Dataflow Analysis

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

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
- Loop invariant code motion
  - Need to know where and when variables are live
- So how do we get this information?

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

Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding
    - Create dead code
      - x = 1;
      - y = x + 2;
      - if (x > z) then y = 5
      - ... y ...
      - x = 1;
      - y = x + 2;
      - if (y > x) then y = 5
      - ... y ...

Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding
    - Create dead code
      - x = 1;
      - y = x + 2;
      - if (x > z) then y = 5
      - ... y ...
      - x = 1;
      - y = x + 2;
      - if (y > x) then y = 5
      - ... y ...
**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 ...
  ```

**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)

**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

**Build CFG**

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

**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

**Symbolic evaluation**

- Evaluate expressions symbolically: `eval(e; V_i)`
  - If `e` evaluates to a constant, return that value. If any input is `⊤` (or `⊥`), return `⊤` (or `⊥`)
  - 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`
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 ⊤
  - Can't make any assumptions about inputs – must assume not constant
- Everything else starts as ⊥, since we have no information about the variable at that point

Wednesday, December 3, 14

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?

Wednesday, December 3, 14

Handling merges

- Have two different V_in coming from two different paths
- Goal: want new value for V_out 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 = ⊥, V_2 = * → V_out = *
  - V_1 = constant x, V_2 = x → V_out = x
  - V_1 = constant x, V_2 = constant y → V_out = ⊤
  - V_1 = ⊤, V_2 = * → V_out = ⊤
  - Generalization:
    - V_out = V_1 

Wednesday, December 3, 14

Result: worklist algorithm

- Associate state vector with each edge of CFG, initialize all values to ⊥, 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[eval(e)/x] to output edge
  - If target node is branch (e?)
    - If 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 V_m to output edge
  - If any output edge state vector has changed, add it to worklist

Wednesday, December 3, 14

Running example

Wednesday, December 3, 14

Running example
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?

Loop example

First time through loop, \( x = 1 \)
Subsequent times, \( x = \top \)

Complexity of algorithm

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

Question

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

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