as previously described [CFR*89a].

Figure 10. Sketch of algorithm to compute pruned SSA form.

5.1 Pruned SSA Form

Although SSA form nicely represents value flow in a program, it has one disadvantage in detecting program components with equivalent behaviors [YHR89]. In the control graph of Figure 4, the construction of SSA Form places a *dead* ϕ-function for `v` at node 2: the value defined by that ϕ-function is never used. Some algorithms based on SSA form depend on such ϕ-functions. For example, `v` might represent a value useful in value numbering, even if the original program did not refer to that value as `v`.

Unfortunately, dead ϕ-functions hinder detection of equivalent behavior. Suppose the edge $12 \rightarrow 2$ were removed from Figure 4, causing the ϕ-function at node 2 to disappear for `v`. Although the two programs might have equivalent behavior as far as `v` is concerned, such equivalence is not readily apparent, because program *slices* with respect to `v` would indicate a ϕ-function in the original program and its absence in the altered program.

Dead code elimination could remove the offensive ϕ-functions, but this would add to the expense of establishing a useful program form prior to optimization. Alternatively, an algorithm has been proposed that does not insert dead ϕ-functions [YHR89], but this algorithm is restricted to certain (structured) program constructs. We are therefore interested in constructing pruned SSA form for arbitrary control flow graphs, in which dead ϕ-functions are never introduced.

Because of space constraints, we cannot state in full an algorithm for constructing pruned SSA form. Instead, Figure 10 shows how to modify the published SSA form construction algorithm [CFR*89a] to obtain pruned form. When computing SSA form with respect to a variable V, liveness is first determined. The placement of ϕ-functions is then limited to nodes at which V is live. During renaming, some nodes will not have a single, valid name available for V, but renaming at such nodes is not required because V is dead.

6 Reaching Uses

In this section we sketch how the Reaching Uses problem, a forward problem, can be solved with the same efficiency as the SSA-based method.

Definition 5 *A use of a variable v at node Z* reaches *a CFG edge $X \rightarrow Y$ if there exist*

- *a (possibly empty) path in CFG $Z \xrightarrow{*} X$, and*
- *no nodes on the path $Z \xrightarrow{*} X$ that* kill *v.*

We show how Reaching Uses is solved for variable `v` in the control flow graph of Figure 4. Consider the data flow framework for this problem:

Flow Graph FG is the control flow graph, because Reaching Uses is a forward data flow problem. Similarly, *root*=`Entry`. DT and DF are in Figure 8.

Meet Semilattice L is a powerset semi-lattice. A, the domain of L, is the set of all subsets of N_{FG}. Here, an element of A represents the set of uses of variable v, indexed by the node in FG in which it appears, that can reach an edge in CFG. We take $\top$ to be the empty set, $\{\}$, and $\bot = N_{FG}$. The meet operator $\wedge$ is set union, $\cup$, and $a \preceq b$ iff $b \subseteq a$.

Transfer Functions With respect to variable v, the transfer function at node X is determined by the effect of node X on v:

Effect of Node X		Transfer Function
`Entry`		$f_X = \{\}$
	Kill v	$f_X = \{\}$
Use v	No kill of v	$f_X(IN) = \{X\} \cup IN$
No use of v	No kill of v	$f_X = \iota$

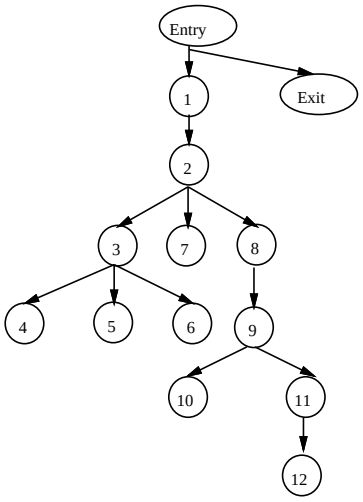
where *no kill* of v means no definition or a *preserving* definition of v. Notice that both uses and killing definitions have non-identity transfer functions.

Before invoking *Search* to compute reaching uses for `v`, the algorithm of Figure 3 performs the following initializations:

1. $N_{SG} = \{$`Entry`$, 4, 5, 7, 11, 12\}$. Node 11 uses v, while nodes 4, 5, 7, and 12 kills v.
2. $MeetNodes = \{2, 6, 8, 9,$ `Exit`$\}$, because 2 is in the dominance frontier of 12, 6 is in the dominance frontier of 4 and 5, 8 is in the dominance frontier of 6 and 7, 9 is in the dominance frontier of 11, and `Exit` is in the dominance frontier of 12.
3. $N_{SG} = \{$`Entry`$, 2, 4, 5, 6, 7, 8, 9, 11, 12,$ `Exit`$\}$

The *Search* procedure then begins at *root* = `Entry`. Figure 11 shows the resulting sparse evaluation graph. We give the final solution to the Reaching Uses problem in the following table.

Edges mapped to node	have solution
`Entry`	{}
2	{}
4	{}
5	{}
6	{}
7	{}
8	{}
9	{11}
11	{11}
12	{}



Dominator tree of CFG

Node	DF(Node)	Node	DF(Node)
1	Exit	7	8
2	Exit, 2	8	Exit, 2
3	8	9	Exit, 2, 9
4	6	10	11
5	8	11	Exit, 2, 9
6	8	12	Exit, 2

Dominance Frontier Table

Figure 8. Structures for solving forward problems for the control flow graph of Figure 4.

