Lecture 16: Static Single Assignment

I. Motivation for SSA
   A. Ideally we want to do local optimization over the whole routine
      1. Between blocks, not just within a block
      2. But that means we need to know what temporaries have what values throughout the routine
      3. This means we need to know what definition(s) of a temporary reach what uses
         a. I.e. computing reaching definitions
         b. And then doing all the optimizations taking these into account
   B. Using reaching definitions
      1. Inside a block, when a temporary is used, you have to look at all reaching definitions of that temporary
      2. Then you have to consider if they have the same value (using value numberings)
      3. Then you can do value numbering, constant propagation and identities
      4. The reaching definitions need to be maintained throughout optimization
         a. As the code changes
         b. Either a full recomputation after each change
         c. Or complex incremental updates that need to propagate through the flow graph
      5. Overall this makes optimization quite complex
   C. To make this simpler
      1. Suppose there is only a single definition for each temporary
      2. Then if a variable is used, we know which definition was used
      3. Value numbering, etc. only has to look at the single definition
      4. A variable is used if it is used
      5. No data flow computation needs to be done; thus no incremental updates need to be maintained
   D. Programs that only do single assignments (single definition) sound useless
      1. No loops, conditional assignments, ...
2. It took a while, but around 1990, people figured out how to achieve this.
3. The result is Static Single Assignment (SSA) form.

II. SSA Basics
A. Make each assignment yield a unique temporary
   1. Each instruction has a unique temporary as its target operand.
   2. We can do this within a block
      a. Suppose there are multiple assignments to some temporaries.
      b. Replace the first one with a new temporary.
      c. Replace all uses of the temporary up to the next definition with the new temporary.
      d. The result has the same semantics as before.
      e. But there are fewer multiple assignments.
      f. This can be repeated until there are no multiple assignments.

   3. How do we generalize this across blocks?
      a. With the tools we have, this can’t be done.
      b. Thus we need a new tool, or a new instruction really.

B. Adding Φ (PHI) operators or instructions
   1. Problems arise when the same temporary is defined in two different blocks and those definitions can reach a common place.
   2. Consider the first block that these definitions can reach in common.
   3. Add a new instruction at the start of that block.
      a. \( T_{new} = \Phi(T1,T1) \)
      b. The semantics of this operator are to take the value from the temporary whose block was used.
         i. This can’t be implemented easily.
         ii. But we’ll worry about this later on.
         iii. By converting out of SSA form.
   4. This allows us to replace each \( T1 \) with a different temporary.
      a. This removes the common definitions in multiple blocks.
      b. All common uses are renamed to use \( T_{new} \).

C. Then we can do optimization using SSA
   1. Convert the program to SSA form.
      a. Add Φ operators with new temporaries where needed.
      b. Relabel all temporaries inside and outside blocks so each assignment is unique.
      c. Ensures that uses use the correct labeled temporary.
   2. Do the optimizations.
a. Local optimizations done globally
b. Dominator-based optimizations
c. Global optimizations for loops
d. Other complex optimizations (pointer, interprocedural)

3. Convert the resultant program back to normal form
   a. Remove Φ operators
   b. Add MOV instructions along edges to set the new temporaries to that of the Φ operator
   c. Simplify by merging names to eliminate these (and other) move instructions

4. Then we can do limiting optimizations, register allocation, and assembler code generation on the (optimized, simplified) resultant program

III. Conversion into SSA form
   A. This is done in two stages
      1. First we add Φ operators where they will be needed
         a. Add these as Ti = Φ(Ti,Ti,Ti,...,Ti)
         b. Need to determine where these locations are
         c. Need to guarantee that only one definition of Ti reaches any block
         d. Want to insert a minimum number of such nodes
      2. Note that the result is semantically correct
         a. Assuming correct semantics for Φ
      3. Then go through the whole program and relabel all temporaries
         a. Each point can be reached by only one definition of an original temporary
         b. Replace each use of an original temporary with that relabeled definition
         c. By keeping a map of old->new temporaries
   B. Inserting Φ operators
      1. Where are these needed
         a. For a temporary T at the beginning of a block B if B has multiple predecessors and different definitions of T occur on distinct paths going through at least two predecessors
         b. Paths must be distinct (no common subpaths)
         c. Must be done for each temporary independently
         d. Must be computable in linear time
      2. Notion of converging paths
a. Two non-null paths, p from \( B_0 \) to \( B_n \) and q from \( B'_0 \) to \( B'_m \) converge at a block Z iff
   i. \( B_0 \neq B'_0 \) (i.e. they start at different points)
   ii. \( B_n = B'_m = Z \) (i.e. they both end at Z)
   iii. If \( B_i = B'_j \) then either \( i = n \) or \( j = m \) (i.e. the only point in common on the paths is the end point)

b. If \( I_1 \) and \( I_2 \) are assignments to T, then any basic block Z that is the conjunction of two converging paths from \( I_1 \) and \( I_2 \) will be a point where a \( \Phi \) node will be inserted.

