Stable Interpretation of Java

Stephan Diehl[1] and Claudia Bieg[1]

FR 6.2 Informatik
Saarland University
66041 Saarbrücken, Germany
diehl@acm.org, cbieg@cs.uni-sb.de

Abstract. In this paper we present the unconventional architecture of JOSH, an interpreter for Java code fragments. It consists of a parser that accepts Java code fragments, a code generator that embeds the fragments into Java classes and exploits inheritance and serialization to preserve state. The classes are then compiled and executed by external processes. To propagate state from the execution of one code fragment to the next, state is serialized and written into and read from temporary files. The major advantage of this approach compared to other Java interpreters is stability, i.e. its ability to recover from errors. We conclude the paper by comparing our approach to previous ones.

1 Introduction

JOSH is a Java interpreter [5]. It was originally designed to ease teaching Java to beginners. We believe that they should learn basic programming concepts, before they are exposed to classes and objects. With JOSH beginners can interactively evaluate simple expressions, execute program statements and define variables (fields) and functions (methods). Not until they are familiar with these concepts they can define classes.

Typically one of the first programs beginners write in Java looks like

```
public class Hello {
  public static void main(String[] args)
    { System.out.println("Hello world");
    }
}
```

To understand the meaning of all the constructs involved the student would have to know about classes, static methods, packages, Strings, arrays, etc. In JOSH the student would simply have to type

```
> println("Hello world");
Hello world
```

The direct feedback provided by the read-eval-print loop[10] as we know it from languages like LISP, ML and even BASIC has not only proven useful

for novices to learn programming, but also for experts to debug more complex programs or APIs. In particular for debugging stability of the interpreter is a very desirable feature. If the evaluation of a statement leads to a runtime error, the interpreter should be able to recover from this error by returning to the previous state. The architecture of JOSH very much differs from previous Java interpreters and by virtue of the way JOSH preserves state it is the first Java interpreter that provides the desired stability. Runtime errors during execution of a statement or expression do not interfere with the rest of the system which just falls back to the previous state.

2 An Example

Consider that you enter subsequently the following fragments into a Java interpreter.

```
> class A { int v; }
Class declared
> int f(A a) { a.v++; return f(a); }
Method added
> A a=new A();
Field Added
> f(a)
```

The evaluation of the expression `f(a)` will lead to a stack-overflow error. Unfortunately, in existing Java interpreters all side effects that occur during the evaluation of this expression will be effective, e.g. the value of `a.v` will be the number of recursive invocations of `f`, instead of its original value `0`. The problem is that in most error cases, it is not as obvious what side effects have occurred and to continue working with the interpreter in such a situation can lead to unpredictable results. To continue debugging the programmer might be forced to redo the whole sequence of inputs up to the point where the error occurred. In JOSH the state is automatically set to that before the execution of the expression, which caused the runtime error.

3 The architecture of JOSH

In the following we use the term *single code fragment* to denote single expressions and statements, or declarations of methods, variables and classes. The term *code fragment* is a sequence of one or more simple code fragments.

Traditional interpreters like DynamicJava [6] parse the code fragments, build an abstract syntax tree or some other internal representation and then traverse this representation and perform semantics actions. JOSH doesn't know anything about the semantics of Java, but relies on the Java compiler and the Java Virtual Machine (JVM) [7, 4]. Both JOSH and DynamicJava use parsers generated with

JavaCC. But the rest of JOSH's implementation takes only 1000 lines of code whereas DynamicJava (without its GUI) takes about 28000 lines of code[1].