Meet Semilattice L is a powerset semi-lattice. A, the domain of L, is the set of all subsets of N_{FG}. Here, an element of A represents the set of definitions of variable v, indexed by the node in FG in which it appears, that can reach an edge in CFG. We take $\top$ to be the empty set, $\{\}$, and $\bot = N_{FG}$. The meet operator $\wedge$ is set union, $\cup$, and $a \preceq b$ iff $b \subseteq a$.

Transfer Functions With respect to variable v, the transfer function at node X is determined by the effect of node X on v:

Effect of Node X	Transfer Function
Entry	$f_X = \{\}$
Kill v	$f_X(IN) = \{X\}$
Preserve v	$f_X(IN) = \{X\} \cup IN$
No def of v	$f_X = \iota$

where, as before, *preserve* means that a variable is only potentially or partially modified, as in the assignment to an array element of `w` in Figure 4.

Before invoking *Search* to compute liveness for `w`, the algorithm of Figure 3 performs the following initializations:

1. $N_{SG} = \{$`Entry`$, 7\}$, because only those nodes have a non-identity transfer function for `w`.

Node visited	Link from	New TOS	Edges mapped	Link to
Entry		Entry	Entry → 1	
			Entry → Exit	Exit
1			1 → 2	2
2		2	2 → 3	
			2 → 7	
3			3 → 4	
			3 → 5	
4			4 → 6	
5			5 → 6	
6			6 → 8	8
7	2	7	7 → 8	8
8		8	8 → 9	
9			9 → 10	
			9 → 11	
10			10 → 11	
11			11 → 9	
			11 → 12	
12			12 → 2	2
			12 → Exit	
Exit		Exit		

Table 3. Reaching Defs for `w`

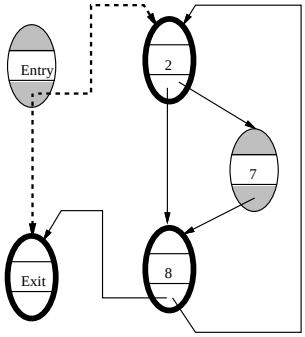


Figure 9. Sparse Graph for Reaching Defs (w)

2. $MeetNodes = \{$`Exit`$, 2, 8\}$, because 8 is in the dominance frontier of 7, and `Exit` and 2 are in the dominance frontier of 8.
3. $N_{SG} = \{$`Entry`$, 2, 7, 8,$ `Exit`$\}$

The *Search* procedure then begins at $root =$ `Entry`. Table 3 shows a possible visitation order for the nodes and the actions performed while visiting a given node. Note that the sparse evaluation graph produced, given in Figure 9, is cyclic. We give the final solution to the Reaching Definitions problem in the following table. Note that in evaluating the cyclic sparse graph to obtain the solution, only a single pass through the nodes is needed.

Edges mapped to node	have solution
Entry	{}
2	{7}
7	{7}
8	{7}

is applied directly to obtain $IN_{\texttt{Entry}} = \bot$.

The dashed edges in Figure 6 represent sparse graph evaluation edges *absent* from our graph due to constant transfer function optimization.

As each flow graph edge is listed in the "edges mapped" column, it is mapped to the node on top of the stack ("New TOS" column).

The sparse graph can now be evaluated by iteration, elimination, or any method of choice. Because the graph of Figure 6 contains no unoptimized edges, evaluation for this example is very fast. When node 2 was visited during sparse graph construction, IN_{12} became $\bot$ when 8 was linked to 12. The lowering of node 12's input causes evaluation of its transfer function to obtain $OUT_{12} = \bot$. Thus, the final solution in terms of the sparse graph is:

Edges mapped to node	have solution
8	Live
12	Live
`Exit`	Dead

For example, edge $6 \leftarrow 4$ is mapped to node 8, whose final solution is *Live*. Inspection of Figure 4 confirms that `w` is live this edge.

Now consider construction of a sparse evaluation graph to determine liveness for `v`. The steps shown in Table 2 result in the sparse graph of Figure 7.

The dashed edges indicate successful optimization around constant transfer functions. After graph construction, IN_3 and IN_2 have not changed from their initial value of $\top$. No propagation is necessary to achieve the final solution:

Node visited	Link from	New TOS	Edges mapped	Link to
`Exit`		`Exit`	`Exit` $\leftarrow$ 12	12
			`Exit` $\leftarrow$ `Entry`	`Entry`
`Entry`		`Entry`		
12		12	12 $\leftarrow$ 11	
11	12	11	11 $\leftarrow$ 10	
			11 $\leftarrow$ 9	
10			10 $\leftarrow$ 9	
9			9 $\leftarrow$ 8	
			9 $\leftarrow$ 11	
8			8 $\leftarrow$ 6	
			8 $\leftarrow$ 7	
7	11	7	7 $\leftarrow$ 2	2
2		2	2 $\leftarrow$ 1	
			2 $\leftarrow$ 12	12
1			1 $\leftarrow$ `Entry`	`Entry`
6		11	6 $\leftarrow$ 4	
			6 $\leftarrow$ 5	
3		3	3 $\leftarrow$ 2	2
4	11	4	4 $\leftarrow$ 3	3
5	11	5	5 $\leftarrow$ 3	3

Table 2. Liveness for `v`.

Edges mapped to node	have solution
2	Dead
3	Dead
4	Dead
5	Dead
7	Dead
11	Live
12	Dead
`Exit`	Dead

5 Reaching Definitions

In this section we sketch how the Reaching Definitions problem, a forward problem, can be solved with the same efficiency as the SSA-based method. We also show how to compute a variant of SSA form that contains no dead ϕ functions. This form is especially attractive for program comparison [YHR89].

