**Advanced Compilers**
CMPS 710
Spring 2003
*Using SSA form*

Emery Berger
University of Massachusetts, Amherst

---

**Topics**
- Last time
  - Computing SSA form
- This time
  - Optimizations using SSA form
    - Constant propagation & dead code elimination
    - Loop invariant code motion

---

**Constant Propagation**
- Lattice for integer addition, multiplication, mod, etc.

```
> |
false 0 1 true
```
- note: false is *bottom*.

---

**Boolean Lattices, AND**
- meet functions
  - example: true AND ?, false AND >

```
<table>
<thead>
<tr>
<th>AND</th>
<th>false</th>
<th>true</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>⊥</td>
<td>true</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
```

---

**Boolean Lattices, OR**
- meet functions
  - example: true OR ?, false OR >

```
<table>
<thead>
<tr>
<th>OR</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>⊥</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
```

---

**Lattice for Φ-Nodes**
- To propagate constants:
  - if constant appears in conditional
  - Insert assignment on true branch

```
<table>
<thead>
<tr>
<th>Φ</th>
<th>T</th>
<th>c₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>c₁</td>
</tr>
<tr>
<td>c₁</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>c₂ ≠ c₁</td>
<td>c₁</td>
<td>⊥</td>
</tr>
<tr>
<td>c₂</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
```
Constant Propagation Using SSA Form

- Initialize all expressions to >
- Two work lists:
  - CFG edges = (entry, x)
  - SSA edges = ∅
- Pick edge from either list until empty
- Propagate constants when visiting either edge
- When we visit a node:
  - φ-node: meet lattice values
  - others: add SSA successors, CFG successors

Sparse Conditional Constant Propagation Example

Loops = frequently-accessed code
- regular patterns – can simplify optimizations
- rule of thumb: loop bodies execute 10<sup>depth</sup> times
- optimizations pay off

But why do we care if we aren't using FORTRAN?
- Loops aren't just over arrays!
- Pointer-based data structures
- Text processing...

Loop Invariant Code Motion

Removing Loop Invariant Code

- Build SSA graph
- Simple test:
  - no operands in statement are defined by φ node
  - or have definition inside loop
  - if match:
    - assign computation new temporary name and move to loop pre-header, and add assignment to temp
    - e.g., l = r<sub>i</sub> op r<sub>j</sub> becomes t<sub>i</sub> = r<sub>i</sub> op r<sub>j</sub>; l = t<sub>i</sub>
Finding More Invariants

- Build SSA graph
- If operands point to definition inside loop and definition is function of invariants (recursively)
  > replace as before

Loop Invariant Code Motion
Example II

- Build SSA graph
- If operands point to definition inside loop and definition is function of invariants (recursively)
  > replace as before

Loop Induction Variables

- Loop induction variable: increases or decreases by constant amount inside loop
  > e.g., for (i = 0; i < 100; i++)
- Opportunity for:
  > strength reduction
    > e.g., \( j = 2 \cdot i \) becomes \( j = j + 2 \)
  > identifying stride of accesses for prefetching
    > e.g., array accesses

Easy Detection of Loop Induction Variables

- Pattern match & check:
  > Search for "\( i = i + b \)" in loop
  > \( i \) is induction variable if no other assignment to \( i \) in loop

- Pros & Cons:
  > Easy!
  > Does not catch all loop induction variables
    > e.g., "\( i = a \cdot c + 2 \)"

Next Time

- Finding loop induction variables
- Strength reduction
- Read ACD I ch. 12, pp 333-342
- Project Design documents due
- March 13: project presentations
  > 5-10 minutes
  > 3 slides

Taxonomy of Induction Variables

- Basic induction variable:
  > only definition in loop is assignment
    > \( j = j \cdot c \), where \( c \) is loop invariant
- Mutual induction variable:
  > definition is linear function of other induction variable \( i \):
    > \( i = c \cdot i + d \)
    > \( i = i \cdot c \cdot i \cdot c \)
- Family of basic induction variable \( j \):
  > set of induction variables \( i \) such that \( i \) always assigned linear function of \( j \)