Generative Programming Techniques for Adaptation of Java Libraries to Embedded Systems

Victor Winter and Azamat Mametjanov
Department of Computer Science
College of Information Science and Technology
University of Nebraska at Omaha

CIST Round Table
Omaha, Nebraska
Friday September 7, 2007
Abstract

- Embedded Java programming needs library support
- New library development and/or manual library adaptation is resource-intensive
- Program transformations are used to remove unsupported instructions from the standard Java libraries
- This approach results in
  - near-perfect ($\approx 100\%$) migration rate
  - rapid adaptation, e.g. 30x faster on the java.lang library
  - re-targeting of the libraries to heterogeneous embedded platforms with different instruction sets
Outline

Java Library Migration
  Problem Statement
  Overview of the Migration

Program Transformations: HATS
  Overview
  Implementation

Results

Conclusion
Java Library Migration
Outline

Java Library Migration
   Problem Statement
   Overview of the Migration

Program Transformations: HATS
   Overview
   Implementation

Results

Conclusion
Embedded programming

Programming for embedded systems is growing in size and complexity

- In 2000, 8 billion microprocessors were built, of which 98% were used in embedded systems
- Embedded systems are deployed in increasingly complex applications: e.g. medical applications, weapons systems, space borne systems, flight control systems

Embedded programming needs high-level language support
Embedded JVM

The Java Virtual Machine architecture can be implemented in hardware

- Program memory to store instructions
- Stack memory to store method frames and the opstack
- Heap memory to store objects and static fields
- Execution core: ALU, registers, instruction decode etc.

Embedded programmers can now code in Java.
Library support

The Java libraries are fundamental for programming in Java

- The java.lang library is a system library with foundational classes such as Object, String, and Class

- The java.util library is the most commonly used utility library providing data structure implementations such as Stack, Queue, Vector etc.

- Other specialized libraries provide abstractions and API to avoid recreating known implementations and encourage reuse: e.g. java.io for I/O, java.util.zip for ZIP compression, and java.util.regex for regular expressions
Problem Statement

Due to resource constraints, embedded implementations of the JVM (jVMs) do not typically implement all 201 JVM instructions: e.g.

- floating-point arithmetic requires FPU hardware: e.g. fadd, fdiv
- multi-threaded execution involves locks, monitors, and other thread synchronization overhead: e.g. monitorenter, monitorexit
- native library methods need to be re-implemented in embedded system code: e.g. Object.hashCode()

The standard libraries assume support for all JVM instructions and features. This makes the libraries *incompatible* with embedded jVMs.
Related Work

Refactoring of Java code

► semantics-preserving structure modification
► used in profiling and optimization of Java libraries

Synchronizing client software with library evolution

► remove references to deprecated code
► convert legacy code with downcasts into generic code with type parameters

Developer of the standard Java libraries, Sun Microsystems, does not provide migrated version of the libraries for use on the embedded Java Card platform.
Outline

Java Library Migration
  Problem Statement
  Overview of the Migration

Program Transformations: HATS
  Overview
  Implementation

Results

Conclusion
Migration Approaches

Removal of unsupported code

- **Advantage:** Simple, generic w.r.t. unsupported features
- **Disadvantage:** Reduced functionality of the libraries

Re-implementation using supported features

- **Advantage:** Retain as much functionality as possible
- **Disadvantage:** Risk of introducing bugs into mature, tested libraries

Since correctness is of utmost importance, the approach based on removal is more attractive.
Methods

- Manually migrate the `java.lang` library
- Record removal patterns
- Implement removal patterns in program transformations
- Compare manual vs. automated migration
Constraints

Removal is a sequence of transformation steps

\[ T_i \overset{\text{def}}{=} C[\ldots t_i \ldots] \rightarrow C[\ldots \epsilon \ldots] \]

subject to the following 4 constraints:

**Syntactic correctness** Terms resulting from transformations must be syntactically well-formed: e.g.

\[
\text{int } x = (\text{int})1.0 \quad \rightarrow \quad \text{int } x = (\text{int})
\]

