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

```
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

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

- Create dead code

```
x = 1;
y = 3;
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 fairly quickly
  
  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

- Evaluate expressions symbolically: eval(e; \(V_i\))
  
  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
    
    - 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 \(\lor\)

  Meet often written as \(\land\)

Symbolic evaluation

- Idea: replace each value with a symbolic

- constant (specify which), maybe constant, definitely not constant

- Can organize these possible values in a lattice (will formalize this later)
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 don't know if the variable is constant or not 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 \( V_{in} \)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:
  - \( V_1 = !, V_2 = * \rightarrow V_{out} = * \)
  - \( V_1 = \text{constant } x, V_2 = x \rightarrow V_{out} = x \)
  - \( V_1 = \text{constant } x, V_2 = \text{constant } y \rightarrow V_{out} = T \)
  - \( V_1 = T, V_2 = * \rightarrow V_{out} = T \)
- Generalization:
  - \( V_{out} = V_1 \vee V_2 \)

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 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

Wednesday, November 4, 2009
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

```
start
x = 1
...
merge
x < 100?
...           ...  
end
x = x + 1
```

First time through loop, x = 1
Subsequent times, x = 7

Complexity of algorithm

- $V = \# \text{ of variables}, E = \# \text{ of edges}$
- Height of lattice = 2 $\rightarrow$ each state vector can be updated at most $2^V$ times.
- So each edge is processed at most $2^V$ times, so we process at most $2^V E^2 V$ 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?
- First, let’s lay the theoretical foundation for dataflow analysis.

First, something interesting

- **Brouwer Fixpoint Theorem**
  - Every continuous function $f$ from a closed disk into itself has at least one fixed point
  - More formally:
    - Domain $D$: a convex, closed, bounded subspace in a plane (generalizes to higher dimensions)
    - Function $f: D \rightarrow D$
    - There exists some $x$ such that $f(x) = x$
Intuition

- Consider the one-dimensional case: mapping a line segment onto itself
- \( x \in [0, 1] \)
- \( f(x) \in [0, 1] \)
- There must exist some \( x \) for which \( f(x) = x \)
- Examples (in 2D)
  - A mall directory
  - Crumpling up a piece of graph paper

Back to dataflow

- Game plan:
  - Finite partially ordered set with least element \( D \)
  - Function \( f : D \to D \)
  - Monotonic function \( f : D \to D \)
  - \( \exists \) fixpoint of \( f \)
  - \( \exists \) least fixpoint of \( f \)
  - Generalization to case when \( D \) has a greatest element, \( T \)
  - \( \exists \) greatest fixpoint of \( f \)
  - Generalization to systems of equations

Partially ordered set (poset)

- Set \( D \) with a relation \( \preceq \) that is
  - Reflexive: \( x \preceq x \)
  - Anti-symmetric: \( x \preceq y \) and \( y \preceq x \Rightarrow y = x \)
  - Transitive: \( x \preceq y, y \preceq z \Rightarrow x \preceq z \)
- Example: set of integers and \( \preceq \)
- Graphical representation of poset
  - Graph in which nodes are elements of \( D \) and relation \( \preceq \) is indicated by arrows
  - Usually omit reflexive and transitive arrows for legibility
  - Not counting reflexive edges, graph is always a DAG (why?)

Another example

- Powerset of any set, ordered by \( \subseteq \) is a poset
  - In the example, poset elements are \( \{\}, \{a\}, \{a, b\}, \{a, b, c\} \), etc.
  - \( X \subseteq Y \) iff \( X \preceq Y \)

Finite poset with least element

- Poset in which
  - Set is finite
  - There is a least element that is below all other elements in poset
- Examples
  - Set of integers ordered by \( \preceq \) is not a finite poset with least element (no least element, not finite)
  - Set of natural numbers ordered by \( \preceq \) has a least element (0), but not finite
  - Set of factors of 12, ordered by \( \preceq \) has a least element as is finite
  - Powerset example from before is finite (how many elements?) with a least element (\( \{\} \))

Domains

- "Finite poset with least element" is a mouthful, so we will abbreviate this to domain
- Later, we will add additional conditions to domains that are of interest to us in the context of dataflow analysis
  - (Goal: what is a lattice?)
Functions on domains

- If $D$ is a domain, we can define a function $f : D \to D$
- Function maps each element of domain on to another element of the domain
- Example: for $D = \text{powerset of } \{a, b, c\}$
  - $f(x) = x \cup \{a\}$
  - $g(x) = x - \{a\}$
  - $h(x) = \{a\} - x$

Monotonic functions

- A function $f : D \to D$ on a domain $D$ is monotonic if $x \subset y \Rightarrow f(x) \subset f(y)$
- Note: this is not the same as $x \subset f(x)$
- This means that $x$ is extensive
- Intuition: think of $f$ as an electrical circuit mapping input to output
  - If $f$ is monotonic, raising the input voltage raises the output voltage (or keeps it the same)
  - If $f$ is extensive, the output voltage is always the same or more than the input voltage

Examples

- Domain $D$ is the powerset of $\{a, b, c\}$
- Monotonic functions:
  - $f(x) = \{\}$ (why?)
  - $f(x) = x \cup \{a\}$
  - $f(x) = x - \{a\}$
- Not monotonic
  - $f(x) = \{a\} - x$ (why?)
- Extensivity
  - $f(x) = x \cup \{a\}$ is monotonic and extensive
  - $f(x) = x - \{a\}$ is monotonic but not extensive
  - $f(x) = \{a\} - x$ is neither
- What is a function that is extensive, but not monotonic?

