JAVA CORE API MIGRATION: CHALLENGES AND TECHNIQUES

Victor Winter, Jonathan Guerrero, Carl Reinke, and James T. Perry
OUTLINE

• Motivation – What is Driving this Effort
• Overview of Our Approach to Migration
• Source Code Analysis Needed
• Technical Details – What to Look Out For
MOTIVATION

The SCore Processor
THE SCORE PROCESSOR

- The **Scalable Core (SCore)** is a hardware implementation of a subset of the JVM, designed and developed at **Sandia National Laboratories (SNL)**, for use in resource constrained high-consequence embedded systems.

- Within the Score:
  - **Supported Java bytecodes** are realized in microcode.
  - **Native methods** used in the JVM and supported by the SCore are also implemented in microcode.
# JAVA LANGUAGE RESTRICTIONS

<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant Keywords</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>strictfp, float, double</td>
<td>unsupported</td>
</tr>
<tr>
<td>Threading</td>
<td>synchronized, volatile</td>
<td>unsupported</td>
</tr>
<tr>
<td>Serialization</td>
<td>transient</td>
<td>unsupported</td>
</tr>
<tr>
<td>Assertions</td>
<td>assert</td>
<td>unsupported</td>
</tr>
<tr>
<td>Multi-dimensional Arrays</td>
<td></td>
<td>unsupported</td>
</tr>
</tbody>
</table>
## VM RESTRICTIONS

<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant Keywords</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native methods</td>
<td>native</td>
<td>Limited support</td>
</tr>
<tr>
<td>Garbage collection</td>
<td></td>
<td>Limited support</td>
</tr>
<tr>
<td>Reflection</td>
<td></td>
<td>Unsupported</td>
</tr>
<tr>
<td>(dynamic) class loading</td>
<td></td>
<td>Unsupported</td>
</tr>
</tbody>
</table>
APPLICATION DEVELOPMENT ON THE SCORE

- API references
- API
- Standard Desktop
- Application
- Monarch
- Migrated API
- Java Compiler
- API and Application Class files
- Interlude
- ROM Image
MIGRATION
OVERVIEW OF MIGRATION

- (Manual) **Re-implementation Stage** – preserve “must have” functionality.
- (Manual) **Preparation Stage** – expand target code base to create a subtype hierarchy rooted at `java.lang.Object`
- (Automatic) **Removal Stage** – transformation driven removal of source code having dependencies on features of Java that are not supported by the SCore.
JAVA SOURCE CODE ANALYSIS
GOAL OF REMOVAL STAGE

From the targeted code base, remove all fields, methods, and constructors having direct or indirect dependencies on

1. features that are not supported by the SCore, or
2. external references.
We define a reference as a source-code expression (i.e., valid Java syntax) referring to a declared element.

Such denotations can be in relative terms, in indirect terms (also known as aliases), and in absolute terms (also known as canonical forms).

In this presentation, the term resolution analysis is used to refer to static analysis whose purpose is to determine the relation $R$ between element references and element declarations.
PRIMARY AND SECONDARY RESOLVENTS

- Java allows for the creation of code structures in which certain declarations are hidden (i.e., not visible) from the environment (T) in which the reference occurs. When they exist, we refer to such hidden declarations as secondary resolvents of the reference.

- Declarations that are not hidden are called primary resolvents.

- A central concern is whether the migration process, when seen as a whole, creates conditions for the reclassification, by the Java compiler, of a secondary resolvent as the (new) primary resolvent for a given reference. If this reclassification occurs, then migration is not correctness preserving.

In order to assure migration correctness, Monarch may remove arbitrary declarations (e.g., secondary resolvents) during migration. However, such removal should be kept to a minimum.
ANALYSIS ISSUES

- **Points-to analysis** – undecidable
- **External references** – ideally, source code migration would be performed on a code base that is reference-closed. Unfortunately, this is unrealistic in practice. A simple “hello world” program when executed via the command `java -verbose:class` demonstrates that this tiny program loads (e.g., has dependencies on) over 400 classes.

