McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

An Elimination-Based Approach to Incremental Data Flow Analysis

Vugranam C. Sreedhar
Guang R. Gao

ACAPS Technical Memo 94
June 21, 1995

A version of this report has been submitted for publication

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7
FTP: ftp-acaps.cs.mcgill.ca • WWW: http://www-acaps.cs.mcgill.ca/

Abstract

In this paper we present a new approach for incremental data flow analysis based on elimination methods. Unlike previous elimination-based incremental data flow analysis, our approach can handle arbitrary non-structural and structural changes to program flowgraphs, including irreducibility. We use properties of dominance frontiers and iterated dominance frontiers for updating data flow solutions. These properties are applicable to both reducible and irreducible flowgraphs. Since we use properties of dominance frontiers (and iterated dominance frontiers) we also give a simple algorithm for updating the dominance frontier relation. The dominance frontier update algorithm is based on our previous work for updating dominator trees.

Contents

List of Figures

1 Introduction

There are two classical approaches to incremental data flow analysis: one based on iteration methods [PS89], and another based on elimination methods [RP88, CR88]. Marlowe and others have extensively studied the relative merits of one approach over the other [Mar89, RMP88, MR90, BR90]. Marlowe has also proposed a hybrid scheme that combines the two approaches [MR90, Mar89].

In this paper we propose a new approach for incremental data flow analysis that is based on our eager elimination method [SGL95a]. Compared to previous elimination-based incremental data flow analyses, our approach can handle arbitrary non-structural and structural program changes, including irreducibility. A novel aspect of our approach is that we use simple properties of dominance frontiers and iterated dominance frontiers for updating the data flow solutions. In [SGL95b] we showed how to use such properties in the context of incremental dominator tree update problem. In this paper we will go one step further and show how to exploit them in the context of incremental data flow analysis.

In a companion paper we proposed a new approach for elimination-based data flow analysis that uses DJ graphs for reduction and variable elimination [SGL95a]. We proposed two variations of our approach: (1) eager elimination method, and (2) delayed elimination method. Both approaches perform reduction and variable elimination on DJ graphs in a bottom-up fashion, ordered by the levels of the nodes on the dominator tree. Unlike the eager elimination method, the delayed elimination method also compresses the dominator tree to improve the worst-case time complexity of the eager elimination method. In this paper we show how to incrementalize our eager elimination method.

To summarize the major features of our algorithm are as follows:

- Unlike many of the previous incremental elimination algorithms, our algorithm can handle arbitrary non-structural and structural program changes, including irreducibility.
- We use simple properties of dominance frontiers and iterated dominance frontiers for updating the data flow solutions. These properties were introduced for constructing Static Single Assignment form and other other Sparse Evaluation Graphs [CFR$^+$91, CCF91, SG95]. In this paper we will show how to exploit them in the context of incremental data flow analysis. To the best of our knowledge this is the first paper that uses such properties in the context of incremental data flow analysis.
- Since our incremental algorithm uses properties of dominance frontiers for updating data flow solutions, we also give a simple incremental algorithm for maintaining the dominance frontier relation.

The work that is most closely related to ours is due to Carroll and Ryder [CR88, Car88]. They use concepts from incremental attributed grammars to update the data flow solutions.

Unlike our algorithm, their algorithm is restricted to only reducible flowgraphs. In this paper we will also show how our algorithm is related to some of the key concepts used in incremental attributed tree problem.

Organization: The rest of the paper is organized as follows. The next section outlines our eager elimination method. In Section 3 we lay the foundation of our approach for incremental data flow analysis. In this section we will introduce the concept of initial and final flow equation, and DF graphs that are fundamental to our approach. In Section 4 we will show how to update the final flow equations for non-structural changes. In Section 5 we will handle structural changes. Here we will also give an algorithm for updating the dominance frontier relation. In Section 6 we will show how to update the final data flow solutions for both structural and non-structural changes. In Section 7 we will prove the correctness of our approach and also analysis its time complexity. In Section 8 we will give a detailed comparison of our approach with Carroll and Ryder's approach. Here we will also show how dominance frontiers and iterated dominance frontiers are intrinsically related to concepts from attributed grammars. Finally, we will conclude in Section 9. Appendix A gives the relevant background material and notation used in the paper.

2 Exhaustive Elimination Method: An Overview

In this section we briefly review our exhaustive eager elimination method [SGL95a]. To simplify the presentation we will initially assume that the flowgraph is reducible, and later show how to handle irreducible flowgraphs. Our exhaustive eager elimination uses DJ graph as its main data structure for reduction and variable elimination. A DJ graph is nothing but the dominator tree of a flowgraph augmented with join edges. An edge $x \rightarrow y$ in a flowgraph is a join edge if x does not immediately dominate y. Figure 1 shows an example of a flowgraph, and the corresponding dominator tree and DJ graph.

Our exhaustive eager elimination method consists of three phases:

1. Reduce the DJ graph to its dominator tree in a bottom-up fashion using $\mathcal{E}$-$rules$.

2. Reduce the system of data flow equations by eliminating variables.

3. Propagate the final data flow solutions in a top-down manner on the dominator tree.

Rather than reducing a DJ graph to a single node we only eliminate J edges in a bottom-up fashion, preserving the structure of the dominator tree for the entire duration of the algorithm. During the bottom-up reduction we apply the $\mathcal{E}$-$rules$ for eliminating flow variables and reducing the DJ graph to its dominator tree. The $\mathcal{E}$-$rules$ are always applied to a J edge $y \rightarrow z$ such that y is a *non-join* node, and there are no other J edges whose source node is greater than

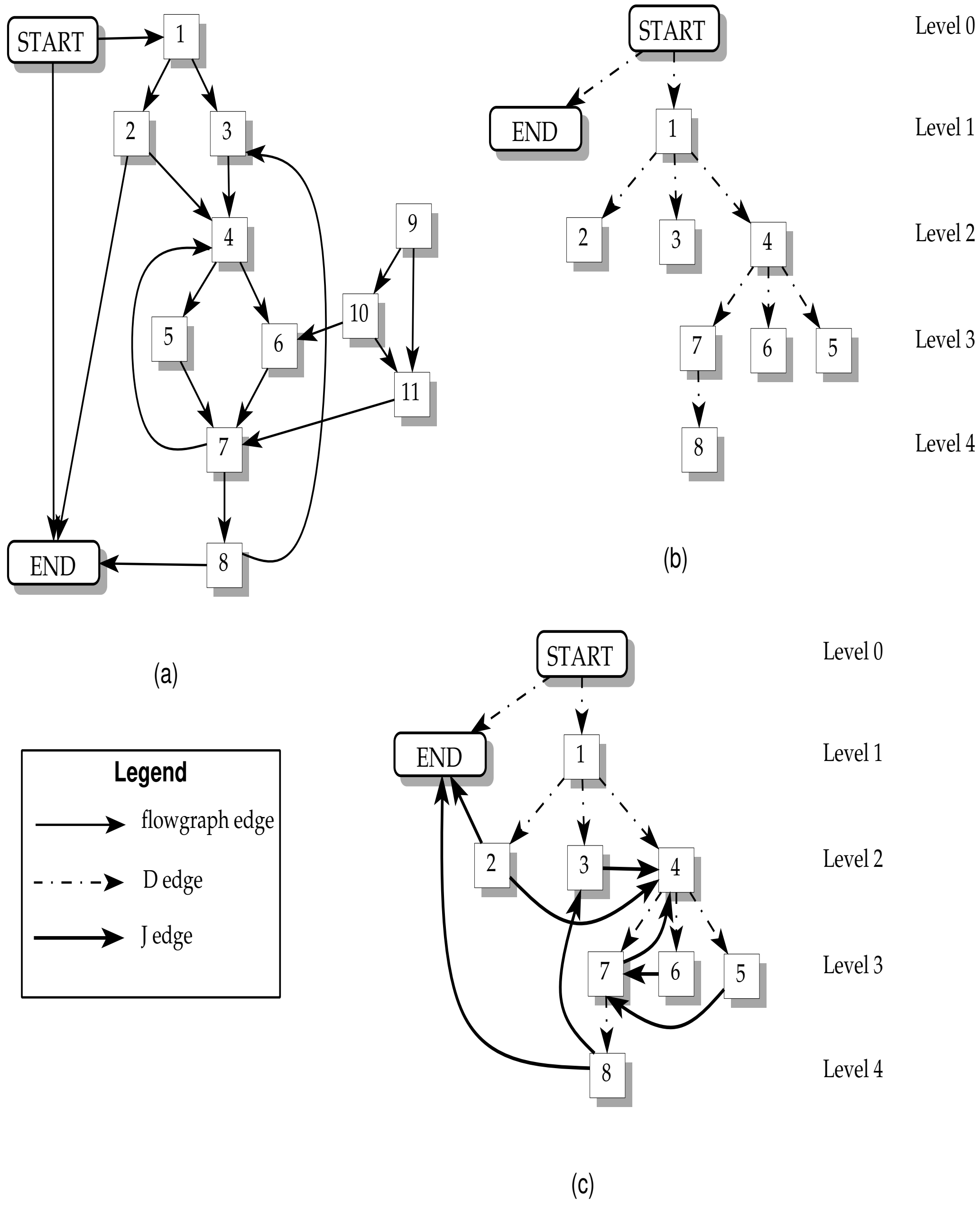

Figure 1: A flowgraph and its dominator tree and DJ graph

$y.level$, the level number of node y. Each application of $\mathcal{E}$-*rules* transforms some reduced DJ graph $\mathcal{G}^i$ to $\mathcal{G}^{i+1}$, until the DJ graph reduces to its dominator tree.

The E1 rule eliminates a self-loop, and computes closure of its recursive equation.

Definition 2.1 (E1 rule) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node such that y contains a self-loop. Let $H_y : O_y = f(O_y)$ be the flow equation at node y. The E1 rule is given below:*

- (i) Graph reduction*:*

$$E1(< \mathcal{G}^i, N, E, y \xrightarrow{J} y >) \ = < \mathcal{G}^{(i+1)}, N, E - \{y \xrightarrow{J} y\} >$$

- (ii) Variable elimination*:*

$$E1(< H_y : O_y = f(O_y) >) \ = < O_y = f^*(O_y) >$$

E2 rule is applied to a J edge $y \rightarrow z$, if y is a non-join node and it do not contain a self-loop. We distinguish between two types of E2 rules depending on the levels of y and z. If $y.level = z.level$ we apply E2a rule; otherwise we apply E2b.

