16 Run-time Program Management

16.1.2 The Common Language Infrastructure

Work on the system that became the Common Language Infrastructure (CLI) began at Microsoft Corporation in the late 1990s, and was able to benefit from experience with Java and the JVM, which were already well established. The most significant differences between the virtual machines, however, stem from Microsoft's emphasis on cross-language interoperability—an emphasis that predates the JVM by many years.

Growing out of earlier work on the DDE, OLE, COM, ActiveX, and DCOM projects, the beta version of .NET was released in 2000. In addition to a virtual machine, it includes libraries, servers, and tools for a wide variety of local and distributed services, including user interface management, database access, networking, and security. A specification for the virtual machine—the CLI—was standardized by ECMA in 2001 and by the ISO in 2003. The standard has been updated several times over the years; version 6 was released in June 2012 [Int12a].

Perhaps the most significant contribution of the CLI is the definition of a Common Type System (CTS) for all supported languages. Encompassing nearly everything described in Chapters 8 and 10 of this book, the CTS provides a superset of what any particular language needs, while requiring common semantics and implementation wherever the type systems of more than one language intersect. In addition to the CTS, the CLI defines a virtual machine architecture, the VES (Virtual Execution System); an instruction set for that machine, the CIL (Common Intermediate Language); and a portable file format for code and metadata, PE (Portable Executable) assemblies.

C# is in some sense the premier language for .NET, and was developed concurrently with it. Several dozen languages have been ported to the CLI, however, and several of these, including Visual Basic, C++/CLI (formerly Managed C++), and F# (a descendant of OCaml) are now in widespread use.

Thanks to the ECMA/ISO standard, it is possible for organizations other than Microsoft to build implementations of the CLI. The leading such implementation is the open-source Mono project, led by Xamarin, Inc. (a Microsoft subsidiary). Mono runs on a wide variety of platforms, but tends to lag slightly behind .NET in

the addition of new features. Outside Microsoft, Java and the JVM still dominate. Within Microsoft, most new development today employs C#. Microsoft calls its CLI implementation the Common Language Runtime (CLR).

Architecture and Comparison to the JVM

In many ways, the CLI resembles the JVM. Both systems define a multithreaded, stack-based virtual machine, with built-in support for garbage collection, exceptions, virtual method dispatch, and interface inheritance. Both represent programs using a platform-independent, self-descriptive, bytecode notation. For languages like C#, the CLI provides all the safety of the JVM, including definite assignment, strong typing, and protection against overflow or underflow of the operand stack.

The biggest contrasts between the JVM and CLI stem from the latter's support for multiple programming languages (the following is not a comprehensive list).

Richer Type System The Common Type System (discussed below) supports both value and reference variables of structured types (the JVM is limited to references). The CTS also has true multidimensional arrays (allocated, contiguously, as a single operation); function pointers; explicit support for generics; and the ability to enforce structural type equivalence.

Richer Calling Mechanisms To facilitate the implementation of functional languages, the CLI provides explicit tail-recursive function calls (Section 6.6.1);

DESIGN & IMPLEMENTATION

16.7 Assuming a just-in-time compiler

Like the JVM, the CLI has behavior defined in terms of an abstract virtual machine. Where Java's virtual machine may in practice be either interpreted or just-in-time compiled, however, the CLI was designed from the outset for just-in-time compilation. Several minor differences between the virtual machines reflect this difference in expected implementations. Arithmetic instructions in Java bytecode generally include an explicit indication of operand type: there are, for example, four separate opcodes for 32- and 64-bit integer and floating-point addition. In the CLI's Common Intermediate Language (CIL), there is only one `add` instruction: it figures out what to do based on the types of its operands. In type-safe code, of course, the type of every operand is statically known, and either a compiler or an interpreter can inspect the types of arguments and figure out what to do. The compiler, however, only has to do this once, at compile time; the interpreter has to do it every time it encounters the instruction. In a similar vein, slots in the local variable array of the CLI VES can be of arbitrary size, and are required to hold a value of a single, statically known type throughout the execution of the method. For the sake of space efficiency and rapid indexing, the JVM reserves exactly 32 bits for every slot (`longs` and `doubles` take two consecutive slots), and a given slot can be used for values of different types at different points in time.

these discard the caller's frame while retaining the dynamic link. The CLI also supports both value and reference parameters, variable numbers of parameters (in the fully general sense of C), multiple return values, and nonvirtual methods, all of which the JVM lacks.

