SWEN430 - Compiler Engineering (2018)

Lecture 10 - Static Analysis II: Definite Assignment/Unassignment

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Definite Assignment?

- Accessing uninitialised variables may produce unpredictable results.

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

  -
    ```
    class X {
     int field;
     int f() { return field; }
    }
    ```
- Can't always tell whether variables are/aren't initialised.

  -
    ```
    int f(int y) {
     int x;
     if (y == 1) { x = 1; }
     return x;
    }
    ```

More Interesting Examples

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

-
  ```
  int f(int y) {
   int x;
   for (int i=0; i!=y; ++i) { x = 2*i; }
   return x+1;
  }
  ```

-
  ```
  int f(int y) {
   int x;
   for (int i=(y-1); i!=y; ++i) { x = 2*i; }
   return x+1;
  }
  ```

What can/should we do about uninitalised varaibles?

- Ignore it — it's the programmer's responsibility,
- Make it a non-problem.

  Some programming languages define default values for all variables, fields, etc., so there is no such thing as an uninitialsied variable.

  Many argue that it is danerous to rely on such defaults.
- Detect possible cases and issue error/warning.

  Again, we can't detect all cases — would need to solve the halting problem!
- Detect definite cases and issue error/warning.

What does Java do?

- Java requires that the compiler can check that local variables and blank final fields are initialised.

  Each local variable (§14.4) and every blank final (§4.12.4) field (§8.3.1.2) must have a definitely assigned value when any access of its value occurs.

  For every access of a local variable or blank final field f, f is definitely assigned before the access; otherwise a compile-time error must occur. (JLS §16)

  `https://docs.oracle.com/javase/specs/jls/se7/html/jls-16.html`

- Also requries that a blank final field is definitely unassigned when it is assigned a value, so that it can only be assigned once.

- Checking is not required for non-final fields.

Overview of Analysis

- Definite assignment algorithm is form of **dataflow analysis**.
- Maybe performed on a Control Flow Graph (CFG), or AST.
- We want to determine for each node in the CFG which variables have been assigned values, and which variables are accessed.
- We can determine which variables are accessed by just analysing the statement/condition at that node.
- To determine which variables have been assigned values, we must consider the effects of all statements on paths to that node.

Overview of Analysis

- To determine which variables have been assigned values, we will keep track of:
  - variables defined on *entry* to the node
  - variables defined by statement at the node
  - variables defined on *exit* from the node
- The variables defined on exit from a node are those defined on entry to the node, along with those defined at the node.
- The variables defined on entry to a node are those defined on exit from *all* nodes with edges leading to that node.
- Method parameters are assumed to be defined on entry to the method.

Overview of Analysis

Definite Assignment Flow Sets

Let $D = (V, E)$ be the control-flow graph of a method.

For each $v \in V$, we define:

- $DEF_{IN}(v)$ — the set of variables defined on *entry* to v
- $DEF_{AT}(v)$ — the set of variables defined by statement at v
- $DEF_{OUT}(v)$ — the set of variables defined on *exit* from v
- $USES(v)$ — the set of variables used by statement at v

- It is an error if $USES(v) - DEF_{IN}(v) \neq \emptyset$ for some $v \in V$.

Calculating $DEF_{AT}(v)$

- $DEF_{AT}(v)$ is computed recursively over statements:

$$\begin{array}{lcrcl}
STMT(v) = \texttt{v = e} & \Rightarrow & DEF_{AT}(v) & = & \{v\} \\
STMT(v) = \texttt{e}_1\texttt{.f = e}_2 & \Rightarrow & DEF_{AT}(v) & = & \emptyset \\
STMT(v) = \texttt{e}_1\texttt{[e}_2\texttt{] = e}_3 & \Rightarrow & DEF_{AT}(v) & = & \emptyset \\
STMT(v) = \texttt{if(e) goto L} & \Rightarrow & DEF_{AT}(v) & = & \emptyset \\
STMT(v) = \texttt{return e} & \Rightarrow & DEF_{AT}(v) & = & \emptyset
\end{array}$$

- $STMT(v)$ gives the statement contained in node v
- Note that field and array element assignments are ignored

Calculating *USES*(v)

- *USES*(v) is computed recursively over statements and expressions:

$$
\begin{array}{lcrcl}
STMT(v) = \texttt{v = e} & \Rightarrow & USES(v) & = & USES_{EXPR}(e) \\
STMT(v) = \texttt{e}_1\texttt{.f = e}_2 & \Rightarrow & USES(v) & = & USES_{EXPR}(e_1) \cup \\
 & & & & USES_{EXPR}(e_2) \\
STMT(v) = \texttt{if(e) goto L} & \Rightarrow & USES(v) & = & USES_{EXPR}(e) \\
STMT(v) = \texttt{return e} & \Rightarrow & USES(v) & = & USES_{EXPR}(e) \\
\ldots & = & \ldots & &
\end{array}
$$

