

# Dataflow Analysis

Monday, November 9, 15

# Program optimizations

- So far we have talked about different kinds of optimizations
  - Peephole optimizations
  - Local common sub-expression elimination
  - Loop optimizations
- What about *global optimizations*
  - Optimizations across multiple basic blocks (usually a whole procedure)
  - Not just a single loop

Monday, November 9, 15

# Useful optimizations

- Common subexpression elimination (global)
  - Need to know which expressions are available at a point
- Dead code elimination
  - Need to know if the effects of a piece of code are never needed, or if code cannot be reached
- Constant folding
  - Need to know if variable has a constant value
- So how do we get this information?

Monday, November 9, 15

# Dataflow analysis

- Framework for doing compiler analyses to drive optimization
- Works across basic blocks
- Examples
  - Constant propagation: determine which variables are constant
  - Liveness analysis: determine which variables are live
  - Available expressions: determine which expressions are have valid computed values
  - Reaching definitions: determine which definitions could “reach” a use

Monday, November 9, 15

# Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding

```
x = 1;  
y = x + 2;  
if (x > z) then y = 5  
... y ...
```

- Create dead code

```
x = 1;  
y = x + 2;  
if (y > x) then y = 5  
... y ...
```

Monday, November 9, 15

# Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding

```
x = 1;  
y = x + 2;  
if (x > z) then y = 5  
... y ...
```

- Create dead code

```
x = 1;  
y = x + 2;  
if (y > x) then y = 5  
... y ...
```

Monday, November 9, 15

## Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding

```

x = 1;
y = x + 2;
if (x > z) then y = 5
... y ...
    
```

→

```

x = 1;
y = 3;
if (x > z) then y = 5
... y ...
    
```

- Create dead code

```

x = 1;
y = x + 2;
if (y > x) then y = 5
... y ...
    
```

→

```

x = 1;
y = 3; //dead code
if (true) then y = 5 //simplify!
... y ...
    
```

Monday, November 9, 15

## How can we find constants?

- Ideal: run program and see which variables are constant
- Problem: variables can be constant with some inputs, not others – need an approach that works for all inputs!
- Problem: program can run forever (infinite loops?) – need an approach that we know will finish
- Idea: run program *symbolically*
- Essentially, keep track of whether a variable is constant or not constant (but nothing else)

Monday, November 9, 15

## Overview of algorithm

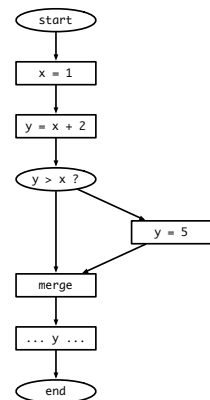
- Build control flow graph
- We'll use statement-level CFG (with merge nodes) for this
- Perform symbolic evaluation
- Keep track of whether variables are constant or not
- Replace constant-valued variable uses with their values, try to simplify expressions and control flow

Monday, November 9, 15

## Build CFG

```

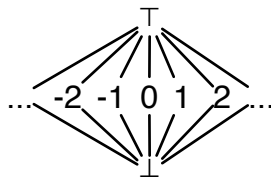
x = 1;
y = x + 2;
if (y > x) then y = 5;
... y ...
    
```



Monday, November 9, 15

## Symbolic evaluation

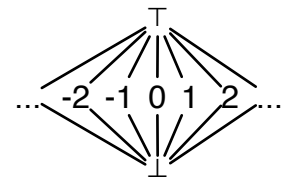
- Idea: replace each value with a symbol
- constant (specify which), no information, definitely not constant
- Can organize these possible values in a *lattice*
- Set of possible values, arranged from least information to most information



Monday, November 9, 15

## Symbolic evaluation

- Evaluate expressions symbolically:  $eval(e, V_{in})$
- If  $e$  evaluates to a constant, return that value. If any input is  $\top$  (or  $\perp$ ), return  $\top$  (or  $\perp$ )
  - Why?
- Two special operations on lattice
  - $meet(a, b)$  – highest value less than or equal to both  $a$  and  $b$
  - $join(a, b)$  – lowest value greater than or equal to both  $a$  and  $b$

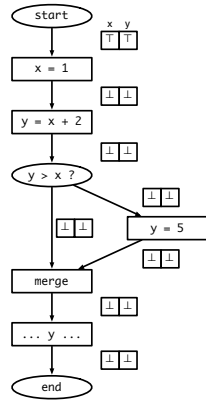


Join often written as a  $\sqcup$  b  
Meet often written as a  $\sqcap$  b

Monday, November 9, 15

## Putting it together

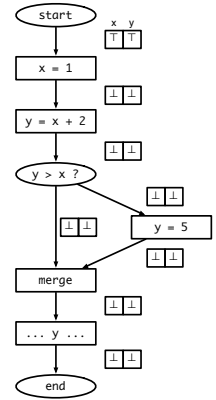
- Keep track of the symbolic value of a variable at every program point (on every CFG edge)
- State vector
- What should our initial value be?
  - Starting state vector is all  $\perp$
  - Can't make any assumptions about inputs – must assume not constant
- Everything else starts as  $\perp$ , since we have no information about the variable at that point



