Analysis of Software Artifacts

Dataflow Analysis: Examples and Correctness

Jonathan Aldrich
Outline

• Dataflow Analysis Frameworks
  • Lattices
  • Abstraction functions
  • Control flow graphs
  • Flow functions
  • Worklist algorithm

• Example Dataflow Analyses
  • Constant Propagation
  • Reaching Definitions
  • Live Variable Analysis

• Dataflow Analysis Correctness
  • Termination
  • Soundness
Worklist Dataflow Analysis Algorithm

\[
\text{worklist} = \text{new } \text{Set}();
\text{for all node indexes } i \text{ do }
\begin{align*}
\text{input}[i] &= \bot_A; \\
\text{input}[\text{entry}] &= \iota_A; \\
\text{worklist.add(all nodes);}
\end{align*}
\]

\[
\text{while (!worklist.isEmpty()) do }
\begin{align*}
i &= \text{worklist.pop();} \\
\text{after} &= f_A(\text{input}[i], \text{node}(i)); \\
\text{for all } k \in \text{succ}(i) \text{ do }
\begin{align*}
\text{newinput} &= \text{input}[k] \sqcup \text{after} \\
\text{if } (!\text{newinput} \sqsubseteq \text{input}[k])) \\
\text{input}[k] &= \text{newinput;} \\
\text{worklist.add}(k);
\end{align*}
\end{align*}
\]
Worklist Dataflow Analysis Algorithm

\[
\text{worklist} = \text{new Set(); for all node indexes } i \text{ do}
\]
\[
\text{input}[i] = \perp_A;
\]
\[
\text{input}[\text{entry}] = \iota_A;
\]
\[
\text{worklist.add(all nodes);}
\]

\[
\text{while (!worklist.isEmpty()) do}
\]
\[
i = \text{worklist.pop();}
\]
\[
\text{after} = f_A(\text{input}[i], \text{node}(i));
\]
\[
\text{for all } k \in \text{succ}(i) \text{ do}
\]
\[
\text{newinput} = \text{input}[k] \sqcup \text{after}
\]
\[
\text{if } !(\text{newinput} \sqsubseteq \text{input}[k])
\]
\[
\text{input}[k] = \text{newinput};
\]
\[
\text{worklist.add(k);}
\]

Ok to just add entry node if flow functions cannot return \( \perp_A \) (examples will assume this)
Worklist Dataflow Analysis Algorithm

```
worklist = new Set();
for all node indexes i do
    input[i] = ⊥_A;
input[entry] = ι_A;
worklist.add(all nodes);

while (!worklist.isEmpty()) do
    i = worklist.pop();
    after = f_A(input[i], node(i));
    for all k ∈ succ(i) do
        newinput = input[k] ⊔ after
        if (!(newinput ⊑ input[k]))
            input[k] = newinput;
            worklist.add(k);
```

Ok to just add entry node if flow functions cannot return ⊥_A (examples will assume this)

Pop removes the most recently added element from the set (performance optimization)
**Example of Worklist**

\[
\begin{align*}
[a := 0]_1 \\
[b := 0]_2 \\
\text{while } [a < 2]_3 \text{ do} \\
\quad [b := a]_4; \\
\quad [a := a + 1]_5; \\
[a := 0]_6
\end{align*}
\]

Control Flow Graph

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \\
\quad 4 \rightarrow 5
\]
### Example of Worklist

\[
egin{align*}
[a := 0]_1 \\
[b := 0]_2 \\
\text{while } [a < 2]_3 \text{ do} \\
\quad [b := a]_4; \\
\quad [a := a + 1]_5; \\
[a := 0]_6
\end{align*}
\]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4,6</td>
</tr>
<tr>
<td>4</td>
<td>5,6</td>
</tr>
<tr>
<td>5</td>
<td>3,6</td>
</tr>
<tr>
<td>3</td>
<td>4,6</td>
</tr>
<tr>
<td>4</td>
<td>5,6</td>
</tr>
<tr>
<td>5</td>
<td>3,6</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>2</td>
<td>Z</td>
<td>MZ</td>
</tr>
<tr>
<td>3</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>5</td>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td>6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
</tbody>
</table>

Control Flow Graph

1 → 2 → 3 → 6

4 ← 5
Quick Quiz

Show how the worklist algorithm given in class operates on the program given, by filling in the table below.

1: x := 0
2: y := 1
3: if (z == 0)
   4: x := x + y
5: else y := y – 1
6: w := y

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>x</th>
<th>y</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Worklist Algorithm Performance

```java
worklist = new Set();
for all node indexes i do
    input[i] = ⊥A;
input[entry] = ιA;
worklist.add(all nodes);

while (!worklist.isEmpty()) do
    i = worklist.pop();
    after = fA(input[i], node(i));
    for all k ∈ succ(i) do
        newinput = input[k] ⊔ after
        if (!(newinput ⊑ input[k]))
            input[k] = newinput;
            worklist.add(k);
```

- How many times might a node get added to the worklist?
  - The node’s input must increase each time
  - The number of increases is bound by the height $h$ of the lattice

- How many times do statements execute?
  - $h*n$ in total: we may run it $h$ times for each node $n$
  - but we must propagate along all successor edges; these statements execute $h*e$ times
  - Assume statement cost is $c$
    - Then performance is $O(h*e*c)$
    - Often $h$, $e$, and $c$ are bounded by $n$. So we get $O(n^3)$
    - Good enough to run on a function, but not on the whole program
Outline

• Dataflow Analysis Frameworks
  • Lattices
  • Abstraction functions
  • Control flow graphs
  • Flow functions
  • Worklist algorithm

• Example Dataflow Analyses
  • Constant Propagation
  • Reaching Definitions
  • Live Variable Analysis

• Dataflow Analysis Correctness
  • Termination
  • Soundness
Constant Propagation

• Goal: determine which variables hold a constant value:

\[
\begin{aligned}
x &:= 3; \\
y &:= x+7; \\
\text{if } b & \quad \text{then } z := x+2 \\
& \quad \text{else } z := y-5; \\
w &:= z-2
\end{aligned}
\]

• What is w?
  • Useful for optimization, error checking
  • Zero analysis is a special case
Constant Propagation Definition

- Constant lattice \((L_C, \sqsubseteq_C, \sqcup_C, \bot, \top)\)
  - \(L_C = \text{Integer} \mid \{ \bot, \top \}\)
  - \(\forall n \in \text{Integer}: \bot \sqsubseteq_C n \&\& n \sqsubseteq_C \top\)

- Constant propagation lattice
  - Tuple lattice formed from above lattice
  - See notes on zero analysis for details

- Abstraction function:
  - \(\alpha_C(n) = n\)
  - \(\alpha_{CP}(\eta) = \{ x \mapsto \alpha_C(\eta(x)) | x \in \text{Var} \}\)

- Initial data:
  - \(\iota_{CP} = \{ x \mapsto \top | x \in \text{Var} \}\)
Constant Propagation Definition

- \( f_{CP}(\sigma, [x := y]) = [x \mapsto \sigma(y)] \sigma \)
- \( f_{CP}(\sigma, [x := n]) = [x \mapsto n] \sigma \)
- \( f_{CP}(\sigma, [x := y \ op \ z]) = [x \mapsto (\sigma(y) \ op_\top \ \sigma(z))] \sigma \)
  - \( n \ op_\top m = n \ op \ m \)
  - \( n \ op_\top \top = \top \)
  - \( \top \ op_\top m = \top \)

- **Note:** we could define for \( \bot \) too, but we won’t actually ever see \( \bot \) during analysis

- \( f_{CP}(\sigma, /* \text{any other */}) = \sigma \)
Constant Propagation Example

\[ [x := 3]_1; \]
\[ [y := x + 7]_2; \]
\[ \text{if } [b]_3 \text{ then } [z := x + 2]_4 \]
\[ \text{else } [z := y - 5]_5; \]
\[ [w := z - 2]_6 \]
Constant Propagation Example

\[ x := 3 \]
\[ y := x + 7 \]
\[ \text{if } [b] \text{ then } [z := x + 2] \]
\[ \text{else } [z := y - 5] \]
\[ w := z - 2 \]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>( x )</th>
<th>( y )</th>
<th>( z )</th>
<th>( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4, 5</td>
<td>3</td>
<td>10</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>4</td>
<td>6, 5</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>( T )</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>( T )</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>3</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>
Constant Propagation Example

\[ \begin{align*}
[x := 3]_1; \\
[y := x+7]_2; \\
\text{if } [b]_3 \\
\quad \text{then } [z := x+1]_4 \\
\quad \text{else } [z := y-5]_5; \\
[w := z-2]_6
\end{align*} \]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>T</td>
<td>T</td>
<td>10</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>4,5</td>
<td>3</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Constant Propagation Example

\[
\begin{align*}
[x := 3]_1; \\
[y := x+7]_2; \\
\text{if } [b]_3 \\
\quad \text{then } [z := x+1]_4 \\
\quad \text{else } [z := y-5]_5; \\
[w := z-2]_6
\end{align*}
\]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>10</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>4,5</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>6,5</td>
<td></td>
<td></td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>3</td>
<td>10</td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>
### Loss of Precision

If \([x = 0]_1\)

then \([y := 1]_2;\)

else \([y := x]_3;\)

\([z := 10/y]_4\)

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>MZ</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>1</td>
<td>2,3</td>
<td>MZ</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>2</td>
<td>4,3</td>
<td>MZ</td>
<td>NZ</td>
<td>MZ</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>MZ</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>MZ</td>
<td>NZ</td>
<td>NZ</td>
</tr>
</tbody>
</table>
Branch Sensitivity for Zero Analysis

- Existing flow functions
  - \( f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\sigma \)
  - \( f_{ZA}(\sigma, [x := n]) = \text{if } n==0 \text{ then } [x \mapsto Z]\sigma \)
    \quad \text{else } [x \mapsto NZ]\sigma
  - \( f_{ZA}(\sigma, [x := y \text{ op } z]) = [x \mapsto MZ]\sigma \)
  - \( f_{ZA}(\sigma, /* \text{ any other */}) = \sigma \)
Branch Sensitivity for Zero Analysis

• **Existing flow functions**
  - \( f_{ZA}^T(\sigma, [x := y]) = [x \mapsto \sigma(y)] \sigma \)
  - \( f_{ZA}^F(\sigma, [x := y]) = [x \mapsto \neg\sigma(y)] \sigma \)

• **Propagate different info on branches**
  - \( f_{ZA}^T(\sigma, [x = 0]) = [x \mapsto Z] \sigma \)
  - \( f_{ZA}^F(\sigma, [x = 0]) = [x \mapsto NZ] \sigma \)
  - Slightly more general:
    - Assume \( \neg Z = NZ; \neg NZ = Z; \neg MZ = MZ \)

- \( f_{ZA}^T(\sigma, [x := n]) = \begin{cases} [x \mapsto Z] \sigma & \text{if } n == 0 \\ [x \mapsto NZ] \sigma & \text{else} \end{cases} \)

- \( f_{ZA}(\sigma, [x := y \text{ op } z]) = [x \mapsto MZ] \sigma \)
- \( f_{ZA}(\sigma, /* \text{ any other */}) = \sigma \)
### Precision Regained

Worklist simplified to the statement level

\[ \text{if } [x = 0]_1 \]
\[ \text{then } [y := 1]_2; \]
\[ \text{else } [y := x]_3; \]
\[ [z := 10/y]_4 \]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>MZ</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>1(^T)</td>
<td>2,3</td>
<td>Z</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>1(^F)</td>
<td>2,3</td>
<td>NZ</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>2 (use 1(^T))</td>
<td>4,3</td>
<td>Z</td>
<td>NZ</td>
<td>MZ</td>
</tr>
<tr>
<td>3 (use 1(^F))</td>
<td>4</td>
<td>NZ</td>
<td>NZ</td>
<td>MZ</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><strong>MZ</strong></td>
<td><strong>NZ</strong></td>
<td><strong>NZ</strong></td>
</tr>
</tbody>
</table>
Outline

• Dataflow Analysis Frameworks
  • Lattices
  • Abstraction functions
  • Control flow graphs
  • Flow functions
  • Worklist algorithm

• Example Dataflow Analyses
  • Constant Propagation
  • Reaching Definitions
  • Live Variable Analysis

• Dataflow Analysis Correctness
  • Termination
  • Soundness
Reaching Definitions Analysis

• Goal: determine which is the most recent assignment to a variable that precedes its use:

\[ y := x \]
\[ z := 1 \]
while \[ y > 1 \]
do
\[ z := z \times y \]
\[ y := y - 1 \]
\[ y := 0 \]

• Example: definitions 1 and 5 reach the use of y at 4

• Applications
  • Simpler version of constant propagation, zero analysis, etc.
  • Just look at the reaching definitions for constants
  • If definitions reaching use include “undefined” sentinel, then we may be using an undefined variable
Reaching Definitions

Set Lattice \((P^{\text{DEFS}}, \sqsubseteq_{\text{RD}}, \sqcup_{\text{RD}}, \emptyset, \text{DEFS})\)
- \text{DEFS} is the set of definitions in the program
- Each element of the lattice is a subset of \text{defs}
  - \(P^{\text{DEFS}}\) is the powerset of \text{DEFS}, i.e. the set of all subsets of \text{DEFS}
- Approximation
  - A definition \(d\) may reach program point \(P\) if \(d\) is in the lattice at \(P\)
  - We call this a \textit{may analysis}
- \(x \sqsubseteq_{\text{RD}} y\) iff \(x \subseteq y\)
- \(x \sqcup_{\text{RD}} y = x \cup y\)
  - This is a direct consequence of the definition of \(\sqsubseteq_{\text{RD}}\)
- Most precise element \(\perp = \emptyset\) (no reaching definitions)
- Least precise element \(\top = \text{DEFS}\) (all definitions reach)
Reaching Definitions

- Initially assume dummy assignments
  - Represents passed values for parameters
  - Represents uninitialized for non-parameters
  - \( \nu_{RD} = \{ x_0 \mid x \in \text{Var} \} \)

- Flow functions
  - \( f_{RD}(\sigma, [x := \ldots]_k) \)
    \[
    = \sigma - \{ x_m \mid x_m \in \text{DEFS}(x) \} \cup \{ x_k \}
    \]
  - Kills (removes from set) all other definitions of \( x \)
  - Generates (adds to set) the current definition \( x_k \)
  - Kill/Gen pattern true in many analyses with set lattices
  - \( f_{RD}(\sigma, /* \text{any other} */ \) = \( \sigma \)
Reaching Definitions Example

\[ y := x \] \(_1\);
\[ z := 1 \] \(_2\);
\text{while } [y>1] \(_3\) do
\[ z := z \times y \] \(_4\);
\[ y := y - 1 \] \(_5\);
\[ y := 0 \] \(_6\);
Reaching Definitions Example

\[ y := x \]_1;
\[ z := 1 \]_2;
while \[ y > 1 \]_3 do
  \[ z := z \times y \]_4;
  \[ y := y - 1 \]_5;
\[ y := 0 \]_6;

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>Lattice Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>{x_0, y_0, z_0}</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>{x_0, y_1, z_0}</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>{x_0, y_1, z_2}</td>
</tr>
<tr>
<td>3</td>
<td>4,6</td>
<td>{x_0, y_1, z_2}</td>
</tr>
<tr>
<td>4</td>
<td>5,6</td>
<td>{x_0, y_1, z_4}</td>
</tr>
<tr>
<td>5</td>
<td>3,6</td>
<td>{x_0, y_5, z_4}</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>{x_0, y_6, z_2, z_4}</td>
</tr>
</tbody>
</table>
Outline

• Dataflow Analysis Frameworks
  • Lattices
  • Abstraction functions
  • Control flow graphs
  • Flow functions
  • Worklist algorithm

• Example Dataflow Analyses
  • Constant Propagation
  • Reaching Definitions
  • Live Variable Analysis

• Dataflow Analysis Correctness
  • Termination
  • Soundness
Live Variables Analysis

- Goal: determine which variables may be used again before they are redefined (i.e. are live) at the current program point:

\[
\begin{align*}
[y := x]_1; \\
[z := 1]_2; \\
\text{while } [y > 1]_3 \\ &\quad [z := z \times y]_4; \\
&\quad [y := y - 1]_5; \\
[y := 0]_6;
\end{align*}
\]

- Example: after statement 1, y is live, but x and z are not
- Optimization applications
  - If a variable is not live after it is defined, can remove the definition statement (e.g. 6 in the example)
Live Variables Definition

- Set Lattice \((P^{\text{Vars}}, \sqsubseteq_{\text{LV}}, \sqcup_{\text{LV}}, \emptyset, \text{Vars})\)
  - \text{Vars} is the set of variables in the program
  - Each element of the lattice is a subset of \text{Vars}
    - \(P^{\text{Vars}}\) is the powerset of \text{Vars}, i.e. the set of all subsets of \text{Vars}
    - \(x \sqsubseteq_{\text{LV}} y\) iff \(x \subseteq y\)
    - \(x \sqcup_{\text{LV}} y = x \cup y\)
  - Most precise element \(\bot = \emptyset\) (no live variables)
  - Least precise element \(\top = \text{DEFS}\) (all variables live)

\[\text{Vars} = \{x, y, z\}\]

- \{x, y\}  \{x, z\}  \{y, z\}
- \{x\}  \{y\}  \{z\}
- \emptyset
Live Variables Definition

- Live Variables is a *backwards* analysis
  - To figure out if a variable is live, you have to look at the future execution of the program
- Will x be used before it is redefined?
  - When x is defined, assume it is not live
  - When x is used, assume it is live
  - Propagate lattice elements as usual, except backwards
- Initially assume return value is live
  - $\iota_{LV} = \{ x \}$ where $x$ is the variable returned from the function
Flow Function Practice

• Write flow functions for Live Variable analysis:

  \[ f_{\text{LV}}(\sigma, [x := e]_k) = \]

  \[ f_{\text{LV}}(\sigma, /* any other */) = \]
Flow Function Practice

• Write flow functions for Live Variable analysis:

  \[ f_{LV}(\sigma, [x := e]_k) = (\sigma - \{ x \}) \cup \text{vars}(e) \]
  - Kills (removes from set) the variable \( x \)
  - Generates (adds to set) the variables in \( e \)
  - Note: must kill first then generate (what if \( e = x \)?)

  \[ f_{LV}(\sigma, [e]_k) = \sigma \cup \text{vars}(e) \]

  \[ f_{LV}(\sigma, /* \text{any other } */ \}) = \sigma \]
Worklist Practice

Show how the worklist algorithm given in class operates on the program given, by filling in the table below.

\[
\begin{align*}
& [y := x]_1; \\
& [z := 1]_2; \\
& \text{while } [y>1]_3 \text{ do} \\
& \quad [z := z \times y]_4; \\
& \quad [y := y - 1]_5; \\
& [y := 0]_6; \\
& \text{return } z;
\end{align*}
\]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>Lattice Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Live Variables Example

\[y := x\]_1;  
\[z := 1\]_2;  
while \([y>1]\)_3 do  
\[z := z \times y\]_4;  
\[y := y - 1\]_5;  
\[y := 0\]_6;  
return z;

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>Lattice Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>6</td>
<td>{z}</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>{z}</td>
</tr>
<tr>
<td>3</td>
<td>5,2</td>
<td>{y,z}</td>
</tr>
<tr>
<td>5</td>
<td>4,2</td>
<td>{y,z}</td>
</tr>
<tr>
<td>4</td>
<td>3,2</td>
<td>{y,z}</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>{y}</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>{x}</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

• Dataflow Analysis Frameworks
  • Lattices
  • Abstraction functions
  • Control flow graphs
  • Flow functions
  • Worklist algorithm

• Example Dataflow Analyses
  • Constant Propagation
  • Reaching Definitions
  • Live Variable Analysis

• Dataflow Analysis Correctness
  • Termination
  • Soundness
What does Correctness Mean?

