Data-flow Frameworks

Data-flow Analysis

• Efficient technique for proving properties about programs
• Not as powerful as theorem provers, but requires less human expertise
• Uses an annotated control-flow graph model of the program
  • Compute facts for each node
• Use the flow in the graph to compute facts about the whole program
  • Facts that are true at a stmt in a program, for all possible executions to that stmt
Previously

- Looked at several data-flow analysis problems
  - Reaching definitions
  - Live variables
  - Constant propagation
- All done in a somewhat informal way
  - This time, define these problems more formally

Deriving global behavior from local information

- We know the effect of each node in isolation
- Global behavior is found by using the structure of the control-flow graph
  - Edges represent flow out from the source into the destination
    - Can flow information either forward or backward over the edges
  - Merge points in the flow graph require that information flowing from multiple sources be combined
    - This combination must be conservative
    - The appropriate combination operator depends on whether we are interested in gathering facts along
      - all paths leading to a node
      - any (some) path leading to a node
**Initial Facts about a node: GEN and KILL sets**

- For each node $i$ associate sets:
  - $\text{GEN}(i)$ - what is to be added (generated)
  - $\text{KILL}(i)$ - ωηατ ισ to be eliminated (killed)

- The definitions of GEN and KILL depend on the problem that is being solved.

- Often the GEN and KILL sets can be derived from the abstract syntax tree:
  - E.g., variables defined in a node
  - variables referenced in a node

**General Approach**

- **Initial node values**
  - for each node define GEN and KILL information

- **In Equations for each node**
  - for each node we have an equation of the form:
    $$\text{IN}(n_i) := \text{Merge} \left( \forall n_j \text{ OUT}(n_j) \right)$$
  - “Merge” operation over the {predecessors|successors} of $n_i$ depending on whether it is a {forward|backward}-flow problem

- **OUT Equations for each node**
  - for each node we have an equation of the form:
    $$\text{OUT}(n_i) := f( \text{IN}(n_i) )$$
  - Transfer function $f$ usually depend on GEN and KILL information that is computed for each node
  - Usually: $\text{OUT} := (\text{IN} \setminus \text{KILL}) \cup \text{GEN}$
General Approach (continued)

- Propagation rule
  - Forward or backward
  - IN value for the initial node
  - Since IN depends on OUT, need to initialize OUT for each node
    - Initial value depends on the problem
  - Merge operator determined by whether it is an all-path or an any-path problem

- Final result rule
  - Usually based on IN, OUT, GEN, and KILL for each node
  - Sometimes only need to look at the final node

Fixed point

- Compute the new In and Out values for each node, until all the values stabilize
- Reaches a fixed point, if
  - there are only a finite number of possible sets that can be associated with a node, and
  - if the function that determines the sets that can be associated with a node is monotonic
- Can visit each node in any order, but algorithm is more efficient if a worklist is used
  - Worklist holds the nodes that need to have new In values computed
**Typical Merge Functions**

<table>
<thead>
<tr>
<th></th>
<th>Forward-flow for i ≠ initial</th>
<th>Backward-flow for i ≠ final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any-path</td>
<td>( \text{IN}(i) = \bigcup_{j \in \text{preds}(i)} \text{OUT}(j) )</td>
<td>( \text{IN}(i) = \bigcup_{j \in \text{succs}(i)} \text{OUT}(j) )</td>
</tr>
<tr>
<td>All-path</td>
<td>( \text{IN}(i) = \bigcap_{j \in \text{preds}(i)} \text{OUT}(j) )</td>
<td>( \text{IN}(i) = \bigcap_{j \in \text{succs}(i)} \text{OUT}(j) )</td>
</tr>
</tbody>
</table>

**Formalizing Data-flow Analysis**

- Compute facts for each node of a control-flow graph
  - The IN and OUT sets
  - Depends on the direction facts are propagated
**Reaching Definitions**

- Which definitions can “reach” a statement
- \( x_i \) means that the definition of variable \( x \) at node \( i \) reached the current node

```plaintext
int x,y;
...
x := foo();
y := x + 2;
if x > 0 then
  x := x + y;
end if;
...
```

**Forward-flow, any-path problem**

**Reaching Definitions - Formalized**

- In what direction are the facts propagated?
  - Choice of forwards or backwards:
    - Could propagate definitions forward to the statements that they reach, or
    - could propagate statements backwards to the definitions
- Reaching definitions
  - Here, easier to identify definitions and propagate them forward
  - Solved as a forward-flow problem
Reaching Definitions - Formalized