Unsafe Code For the benefit of C, C++, and other non-type-safe languages, the CLI supports explicitly unsafe operations: nonconverting type casts, dynamic allocation of non-garbage-collected memory, pointers to non-heap data, and pointer arithmetic. The CLI distinguishes explicitly between *verifiable* code, which cannot use these features, and *unverifiable* code, which can. (Verifiable code must also follow a host of other rules.)

Miscellaneous Again for the sake of multiple languages, the CLI supports global data and functions, local variables whose shapes and sizes are not statically known, optional detection of arithmetic overflow, and rich facilities for "scoped" security and access control.

As in the JVM, every CLI thread has a small set of base registers and a stack of method call frames, each of which contains an array of local variables and an operand stack for expression evaluation. Each frame also contains a local memory pool for variables of dynamic and elaboration-time shape. Incoming parameters have their own separate space in the CLI; in the JVM they occupy the first few slots of the local variable array.

The Common Type System

The VES and CIL provide instructions to manipulate data of certain built-in types. A few additional types are predefined, and have built-in names in CLI metadata. To these, the CTS adds a wide variety of type constructors. For each, it defines both behavior *and* representation. No single language provides all the types of the CTS, but (with occasional compromises) each provides a subset.

The Common Language Specification (CLS) defines a subset of the CTS intended for cross-language interaction. It omits several type constructors provided by the CTS, and places restrictions on others. Standard libraries (collection classes, XML, network support, reflection, extended numerics) restrict themselves (with occasional exceptions) to types in the CLS. Not all languages support the full CLS; code written in those languages cannot make use of library facilities that require unsupported types.

Built-in Types The VES and CIL provide instructions to manipulate the following types:

- Integers in 8-, 16-, 32-, and 64-bit lengths, both signed and unsigned
- "Native" integers, of the length supported by the underlying hardware, again both signed and unsigned
- IEEE floating point, both single and double precision
- Object references and "managed" pointers

Managed pointers are different from references: while typed, they don't necessarily point to the beginning of a dynamically created object. Specifically, they can refer to fields within an object or to data outside the heap. The CIL makes sure these pointers are known to the garbage collector, which must avoid reclaiming any object *O* when a managed pointer refers to a field inside *O*. More details on pointers and references can be found in Sidebar C-16.8.

Beyond the basic hardware-level types, CLI metadata treats Booleans, characters, and strings as built-ins. Booleans and characters are manipulated in the VES using instructions intended for short integers; strings are manipulated by accessing their internal structure.

Constructed Types To the built-in types, the CTS adds the following:

Dynamically allocated instances of class, interface, array, and delegate types. These are the things to which references (the built-in type) can refer. Arrays can be multidimensional, and are stored in row-major order. Delegates are closures (subroutine references paired with referencing environments).

Methods — function types.

Properties — getters and setters for objects.

Events — lists of delegates, associated with an object, that should be called in response to changes to the object.

Value types — records (structures), unions, and enumerations.

Boxed value types — values embedded in a dynamically allocated object so that one can create references to them.

Function pointers — references to static functions: type-safe, but without a referencing environment.

Typed references — pointers bundled together with a type descriptor, used for C-style variable argument lists.

Unmanaged pointers — as in C, these can point to just about anything, and support pointer arithmetic. They *cannot* point to garbage-collectible objects (or parts of objects) in the heap.

