Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science 141 (2005) 35–52
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Translation-Facilitated Comparison Between the Common Language Runtime and the Java Virtual Machine

Sam Shiel[1] ,[2]

Ian Bayley[3]

Department of Computing
Oxford Brookes University
Oxford OX33 1HX, U.K.

Abstract

We describe how programs can be converted from the Common Language Runtime to the Java Virtual Machine, based on our experience of writing an application to do so. We also recount what this experience has taught us about the differences between these two architectures.

Keywords: bytecode semantics, bytecode transformation, bytecode verification

1 Introduction

Two of the most well-known stack-based virtual machines in use today are the Common Language Runtime (CLR) and the Java Virtual Machine (JVM). There are many obvious similarities between the two. Both are statically-typed, provide automatic memory management (garbage collection), have

[1] Thanks to the reviewers who provided additional ideas and pointed out shortcomings in the original draft
[2] Email: sam.shiel@gmail.com
[3] Email: ibayley@brookes.ac.uk

doi:10.1016/j.entcs.2005.02.039

multi-threading abilities and support an object-oriented model at the instruction level. Further to the last point both use a single-inheritance object model, with multiple interface implementation. Both posses the notions of classes, abstract classes, interfaces, methods (both virtual and static) and fields (both instance and static).

It is therefore a matter of some academic interest to know to what extent the JVM and CLR are equivalent. Although Gough has already compared the two [3], we felt the best method of comparison was to write an application to translate programs originally compiled to run on the CLR into programs that run on the JVM. This would force any differences in capability to be examined. One of us (Shiel) has therefore written such an application. As expected, the issues that arose in the design and implementation of this application reflected the differences and similarities between the CLR and the JVM.

Furthermore, the CLR also supports the notion of verification. What is intriguing about this is that the JVM has been the subject of intensive research over the past few years [10] but little has been published on the CLR, even though there are in fact many important differences between the two virtual machines. A detailed comparison between the two would allow researchers to apply to the CLR results they have obtained for the JVM.

A minor first obstacle is the use of different terminology in CLR for otherwise familiar JVM concepts. The collection of classes representing an application is bundled together as an assembly, which is the CLR equivalent of a package or a JAR file. An assembly consists of two types of data: method bodies (sequences of instructions) and metadata. The metadata holds more information than can be found in JVM '`.class`' files but is similar to the constant pool.

As for architecture of the CLR itself, the program counter is renamed the instruction pointer and the operand stack is renamed the evaluation stack. The local variable array is not renamed but this is perhaps misleading as it does not include the arguments with which the method was called; they are held separately in an argument array.

One of the claims made for the CLR is that it is language neutral and enables multi-language applications to be developed. To assist this interoperability, a Common Type System (CTS) is defined together with a Common Language Specification (CLS) a set of rules that each component of the application must obey. Finally, the language used to write the instructions is called the Common Intermediate Language (CIL).

This paper is structured along the lines of the CLR type system. Section 2 considers problems that can be encountered when you are using primitive types alone. Section 3 describes value types, which are among the biggest

CLR Type		JVM Type	Description
Boolean	$\mapsto$	boolean	True or False
SByte	$\mapsto$	byte	Signed 8-bit byte
Int16	$\mapsto$	short	Signed 16-bit integer
Char	$\mapsto$	char	Unsigned 16-bit character, Unicode format
Int32	$\mapsto$	int	Signed 32-bit integer
Int64	$\mapsto$	long	Signed 64-bit integer
Single	$\mapsto$	float	Single precision float, IEEE 754 format
Double	$\mapsto$	double	Double precision float, IEEE 754 format
Byte	$\mapsto$	*none*	Unsigned 8-bit byte
UInt16	$\mapsto$	*none*	Unsigned 16-bit integer
UInt32	$\mapsto$	*none*	Unsigned 32-bit integer
UInt64	$\mapsto$	*none*	Unsigned 64-bit integer

Table 1
Mapping of CLR value types to JVM primitives.

differences between CLR and the JVM. Section 4 discusses arrays. Section 5 introduces objects and is extended by Section 6, which considers inheritance and the class hierarchy. Section 7 mentions other features like pointers and exceptions. Finally, Section 8 concludes.

2 Primitive Types

As Table 1 reveals, there is a reasonably close correspondence between CLR value types and JVM primitive types. The exceptions are the CLR unsigned types, which have no JVM equivalent.

