

The .NET backend of the Scala compiler

© Miguel Garcia, STS, Hamburg University of Technology
<http://www.sts.tu-harburg.de/people/mi.garcia>

March 14, 2010

Contents

1	IDE-based debugging of the cross-compiler	2
1.1	Configuration	2
1.2	Debug session	2
2	From ICode to CIL	4
2.1	Platform-specific elements	4
2.2	How GenMSIL works	5
2.3	Unparsing CIL ASTs to disk	6
3	Generating PDB file(s) for debug	6
3.1	Generation with <code>ilasm</code>	7
3.2	Generation with <code>Reflection.Emit</code>	7
3.3	Generation with CCI	8
4	Quality of the generated code	8
4.1	Emitting for interop: CLS and Design Guidelines	8
4.2	<code>PEVerify</code>	9
4.3	<code>FxCop</code>	9
4.4	Static verification of implicit preconditions	10
4.5	Verification of compiler backend vs. Validation of translation results	10
5	Battle of the CIL-emit libraries (Round 2)	11
5.1	<code>System.Reflection</code> : too eager type resolving	11
5.2	CCI documentation	11
6	Link Sink	11

Abstract

These notes present a tour of the Scala.Net cross-compiler, exploring aspects relevant from both self-hosting and cross-compilation perspectives. A more detailed look at type decoding and CIL generation is pending (these notes provide overviews only) as both topics qualify for dedicated write-ups.

1 IDE-based debugging of the cross-compiler

1.1 Configuration

After downloading the sources of the Scala.Net cross-compiler from its SVN repo¹ (Figure 1), an IntelliJ project can be created to hold them. Some notes about the layout of folders:

- “dotnet-library” is the part of the scala library that compiles successfully using `-target:msil`, i.e. the part that produces `predef.dll` after a run of `ant msil`. It is a branch of the “library” folder, and there’s a nightly script merging all changes from “library” to “dotnet-library”. This library is stand-alone, depending only on `mcorlib.dll`.
- “dotnet-compiler” is a branch of “compiler” also with automatic merges. The sources in “dotnet-compiler” are meant to be the sources for the native compiler (as opposed to the current cross-compilation scheme)
- the source files of the cross-compiler are the files in the “compiler” folder, they are identical with the ones from trunk. So in fact, the trunk compiler *is* a functional cross-compiler, too!

The folders I picked to become source folders are shown in blue in Figure 2. Just one lump-everything IntelliJ module was set-up (creatively called “Msil”) (the other modules contain the sources of `jline`, need not be there and can be ignored). Finally, a dependency was added on a library created to contain all jars in the `lib` folder of the SVN checkout.

This `lib` folder also contains files `mcorlib.dll` and `scalaruntime.dll`. We will need those files (together with `predef.dll`, that can be obtained from the nightly MSIL distribution of Scala²), in order to run the executables produced after cross-compilation.

We’re ready to run the cross-compiler in debug mode!

1.2 Debug session

The minimal debug configuration in IntelliJ (to compile HelloWorld) reads:

- Main class: `scala.tools.nsc.Main`
- Program parameters: `-target:msil -Xassem-extdirs D:\w\m\mscor HelloWorld.scala`
- Use classpath and JDK of module: the lump-everything module
- `D:\w\m\mscor` above is a folder containing the minimum of assemblies where the compiler will look for external types (i.e., containing `mcorlib.dll`, `predef.dll`, and `scalaruntime.dll`).

You may already set breakpoints and watch how Listing 1 gets compiled. Better yet, you can remove one or more of the `dlls` listed under `-Xassem-extdirs` and see where an exception is thrown. It has to do with type decoding³.