- Intuition
  - Analysis will eventually terminate at a fixed point
  - At a fixed point, analysis results are a sound abstraction of program execution
  - program execution must be formally defined
  - abstraction function relates program execution to data flow lattice elements
  - sound means truth \( \sqsubseteq \) analysis results
    - also called conservative or safe
Termination

• Intuition
  • Dataflow information for a statement gets less precise every time we visit the statement
  • Information can only get less precise as many times as the lattice is high
  • When information stops changing, we stop

• Key property: Monotonic flow functions
  • $f$ is monotone iff $\sigma \sqsubseteq \sigma'$ implies $f(\sigma) \sqsubseteq f(\sigma')$
Nonterminating Analysis

(bad) idea: Track set of values for each variable

\[
\begin{align*}
[x := 0]_1 & \\
\text{while } [x < y]_2 \text{ do} & \\
[x := x + 1;]_3 & \\
[x := 0]_4;
\end{align*}
\]

\textit{Moral: make your lattices finite height!}
Dataflow Analysis Termination

- Theorem: If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate
Dataflow Analysis Termination

- Theorem: If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate.

- Lemma: Each time a node is added to the worklist, a dataflow value has increased (and no dataflow value has decreased).
  - Proof outline: by induction
    - Base case: The dataflow value for every statement is $\bot$. This is the lowest point in the lattice. Thus the first time the value changes, it will increase.
    - Inductive case: Assume the last application of the dataflow function mapped $\sigma$ to $f(\sigma)$. By assumption $\sigma \sqsubseteq \sigma'$. By monotonicity $f(\sigma) \sqsubseteq f(\sigma')$. Thus the output value increased.
  - Will not affect others because only the flow value for the current statement is set.
Dataflow Analysis Termination

- Theorem: If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate
  - Proof outline: by induction
    - Base case: The dataflow value for every statement is ⊥. This is the lowest point in the lattice. Thus the first time the value changes, it will increase.
    - Inductive case: Assume the last application of the dataflow function mapped $\sigma$ to $f(\sigma)$. By assumption $\sigma \sqsubseteq \sigma'$. By monotonicity $f(\sigma) \sqsubseteq f(\sigma')$. Thus the output value increased.
  - Will not affect others because only the flow value for the current statement is set.

- Proof outline for theorem:
  - Each time a node is added to the worklist, the dataflow value was raised in the lattice for one statement.
  - If there are $n$ statements in the program and the height of the lattice is $h$, this can happen at most $n \times h$ times.
  - An inspection of the worklist algorithm shows that a finite number of steps occurs between applications of flow functions, and that when the values stop changing the algorithm terminates.
Outline

• Dataflow Analysis Frameworks
• Lattices
• Abstraction functions
• Control flow graphs
• Flow functions
• Worklist algorithm

Example Dataflow Analyses
• Constant Propagation
• Reaching Definitions
• Live Variable Analysis

Dataflow Analysis Correctness
• Termination
• Soundness
Dataflow Analysis Soundness

• Intuition
  • The result of dataflow analysis is a conservative approximation of all possible run time states

• Definition
  • A dataflow analysis is sound if, for all programs P, for all inputs I, for all times T in P’s execution on input I,
  • If P is at statement S at time T, with memory $\eta$, and the analysis returned abstract state $\sigma$ for S,
  • then $\alpha(\eta) \subseteq \sigma$
Local Soundness

\[ \alpha_{DF}(\eta_i) \subseteq \sigma_i \xrightarrow{f_{DF}(\sigma_i, S)} \sigma_o \supseteq \alpha_{DF}(\eta_o) \]

- Local correctness condition for dataflow analysis
  - If applying a transfer function to statement \( S \) and input information \( \sigma_i \) yields output information \( \sigma_o \),
  - Then for all program states \( \eta_i \) such that \( \alpha(\eta_i) \subseteq \sigma_i \) and executing \( S \) in state \( \eta_i \) yields state \( \eta_o \),
  - We must have \( \alpha(\eta_o) \subseteq \sigma_o \)

Intuitively, translating from concrete to abstract and applying the flow function will safely approximate (\( \supseteq \)) taking a step in the trace and translating from concrete to abstract
Finding Errors with Local Soundness

- Consider the incorrect flow function:
  \[ f_{ZA}(\sigma, [x := y \text{ op } z]) = \]
  \[
  \begin{cases} 
  \sigma[y]=Z \mid \sigma[z]=Z & \text{then } [x \mapsto Z] \sigma \\
  \text{else } [x \mapsto MZ] \sigma 
  \end{cases}
  \]