**Semantic refinement** Removal of unsupported code entails removal of all transitive references to the unsupported code:

\[
\begin{align*}
\text{int } x &= 0; \\
\text{int } f()\{\text{double } x = 1.0; \text{ return } (\text{int})x; \} \quad &\overset{\text{wrong}}{\rightarrow} \\
\text{int } x &= 0; \\
\text{int } f()\{\text{return } (\text{int})x; \}
\end{align*}
\]
Constraints

**Full coverage**  The library resulting from the transformations must consist solely of supported elements

**Minimality**  Transformations need to retain as much of the original library as possible. Removals can occur at the level of

- Classes
- Class members (fields, methods, constructors)
- Statements within class members

Inspection of libraries suggests that members are loosely coupled, while statements are tightly coupled. Thus, member removal instead of statement removal is chosen.
Removal policy

A Java class is a list of members. A class member is

- **Atomic**
  - Field
  - Method
  - Constructor
  - Initializer

- **Composite** Inner class

  - If an atomic member contains an unsupported element, then the member is removed.
  - If an inner class member contains an unsupported element, then the member is removed, but the inner class is retained.
Transitive dependencies

- If a member is removed, all references to the removed member also need to be removed.
- For example, if a member $m$ in class $A$ is removed, then the following references are transitive dependencies: $m$, $this.m$, $A.m$, $new A().m$.
- We write $M[\ldots \hat{t} \ldots]$ to denote reference to $t$ within $M$.

Automated computation of the closure of transitively dependent members, set $R_C$, is the key contribution of our approach.
Floating-point dependency rules

\[ \text{Floating-point dependency rules} \]

\[ FpLiteral \in R_C \quad \text{(Axiom-lit)} \]

\[ Modifier[\text{strictfp}] \in R_C \quad \text{(Axiom-strictfp)} \]

\[ \text{BasicType}[\text{float}] \in R_C \quad \text{(Axiom-float)} \]

\[ \text{BasicType}[\text{double}] \in R_C \quad \text{(Axiom-double)} \]

\[ C[\ldots m \ldots] \quad t \in R_C \quad m[\ldots \hat{t} \ldots] \quad \text{ref-to}(m) \in R_C \quad \text{(T-transitive)} \]
**Migration rules**

- **For atomic members** \( m \) (fields, methods, constructors) and initializer blocks \( b \):

\[
\begin{array}{c}
C[\ldots mb\ldots] \quad t \in R_C \quad mb[\ldots \hat{t} \ldots] \\
\hline
C[\ldots mb\ldots] \rightarrow C[\ldots \epsilon \ldots]
\end{array}
\]  

(T-migrate1)

- **For composite members** \( x \) (inner classes):

\[
\begin{array}{c}
C[\ldots x \ldots] \quad t \in R_C \quad x[\ldots \hat{t} \ldots] \\
\hline
x[\ldots \hat{t} \ldots] \rightarrow x[\ldots \epsilon \ldots]
\end{array}
\]  

(T-migrate2)
Example

class Cafe {
    float fabulous = 0xCAFEBABE;
    int innocent = (int) fabulous;

    // initializer block, use-before-def
    innocent = 1;

    // initializer block, def-before-use
    int innocent;
    innocent = 0;
}
}
Program Transformations: HATS
Context of Program Transformations

Outline

Java Library Migration
  Problem Statement
  Overview of the Migration

Program Transformations: HATS
  Overview
  Implementation

Results

Conclusion
Overview of Key Concepts

- Object language
- Matching
- Rewrite rules
- Traversals
- Combinators
Object language

- Grammar $G = (N, T, P, S)$
- Instance of the derivation $B \xrightarrow{*} \alpha$ is denoted by $B[\alpha']$
- E.g. grammar:

```plaintext
Expr ::= Expr "+" Term | Term
Term ::= Term "*" Factor | Factor
Factor ::= id | integer | "(" Expr "")"
```