¹<http://lampsvn.epfl.ch/svn-repos/scala/scala-msil>

²<http://www.scala-lang.org/node/212/distributions-msil>

³<http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/TypeDecoding.pdf>

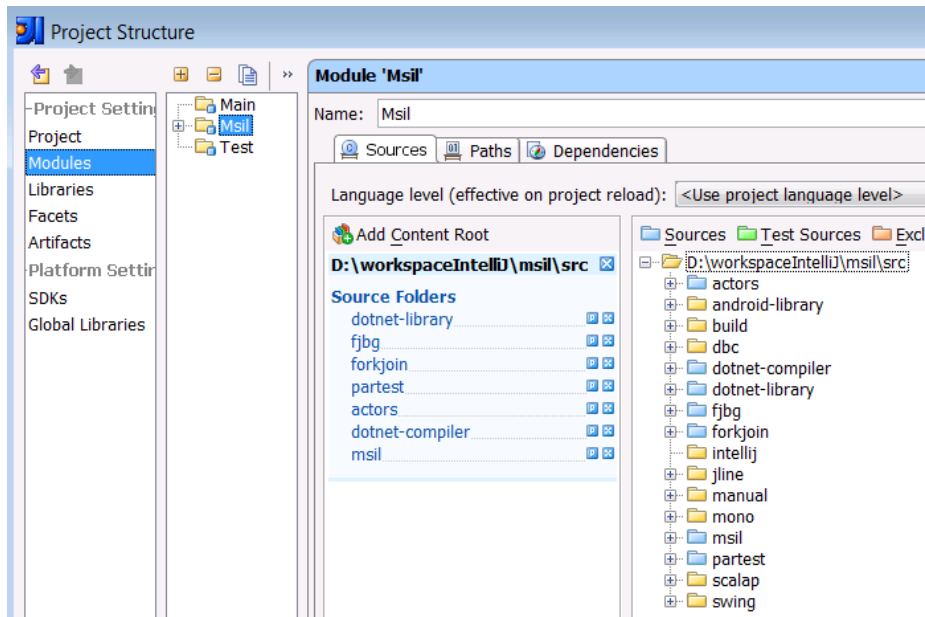


Figure 2: Pickings source folders

Listing 1: You asked for it: Hello World

```
import System.Console

object test extends Application {
  Console.WriteLine("Hello world!")
}
```

After a successful run of the compiler on HelloWorld, two files are generated: HelloWorld.msil and test.msil. After *opening a Visual Studio prompt* they can be compiled into an .exe with `ilasm HelloWorld.msil test.msil`. Want to read ILAsm? There's a book on it [1].

2 From ICode to CIL

When I started writing these notes I planned to cover this topic in one section :-). Now I realize more space is needed, that's why follow-up notes are planned. An introduction is provided in this section.

TODO GenJVM invokes `dce` (if enabled on the command-line) before generation proper starts. GenMSIL should do the same, right?

2.1 Platform-specific elements

Before getting to GenMSIL proper, its connection with Global appears in Figure 3, i.e., the MSILPlatform class. In addition to the genMSIL object, MSILPlatform provides .NET specialized versions of `classPath`, `rootLoader` (for type decoding),

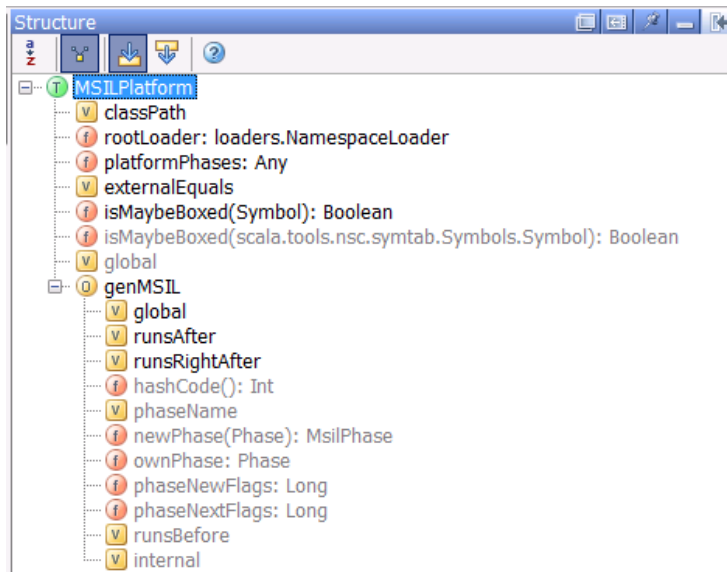


Figure 3: MSILPlatform

and platform phases (only `genMSIL` in comparison with Java's `flatten`, `liftcode`, and `genJVM`). Also the platform-specific `Symbol` for `equals` is accessible through `MSILPlatform` (and the `isMaybeBoxed` utility).

