Dataflow Analysis Frameworks

17-654/17-754
Analysis of Software Artifacts

Jonathan Aldrich

Worklist Dataflow Analysis Algorithm

worklist = new Set();
for all node indexes i do
  results[i] = ⊥\textsubscript{A};
  results[\textit{entry}] = \textit{ι};
  worklist.add(\textit{all nodes});

while (!worklist.isEmpty()) do
  i = worklist.pop();
  before = \bigsqcup_{k \in \text{pred}(i)} results[k];
  after = f\textsubscript{A}(before, node(i));
  if (!(after \sqsubseteq results[i]))
    results[i] = after;
  for all k \in \text{succ}(i) do
    worklist.add(k);

Ok to just add entry node if flow functions cannot return \bot\textsubscript{A} (examples will assume this)

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

```
[a := 0]_1
[b := 0]_2
while [a < 2]_3 do
  [b := a]_4;
  [a := a + 1]_5;
[a := 0]_6
```

<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>6</td>
<td>4, 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>Z</td>
<td>MZ</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>MZ</td>
<td>MZ</td>
</tr>
</tbody>
</table>

**Control Flow Graph**

```
1 -> 2 -> 3 -> 6
```

---

### Dataflow Analysis in Crystal

**17-654/17-754**

**Analysis of Software Artifacts**

Jonathan Aldrich
Framework Operation

- User chooses Crystal | Run Analysis
- Fwk invokes MyMethodAnalysis.analyzeMethod(m)
- analyzeMethod(m) invokes FlowAnalysis.getResultsAfter(ast)
- FlowAnalysis.getResultsAfter(ast)
  - gets CFG node for ast (building CFG if necessary)
  - checks if lattice element exists for CFG node
  - If so, returns lattice element after node (done)
  - If not, runs analysis (continue below)
- Running the analysis
  - Worklist algorithm
  - In reality, implemented as a visitor, but this isn't important to clients
  - Uses MyFlowAnalysisDefinition.getLattice()
    - Lattice.entry() for lattice element at beginning of method
    - Lattice.bottom() for lattice element at loop back edges
  - Uses MyFlowAnalysisDefinition.transfer(a,v)
    - Propagates information across AST node
  - Uses MyLatticeElement
    - atLeastAsPrecise() to detect a fixed point
    - join() to merge data from two CFG branches
    - copy() before each join() or transfer()
Zero Lattice

```java
public class ZeroLatticeElement extends LatticeElement<ZeroLatticeElement> {

  private final String name;

  private ZeroLatticeElement(String n) {
    name = n;
  }

  // lattice element constants
  static final ZeroLatticeElement MZ = new ZeroLatticeElement("MZ");
  static final ZeroLatticeElement bottom = new ZeroLatticeElement("bottom");
  static final ZeroLatticeElement Z = new ZeroLatticeElement("Z");
  static final ZeroLatticeElement NZ = new ZeroLatticeElement("NZ");

  static final Lattice<ZeroLatticeElement> lattice
    = new Lattice<ZeroLatticeElement>(MZ, bottom);

  // lattice element constants
  public boolean atLeastAsPrecise(ZeroLatticeElement other) {
    // true if elements equal
    if (other == this)
      return true;
    // bottom more precise than any other
    else if (this == bottom)
      return true;
    // top less precise than any other
    else if (other == MZ)
      return true;
    // otherwise other is more precise, or no relationship
    else
      return false;
  }
}
```
Zero Lattice

```java
public ZeroLatticeElement join(ZeroLatticeElement other) {
    // join of equal elements is the element
    if (other == this)
        return this;
    // join of X and bottom is X
    else if (other == bottom)
        return this;
    else if (this == bottom)
        return other;
    // any other join is top (MZ)
    else
        return MZ;
}
```

// since our lattice elements are immutable, copying returns this
public ZeroLatticeElement copy() {
    return this;
}
```

Tuple Lattice

```java
public class TupleLatticeElement<LE extends LatticeElement<LE>>
    extends LatticeElement<TupleLatticeElement<LE>> {

    private final LE bot;
    private final LE theDefault;
    // if elements==null, then this element is the bottom tuple lattice
    private final HashMap<ASTNode,LE> elements;

