Global Optimization

Lecture Outline

• Global flow analysis
• Global constant propagation
• Liveness analysis

Local Optimization

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

```
x := 42
y := z * w
q := y + x
```

Global Optimization

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

```
x := 42
b > 0
y := z * w
q := y + x
```

```
x := 42
b > 0
y := z * w
q := y + x
y := 0
```

```
x := 42
b > 0
y := z * w
q := y + x
y := 0
```
Global Optimization

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

\[
x := 42 \\
b > 0 \\
y := z \times w \\
y := 0 \\
q := y + x
\]

Correctness

• How do we know whether it is OK to globally propagate constants?

• There are situations where it is incorrect:

\[
x := 42 \\
b > 0 \\
x := 54 \\
y := z \times w \\
y := 0 \\
q := y + x
\]

Correctness (Cont.)

To replace a use of \( x \) by a constant \( k \) we must know that the following property ** holds:

\( \text{On every path to the use of } x, \text{ the last assignment to } x \text{ is } x := k \) **
Example 1 Revisited

\[
\begin{align*}
x &:= 42 \\
b &> 0 \\
y &:= z \times w \\
q &:= y + x
\end{align*}
\]

Example 2 Revisited

\[
\begin{align*}
x &:= 42 \\
b &> 0 \\
y &:= z \times w \\
x &:= 54 \\
y &:= 0 \\
q &:= y + x
\end{align*}
\]

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 that determines how data flows over the entire control-flow graph of a function/method

Global Analysis

Global optimization tasks share several traits:
- The optimization depends on knowing a property \( P \) at a particular point in program execution
- Proving \( P \) at any point requires knowledge of the entire function body
- Property \( P \) is typically undecidable!
- It is OK to be conservative: If the optimization requires \( P \) to be true, then want to know either
  - that \( P \) is definitely true, or
  - that we don’t know whether \( P \) is true
- It is always safe to say “don’t know”
  - We try to say do not know as rarely as possible
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

• On every path to the use of $x$, the last assignment to $x$ is $x := k$  

• Global constant propagation can be performed at any point where property ** 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 whether $x$ is 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 Analysis 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 \( s \)
  \[
  C_{in}(x,s) = \text{value of } x \text{ before } s \\
  C_{out}(x,s) = \text{value of } x \text{ after } s
  \]

Transfer Functions

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

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

if $C_{out}(x, p_i) = *$ for any $i$, then $C_{in}(x, s) = *$

Rule 2

If $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$
then $C_{in}(x, s) = *$

Rule 3

if $C_{out}(x, p_i) = c$ or $#$ for all $i$,
then $C_{in}(x, s) = c$

Rule 4

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

- Rules 1-4 relate the \textit{out} of one statement to the \textit{in} of the successor statement
  - they propagate information forward across CFG edges

- We also need rules relating the \textit{in} of a statement to the \textit{out} of the same statement
  - to propagate information across statements

\begin{align*}
\text{Rule 5} \\
C_{\text{out}}(x, s) = \# \text{ if } C_{\text{in}}(x, s) = \#
\end{align*}

\begin{align*}
\text{Rule 6} \\
C_{\text{out}}(x, x := c) = c \text{ if } c \text{ is a constant}
\end{align*}

\begin{align*}
\text{Rule 7} \\
x := f(...) \\
C_{\text{out}}(x, x := f(...)) = * & \text{ where } f \text{ is a function other than the one being analyzed}
\end{align*}

This rule says that we do not perform inter-procedural analysis (i.e. we do not look at what other functions do)
Rule 8

\[ \text{Cout}(x, y := ...) = \text{Cin}(x, y := ...) \text{ if } x \neq y \]

An Algorithm

1. For every entry \( s \) to the function, set \( \text{Cin}(x, s) = * \)
2. Set \( \text{Cin}(x, s) = \text{Cout}(x, s) = # \) 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:

- 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:
  - \( x := 42 \)
  - \( q := y + x \)
- But information for \( q := y + x \) 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

\[ x := 42 \]
\[ b > 0 \]
\[ y := z \times w \]
\[ y := 0 \]
\[ q := x + y \]
\[ q < b \]
\[ x = * \]
\[ x = 42 \]
\[ x = # \]
\[ x = 42 \]
\[ x = # \]
\[ x = 42 \]
Example

Orderings

- We can simplify the presentation of the analysis by ordering the values
  \[ \# < c < * \]

- Drawing a picture with "lower" values drawn lower, we get

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

- Simply saying "repeat until nothing changes" doesn't guarantee that eventually we reach a point where 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 = // worst case
Number of $C_\text{(....)}$ 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, $x := 42$ is dead (assuming $x$ is not used elsewhere)

Live and Dead Variables

- The first value of $x$ is dead (never used)
- The second value of $x$ is live (may be used)
- Liveness is an important concept for the compiler

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

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

Liveness Rule 2

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

\[ L_{\text{in}}(x, x := e) = \begin{cases} \text{false} & \text{if } e \text{ does not refer to } x \\ x = ? & \text{if } e \text{ refers to } x \end{cases} \]

Liveness Rule 4

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

Algorithm

1. Let all \( L_{\text{...}} = \text{false} \) 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 information is computed, it is simple to eliminate dead code
# Forward vs. Backward Analysis

We have seen two kinds of analysis:

- An analysis that enables constant propagation:
  - this is a *forwards* analysis: information is pushed from inputs to outputs

- An analysis that calculates variable liveness:
  - this is a *backwards* analysis: information is pushed from outputs back towards inputs

---

# Global Flow Analyses

- 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