Definition 2.2 (E2 rules) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node such y do not contain a self-loop. Let $y \rightarrow z$ be a J edge, and let $x = idom(y)$. Let $H_y : O_y = kO_x + m$ be the equation at node y, such that the parameters k and m does not contain any variables. Finally, let $H_z : O_z = aO_y + b$ be the equation at node z, where a and b does not contain the variable O_y. There are two cases:*

(E2a rule) *If $y.level = z.level$ then*

- (i) Graph reduction:

$$E2a(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \ = < \mathcal{G}^{i+1}, N, E - \{y \xrightarrow{J} z\} >$$

- (ii) Variable elimination:

$$E2a(< H_z : O_z = aO_y + b >) \ = < H_z : O_z = a(kO_x + m) + b >$$

(E2b rule) *If $y.level \neq z.level$ then*

- (i) Graph reduction:

$$E2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \ = < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup \{x \xrightarrow{J} z\} >$$ [1]

- (ii)Variable elimination:

$$E2a(< H_z : O_z = aO_y + b >) \ = < H_z : O_z = a(kO_x + m) + b >$$

[1]We do not insert $x \rightarrow z$ in $\mathcal{G}^{i+1}$ if it is already present in $\mathcal{G}^i$.

The newly inserted edge $idom(y) \rightarrow z$ in E2b is called the **derived edge** of $y \rightarrow z$. An important point to note here is that before an E2 rule is applied to an edge $y \rightarrow z$, we first eliminate the self-loop $y \rightarrow y$, if it exists, using an E1 rule.

Handling Irreducibility

In our exhaustive eager elimination method irreducibility is detected whenever we cannot apply any of the $\mathcal{E}$-*rules*, although there are J edges that originate at the current level. If this is the case, we apply Tarjan's Strongly Connected Component (SCC) algorithm and collapse every non-trivial component to a single node. In applying the Tarjan's algorithm to the reduced DJ graph, we visits only the nodes via J edges whose source and destination nodes are at the current level. This will generate dag(s) of SCCs. It is important to remember there can be more than one disjoint dag at level i. We process each SCC in each dag in topological order. We compute closure of equations of all nodes in S, if needed. We also express the flow equations at all nodes at level i in terms of flow variable of their immediate dominator node. This is possible because we are eliminating the flow variables in the topological order of the SCC. Next we eliminate all J edges whose source and destination nodes are at level i. Finally we apply E2b rule to all J edges whose source nodes are at level i and destination nodes are at levels less i. Once this is done, all the J edges whose source nodes are at this level are eliminated, and so we can continue to apply $\mathcal{E}$-*rules* to nodes at levels less than i.

3 Our Approach

In the next several sections we present our approach for incremental data flow analysis. In this section we will set the stage for our approach. Specifically we will introduce the concept of *initial* and *final* flow equations in Section 3.1, and introduce the steps involved in the updating the final data flow solutions in Section 3.2, and introduce a representation called the DF graph that is useful for simplifying the problem of incremental data flow analysis (Section 3.3).

3.1 Initial and Final Flow Equations

Our exhaustive eager elimination method consists of three steps: (1) reduce the DJ graph to its dominator tree in a bottom-up fashion, (2) eliminate the variables by substitution, and (3) propagate the solution of root node to all other nodes, determining their corresponding solution. In [SGL95a] we showed that at the end of elimination phase (step 2), the flow equation at each node depends only on the output flow variable of its immediate dominator node. In other words, let y be a node and let $x = idom(y)$, the flow equation at node y, at the end of elimination phase, would resemble:

$$H_y^F : f^F(O_x) = O_y = P_y^F O_x + G_y, \tag{1}$$

where P_y^F and G_y^F are final parameters of the equation at node y. We will call the set of equations at the end of elimination phase as the **final flow equations**. In contrast, we will call the set of flow equations prior to DJ graph reduction and variable elimination as the **initial flow equations**. We will denote the initial flow equation at a node y as follows:

$$H_y^0 : f^0(I_y) = O_y = P_y^0\Big(\bigwedge_{z \in Pred_f(y)} O_z\Big) + G_y^0 \tag{2}$$

where P_y^0 and G_y^0 are initial parameters of the initial flow equation at node y, $\bigwedge$ is merge (union) operator, and $Pred_f(y)$ is a set of predecessors in the corresponding flowgraph.

Example: Consider a forward data flow problem with union as the merge operation (e.g., Reaching Definitions). The initial flow equation at each node for the example flowgraph shown in Figure 1 is as follows:

$$
\begin{aligned}
O_S &= G_S^0 \\
O_1 &= P_1^0 O_S + G_1^0 \\
O_2 &= P_2^0 O_1 + G_2^0 \\
O_3 &= P_3^0 (O_1 + O_8) + G_3^0 \\
O_4 &= P_4^0 (O_2 + O_3 + O_7) + G_4^0 \\
O_5 &= P_5^0 O_4 + G_5^0 \\
O_6 &= P_6^0 O_4 + G_6^0 \\
O_7 &= P_7^0 (O_5 + O_6) + G_7^0 \\
O_8 &= P_8^0 O_7 + G_8^0 \\
O_E &= P_E^0 (O_S + O_2 + O_8) + G_E^0
\end{aligned}
$$

The corresponding final flow equation at each node would resemble:

$$
\begin{aligned}
H_S^F : O_S &= G_S^0 \\
H_1^F : O_1 &= P_1^0 O_S + G_1^0 \\
H_2^F : O_2 &= P_2^0 O_1 + G_2^0 \\
H_3^F : O_3 &= P_3^0 O_1 + (P_3^0 P_8^0 P_7^0 P_5^0 + P_3^0 P_8^0 P_7^0 P_6^0) O_4 + P_3^0 P_8^0 P_7^0 G_6^0 + \\
&\quad P_3^0 P_8^0 P_7^0 G_5^0 + P_3^0 P_8^0 G_7^0 + P_3^0 G_8^0 + G_3^0 \\
H_4^F : O_4 &= P_4^0 P_2^0 O_1 + (P_4^0 P_3^0 P_8^0 P_7^0 P_5^0 + P_4^0 P_3^0 P_8^0 P_7^0 P_6^0) O_4 + \\
&\quad P_4^0 P_3^0 P_8^0 P_7^0 G_5^0 + P_4^0 P_3^0 P_8^0 P_7^0 G_6^0 + P_4^0 P_3^0 P_8^0 G_7^0 + P_4^0 P_3^0 G_8^0 + \\
&\quad P_4^0 G_3^0 + G_4^0 \\
H_5^F : O_5 &= P_5^0 O_4 + G_5^0
\end{aligned}
$$

$$
\begin{aligned}
H_6^F : O_6 &= P_6^0 O_4 + G_6^0 \\
H_7^F : O_7 &= (P_7^0 P_5^0 + P_7^0 P_6^0) O_4 + P_7^0 G_5^0 + P_7^0 G_6^0 \\
H_8^F : O_8 &= P_8^0 O_7 + G_8^0
\end{aligned}
$$

In the above system of equations, H_3^F and H_4^F are mutually recursive, and the final equations at nodes 3 and 4 is the closure of the two equations. Once the closure is determined, the final flow equations at these nodes are expressed in terms of their immediate dominator node [SGL95a]

3.2 Basic Steps

Given the notion of initial and final flow equations we are ready to lay the foundation of our approach for incremental data flow analysis. The problem of incremental data flow analysis can be concisely stated as follows [PS89]:

> *Given a program and a correct solution to a data flow problem over that program, update the affected parts of the current solution to reflect a change in the program without unnecessary reinitialization and recalculation of the entire data flow solution.*

To simplify the presentation, we will consider only the following two types of incremental changes:

- Non-structural change: The parameters of the initial flow equation H_y^0 at a node y are modified.
- Structural change: A flowgraph edge $x \rightarrow y$ is either inserted or deleted in the flowgraph.

One can easily extend and implement other types of incremental changes using the results of this paper. It is important to remember that all incremental algorithms rely on having correct solutions at all nodes prior to incremental changes. Once an incremental change is effected, incremental algorithms will update (ideally) only those solutions that are affected due to the incremental change [Mar89]. Let us denote the data flow solution at each node y, prior to incremental change, as α_y^F. We will call α_y^F as the **final flow solution** at a node y. Let $x = idom(y)$, then one can easily show the following input-output relation will hold at node y prior to an incremental change.

$$H_y^0 : \alpha_y^F = f^F(\alpha_x^F) = P_y^F \alpha_x^F + G_y^F \tag{3}$$

The first step in our approach is to associate with each node y: (1) the initial flow equation H_y^0, (2) the final flow equation H_y^F, and (3) the solution final flow solution α_y^F. Now supposing we induce an incremental change (such as updating the parameter of the initial flow equations, inserting a new flowgraph edge, or deleting an existing a flowgraph edge), our incremental data flow analysis will update the data flow solutions in two steps:

- Update the final data flow equations; and
- Update the final data flow solutions.

We will handle structural and non-structural changes separately. The complete algorithm for incremental data flow analysis is given below:

Algorithm 3.1 *The following algorithm updates the data flow solution when the corresponding flowgraph is subjected to structural and non-structural incremental changes.*

MainIDFA()
{

1: **if** (IncrementalChange == Non-Structural) **then**
2: Update final flow equations for non-structural changes (Section 4)
3: **else**
4: Update final flow equations for structural changes (Section 5)
5: **endif**
6: Update final data flow solutions (Section 6).

}

To effectively handle the incremental changes we will introduce a representation called the **Dominance Frontier (DF) graph**. A DF graph is nothing but the dominator tree of a flowgraph augmented with edges $x \rightarrow y$ such that $y \in DF(x)$. We will use DF graph for updating the final flow equations.

3.3 The DF Graph

In our exhaustive eager elimination method when we apply E2b rule to an edge $y \rightarrow z$, we eliminate $y \rightarrow z$ and insert $x \rightarrow z$ (if the edge is not already present), where $x = idom(y)$. In other words we "derive" the edge $x \rightarrow z$ from $y \rightarrow z$ by applying E2b rule, and so we call $x \rightarrow z$ as a derived edge of $y \rightarrow z$. Now if $x.level > z.level$, we will (subsequently) apply E2b rule to $x \rightarrow z$ to derive another edge $w \rightarrow z$, where $w = idom(x)$. We will continue to apply E2b rule to the "derived edges" as long as its source and destination nodes are at the same level, at which point we either apply E1 rule or E2a rule, eliminating the derived edge.

An astute reader may observe the relation between derived edges and dominance frontiers. To see this let $y \to z$ be the original J edge in the initial DJ graph, then $x \to z$ will be a derived edge if $z \in DF(x)$. This is interesting because given a dominator tree we can augment the dominator tree with edges $x \to y$ such that $y \in DF(x)$. We call the resulting graph as the **Dominance Frontier (DF) graph**. We can will immediately observe the relation between the DJ graph and the DF graph of a flowgraph. We can think of a DF graph as the DJ graph augmented with edges $x \to y$, such that $y \in DF(x)$ and $x \to y$ is not already present in the DJ graph. We will call the edges in DF graph that are not dominator tree edges as **DF edges** to distinguish from J edges. Figure 2 gives the DF graph for our example flowgraph.