    /** returns bottom if this lattice is bottom, theDefault if n not found in map */
    public LE get(ASTNode n) {
        if (elements == null)
            return bot;
        LE elem = elements.get(n);
        if (elem == null)
            return theDefault;
        else
            return elem;
    }

    public LE put(ASTNode n, LE l) { return elements.put(n,l); }
}
```

public TupleLatticeElement<LE> join(TupleLatticeElement<LE> other) {
    HashMap<ASTNode,LE> newMap = new HashMap<ASTNode,LE>();
    Set<ASTNode> keys = new HashSet(getKeySet());
    keys.addAll(other.getKeySet());

    // join the tuple lattice by joining each element
    for (ASTNode key : keys) {
        LE myLE = get(key);
        LE otherLE = other.get(key);
        LE newLE = myLE.join(otherLE);
        newMap.put(key, newLE);
    }

    return new TupleLatticeElement<LE>(bot, theDefault, newMap);
}

public boolean atLeastAsPrecise(TupleLatticeElement<LE> other) {
    Set<ASTNode> keys = new HashSet(getKeySet());
    keys.addAll(other.getKeySet());

    // elementwise comparison: return false if any element is not atLeastAsPrecise
    for (ASTNode key : keys) {
        LE myLE = get(key);
        LE otherLE = other.get(key);
        if (!myLE.atLeastAsPrecise(otherLE))
            return false;
    }

    return true;
}

public TupleLatticeElement<LE> copy() {
    return new TupleLatticeElement<LE>(varLattice,
        (HashMap<ASTNode, LE>) (elements==null ? null : elements.clone()));
}
Zero Analysis Definition

```java
public class DBZTransferMethods extends AbstractingTransferFunction<TupleLatticeElement<ZeroLatticeElement>> {

    public Lattice<TupleLatticeElement<ZeroLatticeElement>> getLattice(IMethodDeclarationNode d) {
        TupleLatticeElement<ZeroLatticeElement> entry = new TupleLatticeElement<ZeroLatticeElement>(
            ZeroLatticeElement.bottom, ZeroLatticeElement.MZ);

        return new Lattice<TupleLatticeElement<ZeroLatticeElement>>(
            entry, entry.bottom());
    }

    // if we are visiting a divide or modulus operation
    if (binop.getOperator().equals(BinaryOperation.BinaryOperator.ARIT_DIVIDE) ||
        binop.getOperator().equals(BinaryOperation.BinaryOperator.ARIT_MODULO)) {

        // get the lattice element for the second operand (the divisor)
        DivideByZeroLatticeElement cur_val = value.get(binop.getOperand2());

        // note an error or warning if the divisor is definitely or possibly zero
        if (cur_val.equals(DivideByZeroLatticeElement.ZERO)) {
            problems.put(binop, DivideByZeroLatticeElement.ZERO);
        } else if (cur_val.equals(DivideByZeroLatticeElement.MAYBEZERO)) {
            problems.put(binop, DivideByZeroLatticeElement.MAYBEZERO);
        }
    }

    return super.transfer(binop, value);
}
```
Zero Analysis Definition

/** A copy instruction copies the lattice value for the source
 * variable to the target variable.
 */
@override
public TupleLatticeElement<DivideByZeroLatticeElement> transfer(CopyInstruction instr,
        TupleLatticeElement<DivideByZeroLatticeElement> value) {
    value.put(instr.getTarget(), value.get(instr.getOperand()));
    return value;
}

/** Assignment instructions (other than those explicitly defined
 * with other flow functions) set the result to maybe zero.
 */
@override
public TupleLatticeElement<DivideByZeroLatticeElement> transfer(AssignmentInstruction
        instr, TupleLatticeElement<DivideByZeroLatticeElement> value) {
    value.put(instr.getTarget(), DivideByZeroLatticeElement.MAYBEZERO);
    return value;
}

IntLiteralNode literal = (IntLiteralNode)instr.getLiteral();

if( Integer.parseInt(literal.getToken()) == 0 )
    value.put(instr.getTarget(), DivideByZeroLatticeElement.ZERO);
else
    value.put(instr.getTarget(), DivideByZeroLatticeElement.NONZERO);
return value;
}

else {
    // If it's not an integer literal, handle as usual
    return super.transfer(instr, value);
}
}
Crystal Tricks

- Cache warning messages
  - Can generate during analysis
  - Don’t report right away
    - Analysis may visit a node multiple times
    - Don’t want to report multiple identical warnings!

- Use AbstractingTransferFunction
  - Default implementation of transfer function for a node calls transfer function for node’s superclass
    - e.g. anything that assigns to a variable calls the transfer function for assignment
  - Lets you define a generic case for assignments, and override where needed

Dataflow Analysis Example:
Constant Propagation

17-654/17-754
Analysis of Software Artifacts

Jonathan Aldrich
Constant Propagation

- Goal: determine which variables hold a constant value:

```plaintext
x := 3;
y := x + 7;
if b
   then z := x + 2
   else z := y - 5;
w := z - 2
```

- What is w?
  - Useful for optimization, error checking
  - Zero analysis is a special case

Constant Propagation Definition

- Constant lattice \( (L_C, \sqsubseteq_C, \cup_C, \bot, \top) \)
  - \( L_C = \text{Integer} \cup \{ \bot, \top \} \)
  - \( \forall n \in \text{Integer} : \bot \sqsubseteq_C n \land n \sqsubseteq_C \top \)
- Constant propagation lattice
  - Tuple lattice formed from above lattice
  - See notes on zero analysis for details
- Abstraction function:
  - \( \alpha_C(n) = n \)
  - \( \alpha_{CP}(\eta) = \{ x \mapsto \alpha_C(\eta(x)) \mid x \in \text{Var} \} \)
- Initial data:
  - \( \iota_{CP} = \{ x \mapsto \top \mid x \in \text{Var} \} \)
Constant Propagation Definition

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

<table>
<thead>
<tr>
<th></th>
<th>Position</th>
<th>Worklist</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x := 3]_1;</td>
<td>0</td>
<td>1</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>[y := x+7]_2;</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>if [b]_3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>then [z := x+2]_4</td>
<td>3</td>
<td>4,5</td>
<td>3</td>
<td>10</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>else [z := y-5]_5;</td>
<td>4</td>
<td>6,5</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>$\top$</td>
</tr>
<tr>
<td>[w := z-2]_6</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
## Constant Propagation Example

\[
\begin{array}{cccccccc}
\text{[x := 3]},_1; & \text{Position} & \text{Worklist} & \text{x} & \text{y} & \text{z} & \text{w} \\
& 0 & 1 & T & T & T & T & T \\
[y := x+7],_2; & 1 & 2 & 3 & T & T & T & T \\
\text{if [b]},_3 & 2 & 3 & 3 & 10 & T & T & T \\
& 3 & 4,5 & 3 & 10 & 4 & T & T \\
& 4 & 6,5 & 3 & 10 & 4 & 2 & T \\
& 5 & 6 & 3 & 10 & 5 & T & T \\
[w := z-2],_6 & 6 & 3 & 10 & T & T & T \\
\end{array}
\]

## Loss of Precision

\[
\begin{array}{cccccccc}
\text{if [x = 0]},_1 & \text{Position} & \text{Worklist} & \text{x} & \text{y} & \text{z} \\
& 0 & 1 & MZ & MZ & MZ \\
\text{then [y := 1]},_2; & 1 & 2,3 & MZ & MZ & MZ \\
\text{else [y := x]},_3; & 2 & 4,3 & MZ & \text{NZ} & MZ \\
\text{[z := 10/y]},_4 & 3 & 4 & MZ & \text{MZ} & MZ \\
& 4 & MZ & \text{MZ} & MZ & \text{NZ} \\
\end{array}
\]
Flow Sensitivity for Zero Analysis

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

- 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:
    - \( f_{ZA}^T(\sigma; x = y) = [x \mapsto \sigma(y)] \sigma \)
    - \( f_{ZA}^F(\sigma; x = y) = [x \mapsto \neg \sigma(y)] \sigma \)
      
        Assume \( !Z=\text{NZ};!\text{NZ}=Z;!Z=\text{MZ} \)

---

Precision Regained

Worklist simplified to the statement level

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{if } [x = 0]_1 & \text{Position} & \text{Worklist} & x & y & z \\
\hline
0 & 1 & \text{MZ} & \text{MZ} & \text{MZ} \\
1^T & 2,3 & Z & \text{MZ} & \text{MZ} \\
1^F & 2,3 & \text{NZ} & \text{MZ} & \text{MZ} \\
2 & \text{(use } 1^T) & 4,3 & Z & \text{NZ} & \text{MZ} \\
3 & 3 & Z & \text{NZ} & \text{NZ} \\
4 & \text{(use } 1^F) & 4 & \text{NZ} & \text{NZ} & \text{MZ} \\
\hline
\end{array}
\]
Dataflow Analysis Correctness

Software Analysis
LG Electronics Curriculum
Jonathan Aldrich

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 ⊑ 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 $\sigma \leq \sigma'$ implies $f(\sigma) \leq f(\sigma')$

Nonterminating Analysis

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

<table>
<thead>
<tr>
<th>Iter</th>
<th>Position</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>(0)</td>
<td>Z</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>(0)</td>
<td>Z</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>(1)</td>
