SWEN430 - Compiler Engineering

Lecture 9 - Static Analysis I: Unreachable Code

Linsday Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Static Analysis?

- After doing context context checking and type checking, a compiler can do various other tests to either (i) detect errors or (ii) produce better code. E.g.:
  - Detecting unreachable (dead) code
  - Detecting access to uninitialised variables
  - Dereferencing null pointers
  - Identifying unreachable objects (garbage)
- These usually involve analysing a whole method, or whole program, rather than just single statements/expressions (global v. local).
- Analysis is usually performed by traversing a Control-Flow Graph (CFG), rather than AST.
- Properties checked are usually undecidable, so we need to do a conservative analysis or approximation.
- This means we will get false positives and/or false negatives.

What is Unreachable (aka Dead) Code?

- Consider the following Java method:

```
int f(int x) {
 if(x > 0) {
  return 1;
 } else {
  throw new NullPointerException();
 }
 x = x + 1;
 if(x > 100) { return 3; }
 return 2;
}
```

- Can this method ever return 2 or 3?
- Will this compile under `javac`?

Why is Dead Code a Problem?

- Dead code is usually a sign of a program error.

  What is the code there for if it is unreachable?

- The Java source language does not permit dead code:

  JLS §14.21:

  > *"It is a compile-time error if a statement cannot be executed because it is unreachable. Every Java compiler must carry out the* **conservative** *flow analysis specified here to make sure all statements are reachable."*

Is this Dead Code?

- Consider this Java method:

```java
int f() {
 int x = 0;
 while(x < 10) { if(++x == 10) { return 1; } }
 return 2;
}
```

- Can this method ever return 2? Does it compile with `javac`?

- What about this method:

```java
int f(int x, int y, int z) {
 if(x<=0 || y<=0 || z<=0) { return -1; }
 else if(x > 1290 || y > 1290 || z > 1290) { return 0;}
 else if((x*x*x) != (y*y*y) + (z*z*z)) { return 1; }
 return 2;
}
```

- Can this method ever return 2?

Defining Dead Code

CFG

A *control-flow graph* (CFG) for a method is a directed graph, $G = (V, E)$, with a node for each atomic action (assignment, test, ...), and an edge $u \to v$ (possibly labelled with a condition) if control can pass u to v.

CFG Path

A *path* in a CFG is a sequence $[l_1, \ldots, l_n]$, where $l_k \to l_{k+1} \in (E)$.

Execution Path

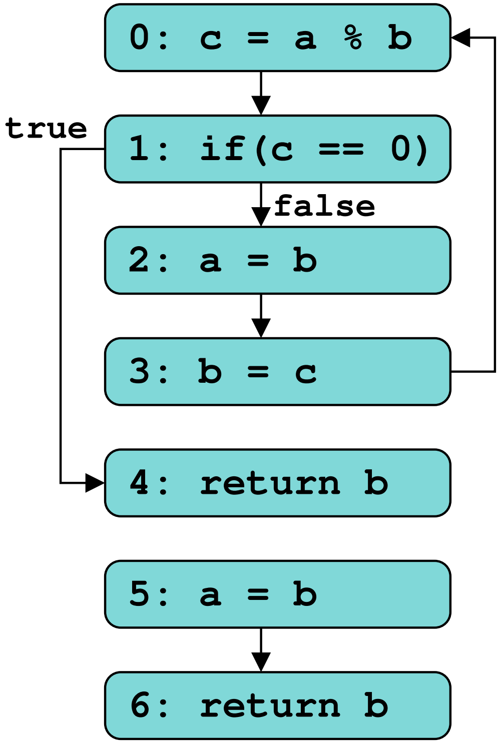
An execution path is a CFG path which starts at statement 0.

Dead Statements

A statement S in a CFG is *dead* iff no valid execution path includes S.

Defining Dead Code (Cont'd)

- Consider this Control-Flow Graph:



- $[1, 2, 3, 0, 1]$ is a valid *CFG path*
- $[4, 5]$ is not a valid *CFG path*
- $[0, 1, 4]$ is a valid *Execution path*
- $[1, 2]$ is not a valid *Execution path*