$$
\begin{array}{lcl}
USES_{EXPR}(\texttt{i}) & = & \emptyset \\
USES_{EXPR}(\texttt{v}) & = & \{v\} \\
USES_{EXPR}(\texttt{(T)e}) & = & USES_{EXPR}(e) \\
USES_{EXPR}(\texttt{e}_1\texttt{.f}) & = & USES_{EXPR}(e_1) \\
USES_{EXPR}(\texttt{e}_1\texttt{[e}_2\texttt{]}) & = & USES_{EXPR}(e_1) \cup USES_{EXPR}(e_2) \\
USES_{EXPR}(\texttt{e.m(}\overline{\texttt{e}}\texttt{)}) & = & USES_{EXPR}(e) \cup USES_{EXPR}(e_1) \cup \ldots \cup USES_{EXPR}(e_n) \\
USES_{EXPR}(\texttt{e}_1 + \texttt{e}_2) & = & USES_{EXPR}(e_1) \cup USES_{EXPR}(e_2) \\
\ldots & = & \ldots
\end{array}
$$

- Recall, *STMT*(v) gives the statement contained in node v

Dataflow Equations

To find $DEF_{IN}(v)$, we need to solve a set of equations.

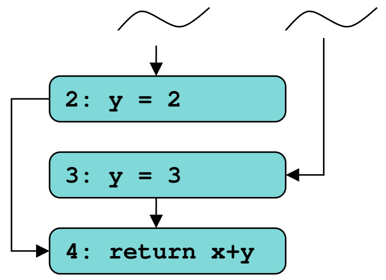
Dataflow Equations

Let $D = (V, E)$ be the control-flow graph of a method.

Then, the dataflow equations for a node $v \in V$ are:

$$\begin{aligned} DEF_{IN}(0) &= ARGS(0) \\ DEF_{IN}(v) &= \bigcap_{w \to v \in E} DEF_{OUT}(w) \\ DEF_{OUT}(v) &= DEF_{IN}(v) \cup DEF_{AT}(v) \end{aligned}$$

- Assume $ARGS(0)$ gives the arguments for the method, and node 0 has no predecessors.
- We compute $DEF_{IN}(v)$ by intersecting $DEF_{OUT}(w)$ for each predecessor w of v:

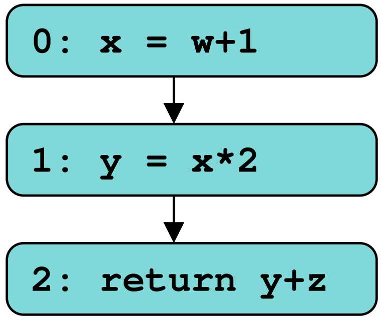


Here, $DEF_{IN}(4) = DEF_{OUT}(2) \cap DEF_{OUT}(3)$

Example

- Consider following method, and CFG:

```
int f(int w) {
 int x,y,z;
 x = w + 1;
 y = x * 2;
 return y + z;
}
```



- The flow sets are:

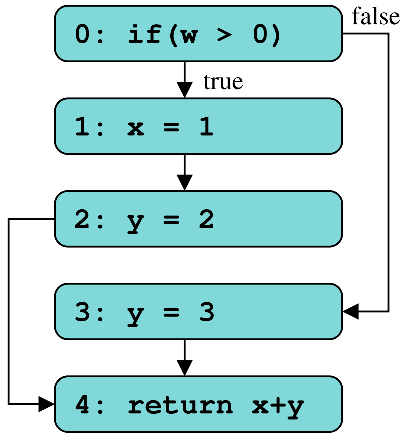
$DEF_{IN}(0) = \{w\}$	$DEF_{AT}(0) = \{x\}$	$USES(0) = \{w\}$	$DEF_{OUT}(0) = \{w, x\}$
$DEF_{IN}(1) = \{w, x\}$	$DEF_{AT}(1) = \{y\}$	$USES(1) = \{x\}$	$DEF_{OUT}(1) = \{w, x, y\}$
$DEF_{IN}(2) = \{w, x, y\}$	$DEF_{AT}(2) = \emptyset$	$USES(2) = \{y, z\}$	$DEF_{OUT}(2) = \{w, x, y\}$

- Note that $DEF_{IN}(0) = \{w\}$ since w is parameter
- Java gives an error because $USES(2) - DEF_{IN}(2) = \{z\}$
- In this example, $DEF_{IN}(i + 1) = DEF_{OUT}(i)$

Another Example

- Consider following method, and CFG:

```
int f(int w) {
 int x,y;
 if(w > 0) {
   x = 1; y = 2;
 } else { y = 3; }
 return x + y;
}
```