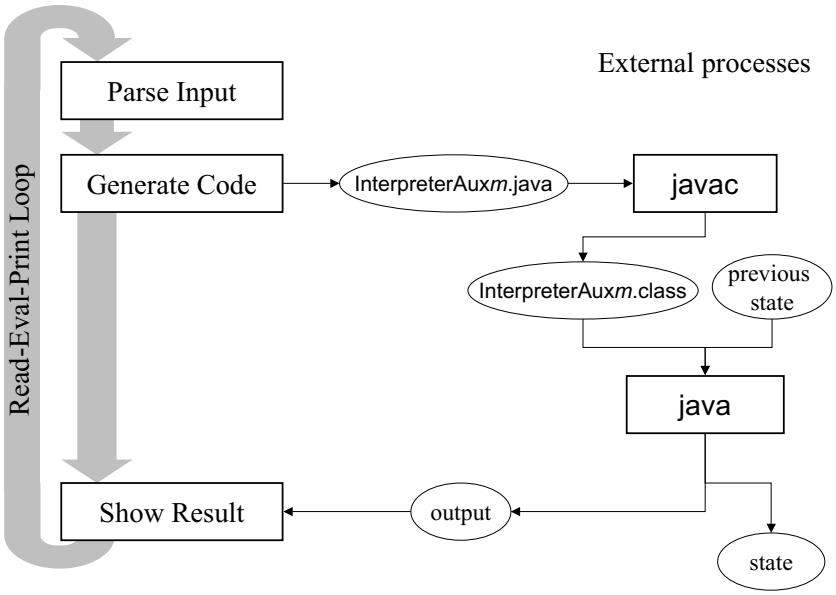


Fig. 1. The read-eval-print loop of JOSH

The key to JOSH's stability is that each code fragment is executed by a JVM in a separate process. More precisely, we generate for each code fragment a new Java class `InterpreterAux`n which is actually an application with its own `main()` method and execute it in a command shell by `java InterpreterAux`n. But how does the application get the final state of the execution of the previous code fragment as its initial state? By state we mean the set of instantiated objects and their attribute values at runtime. In addition the next application has to know all the methods and classes of the previous one.

JOSH inserts code fragments into a class skeleton to produce a running Java application. Consider the declaration of the variable `a` of our example. For this JOSH generates the following Java source code:

```
package tempclasses;

public class InterpreterAux0
       extends jinterpreter.InterpreterBase
{
```

[1] Also the generated parsers considerably differ in size (5900 vs 17100 lines of code), as DynamicJava's parser additionally has to build an abstract syntax tree.

```
  public static void main(String [] argv) {
     InterpreterAux0 self=new InterpreterAux0();
     self.startInterpreterAux(); }

  public void startInterpreterAux() {
     println("Field added");
     dump("/tmp/state.dump"); }

    A a=new A();
}
```

The source code is compiled by starting the Java compiler as an external process. If the compilation is successful, then the generated byte code is executed by the JVM as an external process. To preserve state the application generated for one code fragments inherits from that of the previous one. In addition at the end of the execution of an application the currently instantiated objects are serialized and stored in a file `state.dump`. The generated class inherits methods for serialization from `InterpreterBase`.

```
public class InterpreterBase
       implements Serializable
{ ...

  public void dump(String filename)
   { try { FileOutputStream fs =
             new FileOutputStream(filename);
           ObjectOutputStream os =
             new ObjectOutputStream(fs);
           os.writeObject(this);
         } catch (Exception e) {}
    }

  public static Object undump(String filename)
   { try { FileInputStream fs =
             new FileInputStream(filename);
           ObjectInputStream os =
             new ObjectInputStream(fs);
           return  os.readObject();
         } catch (Exception e)
            { System.out.println(e);
              return new Object(); }
   }
}
```

In our example the next code fragment was the expression `f(a)`. JOSH generates the following source code:

```
package tempclasses;

public class InterpreterAux1
       extends InterpreterAux0 {
  public static void main(String [] argv) {
     InterpreterAux1 self=new InterpreterAux1();
     InterpreterAux0 previousState=
       (InterpreterAux0) undump("/tmp/state.dump");
     self.a=previousState.a;
     self.startInterpreterAux(); }

  public void startInterpreterAux() {
     javaInterpreterEvaluate( f(a) );
     dump("/tmp/state.dump"); }
}
```

Now the class `InterpreterAux`m can access a serialized object of its superclass by calling the method `undump()` and setting the values of those variables that are not redefined in `InterpreterAux`m to the stored values. If the execution of the successfully compiled code fragment leads to a runtime error as the stack overflow in the above example, the state can be reset to that of the previous state. Thus serialization enables us to go backwards in the state history. The problem with serialization is that some member variables of objects cannot be serialized, i.e. those annotated with the attribute `transient` or objects of non-serializable classes like `java.io.FileDescriptor`.