<td>Z</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>(0,1)</td>
<td>Z</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>(1,2)</td>
<td>Z</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>(0,1,2)</td>
<td>Z</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>(1,2,3)</td>
<td>Z</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>(0,1,2,3)</td>
<td>Z</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>(1,2,3,4)</td>
<td>Z</td>
</tr>
</tbody>
</table>

[x := 0]_1
while [x < y]_2 do
[x := x + 1]_3;
[x := 0]_4;

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.
- **Lemma:** Each application of the flow function will increase some dataflow value (and not affect others).
  - **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 \( \sigma \leq \sigma' \). By monotonicity \( f(\sigma) \leq f(\sigma') \).
      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 application of a flow function raises the dataflow value 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 \( n \cdot h \) 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.

Worklist Algorithm Performance

- **Performance**
  - Visits node whenever input gets less precise
    - up to \( h = \) height of lattice
  - Propagates data along control flow edges
    - up to \( e = \) max outbound edges per node
  - Assume lattice operation cost is \( o \)
  - Overall, \( O(h \cdot e \cdot o) \)
    - Typically \( h, o, e \) bounded by \( n = \) number of statements in program
    - \( O(n^2) \) for many data flow analyses
    - \( O(n^2) \) if you assume a number of edges per node is small
  - **Good enough to run on a function**
    - Usually not run on an entire program at once, because \( n \) is too big
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 η, and the analysis returned abstract state σ for S,
  - then $\alpha(\eta) \subseteq \sigma$

Local Soundness

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

- Global soundness follows from local soundness by induction
  - Initial dataflow facts are sound
  - Each step in program execution maintains soundness invariant
Why care about Soundness?

- **Analysis Producers**
  - Writing analyses is hard
    - People make mistakes all the time
    - Need to know how to *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!

---

Additional Slides (for your reference)
Proving Soundness

- Formally define analysis
  - Including abstraction function
  - We already know how
- Formalize trace semantics
- Prove local soundness for flow functions
- Apply global soundness theorem

Execution Traces

- Sequence of \(<pp, \text{mem}>\) pairs
  - pp is a program point
  - Just after statement pp
  - mem is the state of variables in memory

<table>
<thead>
<tr>
<th></th>
<th>pp</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

[y := x]_1;  
[z := 1]_2;  
while [y>1]_3 do  
[z := z * y]_4;  
[y := y – 1]_5;  
[y := 0]_6;
Execution Traces

- Sequence of \(<pp,\text{mem}>\) pairs
- \(pp\) is a program point
  - Just after statement \(pp\)
- \(\text{mem}\) is the state of variables in memory

\[
\begin{array}{cccc}
\text{pp} & x & y & z \\
0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 \\
2 & 1 & 1 & 1 \\
3 & 1 & 1 & 1 \\
6 & 1 & 0 & 1 \\
\end{array}
\]

\[y := x_1;\]
\[z := 1_2;\]
while \([y>1]_3\) do
\[z := z \cdot y]_4;\]
\[y := y - 1]_5;\]
\[y := 0]_6;\]

Analysis of Software Artifacts - Spring 2007

--

Execution Traces

- Sequence of \(<pp,\text{mem}>\) pairs
- \(pp\) is a program point
  - Just after statement \(pp\)
- \(\text{mem}\) is the state of variables in memory

\[
\begin{array}{cccc}
\text{pp} & x & y & z \\
0 & 3 & 0 & 0 \\
1 & 3 & 3 & 0 \\
2 & 3 & 3 & 1 \\
3 & 3 & 3 & 1 \\
4 & 3 & 3 & 3 \\
5 & 3 & 2 & 3 \\
3 & 3 & 2 & 3 \\
4 & 3 & 2 & 6 \\
5 & 3 & 1 & 6 \\
3 & 3 & 1 & 6 \\
6 & 3 & 0 & 6 \\
\end{array}
\]

\[y := x_1;\]
\[z := 1_2;\]
while \([y>1]_3\) do
\[z := z \cdot y]_4;\]
\[y := y - 1]_5;\]
\[y := 0]_6;\]
Execution Traces

- Sequence of <pp,mem> pairs
  - pp is a program point
  - Just after statement pp
  - mem is the state of variables in memory

\[
\begin{align*}
y &:= x_1; \\
z &:= 1_2; \\
\text{while } [y>1]_3 \text{ do } \\
[z &:= z \times y]_4; \\
y &:= y - 1]_5; \\
y &:= 0]_6;
\end{align*}
\]

Repeat for all possible initial values of x,y,z!

\*

**WHILE** Traces, Formally

- A trace for program \(S_1\) and initial state \(\eta_0\) is either:
  - A finite sequence \((\eta_0, S_1), \ldots, (\eta_n, \text{skip})\)
    where \((\eta_i, S_i) \rightarrow (\eta_{i+1}, S_{i+1})\) for \(0 \leq i < n\)
  - An infinite sequence \((\eta_0, S_1), \ldots, (\eta_i, S_i), \ldots\)
    where \((\eta_i, S_i) \rightarrow (\eta_{i+1}, S_{i+1})\) for \(i \geq 0\)

- Slight notational simplification
  - We will abbreviate \((\eta_0, S_0), \ldots, (\eta_n, S_n)\)
    as \((\eta_0, \text{first}(S_0)), \ldots, (\eta_n, \text{first}(S_n))\)
    - \text{first} is the label of the first statement in \(S\)
    - Uses program counter labels instead of complete programs
What does Correctness Mean?

- **Intuition**
  - At a fixed point, analysis results are a *sound abstraction* of program execution

- **Soundness condition**
  - When data flow analysis reaches a fixed point $F$, then for all traces $T$ and all times $t$ in each trace, $\alpha(T(t)) \subseteq \sigma_{pp(T(t))}$ where $\sigma_{pp(T(t))}$ is the analysis results at $pp(T(t))$
  - **Constant propagation**
    - For trace on last slide with $t=10$
    - $\alpha_{CP}(T(10)) = \{x\to 3,y\to 0,z\to 6\}$
    - $\sigma_{pp(T(10))} = \sigma_6 = \{x\to T,y\to 0,z\to T\}$
    - $\{x\to 3,y\to 0,z\to 6\} \subseteq_{CP} \{x\to T,y\to 0,z\to T\}$
      - Because $3 \subseteq T$ and $0 \subseteq 0$ and $6 \subseteq T$ in the CP lattice
    - *To prove soundness, repeat for all times in all traces*

Local Soundness

To show:
- if $(\eta_0, S_0) \rightarrow (\eta_{i+1}, S_{i+1}) \in T$
  - and $\sigma_i = \alpha_{DF}(\eta_i)$
  - and $\sigma_{i+1} = f_{DF}(\sigma_i, \text{first}(S_i))$
  - then $\alpha_{DF}(\eta_{i+1}) \subseteq \sigma_{i+1}$

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

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

- Local Soundness fails!
  - Consider \( \eta_i = [], S_i = [x := 3+0] \)
  - \( \sigma_i = \alpha_{DF}(\eta_i) = \alpha_{DF}([]) = [] \)
  - \( \sigma_{i+1} = f_{DF}(\sigma_i, \text{ first}(S_i)) = [x \mapsto Z] \)
  - \( \alpha_{DF}(\eta_{i+1}) = \alpha_{DF}([x \mapsto 3]) = [x \mapsto NZ] \)
  - \( \alpha_{DF}(\eta_{i+1}) \notin \sigma_{i+1} \) because Z \( \notin \) NZ

Global Soundness

- Intuition
  - We begin with initial dataflow facts \( \iota \) that safely approximate (\( \sqsubseteq \)) 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
Global Soundness

- Theorem (Global Soundness)
  - Assume that $\forall T \in \text{traces}(S)$ $\alpha_{DF}(\eta_T) \subseteq \iota$ and that analysis DF is monotone and locally sound with respect to $\alpha_{DF}$
  - Then for any fixed point $DF_{fix}$ of DF on program $S$, $\forall T \in \text{traces}(S)$ $\forall t \in \text{times}(T)$ we have $\alpha_{DF}(\eta_T) \subseteq DF_{fix}(pp(T(t)))$

- Proof outline: For each trace $T$ we do induction on $t$
  - Induction hypothesis: $\alpha_{DF}(\eta_T) \subseteq DF_{fix}(pp(T(t)))$
  - Base case: $t=0$
    - By assumption $\alpha_{DF}(\eta_T) \subseteq \iota = DF_{fix}(pp(\eta_T))$
  - Inductive case: time $t$ and statement $S_i$
    - Simplifying assumption: straight-line control flow
    - By induction hypothesis we have $\alpha_{DF}(\eta_{t-1}) \subseteq DF_{fix}(pp(T(t-1)))$
    - By monotonicity of DF we have:
      - $f_{DF}(\alpha_{DF}(\eta_{t-1}), S_i) \subseteq f_{DF}(DF_{fix}(pp(T(t-1))), S_i)$
      - By local soundness we have $\alpha_{DF}(\eta_T) \subseteq f_{DF}(\alpha_{DF}(\eta_{t-1}), S_i)$
      - By transitivity we get $\alpha_{DF}(\eta_T) \subseteq f_{DF}(DF_{fix}(pp(T(t-1))), S_i)$
      - But $f_{DF}(DF_{fix}(pp(T(t-1))), S_i) = DF_{fix}(pp(T(t)))$ because it’s a fixed point
      - So we have $\alpha_{DF}(\eta_T) \subseteq DF_{fix}(pp(T(t)))$

Other Dataflow Analyses

- Traditional optimization analyses
  - Reaching Definitions
  - Live Variables
Reaching Definitions Analysis

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

\[ \begin{align*}
[y := x],_1; \\
[z := 1],_2; \\
\text{while } [y>_1, \text{do} \\
[z := z \ast y],_4; \\
[y := y - 1],_5; \\
[y := 0],_6; \\
\end{align*} \]

- 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 \( (\mathcal{P}(\text{DEFS}), \in_{\text{RD}}, \cup_{\text{RD}}, \emptyset, \text{DEFS}) \)
  - \( \text{DEFS} \) is the set of definitions in the program
  - Each element of the lattice is a subset of \( \text{defs} \)
  - \( \mathcal{P}(\text{DEFS}) \) 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}
  - \( x \in_{\text{RD}} y \) iff \( x \subseteq y \)
  - \( x \cup_{\text{RD}} y = x \cup y \)
  - This is a direct consequence of the definition of \( \in_{\text{RD}} \)
  - Most precise element \( \bot = \emptyset \) (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
  - \( \epsilon_{\text{RD}} = \{ x_0 \mid x \in \text{Var} \} \)

- Flow functions
  - \( f_{\text{RD}}(\sigma, [x := \ldots]) = \sigma - \{ x_m \mid x_m \in \text{DEFS} \} \cup \{ x_k \} \)
    - 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
  - \( f_{\text{RD}}(\sigma, \text{ /* any other */}) = \sigma \)

---

Reaching Definitions Example

\[
\begin{array}{c|cccc}
[x := x]_1; & \text{Position} & \text{Worklist} & \text{Lattice Element} \\
[z := 1]_2; & 0 & 1 & \{x_0, y_0, z_0\} \\
\text{while } [y>1]_3 \text{ do} & 1 & 2 & \{x_0, y_1, z_0\} \\
[z := z \ast y]_4; & 2 & 3 & \{x_0, y_1, z_1\} \\
[y := y - 1]_5; & 3 & 4, 6 & \{x_0, y_1, z_4\} \\
[y := 0]_6; & 4 & 5, 6 & \{x_0, y_5, z_4\} \\
\end{array}
\]
Live Variables Analysis

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

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

- 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

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

- Set Lattice \((\mathcal{P}(\text{Vars}), \subseteq_{LV}, \cup_{LV}, \emptyset, \text{Vars})\)
  - \(\text{Vars}\) is the set of variables in the program
  - Each element of the lattice is a subset of \(\text{Vars}\)
    - \(\mathcal{P}(\text{Vars})\) is the powerset of \(\text{Vars}\), i.e. the set of all subsets of \(\text{Vars}\)
    - \(x \subseteq_{LV} y\) iff \(x \subseteq y\)
    - \(x \cup_{LV} y = x \cup y\)
    - Most precise element \(\bot = \emptyset\) (no live variables)
    - Least precise element \(\top = \text{DEFS}\) (all variables live)
**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**
  - \( \ell_{LV} = \{ x \} \) where x is the variable returned from the function

**Flow functions**

- \( f_{LV}(\sigma, \{ x := y \}) = (\sigma \setminus \{ x \}) \cup \{ y \} \)
  - Kills (removes from set) the variable x
  - Generates (adds to set) the variable y
  - Note: must kill first then generate (what if \( y = x \)?)
- \( f_{LV}(\sigma, /\ast \text{ any other } \ast/) = \sigma \)

---

**Live Variables Example**

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

<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,z}</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>{y}</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>{x}</td>
</tr>
</tbody>
</table>