2.1 Constant Loading

Numeric constants are loaded onto the stack in the CLR using the `ldc` family of instructions, while string references are loaded using the `ldstr` instruction. All of these instructions were translated using the `ldc` JVM instruction, which loads a constant from the constant pool onto the stack. However a more efficient solution would be to use, where possible, the instruction `bipush` (or `sipush`) to load an signed byte (or signed short) literal onto the stack.

There are also specialised CIL instructions to push 32-bit integers in the range -1…8, each of which take up a single byte. Most of these can be translated into corresponding single byte JVM instructions, such as `iconst_1`, which pushes the integer constant 1 onto the stack.

2.2 Generic Instructions

Many JVM instructions encode type information as prefixes. For example, `iadd` adds two integers, and `fmul` multiplies two floats. Taken together with the instruction operands (for more generic instructions like `ldc`), it is always possible to infer the types being manipulated.

Translating numeric constant loading was straightforward because the `ldc` family of CIL instructions has a different instruction for each type of value that may be pushed onto the stack (integer, long, float or double). However, most CIL instructions are generic and do not contain information from which the types can be inferred. (For example, for multiplication and addition there is just one `mul` instruction and one `add` instruction). The reason for this is that the CIL was always intended to be compiled rather than interpreted [5]. In order to correctly translate CIL instructions to JVM instructions, it was therefore necessary to maintain a type model of the CLR evaluation stack for each method being translated.

2.3 Arithmetic and Logic

To translate the arithmetic and logic instructions in the CLR to their JVM equivalents, we simply need to identify the types on top of the evaluation stack and generate the appropriate JVM instruction. The CLR also has support for unsigned integer (although this is not CLS compliant) and overflow checked arithmetic, both of which are both lacking in the JVM. Translating these features to the JVM is possible but awkward.

To simulate a `UInt32`, we could use a `long` with the data type invariant that the top 32 bits are always clear; these bits would therefore be masked out after every operation. A similar approach would work for a `Byte` or a `UInt16` but for a `UInt64` we would need to use an auxiliary variable just to store the most significant bit (MSB). Extra instructions would be inserted to set or clear this bit after each operation as a function of the result and the MSBs of the operands. To translate overflow checking, we would again use an oversized workspace (large enough that no genuine overflow can take place), test the result and throw a customised exception if the result is too large.

Value Type	CLR	JVM
Integer	`beq` *destination*	`if_icmpeq` *destination*
Reference	`beq` *destination*	`if_acmpeq` *destination*
Long	`beq` *destination*	`lcmp` `if_eq` *destination*
Float	`beq` *destination*	`fcmpl` `if_eq` *destination*
Double	`beq` *destination*	`dcmpl` `if_eq` *destination*

Table 2
Conditional branch instructions compared.

2.4 Branching

There are two complications surrounding the translation of CLR flow control instructions. Firstly, JVM methods are restricted in size to 65,535 bytes. It is therefore only possible to directly translate CLR methods that will result in a JVM method whose size is less than or equal to this limit.

Secondly, the JVM comparative branching instructions only directly operate on integers and references, while the more generic CLR instructions also operate on floats, long and doubles. This requires the translator to generate two JVM instructions when comparing floats, doubles or longs. As an example, Table 2 shows the equivalent CLR and JVM instructions to branch to a given destination if the two values on top of the evaluation stack are equal. Similar translations apply for the `bne`, `bge`, `bgt`, `ble` and `blt` CIL instructions. The CLR also has conditional branch instructions in which the integer values are compared unsigned, but these can be translated in a manner similar to the unsigned arithmetic instructions.

3 Value Types

Perhaps the most important feature provided by the CLR that is lacking in the JVM is value types. Value types fulfil the role of both primitive types in the JVM and user-defined structures and enumerations such as those found in the C programming language.

Like primitive types in the JVM, value types are passed by value and are stored on the stack, rather than the heap (except in the case of a value type that is a member of a reference type). However, unlike JVM primitive types, a value type can have associated methods. These methods are not stored on the stack, so a value type only takes up as much stack space as is required to represent its actual value.

Translating the semantics of value types from the CLR to the JVM is problematic. One approach would be to translate CLR value types that represent primitive types to their JVM primitive type equivalents, and then create a JVM *box* class that duplicates the functionality and interface of each such CLR value type. Method invocation on value types could then be simulated by boxing the primitive type into an instance of the corresponding box class, invoking the appropriate method and then unboxing the primitive type from the object instance.

