**Dominators, control-dependence and SSA form**

**Organization**

- Dominator relation of CFGs
  - postdominator relation
- Dominator tree
- Computing dominator relation and tree
  - Dataflow algorithm
  - Lengauer and Tarjan algorithm
- Control-dependence relation
- SSA form

**Control-flow graphs**

- CFG is a DAG
- Unique node **START** from which all nodes in CFG are reachable
- Unique node **END** reachable from all nodes
- Dummy edge to simplify discussion

**Dominators**

- In a CFG G, node a is said to dominate node b if every path from **START** to b contains a.
- Dominance relation: relation on nodes
  - We will write a dom b if a dominates b
Computing dominance relation

• Dataflow problem:

- Domain: powerset of nodes in CFG
- Confluence operation: set intersection
- Find greatest solution

Work through example on previous slide to check this.
Question: what do you get if you compute least solution?

Properties of dominance

• Dominance is
  – reflexive: a dom a
  – anti-symmetric: a dom b and b dom a → a = b
  – transitive: a dom b and b dom c → a dom c
  – tree-structured:
  - a dom c and b dom c → a dom b or b dom a
  - intuitively, this means dominators of a node are themselves ordered by dominance

Example of proof

• Let us prove that dominance is transitive.
  – Given: a dom b and b dom c
  – Consider any path P: START →+ c
  – Since b dom c, P must contain b.
  – Consider prefix of P = Q: START →+ b
  – Q must contain a because a dom b.
  – Therefore P contains a.
Dominator tree example

Check: verify that from dominator tree, you can generate full relation

Computing dominator tree

• Inefficient way:
  – Solve dataflow equations to compute full dominance relation
  – Build tree top-down
    • Root is START
    • For every other node
      – Remove START from its dominator set
      – If node is then dominated only by itself, add node as child of START in dominator tree
    • Keep repeating this process in the obvious way

Building dominator tree directly

• Algorithm of Lengauer and Tarjan
  – Based on depth-first search of graph
  – \( O(E \cdot \alpha(E)) \) where \( E \) is number of edges in CFG
  – Essentially linear time
• Linear time algorithm due to Buchsbaum et al
  – Much more complex and probably not efficient to implement except for very large graphs

Immediate dominators

• Parent of node \( b \) in tree, if it exists, is called the immediate dominator of \( b \)
  – written as \( \text{idom}(b) \)
  – \( \text{idom} \) not defined for \( \text{START} \)
• Intuitively, all dominators of \( b \) other than \( b \) itself dominate \( \text{idom}(b) \)
  – In our example, \( \text{idom}(c) = a \)
**Useful lemma**

- Lemma: Given CFG $G$ and edge $a \rightarrow b$, $\text{idom}(b)$ dominates $a$.
- Proof: Otherwise, there is a path $P$: $\text{START} \rightarrow+ a$ that does not contain $\text{idom}(b)$. Concatenating edge $a \rightarrow b$ to path $P$, we get a path from $\text{START}$ to $b$ that does not contain $\text{idom}(b)$ which is a contradiction.

**Postdominators**

- Given a CFG $G$, node $b$ is said to postdominate node $a$ if every path from $a$ to $\text{END}$ contains $b$.
  - We write $b \text{ pdom} a$ to say that $b$ postdominates $a$.
- Postdominance is dominance in reverse CFG obtained by reversing direction of all edges and interchanging roles of $\text{START}$ and $\text{END}$.
- Caveat: $a \text{ dom} b$ does not necessarily imply $b \text{ pdom} a$.
  - See example: $a \text{ dom} b$ but $b$ does not $\text{ pdom} a$.

**Obvious properties**

- Postdominance is a tree-structured relation.
- Postdominator relation can be built using a backward dataflow analysis.
- Postdominator tree can be built using Lengauer and Tarjan algorithm on reverse CFG.
- Immediate postdominator: $\text{ipdom}$.
- Lemma: if $a \rightarrow b$ is an edge in CFG $G$, then $\text{ipdom}(a)$ postdominates $b$.

