Grail and .NET
Version 1.0*

K. MacKenzie

16th April 2002

1 The .NET platform

According to the Microsoft .NET webpage [6], "Microsoft .NET is an XML Web services platform that will enable developers to create programs that transcend device boundaries and fully harness the connectivity of the Internet".

This isn't very helpful, so we'll attempt to give a description of the structure of .NET. The core of .NET consists of the Common Language Infrastructure (CLI), which in turn consists of:

- an "execution environment" (in other words, a virtual machine) called the Common Language Runtime (CLR)
- a comprehensive type system called the Common Type System (CTS) allowing many computer languages to be implemented in the CLR bytecode (which is referred to as the Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL))
- something known as *metadata* which is analogous to the information encoded in the Java class-file format (including the contents of the runtime constant pool)
- a large collection of Java-type class files providing useful computational services.

It is remarkably difficult to find any technical information about .NET without having to pay for it. The only on-line resources available appear to be the ECMA CLI specification [1], which a highly technical set of documents running to almost 500 pages in total, and the paper of Meijer and Miller [5] which provides a useful overview of some aspects of the CLI, enlivened by derogatory remarks about Java. For ease of reference, we will give a brief description of the CLR architecture and then a summary of the entire CIL instruction set. Since

*This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

we are attempting to condense a large amount of information into a small space, we give no guarantees that everything said here is 100% accurate or complete; however, it should suffice to give a general idea of the structure and capabilities of the .NET platform.

1.1 The .NET CLR

The basic unit of distribution for CLR programs is the so-called *assembly*, which is similar to a Java jar file; an assembly contains CIL code and metadata describing properties of the code. The metadata (which is user-extensible) contains a great deal of information (the specification in [1] is 191 pages long), and describes things such as the methods and constants contained in an assembly, attributes of fields and methods, linkage information, constants, type definitions, detailed information about the layout of data in memory, security information and so on.

Code for the CLR comes in several flavours. Code can be *managed* or *unmanaged*. The exact meaning of these terms seems to be rather vague, but managed code appears to be code which has been written with the CLR in mind, whereas unmanaged code consists of, for example, native machine code. Code can also be *verifiable* or *unverifiable*. As in the JVM, the memory safety of CIL bytecode can be checked by a verification algorithm; however, many of the CIL instructions are designed for the implementation of language such as C, and are intrinsically unsafe. Code using such instructions is said to be unverifiable and will immediately be rejected by the verifier; however verifiable code, which uses only safe instructions, can be analysed by the verifier, and if it is accepted then the safety of the code will be guaranteed.

The architecture of the CLR is similar to that of the JVM. Each method invocation has associated with it a stack frame; as in the JVM this has a stack of bounded length where arithmetic and other operations are carried out, together with an array of local variables. There is also a separate array containing the method arguments. The entries of this array are both readable and writable. The argument array is discarded upon method return, so it is not immediately obvious why it is useful for it to be writable. This feature appears to be connected with an optimised form of recursion; see the description of the `jmp` instruction below.

Unlike the JVM, each local variable can only contain data of a fixed type, which is specified in the metadata. This allows the implementation of *polytypic* instructions. For example, there is only one `add` instruction; when such an instruction is encountered the types of its stack arguments can be deduced from the known types of local variables and method arguments (and the results of previous instructions) and the appropriate action can be carried out. This contrasts with the JVM strategy, where the instructions specify the types which they expect. The CLR strategy is designed for JIT compilation; if CIL code was to be interpreted then the interpreter would need to keep track of the types of all objects on the stack at all times during actual execution (with the JVM this information only needs to be tracked during the verification procedure), which

would add a considerable overhead. [It's not clear how the CLR is supposed to function in an environment with limited memory and/or processing resources. There appears to be something called "Windows CE .net" (available for only $995 until 31st May 2002 thanks to a special offer saving up to 65% on the standard retail price), but I haven't been able to find any technical information about this.]

