**Motivation**

- Data flow analysis needs to represent facts at every program point.

- What if
  - There are a lot of facts and
  - There are a lot of program points?
    - \( \Rightarrow \) potentially takes a lot of space/time

- Most likely, we're keeping track of irrelevant facts

**Sparse Representation**

- Instead, we'd like to use a sparse representation
  - Only propagate facts about \( x \) where they're needed
  - Enter *static single assignment* form
    - Each variable is defined (assigned to) exactly once
    - But may be used multiple times

**What About Joins?**

- Add \( \Phi \) functions/nodes to model joins
  - Intuitively, takes meet of arguments
  - At code generation time, need to eliminate \( \Phi \) nodes
  - Add SSA edges from definitions to uses
    - No intervening statements use/define variable
    - Safe to propagate only along SSA edges

---

**Example**

- \( x = 3 \)
  - \( x = 3 \)
  - \( y = a + b \)
  - \( z = 2 \times y \)
  - \( w = y + z \)

- \( a > b \)
  - \( y = a - b \)
  - \( y = y \times 10 \)
  - \( w = w + y \)
  - \( z = w + x \)
**Constant Propagation Revisited**

- Initialize facts at each program point
  - $C(n) := \text{top}$
- Add all SSA edges to the worklist
- While the worklist isn’t empty,
  - Remove an edge $(x, y)$ from the worklist
  - $C(y) := C(y) \cap C(x)$
  - Add SSA edges from $y$ if $C(y)$ changed

**Def-Use Chains vs. SSA**

- Alternative: Don’t do renaming; instead, compute simple def-use chains (reaching definitions)
  - Propagate facts along def-use chains
- Drawback: Potentially quadratic size

**Def-Use Chains vs. SSA (cont’d)**

```plaintext
case (...) of  
  0: a := 1;  
  1: a := 2;  
  2: a := 3;  
end  
case (...) of  
  0: b := a;  
  1: c := a;  
  2: d := a;  
end
```

**Conditional Constant Propagation**

- So far, we assume that all branches can be taken
  - But what if some branches are never taken in practice?
    - Debugging code that can be enabled/disabled at run time
    - Macro expanded code with constants
    - Optimizations
  - Idea: use constant propagation to decide which branches might be taken
    - Fits in neatly with SSA form

**Nodes versus Edges**

- So far, we’ve been hazy about whether data flow facts are associated with nodes or edges
  - Advantage of nodes: may be fewer of them
  - Advantage of edges: can trace differences on multiple paths to same node
- For this problem, we’ll associate facts with edges

**Conditional Execution**

- Keep track of whether edges may be executed
  - Some may not be because they’re on not-taken branch
  - Initially, assume no edges taken
  - At joins, don’t propagate information from not-taken in-edges
  - Side comment: Notice that we always, always start with the optimistic assumption
    - We need proof that a pessimistic fact holds
    - We’re computing a greatest fixpoint
Example

```
x1 := 3
x1 > 2
j1 := 1
j2 := 4
j3 := !(j1, j2)
z
```

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place Φ nodes
  - Naive, impractical step 2: put a Φ function for every variable at the beginning of every block
  - Better: If node X contains assignment to a, put Φ function for a in dominance frontier of X
    - Adding Φ fn may require introducing additional Φ fn
- Step 3: Rename variables so only one definition per name

Dominator Tree

- The dominator relationship forms a tree
  - Edge from parent to child = parent dominates child
  - Note: edges are not same as CFG edges!

Why Are Dominators Useful?

- Computing static single assignment form
- Computing control dependencies
- Identify loops in CFG
  - All nodes X dominated by entry node H, where X can reach H, and there is exactly one back edge (head dominates tail) in loop

Dominator Tree

• Let X and Y be nodes in the CFG
  • Assume single entry point Entry
  • X dominates Y (written X≥Y) if
    - X appears on every path from Entry to Y
  • Write X>Y when X dominates Y but X≠Y
  • Note ≥ is reflexive

Computing Dominator Tree

• Standard algorithm due to Lengauer and Tarjan
  • Runs in time \( O(E \alpha(E, N)) \)
    - \( E \) = # of edges, \( N \) = # of nodes
    - where \( \alpha(\cdot) \) is the inverse Ackerman's function
    - Very slow growing; effectively constant in practice
  • Algorithm quite difficult to understand
    - But lots of pseudo-code available

Why Are Dominators Useful?