E.g. parse expressions: $Expr_1, Expr[Term_1 + 5]$, and $Factor[(Expr_1)]$

- Java’s grammar contains 123 non-terminal symbols, 107 terminal symbols, and 325 productions
Matching

- First-order syntactic *match equations*: e.g.
  \[ expr[expr_1 + term_1] \ll expr[a + b \times c] \]
  succeeds with substitution \( \sigma = \{ (expr_1, expr[a]), (term_1, term[b \times c]) \} \)

- Boolean composition of match equations – *match expressions*:

  \[
  \begin{align*}
  \sigma(t_1 \ll t_2) & \equiv \sigma(t_1) = t_2 \\
  \sigma(e_1 \land e_2) & \equiv \sigma(e_1) \land \sigma(e_2) \\
  \sigma(e_1 \lor e_2) & \equiv \sigma(e_1) \lor \sigma(e_2) \\
  \sigma(\neg e_1) & \equiv \neg \sigma(e_1)
  \end{align*}
  \]
Rewrite rules

\[ \text{lbl} : \text{lhs} \rightarrow \text{rhs} [ \text{if me} ] \]

- Higher-order rewrite rules:
  - Parse expression \( p \) – rule of order 0: \( r^0 \)
  - First-order rule: \( r^0 \rightarrow r^0 \)
  - Higher-order rule of order \( n + 1 \): \( r^0 \rightarrow r^n \)
- Built-in rewrites: identity \( ID \) and no-op \( SKIP \)
- E.g. distributivity of multiplication over addition:

\[
\text{distribute} : \text{term}[\text{term}_1 \ast (\text{expr}_2 + \text{term}_2)] \rightarrow \\
\text{term}[(\text{term}_1 \ast (\text{expr}_2) + \text{term}_1 \ast (\text{term}_2))]}
\]
Traversals

- Parse trees are isomorphic to terms of the form

\[ t ::= f(t_1, t_2, \ldots, t_n), n \geq 0 \]

- Traversals serialize a complex term into a sequence of sub-terms: e.g. bottom-up left-to-right traversal \( t_1, t_2, \ldots, t_n, f \)

- Several traversals are built-in: \textit{TDL, TDR, BUL, BUR}

- Application of a rule \( r \) to term \( t \) using traversal \( v \) is denoted by \( v\{r\}(t) \): e.g. \( \text{TDL\{distribute\}}(\text{expr}_1) \)

- New traversals can be defined using \textit{def}: e.g.

\[
\text{def first\_TDL } s = s <\text{mapL(first\_TDL}\{s\})
\]
Combinators

Transformations are performed incrementally

\[ \mathcal{T}_{1,n} \overset{\text{def}}{=} \mathcal{T}_1 \oplus \mathcal{T}_2 \oplus \ldots \oplus \mathcal{T}_n \]

- Sequential composition \(<;\)

\[ \mathcal{T}_{1,n} \equiv \mathcal{T}_1 <; \mathcal{T}_2 <; \ldots <; \mathcal{T}_n \equiv \mathcal{T}_n \circ \ldots \circ \mathcal{T}_1 \]

- Conditional composition \(\leftarrow\)

\[ \mathcal{T}_{1,n} \equiv \mathcal{T}_1 \leftarrow \ldots \leftarrow \mathcal{T}_n \equiv \]

  if \( \text{check}(\mathcal{T}_1) \)

  then \( \text{ID} \)

  else \ldots

  if \( \text{check}(\mathcal{T}_n) \)

  then \( \text{ID} \)

  else \( \text{SKIP} \)
Definition of Strategy

- **Basis**: A (conditional) rewrite rule is a strategy
- **Induction**: An expression composed of strategies, combinators, and iterators is a strategy
Example: Distributivity

