Lecture 5

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]

•Marlowe & Ryder, Properties of data flow frameworks: a unified model. Rutgers tech report, Apr. 1988

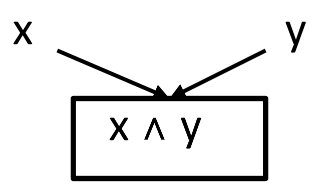
A Unified Framework

- **Data flow problems are defined by**
  - Domain of values: V
  - Meet operator ($V \wedge V \rightarrow V$), 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.
  - Reuse code

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**
  - $x \leq y$ if and only if $x \wedge y = x$
    - Transitivity: if $x \leq y$ and $y \leq z$ then $x \leq z$
    - Antisymmetry: if $x \leq y$ and $y \leq x$ then $x = y$
    - Reflexitivity: $x \leq x$

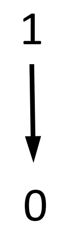
Partial Order

- Example: let **V** = {x | such that x ⊆ { $\mathbf{d_1}$, $\mathbf{d_2}$, $\mathbf{d_3}$}}, ∧ = ∩

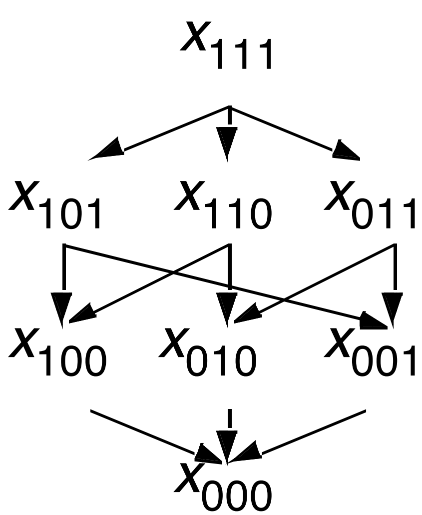
- Top and Bottom elements
  - Top T such that: **x ∧ T = x**
  - Bottom ⊥ such that: **x ∧ ⊥ = ⊥**
- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a T, but not necessarily a ⊥.
- x, y are ordered: x ≤ y then x ∧ y = x
- what if x and y are not ordered?
  - x ∧ 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.

$$x_0 > x_1 > x_2 > \ldots$$

- **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 → 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 $f_1 \cdot f_2 \in F$

Monotonicity

- A framework (*F, V,* ∧) is monotone if and only if
  - x ≤ y implies f(x) ≤ 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,* ∧) is monotone if and only if
  - f(x ∧ y) ≤ f(x) ∧ 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

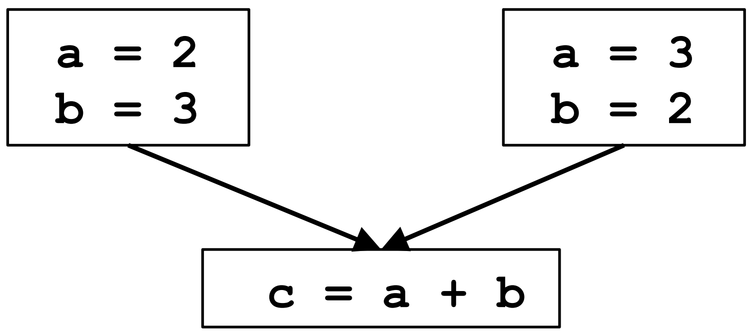
Example

- **Reaching definitions: f(x) = Gen ∪ (x - Kill), ∧ = ∪**
  - Definition 1:
    - $x_1 \leq x_2$, Gen ∪ (x_1 - Kill) ≤ Gen ∪ (x_2 - Kill)
  - Definition 2:
    - (Gen ∪ (x_1 - Kill)) ∪ (Gen ∪ (x_2 - Kill))

      = (Gen ∪ ((x_1 ∪ x_2) - Kill))
- **Note: Monotone framework does not mean that f(x) ≤ x**
    - e.g., reaching definition for two definitions in program
    - suppose: f_x: $Gen_x = \{d_1, d_2\}$; $Kill_x = \{\}$
- **If input(second iteration) ≤ input(first iteration)**
    - result(second iteration) ≤ result(first iteration)

Distributivity

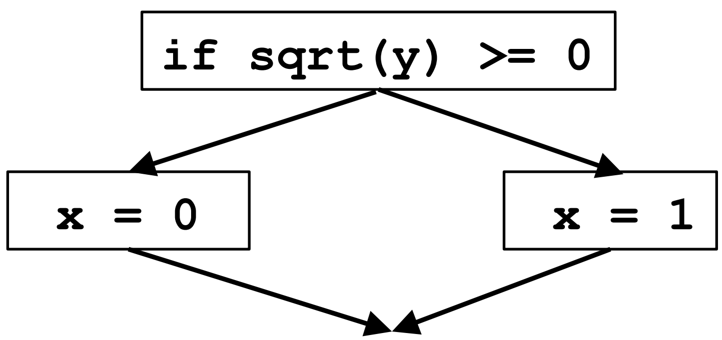
- A framework (F, V, $\wedge$) is **distributive** if and only if
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - i.e. merge input, then apply f is **equal to** apply the transfer function individually then merge result
- Example: Constant Propagation



