Implementing Languages Other than Java on the Java Virtual Machine

K John Gough and Diane Corney
j.gough@qut.edu.au and d.corney@qut.edu.au *

Abstract

The portability and runtime safety of programs which are executed on the Java Virtual Machine (*JVM*) makes the *JVM* an attractive target for compilers of languages other than Java. Unfortunately, the *JVM* was designed with language Java in mind, and lacks many of the primitives required for a straighforward implementation of other languages.

Here, we discuss how the *JVM* may be used to implement other object-oriented languages. As a practical example of the possibilities, we report on a comprehensive case study.

The open source *Gardens Point Component Pascal* compiler compiles the entire Component Pascal language, a dialect of Oberon-2, to *JVM* bytecodes. This compiler achieves runtime efficiencies which are comparable to native-code implementations of procedural languages.

1 Introduction

1.1 Java and the Java Virtual Machine

The runaway success of the Java programming language[1] in the last few years is a phenomenon arguably without parallel in the short history of programming languages. One of the interesting side-effects of this widespread popularity is the ubiquity of the execution engine of Java, the Java Virtual Machine (*JVM*)[2]. Essentially all computing platforms have at least one *JVM* implementation available for them, and there are an increasing number of lightweight, small footprint *JVM* implementations targetted at embedded devices.

The widespread adoption of Java as an implementation language for mainstream applications has ensured that the typical Java execution environment is endowed with a rich supply of APIs. Thus solutions to issues such as security, network programming, wide character support and so on are suddenly available in a relatively uniform fashion across the spectrum of platforms.

A final factor favouring the availability of the Java execution environment is the elimination of the major argument against the virtual machine approach, that is, the runtime inefficiency of the virtual machine interpreter. As will be quantatatively demonstrated below, the use of the more recent just-in-time compilation systems all but removes the runtime overhead of the traditional, interpretative approach to virtual machine implementation.

After discounting the effects of fashion, it seems that the popularity of Java is based in two promises: one is the *write-once, run anywhere* claim of universal portability, the other is the runtime type-safety of Java programs. Although it may be observed that both of these claims are subject to some minor quibbles, they hold true to a very large extent.

For many enthusiasts of Java, it may come as a surprise to learn that the two "key advantages" of Java, are not properties of Java the programming language. Rather the key advantages are properties of the *JVM*. A consequence of this observation is the claim that programs written is *any* programming language would share all of the advantages of Java, once they were translated into the machine code used by the *JVM*.

*Queensland University of Technology, Box 2434 Brisbane 4001, Australia

1.2 Languages other than Java

A number of people, lured by the availability of the *JVM* have written compilers which compile subsets of other languages to the Java bytecode form[3]. In most cases these efforts have been restricted to language subsets, since there is no efficient way of encoding the type-unsafe features of most of the other popular programming languages. This is an intrinsic limitation, since the design philosophy of the *JVM* is based on type safety. Indeed, if an implementor was to find a way of bypassing the type safety guarantees of some implementation of the *JVM*, it seems probable that the exploit could form the basis of a security attack on Java programs. Thus the *JVM* vendor would be obliged to remove the security hole in the next revision, invalidating any programs depending on the flaw.

Leaving aside the issue of type-safety, the question remains as to the extent to which languages of different design philosophy can be efficiently implemented on the *JVM*. In order to explore the answer to this question, in 1998 a project was begun to provide a complete, efficient implementation of another type-safe language.

The central difficulty standing in the path of languages other than java (*LOTJ*s) is the fact that the *JVM* was designed precisely with language Java in mind. The execution engine does not provide the primitives that are required for the simple implementation of many *LOTJ*s. Common programming language features which require some inventiveness include —

- Reference parameters
- Uplevel addressing (access to non-local variables)
- Procedure variables (function pointers)
- Structural compatability of types

All of these issues admit to solutions with more or less difficulty, as is demonstrated by the compiler which forms the main subject of this paper. However, there are other, practical issues which need to be considered as well.

The standard Java runtime environment consist of the *JVM*, together with infrastructure that loads classes as needed. An intrinsic part of this mechanism is the *byte-code-verifier*, which checks the binary form of every class before loading it. JavaSoft describe this tool as a "theorem prover" which refuses verification to any class for which it cannot establish the required properties. As is usual, many of the properties which the verifier attempts to evaluate are incomputable. The analysis is therefore necessarily conservative. This places a novel constraint on the compilers of *LOTJ*s since it is insufficient to generate semantically correct code, instead the code must be generated in such a way that the verifier is able to establish that correctness.

