Parameter Passing for the Java Virtual Machine

K John Gough
Queensland University of Technology
j.gough@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. Most fundamental of these obstacles in the limited range of parameter passing modes offerred by the JVM.

Here, we discuss possible ways of using the JVM to provide parameter passing modes with alternative semantics, and explore one particular architecture in practice. The interaction between these mechanisms and the Java "byte code verifier" is considered also.

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.

Keywords *Compilers, Java Virtual Machine, languages other than Java, parameter passing mechanisms.*

1 Introduction

1.1 Languages other than Java

The runaway popularity of the Java programming language[1] is one of the more notable phenomena of the last few years. The very wide adoption of language Java as an implementation language for systems, and even as a language of instruction in education, seems to be chiefly based on two claims —

- Programs written in Java are immune from certain classes of runtime errors
- Programs written in Java are portable (the *compile once, run anywhere property*)

Each of these properties is indeed important, but it turns out that neither of them is actually a property of Java the language[1]. Both are actually properties of the execution engine: the *Java Virtual Machine*[4] (*JVM*). Any program written in any programming language capable of being translated into the bytecode format used by the *JVM* will share all of the portability and safety properties of programs written in language Java. Indeed, the execution engine has no way of telling in which source language the program was written.

A number of people, making the same observation, have written compilers which compile subsets of other languages to the Java bytecode form[8]. In most cases these efforts have been restricted to language subsets, since there is no efficient way of encoding the type-unsafe features of typical programming languages.

A further difficulty standing in the path of languages other than java (*LOTJ*s) is due to 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

There are other issues that arise in the case of non-procedural languages, although the *JVM* seems well suited to languages such as the Scheme dialect of LISP. Some of the intrinsic limitations of the *JVM* as a compiler target are discussed in Section 5.

All of the issues raised above admit to solutions with more or less difficulty, as is demonstrated by the compiler

[1] And indeed neither of them are fully achieved with Java. Every writer of a substantial GUI program written in Java is aware that there are still annoying differences between the behaviour of the graphics libraries on various platforms. In addition, there is a rather obscure failure of type safety in the case of aliassed arrays.

which forms the subject of this paper. However, there are other, practical issues which need to be considered as well.

The standard Java runtime environment consists 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. All of issues listed above needed to be resolved, in order to achieve this outcome.

1.2 Why Component Pascal?

Component Pascal[7, 6] is a dialect of Oberon-2[9, 5]. 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.

Compared to Oberon-2, Component Pascal has a number of new features which support programming in the large and component-based programming. The most important change is the use of declarative attributes to control the visibility and heritability of types and methods. Thus, types and methods which are intended to be extended or overidden must be declared `EXTENSIBLE`. Unlike Java, the *default* behaviour corresponds to Java's `final`. Methods must also declare whether they are intended to override inherited methods, or are intended to be `NEW`.

Apart from this richer declarative framework, the base semantics of the language follows that of its predecessors Pascal, Modula-2 and Oberon-2. Thus the language supports nested procedures with block scope, reference parameters, and procedure types and values. Although this is a relatively small language, it has all of the needed functionality to be used as a systems implementation language.

The parameter passing mechanisms of the language are more general than its predecessors. Formal parameters may be declared as being of `IN`, `OUT`, or `VAR` (i.e. inout) modes. The default, as with Pascal is value mode. The tighter specification of intended use of parameters allows for stronger static checking by the compiler, but also significantly frees up the use of actual parameters. Recall that for type-safety an `inout` mode formal parameter may have neither a supertype nor a subtype variable passed to it as actual value.

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*.

1.3 Road map

This paper concentrates on just one of the issues, that of achieving the effects of reference parameters. The solutions to the other major issues will be treated elsewhere[2].

In the next section, relevant aspects of the Java runtime environment are briefly described. Following that, in Section 3 a number of different attacks on the problem are evaluated. The chosen solution is described in more detail in Section 4, and some implementation subtleties explored. Finally, Section 5 gives some tentative conclusions.

2 The Java Runtime Environment

At runtime, a *JVM* program may be abstracted as a set of dynamically loaded class "byte-code" files, a stack of method activation records, and an evaluation stack associated with the currently executing method. The class files contain symbolic information in a constant pool, and the executable statements of the methods, encoded as byte-code instructions for the target-independent virtual machine. The virtual machine is an abstract stack machine, with a rich instruction set. The semantics of method invocation and parameter passing are determined by the detailed semantics of the method invocation instructions. These semantics are informally defined in the Java Virtual Machine Specification[4].

