USING PROGRAM TRANSFORMATION, ANNOTATION, AND REFLECTION TO CERTIFY A JAVA TYPE RESOLUTION FUNCTION

Victor Winter, Carl Reinke, Jonathan Guerrero

OUTLINE

- Type Resolution
  - Examples and Errors
  - Overview of Algorithm
- Automated Certification
  - Program Transformation
  - Reflection

TYPE RESOLUTION

TYPE RESOLUTION

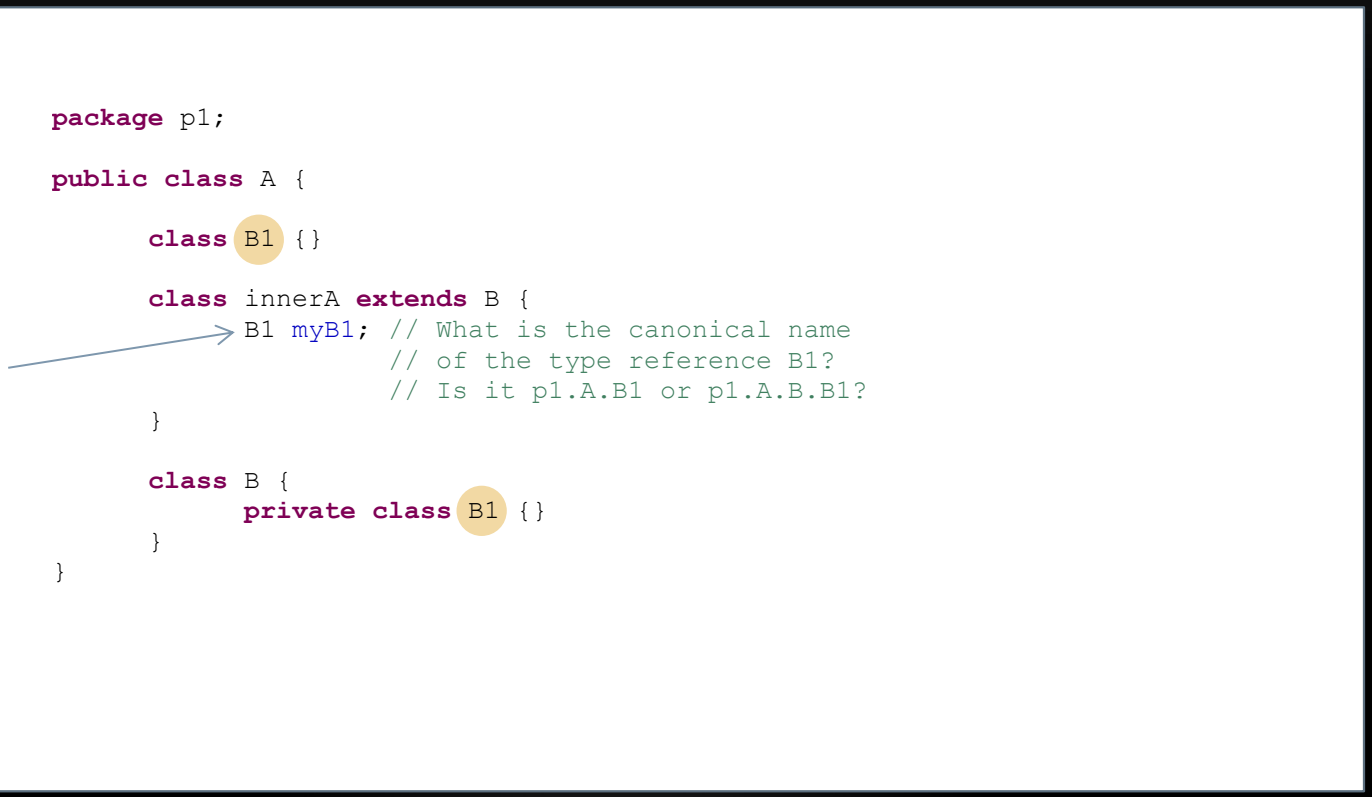
- is a function that takes a reference to a type occurring in a given context as input, and returns the canonical form of that type

  type_reference → canonical form

- is a must have function underlying virtually all forms of semantic-based analysis

