Contents of Lecture 3

- Translation to SSA Form
- Translation from SSA Form
A function is translated to SSA Form in the following steps

1. Compute the dominator tree $DT$ of the function.
2. Compute the dominance frontier of each vertex in the CFG.
3. Insert $\phi$-functions.
4. Rename variables while traversing the dominator tree.
We want to insert a $\phi$-function where two paths from assignments meet.

This formulation of the problem was difficult to use to find an efficient algorithm.

The following is a trick which makes it easier to answer the question of where to insert $\phi$-functions:

**Trick:** Every variable is given a assignment in the start vertex.

That is, a variable $x$ is given an assignment $x_0$ in the start vertex.

No assembler code is produced for the assignment though.
Why would $x_0$ help???

- With the assignment to $x_0$ we can see that two paths from assignments join in the vertices with $x_?$.
- Therefore each of them needs a $\phi$-function.
- Another way to see this is that these vertices are just outside what is dominated by the vertex with $x_1 =$.
Dominance frontier

- We need to insert a $\phi$-function in every vertex which is just outside what is dominated by a vertex with an assignment.
- "Just outside" is called the **dominance frontier** of a vertex $u$.
- It is written $DF(u)$.
- $\mathit{DF}(u) = \{ v \mid \exists p \in \mathit{pred}(v), \ u \gg p, \ u \gg v \}$.
- In words: if $u$ dominates a predecessor of $v$ but does not dominate $v$ strictly, then $v$ is in the dominance frontier of $u$.
- After the dominator tree is found, the dominance frontier for each vertex is computed.
- Each local variable and compiler-generated temporary is inspected: for each vertex $u$ with an assignment to the variable, a $\phi$-function is inserted in $\mathit{DF}(u)$.
- N.B. a $\phi$-function is an assignment — which also needs $\phi$-functions in the dominance frontier of its vertex. More about that below.
Multiple assignments

- The assignment to $x_4$ means that that vertex is dominated by two different assignments.
- Therefore we must rename the variables in a certain order so that after a later assignment the up-to-date version is used during the renaming.
- Obviously it is $x_4$ that should be the $\phi$-operand and not $x_1$.
- This is achieved during the renaming in which a stack of variable versions is used and the current version of a variable is found on the top of the stack.
Using the Dominator Tree and a Stack of Variable Versions

- After $\phi$-functions have been inserted (more details below) the dominator tree is traversed during variable renaming.
- Each variable has its own stack of variable versions.
- At a use of a variable in a statement, the variable is replaced in the statement by the top of variable’s stack.
- At an assignment a new variable version is pushed on the variable’s stack, and the variable is replaced in the statement by the new version.
The new version $x_1$ is pushed on the stack of $x$.

The vertex with $x_4$ is a child in the $DT$ and is inspected next.

The new version $x_4$ is pushed on the stack of $x$.

The $\phi$-function in the successor vertex gets one of its operands replaced to $x_4$ from the current top of the stack.

The vertex with $x_4$ has no child in the $DT$ and $x_4$ is popped from the stack.

$x_1$ is then at the top of the stack and is used next.
Strict Dominance in the Definition of Dominance Frontier

\[ DF(u) = \{ v \mid \exists p \in \text{pred}(v), u \gg p, u \gg v \} \]

- Consider 7 and suppose it contains ++i.
- It then needs \( i = \phi(i, i) \).
- \( DF(7) = \{5, 7\} \).
- When 7 is added to its own DF it is both \( u \), \( p \), and \( v \) in the definition.
- This situation is the reason for using not strict dominance in the definition.
Computing the Dominance Frontiers of a CFG

- \( DF(u) = \{ v \mid \exists p \in \text{pred}(v), \ u \gg p, \ u \gg v \} \).
- Below \( \text{children}(u) \) is the set of children of \( u \) in the dominator tree.
- The dominance frontier is computed bottom up in the dominance tree using:
  \[
  DF(u) = DF_{\text{local}}(u) \cup \bigcup_{c \in \text{children}(u)} DF_{\text{up}}(c)
  \]
- \( DF_{\text{local}}(u) \overset{\text{def}}{=} \{ v \in \text{succ}(u) \mid u \gg v \} \).
- \( DF_{\text{up}}(c) \overset{\text{def}}{=} \{ v \in DF(c) \mid \text{idom}(c) \gg v \} \).
- These formulas can be simplified further as we will see, but first we will build intuition into why they are correct.
The set $DF_{local}(u)$ is defined as $DF_{local}(u) \overset{\text{def}}{=} \{ v \in succ(u) \mid u \gg v \}$.  