Our next concern is the size of the DF graph. The size of the DF graph depends on the size of the dominance frontiers. Theoretically, the size of the dominance frontiers can be quadratic, but Cytron et al. have shown that for most practical programs this set is linear in terms of the number of edges in the flowgraph [CFR$^+$91]. Therefore the space complexity of our algorithm is linear for most practical programs.

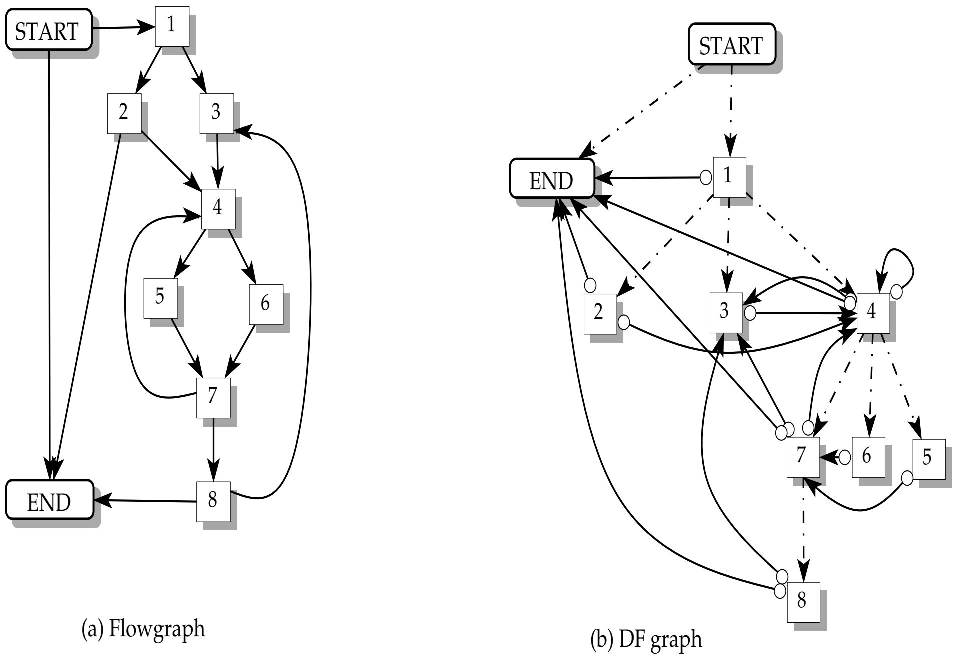

Figure 2: A flowgraph and its DF graph.

4 Updating Data Flow Equations: Non-Structural Updates

In this section we will show how to update the final data flow equations for a non-structural change at a node y in a flowgraph. An example of a non-structural change at a node y is addition or deletion of a new definition. Because of this update, the reaching definition information at other nodes may be affected. By adding or deleting a definition, we are essentially changing the parameters P_y^0 and G_y^0 of the initial flow equation at node y.

Given an incremental change at a node y, the first step is to determine the set of nodes whose final flow equations is affected. To determine the set of affected nodes we made one key observation. Consider the final flow equation of node 7 of the example flowgraph shown in Figure 1.

$$\begin{aligned} H_7^F : O_7 &= (P_7^0 P_5^0 + P_7^0 P_6^0) O_4 + P_7^0 G_5^0 + P_7^0 G_6^0 \\ &= P_7^F O_4 + G_7^F \end{aligned}$$

We notice that the final parameters P_7^F and G_7^F on the RHS of the equation are made up of the initial parameters P_5^0, G_5^0, P_6^0, and G_6^0 of nodes 5 and 6, respectively. It is important to remember that the parameters of both the initial flow equation and the final flow equation at a node are constant values. We will use the term *appears* to mean that the parameters of the final flow equations are computed from the parameters of initial flow equations during the reduction and variable elimination phase of the eager elimination algorithm. Given this notion, the key question to ask is: how are the parameters of the initial flow equations related to the parameters of the final flow equations? We found a surprisingly simple relation between the initial and the final parameters of flow equations.

Claim 4.1 *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$.*

We will prove this claim later in Section 7. The above claim has an important implication in our incremental algorithm. Supposing we make a non-structural change to a node y, thereby affecting P_y^0 and G_y^0. From Claim 4.1 we should update the final flow equation of all nodes that are in the $IDF(y)$. Now since we are changing P_y^0 and G_y^0, the final flow equation H_y^F at node y should also be updated. Let $\mathsf{FEqAffected}(y)$ be the set of all nodes whose final flow equation change due to a non-structural change at y (i.e., $\mathsf{FEqAffected}(y)$ is the set of "affected" nodes). To determine the set nodes whose final flow equations have to be updated, we will use the following key result:

Claim 4.2 *Let the parameters P_y^0 and G_y^0 of the initial flow equation at node y be updated. Then the final flow equation at a node w is* **affected** *(i.e., w is in $\mathsf{FEqAffected}(y)$) if and only if $w \in \{y\} \cup IDF(y)$.*

Therefore, from the above key result, we can see that the first step in updating the final flow equations is to compute the set $IDF(y)$. Computing $IDF(y)$ is much simpler using DF graphs than DJ graphs. We can easily show that a node w is in $IDF(y)$ if and only if there exists a path P from y to w in DF graph that does not contain D edges. Therefore to compute the $IDF(y)$ we determine all nodes that are reachable from y without visiting any D edges.

Once we compute the $IDF(y)$, we next construct a **Projection Graph** $Proj(y)$ of the DF graph with respect to the nodes in $\{y\} \cup IDF(y)$. The projection graph $Proj(y)$ consists of nodes in set $\{y\} \cup IDF(y)$, and we insert an edge $p \to q$ between any two nodes in $Proj(y)$ if and only if $q \in DF(p)$ in DF graph.[2] For example, let us construct the projection graph $Proj(5)$. First we compute $IDF(5)$, which consists of nodes $\{3, 4, 7, \mathrm{END}\}$. The next step is to insert edges $u \to w$ from the DF graph, such that $w \in DF(u)$ and both u and w are in $Proj(5)$. The resulting projection graph $Proj(5)$ is shown in Figure 3(a).

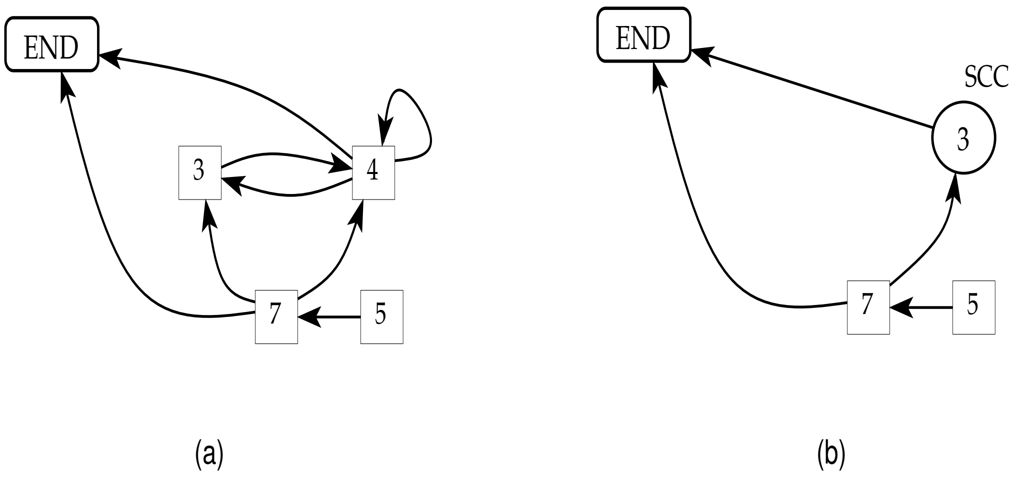

Figure 3: The projection graph $Proj(5)$ graph, and its dag.

Given the projection graph we will next show how to update the final flow equations of the affected nodes. It is important to remember that a node w is affected (because of a non-structural change at node y) if and only if w is in $Proj(y)$. It is also important to note that $Proj(y)$ need not be acyclic. So the first step is to apply Tarjan's Strongly Connected Component (SCC) algorithm and process each component in the topological order of the dag of SCCs. For example, the $Proj(5)$ in Figure 3(a) is not acyclic, and so we determine its dag of SCCs, which is shown in Figure 3(b). Now if a SCC in the dag of $Proj(y)$ contains a cycle, we compute the closure of the equations of the nodes in the SCC (as in the exhaustive case). We can easily show that if an SCC contains more than one node, then all the nodes will be at the same level in the DF graph [Sre95]. Now given the dag of SCCs of $Proj(y)$, *we*

[2] Note that the set $DF(p)$ is same in both the original flowgraph and its DF graph.

process each SCC in the dag in topological order. It is important to remember that the only variable on the RHS of the final flow equation at a node should be the output flow variable of its immediate dominator. Given this we next show how to construct the final flow equations of all the affected nodes. The projection graph $Proj(y)$ helps provide an ordering in which we can update the final flow equations at the affected nodes. As discussed above, we first apply Tarjan's Strongly Connected Component (SCC) algorithm to the projection graph. Then we determine a topological ordering on the strong components. Finally, we visit the strong components S in topological order to update their final flow equations. There are three cases to consider:

Case 1: S is a single node and has no self-loop. Let $Pred_f(S)$ be the set of predecessor of node S in the corresponding flowgraph. Assume that the final flow equation at every predecessor $p \in Pred_f(S)$ is correct (either previously updated or unaffected by the incremental change). To updated final flow equation H_S^F at node S, we start with its initial flow equation. That is, we first construct the following equation at node S:

$$H_S : O_S = P_S^0(\bigwedge_{t \in Pred_f(S)} O_t) + G_S^0, \tag{4}$$

Starting from this equation, we eliminate variables from it in a bottom-up fashion, as in our exhaustive eager elimination, until the only variable in H_S is that of its immediate dominator node. Recall that during the elimination process we create and delete derived edges. The topological ordering of the SCCs ensures that the final equation at source nodes of these derived edges are in its final form. At the end of the elimination process the equation at node S will be in its final form. Notice that the above (incremental) update is nothing but a "selective" exhaustive eager elimination process, but restricted to the equation at node S.

Case 2: S is a single node and has a self-loop. Here we assume that the final flow equation at every predecessor of node S, excluding S itself, is correct. We next perform a "selective" exhaustive variable elimination starting from the initial flow equation H_S of node S. At the end of this selective variable elimination, the only variable that will remain in H_S is O_S and $O_{idom(S)}$. At this point we compute the closure of the recursive equation to break the dependency of O_S on itself (*á la* E1 rule) and obtain the updated final flow equation H_S^F.