Much of the underlying semantics of the Java language are visible in the *JVM* at runtime. For example, the notions of single implementation inheritance, and the implementation of multiple interfaces, are reflected in the operational semantics of the *invokevirtual* and *invokeinterface* instructions.

At runtime, program data consists of primitive data and references to dynamically allocated objects. Primitive data includes various sizes of integers and floating point numbers, unicode characters, and booleans. Every such datum occupies either one or two 32-bit slots in the activation record or evaluation stack. References provide access to instances of class objects and dynamically sized array objects. It is axiomatic that these references provide the only path of access to objects. In particular, there is no notion of "address", and no address arithmetic.

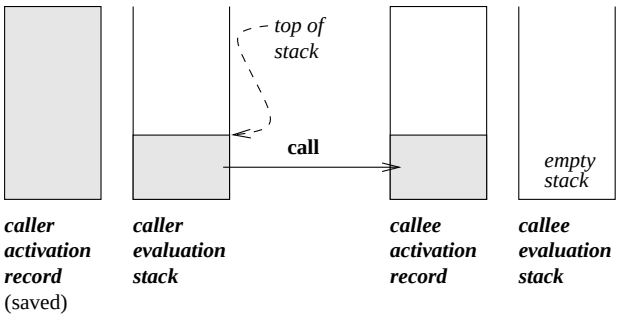

Figure 1. Method call: parameters are taken from the evaluation stack

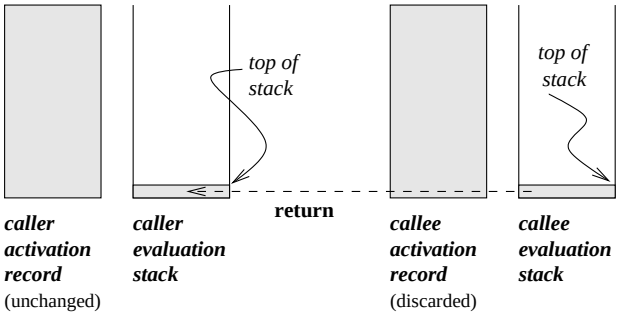

Figure 2. Function value return: stack value is copied onto the stack of the caller

Note that there are no structured data other than the dynamically allocated objects that are accessed by reference.

Language Java passes parameters by value, as illustrated by Figure 1. In the general case if a method requires, say, n parameters, these actual parameters are pushed, in order, onto the evaluation stack. At the call, the n parameter values are transferred into the first n local variable slots of the activation record of the called method.

The only way in which a method can send a value back to its caller is by using the function return-value mechanism. Value returning functions push their return value onto the evaluation stack. The return instruction, as shown in Figure 2, takes the value on the stack and copies it onto the otherwise empty evaluation stack of the calling method to which control returns.

Since these are the only mechanisms whereby values can be passed into and out of method activations, the implementation of alternative parameter passing mechanisms poses an immediate problem. In particular, the implementation of Ada-style `out` and `inout` parameters is problematical, and the realisation of reference semantics even more difficult.

It might be thought, at first sight, that the value-passing semantics of the *JVM* are a perfect match for ANSI C, which also specifies only value semantics. However, this is not the case. Recollect that C idioms rely on the use of the address operator "&" to construct true reference semantics as required. Since the *JVM* does not support the notion of datum address, the attempt fails.

Parenthetically, it might be noted that the nowadays unfashionable *call-by-name* mechanism is readily supported by the *JVM*. In essence, an object is constructed that reveals `get()` and `set()` methods that modify the actual parameter. A reference to this object is passed *by value* to the callee.

3 Reference Parameters: Various Attempts

It is helpful to consider the apparently simplest case of a formal parameter of simple, scalar type. Let us consider a simple integer formal parameter, and an actual parameter which is an integer variable. In order to pass such a parameter by value, the value of the variable is pushed onto the evaluation stack, and the *invoke** instruction does the rest. However, for any parameter semantic which allows for the called procedure to modify the value of the associated actual variable, some other trick is required. In conventional settings, one might consider either passing the address of the actual variable to the called procedure, (passing by reference), or copying the returned value from the called procedure back to the actual variable after the return (passing by copying). The difference between these two mechanisms can only be seen in the case of aliassing, where the same variable is accessible along more than one path. Certain languages (in effect) specify that programs which depend on one or other mechanism are non-conforming.

3.1 Boxing and unboxing

A first attempt at allowing for `inout` semantics consists of placing values in boxes, references to which are then passed by value to the callee. For this mechanism, every type used as a formal parameter has a corresponding wrapper, "box" class defined. This class encapsulates a single single instance field which holds the value. Figure 3 illustrates this concept.