• Computing static single assignment form
• Computing control dependencies
• Identify loops in CFG
  • All nodes X dominated by entry node H, where X can reach H, and there is exactly one back edge (head dominates tail) in loop
**Where do \( \Phi \) Functions Go?**

- We need a \( \Phi \) function at node \( Z \) if
  - Two non-null CFG paths that both define \( v \)
  - Such that both paths start at two distinct nodes and end at \( Z \)

\[
\begin{align*}
v &> 3 \\
v &> 4 \\
Z
\end{align*}
\]

**Dominance Frontiers**

- \( Y \) is in the dominance frontier of \( X \) iff
  - There exists a path from \( X \) to Exit through \( Y \) such that \( Y \) is the first node not strictly dominated by \( X \)
  - Equivalently:
    - \( Y \) is the first node where a path from \( X \) to Exit and a path from Entry to Exit (not going through \( X \)) meet
  - Equivalently:
    - \( X \) dominates a predecessor of \( Y \)
    - \( X \) does not strictly dominate \( Y \)

**Example**

![Diagram](attachment:example.png)

**Computing Dominance Frontiers**

- Two components to \( DF(X) \):
  - \( DF_{local}(X) = \{ Y \in succ(X) \mid X \not\geq Y \} \)
    - Any child of \( X \) not (strictly) dominated by \( X \) is in \( DF(X) \)
  - Let \( Z \) be such that \( idom(Z) = X \)
    - \( idom(Z) \) is the parent of \( Z \) in the dominator tree
  - \( DF_{up}(Z) = \{ Y \in DF(Z) \mid X \not\geq Y \} \)
    - Nodes from \( DF(Z) \) that are not strictly dominated by \( X \) are also in \( DF(X) \)

**Why Is This Sufficient?**

- Suppose \( Y \in DF(X) \)
  - Then there is a \( U \in \text{pred}(Y) \) such that \( X \geq U, X \not\geq Y \)
  - If \( U=X \), then \( U \in DF_{local}(X) = \{ Y \in succ(X) \mid X \not\geq Y \} \)
  - Otherwise \( U \neq X \)
    - Then there is a node \( Z \) such that \( idom(Z)=X \) and \( Z \geq U \)
    - Possibly \( Z=U \)
    - Since \( X \not\geq Y \), \( Z \not\geq Y \), hence \( Y \in DF(Z) \)
    - Therefore \( Y \in DF_{up}(Z) = \{ Y \in DF(Z) \mid X \not\geq Y \} \)
Algorithm

- Let $sdom(X) = \{Y \mid X > Y\}$
- In a postorder traversal on dominator tree
  - $DF(X) = succ(X) - sdom(X)$
    - i.e., $DF(X) = DF_{local}(X)$
  - For each $Z$ such that $idom(Z) = X$ do
    - $DF(X) = DF(X) - (DF(Z) - sdom(X))$
    - i.e., $DF(X) = DF(X) - DF_{up}(Z)$

Equivalent Algorithm

- In a postorder traversal on dominator tree
  - $DF(X) = succ(X)$
  - For each $Z$ such that $idom(Z) = X$ do
    - $DF(X) = DF(X) - sdom(X)$
  - See paper for another equivalent algorithm that runs in $O(E + |DF|)$

Computing SSA Form

- Step 1: Compute the dominance frontier
- Step 2: Use dominance frontier to place $\Phi$ nodes
- Step 3: Rename variables so only one definition per name

Step 2: Placing $\Phi$ Functions for $v$

- Let $S$ be the set of nodes that define $v$
- Need to place $\Phi$ function in every node in $DF(S)$
  - Recall, those are all the places where the definition of $v$ in $S$ and some other definition of $v$ may meet
  - But a $\Phi$ function adds another definition of $v$!
    - $v := \Phi(v, ... , v)$
  - So, iterate
    - $DF_1 = DF(S)$
    - $DF_{i+1} = DF(S \cup DF_i)$

