Assignment 1
Due by 11:55pm on Friday June 14

Overview

The JVM implementation lacks a verification step. What *should* happen is that immediately after a class is loaded, the JVM scans over and analyzes the bytecode of each method to verify that the bytecode can never violate certain operational requirements. The analysis is performed once only, and that will usually be much less expensive than putting an equivalent set of run-time checks into the interpreter (or having the JIT compiler generate code for these checks).

Your task is to implement a simple verifier – one that implements a reasonable subset of the full verification process.

The Checks to be Performed

The basic verification algorithm described on the course slides should support these checks.

- No matter what execution path is followed to reach a point P in the bytecode, the height of the stack will be the same at P for all these paths.
- The height of the stack will never exceed the number determined by the Java compiler. Note: see the `max_stack` field in the `method_info` struct for a method.
- The stack will never underflow.
- When the value of a local variable is used at a point P in the byecode, that local variable should have been assigned a value on all paths which reach P.
- When a value is stored into a field F of a class, the value must be compatible with the type of F.
- When an opcode OP at point P in the bytecode is executed, any operands for OP on the stack must have types which are compatible with OP.

Simplifications (or What Can Be Left Unimplemented)

A full verifier performs many more checks than the above. To keep the problem under control, our verifier will focus mostly on analyzing the states of the stack and local variables.

These checks do not need to be performed for this assignment:

- No instruction accesses a local variable whose number is out of range.
- No instruction uses an out-of-range index into the constant pool.
- Verifying that the constructor gets called for a new class instance.
- Verifying that control flow cannot be transferred into the middle of an operation or past the beginning or end of the bytecode for the current method.
- Verifying that the last operation in a method is complete (i.e. not missing some bytes).
- Handling the `jsr` or `ret` instructions (i.e. for analysis purposes, we just treat `jsr` as a no-op and `ret` as an exit from the program).
- Handling any instructions which are unimplemented in our version of the JVM.
- Handling interface types (which are unsupported in our JVM anyway).
- Analyzing control flow transfers for exception handlings.
- Distinguishing between the high and low halves of double and long values.

Materials Provided

The folder named `Resources/MyJVM` on the course website holds C source code for a rather incomplete JVM. However it works as long as the Java program does not attempt to use any of the unimplemented constructs.

The two files `Verifier.c` and `VerifierUtils.c` contain code that begins the implementation of a verification step. The `Verify` function (in `Verifier.c`) is called whenever a new class is loaded. It in turn calls the local function `verifyMethod` for each method in the class. That function is mostly empty. Your assignment is to complete that function according to the requirements listed above.

The file `OpcodeSignatures.c` defines a large table which has one entry per opcode. The entry contains information about the opcodes signature – what operands with what types need to be on top of the stack to be consumed by the instruction, and what result(s) with what types get pushed by the instruction. A large part of this assignment involves decoding these signatures and applying them to the simulated stack. The great majority of the opcodes can be handled automatically like this.

Hints for Implementation

Representing the State of the Computation

A single array plus an integer for the stack depth can be used to show the state of both the local variables and the stack. Suppose, for example, that we are analyzing a method which takes three arguments (none of which is `long` or `double`), has a `max_locals` value of 8, and has a `max_stack` value of 3. In that case, an array with 11 elements would suffice. That array combines the local variables (which include the arguments) and stack entries into a single entity. At the start of the analysis for the method, the first 3 elements would be initialized to represent values with the datatypes of the 3 arguments and the next 5 elements would be initialized to show uninitialized values. The other 3 elements of the array are initialized "-" meaning *empty* (which will help help debugging). The stack depth value would be 8, which is its minimum value for this method; the maximum value would be 11.

Code has been provided which creates such an array. It is an array of strings. The first letter of the string indicates what kind of data (if any) has held in the variable or stack slot corresponding to the array element. Consider the `Area` method of class `Ass1Ex` shown in Figure 1. When the first opcode in the bytecode of this method is about to be analyzed, the array of strings would have the contents shown in Table 1. The stack bottom is 8 (i.e. the index of the first stack slot in the array).

The coding scheme by which a string represents a datatype is shown in Table 2. When the first code letter is 'A' then it is a reference type, where storage for instances are held on the JVM heap. The remainder of the string following the initial 'A' describes the datatype further. There are two possibilities:

1. The letter 'L' followed by the fully qualified name of a class.
2. The character '[' which indicates that the type is an array; the remainder of the string following the '[' character specifies the element type. If the elements have a simple type, then the possibilities are one of the single letters 'I' 'L' 'F' or 'D' (for `int`, `long`, `float` or `double` respectively). If the elements have a reference type, then element type is shown as a letter 'A' followed by one of the two numbered possibilities. (This is a recursive specification.)

For example, the string `"A[A[Ljava.lang.String"` describes the type `String[][]`.

Note that there is one array element for every word of memory used by the local variables and the stack elements in the JVM. A 64-bit value, which occurs for the `long` and `double` types, occupies two slots in the array. A `long` value is represented as `"L"` in one slot, and `"l"` (a lower-case *L*) in the other; similarly a `double` value is represented as `"D"` and `"d"` in two adjacent slots..

