**Global Optimization**

Lecture 15

**Lecture Outline**

- Global flow analysis
- Global constant propagation
- Liveness analysis

**Local Optimization**

Recall the simple basic-block optimizations

- Constant propagation
- Dead code elimination

\[
\begin{align*}
X &:= 3 \\
Y &:= Z \times W \\
Q &:= X + Y
\end{align*}
\]

**Global Optimization**

These optimizations can be extended to an entire control-flow graph

\[
\begin{align*}
X &:= 3 \\
B &> 0 \\
Y &:= Z \times W \\
Q &:= X + Y
\end{align*}
\]
Correctness

- How do we know it is OK to globally propagate constants?
- There are situations where it is incorrect:

```
X := 3
B > 0
Y := Z + W
(X := 4)
A := 2 * X
Y := 0
```

Correctness (Cont.)

To replace a use of $x$ by a constant $k$ we must know that:

\[
\text{On every path to the use of } x, \text{ the last assignment to } x \text{ is } x := k \quad **
\]

Example 1 Revisited

```
X := 3
B > 0
Y := Z + W
Y := 0
A := 2 * X
```

Example 2 Revisited

```
X := 3
B > 0
Y := Z + W
X := 4
Y := 0
A := 2 * X
```

Discussion

- The correctness condition is not trivial to check
- “All paths” includes paths around loops and through branches of conditionals
- Checking the condition requires global analysis
  - An analysis of the entire control-flow graph

Global Analysis

Global optimization tasks share several traits:
- The optimization depends on knowing a property $X$ at a particular point in program execution
- Proving $X$ at any point requires knowledge of the entire program
- It is OK to be conservative. If the optimization requires $X$ to be true, then want to know either
  - $X$ is definitely true
  - Don’t know if $X$ is true
- It is always safe to say “don’t know”
Global Analysis (Cont.)

- Global dataflow analysis is a standard technique for solving problems with these characteristics.

- Global constant propagation is one example of an optimization that requires global dataflow analysis.

Global Constant Propagation

- Global constant propagation can be performed at any point where ** holds.

- Consider the case of computing ** for a single variable X at all program points.

Global Constant Propagation (Cont.)

- To make the problem precise, we associate one of the following values with X at every program point.

<table>
<thead>
<tr>
<th>value</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This statement never executes</td>
</tr>
<tr>
<td>C</td>
<td>X = constant c</td>
</tr>
<tr>
<td>⊥</td>
<td>X is not a constant</td>
</tr>
</tbody>
</table>

Example

Using the Information

- Given global constant information, it is easy to perform the optimization.
  - Simply inspect the \( x = ? \) associated with a statement using \( x \).
  - If \( x \) is constant at that point replace that use of \( x \) by the constant.

- But how do we compute the properties \( x = ? \)?

The Idea

The analysis of a complicated program can be expressed as a combination of simple rules relating the change in information between adjacent statements.
Explaination

- The idea is to “push” or “transfer” information from one statement to the next.
- For each statement $s$, we compute information about the value of $x$ immediately before and after $s$.
  
  $$
  C(s, x, in) = \text{value of } x \text{ before } s \\
  C(s, x, out) = \text{value of } x \text{ after } s
  $$

Transfer Functions

- Define a transfer function that transfers information one statement to another.
- In the following rules, let statement $s$ have immediate predecessor statements $p_1, ..., p_n$.

Rule 1

$$
if \ C(p_i, x, out) = \top \text{ for any } i, \ then \ C(s, x, in) = \top
$$

Rule 2

$$
C(p_i, x, out) = c \ & \ C(p_j, x, out) = d \ & \ d \neq c \ then \ C(s, x, in) = \top
$$

Rule 3

$$
if \ C(p_i, x, out) = c \ or \ \bot \ for \ all \ i, \ then \ C(s, x, in) = c
$$

Rule 4

$$
if \ C(p_i, x, out) = \bot \ for \ all \ i, \ then \ C(s, x, in) = \bot
$$
The Other Half

• Rules 1-4 relate the out of one statement to the in of the next statement.

• Now we need rules relating the in of a statement to the out of the same statement.

