15-745 Lecture 2

Dataflow Analysis
Basic Blocks
Related Optimizations

Dataflow Analysis

- Last time we looked at code transformations
  - Constant propagation
  - Copy propagation
  - Common sub-expression elimination
  - …
- Today, dataflow analysis:
  - How to determine if it is legal to perform such an optimization
  - (Not doing analysis to determine if it is beneficial)

A sample program

```
int fib10(void) {
  int n = 10;
  int older = 0;
  int old = 1;
  ir
  int i;

  if (n <= 1) return n;
  for (i = 2; i<n; i++) {
      result = old + older;
      older = old;
      old = result;
  }
  return result;
}
```

What are those numbers?

A Comment about the IR

```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13
8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```

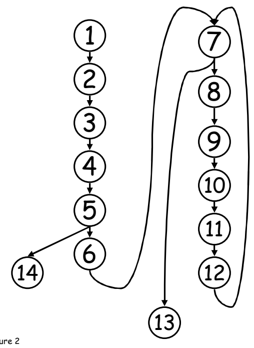

Simple Constant Propagation

- Can we do SCP?
- How do we recognize it?
- What aren't we doing?
- Metanote:
  - keep opts simple!
  - Use combined power

```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13
8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```

Reaching Definitions

- A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.




```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13
8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```


Reaching Definitions (ex)

- 1 reaches 5, 7, and 14

14, Really?

Meta-notes:
- (almost) always conservative
- only know what we know
- Keep it simple:
  - What opt(s), if run before this would help
  - What about:

```
1: x <- 0
2: if (false) x<-1
3: … x …
```

  - Does 1 reach 3?
  - What opt changes this?

```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13
8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```


Calculating Reaching Definitions

- A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.

- Build up RD stmt by stmt
- Stmt s, "d: v <- x op y", generates d
- Stmt s, "d: v <- x op y", kills all other defs(v)

Or,
- Gen[s] = { d }
- Kill[s] = defs(v) – { d }

Gen and kill for each stmt

	Gen	kill
1: n <- 10	1	
2: older <- 0	2	9
3: old <- 1	3	10
4: result <- 0	4	8
5: if n <= 1 goto 14		
6: i <- 2	6	11
7: if i > n goto 13		
8: result <- old + older	8	4
9: older <- old	9	2
10: old <- result	10	3
11: i <- i + 1	11	6
12: JUMP 7		
13: return result		
14: return n		

How can we determine the defs that reach a node?

We can use:
- control flow information
- gen and kill info

Computing in[n] and out[n]

- In[n]: the set of defs that reach the beginning of node n
- Out[n]: the set of defs that reach the end of node n

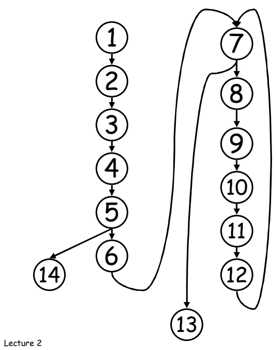
$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

- Initialize in[n]=out[n]={} for all n
- Solve iteratively

What is pred[n]?

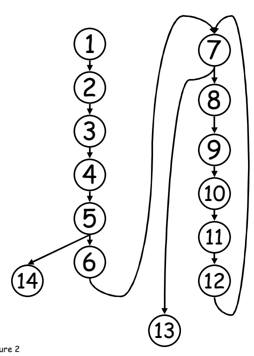
- Pred[n] are all nodes that can reach n in the control flow graph.
- E.g., pred[7] = { 6, 12 }




```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13
8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```


What order to eval nodes?

- Does it matter?
- Lets do: 1,2,3,4,5,14,6,7,13,8,9,10,11,12




```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13
8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```


Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2,3	1-4
4: result <- 0	4	8		
5: if n <= 1 goto 14				
6: i <- 2	6	11		
7: if i > n goto 13				
8: result <- old + older	8	4		
9: older <- old	9	2		
10: old <- result	10	3		
11: i <- i + 1	11	6		
12: JUMP 7				
13: return result				
14: return n				

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$in[n] = \bigcup_{p \in pred[n]} out[p]$ $\quad out[n] = gen[n] \cup (in[n] - kill[n])$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6	1-4,6
8: result <- old + older	8	4	1-4,6	1-3,6,8
9: older <- old	9	2	1-3,6,8	1,3,6,8,9
10: old <- result	10	3	1,3,6,8,9	1,6,8-10
11: i <- i + 1	11	6	1,6,8-10	1,8-11
12: JUMP 7			1,8-11	1,8-11
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$in[n] = \bigcup_{p \in pred[n]} out[p]$ $\quad out[n] = gen[n] \cup (in[n] - kill[n])$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6,8-11	1-4,6,8-11
8: result <- old + older	8	4	1-4,6,8-11	1-3,6,8-11
9: older <- old	9	2	1-3,6,8-11	1,3,6,8-11
10: old <- result	10	3	1,3,6,8-11	1,6,8-11
11: i <- i + 1	11	6	1,6,8-11	1,8-11
12: JUMP 7			1,8-11	1,8-11
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Can we combine the gen and kill sets into one set per BB?

