Lecture 6
Foundations of Data Flow Analysis

I. Meet Operator
II. Transfer Functions
III. Correctness, Precision, Convergence
IV. Efficiency

• Reference: ALSU pp. 613-631
• Background: Hecht and Ullman, Kildall, Allen and Cocke[76]

Review: Reaching Definitions

<table>
<thead>
<tr>
<th>Block</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{1,2}</td>
<td>{3,4,5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Review: Live Variable Analysis

• A variable \( v \) is **live** at point \( p \) if
  - the value of \( v \) is used along some path in the flow graph starting at \( p \).
• A basic block \( b \) can
  - generate live variables: Use[\( b \)]
    - set of locally exposed uses in \( b \)
  - propagate incoming live variables: OUT[\( b \)] \( \cup \) Def[\( b \)]
    - where Def[\( b \)] = set of variables defined in \( b \).
• Backward analysis
  - transfer function for block \( b \):
    \[ \text{in}[b] = \text{Use}[b] \cup \text{out}(b) \cup \text{Def}[b] \]
• meet operator:
  \[ \text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \cdots \cup \text{in}[s_n], \]
  where \( s_1, \ldots, s_n \) are all successors of \( b \)

Review: Data Flow Analysis Framework

<table>
<thead>
<tr>
<th>Domain</th>
<th>Sets of definitions</th>
<th>Sets of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>out[( b )] = Gen[( b )] ( \cup ) in[( b )] ( \land) pred[( b )]</td>
<td>in[( b )] = out[( b )] ( \cup ) in[( b )] ( \land) succ[( b )]</td>
</tr>
<tr>
<td>Backward</td>
<td>in[( b )] = Gen[( b )] ( \cup ) out[( b )]</td>
<td>out[( b )] = in[( b )] ( \cup ) succ[( b )]</td>
</tr>
<tr>
<td>Transfer function</td>
<td>Gen[( b )] ( \cup ) in[( b )] ( \land) pred[( b )]</td>
<td>out[( b )] = in[( b )] ( \cup ) succ[( b )]</td>
</tr>
<tr>
<td>Meet Operation</td>
<td>( \cup )</td>
<td>( \cup )</td>
</tr>
<tr>
<td>Boundary Condition</td>
<td>out[entry] = ( \emptyset )</td>
<td>in[exit] = ( \emptyset )</td>
</tr>
<tr>
<td>Initial interior points</td>
<td>out[( b )</td>
<td>in[( b )</td>
</tr>
</tbody>
</table>
A Unified Framework

- Data flow problems are defined by
  - Domain of values: \( V \)
  - Meet operator: \( (V, \wedge, \top, \bot) \), initial value
  - A set of transfer functions: \( V \rightarrow V \)

- Usefulness of unified framework
  - To answer questions such as correctness, precision, convergence, speed of convergence for a family of problems
    - If meet operators and transfer functions have properties \( X \), then we know \( Y \) about the above.
  - Reuse code

Overview: A Check List for Data Flow Problems

- Semi-lattice
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?

- Transfer functions
  - function of each basic block
  - monotone
  - distributive?

- Algorithm
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph

I. Meet Operator

- Properties of the meet operator
  - commutative: \( x \wedge y = y \wedge x \)
  - idempotent: \( x \wedge x = x \)
  - associative: \( x \wedge (y \wedge z) = (x \wedge y) \wedge z \)
  - there is a Top element \( T \) such that \( x \wedge T = x \)

- Meet operator defines a partial ordering on values
  - \( x \preceq y \) if and only if \( x = y \) \( \text{[y \rightarrow x in diagram]} \)
    - Transitivity: if \( x \preceq y \) and \( y \preceq z \) then \( x \preceq z \)
    - Antisymmetry: if \( x \preceq y \) and \( y \preceq x \) then \( x = y \)
    - Reflexivity: \( x \preceq x \)

- Partial Order

  \[ \begin{array}{c}
  T = (1,1) \\
  (1,0) \\
  (0,1) \\
  (0,0) \\
  \end{array} \]

Meet Operator: Elementwise-min

Meet Operator: Intersection

Meet Operator: Union

- Top and Bottom elements
  - Top \( T \) such that: \( x \wedge T = x \)
  - Bottom \( \bot \) such that: \( x \wedge \bot = \bot \)

- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a \( T \), but not necessarily a \( \bot \)
  - \( x, y \) are ordered: \( x \preceq y \) \( \text{[y \rightarrow x in diagram]} \)
  - what if \( x \) and \( y \) are not ordered?
    - \( x \wedge y \leq x, x \wedge y \leq y, \) and if \( w \leq x, w \leq y \), then \( w \leq x \wedge y \)
One vs. All Variables/Definitions

- Lattice for each variable: e.g. intersection

• Lattice for three variables:

```
+----------------+----------------+----------------+
|                 |      |                 |
|                 |      |                 |
|                 |      |                 |
|                 |      |                 |
+----------------+----------------+----------------+
```