Definition 4 *A definition of a variable v at node Z* reaches *a CFG edge $X \rightarrow Y$ if there exist*

- *a (possibly empty) path in CFG $Z \xrightarrow{*} X$, and*
- *no nodes on the path $Z \xrightarrow{*} X$ other than Z that* kill *v.*

We show how Reaching Definitions is solved for variable `w` in the control flow graph of Figure 4. Consider the data flow framework for this problem:

Flow Graph FG is the control flow graph, because Reaching Definitions is a forward data flow problem. Similarly, *root*=`Entry`. For the control flow graph in Figure 4, we show DT and DF in Figure 8.

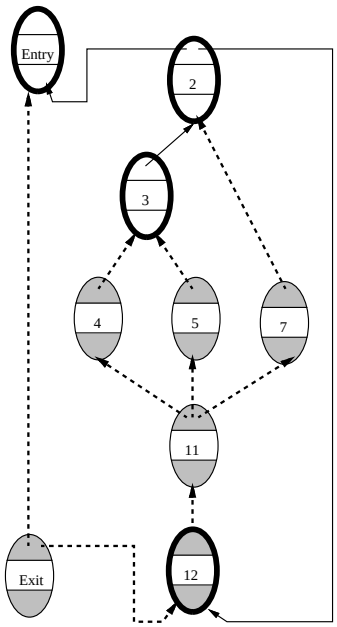


Figure 7. Sparse evaluation graph for Liveness(v).

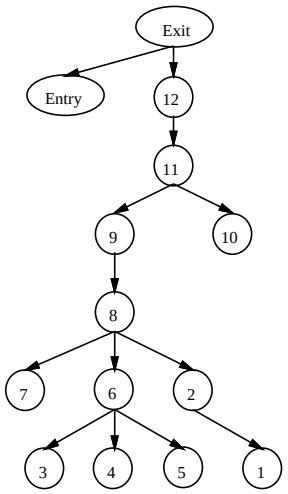
Effect of Node X		Transfer Function
Use v		$f_X = Live$
No use	Kill v	$f_X = Dead$
No use	Preserve v	$f_X = \iota$
No use	No def	$f_X = \iota$

where *preserve* means that a variable is only potentially or partially modified. For example, the assignment to the array `w` in Figure 4 modifies only a single element. With respect to the entire structure, the statement preserves `w`, and liveness of `w` is not affected.

Before invoking *Search* to compute liveness for `w`, the algorithm of Figure 3 performs the following initializations:

1. $N_{SG} = \{\texttt{Exit}, 8\}$, because only node 8 has a non-identity transfer function for `w`.
2. $MeetNodes = \{\texttt{Entry}, 12\}$, because 12 is in the dominance frontier of 8, and `Entry` is in the dominance frontier of 8.
3. $N_{SG} = \{\texttt{Entry}, 8, 12, \texttt{Exit}\}$

The *Search* procedure then begins at $root =$ `Exit`. Table 1 shows a possible visitation order for the nodes.



Dominator Tree of Reverse CFG

Node	DF(Node)	Node	DF(Node)
1	Entry	7	2
2	Entry, 12	8	Entry, 12
3	2	9	Entry, 11, 12
4	3	10	9
5	3	11	Entry, 11, 12
6	2	12	Entry, 12

Dominance Frontier Table

Figure 5. Structures for solving backward problems for the control flow graph of Figure 4.

Node visited	Link from	New TOS	Edges mapped	Link to
`Exit`		`Exit`	`Exit` $\leftarrow$ 12 `Exit` $\leftarrow$ `Entry`	12 `Entry`
`Entry`		`Entry`		
12		12	12 $\leftarrow$ 11	
11			11 $\leftarrow$ 10 11 $\leftarrow$ 9	
10			10 $\leftarrow$ 9	
9			9 $\leftarrow$ 8 9 $\leftarrow$ 11	
8	12	8	8 $\leftarrow$ 6 8 $\leftarrow$ 7	
7			7 $\leftarrow$ 2	
2			2 $\leftarrow$ 1 2 $\leftarrow$ 12	 12
1			1 $\leftarrow$ `Entry`	`Entry`
6			6 $\leftarrow$ 4 6 $\leftarrow$ 5	
3			3 $\leftarrow$ 2	
4			4 $\leftarrow$ 3	
5			5 $\leftarrow$ 3	

Table 1. Liveness for `w`.

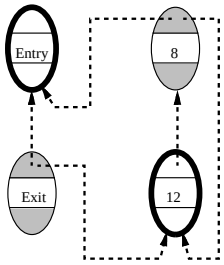


Figure 6. Sparse evaluation graph for Liveness(w). Bold ellipses are in $MeetNodes$. Shaded ellipses have a non-identity transfer function.

The actions performed while visiting a given node are shown in the other columns of this table.

The two "link" columns refer to edges added to the sparse graph of Figure 6 by the algorithm in Figure 3:

Link from shows edges added by step 1. Consider the visit of sparse node 8. Because 8 is not a meet node, the input for its transfer function is the output of whatever node was on top of the stack when node 8 is visited. Therefore, an edge should be added to the sparse graph from 12 to 8. Because node 8 has a constant transfer function, this edge is not necessary.

