LLVM Compiler Infrastructure

Goals
- lifelong analysis and optimization
- modular compiler components
IR is low level, control-flow graph and SSA
- language-independent types: 8-26 bit ints, float, double, pointers, arrays, structures, functions

Status
- Apple is paying main developer (Chris Lattner) to continue development
- built OpenGL JIT compiler with it in two weeks

Reuse Optimization

Idea
- Eliminate redundant operations in the dynamic execution of instructions

How do redundancies arise?
- Loop invariant code (e.g., index calculation for arrays)
- Sequence of similar operations (e.g., method lookup)
- Same value be generated in multiple places in the code

Types of reuse optimization
- Value numbering
- Common subexpression elimination
- Partial redundancy elimination

Local Value Numbering

Idea
- Each variable, expression, and constant is assigned a unique number
- When we encounter a variable, expression or constant, see if it’s already been assigned a number
  - If so, use the value for that number
  - If not, assign a new number
- Same number ⇒ same value

Example
- b := b + c
- d := b
- b + c is #1 + #2 = #3
- a := a
- a = #3
- d := a + c
- d + c = #1 + #2 = #3
- e := d + c
- e = #3

Temporary may be necessary

\[
\begin{align*}
   a & : b + c \\
   a & : b \\
   d & : a + c \\
   b & : \rightarrow 1 \\
   c & : \rightarrow 2 \\
   b + c & : \rightarrow 1 + \#2 = \#3 \\
   a & : \rightarrow #3 \\
   a + c & : \rightarrow #1 + #2 = #3 \\
   d & : \rightarrow #3 \\
   t & : b + c \\
   b & : \rightarrow 1 \\
   c & : \rightarrow 2 \\
   b + c & : \rightarrow 1 + \#2 = \#3 \\
   a & : b \\
   b & : \rightarrow #1 \\
   b + c & : \rightarrow #1 + #2 = #3 \\
   d & : b + c \\
   t & : \rightarrow #3 \\
   a & : a \\
   b + c & : \rightarrow #1 + #3 = #3 \\
   d & : \rightarrow #3
\end{align*}
\]
Global Value Numbering

How do we handle control flow?

\[
\begin{align*}
  w &= 5, \\
  x &= 8, \\
  w &\to #1, \\
  x &\to #1, \\
  y &= w+1, \\
  z &= x+1, \\
  w &\to #2, \\
  x &\to #2.
\end{align*}
\]

Global Value Numbering (cont)

Idea [Alpern, Wegman, and Zadeck 1988]
- Partition program variables into congruence classes
- All variables in a particular congruence class have the same value
- SSA form is helpful

Approaches to computing congruence classes
- Pessimistic
  - Assume no variables are congruent (start with \(n\) classes)
  - Iteratively coalesce classes that are determined to be congruent
- Optimistic
  - Assume all variables are congruent (start with one class)
  - Iteratively partition variables that contradict assumption
  - Slower but better results

Role of SSA Form

SSA form is helpful
- Allows us to avoid data-flow analysis
- Variables correspond to values

- Variables correspond to values

- Variables correspond to values

Congruence classes: \((a_1, b_1), (a_2, c_2), (a_3, d_3)\)

Basis

Idea
- If \(x\) and \(y\) are congruent then \(f(x)\) and \(f(y)\) are congruent

- Use this fact to combine (pessimistic) or split (optimistic) classes

Problem
- This is not true for \(\phi\)-functions

Solution: Label \(\phi\)-functions with join point
Pessimistic Global Value Numbering

Idea
- Initially each variable is in its own congruence class
- Consider each assignment statement (reverse postorder in CFG)
  - Update LHS value number with hash of RHS
- Identical value number ⇒ congruence

Why reverse postorder?
- Ensures that when we consider an assignment statement, we have already considered definitions that reach the RHS operands

Optimistic Global Value Numbering

Idea
- Initially all variables in one congruence class
- Split congruence classes when evidence of non-congruence arises
  - Variables that are computed using different functions
  - Variables that are computed using functions with non-congruent operands

Snag!

Problem
- Our algorithm assumes that we consider operands before variables that depend upon it
- Can’t deal with code containing loops!

Solution
- Ignore back edges
- Make conservative (worst case) assumption for previously unseen variable (i.e., assume its in it’s own congruence class)

Algorithm

```
for each assignment of the form: “x = f(a, b)”
valNum[x] ← UniqueValue() // same for a and b
for each assignment of the form: “x = f(a, b)” (in reverse postorder)
valNum[x] ← Hash(f ⊕ valNum[a] ⊕ valNum[b])
```

Snag!

Problem
- Our algorithm assumes that we consider operands before variables that depend upon it
- Can’t deal with code containing loops!

Solution
- Ignore back edges
- Make conservative (worst case) assumption for previously unseen variable (i.e., assume its in it’s own congruence class)
### Splitting

**Initially**
- Variables computed using the same function are placed in the same class

\[
\begin{align*}
\mathbf{x}_1 &= f(\mathbf{a}_1, \mathbf{b}_1) \\
\mathbf{y}_1 &= f(\mathbf{c}_1, \mathbf{d}_1) \\
\mathbf{z}_1 &= f(\mathbf{e}_1, \mathbf{f}_1)
\end{align*}
\]

**Iteratively**
- Split classes when corresponding operands are in different classes
- Example: assume \( a_1 \) and \( c_1 \) are congruent, but \( e_1 \) is congruent to neither

\[
\begin{align*}
\mathbf{x}_1 &= f(\mathbf{a}_1, \mathbf{b}_1) \\
\mathbf{y}_1 &= f(\mathbf{c}_1, \mathbf{d}_1) \\
\mathbf{z}_1 &= f(\mathbf{e}_1, \mathbf{f}_1)
\end{align*}
\]

\[
\begin{align*}
x_1 &= y_1 = z_1 \\
a_1 &= c_1
\end{align*}
\]

