Using Static Single Assignment Form

Last Time
- Constructing SSA form

Today
- Finish naming algorithm
- Transforming from SSA
- Using SSA for program optimization
  - Dead-code elimination
  - Constant propagation
  - Register allocation
  - Copy propagation
  - Induction variables

Variable Renaming (cont)

Data Structures
- \( \text{Stacks}[v] \forall v \)
  - Holds the subscript of most recent definition of variable \( v \), initially empty
- \( \text{Counters}[v] \forall v \)
  - Holds the current number of assignments to variable \( v \); initially 0

Auxiliary Routine
procedure \( \text{GenName}(\text{variable } v) \)
  \[
i := \text{Counters}[v] \\
push i \text{ onto } \text{Stacks}[v] \\
\text{Counters}[v] := i + 1
\]

Use the Dominance Tree to remember the most recent definition of each variable

Transformation from SSA Form

Proposal
- Restore original variable names (i.e., drop subscripts)
- Delete all \( \phi \)-functions

Complications
- What if versions get out of order? (simultaneously live ranges)

Alternative
- Perform dead code elimination (to prune \( \phi \)-functions)
- Replace \( \phi \)-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies

Variable Renaming Algorithm

procedure \( \text{Rename(block } b) \)
  if \( b \) previously visited return
  Call Rename(entry-node)
  for each \( \phi \)-function \( p \) in \( b \)
    GenName(LHS(p)) and replace \( v \) with \( v_i \), where \( i = \text{Top(Stack}[v]) \)
  for each statement \( s \) in \( b \) (in order)
    for each variable \( v \in \text{RHS}(s) \)
      replace \( v \) by \( v_i \), where \( i = \text{Top(Stack}[v]) \)
    for each variable \( v \in \text{LHS}(s) \)
      GenName(\( v \)) and replace \( v \) with \( v_i \), where \( i = \text{Top(Stack}[v]) \)
      \( j \leftarrow \) position in \( s \)'s \( \phi \)-function corresponding to block \( b \)
      for each \( \phi \)-function \( p \) in \( s \)
        replace the \( j \)th operand of RHS(p) by \( v_i \), where \( i = \text{Top(Stack}[v]) \)
      Rename(s)
      for each \( \phi \)-function or statement \( t \) in \( b \)
        Unwind stack when done with this node
  for each \( v \in \text{LHS}(t) \)
    Pop(Stack[v])
  for each \( \phi \)-function or statement \( t \) in \( b \)
    Rename(s)
Dead Code Elimination for SSA

Dead code elimination
while ∃ a variable v with no uses and whose def has no other side effects
    Delete the statement s that defines v
    for each of s’s ud-chains
        Delete the corresponding du-chain that points to s

If y becomes dead and there are no other uses of x, then the assignment to x becomes dead, too

– Contrast this approach with one that uses liveness analysis
– This algorithm updates information incrementally
– With liveness, we need to invoke liveness and dead code elimination iteratively until we reach a fixed point

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)

Data-Flow Analysis for Simple Constant Propagation

Simple constant propagation

V: \{All constants\} ∪ \{T,⊥\}

F:

- \(F_{x=c}(\text{In}) = c\)
- \(F_{x=y+z}(\text{In}) = \text{if } c_1\text{-In}_p \& c_2\text{-In}_p, \text{then } c_1 \oplus c_2, \text{else } \top \text{ or } \bot\)
- . . .

Using tuples of lattices

- \(\tau\)
- \(-3\) - \(-2\) - \(-1\) - \(0\) - \(1\) - \(2\) - \(3\) . . .

Implementing Simple Constant Propagation

Standard worklist algorithm
– Identifies simple constants
– For each program point, maintains one constant value for each variable
– O(EV) (E is the number of edges in the CFG; V is number of variables)

Problem
– Inefficient, since constants may have to be propagated through irrelevant nodes

Solution
– Exploit a sparse dependence representation (e.g., SSA)
Sparse Simple Constant Propagation

**Reif and Lewis algorithm**
- Identifies simple constants
- Faster than Simple Constants algorithm

**SSA edges**
- Explicitly connect defs with uses
- How would you do this?

**Main Idea**
- Iterate over SSA edges instead of over all CFG edges

```
x = 1
y = x
```

Sparse Simple Constants Algorithm (Ch. 19 in Appel)

```
worklist = all statements in SSA
while worklist ≠ ∅
  Remove some statement S from worklist
  if S is x = phi(c,c,...,c) for some constant c
    replace S with v = c
  if S is x = c for some constant c
    delete x from program
    for each statement T that uses v
      substitute c for x in T
  worklist = worklist union {T}
```

Sparse Simple Constants

**Complexity**
- $O(E') = O(EV)$, $E'$ is number of SSA edges
- $O(N)$ in practice

Backward Analyses vs. Forward Analyses

For forward data-flow analysis, at phi node apply meet function

For backward data-flow analysis?

```
v_0 := ...  
v_1 := ...  
v_2 := \phi(v_0, v_1)
      \phi(v_2, v_1)
```

...v_2...

...v_0...

2

3

4

5
Copy Propagation

**Algorithm**
- worklist = all statements in SSA
- while worklist ≠ ∅
  - Remove some statement S from worklist
  - if S is x = phi(y) or x = y
    - for each statement T that uses x
      - replace all use of x with y
    - worklist = worklist union {T}
  - delete S

Induction Variable Identification

**Types of Induction Variables**
- **Basic** induction variables
  - Variables that are defined once in a loop by a statement of the form,
    \[ i = i + c \] (or \[ i = i \times c \]), where \( c \) is a constant integer
- **Derived** induction variables
  - Variables that are defined once in a loop as a linear function of another induction variable
    - \( j = c_1 \times i + c_2 \)
    - \( j = i / c_1 + c_2 \), where \( c_1 \) and \( c_2 \) are loop invariant
Induction Variable Identification (cont)

**Informal SSA-based Algorithm**
- Build the SSA representation
- Iterate from innermost CFG loop to outermost loop
  - Find SSA cycles
    - Each cycle may be a **basic** induction variable if a variable in a cycle is a function of loop invariants and its value on the current iteration
  - Find **derived** induction variables as functions of loop invariants and basic induction variables

Next Time

**Reading**
- Value Numbering chapter

**Lecture**
- Value Numbering