In this attempt the actual value might be loaded into the box immediately before the call, and copied out to its destination variable immediately following the return. Alternatively, all values which are ever passed as `inout` parameters might be boxed throughout their lifetimes.

Note that the called procedure does not return the reference to the box. (If it could do that, it could pass the modified value back directly.) Instead, the caller must duplicate

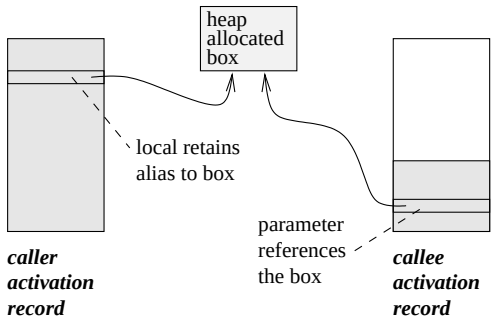

Figure 3. Passing a boxed value to a procedure

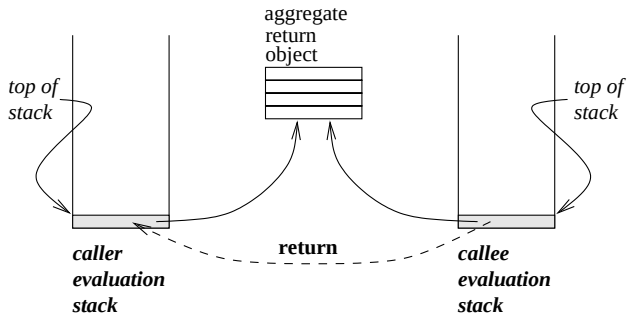

Figure 4. Returning a reference to an aggregate containing all return data

the reference to the box, save one copy in a local variable, and pass the other to the called procedure. When control returns to the caller, the saved reference is used to access and retrieve the boxed value.

The considerations involved here are —

- Copying in and out involves extra overhead for the call
- Copying in and out gives copying rather than reference semantics
- Transient boxing usually involves object creation at call sites
- Permanent boxing creates overheads for **every** access to datum
- Boxing usually involves the generation of *junk classes*

"Junk classes" are classes which are defined for some special purpose. Typically, they are used in one place only, and may only ever have one instance. In the *JVM*, such classes, no matter how simple, require the creation of an additional class file, thus cluttering up the filename space.

It might be noticed that the `RECORD` and `ARRAY` types of Component Pascal are necessarily already boxed in the *JVM* implementation, and thus consume no additional overhead when passed by reference. They also require no additional classes to be defined.

By contrast, `POINTER` types passed to `in-out` formals do require to be boxed. The junk wrapper class corresponding to the pointer type —

```
TYPE Foo = POINTER TO RECORD ... END;
```

behaves as though it originated from the type declaration —

```
TYPE FooBoxJunkClass = RECORD fld:Foo END;
```

3.2 Boxing with refinement

There are possible variations on the copying theme. One such variation would be to pass the value **and** an empty, aliassed box. In this case, the called procedure has direct access to the value, and has to copy the value into the return box immediately prior to the return. This method entails lower performance overheads whenever formal parameters are accessed more than once, on average.

Yet another refinement would entail passing values into the called procedure, and have that procedure construct an aggregate for return. The aggregate would contain all return data, including the conventional "function return value". Figure 4 illustrates the concept.

Before the return, the called procedure creates the return object, and loads the final values of the local formal parameters into the corresponding instance fields. After the return, the caller has the reference to the return object on the top of stack. It must then unpack the instance fields, copying them to their corresponding actual parameter variables. The object is then discarded.

This strategy minimizes the number of boxes which need to be constructed, and allows direct access to the formal parameter values in the called procedure. However, it typically generates massive numbers of junk classes, since it requires a separate class for every separate procedure signature. For simple boxing, we only need a separate wrapper class for every formal type. Aggregate return potentially requires as many classes as there are procedures.

There are further refinements possible. For example every aggregate junk class might define a static field referencing a singleton instance which is repeatedly reused, thus avoiding the overhead of repeated instance creation.

3.3 Using "Thunks"

If providing precise reference semantics is an implementation objective, then it is sufficient to ensure that all effects on aliassed formal parameters modify the same datum. One way of achieving this end is to not pass the actual datum (either boxed or unboxed), but to pass an object

with `get()` and `set()` methods that access the real datum. All read and write operations on the formal parameter within the called procedure are translated into calls of the put and get procedures of the formal parameter object. Because of the correspondence to the usual implementation of call-by-name in *ALGOL-60* we call such access functions "thunks"[3].

