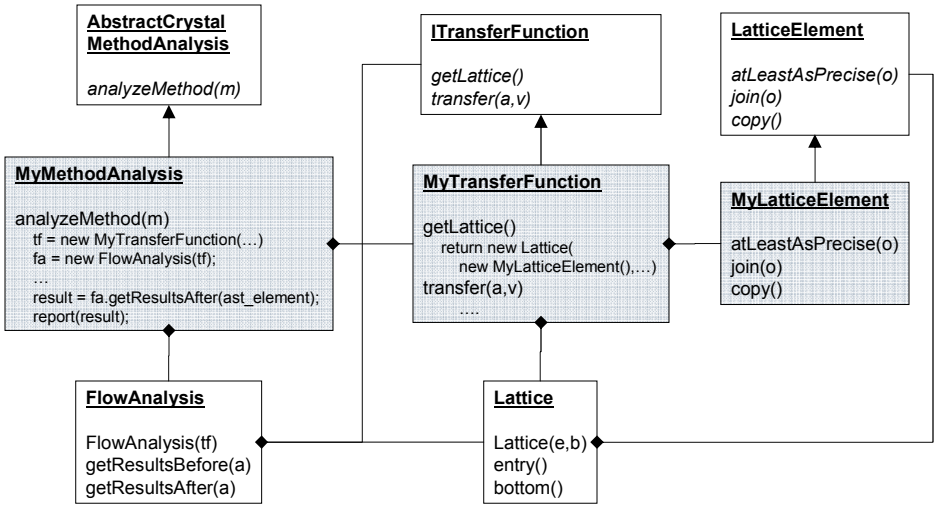
Dataflow Analysis in Crystal

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()

Packages of interest

edu.cmu.cs.crystal
Contains CrystalPlugin

edu.cmu.cs.crystal.flow
Base classes of the flow classes

edu.cmu.cs.crystal.tac
The three address code operations
TAC related classes

edu.cmu.cs.crystal.examples.flow
DivideByZero example
(we'll provide a few more in this package)

edu.cmu.cs.crystal.cs654
where YOUR analyses will go!

What YOU need to do

- Derive from AbstractCrystalMethodAnalysis
  - Set up FlowAnalysis with your transfer functions
  - Call getResultsAfter
  - Report any errors
- Derive from AbstractingTransferFunction
  - Implement your transfer functions
  - Set up the lattice that will be used
- Derive from LatticeElement
  - Implement the methods carefully
- Register your analysis in CrystalPlugin
- Use the example analyses to help you

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
- Watch that lattice!
  - If lattice has infinite height, you can loop forever
  - Code review join, atLeastAsPrecise, and copy
  - Immutable lattices are easier to do than mutable lattices

Dataflow Analysis Example:

Constant Propagation

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich

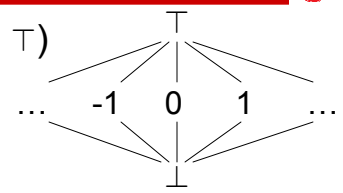
Constant Propagation

- 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, $\sqsubseteq_C$, $\sqcup_C$, $\bot$, $\top$)
  - L_C = **Integer** ⌈ { $\bot$, $\top$ }
  - $\forall n \in$ **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 ↦ $\alpha_C(\eta(x))$ | x∈**Var** }
- Initial data:
  - ι_{CP} = { x ↦ $\top$ | x∈**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*
- $f_{CP}(\sigma$, */* 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
```

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	5	⊤
6	5	3	10	5	3
5	6	3	10	5	⊤
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 $\neg Z$=NZ; $\neg NZ$=Z; $\neg MZ$=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