Chapter 1

Alternative SSA construction algorithms —(*D. Das, U. Ramakrishna, and V. Sreedhar*)

1 Introduction

recall that SSA construction
falls into two phases:
- phi placement
- renaming

Here we study the first
phase for minimal SSA
ie for phi placed at
IDF(Defs)=J(Defs + r)

Provide the layout without
giving the details
(just name the solutions)

The placement of ϕ-functions is an important step in the construction of Static Single Assignment (SSA) form [**?**]. In SSA form every variable is assigned only once. At control flow merge points ϕ-functions are added to ensure that every use of a variable has exactly one definition. A ϕ-function is of the form $x_n = \phi(x_0, x_1, x_2, \ldots, x_{n-1})$, where x_i's ($i = 0 \ldots n-1$) are a set of variables with static single assignment. Consider the example control flow graph where we have shown definitions and uses for a variable x. The SSA form for the variable x is illustrated in Figure 1(a) without explict renaming. Notice ϕ-functions have been introduced at certain join (also called merge) nodes in the control flow graph. In the rest of this chapter we will present three different approaches for placing ϕ-functions at appropriate join nodes. We will begin by introducing concepts of dominance frontiers and iterated dominance frontiers (IDF). Cytron et al. showed that ϕ-functions for a set of nodes in the control flow graph are placed at exactly the IDF of the set of nodes. Next we present Sreedhar and Gao's algorithm for placing ϕ-functions. Sreedhar and Gao's algorithm uses DJ graph representation of a CFG. Then we will discuss the notion of merge set and merge relation, and present Das and Ramakrishna's algorithm for ϕ-functions based on merge relation and using DJ graph. Finally we describe another approach for placing ϕ-functions based on loop nesting forest.

phi placement
phi function
SSA variables
join/merge points (fig 1a)
layout:
- DF+
- Cytron
- Sreedhar & Gao
- DJ graphs
- merge set
- Das & Ramakrishna
- loop nesting forest based

2 Basic Algorithm

DF (and its construction)
is in the appendix.

The original algorithm for placing ϕ-functions was based on first computing the dominance frontier (DF) set for the given control flow graph. The dominance frontier $DF(x)$ of a node x is the set of all nodes z such that x dominates a predecessor of z, without strictly dominating z. For example, $DF(8) = \{6, 8\}$ in Figure 1(b). A straight forward algorithm for computing DF for each node takes $O(N^2)$ since the size of the full DF set in the worst case can be $O(N^2)$.Here N is the number of nodes of the CFG. Cytron et al.'s algorithm for the placement of ϕ-functions consists of computing iterated dominance frontier (IDF) for a set of all definition points (or nodes where variables are defined). Let N_α be the set of nodes where variable x are defined. Given that the dominance frontier for a set of nodes is just the union of the DF of each node, we can compute $IDF(N_\alpha)$ as a limit of the following recurrence equation (where S is initially N_α)

Cytron => DF
DF (exple & complexity)
iDF

$$IDF_1(S) = DF(S)$$
$$IDF_{i+1}(S) = DF(S \cup IDF_i(S))$$

A ϕ-function is then placed at each join node in the $IDF(N_\alpha)$ set. Although Cytron et al.'s algorithm works very well in practice, in the worst case the time complexity of the algorithm is quadratic in the number of nodes in the original control flow graph.

for one variable

WC complexity of phi placement

3 Placing ϕ-functions using DJ graphs

Sreedhar and Gao proposed the first linear time algorithm for computing the IDF set without the need for explicitly computing the full DF set. Sreedhar and Gao's orginal algorithm was implemented using the DJ graph representation of CFG. A DJ graph is ~~nothing more than a CFG augmented with dominator tree edges~~. An edge $x \rightarrow y$, of the CFG, which is not a dominator-tree edge, becomes a join(J) edge The DJ graph for the example CFG is also shown in Figure 1(b). The CFG edge $10 \rightarrow 8$, which is not a dominator-tree edge becomes the J edge, $10 \xrightarrow{J} 8$. Rather than explicitly computing the full DF set, Sreedhar and Gao's algorithm uses ~~a bottom-up and top-down traversal of~~ the DJ graph to compute the $IDF(N_\alpha)$. Even though the time complexity of Sreedhar and Gao's algorithm is linear, in practice it sometimes performs worse than the Cytron et al. algorithm. The main reason for this is that in practice the size of the DF set is linear and sometimes smaller than the size of the DJ graph.

remove dominance forward edges before

use the same notation than in the figure

emphasize

in what?

in what?

Sreedhar linear uses DJ graphs J-edge

Linear but sometimes slower forward+backward on dom-tree

what is important is the complexity for computing iDF for all variables. So this is confusing.
Also DJ graph (dom-tree) is a pre-requisit to compute DF. So I do not understand the argument.

3.1 Key Observation

Let us try to understand how to compute the DF for a single node using DJ graphs. Consider the DJ graph shown in Figure 1(b). To compute $DF(3)$ we simply walk down the dominator (D) tree edges from node 3 and identify all join (J) edges $x \xrightarrow{J} y$ such the $y.level \leq 3.level$, where *level* of a node is the depth of the node from the root of the dominator tree. The *level* of each node is also shown in Figure 1(b). For our example the only J edge that satifies this condition is $7 \xrightarrow{J} 2$. Therefore the $DF(3) = \{2\}$. To generalize the example, we can compute the DF of a node x using

$$\boxed{DF(x) = \{y | z \in SubTree(x) \ \wedge \ z \xrightarrow{J} y \ \wedge \ y.level \leq x.level\}}$$

It would be helpful to give the property: level(DF(x)) always <= level(x)

use the same notations ie x->y & z for 3 DF(z)=...