1.1.1 CLR data types

As the name suggests, the Common Type System (see Partition I of [1]) is a type system which has been designed with the aim of promoting inter-language operability. It is basically object-oriented, with the same model as Java: every class (with the exception of `Object`, the root class) has precisely one immediate superclass, but a class may implement multiple interfaces. The CTS also provides features such as unrestricted pointer arithmetic and non-heap-allocated structures, which are useful for implementation of other languages (C for instance). Some features of the CTS are implemented by explicit CIL instructions, while others are described by metadata. We will now attempt to describe how the type system appears within the CLR.

The CLR handles various primitive types. There are basic types whose meaning should be obvious: `bool`, (Unicode) `char`, `float32`, `float64`, `native float`, integer types `int8`, `int16`, `int32`, `int64`, and `native int`, together with `unsigned` variants. A native int is simply an integer type whose size is the wordsize of the host processor, and similarly with `native float`. The CLR actually only manipulates `int32`,`int64`,`native int` and `native float`; the other types are supported in the metadata, with implicit type-conversions being carried out before method calls. There is another primitive type, `typed reference`, instances of which consist of a managed pointer together with some representation of a type which may be stored at the location specified by the pointer. This is used to represented dynamically-typed data, and appears for example in the implementation of variable-length argument lists.

The types described above are examples of so-called *value types* (as opposed to *reference types* such as objects and arrays); this terminology isn't very striking, but try to remember it as it will reappear several times in the sequel. Value types also include *user-defined types*. These are *structures* and *enumerations*, which are similar to the identically named C constructs (although structures can contain methods, which is not the case in C). An important property of value types is that they are allocated "in place", and not on the heap. They can be passed by value as parameters, and can appear as method return values. The internal structure of user-defined types is specified in the metadata. In particular, the precise layout of fields within structures can be specified, and this can be used to implement unions by using overlapping fields.

Note that every instance of a value type (including structures) occupies a single stack location.

In addition to the value types there are several reference types , including (one-dimensional) arrays and function pointers. Also there are *object references*

(similar to the Java `reference` type), and *managed* and *unmanaged* pointers. A managed pointer is a pointer which points to an array element or object field; this can be used to implement argument passing by reference. An unmanaged pointer is simply an arbitrary memory address, and is usually represented by a `native unsigned int`. Arbitrary arithmetic operations are allowed on unmanaged pointers

Almost all dynamic memory allocation in the CLR is performed by creating a new object or array (the exception to this is the `localloc` instruction described below). Memory allocated by these means is garbage-collected; the CIL has no instruction for explicitly deallocating memory.

The CLR has several other features in common with the JVM. There is a built-in exception-handling mechanism using a `try ... catch ... finally` construct which is implemented partly in the instruction set and partly in the metadata. The CLR also allows bytecode to be multithreaded as in the JVM.

2 The CIL instruction set

2.1 Basic Instructions

These instructions provide basic computational capabilities. The mnemonics used are in the *ilasm* format used in [1]. The `.un` suffix attached to some instructions indicates that the instruction operates on unsigned values, while the `.ovf` suffix on some arithmetic instructions indicates that an exception will be thrown if overflow occurs (as with the JVM, the default strategy is to ignore overflow). Some instructions also have suffixes from the set { `.i1`, `.i2`, `.i4`, `.i8`, `.u1`, `.u2`, `.u4`, `.u8`, `.r4`, `.r8`, `.r.un`, `.i`, `.u`, `.ref` } indicating result types; we will usually denote such a family of instructions by an expression such as `ldelem.*` without going into full detail. Also, some instructions (such as load and store instructions) have short forms for accessing commonly-used values; again, we will not give details in these cases. Each instruction has a set of conditions which must be met in order for the instruction to be verifiable. For instance, the `stobj` instruction requires a pointer and a value type on the stack, and copies the value type into the memory location specified by the pointer. This will clearly be potentially unsafe if the pointer is unmanaged, for example. Correct code requires that the pointer be a managed pointer to a location containing an object of the same type as the value type on the stack; it can be deduced from the metadata whether or not this is the case. Consult [1] for details of verifiability conditions.

2.1.1 Branches

`beq, bge, bge.un, bgt, bgt.un, ble, ble.un, blt, blt.un, bne.un`
: Conditional branches. These compare pairs of integers and pairs of pointers, in various combinations. Managed pointers can also be compared with native integers, although this makes the code unverifiable. `beq` and `bne.un` can also be used for comparison of object references.

