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
- 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
    
    ```
    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 ...
    ```
Example: constant propagation

• Goal: determine when variables take on constant values

• Why? Can enable many optimizations
  • Constant folding

\[ x = 1; \]
\[ y = x + 2; \]
\[ \text{if (}x > z\text{) then } y = 5 \]
\[ \ldots y \ldots \]

\[ x = 1; \]
\[ y = 3; \]
\[ \text{if (}x > z\text{) then } y = 5 \]
\[ \ldots y \ldots \]

• Create dead code

\[ x = 1; \]
\[ y = x + 2; \]
\[ \text{if (}y > x\text{) then } y = 5 \]
\[ \ldots y \ldots \]
Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding
    
    ```
    \begin{align*}
    x &= 1; \\
    y &= x + 2; \\
    \text{if (}x > z\text{) then } y &= 5 \\
    \ldots \ y \ldots
    \end{align*}
    \rightarrow
    \begin{align*}
    x &= 1; \\
    y &= 3; \\
    \text{if (}x > z\text{) then } y &= 5 \\
    \ldots \ y \ldots
    \end{align*}
    ```

  - Create dead code
    
    ```
    \begin{align*}
    x &= 1; \\
    y &= x + 2; \\
    \text{if (}y > x\text{) then } y &= 5 \\
    \ldots \ y \ldots
    \end{align*}
    \rightarrow
    \begin{align*}
    x &= 1; \\
    y &= 3; \quad \text{//dead code} \\
    \text{if (true) then } y &= 5 \quad \text{//simplify!} \\
    \ldots \ y \ldots
    \end{align*}
    ```
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
x = 1;
y = x + 2;
if (y > x) then y = 5;
... y ...

Build CFG
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: 
  \text{eval}(e, V_{in})

• If \( e \) evaluates to a constant, 
  return that value. If any input is \( \top \) (or \( \bot \)), return \( \top \) (or \( \bot \))

• Why?

• Two special operations on lattice
  - \text{meet}(a, b) – highest value less than or equal to both \( a \) and \( b \)
  - \text{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 \( a \sqcap b \)
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 $\top$
  - Can’t make any assumptions about inputs – must assume not constant
- Everything else starts as $\bot$, since we have no information about the variable at that point
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?
Handling merges

- Have two different $V_{\text{in}}$s coming from two different paths
- Goal: want new value for $V_{\text{in}}$ to be safe (shouldn’t generate wrong information), and we don’t know which path we actually took
- Consider a single variable. Several situations:
  - $V_1 = \bot, V_2 = * \rightarrow V_{\text{out}} = *$
  - $V_1 = \text{constant } x, V_2 = x \rightarrow V_{\text{out}} = x$
  - $V_1 = \text{constant } x, V_2 = \text{constant } y \rightarrow V_{\text{out}} = T$
  - $V_1 = T, V_2 = * \rightarrow V_{\text{out}} = T$
- Generalization:
  - $V_{\text{out}} = V_1 \sqcup V_2$
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 \( V_{in}[\text{eval}(e)/x] \) 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 join(all \( V_{in} \)) to output edge
  If any output edge state vector has changed, add it to worklist
Running example

```
x = 1
y = x + 2
y > x ?
```

```
x  y
T  T
```

```
... y ...
```

```
start
x = 1
y = x + 2
y > x ?
```

```
T
```

```
y = 5
```

```
merge
```

```
T
```

```
end
```

Monday, November 9, 15
Running example

\begin{tikzpicture}
  \node (start) [startstop] {start};
  \node (x_eq_1) [process] at (start.150) {$x = 1$};
  \node (y_eq_x_plus_2) [process] at (x_eq_1.210) {$y = x + 2$};
  \node (if_greater) [decision] at (y_eq_x_plus_2.195) {$y > x ?$};
  \node (y_eq_5) [process] at (if_greater.150) {$y = 5$};
  \node (merge) [process] at (y_eq_x_plus_2.290) {... y ...};
  \node (end) [startstop] at (merge.290) {end};

  \draw[->] (start) -- node[above left] {$\top$} (x_eq_1);
  \draw[->] (x_eq_1) -- node [above] {$\top$} (y_eq_x_plus_2);
  \draw[->] (y_eq_x_plus_2) -- (if_greater);
  \draw[->] (if_greater) -- node [above] {$\bot$} (merge);
  \draw[->] (if_greater) -- (y_eq_5);
  \draw[->] (merge) -- (end);
\end{tikzpicture}
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, $x = T$
Complexity of algorithm

- $V = \# \text{ of variables, } E = \# \text{ of edges}$
- Height of lattice = 2 $\rightarrow$ each state vector can be updated at most $2 \times V$ times.
- So each edge is processed at most $2 \times V$ times, so we process at most $2 \times E \times V$ 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