Announcements

- Homework is due Monday
  - Problem 6(b) is Extra credit
  - Submit in Submitty
  - You can lock your team later. Maximal size is 3. Ideal size is 2.

- Quiz 1 today

Outline of Today’s Class

- Dataflow frameworks, conclusion
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if
  - its property space $L$, $\leq$ that satisfies the Ascending Chain Condition
  - its merge operator $V$ is the join of $L$
  - its transfer function space $F : L \rightarrow L$ is monotone

Thus, we can make use of a generic solution procedure, known as the worklist algorithm or the maximal fixpoint algorithm or the fixpoint iteration algorithm

Catch up: Transfer Functions

- The transfer functions: $f : L \rightarrow L$. Formally, function space $F$ is such that
  1. $F$ contains all $f_j$
  2. $F$ contains the identity function $id(x) = x$
  3. $F$ is closed under composition
  4. Each $f$ is monotone

Catch up: Monotonicity

- $F : L \rightarrow L$ is monotone if and only if:
  1. $a, b$ in $L$, $f$ in $F$ then $a \leq b \iff f(a) \leq f(b)$
  or (equivalently):
  2. $x, y$ in $L$, $f$ in $F$ then $f(x) V f(y) \leq f(x V y)$

Theorem: Definitions (1) and (2) are equivalent.
- Show that (1) implies (2)
- Show that (2) implies (1)
Catch up: Distributivity

- **F: L \rightarrow L** is distributive if and only if for all \( x, y \) in \( L \), \( f(x \lor y) = f(x) \lor f(y) \)
- Every distributive function is also monotone but not the other way around
- Distributivity is a very nice property!

Catch up: Monotonicity and Distributivity

- Is classical Reach distributive?
  - Yes
  - To show distributivity:
    For each \( j \):
    \[
    ((X \lor Y) \land \text{pres}(j)) \lor \text{gen}(j) = (X \land \text{pres}(j)) \lor \text{gen}(j) \]
    \[
    (X \land \text{pres}(j)) \lor \text{gen}(j) = (Y \land \text{pres}(j)) \lor \text{gen}(j) \]

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if:
  - Its property space is a lattice \( L \) that satisfies the **Ascending Chain Condition**
  - Its merge operator \( V \) is the join of \( L \)
  - Its transfer function space \( F: L \rightarrow L \) is monotone
- Thus, we can make use of a generic solution procedure, known as the worklist algorithm or the maximal fixpoint algorithm or the fixpoint iteration algorithm

Worklist Algorithm for Forward Dataflow Problems

```c
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f_1(in(1))
for m = 2 to n do in(m) = 0; out(m) = f_m(0)
W = \{ 2, ..., n \} /* put every node but 1 on the worklist */
while W ≠ Ø do {
    remove j from W
    in(j) = V \{ out(i) | i is predecessor of j \}
    out(j) = f_j(in(j))
    if out(j) changed then
        W = W U \{ k | k is successor of j \}
}
```

Worklist Algorithm for Forward Dataflow Problems (slightly different)

```c
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f_1(in(1))
for m = 2 to n do in(m) = 0; out(m) = f_m(0)
W = \{ 2, ..., n \} /* put every node but 1 on the worklist */
while W ≠ Ø do {
    remove j from W
    in(j) = V \{ out(i) | i is predecessor of j \}
    out(j) = f_j(in(j))
    if out(j) changed then
        W = W U \{ k | k is successor of j \}
}
```

Termination Argument

- Why does the algorithm terminate?
  - Sketch of argument:
    At each “phase”, at least one \( \text{out}(j) \) changes. Monotonicity of \( f \) entails that change of is up the chain: \( \text{out}(j) \geq \text{out}(k) \). Since \( \text{out}(j) \) in \( L \) and \( L \) satisfies the **Ascending Chain Condition**, \( \text{out}(j) \) changes at most \( O(h) \) times where \( h \) is the height of the lattice \( L \).
Correctness Argument

- **Theorem:** The worklist algorithm computes a solution that satisfies the dataflow equations
  
  **Why?**
  
  **Sketch of argument:**
  
  Whenever $j$ is processed, algorithms sets $\text{out}(j) = f_j(\text{in}(j))$. Whenever $\text{out}(j)$ changes, algorithm puts successors on the list, so $\text{in}(j) = V\{\text{out}(l)\}$. So final solution will satisfy equations.

Precision Argument

- **Theorem:** The algorithm computes the least solution of the dataflow equations.
  