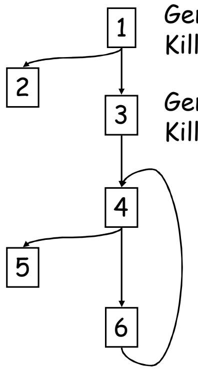
	Gen	kill
1: i <- 1	1	8,4
2: j <- 2	2	
3: k <- 3 + i	3	11
4: i <- j	4	1,8
5: m <- i + k	5	

- Gen[BB]={2,3,4,5}
- Kill[BB]={1,8,11}
- Gen[s1;s2]=
- Kill[s1;s2]=

BB sets

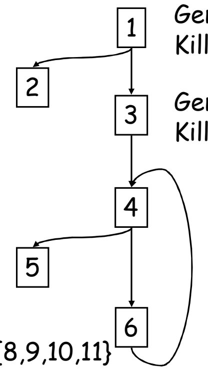
BB		Gen	kill		
	1: n <- 10	1			
	2: older <- 0	2	9		
1	3: old <- 1	3	10		
	4: result <- 0	4	8		
	5: if n <= 1 goto 14			1,2,3,4	8,9,10
3	6: i <- 2	6	11	6	11
4	7: if i > n goto 13				
	8: result <- old + older	8	4		
	9: older <- old	9	2		
6	10: old <- result	10	3		
	11: i <- i + 1	11	6		
	12: JUMP 7			8-11	2-4,6
5	13: return result				
2	14: return n				

BB sets



Block	Gen	Kill	In	out
1	{1,2,3,4}	{8,9,10}		1,2,3,4
3	{6}	{11}		
6	{8,9,10,11}	{2,3,4,6}		

BB sets



Block	Gen	Kill	In	out
1	{1,2,3,4}	{8,9,10}		1,2,3,4
3	{6}	{11}	1,2,3,4	1,2,3,4,6
4			1-4,6,8-11	1-4,6,8-11
6	{8,9,10,11}	{2,3,4,6}	1-4,6,8-11	1,8-11

Forward Dataflow

- Reaching definitions is a forward dataflow problem: It propgates information from preds of a node to the node
- Defined by:
  - Basic attributes: (gen and kill)
  - Transfer function: $out[b]=F_{bb}(in[b])$
  - Meet operator: $in[b]=M(out[p])$ for all $p \in pred(b)$
  - Set of values (a lattice, in this case powerset of program points)
  - Initial values for each node b
- Solve for fixed point solution

How to implement?

- Values?
- Gen?
- Kill?
- F_{bb}?
- Order to visit nodes?
- When are we done?
  - In fact, do we know we terminate?

Implementing RD

- Values: bits in a bit vector
- Gen: 1 in each position generated, otherwise 0
- Kill: 0 in each position killed, otherwise 1
- F_{bb}: out[b] = (in[b] | gen[b]) & kill[b]
- Init in[b]=out[b]=0

- When are we done?
- What order to visit nodes? Does it matter?

RD Worklist algorithm

Initialize: in[B] = out[b] = ∅
Initialize: in[entry] = ∅
Work queue, W = all Blocks in topological order
while (|W| != 0) {
  remove b from W
  old = out[b]
  in[b] = {over all pred(p) ∈ b} ∪ out[p]
  out[b] = gen[b] ∪ (in[b] - kill[b])
  if (old != out[b]) W = W ∪ succ(b)
}

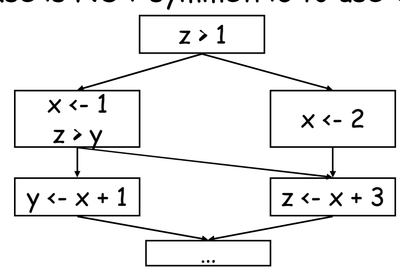
Storing Rd information

- Use-def chains: for each use of var x in s, a list of definitions of x that reach s