- is a complex function – it is quite common to encounter tools where it is implemented incorrectly

ANALYSIS EXAMPLE



```
package p1;

public class A {

	class B1 {}

	class innerA extends B {
		B1 myB1; // What is the canonical name
			 // of the type reference B1?
			 // Is it p1.A.B1 or p1.A.B.B1?
	}

	class B {
		private class B1 {}
	}
}
```

ERROR IN JAVA 1.6.0_26 COMPILER

```
package stackoverflow;

import static stackoverflow.A00.*;

public class A00 extends B00 {
      public static class B00{}
}
```

The discussion that follows gives an approximation to the resolution algorithm used by Java.

RESOLUTION

- In Java, a reference to a type consists of one or more simple identifiers separated by dots.

$$id_1.id_2. \ldots .id_n$$

- The goal of resolution is to convert a reference into its canonical form (e.g., java.lang.Object).

ALGORITHM

- Resolution proceeds in an incremental fashion by resolving the simple identifiers in a reference in a left-to-right fashion.
- Each incremental step produces a result called a resolvent which is then used as the starting point for the incremental step that follows.

- Resolution consists of two algorithms
  - Algorithm I: resolution of primary identifiers
  - Algorithm II: resolution of secondary identifiers

ALGORITHM I

Starting from the type (e.g., class) in which the reference is located do the following:

1. Search the current type for a matching field (or method) declaration.
2. If not found: Search up the hierarchy for a matching field (or method) declaration.
3. If not found: Search static imports of the compilation unit for fields and types with a preference given to fields in contexts where both a field and a type could occur.

4. If not found: Search hierarchy for a matching type declaration.
5. If not found: Search the imports of the compilation unit for a matching single-type import.
6. If not found: Search all compilation units of the current package for a matching type declaration.
7. If not found: Search the imports of the compilation unit for a matching on-demand import. The contents of the package java.lang is implicitly included as an on-demand import.
8. If not found: Search the project for a matching package. If this occurs, continue to process the simple identifiers of the reference until a type identifier is encountered (e.g., java.lang.Object).
9. If not found: Fail.