  Historically though, this solution is called the maximal fixpoint solution (MFP)
  
  I.e., For every node $j$, the worklist algorithm computes a solution $MFP(j) = \{\text{in}(j), \text{out}(j)\}$, such that every other solution $\{\text{in}(j), \text{out}(j)\}$ of the dataflow equations is $\text{in}(j) \leq \text{in}'(j)$, $\text{out}(j) \leq \text{out}'(j)$

Example

1. $z := x + y$
2. if ($z > 500$)
3. skip

$\text{in}(Avail(2)) = \text{out}(Avail(1)) 
\text{V} \text{out}(Avail(3))$

$\text{in}(Avail(3)) = \text{out}(Avail(2))$

$\text{in}(Avail(1)) = \emptyset$

$\text{out}(Avail(2)) = \text{in}(Avail(2))$

$\text{out}(Avail(3)) = \text{in}(Avail(3))$

$\text{out}(Avail(1)) = (\text{in}(Avail(1)) - E_z) \{x+y\}$

Equivalent to: $\text{in}(Avail(2)) = \{x+y\} \text{ V } \text{in}(Avail(2))$ and recall that $V$ is $\cap$ (i.e., set intersection).

Question

Willy Wazoo changed the worklist algorithm to initialize to 1.

```plaintext
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;
in(Reach(1)) = UNDEF
for m = 2 to n do
  in(m) = 1
  in(Reach(m)) = ALL_DEFS
W = {1,2,3,...,n} /* put every node on the worklist */
while W $\neq \emptyset$ do {
  remove j from W
  out(j) = $f_j(\text{in}(j))$
  out(Reach(j)) = (\text{in}(Reach(j)) \cap \text{pres}(j)) \cup \text{gen}(j)$
  for i in successors(j)
    if out(j) $\leq$ in(i) then {
      if out(Reach(i)) $\neq \text{in}(i)$ then {
        in(Reach(i)) = out(Reach(i)) \cup \text{in}(Reach(i))
      } else {
        in(Reach(i)) = \text{out}(Reach(i)) \cup \text{in}(Reach(i))
      }
      W = W \cup \{ i \}
    }
}
```

1. Does Willy’s algorithm compute an over-approximation or an under-approximation of the MFP?

Meet Over All Paths (MOP)

- Desired dataflow information at $n$ is obtained by traversing ALL PATHS from 1 (entry node) to $n$.
  For every path $p = \{1, n_2, n_3, ..., n_k\}$ we compute $f_{n_2}(...f_{n_2}(f_1(\text{init}(1))))$

- The MOP at entry of $n$ is $V f_{n_2}(...f_{n_2}(f_1(\text{init}(1))))$ over all paths $p$ from 1 to $n$

MOP vs. MFP

- The MOP is an abstract model for the best solution computable with this kind of static analysis
  
  It is a common assumption in this kind of static analysis that all program paths are executable
  
  (Abstract interpretation and axiomatic semantics abstract state more precisely, and can rule out some paths)

- The MFP is the solution computed by the worklist algorithm
MOP vs. MFP

- For **distributive** problems MFP = MOP!

- Unfortunately, for **monotone** problems this is not true. But we still have a **safe** solution: it is a theorem that for monotone problems, MFP ≥ MOP

Safety of a Dataflow Solution

- A safe (also, correct or sound) solution X overestimates the “best” possible dataflow solution, i.e., X ≥ MOP

- Historically, an **acceptable** solution X is one that is better than what we can do with the MFP, i.e., X ≤ MFP

**Safe Solutions**

In **may problems** 1 is the universal set of facts, the merge operator is set union. It is **safe** to err by saying that a fact reaches a node when in fact it doesn’t

- E.g., intuitively, it is **safe** to err by saying that a definition \((x,k)\) reaches a node, when in fact it MAY NOT REACH that node

- **Safe** entails “larger” then the MOP under our partial order. Our definition of ≤ is subset inclusion (which is natural). So “safer” solutions are larger sets

**Safe Solutions: Reach**

\(U = \{ (x,1), (x,4), (a,3) \} \)  \(\{ (x,1), (x,4), (a,3) \} \)

| 1. \(x = a \cdot b\) |
| 2. \(if \ y \leq a \cdot b\) |
| 3. \(a = a+1\) |
| 4. \(x = a \cdot b\) |
| 5. goto 3 |