- The flow sets are:

$$
\begin{array}{llll}
DEF_{IN}(0) = \{w\} & DEF_{AT}(0) = \emptyset & USES(0) = \{w\} & DEF_{OUT}(0) = \{w\} \\
DEF_{IN}(1) = \{w\} & DEF_{AT}(1) = \{x\} & USES(1) = \emptyset & DEF_{OUT}(1) = \{w, x\} \\
DEF_{IN}(2) = \{w, x\} & DEF_{AT}(2) = \{y\} & USES(2) = \emptyset & DEF_{OUT}(2) = \{w, x, y\} \\
DEF_{IN}(3) = \{w\} & DEF_{AT}(3) = \{y\} & USES(3) = \emptyset & DEF_{OUT}(3) = \{w, y\} \\
DEF_{IN}(4) = \{w, y\} & DEF_{AT}(4) = \emptyset & USES(4) = \{x, y\} & DEF_{OUT}(4) = \{w, y\}
\end{array}
$$

What About Exceptions?

- Example 1:

```
Object f() {
  Object x;
  try { x = new FileInputStream("Hello World");
  } catch(Exception e) { }
  return x;
}
```

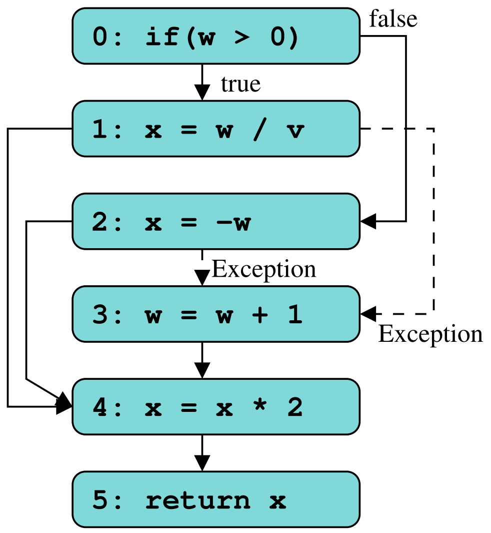
- Example 2:

```
int f(String x) {
  int y;
  try {
    if(x != null) { return x.length();}
    return 0;
  } catch(Exception e) { }
  return y;
}
```

Considering Exceptions

Add edges representing exceptions to the CFG. But their DEF_{OUT} does not add in things defined defined from before.

```
int f(int w, int v) {
 try {
  if(w > 0) { x = w / v; }
  else { x = -w; }
 } catch(Exception e) {
  w = w + 1;
 }
 x = x * 2;
 return x;
}
```



$$
\begin{aligned}
DEF_{IN}(0) &= \{w, v\} & DEF_{OUT}(0) &= DEF_{IN}(0)\\
DEF_{IN}(1) &= DEF_{OUT}(0) & DEF_{OUT}(1) &= DEF_{IN}(1) \cup \{x\}\\
DEF_{IN}(2) &= DEF_{OUT}(0) & DEF_{OUT}(2) &= DEF_{IN}(2) \cup \{x\}\\
DEF_{IN}(3) &= DEF_{IN}(1) \cap DEF_{IN}(2) & DEF_{OUT}(3) &= DEF_{IN}(3) \cup \{w\}\\
DEF_{IN}(4) &= DEF_{OUT}(1) \cap DEF_{OUT}(2) & DEF_{OUT}(4) &= DEF_{IN}(4) \cup \{x\}\\
&\quad \cap DEF_{OUT}(3) & &\\
DEF_{IN}(5) &= DEF_{OUT}(4) & DEF_{OUT}(5) &= DEF_{IN}(5)
\end{aligned}
$$

What is Definite Unassignment?

- JLS §16:
  Similarly, every blank final variable must be assigned at most once; it must be definitely unassigned when an assignment to it occurs.
  Such an assignment is defined to occur if and only if either the simple name of the variable (or, for a field, its simple name qualified by this) occurs on the left hand side of an assignment operator.
  For every assignment to a blank final variable, the variable must be definitely unassigned before the assignment, or a compile-time error occurs.

What is Definite Unassignment?

- JLS §16:
  “The definite unassignment analysis of loop statements raises a special problem. Consider the statement while (e) S. In order to determine whether V is definitely unassigned within some subexpression of e, we need to determine whether V is definitely unassigned before e. One might argue, by analogy with the rule for definite assignment (Â§16.2.10), that V is definitely unassigned before e iff it is definitely unassigned before the while statement. However, such a rule is inadequate for our purposes. If e evaluates to true, the statement S will be executed. Later, if V is assigned by S, then in the following iteration(s) V will have already been assigned when e is evaluated. Under the rule suggested above, it would be possible to assign V multiple times, which is exactly what we have sought to avoid by introducing these rules”

Examples

- Example 1:

```
void f(boolean flag) {
 final int k;
 if(flag) { k = 3; }
 if(!flag) { k = 4;}
}
```

- Example 2:

```
void f(boolean flag, int x) {
 final int k;
 if(flag || (k=x) == 0) { x = 0; }
 k = 1;
}
```

- Example 3:

```
class X {
 final int field;
 void f() { field = 0; }
}
```