3. Compute the join points for a temporary iteratively
   a. \( S = \{ B \mid B \text{ contains an assignment to } X \} \)
   b. \( J_1(S) = \{ Z \mid \exists Z_1, Z_2 \text{ in } S \text{ such that } Z \text{ is a merge point for two paths } Z_1 \rightarrow Z \text{ and } Z_2 \rightarrow Z \} \)
   c. \( J_{i+1}(S) = J_1(S \cup J_i(S)) \)
   d. \( J^+(S) = \text{MAX } J_i(S) \)

4. But this can’t be computed quickly
   a. But we can quickly approximate it

C. Practical computation of merge points

1. Consider definitions of a temporary in blocks B and B’
   a. If B dominates B’, then consider a merge point Z for these blocks
      i. There must be disjoint paths from B to Z and B’ to Z
      ii. Suppose B’ dominates Z; then all executions from Start must pass through B’ before getting to Z. Since B dominates B’, there is a path from Start to B that doesn’t go through B’. Moreover, since the paths are disjoint, there is a path from B to Z that doesn’t go through B’. Hence B’ cannot dominate Z.
   b. If B’ dominates B, by similar arguments B cannot dominate Z
   c. If neither B nor B’ dominates the other, then a merge point will not be dominated by either because two distinct paths reach it.

2. This implies that merge points are related to the dominance frontier
   a. Recall what the dominance frontier is
      i. The dominance frontier DF(B) of a block B is the set of all blocks C such that B dominates a predecessor of C but either B equals C or B does not dominate C.
      ii. Can be generalized to sets of blocks
   b. We actually want the iterated dominance frontier DF^*
      i. \( S = \{ B \mid B \text{ contains an evaluation of } T \} \cup \{ \text{Start} \} \)
ii \(DF_1(S) = DF(S)\)
iii \(DF_{i+1}(S) = DF(S \cup DF_i(S))\)
iv \(DF^+(S) = \bigcup_{i} DF_i(S)\)

c. Note that start is in here
d. Note that we keep adding nodes
e. Since we know \(DF(B)\) for all \(B\), this is an easy computation

3. We can show \(Z\) is going to be a block on \(DF^+\) of the definitions
   a. Showing this is non-trivial however
   b. First show if \(p\) is a non-null path from \(B\) to \(Z\), then either \(B\) dominates each node on the path including \(Z\) or there is a block \(B'\) in the dominance frontier of \(B\) that is on the path \(p\) and that dominates \(Z\)
   c. Then show that if \(B\) and \(C\) are distinct blocks, and there are path \(p\) from \(B\) to \(Z\) and \(q\) from \(C\) to \(Z\) that converge at \(Z\), then \(Z\) is in either \(DF^+(B)\) or \(DF^+(C)\)

D. Basic Approach
1. For each temporary
   a. Let \(S\) to be the set of blocks containing definitions plus the entry block
   b. Compute \(DF^+(S)\)
   c. For each block in the result, add a \(\Phi\) operator

2. What is the problem with this
   a. Inserts too many \(\Phi\) operators
   b. It doesn’t take into account where variables are used

3. Consider first temporaries only used in one block
   a. These get \(\Phi\) operators at the DF of that block
   b. But such nodes are not needed
   c. We can restrict our loop to only look at global temporaries

4. Consider next temporaries that aren’t used
   a. We’ve previously computed where things are use
   b. This is reflected in the flow set \(LIVE\)
   c. We can restrict \(\Phi\) operators to blocks where the temporary is live

