Sable
Research Group
http://www.sable.mcgill.ca

Decompile Java Bytecode with Soot!

McGill
University

Motivation

Software Engineering: results of compiler analyses can be included as comments in regenerated source.

Translation: "to Java" translator if original source was not written in Java.

Development: Libraries may be browsed for bug and performance issues.

Goals

To be able to decompile any verifiable Java class.

To output a pure Java representation.

To produce highly readable output.

Using Soot

Soot already had a typeless, 3-address representation called Jimple, and a high level representation called Grimp that provides aggregated expressions

The remaining major tasks for building a decompiler were static type inference and control flow restructuring.

Type Inference

Stage 1

Constraint System

Constraints are extracted from program statements:
- hard nodes represent types
-soft nodes represent type variables
- directed edges represent assignability

Connected Components

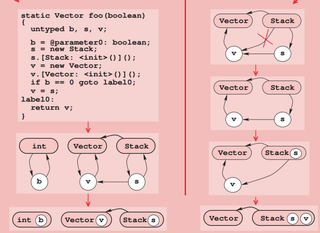
All connected components (cycles) are found:
- all nodes of a cycle are merged
- if a cycle contains more than one hard node, stage 1 fails

Transitive Constraints

All transitive constraints between a soft node and another node are eliminated.

Single Constraints

All soft nodes with a single constraint are merged:
- if any unmerged soft node remains, stage 1 fails



Stage 2

The "new" Problem

Sometimes, stage 1 fails because of a semantic difference between the Java "new" operator, and the bytecode "new" operator

Java
```
Object v = new Hashtable(10);
```
Jimple
```
v = new Hashtable;
v.[Hashtable: <init>(int)](10);
```

In Jimple, v cannot have type Object because of <init> call constraints.

To solve this problem, when stage 1 fails, new instances are assigned to temporary variables as follows, then stage 1 is repeated.

```
v = new Hashtable;
v.[Hashtable: <init>(int)](10);
```
```
temp = new Hashtable;
v = temp;
temp.[Hashtable: <init>(int)](10);
```

Integer Types

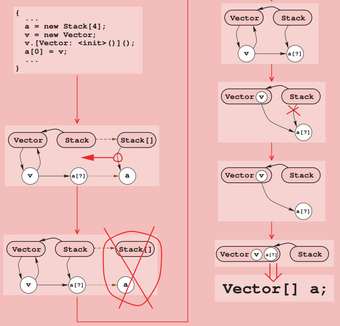
Precise inference of integer types (boolean, byte, short, char and int) is performed at a later step. Steps 1 and 3 are reapplied to variables of type int, using the following type hierarchies, respectively.

Array Constraints

Array operations introduce another kind of constraint. They impose constraints between the element type of a variable and another type (or type variable).

Nodes representing element types and "element of" constraints are collected along normal constraints. Then, constraints between array types are replaced by constraints between their element types.

The resulting simpler problem is then solved, and finally, results are propagated back up "element of" constraints.



Stage 3

In theory, there exist programs for which stages 1 and 2 will not deliver a solution. Such programs do not happen in practice, but for theoretical completeness, a third stage is provided that handles these unlikesly cases
This stage collects "definition constraints", solves the resulting type constraint system, and introduces type casts whenever a "use constraint" is violated.

Restructurer

Problems

Java bytecode control flow is unstructured.

The control flow graph may not represent a structured program.

Java language Thread synchronization is structured while the bytecode monitors are only weakly structured.

Target

A highly compact structuring is likely to be the most readable one.

Java provides multi-level breaks to arbitrary encapsulating blocks. This allows any control flow DAG to be written in pure Java.

Overall Strategy

We build an Abstract Syntax Tree from the control flow graph and emit Java source code from it. The tree is built in 5 phases:

1) Convert monitor statements to "synchronized" blocks
2) Infer Java loops from graph cycles
3) Structure conditions not controlling Java loops
4) Build "try - catch - finally" blocks
5) Determine which edges represent "breaks" and "continues"

Conclusions

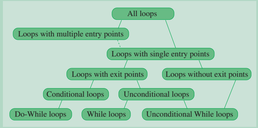
A correct compact representation is alway possible.

When program semantics cannot be preserved, goto elimination with new control flow variables is effective

A best restructuring is performed by minimizing the number of control flow constructs used.

Loops

- Every cycle in the control flow graph is a subset of a Java loop.
- Java loops are characterized by their entry and exit points.
- All multi-entry point cycles must be transformed to single entry point cycles before loops can be inferred from them.



A taxonomy of Java language loops.

