Program Optimizations using Data-Flow Analysis

Last time
– Lattice theoretic framework for data-flow analysis

Today
– Dead-code elimination
– Common sub-expression elimination (CSE)
– Copy propagation
– Constant propagation

Dead Code Elimination

Remove statements that define only one variable and the variable being defined is not in the live out set.

Algorithm
1) generate a FlowGraph from the list of instructions
   do 
   2) perform liveness on FlowGraph
   3) for each node in FlowGraph
      if the defs set contains only one temporary
         if the temporary being defined is not in the live out set
            remove the node from the FlowGraph
   } while (changes);
4) generate a list of instructions from the modified FlowGraph
Common Subexpression Elimination

Idea
– Find common subexpressions whose range spans the same basic blocks and eliminate unnecessary re-evaluations
– Leverage available expressions

Recall available expressions
– An expression (e.g., \(x+y\)) is available at node \(n\) if every path from the entry node to \(n\) evaluates \(x+y\), and there are no definitions of \(x\) or \(y\) after the last evaluation along that path

Strategy
– If an expression is available at a point where it is evaluated, it need not be recomputed

CSE Example

<table>
<thead>
<tr>
<th>Before CSE</th>
<th>After CSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c := a + b)</td>
<td>(c := a + b)</td>
</tr>
<tr>
<td>(d := m &amp; n)</td>
<td>(t1 := c)</td>
</tr>
<tr>
<td>(e := b + d)</td>
<td>(d := m &amp; n)</td>
</tr>
<tr>
<td>(f := a + b)</td>
<td>(t2 := d)</td>
</tr>
<tr>
<td>(g := -b)</td>
<td>(e := b + d)</td>
</tr>
<tr>
<td>(h := b + a)</td>
<td>(t3 := e)</td>
</tr>
<tr>
<td>(a := j + a)</td>
<td>(f := t1)</td>
</tr>
<tr>
<td>(k := m &amp; n)</td>
<td>(g := -b)</td>
</tr>
<tr>
<td>(j := b + d)</td>
<td>(h := t1)</td>
</tr>
<tr>
<td>(a := -b)</td>
<td>(a := j + a)</td>
</tr>
<tr>
<td>if (m &amp; n) goto L2</td>
<td>(k := t2)</td>
</tr>
<tr>
<td></td>
<td>(j := t3)</td>
</tr>
<tr>
<td></td>
<td>(a := -b)</td>
</tr>
</tbody>
</table>

Summary
11 instructions
12 variables
9 binary operators

<table>
<thead>
<tr>
<th>After CSE</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c := a + b)</td>
<td>14 instructions</td>
</tr>
<tr>
<td>(t1 := c)</td>
<td>15 variables</td>
</tr>
<tr>
<td>(d := m &amp; n)</td>
<td>4 binary operators</td>
</tr>
<tr>
<td>(t2 := d)</td>
<td></td>
</tr>
<tr>
<td>(e := b + d)</td>
<td></td>
</tr>
<tr>
<td>(t3 := e)</td>
<td></td>
</tr>
<tr>
<td>(f := t1)</td>
<td></td>
</tr>
<tr>
<td>(g := -b)</td>
<td></td>
</tr>
<tr>
<td>(h := t1)</td>
<td></td>
</tr>
<tr>
<td>(a := j + a)</td>
<td></td>
</tr>
<tr>
<td>(k := t2)</td>
<td></td>
</tr>
<tr>
<td>(j := t3)</td>
<td></td>
</tr>
<tr>
<td>(a := -b)</td>
<td></td>
</tr>
<tr>
<td>if (t2) goto L2</td>
<td></td>
</tr>
</tbody>
</table>
CSE Approach 1

Notation
– IN_avail(s) is the set of expressions available at statement s
– GEN(s) is the set of expressions generated and not killed at statement s

If we use e and e ∈ IN_avail(s)
– Allocate a new name n
– Search backward from s (in CFG) to find statements (one for each path) that most recently generate e (e ∈ GEN)
– Insert copy to n after generators
– Replace e with n

Example

```
a := b + c
t2 := a
t1 := a
e := b1 + c
f := b2 + c
```

Problems
– Backward search for each use is expensive
– Generates unique name for each use
– |Uses| > |Avail|
– Each generator may have many copies