Case 3: S contains more than one node. In this case we have irreducibility, and so we have to simultaneously determine the final flow equation of all nodes in S. These nodes have the same immediate dominator. In this case we will assume that the final flow equation at all nodes in $N - S$ are correct (where N is a set of all nodes in the DF graph). As before, we again perform "selective" variable elimination, for each equation at w nodes in S, until the only variables remaining in the system of equations are those of nodes in S and the output variable their immediate dominator node. Finally we perform fixed-point iterations over all the mutually recursive equation and determine their closure.

The complete algorithm is given below.

UpdateFlowEq(y)
{

7: Compute $IDF(y)$
8: Determine $Proj(y)$
9: Apply Tarjan's SCC algorithm determine the SCCs in $Proj(y)$.
10: For each SCC S in topological order **do**
11: **Switch** (S)
12: **Case 1:** S is a single node and does not contain a self-loop.
Compute the final flow equation as described in the main text.
13: **Case 2:** S is a single node and contain a self-loop.
Compute the final flow equation as described in the main text.
14: **Case 3:** S contains more than one node.
Compute the final flow equation as described in the main text.
15: **EndSwitch**

}

Example: Consider the previous example where we induce a non-structural change to node 5. The corresponding $Proj(5)$ and its dag is shown in Figure 3. We will process the nodes in the dag order. So we will first process node 5. Node 5 is a single node with no self-loop, and so corresponds to case 1 (step $\boxed{\mathbf{12}}$). The only predecessor of node 5 (on the flowgraph) is node 4. The new final flow equation of node 5 is

$$H_5^{nF} = P_5^{n0} O_4 + G^{n0},$$

where the superscript n means *new* equation (or parameters).

Next we process node 7, and this also corresponds to case 1 in the algorithm (step $\boxed{\mathbf{12}}$). The two predecessor nodes of node 7 are 5 and 6. The initial flow equation of node 7 depends only on equations at node 5 and node 6, and is given below:

$$O_7 = P_7^0(O_5 + O_6) + G_7^0$$

Note that the equation at node 6 is unaffected, but at node 5 it is affected (and updated). After eliminating O_5 and O_6 in above equation we get the new final flow equation for node 7:

$$\begin{aligned} H_7^{nF} : O_7 &= (P_7^0 P_5^{n0} + P_7^0 P_6^0) O_4 + P_7^0 G_5^{n0} + P_7^0 G_6^0 \\ &= P_7^{nF} O_4 + G_7^{nF} \end{aligned}$$

Next we process the non-trivial SCC 3, which corresponds to case 3 in the algorithm (step $\boxed{\mathbf{14}}$). The SCC 3 consists of nodes 3 and 4. We first set up the intial flow equations for these two nodes:

$$
\begin{aligned}
O_3 &= P_3^0(O_1 + O_8) + G_3^0 \\
O_4 &= P_4^0(O_2 + O_3 + O_7) + G_4^0
\end{aligned}
$$

Next we perform variable elimination, and reduce the equation to the following form:

$$
\begin{aligned}
H_3^{nF} : O_3 &= P_3^0 O_1 + (P_3^0 P_8^0 P_7^0 P_5^{n0} + P_3^0 P_8^0 P_7^0 P_6^0) O_4 + P_3^0 P_8^0 P_7^0 G_6^0 + \\
&\quad P_3^0 P_8^0 P_7^0 G_5^{n0} + P_3^0 P_8^0 G_7^0 + P_3^0 G_8^0 + G_3^0 \\
H_4^{nF} : O_4 &= P_4^0 P_2^0 O_1 + (P_4^0 P_3^0 P_8^0 P_7^0 P_5^{n0} + P_4^0 P_3^0 P_8^0 P_7^0 P_6^0) O_4 + \\
&\quad P_4^0 P_3^0 P_8^0 P_7^0 G_5^{n0} + P_4^0 P_3^0 P_8^0 P_7^0 G_6^0 + P_4^0 P_3^0 P_8^0 G_7^0 + \\
&\quad P_4^0 P_3^0 G_8^0 + P_4^0 G_3^0 + G_4^0
\end{aligned}
$$

Notice that the only variables in the above two equations are those of the nodes in the SCC 3 and their immediate dominator node. Next we compute the closure of the two equations using fixed-point iterations, and express them only in terms of O_4, the immediate dominator of the nodes in SCC 3.

5 Updating Data Flow Equations: Structural Updates

Next we will show how to update the final flow equations for structural changes (i.e, insertion and deletion of a flowgraph edge). Our algorithm for structural changes consists of the following steps:

- Update the dominator tree of the flowgraph.
- Update the dominance frontier relation of the flowgraph.
- Update the final flow equations.

In a previous paper we gave a simple algorithm for updating the dominator tree of a flowgraph [SGL95b]. In this paper we will use that result to incrementally update the DF graph. A brief introduction to our incremental dominator algorithm is given in the next section (for details, please see [SGL95b]). In Sections 5.2 and 5.3, we present incremental algorithms for updating the dominance frontier relation for edge insertion and for edge deletion, respectively. Recall that $p \rightarrow q$ is a DF edge in DF graph iff $q \in DF(p)$. Therefore the problem of updating DF graphs is isomorphic to the problem of updating the dominance frontier relation.

Let $x \rightarrow y$ be some edge that is inserted/deleted in the flowgraph. Without loss of generality, we will initially assume that both x and y are reachable from the START before and after update. We will handle other cases, where the reachability of x and/or y changes, as special cases.

5.1 Updating Dominator Trees: An Overview

In [SGL95b] we presented a simple incremental algorithm for updating a dominator tree when the corresponding flowgraph is subjected to incremental changes (such as insertion and/or deletion of an edge). The solution procedure consists of first identifying the set of nodes that are "affected" because of the update. A node is said to *DomAffected* if its immediate dominator changes because of an update. For both the insertion case and deletion case, we can determine the exact set of affected nodes when $x \rightarrow y$ is updated, even prior to restructuring the DJ graph. In this paper we will use the notation $\mathsf{DomAffected}(y)$ to represent the (exact) set of *DomAffected* nodes when $x \rightarrow y$ is updated [SGL95b].

In [SGL95b] we showed, for the insertion case, that the new immediate dominator of all the nodes that are *DomAffected* is $nca(x, y)$, the nearest common ancestor of x and y on the dominator tree. For the deletion case, a single node does not immediately dominate all the affected nodes. But we can still determine the new immediate dominators of the affected nodes prior to restructuring the DJ graph [SGL95b]. Another interesting point to notice is that when an edge is inserted all the affected nodes move closer to the root, whereas when an edge is deleted all the affected nodes move away from the root.

Given the set $\mathsf{DomAffected}(y)$ and $nca(x, y)$ we will next show how to incrementally update the dominance frontiers of all the "affected" nodes. A node is *PossiblyDFAffected*, if its dominance frontier set possibly changes because of an update in the flowgraph. In the next two sections we give our algorithm for updating the dominance frontier relation. To simplify our presentation we will initially assume that both x and y are reachable from the START node; and then we will handle other cases, where the reachability of x and y differ, as special cases.

Example Consider the flowgraph shown in Figure 1. Let us insert an edge from node 2 to node 5, resulting in the flowgraph shown in Figure 4(a). Using the algorithm given in [SGL95b] we get $\mathsf{DomAffected}(5) = \{5, 7\}$. The new immediate dominator of all the affected is $nca(2, 5) = 1$. The updated DJ graph is shown in Figure 4(b).

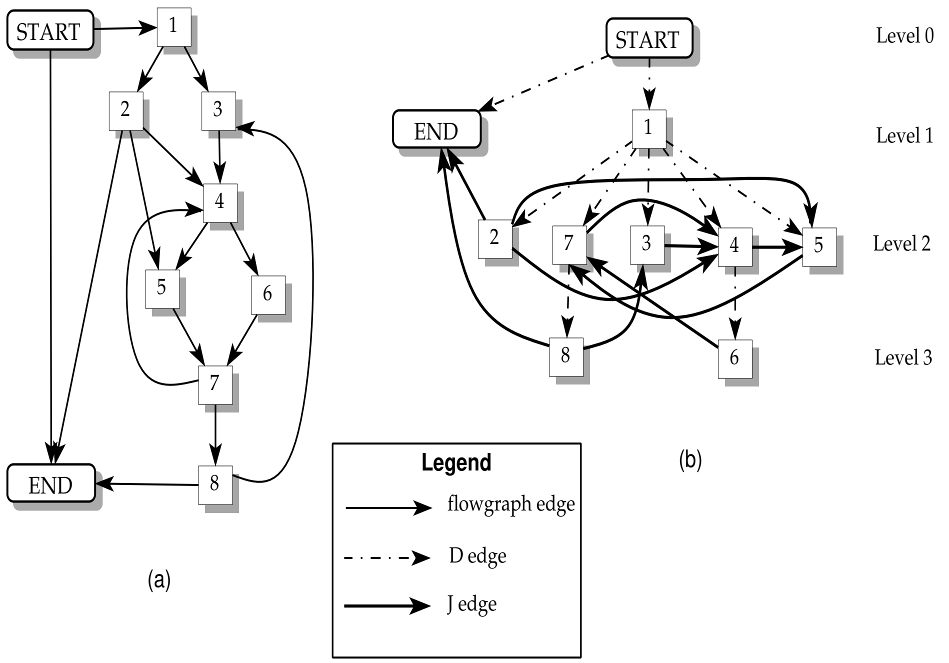

Figure 4: Flowgraph and its DJ graph after inserting $2 \to 5$

5.2 Updating Dominance Frontiers: Insertion of an Edge

In this section we give a simple algorithm for updating the dominance frontiers of all the nodes in $\mathsf{DFAffectedI}(y)$, when an edge $x \to y$ is inserted in the flowgraph. The key question to ask is: at which nodes the dominance frontiers may change when $x \to y$ is inserted. Recall that a node $w \in \mathsf{DomAffected}(y)$ will move up in the dominator tree after the DJ graph is updated. When this happens, the dominance frontiers of all nodes that dominate node w or x prior to updating will possibly be affected. Now let $z = nca(x, y)$, and let $\mathsf{DFAffectedI}(y) = \{u | z$ **stdom** u **and** u dominates a node $w \in \{x\} \cup \mathsf{DomAffected}(y)$ *prior* to updating the DJ graph$\}$. We will claim that if a node is not in $\mathsf{DFAffectedI}(y)$ then its dominance frontiers will not change. But if a node is in $\mathsf{DFAffectedI}(y)$ we cannot say for sure whether its dominance frontiers will change or not. We will formally prove our claim later in Section 7. Notice that we can easily compute the set $\mathsf{DFAffectedI}(y)$ by a simple bottom-up traversal of the (old) dominator tree starting from nodes in $w \in \{x\} \cup \mathsf{DomAffected}(y)$.