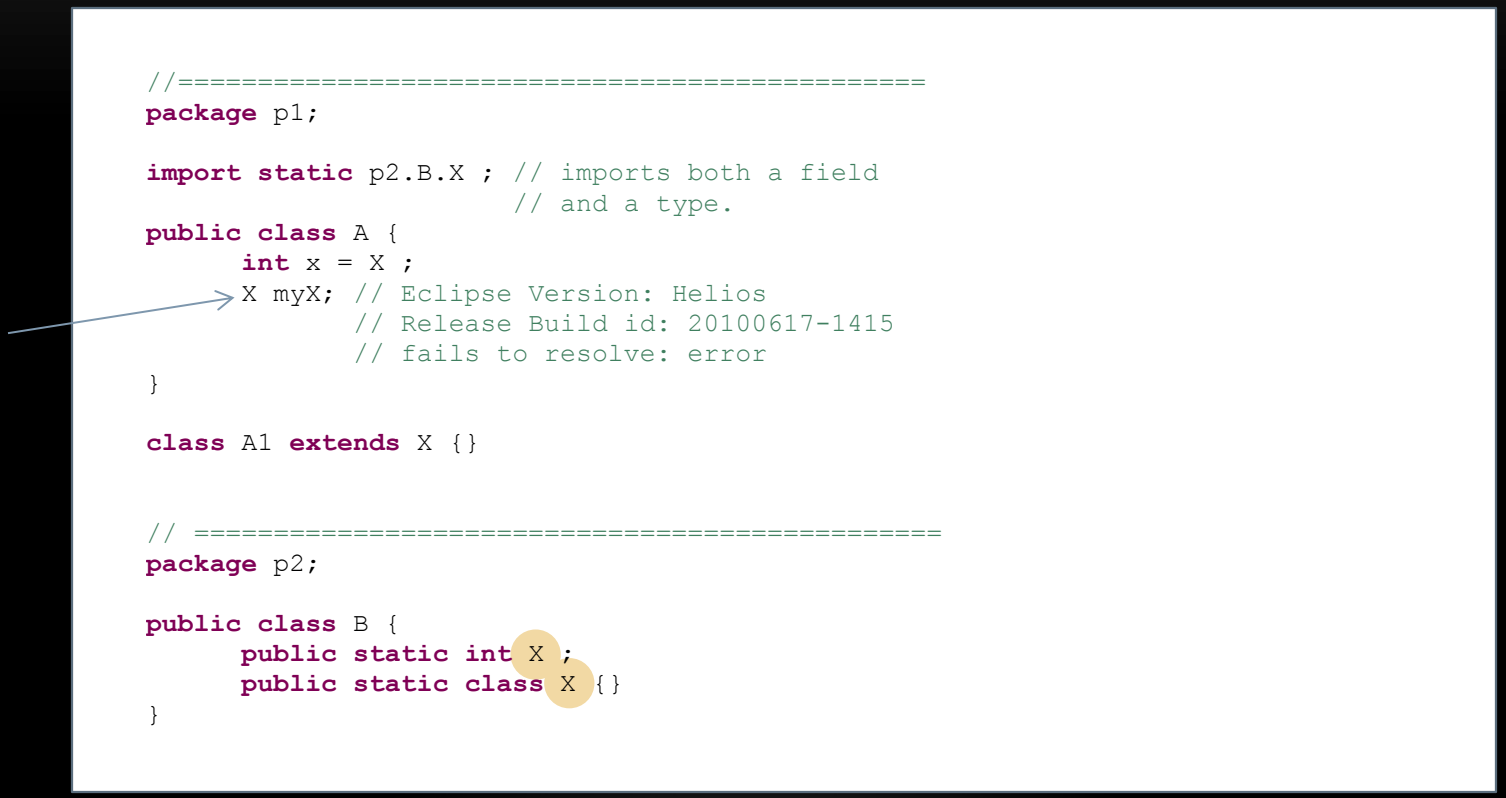
ALGORITHM II

Given the resolvant, r1, obtained from Algorithm I. To resolve the identifier id do the following:

1. r1 = field
    a) Obtain the canonical form for the type of the field and search the hierarchy of this type for a field matching id.

2. r1 = type
    a) Search the inheritance hierarchy of the type for a member declaration (which could be a field or a type) matching id.

ECLIPSE ERROR



```
//================================================
package p1;

import static p2.B.X ; // imports both a field
                       // and a type.
public class A {
      int x = X ;
      X myX; // Eclipse Version: Helios
             // Release Build id: 20100617-1415
             // fails to resolve: error
}

class A1 extends X {}


// ================================================
package p2;

public class B {
      public static int X ;
      public static class X {}
}
```

ADDITIONAL CONSIDERATIONS

- Static imports – introduced into Java in version 5.0
- Nested Types – introduced into Java in version 1.1

From an operational standpoint, it is worth mentioning that the introduction of nested types did not change the JVM. During compilation, all types are flattened leaving only top-level types. The Java compiler accomplishes this by inserting hidden fields, methods, and constructor arguments (as needed) into the newly generated types. The javap disassembler can be used to get a clearer picture of what the compiler actually does in such cases.

ACCESS CONTROL

- Access Control Modifiers
  - public, protected, “package-private”, private

- Protection state can be modeled as a tuple of the form (p1,p2) where
  - p1 is the non-inheritance based protection state (public, package-private, private)
  - p2 is the inheritance based protection state (protected)

- Visibility boundaries affecting p1 occur at: intraclass, intrapackage, interpackage
- Transitions on p1 are monotonic
- The rules governing p2 are orthogonal to those governing p1.

```java
package p1;

import p3.C;
import p3.D;
import p3.E;

public class A extends p2.B {
      public class nestedA {
            C c;
            D d;
            E e;
            int x1 = c.Bi;
            int x2 = d.c.Bi;
            int x3 = e.Bi ; // not visible
            int x4 = c.e.Bi ; // not visible
      }
}
```

```java
// ================================================
package p2;

public class B {
            protected int Bi ;
}

// ================================================
package p3;

public class C extends p1.A {
      public E e;
}

// ================================================
package p3;

public class D {
      public C c;
}

// ================================================
package p3;

public class E extends p2.B {}
```

AUTOMATED CERTIFICATION

AUTOMATED CERTIFICATION

- a novel approach for certifying the correctness of a given type resolution function with respect to an arbitrary Java source code base
- The approach uses program transformation to instrument a subject code base in such a way that reflection can then be used to certify the correctness of the type resolution function against the function used by the Java compiler.
- In this form of certification, the type resolution function of the Java compiler serves as the test oracle.

