Global Optimization

Lecture 22
Lecture Outline

- **Global flow analysis**
- **Global constant propagation**
- **Liveness analysis**
Local Optimization

Recall the simple basic-block optimizations
- Constant propagation
- Dead code elimination

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

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

\[
\begin{align*}
X &:= 3 \\
B &> 0 \\
Y &:= Z + W \\
A &:= 2 \times X \\
A &:= 2 \times X \\
Y &:= 0
\end{align*}
\]
Global Optimization

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

\[
\begin{align*}
X &:= 3 \\
B &> 0 \\
Y &:= Z + W \\
Y &:= 0 \\
A &:= 2 \times X
\end{align*}
\]
Global Optimization

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

\[ X := 3 \]

\[ B > 0 \]

\[ Y := Z + W \]

\[ Y := 0 \]

\[ A := 2 \times 3 \]
Correctness

- How do we know it is OK to globally propagate constants?
- There are situations where it is incorrect:
Correctness (Cont.)

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

On every path to the use of $x$, the last assignment to $x$ is $x := k$ **
Example 1 Revisited

X := 3
B > 0

Y := Z + W

A := 2 * X

Y := 0
Example 2 Revisited

- $X := 3$
- $B > 0$
- $Y := Z + W$
- $X := 4$
- $A := 2 \times X$
- $Y := 0$
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 function body
- 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>#</td>
<td>This statement never executes</td>
</tr>
<tr>
<td>c</td>
<td>$X = \text{constant c}$</td>
</tr>
<tr>
<td>*</td>
<td>Don’t know if $X$ is a constant</td>
</tr>
</tbody>
</table>
Example

\[ X := 3 \]
\[ B > 0 \]

\[ Y := Z + W \]
\[ X := 4 \]

\[ A := 2 \times X \]

\[ Y := 0 \]
\[ X = 3 \]

\[ X = 4 \]
\[ X = * \]
\[ X = * \]
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.
Explanation

• 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 \( s \)

\[
C_{\text{in}}(x,s) = \text{value of } x \text{ before } s \\
C_{\text{out}}(x,s) = \text{value of } x \text{ after } s
\]
Transfer Functions

• Define a transfer function that transfers information one statement to another

• In the following rules, let statement \( s \) have immediate predecessor statements \( p_1, \ldots, p_n \)
Rule 1 (unknown predecessor)

If $C_{out}(x, p_i) = *$ for any predecessor $i$,
then $C_{in}(x, s) = *$
Rule 2 (different predecessors)

If \( C_{\text{out}}(x, p_i) = c \) and \( C_{\text{out}}(x, p_j) = d \) and \( d \neq c \)
then \( C_{\text{in}}(x, s) = * \)
Rule 3 (exclude unreachable values)

\[ C_{\text{out}}(x, p_i) = c \text{ or } \# \text{ for all } i, \]
then \[ C_{\text{in}}(x, s) = c \]
Rule 4 (unreachable propagation)

\[
\text{if } C_{out}(x, p_i) = \# \text{ for all } i, \\
\text{then } C_{in}(x, s) = \#
\]
The Other Half

• Rules 1-4 relate the *out* of one or more statements to the *in* of the successor statement

• Now we need rules relating the *in* of a statement to the *out* of the same statement
Rule 5

\[ C_{out}(x, s) = \# \text{ if } C_{in}(x, s) = \# \]
Rule 6

\[ C_{out}(x, x := c) = c \text{ if } c \text{ is a constant} \]
Rule 7

\[ C_{\text{out}}(x, x := f(...)) = * \]
Rule 8

\[ C_{\text{out}}(x, y := \ldots) = C_{\text{in}}(x, y := \ldots) \text{ if } x \not\equiv y \]
An Algorithm

1. For every entry \( s \) to the program, set \( C_{in}(x, s) = * \)

2. Set \( C_{in}(x, s) = C_{out}(x, s) = # \) everywhere else

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

- To understand why we need #, 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 # (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 # means “So far as we know, control never reaches this point”
Example

\[
\begin{align*}
X &:= 3 \\
B &> 0 \\
Y &:= Z + W \\
Y &:= 0 \\
A &:= 2 \times X \\
A &< B
\end{align*}
\]
Example

X := 3
B > 0
Y := Z + W
A := 2 * X
A < B
Y := 0
X := *
X := 3
X := 3
X := 3
X := 3
X := #
X := #
Example

\[
X := 3 \\
B > 0 \\
Y := Z + W \\
A := 2 \times X \\
Y := 0 \\
A < B
\]
Example

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

\( X = * \)
\( X = 3 \)
\( X = 3 \)
\( X = 3 \)
\( X = 3 \)
\( X = 3 \)
Orderings

• We can simplify the presentation of the analysis by ordering the values
  $$\# < c < *$$

• Drawing a picture with “lower” values drawn lower, we get

```
*----*----*
|     |
*----*----*
  |     |
  |     |
-1-----0-----1
    |     |
    |     |
#     #     #
```
Orderings (Cont.)

• * is the greatest value, # is the least
  - All constants are in between and incomparable

• Let lub be the least-upper bound in this ordering

• Rules 1-4 can be written using lub:
  \[ C_{in}(x, s) = \text{lub} \{ C_{out}(x, p) \mid p \text{ is a predecessor of } s \} \]
Termination

• In general, simply saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes.

• The use of lub explains why the algorithm terminates
  - Values start as # and only increase
  - # can change to a constant, and a constant to *
  - Thus, $C(x, s)$ can change at most twice
Termination (Cont.)

Thus the algorithm is linear in program size

Number of steps =
Number of C_(....) values 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 \textit{dead} (never used)

• The second value of $x$ is \textit{live} (may be used)

• Liveness is an important concept
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

\[ L_{out}(x, p) = \bigvee \{ L_{in}(x, s) \mid s \text{ a successor of } p \} \]
Liveness Rule 2

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

\[ L_{in}(x, x := e) = \text{false} \quad \text{if e does not refer to x} \]
Liveness Rule 4

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

1. Let all $L(...) = \text{false}$ 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 \textit{false} to \textit{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
Other parts of optimization

• Register allocation, graph coloring
• Instruction scheduling on pipelined machines
  - Removing bubbles from the pipeline
  - Unrolling loops to fill bubbles