Once we have determined the set of possibly affected nodes, next we update the dominance frontiers of these affected nodes. For this we will first update the DJ graph using the algorithm given in [SGL95b]. Next, for each node $w \in \mathsf{DFAffectedI}(y)$, we will recompute $DF(w)$ in a bottom-up fashion on the new DJ graph as follows: Let w be a node in $\mathsf{DFAffectedI}(y)$ and assume that the dominance frontiers of all the children nodes on w are correct. Let

$p \in Children(w)$, then the new $DF(w)$ is given by the following formula [CFR+91].

$$\begin{aligned}
DF_{local}(w) &= \{r \mid w \rightarrow r \text{ is a J edge}\} \\
DF_{up}(p) &= \{q \mid q \in DF(p) \textbf{ and } q.level \leq idom(p).level\} \\
DF(w) &= DF_{local}(w) \cup \bigcup_{p \in Children(w)} DF_{up}(p)
\end{aligned} \tag{5}$$

Notice that the above formula is equivalent to the one given by Cytron et al. for (exhaustively) computing the dominance frontiers of all nodes [CFR+91].[3] But, unlike in the exhaustive case, we apply the above formula only to the nodes in DFAffectedI(y), the set of affected nodes. The complete algorithm is given below.

UpdateDFIns(x, y)
{

Insert $x \rightarrow y$ in the flowgraph
$z = nca(x, y)$
Compute DomAffected(y) ;
Compute DFAffectedI(y) ;
Update the DJ graph ;
For each node $w \in$ DFAffectedI(y) in a bottom-up fashion
  (ordered by their levels) **do**
  Compute $DF(w)$ using the formula given in the main text.
endfor

}

Example: Consider the flowgraph and its DJ graph shown in Figure 1. Let us insert an edge from node 2 to node 5. The resulting flowgraph and the updated DJ graph is shown in Figure 4. Using the algorithm given in [SGL95b], the set DomAffected(5) is $\{5, 7\}$, and $nca(2, 5) = 1$. Next we compute the DFAffectedI(5). This set consists of all the nodes that dominate the set of nodes in DomAffected(5) (i.e., 5 and 7), and are strictly dominated by $nca(2, 5)$ (i.e., 1). We compute the set DFAffectedI(5) before updating the dominator tree. To compute the set DFAffectedI(5) we simply perform a bottom-up traversal of the dominator starting from nodes in DomAffected(5) until we reach $nca(2, 5)$. We include all the nodes, except $nca(2, 5)$, that are visited during this bottom-up traversal into the set DFAffectedI(5). By doing this we get DFAffectedI(5) $= \{2, 4, 5, 7\}$.

The dominance frontiers of the nodes in DFAffectedI(5), prior to edge insertion are: $DF(2) = \{4, \text{END}\}$, $DF(4) = \{3, \text{END}\}$, $DF(5) = \{7\}$, and $DF(7) = \{3, 4, \text{END}\}$. After inserting the edge the new dominance frontiers for these nodes are: $DF(2) = \{4, 5, \text{END}\}$, $DF(4) = \{5, 7\}$, $DF(5) = \{7\}$, and $DF(7) = \{3, 4, \text{END}\}$. Notice that the dominance frontiers

[3] The difference between the two formulas is that we use level information whereas Cytron et al. use dominance relation for computing dominance frontiers.

for nodes 5 and 7 did not change because of the update, but we still have to recompute its dominance frontiers. In general, it is much more difficult to determine the exact set of nodes whose dominance frontiers will definitely change. Also, since computing the dominance frontier relation using above formula is linear practice it may not be worth the effort to determine the exact set of affected node.

Other Cases

Next we will extend the insertion algorithm for cases where both x and y are not reachable. The first case is where x is not reachable, for which we do nothing because we maintain the DJ graph and the dominance frontiers only for the reachable sub-graph of the flowgraph. In the second case, y becomes reachable only after the edge insertion. For this we first build (using exhaustive algorithm) the DJ sub-graph and the corresponding dominance frontier set for the sub-flowgraph induced by nodes reachable from y but not reachable from START. In constructing this DJ sub-graph and the dominance frontiers, we treat y as its root. Since x must dominate y, we then insert a D edge from x to y (to connect the newly built DJ sub-graph with the existing DJ graph). Finally, from the viewpoint of updating DJ graph and dominance frontiers, we pretend each edge $u \to v$ to be a newly inserted flowgraph edge, where u becomes reachable only after $x \to y$ is inserted, and v is reachable even before the edge insertion. This corresponds to the case that we discussed earlier.

5.3 Updating Dominance Frontiers: Deletion of an Edge

In this section, we show how to update the dominance frontiers when a flowgraph edge $x \to y$ is deleted. The effect of deleting an edge is opposite and complementary to that of inserting the same edge. Recall that when inserting an $x \to y$, we pull up all the nodes whose immediate dominator node changes. By contrast, when deleting the edge $x \to y$, we pull down these affected nodes. So we have to recompute the dominance frontiers of all possibly affected nodes. Again let $z = nca(x, y)$, and let $\mathsf{DFAffectedD}(y) = \{u | z$ **stdom** u **and** u dominates a node $w \in \{x\} \cup \mathsf{DomAffected}(y)$ *after* updating the DJ graph$\}$. Now we claim that if a node is not in $\mathsf{DFAffectedD}(y)$ then its dominance frontiers does not change.

In the deletion case we compute the set $\mathsf{DFAffectedD}(y)$ after updating the DJ graph. Once the $\mathsf{DFAffectedD}(y)$ set is determined then we update the dominance frontiers of the nodes in $\mathsf{DFAffectedD}(y)$ as in the insertion case. The complete algorithm is given below.
UpdateDFDel(x, y)
{

24: Delete $x \to y$ in the flowgraph
25: $z = nca(x, y)$
26: Compute $\mathsf{DomAffected}(y)$;

```
    Update the DJ graph ;
    Compute DFAffectedD(y) ;
    For each node s ∈ DFAffectedD(y) in a bottom-up fashion
      (ordered by their levels) do
      Compute DF(s) using the formula given in the main text.
    endfor
}
```

Other Cases

Here we extend the deletion algorithm for cases where both x and y are not reachable. Let $x \rightarrow y$ be the deleted edge in the flowgraph. If x is not reachable, then we do nothing because we only maintain the DJ graph and the dominance frontier relation for the reachable sub-graph. Now assume that x is reachable and y becomes unreachable after deletion. In this case, we remove from the DJ graph all the nodes in $SubTree(y)$ and their incident edges. This is because if removing $x \rightarrow y$ makes y become unreachable, then all the nodes strictly dominated by y will not be reachable either. Next, we remove the D-edge $x \rightarrow y$. We update the reachability status for all the nodes that become unreachable due to the edge deletion. Finally, notice that we never need to update the dominance frontiers of nodes that are reachable from the START node. This is because none of the nodes in $SubTree(y)$ will be in the dominance frontiers of any node that are reachable from the START node.

5.4 Updating Final Flow Equations

Once we have updated the dominance frontier relation, we will next show how to update the final flow equations at all the 'affected' nodes. Again, let the edge $x \rightarrow y$ be the structural update (either inserted or deleted). As in the previous section, we will first handle the case where both x and y are reachable from the START node before and after the update. Now the key question to ask is: at which nodes are the final data flow equations affected when a new edge $x \rightarrow y$ is updated. The answer to the above question is given in the following claim:

Claim 5.1 *Let $x \rightarrow y$ be the edge that is updated in the flowgraph. The final data flow equation at a node w is affected if and only if $w \in \{y\} \cup IDF(y)$.*

This is very interesting. Recall in Section 4 we made a similar claim (Claim 4.2) for non-structural updates. This means that we can essentially use the same algorithm as given in Section 4 for updating the final data flow equations even for structural changes. But rather than using the 'old' DF graph we will have to use the updated DF graph, to compute the new set of final flow equations. In other words we first compute the $IDF(y)$ on the new DF graph and, as before, construct the projected graph $Proj(y)$. Once the projected graph is constructed, we update the final flow equations at all nodes in $Proj(y)$, using the same strategy described in Section 4.

6 Updating Data Flow Solutions

In this section we will show how to update the final data solutions α^F, once the final data flow equations have been updated. The first key question to ask is: at which nodes the final data flow solutions are affected because of incremental changes? As before let $Proj(y)$ be the set of nodes where data flow equations have been updated. Once the data flow equations have been updated at these nodes, we may have to update their final data flow solution. It is important to remember that the final data flow equation at each node will depend only on the output flow variable of its immediate dominator node.

Let w be a node in $Proj(y)$ whose *new* final data flow equation is $f_w^{Fnew}(I_w)$. Let α_w^{Fold} be the old final solution at this node (which may not be a correct solution). Let $u = idom(w)$ whose data flow solution α_u^{Fcor} is a 'correct" solution.[4] If the following relation holds at node w

$$\alpha_w^{Fold} = f_w^{Fnew}(\alpha_u^{Fcor}),$$

then we need not update the final solutions of the children nodes of w. Otherwise, we have to compute α_w^{Fcor} and mark the immediate dominee of w (i.e., children of w on the dominator) as potentially being affected, and repeat the process for each children node. Since the flow equation at node w is depends only on its immediate dominator node, and since the solution of its immediate dominator is correct, we can compute the new correct final solution at w as follows:

$$\alpha_w^{Fcor} = f_w^{Fnew}(\alpha_u^{Fcor})$$

It is important to observe that before we can update the final solution at a node w, we have to ensure that the final solution of its immediate dominator node is correct. Also, once (and if) we update w's final solution we have to mark all its children nodes to be potentially affected, and so their final solutions have to be updated. Therefore, we order the nodes in $Proj(y)$ in terms of the levels of the nodes on the dominator tree, and process the nodes in a top-down fashion. We will use a data structure akin to $OrderedBuckets$ to keep track nodes where the final solutions are possibly affected. $OrderedBuckets$ is an array of buckets ordered by levels of the nodes on the dominator tree. When a node x is inserted in the bucket, it will be inserted at the bucket $OrderedBuckets[x.level]$. We will initially insert all the nodes whose final flow equation was updated into the $OrderedBuckets$. We then iterate by picking out one node at a time in a *top-down* fashion and updating its solution. We check if the old solution

[4]By correct solution we mean that either its original solution was unaffected because of the incremental change, or some how has been correctly updated.

at the node is consistent with the final data flow equation at the node, if so we pick the next node from the *OrderedBuckets*. Otherwise we compute the new final solution and insert all the children nodes into the *OrderedBuckets*. The complete algorithm is given below.

