Local Optimizations

I. Basic blocks/Flow graphs
II. Abstraction 1: DAG
III. Abstraction 2: Value numbering

Basic block = a sequence of 3-address statements
- only the first statement can be reached from outside the block
  (no branches into middle of block)
- all the statements are executed consecutively if the first one is
  (no branches out or halts except perhaps at end of block)
- We require basic blocks to be maximal, i.e., they cannot be made larger
  without violating the conditions

Flow graph
- Nodes: basic blocks
- Edges: \( B_i \rightarrow B_j \) iff \( B_j \) can follow \( B_i \) immediately in some execution
  - Either first instruction of \( B_j \) is target of a goto at end of \( B_i \)
  - Or, \( B_j \) physically follows \( B_i \), which does not end in an unconditional goto.

Partitioning into Basic Blocks
Identify the leader of each basic block
- First instruction
- Any target of a jump
- Any instruction immediately following a jump

Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)
II. Local Optimizations (within basic block)

- Common subexpression elimination
  - array expressions
  - field access in records
  - access to parameters

Graph Abstractions

Example 1:
- grammar (for bottom-up parsing): $E \rightarrow E + T \mid E - T \mid T, \ T \rightarrow T^* F \mid F, \ F \rightarrow (E) \mid id$
- expression: $a + a \cdot (b - c) + (b - c) \cdot d$

Optimized code:

$t1 = b - c$
$t2 = a \cdot t1$
$t3 = a + t2$
$t4 = t1 \cdot d$
$t5 = t3 + t4$

Example 2:

$\begin{align*}
a &= b + c; \\
b &= a - d; \\
c &= b + c; \\
d &= a - d;
\end{align*}$

Is this optimized code correct?

$\begin{align*}
a &= b + c; \\
d &= a - d; \\
c &= d + c; \\
\text{Depends on whether } b \text{ is live on exit from the block}
\end{align*}$
Critique of DAGs

- **Cause of problems**
  - Assignment statements
  - Value of variable depends on time

- **How to fix problem?**
  - Build graph in order of execution
  - Attach variable name to latest value

- **Final graph created is not very interesting**
  - Key: variable-value mapping across time
  - Loses appeal of abstraction

### III. Value Number: Another Abstraction

- More explicit with respect to VALUES, and TIME

#### Static Variables

- Each value has its own “number”
  - Common subexpression means same value number
- var2value: current map of variable to value
  - Used to determine the value number of current expression

\[
\text{var2value} (r1 + r2) = \text{var2value}(r1) + \text{var2value}(r2)
\]

#### Dynamic Values

- **Value Numbering: Expression Example**

Expression: \(a + a \times (b - c) + (b - c) \times d\)

Optimized code:

\[
\begin{align*}
t4 &= b - c \\
t5 &= a \times t4 \\
t6 &= a + t5 \\
t8 &= t4 \times d \\
t9 &= t6 + t8
\end{align*}
\]

Value Numbering Algorithm

- **Data structure:**
  - VALUES = Table of expression /* [OP, valnum1, valnum2] */
  - var /* name of variable currently holding expr */

For each instruction (dst = src1 OP src2) in execution order

\[
\begin{align*}
\text{valnum}1 &= \text{var2value(src1)}; \text{valnum}2 &= \text{var2value(src2)} \\
\text{IF} \ [\text{OP}, \text{valnum}1, \text{valnum}2] \text{ is in VALUES} \\
\text{v} &= \text{the index of expression} \\
\text{Replace instruction with: dst = VALUES[v].var}
\end{align*}
\]

**ELSE**

- Add expression = [OP, valnum1, valnum2] to VALUES
- v = index of new entry
- \text{set_var2value} (dst, v)
More Details

- What are the initial values of the variables?
  - values at beginning of the basic block

- Possible implementations:
  - Initialization: create "initial values" for all variables
  - Or dynamically create them as they are used

- Implementation of VALUES and var2value: hash tables

Value Numbering: Basic Block Example

\[
\begin{align*}
a &= b + c & t1 &= b + c \\
b &= a - d & a &= t1 \\
c &= b + c & t2 &= t1 - d \\
d &= a - d & b &= t2 \\
c &= t2 + c & c &= t3 \\
d &= t2 \\
\end{align*}
\]

Q: Assigning to a temporary and then copying to the destination increases the number of instructions—so why do it?

A: If dst is overwritten later, would lose opportunity to eliminate common subexpression since no variable would hold the result

Value Numbering Algorithm

```c
Data structure:
VALUES = Table of
expression /* [OP, valnum1, valnum2] */
var /* name of variable currently holding expr */

For each instruction (dst = src1 OP src2) in execution order
valnum1=var2value(src1); valnum2=var2value(src2)
IF [OP, valnum1, valnum2] is in VALUES
  v = the index of expression
  Replace instruction with: dst = VALUES[v].var
ELSE
  Add
  expression = [OP, valnum1, valnum2]
  var = dst
  to VALUES
  v = index of new entry; tv is new temporary for v
  Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
  dst = tv

set_var2value (dst, v)
```

Question

- How do you extend value numbering to constant folding?

```
  a = 1 \\
b = 2 \\
c = a+b
```

Answer: Can add a field to the VALUES table indicating when an expression is a constant and what its value is
Conclusions

• Comparisons of two abstractions
  – DAGs
  – Value numbering

• Value numbering
  – VALUE: distinguish between variables and VALUES
  – TIME
    • Interpretation of instructions in order of execution
    • Keep dynamic state information

Monday's Class

• Data Flow Analysis
  – ALSU 9.2