The set $DF_{local}(u)$ is the contribution to $DF(u)$ which can be determined by only looking at the successors of $u$ in the CFG.

Since $u$ does not dominate $v$ strictly, but clearly it dominates a predecessor of $v$ (namely itself), $v \in DF(u)$.

For example, $3 \in DF(2)$ and $7 \in DF(7)$

But e.g. $3 \notin DF(1)$ since $1 \gg 3$. 

\[ \text{DF}_{local}(u) \]
$DF_{up}(c)$

- $DF_{up}(c) \overset{\text{def}}{=} \{ v \in DF(c) \mid idom(c) \gtrsim v \}$.
- The set $DF_{up}(c)$ is the contribution from a vertex $c$ to the $DF$ of $idom(c)$.
- To see that $DF_{up}(c) \subseteq DF(idom(c))$, consider any vertex $v \in DF(c)$.
- From the definition of $DF(c)$ there must be a $p \in \text{pred}(v)$ such that $c \gtrsim p$.
- Since dominance is transitive and obviously $idom(c) \gtrsim c$ we must have $idom(c) \gtrsim p$.
- Thus the vertices in $DF(c)$ which are not strictly dominated by $idom(c)$ should be added to $DF(idom(c))$ and this is what $DF_{up}(c)$ achieves.
In the book is also shown that every vertex in $DF(v)$ is accounted for in either $DF_{local}(v)$ or $DF_{up}(c)$ where $idom(c) = v$.

One can also show that instead of:

$$DF_{local}(u) \overset{\text{def}}{=} \{ v \in succ(u) | u \gg v \}.$$  

we can use:

$$DF_{local}(u) \overset{\text{def}}{=} \{ v \in succ(u) | u \neq idom(v) \}.$$  

and:

$$DF_{up}(c) \overset{\text{def}}{=} \{ v \in DF(c) | idom(c) \neq idom(v) \}.$$
Computing the Dominance Frontiers of a CFG

```
procedure df(G, DT)
    for each u in a postorder traversal of DT do
        DF(u) ← ∅
        for each v ∈ succ(u) do
            if (idom(v) ≠ u)
                add v to DF(u)
        for each w ∈ children(u) do
            for each v ∈ DF(w) do
                if (idom(v) ≠ u)
                    add v to DF(u)
```

DT
By postorder traversal is meant that when we visit vertex $u$, we first compute the dominance frontier of each child $c$ of $u$ in $DT$ before we compute $DF(u)$.

You will implement this function in Lab 2.

Recursively walk through the dominator tree.

The first computed set will be $DF_{local}(7) = \{5, 7\}$.

$DF_{up}(c)$ is never explicitly stored but computed by inspecting $DF(c)$

The first complete computed dominance frontier will be $DF(7) = \{5, 7\}$.

Then the $DF(6), DF(2), DF(4)$ etc...
Inserting $\phi$-functions

- $\phi$-functions are inserted for one variable at a time.
- A counter iteration is incremented when the next variable is processed — i.e. gets its $\phi$-functions inserted into the CFG.
- Each vertex has two attributes for the $\phi$-function insertion which keeps track of for which iteration it was processed:
  - has_already — used to determine whether a $\phi$-function for a certain variable has already been inserted in that vertex.
  - work — used to determine whether that vertex has been put in a worklist called $W$.
- These variables are all set to zero initially.
procedure insert-\(\phi\)

\[
W \leftarrow \emptyset
\]

for each variable \(V\) do

\[
iteration \leftarrow iteration + 1
\]

for each \(u \in \text{vertex\_with\_assignment}(V)\) do

\[
work[u] \leftarrow iteration
\]

add \(u\) to \(W\)

while \((W \neq \emptyset)\) do

take \(u\) from \(W\)

for each \(v \in DF(u)\) do

if \((\text{has\_already}[v] < iteration)\)

place \(V \leftarrow \phi(V, ..., V)\) at \(v\)

\[
\text{has\_already}[v] \leftarrow iteration
\]

if \((\text{work}[v] < iteration)\)

\[
\text{work}[v] \leftarrow iteration
\]

add \(v\) to \(W\)
The use of an explicit counter and the attributes **work** and **has_already** is how the algorithm was originally described by researchers from IBM.

