Decompiling Java: Problems, Traps, and Pitfalls

Jerome Miecznikowski and Laurie Hendren

Sable Research Group
School of Computer Science
McGill University, Montreal, Canada

http://www.sable.mcgill.ca
Overview

1. Motivation

2. Basic issues for typed statements
   - simple statements
   - types

3. Advanced issues for restructuring
   - multi-entry point loops
   - labeled blocks and break statements
   - exceptions & thread synchronization

4. Putting it all together

5. Conclusions
Motivation

The facts are ...

- Java bytecode is rich in type information, and is much higher level than traditional machine code

- Bytecode generated from javac follows specific code generation patterns

So shouldn’t decompiling simply be a matter of inverting javac’s compilation strategy?
No!

What we found:

- Java bytecode is much more flexible than what can be expressed in any structured language

- Bytecode optimizers and compilers for other languages will produce radically different patterns in code generation. These patterns can get very complex

- Type information for locals has to be treated carefully regardless of source

Conclusion: We want to show some interesting problems in decompiling to Java, see how other decompilers fare, and suggest our own workable strategies
Background Questions

• What are these “other” decompilers?
  – Jasmine version 1.10, Jad version 1.5.8, Wingdis version 2.16, and SourceAgain version 1.1

• What does Java bytecode look like?
  1. uses an expression stack
  2. has explicit control flow
  3. supports exceptions
  4. supports thread synchronization
Some bytecode (javap -c)

Method int f(java.lang.Object, int)
  0 iconst_5
  1 istore_3
  2 goto 32
  5 aload_1
  6 astore 4
  8 aload 4
 10 monitorenter
 11 iload_3
 12 iload_2
 13 iinc
 16 imul
 17 istore_3
 18 aload 4
 20 monitorexit
 21 goto 32
 24 astore 5
 26 aload 4
 28 monitorexit
 29 aload 5
 31 athrow
 32 iload_2
 33 bipush 10
 35 if_icmplt 5
 38 iload_3
 39 ireturn

Exception table:
  from    to  target  type
    11    24    24  any
Basic Issues for Typed Statements

1. Simple Statements

- The Java virtual machine uses an expression stack

- javac compilation pattern: the expression stack will be empty after every program statement

- Even simple optimizations can leave values on the stack after a “program statement” (example is given in paper)

- All other tested decompilers were confused by this and produced incorrect output. (dropped statements, lost locals, error messages in code, etc.)
Our working solution is to ...

1. represent stack positions as locals

2. split locals by using U-D webs

3. build 3-address code using the locals

4. aggregate expressions of 3-address code

See Raja Valée-Rai’s Master’s Thesis:

_Soot: A Java Bytecode Optimization Framework_
2. Types

In bytecode, **fields** have types but **locals** don’t

```
public static void f(short i)
{  <unknown> c; <unknown> r; <unknown> d;
   <unknown> is_fat;

   if (i>10)
   {  r = new Rectangle(i, i);
       is_fat = r.isFat();
       d = r;
   }
   else
   {  c = new Circle(i);
       is_fat = c.isFat();
       d = c;
   }
   if (is_fat == 0) d.draw();
}
```
**Problem:** Given the following class hierarchy, how to determine the type of “d”?

**Solution:**
1) Create a type constraint graph based on the class hierarchy, local assignments, and local uses
2) Prune and collapse the graph to get precise types

See Gagnon, et.al. from SAS2000:

_Efficient Inference of Static Types for Java Bytecode_
All other decompilers failed in both:

- Handling simple statement creation on stack optimized code
- Correctly finding that \texttt{a} is of type \textit{Drawable}

The following 5 slides show all tested decompilers’ output on this example.

1. The class was first compiled with \texttt{javac}

2. Then it was optimized by a simple peephole optimizer
Output for: **Jasmine**

```java
public static void f(short s)
{
    Object object;
    if (s <= 10) goto 24 else 6;
    expression new Rectangle
dup 1 over 0
expression s
dup 1 over 0
invoke Rectangle.<init>
dup 1 over 0
invoke isFat
swap
pop object
expression new Circle(s)
dup 1 over 0
invoke isFat
swap
pop object
if != goto 47
object.draw();
}
```
Output for: **Wingdis**

```java
public static void f(short short0)
{
    if (((((byte)short0) <= 10))?
        (Circle circle1= new Circle(short0)):
            (Rectangle rectan1=
                new Rectangle(
                    ((short)short0), ((short)short0)))
            == false)
    { Drawable.draw();
    }
}
```
public static void f(short word0)
{
    Rectangle rectangle;
    if(word0 <= 10)
        break MISSING_BLOCK_LABEL_24;
    rectangle =
        new Rectangle(word0, word0);
    rectangle.isFat();
    Object obj;
    obj = rectangle;
    break MISSING_BLOCK_LABEL_38;
    Circle circle =
        new Circle(word0);
    circle.isFat();
    obj = circle;
    JVM INSTR ifne 47;
    goto _L1 _L2
_L1:
    break MISSING_BLOCK_LABEL_41;
_L2:
    break MISSING_BLOCK_LABEL_47;
    ((Drawable) (obj)).draw();
}
public static void f(short si)
{
    Object obj;
    Object tobj;
    Object tobj1;

