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 directed graph
- Unique node START from which all nodes in CFG are reachable
- Unique node END reachable from all nodes
- Dummy edge to simplify discussion \( \text{START} \rightarrow \text{END} \)
- Path in CFG: sequence of nodes, possibly empty, such that successive nodes in sequence are connected in CFG by edge
  - If \( x \) is first node in sequence and \( y \) is last node, we will write the path as \( x \rightarrow^* y \)
  - If path is non-empty (has at least one edge) we will write \( x \rightarrow^+ y \)

Dominators

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

• Dataflow problem:

\[ \text{Dom}(N) = \{N\} \cup \bigcap_{M \in \text{pred}(N)} \text{Dom}(M) \]

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: \( a \text{ dom } a \)
  – anti-symmetric: \( a \text{ dom } b \) and \( b \text{ dom } a \Rightarrow a = b \)
  – transitive: \( a \text{ dom } b \) and \( b \text{ dom } c \Rightarrow a \text{ dom } c \)

  – tree-structured:
    • \( a \text{ dom } c \) and \( b \text{ dom } c \Rightarrow a \text{ dom } b \) or \( b \text{ dom } a \)
    • intuitively, this means dominators of a node are themselves ordered by dominance

Example of proof

• Let us prove that dominance is transitive.
  – Given: \( a \text{ dom } b \) and \( b \text{ dom } c \)
  – Consider any path \( P: \text{START} \rightarrow c \)
  – Since \( b \text{ dom } c \), \( P \) must contain \( b \).
  – Consider prefix of \( P = Q: \text{START} \rightarrow b \)
  – \( Q \) must contain \( a \) because \( a \text{ dom } b \).
  – Therefore \( P \) contains \( a \).
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
  - \( O(E \alpha(E)) \) 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 $P$: START $\rightarrow^+ a$ that does not contain $\text{idom}(b)$. Concatenating edge $a \rightarrow b$ to path $P$, we get a path from 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 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 START and END.
• Caveat: $a \text{ dom } b$ does not necessarily imply $b \text{ pdom } a$.
  – See example: $a \text{ dom } b$ but $b$ does not 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: $e \rightarrow S_1 \rightarrow m \rightarrow \text{START}$ $\rightarrow \ldots$ if $e$ then $S_1$ else $S_2$ $\ldots$ END
  We would say $S_1$ and $S_2$ are control-dependent on $e$
Examples (contd.)

```
while e do S1;
```

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

Example (contd.)

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 \((u \rightarrow v)\) 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 \(u \rightarrow v\)?
  – 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 \((u \rightarrow v)\) if
  – w postdominates v
  – if w is not u, w does not strictly postdominate u

Strict postdominance

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

Example
Computing control-dependence relation

- Control dependence: given a CFG G, a node w is control-dependent on an edge \((u \rightarrow v)\) if
  - w postdominates v
  - w does not strictly postdominate u

- 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 \([v, \text{ipdom}(u))\)
  - half-open interval in tree

Nodes control dependent on edge \((u \rightarrow v)\) are nodes on path up the postdominator tree from v to ipdom(u), excluding ipdom(u)

Computing control-dependence relation

- Compute the postdominator tree
- Overlay each edge \(u \rightarrow v\) on pdom tree and determine nodes in interval \([v, \text{ipdom}(u))\)
- Time and space complexity is \(O(EV)\).
- Faster solution: in practice, we do not want the full relation, we only make queries
  - \(cd(e)\): what are the nodes control-dependent on an edge \(e\)?
  - \(conds(w)\): 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 $Z_3$ 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**

- Compute $M: V \rightarrow P(V)$
- If node $N$ contains an assignment to a variable $x$, then node $Z$ is in $M(N)$ if:
  1. There is a non-null path $P_1 \Rightarrow N \Rightarrow Z$
  2. The value computed at $X$ reaches $Z$
  3. $P_1$ and $P_2$ are disjoint except for $Z$
- If $S \subseteq V$ where there are assignments to variable $x$, then place $\phi$ functions for $x$ in nodes $\bigcup_{N \in S} M(N)$
Computing Merge(v)

• If $u \in \text{Merge}(w)$, $w$ does not strictly dominate $u$
  – Proof: there is a path from START to $v$ that does not contain $w$
• Conversely
  – if $w$ dominates $u$, $u \not\in \text{Merge}(w)$
• Idea:
  – compute nodes on the dominance frontier of $w$
  • $w$ does not strictly dominate $u$
  but dominates some CFG predecessor of $u$
  – iterate

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

Running example:

Iterated dominance frontier

• Irreflexive closure of dominance frontier relation
• Related notion: iterated control dependence in reverse graph
• Where to place $\phi$-functions for a variable $Z$
  – Let Assignments = {START} U {nodes with assignments to $Z$ in original CFG}
  – Find set $I = \text{iterated dominance frontier of nodes in Assignments}$
  – Place $\phi$-functions in nodes of set $I$
• For example
  – Assignments = {START,A,C}
  – DF(Assignments) = {B}
  – DF(DF(Assignments)) = {B}
  – DF(DF(DF(Assignments))) = {B}
  – So $I = \{E,B\}$
  – This is where we place $\phi$-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
• 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 Bilardi 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 := ...
      ...x...
    ....
    ```
    - flow-dependence
    
    This is called a loop-carried dependence

Anti-dependences

- Anti-dependence (write-after-read): S1 $\rightarrow$ S2
  - Execution of S2 may follow execution of S1 in program order
  - S1 may read from a memory location that may be (over)written by S2
  - Example:
    ```
    x := ...
    ...x....
    x:= ...
    ```
    - anti-dependence
Output-dependence

- Output-dependence (write-after-write): S1 → S2
  - Execution of S2 may follow execution of S1 in program order
  - S1 and S2 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