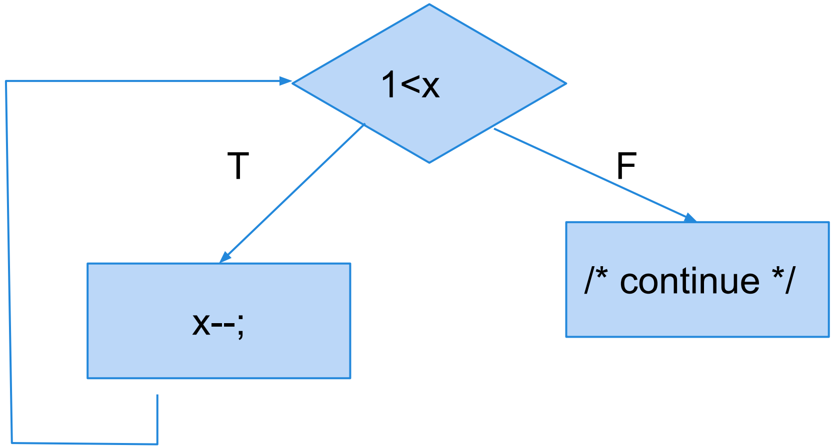
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 9: Data-Flow Analysis and Testing

Gregory Gay
DIT635 - February 19, 2020

Control Flow

- Capture dependencies in terms of how control passes between parts of a program.
- We care about the effect of a statement when it affects the path taken.
  - but deemphasize the information being transmitted.



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

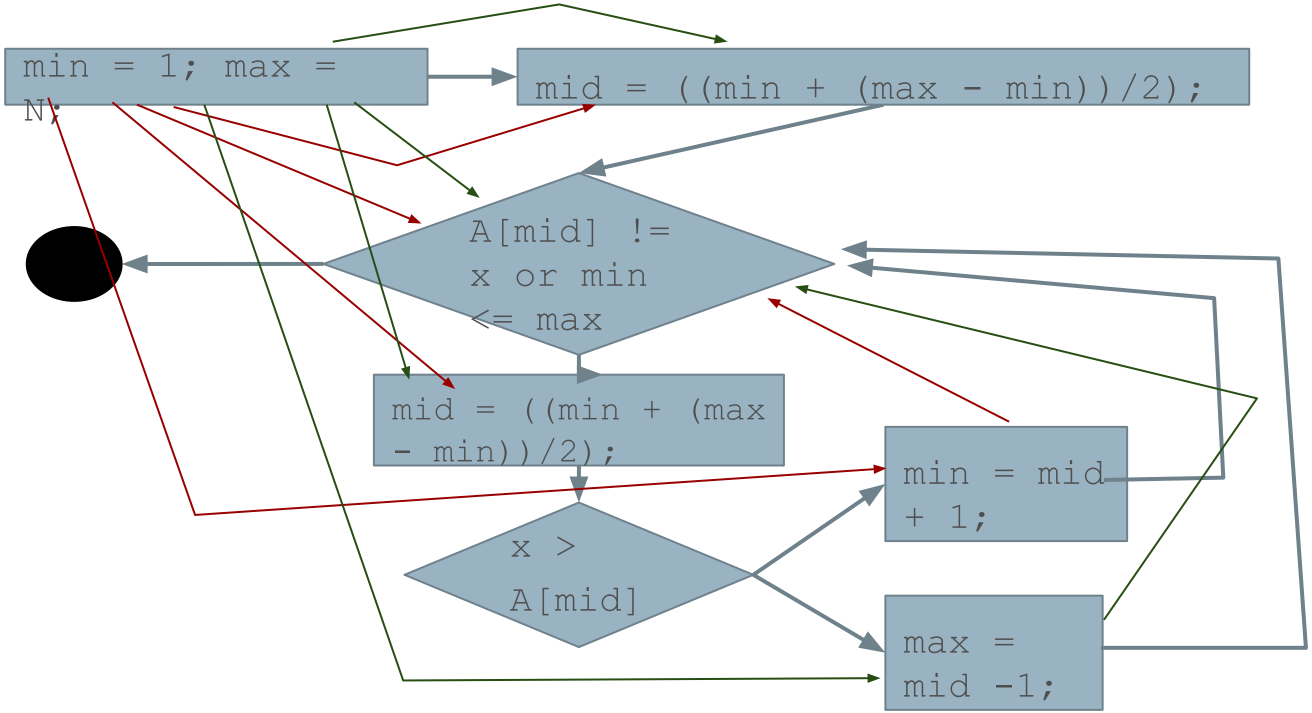
- Data is defined.
  - Variables are declared and assigned values.
- … and data is used.
  - Those variables are used to perform computations.
- Associations of definitions and uses capture the flow of information through the program.
  - Definitions occur when variables are declared, initialized, assigned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

