HOW CAN JAVA CLASS FILES BE DECOMPILED?

David Foster
Chamblee High School
11th Grade

Many computer science courses feature building a compiler as a final project. I have decided to take a spin on the idea and instead develop a *decompiler*. Specifically, I decided to write a *Java* decompiler because: one, I am already familiar with the Java language, two, the language syntax does not suffer from feature-bloat, and three, there do not appear to be many Java decompilers currently on the market.

I have approached the task of writing a decompiler as a matter of translating between two different languages. In our case, the source language is Java Virtual Machine (VM) Bytecode, and the target language is the Java Language. As such, our decompiler will attempt to analyze the *semantics* of the bytecode and convert it into Java source code that has equivalent *semantics*. A nice sideeffect of this approach is that our decompiler will not be dependent upon the *syntax* of the bytecodes, the *compiler* used to generate the bytecodes, or the *source language* (possibly not Java). It will only care whether a class-file's semantics can be represented in the Java language.

Decompilation is divided up into six phases:
1. Read opcodes
2. Interpret opcode behavior
3. Identify Java-language "idioms"
4. Identify patterns of control-flow
5. Generate Java-language statements
6. Format and output to file

The first four phases will be discussed in detail within this paper, as they were the primary focus of my research. Further details pertaining to these phases can be gleaned from the accompanying log book.

PHASE 1: READ OPCODES

First the input Java class file is read, converting each individual opcode within each method to its corresponding Instruction object. Instruction objects are an abstraction used internally by the decompiler to represent Java-opcodes in a more programmatically-friendly manner.

Instruction objects alone are not very good at representing their *semantics*. There are just too many Java-opcodes that perform nearly the same operation. For example, **iadd**, **ladd**, and **fadd** all add two values together, they just operate on different variable types.

To remedy this problem, I have written my own mini-language to address this lack of genericity. The language consists of a small set of basic *commands* and a larger set of *expressions*. Commands perform basic operations such as assignment or branching. Expressions may be used by commands to represent types of manipulations or values to be manipulated.

Through the usage of the mini-language, the decompiler can represent the 200+ Java-opcodes using a handful of commands (about 15) and a rich set of expressions (about 30). By representing the highly specialized opcodes in a more generic fashion, it becomes a lot easier to analyze and transform the code's semantics.

So after the Java-opcodes have been converted to Instruction objects, those Instruction objects are then converted into Commands and Expressions, the decompiler's native language for semantic analysis. During this time, VM exception handlers are also converted into ExceptionHandler objects.

PHASE 2: INTERPRET OPCODE BEHAVIOR

The next item on the agenda is to remove all stack-manipulating Commands and Expressions (**dup**, **pop**, **push**, **swap**). This is necessary because the Operand Stack, which is used by such stack-manipulators, is a VM-only construct; no such construct exists in the Java language.

To deal with this, all stack-manipulations will be transformed into manipulations of special temporary-variables that are inserted by the decompiler. This is done via *operand stack modeling*, a method of simulating the effect that the execution of Commands has on the state of the Operand Stack at every point within the method. During operand-stack-modeling, all **push** commands are converted to assignments to temporary-variables and all **pop** expressions are replaced with the appropriate temporary-variable expression.

The last hurdle to overcome in interpreting the Java-opcodes is *type resolution*; that is, the process of determining the (return-) type of local variables. Such information is not always provided explicitly in the Java class file, so we must be able to determine it ourselves in such cases. Type information about an individual variable can be deduced by looking at what contexts the variable is accessed by Commands or Expressions. For example, if a variable B of type **short** is assigned to a variable A, whose type is unknown, we can ascertain that A's type must be either **short** or **int**, based on the rules for VM type conversion.

PHASE 3: IDENTIFY JAVA-LANGUAGE IDIOMS

Like most other languages, the Java Language has its own set of idioms, each of which serves as shorthand for a common, more drawn out, phrase. Here we address single statements/expressions within the Java Language that translate into multiple Java Bytecodes.

First of all, we have the *conditional expressions*, **&& (and)**, **|| (or)**, and **? (if)**. The first two are boolean expressions which exhibit the behavior of *short-circuit* evaluation under certain circumstances. For example, if the first operand of the **&&** operator is evaluated to be **false**, then the runtime knows that the **&&** expression will evaluate to **false**, so it does not bother evaluating the second operand. Similarly, if the first operand of || is evaluated to be **true**, then the || expression will be automatically evaluated to **true** without evaluating the second operand. Conditional expressions can be identified by searching for certain arrangements of **if** commands. See the log book for more detailed information.

