Dependence and Data Flow Models
Why Data Flow Models?

• Models from Chapter 5 emphasized control
  • Control flow graph, call graph, finite state machines

• We also need to reason about dependence
  • Where does this value of x come from?
  • What would be affected by changing this?
  • ...

• Many program analyses and test design techniques use data flow information
  - Often in combination with control flow
    • Example: “Taint” analysis to prevent SQL injection attacks
    • Example: Dataflow test criteria (Ch.13)
Learning objectives

• Understand basics of data-flow models and the related concepts (def-use pairs, dominators...)

• Understand some analyses that can be performed with the data-flow model of a program
  - The data flow analyses to build models
  - Analyses that use the data flow models

• Understand basic trade-offs in modeling data flow
  - variations and limitations of data-flow models and analyses, differing in precision and cost
Def-Use Pairs (1)

- A def-use (du) pair associates a point in a program where a value is produced with a point where it is used.
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value “uninitialized”)
  - Variable initialization
  - Assignment
  - Values received by a parameter
- **Use**: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns
Def-Use Pairs

```plaintext
... if (...) {
    x = ... ;
... } y = ... + x + ... ;
```

Definition: x gets a value

Use: the value of x is extracted

Def-Use path
/** Euclid's algorithm */
public class GCD {
    public int gcd(int x, int y) {
        int tmp;               // A: def x, y, tmp
        while (y != 0) {       // B: use y
            tmp = x % y;       // C: def tmp; use x, y
            x = y;             // D: def x; use y
            y = tmp;           // E: def y; use tmp
        }
        return x;             // F: use x
    }
}
Def-Use Pairs (3)

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without* another definition of the variable between.
  - If, instead, another definition is present on the path, then the latter definition **kills** the former.

- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use.

*There is an over-simplification here, which we will repair later.
Definition-Clear or Killing

Path A..C is not definition-clear
Path B..C is definition-clear

\[
x = \ldots \quad \text{// A: def } x
\]
\[
q = \ldots
\]
\[
x = y; \quad \text{// B: kill } x, \text{ def } x
\]
\[
z = \ldots
\]
\[
y = f(x); \quad \text{// C: use } x
\]
(Direct) Data Dependence Graph

- A direct data dependence graph is:
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (du) pairs, labelled with the variable name

(Figure 6.3, page 80)
Data Flow Analysis

Computing data flow information
Calculating def-use pairs

- Definition-use pairs can be defined in terms of paths in the program control flow graph:
  - There is an association \((d,u)\) between a definition of variable \(v\) at \(d\) and a use of variable \(v\) at \(u\) iff
    - there is at least one control flow path from \(d\) to \(u\)
    - with no intervening definition of \(v\).
  - \(v_d\) reaches \(u\) \((v_d\text{ is a reaching definition at }u)\).
  - If a control flow path passes through another definition \(e\) of the same variable \(v\), \(v_e\text{ kills }v_d\) at that point.

- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.
Exponential paths
(even without loops)

Tracing each path is not efficient, and we can do much better.

2 paths from A to B
4 from A to C
8 from A to D
16 from A to E
...
128 paths from A to V
DF Algorithm

An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node.

Suppose we are calculating the reaching definitions of node n, and there is an edge (p, n) from an immediate predecessor node p.

- If the predecessor node p can assign a value to variable \( v \), then the definition \( v_p \) reaches n. We say the definition \( v_p \) is generated at p.
- If a definition \( v_p \) of variable \( v \) reaches a predecessor node p, and if \( v \) is not redefined at that node (in which case we say the \( v_p \) is killed at that point), then the definition is propagated on from p to n.
Equations of node E \((y = \text{tmp})\)

```
public class GCD {
    public int gcd(int x, int y) {
        int tmp;               // A: def x, y, tmp
        while (y != 0) {     // B: use y
            tmp = x % y;     // C: def tmp; use x, y
            x = y;           // D: def x; use y
            y = tmp;           // E: def y; use tmp
        }
        return x;              // F: use x
    }
}
```

Calculate reaching definitions at E in terms of its immediate predecessor D

Reach\((E) = \text{ReachOut}(D)\)
ReachOut\((E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}\)

(c) 2007 Mauro Pezzè & Michal Young
Equations of node B (while (y != 0))

public class GCD {
    public int gcd(int x, int y) {
        int tmp;               // A: def x, y, tmp
        while (y != 0) {     // B: use y
            tmp = x % y;     // C: def tmp; use x, y
            x = y;               // D: def x; use y
            y = tmp;           // E: def y; use tmp
        }
        return x;              // F: use x
    }
}