UpdateFlowSol()
{

32: Insert the set of nodes $Proj(y)$ in *OrderedBuckets*
33: **foreach** $i = 1$ to $NumLevel - 1$ **do**
34:   **while**(($u =$ **GetNode**$(i)) \neq NULL$) **do**
35: $w = idom(u)$
36:   **if**($\alpha_u^{Fold} \neq f_u^{Fnew}(\alpha_w^{Fcor})$) **then**
37: $\alpha_u^{Fcor} = f_u^{Fnew}(\alpha_w^{Fcor})$
38:   Insert the children nodes of u in *OrderedBuckets*.
39:   **else**
40: $\alpha_u^{Fcor} = \alpha_u^{Fold}$ `/* Old solution is the correct solution */`
41:   **endif**
42:   **endwhile**
43: **endfor**

}

The function **GetNode**(i) returns a node x if one exists in the ith bucket, otherwise it returns $NULL$. Notice at step $\boxed{\textbf{40}}$ if the old solution is "correct" then the children nodes are not affected (unless they are already in the *OrderedBuckets*).

7 Correctness and Complexity

In this section we will prove the correctness of our incremental algorithm and analyze its time complexity. The main theorem which establishes the correctness of Algorithm 3.1 is Theorem 7.2. The proof of Theorem 7.2 relies on Lemma 7.8 and Lemma 7.7. Lemma 7.7 establishes the correctness of updating final flow equations for both structural and non-structural changes. To update the final flow equation of a node we must first know that it is affected. Lemma 7.3 and Lemma 7.6 gives the necessary and sufficient condition to determine the *exact* set of nodes whose final flow equations have to be updated for non-structural and structural changes, respectively. The validity of these two lemmas is based on another key lemma (Lemma 7.2) which establishes a relation between the parameters of initial flow equations and the parameters of the final flow equations. The validity of Lemma 7.7 for structural changes relies on the correctness of dominance frontier update algorithm (Theorem 7.1). Theorem 7.1 relies on two other lemmas, Lemma 7.4 and Lemma 7.5, which gives conditions for determining the set of nodes whose dominance frontiers is possibly affect when an edge is inserted and when an edge is deleted, respectively. Finally, Lemma 7.8 establishes the correctness of updating final flow solutions once the final flow equations have been updated.

In our chain of proofs, we begin with Lemma 7.1, that relates the concept of dominance frontiers and derived edges.

Lemma 7.1 *In the exhaustive eager elimination method, a derived $u \rightarrow w$ will created and processed at some stage in the elimination phase iff $w \in DF(u)$.*

Proof:

The "if" part: In [SG95] we showed that if $w \in DF(u)$ then there exists a J edge $t \rightarrow w$ such that u **dom** t and $w.level \leq u.level$. Now if $t \rightarrow w$ is a J edge then this edge will be processed during some stage in the elimination phase (i.e. one of $\mathcal{E}$-*rules* will be applied to this edge and the edge will be eliminated). Now if $t = u$, then we are done (i.e, $u \rightarrow w$ is a derived edge). Otherwise we will apply one or more E2b rules to the derived edges of $t \rightarrow w$ until the source node of the derived edge is at the same level as the destination. Since u **dom** t, and $w.level \leq u.level$, eventually a derived edge $u \rightarrow w$ will created and processed.

The "only if" part: If $u \rightarrow w$ is a derived edge then $u \rightarrow w$ was created and processed at some stage during the elimation phase [SGL95a]. In other words, it was derived from some J edge $t \rightarrow w$ such that u **dom** t, and the level of u is greater than or equal to the level of w. But we know that if $t \rightarrow w$ is a J edge such that u **dom** t and $u.level \geq w.level$, the $w \in DF(u)$. Hence the result.

∎

Next we prove Lemma 7.2. This lemma is exactly the same as Claim 4.1. This lemma states that the parameter P_u^0 and G_u^0 in the initial flow equation of node u will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$. It is important to remember that the parameters of both the initial flow equation and the final flow equation at a node are constant values. We use the term *appears* to mean that the parameters of the final flow equations are computed from the parameters of initial flow equations during the reduction and variable substitution phase of the eager elimination algorithm.

Lemma 7.2 *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$.*

Proof: It is obvious to see that if $w = u$ then P_u^0 and G_u^0 will appear in P_w^F and G_w^F, and vice versa. So let us assume that $w \neq u$.

The "if" part: We want to show that if $w \in IDF(u)$, then P_u^0 and G_u^0 will appear in P_w^F and G_w^F.

First we will show that if $w \in DF(u)$ then P_u^0 and G_u^0 will appear in P_w^F and G_w^F. Using Lemma 7.1 we can see that if $w \in DF(u)$ then $u \rightarrow w$ is a derived edge. When this edge is processed during the elimination phase, we will eliminate O_u in the flow equation H_w at node w, by substituting it with $P_u^F O_r + G_u^F$, where $r = idom(u)$. The parameters P_u^F and G_u^F will contain P_u^0 and G_u^O and so will appear in H_w, and hence H_w^F, the final flow equation of node w.

Now to show that if $w \in IDF(u)$, then P_u^0 and G_u^0 will appear in P_w^F and G_w^F, we can use inductive definition of iterative dominance frontiers. Now, if $w \in IDF(u)$, then there exist nodes $t_0, \ldots, t_k$ such that $w = t_0$, $u = t_k$, and $t_i = DF(t_{i+1})$, where $0 \leq i \leq k-1$. We can use induction on i to show the result.

The "only if" part: We want to show that if P_u^0 and G_u^0 appear in P_w^F and G_w^F, then $w \in IDF(u)$.

Now assume to the contrary that $w \notin IDF(u)$. Then either w is not reachable from u or there exists a node s that strictly dominates w and $s \in IDF(u)$. But if w is not reachable then P_u^0 and G_u^0 will not appear in P_w^F and G_w^F contradicting our assumption. Assume that w is reachable from u but is not in $IDF(u)$. Now we know that there exists a node s closest to w that strictly dominates w and $s \in IDF(u)$ [SGL95b, CFR+91]. Therefore all paths from u to w must pass through s. Now if s **stdom** w then a derived edge will never be created between s and w (since s can never be in $DF(w)$), and so P_u^0 and G_u^0 will never propagate to w via node s, and since all paths from u to w must pass through s, P_u^0 and G_u^0 can never appear in P_w^F and G_w^F, contradicting our initial assumption. Therefore w must be in $IDF(u)$.

∎

Next we will prove Lemma 7.3, which is same as Claim 4.2.

Lemma 7.3 *Let a non-structural change be induced at a node y. The final flow equation at a node w is* **affected** *(i.e., w is in* FEqAffected(y)*) if and only if $w \in \{y\} \cup IDF(y)$.*

Proof: First of all observe that when we induce a non-structural change at a node y, we are essentially changing the parameters P_y^0 and G_y^0 of the initial flow equation at y. From Lemma 7.2 we know that if $w \in IDF(y)$ then P_y^0 and G_y^0 will appear in P_w^F and G_w^F, and vice-versa. Therefore if P_y^0 and G_y^0 are updated we have to update the parameters P_w^F and G_w^F of all final flow equations that appear in $y \cup IDF(y)$. The converse is also true i.e., if a node is in $\{y\} \cup IDF(y)$ then its final flow equation is affected. ∎

Next we will establish the correctness for structural updates. Let $x \rightarrow y$ be the edge that is updated. Without loss of generality assume that both x and y are reachable from START.

We will first show that the dominance frontier update algorithm is correct. To establish this result we will show that if a node is not in $\mathsf{DFAffectedI}(y)$ ($\mathsf{DFAffectedD}(y)$) then it is not $PossiblyDFAffected$. By possibly affected we mean its dominance frontier may or may not change due to the insertion (deletion) of the edge $x \rightarrow y$. In the following lemma we will show the result for the insertion case, and in Lemma 7.5 we will show the result for the deletion case.

Lemma 7.4 *If a node w is not in* $\mathsf{DFAffectedI}(y)$ *then its dominance frontiers does not change due to the insertion of the edge $x \rightarrow y$.*

Proof:

Let $z = nca(x, y)$. It is obvious to see that if w is not in $SubTree(z)$ then none of the nodes in $SubTree(z) - \{z\}$ will be in $DF(w)$, and so its dominance frontier will not be affected. So we will consider only nodes within $SubTree(z)$.

Case 1: $w = z$. In this case again w will not be affected. This is because w will strictly dominate all the nodes whose immediate dominator change.

Case 2: w is some node that is strictly dominated by x. Assume that x is not in $\mathsf{DomAffected}(y)$. Let $s \in DF(w)$. So there must be a J edge $t \rightarrow s$ such that w **dom** t. Now if $s \notin \mathsf{DomAffected}(y)$ then s will not be $DomAffected$ and so s will still be in $DF(w)$. Now if $s \in \mathsf{DomAffected}(y)$, then it will move up, and so its level number never increases [SGL95b]. Therefore s will still be in $DF(w)$ and so dominance frontier of w is not affected due to s's movement. Therefore w is not in $\mathsf{DFAffectedI}(y)$.

Case 3: w is not in $\mathsf{DomAffected}(y)$ and is strictly dominated by some node in $\mathsf{DomAffected}(y)$. Let $s \in DF(w)$. Again there must be a J edge $t \rightarrow s$ such that w **dom** t. Using the same argument as in Case 2 we can see that w is not affected.

∎

Now consider the deletion case. In the next lemma we show that if a node is affected then it is in $\mathsf{DFAffectedD}(y)$.

Lemma 7.5 *If a node w is not in* $\mathsf{DFAffectedD}(y)$ *then its dominance frontiers does not change due to the deletion of the edge $x \rightarrow y$.*

Proof: We can use a similar argument as in Lemma 7.4 to prove this lemma. ∎

Given the proof of Lemmas 7.4 and Lemma 7.5, we will next establish the correctness of updating the dominance frontier relation.

Theorem 7.1 *The procedures* **UpdateDFIns**(x, y) (**UpdateDFDel**(x, y)) *correctly updates the dominance frontier relation when an edge* $x \to y$ *is inserted (deleted).*

Proof: Cytron et al. have shown that the Equation 5 correctly computes the dominance frontiers for all nodes when the nodes are processed in a bottom-up fashion. Using this result in conjunction with Lemma 7.4 (Lemma 7.5) we can easily see the procedures correctly updates the dominance frontiers of all the affected nodes when performed in a bottom-up fashion. ∎

Next we will establish the correctness of Lemma 7.6, which which is same as Claim 5.1.

Lemma 7.6 *Let* $x \to y$ *be the edge that is updated in the flowgraph. The final data flow equation at a node* w *is affected if and only if* $w \in y \cup IDF(y)$.

