**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 \( (V \times V \rightarrow V) \), 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.
  - Re-use code

---

**I. Meet Operator**

- Properties of the meet operator
  - Commutative: \( x \land y = y \land x \)
  - Idempotent: \( x \land x = x \)
  - Associative: \( x \land (y \land z) = (x \land y) \land z \)
  - There is a Top element \( \top \) such that \( x \land \top = x \)

- Meet operator defines a partial ordering on values
  - \( x \leq y \) if and only if \( x \land y = x \)
    - Transitivity: if \( x \leq y \) and \( y \leq z \) then \( x \leq z \)
    - Antisymmetry: if \( x \leq y \) and \( y \leq x \) then \( x = y \)
    - Reflexivity: \( x \leq x \)

---

**Partial Order**

- Example: let \( V = \{ x \mid \text{such that } x \subseteq \{ d_1, d_2, d_3 \} \} \), \( \land = \cap \)

- Top and Bottom elements
  - Top \( \top \) such that \( x \land \top = x \)
  - Bottom \( \bot \) such that \( x \land \bot = \bot \)

- Values and meet operator in a data flow problem define a semi-lattice: there exists a \( \top \), but not necessarily a \( \bot \).
- \( x, y \) are ordered: \( x \leq y \) then \( x \land y = x \)
- What if \( x \) and \( y \) are not ordered?
  - \( x \land y \leq x \), \( x \land y \leq y \), and if \( x \leq \bot \), \( \bot \leq y \), then \( w \leq x \land y \)

---
One vs. All Variables/Definitions

- Lattice for each variable: e.g. intersection

1

0

- Lattice for three variables:

Lattice for three variables:

\[
\begin{array}{cccc}
 & x_{111} & & \\
& x_{110} & x_{11} & \\
& x_{10} & x_{01} & x_{00} \\
& x_{000} & & \\
\end{array}
\]

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
  - undef, ... -1, 0, 1, 2, ..., not-a-constant

II. Transfer Functions

- Basic Properties \( f : V \to V \)
  - 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 smaller than or equal to apply the transfer function individually then merge result
Example

- Reaching definitions: \( f(x) = \text{Gen} \cup (x - \text{Kill}) \), \( \land = \cup \)
  - 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: \( f_1 : \text{Gen}_x = \{d_1, d_2\}; \text{Kill}_x = \{\} \)

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

III. Data Flow Analysis

- Definition
  - Let \( f_1, ..., f_m : e \rightarrow F \), \( f_i \) is the transfer function for node \( i \)
    - \( p = f_{n_k} \cdot ... \cdot f_{n_1} \cdot p \) is a path through nodes \( n_1, ..., n_k \)
    - \( f_p = \) identify function, if \( p \) is an empty path
- Ideal data flow answer:
  - For each node \( n \) :
    - \( \land f_p (\top) \), for all possibly executed paths \( p \) reaching \( n \).

- Determine all possibly executed paths is undecidable

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

\[
\begin{align*}
  f(x) & = \{a=2, b=3\} \\
  f(x) & = \{a=3, b=2\} \\
  c & = a + b
\end{align*}
\]

Meet-Over-Paths MOP

- Err in the conservative direction
- Meet-Over-Paths MOP
  - For each node \( n \):
    - \( \text{MOP} (n) = \land f_{p_i} (\top) \), for all paths \( p_i \) reaching \( n \)
  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - \( \text{MOP} = \) Perfect-Solution \( \land \) Solution-to-Unexecuted-Paths
  - \( \text{MOP} \leq \) Perfect-Solution
  - 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
  - \( \text{out(entry)} = {} \)
  - Values = (subsets of definitions)
  - Meet operator: \( \cup \)
    - \( \text{in(b)} = \cup \text{out(p)}, \text{for all predecessors p of b} \)
  - 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 \( \text{out(b)} \) to \( {} \)
  - If converges, it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations

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

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 \( \text{IN[entry]} = \text{OUT[entry]} \)
    and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of \( \text{IN[entry]} \)
  - If true for path of length \( k \), \( \rho_{k} = (n_{1}, ..., n_{k}) \)
    - Assume: \( \text{IN}[n_{k}] \leq f_{n_{k}-1}(f_{n_{k-2}}(...f_{n_{1}}(\text{IN}[\text{entry}]))) \)
    - \( \text{IN}[n_{k+1}] = \text{OUT}[n_{k}] \land ... \leq \text{OUT}[n_{k}] \)
    - \( \leq f_{n_{k}}(\text{IN}[n_{k}]) \)
    - \( \leq f_{n_{k}}(f_{n_{k-2}}(...f_{n_{1}}(\text{IN}[\text{entry}]))) \)

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

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 \( {} \)
    - 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)

```
input: control flow graph CFG = (N, E, Entry, Exit)

/* 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] = \land (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