Gardens Point Component Pascal (**gpcp**) is a compiler for the whole of the language Component Pascal[4, 5]. All of the issues listed above needed to be resolved, in order to achieve this outcome.

1.3 Why Component Pascal?

Component Pascal is a dialect of Oberon-2[6, 7]. The language was designed by Clemens Szyperski and others for Oberon Microsystems' BlackBox Component Builder framework. Like Oberon-2 it is a small, object oriented language supporting single inheritance based on extensible records. Its ancestors are Oberon, Modula-2 and Pascal.

As a vehicle for the exploration of the issues involved in compiling *LOTJ*s to the *JVM*, Component Pascal seems an ideal choice. The language poses all of the significant issues that were itemized in the introduction. Furthermore, since the language is completely statically type-safe there was reason to believe that this is one of the few languages for which *the complete language* could be efficiently implemented by the *JVM*.

1.4 Overview

It is the objective of this paper to review the main issues of compilation of *LOTJ*s to the *JVM*, and give some performance figures. A more extensive treatment of the detail of some of the required techniques has been given elsewhere[8], and other papers in preparation.

Component Pascal Construct	Java Virtual Machine Construct
module level scalars and pointers	static fields of the class corresponding to the CP module
module level records and arrays	objects referenced by static variables of the module class, and allocated at load time of the class
scalar or pointer local variables of procedures	local variables of corresponding *JVM* method
record and array variables of procedures	objects reference by local variables of the *JVM* method, and allocated during the procedure prolog

Figure 1: Mapping of data types

2 Gardens Point Component Pascal

2.1 The Compiler

Gardens Point Component Pascal (**gpcp**) is a compiler for the language Component Pascal, which targets the *JVM*. It is able to produce either applications or applets, and is able to make use of the Java API to access utilities such as network services and GUI support.

The compiler is written in Java currently, but a future project will rewrite the compiler in its own language. Both the current and the future versions of the compiler will be released as open source products. Thus the community may use the compiler directly, or as an example of the techniques of compiling *LOTJ*s to the *JVM*.

2.2 Data Representation and Module Structure

There are only two kinds of data known to the *JVM*. These are the local variables of methods, and dynamically allocated instances of classes and arrays. Local variables are simple scalars or references to objects, and can only be accessed from within their owning method. There is no concept of data address, nor of address arithmetic. It is possible to store and pass references, but these references can only originate from the allocation of objects of known type. In particular, it is not possible to obtain a reference which points to the interior of an object, or points to a local variable of a method.

It follows that all the program data of a *LOTJ* must be mapped onto the available types of the *JVM*. Component Pascal has the usual scalars, arrays, records and has pointers to dynamically allocated arrays and records. Data of all these types may be *static*, that is, allocated at load time, or *automatic*, that is, allocated on procedure invocation.

The mapping of modules onto classes is performed as follows. Each module corresponds to a single class in the *JVM*. Ordinary, that is, non-virtual procedures of the module become static procedures of the class, and module data becomes static data of the class. Each record type of the module becomes a class in the *JVM*, carrying with it all of the type-bound (virtual) procedures of the module. Details are given in Figure 1.

2.3 Parameter Passing

Parameters are passed in the *JVM* only by value. Since there is no notion of data address, it is not possible to obtain the effect of reference parameters by passing addresses using the language-C idiom.

In this case, *OUT* and *VAR* (inout) parameters are passed by copying in and out. Since the *JVM* allows only a single return value this effect is obtained by "boxing" the outgoing value in a unit length, dynamically allocated array. The caller loads the value of the actual parameter variable into the box, and saves a reference to the box through the call. The called procedure updates the boxed value. After the return, the caller uses the saved reference to access the updated value in the box. It finally copies the new value to the actual parameter location. Figure 2 illustrates this concept.

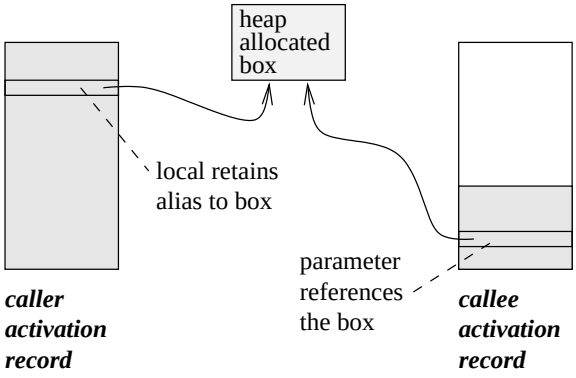

Figure 2: Passing a boxed value to a procedure

2.4 Dataflow Analysis

As pointed out in the introduction, it is necessary for any compiler of a *LOTJ* to ensure that the class files which it produces are able to be verified. In particular, the verifier will insist on being able to prove that every local variable is properly initialized before use. This requires conservative dataflow analysis the details of which are implied by the *Definite Assignment* rules of the Java Language Specification. This is a relatively standard *backward-flow, all-paths* dataflow analysis problem.