With these type constructors come extensive semantic rules, covering such topics as identity and equality,[1] casting and coercion, scoping and visibility, interface inheritance, hiding and overriding of members, memory layout, initialization, type safety, and verification. The details occupy hundreds of pages in the CLI documentation.

The Common Language Specification Because no single language implements the entire CTS, one cannot use arbitrary CTS types in a general-purpose interface intended for use from many different languages. The Common Language Specification (CLS) defines a subset of the CTS that most (though not all) languages

---

1 These are reminiscent of the relationships discussed in Sections 7.5 and 11.3.3.

can accommodate. Among other things, it omits several of the types provided by the CTS, including signed 8-bit integers; unsigned native, 16-, 32-, and 64-bit integers; boxed value types; global static fields and methods; unmanaged pointers; typed references; and methods with variable numbers and types of arguments. The CLS also imposes a variety of restrictions on the use of other types. It establishes naming conventions, limits the use of overloading, and defines the operators and conversions that programs can assume are supported on built-in types. It requires a lower bound of zero on each dimension of array indexing. It prohibits fields and static methods in interfaces. It insists that a constructor be called exactly once for each created object, and that each constructor begin with a call to a constructor of its base class. None of these restrictions applies to program components that operate only within a given language.

Generics As described in Section C-7.3.5, generics were added to Java and C# in very different ways. Partly to avoid the need to modify the JVM, Java generics were defined in terms of *type erasure*, which effectively converts all generic types to `Object` before generating bytecode. C# generics were defined in terms of *reification*, which creates a new concrete type every time a generic is instantiated with different arguments. Reified generics have been supported directly by the CLI since .NET

DESIGN & IMPLEMENTATION

16.8 References and pointers

The reference and pointer types of the CTS are a source of potential confusion. In a language like Java, reference types provide the only means of indirection. They refer to dynamically allocated instances of class, interface, and array types. Managed pointers provide additional functionality for languages like C# and Microsoft's C++/CLI, which permit references to the insides of objects and to values outside the CLI heap. Managed pointers are understood by the garbage collector, and can be used in type-safe code: If a managed pointer p refers to a field of object O, then the collector will know that O is live. It will also update p automatically whenever it moves O.

Unmanaged pointers exist for the sake of languages like C. They are incompatible with garbage collection, and cannot point to objects in the heap. They are also incompatible with type safety, and cannot be used in verifiable code.

Typed references (`typedrefs`) in the CLI include the information needed to correctly manipulate references to values (e.g., in variable argument lists) whose type cannot be statically determined.

Version 2.0 of the CLI introduced *controlled-mutability* managed pointers (also known, somewhat inaccurately, as *read-only* pointers). Operations on these pointers are constrained to prevent modification of the referenced object. Read-only pointers are used in boxing and array contexts where generics require the ability to generate a pointer to data of a value type, but modification of that data might not be safe.

version 2.0, introduced by Microsoft in 2005 and codified by ECMA and ISO in 2006.

EXAMPLE 16.39
Generics in the CLI and JVM

Reified generic types are fully described in CLI metadata, allowing full type checking and reflection. Consider the following code in C#:

```
class Node<T> {
    public T val;
    public Node<T> next;
}
...
Node<int> n = new Node<int>();
Console.WriteLine(n.GetType().ToString());
```

If `Node` is an outermost class, the final line will print `Node`1[System.Int32]`. The equivalent code in Java (running on the JVM) will simply print `class Node`. To support generics, CLI version 2 extended the rules for type compatibility and verification, and introduced new versions of several CIL instructions. ■

Metadata and Assemblies

Portable Executable (PE) *assemblies* are the rough equivalent of Java `.jar` files: they contain the code for a collection of CLI classes. PE is based on the Common Object File Format (COFF), originally developed for AT&T's System V Unix. It is the native object file format for Windows systems, extended to accommodate CIL as an optional instruction set. Given the requirements of native-code executable files (e.g., relocation—see Section 15.4), PE is quite a bit more complicated than Java `.class` and `.jar` format. A PE assembly contains a general-purpose PE header, a special CLI header, metadata describing the assembly's types and methods, and CIL code for the methods.

