SWEN430 - Compiler Engineering (2018)

Lecture 10 - 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.:
  - Unreachable (dead) code
  - Access to uninitialised variables
  - Dereferencing null pointers
  - Unreachable objects (garbage)

  (NB: These are not the same!)
- These usually involve analysing a whole method, or whole program, rather than just single statements/expressions (global v. local).
- Analysis is often 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 Code?

- ```
  int f(int x) {
    return 1;
    return 2;
  }
  ```

- ```
  int f(int x) {
    if (x==0)
      return 1;
    else
      return 2;
    return 3;
  }
  ```

What is Unreachable Code?

- ```
  int f(int x) {
    if (true)
      return 1;
    return 3;
  }
  ```

- ```
  int f(int x) {
    while (true)
      return 1;
    return 3;
  }
  ```

- ```
  int f(int x) {
    if (false)
      return 1;
    return 3;
  }
  ```

- ```
  int f(int x) {
    while (false)
      return 1;
    return 3;
  }
  ```

What is Unreachable 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`?

What is Unreachable Code?

- Consider this Java method:

```
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:

```
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?

So what?

- Why is unreachable code a problem?
  - Unreachable code is usually a sign of a program error.

    What is the code there for if it is unreachable?
  - Increases size of object code, and can affect cache behaviour adversely.

  Ex: Is there ever a case where unreachable code is not an error?

- What can a compiler do about unreachable code?
  - Can't detect all cases — would need to solve the halting problem!
  - But can detect many common cases.

- What should a compiler do about unreachable code?
  - Ignore it — it's the programmer's responsibility,
  - Detect possible cases and issue error/warning.
  - Detect definite cases and issue error/warning.

What does Java do?

- The Java source language does not permit unreachable code:

  It is a compile-time error if a statement cannot be executed because it is unreachable.

  The idea is that there must be some possible execution path from the beginning of the constructor, method, instance initializer, or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of while, do, and for statements whose condition expression has the constant value true, the values of expressions are not taken into account in the flow analysis. (JLS §14.21)

  `https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.21`

- Why are while, do, and for statements treated specially?
- What is omitted here?
- Java bytecode does permit unreachable-code.

Detecting Unreachable Code

We can treat this as a reachability problem on graphs.

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 $l_k \to l_{k+1} \in E$.

Do we need to incorporate edge labels?

Execution Path

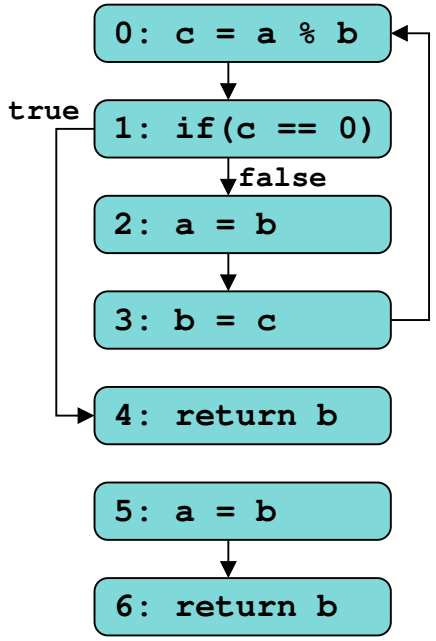
An execution path is a CFG path which starts at the root of the CFG.

Unreachable Statements

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

Detecting Unreachable Code

Consider this Control-Flow Graph:



E.g. from source code:

```
   int f(int x) {
     while (true) {
0:     c = a % b;
1:     if (c == 0) break;
2:     a = b;
3:     b = c;
     }
4:   return b;
5:   a = b;
6:   return b;
   }
```