\[
\begin{align*}
\text{deparen}\_\text{expr} & : \text{expr}[(\text{expr}_0)] \rightarrow \text{expr}[\text{expr}_0] \\
\text{fuse}\_\text{expr} & : \text{expr}[\text{expr}_0 + (\text{expr}_1 + \text{term}_0)] \rightarrow \\
& \quad \text{expr}[\text{expr}_{\text{new}} + \text{term}_0] \\
\text{if} & \text{ expr}_{\text{new}} \ll \text{first}\_\text{TDL}\{	ext{expr}[\text{term}_{\text{some}}]\} \rightarrow \\
& \quad \text{expr}[\text{expr}_0 + \text{term}_{\text{some}}]\}(\text{expr}_1) \\
\text{compose} & : \text{distribute} <; \text{fuse}\_\text{expr} <; \text{deparen}\_\text{expr} \\
\text{main} & : \text{FIX}\{\text{TDL}\{\text{compose}\}\}\}
\end{align*}
\]

- Input: \((a + b) \ast (p + (q + r))\)
- Output: \(a \ast p + b \ast p + a \ast q + a \ast r + b \ast q + b \ast r\)
Outline

Java Library Migration
   Problem Statement
   Overview of the Migration

Program Transformations: HATS
   Overview
   Implementation

Results

Conclusion
class A {
    float x;
    native int foo();
}

class B extends A {
    int global() { return foo(); }
    int local() { return global(); }
    int fix() { return new Z().bar(); }
}

class Z {
    int bar() {
        Class claz = this.getClass();
        return claz.hashCode();
    }
}

Step 0: field: A.x
Step 1: method: B, global
Step 4: method: B, fix
Step 2: method: Z, bar
Step 3: method: B, local

Transformations

Fixed-point

Iteration: i

Step: 3i 3i+1 3i+2

assert/synchronized/transient/volatile

Keywords

Base filter

float/double/strictfp native floating-point literals unsupported classes

Removed member log
method: Object, getClass
method: Class, getClassLoader ...

Schematic Representation
## Driver strategy

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>migration</td>
<td>$\text{CompilationUnit}<em>{in} \rightarrow \text{closure}{\text{fold} \leftrightarrow (\text{base} \leftrightarrow \text{add})}(\text{CompilationUnit}</em>{in})$</td>
</tr>
<tr>
<td>def closure src</td>
<td>$\text{FIX}{ \text{proc_mem}{s_rc}}$</td>
</tr>
<tr>
<td>def proc_mem src</td>
<td>$\text{(occurs_t_hat}{s_rc} \leftrightarrow (\text{deduct_step}{s_rc} \text{; } \text{proc_mem}{s_rc})) \leftrightarrow \text{proc_next_mem}{s_rc}$</td>
</tr>
<tr>
<td>def deduct_step src</td>
<td>$s_rc \leftrightarrow s_rc \quad (* \text{Apply } T\text{-migrate and } T\text{-transitive} *)$</td>
</tr>
<tr>
<td>def occurs_t_hat src</td>
<td>$\text{Members}[\text{Comments}_1 \text{ Member}_1 \text{ Members}_1 ] \rightarrow \text{Members}[\text{Comments}_1 \text{ Member}_1 \text{ Members}_1 ]$&lt;br&gt;if $T\text{DL}{s_rc}(\text{Member}_1)$&lt;br&gt;$\leftrightarrow \ldots$&lt;br&gt;($* \text{ rules for inner classes and interface members } *$)</td>
</tr>
<tr>
<td>def proc_next_mem src</td>
<td>$\text{Members}[\text{Comments}_1 \text{ Member}_1 \text{ Members}_1 ] \rightarrow \text{Members}[\text{Comments}_1 \text{ Member}_1 \text{ Members}_2 ]$&lt;br&gt;if $\text{Members}_2 \ll \text{process_member}{s_rc}{\text{Members}_1}$&lt;br&gt;$\leftrightarrow \ldots$&lt;br&gt;($* \text{ similar rules for interface members } *$)</td>
</tr>
</tbody>
</table>
Base filter strategies

axiom_lit: \( FpLiteral_1 \rightarrow FpLiteral_1 \)

axiom_strictfp: \( Modifier[\text{strictfp}] \rightarrow Modifier[\text{strictfp}] \)

axiom_float: \( \text{BasicType}[\text{float}] \rightarrow \text{BasicType}[\text{float}] \)

