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]

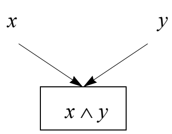
Marlowe&Ryder, Properties of data flow frameworks: a unified model
Rutgers tech report, Apr. 1988

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

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 $\top$ such that $x \wedge \top = x$
- **Meet operator defines a partial ordering on values**
  - $x \le y$ if and only if $x \wedge y = x$
    - Transitivity: if $x \le y$ and $y \le z$ then $x \le z$
    - Antisymmetry: if $x \le y$ and $y \le x$ then $x = y$
    - Reflexitivity: $x \le x$

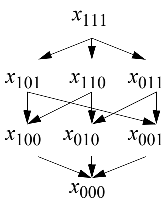
Partial Order

- Example: let V = {x | such that x $\subseteq$ { d_1, d_2, d_3}}, $\wedge = \cap$
- Top and Bottom elements
  - Top $\top$ such that $x \wedge \top = x$
  - Bottom $\perp$ such that $x \wedge \perp = \perp$
- Values and meet operator in a data flow problem define a semi-lattice: there exists a $\top$, but not necessarily a $\perp$.
- x, y are ordered: $x \le y$ then $x \wedge y = x$
- what if x and y are not ordered?
  - $x \wedge y \le x$, $x \wedge y \le y$, and if $w \le x$, $w \le y$,then $w \le x \wedge y$
  - .

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**
  - undef, ... -1, 0, 1, 2, ..., 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 $f_1, f_2 \in F$, $f_1 \bullet f_2 \in F$

Monotonicity

- A framework $(F, V, \wedge)$ 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 $(F, V, \wedge)$ is monotone if and only if
  - $f(x \wedge y) \leq f(x) \wedge f(y)$,

  i.e. merge input, then apply f is **smaller than or equal to** apply the transfer function individually then merge result

Example

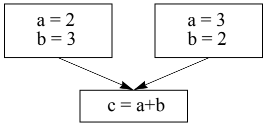
- **Reaching definitions: f(x) = Gen ∪ (x - Kill), ∧ = ∪**
  - Definition 1:
    - $x_1 \le x_2$, Gen ∪ (x_1 - Kill) ≤ Gen ∪ (x_2 - Kill)
  - Definition 2:
    - (Gen ∪ (x_1 - Kill)) ∪ (Gen ∪ (x_2 - Kill))

      = (Gen ∪ (($x_1 \cup x_2$) - 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*, ∧) is distributive if and only if**
  - $f(x \wedge y) = f(x) \wedge f(y)$,

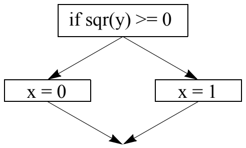
    i.e. merge input, then apply *f* is **equal to**
    apply the transfer function individually then merge result
- **Example: Constant Propagation**



III. Data Flow Analysis

- **Definition**
  - Let $f_1, ..., f_m : \in F$, f_i is the transfer function for node *i*
    - $f_p = f_{n_k} \bullet \ldots \bullet f_{n_1}$, 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 :

    $\wedge\ f_{p_i}(\top)$, 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
    => 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: $\cup$
    $in(b) = \cup\ out(p)$, for all predecessors p of b
  - Transfer functions:
    $out(b) = gen_b \cup (in(b) - kill_b)$
- **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 ≤ 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, $p_k = (n_1, ..., n_k)$,
    true for path of length $k+1$: $p_{k+1} = (n_1, ..., n_{k+1})$
    - Assume: $IN[n_k] \le f_{n_{k-1}}(f_{n_{k-2}}(... f_{n_1}(IN[entry])))$
    - $IN[n_{k+1}] = OUT[n_k] \wedge ...$

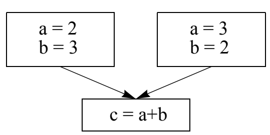
      $\le OUT[n_k]$

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

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

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

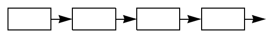


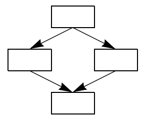
Additional Property to Guarantee Convergence

- **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 $\top$)
  - 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
    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