There is no advantage in performing a more accurate analysis then the verifier, since the verifier has the final say. However, as it turns out the computational framework which is required to perform the analysis may be used for additional helpful compile-time diagnostics. Figure 3 is an example where an interprocedural extension of the dataflow analysis allows **gpcp** to detect an incorrect program construction.

```
    PROCEDURE Bar();
      VAR abc,xyz : INTEGER;

      PROCEDURE Fee; BEGIN abc := 0 END Fee;
      PROCEDURE Foo; BEGIN INC(xyz) END Foo;

    BEGIN
      Fee; (* this one is ok *)
      Foo; (* but this is bad *)
****  ^ Non-locally accessed variable may be uninitialised
****  ^ <xyz> not assigned before this call
      ...
    END Bar;
```

Figure 3: Incorrect code requiring interprocedural analysis

2.5 Procedure Variables

Perhaps the most difficult aspect of the whole project turned out to be the implementation of procedure variables (function pointers in language-C).

The problem is twofold. First, the *JVM* knows of no such construct. Second, procedure variables in those languages which possess them typically are determined to be compatible according to structural equivalence rules. Since the *JVM* knows only of equivalence according to name, the problem follows.

The first issue is easily solved by representing a procedure value as an object with a single "invoke" method. However, when such a value has to be assigned the compiler cannot guarantee

that the value is of the same named type as the destination value.

A number of complex schemes were prototyped in an attempt to resolve this issue. In the current version, the solution is simple, but sometimes inefficient. Procedure types correspond to *JVM* interface types, while procedure values are declared to be of some unique class type. The trick is that the class corresponding to each value implements every conforming interface known to the compiler. There is thus a high probability that any particular invocation of a procedure value will find the value implements the expected interface type. The exceptional cases are trapped at runtime, and the call made using the reflection mechanisms. The compile-time guarantee of structural compatability guarantees the success of the reflection-mediated invocation.

2.6 Accessing the Java API

The attractiveness of any *LOTJ* is likely to depend critically on the ease with which the Java API is able to be accessed. In particular, for a language such as Component Pascal, it is important that components are able to interwork seamlessly with the Java component framework – Java Beans.

Following previous experience with Gardens Point compiler systems, we defined a *foreign language interface* which allows declaration of Java API access. As it turned out, several problems surfaced with constructs such as interface types, protected methods, and name overloading.

A fundamental issue is that languages such as Component Pascal enforce a strict partial order on compilation order. Java has no such strictures.[1] Rather than write a new compiler especially for interfaces we are now constructing a tool which directly produces binary symbol files from corresponding Java .class files.

In response to the other issues, we have enriched the attribute evaluation of the compiler so that it understands all necessary semantics of Java as well as Component Pascal. As an example, the compiler understands what it means for a Component Pascal type to extend a Java API class. It also permits Component Pascal types which are extensions of Java classes to contract to implement interfaces. Such obligations are fully enforced.

These choices have certainly added some additional complexity to the compiler. However, they are a necessary addition. Consider for example that the *LOTJ* classes cannot participate in the Java 1.1+ event handling model unless there is a mechanism for declaring that they implement the necessary event handling interfaces.

3 Performance

Some figures available from preliminary testing suggest that for procedural (i.e. non object-oriented) code the performance of programs is comparable to native code Modula-2 on the same platform.

Here, we present two rather different synthetic benchmarks, to illustrate the range of possibilities. The first of these benchmarks is a program which discovers all solutions of the *N-queens* problem for all board sizes from 8 to 13. The algorithm is recursive backtracking, so the procedure call and return mechanism is worked extremely hard. Figure 4 shows normalised results for several different platforms.