Finally, in `Global` we find:

```
// platform specific elements

type ThisPlatform = Platform[_] { val global: Global.this.type }

lazy val platform: ThisPlatform =
  if (forMSIL) new { val global: Global.this.type = Global.this } with MSILPlatform
  else new { val global: Global.this.type = Global.this } with JavaPlatform

def classPath: ClassPath[_] = platform.classPath
def rootLoader: LazyType = platform.rootLoader
```

2.2 How GenMSIL works

The `ch.epfl.lamp.compiler.msil` library is used by `GenMSIL` to build CIL ASTs for incoming `ICode` ASTs and to unparse the thus generated CIL ASTs into `.il` text files (Sec. 2.3).

This library is based on the ECMA-335 Standard: Common Language Infrastructure (CLI), 4th edition (June 2006). PDFs for CLR standards can be found at: <http://msdn.microsoft.com/en-us/netframework/aa569283.aspx>.

`GenMSIL` lexically comprises two classes: `MsilPhase` (stretching from lines 33 to 63) and `BytecodeGenerator` (from 68 to a whopping 2224).

TODO More detailed description.

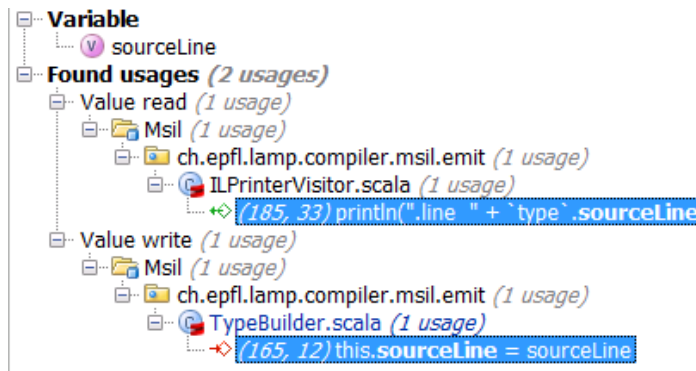


Figure 4: How GenMSIL marks source locations for debug

2.3 Unparsing CIL ASTs to disk

This functionality is supported by `ILPrinterVisitor`'s companion object. The class has two subclasses: `SingleFileILPrinterVisitor` (prints a complete assembly in a single file), and `MultipleFilesILPrinterVisitor` (prints a complete assembly into as many files as `TODO` in the assembly).

Unparsing (of the multi-file variety) is invoked from `AssemblyBuilder.Save()` (in turn invoked from `BytecodeGenerator.writeAssembly()`), i.e. the second of the following methods gets invoked:

```
@throws(classOf [IOException])
def printAssembly(assemblyBuilder: AssemblyBuilder, fileName: String) {
  assemblyBuilder.apply(new SingleFileILPrinterVisitor(fileName))
}

@throws(classOf [IOException])
def printAssembly(assemblyBuilder: AssemblyBuilder, destPath: String, sourceFilesPath: String) {
  assemblyBuilder.apply(new MultipleFilesILPrinterVisitor(destPath, sourceFilesPath))
}
```

3 Generating PDB file(s) for debug

The C# debugger highlights a *text range* with each debug step, thus giving better feedback when debugging closures, for example. This capability is also supported by ILAsm [1, p. 403]:

The `.line <start_line>[,<end_line>][:<start_col> [,<end_col>]] [<file_name>]` directive identifies the line and column in the original source file that are responsible for the IL code that follows the `.line` directive.

GenMSIL outputs line numbers only, the relevant code is shown in Figure 4.

A forum with related information is “Building Development and Diagnostic Tools for .Net”⁴.