Example - Definition-Use Pairs

```
1.  min = 1;
2.  max = N;
3.  mid = ((min + (max - min))/2);
4.  while (A[mid] != x or min <= max){
5.      mid = ((min + (max - min))/2);
6.      if (x > A[mid]){
7.          min = mid + 1
8.      } else {
9.          max = mid - 1;
10.     }
11. }
```

1. **def** - min
2. **def** - max, **use** - N
3. **def** - mid, **use** - min, max
4. **use** - A[mid], mid, x, min, max
5. **def** - mid, **use** - min, max
6. **use -** x, A[mid], mid
7. **def -** min, **use** - mid
8. -
9. **def -** max, **use -** mid

Example - Definition-Use Pairs



1. **def** - min
2. **def** - max, **use** - N
3. **def** - mid, **use** - min, max
4. **use** - A[mid], mid, x, min, max
5. **def** - mid, **use** - min, max
6. **use -** x, A[mid], mid
7. **def -** min, **use** - mid
8. -
9. **def -** max, **use -** mid

Def-Use Pairs

- We can say there is a def-use pair when:
  - There is a *def* (definition) of variable *x* at location A.
  - Variable *x* is *use*d at location B.
  - A control-flow path exists from A to B.
  - and the path is *definition-clear* for x.
    - If a variable is redefined, the original def is *killed* and the pairing is between the new definition and its associated use.

Example - Definition-Use Pairs

```
1.  min = 1;
2.  max = N;
3.  mid = ((min + (max - min))/2);
4.  while (A[mid] != x or min <= max){
5.      mid = ((min + (max - min))/2);
6.      if (x > A[mid]){
7.          min = mid + 1
8.      } else {
9.          max = mid - 1;
10.     }
11. }
```

DU Pairs

min: (1, 3), (1, 4), (1, 5), (7, 4), (7, 5)

max: (2, 3), (2, 4), (1, 5), (9, 4), (9, 5)

N: (0, 2)

mid: (3, 4), (5, 6), (5, 7), (5, 9), (5, 4)

x: (0, 4), (0, 6)

A: (0, 4), (0, 6)

Example - GCD

```
1.  public int gcd(int x, int y){
2.      int tmp;
3.      while(y!=0){
4.          tmp = x % y;
5.          x = y;
6.          y = tmp;
7.      }
8.      return x;
9.  }
```

1. def: x, y
2. def: tmp
3. use: y
4. use: x, y
   def: tmp
5. use: y
   def: x
6. use: tmp
   def: y
7. -
8. use: x

Example - GCD

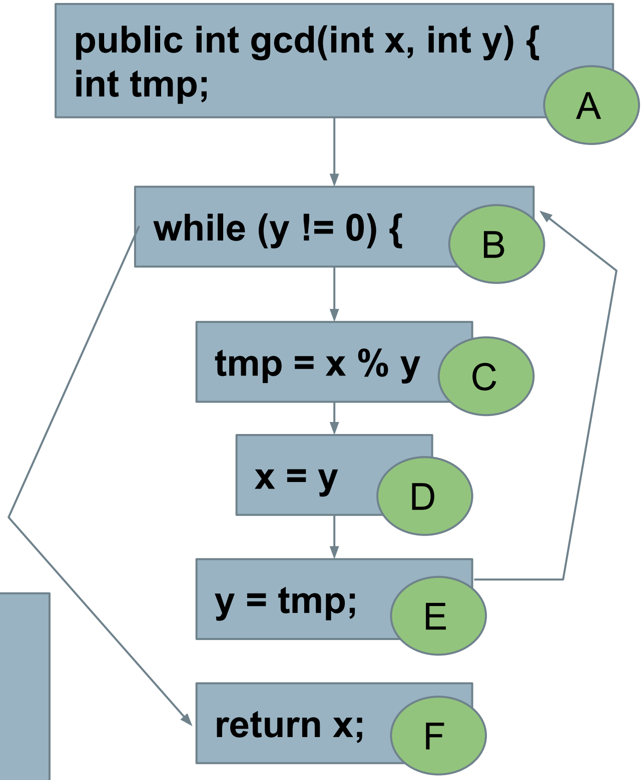
```
1.  public int gcd(int x, int y){
2.      int tmp;
3.      while(y!=0){
4.          tmp = x % y;
5.          x = y;
6.          y = tmp;
7.      }
8.      return x;
9.  }
```