Link to shows edges added by step 3b. The algorithm considers CFG edge 1 $\leftarrow$ `Entry` while visiting node 1. The meet contribution at `Entry` due to this edge is the output of whatever node is on top of the stack when this edge is considered. Therefore, an edge should be added from 8 to the meet input of `Entry`. Because the output of 8 is constant, its value of $\perp$

denotes the size of the set or relation A. Performing the recursive calls to *Search* is in total

$$O(\mid MeetNodes \mid + \mid DF \mid) = O(E + N^2)$$

and thus the total time and space cost of the algorithm is $O(E + N^2)$. Since for many problems $\mid MeetNodes \mid$ is small and and $\mid DF \mid$ is expected to be linear for real programs, we expect the computation to be linear in many cases.

We now consider the relationship of this work with control dependence. Recall that for forward data flow problems, the flow graph is the control flow graph, and for backward problems, the flow graph is the *reverse* control flow graph. When presented with this flow graph, the first step of our algorithm generates the dominance frontier graph. Previous work has shown the relationship between dominance frontiers and the *control dependence graph* [FOW87]: the control dependence graph is the dominance frontier graph of the *reverse* control flow graph. Thus, for backward data flow problems, the first step of our algorithm generates the control dependence graph for the given control flow graph. If node Y in CFG has a non-identity transfer function, the first step of our algorithm makes the control dependence predecessor of Y a meet node. We give an intuitive explanation of why this relationship holds.

Informally, for nodes X and Y in CFG, Y is control dependent on X if there is some edge $X \rightarrow Z$ such that Y must be on any path $X \rightarrow Z \xrightarrow{*} \texttt{Exit}$, and there is some other path

$$p : X \xrightarrow{+} \texttt{Exit}$$

that avoids Y. (A formal definition can be found in [FOW87].) Assume Y is control dependent on X. Given a path

$$q : X \rightarrow Z \xrightarrow{*} Y$$

p and q have no nodes in common except X. This follows since if they had a node $W \neq Y$ in common, the path $X \rightarrow Z \xrightarrow{*} W \xrightarrow{*} \texttt{Exit}$ would avoid Y, thereby contradicting the definition of control dependence.

Now suppose Y has a non-identity transfer function. Since `Exit` is *root* for backward problems, it is necessary for information originating at Y and `Exit` to combine at some meet node. Given that these paths p and q have no node in common, X is the first node where information from `Exit` and information from Y could meet. Therefore, it makes sense to make X, the control dependence predecessor of Y, a meet node.

Each of the next three sections presents and solves a data flow problem using our method. We begin with *Live Variables* to show how backward problems are solved. We then present *Reaching Definitions*, which is a forward problem that could be easily solved using SSA form. We then show how to compute *Reaching Uses*, which is a forward problem that cannot be solved using def-use or SSA form.

4 Live Variables

In this section we show how our method accommodates the *Live Variables* problem [ASU86], which is not readily solvable using SSA form:

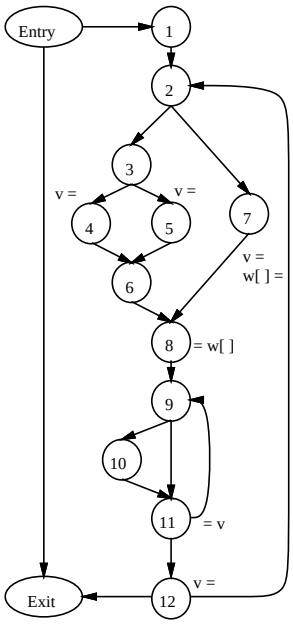


Figure 4. Control Flow Graph

Definition 3 *A variable v is* live *on CFG edge $X \rightarrow Y$ if there exist*

- *a node Z that uses v,*
- *a (possibly empty) path in CFG $Y \xrightarrow{*} Z$, and*
- *no nodes on the path $Y \xrightarrow{*} Z$ that* kill *v.*

This problem is a well-known backward data flow problem, whose solution is not simplified for programs in SSA form. Programs in SSA form could be analyzed for liveness using traditional methods. However, the ϕ-functions can actually *hinder* such analysis. In Figure 4, ϕ-function assignments would be placed at node 6, 8, and 2 due to the definitions of v. If treated as ordinary assignments, each ϕ-function assignment is a *kill* and a use of v, thereby obscuring the liveness properties of the original assignments.

We now show how Live Variables is solved for variables `v` and `w` in the control flow graph of Figure 4. Consider the data flow framework for this problem:

Flow Graph FG is the reverse control flow graph, because Live Variables is a backward data flow problem. Similarly, *root*=`Exit`, and DT and DF are computed with respect to the reverse control flow graph, as shown in Figure 5.

Meet Semilattice L is a two-level lattice, where $\top$ represents *Dead* and $\bot$ represents *Live*. The meet operator $\wedge$ is therefore implicitly defined:

$\wedge$	*Dead* ($\top$)	*Live* ($\bot$)
Dead ($\top$)	*Dead*	*Live*
Live ($\bot$)	*Live*	*Live*

Transfer Functions With respect to variable v, the transfer function at node X is determined by the effect of node X on v:

of X, and therefore in the sparse graph. By the induction hypothesis, Z is the first such node. By the definition of dominance frontiers, X must dominate Y, causing step 3b to add the edge (X, Z) to the sparse graph.

□

Lemma 2 *If P_{FG} and P_{SG} are associated paths from root through the flow and sparse graphs, respectively, then for any $l \in L$,*

$$f^{P_{SG}}(l) = f^{P_{FG}}(l)$$

Proof:

$f^{P_{SG}}(l) \preceq f^{P_{FG}}(l)$: Both paths begin with the *root* node, whose transfer function produces $\top$ for any input. By the construction given in Definition 1 P_{SG} omits only nodes from P_{FG} whose transfer functions are identity. Such nodes cannot contribute to the solution for P_{FG}.