⁴<http://social.msdn.microsoft.com/Forums/en/netfxtoolsdev/threads>

```
Visual Studio Command Prompt (2010) - Mdbg
d:\Music\MSTL>Mdbg
Mdbg (Managed debugger) v4.0.30128.1 (RC1Rel.030128-0100) started.
Copyright (C) Microsoft Corporation. All rights reserved.

For information about commands type "help";
to exit program type "quit".

mdbg> run HelloWorld
STOP: Breakpoint Hit
22: IL 0000: ldsfld class 'test$' 'test$': 'MODULE$'
[p#:0, t#:0] mdbg> _
```

Figure 5: mdbg

3.1 Generation with ilasm

Summary on using `ilasm` (from “Compiling in Debug Mode” in Ch. 19 of [1]):

- *If your compiler generates ILAsm source code, it must insert `.language` and `.line` directives at the appropriate points.*
- *If you are round-tripping a module compiled from a high-level language, use the disassembler option `/LINENUM` (or `/LIN`).*
- *In any case, don't forget to use one of the PDB-generating options of the ILAsm compiler: `/DEB`, `/DEB=OPT`, `/DEB=IMP`, or `/PDB` (the last option generates the PDB file but doesn't emit the `DebuggableAttribute`).*

It works, as the `mdbg` session in Figure 5 shows.

The next question is: How to programmatically invoke `ilasm` for a single (big) in-memory `.il` file? Apparently, `ilasm` provides no programmatic interface⁵.

- A utility (`InlineIL`⁶) invokes `ilasm` behind the scenes, using the snippet⁷ shown in Listing 2. `InlineIL` allows mixing source and bytecode.
- Another alternative is to use an ILAsm MSBuild task (TODO).

3.2 Generation with Reflection.Emit

From a PDB perspective, `Reflection.Emit` can be used, as discussed in the blog entry⁸ “Debugging Dynamically Generated Code”. The key API usages are shown in Listing 3.

The `ILGenerator.MarkSequencePoint` method⁹ supports associating a *text range* for a debug step, similar to the mechanism discussed for `ilasm` in Sec. 3.1.

⁵<http://social.msdn.microsoft.com/Forums/en-US/netfxtoolsdev/thread/9f31b40f-7e42-47b6-9bc4-14e55a7588a4/>

⁶<http://blogs.msdn.com/jmstall/archive/2005/02/21/377806.aspx>

⁷http://timstall.dotnetdevelopersjournal.com/using_systemdiagnostics_to_run_external_processes.htm

⁸<http://blogs.msdn.com/jmstall/archive/2005/02/03/366429.aspx>

⁹<http://msdn.microsoft.com/en-us/library/system.reflection.emit.ilgenerator.marksequencepoint.aspx>

Listing 2: RunHiddenConsole

```
public static void RunHiddenConsole(string strFileName, string strArguments, bool blnWaitForExit)
{
    //run in process without showing dialog window:
    ProcessStartInfo psi = new ProcessStartInfo();
    //psi.CreateNoWindow = true;
    psi.WindowStyle = ProcessWindowStyle.Hidden;
    psi.FileName = strFileName;
    psi.Arguments = strArguments;

    Process p = System.Diagnostics.Process.Start(psi);

    if (blnWaitForExit)
        p.WaitForExit();
} //end of method
```

Listing 3: Emitting for Debugging

```
using System.Diagnostics.SymbolStore;
using System.Diagnostics;
. . .
CustomAttributeBuilder daBuilder = new CustomAttributeBuilder(daCtor, new object[] {
    DebuggableAttribute.DebuggingModes.DisableOptimizations |
    DebuggableAttribute.DebuggingModes.Default });
. . .
// Tell Emit about the source file that we want to associate this with.
ISymbolDocumentWriter doc = module.DefineDocument(@"Source.txt", Guid.Empty, Guid.Empty, Guid.Empty);
. . .
// Create a local variable of type 'string', and call it 'xyz'
LocalBuilder localXYZ = ilGenerator.DeclareLocal(typeof(string));
localXYZ.SetLocalSymInfo("xyz"); // Provide name for the debugger.

// Emit sequence point before the IL instructions. This is start line, start col, end line, end column,
// Line 2: xyz = "hello";
ilGenerator.MarkSequencePoint(doc, 2, 1, 2, 100);
```

3.3 Generation with CCI

CCI uses the similarly named `ILGenerator.MarkSequencePoint`, which receives an `ILocation` as argument. Additionally, `MarkLabel(ILGeneratorLabel label)` is available. The pseudo-instruction `IILocation` represents a location in the instruction stream.