E. Final algorithm
1. Top level code
   Foreach variable temporary \(T\)
   If \(T\) is in \(GLOBALS\) then
     \(S = EVALIN(T) \cup \{START\}\)
     \(S = computeFrontier(S)\)
   For each block \(B\) in \(S\)
     If \(B\) is in \(LIVEIN(T)\) and \(B\) has > 1 predecessor :
       Insert a \(\Phi\) node at the start of \(B\) for \(T\)
   endif
2. Computing the iterated dominance frontier

```java
computeFrontier(BlockSet D)
BlockList worklist = empty
For each Block B in D
    Push B onto end of worklist
next
While ( worklist not empty ) DO
    Remove block B from front of worklist
    Foreach Block C in DF(B) do
        If C is not in D then
            Add C to D
            Add C to end of worklist
        endif
    next
next
return D
```

IV. Renaming the temporaries

A. Basic idea
1. Go through the blocks in DFS order
2. Maintain the current new temporary for each old temporary
3. Replace any use of a temporary with its new one
4. Create a new temporary at each definition and update the mapping from old to new
5. Push and pop the mappings as you go through blocks
6. Handle Φ nodes explicitly to get the right labels from incoming blocks

B. Data structures
1. name_stack : list (stack) of new temporaries indexed by old
   a. Front of the list is the current assignment
2. def_map : instruction that defines a given temporary
3. use_set : set of instructions where a temporary is used

C. Algorithm
```java
renameBlock(B)
foreach Instruction I in B in execution order
    if I is not a Φ instruction then
        foreach temporary operation T of I
            Tnew = name_stack[T].front()
            Replace T with Tnew in I
            add I to use_set[Tnew]
    next
```
Let T = target(I)
If T exists then
    Tnew = newTemporary()
    push Tnew onto front of name_stack[T]
    def_map[Tnew] = T
endif

next

foreach Block S in SUCC(B)
    j = whichPrecessor(S,B)
    foreach Φ instruction I in S do
        Let Tj be the jth operation of I
        Let Tnew be name_stack[Tj].top()
        Replace Tj by Tnew in I
        add I to use_set[Tnew]
    next
next

foreach Block C in DominanceChildren(B)
call renameBlock(C)
next

foreach instruction I in B in reverse execution order
    Let T = target(I)
    if T is defined then
        Pop Tnew from the front of name_stack[T]
        Replace T with Tnew in I
    endif
next

D. Notes
1. Forward pass over block to make the necessary changes
2. Φ operators are handled differently
3. Backward pass handles targets and restores the stack
4. Remember the overall approach used here: it will be reused later

V. Conversion out of SSA form
A. Basic idea
   1. Replace the Φ nodes with move instructions
   2. Relabel the temporaries consistently
   3. But a consistent relabeling can be tricky
B. Problems:
   1. Can’t place assignments on abnormal edges
C. Actual algorithm
   1. First build the set of all used temporaries
      a. Temporaries used as an operand in some instruction
2. Then create a partition of temporaries
   a. Those that should be merged
   b. Using a SET-UNION algorithm
      i. Map from temporary to temporary
      ii. Operations:
         ➢ insert (add new temporary, it gets its own partition)
         ➢ merge(T1,T2) :: equate two temporary sets
      iii. Done by maintaining a tree
      iv. Merge sets the parent of T2 to be T1
      v. Lookup climbs (and collapses) the parent hierarchy
   c. In each block B, if two equivalent temporaries are targets of Φ nodes, then the corresponding arguments must be equivalent
   d. For each abnormal critical edge, if T0 = Φ(T1,…Tm) then the Ti for the edge and T0 must be equivalent
3. Then normalize the code
   a. For each block B, for each instruction in execution order
      i. Replace each operand t with FIND(t)
      ii. Replace target t with FIND(t)
      iii. If this yields MOV Ti,Ti, remove the instruction
      iv. If this yields LDC and the target isn’t used, remove the instruction
      v. For each predecessor block C, call eliminatePhi(C,B,#)
   b. EliminatePhi
      i. Add move instruction on arc leading to block
      ii. Take care to preserve ordering
      iii. Due to optimizations, things can go wrong
         ➢ Need to remove critical edges and add blocks
      iv. Consider what happens with Loop(Swap(A,B))
         ➢ Block 1: a = ..., b= ... Link to 2
         ➢ Block 2: c = a, a = b, b = c; Link to 2, 3
         ➢ Block 3: ... = a
      v. Optimized result
         ➢ Block 2: b1 = PHI(b0,a1), a1 = PHI(a0,b1)
      vi. What does this result in
         ➢ Need to remove critical edge 2->2 and add block
   c. Finally remove Φ nodes

What is actual complexity of union-find. Need to be more explicit in the algorithm for removing SSA form (when to merge, etc). Can remove the example on eliminatephi and just say that you have to remove critical edges in order for this to work.