Data Flow

- Another view - program statements compute and transform data…
  - So, look at how that data is passed through the program.
- Reason about **data** dependence
  - A variable is used here - where does its value come from?
  - Is this value ever used?
  - Is this variable properly initialized?
  - If the expression assigned to a variable is changed what else would be affected?
Data Flow

- Basis of the optimization performed by compilers.
- Used to derive test cases.
  - Have we covered the dependencies?
- Used to detect faults and other anomalies.
  - Is this string tainted by a fault in the expression that calculates its value?
Definition-Use Pairs

● Data is defined.
  ○ Variables are declared and assigned values.
● … and data is used.
  ○ Those variables are used to perform computations.
● Associations of definitions and uses capture the flow of information through the program.
  ○ Definitions occur when variables are declared, initialized, assigned values, or received as parameters.
  ○ Uses occur in expressions, conditional statements, parameter passing, return statements.
Data Dependence

- If a definition is impacted by a fault, all uses of that definition will be too.
- Uses are dependent on definitions.
- Tests and analyses that focus on these dependencies are likely to detect faults.
- Tests and analyses can be designed to cover different def-use pairs.
Data Dependence

Data dependency can be visualized.
  - Data dependence graph
  - Paired with control-flow graph.
  - Nodes = statements
  - Edges = data dependence

```java
public int gcd(int x, int y) {
    int tmp;
    tmp = x % y;
    y = tmp;
    while (y != 0) {
        x = y;
        y = tmp;
    }
    return x;
}
```
public int gcd(int x, int y) {
    int tmp;
    tmp = x % y;
    y = tmp;
    while (y != 0) {
        x = y;
        y = tmp;
    }
    return x;
}
Control-Dependence

● A node that is reached on every execution path from entry to exit is control dependent only on the entry point.
● For any other node N, that is reached on some - but not all - paths, there is some branch that controls whether that node is executed.
● Node M dominates node N if every path from the root of the graph to N passes through M.
Control Dependence Graph

Which statement controls the execution of a statement of interest?

- In a CFG, order is imposed whether it matters or not.
  - If there is dependency, then the order does matter.
- CDG shows only dependencies.
- Often combined with DDG.

```java
public int gcd(int x, int y) {
    int tmp;
    tmp = x % y;
    y = tmp;
    while (y != 0) {
        x = y;
        y = tmp;
    }
    return x;
}
```
Forming the Control Dependence Graph

```java
public int gcd(int x, int y) {
    int tmp;
    tmp = x % y
    y = tmp;
    while (y != 0) {
        x = y
        return x;
    }
}
```
Domination

- Nodes typically have many dominators.
- Except for the root, a node will have a unique **immediate dominator**.
  - Closest dominator of N on any path from the root and which is dominated by all other dominators of N.
  - Forms a dependency tree.
- **Post-Domination** can also be calculated in the reverse direction of control flow, using the exit node as root.
Domination Example

- A pre-dominates all nodes
- G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
- C does not post-dominate B
- B is the immediate pre-dominator of G
- F does not pre-dominate G
Post-Dominators and Control Dependency

- Node $N$ is reached on some paths.
- $N$ is control-dependent on a node $C$ if that node:
  - Has two or more successor nodes.
  - Is not post-dominated by $N$.
  - Has a successor that is post-dominated by $N$. 
Control-Dependency Example

- Execution of F is not inevitable at B.
- Execution of F is inevitable at E.
- F is control-dependent on B - the last point at which it is not inevitable.
GCD Example

- B and F are inevitable
  - Only dependent on entry (A).
- C, D, and E (nodes in the loop) depend on the loop condition (B).

```java
public int gcd(int x, int y) {
    int tmp;
    tmp = x % y
    y = tmp;
    while (y != 0) {
        x = y
        tmp = x % y
        x = y
        y = tmp;
    }
    return x;
}
```
Data Flow Analysis
Reachability

- Def-Use pairs describe paths through the program’s control flow.
  - There is a \((d,u)\) pair for variable \(V\) only if at least one path exists between \(d\) and \(u\).
  - If this is the case, a definition \(V_d\) reaches \(u\).
    - \(V_d\) is a reaching definition at \(u\).
  - If the path passes through a new definition \(V_e\), then \(V_e\) kills \(V_d\).
Computing Def-Use Pairs

- One algorithm: Search the CFG for paths without redefinitions.
  - Not practical - remember path coverage?
- Instead, summarize the reaching definitions at a node over all paths reaching that node.
Computing Def-Use Pairs

- If we calculate the reaching definitions of node $n$, and there is an edge $(p, n)$ from an immediate predecessor node $p$.
  - If $p$ can assign a value to variable $v$, then definition $v_p$ reaches $n$.
    - $v_p$ is generated at $p$.
  - If a definition $v_d$ reaches $p$, and if there is no new definition, then $v_d$ is propagated from $p$ to $n$.
    - If there is a new definition, $v_p$ kills $v_d$ and $v_p$ propagates to $n$. 
Computing Def-Use Pairs

- The reaching definitions flowing out of a node include:
  - All the reaching definitions flowing in
  - Minus the definitions that are killed
  - Plus the definitions that are generated
Flow Equations

- As node $n$ may have multiple predecessors, we must merge their reaching definitions:
  - $\text{ReachIn}(n) = \bigcup_{p \in \text{pred}(n)} \text{ReachOut}(p)$

- The definitions that reach out are those that reach in, minus those killed, plus those generated.
  - $\text{ReachOut}(n) = (\text{ReachIn}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$
Computing Reachability

- **Initialize**
  - $ReachOut$ is empty for every node.

- **Repeatedly update**
  - Pick a node and recalculate $ReachIn$, $ReachOut$.

- **Stop when stable**
  - No further changes to $ReachOut$ for any node
  - Guaranteed because the flow equations define a monotonic function on the finite lattice of possible sets of reaching definition.
Iterative Worklist Algorithm

- Initialize the reaching definitions flowing out to an empty set.
- Keep a *worklist* of nodes to be processed.
- At each step remove an element from the *worklist* and process it.
- Calculate the flow equations.

If the recalculated value is different for the node add its successors to the worklist.

```
for(n ∈ nodes)
    ReachOut(n) = {};
}
workList = nodes;
while(workList != {}){
    n = a node from the workList;
    workList = workList \ {n};
    oldVal = ReachOut(n);
    ReachIn(n) = \bigcup_{p ∈ \text{pred}(n)} \text{ReachOut}(p);
    ReachOut(n) = (ReachIn(n) \ kill(n)) \cup \text{gen}(n)
    if(ReachOut != oldVal){
        workList = workList \cup \text{succ}(n);
    }
}
```
Can this algorithm work for other analyses?

- ReachIn/ReachOut are flow equations.
  - They describe passing information over a graph.
  - Many other program analyses follow a common pattern.
- Initialize-Repeat-Until-Stable Algorithm
  - Would work for any set of flow equations as long as the constraints for convergence are satisfied.
- Another problem - expression availability.
Available Expressions

- When can the value of a subexpression be saved and reused rather than recomputed?
  - Classic data-flow analysis, often used in compiler construction.
- Can be defined in terms of paths in a CFG.
- An expression is *available* if - for all paths through the CFG - the expression has been computed and not later modified.
  - Expression is *generated* when computed.
  - … and *killed* when any part of it is redefined.
Available Expressions

- Like with reaching, availability can be described using flow equations.
- The expressions that become available (gen set) and cease to be available (kill set) can be computed simply.
- Flow equations:
  - $\text{AvailIn}(n) = \bigcap_{p \in \text{pred}(n)} \text{AvailOut}(p)$
  - $\text{AvailOut}(n) = (\text{AvailIn}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$
Iterative Worklist Algorithm

- **Input:**
  - A control flow graph $G = (\text{nodes}, \text{edges})$
  - $\text{pred}(n)$
  - $\text{succ}(n)$
  - $\text{gen}(n)$
  - $\text{kill}(n)$

- **Output:**
  - $\text{AvailIn}(n)$

```plaintext
for(n ∈ nodes) {
    $\text{AvailOut}(n) =$ set of all expressions defined anywhere;
}
workList = nodes;
while(workList != {}){
    n = a node from the workList;
    workList = workList \ {n};
    oldVal = $\text{AvailOut}(n)$;
    $\text{AvailIn}(n) =$ $\bigcap_{p \in \text{pred}(n)} \text{AvailOut}(p)$
    $\text{AvailOut}(n)$ = $(\text{AvailIn}(n) \ \text{\setminus} \ \text{kill}(n)) \cup \text{gen}(n)$
    if($\text{AvailOut} \neq \text{oldVal}$){
        workList = workList $\cup$ $\text{succ}(n)$;
    }
}
```
Analysis Types

● Both reaching definitions and expression availability are calculated on the CFG in the direction of program execution.
  ○ They are *forward* analyses.

● Definitions can reach across *any path*.
  ○ The in-flow equation uses a union.
  ○ This is a *forward, any-path* analysis.

● Expressions must be available on *all paths*.
  ○ The in-flow equation uses an intersection.
  ○ This is a *forward, all-paths* analysis.
Forward, All-Paths Analyses

- Encode properties as tokens that are generated when they become true, then killed when they become false.
  - The tokens are “used” when evaluated.
- Can evaluate properties of the form:
  - “G occurs on all execution paths leading to U, and there is no intervening occurrence of K between G and U.”
  - Variable initialization check:
    - G = variable-is-initialized, U = variable-is-used
    - K = variable-is-uninitialized (kill set is empty)
Backward Analysis - Live Variables

- Tokens can flow backwards as well.
- Backward analyses are used to examine what happens after an event of interest.
- “Live Variables” - analysis to determine whether the value held in a variable may be used.
  - A variable may be considered live if there is any possible execution path where it is used.
Live Variables

- A variable is live if its current value may be used before it is changed.
- Can be expressed as flow equations.
  - \( \text{LiveIn}(n) = \bigcup_{p \in \text{succ}(n)} \text{LiveOut}(p) \)
  - Calculated on successors, not predecessors.
  - \( \text{LiveOut}(n) = (\text{LiveIn}(n) \setminus \text{kill}(n)) \cup \text{gen}(n) \)
- Worklist algorithm can still be used, just using successors instead of predecessors.
Backwards, Any-Paths Analyses

- General pattern for backwards, any-path:
  - “After D occurs, there is at least one execution path on which G occurs with no intervening occurrence of K.”
    - D indicates a property of interest. G is when it becomes true. K is when it becomes false.
    - Useless definition check, D = variable-is-assigned, G = variable-is-used, K = variable-is-reassigned.
Backwards, All-Paths Analyses

● Check for a property that must inevitably become true.

● General pattern for backwards, all-path:
  ○ “After D occurs, G always occurs with no intervening occurrence of K.”
  ○ Informally, “D inevitably leads to G before K”
    ■ D indicates a property of interest. G is when it becomes true. K is when it becomes false.
    ■ Ensure interrupts are reenabled, files are closed, etc.
## Analysis Classifications

<table>
<thead>
<tr>
<th></th>
<th>Any-Paths</th>
<th>All-Paths</th>
</tr>
</thead>
</table>
| **Forward (pred)**  | **Reach**  
*U* may be preceded by *G* without an intervening *K*  
|                     | **Avail**  
*U* is always preceded by *G* without an intervening *K*  |
| **Backward (succ)** | **Live**  
*D* may lead to *G* before *K*  
|                     | **Inevitability**  
*D* always leads to *G* before *K*  |
Crafting Our Own Analysis

- We can derive a flow analysis from run-time analysis of a program.
- The same data flow algorithms can be used.
  - Gen set is “facts that become true at that point”
  - Kill set is “facts that are no longer true at that point”
  - Flow equations describe propagation
Monotonicity Argument

- **Constraint**: The outputs computed by the flow equations must be monotonic functions of their inputs.

- When we recompute the set of “facts”:
  - The gen set can only get larger or stay the same.
  - The kill set can only grow smaller or stay the same.
Example - Taint Analysis

● Built into Perl. Prevents program errors from data validation by detecting and preventing use of “tainted” data in sensitive operations.
● Tracks sources that variables are derived from. Looks for data derived from tainted data, and tracks corrupted program state.
  ○ String created from concatenating a tainted and a safe string is corrupted by the tainted string.
● Signals an error if tainted data is used in a potentially dangerous way.
Taint Analysis Variant

- Perl monitors values dynamically.
- Alternative - analysis that prevents data that could be tainted from ever being used in an unsafe manner.
- Forward, any-path analysis.
  - Tokens = tainted variables
  - Gen set = any variable assigned a tainted value
  - Kill set = variable cleansed of taintedness
Taint Analysis Variant

● Gen and kill sets depend on the set of tainted variables, which is not constant.
  ○ Circularity - tainted variable set also depends on gen and kill sets.

● Monotonicity property ensures soundness of the analysis.
  ○ We evaluate taintedness of an expression with the set \{a,b\}, then again with \{a,b,c\}. If it is tainted the first time, it must be tainted the second time.
We Have Learned

- Control-flow and data-flow both capture important paths in program execution.
- Analysis of how variables are defined and then used and the dependencies between definitions and usages can help us reveal important faults.
- Many forms of analysis can be performed using data flow information.
We Have Learned

- Analyses can be *backwards* or *forwards*.
  - … and require properties be true on *all-paths* or *any-path*.
  - Reachability is forwards, any-path.
  - Expression availability is forwards, all-paths.
  - Live variables are backwards, any-path.
  - Inevitability is backwards, all-paths.

- Many analyses can be expressed in this framework.
Next Class

- Data flow test adequacy criteria
- Data flow analysis with arrays and pointers.

- Reading: Chapter 13
- Assignment 2 out - due February 23