Finding Dead Code

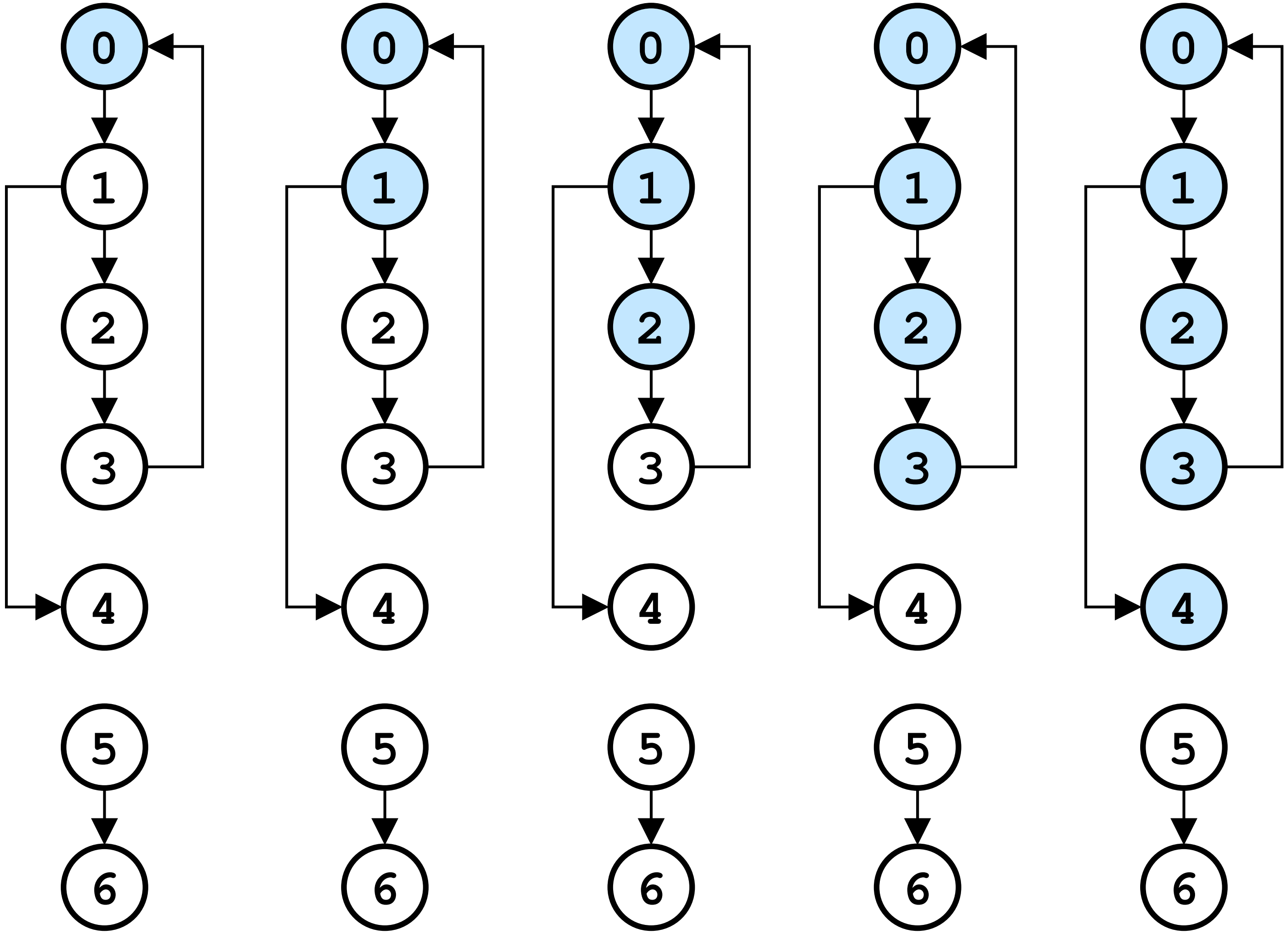
- To find dead-code, perform depth-first traversal from statement 0

```
procedure DFS
for all $v \in V$ do
    visited$(v)$ = false;
VISIT(0)

procedure VISIT$(v)$
visited$(v)$ = true;
for all $(v, w) \in (E \cup E^*)$ do
    if $\neg visited(w)$ then VISIT$(w)$
```

- After running DFS(), any vertex w where visited(w) = *false* is dead-code

Example DFS



Why is this Approximate?

- This analysis assumes that both outcomes for a test are possible.
- Consider:

  ```
  int f(int x) {
   if(B)
     ...
   else if(B)
       ...  // Is this reachable?
  }
  ```

  Will this be detected?
  Draw the CFG and see what the algorithm does.
- To detect such cases we need to reason about the outcomes of tests, niot just about the existence of (potential) execution paths.
- Look again at the examples on slide 5.

What does Java do?

- The following **does not** compile under `javac`:

```java
void f(int x) {
 while(false) { x=3; }
}
```

- The following **does** compile under `javac`:

```java
void f(int x) {
 if (false) { x=3; }
}
```

- Why is this a problem?
- Why does Java support such differing behaviour?
- See JLS §14.21