DF characterization using subtree and level

Now we can extend the above idea to compute the IDF for a set of nodes, and hence the set of ϕ-functions. Given a set of initial nodes N_α to compute the relevant set of ϕ-functions, Sreedhar and Gao made two key observations: (1) Let y be an ancestor node of a node x on the dominator tree. If $DF(x)$ has already been computed before the computation of $DF(y)$, $DF(x)$ need not be recomputed when computing $DF(y)$. However, the reverse may not be true, and therefore the order of the computation of DF is crucial. ~~(2) When computing $DF(x)$ we only need to examine J edges $y \xrightarrow{J} z$, where $y \in SubTree(x)$ and z is a node such that $z.level \leq x.level$.~~

To illustrate the two key observations consider the example DJ graph in Figure 1(b), and let us compute $IDF(\{8, 9\})$. Now supposing we start with node 8 and compute $DF(8)$ using the second key observation. The resulting DF set is $DF(8) = \{6, 8\}$. Now supposing we next compute the DF set for node 9, and the resulting set is $DF(9) = \{6, 8\}$. Notice here that we visited the edges $9 \xrightarrow{J} 6$ and $9 \xrightarrow{J} 6$ twice, once during the

difficult to understand because you did not give the main lines of the algo

redundant

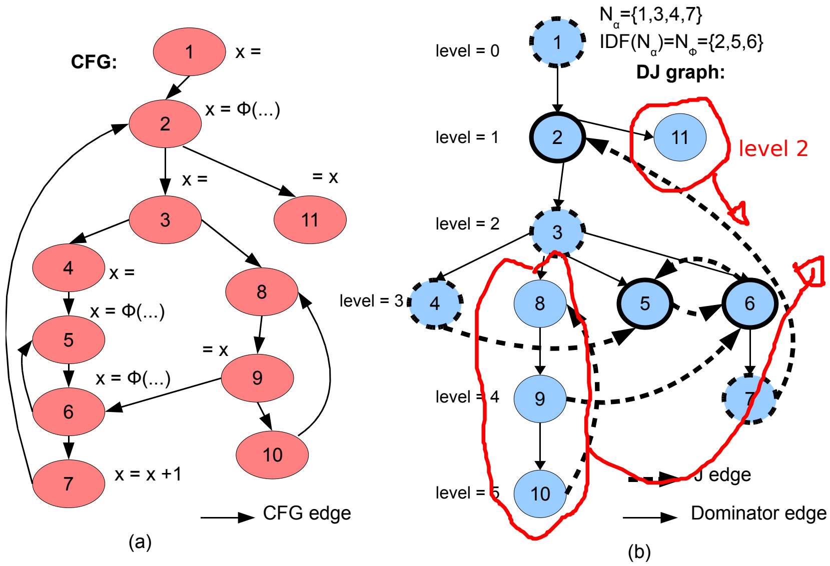
not well chosen example

key property that the ordering is important

From review 5/1:
I think the example at the end of paragraph 3.1 is useless for that. We do not see why we could not first compute DF(8), mark 8, 9, and 6 and then compute DF(9) and stop (because 9 have been traversed already). To see this point maybe you could consider IDF({3,8}) instead.

The key property is the following:
DF(x,x')=DF(x') U {y|z in Subtree(x) \ Subtree(x') AND z -j->y AND y.level <= w.level } whenever x ancestor of x'

You need to outline first that you do not want to precompute DF. You consider the complexity of IDF(N_alpha) not the overall complexity of placing all the phi nodes for all variables. To do so (because this is the _iterated_ dom frontier) you use a working list (initiated with N_alpha) of node for which you need to compute the DF. If x & x' are in the working list, x ancestor of x' then DF(x) does not need to be computed entirely. The nodes reachable from Subtree(x') are already in DF(x').



put it on the right

Fig. 1. A Motivating Example (a) CFG (b) DJ graph

computation of $DF(8)$ and once again during the computation of $DF(9)$. We can avoid such duplicate visits using the first key observation by ordering the computation of DF set so that we first compute $DF(9)$ and then during the computation of $DF(8)$ we avoid visiting the sub-tree of node 9, and use the result $DF(9)$ that was previously computed. In the next section we present the Sreedhar and Gao's algorithm based on the above key observations.

3.2 Main Algorithm

In this section we present Sreedhar and Gao's algorithm. Let $x.level$ be the depth of the node from the root node with $root.level = 0$. To ensure that the nodes are processed according to the first key observation we use a simple array of sets $OrderedBucket$, and two functions defined over the array of sets: (1) InsertNode(n) that inserts the node n in the set $OrderedBucket[n.level]$, and (2) GetNode() that returns a node from the $OrderedBucket$ with highest level number. The complete Sreedhar and Gao's algorithm is shown in Figure 2.

First we insert all nodes in N_α in the $OrderedBucket$. The nodes are processed in a bottom-up fashion over the dominator tree from highest node level to least node level (step 5). The procedure Visit(x) essentially walks down the DJ graph and identifies candidate J edges whose destination node are in the IDF set (step 14). Notice that at step 16 a node is inserted in the $OrderedBucket$ if it was never inserted before. Finally

Sreedhar & Gao algo

initialize and define variables

Sreedhar & Gao's algo

```
Input: A DJ graph representation of a program and $N_\alpha$. Output: The set $IDF(N_\alpha)$.
Procedure IDFMain(Set $N_\alpha$) {
1:     $IDF = \{\}$
2:     foreach ( node $x \in N_\alpha$) do
3:         InsertNode($x$)
4:     end for
5:     while (($z = GetNode()) \neq NULL$)
6:         $croot = z$
7:         $z.visited = true$ ;
8:         Visit($z$)
9:     end while
}
Procedure Visit($x$) {
10:    foreach (node $y \in Succ(x)$) do
11:        if ($x \rightarrow y$ is a J edge) then
12:            if ($y.level \leq croot.level$) then
13:                if ($y \notin IDF$) then
14:                    $IDF = IDF \cup y$
15:                    if ($y \notin N_\alpha$) then
16:                        InsertNode($y$)
17:                    endif
18:                end if
19:            end if
20:        else // visit D edges
21:            if ($y.visited == false$) then
22:                $y.visited = true$
23:                // if($y.boundary == false$)
24:                    Visit($y$)
25:                // end if
26:            end if
27:        end if
28:    end for
}
```

explain that it is not part of Sreedhar's algorithm
This is Pingali's

Fig. 2. Sreedhar and Gao's algorithm for computing IDF set.

at step 24 we continue to process the nodes in the sub-tree by visiting over the D edges. When the algorithm terminates the set IDF will contain the IDF set for the initial N_α.