The metadata of an assembly has a complex internal structure. (A diagram of the interconnections among some two dozen different kinds of tables fills two pages of the annotated CLI standard [MR04, pp. 322–323].) The metadata begins with a *manifest* that specifies the files included and directly referenced, the types exported and imported, versioning information, and security permissions. This is followed by descriptions of all the types, and signatures for all the methods. Unlike the Java constant pool, the metadata of an assembly is not directly visible to the assembly's code; it may be rearranged by the JIT compiler in implementation-dependent ways, so long as it remains available to reflection routines at run time (obviously, those routines are also implementation dependent).

The Common Intermediate Language

Just as the CLI VES bears a strong resemblance to the JVM, CIL bears a strong resemblance to Java bytecode. Version 6 of the ECMA standard defines some 219 instructions, most with single-byte opcodes. Most instructions take their arguments from, and return results to, the operand stack of the current method frame. Others take explicit arguments representing variables, types, or methods.

Java bytecode and CIL are similarly dense—they require roughly the same number of bytes per instruction on average.

Many of the differences between the two intermediate languages are essentially trivial. Java bytecode is big-endian; CIL is little-endian. Java bytecode has explicit instructions for monitor entry and exit; these are method calls in the CLI. CIL allows arbitrary offsets for branches; Java bytecode limits them to 64K bytes.

A few more significant differences stem from the assumption that CIL will always be JIT-compiled, as described in Sidebar C-16.7. The most obvious difference here is that Java bytecode encodes type information explicitly in opcodes, while CIL requires it to be inferred from arguments. CIL also includes an explicit instruction (`ldtoken`) that will push a "run-time handle" for a method, type, or field. While the metadata of a CIL assembly must all be available at run time, its format may be implementation dependent; the JIT compiler translates `ldtoken` into machine code consistent with that format. In the JVM, the class file constant pool is assumed to be available at run time, in its standard format; an ordinary "load constant" instruction suffices to push the desired reference.

A more subtle difference is the separation of arguments from local variables in the CLI (they share one array in the JVM). Separate arrays admit special one-byte load instructions for both the first few arguments and the first local variables, without requiring that they have interleaved slots; this in turn may make it easier to generate object code in which arguments occupy contiguous locations in memory (as, for example, in the argument build area of the stack described in Section C-9.2.2).

Finally, as already suggested, several features of CIL, not found in Java bytecode, stem from the need to support multiple source languages. We have noted that the CLI provides value types, reference parameters, and optional overflow checking on arithmetic; all of these are reflected in the CIL instruction set. There are also several extra ways to make subroutine calls. Where Java bytecode supports only static, virtual, and dynamic method invocations, CIL has (1) nonvirtual method calls, as in C++ (these implicitly pass `this`, as virtual calls do); (2) indirect calls (i.e., calls through function pointers); (3) tail calls, which discard the caller's frame; and (4) *jumps*, which redirect control to a method after executing some optional prologue (e.g., for `this` pointer adjustment in languages with multiple inheritance; see Section C-10.6).