Rule 5

\[ C(s, x, \text{out}) = \perp \text{ if } C(s, x, \text{in}) = \perp \]

Rule 6

\[ C(x := c, x, \text{out}) = c \text{ if } c \text{ is a constant} \]

Rule 7

\[ C(x := f(\ldots), x, \text{out}) = \top \]

Rule 8

\[ C(y := \ldots, x, \text{out}) = C(y := \ldots, x, \text{in}) \text{ if } x \leftrightarrow y \]

An Algorithm

1. For every entry \( s \) to the program, set \( C(s, x, \text{in}) = \top \).

2. Set \( C(s, x, \text{in}) = C(s, x, \text{out}) = \perp \) everywhere else.

3. Repeat until all points satisfy 1-8:
   - Pick \( s \) not satisfying 1-8 and update using the appropriate rule.
The Value $\bot$

- To understand why we need $\bot$, look at a loop

Discussion

- Consider the statement $Y := 0$
- To compute whether $X$ is constant at this point, we need to know whether $X$ is constant at the two predecessors
  - $X := 3$
  - $A := 2 \times X$
- But info for $A := 2 \times X$ depends on its predecessors, including $Y := 0$

The Value $\bot$ (Cont.)

- Because of cycles, all points must have values at all times
- Intuitively, assigning some initial value allows the analysis to break cycles
- The initial value $\bot$ means “So far as we know, control never reaches this point”

Example
**Example**

- \( X := 3 \)
- \( B > 0 \)
- \( Y := Z + W \)
- \( Y := 0 \)
- \( A := 2 \times X \)
- \( A < B \)

**Orderings**

- We can simplify the presentation of the analysis by ordering the values
  \( \bot < c < T \)
- Drawing a picture with “lower” values drawn lower, we get

**Orderings (Cont.)**

- \( T \) is the greatest value, \( \bot \) is the least
  - All constants are in between and incomparable
- Let \( \text{lub} \) be the least-upper bound in this ordering
- Rules 1-4 can be written using \( \text{lub} \):
  \[
  C(s, x, \text{in}) = \text{lub} \{ C(p, x, \text{out}) \mid p \text{ is a predecessor of } s \}
  \]

**Termination**

- Simply saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes
- The use of \( \text{lub} \) explains why the algorithm terminates
  - Values start as \( \bot \) and only increase
    \( \bot \) can change to a constant, and a constant to \( T \)
  - Thus, \( C(s, x, _) \) can change at most twice

**Termination (Cont.)**

Thus the algorithm is linear in program size

Number of steps =  
Number of \( C(...) \) value computed * 2 =  
Number of program statements * 4

**Liveness Analysis**

Once constants have been globally propagated, we would like to eliminate dead code

After constant propagation, \( X := 3 \) is dead (assuming \( X \) not used elsewhere)
Live and Dead

• The first value of \( x \) is dead (never used)
  \[
  X := 3
  \]

• The second value of \( x \) is live (may be used)
  \[
  X := 4
  \]

• Liveness is an important concept
  \[
  Y := X
  \]

Liveness

A variable \( x \) is live at statement \( s \) if
- There exists a statement \( s' \) that uses \( x \)
- There is a path from \( s \) to \( s' \)
- That path has no intervening assignment to \( x \)

Global Dead Code Elimination

• A statement \( x := \ldots \) is dead code if \( x \) is dead after the assignment
• Dead statements can be deleted from the program
• But we need liveness information first . . .

Computing Liveness

• We can express liveness in terms of information transferred between adjacent statements, just as in copy propagation
• Liveness is simpler than constant propagation, since it is a boolean property (true or false)

Liveness Rule 1

\[
L(p, x, \text{out}) = \bigvee \{ L(s, x, \text{in}) \mid s \text{ a successor of } p \}
\]

Liveness Rule 2

\[
L(s, x, \text{in}) = \text{true} \text{ if } s \text{ refers to } x \text{ on the rhs}
\]
**Liveness Rule 3**

\[ L(x := e, x, \text{in}) = \text{false} \text{ if } e \text{ does not refer to } x \]

**Liveness Rule 4**

\[ L(s, x, \text{in}) = L(s, x, \text{out}) \text{ if } s \text{ does not refer to } x \]

**Algorithm**

1. Let all \( L(\ldots) = \text{false} \) initially
2. Repeat until all statements \( s \) satisfy rules 1-4
   - Pick \( s \) where one of 1-4 does not hold and update using the appropriate rule

**Termination**

- A value can change from \( \text{false} \) to \( \text{true} \), but not the other way around
- Each value can change only once, so termination is guaranteed
- Once the analysis is computed, it is simple to eliminate dead code

**Forward vs. Backward Analysis**

We've seen two kinds of analysis:
- Constant propagation is a *forwards* analysis: information is pushed from inputs to outputs
- Liveness is a *backwards* analysis: information is pushed from outputs back towards inputs

**Analysis**

- There are many other global flow analyses
- Most can be classified as either forward or backward
- Most also follow the methodology of local rules relating information between adjacent program points