The parser of JOSH does not build an internal data structure like an abstract syntax tree. The parser just detects what kind of fragment was entered. For declarations JOSH then looks in the input for the class, method or variable names declared. In particular for all variable names that have been declared, JOSH generates code to copy their values from the previous state. In the example above this is done for the variable `a` by the assignment

```
self.a=previousState.a;
```

Like interpreters for ML and other functional languages JOSH does not only return the value of an expression, but also its type:

```
> (3+'c')*'a'
9894 : int
```

This is achieved in the class `InterpreterBase` by simply overloading the method `javaInterpreterEvaluate()`:

```
public static void javaInterpreterEvaluate(byte b)
  { System.out.println(b+" : byte"); }

public static void javaInterpreterEvaluate(int i)
  { System.out.println(i+" : int"); }
...
```

4 Related Work

Recently several interpreters for Java have been developed including MiniJava [9], DynamicJava [6] and BeanShell [8]. While MiniJava is not publically available and there are no details available on its implementation, both DynamicJava and BeanShell are written in Java. Both are open source. They build an abstract syntax tree and then traverse the nodes of these trees and execute their semantics functions. JIN [11] is a commercial Java interpreter also written in Java that presumably works in a similar way. While Dynamic Java supports class and method declarations, BeanShell only supports method declarations and JIN supports neither of both.

DrJava [1, 2] is a Java programming environment that uses DynamicJava within its interaction window. For classes defined in other windows it calls the Java compiler.

To integrate JOSH into a web-based tutorial on Java programming, we developed JOSH-online, a server-based interpreter[3] which has been integrated into an online tutorial for learning Java programming from basic principles. JOSH-online applet sends the code fragments to the server for compilation and dynamically loads the generated classes to execute the fragments. In JOSH-online the execution is thus not done in a separate process and thus runtime errors can interfere with the rest of the interpreter.

5 Comparison of 3 Architectures

In Table 1 we compare three different architectures:

DynamicJava: classical interpreter architecture which builds an abstract syntax tree for each fragment and performs semantics actions while traversing the tree.

JOSH: Java classes are generated for each fragment, externally compiled and executed.

JOSH-online: Java classes are generated for each fragment, externally (on the server) compiled, dynamically loaded and executed.

Although the implementation of DynamicJava that is used in DrJava seems to be the most elaborate classical Java interpreter available, we found that it doesn't implement resolution of overloaded methods correctly, in particular it did not detect ambiguous method invocations.

When it comes to performance the generation-based approaches suffer from an initial generation overhead. On a Pentium III with 700MHz generation takes about 1 second. For method invocations we found that the interpretation overhead of DynamicJava compared to JOSH was about a factor 600 and for memory access (reading and writing to an array) it was about a factor 45. Some performance results (including those for the BeanShell interpreter) are shown in Figure 2 and Figure 3.

	Dynamic Java	JOSH	JOSH-online
Error Recovery	no	yes	no
Preserving transient state information	yes	no	yes
Semantics	hand coded	relies on compiler and JVM	relies on compiler and JVM
Size of Implementation (including parser)	28 (45) KLOC without GUI	1 (7) KLOC	3 (10) KLOC without GUI
Performance	no generation overhead, interpretation overhead during execution	generation overhead, fast execution of compiled code	generation overhead, fast execution of compiled code

Table 1. Comparison of three interpreters

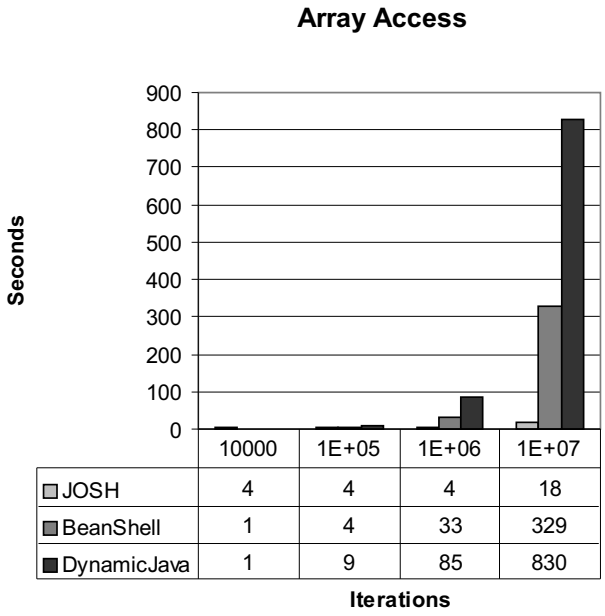


Fig. 2. Performance for accessing arrays

By relying on the compiler and the JVM the source code of JOSH's implementation is only 4 percent in size than that of Dynamic Java. The implementation