Def-Use Pairs

x: (1, 4), (5, 4), (5, 8), (1, 8)

y: (1, 3), (1, 4), (1, 5), (6, 3), (6, 4), (6, 5)

tmp: (4, 6)



Example - collapseNewlines

```
7. public static String collapseNewlines(String argStr)
8. {
9.    char last = argStr.charAt(0);
10.   StringBuffer argBuf = new StringBuffer();
11.
12.   for(int cldx = 0; cldx < argStr.length(); cldx++)
13.   {
14.         char ch = argStr.charAt(cldx);
15.         if(ch != ‘\n’ || last != ‘\n’)
16.         {
17.               argBuf.append(ch);
18.               last = ch;
19.         }
20.   }
21.
22.   return argBuf.toString();
23. }
```

Variable	**D-U Pairs**
argStr	(7, 9), (7,12), (7, 14)
last	(9, 15), (18, 15)
argBuf	(17, 22)
cldx	(12, 12), (12, 14)
ch	(14, 15), (14, 17), (14, 18)

Dealing With Arrays/Pointers

- Arrays and pointers (including object references and arguments) introduce issues.
  - It is not possible to determine whether two access refer to the same storage location.
    - `a[x] = 13;`
      `k = a[y];`
      - Are these a def-use pair?
    - `a[2] = 42;`
      `i = b[2];`
      - Are these a def-use pair?
        - **Aliasing** = two names refer to the same memory location.

Aliasing

- ***Aliasing*** is when two names refer to the same memory location.
  - ```
    int[] a = new int[3]; int[] b = a;
    a[2] = 42;
    i = b[2];
    ```
  - a and b are aliases.
- Worse in C:
  ```
  p = &b;
  *(p + i) = k;
  ```
  ```

Uncertainty

- Dynamic references and aliasing introduce uncertainty into data flow analysis.
  - Instead of a definition or use of one variable, may have a potential def or use of a set of variables.
- Proper treatment depends on purpose of analysis:
  - If we examine variable initialization, might not want to treat assignment to a potential alias as initialization.
  - May wish to treat a use of a potential alias of *v* as a use of *v*.

Dealing With Uncertainty

- Basic option: Treat all potential aliases as definitions and uses of the same variable:

```
a[1] = 13;
k = a[2];
```

Def of a[1], use of a[2].

```
a[x] = 13;
k = a[y];
```

Def and use of array a.

- Easiest and cheapest option when performing analyses.
- Can be very imprecise.
    - They are only the same if x and y are the same.

Dealing With Uncertainty

- Treat uncertainty about aliases like uncertainty about control flow.

```
a[x] = 13;
k = a[y];
```

```
a[x] = 13;
if(x == y)    k = a[x];
else          k = a[y];
```

- In transformed code, all array references are distinct.

Situational Def-Use Pairs

- `++counter, counter++, counter+=1`
  `counter = counter + 1`
  - These are equivalent. They are a *use* of counter, then a new *definition* of counter.
- `char *ptr = *otherPtr`
  - Need a policy for how you deal with aliasing. Ad-hoc option:
    - Definition of **string** *ptr
    - Use of **index** ptr, string *otherPtr, and index otherPtr.
- `ptr++`
  - Use of index ptr, and definition of both index and string *ptr.
  - Change to index moves the pointer to a new location.

Dealing With Nonlocal Information

- fromCust and toCust may be references to same object.
    - from/toHome and from/toWork may also reference same object.
- Option - treat all nonlocal information as unknown.
    - Treat Customer/PhoneNum objects as potential aliases.
    - Be careful - may result in results so imprecise they are useless.

```
public void transfer(Customer fromCust,
Customer toCust){
    PhoneNum fromHome =
        fromCust.getHomePhone();
    PhoneNum fromWork =
        fromCust.getWorkPhone();
    PhoneNum toHome =
        toCust.getHomePhone();
    PhoneNum toWork =
        toCust.getWorkPhone();
}
```

Data Flow Testing

Overcoming Limitations of Path Coverage

- We can potentially expose many faults by targeting particular *paths* of execution.
- Full path coverage is impossible.
- What are the important paths to cover?
  - Some methods impose heuristic limitations.
    - Loop boundary coverage
  - Can also use data flow information to select a subset of paths based on how one element can affect the computation of another.

