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

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

- Create dead code

\[
\begin{align*}
x &= 1; \\
y &= x + 2; \\
\text{if} \ (y > x) \ \text{then} \ y &= 5 \\
\ldots \ y \ldots
\end{align*}
\]
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

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

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

  • Create dead code

    \[
    \begin{align*}
    x &= 1; \\
    y &= x + 2; \\
    \text{if} \ (y > x) \ \text{then} \ y &= 5 \\
    \ldots \ y \ \ldots
    \end{align*}
    \]

    \[
    \begin{align*}
    x &= 1; \\
    y &= 3; \ //\text{dead code} \\
    \text{if} \ (\text{true}) \ \text{then} \ y &= 5 \ //\text{simplify!} \\
    \ldots \ y \ \ldots
    \end{align*}
    \]
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
Build CFG

\[
x = 1; \\
y = x + 2; \\
if (y > x) \text{ then } y = 5; \\
... y ... \\
\]
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)
Symbolic evaluation

- Evaluate expressions symbolically: `eval(e, V_{in})`
  - 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`

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 don’t know if the variable is constant or not at that point

\[
\begin{align*}
\text{start} & \quad x \quad y \\
x = 1 & \quad \bot \quad \bot \\
y = x + 2 & \quad \bot \quad \bot \\
y > x ? & \quad \bot \quad \bot \quad \bot \\
y = 5 & \quad \bot \quad \bot \quad \bot \\
\text{merge} & \quad \bot \quad \bot \quad \bot \\
\ldots y \ldots & \quad \bot \quad \bot \quad \bot \\
\text{end} & \quad \bot \quad \bot \quad \bot
\end{align*}
\]
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:
  - $V_1 = \perp, V_2 = * \rightarrow V_{\text{out}} = *$
  - $V_1 = \text{constant } x, V_2 = x \rightarrow V_{\text{out}} = x$
  - $V_1 = \text{constant } x, V_2 = \text{constant } y \rightarrow V_{\text{out}} = T$
  - $V_1 = T, V_2 = * \rightarrow V_{\text{out}} = T$
- 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 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

start

\[ x = 1 \]

\[ y = x + 2 \]

\[ y > x? \]

\[ y = 5 \]

merge

... y ...

down

down

down

down

down

down

down

down

down

down

down

down

down

down

end
Running example

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

```
y = 5
... y ...
end
```
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

- \( V = \# \text{ of variables}, \ E = \# \text{ of edges} \)
- 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?
- First, let’s lay the theoretical foundation for dataflow analysis.
Lattice Theory
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 $f : D \rightarrow D$
  - Monotonic function $f : D \rightarrow D$
  - $\exists$ fixpoint of $f$
    - $\exists$ least fixpoint of $f$
  - Generalization to case when $D$ has a greatest element, $\top$
    - $\exists$ greatest fixpoint of $f$
  - Generalization to systems of equations
Partially ordered set (poset)

- Set $D$ with a relation $\sqsubseteq$ that is
  - Reflexive: $x \sqsubseteq x$
  - Anti-symmetric: $x \sqsubseteq y$ and $y \sqsubseteq x \Rightarrow y = x$
  - Transitive: $x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- Example: set of integers and $\leq$
- Graphical representation of poset
  - Graph in which nodes are elements of $D$ and relation $\sqsubseteq$ 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 \( \{\}, \{a\}, \{a, b\}, \{a, b, c\}, \text{etc.} \)
- \( X \subseteq Y \) iff \( X \subseteq Y \)
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 $\leq$ is not a finite poset with least element (no least element, not finite)
  - Set of natural numbers ordered by $\leq$ has a least element (0), but not finite
  - Set of factors of 12, ordered by $\leq$ 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 $D =$ powerset of $\{a, b, c\}$
  • $f(x) = x \cup \{a\}$
  • $g(x) = x - \{a\}$
  • $h(x) = \{a\} - x$
Monotonic functions

• A function \( f : D \rightarrow D \) on a domain \( D \) is monotonic if

• \( x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \)

• Note: this is not the same as \( x \sqsubseteq f(x) \)

• 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?
Fixpoints

• Suppose $f : D \rightarrow D$.
  • 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

  $\bot, f(\bot), f(f(\bot)), f(f(f(\bot))), \ldots f^n(\bot)$

• Least fixpoint: a fixpoint of $f$, $x$ such that, if $y$ is a fixpoint of $f$, then $x \sqsubseteq y$
Examples

- For domain of powersets, \{ \} is the least element
- For identity function, \( f^n(\{ \}) \) is the chain
  \{ \}, \{ \}, \{ \}, ... so least fixpoint is \{ \}, which is correct
- For \( f(x) = x \cup \{a\} \), we get the chain
  \{ \}, \{a\}, \{a\}, ... so least fixpoint is \{a\}, which is correct
- For \( f(x) = \{a\} - x \), function is not monotonic, so not guaranteed to have a fixpoint!
- Important observation: as soon as the chain repeats, we have found the fixpoint (why?)
Proof of fixpoint theorem

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

• Second, prove that $f^n(\bot)$ is the least fixpoint
Solving equations

• If \( D \) is a domain and \( f : D \rightarrow D \) 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

\[ \bot, f(\bot), f(f(\bot)), f(f(f(\bot))), \ldots \]

• 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, $x \sqsubseteq \top$

• 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

  $\top, f(\top), f(f(\top)), ...$

• Proof?
Multi-argument functions

- If $D$ is a domain, a function $f : D \times D \rightarrow D$ 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 \( \bot \) and \( \top \), with two special functions for any pair of points \( x \) and \( y \) in the lattice:

  • A join: \( x \sqcup y \) is the least element that is greater than \( x \) and \( y \) (also called the least upper bound)

  • A meet: \( x \sqcap y \) is the greatest element that is less than \( x \) and \( y \) (also called the greatest lower bound)

• Are \( \sqcup \) and \( \sqcap \) 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 $\sqcap$ and $\sqcup$) 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:

\[ \bot \quad f(\bot, \bot, \bot) \quad \ldots \]
\[ \bot \quad g(\bot, \bot, \bot) \quad \ldots \]
\[ \bot \quad h(\bot, \bot, \bot) \quad \ldots \]
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: eval(e, V_{in})
    - These are called *transfer functions*
  - Need to make sure this is monotonic

- Branches
  - Propagates input state vector to output – trivially monotonic

- 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