In Figure 3, some of the phases of the algorithm are depicted for clarity. The $OrderedBucket$ is populated with the nodes 1, 3, 4 and 7 corresponding to $N_\alpha = \{1, 3, 4, 7\}$. The nodes are placed in the buckets corresponding to the levels at which they appear. Hence, node 1 which appears at level 0 is in the zero-th bucket, node 3 is in the second bucket and so on. Since the nodes are processed bottom-up, the first node that is visited is node 7. The successor of node 7 is node 2 and since there exists a J edge $7 \xrightarrow{J} 2$, and the IDF set is empty, the IDF set is updated to hold node 2 according to step 14 of the Visit procedure. In addition, InsertNode(2) is invoked and node 2 is placed in the second bucket. The next node visited is node 4. The successor of node 4 which is node 5 has an incoming

running example

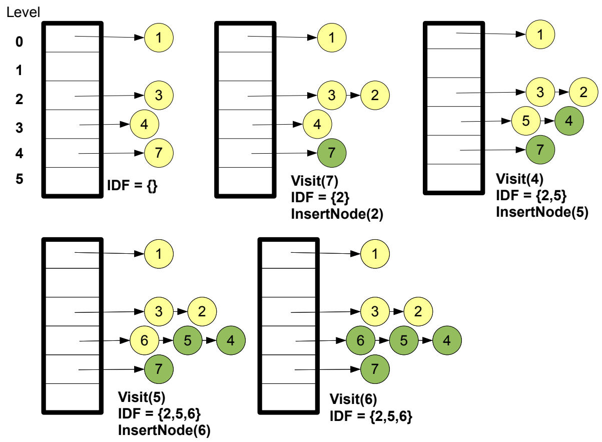


Fig. 3. Phases of Sreedhar and Gao's algorithm for $N_\alpha = \{1, 3, 4, 7\}$.

running example

J edge $4 \xrightarrow{J} 5$ which results in the new $IDF = \{2, 5\}$. The final IDF set converges to $\{2, 5, 6\}$ when node 5 is visited. Subsequent visits of other nodes do not add anything to the IDF set. An interesting case arises when node 3 is visited. Node 3 finally causes nodes 9 and 10 also to be visited. However, when node 10 is visited, its successor node which is node 8 and which also corresponds to the J edge $10 \xrightarrow{J} 8$, does not result in an update of the IDF set as the level of node 8 is higher than that of node 3.

What is precomputed?

which DF set (of one node, a set of nodes, DF-edges)? Not precise enough. We do not understand

I do not understand:
- what are the rules of the game. What do they precompute.
- what is a boundary node. What is it for.
- What do you mean by "cached".

Pingali and Bilardi made one key observation that the DF set can be constructed in space and time proportional to the number of nodes in the CFG and which supports DF query in time proportional to the output size of result of the query. Pingali and Bilardi used a representation called APT(Augmented Postdominator Tree) for representing DF set. The APT representation can be thought as cached DJ graph, where J edges are cached at certain points, called "boundary nodes", in the dominator tree. Given the APT (or cached DJ graph) of a control flow graph, it is straight forward to modify Sreedhar and Gao's algorithm for computing ϕ-functions in linear time. The only modification that is needed is to ensure that we need not visit all the nodes of a sub-tree rooted at a node y. At step 23 if a node y is a boundary node, we avoid visiting the subtree rooted at y. The reason for this is that at a boundary all the required J edges are cached. Finally the time complexity of Sreedhar and Gao's algorithm is linear with respect to the number of DJ graph edges, each edge in the DJ graph is visited at most once. Details of cached DJ graph can be found in the next section.

why "post" dominance?

Pingali & Bilardi's algorithm that use the notion of boundary node and cached DJ-graph

J-edges or DF-edges?

linear in what? Sreedhar is already linear.

phi-functions of one variable?

4 Placing ϕ-functions using Merge Sets

I do not see the difference

In the previous section we described ϕ-function placement algorithm in terms of iterated dominance frontier. There is another way to approach the problem of ϕ-function placement using the concept of merge relation and merge set. In this section we will first introduce the notion of merge set and merge relation and then show how merge relation is related to DF relation. We will then show how to compute M (merge) graph using DJ graph and then use M graph for placing ϕ-functions.

I would remove all this section

4.1 Merge Set and Merge Relation

Join Set

Let us define first the notion of a *join* set $J(S)$ for a given set of nodes S in a control flow graph.[1] Consider two nodes u and v and distinct paths from $u \overset{+}{\to} w$ and $v \overset{+}{\to} w$, where w is some node in the CFG. If the two paths meet only at w then w is in the join set of the nodes $\{\cancel{r}, u, v\}$~~, where r is the root of the CFG~~. For instance, consider nodes 1 and 9 in Figure 1(a). The paths $1 \to 2 \to 3 \to 8$ and $9 \to 10 \to 8$ meet at 8 for the first time and so $\{8\} \in J(\{1, 9\})$. Let $S \subseteq V$ and let $S_r = S \cup \{root\}$. we can compute the join set $J(S_r)$ for all possible pairs $u, v \in S_r$ and determine all such w's where the two distinct paths $u \overset{+}{\to} w$ and $v \overset{+}{\to} w$ will meet for the first time.

why don't you say that M(u)=iDF(u)?