These instructions cannot be used to jump into or out of `try`, `catch` or `finally` blocks. Note that in the JVM, jumps are limited to 64 kilobytes. There is no limit on the length of branches in the CLR.

`br` Unconditional branch.

`brfalse, brtrue`
`brfalse` causes a branch if the value on the top of the stack is the integer value 0, or if it is a null pointer or object reference. `brtrue` has the opposite behaviour.

`call` This is used for calling a static method, instance method, or global function, where the destination address can be statically determined (ie, `call` implements method call via early binding). The destination is described by a metadata token (similar to a JVM method descriptor) embedded in the instruction stream; the arguments for the method call must be loaded onto the stack prior to invocation.

`calli` This is used for an indirect method call to native code. [This is what it says in the CLI specification, but the specification isn't very clear about function pointers. It appears that in fact you can have a pointer to any method; maybe the method has to have been JIT-compiled into native code first?] The stack contains the method arguments along with a pointer to a method entry point. The instruction stream contains a descriptor specifying the method signature.

`jmp` This appears to be used for an optimised form of tail call. Control is transferred to a method specified by a metadata token; the called method must have an identical signature to the calling method. The execution stack must be empty, and the arguments of the calling method are used as the arguments of the called method; this is presumably why method arguments are writable. The `jmp` instruction is never verifiable.

`ret` Return form the current method. If a value is to be returned, the value on the top of the stack is used. Apart from any return value, the execution stack for the current method must be empty (unlike the JVM, where the stack can contain an arbitrary amount of extra data, which will be discarded).

`switch` The switch instruction.

2.1.2 Comparisons

`ceq, cgt, cgt.un clt, clt.un`
Push the integer 0 or 1 onto the stack according to the result of the comparison. These instructions don't seem to offer much extra functionality over the branch instructions.

2.1.3 Arithmetic

`add, add.ovf, add.ovf.un`
`div, div.un`
`mul, mul.ovf, mul.ovf.un`
`neg`
`rem, rem.un`
`sub, sub.ovf, sub.ovf.un`
: These carry out the obvious arithmetic operations.

2.1.4 Logic

`and, not, or, shl, shr, shr.un, xor`
: Bitwise logical operations.

2.1.5 Store

`ldarg` Push the nth incoming argument onto the stack. (n is embedded in the instruction stream).

`ldarga` Push the address of nth incoming argument onto the stack. This enables parameter passing by reference.

`ldc.*` Push a numeric constant onto the stack.

`ldftn` Push a pointer to native code onto the stack. [As in the case of `calli` above, it seems that you can have a pointer to any method]

`ldind.*` Dereference a pointer and push the result onto the stack.

`ldloc` Push the nth local variable onto the stack.

`ldloca` Push the address of nth local variable onto the stack.

`ldnull` Push null reference.

`starg` Pop a value from the stack and store it in the nth argument slot.

`stind.*` Store a value in the address specified by a pointer (both arguments are on the stack).

`stloc` Pop a value from the stack and store it in the nth local variable.

2.1.6 Other

`arglist` This is used for the implementation of C's `varargs` construct.

`break` Breakpoint instruction for debugging.

`ckfinite`
: Check whether a floating-point number represents a finite value. If not, throw `ArithmeticException`.

`conv.*, conv.ovf.*, conv.ovf.*.un`
: A total of 33 instructions for type conversion.

`cpblk` Copy a block of memory to another location. This instruction is intended for copying structures. This instruction is never verifiable.

`dup` dup.

`endfilter`
: This appears to be used to mark the end of a `catch` block.

`endfinally`
: Mark the end of a `finally` block.

`initblk` Initialise a block of memory (usually a structure) to a particular value. This instruction is never verifiable.

`leave` Exit a `try`, `catch` or `finally` block.

`localloc`
: Each method has associated with it a *local dynamic memory pool*, which is freed on method exit; the `localloc` instruction allocates memory within this pool. The precise purpose of this pool is not clear; it may be required for allocating space for automatic variables whose size cannot be determined at compile time, while avoiding the overhead of garbage collection. This instruction is never verifiable.