4 Quality of the generated code

4.1 Emitting for interop: CLS and Design Guidelines

Library designers are asked¹⁰ to consider the advantages of CLS conformance, to allow libraries to be easily consumed from other languages:

¹⁰<http://blogs.msdn.com/brada/archive/2004/03/20/93341.aspx>

The CLS (or Common Language Specification) is a simply a contract between programming language designers and class library authors. ... The CLS is basically just a subset of the entire set of features supported by the CLR. The CLS includes things such as calling virtual methods, overloading methods and does not include things such as unsigned types. ... FxCop enforces the Design Guidelines which includes things such as naming conventions for publicly exposed identifiers etc. Another way to look at this is that the CLS is a normative (binding) part of the CLI Specification whereas the Design Guidelines are a normative (informative, non-binding) part of the spec.

TODO How much of that is required/ensured out-of-the-box by the compiler?

TODO Perhaps an <code>-X</code> option can be provided to test CLS compliance for user programs. How to programmatically ask .NET to check such compliance?

TODO Not sure if we want to make the compiler itself CLS compliant. Who is going to call it as a component from, say, IronPython?

4.2 PEVerify

Running `peverify /MD /IL /VERBOSE HelloWorld.exe` reports “All Classes and Methods in HelloWorld.exe Verified”.

From a blog entry:

PEVerify itself is but a shell calling `IMetaDataValidate` interface and `ICLRRuntimeHost::VerifyAssemblyFile`. You can have a look at how PEVerify uses these interfaces in SSCLI (Rotor).

TODO How to programmatically invoke the <code>MDValidator</code> and <code>ILVerifier</code> components of PEVerify ([1] mentions them)

4.3 FxCop

This experiment involves running FxCop¹¹ on the just compiled HelloWorld (Figure 6). FxCop goes beyond well-formedness checking, comprising checks for style and other guidelines:

FxCop is a code analysis tool that checks .NET managed code assemblies for conformance to the Microsoft .NET Framework Design Guidelines. It uses MSIL parsing, and callgraph analysis to inspect assemblies for more than 200 defects in the following areas: Library design, Globalization, Naming conventions, Performance, Interoperability and portability, Security, Usage

TODO What are those guidelines? How much overlap with CheckStyle?

TODO How much overlap with FindBugs?

TODO How to invoke FxCop programmatically (and retrieve its results)
--

- It’s possible to add custom rules (or a custom rules library) to FxCop <http://blogs.msdn.com/fxcop/archive/2006/03/11/549611.aspx>