$f^{P_{FG}}(l) \preceq f^{P_{SG}}(l)$: We must prove that there are no spurious paths in the sparse graph. Edges are only added by steps 1 and 3b in Figure 3. By Lemma 1, each of these cases corresponds to a path in the flow graph.

□

Theorem 1 *Function composition along any path in FG produces the same result as function composition along the corresponding path in SG.*

Proof: Consider the flow graph path $X_{FG} \xrightarrow[FG]{*} Z$ and its corresponding sparse graph path $X_{SG} \xrightarrow[SG]{*} Y$. Without loss of generality, assume $X_{FG} = X_{SG} = X$. From Lemma 2, we know that composition along the path $root \xrightarrow[FG]{*} X \xrightarrow[FG]{*} Z$ produces the same result as composition along $root \xrightarrow[SG]{*} X \xrightarrow[SG]{*} Y$. Composition along the path prior to X can produce *any* lattice element. Therefore, compositional equality must be preserved by the path from X. □

Lemma 3 *Let $X \neq Y$ be two nodes in the flow graph. If the nonnull flow graph paths $X \xrightarrow[FG]{+} Z$ and $Y \xrightarrow[FG]{+} Z$ have Z as their first node in common, then $Z \in MeetNodes$ as constructed by step 2 of the* SparseGraph *algorithm.*

Proof: Follows from Lemma 5 in [CFR*89a].

Theorem 2 *Any combination of function composition and meets using paths in the flow graph produces the same result as composition and meets using the associated paths in the sparse graph.*

Proof: Consider two flow graph paths with node Z as their first node in common. If the two paths have identity transference, then evaluation is trivially the same. If not, then some node has non-identity transference, and by Theorem 1 we have compositional equivalence along the associated paths. By Lemma 3, we know the meet will occur at $Z \in N_{SG}$. □

We turn to our treatment of constant transfer functions by steps 1 and 2a of procedure $Link(Z)$. We show that step 1 does not affect any manner of composition or meet in evaluating the sparse graph. However, step 2a may affect an evaluation whose solution is more accurate than best, generally decidable solution MFP.

Definition 2 *The* relevant suffix *of a sparse graph path $P_{SG} = root \xrightarrow[SG]{*} Y$ is the path $K \xrightarrow[SG]{*} Y$, where K is the closest node to Y in the path with constant transference.*

A relevant suffix always exists for a path from *root*, because *root* has constant transference.

Lemma 4 *Optimization by step 1 does not affect sparse graph evaluation.*

Proof: Function composition prior to K is ignored by the transfer function at K, so that function composition along the relevant suffix of a sparse graph path produces the same result as function composition along the complete path. The only effect of step 1 is to eliminate edges from the sparse graph that precede a relevant suffix. Therefore, Theorem 2 holds even if such edges are missing in the sparse graph. □

We next consider optimization by step 2a, which eliminates an edge (K, Z) from the sparse graph in favor of directly forwarding constant information from K to Z. If node Z has constant transference, then optimization by step 1 applies, so we may assume nonconstant transference at Z. In general, we cannot prove equivalence of evaluation. Suppose node Z has predecessors K_1 and K_2 in the sparse graph, where both K_1 and K_2 have constant transference: K_1 transfers $\bot$ and K_2 transfers $\top$. After optimized graph construction, information from both nodes is already incorporated into IN_Z, so that $IN_Z = \bot$. Evaluation along the path $root \xrightarrow[SG]{*} K_2 \xrightarrow[SG]{} Z$ may produce a more conservative answer (closer to $\bot$) than evaluation along this path in the unoptimized sparse graph. In general, when the best computable solution is MFP convergence, this discrepancy is not a problem:

Lemma 5 *Optimization due to step 2a does not affect the MFP solution.*

Proof: The meet operator is by definition commutative and associative. Thus, predecessors of Y can be considered in any order in forming IN_Y. Because IN_Y starts at $\top$, optimization by step 2a cannot bring IN_Y any closer to $\bot$ than it would reach during maximum fixed point evaluation. □

We now consider the time and space complexity of computing the sparse data flow evaluation graph. We assumed the dominator tree and the dominance frontier relation DF were precomputed, which requires in total time and space $O(E + N^2)$. The time and space to carry out the algorithm in Figure 3 is dominated by the computation of $MeetNodes$ and the recursive calls to $Search$. Computing $MeetNodes$ can be done at worst in time and space $O(\mid DF \mid) = O(E + N^2)$, where $\mid A \mid$

Algorithm $SparseGraph(\mathcal{D}, DT, DF)$:

Input: Data flow framework $\mathcal{D}$, dominator tree DT, dominance frontier DF

Output: Sparse graph $SG =< N_{SG}, E_{SG} >$ and mapping function $M : E_{FG} \mapsto N_{SG}$

Steps:

1. $N_{SG} = \{root\} \cup \{ Y \mid f_Y \neq \iota \}$
2. $MeetNodes = \bigcup_{X \in N_{SG}} DF^+(X)$
3. $N_{SG} = N_{SG} \cup MeetNodes$
4. $\forall Y \in N_{SG},\ IN_Y = \top$
5. $Stack = empty$, with operators $push$ and pop, and top of stack pointer TOS.
6. $Search(root)$

Procedure $Search(Y)$:

1. if $Y \notin MeetNodes$ then $Link(Y)$
2. if $Y \in N_{SG}$ then $push(Y)$
3. for each $Z \in Succ(Y)$
   (a) $M((Y,Z)) = TOS$
   (b) if $Z \in MeetNodes$ then $Link(Z)$
4. for each child C of Y in DT do $Search(C)$
5. if $Y \in N_{SG}$ then $pop(Y)$

Procedure $Link(Z)$:

if $Z \in N_{SG} - \{root\}$ then
1. if $f_Z = K_Z$ then
   /* No Edge Added */
2. else
   (a) if $f_{TOS} = K_{TOS}$ then
   $IN_Z = IN_Z \wedge K_{TOS}$
   (b) else
   $E_{SG} = E_{SG} \cup (TOS, Z)$

Figure 3. Algorithm to Generate Sparse Evaluation Graph

3.2 Analysis

To establish the correctness of our algorithm, we must prove that a monotone data flow evaluation of the sparse graph produces the same result as evaluation of the flow graph. We first prove correctness assuming no optimization due to constant transfer functions, so that procedure $Link(Z)$ always adds the edge (TOS, Z) to E_{SG} if $Z \in N_{SG} - \{root\}$, in the following manner:

1. For any flow graph path $root \xrightarrow[FG]{*} Y$, we define and prove the existence of an associated sparse graph path $root \xrightarrow[SG]{*} X$.
2. Consider transfer function composition along a flow graph path P_{FG} and its associated sparse graph path P_{SG}, which we write as $f^{P_{FG}}$ and $f^{P_{SG}}$, respectively. We prove that
$$\forall l \in L, f^{P_{SG}}(l) \preceq f^{P_{FG}}(l)$$
so that the fixed point achieved by evaluation of the sparse graph produces at least as conservative a result as the same evaluation of the flow graph.
3. We prove that each path P_{SG} has at least one corresponding path P_{FG}, so that
$$\forall l \in L, f^{P_{FG}}(l) \preceq f^{P_{SG}}(l)$$
Therefore, evaluation of the sparse graph produces the same result as evaluation of the flow graph.

We subsequently show that our treatment of constant transfer functions does not affect evaluation of the sparse graph.

The sparse graph path corresponding to a given flow graph path is just the flow graph path with non-sparse nodes deleted:

Definition 1 *The sparse graph path P_{SG} associated with a given flow graph path $P_{FG} = root \xrightarrow[FG]{*} Y$ is constructed as follows:*

1. *If $P_{FG} = root$, then $P_{SG} = root$.*
2. *Suppose $P_{FG} = root \xrightarrow[FG]{*} Y \xrightarrow[FG]{} Z$ and the sparse path associated with $root \xrightarrow[FG]{*} Y$ is $root \xrightarrow[SG]{*} X$. Then*
$$P_{SG} = \begin{cases} root \xrightarrow[SG]{*} X \xrightarrow[SG]{} Z & \text{if } Z \in N_{SG} \\ root \xrightarrow[SG]{*} X & \text{otherwise} \end{cases}$$

Lemma 1 *If P_{FG} is a path in the flow graph, then the construction of P_{SG} by Definition 1 yields a path in the sparse graph.*

Proof: We use induction along the path P_{FG}. The path $root$ is a valid path in both graphs, because $root$ is always included in N_{SG}. As induction hypothesis, assume the flow graph path $root \xrightarrow[FG]{+} X \xrightarrow[FG]{*} Y$ has a valid sparse graph path $root \xrightarrow[SG]{*} X$, so that all nodes added to the flow graph path after X do not affect the sparse graph path. Now consider extending the flow graph path by one node Z to $root \xrightarrow[FG]{+} X \xrightarrow[FG]{*} Y \xrightarrow[FG]{} Z$. By Definition 1, the resulting sparse graph path either remains $root \xrightarrow[SG]{*} X$, or else the sparse graph path is extended by Z if $Z \in N_{SG}$. In this case, we must prove $(X, Z) \in E_{SG}$. There are two possibilities:

X dominates Z: The path $root \xrightarrow[FG]{+} X \xrightarrow[FG]{*} Y \xrightarrow[FG]{} Z$ could not avoid any dominator of Z. By the induction hypothesis, nodes between X and Z are not in the sparse graph. Therefore, X must be the closest proper ancestor of Z in the dominator tree that is included in the sparse graph. Thus, $Z \neq root$ and step 1 adds the edge (X, Z) to the sparse graph.

X does not dominate Z: In this case, some node on the path after X must be in the dominance frontier

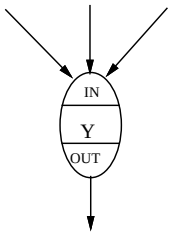


Figure 2. Data flow effect of a node.

which we call the MFP solution. During such evaluation, IN_Y travels down the lattice from $\top$ to the element that represents the best computable solution prior to Y, regardless of flow path taken. The view from a particular node Y is shown in Figure 2.

In traditional (bit-vectoring) graphs, edges of the flow graph retain the best available solution. Thus, the representative lattice element for IN_Y is

$$IN_Y \equiv \bigwedge_{X \in Pred(Y)} OUT_X$$

and the number of lattice elements combined by the meet at node Y is limited to the in-degree of Y in FG. Direct-connection methods compute IN_Y by identifying nodes that affect the solution prior to Y:

$$IN_Y \equiv \bigwedge_{X \in Affects(Y)} OUT_X$$

The expense of this meet could be proportional to the number of nodes in CFG.