Proof: The proof of lemma is based on the following observation. When $x \to y$ is updated we are essentially changing the input flow information of node y. Therefore the parameters of the final flow equation at node y is affected. This situation exactly corresponds to non-structural updates, except that we do not change the parameters of the initial flow equation of node y. The rest of the proof is exactly same as in the proof of Lemma 7.3 ∎

Recall that once we identify the set of nodes whose final flow equations are affected we have to re-evaluate their new final flow equation. Next we will show that steps $\boxed{\mathbf{9}}$ to $\boxed{\mathbf{15}}$ correctly re-evaluates the final flow equation for both structural and non-structural changes.

Lemma 7.7 *The steps* $\boxed{\mathbf{9}}$ *to* $\boxed{\mathbf{15}}$ *correctly updates the final flow equations for both structural and non-structural changes.*

Proof:

From Lemmas 7.3 and 7.6 we know exactly at which nodes the final flow equations are affected. For non-structural updates we do not change the structure of the DF graph. For structural changes we first update the DF graph, and then update the final flow equations. From Theorem 7.1 we know that DF graph is correctly updated. Therefore in the rest of the lemma we will not distinguish between the two incremental changes.

First of all observe that we are processing SCCs in topological order of the dag obtained by collapsing the non-trivial SCCs in $Proj(y)$ (step $\boxed{\mathbf{10}}$). Therefore when processing an SCC S we are ensured that final flow equations at all the nodes u such that $u \to w$ is an edge in DF graph, $u \notin S$, and $w \in S$ are correct (either previously updated or is unaffected). This topological order also ensures that the algorithm will terminate in finite time.

Given this it is enough to show that the final flow equation at affected node that is derived by eliminating variables from the corresponding initial equation is the correct final flow equation. The derivation of the final equation depends on the type of SCC S, and we will handle them seperately.

Case 1: s is a single node and does not contain a self-loop. Its initial equation is given by

$$H_S^0 : O_S = P_S^0(\bigwedge_{t \in T} O_t) + G_S^0, \tag{6}$$

where $T = \{t | t \in Pred_f(S)\}$. To eliminate variables from the above equation we perform selective exhaustive eager elimination. Since we are processing the nodes in the topological order, we are ensured that the final flow equation at the destination node of every derived edge (generated and processed during the selective eager elimination process) is correct (either updated or unaffected). Since we showed the correctness of the exhaustive eager elimination in [SGL95a], the correctness of the selective elimination directly follows from it.

Case 2: S is a single node and contains a self-loop The only difference between this case and Case 1 is that we also compute closure of the recursive equation. In the E1 rule we also compute closure whenever there is a self-loop at a non-join node. Once the closure is computed, the equation at node s is the final flow equation.

Case 3: S contains more than one node. In this case we first form a set of mutually recursive equation by eliminating all output flow variables O_p such that p is not in s. This situation corresponds to the irreducible case in our exhaustive eliminationmethod. As in the exhaustive case we determine the closure of all the mutually dependent equations, and then express the final flow equation at all nodes in s in terms of their immediate dominator.

∎

Next we will show that the algorithm for updating final flow solution is correct. First of all observe that the data flow solutions at all nodes whose final flow equation is updated is potentially affected. So we may have to update their solution. Now let α_u^{Fold} be the old solution at a node u, i.e., solution of node u prior to incremental change. Let $w = idom(u)$, and assume that its solution α_w^{Fcor} is correct. If $\alpha_u^{Fold} \neq f_u^{Fnew}(\alpha_w^{Fcor}))$ then we need not update the solution at node u. Otherwise we have to update its solution and mark the solutions of all its children node as being affected.

Lemma 7.8 *The procedure* **UpdateFlowSol**() *correctly updates the final flow solutions at all the nodes whose final solutions are incorrect.*

Proof:

First of all notice that if a node u is returned by **GetNode**() then it is possibly affected. A node u is returned by **GetNode**() if and only if it was previously inserted in *OrderedBuckets*; and a node u is inserted in *OrderedBuckets* if it either in $Proj(y)$ (step $\boxed{\mathbf{32}}$ or the solution of its parent node was previously updated (step $\boxed{\mathbf{38}}$). Since we are processing the nodes in a top-down manner we will eventually update the final flow solutions at all the affected nodes.

∎

Finally we prove the correctness of our incremental data flow analysis.

Theorem 7.2 *The Algorithm 3.1 correctly updates the data flow solutions for both structural and non-structural changes to flowgraphs.*

Proof: Follows from Thereom 7.1, Lemma 7.7, and Lemma 7.8. ∎

Next we will analyze the time complexity of our approach.

Theorem 7.3 *The worst case time complexity of Algorithm 3.1 is* $O(|E| \times |N|)$.

Proof: (1) For both structural and non-structural updates selective eager elimination could, in the worst case, be performed over all nodes. And so the worst case time complexity for selective elimination is $O(|E| \times |N|)$ [SGL95a].

(2) the worst case time complexity of updating both dominance frontiers and dominator trees is bounded by $O(|E| \times |N|)$.

(3) The worst-case time for updating the final solution is $O(|N|)$, since we are propagating the solution on the dominator tree.

Combining (1), (2), and (3) we can see that the worst case time complexity of Algorithm 3.1 is $O(|E| \times |N|)$.

∎

For both non-structural changes and for insertion of an edge our algorithm is expected to behave linearly in practice since the size of the dominance frontier is linear in practice [CFR+91]. The cost for updating the data flow solution for deletion case is dominated by the cost for updating the dominator tree. Since we use Purdom and Moore algorithm for this step, the time complexity is quadratic in the worst-case for the deletion.

8 Discussion and Related Work

In this paper we proposed a new approach for incremental data flow analysis based on elimination methods. Previous work most relevant to ours is due to Carroll and Ryder [CR88]. We will first give a detailed comparison of our approach with theirs, and then compare with other related work.

Carroll and Ryder's algorithm is based on the notion of *reduce and borrow* concept for updating the data flow solutions [CR88]. They *reduce* a monotone data flow problem to an attributed (dominator) tree problem, and then *borrow* the well-known Reps's attribute update algorithm for updating the data flow solutions [Rep82, RTD83]. They use Graham-Wegman elimination algorithm as a starting point for mapping data flow problems to attributed dominator tree problems [GW76]. They decorate each node in the dominator tree with its (1) initial flow equation (2) final flow equation, and (3) the correct solution. These decorations are treated as attributes of the dominator tree. Once they construct an attributed dominator tree, they modify the well-known Reps's algorithm for updating the attributed dominator tree [Rep82, RTD83]. Reps's original algorithm can only handle updates to attributed parse tree, which are derived from attributed grammars. Since dominator trees are not parse trees, Carroll and Ryder generalize Reps's algorithm for handling updates to arbitrary trees.

In an attributed parse tree problem, we associate with each node a *semantic function* which defines the value of that attribute in terms of values of other attributes [ASU86]. Given an attributed parse tree, if the attribute at node y uses the value of attribute node at x, then we say that node (attribute) y depends on node (attribute) x. The *dependency graph* of a set of attributes A is a graph whose nodes are the elements of A, and there is an edge $u \rightarrow w$ if w depends on u. The value of an attribute is *consistent* if it equals the value returned by the attribute's semantic function. A *solution* to an attributed parse tree is a set of consistent values for all its attributes [ASU86].

Reps's original algorithm can handle updates only if the dependence graph of the attributed tree is acyclic [Rep82, RTD83]. Carroll and Ryder show that if the original flowgraph is reducible then the dependence graph of the attributed dominator tree is also acyclic. Presence of irreducibility in the original flowgraph introduces cycles in the dependence graph of the attributed dominator tree, and so we cannot use Reps's algorithm for updating such trees.

Reps's algorithm basically consists of replacing the affected sub-parse tree with a correct sub-parse tree and propagating the attribute values of the new sub-tree to all other nodes that depend on it. To ensure optimality the attributes are propagated on a projected graph of the dependence graph, called the sub-ordinate and superior *characteristic graph* [Rep82, RTD83]. Carroll and Ryder show how to construct these characteristic graphs for attributed dominator problem, and use them for updating and propagating final data flow solutions.

The sub-parse tree replacement in Reps's algorithm corresponds to restructuring of the dominator tree in Carroll and Ryder's algorithm. Carroll and Ryder also propose an algorithm for updating the dominator tree of the flowgraph. While updating the dominator tree they compute, what they call as *representative edges*, which are central to their update algorithm. These representative edges are then used for updating both the dominator tree and the attributes of the dominator tree. Projection of these representative edges with respect to the root of the affected sub-tree corresponds to the characteristic graphs in Reps's algorithm. For reducible flow graphs the projection of the representative edges form a dag, and so they can update the attributes of the dominator tree in the dag order of the projection graph.

In our algorithm too we "reduce" the problem to an attributed tree problem (since we are annotating the DJ graph with initial flow equation, final flow equation, and the final flow solution). But, unlike Carroll and Ryder's approach, we use simple properties of dominance frontiers and iterated dominance frontiers for updating the final data flow solution, and these properties are valid for both reducible and irreducible flowgraphs.

Although we do not use Reps's update algorithm for updating data flow solutions, we will show how the notion of dominance frontiers and iterated dominance frontiers are tied to the notion of dependence graphs and characteristic graphs used in Reps's algorithm. As in Carroll and Ryder's algorithm the dominator tree in our algorithm corresponds to the parse tree in Reps's algorithm. The DF graph in our algorithm correspond to the dependence graph in Reps's algorithm. Interestingly enough, the representative edges used in Carroll and Ryder's algorithm are nothing but DF edges in our DF graph. The superior characteristic graphs in Reps's and Carroll and Ryder's algorithm correspond to the projection graph $Proj(y)$ in our algorithm, although $Proj(y)$ can contain cycles.[5] Recall that $Proj(y)$ is derived from $IDF(y)$, the iterated dominance frontiers of y. This suggests that the concept of iterated dominance frontiers is deeply related to the concept of superior characteristic graphs used in Reps's algorithm. These two concepts were developed independently, and for different problems—dependency graphs and the characteristic graphs were introduced in the context of attributed grammars, whereas dominance frontiers and iterated dominance frontiers were introduced in the context of Static Single Assignment form [CFR+91]. An interesting direction for future research would be to further explore the possible relation between these two concepts.

Burke proposes an algorithm for elimination-based incremental data flow analysis that use interval graphs for updating and propagating data flow solutions [Bur90]. His algorithm can only handle structural changes (to flowgraphs) that does not change the depth-first spanning tree of the flowgraph. Marlowe and Ryder propose a hybrid incremental algorithm that combines iteration and elimination methods [MR90]. They first identify strongly connected components in the flowgraph, and they use iteration method within each component, but propagate the solutions to other components using elimination-like method. Although they can handle program arbitrary updates, their incremental algorithm is more coarse-grained; they update and propagate solutions to a much larger set of nodes than our algorithm or Carroll and Ryder's algorithm.