EXAMPLE 16.40
CIL for a list insert operation

To illustrate CIL, let us return to the linked-list set of Example **??**. The declarations given there are valid in both Java and C#. The `insert` method for this class appears in Figure C-16.7. C# source (which is again identical to the Java version) is on the left; a symbolic representation of the corresponding CIL is on the right. As in Example **??**, there are many examples of special one-byte load and store instructions (here specified with a *.index* suffix on the opcode), and of instructions that operate implicitly on the operand stack. ■

Verification As we have noted, the CLI distinguishes between *verifiable* and *unverifiable* code. Verifiable code must satisfy a large variety of constraints that guarantee type safety and catch many common programming errors. In particular,

```
public void insert(int v) {

    node n = head;

    while (n.next != null
           && n.next.val < v) {

        n = n.next;
    }
    if (n.next == null
        || n.next.val > v) {

        node t = new node();

        t.val = v;

        t.next = n.next;

        n.next = t;

    } // else v already in set
}
```

```
.method private hidebysig
       instance default void insert (int32 v)  cil managed
{
    // Method begins at RVA 0x2070      // RVA == relative
    // Code size 108 (0x6c)             //     virtual address
    .maxstack 3
    .locals init (
        class LLset/node    V_0,        // n
        class LLset/node    V_1)        // t
    IL_0000:  ldarg.0
    IL_0001:  ldfld class LLset/node LLset::head
    IL_0006:  stloc.0
    IL_0007:  br IL_0013        // jump to header of rotated loop
    IL_000c:  ldloc.0           // n -- beginning of loop body
    IL_000d:  ldfld class LLset/node LLset/node::next
    IL_0012:  stloc.0           // n = n.next
    IL_0013:  ldloc.0           // n -- beginning of loop test
    IL_0014:  ldfld class LLset/node LLset/node::next
    IL_0019:  brfalse IL_002f   // exit loop if n null
    IL_001e:  ldloc.0           // n
    IL_001f:  ldfld class LLset/node LLset/node::next
    IL_0024:  ldfld int32 LLset/node::val
    IL_0029:  ldarg.1           // v
    IL_002a:  blt IL_000c       // continue loop
    IL_002f:  ldloc.0           // n
    IL_0030:  ldfld class LLset/node LLset/node::next
    IL_0035:  brfalse IL_004b
    IL_003a:  ldloc.0           // n
    IL_003b:  ldfld class LLset/node LLset/node::next
    IL_0040:  ldfld int32 LLset/node::val
    IL_0045:  ldarg.1           // v
    IL_0046:  ble IL_006b
    IL_004b:  newobj instance void class LLset/node::'.ctor'()
    IL_0050:  stloc.1           // t
    IL_0051:  ldloc.1           // t
    IL_0052:  ldarg.1           // v
    IL_0053:  stfld int32 LLset/node::val
    IL_0058:  ldloc.1           // t
    IL_0059:  ldloc.0           // n
    IL_005a:  ldfld class LLset/node LLset/node::next
    IL_005f:  stfld class LLset/node LLset/node::next
    IL_0064:  ldloc.0           // n
    IL_0065:  ldloc.1           // t
    IL_0066:  stfld class LLset/node LLset/node::next
    IL_006b:  ret
} // end of method LLset::insert
```

Figure 16.7 C# source and CIL for a list insertion method. Output on the right was produced by the Mono project's `mcs` (compiler) and `monodis` (disassembler) tools, with additional comments inserted by hand. Note that the compiler has rotated the test to the bottom of the `while` loop, which occupies lines `IL_000c` through `IL_002a` in the output code.

the VES can be sure that a verifiable program will never access data outside its logical address space. Among other things, this guarantee ensures fault containment for verifiable modules that share a single physical address space.

Unverifiable code can make use of unsafe language features (e.g., unions and pointer arithmetic in C), but must still conform to more basic rules for validity (well-formedness) of CIL. Together, the components of the VES (i.e., the JIT compiler, loader, and run-time libraries) *validate* all loaded assemblies, and *verify* those that claim to be verifiable. Any standard-conforming implementation of the CLI must run all verifiable programs. Optionally, it may also run validated but not verifiable programs.

