CSCI 742 - Compiler Construction

Lecture 36
Range Analysis, Initialization Analysis
Instructor: Hossein Hojjat

April 23, 2018
Recap: Lattices

- Poset: set augmented with a partial order relation \( \preceq \)
- Lattice: poset where each two-element subset has a lub and a glb
- Can define: meet \( \sqcap \) and join \( \sqcup \)
- Use lattice to express information about a point in a program
- \( x \preceq y \) means “\( x \) is less or equally precise as \( y \)”
- To compute information: build constraints that describe how the lattice information changes
  - Effect of instructions: transfer functions
  - Effect of control flow: meet operation
Recap: Transfer Functions

• $L$: data-flow information lattice
• Transfer function $F_S : L \rightarrow L$ for each instruction $S$
• Describes how $S$ modifies the information in the lattice
• If $in(S')$ is info before $S$ and $out(S')$ is info after $S$ then
• Forward analysis: $out(S) = F(in(S))$
• Backward analysis: $in(S) = F(out(S))$
Recap: Control Flow

- Meet operation models how to combine information at split/join points in the control flow.
- Forward analysis:
  $$in(S) = \bigcap \{out(S') | S' \in pred(S)\}$$
- Backward analysis:
  $$out(S) = \bigcap \{in(S') | S' \in succ(S)\}$$
• Try to determine the possible range of integer values of a variable
• Elements: $[a, b]$ where $a \leq b$ or $\emptyset$
• We allow $a = -\infty$ and/or $b = \infty$
  • $(-\infty, +\infty)$ set of all integers
• $[a, b] \cup [a', b'] = [\min(a, a'), \max(b, b')]$
• Forward analysis with $\cup$ as the meet operator
Domain of Intervals $[a, b]$ where $a, b \in \{-\infty, 2, 3, 4, \infty\}$
Transfer Function: Addition

- Suppose we have only two integer variables: \(x, y\)

\[
\begin{align*}
\text{x : } [a, b] & \quad \text{y : } [c, d] \\
x &= x + y
\end{align*}
\]

- So we can let

\[
\begin{align*}
a' &= a + c \\
c' &= c \\
b' &= b + d \\
d' &= d
\end{align*}
\]
Suppose we have only two integer variables: $x$, $y$

- $x : [a, b]$  $y : [c, d]$
  
  \[ y = x - y \]

- $x : [a', b']$  $y : [c', d']$

So we can let

\[
\begin{align*}
a' &= a \\
c' &= a - d \\
b' &= b \\
d' &= b - c
\end{align*}
\]
Combine Data-flow Facts

\[
\begin{align*}
x & : [-10, 10] \quad y : [-1000, 1000] \\
\text{if} \ (x > 0) \ {\} \\
\quad y &= x + 100; \\
\text{else} \ {\} \\
\quad y &= -x - 50; \\
\end{align*}
\]
Handling Loops

Iterate until stabilizes

\begin{itemize}
  \item \(x = 1\);
  \item \textbf{while} \((x < 10)\) \{
    \item \(x = x + 2;\)
  \}
\end{itemize}
Exercise

• Run range analysis, prove error is unreachable

```java
int M = 16;
int a[] = new int[M];
int x = 0;
while (x < 10) {
    x = x + 3;
}
if (x >= 0) {
    if (x <= 15)
        a[x] = 7;
    else
        error;
} else {
    error;
}
```
Exercise

- Run range analysis, prove error is unreachable

```java
int M = 16;
int a[] = new int[M];
int x = 0;
while (x < 10) {
    x = x + 3;
}  // 
if (x >= 0) {
    if (x <= 15)
        a[x] = 7;
    else
        error;
} else {
    error;
}
```

- Benefits: faster execution (no checks)
- Program cannot crash with error
Exercise

- Run range analysis, prove error is unreachable

```java
int M = 16;
int a[] = new int[M];
int x = 0;
while (x < 10) {
    x = x + 3;
}
if (x >= 0) {
    if (x <= 15)
        a[x]=7;
    else
        error;
} else {
    error;
}
```

- Benefits: faster execution (no checks)
- Program cannot crash with error
Exercise

- Run range analysis, prove error is unreachable

```java
int M = 16;
int a[] = new int[M];
int x = 0;
while (x < 10) {
    x = x + 3;
}
if (x >= 0) {
    if (x <= 15)
        a[x]=7;
    else
        error;
} else {
    error;
}
```

- Benefits: faster execution (no checks)
- Program cannot crash with error
class Test {
    static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
            n = 100;
        }
        while (n != 0) {
            System.out.println(n);
            n = n - p;
        }
    }
}

• Does javac compile this program without error?
Initialization Analysis

```java
class Test {
    static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
            n = 100;
        }
        while (n != 0) {
            System.out.println(n);
            n = n - p;
        }
    }
}
```

- Does javac compile this program without error?

Test.java:8:error: variable n might not have been initialized while (n != 0) {
    ^
Initialization Analysis

• We would like variables to be initialized on all execution paths.
• Otherwise, the program execution could be undesirably affected by the value that was in the variable initially.
• We can enforce such check using initialization analysis.

```java
class Test {
    static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
            n = 100;
        } else {
            n = -100;
        }
        while (n != 0) {
            System.out.println(n);
            n = n - p;
        }
    }
}
```
Initialization Analysis

- Does javac compile this program without error?

```java
static void test(int p) {
    int n;
    p = p - 1;
    if (p > 0) {
        n = 100;
    }
    System.out.println("Hello!");
    if (p > 0) {
        while (n != 0) {
            System.out.println(n);
            n = n - p;
        }
    }
}
```
class Test {
    static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
            n = 100;
        } else {
            n = -100;
        }
        while (n != 0) {
            System.out.println(n);
            n = n - p;
        }
    }
}

• ⊥ indicates presence of flow from states where variable was not initialized
• If variable is possibly uninitialized, we use ⊥
• Otherwise (initialized, or unreachable): ⊤

If var occurs anywhere but LHS of an assignment and has value ⊥, report error
• Domain: for each variable, for each program point: \( D = \{ \bot, \top \} \)
• At program entry, local variables: \( \bot \), parameters: \( \top \)
• At other program points: each variable: \( \top \)
• An assignment \( x = e \) sets variable \( x \) to \( \top \)
• glb (\( \sqcap \)) of any value with \( \bot \) gives \( \bot \)
• Uninitialized values are contagious along paths
• \( \top \) value for \( x \) means there is definitely no possibility for accessing uninitialized value of \( x \)
Run initialization analysis

```c
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
while (n != 0) {
    n = n - p;
}
```