Choosing the Paths

- Branch or MC/DC coverage already cover many paths. What are the remaining paths that are important to cover?
- **Computing the wrong value leads to a failure only when that value is *used*.**
  - Pair definitions with usages.
  - Ensure that definitions are actually used.
  - Select a path where a fault is more likely to propagate to an observable failure.

All DU Pair Coverage

- Requires each DU pair be exercised in at least one program execution.
  - Erroneous values produced by one statement might be revealed if used in another statement.

$$\text{Coverage} = \frac{\text{number exercised DU pairs}}{\text{number of DU pairs}}$$

- Can easily achieve structural coverage without covering all DU pairs.

All DU Paths Coverage

- One DU pair might belong to many execution paths. Cover all simple (non-looping) paths at least once.
  - Can reveal faults where a path is exercised that should use a certain definition but doesn't.

$$\text{Coverage} = \frac{\text{number of exercised DU paths}}{\text{number of DU paths}}$$

Path Explosion Problem

- Even without looping paths, the number of SU paths can be exponential to the size of the program.
- When code between definition and use is irrelevant to that variable, but contains many control paths.

```
void countBits(char ch){
    int count = 0;
    if (ch & 1)   ++count;
    if (ch & 2)   ++count;
    if (ch & 4)   ++count;
    if (ch & 8)   ++count;
    if (ch & 16)  ++count;
    if (ch & 32)  ++count;
    if (ch & 64)  ++count;
    if (ch & 128) ++count;
    printf("'%c' (0X%02X) has %d bits set to 1\n", ch, ch, count);
}
```

All Definitions Coverage

- All DU Pairs/All DU Paths are powerful and often practical, but may be too expensive in some situations.
- Pair each definition with at least one use.

$$\text{Coverage} = \frac{\text{number of covered definitions}}{\text{number of definitions}}$$

Infeasibility Problem

- Metrics may ask for impossible test cases.
- Path-based metrics aggravates the problem by requiring infeasible combinations of feasible elements.
  - Alias analysis may add additional infeasible paths.
- All Definitions Coverage and All DU-Pairs Coverage often reasonable.
  - All DU-Paths is much harder to fulfill.

Activity - DU Pairs

- **Take a break, then...**
- Identify all DU pairs and write test cases to achieve All DU Pair Coverage.
  - Hint - remember that there is a loop.

```
int doSomething(int x, int y)
{
    while(y > 0) {
        if(x > 0) {
            y = y - x;
        }else {
            x = x + 1;
        }
    }
    return x + y;
}
```

Activity - DU Pairs

```
1.  int doSomething(int x, int y)
2.  {
3.      while(y > 0) {
4.          if(x > 0) {
5.              y = y - x;
6.          }else {
7.              x = x + 1;
8.          }
9.      }
10.     return x + y;
11. }
```

Variable	Defs	Uses
x	1, 7	4, 5, 7, 10
y	1, 5	3, 5, 10

Variable	D-U Pairs
x	(1, 4), (1, 5), (1, 7), (1, 10), (7, 4), (7, 5), (7, 7), (7, 10)
y	(1, 3), (1, 5), (1, 10), (5, 3), (5, 5), (5, 10)

Activity - DU Pairs

```
1.  int doSomething(int x, int y)
2.  {
3.      while(y > 0) {
4.          if(x > 0) {
5.              y = y - x;
6.          }else {
7.              x = x + 1;
8.          }
9.      }
10.     return x + y;
11. }
```

Variable	Defs	Uses
x	1, 7	4, 5, 7, 10
y	1, 5	3, 5, 10

Variable	D-U Pairs
x	~~(1, 4)~~, ~~(1, 5)~~, ~~(1, 7)~~, ~~(1, 10)~~, ~~(7, 4)~~, ~~(7, 5)~~, ~~(7, 7)~~, ~~(7, 10)~~
y	~~(1, 3)~~, ~~(1, 5)~~, ~~(1, 10)~~, ~~(5, 3)~~, ~~(5, 5)~~, ~~(5, 10)~~

Test 1: (x = 1, y = 2)
Covers lines 1, 3, 4, 5, 3, 4, 5, 3, 10
Test 2: (x = -1, y = 1)
Covers lines 1, 3, 4, 6, 7, 3, 4, 6, 7, 3, 4, 5, 3, 10
Test 3: (x = 1, y = 0)
Covers lines 1, 3, 8

