Lecture 7

Global Common Subexpression Elimination; Constant Propagation/Folding

I. Available Expressions Analysis

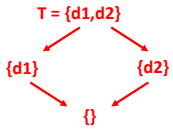
II. Eliminating CSEs

III. Constant Propagation/Folding

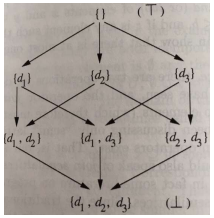
[ALSU 9.2.6, 9.4]

Review: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values **V**
  - meet operator ∧
  - Top **T**
  - finite descending chain?



Meet Operator: Intersection



Meet Operator: Union

Review: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values **V**
  - meet operator ∧
  - Top **T**
  - finite descending chain?
- **Transfer functions**
  - function of a basic block **f: V → V**
  - closed under composition
  - meet-over-paths **MOP**
  - monotone
  - distributive?

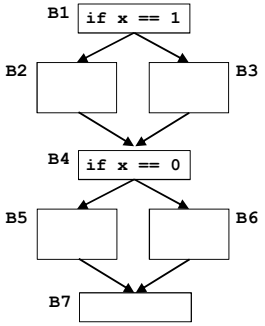
For each node *n*: MOP(*n*) = $\wedge f_{p_i}$ (T), for all paths p_i reaching *n*

If data flow framework is monotone (i.e., x ≤ y implies f(x) ≤ f(y)) then if the algorithm converges, IN[b] ≤ MOP[b]

Data flow framework (monotone) converges if there is a finite descending chain

If data flow framework is distributive (i.e., f(x ∧ y) = f(x) ∧ f(y)) then if the algorithm converges, IN[b] = MOP[b]

Example: MOP considers more paths than Ideal



Assume: B2 & B3 do not update x

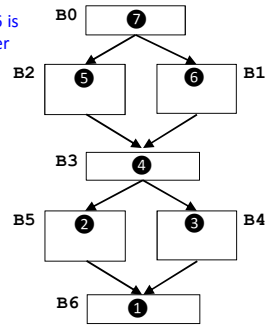
Ideal: Considers only 2 paths
B1-B2-B4-B6-B7 (i.e., x=1)
B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths
B1-B2-B4-B5-B7
B1-B3-B4-B6-B7

Review: A Check List for Data Flow Problems

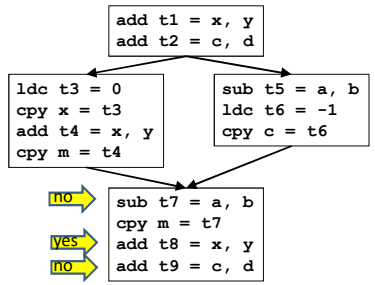
- **Semi-lattice**
  - set of values **V**
  - meet operator **∧**
  - Top **T**
  - finite descending chain?
- **Transfer functions**
  - function of a basic block **f: V → V**
  - closed under composition
  - meet-over-paths **MOP**
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: **rPostOrder**
  - depth of the graph

B0,B1,…,B6 is rPostOrder



Number of iterations = number of back edges in any acyclic path + 2

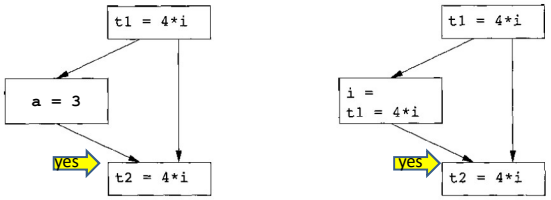
Global Common Subexpressions



Is expression available?

- **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
  - Observation: E may have different values on different paths (e.g., x+y above)

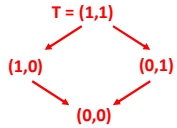
Available Expressions Example



Is `4*i` available at this point?

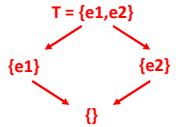
Formulating the Problem

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



Meet Operator: Elementwise-min

- **Top?** (1,1,…,1)
- **Bottom?** (0,0,…,0)
- **Boundary condition: entry/exit node?** out[entry]=(0,…,0)
- **Initialization for iterative algorithm?** Coming soon…



Meet Operator: Intersection

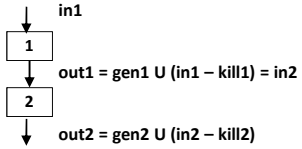
Transfer Functions

- **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

Statement	Available Expressions
	{}
a = b + c	{b + c}
b = a - d	{a-d}
c = b + c	{a-d}
d = a - d	{}