In more detail, boxing a value type involves removing it from the evaluation stack, creating a reference type on the heap and storing the value inside the reference type, before placing the reference to the boxed value on the evaluation stack. Unboxing removes the reference type from the stack and retrieves the value type from inside it, before placing the value type on the evaluation stack. This of course entails all the expense involved in instantiating and then garbage collecting the box object, and so is far from ideal.

Before we describe an alternative approach, note that boxing and unboxing would be required to translate the specific CLR instructions `box` and `unbox` that box and unbox value types to and from references. Each JVM box class would be given additional get and set methods so that the `box` instruction can be translated by instantiating the appropriate box class and invoking the set method, while the `unbox` instruction can be translated by invoking the get method. An example use of boxing and unboxing is given in Section 7.1.

The alternative approach takes advantage of the fact that CLR value types must be final, with a fixed number of methods. This allows us to provide a set of JVM classes identical to those CLR value types that represent the primitive types, but with one crucial difference: all the methods are defined as static, with an extra parameter added whose purpose is to pass the value of the value type to the method. Thus we could translate a virtual method invocation on a CLR value type into a JVM static method invocation with the value as a parameter. This static method invocation would not require the overhead of boxing and unboxing the primitive type and would therefore be much less expensive.

In both approaches, structure value types would have to be translated as reference types and a *deep-cloning* copy method provided to enable simulation of value type semantics (since a structure may contain other value types that are themselves emulated using reference types). When the item on top of the evaluation stack is a structure, the CLR `dup` instruction would be translated by invoking its copy method, which would recursively invoke the copy method on all of its structure members.

These issues and surrounding complications have been explored by Gough [2], where he uses the term ‘reference surrogates’.

4 Arrays

The CLR provides two types of arrays: single-dimensional arrays (also known as vectors) and multidimensional arrays. A vector can contain references to other vectors, enabling (possibly ragged) arrays-of-arrays, which is how multidimensional arrays are implemented in the JVM, as it has no direct equivalent of CLR multidimensional arrays.

CLR vectors are subtypes of the `System.Array` abstract class, so for every CLR type there is a corresponding vector type derived from `System.Array`. There are specific CIL instructions for creating vectors, manipulating (loading and storing) elements within them and obtaining the length of a vector. Fortunately, each of these has a direct JVM counterpart, so translating them is easy.

The next issue is how to deal with methods invoked on vector type instances. This is probably most easily dealt with by creating a family of JVM classes (one for each CLS type) that implement the interface defined by the `System.Array` abstract class. The non-static methods defined in the `System.Array` class would need to be changed to be static, and an extra parameter added to pass the array instance reference. The translation would then involve replacing any vector instance method invocations in the CIL source to the corresponding JVM static method.

Multidimensional arrays in the CLR are also represented as subtypes of the `System.Array` abstract class. However, there are no special CIL instructions for dealing with multidimensional arrays. Instead, they are treated just as any other reference types. Translating CLR multidimensional arrays to the JVM would therefore be a matter of creating JVM classes that duplicate the interface and functionality of the CLR multidimensional array classes.

A final complication is the way arrays are initialised in the CLR. Rather than generating a sequence of instructions to initialise the elements of array (as is done on the JVM), the CLR instead uses a special helper method which takes the array to initialise and a structure representing the initial element values as arguments. This structure would need to be translated into the appropriate sequence of JVM instructions.

5 Objects

5.1 Properties

Unlike the JVM, the CLR has the notion of properties as members of types. However, properties are actually implemented as regular accessor methods, specially identified in the metadata as properties. All references to properties in the CIL code are made by invoking the accessor methods, making translation of properties entirely transparent.

5.2 Member Access Modifiers

The CLR modifiers public and private have the same meaning as the JVM public and private modifiers and are translated as such. In addition, the CLR provides more finely-grained member access modifiers than the JVM. As usual, all class members are accessible by the enclosing class. The modifier 'Family' means the class member is also accessible by derived classes. The modifier 'Assembly' means the class member is accessible by classes in the same assembly (which, remember, is the CLR equivalent of a package). In addition, the modifiers 'FamAndAssem' (respectively 'FamOrAssem') hold when the conditions for family access and (respectively inclusive or) assembly access both hold. So 'FamOrAssem' translates to protected and 'Assembly' translates to package visibility which, being the default, is denoted by the absence of a modifier. The JVM has no equivalents for either the 'FamAndAssem' or the 'Family' access modifiers.