Step 3: Renaming Variables

- Top-down (DFS) traversal of dominator tree
  - At definition of $v$, push new # for $v$ onto the stack
  - When leaving node with definition of $v$, pop stack
  - Intuitively: Works because there's a $\Phi$ function, hence a new definition of $v$, just beyond region dominated by definition
  - Can be done in $O(E + |DF|)$ time
    - Linear in size of CFG with $\Phi$ functions

Example

```
Entry
1: x := 3
3
5: x := 4
8
11
4
7
10
Exit
```

```
= need $\Phi$ function
```

```
Top-down (DFS) traversal of dominator tree
- At definition of $v$, push new # for $v$ onto the stack
- When leaving node with definition of $v$, pop stack
  - Intuitively: Works because there's a $\Phi$ function, hence a new definition of $v$, just beyond region dominated by definition
  - Can be done in $O(E + |DF|)$ time
    - Linear in size of CFG with $\Phi$ functions
```
Eliminating $\Phi$ Functions

- Basic idea: $\Phi$ represents facts that value of join may come from different paths
  - So just set along each possible path

\[ W_2 := y_1 + z_1 \quad W_3 := W_2 + y_3 \quad W_4 := W_3 + y_3 \]

Eliminating $\Phi$ Functions in Practice

- Copies performed at $\Phi$ fns may not be useful
  - Joined value may not be used later in the program
    - (So why leave it in?)
  - Use dead code elimination to kill useless $\Phi$s
  - Subsequent register allocation will map the (now very large) number of variables onto the actual set of machine register

Efficiency in Practice

- Claimed:
  - SSA grows linearly with size of program
  - No correlation between ratio and program size

  Convincing?

Arrays

- Need to handle array accesses

  Problem: How do we know whether $A[i], A[j], \text{ and } B[k]$ are all distinct?
    - Could have $A=B$, e.g., `foo(int A[], int B[]){ ... foo(a,a)`
    - Could have $i=j$

  History: significant research on determining array dependencies, for parallelizing compilers

Arrays (cont’d)

- This paper’s suggestion: make arrays immutable
  - Then don’t need to worry about updates to them

\* := A.f;
A.g := V;
\* := A.f + A.g

- Update($A, j, V$) makes a copy of $A$
  - Then try to collapse unnecessary copies

Convincing?

Structures

- Can treat structures as sets of variables

\* := X;
Y := V;
\* := X + Y

Problems!
**Pointers**

- For each statement $S$, let
  - $\text{MustMod}(S)$ = variables always modified by $S$
  - $\text{MayMod}(S)$ = variables sometimes modified by $S$
    - So if $v \notin \text{MayMod}(S)$, then $S$ must not modify $v$
  - $\text{MayUse}(S)$ = variables sometimes used by $S$

- Then assume that statement $S$
  - writes to $\text{MayMod}(S)$
  - reads $\text{MayUse}(S) \cup (\text{MayMod}(S) \setminus \text{MustMod}(S))$

- Convincing? We'll talk more about pointers later in the course

**Control Dependence**

- $Y$ is control dependent on $X$ if whether $Y$ is executed depends on a test at $X$

```
X
  \(\rightarrow\)
A
  \(\rightarrow\)
B
  \(\rightarrow\)
C
```

- $A$, $B$, and $C$ are control dependent on $X$

**Postdominators and Control Dependence**

- $Y$ postdominates $X$ if every path from $X$ to Exit contains $Y$
  - i.e., if $X$ is executed, then $Y$ is always executed

- Then, $Y$ is control dependent on $X$ if
  - There is a path $X \rightarrow Z_1 \rightarrow \cdots \rightarrow Z_n \rightarrow Y$ such that $Y$ postdominates all $Z_i$ and
  - $Y$ does not postdominate $X$
  - i.e., there is some path from $X$ on which $Y$ is always executed, and there is some path on which $Y$ is not executed

**Dominance Frontiers, Take 2**

- Postdominators are just dominators on the CFG with the edges reversed

- To see what $Y$ is control dependent on, we want to find the $X$s such that in the reverse CFG
  - There is a path $X \leftarrow Z_1 \leftarrow \cdots \leftarrow Z_n \leftarrow Y$ where
    - for all $i, Y \geq Z_i$ and
    - $Y \neq X$
  - i.e., we want to find $DF(Y)$ in the reverse CFG!