**Control dependence**

- Intuitive idea:
  - node $w$ is control-dependent on a node $u$ if node $u$ determines whether $w$ is executed.
- Example:
  
  ![Control dependence diagram](image)

  We would say $S1$ and $S2$ are control-dependent on $e$. 

Examples (contd.)

We would say node S1 is control-dependent on e. It is also intuitive to say node e is control-dependent on itself:
- execution of node e determines whether or not e is executed again.

Example (contd.)

- S1 and S3 are control-dependent on f
- Are they control-dependent on e?
- Decision at e does not fully determine if S1 (or S3 is executed) since there is a later test that determines this
- So we will NOT say that S1 and S3 are control-dependent on e
  - Intuition: control-dependence is about "last" decision point
- However, f is control-dependent on e, and S1 and S3 are transitively (iteratively) control-dependent on e

Example (contd.)

- Can a node be control-dependent on more than one node?
  - yes, see example
  - nested repeat-until loops
    - n is control-dependent on t1 and t2 (why?)
- In general, control-dependence relation can be quadratic in size of program

Formal definition of control dependence

- Formalizing these intuitions is quite tricky
- Starting around 1980, lots of proposed definitions
- Commonly accepted definition due to Ferrane, Ottenstein, Warren (1987)
- Uses idea of postdominance
- We will use a slightly modified definition due to Bilardi and Pingali which is easier to think about and work with
Control dependence definition

- First cut: given a CFG G, a node w is control-dependent on an edge (u→v) if
  - w postdominates v
  - ... w does not postdominate u
- Intuitively,
  - first condition: if control flows from u to v it is guaranteed that w will be executed
  - second condition: but from u we can reach END without encountering w
  - so there is a decision being made at u that determines whether w is executed

Small caveat: what if w = u in previous definition?
- See picture: is u control-dependent on edge u→v?
- Intuition says yes, but definition on previous slides says "u should not postdominate u" and our definition of postdominance is reflexive
- Fix: given a CFG G, a node w is control-dependent on an edge (u→v) if
  - w postdominates v
  - if w is not u, w does not postdominate u

Strict postdominance

- A node w is said to strictly postdominate a node u if
  - w ≠ u
  - w postdominates u
- That is, strict postdominance is the irreflexive version of the dominance relation
- Control dependence: given a CFG G, a node w is control-dependent on an edge (u→v) if
  - w postdominates v
  - w does not strictly postdominate u

Example

```
START  f  c  d  e  a  b  x  x  x  x
START→a
f→b
x  x  x  x

START  f  c  d  e  a  b  x  x  x  x
START→a
f→b
x  x  x  x
```
Computing control-dependence relation

- Nodes control dependent on edge \((u \rightarrow v)\) are nodes on path up the postdominator tree from \(v\) to \(ipdom(u)\), excluding \(ipdom(u)\)
  - We will write this as \([v, ipdom(u))\)
  - half-open interval in tree

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>START→a</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>f→b</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>c→d</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>c→e</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>a→b</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Computing control-dependence relation

- Compute the postdominator tree
- Overlay each edge \(u \rightarrow v\) on \(pdom\) tree and determine nodes in interval \([v, ipdom(u))\)
- Time and space complexity is \(O(EV)\).
- Faster solution: in practice, we do not want the full relation, we only make queries
  - \(cde(e)\): what are the nodes control-dependent on an edge \(e\)?
  - \(cde(w)\): what are the edges that \(w\) is control-dependent on?
  - \(cdequiv(w)\): what nodes have the same control-dependences as node \(w\)?
- It is possible to implement a simple data structure that takes \(O(E)\) time and space to build, and that answers these queries in time proportional to output of query (optimal) (Pingali and Bilardi 1997).

