

# Crystal-izing Sophisticated Code Analyses

---

Ciera Jaspán

Kevin Bierhoff

Jonathan Aldrich

**Carnegie Mellon**



<http://code.google.com/p/crystalsaf>

# Installation

---

- Pre-requisites
  - Eclipse with
    - Java Development Tools (JDT)
    - Plugin Development Environment (PDE)
  - Crystal
    - Available from our Eclipse update site
    - <http://crystalsaf.googlecode.com/svn/trunk/EclipseUpdate/>

# Crystal

---

- A framework for creating static analyses
  - AST walkers
  - Simple dataflow
  - Branch-sensitive dataflow
  - Use of specifications
- Primarily for educational purposes
  - Easy startup into a static analysis
  - Direct from theory to implementation
  - Incremental development of analyses
  - “Wow factor”: power of Eclipse
- Also found to be useful for research prototypes

# Crystal in the classroom

---

- Crystal is used in CMU graduate analysis courses
  - Courses: Analysis of Software Artifacts, Program Analysis
  - Students: Ph.D. students, professional masters' students
- Students will learn
  - What can affect the usability and precision of a static analysis
  - What kind of problems static analysis can solve

# Crystal in research

---

- High transferability from paper to code made Crystal natural choice for research
- Currently 4 research analyses written in Crystal
  - 3 are published (OOPSLA 08, ECOOP 09, OOPSLA 09)
- Allows incremental development to more sophisticated features
  - Annotations with custom parsers
  - Branch-sensitivity
  - Automated testing
  - Widening v. regular join

# Demonstration

---

- Install Crystal
- Create an Eclipse Plugin
- Create an Analysis class
- Register the analysis
- Run the analysis
- Create a visitor
- Run the analysis again

# Register and Run

---

- Create a new plugin project
  - Make it depend on Crystal and JDT
- Implement `ICrystalAnalysis`
- Register with the extension-point `CrystalAnalysis` in the `plugin.xml` file
- Select Run -> Run Configuration...
- Make a new Eclipse configuration
- Run! Your analysis name should appear in the Crystal menu.

# Steps for making an analysis

---

- Install Crystal
- Register an analysis
- **Create a simple AST walker for nullness**
- Add a simple flow analysis
- Add annotations
- Add branch-sensitivity



# Use an AST walker

---

- We're ready to make an analysis now.
- **What kinds of expressions do we want to check for an null pointer analysis?**
  - Method calls
  - Field access
  - Array access
- **In analyzeMethod(), create an ASTVisitor that gives an error when it encounters these operations.**

# Everything running?

---

- Create a new project in the child Eclipse
- Add code to analyze
- Test with Crystal->MyAnalysis
- Can also run automated tests in JUnit
  - Will not cover that today
  - See wiki for more information

# Steps for making an analysis

---

- Install Crystal
- Register an analysis
- Create a simple AST walker for nullness
- **Add a simple flow analysis**
- Add annotations
- Add branch-sensitivity

# Abstract interpretation concepts

---

- Lattice
  - The abstract states the program can be in
- Control flow graph
  - The order of control flow through the nodes of the AST
- Transfer functions
  - How the states change as the analysis encounters new program instructions
- Worklist algorithm
  - Traverses the control flow graph and runs the transfer functions

# Abstract interpretation concepts

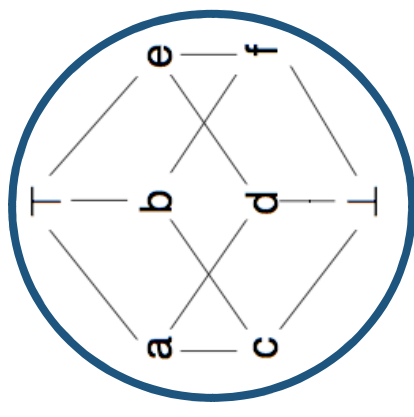
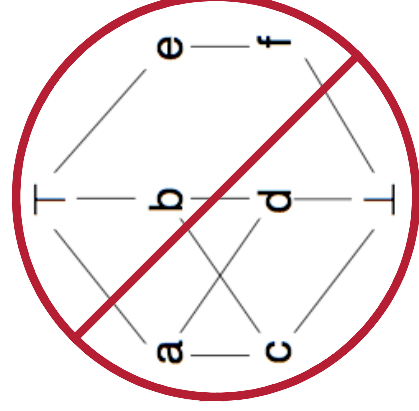
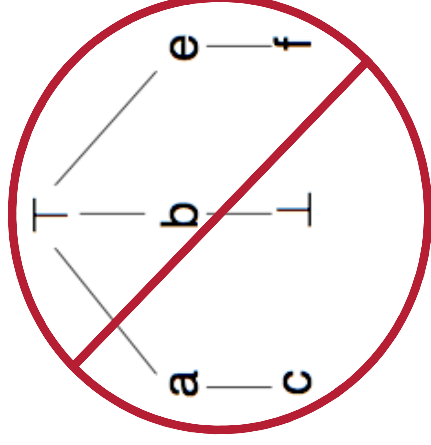
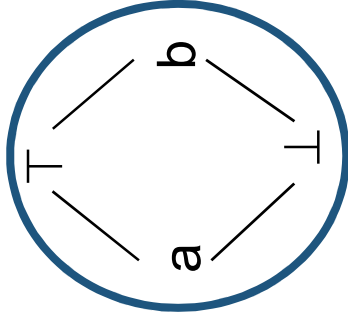
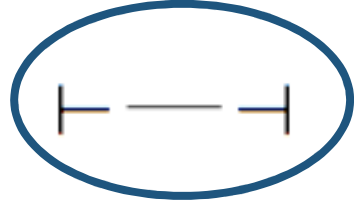
---

- Lattice
  - A finite lattice of the abstract states the program can be in
- Control flow graph
  - The order of control flow through the nodes of the AST
- Transfer functions
  - How the states change as the analysis encounters new program instructions
- Worklist algorithm
  - Traverses the control flow graph and runs the transfer functions

# Lattice review

---

- Top of lattice represents least precise info
- Bottom of lattice represents an unanalyzed element
- Must have finite height to ensure termination
- Unique least upper bound must exist for any two elements



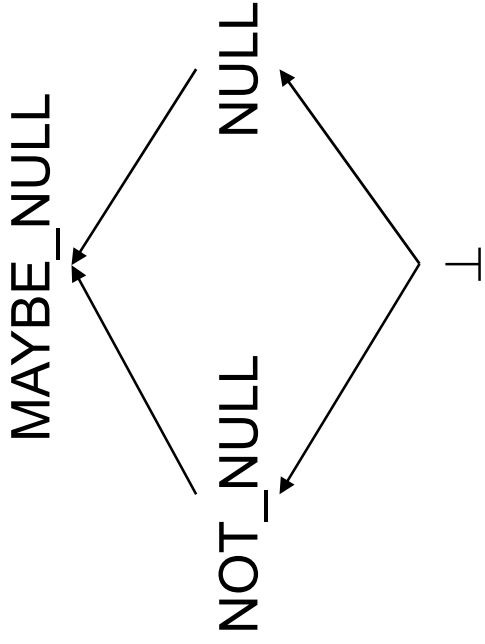
# Transfer function review

---

- Given
  - An instruction
  - An incoming lattice element  $\sigma$
- Produce
  - An outgoing lattice element  $\sigma'$
- $f(\text{instr}, \sigma) = \sigma'$
- Make a different transfer function on each type of instruction

# A simple null analysis

---



$\sigma$  is a map of every program variable to a null lattice element

- $f(x = \text{null}, \sigma) = \sigma[x \rightarrow \text{NULL}]$
- $f(x = y, \sigma) = \sigma[x \rightarrow \sigma(y)]$
- $f(x = \text{new } C(), \sigma) = \sigma[x \rightarrow \text{NOT\_NULL}]$
- $f(x = y.m(z_1, \dots, z_n), \sigma) = \sigma[y \rightarrow \text{NOT\_NULL}]$   
(array access, field access, etc.)

- Map all variables to an element in the lattice above
- Tuple Lattice: A lattice which maps a key to an element in another lattice



# The lattice

---

- **What are the elements in the lattice?**
  - Bottom, null, not null, and maybe null
- **Create a type which represents the elements**
  - Crystal allows this to be an arbitrary type
  - This is likely an immutable type, like an enum

```
public enum NullLatticeElement {  
    BOTTOM, NULL, NOT_NULL, MAYBE_NULL;  
}
```

- Tuple Lattices
  - We will use `TupleLatticeElement`
  - Just create the type which represents the sub-lattice

# The lattice

---

- **What is the bottom-most element?**
- **What is the the top-most?**
- **What is the ordering of elements?**
- **What does the join operation look like?**
  
- **Extend SimpleLatticeOperations<LE>**
  - LE bottom()
  - boolean atLeastAsPrecise(LE, LE)
  - LE join(LE, LE)
  - LE copy(LE)

# Setting up the flow analysis

---

- **What is the lattice element at the start of the method?**
  - Everything may be null (except **this** is not null)
- **Extend AbstractingTransferFunction**
  - Implement `createEntryValue()` and `getLatticeOperations()`
  - (Don't override the transfer functions yet)
- We'll also now have the visitor call the flow analysis

# Transfer functions

---

- **Which instructions cause the lattice element to change? How do they change the lattice element?**
  - Null: makes target null
  - Constructor: makes target non-null
  - Copying assignment: makes target same as right side
- **In the derived transfer function, override the relevant instructions**

# You now have a Crystal flow analysis

---

- **And that's it!**
- From here, we're just going to improve it
  - Annotations: teach students power of specifications
  - Branch-sensitivity: teach students power of abstractions closer to code
- We'll move a little faster
- Assistants are ready to help if you wish to follow along

# Why Three Address Code

---

- Does mean students work with both Eclipse AST and TAC
- However
  - TAC has no sub-expressions
  - TAC has many fewer kinds of nodes
- Students able to understand TAC as it matched what they wrote down on paper

# Relevant packages

---

- **edu.cmu.cs.crystal**
  - Core package for analyses
- **org.eclipse.jdt.core.dom**
  - The Eclipse AST
- **edu.cmu.cs.crystal.simple**
  - Simple interfaces for flow analyses
- **edu.cmu.cs.crystal.tac.model**
  - The interfaces for three address code instructions

# Steps for making an analysis

---

- Install Crystal
- Register an analysis
- Create a simple AST walker for nullness
- Add a simple flow analysis
- **Add annotations**
- Add branch-sensitivity



# Annotations

---

- **What specifications could we add to make the analysis more precise?**
  - Non-null on method parameters
- **Create a Java annotation**
  - Put it in a jar separate from your analysis
  - Make it available to the code being analyzed

```
@Target({ElementType.METHOD, ElementType.PARAMETER})  
public @interface NonNull { }
```

# Annotations

---

- **What transfer functions can use this annotation to improve precision?**
  - Initial lattice information
  - Method call instruction (return value)
- Annotations are available from the Eclipse AST
  - But hard to get
  - No desugaring
- **Use the AnnotationDatabase**
  - Pass in an AnnotationDatabase to the transfer functions
  - Query it to find instances of the `@NotNull` annotation
- Can also give Crystal a custom parser for complex annotations
  - `@Invariant("x == foo and y != bar")`

# Annotations

---

- **Where can the visitor use annotations for additional checking?**
  - Method call parameters and return value
  - Constructor call parameters
- **Use the AnnotationDatabase**
  - Query it to find instances of the `@NotNull` annotation
  - Check that parameter is not null

# Steps for making an analysis

---

- Install Crystal
- Register an analysis
- Create a simple AST walker for nullness
- Add a simple flow analysis
- Add annotations
- **Add branch-sensitivity**

# Branch-sensitivity

---

- Take advantage of knowledge gained through tests
- Specify different exit paths through a method
  - An invariant that doesn't hold on exceptional exit
- Labeled branches let us distinguish these

```
if (x != null) {  
    //hey, it's safe  
    //to use x in here!  
}  
else {  
    //but it's an  
    //error in here!  
}
```

- **Must return different lattice elements for each label**

# On paper...

---

- No branch sensitivity

$$f(x == y, \sigma) = \sigma$$

- Branch sensitivity

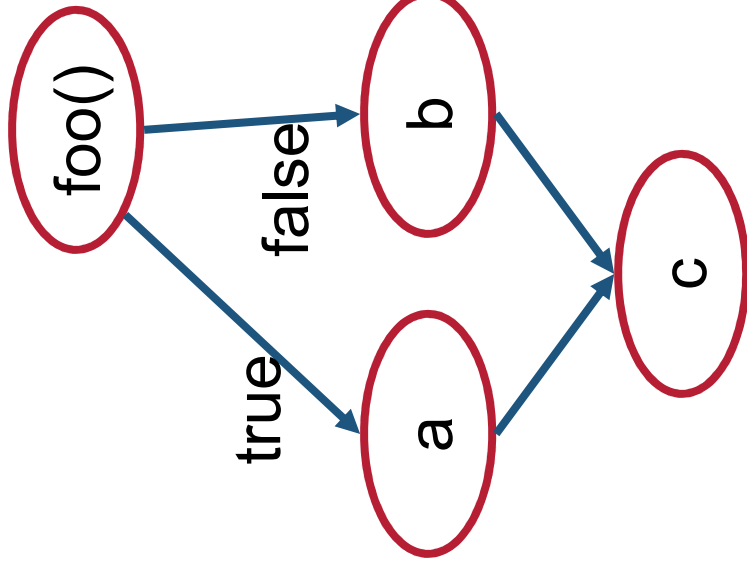
$$f^T(x == y, \sigma) = \begin{cases} \sigma[y \rightarrow \sigma(x)] & \text{if } (\sigma(x) < \text{MAYBE\_NULL}) \\ \text{else if } (\sigma(y) < \text{MAYBE\_NULL}) & \sigma[x \rightarrow \sigma(y)] \\ \text{else} & \sigma \end{cases}$$

- Separate definition for the false branch  $f^F(x == y, \sigma)$

# Branching example

---

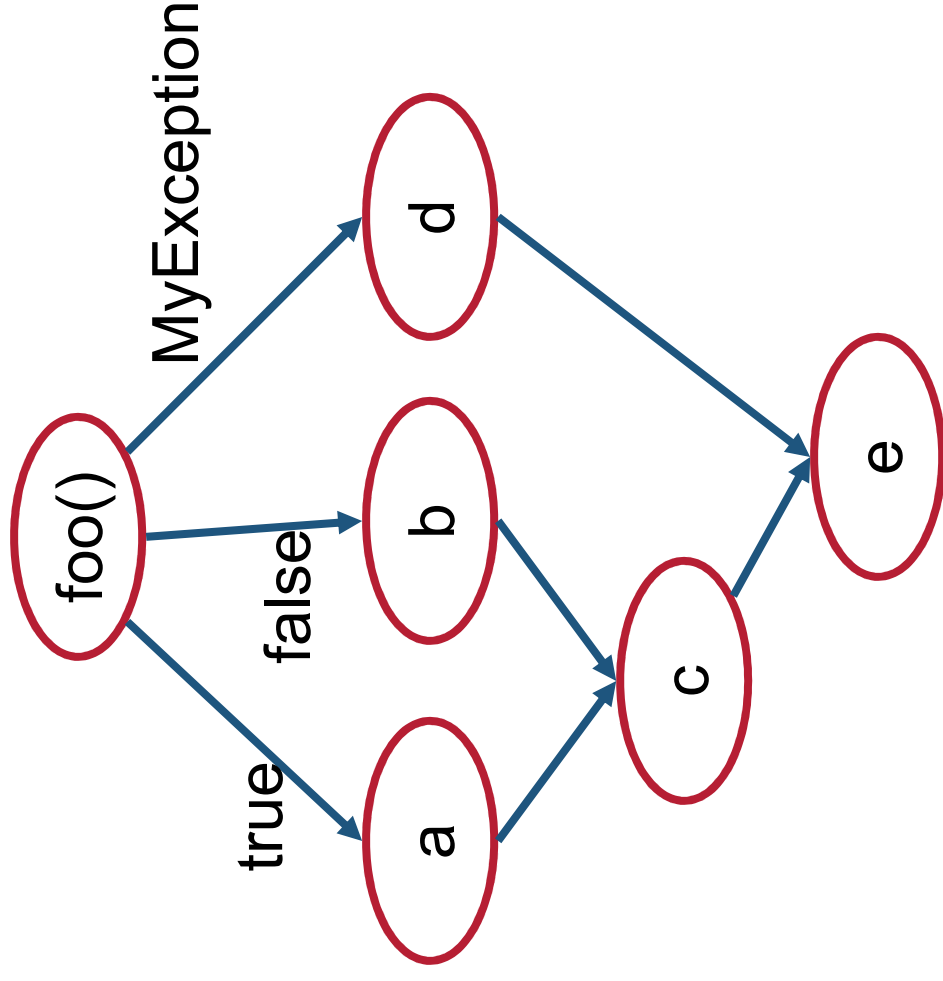
```
if (foo()) {  
  a;  
} else {  
  b;  
}  
c;
```



```
public boolean foo()
```

# Branching example, with exceptions

```
try {  
    if (foo()) {  
        a;  
    } else {  
        b;  
    }  
    c;  
} catch (MyException exp) {  
    d;  
}  
e;
```



**public boolean foo() throws MyException;**



# Types of labels

---

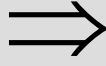
- True/false
  - All conditionals (if, while, ?:, etc.)
  - Methods calls that return a boolean
  - Binary relational operators (&&, <, ==, etc.)
- Exceptional
  - Methods calls that throw exceptions
  - Throw statements
  - Catch and Finally statements
- Switch (used on switch)
- Iterator (used on enhanced for)
- Normal

# Changing to branch-sensitive analyses

---

1. Implement `AbstractTACBranchSensitiveTransferFunction`
2. Change signatures on transfer functions

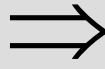
```
public LE transfer(TACInstruction instr, LE value)
```



```
public IResult<LE> transfer(TACInstruction instr,  
    List<ILabel> labels, LE value)
```

3. Wrap return lattice in an `IResult`

```
return value;
```



```
return LabeledSingleResult.createResult(value, labels);
```

At this point, transfer functions run as they did before

# Using the branches

---

- **Which instructions can provide different information on each branch?**

- $x == y$
- $x != y$

- **Create a new LabeledResult with the labels and a default value**

- Copy the lattice element for each branch
- Change the lattice elements
- Put them into the labeled result with the right label

# Crystal Static Analysis Framework

---

- Fast startup into a simple analysis
- Direct from theory to implementation
- Incremental sophistication of analysis
- Full power of Eclipse infrastructure
- **Proven useful for both teaching and research**
  - 4 research analyses
  - Used for several years in a professional master's course