axiom_double: \( \text{BasicType}[\text{double}] \rightarrow \text{BasicType}[\text{double}] \)

axiom_supported: \( \text{QualifiedType}[\text{Id}_1, \text{TypeArgsOpt}_1] \rightarrow \text{QualifiedType}[\text{Id}_1, \text{TypeArgsOpt}_1] \)
if \( \text{TypeList}_1 \ll \text{TypeList}[\text{Object, System, Integer, ...}] \) (* list of supported classes *)
and not \( \text{TDL}\{\text{Id}_1 \rightarrow \text{Id}_1}\}(\text{TypeList}_1) \)

base: \( \text{axiom_lit} \leftrightarrow \text{axiom_strictfp} \leftrightarrow \text{axiom_float} \leftrightarrow \text{axiom_double} \leftrightarrow \text{axiom_supported} \)
New filter strategies

**add:**
- remove_field_and_add_dependency
- remove_method_and_add_dependency
- remove_constructor_and_add_dependency
- remove_init_block

**remove_field_and_add_dependency:**

Members\\[
\text{Comments}_1 \text{ Field}_1 \text{ Members}_1 ]

\rightarrow

(\text{transient}(\text{Members}[\text{Comments}_1 \text{ Field}_1 \text{ Members}_1 ] \rightarrow \text{Members}_1)) (**T\text{-migrate: field removal rule}**) \\
\leftrightarrow

\text{gen\_field\_reference\_recognizer}[\text{Field}_1]

(* T-transitive: adding the reference Field}_1 to s\_rc *)

**remove_method_and_add_dependency:** . . .

**remove_constructor_and_add_dependency:** . . .

**remove_init_block:** . . .
class Cafe {
    float fabulous = 0xC0FEBABE; // catch me if you can
    int innocent = (int) fabulous; // masquerade as int

    Cafe(int innocent) {
        this.fabulous = 0xDD; // explicit ref to FP field
        this.innocent = innocent;
    }

    int environment(int innocent) { // re-decl in params
        innocent = 0xFACADE;
        return innocent;
    }

    int environment() {
        int harmless = innocent; // use comes before decl
        int innocent = harmless;
        return innocent;
    }

    int order(int choice) {
        int waitress;
        if (choice == 0xC0FFEE) {
            int innocent = 0xBABE; // re-decl in local if-block
            waitress = innocent;
        } else {
            waitress = innocent; // indirect ref to FP field
        }
        return waitress;
    }
}
Example – Output

class Cafe {
    int environment(int innocent) {
        // re-decl in params
        innocent = 0xFACADE;
        return innocent;
    }
}

Results
Unsupported Feature Profile

**synchronized** serialized access to data or execution of code; compiles to unsupported opcodes monitorenter, monitorexit

**volatile** thread coherence; multi-threading is no supported

**transient** I/O persistence; persistence is not supported

**assert** assertion-checking with dependencies on reflection features of a JVM; reflection is not supported

**native** code implemented in target platform’s native code; not supported unless implemented on the embedded platform

**floating-point** keywords float, double, strictfp

**other** unsupported classes: e.g. Classloader, Compiler, etc.
Experiment on java.lang package

- Originally, 107 source files with a total of 41620 SLOC
- 35 classes are unsupported (e.g. Float, Double, Thread etc.)
- 53 classes remained intact after the transformations
- 19 classes contained unsupported elements
Example: Throwable