1:	n <- 10	1	
2:	older <- 0	1	1,2
3:	old <- 1	1,2	1,2,3
4:	result <- 0	1-3	1-4
5:	if n <= 1 goto 14	1-4	1-4
6:	i <- 2	1-4	1-4,6
7:	if i > n goto 13	1-4,6,8-11	1-4,6,8-11
8:	result <- old + older	1-4,6,8-11	1-3,6,8-11
9:	older <- old	1-3,6,8-11	1,3,6,8-11
10:	old <- result	1,3,6,8-11	1,6,8-11
11:	i <- i + 1	1,6,8-11	1,8-11
12:	JUMP 7	1,8-11	1,8-11
13:	return result	1-4,6	1-4,6
14:	return n	1-4	1-4

Def-use chains are valuable too

- Def-use chain: for each definition of var x, a list of all uses of that definition
- Computed from liveness analysis, a backward dataflow problem
- Def-use is NOT symmetric to use-def



Using RD for Simple Const. Prop.

```
1:  n <- 10                      1
2:  older <- 0                   1            1,2
3:  old <- 1                     1,2          1,2,3
4:  result <- 0                  1-3          1-4
5:  if n <= 1 goto 14            1-4          1-4
6:  i <- 2                       1-4          1-4,6
7:  if i > n goto 13             1-4,6,8-11   1-4,6,8-11
8:  result <- old + older        1-4,6,8-11   1-3,6,8-11
9:  older <- old                 1-3,6,8-11   1,3,6,8-11
10: old <- result                1,3,6,8-11   1,6,8-11
11: i <- i + 1                   1,6,8-11     1,8-11
12: JUMP 7                       1,8-11       1,8-11
13: return result                1-4,6        1-4,6
14: return n                     1-4          1-4
```

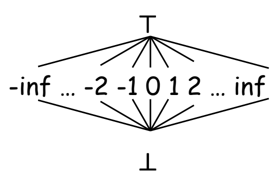

Better Constant Propagation

- What about:

  x <- 1
  if (y > z)
      x <- 1
  a <- x

Better Constant Propagation

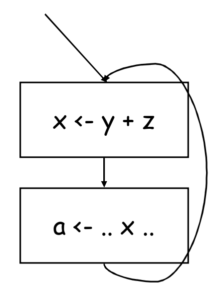
- What about: x <- 1
  if (y > z)
      x <- 1
  a <- x
- Use a better lattice



- Meet: a <- a ∧ top
  bot <- a ∧ bot
  c <- c ∧ c
  bot <- c ∧ d (if c ≠ d)
- Init all vars to: bot or top?

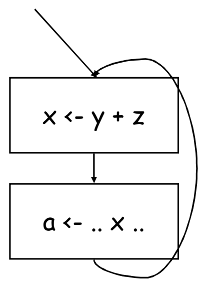
Loop Invariant Code Motion

- When can expression be moved out of a loop?



Loop Invariant Code Motion

- When can expression be moved out of a loop?
- When all reaching definitions of operands are outside of loop, expression is loop invariant
- Use ud-chains to detect
- Can du-chains be helpful?



Liveness (def-use chains)

- A variable x is live-out of a stmt s if x can be used along some path starting a s, otherwise x is dead.
- Why is this important?
- How can we frame this as a dataflow problem?

Liveness as a dataflow problem

- This is a backwards analysis
  - A variable is live out if used by a successor
  - Gen: For a use: indicate it is live coming into s
  - Kill: Defining a variable v in s makes it dead before s (unless s uses v to define v)
  - Lattice is just live (top) and dead (bottom)
- Values are variables
- In[n] = variables live before n
  = $out[n] - kill[n] \cup gen[n]$
- Out[n] = variables live after n
  = $\bigcup_{s \in succ(n)} In[s]$

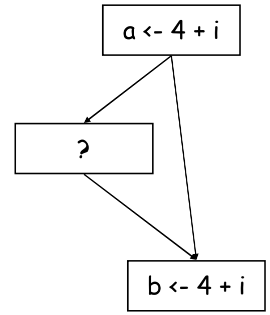
Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?

Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?
  - When the definition is dead, and
  - When the instruction has no side effects
- So:
  - run liveness
  - Construct def-use chains
  - Any instruction which has no users and has no side effects can be eliminated

When can we do CSE?



Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

a <- b+c

b <- a-d

c <- b+c

d <- a-d

Computing Available Expressions

- Forward or backward?
- Values?
- Lattice?
- gen[b] =
- kill[b] =
- in[b] =
- out[b] =
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, then all exprs using x are killed.
- out[b] = in[b] – kill[b] ∪ gen[b]
- in[b] = what to do at a join point?
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, exprs(x) are killed
  out[b] = in[b] – kill[b] ∪ gen[b]
- in[b] = An expr is avail only if avail on ALL edges, so: in[b] = ∩ over all p∈ pred(b), out[p]
- Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail