

Interprocedural Analysis in PREfix

Reading: ***A Static Analyzer for Finding
Dynamic Programming Errors***

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2006

Motivation: Interprocedural Analysis



```
void exercise_deref() {  
    int v = 5;  
    int x = deref(&v);  
    int y = deref(NULL);  
    int z = deref((int *) 5);  
}
```

- Are there errors in this code?
 - Depends on what the function does
 - Second call: error if dereference w/o NULL check
 - Third call: error if any dereference

Analysis of Software Artifacts -
Spring 2006

3

Interprocedural Analysis



- ***Any analysis where the analysis results for a caller depend on the results for a callee, or vice versa***

Summaries



- Summarize what a function does
 - Maps arguments to results
 - May case-analyze on argument information
 - Simulateable
 - Given information about arguments, will yield:
 - Any errors
 - Information about results

PREfix: Building a Summary

(syntax slightly de-LISP-ified)



```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- **Return statement**
deref (param p)
alternate return_0
guard p==NULL
constraint initialized(p)
result return==NULL
- alternate return_X
guard p != NULL
constraint initialized(p)
constraint valid_ptr(p)
constraint initialized(*p)
result return==*p

PREfix: Using a Summary

(syntax slightly de-LISP-ified)



```
void exercise_deref(int v) {  
    int v = 5;  
    int x = deref(&v);  
    int y = deref(NULL);  
    int z = deref((int *) 5);  
}
```

```
deref (param p)  
alternate return_0  
guard p==NULL  
constraint initialized(p)  
result return==NULL  
alternate return_X  
guard p != NULL  
constraint initialized(p)  
constraint valid_ptr(p)  
constraint initialized(*p)  
result return==*p
```

- **Apply summary**
exercise_deref
fact initialized(v), v==5
fact initialized(&v), valid_ptr(&v)
fact x==5
- only return_X applies
 - **constraint initialized(&v) – PASS**
 - **constraint valid_ptr(&v) – PASS**
 - **constraint initialized(*&v) – PASS**
 - **apply result**

PREfix: Using a Summary

(syntax slightly de-LISP-ified)



```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary
- ```
exercise_deref
 fact initialized(v), v==5
 fact initialized(&v), valid_ptr(&v)
 fact x==5
 fact y==NULL
```
- only return\_0 applies
  - **constraint initialized(p) – PASS**
  - **apply result**

## PREfix: Using a Summary

(syntax slightly de-LISP-ified)



```
void exercise_deref(int v) {
 int v = 5;
 int x = deref(&v);
 int y = deref(NULL);
 int z = deref((int *) 5);
}
```

```
deref (param p)
 alternate return_0
 guard p==NULL
 constraint initialized(p)
 result return==NULL
 alternate return_X
 guard p != NULL
 constraint initialized(p)
 constraint valid_ptr(p)
 constraint initialized(*p)
 result return==*p
```

- Apply summary
- ```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
  fact y==NULL
  fact !valid_ptr((int *) 5), (int *) 5 !=
  NULL
```
- return_0 does not apply
 - return_X does apply
 - constraint initialized((int *) 5) – PASS
 - **constraint valid_ptr((int *) 5) – FAIL**
 - Generate error

PREfix Scaleability



Program	Language	number of files	number of lines	PREfix parse time	PREfix simulation time
Mozilla	C++	603	540613	2 hours 28 minutes	8 hours 27 minutes
Apache	C	69	48393	6 minutes	9 minutes
GDI Demo	C	9	2655	1 second	15 seconds

Table I: Performance on Sample Public Domain Software

- Analysis cost = 2x-5x build cost
 - Scales linearly
 - Probably due to fixed cutoff on number of paths

Value of Interprocedural Analysis

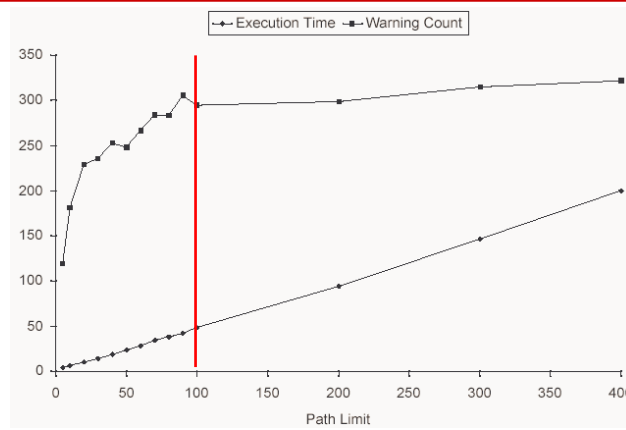


model set	execution time (minutes)	statement coverage	branch coverage	predicate coverage	total warning count	using uninit memory	NULL pointer deref	memory leak
none	12	90.1%	87.8%	83.9%	15	2	11	0
system	13	88.9%	86.3%	82.1%	25	6	12	7
system & auto	23	73.1%	73.1%	68.6%	248	110	24	124

Table III: Relationships between Available Models, Coverage, Execution Time, and Defects Reported

- 90% of errors require models (summaries)

You don't need every path



- Get most of the warnings with 100 paths

Empirical Observations



- PREFIX finds errors off the main code paths
 - Main-path errors caught by careful coding and testing
- UI is essential
 - Text output is hard to read
 - Need tool to visualize paths, sort defect reports
- Noise warnings
 - Real errors that users don't care about
 - E.g., memory leaks during catastrophic shutdown

PREfix Summary



- Great tool to find errors
 - Can't guarantee that it finds them all
 - Role for other tools (e.g., Fluid)
 - Complements testing by analyzing uncommon paths
 - Focuses on low-level errors, not logic/functionality errors
 - Role for functional testing
- Huge impact
 - Used widely within Microsoft
 - Lightweight version is part of new Visual Studio

Dataflow Analysis in Crystal

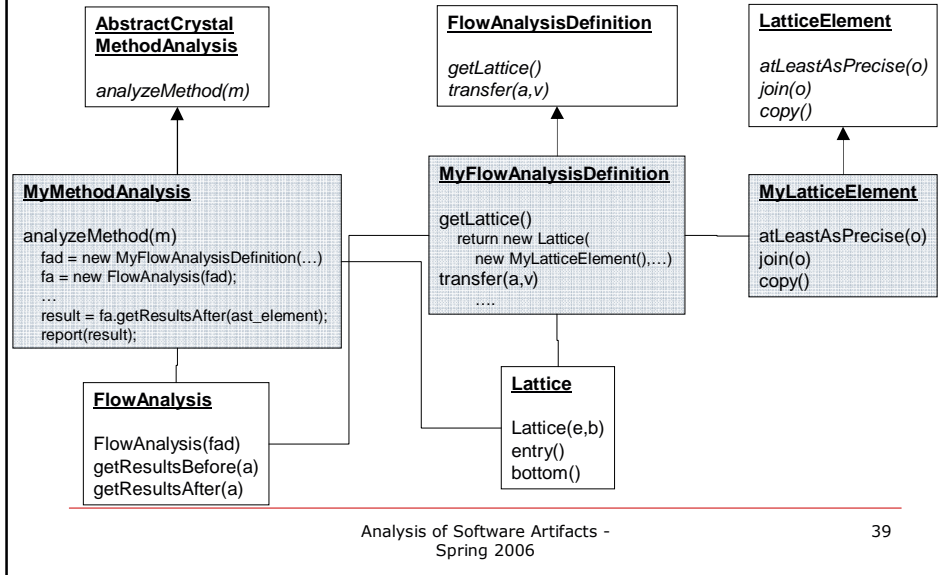
17-654/17-754

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)
- FlowAnalysis.performAnalysis()
 - Worklist algorithm
 - 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 ZeroAnalysisDefinition extends
    FlowAnalysisDefinition<TupleLatticeElement<ZeroLatticeElement>> {

    public Lattice<TupleLatticeElement<ZeroLatticeElement>>
    getLattice(MethodDeclaration 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<ZeroLatticeElement> transfer(
    NumberLiteral node, TupleLatticeElement<ZeroLatticeElement> value) {
    if (((Integer) node.resolveConstantExpressionValue()).intValue() == 0)
        value.put(node, ZeroLatticeElement.Z);
    else
        value.put(node, ZeroLatticeElement.NZ);
    return value;
}
```

Zero Analysis Definition



```
/** common code for handling assignment and initialization */
private void handleAssign( ASTNode left, ASTNode right,
    TupleLatticeElement<ZeroLatticeElement> value) {
    if (left instanceof SimpleName) {
        ASTNode declNode = Utilities.getASTNode(((SimpleName)left).resolveBinding());
        value.put(declNode, value.get(right));
    }
}

/** assignment case (true assignment) */
public TupleLatticeElement<ZeroLatticeElement> transfer(
    Assignment node, TupleLatticeElement<ZeroLatticeElement> value) {
    handleAssign(node.getLeftHandSide(), node.getRightHandSide(), value);
    return value;
}

/** assignment case (variable initializer) */
public TupleLatticeElement<ZeroLatticeElement> transfer(
    SingleVariableDeclaration node, TupleLatticeElement<ZeroLatticeElement> value) {
    handleAssign(node.getName(), node.getInitializer(), value);
    return value;
}
```

Zero Analysis Definition



```
/** variable case */
public TupleLatticeElement<ZeroLatticeElement> transfer(
    SimpleName node,
    TupleLatticeElement<ZeroLatticeElement> value) {

    // if this is the decl point, skip it; this TF is only for uses
    if (node.getParent() instanceof VariableDeclaration
        && ((VariableDeclaration)node.getParent()).getName() == node)
        return value;

    ASTNode declNode = Utilities.getASTNode(node.resolveBinding());
    value.put(node, value.get(declNode));
    return value;
}
```