III. Data Flow Analysis

- **Definition**
  - Let $f_1, \ldots, f_m : \in F$, where f_i is the transfer function for node i
    - $f_p = f_{n_k} \cdot \ldots \cdot f_{n_1}$, where p is a path through nodes $n_1, \ldots, n_k$
    - f_p = identify function, if p is an empty path
- **Ideal data flow answer:**
  - For each node *n:*

    $\wedge f_{p_i}(\top)$, 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:*

    MOP(n) = $\wedge f_{p_i}(\top)$, for all paths p_i reaching n

  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - MOP = Perfect-Solution ∧ Solution-to-Unexecuted-Paths
  - MOP ≤ 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**

Solving Data Flow Equations

- **Example: Reaching definitions**
  - out[entry] = {}
  - Values = {subsets of definitions}
  - Meet operator: ∪
    - in[b] = ∪ out[*p*], for all predecessors *p* of b
  - Transfer functions: out[b] = gen_b ∪ (in[b] -$kill_b$)
- **Any solution satisfying equations = Fixed Point Solution (FP)**
- **Iterative algorithm**
  - initializes out[b] to {}
  - if converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations
- **Properties:**
  - FP ≤ MFP ≤ MOP ≤ Perfect-solution
  - FP, MFP are safe
  - in(b) ≤ MOP(b)

Partial Correctness of Algorithm

- **If data flow framework is monotone, then if the algorithm converges, IN[b] ≤ 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, $p_k = (n_1, ..., n_k)$, then true for path of length k+1: $p_{k+1} = (n_1, ..., n_{k+1})$
    - Assume: $IN[n_k] \leq f_{n_{k-1}}(f_{n_{k-2}}(... f_{n_1}(IN[entry])))$
    - $IN[n_{k+1}] = OUT[n_k] \wedge ...$

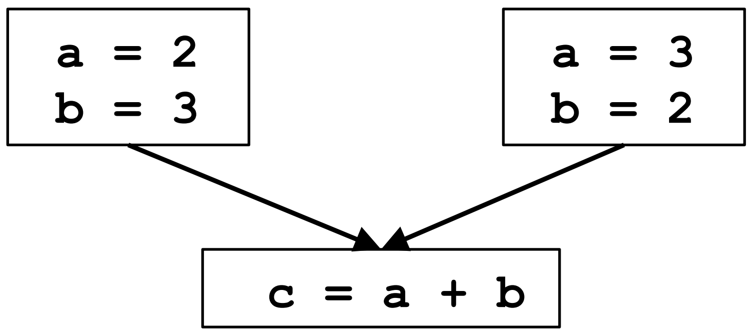
      $\leq OUT[n_k]$

      $\leq f_{n_k}(IN[n_k])$

      $\leq f_{n_{k-1}}(f_{n_{k-2}}(... f_{n_1}(IN[entry])))$

Precision

- **If data flow framework is distributive,then if the algorithm converges, IN[b] = 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**

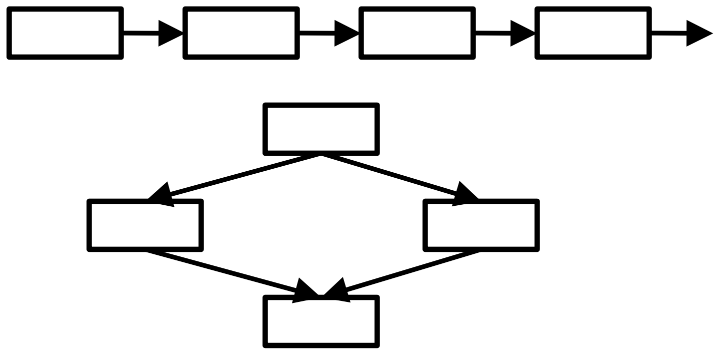
- For each variable IN[b], OUT[b], consider the sequence of values set to each variable across iterations:

  – if sequence for in[b] is monotonically decreasing
    - sequence for out[b] is monotonically decreasing
      - (out[b] initialized to T)

  – if sequence for out[b] is monotonically decreasing
    - sequence of 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] = ⊤
    Change = True

/* iterate */
    While Change {
       Change = False
       For each node i in rPostOrder {
          in[i] = ∧(out[p]), for all predecessors p of i
          oldout = out[i]
          out[i] = f_i(in[i])
          if oldout ≠ 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:
    - e.g., 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
    - (2 are 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 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