One-Slide Summary

- A global optimization changes an entire method (consisting of multiple basic blocks).
- We must be conservative and only apply global optimizations when they preserve the semantics.
- We use global flow analyses to determine if it is OK to apply an optimization.
- Flow analyses are built out of simple transfer functions and can work forwards or backwards.

Lecture Outline

- Global flow analysis
- Global constant propagation
- Liveness analysis
Local Optimization
Recall the simple basic-block optimizations
- Constant propagation
- Dead code elimination

\[ X := 3 \]
\[ Y := Z \times W \]
\[ Q := X + Y \]

\[ X := 3 \]
\[ Y := Z \times W \]
\[ Q := 3 + Y \]

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

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

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

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

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

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 this correctness condition:

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

Example 2 Revisited

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
  - Global = an analysis of the entire control-flow graph for one method body
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 method body
- Property $P$ is typically undecidable!

Undecidability of Program Properties

- **Rice’s Theorem**: Most interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the halting problem
  - Is the result of a function $F$ always positive?
    - Assume we can answer this question precisely
      - Take function $H$ and find out if it halts by testing function $F(x)$ where $H(x); \text{return 1;}$ whether it has positive result
  - Syntactic properties are decidable!
    - e.g., How many occurrences of “x” are there?
  - Programs without looping are also decidable!

Conservative Program Analyses

- So, we cannot tell for sure that “x” is always 3
  - Then, how can we apply constant propagation?
- It is OK to be **conservative**. If the optimization requires $P$ to be true, then want to know either
  - $P$ is definitely true
  - Don’t know if $P$ is true
- It is always correct to say “don’t know”
  - We try to say don’t know as rarely as possible
- All program analyses are conservative
• 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 $x := k$*
Review

- The correctness condition is not trivial to check
- Checking the condition requires global analysis
  - An analysis of the entire control-flow graph for one method body

Global Analysis

- **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 \( \star \) holds
- Consider the case of computing \( \star \) 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 is not reachable</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 * X
```

It's rarely this easy to find key locations ...
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 $s$
  
  $C_{in}(x,s) =$ value of $x$ before $s$
  
  $C_{out}(x,s) =$ value of $x$ after $s$
Transfer Functions

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

Rule 1

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

Rule 2

\[ C_{\text{out}}(x, x := c) = c \text{ if } c \text{ is a constant} \]
Rule 3

\[ \text{Rule 3} \]

\[ \begin{align*}
  \text{C}_{\text{out}}(x, x := f(...)) &= \ast \\
  x := f(...) &\Rightarrow x \ast
\end{align*} \]

Rule 4

\[ \text{Rule 4} \]

\[ \begin{align*}
  \text{C}_{\text{out}}(x, y := ...) &= \text{C}_{\text{in}}(x, y := ...) \quad \text{if } x \neq y \\
  y := ... &\Rightarrow x \ast a \\
  \end{align*} \]

The Other Half

- Rules 1-4 relate the in of a statement to the out of the same statement.
  - They propagate information across statements.

- Now we need rules relating the out of one statement to the in of the successor statement.
  - To propagate information forward across CFG edges.

- In the following rules, let statement \( s \) have immediate predecessor statements \( p_1, ..., p_n \).
Rule 5

if $C_{out}(x, p_i) = \ast$ for some $i$, then $C_{in}(x, s) = \ast$

Rule 6

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

Rule 7

if $C_{out}(x, p_i) = c$ or $\#$ for all $i$,
then $C_{in}(x, s) = c$
Rule 8

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

An Algorithm

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

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
The Value 

- To understand why we need #, look at a loop

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

The Value # (Cont.)

- Because of cycles, all points must have values at all times during the analysis
- 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”

Sometimes all paths lead to the same place.
Thus you need #.
Example

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

We are done when all rules are satisfied!

Another Example

\[ X := 3 \]
\[ B > 0 \]
\[ Y := Z + W \]
\[ Y := 0 \]
\[ A := 2 \times X \]
\[ X := 4 \]
\[ A < B \]

Another Example

\[ X := 3 \]
\[ B > 0 \]
\[ Y := Z + W \]
\[ Y := 0 \]
\[ A := 2 \times X \]
\[ X := 4 \]
\[ A < B \]

Must continue until all rules are satisfied!
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 \textit{lub} be the least-upper bound in this ordering

- Rules 5-8 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 nothing changes

- The use of lub explains why the algorithm \textit{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 = 
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

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

After constant propagation, \( X := 3 \) is dead ?
(assuming this is the entire CFG)

Live and Dead

- The first value of \( x \) is **dead** (never used)
- The second value of \( x \) is **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 constant propagation
- Liveness is simpler than constant propagation, since it is a boolean property (true or false)
Liveness Rule 1

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

Liveness Rule 2

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

Liveness Rule 3

\[ L_{in}(x, s) = L_{out}(x, s) \text{ if } s \text{ does not refer to } x \]
Liveness Rule 4

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

Algorithm

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

2. Repeat process until all statements \( s \) satisfy rules 1-4:
   - Pick \( s \) where one of 1-4 does not hold and update using the appropriate rule

Another Example

- \( X := 3 \)
- \( B \geq 0 \)
- \( Y := Z + W \)
- also dead code?
- \( X := X \times X \)
- \( X := 4 \)
- \( A \times B \)
Termination

- A value can change from false to 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 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
Homework

• PA4 due this Friday March 30th (tomorrow)
• For Tuesday - Read Chapter 7.7
  - Optional David Bacon article
• Midterm 2 - Thursday April 12 (15 days)