M(u)=J(u,r)=iDF(u)

caracterization of M(u)

Now let us define *merge* relation as a relation $v = M(u)$ that holds between two nodes u and v whenever $v \in J(\{root, u\})$. We insert a ϕ-function at v for a variable that is assigned only at $root$ and u. One can show that $J(S_r) = \cup_{u \in S_r} M(u)$. ~~It is straight forward to show that~~ for any node $u \in V$, $v \in M(u)$ if and only if there is a path $u \overset{+}{\to} v$ that does not contain $idom(v)$. This relationship between dominance and merge can be conveniently encoded using DJ graph and used for placing ϕ-functions. First we will construct DF (Dominance Frontier) graph using DJ graph. For each J edge $u \to v$ in the DJ graph insert a new $w \overset{J}{\to} v$ where $w = idom(u)$ and w does not strictly dominate v. Another way to look at this is to insert a new J edge $w \to v$ if $w = idom(u)$ and $w.level \geq v.level$.[2] We repeatedly insert new J edges in a bottom-up fashion over the DJ graph until no more J edges can be inserted. The resulting DJ graph is a "fully cached" DJ graph, from which we can easily compute the dominance frontiers for a set of nodes. We will call the fully cached DJ graph as the DF graph. Figure 4(a) shows the DF graph that is derived from the DJ graph. Notice that for the J edge $10 \overset{J}{\to} 8$, additional J edges are inserted for the DF graph. These are $9 \overset{J}{\to} 8$ and $8 \overset{J}{\to} 8$ respectively. Given the DF graph we can compute the dominance frontier as $DF(u) = \{v | u \overset{J}{\to} v\}$.

DF graph =
Dom graph +
Dom Frontier edges

+example

Next we can compute the M relation easily from the DF graph. Recall that we insert an edge from a node u to a node v if $v \in M(u)$. Using DF graph we compute the M graph as transitive closure using only the J edges. Now given the M graph of a CFG we can place ϕ-functions to a set $S \subseteq V$ at the neighbouring nodes of the nodes in S. In other words for each node $u \in S$ we place a ϕ-function at v if $u \overset{J}{\to} v$ in M graph. Figure 4(d) shows the M graph where the merge sets $M(x)$ for each node x is shown. For node 8, $M(8) = \{2, 5, 6, 8\}$, as the transitive closure on the DF graph results in the nodes 2, 5, 6 and 8 becoming reachable from node 8 via J edges.

M (iDF), transitive closure of DF.

+example

It can also be shown that the merge, join and IDF are equivalent. Thus, for a node x, $M(x) = IDF(x) = J(x \cup \{root\})$.

[1] In English 'join' and 'merge' are synonyms, but unfortunately in the literature, due to lack of better terms, these two synonyms are used to mean distinct but related concepts.

[2] Recall that $w.level$ is the depth of w from $root$ of the dominator tree.

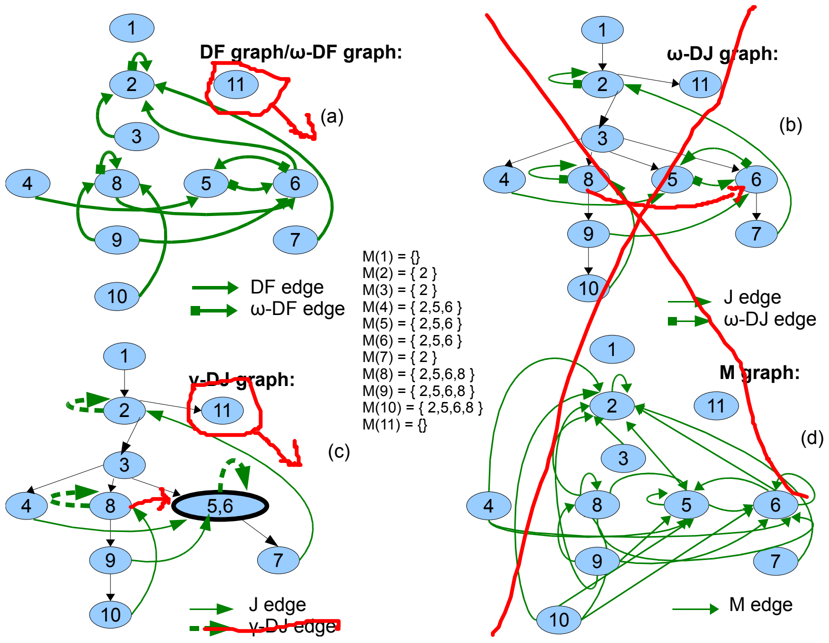


Fig. 4. (a)DF-graph/ω-DF graph (b)ω-DJ graph (c) γ-DJ graph (d)M graph

construction of M graph is quadratic
as soon as we use reduced DJ-graph
or a technique not described in
this chapter.

4.2 Reduced DJ graph

which algorithm?
describe the problem.

The algorithm for placing ϕ-functions based on the construction of DF graph or M graph in the worst case can be quadratic. In this section we will describe an approach for computing ϕ-functions using reduced DJ graph. Recall that Sreedhar and Gao's algorithm for computing ϕ-functions is linear but sometimes perform worse than the one based on DF graph. Sreedhar and Gao's algorithm is a fully lazy algorithm in the sense it does not require explicit computation of DF relation. On the other hand Cytron et al.'s algorithm, which is based on DF graph, explicitly computes DF graph and then uses this graph to compute the set of ϕ-functions. We can first perform a simple preprocessing step to get rid of irreducible cycles using DJ graphs as follows: For each J edge $u \rightarrow v$ in the DJ graph we insert a new J edge $w \rightarrow v$ such that $w.level = v.level$ and w dominates u. We will call the resulting graph as ω-DJ graph. Figure 4(a) shows that certain edges are marked ω-DF edges. These are the edges that correspond to DF when the source and the target nodes are at the same level as defined in [**?**]. The ω-DJ graph shown in Figure 4(b) is an augmented DJ graph with additional edges as described above. It should be noted that the edges marked as ω-DJ edges correspond exactly to the ones marked as ω-DF edges in Figure 4(a).