¹¹FxCop website, <http://go.microsoft.com/fwlink/?LinkId=70294>

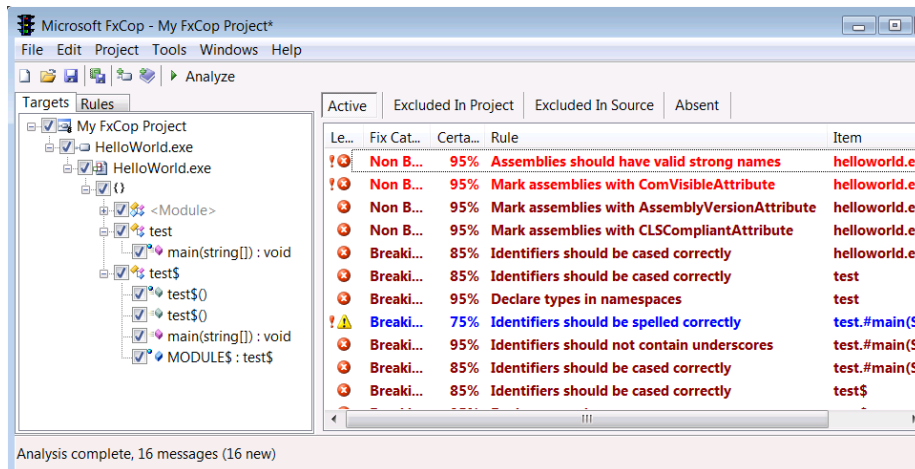


Figure 6: FxCop on HelloWorld

- The Mono equivalent of FxCop is Gendarme
<http://www.mono-project.com/Gendarme>
- Blog with related info: “The Visual Studio Code Analysis Team Blog”
<http://blogs.msdn.com/fxcop/>.

4.4 Static verification of implicit preconditions

It can be argued whether it’s the compiler’s job or not. An approach is discussed at the Boogie forum¹²:

The problem with FxCop is that the Abstract Syntax Tree (AST) it builds is very low level. Just trying to figure out if the call to Dispose above is inside an if statement is difficult. Besides that, the AST’s are very different for Debug or Release compiled code. ... a deeper property that you might want to enforce is that most methods on a disposable object have an (implicit) precondition that the object has not already been disposed. So a type-state property is that once Dispose has been called, some set of methods may no longer be called.

4.5 Verification of compiler backend vs. Validation of translation results

The former is a one-time activity (at compiler-construction time) while the latter is performed on each input-output pair (i.e., once after each compiler run). Per-run validation makes sense for optional optimizations (if a counterexample is found, the optimization is unrolled, where the counterexample may be conservative regarding behavioral equivalence).

- TODO <http://pauillac.inria.fr/~xleroy/>
- TODO search in Google Scholar for “translation validation +compiler”

¹²<http://boogie.codeplex.com/Thread/View.aspx?ThreadId=203344>

5 Battle of the CIL-emit libraries (Round 2)

If you've been following these write-ups you'll have noticed that the dispute among competing reflection-like libraries goes on and on (CCI, System.Reflection, IKVM.Reflection, etc.) Yet more opinions on this topic follow.

5.1 System.Reflection: too eager type resolving

From Mike Stall's Debugging Blog¹³:

*For example, Reflection does not expose `TypeRef`, `MemberRef`, `AssemblyRef`, or other `Ref` tokens. These tokens are references to things in other assemblies. Reflection just resolves them for you (potentially invoking an event to get help from your app) and hands back the resolved object. Similarly, reflection is also missing `TypeSpecs`. `TypeSpecs` are just binary signature blobs that describe compound types (arrays, **generics**, etc). Reflection will parse the blob and resolve it to a real `System.Type`.*

TODO If Reflection resolves a `TypeSpec` to a `System.Type`, this means an assembly containing the resolved type's declaration should be on the compiler's classpath (in addition to `-Xassem-extdirs`). Example needed

5.2 CCI documentation

The Sample Walkthroughs at the CCI wikis (one per subproject) showcase important API details. Two good starting points are:

- HelloIL <http://ccimetadata.codeplex.com/wikipage?title=HelloIL%20Sample%20Walkthrough>
- HelloCodeModel <http://cciast.codeplex.com/wikipage?title=HelloCodeModel%20SampleWalkthrough>

Reference information is provided in the API Overview,
<http://ccimetadata.codeplex.com/wikipage?title=API%20overview>

6 Link Sink

- ILMerge,
<http://research.microsoft.com/en-us/people/mbarnett/ilmerge.aspx>
- Tools and Utilities for .NET (CLR Profiler, FxCop, ILMerge, LibCheck),
<http://msdn.microsoft.com/en-us/netframework/aa569269.aspx>

References

- [1] Serge Lidin. *Expert .NET 2.0 IL Assembler*. Apress, Berkely, CA, USA, 2006.

¹³<https://blogs.msdn.com/jmstall/archive/2008/03/15/things-in-metadata-that-are-missing-in-reflection.aspx>