![Diagram showing object relationships to target code base](image-url)
APPLICATION INDEPENDENCE

• Migration may not extend its dependency analysis to applications.

• Rationale
  • Migration would need to be performed on a per-application basis.
  • Security policy of a project may prohibit tools from accessing application code.
Migration of a targeted code base C must be performed in an **application independent fashion**. Specifically, migration may not make any assumptions about how an application might use C.

The Interlude classloader **should fail** to create a ROM image when the class files of an application contain a reference to a field, method or constructor that does not exist in the migrated target code base.
Shadowing, overriding, overloading, constructor removal, and initialization blocks.
package p1;

public class A {
    int y1 = new B2().x1; // p1.B2.x1
    int y2 = (int) new B2().x2; // p1.B2.x2
    int y3 = (int) new B2().x3; // p1.B1.x3
}

class B1 {
    int x1 = 1;
    int x2 = 2;
    int x3 = 3;
}

class B2 extends B1 {
    // will be removed by Monarch, shadows p1.B1.x1
    int x1 = (int) 1.0;
    // will be removed by Monarch, shadows p1.B1.x2
    double x2 = 2.0;
}
In order to assure the correctness of migration, when removing the field declaration T.x it is also necessary to remove all field declarations shadowed by T.x.
METHOD SHADOWING

```java
package p1;

public class A {
    public int f() { return 1; }
    public int g() { return 1; }

    public class InnerA {
        public int f() { return (int) 2.0; } // shadowing method removed during migration
        public int y1 = f(); // depends on p1.A.InnerA.f()
        public int y2 = g();
        public int y3 = z1;
    }

    public int z1 = 1;
    InnerA myThing = new InnerA();
    // int z2 = myThing.g(); // compile error
    // int z3 = myThing.z1; // compile error
}

public int x1 = myThing.f();
// int x2 = myThing.g(); // compile error
int y1 = myThing.y1;
int y2 = myThing.y2;
// int y3 = myThing.z1; // compile error
```

//External to Monarch analysis
package app;

import p1.A;

class App {
    A.InnerA myThing = (new A()).new InnerA();
    int x1 = myThing.f();
    // int x2 = myThing.g(); // compile error
    int y1 = myThing.y1;
    int y2 = myThing.y2;
    // int y3 = myThing.z1; // compile error
}
METHOD SHADOWING

- A target code base C satisfies the **containment property** if whenever a top-level class belongs to C, all its internals (e.g., nested classes) also belong to C.

In order to be suitable for migration, a target code base C must satisfy the containment property.
package p1;  // Target code base

class A extends B {
    public int f() { return 1; }  // overrides p1.B.f()
    public int g() { return (int)1.0; }  // overrides p1.B.g() will be removed during migration
}

class B {
    public int f() { return (int)2.0; }  // overridden by p1.A.f() will be removed during migration
    public int g() { return 2; }  // overridden by p1.A.f()
}

package app;
import p1.*;

class App {
    A myA = new A();
    int x1 = myA.f();  // p1.A.f()
    int x2 = myA.g();  // p1.A.g()
    B myB = new B();
    int x3 = myB.f();  // p1.B.f()
    int x4 = myB.g();  // p1.B.g()
    B myThing = new A();
    int x5 = myThing.f();
    int x6 = myThing.g();
}
METHOD OVERRIDING

When removing method T.m all methods overridden by T.m must also be removed.
METHOD OVERLOADING

- public StringBuilder append(Object obj) ...
- public StringBuilder append(String str) ...
- ...
- public StringBuilder append(float f) ...
- public StringBuilder append(double d) ...
Secondary resolvents resulting from references to overloaded methods/constructors impose no additional constraints on migration because Interlude will fail to produce a ROM image if it encounters a reference to a non-existent method or constructor.
Migration may not exhaustively remove all explicitly declared constructors belonging to a type.
INITIALIZATION BLOCKS

- Static and Instance

The removal phase of migration classifies initialization blocks as “must have” functionality. Furthermore, Monarch will migrate all initialization blocks without performing dependency analysis. It is left to the re-implementation phase to assure that all initialization blocks are free from unwanted dependencies.
The End