Sreedhar Gao
do not precompute DF
while Cytron does.

you talk about irreducible
cycles while did not
explain why they are
problematic

The naming J-edge is
confusing here.

w-DF edge ==
DF edge at the same level

Now given the ω-DJ graph we collapse all strongly connected cycles of J edges whose source and destination nodes are at the same level. In other words, find all strongly connected cycles in the ω-DJ graph at a particular level l and collapse them and create a representative node s at level l. Let $rep(s)$ denote the set of nodes for which

s is the representative node at a particular level. Now let $t \in rep(s)$ and consider any outgoing edge $t \to x$ (either D or J) edges of t. We will insert a new corresponding (either D or J) edge from $s \to x$. We will process the nodes in the DJ graph in a bottom-up fashion to construct a reduced DJ graph, which we will call it as γ-DJ graph. The γ-DJ graph in Figure 4(c) is created by merging the nodes/edges of the ω-DJ graph when the strongly connected components are folded to single nodes. In this case, ignoring the self-loops for nodes 2 and 8, the nodes 5 and 6 get folded to a single representative node (5,6), as nodes 5 and 6 form a strongly connected component in the ω-DJ graph. Notice that γ-DJ graph is a reducible graph and so now can efficiently compute ϕ-functions using a simpler approach due to Cytron, Lowry, and Zadeck [**?**]. Ramalingam has given a better exposition of this algorithm which can handle reducible graphs. It is important to notice that all nodes in strongly connected cycles as being equivalent as far as placement of ϕ-functions is concerned. Notice that one can also now use Sreedhar and Gao's algorithm on the γ-DJ graph.

merge all nodes of a SCC of w-edges into one => "reduced graph" which is reducable.

if Ramalingam is better then don't talk about Cytron Lowry, Zadech

?

Why would it be better than others?

5 Fast and Practical ϕ-placement using Merge Sets and DJ graphs

In this section we describe Das and Ramakrishna's algorithm for placing ϕ-functions based on merge relation and using DJ graphs. Das and Ramakrishna made an observation that the ϕ-functions can be placed by storing $M(x)$-sets and using an iterative formulation involving a top-down pass over the DJ-graph. Empirical observation suggests that the space complexity of $M(x)$ sets is comparable to that of DF-sets and hence can be used instead of DF to drive the ϕ- placement.

the top-down pass and iteration is to compute iDF (M) graph.

5.1 Iterative Merge Set Computation

The following definition is needed for explaining Das and Ramakrishna's algorithm.

defines shadow

Shadow: Given a J edge $s \overset{J}{\to} t$, $Shadow(s \overset{J}{\to} t)$ is the set of nodes on the simple path in the dominator tree from node $idom(t)$ to node s, excluding node $idom(t)$ itself i.e., $\forall x \in Shadow(s \overset{J}{\to} t)$, $x\, dominates\, s$ but does not strictly dominate t. Consider Figure 1(b). Let us consider the J edge $9 \overset{J}{\to} 6$. Here the source node is 9 and the target node is 6. $idom(6)$ is 3. $Shadow(9 \overset{J}{\to} 6)$ consists of the path $8 \to 9$. Similarly, for $7 \overset{J}{\to} 2$ the path is $2 \to 3 \to 6 \to 7$, while for $4 \overset{J}{\to} 5$ the path consists of only node 5. Note that the concept of *Shadow* is used (below) to capture the DF values of the nodes and has similarities with the cached-DJ graph mechanism.

The following property of DJ graphs is important for computing the $M(x)$ values of nodes:

$$\boxed{\forall x \in V, DF(x) = \bigcup_{s \overset{J}{\to} t} \{t | x \in Shadow(s \overset{J}{\to} t)\}}$$

provides DF^-1

This property points to how the DF value of a node x and the J edges are related. The DF value is nothing else but a conglomeration of the target nodes of the J edges in whose shadow node x lies. Also note that for any target node t, $level(t) \leq level(s)$. Thus, for a node x all the nodes in the $DF(x)$ lie at a level equal or lower than x. In Figure 1(b),

$DF(6) = \{2, 5\}$ as node 6 belongs to $Shadow(7 \xrightarrow{J} 2)$ and $Shadow(6 \xrightarrow{J} 5)$. Here node 6 is at the same level as 5 while node 2 is at a lower level.

Since $M(x) = IDF(x) = DF^+(x), x \in V$, we can define the merge set for node x by the following recursive definition :

$\boxed{M(x) = DF(x) \bigcup (\cup_{y \in DF(x)} M(y)) \text{ and } level(y) \le level(x)}$

M is transitive closure of DF. DF goes always up (or stay at the same level)

Here, as $level(y) \le level(x)$, and since we should enumerate the values of $M(y)$, before we can enumerate $M(x)$, we should visit the nodes top-down in the DJ graph. Now, we can write, using the above property of $DF(x)$, substituting for $DF(x)$:

$\boxed{M(x) = \bigcup_{s \xrightarrow{J} t} (\{t | x \in Shadow(s \xrightarrow{J} t)\} \cup M(t))}$

redundant

We can build up the $M(x)$ values incrementally for each node x lying in the shadow of J edges - as each J edge is encountered. This is done by capturing the $M(t)$ values of the target nodes of those J edges and then perforing a union operation with the target nodes for constructing the corresponding $M(x)$. Thus, if $M^{s \xrightarrow{J} t}(x)$ is the contribution of a J edge $s \xrightarrow{J} t$ to $M(x)$, then we can write:

x in shadow(s->t) add M(t) and {t} to M(x)

$\boxed{M^{s \xrightarrow{J} t}(x) = \{t\} \cup M(t)}$

As, $\boxed{M(x) = \bigcup_{s \xrightarrow{J} t} M^{s \xrightarrow{J} t}(x)}$, substituting from above:

$\boxed{M(x) = \bigcup_{s \xrightarrow{J} t} (\{t\} \cup M(t))}$

Thus, $M(9) = \{6 \cup M(6)\} \cup \{8 \cup M(8)\}$, since node 9 is in $Shadow(9 \xrightarrow{J} 6)$ and in $Shadow(10 \xrightarrow{J} 8)$. The $M(6)$ and $M(8)$ values should reach their fixed-points solutions before $M(9)$ can reach its fixed-point solution. Proceeding top-down over the dominator tree level by level allows us to compute the $M(x)$ for a node x in a single pass. However, iterative computation may cause multiple passes for cases where cycles of edges appear in the DF graph at the same level - like the ones involving nodes 5 and 6 in Figure 1(b). The cycles are not collapsed(as in the reduced DJ graph case) as they may turn out to be costly in some cases.

