VISVESWARAYA TECHNOLOGICAL UNIVERSITY

BELGAUM

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING, MYSORE-570006

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Report on

IMPLEMENTATION OF STATIC SINGLE-ASSIGNMENT (SSA) FORM

Guidance of

N.R. PRASHANTH
Professor
Department of CS&E, SJCE, Mysore.

Done By:

VIKRAM T.V.
5th Semester,
Computer Science and Engineering,
S.J.C.E, Mysore

Contents

1 Introduction **1**

2 Converting to SSA Form **1**
2.1 Dominators 1
2.2 The Dominance Frontier 1
2.3 Inserting ϕ-functions 2
2.4 Renaming the Variables 2

3 Some examples **2**

1 Introduction

Data flow analysis needs the *def-use* chain to find the use-sites of each defined variable or the def-sites of each variable used in an expression. An improvement on the idea of def-use chains is *static single-assignment form*, or SSA form, which is an intermediate form in which each variable has only one definition in the program text. The static definition-site may be in a loop that is executed many times, thus the name - static single assignment form, where the variables are never redefined. Some of the advantages of using SSA form are: optimizations and dataflow analysis becomes simpler, space complexity reduces compared to def-use chains and unrelated uses of same variable (inside a loop) in source program become different in SSA form, eliminating needless relationships.

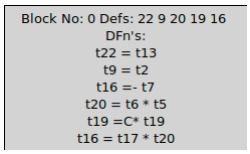

Figure 1: A straight line program

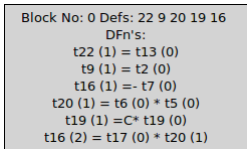

Figure 2: Program in SSA form

Figure 1 shows a straight line program and figure 2 shows the corresponding program in SSA form. When two control flow paths merge together, with each path having a definition of a as $a1$ (across left path) and $a2$ (across right path), it is essential to have only one definition of a at the merge point to have the program in SSA form. A new definition of a, say $a3$, is defined at the merged point which is the result of ϕ-function. Thus the statement $a3 \leftarrow \phi(a1, a2)$ defines $a3$ with $a1$, if the control flow was through left path, or with $a2$ if control flow took right path. The ϕ functions are eliminated before generating executable by placing MOVE instructions on each incoming edge with corresponding definitions. Figure 3 shows an example of ϕ placement for the variable t13.

2 Converting to SSA Form

Converting a program to SSA form involves 4 stages: creating dominators for each basic block, find the places where ϕ functions have to be placed, placing of ϕ functions and finally renaming of uses and definitions of variables. The input program is the intermediate representation represented using basic blocks and control flow graph.

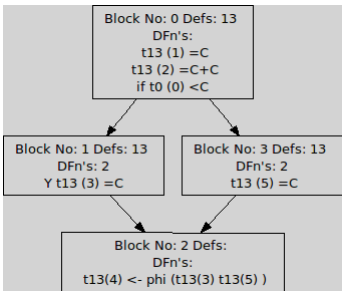

Figure 3: ϕ-function placement

2.1 Dominators

The dominators for the basic blocks of the input program are calculated using the Lenguer-Tarjan algorithm. An immediate dominator is found for each of the nodes. This is used to find places needed for ϕ placement.

2.2 The Dominance Frontier

Finding places for ϕ placement can be calculated iteratively. The criterion for this is to find a blocks x and y (with $x \neq y$) containing definitions of a. A ϕ function should be at node z, if there are non-empty paths P_{xz} and P_{yz} and both paths do not have any node in common other than z. Also node z does not appear within both paths. But we can use the dominance properties on SSA form to find a better method of placing ϕ functions, known as the Dominance Frontier criterion, which relies on following properties:

- If x is used in a ϕ-function in block n, then the definition of x dominates every predecessor of n.
- If x is used in a non-ϕ-statement in block n, then the definition of x dominates n.