III. Transfer Functions

- Basic Properties \( f : V \rightarrow V \)
  - Has an identity function
    • There exists an \( f \) such that \( f(x) = x \), for all \( x \).
  - Closed under composition
    • if \( f_1, f_2 \in F \), then \( f_1 \circ f_2 \in F \)

 out[b] = Gen[b] U (in(b)-Kill[b])

Monotonicity

- A framework \( \langle F, V, \circ \rangle \) is monotone if and only if
  • \( x \leq y \) implies \( f(x) \leq f(y) \)
    • i.e. a “smaller or equal” input to the same function will always give a “smaller or equal” output
  - Equivalently, a framework \( \langle F, V, \circ \rangle \) is monotone if and only if
    • \( f(x \circ y) \leq f(x) \circ f(y) \)
    • i.e. merge input, then apply \( f \) is small than or equal to apply the transfer function individually and then merge the result

```
out[b] = Gen[b] U (in(b)-Kill[b])
```

Descending Chain

- Definition
  • The height of a lattice is the largest number of \( > \) relations that will fit in a descending chain.

\[ x_0 > x_1 > x_2 > \ldots \]

- Height of values in reaching definitions?
  Height-\( n \), where \( n \) is the number of definitions

- Important property: finite descending chain
  • Can an infinite lattice have a finite descending chain? yes
  • Example: Constant Propagation/Folding
    • To determine if a variable is a constant
    - Data values
      • undef, ... -1, 0, 1, 2, ..., not-a-constant
Example

- Reaching definitions: \( f(x) = \text{Gen} \cup (x \cdot \text{Kill}) \land \) \( \bigwedge \)
  - Definition 1:
    - \( x_1 \leq x_2, \text{Gen} \cup (x_1 \cdot \text{Kill}) \leq \text{Gen} \cup (x_2 \cdot \text{Kill}) \)
  - Definition 2:
    - \( (\text{Gen} \cup (x_1 \cdot \text{Kill})) \lor (\text{Gen} \cup (x_2 \cdot \text{Kill})) = (\text{Gen} \cup ((x_1 \cup x_2) \cdot \text{Kill})) \)

- Note: Monotone framework does not mean that \( f(x) \leq x \)
  - e.g., reaching definition for two definitions in program
  - suppose: \( f_{x_1}: \text{Gen} \cdot x_1 = \{d_1, d_2\} ; \text{Kill} \cdot x_1 = {} \)
    - then \( f(x) = \{d_1, d_2\} \) for any \( x \), including \( x = {} \)

- If input(second iteration) \( \leq \) input(first iteration)
  - result(second iteration) \( \leq \) result(first iteration)

Meet-Over-Paths (MOP)

- Err in the conservative direction
- Meet-Over-Paths (MOP):
  - For each node \( n \):
    - \( \text{MOP}(n) = \bigwedge \mathcal{P}_n(T) \), for all paths \( \mathcal{P}_n \) reaching \( n \)
  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - \( \text{MOP} = \text{Perfect-Solution} \land \text{Solution-to-Unexecuted-Paths} \)
  - \( \text{MOP} \leq \text{Perfect-Solution} \)
  - Potentially more constrained, solution is small
    - hence conservative
  - It is not safe to be > Perfect-Solution!
  - Desirable solution: as close to MOP as possible

Distributivity

- A framework \( (\mathcal{P}, \mathcal{V}, \land) \) is distributive if and only if
  - \( f(x \land y) = f(x) \land f(y) \)
    - i.e. merge input, then apply \( f \) is equal to apply the transfer function individually then merge result
  - Example: Constant Propagation is NOT distributive

III. Data Flow Analysis

- Definition
  - Let \( f_0, ..., f_n : \mathcal{P} \) where \( f_i \) is the transfer function for node \( i \)
    - \( f_0 = f_{a_0} \leftarrow f_{a_0} \), where \( p \) is a path through nodes \( n_0, ..., n_k \)
    - \( f_0 \) = identify function, if \( p \) is an empty path
  - Ideal data flow answer:
    - For each node \( n \):
      - \( \bigwedge f_{a_0}(T) \), for all possibly executed paths \( \mathcal{P}_n \) reaching \( n \).

- But: Determining all possibly executed paths is undecidable
**Example: MOP considers more paths than Ideal**

Ideal: Considers only 2 paths
- B1-B2-B4-B6-B7 (i.e., x=1)
- B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths
- B1-B2-B4-B5-B7
- B1-B3-B4-B6-B7

Assume: B2 & B3 do not update x

---

**Solving Data Flow Equations**

- **Example: Reaching definitions**
  - out[entry] = {}
  - Values = (subsets of definitions)
  - Meet operator: \( \cup \)
    - \( in(b) = \bigcup out(p) \) for all predecessors \( p \) of \( b \)
  - Transfer functions: \( out(b) = gen_b \cup \bigcup \{ in(b) \setminus kill_b \} \)

- Any solution satisfying equations = **Fixed Point Solution (FP)**

- **Iterative algorithm**
  - initializtes out[b] to {}
  - if converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations

- **Properties**:
  - FP \( \leq \) MFP \( \leq \) MOP \( \leq \) Perfect-solution
  - FR, MFP are safe
  - in(b) \( \leq \) MOP(b)

---

**Partial Correctness of Algorithm**

- If data flow framework is **monotone** (i.e., \( x \leq y \) implies \( f(x) \leq f(y) \))
  - then if the algorithm converges, \( IN[b] \leq MOP[b] \)

- **Proof**: Induction on path lengths
  - Define \( IN[entry] = OUT[entry] \)
  - and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of \( IN[entry] \)
  - If true for path of length \( k \), \( p_k = (n_1, \ldots, n_k) \), then true for path of length \( k+1 \): \( p_k+1 = (n_1, \ldots, n_{k+1}) \)
    - Assume: \( IN[n_k] \leq f_{n_k}(IN[n_k] \setminus \cdots \setminus f_{n_1}(IN[entry])) \)
    - \( IN[n_{k+1}] = OUT[n_k] \land \cdots \leq OUT[n_k] = f_{n_k}(IN[n_k]) \)
    - \( \leq f_{n_k}(IN[n_k] \setminus \cdots \setminus f_{n_1}(IN[entry])) = \text{by inductive assumption & monotonicity} \)

---

**Precision**

- If data flow framework is **distributive** (i.e., \( f(x \land y) = f(x) \land f(y) \))
  - then if the algorithm converges, \( IN[b] = MOP[b] \)

- Monotone but not distributive: behaves as if there are additional paths

```
\begin{align*}
a &= 2 \\
b &= 3 \\
c &= a + b &= 5
\end{align*}
```
Additional Property to Guarantee Convergence

- Data flow framework (monotone) converges if there is a finite descending chain

- For each variable \( \text{IN}[b] \), \( \text{OUT}[b] \), consider the sequence of values set to each variable across iterations:
  - if sequence for \( \text{IN}[b] \) is monotonically decreasing
    - sequence for \( \text{OUT}[b] \) is monotonically decreasing
    - \( \text{OUT}[b] \) initialized to \( T \)
  - if sequence for \( \text{OUT}[b] \) is monotonically decreasing
    - sequence of \( \text{IN}[b] \) is monotonically decreasing

IV. Speed of Convergence

- Speed of convergence depends on order of node visits
- Reverse “direction” for backward flow problems

Reverse Postorder

- **Step 1: depth-first post order**
  ```
  main() {
    count = 1;
    Visit(root);
  }
  Visit(n) {
    for each successor s that has not been visited
      Visit(s);
    PostOrder(n) = count;
    count = count+1;
  }
  ```

- **Step 2: reverse order**
  ```
  rPostOrder(i) = NumNodes - PostOrder(i)
  ```

Depth-First Iterative Algorithm (forward)

**Input**: control flow graph \( \text{CFG} = (N, E, \text{Entry}, \text{Exit}) \)

/* Initialize */
- \( \text{OUT}[\text{Entry}] = \text{init_value} \)
- For all nodes \( i \)
  - \( \text{OUT}[i] = T \)
  - \( \text{Change} = \text{True} \)

/* iterate */
- While \( \text{Change} \) is not false
  - For each node \( i \) in \( rPostOrder \)
    - \( \text{in}[i] = \land (\text{OUT}[p]) \), for all predecessors \( p \) of \( i \)
    - \( \text{oldout} = \text{OUT}[i] \)
    - \( \text{OUT}[i] = f_i(\text{IN}[i]) \)
    - \( \text{if} \) \( \text{oldout} \neq \text{OUT}[i] \)
      - \( \text{Change} = \text{True} \)

```
Speed of Convergence

- If cycles do not add information
- Information can flow in one pass down a series of nodes of increasing order number:
  - \(1 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 2 \rightarrow 4\) ...
  - First pass
- Passes determined by number of back edges in the path
  - Essentially the nesting depth of the graph
- Number of iterations = number of back edges in any acyclic path + 2
  - (2 are necessary even if there are no cycles)
  - (2 not 1 since need a last pass where nothing changed)
- What is the depth?
  - Corresponds to depth of intervals for “reducible” graphs
  - In real programs: average of 2.75

Summary: A Check List for Data Flow Problems

- Semi-lattice
  - Set of values
  - Meet operator
  - Top, bottom
  - Finite descending chain?
- Transfer functions
  - Function of each basic block
  - Monotone
  - Distributive?
- Algorithm
  - Initialization step (entry/exit, other nodes)
  - Visit order: \(r\text{PostOrder}\)
  - Depth of the graph

Friday’s Class

- Global common subexpression elimination
- Constant propagation/folding