Considering first the case of an actual parameter which is a static integer variable, named `cls.var` say. The access object would correspond to the following Java class —

```
class cls_var_access {
    public int get() {return cls.var;}
    public void put(int val) {cls.var = val;}
}
```

Note that in the case of parameters which are aliassed, passing multiple copies of the object reference does no harm. In this should happen the calls of `put()` made on different formal parameters will all modify the same underlying datum.

Since variables of inactive activation records are inaccessible in the *JVM*, the above code does not work for local variables. In such cases, the actual parameter must be copied to a box in the heap. The thunks then access the boxed datum. Clearly, it is important to ensure that each datum is boxed only once, in the case of aliasses. Fortuitously, this property is statically computable.

As noted, a careful implementation of this mechanism can have precise reference semantics. However, this comes at a steep cost. The cost of method call is high compared to either direct or indirect access to a local variable. Equally serious is the extreme proliferation of junk classes. The methods of the previous Subsection created one junk class per distinct formal type, or one junk class per distinct procedure signature. The method of this section requires one junk class *per actual parameter occurrence*, in the worst case.

4 Implementation Issues

4.1 *GPCP* Implementation strategy

The Gardens Point Component Pascal compiler has opted for efficiency in its solution to the `inout` parameter issue. Parameters are copied in, and if necessary out as well. There is some refinement here, beyond the concept illustrated in Figure 3.

As a first step, the boxes which are used to return values are length-1 arrays, rather than instances of some special junk class. The *JVM* does not require a new class file to define how to create an array of a class that is already defined. Thus the problem of junk class creation is averted. Data are placed into the box by inserting the value as the zero-th element of the array, and the value is retrieved from the zero index position.

```
      PROCEDURE Foo(VAR i : INTEGER);
      BEGIN i := 0 END Foo;

      PROCEDURE Bar();
        VAR x : INTEGER;
      BEGIN
        Foo(x);
****          ↑ variable may not be initialized
        ...
      END Bar;
```

Figure 5. Correct code which fails verification

The additional declarative information of Component Pascal formal parameters leads to a useful efficiency here. In the case of `IN` formals just the value is passed. In the case of `OUT` formals an empty, unit-length array is passed to hold the return value. However, in this case the called procedure has an additional local variable defined which acts as a local proxy for the value. Just prior to executing the return, the code of the procedure loads the current value of the local proxy into the zero-th position of the formal parameter return box.

In the case of `VAR`-mode formals the value is passed first, and the return box reference is passed as an additional parameter. The value is used throughout the body of the called procedure, thus avoiding the overheads of indirect access. The final value is copied into the box immediately prior to the return, just as in the case of `OUT` mode.

4.2 Interaction with class verification

The class verifier of the *JVM*, among other checks, ensures that local variables of methods are provably initialized prior to use. Passing a value as a parameter constitutes a visible use. Equally, the copying of a value from a return-box into a local variable location constitues a visible, provable initialization. The class verifier would thus reject a straightforward encoding of the code fragment in Figure 5

The *JVM* will reject the code in the figure, because the code visibly copies the uninitialised local variable x as an argument to the call of *Foo*. It does not help to insist that interprocedural analysis would show that *Foo* does not use the value until after it has been assigned to. The verifier does not perform interprocedural analysis.

As discussed in Section 1.1, since the *JVM* will reject the resulting bytecodes, our compiler must reject the source code, as it has done in the figure.

As it turns out, the enhanced semantics of Component Pascal provide exactly the right amount of declarative support to enable sensible error reporting in those cases where

programming errors involve incorrect initialization of parameters. Thus **gpcp** detects cases where variables are not initialized along all paths leading to use as an actual parameter passed to an input mode formal. Equally, the compiler rejects programs which fail to fulfill their obligation to assign a value to OUT-mode formals along every path which reaches a procedure return. The onus is on the programmer to declare the correct mode for each formal parameter, as part of the declarative contract of that procedure. The compiler will then verify that the obligations of the contract have been provably fulfilled.

In Component Pascal, the example procedure *Foo* should have been declared with OUT mode.

For earlier Pascal dialects, which only have value and VAR modes, we might avoid verification failure by the brute-force strategy of inserting a dummy initialization for every local variable for which dataflow analysis detected an initialization "error". We would thereby avoid spurious error reports in the case of correct code, but would forego the chance of catching genuine program errors. Within a single compilation unit, we might restore some error detection capability by implementing interprocedural analysis, and give a friendly warning to the programmer in the uncheckable case of intermodule calls.