CSE Example

```
s1: a = 3
ns2: b = a + 2
ns3: c = fread()
s4: c = c + 1
ns5: if (c > a)   
    s5a: c = c + 1
    s5b: r = a * b
ns6: c = c + 1
ns7: r = a * b
```
CSE Approach 2

Idea
– Reduce number of copies by assigning a unique name to each unique expression

Summary
– \( \forall e \) Name\([e]\) = unassigned
– if we use \( e \) and \( e \in \text{Avail}(b) \)
  – if Name\([e]\) = unassigned, allocate new name \( n \) and Name\([e]\) = \( n \)
  else \( n = \text{Name}[e] \)
  – Replace \( e \) with \( n \)
– In a subsequent traversal of statement \( s \), if \( e \in \text{Gen}(s) \) and Name\([e]\) ≠ unassigned, then insert a copy to \( \text{Name}[e] \) after the generator of \( e \)

Problem
– May still insert unnecessary copies
– Requires two passes over the code

Example

\[
\begin{align*}
\text{s1: } a & := 3 \\
\text{s2: } b & := a + 2 \\
\text{s3: } c & := \text{read}() \\
\text{s4: } c & := c + 1 \\
\text{s5: } \text{if } (c > a) \\
\text{s6: } c & := c + 1 \\
\text{s7: } r & := a \times b
\end{align*}
\]
CSE Approach 3

Idea
– Don’t worry about temporaries
– Create one temporary for each unique expression
– Let subsequent pass eliminate unnecessary temporaries

At an evaluation/generation of e
– Hash e to a name, n, in a table
– Insert an assignment of e to n

At a use of e in b, if e ∈ Avail(b)
– Lookup e’s name in the hash table (call this name n)
– Replace e with n

Problems
– Inserts more copies than approach 2 (but extra copies are dead)
– Still requires two passes (2nd pass is very general)

CSE Example
Extraneous Copies

Extraneous copies degrade performance

Let other transformations deal with them
– Copy propagation
– Dead code elimination
– Coalescing

Coalesce assignments to \( t_1 \) and \( t_2 \) into a single statement

\[
\begin{align*}
    t_1 & := b + c \\
    t_2 & := t_1
\end{align*}
\]

– Greatly simplifies CSE

Copy Propagation

Propagate the results of a move statement.

Data-Flow Equations for reaching copies analysis

\[
\begin{align*}
    \text{in}[n] & = \bigcap \text{out}[p] \\
    \text{out}[n] & = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \\
    \text{gen}[n] & = (\text{target}, \text{source}) \text{ tuple if } n \text{ contains a move statement} \\
    \text{kill}[n] & = \text{all (target, source) tuples where either one is defined in } n
\end{align*}
\]

Algorithm

1) generate a FlowGraph from the list of instructions
   do {
   2) perform reaching copies analysis
   3) for each node in FlowGraph
       for each use
           if the use is reached by a copy where it is the target
               change the use to the source in the move statement tuple
   } while (changes);
4) generate a list of instructions from the modified FlowGraph
Constant Propagation using Reaching Definitions

Propagate the results of a CONSTANT move statement.

**Algorithm**

1) generate a FlowGraph from the list of instructions
do {
    2) perform reaching definitions
    3) for each node in FlowGraph
        for each use
            if the use is reached by only constant defs with same constant
                change the use to the rhs in the move statement
} while (changes);
4) generate a list of instructions from the modified FlowGraph

Constant Propagation

**Goal**

– Discover constant variables and expressions and propagate them forward through the program

**Uses**

– Evaluate expressions at compile time instead of run time
– Eliminate dead code (*e.g.*, debugging code)
– Improve efficacy of other optimizations (*e.g.*, value numbering and software pipelining)
Roadmap

Kinds of Constants

**Simple constants** Kildall [1973]
- Constant for all paths through a program

**Conditional constants** Wegbreit [1975]
- Constant for actual paths through a program (when only one direction of a conditional is taken)
### Data-Flow Analysis for Simple Constant Propagation

**Simple constant propagation: analysis is “reaching constants”**

- **D:** \(2^{\mathbb{N} \times \mathbb{C}}\)
- **\(\cap\):** \(\cap\)
- **F:**
  - \(\text{Kill}(x \leftarrow \ldots) = \{(x, c) \, \forall c\}\)
  - \(\text{Gen}(x \leftarrow c) = \{ (x, c) \}\)
  - \(\text{Gen}(x \leftarrow \gamma \oplus z) = \text{if } (y, c_y) \in \text{In} \, \& \, (z, c_z) \in \text{In}, \{ (x, c_y \oplus c_z) \}\)
  - \ldots