    if( si > 10 )
    {
        Object tobj2;
        tobj = new Rectangle( si, si );
        tobj2 = ((Rectangle) tobj).isFat();
        obj = new Rectangle( si, si );
    }
    else
    {
        tobj = new Circle( si );
        tobj1 = ((Circle) tobj).isFat();
        obj = new Circle( si );
    }
    if( tobj1 == 0 )
        ((Drawable) obj).draw();
}
Output from our decompiler: **Dava**

```java
public static void f(short s0)
{
  boolean $z0;
  Drawable r0;
  Rectangle $r1;
  Circle $r2;

  if (s0 <= 10)
  {
    $r2 = new Circle(s0);
    $z0 = $r2.isFat();
    r0 = $r2;
  }
  else
  {
    $r1 = new Rectangle(s0, s0);
    $z0 = $r1.isFat();
    r0 = $r1;
  }
  if ($z0 == false)
  {
    r0.draw();
    return;
  }
}
```
Advanced Issues for Restructuring

1. Multi-entry point loops

- **Problem:** Loops in the control flow graph may have more than one entry point

- **Two solutions:** both perform a transform on the control flow graph
1. No other decompiler produced correct output, they generally ignore this possibility.

2. We chose to use the artificial entry point solution due to scaling issue.

3. Artificial entry point problem: One entry point is selected as *natural* and the other are treated as the product of *gotos*. Which do we select as natural?
1. For each entry point, do a DFS

2. Select the entry point that yields the minimum number of targets of back-edges
2. Labeled blocks and break statements

A combination of labeled blocks and break statements can act like a limited goto!

```
L1:
{
    if (a) {
        if (b)
            break L1;
    }
    else {
        if (c)
            break L1;
    }
    d;
}
e;
```
Any control flow DAG can be represented in pure Java.

1. Topologically sort the DAG

2. Place labeled blocks around the statements of the DAG

3. Represent all control flow with break statements

```java
a b c d e ...
{a} b c d e ...
{{a} b} c d e ...
{{{{a} b} c} d} e ...
```
3. Exceptions

Problems:

- Areas of protection may overlap, but not nest

- An area of protection may have several entry points

- Several areas might share the same handler statement

- Their handlers may reside in the area of protection itself!

- Any combination of the above all at once.
public void m()
{
    mException r0;
    java.lang.RuntimeException r1;
    java.lang.Throwable r2;

    r0 := @this;

    label_a:
        java.lang.System.out.println("a");
        goto label_c;

    label_b:
        r1 := @caughtexception;
        java.lang.System.out.println("b");

    label_c:
        java.lang.System.out.println("c");
        goto label_e;

    label_d:
        r2 := @caughtexception;
        java.lang.System.out.println("d");

    label_e:
        java.lang.System.out.println("e");

    label_f:
        java.lang.System.out.println("f");

    catch java.lang.RuntimeException from label_a to label_d with label_b;
    catch java.lang.Throwable from label_b to label_f with label_d;
}
Control flow graph

normal control flow -> exceptional control flow
Solution: Version the control flow graph
4. Thread Synchronization

Problems:

- Object lock releases may be unstructured
- Critical sections may intersect but not nest
- Multiple entry points, etc.

Solution:

- Restructure only nice candidates
- Use a fallback mechanism for all other cases

**Fallback mechanism:** Replace monitor instructions with static method calls to a class that implements monitors in pure Java.
Example of fallback mechanism

...  ...  ...  ...
monitorenter a;  synchronized (a) {
...  ...  ...  ...
monitorenter b;  Monitor.v().enter( b);
...  ...  ...  ...
monitorexit a;  }
...  ...  ...
monitorexit b;  Monitor.v().exit( b);
...  ...
Putting it All Together

Problem: Since it is difficult to resolve these issues singly, solving them simultaneously would likely be extremely difficult, maybe impossible

Solution: Deal with issues one at a time

Our decompiler uses an ordering of phases that allows us to tackle each problem on its own.

For example, all Java loops are found in a single phase. The benefit is that once we have completed this phase, we know we have solved all the potential restructuring problems caused by multi-entry point loops.

See Miecznikowski et.al. from WCRE2001:

*Decompiling Java using Staged Encapsulation*
The ordering of phases in **Dava**:

1. Find simple statements
2. Perform local typing
3. Create a control flow graph of typed simple statements
4. Modify control flow graph to accommodate exceptional problems
5. Find loops
6. Find if and switch statements
7. Find exceptions
8. Find synchronized statements and their fallbacks
9. Determine if we need labeled blocks and break statements
10. Emit Java source
Conclusions

• The Java bytecode specification is much more flexible than the Java language specification

• There are plenty more problems that I haven’t shown (throws declarations, spurious try block removal, class literals, package and class resolution, etc.)

• Even bytecode that comes from javac can pose difficulties

• Many sources can produce bytecode which doesn’t follow javac’s code production patterns

• All these problems have been solved in our decompiler!
If you’d like to try it out

- Our website:
  http://www.sable.mcgill.ca

- My public directory:

Thank you!