Single entry point cycles

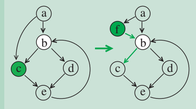
Every single entry point strongly connected component can be represented by a Java loop construct. The type of the loop is determined by the set of conditionals in the component that could control the loop. Loop nestings are found by re-evaluating the component while ignoring the chosen conditional.

an unconditional cycle

Multi-entry point cycles

All control flow constructs in Java have only one entry point. Multi-entry point strongly connected components are converted to single by selecting one of the entry points as natural, and treating the rest as the targets of "goto"s.

The goto is then simulated by redirecting the goto edge to a new "control flow node" that sets a flag stating the destination of the goto. Control is then passed to the natural entry point which is modified to redirect execution along existing control flow paths to the goto target.



entry point removal

Abstract Syntax Tree Construction

As Java control flow contructs are found, the contents of their bodies are identified. The structure of the Abstract Syntax Tree is built as a tree of body content sets, where the children of any node are those constructs whose bodies are subsets of the current node s body.

The order of execution of phases is necessary to this algorithm, however within each phase, order of identification does not affect performance.

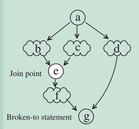
DAGs

Once all conditions that govern Java loops have been identified, all remaining conditions must be represented by "if" and "switch" constructs.

DAG bodies are found looking for all nodes that are reachable by the condition node and not reachable by the join point node, without violating any loop bodies already established.

"if" bodies are found by looking for the join point of the two successors to the conditional statement.

"switch" bodies involve a more exacting set of heuristics to determine the join point.



finding a switch statement

Exception Handling

"Try-catch-finally" blocks are defined as sequences of instructions that are under exception protection and nest well within already established structured control flow

Code under an exception handler may be unstructured. Therefore "try-catch-finally" blocks are identified only after all other structured contructs have been found.

Synchronization

Although they perform no control flow function,synchronized blocks demand that they nest properly within structured control flow. Java bytecode monitor enter and exit instructions should balance within a method but are otherwise unstructured.

Soot allows the user to choose how aggressively synchronized blocks are reconstructed. Choices range from those blocks that could have been generated by a Java compiler, all the way to trying to accomodate instructions with limited variablity in potential monitor level.

In some cases it is impossible to represent the unstructured use of monitors with sychronized blocks. In this case we replace the monitor instructions with calls to our own monitor library written in pure Java.

Breaks and Continues

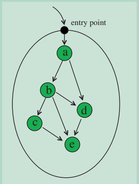
A body in the AST may have a number of children. Connectivity between these child bodies is expressed as a digraph. Since all cycles have been expressed as loop bodies, this new graph is a DAG.

To find out which edges in graph represent natural control and which are breaks and continues, we topologically sort the graph, and check whether the edges span neighbors in the sorting. If they do, the source is recursively checked to see if the edge is natural.

In the given example there are two possible topological sortings:

```
{{a b c} d} e
{{a b} d} c} e
```

The sorting that minimizes the introduction of labeled blocks (denoted by the brace-brackets) is the one that is chosen.



Child digraph

Soot

Soot is a Java bytecode analysis and transformation framework with several powerful intermediate representations

Java bytecode → Baf → Jimple → typed Jimple → Grimp → Java source code

```
1500 6a4c 7661 2f61 6f69 502f 6972 746e
7453 6572 6d61 013b 0a00 6f53 7275 6563
6946 656c 0001 6507 6178 706d 656c 0001
650c 6178 706d 656c 6a2e 7661 0161 0300
6f66 016f 1300 616a 6176 692f 2f6f 7250
```

```
        <unkown> $r0;
label0: load.i i0;
        load.i i3;
        load.i i0;
        add.i;
```

```
        <unknown> $r0;
label0: $i5 = i3 + i0;
        $i6 = i4 + i2;
        $i7 = $i5 * $i6;
        i0 = i0 + $i7;
```

```
        A $r0;
label0: $i5 = i3 + i0;
        $i6 = i4 + i2;
        $i7 = $i5 * $i6;
        i0 = i0 + $i7;
```

```
        A r1;
label0: i0 = i0 + (i3 + i0) * (i4 + i2);
        r1.foo(i0);
        i3 = i3 + 1;
label1: if i3 < i1 goto label0
```

```
A r1;
for (i3 = 0; i3 < i1; i3++) {
    i0 = i0 + (i3 + i0) * (i4 + i2);
    r1.foo(i0);
}
```

Other technical issues for decompiling

Overloaded methods with differing return types (illegal in Java)
Introduction of "import" statements
Syntactic sugaring with "for" loop constructs and ternary statement
Handling compound conditionals

Credits

Authors
Jerome Miecznikowski jerome@sable.mcgill.ca
Etienne Gagnon gagnon@sable.mcgill.ca

Thanks to:
FCAR and NSERC
for their support towards this project
and
The Sable Research Group s
Soot Development Team
Raja Vallee-Rai, Patrick Lam
Patrice Pominville, Laurie Hendren