**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

---

**Example**

\[
\begin{align*}
x &= 3 \\
y &= a + b \\
z &= 2 \times y \\
w &= y + z \\
\end{align*}
\]

\[
\begin{align*}
x &= 3 \\
y &= a_1 + b_1 \\
z &= 2 \times y_1 \\
w &= y_1 + z_1 \\
\end{align*}
\]

\[
\begin{align*}
x &= 3 \\
y &= a_1 - b_1 \\
z &= w + x_1 \\
w &= w + y_1 \\
\end{align*}
\]

---

**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

---

**Example: SSA**

- Add SSA edges from definitions to uses
  - No intervening statements use/define variable
  - Safe to propagate only along SSA edges

---

**What About Joins?**

- Add \( \Phi \) functions/nodes to model joins
  - Intuitively, takes meet of arguments
  - At code generation time, need to eliminate \( \Phi \) nodes
### 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)

#### Use-Def Chains

<table>
<thead>
<tr>
<th>$a := 1$</th>
<th>$a := 2$</th>
<th>$a := 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b := a$</td>
<td>$b := a$</td>
<td>$b := a$</td>
</tr>
</tbody>
</table>

#### SSA Form

- $a_1 := 1$
- $a_2 := 2$
- $a_3 := 3$
- $b := \Phi(a_1, a_2, a_3)$

Quadratic vs. (in practice) linear behavior

### 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 := Φ(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

Dominators

- Let X and Y be nodes in the CFG
  - Assume single entry point E
  - X dominates Y (written X≥Y) if
    - X appears on every path from E 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(Eα(E, N))
    - where α(·) 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 $Z$

![Diagram showing $v := 3$ and $v := 4$ leading to $Z$](image)

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 $Y$

![Diagram illustrating dominance frontiers](image)

Computing Dominance Frontiers

- Two components to $DF(X)$:
  - $DF_{local}(X) = \{ Y \in succ(X) \mid X \not\succ Y \}$
    - 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
  - $DF_{up}(Z) = \{ Y \in DF(Z) \mid X \not\succ Y \}$
    - Nodes from $DF(Z)$ that are not strictly dominated by $X$ are also in $DF(X)$

![Diagram illustrating dominance frontiers](image)

Why Is This Sufficient?

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

- Let $sdom(X) = \{Y \mid X \succ Y\}$
- 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
    - $DF(X) = DF(X) \Delta (DF(Z) - sdom(X))$
    - i.e., $DF(X) = DF(X) \Delta DF_{up}(Z)$

Equivalent Algorithm

- In a postorder traversal on dominator tree
  - $DF(X) = succ(X)$
  - For each $Z$ such that $idom(Z) = X$ do
    - $DF(X) = DF(X) \cup DF(Z)$
    - $DF(X) = DF(X) - sdom(X)$
    - 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

Example

- 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

\[
\begin{align*}
w_2 &= y_1 + z_1 \\
w_3 &= w_1 + y_3 \\
w_4 &= \Phi(w_2, w_3) \\
z &= w_2 = w_4 = w_3
\end{align*}
\]

Efficiency in Practice

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

Arrays (cont’d)

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

\[
\begin{align*}
* &= A[i]; \\
A[j] &= V; \\
* &= A[k] + z; \\
T &= A[k]; \\
* &= T + 2;
\end{align*}
\]

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

Arrays

- Need to handle array accesses

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

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

Structures

- Can treat structures as sets of variables

\[
\begin{align*}
* &= A.f; \\
* &= X; & X &= A.f \\
A.g &= V; \\
Y &= V; & Y &= A.g \\
* &= A.f + A.g \\
* &= X + Y
\end{align*}
\]

- Problems!
Points

- 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 $\text{MayUse}(S) \cup (\text{MayMod}(S) - \text{MustMod}(S))$
- 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 $X$

  \[
  \begin{tikzpicture}
  \node (A) at (0,0) {A};
  \node (B) at (1,0) {B};
  \node (C) at (2,0) {C};
  \node (X) at (-1,1) {X};
  \draw[->] (X) -- (A);
  \draw[->] (X) -- (B);
  \draw[->] (B) -- (C);
  \end{tikzpicture}
  \]
- $A$, $B$, 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 $X \rightarrow Z_1 \rightarrow \cdots \rightarrow Z_n \rightarrow Y$ from such that $Y$ postdominates all $Z_i$
  - $Y$ 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!