In our algorithm, we are interested in two common types of transfer functions:

Identity If the transfer function at node Y satisfies

$$\forall\, l \in L, f_Y(l) = l$$

then we say node Y has *identity transference* and we write

$$f_Y = \iota$$

where ι is the identity function. Transfer functions at such nodes cannot affect data flow information.

Constant If the transfer function at node Y satisfies

$$\exists\, k \in L \text{ such that } \forall\, l \in L, f_Y(l) = k$$

then we say node Y has *constant transference* and we write

$$f_Y = K_Y$$

The transfer function is single-valued and therefore independent of any other node's solution. An example of a constant transfer function that occurs in all frameworks is

$$f_{root} = K_{root} = \top$$

3 Sparse Graph Construction

Given a monotone data flow framework, we now show how to construct a sparse evaluation graph:

- Its nodes are flow graph nodes that *generate* information for the data flow problem, or represent the earliest node where new information must be *combined*.
- Its edges directly propagate information to nodes that *use* or *combine* the information.
- If a node transfers "constant" information, then no information is combined at its input and its output is directly propagated.

3.1 Algorithm

Our algorithm requires prior computation of the following structures:

Dominator tree DT of FG is computed in $O(E\alpha(E))$ time [LT79]. Node X dominates node Y if X is an ancestor of Y in the dominator tree; X *strictly* dominates Y if X is a *proper* ancestor of Y.

Dominance frontier [CFR*89a] DF of FG is computed in time $O(E + N^2)$ for arbitrary FG, but only linear time is required for programs restricted to `do-while` and `if-then-else` branching. The set nodes in the dominance frontier of X is

$$DF(X) = \{\, Z \mid X \text{ dominates some } Y \in Preds(Z) \text{ but } X \text{ does not strictly dominate } Z \,\}$$

We write $DF^+(X)$ as the *iterated* dominance frontier of X, which is the limit of the sequence:

$$\begin{aligned} DF_1 &= DF(X) \\ DF_{i+1} &= DF_i \cup \bigcup_{Z \in DF_i} DF(Z) \end{aligned}$$

These structures depend only on FG and not on any other aspect of the data flow framework. Once DT and DF have been computed for CFG to solve a particular forward (backward) data flow problem, subsequent data flow problems do not require their recomputation.

The algorithm for constructing the sparse graph is shown in Figure 3. Intuitively, nodes with identity transference are not included in the sparse graph at step 1. Some of these nodes may be included in the sparse graph by step 2 because information must be combined at such nodes.

The *Link* procedure is called when edges might be required in the sparse graph. The procedure tries two optimizations related to constant transference to eliminate edges from the sparse graph. Proofs concerning the properties of this optimization are found in Section 3.2: the optimization performed at step 1 is valid for any evaluation of the sparse graph, while the optimization at step 2a is valid only for the prevalent maximum fixed point evaluation.

bit-vectors are overly consumptive of space. Moreover, propagation occurs *throughout* a graph, sometimes in regions that neither affect nor care about the solution. Consider constant propagation for `x` in Figure 1. Information associated with the two definitions of `x` initially propagates through nodes and edges uninterested in the value of `x`. Eventually, the two definitions "meet" at node A, and the combined information is then propagated to the uses of `x`. Our method directly connects the definitions of `x` to node A, thereby avoiding superfluous nodes.

The other prevalent solution method uses *direct-connections* that shorten the propagation distance between nodes that generate and use data flow information. Such solutions are typically based on *def-use chains* [ASU86], shown in Figure 1 by the non-solid lines. In this example, direct connections unfortunately require combining the same information at each use of `x`, rather than just once at node A. In the worst case, a quadratic number of "meets" can occur where a linear number suffices. Experiments show that such quadratic behavior is noticeable especially for arrays, aliased variables, and variables modified at procedure call sites [CCF]. Def-use propagation based on SSA form [CFR*89a, CFR*89b] or its precursors [RL77, RT82] can usually avoid this expense by combining information as early as possible. However, if def-use chains are explicitly computed by solving *Reaching Definitions* and *Live Variables*, then bit vectors would still be required. Moreover, our method is applicable to all monotone data flow problems, and not just those based on def-use chains.

Section 2 contains a brief summary of data flow frameworks. Section 3 presents and analyzes the sparse graph construction algorithm. We also show how the control dependence graph is related to solving backward data flow problems. In Section 4, we apply our method to solve the backward data flow problem of *Live Variables*, which has no known solution in SSA form. In Section 5, we turn to the forward problem of *Reaching Definitions*. We also show how to construct the *pruned* SSA form conducive for program comparison. In Section 6, we show how to compute the use-def chains from which anti-dependences are tested. Although this is a forward data flow problem, use-def chains are not easily obtained using SSA form. Finally, we discuss conclusions and open problems in Section 7.

2 Background

The statements of a program are organized into (not necessarily maximal) basic blocks, where program flow enters a basic block at its first statement and leaves the basic block at its last statement. A *control flow graph* is a directed graph $CFG =< N_{CFG}, E_{CFG}, \texttt{Entry} >$. The nodes N_{CFG} are the basic blocks of a program and two additional nodes, `Entry` and `Exit`. The edges E_{CFG} represent transfers of control (jumps) between the basic blocks. Also, there is an edge from `Entry` to any basic block at which the program can be entered, and there is an edge to `Exit` from any basic block that can exit the program. For reasons explained elsewhere [CFR*89a], there is also an edge from `Entry` to `Exit`. We assume that each node is on a path from `Entry` and on a path to `Exit`. If (Y, Z) is an edge of CFG, then we write $Y \rightarrow Z$. Because we are concerned with backward as well as forward data flow problems, we write $Z \leftarrow Y$ to indicate that information flows from Z to Y even though the control flow graph edge is oriented from Y to Z.