SSA form

- Static single assignment form
  - Intermediate representation of program in which every use of a variable is reached by exactly one definition
  - Most programs do not satisfy this condition
    - (eg) see program on next slide: use of \(Z\) in node \(F\) is reached by definitions in nodes \(A\) and \(C\)
  - Requires inserting dummy assignments called \(\Phi\)-functions at merge points in the CFG to "merge" multiple definitions
  - Simple algorithm: insert \(\Phi\)-functions for all variables at all merge points in the CFG and rename each real and dummy assignment of a variable uniquely
    - (eg) see transformed example on next slide

SSA example
Minimal SSA form

- In previous example, dummy assignment Z3 is not really needed since there is no actual assignment to Z in nodes D and G of the original program.
- Minimal SSA form
  - SSA form of program that does not contain such “unnecessary” dummy assignments
  - See example on next slide
- Question: how do we construct minimal SSA form directly?

**Minimal-SSA form Example**

**Dominance frontier**

- Dominance frontier of node w
  - Node u is in dominance frontier of node w if w dominates a CFG predecessor v of u, but does not strictly dominate u
- Dominance frontier = control dependence in reverse graph!

**Iterated dominance frontier**

- Irreflexive closure of dominance frontier relation
- Related notion: iterated control dependence in reverse graph
- Where to place Φ-functions for a variable Z
  - Let Assignments = {START} U {nodes with assignments to Z in original CFG}
  - Find set I = iterated dominance frontier of nodes in Assignments
  - Place Φ-functions in nodes of set I
- For example
  - Assignments = {START,A,C}
  - DF(Assignments) = {E}
  - DF(DF(Assignments)) = {B}
  - So I = {E,B}
  - This is where we place Φ-functions, which is correct
Why is SSA form useful?

• For many dataflow problems, SSA form enables sparse dataflow analysis that
  – yields the same precision as bit-vector CFG-based dataflow analysis
  – but is asymptotically faster since it permits the exploitation of sparsity
  – see lecture notes from Sept 6th
• SSA has two distinct features
  – factored def-use chains
  – renaming
  – you do not have to perform renaming to get advantage of SSA for many dataflow problems

Computing SSA form

• Cytron et al algorithm
  – compute DF relation (see slides on computing control-dependence relation)
  – find irreflexive transitive closure of DF relation for set of assignments for each variable
• Computing full DF relation
  – Cytron et al algorithm takes \(O(|V| + |DF|)\) time
  – \(|DF|\) can be quadratic in size of CFG
• Faster algorithms
  – \(O(|V| + |E|)\) time per variable: see Biliardi and Pingali

Dependences

• We have seen control-dependences.
• What other kind of dependences are there in programs?
  – Data dependences: dependences that arise from reads and writes to memory locations
• Think of these as constraints on reordering of statements

Data dependences

• Flow-dependence (read-after-write): \(S1 \rightarrow S2\)
  – Execution of \(S2\) may follow execution of \(S1\) in program order
  – \(S1\) may write to a memory location that may be read by \(S2\)
  – Example:
    
    ```
    x := 3
    ...
x
    ```
    
    ```
    flow-dependence
    ```
    
    while e do
    
    ```
    x := ...
    ```
    
    ```
    flow-dependence
    ```
    
    ```
    ....
    ```
    
    This is called a loop-carried dependence
**Anti-dependences**

- **Anti-dependence (write-after-read):** $S_1 \rightarrow S_2$
  - Execution of $S_2$ may follow execution of $S_1$ in program order
  - $S_1$ may read from a memory location that may be (over)written by $S_2$
  - Example:
    
    ```
    x := ... 
    ...x... 
    x := ...  \rightarrow \text{anti-dependence}
    ```

**Output-dependence**

- **Output-dependence (write-after-write):** $S_1 \rightarrow S_2$
  - Execution of $S_2$ may follow execution of $S_1$ in program order
  - $S_1$ and $S_2$ may both write to same memory location

**Summary of dependences**

- **Dependence**
  - **Data-dependence:** relation between nodes
    - Flow- or read-after-write (RAW)
    - Anti- or write-after-read (WAR)
    - Output- or write-after-write (WAW)
  - **Control-dependence:** relation between nodes and edges