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] = $\perp_A$;
results[*entry*] = ι_A;
worklist.add(*all nodes*);

Ok to just add entry node if flow functions cannot return $\perp_A$ (examples will assume this)

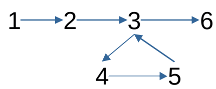
while (!worklist.isEmpty()) do
  i = worklist.pop();
  before = $\bigsqcup_{k\in \text{pred}(i)}$ results[k];
  after = f_A(before, node(i));
  if (!(after $\sqsubseteq$ results[i]))
    results[i] = after;
    for all k$\in$succ(i) do
      worklist.add(k);

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
    $[b := a]_4$;
    $[a := a + 1]_5$;
$[a := 0]_6$

Control Flow Graph



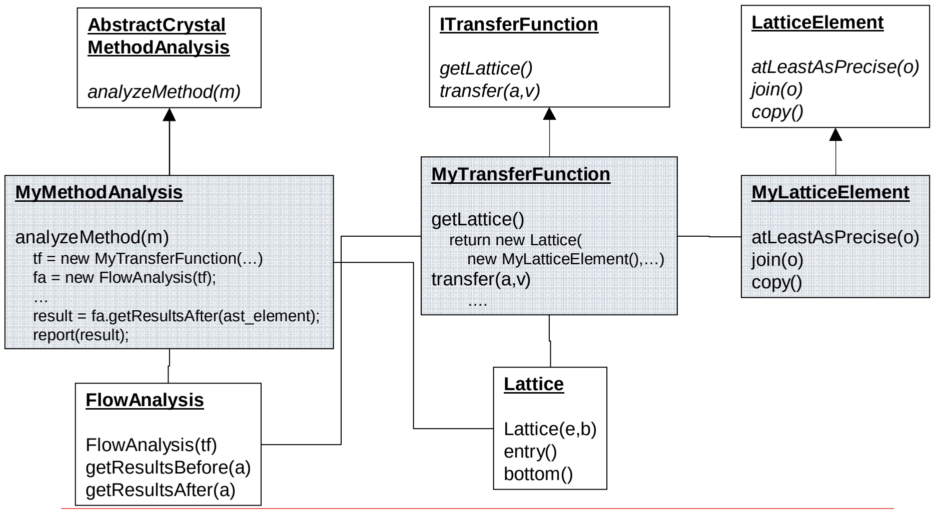
Position	**Worklist**	**a**	**b**
0	1	MZ	MZ
1	2	Z	MZ
2	3	Z	Z
3	4,6	Z	Z
4	5,6	Z	Z
5	3,6	MZ	Z
3	4,6	MZ	Z
4	5,6	MZ	MZ
5	3,6	MZ	MZ
3	4,6	MZ	MZ
4	6	MZ	MZ
6		Z	MZ

Dataflow Analysis in Crystal

17-654/17-754
Analysis of Software Artifacts

Jonathan Aldrich

Crystal Analysis Architecture



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

```
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);
    …
```


Zero Lattice

```
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

```
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

```
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); }
```

Tuple Lattice

```
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);
}
```


Tuple Lattice

```
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;
}

// must copy the underlying elements because lattice is mutable
public TupleLatticeElement<LE> copy() {
    return new TupleLatticeElement<LE>(varLattice,
            (HashMap<ASTNode, LE>) ((elements==null) ? null : elements.clone()));
}
```

Zero Analysis Definition

```
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());
    }
```


Zero Analysis Definition

```
/** constant case */
public TupleLatticeElement<DivideByZeroLatticeElement
transfer(BinaryOperation binop, TupleLatticeElement<DivideByZeroLatticeElement> value) {

    // 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;
}
```


Zero Analysis Definition

```
/** If we assign a literal integer expression to a variable, we use the
 * value of that expression to determine the analysis lattice value for
 * that variable.
 */
@Override
public TupleLatticeElement<DivideByZeroLatticeElement> transfer(LiteralInstruction instr,
        TupleLatticeElement<DivideByZeroLatticeElement> value) {
    if( instr.getLiteral() instanceof IntLiteralNode ) {
        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

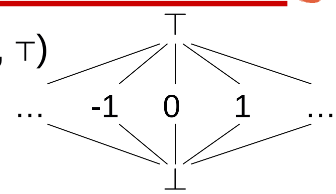
Constant Propagation

- Goal: determine which variables hold a constant value:

```
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, $\sqsubseteq_C$, $\sqcup_C$, $\perp$, $\top$)
  - L_C = **Integer** $\cup \{ \perp, \top \}$
  - $\forall n \in$ **Integer** : $\perp \sqsubseteq_C n$ && $n \sqsubseteq_C \top$
- Constant propagation lattice
  - Tuple lattice formed from above lattice
  - See notes on zero analysis for details
- Abstraction function:
  - $\alpha_C(n) = n$
  - $\alpha_{CP}(\eta) = \{ x \mapsto \alpha_C(\eta(x)) \mid x \in$ **Var** $\}$
- Initial data:
  - $\iota_{CP} = \{ x \mapsto \top \mid x \in$ **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 $\perp$ too, but we won't actually ever see $\perp$ during analysis*
- $f_{CP}(\sigma,$ /* any other */$) = \sigma$

Constant Propagation Example

```
[x := 3]1;
[y := x+7]2;
if [b]3
   then [z := x+2]4
   else [z := y-5]5;
[w := z-2]6
```

$[x := 3]_1$;
$[y := x+7]_2$;
if $[b]_3$
then $[z := x+2]_4$
else $[z := y-5]_5$;
$[w := z-2]_6$

Position	Worklist	x	y	z	w
0	1	$\top$	$\top$	$\top$	$\top$
1	2	3	$\top$	$\top$	$\top$
2	3	3	10	$\top$	$\top$
3	4,5	3	10	$\top$	$\top$
4	6,5	3	10	5	$\top$
6	5	3	10	5	3
5	6	3	10	5	$\top$
6		3	10	5	3

Constant Propagation Example

$[x := 3]_1$;
$[y := x+7]_2$;
if $[b]_3$
  then $[z := x+1]_4$
  else $[z := y-5]_5$;
$[w := z-2]_6$

Position	Worklist	x	y	z	w
0	1	⊤	⊤	⊤	⊤
1	2	3	⊤	⊤	⊤
2	3	3	10	⊤	⊤
3	4,5	3	10	⊤	⊤
4	6,5	3	10	4	⊤
6	5	3	10	4	2
5	6	3	10	5	⊤
6		3	10	⊤	⊤

Loss of Precision

if $[x = 0]_1$
  then $[y := 1]_2$;
  else $[y := x]_3$;
$[z := 10/y]_4$

Position	Worklist	x	y	z
0	1	MZ	MZ	MZ
1	2,3	MZ	MZ	MZ
2	4,3	MZ	**NZ**	MZ
4	3	MZ	NZ	NZ
3	4	MZ	**MZ**	MZ
4		MZ	**MZ**	NZ

Flow Sensitivity for Zero Analysis

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

Precision Regained

Worklist simplified to the statement level

if $[x = 0]_1$
then $[y := 1]_2$;
else $[y := x]_3$;
$[z := 10/y]_4$

Position	Worklist	x	y	z
0	1	MZ	MZ	MZ
1^T	2,3	**Z**	MZ	MZ
1^F	2,3	**NZ**	MZ	MZ
2 (use 1^T)	4,3	Z	**NZ**	MZ
4	3	Z	NZ	NZ
3 (use 1^F)	4	NZ	**NZ**	MZ
4		**MZ**	**NZ**	NZ

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 $\sqsubseteq$ 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

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

Iter	Position	x	y
1	0	Z	Z
2	1	{0}	Z
3	2	{0}	Z
4	3	{1}	Z
5	2	{0,1}	Z
6	3	{1,2}	Z
7	2	{0,1,2}	Z
8	3	{1,2,3}	Z
9	2	{0,1,2,3}	Z
10	3	{1,2,3,4}	Z
…			

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 $\perp$. 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 σ 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 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*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*e*o)
    - Typically h,o,e bounded by n = number of statements in program
    - $O(n^3)$ 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 $\alpha(\eta) \sqsubseteq \sigma$

Local Soundness

$$\begin{tikzcd}
\alpha_{DF}(\eta_o) \sqsubseteq \sigma_i \arrow[r, "f_{DF}(\sigma_i, S)"] & \sigma_o \sqsupseteq \alpha_{DF}(\eta_o) \\
\eta_i \arrow[u, dashed, "\alpha_{DF}"'] \arrow[r, "S \rightarrow"] & \eta_o \arrow[u, dashed, "\alpha_{DF}"']
\end{tikzcd}$$

- Local correctness condition for dataflow analysis
  - If applying a transfer function to statement S and input information σ_i yields output information σ_o,
  - Then for all program states η_i such that $\alpha(\eta_i) \sqsubseteq \sigma_i$ and executing S in state η_i yields state η_o,
  - We must have $\alpha(\eta_o) \sqsubseteq \sigma_o$
- 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,mem> pairs
  - pp is a program point
    - Just after statement pp
  - mem is the state of variables in memory

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

pp	x	y	z
0	2	0	0
1	2	2	0
2	2	2	1
3	2	2	1
4	2	2	2
5	2	1	2
3	2	1	2
6	2	0	2

Execution Traces

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

pp	x	y	z
0	1	0	0
1	1	1	0
2	1	1	1
3	1	1	1
6	1	0	1

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

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

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

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

pp	x	y	z

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

WHILE Traces, Formally

- A trace for program S_1 and initial state η_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 $(\eta_0, \mathit{first}(S_0)), \ldots, (\eta_n, \mathit{first}(S_n))$
    - *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, $\alpha(T(t)) \sqsubseteq \sigma_{pp(T(t))}$ where $\sigma_{pp(T(t))}$ is the analysis results at $pp(T(t))$
  - Constant propagation
    - For trace on last slide with t=10
    - $\alpha_{CP}(T(10)) = \{x \mapsto 3, y \mapsto 0, z \mapsto 6\}$
    - $\sigma_{pp(T(t))} = \sigma_6 = \{x \mapsto \top, y \mapsto 0, z \mapsto \top\}$
    - $\{x \mapsto 3, y \mapsto 0, z \mapsto 6\} \sqsubseteq_{CP} \{x \mapsto \top, y \mapsto 0, z \mapsto \top\}$
      - Because $3 \sqsubseteq_C \top$ and $0 \sqsubseteq_C 0$ and $6 \sqsubseteq_C \top$ in the CP lattice
    - To prove soundness, repeat for all times in all traces

Local Soundness

To show:

if $(\eta_i, S_i) \rightarrow (\eta_{i+1}, S_{i+1}) \in T$
and $\sigma_i = \alpha_{DF}(\eta_i)$
and $\sigma_{i+1} = f_{DF}(\sigma_i, first(S_i))$
then $\alpha_{DF}(\eta_{i+1}) \sqsubseteq \sigma_{i+1}$

$$\begin{tikzcd}
\sigma_i \arrow[r, "f_{DF}(\sigma_i,\, first(S_i))"] & \sigma_{i+1} \sqsupseteq \alpha_{DF}(\eta_{i+1}) \\
(\eta_i, S_i) \arrow[u, dashed, "\alpha_{DF}"] \arrow[r, "\rightarrow"] & (\eta_{i+1}, S_{i+1}) \arrow[u, dashed, "\alpha_{DF}"']
\end{tikzcd}$$

Intuitively, translating from concrete to abstract and applying the flow function will safely approximate ($\sqsupseteq$) taking a step in the trace and translating from concrete to abstract

Finding Errors with Local Soundness

- Consider the **incorrect** flow function:
  $f_{ZA}(\sigma, [x := y\ op\ z]) =$
  if $\sigma[y]$=Z || $\sigma[z]$=Z
  then $[x \mapsto Z]\sigma$ else $[x \mapsto MZ]\sigma$
- Local Soundness fails!
  - Consider $\eta_i = []$, $S_i = [x := 3+0]_k$
  - $\sigma_i = \alpha_{DF}(\eta_i) = \alpha_{DF}([]) = []$
  - $\sigma_{i+1} = f_{DF}(\sigma_i, \mathit{first}(S_i)) = [x \mapsto Z]$
  - $\alpha_{DF}(\eta_{i+1}) = \alpha_{DF}([x \mapsto 3]) = [x \mapsto NZ]$
  - $\alpha_{DF}(\eta_{i+1}) \not\sqsubseteq \sigma_{i+1}$ because Z $\not\sqsubseteq$ NZ

Global Soundness

- Intuition
  - We begin with initial dataflow facts ι that safely approximate ($\sqsupseteq$) all initial stores η_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 $\forall T \in traces(S_s)$ $\alpha_{DF}(\eta_0) \sqsubseteq \iota$ and that analysis DF is monotone and locally sound with respect to α_{DF}
  - Then for any fixed point DF_{fix} of DF on program S, $\forall T \in traces(S)$ $\forall t \in times(T)$ we have $\alpha_{DF}(\eta_t) \sqsubseteq DF_{fix}(pp(T(t)))$
- Proof outline: For each trace T we do induction on t
  - Induction hypothesis: $\alpha_{DF}(\eta_t) \sqsubseteq DF_{fix}(pp(T(t)))$
  - Base case: t=0
    - By assumption $\alpha_{DF}(\eta_0) \sqsubseteq \iota = DF_{fix}(pp(\eta_0))$
  - Inductive case: time t and statement S_t
    - *Simplifying assumption: straight-line control flow*
    - By induction hypothesis we have $\alpha_{DF}(\eta_{t-1}) \sqsubseteq DF_{fix}(pp(T(t-1)))$
    - By monotonicity of DF we have: $f_{DF}(\alpha_{DF}(\eta_{t-1}), S_t) \sqsubseteq f_{DF}(DF_{fix}(pp(T(t-1))), S_t)$
    - By local soundness we have $\alpha_{DF}(\eta_t) \sqsubseteq f_{DF}(\alpha_{DF}(\eta_{t-1}), S_t)$
    - By transitivity we get $\alpha_{DF}(\eta_t) \sqsubseteq f_{DF}(DF_{fix}(pp(T(t-1))), S_t)$
    - But $f_{DF}(DF_{fix}(pp(T(t-1))), S_t) = DF_{fix}(pp(T(t)))$ because it's a fixed point
    - So we have $\alpha_{DF}(\eta_t) \sqsubseteq DF_{fix}(pp(T(t)))$

Other Dataflow Analyses

- Traditional optimization analyses
  - Reaching Definitions
  - Live Variables

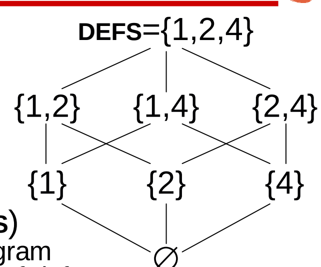
Reaching Definitions Analysis

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

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

- 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" sentinal, then we may be using an undefined variable

Reaching Definitions



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

Reaching Definitions

- Initially assume dummy assignments
  - Represents passed values for parameters
  - Represents uninitialized for non-parameters
  - $\iota_{RD} = \{ x_0 \mid x \in \mathbf{Var} \}$
- Flow functions
  - $f_{RD}(\sigma, [x := \ldots])$

    $= \sigma - \{ x_m \mid x_m \in \textbf{DEFS}\} \cup \{ x_k \}$
    - 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
  - $f_{RD}(\sigma, /^{*}\ \textit{any other}\ ^{*}/) = \sigma$

Reaching Definitions Example

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

Position	Worklist	Lattice Element
0	1	$\{x_0, y_0, z_0\}$
1	2	$\{x_0, y_1, z_0\}$
2	3	$\{x_0, y_1, z_1\}$
3	4,6	$\{x_0, y_1, z_1\}$
4	5,6	$\{x_0, y_1, z_4\}$
5	3,6	$\{x_0, y_5, z_4\}$
3	4,6	$\{x_0, y_1, y_5, z_1, z_4\}$
4	5,6	$\{x_0, y_1, y_5, z_4\}$
5	6	$\{x_0, y_5, z_4\}$
6		$\{x_0, y_6, z_1, z_4\}$

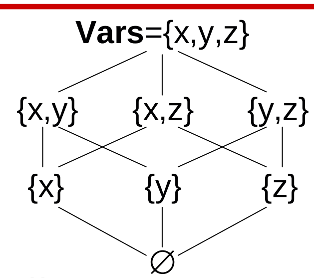
Live Variables Analysis

- 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 * 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



- Set Lattice ($\mathbb{P}^{\textbf{Vars}}$, $\sqsubseteq_{LV}$, $\sqcup_{LV}$, $\emptyset$, **Vars**)
  - **Vars** is the set of variables in the program
  - Each element of the lattice is a subset of **Vars**
    - $\mathbb{P}^{\textbf{Vars}}$ is the powerset of **Vars**, i.e. the set of all subsets of **Vars**
  - $x \sqsubseteq_{LV} y$ iff $x \subseteq y$
  - $x \sqcup_{LV} y = x \cup y$
  - Most precise element $\bot = \emptyset$ (no live variables)
  - Least precise element $\top$=**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
  - $\iota_{LV} = \{ x \}$ where x is the variable returned from the function
- Flow functions
  - $f_{LV}(\sigma, [x := y]_k) = (\sigma - \{ x \}) \cup \{ y \}$
    - Kills (removes from set) the variable x
    - Generates (adds to set) the variable y
    - Note: must kill first then generate (what if $y = x$?)
  - $f_{LV}(\sigma,$ */* any other */*$) = \sigma$

Live Variables Example

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

Position	**Worklist**	**Lattice Element**
exit	6	{z}
6	3	{z}
3	5,2	{y,z}
5	4,2	{y,z}
4	3,2	{y,z}
3	2	{y,z}
2	1	{y}
1		{x}