As in the JVM, verification requires data flow analysis to check type consistency and lack of underflow and overflow in the operand stack. The CLI standard requires verifiable routines to specify that all local variables are initialized to zero. CLI implementations typically perform definite assignment data flow analysis anyway, to identify cases in which those initializations can safely be omitted. The standard also requires numerous checks on individual instructions. Many of these are also performed by the JVM. Local variable references, for example, are statically checked to make sure they lie within the declared bounds of the stack frame. Other checks stem from the presence of unsafe features in the CLI. Verifiable code cannot use unmanaged pointers or unions, for example, nor can it perform most indirect method calls.

✓ CHECK YOUR UNDERSTANDING

38. Summarize the architecture of the Common Language Infrastructure. Contrast it with the JVM. Highlight those features intended to facilitate cross-language interoperability.

39. Describe how the choice of just-in-time compilation (and the rejection of interpretation) influenced the structure of the CLI.

40. Describe several different kinds of references supported by the CLI. Why are there so many?

41. What is the purpose of the Common Language Specification? Why is it only a subset of the Common Type System?

42. Describe the CLI's support for *unsafe* code. How can this support be reconciled with the need for safety in embedded settings?

16 Run-time Program Management

16.5 Exercises

16.14 Using Oracle's `jaotc` compiler and `mono --aot`, compile the code of Figures 16.2 and C-16.7 all the way to machine language. Disassemble and compare the results. Can all the differences be attributed to variations in the quality of the compilers, or are any reflective of more fundamental differences between the source languages or virtual machines?

16.15 Rewrite the list insertion method of Example C-16.40 in F# instead of C#. Compile to CIL and compare to the right side of Figure C-16.7. Discuss any differences you find.

16.16 Building on the previous exercise, rewrite your list insertion routine (both C# and F# versions) to be generic in the type of the list elements. Compare the generic and nongeneric versions of the resulting CIL and discuss the differences.

16.17 Extend your F# code from Exercise C-16.16 to include list removal and search routines. After finding and reading appropriate documentation, package these routines in a library that can be called in a natural way not only from F# but also from C#.

16 Run-time Program Management

16.6 Explorations

16.26 Learn the details of the CLI verification algorithm (Partition III, Section 1.8 of the ECMA standard, version 4 [Int12a]). Pay particular attention to the rules for *merging* compatible types at joins in the control flow graph, and for dealing with generics.

16.27 Learn more about the .NET Language-Integrated Query mechanism (LINQ), mentioned in Example 16.29. Discuss its use of attributes. Write a program that uses it to interface to a database through SQL. Write another program that uses it to process the elements of a set from the `System.Collections` library.

16.28 Like most scripting languages, Perl 5 compiles its input to an internal syntax tree format, which it then interprets. Explore this implementation, and characterize the circumstances under which the interpreter may need to call back into the compiler during execution. Also explore the `perlcc` command-line script (itself written in Perl), which translates source code to either bytecode or machine code.

In several cases, the interpreter may need to call back into the compiler during execution. Features that force such dynamic compilation include `eval`, which compiles and then interprets a string; `require`, which loads a library package; and the `ee` version of the substitution command, which performs expression evaluation on the replacement string:

```
$foo = "abc";
$foo =~ s/b/2 + 3/ee;       # replace b with the value of 2 + 3
print "$foo\n";             # prints a5c
```

Perl can also be directed, via library calls or the `perlcc` command-line script (itself written in Perl), to translate source code to either bytecode or machine code. In the former case, the output is an "executable" file

beginning with `#! /usr/bin/perl` (see Sidebar 14.4 for a discussion of the `#!` convention). If invoked from the shell, this file will feed itself back into Perl 5, which will notice that the rest of the file contains bytecode instead of source, and will perform a quick reconstruction of the syntax tree, ready for interpretation.

If directed to produce machine code, `perlcc` generates a C program, which it then runs through the C compiler. The C program builds an appropriate syntax tree and passes it directly to the Perl interpreter, bypassing both the compiler and the byte-code-to-syntax-tree reconstruction. Both the bytecode and machine code back ends are considered experimental; they do not work for all programs.