- How should facts be merged when a node has multiple predecessors?
  - Union for an any-path problem
  - Intersection for an all-path problem

- Reaching definitions
  - Since both $X_1$ and $X_4$ can reach the highlighted node, we want both definitions to be propagated
  - Use union to merge facts

$IN(i) = \bigcup_{j \in \text{preds}(i)} OUT(j)$

Use predecessors for a forward-flow problem

Reaching Definitions - Formalized

- What new facts are known after a node is processed?
  - Called the GEN set of a node

- For reaching definitions
  - If node $i$ defines a variable $v_i$, then we want to add $v_i$ to the set of reaching definitions after the node is processed

$\text{GEN}(i) = \begin{cases} v_i & \text{if node } i \text{ defines } v \\ \emptyset & \text{otherwise} \end{cases}$
Reaching Definitions - Formalized

• What facts are no longer known after a node is processed?
  • Called the KILL set of a node

• For reaching definitions
  • If node i defines a variable v, then we know that for any j, v_j is no longer a potential reaching definition after the node is processed

\[ \text{KILL}(i) = \begin{cases} \forall j, v_j & \text{if node } i \text{ defines } v \\ \emptyset & \text{otherwise} \end{cases} \]

Reaching Definitions - Formalized

• How should facts be propagated over a node?
  • Called the transfer function of a node

• For each node i, have an equation
  • \( \text{OUT}(i) = f_i(\text{IN}(i)) \)
  • Usually depends on GEN and KILL for each node

• Usually:
  \[ \text{OUT}(i) = (\text{IN}(i) \setminus \text{KILL}(i)) \cup \text{GEN}(i) \]

• Reaching definitions
  • The usual equation works
Reaching Definitions - Formalized

• What facts are known initially?

• Reaching definitions
  • Assuming nothing is defined before this program runs, no definitions can flow in, so no facts are known initially

IN(1) = ∅

∀j, OUT(j) = ∅

Reaching Definitions - Formalized

• How should the facts be interpreted once data-flow analysis stops propagating?

• Reaching definitions
  • The definitions that reach a node are the IN set of that node

REACH(i) = IN(i)
Reaching Definitions - Formalized

• Reaching definitions summary
  - Forward-flow problem
  - Any-path problem

\[
\begin{align*}
x &= \text{foo()} & \{X_1, y_1\} \\
y &= x + 2 & \{X_1, y_2\} \\
\text{if}(x > 0) & & \{X_1, y_2\} \\
x &= x + y & \{X_1, y_2, y_3\} \\
& & \{X_1, y_2, x_4\}
\end{align*}
\]

Initial values: \(\forall j, \text{OUT}(j) = \emptyset\)

\[
\begin{align*}
\text{GEN}(i) &= \begin{cases} v_i & \text{if node } i \text{ defines } v \\ \emptyset & \text{otherwise} \end{cases} \\
\text{KILL}(i) &= \begin{cases} \forall j, v_j & \text{if node } i \text{ defines } v \\ \emptyset & \text{otherwise} \end{cases} \\
\text{IN}(i) &= \begin{cases} \emptyset & \text{if } i = n_s \\ \bigcup_{j \in \text{preds}(i)} \text{OUT}(j) & \text{otherwise} \end{cases} \\
\text{OUT}(i) &= (\text{IN}(i) \setminus \text{KILL}(i)) \cup \text{GEN}(i) \\
\text{REACH}(i) &= \text{IN}(i)
\end{align*}
\]

Using Reaching Definitions for Anomaly Detection

• Undefined referenced variable:
  For each node \(i\),
  for each \(v \in \text{REF}(i)\)
  if there is no \(j\),
  such that \(v_j \in \text{REACH}(i)\)
  then \(v\) is an undefined ref at node \(i\)
Example: Live Variables

- What definitions are live at a statement?
- A variable, $x$, is live at node $i$ if there exists a definition path for $x$ from node $i$ to a use of $x$

```
int x,y;
...  
x := foo();
y := x + 2;
if(x > 0 then
   x := x + y;
end if;
...
```

Live Variables - Formalized

- In what direction are the facts propagated?
- Live variables
  - Keep track of which variables have a use on some path in the future
  - Solve as a backward-flow problem
  - Flow the uses of variables backwards to their definitions
Live Variables - Formalized