`nop` nop.

`pop` pop.

2.2 Instructions for higher-level programming constructs

These instructions provide advanced capabilities for high-level languages.

`box` Convert a primitive value into an instance of class `Object`

`callvirt`
: Call a virtual method associated with an object. The point is that `callvirt` uses late binding: the method used is determined by the runtime type of the object

`castclass`
: Attempt to cast object to instance of class. `InvalidCastException` if the cast fails.

`cpobj` Copy an instance of a value type.

`initobj` Initialise an instance of a value type.

`isinst` Test whether object is an instance of a particular class or interface. If successful, perform a cast; otherwise return `NULL`.

`ldelem.*`
: Push array element onto stack.

`ldelema` Push address of array element onto stack.

`ldfld` Push object field onto stack.

`ldflda` Push address of object field onto stack.

`ldlen` Push array length.

`ldobj` Copy value type onto stack

`ldsfld` Push static object field onto stack.

`ldsflda` Push address of static object field onto stack.

`ldstr` Push literal string.

`ldtoken` Load runtime representation of metadata token; used for reflection.

`ldvirtftn`
: Load unmanaged pointer to native code implementing a virtual method. This can then be called using `calli`.

`mkrefany`
: Push a typed reference onto the stack; this supports the passing of dynamically typed references.

`newarr` Create one-dimensional array. (True multi-dimensional arrays (as opposed to arrays of arrays) can be created using `newobj` or methods from the .NET `Array` class.)

`newobj` Create a new object and initialise it. This instruction is simpler than the JVM version, since initialisation must be done in a separate step in the JVM.

`refanytype`
: Retrieve the type token from a typed reference; used in conjunction with `mkrefany`.

`refanyval`
: Retrieve the address from a typed reference; used in conjunction with `mkrefany`.

`rethrow` Rethrow exception.

`sizeof` Load size of value type.

`stelem.*`
: Store element in array.

`stfld` Store element in object field.

`stobj` Store value type in memory.

`stsfld` Store element in static field.

`throw` Throw an exception.

`unbox` Undo the `box` instruction.

2.3 Prefixes

The following instructions can be used to prefix other instructions in order to alter their behaviour.

`tail.` This can precede the `call`, `calli` or `callvirt` instructions; the evaluation stack must contain nothing except for the method arguments to the method call, and the method call must be followed by the `ret` instruction. The effect of the `tail.` prefix is to cause the current method's stack frame to be discarded prior to the method call, and this allows cross-jumping between mutually recursive functions without a large stack overhead. In certain circumstances (related to, among other things, security considerations and problems when exiting synchronised methods) the `tail.` instruction may be ignored.

`unaligned.`
This can only precede a `ldind`, `stind`, `ldfld`, `stfld ldobj stobj cpblk` or `initblk` instruction, and specifies that the address on the stack may not be properly aligned in memory.

`volatile.`
This can be combined with the `unaligned.` prefix, and may precede any of the same instructions. In addition, `volatile.` may precede `ldsfld` and `stsfld`. `volatile.` specifies that an address may be referenced from outside the current thread: thus multiple reads from this address may not be cached, and multiple writes cannot be suppressed.

3 Grail and .NET

3.1 Grail as a subset of .NET

It should be clear that the CIL instruction set contains a subset which is very similar to the JVM instruction set, and that most of the other instructions deal with the extra data types defined in the Common Type System.

We now present a set of CIL instructions which corresponds closely to the Grail subset of JVML defined in [4].

JVML instruction	CIL instruction
aconst_null	ldnull
sipush, ldc	ldc, ldstr
aload, iload	ldarg, ldloc
astore, istore	starg, stloc
dup	dup
nop	nop
iinc	ldloc, ldconst, add, stloc
iadd	add
isub	sub
imul	mul
if_icmpeq	beq
if_icmplt	blt
if_acmpeq	beq
goto	br
invokevirtual	callvirt
invokestatic	call or callvirt
invokespecial	newobj, call, callvirt
ireturn	ret
areturn	ret
return	ret
new	newobj
putfield	stfld
getfield	ldfld
putstatic	stsfld
getstatic	ldsfld
checkcast	castclass
instanceof	isinst

