Lecture 6

More Examples of Data Flow Analysis: Global Common Subexpression Elimination; Constant Propagation/Folding

I. Available Expressions Analysis

II. Eliminating CSEs

III. Constant Propagation/Folding

Reading: 9.2.6, 9.4

Global Common Subexpressions

```
add t1 = x, y
add t2 = c, d
```

```
ldc t3 = 0
cpy x = t3
add t4 = x, y
cpy m = t4
```

```
sub t5 = a, b
ldc t6 = -1
cpy c = t6
```

```
sub t7 = a, b
cpy m = t7
add t8 = x, y
add t9 = c, d
```

- **Availability of an expression E at point P**
  - DEFINITION: Along every path to P in the flow graph:
    - E must be evaluated at least once
    - no variables in E redefined after the last evaluation
  - Observations: E may have different values on different paths

Formulating the Problem

- **Domain:**
  - a bit vector, with
    a bit for each **textually unique** expression in the program
- **Forward or Backward?**
- **Lattice Elements?**
- **Meet Operator?**
  - check: commutative, idempotent, associative
- **Partial Ordering**

- **Top?**
- **Bottom?**
- **Boundary condition: entry/exit node?**
- **Initialization for iterative algorithm?**

Transfer Functions

- **Can use the same equation as reaching definitions**
  - out[b] = gen[b] ∪ (in[b] - kill[b])
- **Start with the transfer function for a single instruction**
  - When does the instruction generate an expression?
  - When does it kill an expression?
- **Calculate transfer functions for complete basic blocks**
  - Compose individual instruction transfer functions

Composing Transfer Functions

- **Derive the transfer function for an entire block**

in1

1

out1 = gen1 U (in1 - kill1) = in2

2

out2 = gen2 U (in2 - kill2)

- **Since out1 = in2 we can simplify:**
  - out2 = gen2 U ((gen1 U (in1 - kill1)) - kill2)
  - out2 = gen2 U (gen1 - kill2) U (in1 - (kill1 U kill2))
  - out2 = gen2 U (gen1 - kill2) U (in1 - (kill2 U (kill1 - gen2)))
- **Result**
  - gen = gen2 U (gen1 - kill2)
  - kill = kill2 U (kill1 - gen2)

II. Eliminating CSEs

- ***Available expressions (across basic blocks)***
  - provides the set of expressions available at the start of a block
- **Value Numbering (within basic block)**
  - Initialize Values table with available expressions
- **If CSE is an "available expression", then transform the code**
  - Original destination may be:
    - a temporary register
    - overwritten
    - different from the variables on other paths
  - One solution: Copy the expression to a new variable at each evaluation reaching the redundant use

III. Limitation: Textually Identical Expressions

- **Commutative operations**

```
add t1 = x, y        add t2 = y, x

          add t3 = x, y
```

  - sort the operands

Further Improvements

- **Examples**
  - Expressions with more than two operands

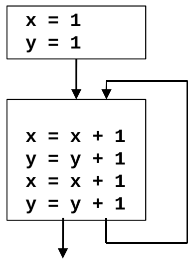
```
add t1 = x, y        add t3 = y, x
add t2 = t1, z       add t4 = t3, z

          add t5 = x, y
          add t6 = t5, z
```

  - Textually different expressions may be equivalent

```
add t1 = x, y
beq t1, t2, L1
cpy z = x
add t3 = z, y
```

Another Example

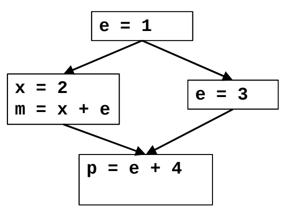


Summary

	Reaching Definitions	Available Expressions
Domain	Sets of definitions	Sets of expressions
Transfer function $f_b(x)$ Generate U Propagate		
direction of function	forward: out[b] = f_b(in[b])	forward: out[b] = f_b(in[b])
Generate	Gen_b: exposed definitions	Gen_b: expressions evaluated
Propagate	in[b]-$Kill_b$: definitions killed	in[b]-$Kill_b$: expressions killed
Meet operation	U (in[b]= U out[predecessors])	$\cap$ (in[b]= $\cap$ out[predecessors])
Initialization	out[entry] = $\varnothing$ out[b] = $\varnothing$	out[entry] = $\varnothing$ out[b] = all expressions

III. Constant Propagation/Folding

- **At every basic block boundary, for each variable v**
  - determine if v is a constant
  - if so, what is the value?



Semi-lattice Diagram

– Finite domain?
– Finite height?

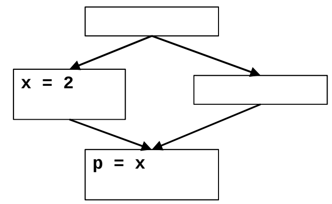
Equivalent Definition

- **Meet Operation:**

v1	v2	v1 ∧ v2
undef	undef	
	c_2	
	NAC	
c_1	undef	
	c_2	
	NAC	
NAC	undef	
	c_2	
	NAC	

  – Note: undef ∧ c2 = c2!

Example



Transfer Function

- **Assume a basic block has only 1 instruction**
- **Let IN[b,x], OUT[b,x]**
  – be the information for variable x at entry and exit of basic block b
- **OUT[entry, x] = undef, for all x.**
- **Non-assignment instructions: OUT[b,x] = IN[b,x]**
- **Assignment instructions: (next page)**

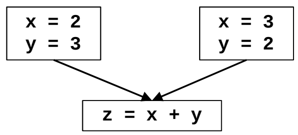
Constant Propagation (Cont.)

- **Let an assignment be of the form $x_3 = x_1 + x_2$**
  - "+" represents a generic operator
  - OUT[b,x] = IN [b,x], if $x \neq x_3$

IN[b,x_1]	IN[b,x_2]	OUT[b,x_3]
undef	undef	
	c_2	
	NAC	
c_1	undef	
	c_2	
	NAC	
NAC	undef	
	c_2	
	NAC	

- **Use:** $x \leq y$ implies $f(x) \leq f(y)$ to check if framework is monotone
  - $[v_1\ v_2 \ldots] \leq [v_1'\ v_2' \ldots]$, $f([v_1\ v_2 \ldots]) \leq f([v_1'\ v_2' \ldots])$

Distributive?



Summary of Constant Propagation

- **A useful optimization**
- **Illustrates:**
  - abstract execution
  - an infinite semi-lattice
  - a non-distributive problem

Other Optimizations

- **Copy Propagation:**

- **Dead Code Elimination:**