- What new facts are known after a node is processed?
- For live variables
  - If node i uses a variable v, then that variable is live before the node is executed
    - Assume that uses occur before definitions on a node
  - Let REF(i) be the set of variables referenced on node i
    
    \[
    \text{GEN}(i) = \text{REF}(i)
    \]

Live Variables - Formalized

- What facts are no longer known after a node is processed?
- For live variables
  - If node i defines a variable v, then this means the current path is not def-clear with respect to that variable
  - Definitions kill liveness
  - Let DEF(i) be the set of variables defined on node i
    
    \[
    \text{KILL}(i) = \text{DEF}(i)
    \]
Live Variables - Formalized

- How should facts be propagated over a node?
- For live variables
  - The usual equation works
  \[ \text{OUT}(i) = (\text{IN}(i) \setminus \text{KILL}(i)) \cup \text{GEN}(i) \]

Live Variables - Formalized

- How should facts be merged when a node has multiple successors?
- Live variables
  - A variable is live if there is some def-clear path
    - This is an any-path problem
    - Use union to merge facts

\[ \text{IN}(i) = \bigcup_{j \in \text{succs}(i)} \text{OUT}(j) \]

Use successors for a backward-flow problem
Live Variables - Formalized

- What facts are known initially?
  - Live variables
    - Assuming nothing is used after this program runs, no live variables can flow in, so no facts are known initially

\[
\text{IN}(5) = \emptyset
\]

\[
\forall_j, \text{OUT}(j) = \emptyset
\]

Live Variables - Formalized

- How should the facts be interpreted once data-flow analysis stops propagating?
  - Live variables
    - The variables that are live on a node are the OUT set for that node

\[
\text{LIVE}(i) = \text{OUT}(i)
\]
Live Variables - Formalized

- Live variables summary
  - Backward-flow problem
  - Any-path problem

\[
\begin{align*}
x &= \text{foo()} \\
y &= x + 2 \\
\text{if}(x > 0) & \\
x &= x + y
\end{align*}
\]

\[
\begin{align*}
\{ \} & \quad \{x\} & \quad \{x\} & \quad \{x,y\} & \quad \{x,y\} \\
\{x\} & \quad \{x,y\} & \quad \{x,y\} & \quad \{x,y\} \\
\{x,y\} & \quad \{x,y\} & \quad \{x,y\} & \quad \{x,y\} \\
\emptyset & \quad \emptyset & \quad \emptyset & \quad \emptyset \\
\emptyset & \quad \emptyset & \quad \emptyset & \quad \emptyset 
\end{align*}
\]

Initial values: \( \forall j, \text{OUT}(j) = \emptyset \)

\[
\begin{align*}
\text{GEN}(i) &= \text{REF}(i) & \text{KILL}(i) &= \text{DEF}(i) \\
\text{IN}(i) &= \begin{cases} \\
\emptyset & \text{if } i = n_f \\
\bigcup_{j \in \text{succs}(i)} \text{OUT}(j) & \text{otherwise} \\
\end{cases} \\
\text{OUT}(i) &= (\text{IN}(i) \setminus \text{KILL}(i)) \cup \text{GEN}(i) \\
\text{LIVE}(i) &= \text{OUT}(i)
\end{align*}
\]

Example - Uninitialized Variables

- Want to determine if it may be possible for a variable to be used without having been defined
- Use data-flow analysis to determine which variables are undefined at each node in the control-flow graph
  - A variable is uninitialized on a node if it is undefined and referenced on that node

- Similar to undefined referenced variables shown previously, but will incorporate variables becoming undefined and not be concerned about where (just if) they are defined
Example - Uninitialized Variables

- In what direction are the facts propagated?
  - Flow the set of undefined variables forward to potential references
  - A forward-flow problem

- What new facts are known after a node is processed?
  - If a variable becomes undefined on a node, then that variable should be added to the OUT set of that node
    - A variable becomes undefined when it is no longer in scope or declared without an initial value

\[ \text{GEN}(i) = \text{UNDEF}(i) \]

Example - Uninitialized Variables

- What facts are removed after a node is processed?
  - If a node defines a variable, it is no longer undefined

\[ \text{KILL}(i) = \text{DEF}(i) \]

- How should facts be propagated over a node?
  - The usual equation works

\[ \text{OUT}(i) = (\text{IN}(i) \setminus \text{KILL}(i)) \cup \text{GEN}(i) \]
Example - Uninitialized Variables