Next, we have the *postfix increment/decrement expressions*. These expressions are highly unusual in that they perform a modification on a variable's value, yet return the <u>old</u> value of the modified variable. These can be identified by looking for **temp = var** followed by **var = var + 1** or **var = var - 1**. These two commands would then be replaced by **temp = var++** or **temp = var--**.

Finally we have *constructor invocations*. As Commands, these statements appear as **this = ThisClass.<init>(...)** or as **this = SuperClass.<init>(...)**, both completely illegal Java Language statements (as the special identifier **this** cannot be reassigned in the Java Language). Commands in those forms are transformed into ConstructorInvokationCommands that look like **this(...)** or **super(...)**.

PHASE 4: IDENTIFY PATTERNS OF CONTROL FLOW

Arguably, the most difficult part of decompilation is the identification of control flow patterns that can be mapped onto Java-Language control flow structures.

Normal control flow encompasses all control flow involving **if** and **switch** statements, the two basic types of conditional branches that exist in the Java Language.

Exceptional control flow involves Java-Language **try** statements. VM exception handlers can be analyzed to infer the existence of **catch** blocks. **Try** statements are also associated with one of the most annoying structures to decompile in the Java Language, the **finally** block.

Synchronized control flow involves Java-Language **synchronized** statements. **Synchronized** statements, like the **finally** blocks of **try** statements, are incredibly difficult to detect.

The current scheme for performing control flow analysis (still in progress) revolves around the strategy of sectioning the set of Commands in a method into different regions, each region corresponding to a Java-Language *statement-block*. The region of the method's root statement-block contains all every command. Nested statement-blocks within the root statement-block are associated with regions of commands that are mutually exclusive subsets of the root statement-block's region. Even more nested statement-blocks have regions that are subsets of their respective parent statement-block's region.

Currently, the only methods that exist for finding regions involve *normal control flow* only; they do not encompass the other types of control flow as of this writing. These methods, being complex and incomplete, are beyond the scope of this paper. See the log book for further information.

PHASE 5: GENERATE JAVA-LANGUAGE STATEMENTS

After performing control flow analysis on all of the commands, generation of Java-language statements can begin. To store the programmatic representation of Java-Language statements and statement-blocks, a new set of classes will be created, known collectively as the *Statement architecture* that provides this functionality.

The statement-block regions determined via control flow analysis will be used to generate Java-statements using the commands within the region, starting at the region's solitary entry-point.

All commands within the statement-block regions will be simple enough to perform a nearly one-to-one translation from the contained Command objects to their corresponding Statement objects. These Statement objects will be linearly arranged within a new StatementBlock, and the block will be nested within its appropriate nesting Statement.

PHASE 6: FORMAT AND OUTPUT TO FILE

A few finishing touches still need to be applied before the Statements are ready to be output into a Java source file.

First and foremost, any variables whose names are not specified by the class-file need to have new names generated for them by the decompiler. There will be a standard convention for generating such names (not defined yet). Variables being used in certain contexts might use different naming conventions than the norm (ex: loop counters).

Next, a final syntax check will be performed by the decompiler to detect any illegal constructs that may have been inserted in error.

If feasible, it would be prudent to have a semantic check that would compare the Command-based and Statement-based representations of the method code to verify that they are consistent. However, this is potentially an expensive operation to perform, in terms of execution time and reliability. This check is not currently implemented.

At long last, the contents of the Statement-architecture is written out to disk.

CONCLUSION

Even though not all of the details of the final phases have been fleshed out, all of the important phases are covered. Overall, I am quite satisfied with the decompiler architecture and design that has resulted from my research.

BIBLIOGRAPHY

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Co., 1986.

Gosling, James; Joy, Bill; Steele, Guy; and Bracha, Gilad. The Java Language Specification (2nd Edition). Addison-Wesley Pub Co., 2000.

Lindholm, Tim and Yellin, Frank. The Java Virtual Machine Specification (2nd Edition). Addison-Wesley Pub Co., 1999.

Nolan, Godfrey. Decompiling Java. APress, 1994.