5.3 Method Arguments and Local Variables

In the JVM, each method has access to set of 32-bit implicit registers, which are used as storage for both its arguments and local variables. Since each register can only store a 32-bit value, long and double sized arguments and local variables consume two adjacent registers. In the case of a static method, the arguments to a method will reside in registers $\{0, \ldots, N + D\}$, where N is the total number of arguments and D is the number of long or double sized arguments (which consume two registers). In the case of a non- static method, the instance reference is implicitly placed in register $\{0\}$, so the method arguments will reside in registers $\{1, \ldots, N + D\}$, where N and D are defined as above.

Any local variables belonging to the method occupy the subsequent registers, with long and double local variables again requiring two adjacent registers. Since method arguments and local variables are placed in the same set of registers, they are both accessed with the appropriately typed load and store instructions. Note that when loading or storing a double or long, the load and store instructions always reference the lower of the two registers within which the value is stored, but they nevertheless load the contents of both registers as a single value onto the evaluation stack.

In the CLR, method arguments and local variables use two distinct sets of registers and are accessed using different instructions. Unlike the JVM, the register sizes are variable, so each argument and local variable only consumes a single register, regardless of its size. Arguments to methods are accessed with the `ldarg` and `starg` instructions (in the case of instance methods the argument register {0} contains the instance reference, as with the JVM). Local variables are accessed using the generic `ldloc` and `stloc` instructions.

Translating method arguments and local variables from the CLR to the JVM therefore requires the two sets of registers in the CLR to be mapped to the single set of registers in the JVM, taking into account the need to allocate two adjacent registers for each long and double sized argument or local variable.

5.4 Accessing Fields

Fields are accessed very similarly on the CLR and JVM. The CLR uses the `ldfield` and `stfld` instructions to load and store instance fields, respectively, while the JVM uses the directly equivalent `getfield` and `putfield` instructions. For static fields the CLR uses the `ldsfld` and `stsfld` instructions, which also have direct JVM counterparts in the `getstatic` and `putstatic` instructions.

6 Inheritance

6.1 Classes

CLR classes must ultimately derive from `System.Object`, while JVM classes must derive from `java.lang.Object`. In both cases a class can only inherit from a single superclass but may implement multiple interfaces. Ignoring the special case of nested classes the default visibility of a class on both platforms is private (meaning it can only be accessed from the assembly or package within which it is declared) but it can also be specified to be public.

`System.Object` provides similar functionality to that of `java.lang.Object`. The most complete solution for translating CLR classes to JVM classes would be to create a JVM class whose immediate superclass is `java.lang.Object` but which mimics the interface of `System.Object`. All translated CLR classes would then derive from the JVM version of the `System.Object` class, which would itself derive from `java.lang.Object`. A simpler but limited approach, which can only be used when no members of the `System.Object` class are accessed, is to translate a CLR class whose immediate superclass is `System.Object` into a JVM class with `java.lang.Object` as its immediate superclass.

6.2 Interfaces

In both cases an interface is simply a pure abstract class flagged as an interface, and may only define public members. In both the CLR and JVM, an interface cannot be derived from any type but it may have one or more superinterfaces. A CLR interface can therefore be directly translated into a JVM interface.

6.3 Method Invocation

The CLR and JVM both support static and instance methods. On the JVM, all instance methods are virtual, but the CLR has both virtual and non-virtual instance methods. In addition, the CLR allows virtual methods to be defined with the `newslot` directive, which means that the method will occupy a new slot in the v-table instead of overriding any inherited method. Since the JVM has no notion of non-virtual instance methods and the `newslot` directive, these cannot be directly translated in all cases, although name-mangling may be a solution.

On both platforms method arguments are pushed left to right, simplying translation. The CIL `call` instruction performs an early-bound call and is typically used to invoke static methods and methods in superclasses [8]. It therefore fulfills the role of both the JVM `invokestatic` and `invokespecial` instructions.

The CIL `callvirt` instruction performs a late-bound call [8] and is used to invoke virtual methods. It is the equivalent of the JVM `invokevirtual` and `invokeinterface` instructions.