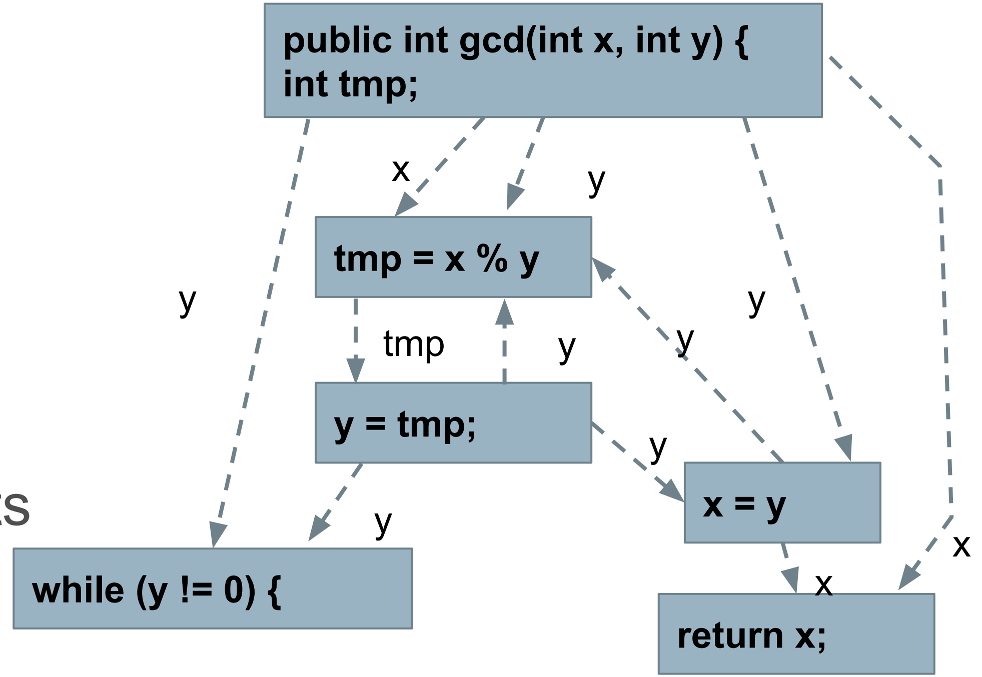
Data and Control Dependence

Data Dependence

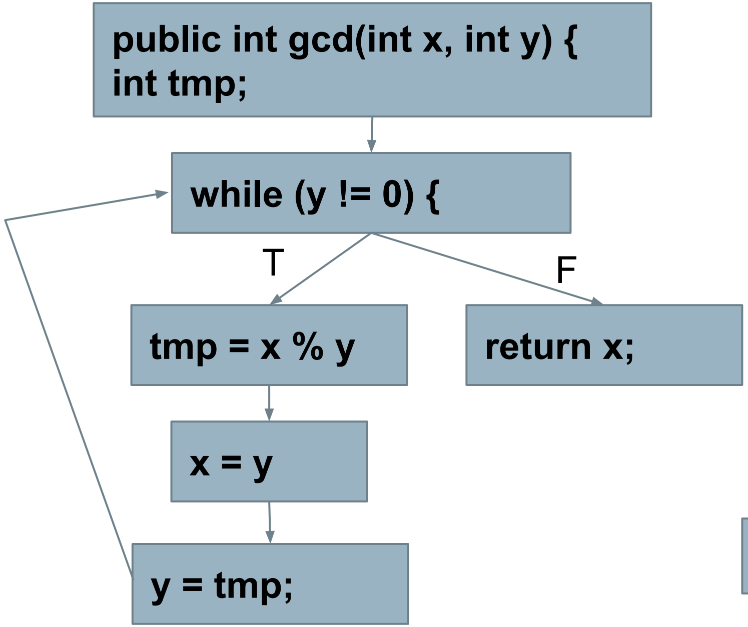
- If a definition is impacted by a fault, all uses of that definition will be too.
- Uses are *dependent* on definitions.
- Tests and analyses that focus on these dependencies are likely to detect faults.
- Tests and analyses can be designed to cover different def-use pairs.