- Challenge: find an example where local soundness fails
Finding Errors with Local Soundness

- Consider the incorrect flow function:
  \[ f_{ZA}(\sigma, [x := y \ op z]) = \]
  \[
  \begin{align*}
  &\text{if } \sigma[y] = Z \text{ or } \sigma[z] = Z \\
  &\text{then } [x \mapsto Z] \sigma \\
  &\text{else } [x \mapsto MZ] \sigma
  \end{align*}
  \]

- Local Soundness failure:
  - Let \( \sigma_i = [] \), \( S = \text{"x := 3+0"} \)
  - Consider \( \eta_i = [] \). As required, \( \alpha_{DF}(\eta_i) = [] \sqsubseteq \sigma_i \)
  - Now \( \sigma_o = f_{DF}(\sigma_i, S) = [x \mapsto Z] \)
  - And \( \eta_o = S(\eta_i) = [x \mapsto 3] \)
  - So \( \alpha_{DF}(\eta_o) = \alpha_{DF}([x \mapsto 3]) = [x \mapsto NZ] \)
  - BUT \( \alpha_{DF}(\eta_o) \not\sqsubseteq \sigma_o \) because \( Z \not\sqsubseteq NZ \), so local soundness is violated
Proving Correctness

- Consider a Zero Analysis flow function
  \[ f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)] \sigma \]

- Monotonicity
  - Assume \( \sigma' \sqsubseteq \sigma \).
  - \( f_{ZA}(\sigma, [x := y]) \) changes only the value for \( x \)
  - Therefore for all variables \( z \neq x \) we have
    \[ f_{ZA}(\sigma', [x := y])(z) \sqsubseteq f_{ZA}(\sigma, [x := y])(z) \]
  - Since \( f_{ZA}(\sigma, [x := y])(x) = \sigma(y) \) and \( \sigma'(y) \sqsubseteq \sigma(y) \) we have \( f_{ZA}(\sigma', [x := y])(x) \sqsubseteq f_{ZA}(\sigma, [x := y])(x) \)
  - Thus \( f_{ZA}(\sigma', [x := y]) \sqsubseteq f_{ZA}(\sigma, [x := y]) \)
Proving Correctness

• Consider a Zero Analysis flow function
  \[ f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)] \sigma \]

• Local Soundness
  • Assume \( \alpha(\eta_i) \subseteq \sigma_i \).
  • By Java’s semantics \( \eta_o = [x \mapsto \eta_i(y)] \eta_i \)
  • By the flow function, \( \sigma_o = [x \mapsto \sigma_i(y)] \sigma_i \)
  • Since both maps changed only in their \( x \) value, for all variables \( z \neq x \) we have \( \alpha_{DF}(\eta_o)(z) \subseteq \sigma_o(z) \)
  • Since \( \alpha(\eta_i)(y) \subseteq \sigma_i(y) \), \( \alpha(\eta_o)(x) = \alpha(\eta_i)(y) \), and \( \sigma_o(y) = \sigma_i(y) \), we also know that \( \alpha(\eta_o)(x) \subseteq \sigma_o(x) \)
  • Thus \( \alpha(\eta_o) \subseteq \sigma_o \)
Global Soundness

- **Intuition**
  - We begin with initial dataflow facts $\iota$ that safely approximate ($\supset$) all initial stores $\eta_0$
  - By local soundness, each transfer function when given safe input information yields safe output information
  - By induction, any fixed point of the analysis is sound
Why care about Soundness?

• Analysis Producers
  • Writing analyses is hard
  • People make mistakes all the time
  • Need to know how to \textit{think} about correctness
  • When the analysis gets tricky, it’s useful to prove it correct formally

• Analysis Consumers
  • Sound analysis provides guarantees
    • Optimizations won’t break the program
    • Finds all defects of a certain sort
  • Decision making
    • Knowledge of soundness techniques lets you ask the right questions about a tool you are considering
    • Soundness affects where you invest QA resources
      • Focus testing efforts on areas where you don’t have a sound analysis!