The dominance frontier is calculated using the dominator tree of the flow graph. x *strictly dominates* w if x dominates w and $x \neq w$. A dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w. A dominance frontier of a node is the border between the dominated and undominated nodes, and it is here the ϕ-functions have to be placed. Hence, the criterion is: whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a ϕ-function for a.

Computing the dominance frontier: To insert all the necessary ϕ-functions, for every node n in the flow graph, DF[n] is needed, its dominance frontier, which can be computed efficiently in one pass using the dominator tree. The following sets are used:

- DF_{local}[n]: The successors of n that are not strictly dominated by n.
- DF_{up}[n]: Nodes in the dominance frontier of n that are not strictly dominated by n's immediate dominator.

Algorithm 1 Psuedocode for computeDF[n]

```
1: computeDF[n] ( )
2: S ← { }
3: for each node y in successor[n] do
4:    if idom(y) ≠ n then
5:       S ← S ∪ y
6:    end if
7: end for
8: for each child c of n in the dominator tree do
9:    computeDF[c]
10:   for each element w of DF[c] do
11:      if n does not dominate w or n = w then
12:         S ← S ∪ w
13:      end if
14:   end for
15: end for
16: DF[n] ← S
```

The dominance frontier of n can be computed from DF_{local} and DF_{up}:

$$\mathrm{DF[n]} = \mathrm{DF}_{local}\mathrm{[n]} \bigcup \cup_{c \in children[n]} \mathrm{DF}_{up}[c]$$

where children[n] are the nodes whose immediate dominator is n. Using immediate dominators DF_{local}[n] is calculated as the set of those successors of n whose immediate dominator is not n. A pseudocode to compute DFn is given in Algorithm 1. Its complexity is proportional to the number of edges of the original graph plus the size of the dominance frontiers it computes. But practically, the control flow graphs for the programs are not that complex and hence this psuedocode runs in linear time.

2.3 Inserting ϕ-functions

A pseudocode for inserting ϕ-functions is shown in Algorithm 2. It uses worklists to avoid reexamining of nodes where no ϕ-function has been inserted. First the definition sites of the variables in each block is computed. A_{orig}[n] contains the variable definitions at node n. A_{ϕ}[n] gives the variables that have ϕ-functions at node n. Y contains a dominance frontier of node n. For each variable a, the definition sites of a are loaded into the worklist. For each node n in the worklist, until the worklist becomes empty, for each node in DF[n], if variable a is not in A_{ϕ}[n], then insert the statement a ← ϕ(a, a, ..., a) at the top of block Y, where ϕ-function has as many arguments as Y has predecessors. Add the variable a to A_{ϕ}[Y] and check if a is not in A_{orig}[n], that is if there is no definition of a, then add the current node into the worklist.

For a program of size of N, the work done for each node and edge in control flow graph and for each statement in program are proportional to N. For each element of every dominance frontier, the complexity is approximately linear in N and the number of inserted ϕ-functions could be N^2 in worst case, but practically it is proportional to N. Thus, the algorithm works in linear time.

Algorithm 2 Psuedocode for Inserting ϕ-functions

```
1: Place-φ-functions ( )
2: for each node n do
3:    for each variable a in A_orig[n] do
4:       defsites[a] ← defsites[a] ∪ n
5:    end for
6: end for
7: for each variable a do
8:    W ← defsites[a]
9:    while W not empty do
10:      remove some node n from W
11:      for each Y in DF[n] do
12:         if a ∉ A_φ[Y] then
13:            insert the statement a ← (a, a, ..., a) at the
               top of block Y, where the φ-function has as
               many arguments as Y has predecessors
14:            A_φ[Y] ← A_φ[Y] ∪ a
15:            if a ∉ A_orig[n] then
16:               W ← W ∪ Y
17:            end if
18:         end if
19:      end for
20:   end while
21: end for
```

2.4 Renaming the Variables

After ϕ-function placement, renaming of different definitions (including ϕ-function definitions) of variable a to *a1*, *a2*, *a3* and so on are done using the dominator tree. Each use of a is renamed to use the closest definition of a that is above a in the dominator tree and this handling is done using separate stack for each variable. For each statement S in each block n, if the statement is not a ϕ-function then replace each use of the statement with its latest definition. That is replace the use x with x_i, where i denotes the latest definition of x. For each definition in each statement (including the ϕ-functions, increment the count of the new definition and push it (i) onto corresponding stack. Replace definition of a with a_i. For each successor Y of block n, if n is the jth predecessor of Y, then replace the jth use of variable a in ϕ-function at block n with a_i, where i is the latest defined value of a. A psuedocode is given in algorithm 3.

