Understanding CIL

James Crowley

Developer Fusion

http://www.developerfusion.co.uk/

Overview

- Generating and understanding CIL
- De-compiling CIL
- Protecting against de-compilation
- Merging assemblies

Common Language Runtime (CLR)

- Core component of the .NET Framework on which everything else is built.
- A runtime environment which provides
  - A unified type system
  - Metadata
  - **Execution engine**, that deals with programs written in a Common Intermediate Language (CIL)

Common Intermediate Language

- All compilers targeting the CLR translate their source code into CIL
- A kind of assembly language for an abstract stack-based machine, but is not specific to any hardware architecture
- Includes instructions specifically designed to support object-oriented concepts

Platform Independence

- The intermediate language is not interpreted, but is not platform specific.
- The CLR uses JIT (Just-in-time) compilation to translate the CIL into native code
- Applications compiled in .NET can be moved to any machine, providing there is a CLR implementation for it (Mono, SSCLI etc)

Demo

- Generating IL using the C# compiler

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       31 (0x1f)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  ldc.i4.s   50
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   20
  IL_0005:  stloc.1
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  call       int32 ILDemo.Demo::Remainder(int32,
                                                    int32)
  IL_000d:  stloc.2
  IL_000e:  ldstr      "Remainder is: {0}"
  IL_0013:  ldloc.2
  IL_0014:  box        [mscorlib]System.Int32
  IL_0019:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                object)
  IL_001e:  ret
} // end of method Demo::Main
```

Some familiar keywords with some additions:

.method – this is a method
hidebysig – the method hides other methods with the same name and signature.
cil managed – written in CIL and should be executed by the execution engine (C++ allows portions that are not)

```
.method private hidebysig static void  Main(string[] args) cil managed
{
 .entrypoint
 // Code size       31 (0x1f)
 .maxstack  2
 .locals init (int32 V_0,
          int32 V_1,
          int32 V_2)
 IL_0000:  ldc.i4.s   50
 IL_0002:  stloc.0
 IL_0003:  ldc.i4.s   20
 IL_0005:  stloc.1
 IL_0006:  ldloc.0
 IL_0007:  ldloc.1
 IL_0008:  call       int32 ILDemo.Demo::Remainder(int32,
                                                   int32)
 IL_000d:  stloc.2
 IL_000e:  ldstr      "Remainder is: {0}"
 IL_0013:  ldloc.2
 IL_0014:  box        [mscorlib]System.Int32
 IL_0019:  call       void [mscorlib]System.Console::WriteLine(string,
                                                               object)
 IL_001e:  ret
} // end of method Demo::Main
```

.entrypoint – the program’s entry point
.maxstack 2 – specifies maximum depth of the stack at any point during execution
.locals – defines storage locations for variables local to this method, with new names V_0, V_1, V_2 (replacing a, b, result)

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       31 (0x1f)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  ldc.i4.s   50
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   20
  IL_0005:  stloc.1
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  call       int32 ILDemo.Demo::Remainder(int32,
                                                   int32)
  IL_000d:  stloc.2
  IL_000e:  ldstr      "Remainder is: {0}"
  IL_0013:  ldloc.2
  IL_0014:  box        [mscorlib]System.Int32
  IL_0019:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                 object)
  IL_001e:  ret
} // end of method Demo::Main
```

ldc.i4.s – loads the 4-byte integer constant “50” onto the stack (“s” defines some additional behaviours to keep the number of op-codes down)

stloc.0 – takes the top value on the stack (ie 50) and stores it in the local variable at index 0 (ie V_0, or “a” with our original naming)

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       31 (0x1f)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  ldc.i4.s   50
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   20
  IL_0005:  stloc.1
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  call       int32 ILDemo.Demo::Remainder(int32,
                                                    int32)
  IL_000d:  stloc.2
  IL_000e:  ldstr      "Remainder is: {0}"
  IL_0013:  ldloc.2
  IL_0014:  box        [mscorlib]System.Int32
  IL_0019:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                object)
  IL_001e:  ret
} // end of method Demo::Main
```

ldloc.0 and **ldloc.1** – loads the value of the local variable at index 0 (“a”) and index 1 (“b”) onto the stack

call – makes a call to our Remainder method. The two arguments are popped off the stack during the call, and we get the result of the method execution pushed back on.

Stloc.2 – store the result (which is at the top of the stack) in local variable at index 2 (“result”)

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       31 (0x1f)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  ldc.i4.s   50
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   20
  IL_0005:  stloc.1
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  call       int32 ILDemo.Demo::Remainder(int32,
                                                      int32)
  IL_000d:  stloc.2
  IL_000e:  ldstr      "Remainder is: {0}"
  IL_0013:  ldloc.2
  IL_0014:  box        [mscorlib]System.Int32
  IL_0019:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                object)
  IL_001e:  ret
} // end of method Demo::Main
```

ldstr – loads the string constant onto a stack
ldloc.2 – loads the variable “result” onto the stack
box – turns the “result” variable (a value type) into an object (reference type)
call – makes the call to WriteLine
ret – returns execution to the callee

Things to note…

- Optimisation largely occurs at the JIT compilation stage, rather than when we are generating IL – so that all languages targeting the CLR can benefit.
- The underlying IL contains all the info required to reconstruct your original source code (minus comments and variable names)

Demo

- Decompilation using .NET Reflector

De-compilation & Obfuscation

- Can’t easily prevent code from being decompiled, but we can make it harder to “understand” the intention of the code.
- Various techniques, including
  - variable renaming
  - control flow obfuscation
  - string encryption

Obfuscation Software

- PreEmptive - Dotfuscator (basic community edition included in VS 2003)
- Remotesoft Obfuscator
- WiseOwl - Demeanor for .NET

Merging Assemblies

- We can combine the IL of multiple assemblies to combine assemblies, without access to the original source code
- For example, merging a required COM interop wrapper into our main assembly.

ILMerge

- Utility from Microsoft Research that automatically merges the IL and re-compiles the assembly.

Demo

- Merging Assemblies

Wrapping Up

- Any questions?

Why do we care?

- Decompilation
  - The underlying IL contains all the info required to reconstruct your original source code (minus comments and variable names)
  - .NET Reflector
  - ILDASM/ILASM
- Merging multiple assemblies
  - We can merge assemblies by merging their IL (ILMerge)
- New Languages
  - We can implement new .NET languages provided we can emit the correct IL