possible circuits at the same level

The following definition is needed to find out whether multiple passes are required for the algorithm.

we do not understand why shadow is conceptually usefull compared to DF^-1

Inconsistency Condition: If the *source*(s) and *target*(t) of a J-edge do not satisfy $M(s) \supseteq M(t)$, then the nodes in the shadow of the J-edge are said to be inconsistent. This directly follows from the statement that $M(s) = \bigcup_{s \xrightarrow{J} t} (\{t\} \cup M(t))$, as $s \in Shadow(s \xrightarrow{J} t)$.

iteration necessary when M(x) grows

we do not understand what is iterated (the whole traversal?)

The algorithm described in the next section is directly based on the above method of building up the $M(x)$ sets of the nodes as each J edge is encountered in an iterative fashion by traversing the DJ graph top-down. If no node is found to be **inconsistent** after a single top-down pass, all the nodes are supposed to have reached fixed-point solutions. If some node is found to be inconsistent, multiple passes are required till fixed-point solutions can be reached.

top-down with possible iterations

we do not understand what are all those notations for

This is nothing else than dataflow analysis along shadow. Whenever the content of a node is modified if the shadow set of DF edges have already been considered in this pass, checks if it should be considered again (so iteration is required).
You should probably define shadow as a set of DF edges instead so that you do not have to reexplain all the notions (already well-known) again in another context.

define and initialize all variables that you use

Input: A DJ graph representation of a program. **Output:** The merge sets for the nodes.

```
Procedure TopDownMergeSetComputation_TDMSC-I(DJ graph) {
1:  $RequireAnotherPass = False$;
2:  while (in B(readth) F(irst) S(earch) order) do
3:    $n$ = Next Node in BFS list
4:    for (all incoming edges to $n$) do
5:      Let $e = s \xrightarrow{J} n$ , be an incoming J edge
6:      if ($e$ not visited) then
7:        Visit($e$)
8:        $tmp = s$;
9:        $lnode = NULL$;
10:       while ($level(tmp) \geq level(t)$) do
11:         $M(tmp) = M(tmp) \cup M(tnode) \cup \{t\}$;
12:         $lnode = tmp$;
13:         $tmp = parent(tmp)$; //dominator tree parent
14:       end while
15:       for (all incoming edges to $lnode$) do
16:         Let $e' = s' \xrightarrow{J} lnode$, be an incoming J edge
17:         if ($e'$ visited) then
18:           if ($M(s') \not\supseteq M(lnode)$) then //Check inconsistency
19:             $RequireAnotherPass = True$;
20:           end if
21:         end if
22:       end for
23:     end if
24:   end for
25: end while
26: return $RequireAnotherPass$;
}
```

of what?

?

propagate on DF^{-1}

check for consistency along already visited J-edges.

Fig. 5. Top Down Merge Set Computation algorithm for computing Merge sets.

5.2 TDMSC-I

TDMSC-I works by scanning the DJ graph in a top-down fashion as shown in step 2 using a breadth-first traversal mechanism. This implies that the DJ graph is visited level-by-level. During this process, for each node n encountered, if there are **incoming** J edges to n as in step 4, then a separate bottom-up pass starts at step 10. This bottom-up pass traverses the nodes in the $Shadow(s \xrightarrow{J} n)$ updating the $M(x)$ values of these nodes. This follows from the contribution of each J edge $M^{s \xrightarrow{J} t}(x)$ to $M(x)$ as discussed earlier. step 15 is used for inconsistency check. $RequireAnotherPass$ is set to true only if a fixed point is not reached and the inconsistency check succeeds for some node.

There are some subtleties in the algorithm that should be noted. step 15 of the algorithm visits incoming edges to $lnode$ only as the the incoming edges to $lnode$'s posterity are at a level greater than that of node n and are unvisited yet.

description of the algo (did not check it)

Here, we will briefly walk through TDMSC-I using the DJ graph of Figure 1(b). Moving top-down over the graph, the first J edge encountered is $7 \xrightarrow{J} 2$. As a result, a bottom-up climbing of the nodes happen, starting at node 7 and ending at node 2 and the merge sets of these nodes are updated so that $M(7) = M(6) = M(3) = M(2) = \{2\}$. The next J edge to be visited can be any of $4 \overset{J}{\rightarrow} 5$, $5 \overset{J}{\rightarrow} 6$ or $6 \overset{J}{\rightarrow} 5$. Assume it is $5 \overset{J}{\rightarrow} 6$. This results in $M(5) = M(6) \cup \{6\} = \{2, 6\}$. Now, let $6 \overset{J}{\rightarrow} 5$ be visited. Hence, $M(6) = M(5) \cup \{5\} = \{2, 5, 6\}$. At this point, the **inconsistency check** comes into picture for the edge $6 \overset{J}{\rightarrow} 5$ as $5 \overset{J}{\rightarrow} 6$ is another J edge that is already visited and is an incoming edge of node 6. Checking for $M(5) \supseteq M(6)$ fails, implying that the $M(5)$ needs to be computed again. This is done in a separate pass as suggested by the *RequireAnotherPass* value of true. In a second iterative pass, the J edges are visited in the same order. Now, when $5 \overset{J}{\rightarrow} 6$ is visited, $M(5) = M(6) \cup \{6\} = \{2, 5, 6\}$. The $M(5)$ value is different in this pass as $M(6)$ has become $\{2, 5\}$ instead of just $\{2\}$ in the first pass. On a subsequent visit of $6 \overset{J}{\rightarrow} 5$, $M(6)$ is also set to $\{2, 5, 6\}$. The inconsistency does not appear any more and the algorithm proceeds to handle the edges $4 \overset{J}{\rightarrow} 5$, $9 \overset{J}{\rightarrow} 6$ and $10 \overset{J}{\rightarrow} 8$ which have also been visited in the earlier pass. **TDMSC-I** is invoked repeatedly by a different function which calls it in a loop till *RequireAnotherPass* is returned as $false$.

running example
(did not check it)

The algorithm of TDMSC-I is enclosed in another algorithm that repeatedly calls it till **RequireAnotherPass** is false.