Figure 1: Example Method

```
public class Ass1Ex {
    float Area( Square[] sqs, float scaleFactor ) {
        float scaledArea = 0.0F;
        try {
            for( int i = 0;  i < sqs.length;  i++ ) {
                Square sq = sqs[i];
                float area = sq.size * sq.size;
                scaledArea += area;
            }
        } finally {
            System.out.println("We are done");
            return scaledArea * scaleFactor;
        }
    }
    ... // remainder of class omitted
}
```

Table 1: State Represented as an Array of Strings

Index	Meaning	String
0	V0	`"ALAss1Ex"`
1	V1	`"A[ALSquare"`
2	V2	`"F"`
3	V3	`"U"`
4	V4	`"U"`
5	V5	`"U"`
6	V6	`"U"`
7	V7	`"U"`
8	S0	`"-"`
9	S1	`"-"`
10	S2	`"-"`

Table 2: Coding Scheme for Datatypes

`"I"`	Integral number (includes char, byte and short values)
`"L"`	One half of a long value
`"l"`	Other half of a long value (that's a lower-case L)
`"D"`	One half of a double value
`"d"`	Other half of a double value
`"F"`	Float value
`"N"`	a `null` reference value
`"A<`*type*`>"`	Reference to any object or any array instance
`"U"`	Uninitialized value
`"X"`	Incompatible mixed types
`"-"`	An empty stack slot (if this is accessed, there is a bug in your verification code)

Obtaining Information about Method Arguments and Result

To initialize the array which represents the state of the computation, we need to know the datatypes of the method's arguments. Some code to get you started has been provided. It initializes an array (with the format proposed in the previous section) to represent the initial state of the computation.

When your verification algorithm reaches a bytecode instruction which calls a method, the datatypes of the arguments and of the result (if any) need to found. Your algorithm has to pop values off the stack for the arguments, checking that their types are compatible, and push the code(s) for a function result back on the stack. Some code to retrieve the datatype information has been provided for you.

Obtaining Information about Fields of Classes

When your verification algorithm reaches a bytecode instruction which accesses a field in a class (either a class field or an instance field), the datatype of that field needs to be known. Some code to provide that datatype information has also been provided.

Merging States

Two arrays representing the states can be merged element by element. The rules for merging two type codes to produce the type code of the combined result are shown in Table 3. Note that the merging operation is commutative, so that a table entry such as t&X is meant to include the X&t combination too. The rules should be applied in order from top to bottom using the first pattern which matches.

The *lub* operator (*least upper bound*) is as described on the lecture slides. An untested function named `LUB` and provided in the `VerifierUtils.c file` determines the *lub* of any two reference types.

Table 3: Confluence Rules for Merging Two Type Codes

t & t	t	in all these rules, *t* represents any type whatsoever
t & X	X	
t & U	U	
s1 & s2	X	s1, s2 are two different simple types
a & s	X	s is any simple type; a is any reference type
a & N	a	a is any reference type (this confluence is correct because the JVM performs run-time checks for null pointers)
a1 & a2	a3	a1, a2 are any reference types; a3 is the *lub* of a1 and a2

Some Error Conditions

If a bytecode operation *uses* a value whose type code is U or X, that is always an error. The former case is reported as a possible access to an unitialized variable. The latter is reported as a verification failure (a possible use of an incompatible operand by the current operation).

Any attempt by an operation to dereference a null value (type code N) is an error. (Dereferencing occurs when there is an access to some value in the object referenced by the value; e.g. invoking an instance method when the 'this' pointer is null, obtaining the length of an array when the array reference is null, etc.) This is reported as a definitely invalid use of a null pointer.

The Verification Algorithm

The course slides sketch an algorithm which can be programmed in any manner you wish, as long as it works. However, you may like to follow the algorithm whose pseudocode is shown in Figure 2. The benefit of this algorithm is that it does not require a pre-pass over the bytecode to discover all the jump instructions and where they jump to. The algorithm maintains a dictionary D of entries of the form:

```
< bytecode position, change bit, stack height, typecode list >
```

where the stack height and typecode list are the representation of the computational state described earlier, and the bytecode position is an index into the bytecode array. The bytecode position is used as a key for this dictionary.

Testing

The Java compiler should only ever generate bytecode which passes the verification tests. If you want to discover whther your verifier actually detects errors, you will need some Java class files where the bytecode has been manually generated. The Jasmin Java assembler is recommended for producing such class files.

It can be found at this URL: `http://sourceforge.net/projects/jasmin/files/`

Figure 2: Verification Algorithm

```
Initialize D to <0, 1, intial stack height, initial typecode list>;
while D contains an entry with a set change bit do
   set change bit to 0 in this entry;
   p = bytecode position in this entry;
   h = stack height in this entry;
   t = typecode list in this entry;
   op = opcode at position p in byecode;
   if op comtains any immediate operands then
      check that any operands which are indexes are in range;
   foreach stack operand accessed by op do
      decrement h and check that stack does not underflow;
      pop stack t and check datatype compatibility of that operand;
   foreach stack result generated by op do
      increment h and check that stack does not overflow;
      push typecode of result onto the stack t;
   for q = the bytecode position of each instruction that
           can execute immediately after op at position p do
      if an entry with position q exists in D then
         merge h and t with the entry in D, updating that entry;
         if any incompatibilities were found then
            report an error;
         if anything changed in that entry then
            set its change bit;
      else // no q entry exists
            add a new entry <q, 1, h, t> to D
```

Rules for performing the work and for submission

- You may form teams of two people for performing the work. Both names must appear in a comment at the beginning of every source code file which you create or modify.
- Your version of the Java interpreter must work correctly on at least a 64-bit Linux platform.
- All files needed to build the Java interpreter with the verifier incorporated (including an updated Makefile) should be combined into a single zipfile or a singled gzipped tarfile and uploaded to the course's conneX account.
- Feel free to add new files to the project and to edit existing ones to fix bugs and/or re-arrange the code to suit your needs.
- Include a `README.txt` file which identifies the team members and describes any deviations from the assignment requirements (and also any unusual problems encountered).
- Experience says that there will be some bugs in the code you have been given. A file named `Corrections.txt` will appear in the Assignment1 folder where there are bugs and bug fixes to report, and the file will be updated as needed.