1. How should facts be merged when a node has multiple predecessors?
   - Want to determine if a variable may be uninitialized
   - This is an any-path problem, use union

2. What facts are known initially?
   - Every variable starts out undefined
   
   \[ \text{IN}(i) = \begin{cases} \text{All vars} & \text{if } i = 1 \\ \bigcup_{j \in \text{preds}(i)} \text{OUT}(j) & \text{otherwise} \end{cases} \]

3. How should the facts be interpreted once data-flow analysis stops propagating?
   - A variable may be uninitialized on a node if it is undefined and referenced
   - UNINIT(i) = IN(i) \cap REF(i)

Example - Uninitialized Variables

1. Summary
   - Forward-flow problem
   - Any-path problem

   Initial values: \forall j, \text{OUT}(j) = \emptyset
   \text{GEN}(i) = \text{UNDEF}(i)
   \text{KILL}(i) = \text{DEF}(i)
   \text{IN}(i) = \begin{cases} \text{All vars} & \text{if } i = 1 \\ \bigcup_{j \in \text{preds}(i)} \text{OUT}(j) & \text{otherwise} \end{cases}
   \text{OUT}(i) = (\text{IN}(i) \setminus \text{KILL}(i)) \cup \text{GEN}(i)
   \text{UNINIT}(i) = \text{IN}(i) \cap \text{REF}(i)
Example: Uninitialized Variables

1. pos int: numVal;
2. for i in 1..numVal loop
3.   read(x);
4.   if i = 1 then
5.     sum := x;
6.   else
7.     sum := sum + x;
8.   endif;
9. end loop;
10. write(sum);

Node 2 is the loop init. ($i := 1$)
Node 3 is the loop test ($i \leq \text{numVal}$)
Node 8 is the loop increment ($i := i+1$)
Node 9 is leaving the loop ($i$ goes out of scope)

Example: Uninitialized Variables

1. pos int: numVal;
2. for i in 1..numVal loop
3.   read(x);
4.   if i = 1 then
5.     sum := x;
6.   else
7.     sum := sum + x;
8.   endif;
9. end loop;
10. write(sum);

numVal will be abbreviated as $n$
sum will be abbreviated as $s$
**Ref and Def facts**

1. `pos int: numVal;`
2. for `i` in `1..numVal` loop
3. `read(x);`
4. if `i = 1` then
5. `sum:= x;`
6. else
7. `sum:= sum + x;`
8. endif;
9. end loop;
10. `write(sum);`

**Visualizing the iterative worklist algorithm**

- **GEN(i) = UNDEF(i)**
- **KILL(i) = DEF(i)**
- **OUT(i) = (IN(i) \ KILL(i)) \ GEN(i)**

Worklist:
- `def(n)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `def(x)`
- `ref(i)`
- `ref(x)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(x)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(i)`
- `def(x)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undef(i)`
- `ref(s)`
- `def(s)`
- `ref(x,s)`
- `def(s)`
- `ref(s)`
- `def(i)`
- `ref(i,n)`
- `undefined`
### Skipping Ahead to the End

Worklist: \( X \times X \times X \times X \times X \times X \times X \times X \)

<table>
<thead>
<tr>
<th>NODE</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i,n,s,x</td>
<td>i,s,x</td>
</tr>
<tr>
<td>2</td>
<td>i,s,x</td>
<td>s,x</td>
</tr>
<tr>
<td>3</td>
<td>s,x</td>
<td>s,x</td>
</tr>
<tr>
<td>4</td>
<td>s,x</td>
<td>s</td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>6</td>
<td>s</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>s</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>s,x</td>
<td>i,s,x</td>
</tr>
<tr>
<td>10</td>
<td>i,s,x</td>
<td>i,n,s,x</td>
</tr>
</tbody>
</table>

- Processing stops when there are no more nodes on the Worklist
  - Some nodes processed more than once
  - Most nodes processed once

### Determining the result

UNINIT(i) = IN(i) \( \cap \) REF(i)

<table>
<thead>
<tr>
<th>NODE</th>
<th>IN</th>
<th>REF</th>
<th>UNINIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i,n,s,x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>i,s,x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>s,x</td>
<td>i,n</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>s,x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>i</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>s</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
<td>s,x</td>
<td>s</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>i</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>s,x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>i,s,x</td>
<td>s</td>
<td>s</td>
</tr>
</tbody>
</table>