of JOSH is simple in the sense that we do not have to implement the semantics of each Java construct. We even could use the Java grammar provided with JavaCC without any modifications. We only had to invoke some internals of the generated parser in addition to the standard way of using it to be able to detect valid prefixes and extract class, method and variable names from declarations. The non-trivial problem to solve was preserving state.

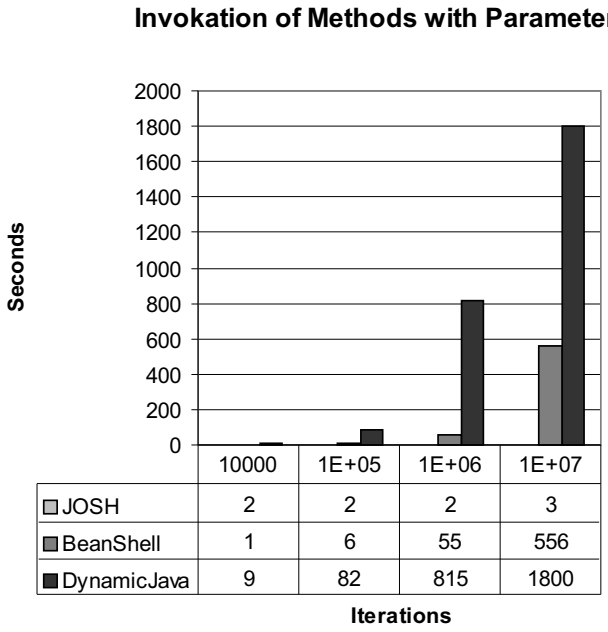


Fig. 3. Performance for method invocations

In the file `josh.properties` the user can actually configure what compiler, APIs and JVM are used by JOSH for external compilation and execution:

```
classpath = c:\JOSH\josh0.02.jar;.
java.command = c:\programs\java\jdk1.4\bin\java
javac.command = c:\programs\java\jdk1.4\bin\javac
```

As JOSH compiles fragments into byte-code and then executes this byte-code on the JVM, it is even possible to use the Java Debugger instead of the standard JVM in combination with JOSH. If we try this with a classical interpreter the whole interpreter would have to be executed by the debugger and we would find ourselves debugging the meta-level and not our actual program.

6 Conclusion

We presented the unconventional architecture of JOSH, a Java interpreter that compiles and executes code fragments externally. We compared JOSH with other architectures. JOSH does not preserve transient values and generation takes some extra time. But it provides better stability, doesn't need hand coding of the Java semantics and doesn't suffer from the interpretation overhead of classical interpreters.

References

1. E. Allen, R. Cartwright, and B. Stoler. DrJava. `http://drjava.sourceforge.net/`.
2. E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *Proceedings of ACM SIGCSE 2002*. ACM SIGCSE Bulletin 34(1), 2002.
3. C. Bieg. A server-based interpreter for an online tutorial on programming in Java (in German). Master's thesis, University of Saarland, Saarbrücken, Germany, 2003.
4. S. Diehl. A Formal Introduction to the Compilation of Java. *Software - Practice and Experience*, 28(3), 1998.
5. S. Diehl and C. Bieg. JOSH Homepage. `http://www.cs.uni-sb.de/~diehl/JOSH`.
6. S. Hillion. DynamicJava. `http://koala.ilog.fr/djava/`.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
8. P. Niemeyer. BeanShell. `http://www.beanshell.org/`.
9. E. Roberts. An overview of MiniJava. In *Proceedings of ACM SIGCSE 2001*. ACM SIGCSE Bulletin 33(1), 2001.
10. E. Sandewall. Programming in an interactive environment: the Lisp experience. *Computing Surveys*, 10(1), 1978.
11. L. Vanhelsuwé. JIN. `http://www.lv2.clara.co.uk/products/Jin.html`.