**Safe Solutions: Avail**

\(U = \{ a \cdot b, a+1, y \cdot z \} \)  \(\{ a \cdot b, a+1, y \cdot z \} \)

| 1. \(x = a \cdot b\) |
| 2. \(if \ y \cdot z \leq a \cdot b\) |
| 3. \(a = a+1\) |
| 4. \(x = a \cdot b\) |
| 5. goto 2 |
**Precision of a Dataflow Solution**

- **Precise** solution is one that is "close" to MOP
  - A precise solution contains few spurious dataflow facts (spurious facts is what we call noise)
  - Unfortunately, for most problems even the MOP (an approximation itself) is undecidable

- MOP ≤ X ≤ Y: X is more precise than Y
  - Usually we can compare two solutions X and Y
  - But, for most problems, we have no way of knowing the "ground truth"

**Outline of Today’s Class**

- Dataflow frameworks
- MOP vs. MPF
- Non-distributive analyses
  - Constant propagation

**Constant Propagation (Simple)**

- Problem statement: What variables always hold constant values at a given program point

- Example:

```
1. x = 1
   if (b>0)
   in(1): x is not const
   out(1): x is 1

2. y = z + w
   x = 2
   in(2): x is 2
   out(2): x is z
   out(3): x is 1

3. y = 0
   in(3): x is 1
   out(3): x is 1

4. z = 10*x
   in(4): x is NOT a const!
   out(4): x is 1
   out(2): x is 2
   out(3): x is 1
   out(4): x is 1
```

**Aside: Defining an Analysis**

- Define program syntax
  - In practice, we deal with a lot more than the simple abstraction 😊
- Define property space
  - The abstract program state that approximates the concrete program state
- Define transfer function space over syntax
  - Symbolically execute program over abstract state

**Aside: Defining an Analysis**

- If property space has desired properties
  - is a lattice L, ≤ that satisfies the Ascending Chain Condition
  - merge operator V is the join of L
  - Function space F: L → L is monotone
  - then analysis fits the monotone dataflow framework and can be solved using the worklist algorithm

**Constant Propagation: Syntax**

\[
S ::= S; S | \text{while (b) } \{ S \} | \text{if (b) } \{ S \} \text{ else } \{ S \}
\]

\[
| x = V | x = V \text{ Op } V
\]

\[
Op ::= + | - | * | /
\]

\[
V ::= x | y | z | C
\]

- x, y, z are program variables
- C is constant
- We have to define transfer functions for x = V and x = V Op V
Constant Propagation: Property Space

- Associate one of the following values with variable \( x \) at each program point

<table>
<thead>
<tr>
<th>value</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (or ( T ))</td>
<td>( x ) is NOT a constant</td>
</tr>
<tr>
<td>( c )</td>
<td>( x ) has constant value ( c )</td>
</tr>
<tr>
<td>0 (or ( \perp ))</td>
<td>( x ) is unknown</td>
</tr>
</tbody>
</table>

Constant Propagation: Lattice

- Lattice \( L_x \leq \)

- Dataflow lattice \( L \) is the product of \( L_x \)
  - \( l_1, l_2 \) in \( L \), \( l_1 \leq l_2 \) if \( l_1, l_2 \) for every variable \( x \)
  - \( l_1 \lor l_2 \) amounts to \( l_1_x \lor l_2_x \) for every variable \( x \)
  - Merge operator is join of \( L \)
  - Does the product lattice satisfy the ACC?

Constant Propagation: Transfer Functions

- \( j \): \( x = c \)
  - \( f_j \): kill \( x \to \text{val} \), generate \( x \to c \)
- \( j \): \( x = y \)
  - \( f_j \): kill \( x \to \text{val} \), add \( x \to \text{val'} \), s.t. \( y \to \text{val'} \) in \( \text{in}(j) \)
- \( \text{val} \) and \( \text{val'} \) are one of
  - \( \perp \): bottom (unknown)
  - \( c \): constant
  - \( T \): top (not a constant)

Example

1. if (\( b > 0 \))
2. \( x = 1 \)
   \( y = 2 \)
3. \( x = 2 \)
   \( y = 1 \)
4. \( z = x + y \)
5. \( w = 10 * z \)

Not Distributive! A Counter Example

1. if (\( b > 0 \))
2. \( x = 1 \)
   \( y = 2 \)
3. \( x = 2 \)
   \( y = 1 \)
4. \( z = x + y \)
5. \( w = 10 * z \)

MFP at 5 gave us \( z \to T \)
(i.e., \( z \) is NOT a const)
Next Class

- We’ll continue with non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Introduction to Soot