TDMSC-II Das and Ramakrishna term as TDMSC-II, an improvement to algorithm TDMSC-I. This improvement is fueled by the observation that for an inconsistent J edge, the merge sets of all nodes in the shadow of that edge can be locally corrected for some special cases. This heuristic works very well for certain class of problems – especially for CFGs with DF graphs having cycles consisting of a few edges. This eliminates an extra pass as an inconsistent node is made consistent immediately on being detected.

Improvement (not described)
for small circuits.

Das and Ramakrishna's TDMSC-II has this additional step immediately after the inner while loop of TDMSC-I.

6 Computing Iterated Dominance Frontier Using Loop Nesting Forests

This section illustrates the use of **loop nesting forests** for construction of the iterated dominance frontier (IDF) of a set of vertices in a CFG, which may contain both reducible as well as irreducible loops.

6.1 Loop nesting forest

loop nesting forests are
used in other chapters.
will be defined (using
Ramalingam's formulation)

Loop nesting forest is a data structure that respresents the loops in a CFG and the containment relation between them. However, while there is a fairly accepted notion of loop nesting forest in a reducible graph [**?**], there is less agreement about their definition in arbitrary graphs. Steensgard [**?**], Sreedhar [**?**] and Havlak [**?**] all provide different definitions which have merits under different situations. For the example shown in Figure 6(a) the loops with backedges 11 → 9 and 12 → 2 are both reducible loops. The

loop nesting forest

corresponding loop nesting forest shown in Figure 6(b), consists of two loops with their header nodes being 2 and 9. The loop with header node 2 contains the loop with header node 9.

loop nesting forest
on an example
(did not check)

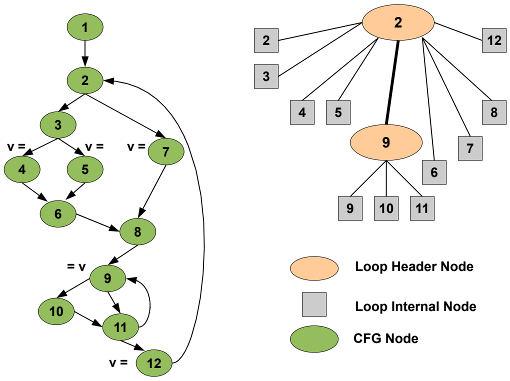


Fig. 6. An example (a) CFG and (b) IDF Computation Using Loop Nesting Forest

instead of saying "can be computed as"... say that iDF have been developped to make use of the notion of DF, but iDF is nothing else than J(X + r). It is possible to reason directly on Join sets. This leads to a simple algorithm based on forward dataflow algorithm that you describe in english in a few sentences.

6.2 Main Algorithm

always use the same notations such as N_alpha.

you can give the principles of the algo directly.

The IDF for a set of vertices X in an acyclic graph G_{ac}, where a certain variable v is "defined", can be computed as the set of nodes where multiple "definitions" of v reach. Let us say that a definition node d "reaches" another node u if there is a path in the graph from d to u which does not contain any definition node (for v) other than d or u. Hence, at least one definition node must reach any vertex $u \neq entry(G)$ [3]. If at least two definitions reach a node u, then u belongs to $IDF(X$ ~~$\cup\, entry(G)$~~$)$. This suggests the algorithm in Figure 7 which visits nodes of G_{ac} in toplogical order and computes their reaching definitions (for a variable v) in step 7 or step 9. If the number of reaching definitions is unique as in step 12, then the reaching definition set of the node is updated with the single reaching definition. Vertices with more than one reaching definition are added to $IDF(X$ ~~$\cup\, entry(G)$~~$)$. It should be noted that a vertex is either a definition node or has a unique reaching definition if it is not in $IDF(X$ ~~$\cup\, entry(G)$~~$)$.

On a DAG for a given set X, compute IDF(X) as follow:
- initialize IDF to emptyset
- dataflow compute using topological order the subset of IDF+X+r that can reach a node
- add a node to IDF if reachable from multiple nodes

For Figure 6, the G_{ac} is formed by dropping the backedges 11 → 9 and 12 → 2. Also, $entry(G) = entry$. For the definitions of v in nodes 4, 5, 7 and 12 in Figure 6, the subsequent nodes where multiple definitions reach turn out to be at 6 and 8. For node 6 any one of two definitions in nodes 4 or 5 can reach. For node 8, it can be either the definition from node 7 or one of 4 or 6. Note that the backedges do not exist in the acyclic graph and hence node 2 is not part of the IDF set. We will see later how the IDF set for the entire graph is computed by factoring in the contribution of the backedges.

running example

[3] Here $entry(G)$ is a dummy node for all those nodes without predecessors

this technique is (for a DAG) the same as C. Click. I have been asked to describe it. C. Click's technique is minimal for DAG but loops are not handled optimally (this allows me to talk about making a non minimal SSa form, minimal). So it differs slightly from this. We should discuss what we do regarding this technique to avoid redundancies.

```
Input: An acyclic CFG $G_{ac}$, $X$: subset of $V(G)$. Output: The set $IDF(X)$.
Procedure ComputeIteratedDominanceFrontier($G_{ac}$:Acyclic graph,$X$: subset of $V(G)$) {
1:   $IDF = \{\}$;
2:   $\forall$ $node$ of $V(G)$, $UniqueReachingDefs(node) = \{\ \}$;
3:   for $\forall u$ of $V(G) - entry(G)$ in topological sort order do {
4:     $ReachingDefs = \{\}$;
5:     for $\forall$ predecessor $v$ of $u$ do {
6:       if $v \in (\ IDF \cup X \cup entry(G)\ )$ then
7:         $ReachingDefs = ReachingDefs \cup \{v\}$;
8:       else
9:         $ReachingDefs = ReachingDefs \cup UniqueReachingDefs(v)$;
10:      end if
11:    end for
12:    if ( $\|ReachingDefs\| == 1$ ) then
13:      $UniqueReachingDefs(u) =$ only element of $ReachingDefs$;
14:    else
15:      $IDF = IDF \cup \{u\}$;
16:    end if
17:  end for
18:  return $IDF$;
}
```

pseudo code on a DAG

Fig. 7. Ramalingam's algorithm for computing the IDF of an acyclic graph.