The input to our algorithm is a data flow framework [Mar89] $\mathcal{D} =< FG, L, \mathcal{F} >$:

Flow Graph $FG =< N_{CFG}, E_{FG}, root >$ has the same nodes as CFG but its edges are oriented in the direction of the data flow problem. For forward flow problems, $FG = CFG$ and $root =$ `Entry`, but for backward flow problems

$$E_{FG} = \{ (Z, Y) \mid (Y, Z) \in E_{CFG} \}$$

and $root =$ `Exit`. We define successors and predecessors with respect to the direction of flow:

$$\begin{aligned} Succs(Y) &= \{ Z \mid (Y, Z) \in E_{FG} \} \\ Preds(Y) &= \{ X \mid (X, Y) \in E_{FG} \} \end{aligned}$$

Meet Semilattice $L =< A, \top, \bot, \preceq, \wedge >$, where

- A is a set, 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 $a, b \in A$,

$$\begin{aligned} a \preceq b &\iff a \wedge b = a \\ a \wedge a &= a \\ a \wedge b &\preceq a \\ a \wedge \top &= a \\ a \wedge \bot &= \bot \end{aligned}$$

Transfer Functions $\mathcal{F}$ is a set of functions

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

containing the identity function ι and closed under composition and pointwise meet. The data flow effect of node Y is described by its transfer function $f_Y \in \mathcal{F}$. Because our control flow graph is augmented with `Entry` and `Exit`, we can assume

$$f_{root} = \top$$

The local properties of node Y are captured by its transfer function:

$$OUT_Y = f_Y(IN_Y), where\ IN_Y, OUT_Y \in L$$

After a framework has been globally *evaluated*, each node Y has a solution OUT_Y that is consistent with transfer functions at every node. In general, the best computable solution for a data flow framework is the maximum fixed point convergence of the equations [Hec77, Ken81, Ros81]:

$$\begin{aligned} OUT_{root} &= \top \\ IN_Y &= \bigwedge_{X \in Preds(Y)} OUT_X \\ OUT_Y &= f_Y(IN_Y) \end{aligned}$$

Automatic Construction of Sparse Data Flow Evaluation Graphs

Jong-Deok Choi*
Ron Cytron*
Jeanne Ferrante*

Abstract

In this paper, we present an algorithm that constructs sparse evaluation graphs for forward or backward monotone data flow problems. The sparse graph combines information as early as possible, yet directly connects nodes that generate and use information. This allows problems from the large, general class of monotone data flow problems to enjoy the advantages of solutions based on Static Single Assignment (SSA) form.

1 Introduction

Compiler optimization problems are typically formulated as data flow frameworks, in which the solution of a given problem at a given program point is related to the solution at other points [Ros79, Tar81]. The quality and speed of evaluating these frameworks are well-understood, and data flow methods are understandably prevalent in most optimizing compilers. Unfortunately, propagation methods commonly used in data flow evaluation are unduly inefficient with respect to time or space.

Static Single Assignment (SSA) Form [CFR*89a, CFR*89b] has recently yielded more efficient and powerful solutions for data flow problems like constant propagation [WZ85], global value numbering [AWZ88], redundancy elimination [RWZ88], and invariance detection [CLZ86]. Once programs are cast into SSA form, data flow solutions for these problems have the following advantages:

1. Information is combined as early as possible.
2. Information is forwarded directly to where it is needed.
3. Useless information is not represented.

These advantages follow from the way *definitions* are connected to *uses* in a program. The extant SSA-based data flow solutions essentially use SSA form as a sparse evaluation graph that embodies this connection. Unfortunately, SSA form does not appear to be sufficiently general to afford an efficient solution for problems *not* based on this connection, such as *Live Variables*.
This paper contains the following results:

1. We show how to construct a sparse data flow evaluation graph for forward or backward monotone data flow problems. Once evaluated, this graph provides solutions at these representative sparse nodes. To determine solutions throughout the flow graph, the algorithm constructs the required mapping from flow graph edges to the sparse nodes.
2. The control dependence graph [FOW87] serves as the basis for computing the sparse evaluation graph for backward problems.
3. We show how to construct *pruned* SSA form, which contains no dead ϕ-functions. This form is particularly useful for program comparison [YHR89], yet previous attempts to efficiently eliminate dead ϕ-functions were restricted to special program structures.

Previous methods for solving data flow problems fall into one of two categories. Traditional bit-vectoring methods [Kil73] propagate the solution at a given node to the control flow graph successors or predecessors of that node. Compiler writers generally acknowledge that

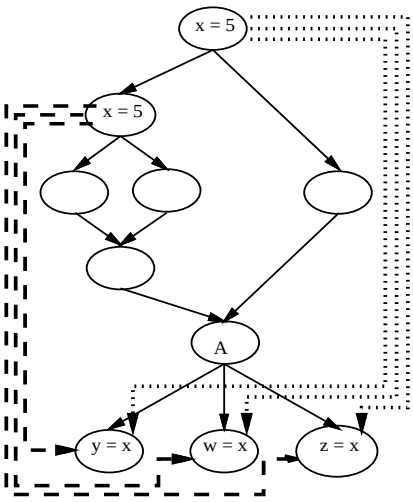


Figure 1. Flow graph for Constant Propagation. Solid lines represent control flow edges; non-solid lines represent def-use chains.

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.