Platform	Version	Optimised M2	Default M2	CP with JIT	CP interpreted
SPARC/Solaris	JDK 1.2.1	100%	78%	91%	4%
SPARC/Solaris	JDK 1.1.6	100%	84%	72%	6%
Pentium/Win98	JDK 1.2.0	100%	77%	106%	13%
Pentium/Linux	JDK 1.1.7	100%	82%	—	13%

Figure 4: Relative speeds for NQueens program

In this figure the results have been normalised to factor away the relative speeds of the various platforms, although these all fell withing a factor of two in absolute speed. Several factors are

[1] Consider `java.lang.Object` This class has methods which presume that the properties of `java.lang.String` and `java.lang.Class` are known to the compiler of the interface. But both of these types are extensions of `Object` and presume prior compilation of that class' interface.

worthy of mention here. Firstly it may be seen that for the *SPARC* platform significant improvements have been made to the just in time compiler (JIT) between version 1.1.6 and 1.2.1. Even the interpreter is somewhat faster.

Notice also that Component Pascal, with the aid of the JIT, is faster than native code with the default level of optimisation. Only with the highest level of optimisation turned on does the native code run faster.

For the Intel architecture is is clear that the interpreters are more efficient than on SPARC. In the case of verion 1.2 is appears that the JIT produces better code than the GPM compiler with all optimisations turned on. This is a very significant achievment. We did not have access to a JIT for the Linux platform.

The NQueens benchmark is a little unusual, since it involves no object creation at all. A rather different impression is given by the (in)famous *Dhrystone* program. In this case, although this is a purely procedural benchmark, the program requires some object creation for the passing of parameters, as was shown diagramatically in Figure 2. Figure 5 shows the performance numbers for the Dhrystone program, comparable to the previous table for NQueens.

Platform	Version	Optimised M2	Default M2	CP with JIT	CP interpreted
SPARC/Solaris	JDK 1.2.1	100%	82%	25%	4%
SPARC/Solaris	JDK 1.1.6	100%	87%	11%	4%
Pentium/Win98	JDK 1.2.0	100%	82%	11%	3%
Pentium/Linux	JDK 1.1.7	100%	80%	—	3%

Figure 5: Relative speeds for Dhrystone program

In this figure the results have been normalised to factor away the relative speeds of the various platforms. Taken at face value, these figures are somewhat discouraging. However, further investigation showed that a majority of the runtime of the program was spent in garbage collection of the parameter boxes. We have reason to believe that most of this overhead can be optimised away by static analysis in the compiler. If this is done, we see no theoretical reason that the comparative figures for benchmarks such as this should not approach those for the NQueens program.

4 Conclusions

The compiler **gpcp** convincingly demonstrates that it is possible to execute at least suitable *LOTJ*s using the *JVM* as an execution platform. As indicated here, the whole of the language may be successfully translated with reasonable efficiency. The advantages of the *JVM* as an execution mechanism are thus available to a wider range of languages.

It is interesting to consider which other languages might be candidates for complete translation. Certainly languages with a high degree of type safety are candidates, with perhaps Scheme, Sather and Eiffel springing to mind. The situation with other languages is not so clear. There are certainly useful subsets of many other languages which might be successfully translated. However, the chance of capturing large quantities of legacy code by this mechanism seem dubious. The problem is that many unnecessary but prevalent idioms in programming praxis use non type-safe mechanisms. For example, almost all uses of union types are intractable to the *JVM*.

There is another language dimension which needs consideration, that of immediate translation languages. These languages occur, for example, whenever a command language is translated on-the-fly to some intermediate form and then immediately executed. Given the amount of technological advancement in JIT compilation sparked by the Java revolution, the use of *JVM* byte-codes as an intermediate form for such dynamic compilation systems seems without parallel. It may transpire that the most important application of this research into compiling *LOTJ*s is for such dynamic compilation languages.

5 Acknowledgements

The work reported here was partially supported by ARC grant A49700626. Useful discussions with Clemens Szyperski, Nam Tran, Wayne Kelly and Paul Roe are acknowledged. The work on the interface to the Java API depends heavily on the Java Tools project of DSTC, particularly the work of Ken Baker.

References

[1] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading MA, 1997.

[2] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading MA, 1997.

[3] R. Tolksdorf, *Programming Languages for the Java Virtual Machine* (Web bibliography) `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`

[4] Oberon Microsystems, 'Component Pascal Language Report' available at — `http://www.oberon.ch/resources`

[5] Cuno Pfister, 'The Evolution of Oberon-2 to Component Pascal'. Oberon Microsystems (technical report), available at — `http://www.oberon.ch/resources`

[6] N. Wirth, 'The Programming Language Oberon'; *Software Practice and Experience* 18:7, 671–690; 1988.

[7] H Mössenböck and N. Wirth, 'The Programming Language Oberon-2'; *Structured Programming* 12, 179–195.

[8] K John Gough, 'Parameter Passing for the Java Virtual Machine' *Australian Computer Science Conference* ACSC2000, Canberra, February 2000.