Lecture 5

Foundation of Data Flow Analysis

I  Semi-lattice (set of values, meet operator)
II Transfer functions
III Problem statement
IV Correctness, Convergence, Precision
V Efficiency

Reference: ASU 10. 11 (pp. 672-673, 680-694)
Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
Allen, Rosen, Zadeck’s Optimization in Compilers (forthcoming?)
Marlowe&Ryder, Properties of data flow frameworks: a unified model
Rutgers tech report, Apr. 1988
A Unified Framework

• Data flow problems are defined by
  • Semilattice
    • domain of values
    • meet operator
  • 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.
  • Re-use code
I. Semi-lattice

- A semi-lattice $S = \{ \text{a set of values } V, \text{ a meet operator } \wedge \}$

- Properties of the meet operator
  - idempotent: $x \wedge x = x$
  - commutative: $x \wedge y = y \wedge x$
  - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
Example: Reaching Definition

• Semi-lattice
  • \( V = \{ x \mid \text{such that } x \subseteq \{ d_1, d_2, d_3 \} \} \)
  • \( \land = \bigcup \)

\[
\begin{align*}
  \{ \} & \quad (\top) \\
  \{ d_1 \} & \quad \{ d_2 \} & \quad \{ d_3 \} \\
  \{ d_1, d_2 \} & \quad \{ d_2, d_3 \} & \quad \{ d_1, d_3 \} \\
  \{ d_1, d_2, d_3 \} & \quad (\bot)
\end{align*}
\]

• \( x \land y \): their first common descendant
• Define top element \( \top \), such that \( x \land \top = x \)
• Define bottom element \( \bot \), such that \( x \land \bot = \bot \)
Another Useful View

• Define: \( x \leq y \) if and only if \( x \land y = x \)

• Properties of meet operator guarantee that \( \leq \) is a partial order
  • Reflexive: \( x \leq x \)
  • Antisymmetric: if \( x \leq y \) and \( y \leq x \) then \( x = y \)
  • Transitive: if \( x \leq y \) and \( y \leq z \) then \( x \leq z \)

• Exercise: prove that \( x \land y \) is the first common descendant
  • \( x \land y \leq x, x \land y \leq y, \text{ and if } w \leq x, w \leq y, \text{ then } w \leq x \land y \)
  • Let \( x \land y = w \)
  • \( x \land w = w \)
  • \( w \land x = w, w \land y = w, (w \land x) \land (w \land y) = w \land (x \land y) = w, w \leq (x \land y) \).
Another Example

- Semi-lattice
  - $V = \{x \mid \text{such that } x \subseteq \{d_1, d_2, d_3\}\}$
  - $\land = \cap$

- $\leq$ is ?

```
```

$\{d_1, d_2, d_3\}$

$\{d_1, d_2\}$ $\{d_2, d_3\}$ $\{d_1, d_3\}$

$\{d_1\}$ $\{d_2\}$ $\{d_3\}$

$\{\}\quad (\perp)$

$(\top)$
One vs. All Variables/Definitions

- Semi-lattice can get quite large: $2^n$ elements for n var/definition
- Or, define semi-lattice for each variable/definition & compose
- Example: 0: not a reaching definition; 1: a reaching definition

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>$x \land y$</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Semi-lattice: ( {0, 1}, “OR” }
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 chains

• infinite lattice \(\Rightarrow\) infinite descending chains?
Example: Constant Propagation/Folding

• At every basic block boundary, for each variable $v$
  • determine if $v$ is a constant
  • if so, what is the value?

  • **Data values**
    • $undef$, $... -1$, $0$, $1$, $2$, $...$, $NAC$(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 \((F, V, \wedge)\) 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. meet inputs, then apply \(f\) is **smaller than or equal to**
  apply \(f\) individually then meet results
Example

• Reaching definitions: \( f(x) = \text{Gen} \cup (x - \text{Kill}), \wedge = \cup \)
  • Definition 1:
    • Let \( x_1 \leq x_2, \)
      \[
      f(x_1): \text{Gen} \cup (x_1 - \text{Kill})
      \]
      \[
      f(x_2): \text{Gen} \cup (x_2 - \text{Kill})
      \]
  • Definition 2:
    • \( f (x_1 \wedge x_2) = (\text{Gen} \cup ((x_1 \cup x_2) - \text{Kill})) \)
      \[
      f(x_1) \wedge f(x_2) = (\text{Gen} \cup (x_1 - \text{Kill}) ) \cup (\text{Gen} \cup (x_2 - \text{Kill}) )
      \]
Important Note

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


\[
\begin{array}{c|c|c|c|c}
   & x & \{d_1, d_2\} & \{d_2\} & \{\}\n\hline
f(x) & \{d_1\} & \{d_1\} & \{d_1\} & \{\}\n\end{array}
\]

• **Purpose of monotone framework:**
  
  • If input(second iteration) \( \leq \) input(first iteration)
  
  • result(second iteration) \( \leq \) 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

- Semi-lattice for 1 variable:

```
<table>
<thead>
<tr>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>... -3 -2 -1 0 1 2 3 ...</td>
</tr>
</tbody>
</table>
```

- Transfer functions:

- Assume a basic block has only 1 instruction
  - Can handle basic blocks with multiple instructions by
    - composing the transfer function
    - breaking down basic blocks to 1-inst. basic blocks

- Let $IN[b,x]$, $OUT[b,x]$
  - be the information for variable $x$ at entry and exit of basic block $b$

- Non-assignment instructions: $OUT[b,x] = IN[b,x]$
Constant Propagation (Cont.)

- Let an assignment be of the form $x_3 = x_1 + x_2$
  - $+$ represents a generic operator
  - $\text{OUT}[b, x] = \text{IN}[b, x]$, if $x \neq x_3$
  - Otherwise:

<table>
<thead>
<tr>
<th>IN[b,x₁]</th>
<th>IN[b,x₂]</th>
<th>OUT[b,x₃]</th>
</tr>
</thead>
<tbody>
<tr>
<td>undef</td>
<td></td>
<td>undef</td>
</tr>
<tr>
<td>c₂</td>
<td></td>
<td>undef</td>
</tr>
<tr>
<td>NAC</td>
<td></td>
<td>undef/NAC</td>
</tr>
<tr>
<td>c₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c₂</td>
<td></td>
<td>+c₁+c₂</td>
</tr>
<tr>
<td>NAC</td>
<td></td>
<td>NAC</td>
</tr>
<tr>
<td>NAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Monotone framework?
Distributive?

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure.png}
\caption{Iterative solution is not precise! It is also not wrong. It is conservative.}
\end{figure}
III. Data Flow Analysis

• Semi-lattice (set of values, meet operator)

• Transfer functions
  • Let $f_1, ..., f_m : \in F$, $f_i$ is the transfer function for node $i$
    
    $f_p = f_{n_k} \circ ... \circ f_{n_1}$, $p$ is a path through nodes $n_1, ..., n_k$

    $f_p = $ identify function, if $p$ is an empty path

• Boundary condition
  • can be represented as a constant transfer function
    for entry/exit node
IDEAL Data Flow

- For each node $n : \bigwedge f_{p_i} (\text{start-val})$,
  for all possibly executed paths $p_i$ reaching $n$

- Example

```
if sqr(y) >= 0
```

```
false  x = 0  true  x = 1
```

- Determining all possibly executed paths is undecidable
Meet-Over-Paths MOP

- Err in the conservative direction

- Meet-Over-Paths MOP
  - Assume a path exists as long there is an edge in the code
  - For each node \( n \): \n    \[
    \text{MOP}(n) = \bigwedge f_{p_i} \text{(start-val), for all paths } p_i \text{ reaching } n
    \]

- Compare MOP with IDEAL
  - MOP includes more paths than IDEAL
  - \( \text{MOP} = \text{IDEAL} \ igwedge \ \text{Result(Unexecuted-Paths)} \)
  - \( \text{MOP} \leq \text{IDEAL} \)
  - MOP is a “smaller” solution, more conservative, safe

- Desirable solution \( \leq \text{MOP} \leq \text{IDEAL} \)
  - as close to MOP from below as possible
Solving Data Flow Equations

• Example: Reaching definition
  • 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 T
  • If converges, it computes Maximum Fixed Point (MFP)
  • MFP is the largest of all solutions to equations

• Properties:
  • FP ≤ MFP ≤ MOP ≤ IDEAL
  • FP, MFP, MOP are safe
  • ∀b, IN[b] ≤ MOP(b)
IV. Properties of Iterative Algorithm

- Let $\text{IN}_k[b]$, $\text{OUT}_k[b]$ be result after step $k$ in iterative algorithm
- $\text{IN}[b]$ refers to the final answer.

  - Monotonicity: $\text{IN}_{k-1}[b] \geq \text{IN}_k[b]$, $\text{OUT}_{k-1}[b] \geq \text{OUT}_k[b]$  
    induction proof: by iteration steps
  - Convergence: finite descending chains $\Rightarrow$ converge
  
- Correctness: $\text{IN}[b] \leq \text{MOP}[b]$  
  induction proof: by path length
  
- Precision: If data flow framework is distributive, $\text{IN}[b] = \text{MOP}[b]$
  - Monotone but not distributive: behaves as if additional paths exist
V. Efficiency

• Speed of convergence depends on order of node visits

Reverse post-order: d, b, c, a -> a, c, b, d
Breadth-first: a, b, d, c

• 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 = count - 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] = \lor (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 (top, bottom)
  - meet operator
  - finite descending chains?

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

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