Monday, November 9, 15

## Executing symbolically

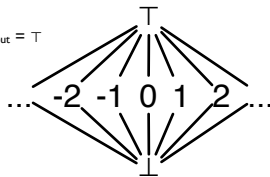
- For each statement  $t = e$  evaluate  $e$  using  $V_{in}$ , update value for  $t$  and propagate state vector to next statement
- What about switches?
  - If  $e$  is true or false, propagate  $V_{in}$  to appropriate branch
  - What if we can't tell?
    - Propagate  $V_{in}$  to both branches, and symbolically execute both sides
- What do we do at merges?



Monday, November 9, 15

## Handling merges

- Have two different  $V_{in}$ s coming from two different paths
- Goal: want new value for  $V_{in}$  to be *safe* (shouldn't generate wrong information), and we don't know which path we actually took
- Consider a single variable. Several situations:
  - $V_1 = \perp, V_2 = * \rightarrow V_{out} = *$
  - $V_1 = \text{constant } x, V_2 = x \rightarrow V_{out} = x$
  - $V_1 = \text{constant } x, V_2 = \text{constant } y \rightarrow V_{out} = \top$
  - $V_1 = \top, V_2 = * \rightarrow V_{out} = \top$
- Generalization:
  - $V_{out} = V_1 \sqcup V_2$



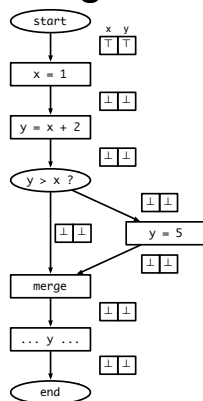
Monday, November 9, 15

## Result: worklist algorithm

- Associate state vector with each edge of CFG, initialize all values to  $\perp$ , worklist has just start edge
- While worklist not empty, do:
  - Process the next edge from worklist
  - Symbolically evaluate target node of edge using input state vector
  - If target node is assignment ( $x = e$ ), propagate  $V_{in}[\text{eval}(e)/x]$  to output edge
  - If target node is branch ( $e$ ?)
    - If  $\text{eval}(e)$  is true or false, propagate  $V_{in}$  to appropriate output edge
    - Else, propagate  $V_{in}$  along both output edges
  - If target node is merge, propagate  $\text{join}(\text{all } V_{in})$  to output edge
  - If any output edge state vector has changed, add it to worklist

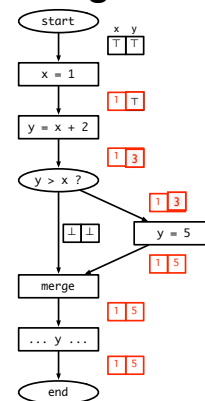
Monday, November 9, 15

## Running example



Monday, November 9, 15

## Running example



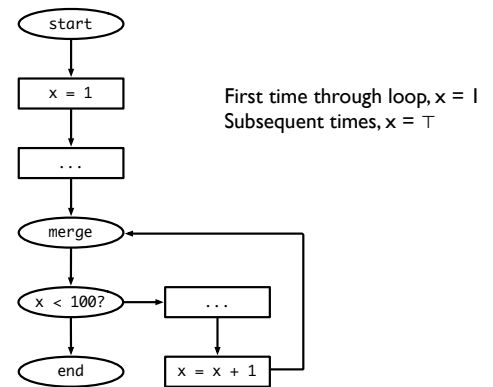
Monday, November 9, 15

## What do we do about loops?

- Unless a loop never executes, symbolic execution looks like it will keep going around to the same nodes over and over again
- Insight: if the input state vector(s) for a node don't change, then its output doesn't change
  - If input stops changing, then we are done!
- Claim: input will eventually stop changing. Why?

Monday, November 9, 15

## Loop example



Monday, November 9, 15

## Complexity of algorithm

- $V = \#$  of variables,  $E = \#$  of edges
- Height of lattice = 2  $\rightarrow$  each state vector can be updated at most  $2 * V$  times.
- So each edge is processed at most  $2 * V$  times, so we process at most  $2 * E * V$  elements in the workload.
- Cost to process a node:  $O(V)$
- Overall, algorithm takes  $O(EV^2)$  time

Monday, November 9, 15

## Question

- Can we generalize this algorithm and use it for more analyses?

Monday, November 9, 15

## Constant propagation

- Step 1: choose lattice (which values are you going to track during symbolic execution?)
  - Use constant lattice
- Step 2: choose direction of dataflow (if executing symbolically, can run program backwards!)
  - Run forward through program
- Step 3: create *transfer functions*
  - How does executing a statement change the symbolic state?
- Step 4: choose *confluence operator*
  - What do do at merges? For constant propagation, use join

Monday, November 9, 15