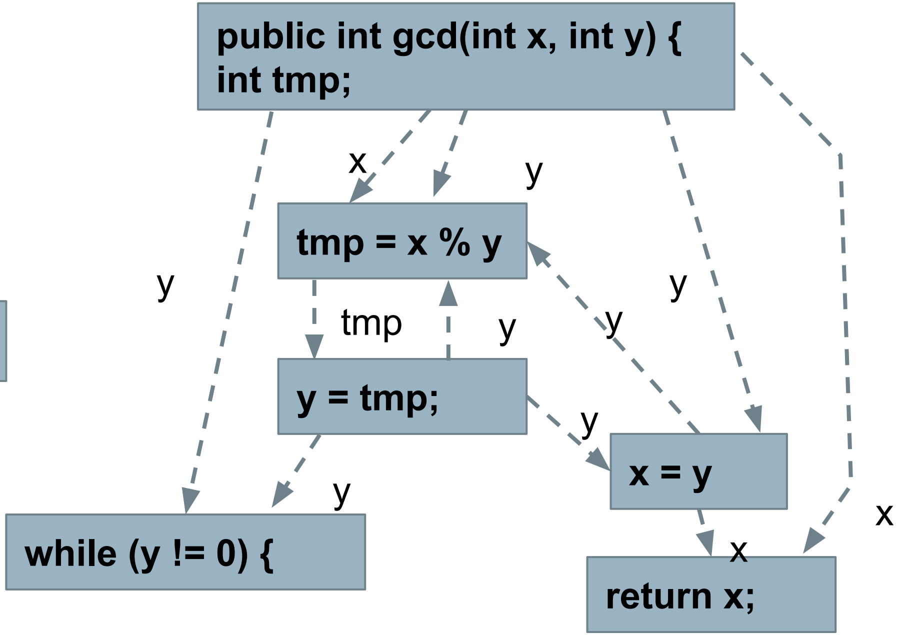
Data Dependence

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



Forming Data Dependence Graphs





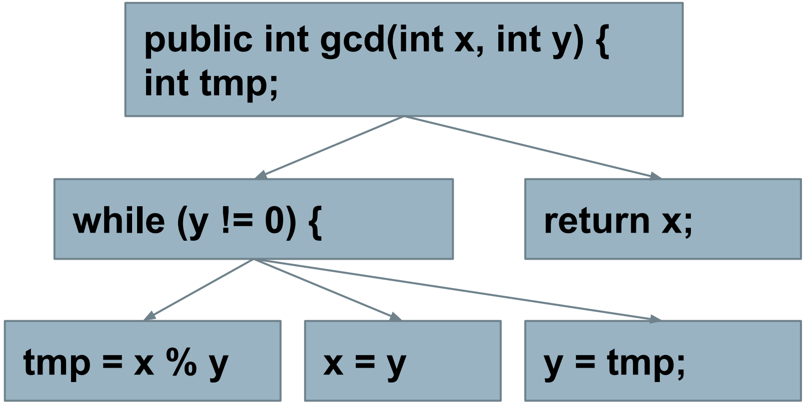
Control-Dependence

- 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.

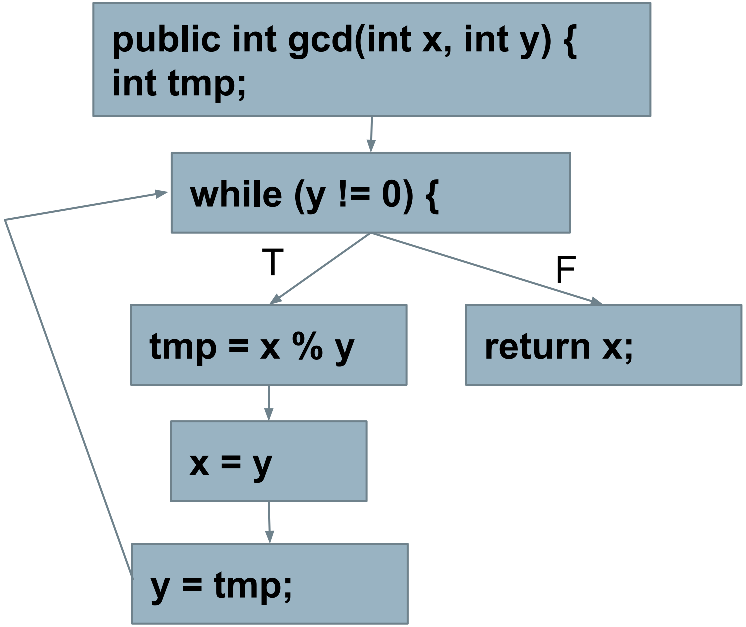
Control Dependence Graph

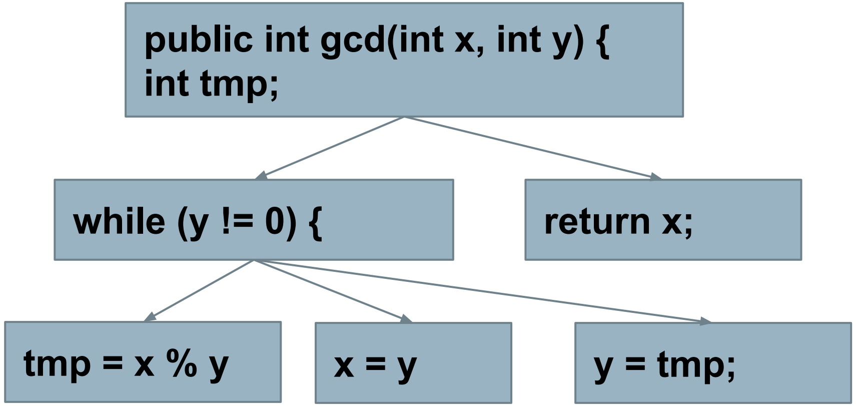
Which statement controls the execution of a statement of interest?

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



Forming Control Dependence Graphs



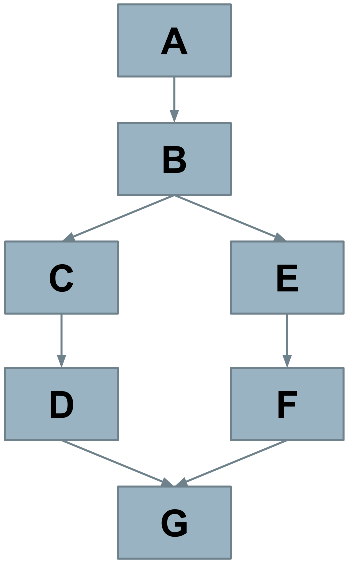


Domination

- 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

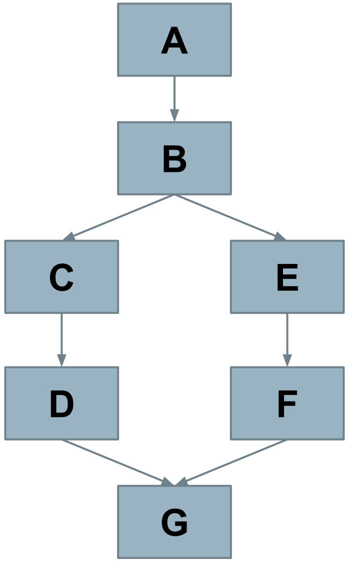


Post-Dominators and Control Dependency

- 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.

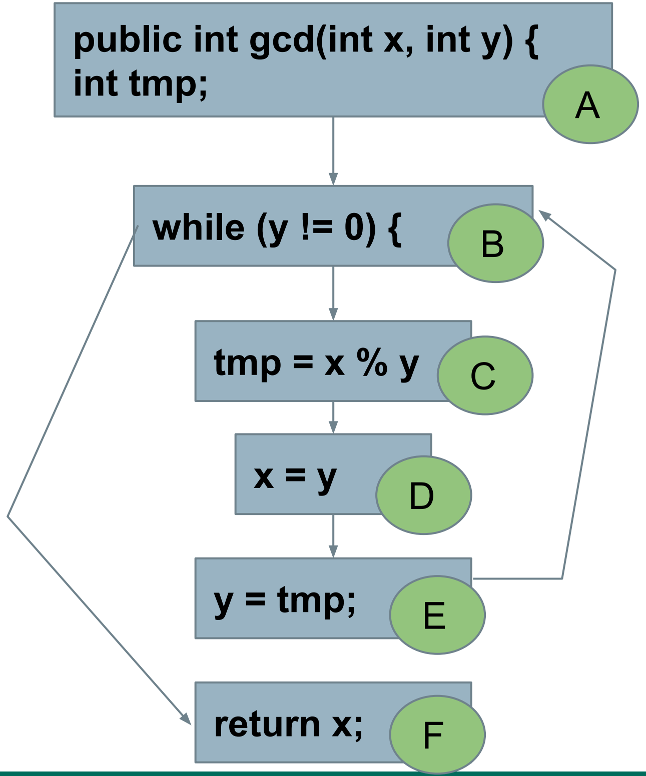
Control-Dependency Example

- 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).