6.4 Constructor Invocation

Both the CLR and the JVM support the notion of static and instance constructors. Static constructors (also known as class constructors) are typically used to initialise static fields belonging to a class and are called implicitly, before any instances of the class are created or any static members are accessed. Instance constructors are used to create and initialise class instances and are called explicitly when instantiating a class.

The CLR names all static constructors `.cctor` and all instance constructors `.ctor`, while on the JVM they are named <`clinit`> and <`init`>, re-

CLR	JVM

`ldc.i4.3`
`newobj void Polygon::.ctor(int32)`
	`new Polygon`
	`dup`
	`iconst_3`
	`invokespecial Polygon/<init>(I)V`

Table 3
Constructor invocation on the CLR and JVM.

spectively. Static constructors behave the same way on both platforms, and are easily translated. However, instance constructors are invoked quite differently complicating translation.

Consider a class representing a polygon, with a constructor that takes a single integer argument specifying the number of sides for the polygon instance. Table 3 compares the equivalent CLR and JVM instructions necessary to instantiate a polygon object representing a triangle.

As can be seen, the CLR version pushes the argument to the instance constructor onto the evaluation stack before a single CIL instruction, `newobj`, both creates the new instance and calls its instance constructor. In contrast, the JVM version creates a new reference, duplicates it, loads the operand stack and then invokes the instance constructor explicitly.

To translate the CIL `newobj` instruction in the case when the constructor being invoked takes a single argument, the translator would need to insert the `new` and `dup` JVM instructions just before the single argument is loaded (or for any N-ary constructor just before the N arguments are pushed by N consecutive loads). This of course assumes that all arguments are explicitly loaded rather than calculated. If the argument 3 was the result of pushing and adding the `ints` 1 and 2, for example, then the `new` and `dup` instructions would have to be inserted before these two pushes. More generally, we need to perform a data-flow analysis to find the correct location in the instruction sequence to push the object reference. A simpler (though inefficient) 'brute-force' solution would be to translate the `newobj` instruction as a sequence of instructions that first stores the N constructor arguments on top of the stack into local variables, then instantiates the object and duplicates its reference on the stack, before loading the constructor arguments back onto the stack and then finally invoking the constructor.

7 Other Issues

7.1 Reference Parameters

The CLR allows value types to be passed by reference as well as by value, whilst the JVM primitive types can only be passed by value. The address of a local variable or method parameter can be loaded as a *managed pointer* onto the CLR evaluation stack using the `ldloca` and `ldarga` CIL instructions respectively. The `stind` family of CIL instructions takes a value and a managed pointer from the stack and stores the value in the location referenced by the managed pointer, while the `ldind` instructions take a managed pointer from the stack and place the value found at the location referenced by it on the stack. The following CIL code provides an implementation of the archetypal *swap* method using reference parameters (C-like pseudocode has been added to the comments):

```
.method static void swap(int32& X, int32& Y)
 ldarg.0  // load address of X // temp = *x; ...
 ldind.i4 // indirectly load value of X
 stloc.0  // save value of X temporarily
 ldarg.0  // load address of X // *x=*y; ...
 ldarg.1  // load address of Y
 ldind.i4 // indirectly load value of Y
 stind.i4 // store value of Y to address of X
 ldarg.1  // load address of Y // *y = temp; ...
 ldloc.0  // load original value of X
 stind.i4 // store original value of X to address of Y
 ret      // return from method
```

In order to swap two local variables using the above code, we must first load their addresses on the evaluation stack using the `ldloca` instruction, before invoking the *swap* method:

```
...
ldloca 0 // load address of X
ldloca 1 // load address of Y
call void swap(int32&, int32&)
...
```

CLR verifiability constraints forbid managed pointers to pointers, so reference parameter passing can be translated to the JVM using boxing and unboxing (this technique is one of several proposed by Gough [1]). For each primitive type, an appropriate box class must be defined with get and set methods as described in Section 3.

