SWEN430 - Compiler Engineering

Lecture 10: Definite Assignment/Unassignment

Erin Greenwood-Thessman

with thanks to Linsday Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Definite Assignment?

- Consider the following Java method:

```
int f(int x) {
 int y;

 if(x > 0) {
  return y;
 }

 return 1;
}
```

- Will this compile under `javac`?
- What value could it output when `x > 0`?

What is Definite Assignment? (cont'd)

- JLS §16:
  "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. An access to its value consists of the simple name of the variable occurring anywhere in an expression except as the left-hand operand of the simple assignment operator.
  A **Java compiler** *must carry out a specific conservative flow analysis to make sure that, 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."*

Simple Examples

- Example 1:

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

- Example 2:

```
class X {
 int field;
 int f() { return field; }
}
```

- Example 3:

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

More Interesting Examples

- Example 4:

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

- Example 5:

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

- Example 6:

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

Overview of Analysis

- Definite assignment algorithm is form of **dataflow analysis**, operating on a Control Flow Graph (CFG).
- We want to determine for each node in the CFG which variables have been assigned values, and which variables are accessed.
- To do this, we will keep track of:
  - variables defined on *entry* to the node
  - variables defined by statement at the node
  - variables used by statement at the node
  - variables defined on *exit* from the node

Overview of Analysis

Definite Assignment Flow Sets

Let $D = (V, E)$ be the control-flow graph of a method. Then, for each $v \in V$, we define:

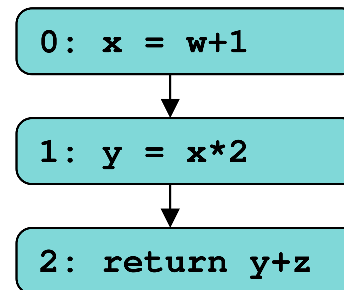
- $DEF_{IN}(v)$ — the set of variables defined on *entry* to v
- $DEF_{AT}(v)$ — the set of variables defined by statement at v
- $USES(v)$ — the set of variables used by statement at v
- $DEF_{OUT}(v)$ — the set of variables defined on *exit* from v

- Note that $DEF_{OUT}(v) = DEF_{IN}(v) \cup DEF_{AT}(v)$
- It is an error if $USES(v) - DEF_{IN}(v) \neq \emptyset$ for some $v \in V$.

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, $DEF_{IN}(i + 1) = DEF_{OUT}(i)$

Calculating $DEF_{AT}(v)$

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

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

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

Calculating $USES(v)$

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

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

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

Dataflow Equations

Dataflow Equations

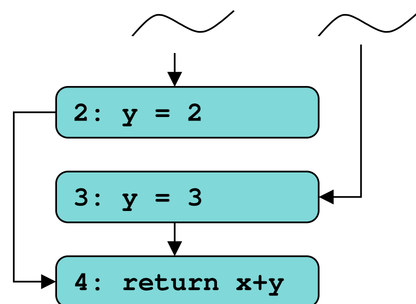
Let $D = (V, E)$ be the control-flow graph of a method. Then, the dataflow equations for a node $v \in V$ are:

$$DEF_{IN}(0) = ARGS(0)$$

$$DEF_{IN}(v) = \bigcap_{w \to v \in E} DEF_{OUT}(w)$$

$$DEF_{OUT}(v) = DEF_{IN}(v) \cup DEF_{AT}(v)$$

- 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:

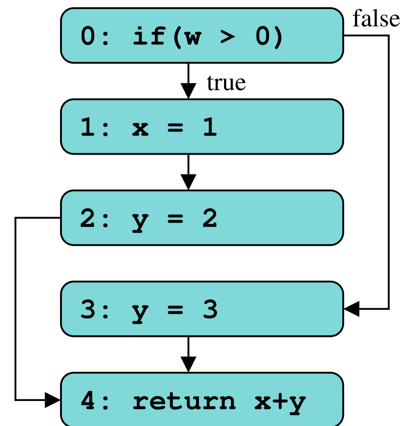


Here, $DEF_{IN}(4) = DEF_{OUT}(2) \cap DEF_{OUT}(3)$

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:

$$DEF_{IN}(0) = \{w\} \quad DEF_{AT}(0) = \emptyset \quad USES(0) = \{w\} \quad DEF_{OUT}(0) = \{w\}$$

$$DEF_{IN}(1) = \{w\} \quad DEF_{AT}(1) = \{x\} \quad USES(1) = \emptyset \quad DEF_{OUT}(1) = \{w, x\}$$

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

$$DEF_{IN}(3) = \{w\} \quad DEF_{AT}(3) = \{y\} \quad USES(3) = \emptyset \quad DEF_{OUT}(3) = \{w, y\}$$

$$DEF_{IN}(4) = \{w, y\} \quad DEF_{AT}(4) = \emptyset \quad USES(4) = \{x, y\} \quad DEF_{OUT}(4) = \{w, y\}$$

What About Exceptions?

- Example 1:

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

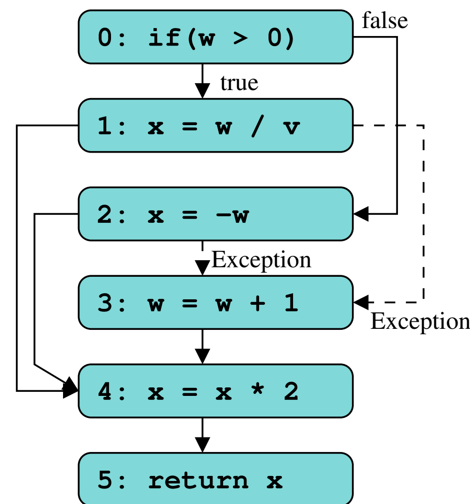
- Example 2:

```
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 (Âğ16.2.10), 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; }
}
```