This line has two predecessors:
Before the loop, end of the loop

• Reach(B) = ReachOut(A) \( \cup \) ReachOut(E)
• ReachOut(A) = gen(A) = \{x_A, y_A, tmp_A\}
• ReachOut(E) = (Reach(E) \ \{y_A\}) \cup \{y_E\}
General equations for Reach analysis

\[ \text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m) \]

\[ \text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n) \]

\[ \text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \} \]

\[ \text{kill}(n) = \{ v_x \mid v \text{ is defined or modified at } x, x \neq n \} \]
Avail equations

Avail \( (n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m) \)

\( \text{AvailOut}(n) = (\text{Avail} (n) \setminus \text{kill} (n)) \cup \text{gen}(n) \)

\( \text{gen}(n) = \{ \text{exp} | \text{exp is computed at n} \} \)

\( \text{kill}(n) = \{ \text{exp} | \text{exp has variables assigned at n} \} \)
Live variable equations

\[ \text{Live}(n) = \bigcup_{m \in \text{succ}(n)} \text{LiveOut}(m) \]

\[ \text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n) \]

\[ \text{gen}(n) = \{ v \mid v \text{ is used at } n \} \]
\[ \text{kill}(n) = \{ v \mid v \text{ is modified at } n \} \]
Classification of analyses

- **Forward/backward**: a node’s set depends on that of its predecessors/successors
- **Any-path/all-path**: a node’s set contains a value iff it is coming from any/all of its inputs

<table>
<thead>
<tr>
<th></th>
<th>Any-path ((\cup))</th>
<th>All-paths ((\cap))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward (pred)</td>
<td>Reach</td>
<td>Avail</td>
</tr>
<tr>
<td>Backward (succ)</td>
<td>Live</td>
<td>“inevitable”</td>
</tr>
</tbody>
</table>
Iterative Solution of Dataflow Equations

- Initialize values (first estimate of answer)
  - For “any path” problems, first guess is “nothing” (empty set) at each node
  - For “all paths” problems, first guess is “everything” (set of all possible values = union of all “gen” sets)

- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a “fixed point” solution where every new calculation produces the same value as the previous guess.
Cooking your own: From Execution to Conservative Flow Analysis

- We can use the same data flow algorithms to approximate other dynamic properties
  - Gen set will be “facts that become true here”
  - Kill set will be “facts that are no longer true here”
  - Flow equations will describe propagation

- Example: Taintedness (in web form processing)
  - “Taint”: a user-supplied value (e.g., from web form) that has not been validated
  - Gen: we get this value from an untrusted source here
  - Kill: we validated to make sure the value is proper
Data flow analysis with arrays and pointers

- Arrays and pointers introduce uncertainty: Do different expressions access the same storage?
  - a[i] same as a[k] when i = k
  - a[i] same as b[i] when a = b (aliasing)

- The uncertainty is accommodated depending on the kind of analysis
  - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
  - All-path: vice versa
Scope of Data Flow Analysis

- **Intraprocedural**
  - Within a single method or procedure
    - as described so far

- **Interprocedural**
  - Across several methods (and classes) or procedures

- Cost/Precision trade-offs for interprocedural analysis are critical, and difficult
  - context sensitivity
  - flow-sensitivity
A context-sensitive (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();

A context-insensitive (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()
Flow Sensitivity

• Reach, Avail, etc. were flow-sensitive, intraprocedural analyses
  - They considered ordering and control flow decisions
  - Within a single procedure or method, this is (fairly) cheap – $O(n^3)$ for $n$ CFG nodes

• Many interprocedural flow analyses are flow-insensitive
  - $O(n^3)$ would not be acceptable for all the statements in a program!
    • Though $O(n^3)$ on each individual procedure might be ok
  - Often flow-insensitive analysis is good enough … consider type checking as an example
Summary

- Data flow models detect patterns on CFGs:
  - Nodes initiating the pattern
  - Nodes terminating it
  - Nodes that may interrupt it
- Often, but not always, about flow of information (dependence)
- Pros:
  - Can be implemented by efficient iterative algorithms
  - Widely applicable (not just for classic “data flow” properties)
- Limitations:
  - Unable to distinguish feasible from infeasible paths
  - Analyses spanning whole programs (e.g., alias analysis) must trade off precision against computational cost