In our translation for the JVM below, we have assumed an appropriate box class for `int` equipped with get and set methods. Observe that the CLI instruction `ldind` is translated by invoking the get method and `stind` is translated by invoking the set method. Here now is the JVM version of `swap`:

```
aload_0                     // load reference to boxed X
invokevirtual Box.get:()I   // retrieve value of X from box
istore_2                    // save value of X temporarily
aload_0                     // load reference to boxed X
aload_1                     // load reference to boxed Y
invokevirtual Box.get:()I   // retrieve value of Y from box
invokevirtual Box.set:(I)V  // set value of boxed X to Y
aload_1                     // load reference to boxed Y
iload_2                     // load temporarily saved value of X
invokevirtual Box.set:(I)V  // set value of boxed Y to X
return                      // return from method
```

Each occurrence of a `ldloca` and `ldarga` instruction is translated to the sequence of JVM instructions to instantiate the appropriate box object and invoke the set method with the local variable or argument as the parameter.

```
...
new Box                          // create box for X
dup
invokespecial Box."<init>":()V
astore_2                         // save reference to X box
aload_2                          // load reference to X box
iload_0                          // load value of X
invokevirtual Box.set:(I)V       // box value of X
new Box                          // create box for Y
dup
invokespecial Box."<init>":()V
astore_3                         // save reference to Y box
aload_3                          // load reference to Y box
iload_1                          // load value of Y
invokevirtual Box.set:(I)V       // box value of Y
aload_2                          // load boxed X reference
aload_3                          // load boxed Y reference
invokestatic swap:(LBox;LBox;)V
aload_2                          // load address of boxed X
invokevirtual Box.get:()I        // unbox value of X
istore_0                         // store new value of X
```

```
aload_3                          // load address of boxed Y
invokevirtual Box.get:()I        // unbox value of Y
istore_1                         // store new value of Y
...
```

Although it is also possible to obtain a managed pointer to an instance or static field (with the `ldflda` and `ldsflda` instructions respectively) and an array element (with the `ldelema` instruction), we have not yet considered how to translate these features to the JVM.

7.2 Exception Handling

At first glance exception handling on the CLR and JVM is very similar, with both supporting the familiar 'try-catch-finally' model. However, they differ in several ways. Firstly, the CLR allows any instance or subclass of `System.Object` to be thrown, but the JVM only allows the throwing of an instance or subclass of `java.lang.Throwable`.

In addition to catch and finally clauses which mirror those found in the JVM, the CLR supports *fault* and *filter* clauses. A fault clause differs from a finally clause in that it is executed only if an exception is thrown within the associated try block. A filter clause is effectively a predicate that can decide whether the exception should be handled or ignored.

There is also a difference in the way that finally clauses are dealt with on the CLR. Consider the example below (taken from [7]):

```
public void tryFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}
```

The JVM instructions corresponding to the above code are as follows:

```
00 aload_0
01 invokevirtual tryItOut()V
04 jsr 14
07 return
08 astore_1
09 jsr 14
12 aload_1
13 athrow
```

```
14 astore_2
15 aload_0
16 invokevirtual wrapItUp()V
19 ret 2
```

The `try` block, between offsets 0 and 7, calls `tryitOut()`, then calls the finally block as a subroutine, and subsequently returns. The `finally` block, between offsets 14 and 19, stores the return address in register {2}, calls `wrapitUp()` and then returns using the location stored in register {2}.

Surprisingly, there is also a `catch` block, between offsets 8 and 13, which stores the exception thrown, calls the `finally` block, retrives the exception, and then throws it again.

If you consider the number of instructions that just store and retrieve exceptions and return addresses, it is unsurprising that the same example in the CLR is more concise:

```
00 ldarg.0    // start of try
01 call instance void tryItOut()
06 leave.s 15 // end of try
08 ldarg.0    // start of finally
09 call instance void wrapItUp()
14 endfinally
15 ret
```

The specialised instructions `leave` and `endfinally` quit the `try` block and `finally` block respectively. Note that we do not need to translate the rethrowing of an uncaught exception.

7.3 Issues Not Examined

There are several features of the CLR that we have not examined at all. These include tail calls (a detailed discussion of tail calls and the JVM is given by Schinz and Odersky [11]), threading, delegates, coercion, enumerations and events.

8 Conclusions and Further Work

The translation application developed is able to translate a significant subset of the constructs and functionality of the CLR to the JVM, including nearly half of the CIL instruction set. It is able to translate:

(i) Class, abstract class and interface definitions.

(ii) Static and instance (nullary) constructor invocation.

(iii) Virtual and static method definitions and invocation.

(iv) Field definitions and access.

(v) Value types corresponding to JVM primitive types.

(vi) Flow control instructions.