[5] Assuming that a non-structural change is induced at node y or we update an edge $x \to y$.

9 Conclusion

We have presented a new incremental data flow analysis based on elimination methods which can handle arbitrary program edits including irreducibility. We also gave an incremental algorithm for updating the dominance frontiers of a flowgraph. To incrementally update the data flow solutions we use simple properties of dominance frontiers and iterated dominance frontiers. Our approach to incremental analysis is based on incrementalizing our previous work on eager elimination method. Since the time complexity of our algorithm (except updating dominator tree for deletion case) depends on the size of the dominance frontiers, we expect our algorithm to behave linearly in practice. We currently have implemented our exhaustive eager elimination algorithm. Our immediate future work would be to implement the incremental algorithm and compare it with our exhaustive algorithm.

Acknowledgement

The first and second authors thank the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS) for their continued support of this research.

References

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

[BR90] Burke, M. G., and Ryder, B. G. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7):723–728, July 1990.

[Bur90] Burke, M. 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.

[Car88] Carroll, M. D. *Data Flow Update via Dominator and Attribute Updates.* PhD thesis, Rutgers University, New Brunswick, NJ, May 1988.

[CCF91] Choi, J.-D., Cytron, R., and Ferrante, J. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, 1991.

[CFR+91] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):452–490, October 1991.

[CR88] Carroll, M., and Ryder, B. G. Incremental data flow update via attribute and dominator updates. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 274–284, January 1988.

[GW76] Graham, S. L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[Kil73] Kildall, G. A. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, 1973. ACM SIGACT and SIGPLAN.

[KU76] Kam, J. B., and Ullman, J. D. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

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

[MR90] Marlowe, T. J., and Ryder, B. G. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196. ACM SIGACT and SIGPLAN, January 1990.

[PS89] Pollock, L., and Soffa, M. L. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 11(4), April 1989.

[Rep82] Reps, T. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages.* ACM SIGACT and SIGPLAN, January 1982.

[RMP88] Ryder, B. G., Marlowe, T. G., and Paull, M. C. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11(1):1–15, October 1988.

[RP86] Ryder, B. G., and Paull, M. C. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[RP88] Ryder, B. G., and Paull, M. C. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[RTD83] Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[SG95] Sreedhar, V. C., and Gao, G. R. A linear time algorithm for placing ϕ-nodes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995. A longer version accepted in Journal of Programming Languages.

[SGL95a] Sreedhar, V. C., Gao, G. R., and Lee, Y. Efficient data flow analysis using DJ graphs: Elimination methods revisited. Technical Report ACAPS Memo 93, McGill University, June 1995.

[SGL95b] Sreedhar, V. C., Gao, G. R., and Lee, Y. Incremental computation of dominator trees. In *ACM SIGPLAN Workshop on Intermediate Representation (in conjunction with the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages)*, January 1995. Also in appear as SIGPLAN Notices Volume 30, Number 4.

[Sre95] Sreedhar, V. C. *Efficient Program Analyses Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, 1995. In preparation.

A Background and Notation

In this section we introduce some background material and notation that are used in the paper.

A.1 Flowgraph Terminology

A flowgraph is a connected directed graph $G = (N, E, \mathrm{START}, \mathrm{END})$, where N is the set of nodes, E is the set of edges, START $\in N$ is a distinguished start node, and END $\in N$ is a distinguished end node. We represent program texts using flowgraphs, where nodes represent basic blocks and edges flow of control. For forward problems (such as Reaching Definitions) flowgraph is the forward control flow graph with START representing program entry point and END representing program exit point. For backward problems (such as Live Uses of Variables) flowgraph is the reverse control flow graph, with START representing program exit point and END representing program entry point. Figure 1(a) shows an example flowgraph.

In a flowgraph, if $x \to y \in E$, then x is called the *source* node and y is called the *destination* node of the edge. We define $Succ(x)$, the successor nodes of x, to be all nodes y such that $x \to y \in E$. Similarly we define $Pred(x)$, the predecessor nodes of x, to be all nodes y such that $y \in x \in E$.

A *path* of length n is a sequence of edges $(x_0 \to x_1 \to \ldots \to x_n)$, where each $x_i \to x_{i+1} \in E$. We will use $P : x_0 \xrightarrow{+} x_n$ to denote a path of length one or more. We will also assume that for every node $x \in N$ there exists a path from START to x, and similarly there is a path from x to END.

If S is a set, we will use $|S|$ to represent the number of elements in the set.

In a flowgraph, a node x **dominates** another node y iff all paths from START to y pass through x. We write x **dom** y to indicate that x dominates y. The dominance relation is reflexive and transitive, and can be represented by a tree, called the dominator tree. If x is a parent node of y in the dominator tree, then x **immediately dominates** y, and we write $idom(y)$ to denote the immediate dominator of y. Figure 1(b) shows the dominator tree for the flowgraph shown in Figure 1(a). We associate a *level number* with each dominator tree node. The level number of a node x, $x.level$, is defined to be its depth from the root.

The **dominance frontier** $DF(x)$ of a node x is the set of all nodes y such that x dominates a predecessor of y, but x does not strictly dominate y [CFR+91]. We can extend the definition of dominance frontier $DF(S)$ to a set of nodes S:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

We define **iterated dominance frontier** $IDF(S)$ for a set of nodes S as the limit of the increasing sequence:

$$IDF_1(S) = DF(S), \tag{7}$$

$$IDF_{i+1}(S) = DF(S \cup IDF_i(S)) \tag{8}$$

A.2 Data Flow Analysis

Data flow analysis is a process of collecting facts about a program from its intermediate representation. These facts or data flow information can be modeled by elements of a semi-lattice L. Associated with each node x is a flow function f_x that maps input information to output information [Kil73, Mar89].[6]

Let $I_x \in L$ be the information at the entry of a node x, and let $O_x \in L$ be the information at the exit of the node. Then the input-output relation can be expressed as

$$O_x = f_x(I_x)$$

We can rewrite this equation to be as follows:

$$O_x = f_x(I_x) = I_x \cap P_x \cup G_x \tag{9}$$

where $P_x, G_x \in L$.[7] We can interpret the above equation as follows: The Ooutput flow information at a node's exit is either (1) what is Ggenerated within the node or (2) what arrives at

[6] For some problems, it is more convenient to associate flow functions with flowgraph edges instead of nodes.

[7] In general G_x may not be a constant. It can also depend on I_x. That is, what information is generated at a node depends on (1) the local data flow information at the node and (2) the input data flow information to the node.

its Input and is Preserved through the node. For convenience, in this paper we will use the following notation instead of the notation in Equation 9.

$$O_x = f_x(I_x) = P_x I_x + G_x \tag{10}$$

where $+$ is the union operation and juxtaposition is the intersection operation.

We need another set of equations to relate the output information at a node y to the input information at x when an edge $y \rightarrow x$ exists. The input information I_x is the merge of all the output information O_y of nodes y in $Pred(x)$; i.e.,

$$I_x = \bigwedge_{y \in Pred(x)} O_y \tag{11}$$

$\bigwedge$ is usually a union or intersection operation, depending upon the data flow problem being solved.

Combining Equations 10 and 11, we obtain the following equation, denoted H_x, for each $x \in N$.

$$\begin{aligned} H_x : O_x &= f_x(\bigwedge_{y \in Pred(x)} O_y) \\ H_x : O_x &= P_x(\bigwedge_{y \in Pred(x)} O_y) + G_x \end{aligned} \tag{12}$$

Since there is one equation for each node, we have a total of $|N|$ equations.

Equation 12 has two variations: The first variation is more general than the second, since for some data flow problems the information generated within a node x is not independent of I_x [Mar89]. We use the term **output variable** to name the variable O_x that appears on the left-hand side (LHS) of equation H_x. Any variable appearing on the right-hand side (RHS) of the equation is called an **input variable**. Furthermore, P_x and G_x are called the **parameters** of the equation.

Given any two equations H_x and H_y, we say that H_x **depends on** H_y if the output variable of H_y appears on the RHS of H_x. That is, we will need the solution of O_y in order to compute the solution of O_x. When we examine the initial set of equations, H_x depending on H_y also indicates that there exists an edge $y \rightarrow x$ in the flowgraph.

Now we will discuss the concept of *variable elimination*, which is fundamental to all elimination methods. Consider the following two equations for example:

$$\begin{aligned} H_y : O_y &= P_y(\bigwedge_{z \in Pred(y)} O_z) + G_y \\ H_x : O_x &= P_x O_y + G_x \end{aligned}$$

In this example, equation H_x depends only on H_y. This also means that x has only one incoming edge $y \rightarrow x$. To eliminate the variable O_y from the RHS of H_x, we can replace it with the RHS of H_y. The resulting H_x equation thus becomes:

$$H_x : O_x \quad = \quad P_x(P_y(\bigwedge_{z \in Pred(y)} O_z) + G_y) + G_x$$

Here we have eliminated the dependence of H_x on H_y but introduced dependences from H_z to H_x for each predecessor of y. In the corresponding flowgraph we also eliminate the edge $y \rightarrow x$ and introduce an edge $z \rightarrow x$ for each $z \in Pred(y)$.

For the more general variation in Equation 12, variable elimination corresponds to function composition. To illustrate this, let us consider the following equations:

$$\begin{aligned} H_y : O_y &= f_y(\bigwedge_{z \in Pred(y)} O_z) \\ H_x : O_x &= f_x(O_y) \end{aligned}$$

After eliminating O_y in H_x we get

$$H_x : O_x \quad = \quad f_x(f_y(\bigwedge_{z \in Pred(y)} O_z))$$

Finally we will define closure operation for recursive equations. If at any node y, the data flow equation is of the form

$$H_y : O_y \quad = \quad mO_y + k = f(O_y), \tag{13}$$

where m and k are terms that do not contain O_y, then the closure operation of this equation is

$$H_y : O_y \quad = \quad f^*(O_y) \tag{14}$$

In other words, O_y is a fixed point of the above recursive equation. Such a fixed point exists if the flow function associated with each node is monotone, and the semi-lattice does not contain an infinite descending chain [Mar89, KU76, Kil73].

For many of the classical problems, such as Reaching Definitions and Available Expressions, f^* can be computed very fast (essentially in constant time) [Mar89]. Ryder and Paull call such closure operations the loop-breaking rules [RP86]. They show how to compute f^* very fast for the classical bit-vector union and intersection problems. Marlowe gives an in-depth treatment on the computational complexity for various classes of functions and domains that can occur in data flow analysis [Mar89]. Throughout our discussion we will assume a monotone data flow framework in which all flow functions are monotone; i.e., for any flow information α and β, $\alpha < \beta \Rightarrow f(\alpha) < f(\beta)$ for any flow function f [Mar89, KU76, Kil73].