Emitting binary assemblies (`GenBinMSIL`) via `IKVM.Reflection`

© Miguel Garcia, LAMP, EPFL
`http://lamp.epfl.ch/~magarcia`

October 28th, 2011

Abstract

These notes cover the differences between `GenMSIL` and `GenBinMSIL`. Both emit *semantically* the same information (a .Net assembly) however in different formats (textual ILAsm vs. binary).

- A big plus of `IKVM.Reflection` is that it doesn't depend on any of the remaining IKVM libraries (it just depends on `System.dll` and `mscorlib.dll`).
- The cross-compiler will retain its dependency on `ch.epfl.lamp.compiler.msil`, but the bootstrapped compiler can have it removed (once `TypeParser` is updated to use `IKVM.Reflection`) For now, given that our custom library is working fine, it remains in use in `TypeParser` in the bootstrapped compiler too.

Finally, these notes cover API-diffing tools and future work (emtiting debug-mode assemblies).

Contents

1 Differences with GenMSIL

1.1 Bug fixes to make peverify happy

In GenICode:

```
  else (from, to) match {
    case (BYTE, LONG) | (SHORT, LONG) | (CHAR, LONG) | (INT, LONG) => ctx.bb.emit(CALL_PRIMITIVE(Conversion(INT,
    case (REFERENCE(clsFrom), REFERENCE(clsTo))
    if(forMSIL && loaders.clrTypes.isValueType(clsFrom) && (clsTo eq definitions.ObjectClass)) =>
      ctx.bb.emit(BOX(from))
    case _ => ()
  }
```

1.2 TODO: Pending type discovery of unsigned integers

In GenBinMSIL:

```
def loadI4(value: Int, code: ILGenerator): Unit = value match {
  case -1 => code.Emit(OpCodes.Ldc_I4_M1)
  case 0 => code.Emit(OpCodes.Ldc_I4_0)
  case 1 => code.Emit(OpCodes.Ldc_I4_1)
  case 2 => code.Emit(OpCodes.Ldc_I4_2)
  case 3 => code.Emit(OpCodes.Ldc_I4_3)
  case 4 => code.Emit(OpCodes.Ldc_I4_4)
  case 5 => code.Emit(OpCodes.Ldc_I4_5)
  case 6 => code.Emit(OpCodes.Ldc_I4_6)
  case 7 => code.Emit(OpCodes.Ldc_I4_7)
  case 8 => code.Emit(OpCodes.Ldc_I4_8)
  case _ =>
    if (value >= -128 && value <= 127) {
      /*- (OpCodes.Ldc_I4_S, value.toByte) would be the right thing to do
          but right now TypeParser maps both System.Byte and System.SByte to scala.Byte */
      code.Emit(OpCodes.Ldc_I4, value)
    } else {
      code.Emit(OpCodes.Ldc_I4, value)
    }
}
```

1.3 TODO: More complete assembly initialization

Quoting from Monos's mcs:

```
       const int IMAGE_SUBSYSTEM_WINDOWS_GUI = 2;
       const int IMAGE_SUBSYSTEM_WINDOWS_CUI = 3;
       PEFileKinds fileKind;
       switch (inputAssembly.ManifestModule.__Subsystem)
       {
              case IMAGE_SUBSYSTEM_WINDOWS_GUI:
                     fileKind = PEFileKinds.WindowApplication;
                     break;
              case IMAGE_SUBSYSTEM_WINDOWS_CUI:
              default:
                     fileKind = PEFileKinds.ConsoleApplication;
                     break;
       }
       ab.SetEntryPoint((MethodBuilder)ResolveMethod(inputAssembly.EntryPoint), fileKind);
```

```
PortableExecutableKinds peKind;
ImageFileMachine machine;
inputAssembly.ManifestModule.GetPEKind(out peKind, out machine);
ab.Save(outputFile, peKind, machine);
```

2 Write your own disassembler

Before bootstrapping, the cross-compiler's `ILPrinterVisitor` was the only means to emit an assembly, in the form of textual ILAsm to be compiled with `ilasm.exe`. The `forJVM` counterpart to that functionality is `-Ygen-javap`:

```
val Ygenjavap    = StringSetting   ("-Ygen-javap", "dir",
                                    "Generate a parallel output directory of .javap files.", "")
```

With `GenBinMSIL`, the same effect can be achieved by disassembling the output (binary) assembly. This is *sometimes* useful (e.g., to compare the bytecode emitted by different versions of the compiler).