APPROACH

- Given a set S of Java programs P_i.

$$S = \{ P_1, P_2, \dots, P_n \}$$

- Each P_i is a folder hierarchy containing compilation units (i.e., dot-java files).
- Instrument S yielding

$$S' = \{ P'_1, P'_2, \dots, P'_n \}$$

- Generate a test execution engine: $\text{Certify}_{S'}$
- To certify that our type resolution function is correct for S, simply compile and execute Certify.

SAMPLE OUTPUT

Test result: PASSED

Files inspected = 70
Classes inspected = 350

References correctly resolved = 380
References incorrectly resolved = 0
Unresolved references = 0

Canonical names processed:
p001.A
p001.B
p001.Bucketx29
p002.A
...

INSTRUMENTATION

- Achieved via program transformation within a Java source code analysis and manipulation tool called Sextant.
- Consists of:
  - creating an informatively named corresponding field declaration (CFD) for each type reference encountered within the targeted code base
  - annotating all (non-generic) field declarations with the canonical form of the field type as determined by our resolution function
  - For each compilation unit:
    - add a _validate() method to the primary class (i.e., the public class) of the compilation unit
    - when necessary, add a top-level bucket class to hold CFD's for type references occurring in the extends or implements portions of top-level classes

ANNOTATION

```
@validation.Validator(
                author = "automatic",
                date = "2013-04-11",
                type = "example01.A.C.C1"
)

C.C1 myC3;
```

A MORE TECHNICAL LOOK

- The method Certify contains a static field which is a 2-dimensional array of type string specifying all targeted CU's (.java files). This information is inserted into Certify during transformation.
- When executed, Certify will
  - Move all targeted CU's to a special folder and compile them.
  - Use a ClassLoader to load the primary class of each CU.
    - This primary class contains a method called _validate whose body consists of a call to a method validateClasses. The call to validateClasses is performed with all top-level classes existing within the CU. This information was inserted during transformation.
- The method validateClasses validates, for each class, that the (reflective) type of each field within the class is equal to the annotated type of the field.
  - This validation is recursive, thus the fields of member classes are also checked.
  - A variety of metrics are reported.

```
package example;

public class A extends B {
	C1 f (C2 arg) {
		C1 myC1 = new C1();
		return myC1;
	}

	class nestedA extends innerB {
		C1 g (C2 arg) {
			C1 myC1 = new C1();
			return myC1;
		}
	}
}

class B {
	class innerB {}
	class C1 {}
}

class C1 {}
class C2 {}
```

```
package example;
class Bucketx1 {
      @validation.Validator(…type = "example.B")
      B extensionOfClass_A_1 ;
}

public class A extends B {
      public static boolean __validate() {
            return validation.ClassValidator.validateClasses(Bucketx1.class,
                        A.class,B.class,C1.class,C2.class);
      }

      C1 f( C2 arg ) {
            C1 myC1 = new C1 () ;
            return myC1;
      }

      @validation.Validator(…type = "example.B.C1")
      C1 fromMethod_f_Body_4 ;

      @validation.Validator(…type = "example.C2")
      C2 fromMethod_f_Param_3 ;

      @validation.Validator(…type = "example.B.C1")
      C1 fromMethod_f_Return_2 ;

      class nestedA extends innerB {
            C1 g( C2 arg ) {
                  C1 myC1 = new C1 () ;
                  return myC1;
            }

      @validation.Validator(…type = "example.B.C1")
      C1 fromMethod_g_Body_7 ;

      @validation.Validator(…type = "example.C2")
      C2 fromMethod_g_Param_6 ;

      @validation.Validator(…type = "example.B.C1")
      C1 fromMethod_g_Return_5 ;
}

@validation.Validator(…type = "example.B.innerB")
innerB superTypeOfClass_nestedA_8 ;
}
…
```

THE END

Questions?