Let’s Take a Break

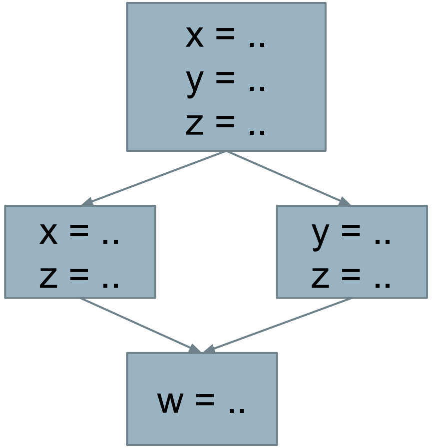
Data Flow Analysis

Reachability

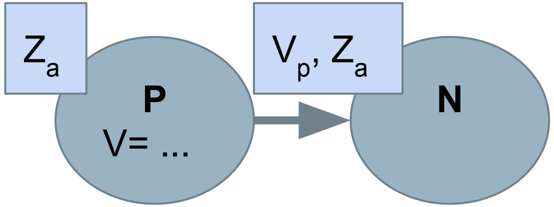
- Def-Use pairs describe paths through the program's control flow.
  - There is a (*d*,*u*) pair for variable *V* only if at least one path exists between *d* and *u*.
  - If this is the case, a definition V_d **reaches** *u*.
    - V_d is a *reaching definition* at *u*.
  - If the path passes through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



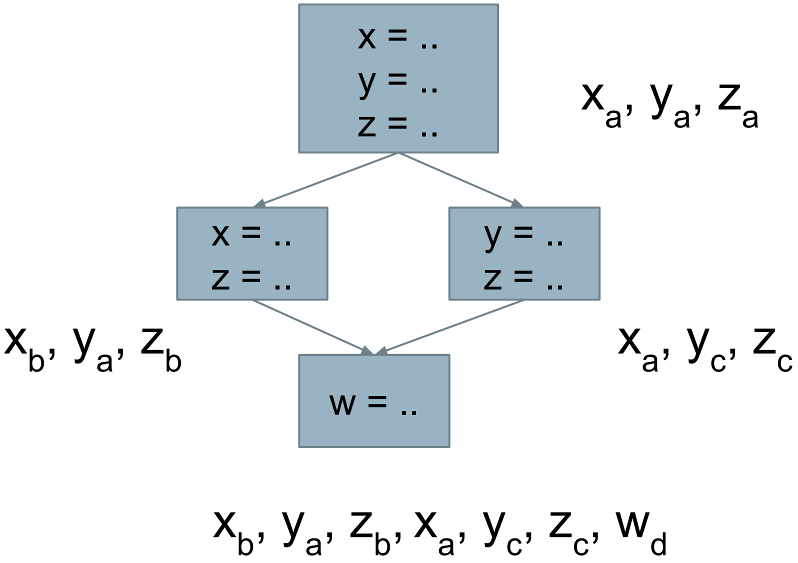
Computing Def-Use Pairs



- 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 reaching definitions flowing in.
    - Minus definitions that are killed.
    - Plus 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

Keep a *worklist* of nodes to be processed.

At each step remove an element from the *worklist*

Calculate the flow equations.

- Output:

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) = ∪p∈pred(n) ReachOut(p);
    ReachOut(n) = (ReachIn(n) \
                      kill(n)) ∪ gen(n);
    if(ReachOut != oldVal){
        workList = workList ∪ 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.
- 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:
  - $\mathrm{AvailIn}(n) = \bigcap_{p \in \mathrm{pred}(n)} \mathrm{AvailOut}(p)$
  - $\mathrm{AvailOut}(n) = (\mathrm{AvailIn}(n) \setminus \mathrm{kill}(n)) \cup \mathrm{gen}(n)$

Iterative Worklist Algorithm

- Input:
  - A control flow graph G = (nodes, edges)
  - pred(n)
  - succ(n)
  - gen(n)
  - kill(n)
- Output:
  - AvailIn(n)

```
for(n ∈ nodes){
    AvailOut(n) = set of all expressions
    defined anywhere;
}
workList = nodes;
while(workList != {}){
    n = a node from the workList;
    workList = workList \ {n};
    oldVal = AvailOut(n);
    AvailIn(n) = ∩_{p∈pred(n)} AvailOut(p)
    AvailOut(n) = (AvailIn(n) \ kill(n)) ∪
                  gen(n);
    if(AvailOut != oldVal){
        workList = workList ∪ 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.
  - Other analyses backtrack from exit to entrance (backwards analyses).

Analysis Types

- 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

	Any-Paths	**All-Paths**
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.

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.

We Have Learned

- If there is a fault in a computation, we can observe it by looking at where the computation is used.
- By identifying DU pairs and paths, we can create tests that trigger faults along those paths.
    - All DU Pairs coverage
    - All DU Paths coverage
    - All Definitions coverage

Next Time

- Integration Testing and Testing of OO Systems
  - Reading: Pezze and Young, Chapters 15, 21, 22.2
- Exercise Session (Friday) - Structural Testing
  - Bring laptops, download Meeting Planner code.
- Assignment 2
  - Due March 1!

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY