Class Hierarchy Complementation: Soundly Completing a Partial Type Graph

George Balatsouras
Yannis Smaragdakis
University of Athens
OOPSLA 2013
Motivation: Static Analysis

- Static Analysis using the Doop framework
  - Analyzes Java programs
  - Uses Soot to analyze bytecode
  - Whole-program analysis
  - External dependencies
  - Missing libraries / class defs
Soot FAQ

How do I modify the code in order to enable soot to continue loading a class even if it doesn’t find some of its references? Can I create a dummy soot class so it can continue with the load? How?
Motivation: Static Analysis

Soot FAQ

How do I modify the code in order to enable soot to continue loading a class even if it doesn’t find some of it[s] references? Can I create a dummy soot class so it can continue with the load? How?

“You can try -use-phantom-refs but often that does not work because not all analyses can cope with such references.”
Class Hierarchy Complementation: Soundly Completing a Partial Type Graph

Complementation Problem

Partial Program

+

Phantom Classes

Complete Program

- Valid Java bytecode
- JVM Standard
- Verifiable

G. Balatsouras

Class Hierarchy Complementation: Soundly Completing a Partial Type Graph

5
Complementation Problem

Partial Program + Phantom Classes = Complete Program

- Valid Java bytecode
- JVM Standard
- Verifiable

G. Balatsouras

Class Hierarchy Complementation: Soundly Completing a Partial Type Graph
JPhantom: Approach

1. Detect every phantom reference
2. Generate minimal classes (empty method bodies) that respect the:
   i. referenced member signatures
   ii. implied type hierarchy
Motivating Example

X, Y, Z phantom classes

```java
public void foo(X, Y) :
aload_2   // load 2\textsuperscript{nd} arg (Y) into stack
aload_1   // load 1\textsuperscript{st} arg (X) into stack
invokevirtual `Z X.bar(A)`
invokevirtual `void B.baz()`
...```

G. Balatsouras Class Hierarchy Complementation: Soundly Completing a Partial Type Graph
public void foo(X, Y) :
aload_2  // load 2nd arg (Y) into stack
aload_1  // load 1st arg (X) into stack
invokevirtual `Z X.bar(A)`
invokevirtual `void B.baz()`
...
Symbolic Execution: Step 2

X, Y, Z phantom classes

```
public void foo(X, Y) :
    aload_2    // load 2^nd arg (Y) into stack
    aload_1    // load 1^st arg (X) into stack
    invokevirtual `Z X.bar(A)`
    invokevirtual `void B.baz()`
    ...
```

Stack

Y
Symbolic Execution: Step 3

X, Y, Z phantom classes

public void foo(X, Y) :
aload_2  // load 2\textsuperscript{nd} arg (Y) into stack
aload_1  // load 1\textsuperscript{st} arg (X) into stack
invokevirtual `Z.X.bar(A)`
invokevirtual `void B.baz()`
...

Stack

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>
Symbolic Execution: Step 3

X, Y, Z phantom classes

public void foo(X, Y):
aload_2    // load 2nd arg (Y) into stack
aload_1    // load 1st arg (X) into stack
invokevirtual `Z X.bar(A)`
invokevirtual `void B.baz()`
...

Method bar:
• expects an argument of type A
• receives an argument of type Y

Stack

Constraints

• X has to be a class (and not an interface).
• X has to provide a method: Z bar(A)
• Y has to be a subtype of A
Symbolic Execution: Step 4

X, Y, Z phantom classes

Stack

Constraints

- X has to be a class (and not an interface).
- X has to provide a method: Z bar(A)
- Y has to be a subtype of A

public void foo(X, Y) {
 aload_2  // load 2\textsuperscript{nd} arg (Y) into stack
 aload_1  // load 1\textsuperscript{st} arg (X) into stack
  invokevirtual `Z X.bar(A)`
  invokevirtual `void B.baz()`
  ...
}
Symbolic Execution: Step 4

**X, Y, Z phantom classes**

