Lecture 5
Foundations of Data Flow Analysis

I. Meet operator
II. Transfer functions
III. Correctness, Precision, Convergence
IV. Efficiency

Reference: Muchnick 8.2-8.5
Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
Marlowe&Ryder, Properties of data flow frameworks: a unified model
Rutgers tech report, Apr. 1988

A Unified Framework

- Data flow problems are defined by
  - Domain of values: V
  - Meet operator (V x V -> V), initial value
  - A set of transfer functions (V -> 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

Partial Order

- Example: let V = {x | such that x ⊆ {d1, d2, d3}}, ∧ = ∩

- Top and Bottom elements
  - Top ⊤ such that x ∧ ⊤ = x
  - Bottom ⊥ such that x ∧ ⊥ = ⊥

- Values and meet operator in a data flow problem define a semi-lattice: there exists a ⊤, but not necessarily a ⊥.

- x, y are ordered: x ≤ y then x ∧ y = x
- what if x and y are not ordered?
  - x ∧ y ≤ x, x ∧ y ≤ y, and if w ≤ x, w ≤ y, then w ≤ x ∧ y
- ...
One vs. All Variables/Definitions

- Lattice for each variable: e.g. intersection

- Lattice for three variables:

Descending Chain

- Definition
  - The height of a lattice is the largest number of > relations that will fit in a descending chain.
    \[ x_0 > x_1 > \ldots \]

- Height of values in reaching definitions?

- Important property: finite descending chain

- Can an infinite lattice have a finite descending chain?

- Example: Constant Propagation/Folding
  - To determine if a variable is a constant

- Data values
  - \text{undef, ... -1, 0, 1, 2, ...}, \text{not-a-constant}

II. 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 \( f_1, f_2 \in F \), \( f_1 \circ f_2 \in F \)

Monotonicity

- A framework \((F, V, \land)\) 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 \((F, V, \land)\) is monotone if and only if
  - \( f(x \land y) \leq f(x) \land f(y) \),
    i.e. merge input, then apply \( f \) is \textbf{smaller than or equal to}
    apply the transfer function individually then merge result
Example

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

- Note: Monotone framework does not mean that \( f(x) \leq x \)
  - e.g. Reaching definition for two definitions in program
    - suppose: \( f_x: \text{Gen}_x = \{d_1, d_2\}; \text{Kill}_x = \{} \)

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

Distributivity

- A framework \((F, V, \land)\) 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

\[
\begin{array}{c}
a = 2 \\
b = 3 \\
c = a + b \\
a = 3 \\
b = 2 \\
\end{array}
\]

III. Data Flow Analysis

- Definition
  - Let \( f_1, ..., f_m : F \rightarrow F \), \( f_i \) is the transfer function for node \( i \)
    - \( f_p = f_{n_k} \ldots f_{n_1} \cdot p \) is a path through nodes \( n_k, ..., n_1 \)
    - \( f_p \) = identify function, if \( p \) is an empty path

- Ideal data flow answer:
  - For each node \( n \):
    - \( \land f_{p_i} (\lor) \), for all possibly executed paths \( p_i \) reaching \( n \).

- Determining all possibly executed paths is undecidable

Meet-Over-Paths MOP

- Err in the conservative direction
- Meet-Over-Paths MOP
  - For each node \( n \):
    - \( \text{MOP} (n) = \land f_{p_i} (\lor) \), for all paths \( p_i \) reaching \( n \)
    - a path exists as long there is an edge in the code
    - consider more paths than necessary
    - \( \text{MOP} = \text{Perfect-Solution} \land \text{Solution-to-Unexecuted-Paths} \)
    - POTentially more constrained, solution is small
    - => conservative
    - It is not safe to be \( > \) Perfect-Solution!
- Desirable solution: as close to MOP as possible
Solving Data Flow Equations

- **Example: Reaching definition**
  - out(entry) = {}
  - Values = {subsets of definitions}
  - Meet operator: \( \cup \)
    \[ \text{in}(b) = \cup \text{out}(p), \text{for all predecessors } p \text{ of } b \]
  - Transfer functions:
    \[ \text{out}(b) = \text{gen}_b \cup (\text{in}(b) - \text{kill}_b) \]

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

- **Iterative algorithm**
  - initializes out(b) to {}
    If converges, it computes Maximum Fixed Point (MFP):
    MFP is the largest of all solutions to equations

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

Partial Correctness of Algorithm

- **If data flow framework is monotone**
  then if the algorithm converges, \( \text{IN}[b] \leq \text{MOP}[b] \)

- **Proof: Induction on path lengths**
  - Define \( \text{IN}[\text{entry}] = \text{OUT}[\text{entry}] \)
    and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of \( \text{IN}[\text{entry}] \)
  - If true for path of length \( k \), \( p_k = (n_1, ..., n_k) \),
    true for path of length \( k+1 \): \( p_{k+1} = (n_1, ..., n_{k+1}) \)
    - Assume: \( \text{IN}[n_k] \leq f_{n_{k-1}}(f_{n_{k-2}}(...f_{n_1}(\text{IN}[\text{entry}]))) \)
    - \( \text{IN}[n_{k+1}] = \text{OUT}[n_k] \wedge ... \)
      \[ \leq \text{OUT}[n_k] \]
      \[ \leq f_{n_k}(\text{IN}[n_k]) \]
      \[ \leq f_{n_k}(f_{n_{k-1}}( ... f_{n_1}(\text{IN}[\text{entry}]))) \]

Precision

- **If data flow framework is distributive**
  then if the algorithm converges, \( \text{IN}[b] = \text{MOP}[b] \)

- Monotone but not distributive: behaves as if there are additional paths
  
  \[
  \begin{align*}
  a &= 2 \\
  b &= 3 \\
  c &= a + b
  \end{align*}
  \[
  \begin{align*}
  a &= 3 \\
  b &= 2 \\
  c &= a + b
  \end{align*}
  \]

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
  - 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
  
  ```
  For each node i
  rPostOrder = NumNodes - PostOrder(i)
  ```

Depth-First Iterative Algorithm (forward)

```c
/* Initialize */
out(Entry) = init_value
For all nodes i
  out(i) = T
change = True

/* iterate */
While Change {
  Change = False
  For each node i in rPostOrder {
    in[i] = \( \wedge \) (out[p]), for all predecessors p of i
    oldout = out[i]
    out[i] = f_i(in[i])
    if oldout \neq out[i]
      Change =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 -> 4 -> 5 -> 7 -> 2 -> 4 ...
  - 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
    (two is necessary even if there are no cycles)

- What is the depth?
  - corresponds to depth of intervals for “reducible” graphs
  - In real programs: average of 2.75
A Check List on 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