**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.

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

**Global Optimization**

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

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

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

5

Correctness

• How do we know whether 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 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

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

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.”
  (But goal is to try to say “don’t 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>$c_i$</td>
<td>$x = \text{constant } c_i$</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 $s$
  
  \[ C_{in}(x,s) = \text{value of } x \text{ before } s \]
  \[ C_{out}(x,s) = \text{value of } x \text{ after } s \]
**Rule 1**

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

**Rule 2**

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**

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

**Rule 4**

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

- Rules 1-4 relate the *out* of one statement to the *in* of the successor statement.
  - They propagate information **forward** across CFG edges.

- We also need rules relating the *in* of a statement to the *out* of the same statement.
  - To propagate information across statements.

**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**

where \( f \) is a function other than the one being analyzed

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

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 an \( s \) 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
  - \( x := 42 \)
  - \( q := y + x \)
- 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

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

37

**Orderings**

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

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

38

**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:
  
  \[ C_{in}(x, s) = lub \{ C_{out}(x, p) \mid p \text{ is a predecessor of } s \} \]

39

**Termination**

- Simply saying “repeat until nothing changes” does not 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, \( C_{in}(x, s) \) can change at most twice.
Termination (Cont.)

Thus, the algorithm is linear in program size.

Number of steps = // worst case
Number of \( C_{(\ldots)} \) 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 \text{dead} (never used).
- The second value of \( x \) is \text{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 constant propagation.

• Liveness is simpler than constant propagation, since it is a boolean property (true or false).

Liveness Rule 1

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

Liveness Rule 2

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

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

**Liveness Rule 4**

\[ \text{Lin}(x, s) = \text{Lout}(x, s) \]  if \( s \) does not refer to \( x \)

**Algorithm**

1. Let all \( \text{L}(\ldots) = \text{false} \) initially

2. Repeat until all statements \( s \) satisfy rules 1-4
   - pick an \( s \) 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.