---

### Data-Flow Analysis for Simple Constant Propagation (cont)

**Reaching constants for simple constant propagation**

- **D:** \{All constants\} \(\cup\) \{ \top, \bot \}
- **\(\cap\):**
  - \(c \cap \top = c\)
  - \(c \cap \bot = \bot\)
  - \(c \cap d = \bot \text{ if } c \neq d\)
  - \(c \cap d = c \text{ if } c = d\)
- **F:**
  - \(F_{x \leftarrow c}(\text{In}) = c\)
  - \(F_{x \leftarrow \gamma \oplus z}(\text{In}) = \text{if } c_y = \text{In}_y \, \& \, c_z = \text{In}_z, \text{ then } c_y \oplus c_z, \text{ else } \bot\)
  - \ldots

**Using tuples of lattices**

![Using tuples of lattices diagram](attachment:image.png)
Initialization for Reaching Constants

**Pessimistic**
- Each variable is initially set to $\bot$ in data-flow analysis
- Forces merges at loop headers to go to $\bot$ conservatively

**Optimistic**
- Each variable is initially set to $\top$ in data-flow analysis
- Each variable is set to $\bot$ at the entry node in the flow graph.

Why initialize to bottom upon entry?

Change $s_6$ to $c = 3$, remove $s_4$, and try it both ways

```
s1: a = 3
\downarrow
s2: b = a + 2
\downarrow
s3: c = \text{read}()
\downarrow
s4: c = c + 1
\downarrow
s5: \text{if } (c > a)
\downarrow
s6: c = c + 1
\downarrow
s7: r = a * b
```
Next Time

Reading
– Ch 9.5

Lecture
– Partial redundancy elimination (PRE)

Dead code elimination with the MiniJava compiler

1) Method in FlowGraph for removing a node that attaches predecessors with successors. (implement rmFlowGraphNode())

2) Method to trace through a FlowGraph to generate an Assem.Instr list. (implement getInstrList())

4) DataFlowSet interface (has be provided)

5) DataFlowProblem interface (has be provided)

6) DataFlowSolver class where the constructor takes a DataFlowProblem as input and solves it with IDFA. (has be provided)

7) Liveness using the data-flow framework (an implementation for LiveDataFlowSet has be provided)
CSE with the MiniJava compiler

- Subclass Assem.Instr with a BINOPMOVE instruction type that can be queried for the operator type, each operand, and the temporary being defined.

- The AvailExpr class should implement the DataFlowProblem interface and use the DataFlowSolver.

- The CSE class should take a reference to a list of Assem.Instrs as input and generate a new list as output.

- CSE will generate new Assem.MOVE instructions and Assem.BINOPMOVE instructions. The CodeGen class must be able to generate code for them.

Copy propagation with the MiniJava compiler

- The ReachCopies class should implement the DataFlowProblem interface.

- Implement a ReachCopiesDataFlowSet class with (target, source) tuples.

- Implement a CopyProp class that takes a list of Assem.Instr and ReachCopies results as input and returns a resulting instruction list.
Constant Propagation with the MiniJava compiler

Assume the DataFlowSolver has already been written.

- Subclass Assem.Instr with a CONSTMOVE instruction type.

- The ReachDefs class should implement the DataFlowProblem interface.

- Implement a ReachDefsDataFlowSet class with reaching def statements paired with the temporary they define.

- Implement a ConstPropReachDef class that takes a list of Assem.Instr as input and has a public member variable for the resulting instruction list.

Simple Constants with the MiniJava compiler

- Subclass Assem.Instr with a CONSTMOVE instruction type.

- Subclass Assem.Instr with a BINOPMOVE instruction type that can be queried for the operator type, each operand, and the temporary being defined.

- The SimpleConstants class should implement the DataFlowProblem interface.
  - In the transfer function, BINOPMOVE instructions may be replaced with CONSTMOVE instructions.
  - Can we replace operands to a BINOPMOVE instruction with a constant?
  - The Frame class must be able to generate code for CONSTMOVE Assem.Instrs.

- The SimpleConstants class should take a reference to a list of Assem.Instrs as input and return a modified list as output.