This is more efficient than using lookup-functions to determine whether a vertex has a certain $\phi$-function or a vertex is in the worklist.

For optimizing compilers research the speed of the compiler at normal optimization levels, e.g. `-O2` is extremely important.

However, some optimizations which analyze the whole program is sometimes allowed to take hours.
Rename performs a traversal of the dominator tree.

In a vertex \( u \) the sequence of three-address statements is examined one statement at a time:

- First the source operands (right hand side, or RHS) are renamed by replacing the operand with the version of the variable on the top of the variable’s rename stack.
- Then the destination operand (left hand side, or LHS) is renamed by creating a new variable version, pushing it on the rename stack, and replacing the operand with the new version of the variable.

Then the \( \phi \)-functions of each successor vertex \( v \) in the CFG is inspected and the operand corresponding to the edge \( (u, v) \) is renamed.

Then each child \( c \) in the DT is processed.

Finally every new version created and pushed on a rename stack in \( u \) is popped from its rename stack.
procedure rename(u)
    for each statement t in u do
        for each variable V ∈ RHS(t) do
            replace use of V by use of Vi where i = top(S(V))
        for each variable V ∈ LHS(t) do
            i ← C(V)
            replace V by Vi
            push i onto S(V)
            C(V) ← C(V) + 1
        for each v ∈ succ(u) do
            j ← which_pred(u, v)
            for each ϕ-function in v do
                replace the j-th operand in RHS(ϕ) by Vi where i = top(S(V))
        for each v ∈ children(u) do
            rename(v)
    pop every variable version pushed in u
Unnecessary $\phi$-functions

- It’s unnecessary to insert a $\phi$-function if its value is never used:
  
  ```java
  if (a > 0) {
      a = a + 1;
      f(a);
  }
  return b;
  ```

- Before the return, there will be a $\phi$-function due to the assignment to $a$.

- In general the cost to determine whether the value will be used is not worth the effort.

- It’s not uncommon that a $\phi$-function is inserted in a vertex where the value is overwritten before being used. This special case can be easy to determine and may be worth the effort of avoiding inserting an unnecessary $\phi$-function.
Most optimization algorithms ignore the variable version number and treat for instance $a_i$ and $a_j$ as completely different variables which have no more in common than $a_i$ and $b_k$ have.

Therefore no counter is usually needed: it’s sufficient to simply create a new temporary variable.

However, Partial Redundancy Elimination, SSAPRE, needs to know from which original variable such a temporary comes.
The basic idea when translating from SSA Form is to replace the \( \phi \)-functions with copy statements in the predecessor vertices.
Translation from SSA Form

- \( a_1 = u_3 + v_1 \)
- \( a_1 > b_4 \) ??

- \( a_3 = a_1 \)
- \( b_6 = b_4 \)

- \( a_2 = u_3 - v_2 \)
- \( b_5 = a_2 - 1 \)
- \( a_3 = a_2 \)
- \( b_6 = b_5 \)

- \( y_1 = a_3 \times b_6 \)

- It’s thus necessary to have a vertex to insert the copy statements into!
- Without the leftmost vertex, there is an edge from a vertex with multiple successors to a vertex with multiple predecessors and such an edge is called a critical edge.
- Critical edges are removed by inserting an extra empty vertex.
- This is done before dominance analysis.
The $\phi$-functions are parallel copy statements.

Conceptually all $\phi$-functions are executed concurrently by first reading all operands and then writing all destinations.

So what will go wrong here with a ”naive” translation from SSA Form?
Translation from SSA Form

What is wrong here?
Translation from SSA Form

- $a_2 = a_1$
- $b_2 = b_1$

- $x = a_2$
- $a_2 = b_2$
- $b_2 = x$

- The value of $a_2$ must be saved before being overwritten!
If version zero is used and there was no explicit initializer for the variable (i.e. no `int a = 1`) it means we have discovered a buggy program with undefined behavior!
During Translation to SSA Form, a copy statement \( a = b \) can be optimized as follows:

- The current value of \( b \), i.e. the version on the top of \( b \)'s rename stack is pushed on \( a \)'s rename stack and is copy statement can then be removed.

- You will do this during Lab 2.