A smaller example is probably enough

We will walk through some of the steps of this algorithm for $IDF(\{4,5,7,12\})$. We will start at vertex 4. For this vertex, $predecessors = \{3\}$. But as $3 \notin IDF \cup X \cup \{entry\})$, $ReachingDefs = UniqueReachingDefs(3)$ according to step 9. Hence, $ReachingDefs = \{entry\}$. As $\|ReachingDefs\|$ equals 1, $UniqueReachingDefs(4) = \{entry\}$. The same logic applies for vertices 5 and 7, and so, $UniqueReachingDefs(5) = UniqueReachingDefs(7) = \{entry\}$. For vertex 6, $predecessors = \{4,5\}$. Since both vertices 4 and 5 belong to X, according to step 7, $ReachingDefs = \{4,5\}$. As $\|ReachingDefs\| \neq 1$, according to step 15, $IDF = \{6\}$. Following similar arguments, $IDF = \{6,8\}$, when node 8 is visited. When the rest of the vertices are visited, the IDF set remains unchanged.

running example (cont.)
(did not check)

How can the algorithm to find IDF for acyclic graphs be extended to handle reducible graphs? A reducible graph can be decomposed into an acyclic graph and a set of backedges. The contribution of backedges to the iterated dominance frontier can be identified by using the loop nesting forest. If a vertex u is contained in a loop then $IDF(u)$ will contain the loop header. For any vertex u, let $HLC(u)$ denote the set of loop headers of the loops containing u. Given a set of vertices X, it turns out that $IDF(X) = HLC(X) \cup IDF_{ac}(X \cup HLC(X))$ where $IDF(X)$ and IDF_{ac} denote the IDF over the original graph G and the acyclic graph G_{ac} respectively. Reverting back to Figure 6 we see that in order to find the IDF for the nodes where the variable v is defined, we need to evaluate $IDF(\{4,5,7,12\})$. Firstly, we would need to evaluate $IDF_{ac}(\{4,5,7,12\} \cup HLC(\{4,5,7,12\}))$. $HLC(\{4,5,7,12\}) = \{2\}$ as all these nodes are contained in a single loop with header 2. Hence, we would need to find $IDF_{ac}(\{2,4,5,7,12\})$ which turns out to be the set $\{6,8\}$. Finally, $IDF(\{4,5,7,12\}) = HLC(\{4,5,7,12\}) \cup \{6,8\} = \{2,6,8\}$.

on a reducible graph
add the loop headers

running example
(did not check)

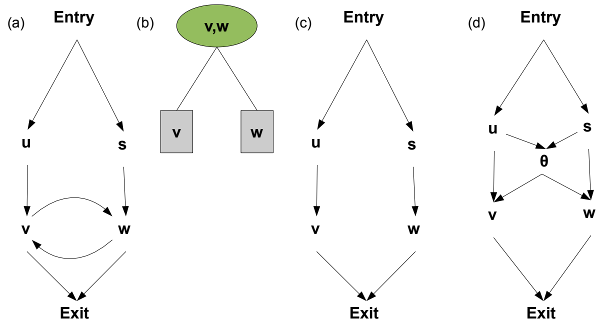


Fig. 8. (a) An irreducible graph (b) The Loop Nesting Forest (c) The acyclic subgraph (c) Transformed graph

Now that we have shown how IDF can be computed for graphs with reducible loops using loop nesting forests, we will briefly touch upon how graphs containing irreducible loops can be handled. The trick behind the implementation is to transform the irreducible loop in such a way that an acyclic graph is created from the loop without changing the dominance properties of the nodes. Referring to Figure 8, we see that the irreducible loop comprising of nodes v and w in (a) can be transformed to the acyclic graph in (c) by removing the edges between nodes v and w that create the irreducible loop. We can now create a new dummy node θ and add edges from the predecessor of all the header nodes of the cycles of the irreducible graph to θ, as well as add edges from θ to all the header nodes. This results in additional edges from u and s which are the predecessors of the header nodes v and w respectively to θ. In addition, extra edges are added from θ to the header nodes v and w.The transformed graph is shown in (d). Following this transformation, computing IDF for the nodes in the original irreducible graph translates to computing IDF for the transformed graph. Since this graph is acyclic, the algorithm depicted in Figure 7 can be applied by noting the loop nesting forest structure as shown in (b). The crucial observation that allows this transformation to create an equivalent acyclic graph is the fact that the dominator tree of the transformed graph remains identical to the original graph containing an irreducible cycle. One of the drawbacks of the transformation is the possible explosion in the number of dummy nodes and edges for graphs with many irreducible cycles as a unique dummy node needs to be created for each irreducible loop. However, such cases may be rare for practical applications.

irreducible graph to reducible with dominance maintained (additional header node)

run the same algo...

note concerning the complexity

7 Summary

This chapter provides an overview of some advanced construction algorithms for SSA. These include the first proposed linear-time algorithm by Sreedhar using the concept of DJ graphs, the merge relation by Pingali and Bilardi, a DJ graph and merge set based algorithm by Das and Ramakrishna and finally an algorithm by Ramalingam based on the concept of loop nesting forests. Though all these algorithms claim to be better than the original CFR algorithm, they are difficult to compare due to the unavailability of these algorithms in a common compiler framework. However, some of these algorithms are known to perform better than CFR for several well-known benchmarks.

summary.
do not know which is better
in practice

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References