- **Input:** 26 members in 651 SLOC
- **Transform:** 16 members were removed
  - No members containing floating point ops (FP)
  - 5 members removed due to references to unsupported classes (UC)
  - 11 members removed due to references to native code (NM)
- **Manual adjustment:** removal of 4 constructors resulted in compile errors in all Error and Exception classes. The constructors were put back; statements referencing native code were removed (`fillInStackTrace()`)
- **Output:** 14 members in 356 SLOC
<table>
<thead>
<tr>
<th>Class</th>
<th>SLOC</th>
<th>Orig</th>
<th>Auto</th>
<th>FP</th>
<th>UC</th>
<th>NM</th>
<th>Post</th>
<th>End</th>
<th>SLOC'</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsStringBuild</td>
<td>1423</td>
<td>59</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>1361</td>
</tr>
<tr>
<td>AssertionError</td>
<td>134</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>111</td>
</tr>
<tr>
<td>Boolean</td>
<td>247</td>
<td>18</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>210</td>
</tr>
<tr>
<td>Byte</td>
<td>442</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>415</td>
</tr>
<tr>
<td>Character</td>
<td>5465</td>
<td>158</td>
<td>33</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>125</td>
<td>2141</td>
</tr>
<tr>
<td>Enum</td>
<td>200</td>
<td>13</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>132</td>
</tr>
<tr>
<td>EnumCNotPEx</td>
<td>55</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>44</td>
</tr>
<tr>
<td>Integer</td>
<td>1163</td>
<td>50</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>44</td>
<td>1014</td>
</tr>
<tr>
<td>Long</td>
<td>1145</td>
<td>46</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>1011</td>
</tr>
<tr>
<td>Math</td>
<td>1513</td>
<td>61</td>
<td>52</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>179</td>
</tr>
<tr>
<td>Number</td>
<td>95</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>79</td>
</tr>
<tr>
<td>Object</td>
<td>536</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>6</td>
<td>250</td>
</tr>
<tr>
<td>Readable</td>
<td>38</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Short</td>
<td>457</td>
<td>27</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>430</td>
</tr>
<tr>
<td>String</td>
<td>3003</td>
<td>93</td>
<td>28</td>
<td>2</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>65</td>
<td>1834</td>
</tr>
<tr>
<td>StringBuilder</td>
<td>463</td>
<td>43</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>-1</td>
<td>35</td>
<td>351</td>
</tr>
<tr>
<td>System</td>
<td>1150</td>
<td>46</td>
<td>39</td>
<td>0</td>
<td>25</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>323</td>
</tr>
<tr>
<td>Throwable</td>
<td>651</td>
<td>26</td>
<td>16</td>
<td>0</td>
<td>5</td>
<td>11</td>
<td>-4</td>
<td>14</td>
<td>356</td>
</tr>
<tr>
<td>Void</td>
<td>31</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>18211</td>
<td>703</td>
<td>220</td>
<td>80</td>
<td>107</td>
<td>33</td>
<td>-5</td>
<td>488</td>
<td>10287</td>
</tr>
</tbody>
</table>
Summary

Initial investment of 3 man/months:
- 132 SLOC of Java lexer
- 277 SLOC of Java BNF
- 2655 SLOC of Java pretty-printer
- 1133 SLOC of transformation code

Payoff:
- 1 day automated vs. 30 days manual transformation of the java.lang library
- 1 day automated vs. X days manual transformation of java.util, java.io, java.nio, etc. libraries
- transformations are reusable for a single evolving platform or multiple different platforms
Conclusion
Future Work

▶ Industrial rule-of-thumb: investment into automation is well-justified if 75% of legacy code is migrated using automation
▶ Our lightweight syntactic analysis achieved 97.67% migration rate
▶ Further improvement can be achieved using semantic analysis to track dependencies across type hierarchies:
  ▶ `super.m`: if `m` is removed in the super class, then `super.m` should be removed in subclasses
  ▶ `DeclaringAbstractClassOrInterface.m`: if `m` is removed in an implementing class, then the declaration of `m` should be removed in the abstract class or interface
  ▶ `InheritingClass.m`: if `m` is removed in the super class, then references to `m` should be removed in all subclasses
Conclusion

- Embedded Java programming needs library support
- New library development and/or manual library adaptation is resource-intensive
- Program transformations were used to remove unsupported instructions from the standard Java libraries
- This approach produced
  - near-perfect ($\approx 100\%$) migration rate
  - rapid adaptation, e.g. 30x faster on the java.lang library
  - re-targeting of the libraries to heterogeneous embedded platforms with different instruction sets
References and Details