Examples

- Domain $D$ is the powerset of $\{a, b, c\}$
- Monotonic functions:
  - $f(x) = \{\}$ (why?)
  - $f(x) = x \cup \{a\}$
  - $f(x) = x - \{a\}$
- Not monotonic
  - $f(x) = \{a\} - x$ (why?)
- Extensivity
  - $f(x) = x \cup \{a\}$ is monotonic and extensive
  - $f(x) = x - \{a\}$ is monotonic but not extensive
  - $f(x) = \{a\} - x$ is neither
- What is a function that is extensive, but not monotonic?

Fixpoints

- Suppose $f : D \to D$.
- A value $x$ is a fixpoint of $f$ if $f(x) = x$
- $f$ maps $x$ to itself
- Examples: $D$ is a powerset of $\{a, b, c\}$
  - Identity function: $f(x) = x$
    - Every element is a fixpoint
  - $f(x) = x \cup \{a\}$
    - Every set that contains $a$ is a fixpoint
  - $f(x) = \{a\} - x$
    - No fixpoints

Fixpoint theorem

- One form of Knaster-Tarski Theorem:
  - If $D$ is a domain and $f : D \to D$ is monotonic, then $f$ has at least one fixpoint
- More interesting consequence:
  - If $\bot$ is the least element of $D$, then $f$ has a least fixpoint, and that fixpoint is the largest element in the chain $\bot, f(\bot), f(f(\bot)), f(f(f(\bot))) \ldots$ $f^*(\bot)$
- Least fixpoint: a fixpoint of $f$, $x$ such that, if $y$ is a fixpoint of $f$ then $x \subset y$
Proof of fixpoint theorem

• First, prove that largest element of chain \( f^n(\perp) \) is a fixpoint

• Second, prove that \( f^n(\perp) \) is the least fixpoint

Solving equations

• If \( D \) is a domain and \( f: D \to D \) is a monotone function on that domain, then the equation \( f(x) = x \) has a least fixpoint, given by the largest element in the sequence \( \perp, f(\perp), f(f(\perp)), f(f(f(\perp))) \ldots \)

• Proof follows directly from fixpoint theorem

Adding a top

• Now let us consider domains with an element \( \top \), such that for every point \( x \) in the domain, \( x \subseteq \top \)

• New theorem: if \( D \) is a domain with a greatest element \( \top \) and \( f: D \to D \) is monotonic, then the equation \( x = f(x) \) has a greatest solution, and that solution is the smallest element in the sequence \( \top, f(\top), f(f(\top)), \ldots \)

• Proof?

Multi-argument functions

• If \( D \) is a domain, a function \( f: D \times D \to D \) is monotonic if it is monotonic in each argument when the other is held constant

• Intuition:
  - Electrical circuit has two inputs
  - If you raise either input while holding the other constant, the output either goes up or stays the same

Fixpoints of multi-arg functions

• Can generalize fixpoint theorem in a straightforward way

• If \( D \) is a domain and \( f, g: D \times D \to D \) are monotonic, the following system of equations has a least fixpoint solution, calculated in the obvious way

  \[ x = f(x, y) \text{ and } y = g(x, y) \]

• Can generalize this to more than two variables and domains with greatest elements easily

Lattices

• A bounded lattice is a partially ordered set with a \( \perp \) and \( \top \), with two special functions for any pair of points \( x \) and \( y \) in the lattice:
  - A join \( x \lor y \) is the least element that is greater than \( x \) and \( y \) (also called the least upper bound)
  - A meet \( x \land y \) is the greatest element that is less than \( x \) and \( y \) (also called the greatest lower bound)

• Are \( \lor \) and \( \land \) monotonic?
More about lattices

- Bounded lattices with a finite number of elements are a special case of domains with $\top$ (why are they not the same?)
- Systems of monotonic functions (including $\lor$ and $\land$) will have fixpoints
- But some lattices are infinite! (example: the lattice for constant propagation)
- It turns out that you can show a monotonic function will have a least fixpoint for any lattice (or domain) of finite height
- Finite height: any totally ordered subset of domain (this is called a chain) must be finite
- Why does this work?

Solving system of equations

- Consider
  
  \[ x = f(x, y, z) \]
  
  \[ y = g(x, y, z) \]
  
  \[ z = h(x, y, z) \]

- Obvious iterative solution: evaluate every function at every step:
  
  \[
  \begin{array}{c}
  x \leftarrow f(\bot, \bot, \bot) \\
  y \leftarrow g(\bot, \bot, \bot) \\
  z \leftarrow h(\bot, \bot, \bot)
  \end{array}
  \]

Worklist algorithm

- Obvious point: only necessary to re-evaluate functions whose inputs have changed

- Worklist algorithm
- Initialize worklist with all equations
- Initialize solution vector $S$ to all $\bot$
- While worklist not empty
  - Get equation from worklist
  - Re-evaluate equation based on $S$, update entry corresponding to lhs in $S$
  - Put all equations which use this lhs on their rhs in the worklist
- Claim: the worklist algorithm for constant propagation is an instance of this approach

Mapping worklist algorithm

- Careful: the “variables” in constant propagation are not the individual variable values in a state vector. Each variable (from a fixpoint perspective) is an entire state vector – there are as many variables as there are edges in the CFG
- Functions:
  - Program statements: $\text{eval}(e, V_e)$
  - Need to make sure this is monotonic
  - Branches
  - Merges
    - Use join or meet to combine multiple input variables – monotonic by definition

Constant propagation

- Step 1: choose lattice
  - Use constant lattice (infinite, but finite height)
- Step 2: choose direction of dataflow
  - Run forward through program
- Step 3: create monotonic transfer functions
  - If input goes from $\bot$ to constant, output can only go up. If input goes from constant to $\top$, output goes to $\top$
- Step 4: choose confluence operator
  - What do do at merges? For constant propagation, use join