```java
public void foo(X, Y) {
   aload_2  // load 2\textsuperscript{nd} arg (Y) into stack
   aload_1  // load 1\textsuperscript{st} arg (X) into stack
   invokevirtual `Z X.bar(A)`
   invokevirtual `void B.baz()`
    ...
}
```

**Stack**

- `Z`

**Constraints**

- X has to be a class (and not an interface).
- X has to provide a method: Z bar(A)
- Y has to be a subtype of A
- Z has to be a subtype of B

**Method baz:**

- declares a receiver of type B
- is called by an object of type Z
public void foo(X, Y) :
aload_2    // load 2\text{nd} arg (Y) into stack
aload_1    // load 1\text{st} arg (X) into stack
invokevirtual `Z X.bar(A)`
invokevirtual `void B.baz()`
...
Hierarchy Complementation

Partial Type Graph

+ subtyping constraints

E.g., \(<A>\) has to be a (transitive) subtype of \(<B>\)

Complete Hierarchy

JPhantom
Multiple Inheritance
Multiple Inheritance

Wait a minute. Aren't we talking about Java?
Wait a minute. Aren't we talking about Java?
Class Hierarchy Complementation: Soundly Completing a Partial Type Graph

Multiple Inheritance Problem

Constraint Graph

Input

Cannot alter outgoing edges of known nodes

Output
The phantom projection set of A is \{B, C, D\}.

In order to satisfy path-edge \((A, E)\) we can either add an edge \((B, E)\), \((C, E)\), or \((D, E)\).
Key Idea

- Stratification exists for any solution
- Edges facing upwards property
  - No cycles
Multiple Inheritance

Algorithm

- Construct valid stratification iteratively
  - Keep nodes at minimum height
  - Keep edges facing upwards
  - Advance node only to satisfy constraint
  - Fixpoint
- Add upward edges to satisfy path constraints
Multiple Inheritance Example

Step 1

- Known Class
- Phantom Class
- Direct Edge
- Path Edge

Diagram:

A → B → C → D → E → F
Step 1

Multiple Inheritance Example

G. Balatsouras  Class Hierarchy Complementation: Soundly Completing a Partial Type Graph  26
Multiple Inheritance Example

Step 1

Not really. Happens to be topologically sorted.

Random?
Multiple Inheritance Example

Step 1

Isn't that enough?

Happens to be topologically sorted.
Multiple Inheritance Example

Step 1

Isn’t that enough?

Path-edge (A,B) cannot be satisfied since both C and E are after B.

Happens to be topologically sorted.
Step 1

Isn’t that enough? **NO!!**

Path-edge (A,B) cannot be satisfied since both C and E are after B.

Happens to be topologically sorted.
Multiple Inheritance Example

Step 1

✗ All nodes except A, and D have incoming (horizontal) edges
Multiple Inheritance Example

Step 2

Known Class
Phantom Class
Direct Edge
Path Edge
Step 2

× Nodes C, F have incoming (horizontal) edges
× Node B is not yet higher than neither C nor E
Step 3

Multiple Inheritance Example

Known Class
Phantom Class
Direct Edge
Path Edge
Step 3

✗ Nodes C, F have incoming (horizontal) egdes

Known Class
Phantom Class
Direct Edge
Path Edge
Multiple Inheritance Example

Step 4

- A
- B
- C
- D
- E
- F

Known Class
Phantom Class
Direct Edge
Path Edge
Multiple Inheritance Example

Step 4

✓ Final stratification
Multiple Inheritance Example

Solution

✔ Path-edge (A,B) satisfied through path (A,E,B)
Single Inheritance

Classes
Multiple vs Single Inheritance

Just one additional constraint on the output...which now has to form a

DAG

Tree

Multiple Inheritance

Single Inheritance
* X, Y subtype-related if X subtype or supertype of Y

**Known Class**

**Phantom Class**

**Direct Edge**

**Path Edge**

---

**Must be subtype-related**

---

**Cannot be subtype-related**
Solvers

- Single inheritance
  - Polynomial if no direct-edges to phantom-nodes
  - Worst-case exponential (backtracking)
  - Quite effective in practice
- Multiple inheritance
  - Polynomial
- Single inheritance, multiple subtyping (e.g., Java)
  - Decompose into a single and a multiple inheritance subproblems
JPhantom: Overview

About JPhantom

- Solves the hierarchy complementation problem for all 3 settings
- Uses the ASM framework to operate on bytecode
- Constraint Extraction Step
  - Detects type constraints and missing member references
- Code Generation Step
  - Generates dummy classes, yet consistent with our input
JPhantom: Performance

About JPhantom

• Highly Scalable
  – runs in mere seconds even for large applications and complex constraints
  – 148 phantom classes and 212 constraints, where execution time < 2 sec, for logback-classic
  – Maximum execution time of 14 s for JRuby

• 19 MB binary
In summary, we:

- Introduce the class hierarchy complementation problem
- Provide algorithms for:
  1. single inheritance
  2. multiple inheritance, and
  3. single inheritance multiple subtyping
- Implement our algorithms in JPhantom, a practical tool for program complementation
  - highly scalable
  - meets Java bytecode requirements