- $[1, 2, 3, 0, 1]$ 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*

Detecting Unreachable Code

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

  procedure DFS

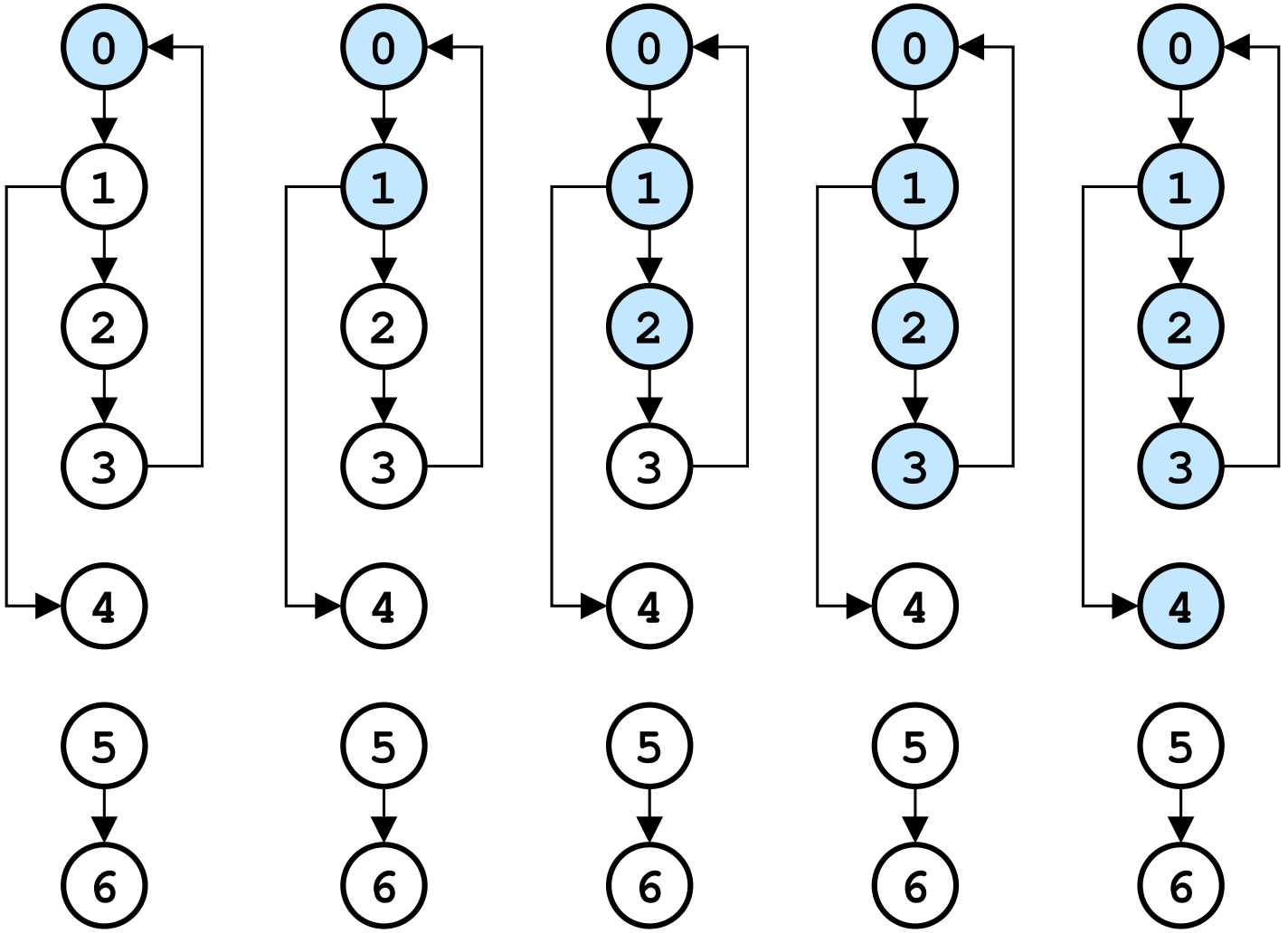
  1: **for all** $v \in V$ **do**
  2: visited(v) = false;
  3: VISIT(0)

  procedure VISIT(v)

  4: visited(v) = true;
  5: **for all** $(v, w) \in E$ **do**
  6: **if** $\neg$*visited*(w) **then** VISIT(w)

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

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, not 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`:

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

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

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

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

Ex: Extend the above definitions/algorithm to handle Java's special cases.

Ex: How do you construct the CFG for `switch` statements? For `try`/`throw` blocks?

What does the While compiler do?

- Analyses the AST, not CFG, so analysis is more aware of program structure.
- For each statement analysed, return a "token" indicating whether it does a break or return, or continues to the next statement.
- When analysing a sequence, if preceding statement returns break or return, this statement is unreachable.
- When analysing an `if` statement, must combine the results of analysing the true and false branches.

$\Longrightarrow$ See code.

Can we do better?

- Extend While's checker to check for `false` conditions in conditional and loop statements.

  Including results of constant expressions, constant definitions, ...

- Look for complementary conditions on paths.
  E.g. `if (x==0) { ... if (x==0) ... else ... }`

  What else do you need to check?

- Look for implied conditions on paths.
  E.g. `if (x > 0) { ... if (x >= 0) ... else ... }`

- Calculate weakest precondition for condition to be `false`, and give it to a theorem prover?

- How would tests change if we wanted to find possible cases?

- Another way to find possible unreachable code is via statement coverage in testing.