- \( s \) may be uninitialized on nodes 7 and 10
Finding a Violating Path

```
1  pos int: numVal;
2,3,9 for i in 1..numVal loop
4       read(x);
5       if i = 1 then
6           sum:= x;
7           else
8               sum:= sum + x;
9           endif;
10      end loop;
10      write(sum);
```

- 1, 2, 3, 9, 10
- 1, 2, 3, 4, 5, 7, 8,…
  - An infeasible path

A Few Loose Ends to Tie Off

- Order of operations in a node
- Termination
- Constant propagation
- Efficient representations for facts
The Order of Operations in a Node

• When defining the GEN sets for live variables, we assumed that uses occur before definitions on a node
  • \( x = x + 1 \)

• Sometimes control-flow graphs have multiple statements contained within a single node
  • Need to consider the order of operations within a node in this case

• For simplicity, the examples have nodes with single statements

---

The Order of Operations in a Node

A variable, \( x \), is live at node \( i \) if there exists a def-clear path for \( x \) from node \( i \) to a use of \( x \)

\[
\begin{align*}
X &:= Y+Z \\
Y &:= \ldots X\ldots
\end{align*}
\]

\( X \) is defined but only after it is referenced. Needs to be live on entry to node

\( Y \) is defined but only after it is referenced.

\( \text{GEN} := \text{referenced variables that are not defined previously in the node e.g.} \ Y, \ Z \)

\( \text{KILL} := \text{defined variables that are not referenced previously in the node e.g.,} \ X \ \text{but not} \ Y \)
**Facts in Data-flow Analysis**

- Facts are represented by a meet semi-lattice
  - a set of values
  - a partial order on the values
  - a meet operator that computes the greatest lower bound of two values with respect to the partial order
- Practical lattices have:
  - a finite set of values
  - distinguished top (T) and bottom (⊥) elements
- For the uninitialized variables example, the values of the lattice are the power set of the variables in the program
  - Variables are i, n, s, x

**Example Meet Semilattice**

values=PowerSet({i,n,s,x})

\[ T = \{ \} \]

\[ \bot = \{i,n,s,x\} \]

Ordering = \( \subseteq \)

Meet = \( \cup \)
Example Meet Semilattice

Moves values down the lattice, from top (T) to bottom (⊥)

\[ \{n\} \cup \{s, x\} = \{n, s, x\} \]

Well-formed Fixed Point Problem

- Conditions under which the data-flow analysis will always terminate
  - If there are only a finite number of possible sets that can be associated with a node
    - Lattice is finite
  - If the function that determines the sets that can be associated with a node is monotonic
    - Always move down or stay at the same place on the lattice
Facts for Constant Propagation

values = \{Unk,C,NC\}

T = \{ Not Constant \}

⊥ = \{ Unknown \}

Meet = \cap

At each node in the control-flow graph, each variable has one of these 3 values

Limitations of Data-flow Analysis

- Conservative, approximate approach
  - Assumes that all paths in the program model are executable

- Aliasing problem
  - A[I] := 2
    - If there is an action on any element of an array, to be conservative, assume that there is an action on all elements of that error

- Thus, may get false positives (error reports)
  - Spurious results, as seen in the uninitialized variable example
  - Accurate up to "symbolic execution"
  - Techniques are available for incrementally improving accuracy
Representing sets of variables

• Typically use bit vectors

\[
\begin{array}{ccccccc}
i & n & s & x & \ldots \\
1 & 0 & 1 & 1 & 0 & \ldots \\
\end{array} = \{i, s, x\}
\]

• GEN, KILL, DEF, REF, IN, and OUT are represented by bit vectors
  • Union becomes logical OR
  • Intersection becomes logical AND

Data-flow Frameworks

• We can reason about the structure of the mathematical objects and infer properties of the analysis
  • if we can prove monotonicity and boundedness we get solution techniques with small upper bounds
  • There other categories of function spaces, lattices, and flow graphs
• We can generate implementations of an analyzer from a specification of the mathematical objects
In Summary

- Automated static analysis approaches provide a means for "knowing" something about a system
- Examples shown so far are useful in program optimization
  - Transform program so it executes faster
  - Determine potential use to optimize resource usage
- Information can all be derived automatically
  - Definitions and references to variables
  - Constant propagation: evaluate expressions
- Desired information is application independent