In case one insists on having Scala.Net emit *all by itself* the venerable textual ILAsm files, then basically we have to include a disassembler in Scala.Net (alternatives listed below). In all cases, IKVM.Reflection should be used, which is up to the task. Quoting from the IKVM weblog:

> *It should now be possible to write an ILDASM clone using IKVM.Reflection, however to be able to write ILASM there are still a couple of things missing: (a) function pointers (used by C++/CLI); (b) API to create missing type, (c) preserving interleaved modopt/modreq ordering; (d) various C++/CLI quirks (e.g. custom modifiers on local variable signatures); (e) ability to set file alignment.*

As of these writing, some alternatives are:

- adapt `ch.epfl.lamp.compiler.msil.emit.ILPrinterVisitor` to use the `IKVM.Reflection` API. Currently `ILPrinterVisitor` uses custom data structures (non-`Reflection`-based)
- Port an existing disassembler (most of them are based on `Mono.Cecil`):
  - ILSpy's disassembler (it also appears to be Monodevelop's disassembler), `https://github.com/icsharpcode/ILSpy/tree/master/ICSharpCode.Decompiler/Disassembler`
  - Zor's disassembler, `http://xtzgzorex.wordpress.com/2011/08/23/gsoc-and-the-state-of-ilasm/`
  - An old version by JB Evain, `https://github.com/mono/cecil/tree/master/ildasm`

Other resources:

- Reflection and Generic types, `http://msdn.microsoft.com/en-us/library/ms172334.aspx`
- CodeDOM MSIL Code Provider, `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=4866#overview`

3 API-diffing

The output of `GenMSIL` and `GenBinMSIL` was diffed to confirm they emit assemblies exposing the same public API.

3.1 `ApiChange`

`ApiChange` does a great job, without the complications of other tools (Sec. 3.2 and Sec. 3.3). Usage is simply:

```
ApiChange.exe -new x:\Input\GenB\scalalib.dll -old x:\Input\GenM\scalalib.dll -diff
```

3.2 LibCheck

`LibCheck` is geared towards detecting API-differences across versions of the .NET framework. As to comparing different versions of individual assemblies, blogs are littered with horror stories about inescrutable command-line parameters. I agree with those. An older version of LibCheck is covered in detail at `http://www.oreilly.de/catalog/9780596527549/chapter/ch04.pdf`.

Quoting from `http://mattonsoftware.com/archive/2006/09/28/21.aspx`:

> *Step 1: Generating Metadata for Comparison*
>
> ```
> libcheck.exe -store AssemblyName FolderNameToStoreResults -full FolderContainingAssembly
> ```
>
> *For example:*
>
> ```
> libcheck.exe -store HelloWorld.dll 1.0.0.0 -full C:\HelloWorldV1\
>
> libcheck.exe -store HelloWorld.dll 2.0.0.0 -full C:\HelloWorldV2\
> ```
>
> *Step 2: Performing the Comparison*
>
> ```
> libcheck.exe -compare FolderContainingFirstStoreResults FolderContainingSecondStoreResults
> ```
>
> *For example:* `libcheck.exe -compare 1.0.0.0 2.0.0.0`

3.3 Framework Design Studio

- Framework Design Studio is the successor to LibCheck. It includes a GUI as well as command-line tool: `http://code.msdn.microsoft.com/fds`
- The author's comments on it: `http://blogs.msdn.com/b/kcwalina/archive/2008/04/04/8357773.aspx`

4 Future Work: Emitting debug-mode assemblies (A story about `csc.exe` optimizations and CLR JITing)

One of the lessons learnt from `GenBinMSIL` is that the `Emit` functionality of `IKVM.Reflection` can be invoked directly from `GenICode` (because `Reflection.Emit` provides API for things like "`BeginExceptionBlock()`", "`BeginScope()`", and so on).