4.3 Uplevel addressing

Providing access to non-local data in languages with nested procedure scopes poses further challenges for *JVM* implementation. A future *FIT* report will describe the somewhat arcane details. Nevertheless, it suffices here to note that one possible solution is to pass the non-local variables as additional, hidden "quasi-parameters". This is the approach that **gpcp** adopts. In this case, of course, there is no declarative framework to define the mode of such quasi-parameters. In **gpcp** the problem is resolved by using the same dataflow analysis which checks for errors in variable initialization. The analysis discovers the required formal mode for each quasi-parameter. The resulting "mode" information is propagated interprocedurally, so that enclosing procedures will use the correct parameter modes. In this way incorrect initialization of non-locally accessed data is detected, while avoiding spurious error messages.

5 Conclusions

The current version of the compiler uses the modified parameter passing mechanisms described here. From one point of view it seems a pity to accept imprecise semantics in this way. An alternative point of view is that the specification of precise reference semantics is a design flaw. In many ways the issues involved in parameter passing to remotely invoked procedures mirror those of the *JVM*. In both cases any actual parameter variables are inaccessible during the execution of the called function. Thus any distributed extension of Component Pascal would necessarily need to loosen the strictness of the specification of parameter passing mechanisms.

The compiler, in its current form, has achieved all of its initial goals. The whole of the language is compiled, and the compiler is capable of generating either applications or "applets". Early benchmarking shows that the runtime efficiency of programs is equivalent to that achieved with the Java language. Furthermore, if programs are run in an environment which uses a just-in-time compiler, the runtime efficiency is very close to that of the same programs directly compiled to native code.

For the future, it is intended to use the compiler as a platform for investigation of those optimizations that are unlikely to be discovered by a just-in-time compiler. It is also intended, in time, to produce compilers for other languages using the technology developed for this project.

Finally, we return to the question of the intrinsic limitations of the *JVM* as a compiler target. Some popular programming language features are inherently type-unsafe, and do not admit of simple translation for the *JVM*. For example, undistinguished unions of the kind that occur in ANSI C, or in the tagless variant types of Pascal seem particularly intractable. *Tagged* variants admit to an elegant and type-safe implementation in the *JVM*, although most programs using this construct use type-unsafe idioms which just happen to be tolerated by most compilers[2] using the *caveat emptor* principle. Pointer type-casts are similarly intractable.

In all such cases, if one is to be absolutely correct, it is possible to achieve the desired semantics by mapping the program memory space to a large array of untyped values, and use another level of interpretation. This "solution" is unattractive for efficiency reasons, although it has some application in reverse engineering.

Acknowledgement

The **gpcp** compiler was written in one continuous, intensive burst during the final three months of 1998. Since then it has acquired its foreign language interface, and has started to become more robust. The compiler, together with all of its sources, will be placed in the public domain. Most users will wish for additional tool support to use this system. Diane Corney, who joined to project in mid-1999, is addressing this lack.

The project is part of the ongoing work of the Programming Languages and Systems Research Centre at QUT. I wish to acknowledge important discussions with Clemens Szyperski regarding the fine points of Component Pascal

[2] Including **gardens point modula-2**, to our shame!

semantics. Anthony Sloane, William Waite, Cristina Cifuentes and Wayne Kelly all contributed useful discussion at critical points.

Some of the insights and outcomes of the Java Tools project at the CRC for Distributed Systems were critical to the understanding of the issues of compiling for the *JVM*. The contributions of Nam Tran, Ken Baker and Diane Corney are hereby acknowledged.

This project was partly supported by ARC grant number A49700626.

References

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

[2] K. J. Gough and D. Corney. Translating languages other than java for the java virtual machine. Technical report, Faculty of Information Technology, QUT, 1999. Preprint available at `http://www.fit.qut.edu.au/~ gough`.

[3] P. Z. Ingerman. Thunks. *Communications of the ACM*, 4(1), 1961.

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

[5] H. Mössenböck and N. Wirth. The programming language oberon-2. *Structured Programming*, 12:179–195, 1900.

[6] C. Pfister. The evolution of oberon-2 to component pascal. Technical report, Oberon Microsystems, 1998. Available at `http://www.oberon.ch/resources`.

[7] C. Szyperski. Component pascal language report. Technical report, Oberon Microsystems, 1997. Available at `http://www.oberon.ch/resources`.

[8] R. Tolksdorf. Programming languages for the java virtual machine. Web bibliography. Available at `http://grunge.cs.tu-berlin.de/~ tolk/-vmlanguages.html`.

[9] N. Wirth. The programming language oberon. *Software Practice and Experience*, 18(7):671–690, 1988.