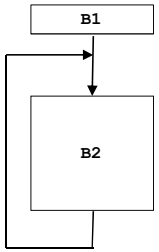
Composing Transfer Functions

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

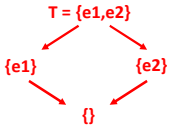


- **Since out1 = in2 we can simplify:**
  - out2 = gen2 U ((gen1 U (in1 - kill1)) - kill2)
  - out2 = gen2 U (gen1 - kill2) U (in1 - (kill1 U kill2))
  - out2 = gen2 U (gen1 - kill2) U (in1 - (kill2 U (kill1 - gen2)))
- **Result**
  - gen = gen2 U (gen1 - kill2)
  - kill = kill2 U (kill1 - gen2)

Initialization for Interior Nodes



out[b] = Gen[b] U (in(b)-Kill[b])



Meet Operator: Intersection

What if initialize out[B2] = {}? Incorrect: in[B2]={}

What if initialize out[B2] = T? Correct: in[B2]=out[B1]

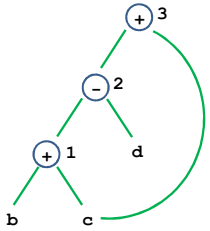
Initialize out[b]=T for all interior b

II. Eliminating CSEs

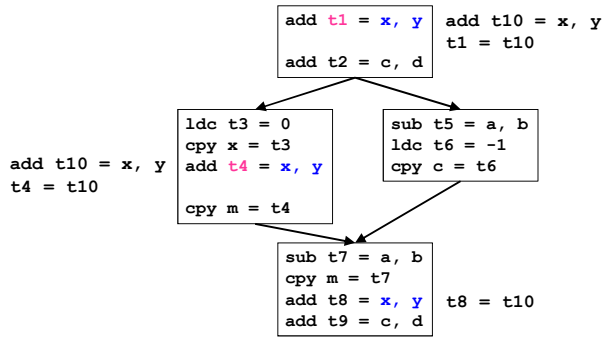
- **Available expressions (across basic blocks)**
  - provides the set of expressions available at the start of a block
- **Value Numbering (within basic block)**
  - Initialize Values table with available expressions
- **If CSE is an "available expression", then transform the code**
  - Original destination 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

Review: Value Numbering

```
a = b+c        t1 = b + c
               a = t1
b = a-d        t2 = t1 - d
               b = t2
c = b+c        t3 = t2 + c
               c = t3
d = a-d        d = t2
```

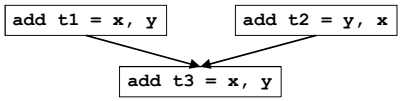


Example Revisited



Limitation: Textually Identical Expressions

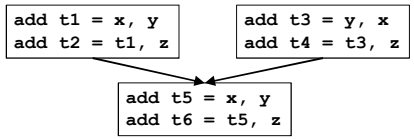
- **Commutative operations**



  - sort the operands

Further Improvements

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



  - Textually different expressions may be equivalent

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

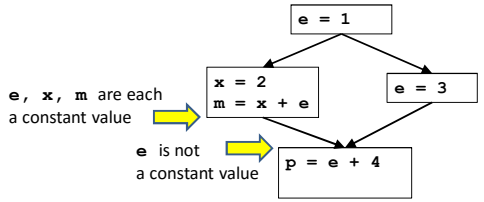
Use multiple passes of GCSE combined with copy propagation

Summary

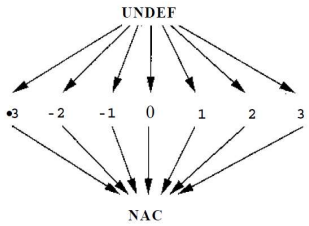
	Reaching Definitions	Available Expressions
Domain	Sets of definitions	Sets of expressions
Transfer function $f_b(x)$ Generate U Propagate		
direction of function	forward: out[b] = f_b(in[b])	forward: out[b] = f_b(in[b])
Generate	Gen_b: exposed definitions	Gen_b: expressions evaluated
Propagate	in[b]-$Kill_b$: definitions killed	in[b]-$Kill_b$: expressions killed
Meet operation	U (in[b]= U out[predecessors])	∩ (in[b]= ∩ out[predecessors])
Initialization	out[entry] = ∅ out[b] = ∅	out[entry] = ∅ 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? No (unless bound number of bits)
- Finite height? Yes (2)

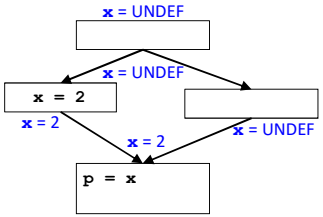
Equivalent Definition

- **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)**

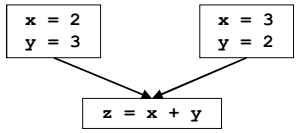
Constant Propagation (Cont.)

- **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 ≤ y implies f(x) ≤ 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?



- Not Distributive
- Iterative solution is not precise!
  - it is also not wrong
  - it is conservative

Summary of Constant Propagation

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

Monday's Class

- Static Single Assignment (SSA) [ALSU 6.2.4]