(vii) Arithmetic (not including unsigned and overflow checked) and logic instructions.

(viii) Arrays of value types corresponding to JVM primitive types.

This investigation has thrown up a number of differences between the CLR and the JVM. Of particular interest to us are the typing rules and operational semantics associated with any formal study of verification.

(i) CIL instructions are generic so the conditions on typing rules will be more complex.

(ii) Verification can be done in a single pass without the need for a fixpoint iteration thanks to the following extra restriction: the operand stack after an unconditional jump is assumed to be empty unless that location is itself the target of a forward jump ([9], chapter 6.1.7.5). Due to this characteristic alone, it would interesting to see how much easier it would be for theorem provers to check the soundness of verification.

(iii) The CLR type system is richer in that data items can either be values or references, and this information must be consulted in almost every typing rule. The instructions `box` and `unbox` must distinguish values and references, and their locations will also be different.

(iv) The CLS does not need a rule that an object is initialised before it is used, because it cannot be created without first being initialised. It does, however, like the JVM require that no object is initialised twice, but this must also be relatively easy to ensure, since every object outside a constructor method is already initialised.

(v) Instead of using the generic concept of a subroutine, the CLR has specialised constructs for exception handling and the use of ordinary control flow instructions is restricted. For example, only `throw`, `leave`, `endfilter`, `endfinally` and `rethrow` can be used to leave a protected block, and not any of the conventional branch instructions. The JVM uses return addresses as first class values so a data-flow analysis is needed to confirm that subroutines are used in a last-in-first-out manner. Most restrictions on protected blocks are entirely lexical and easy to check.

We now plan to use these insights to formalise the semantics of the CLR, in a manner similar to what many have done with the JVM [10]. Then we intend to derive the transformation outlined in this paper as a data refinement. More precisely, we will define a function from CLR state to JVM state, and derive equivalents for each CLI instruction, along with the guards that define their applicability.

We shall also derive a data type refinement going the other way, from the JVM to the CLR. We chose the other direction for our research simply because the latter is considered more expressive. Aspects of this paper that would benefit from formal reasoning include dynamic method dispatch, overflow arithmetic and registers. In particular, the treatment of object initialisation that we have advocated, could be proven correct by deriving 'commuting' conditions that dictate when two instructions could be swapped.

We also intend to construct a formal model of exception handling for the CLR, as a way of illuminating the official specification. This would enable us to examine the relative expressivity of JVM and CLR exception handling.

It would be interesting too to see a formal specification of how managed pointers are managed, together with proofs that no illegal memory is ever accessed.

We hope this introduction will inspire other researchers to investigate the semantics and verification of CIL more closely. Further enquiry may discover whether verification is too loose, too restrictive, both or neither and may contribute greatly to the design of future virtual machines.

References

[1] Gough, J., *Parameter passing for the java virtual machine* (1998).
URL citeseer.ist.psu.edu/gough98parameter.html

[2] Gough, J., "Compiling for the .NET Common Language Runtime," Prentice-Hall, 2001.

[3] Gough, K. J., *Stacking them up: a comparison of virtual machines*, in: *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture* (2001), pp. 55–61.

[4] Gough, K. J. and D. Corney, *Implementing languages other than java on the java virtual machine*.
URL citeseer.ist.psu.edu/499197.html

[5] Hamilton, J., *Language integration in the common language runtime*, SIGPLAN Not. **38** (2003), pp. 19–28.

[6] Lidin, S., "Inside Microsoft .NET IL Assembler," Microsoft Press, 2002.

[7] Lindholm, T. and F. Yellin, "Java Virtual Machine Specification," Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] Meijer, E. and J. Gough, *Technical overview of the common language runtime* (2000).
URL citeseer.ist.psu.edu/meijer00technical.html

[9] Miller, J. S. and S. Ragsdale, "The Common Language Runtime Annotated Standard," Addison-Wesley, 2004.

[10] Qian, Z., “A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines,” Number 1523 in Lecture Notes in Computer Science, Springer Verlag, 1999.

[11] Schinz, M. and M. Odersky, *Tail call elimination on the java virtual machine*, in: *Proc. ACM SIGPLAN BABEL’01 Workshop on Multi-Language Infrastructure and Interoperability.*, Electronic Notes in Theoretical Computer Science **59** (2001), pp. 155–168, http://www.elsevier.com/locate/entcs/.