Status

• Project 4 due June 5, 11:55pm

• This Week:
  • Office Hours Wednesday at 4pm and Friday at 11am
  • Friday: Register Allocation

• Next Week:
  • Monday: Automatic Memory Management
  • Wednesday: Static Analysis for Security
  • Friday: Final Exam Review
Global Optimization
Diagrams
Correctness Diagrams
Correctness

• 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 & 2 Diagram
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 method 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>
Constant Propagation Diagram
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 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 one statement to another

• In the following rules, let statement save immediate predecessor statements $p_1, \ldots, p_n$
Rules

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

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

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

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

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

• they propagate information forward across a CFG edge

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

• to propagate information across statements
Rules

5. $C_{out}(x, s) = \# \text{ if } C_{in}(x, s) = \#

6. $C_{out}(x, x := c) = c \text{ if } c \text{ is a constant}

7. $C_{out}(x, x := (f...)) = *$

8. $C_{out}(x, y := ...) = C_{in}(x, y := ...) \text{ if } x \neq y$
More Rule Diagrams
An Algorithm

• For every entry $s$ to the program, set $C_{in}(x, s) = *$

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

• Repeat until all points satisfy 1-8:
  • Pick $s$ not satisfying 1-8 and update using the appropriate rule
Value Diagram

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 $Y := 0$!
Value #

• 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 Diagrams
Orderings

- We can simplify the presentation of the analysis by ordering the values

  \[ # \ < \ c \ < \ * \]

- (example)
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 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 $C_{\ldots}$ values computed * 2 =
  - Number of program statements * 4
Liveness Diagrams
Liveness Rules

- \( L_{\text{out}}(x, p) = \bigvee \{ L_{\text{in}}(x, s) \mid s \text{ a successor of } p \} \)
- \( L_{\text{in}}(x, s) = \text{true} \) if \( s \) refers to \( x \) on the rhs
- \( L_{\text{in}}(x, x := e) = \text{false} \) if \( e \) does not refer to \( x \)
- \( L_{\text{in}}(x, s) = L_{\text{out}}(x, s) \) if \( s \) does not refer to \( x \)
Liveness

- The 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 Diagrams
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 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
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
Work on Project 4