### Splitting (cont)

**Definitions**
- Suppose \( P \) and \( Q \) are sets representing congruence classes
- \( Q \) splits \( P \) for each \( i \) into two sets
- \( P \setminus i Q \) contains variables in \( P \) whose \( i \)th operand is in \( Q \)
- \( P / i Q \) contains variables in \( P \) whose \( i \)th operand is not in \( Q \)
- \( Q \) properly splits \( P \) if neither resulting set is empty

\[
\begin{align*}
\mathbf{x}_1 &= f(\mathbf{a}_1, \mathbf{b}_1) \\
\mathbf{y}_1 &= f(\mathbf{c}_1, \mathbf{d}_1) \\
\mathbf{z}_1 &= f(\mathbf{e}_1, \mathbf{f}_1)
\end{align*}
\]

\[
\begin{align*}
x_1 &= y_1 = z_1 \\
a_1 &= c_1
\end{align*}
\]

### Algorithm

\[
\begin{align*}
\text{worklist} &\leftarrow \emptyset \\
\text{for each function } f &
\begin{align*}
\mathbf{C}_f &\leftarrow \emptyset \\
\text{for each assignment of the form } \mathbf{x} = f(\mathbf{a}, \mathbf{b}) &
\begin{align*}
\mathbf{C}_f &\leftarrow \mathbf{C}_f \cup \{ \mathbf{x} \} \\
\text{worklist} &\leftarrow \text{worklist} \cup \{ \mathbf{C}_f \} \\
\text{CC} &\leftarrow \text{CC} \cup \{ \mathbf{C}_f \}
\end{align*}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{while } \text{worklist} \neq \emptyset &
\begin{align*}
\text{Delete some } \mathbf{D} \text{ from worklist} &
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{for each class } \mathbf{C} \text{ properly split by } \mathbf{D} \text{ (at operand } i) &
\begin{align*}
\text{CC} &\leftarrow \text{CC} \setminus \mathbf{C} \\
\text{worklist} &\leftarrow \text{worklist} \setminus \mathbf{C}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{Create new congruence classes } \mathbf{C}_j &\leftarrow \{ \mathbf{C} \setminus \mathbf{D} \} \text{ and } \mathbf{C}_k &\leftarrow \{ \mathbf{C} / \mathbf{D} \}
\end{align*}
\]

\[
\begin{align*}
\text{CC} &\leftarrow \text{CC} \cup \mathbf{C}_j \cup \mathbf{C}_k \\
\text{worklist} &\leftarrow \text{worklist} \cup \mathbf{C}_j \cup \mathbf{C}_k
\end{align*}
\]

Note: see paper for optimization

### Example

**SSA code**

\[
\begin{align*}
x_0 &= 1 \\
y_0 &= 2 \\
x_1 &= x_0 + 1 \\
y_1 &= y_0 + 1 \\
z_1 &= x_0 + 1
\end{align*}
\]

**Congruence classes**

\[
\begin{align*}
S_0 &\leftarrow \{ x_0 \} \\
S_1 &\leftarrow \{ y_0 \} \\
S_2 &\leftarrow \{ x_1, y_1, z_1 \} \\
S_3 &\leftarrow \{ x_1, z_1 \} \\
S_4 &\leftarrow \{ y_1 \}
\end{align*}
\]

**Worklist:** \( S_0 \setminus S_1 S_2 \setminus S_1 S_3 \setminus S_2 \setminus S_1 S_4 \setminus S_2 \setminus S_1 S_3 \setminus S_2 \setminus S_1 S_4 \)

\[
\begin{align*}
S_0 \text{ splits } S_2? &\quad \text{no} \\
S_0 \text{ splits } S_2? &\quad \text{no} \\
S_0 \text{ splits } S_2? &\quad \text{yes!}
\end{align*}
\]

\[
\begin{align*}
S_2 \setminus S_1 S_1 &\leftarrow \{ x_1, z_1 \} = S_3 \\
S_2 \setminus S_1 S_3 &\leftarrow \{ y_1 \} = S_4
\end{align*}
\]
Comparing Optimistic and Pessimistic

**Differences**
- Handling of loops
- Pessimistic makes worst-case assumptions on back edges
- Optimistic requires actual contradiction to split classes

\[
\begin{align*}
  w_0 &= 5 \\
  x_0 &= 5 \\
  w_1 &= \phi(w_0, w_1) \\
  x_1 &= \phi(x_0, x_1) \\
  w_2 &= w_1 + 1 \\
  x_2 &= x_1 + 1
\end{align*}
\]

Role of SSA

**Single global result**
- Single def reaches each use
- No data (flow value) at each point

**No data flow analysis**
- Optimistic: Iterate over congruence classes, not CFG nodes
- Pessimistic: Visit each assignment once

**\(\phi\)-functions**
- Make data-flow merging explicit
- Treat like normal functions after subscripting them for each merge point

Next Time

**Lecture**
- Parallelism and Data Locality, start reading chapter 11 in dragon book

**Suggested Exercise**
- Use any programming language to write the class declarations for the equivalence class and variable structures described at the bottom of page 6 in the value numbering chapter.
- Instantiate the data structures for the small example programs in Figures 7.2 and 7.6 and perform optimistic value numbering. Remember to convert to SSA first.
- After performing value numbering, how will the program be transformed?
- Now perform pessimistic value numbering on the two examples. Transform the code as well.