The possible extensions to Grail discussed in [4, §3.1] can also easily be expressed within the CIL framework. Floats would require no extra instructions, and CIL provides a set of array functions basically identical to the JVM ones:

JVML instruction	CIL instruction
newarray	newarr
anewarray	newarr
arraylength	ldlen
iastore	ldelem.i8
iaload	stelem.i8
aastore	ldelem.ref
aaload	stelem.ref

The inclusion of floats and arrays might be slightly more troublesome in CIL than in JVML. The JVML opcodes are explicitly typed, but one must examine

metadata and perhaps keep track of the types of stack elements to find out precisely what an `add` instruction (say) is supposed to do. This might or might not have implications for the complexity our cost model, for example.

3.2 Discussion

The functionality of the CIL subset above (which for brevity we will refer to as .Grail) is very similar to that of the Grail subset of JVML. It would probably be possible to write a program which would translate a compiled Grail class file into a .Grail assembly, and vice-versa. This task wouldn't be entirely straightforward. For example, care would have to be taken to separate the local variables in the Grail file into those which were method arguments and those which were genuinely local variables, and re-use of local variables for differently typed values would have to be resolved. Going in the other direction, a careful analysis would be required to determine the types of variables on the stack so that the correct JVM instructions could be selected.

The question of which platform to target will have to be decided at some point. It can be argued that the .NET CLR provides better support for language compilation than does the JVM; for example, the availability of tail calls and the boxing and unboxing operations might be useful for implementing a functional language, and CIL value types could ease the implementation of datatypes for such a language. On the other hand, there are many practical arguments in favour of the JVM. There are plenty of freely-available JVM implementations, but there is only one CLI implementation whose source-code is available. This is Rotor (see [9]), which is available under Microsoft's "shared-source" license. Rotor runs on Windows XP or FreeBSD, and consists of about 9,700 files and 1.9 million lines of code (see [3], or more generally [8]). In contrast the source code of the Java KVM (excluding API files) is available for Linux, and consists of 21 files and less than 24,000 lines of code.

Although the JVM may not be ideally-suited as a target for a functional language, the MLj project [7] demonstrates that the situation is far from hopeless. Furthermore, a great deal of work has been done on formalising the JVM and is available for us to make use of. In contrast, there appears to be only one paper which attempts to formalise part of the CLI (see [2]). The lack of formalisation might be regarded as an opportunity rather than an obstacle , but formalising the CLI would be a task not directly related to the present project. Furthermore, providing a semantics (for example) for CIL would be a non-trivial task as many of the properties of CIL (eg, the type-system) are mediated by the metadata and are not directly represented in the instruction set.

However, the table above demonstrates that Grail could essentially be executed in either environment. Perhaps a reasonable solution would be to make the original version of Grail slightly more abstract by means of minor modifications such as assuming that local variables and method arguments are stored in separate arrays. We could then obtain a bytecode language which could easily be translated into bytecode for either platform. This strategy would enable us to defer the choice of platform, or perhaps even to target both the JVM and

the .NET CLR simultaneously.

References

[1] ECMA. Standard ECMA-335: Common language infrastructure (CLI). See `http://www.ecma.ch`.

[2] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. Available from `http://research.microsoft.com/~adg/Publications/popl01.ps`, 2000.

[3] Brian Jepson. Get your rotor running. See `http://www.oreillynet.com/pub/a/dotnet/2002/03/27/gettingstarted.html`.

[4] K. MacKenzie. A virtual machine platform for resource-bounded computation. MRG internal report, LFCS, University of Edinburgh, February 2002.

[5] Erik Meijer and Jim Miller. Technical overview of the common language runtime (or why the JVM is not my favorite execution environment), 2001. Available at `http://docs.msdnaa.net/ark/Webfiles/whitepapers.htm`.

[6] Microsoft. .NET development. See `http://msdn.microsoft.com/net`.

[7] MLj. See `http://www.dcs.ed.ac.uk/~mlj/`.

[8] O'Reilly Network. See `http://www.oreillynet.com/dotnet/`.

[9] Rotor. See `http//msdn.microsoft.com/net/sscli/`.