This algorithm takes time proportional to the size of the program (after ϕ-functions are inserted), so in practice the algorithm runs in linear time. Finally, if the Lenguer-Tarjan algorithm is implemented efficiently with approximate linear complexity, then all the four stages of SSA works in around linear time, making the whole SSA implementation to work in linear time.

3 Some examples

Figure 3 shows the control flow graph after ϕ-function placement at node 2 for variable t13. Variable t13 is defined along both the paths of the if-else construct - once

Algorithm 3 Psuedocode for Renaming Variables

```
1: initialize ( )
2: for each variable a do
3:    count[a] ← 0
4:    stack[a] ← empty
5:    push 0 onto stack[a]
6: end for
7: rename (n)
8: for each statement S in block n do
9:    if S is not a φ-function then
10:      for each use of some variable x in S do
11:         i ← top(stack[x])
12:         replace the use of x with x_i in S
13:      end for
14:   end if
15:   for each definition of some variable a in S do
16:      count[a] ← count[a] + 1
17:      i ← count[a]
18:      push i onto stack[a]
19:      replace definition of a with definition of a_i in S
20:   end for
21: end for
22: for each successor Y of block n do
23:   suppose n is the jth predecessor of Y
24:   for each φ-function in Y do
25:      suppose the jth operand of the φ-function is a
26:      i ← top(stack[a])
27:      replace the jth operand with a_i
28:   end for
29: end for
30: for each child X of n do
31:   rename(X)
32: end for
33: for each definition of some variable a in the original
    S do
34:   pop stack[a]
35: end for
```

along the 'if' path and along 'else' path. Thus, there needs a ϕ-function at the convergence of the two paths - node 2. All the figures show the node or block numbers, variables defined in the corresponding block and the DF[node]. In the above example, DF[1] is 2, which means there needs to be a ϕ-function placement at node 2.

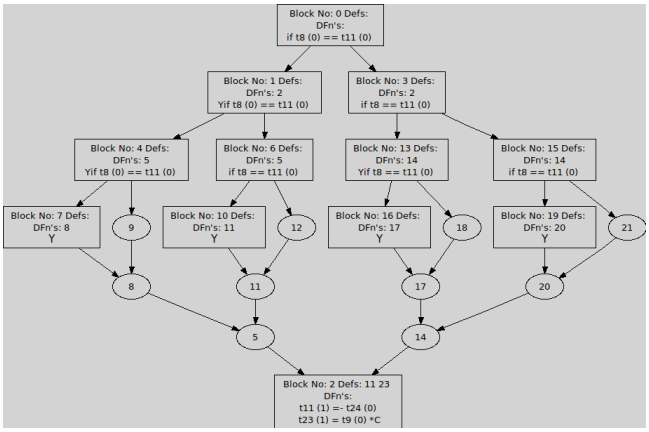

Figure 4: A complex if-else construct

Figure 4 shows the graph for a complex if-else construct with no necessity of ϕ-function placement and figure 5 shows ϕ-function placement for a simple loop.

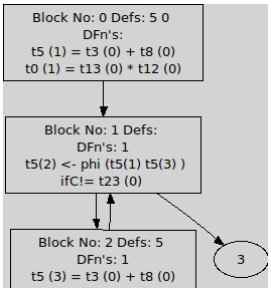

Figure 5: ϕ-function placement for a loop

References

[1] appel andrew w. appel, *modern compiler implementation in c*, Revised Edition, Cambridge University Press, 2008