Debug-mode assemblies refers to assemblies that .Net compilers emit when given the `/debug` option. Structurally, they are .NET assemblies all right. Unlike their Release-mode counterparts, only simple optimizations were applied. Emitting debug-mode assemblies (where only simple optimizations are applied) results in shorter edit-compile-debug cycles. Guaranteed.

Which brings us to the topic of CLR optimizations. In most cases, a debug-mode assembly will run after JIT-ing just as fast as a release-mode one because of JIT optimizations. Quoting from `http://blogs.msdn.com/b/davidnotario/archive/2004/10/26/247792.aspx`:

> *3. Flowgraph analysis: The JIT performs a traditional flowgraph analysis, to determine the liveness of variables and gen/kill sets, dominator information, loop detection, etc.... This information is used in all subsequent stages.*
>
> *4. Optimization phase: In this stage, the heavyweight optimizations happen: Common Subexpression and Range Check Elimination, loop hoisting, etc...*

Other resources on CLR JIT'ing:

- `http://www.codeproject.com/KB/dotnet/JITOptimizations.aspx`
- `http://blogs.msdn.com/b/davbr/`

The definitive source of information on `csc.exe` optimizations is Eric Lippert's blog entry "*What does the optimize switch do?*" at `http://blogs.msdn.com/b/ericlippert/archive/2009/06/11/what-does-the-optimize-switch-do.aspx`.

All right, a caveat on "JITing can remedy the lack of advanced optimizations". That's true for desktop and server VMs. However in resource-constrained environments (mobile devices) the JITer is way less aggressive.

A Appendix: Limitations of `System.Reflection.Emit`

Quoting from the examples and discussion at: `http://blogs.msdn.com/b/lucian/archive/2009/11/29/the-limitations-of-reflection-emit.aspx`:

> *We're porting the VB compiler from C++ into VB, and we wanted to know if we could use Reflection.Emit for the back-end (answer "no") ... we're exploring the idea of a "REPL loop" and want to know how best to implement it (no solid answers yet).*

Lessons learnt from migrating `vbnc` (Mono's Visual Basic compiler) from `S.R.E` to `Mono.Cecil`: `http://rolfkvinge.blogspot.com/2010/06/hackweeek.html`. BTW, Cecil is released under the MIT/X11 license.

> *This makes it possible to fix a few long standing bugs, but most importantly (to me at least, the bugs are quite corner-case), a lot of unnecessary code has been deleted (˜10k fewer lines of code in the compiler). Another advantage is that it's now trivial to add support for compiling to different runtime versions - `vbnc` won't ever suffer from multiple personalities like the mono's C# compiler does: `mcs/gmcs/smcs/dmcs ...`*

Other disadvantages of `S.R.E`:

- not all .NET platforms support it, e.g. the Compact Framework doesn't.
- Dreaded "bug by design". Somewhat outdated but worth checking: `http://nemerle.org/Runtime_issues`
- less than ideal in connection wih generics: `http://joltdev.blogspot.com/2008/10/frustration-with-reflectionemit-api.html`
- Problem with generics and `DefineMethodOverride`. Fixed? `http://connect.microsoft.com/VisualStudio/feedback/details/97425/problem-with-generics-and-definemet`

Advantages of `S.R.E` (shared with other approaches):

- detailed walkthrough of `S.R.E` debugging support, `http://blogs.msdn.com/b/jmstall/archive/2005/02/03/366429.aspx`