Making `ILPrinterVisitor` ready for Generics

© Miguel Garcia, LAMP,
École Polytechnique Fédérale de Lausanne (EPFL)
`http://lamp.epfl.ch/~magarcia`

June 24th, 2010

Abstract

The process by which `.msil` files are written to disk starts with `GenMSIL` building an internal representation of the assembly to serialize. This representation comprises instances of `AssemblyBuilder` and `TypeBuilder`, builders for type members, and `ILGenerator` instances. The resulting tree (rooted at `AssemblyBuilder`) resembles a concrete syntax tree, with each `MethodBuilder` and `ConstructorBuilder` (for non-abstract members) owning an `ILGenerator` to record a stream of CIL instructions. Serialization into an `.msil` file is the responsiblity of `ILPrinterVisitor`. In order to make this visitor ready for Generics, it should serialize the additions (for Generics) that the representation classes will exhibit. After these updates, switching to CCI (Common Compiler Infrastructure) should be easier: only `ILPrinterVisitor` and its two subclasses would need to be re-written, while `AssemblyBuilder` & Co. will stay the same.

Contents

1 IDE Setup

For those just arriving to *The Scala Compiler Corner*, here go the instructions to set up a debugging session:

```
svn co http://lampsvn.epfl.ch/svn-repos/scala/scala-msil/trunk sn2
ant clean build
```

I've unpacked the `.dll`s from ikvmbin-0.43.3824 into `Z:\scalaproj\mscor\sn2`, adding `scalaruntime.dll`, `mscorlib.dll`, and `System.dll` (all from the `lib` folder in `scala-msil`)

- Main class: `scala.tools.nsc.Main`
- VM parameters:

  ```
  -Xbootclasspath/a:Z:\scalaproj\sn2\build\pack\lib\scala-compiler.jar;
  Z:\scalaproj\sn2\build\pack\lib\scala-library.jar
  ```

  When compiling the compiler for JVM we also included:

  ```
  Z:\scalaproj\sn2\lib\fjbg.jar;
  Z:\scalaproj\sn2\lib\msil.jar
  ```

  but they are not necessary now (in fact, they were not necessary then either, somehow they managed to slip through all along).

- Program parameters:

  ```
  -target:msil -Xassem-extdirs Z:\scalaproj\mscor\sn2
  -no-specialization -Ystruct-dispatch:no-cache
  -Ydebug
  @Z:\scalaproj\sn2\allLibraryFiles.txt
  ```

  As you can see, rather than listing all the `.scala` files as in previous write-ups, I've become in the meantime a big fan of `@` parameters to `scalac` (I was getting funky errors due to the interplay of source folder and whatever when sticking to the old habit).

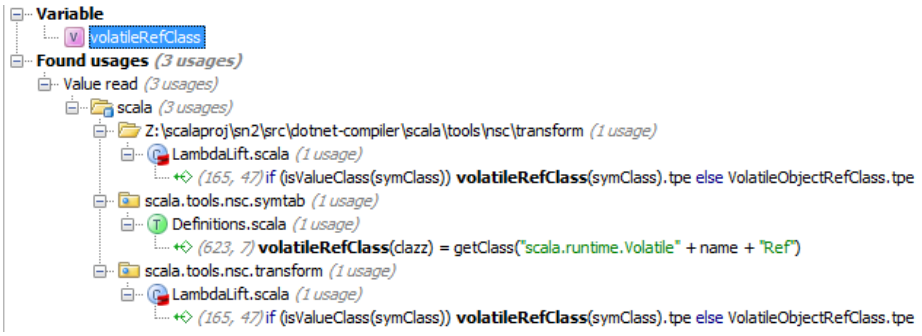
- Working directory: `Z:\scalaproj\sn\src`

1.1 `Volatile`s make their splash

So that you see the *realities of software*, on first attempt we get an exception (`scala.tools.nsc.MissingRequirementError`) because `scala.runtime.VolatileBooleanRef` can't be found (`Definitions.scala`). The following:

```
volatileRefClass(clazz) = getClass("scala.runtime.Volatile" + name + "Ref")
```

causes trouble because `VolatileByteRef` and similar (Java) classes were not there last time. The `volatileRefClass` map is used at:



Two write-ups ago we prepared `JavaFilesInScalaLib.dll`, containing just `scala.math.ScalaNumber` and `scala.runtime.BoxesRunTime` because in those times those were the only `.java` files not delivered out-of-the-box in `scalaruntime.dll`. But now some more `.java` files show up under `src\dotnet-full-library`:

```
scala.reflect.ScalaLongSignature
scala.reflect.ScalaSignature

scala.runtime.ArrayRuntime
scala.runtime.VolatileBooleanRef
scala.runtime.VolatileByteRef
scala.runtime.VolatileCharRef
scala.runtime.VolatileDoubleRef
scala.runtime.VolatileFloatRef
scala.runtime.VolatileIntRef
scala.runtime.VolatileLongRef
scala.runtime.VolatileObjectRef
scala.runtime.VolatileShortRef
```

The `scala-lib.dll` compiled in a previous write-up also does not contain the `Volatile` classes. They need to be compiled. The `.bat` file in Listing 1 just got longer.

An update:

In the repository-based build, `scalaruntime.dll` is not created with `ikvmc`. Instead, for each `.java` file in the `src/library` folder a corresponding C# file is created in `src/dotnet-library` and then compiled with:

```
mcs -target:library -out:scalaruntime.dll ‘find src/dotnet-library -name *.cs‘
```

Documentation about this at `http://lampsvn.epfl.ch/trac/scala/browser/scala-msil/trunk/src/dotnet-library/README`

1.2 Bringing in `predef.dll` and `JavaFilesCompilerMSIL.dll`

We copy `scala-lib.dll` into `Z:\scalaproj\mscor\sn2` and rename it to `predef.dll`. Together with `JavaFilesCompilerMSIL.dll` (details in the “*Bootstrap (1 of 2)*” write-up). If you prefer, the command-line to compile the compiler is shown in Listing 2. To reproduce, run `Z:\scalaproj\sn2>compCompiler.bat` without forgetting to place `JavaFilesCompilerMSIL.dll` alongside the other `dll`s, that one contains the `ch.epfl.lamp.msil` package.

Listing 1: So that `Definitions` can find the `Volatile`s it looks for

```
rd /Q /S .\classes
mkdir classes

javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\math\ScalaNumber.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\BoxesRunTime.java

javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\reflect\ScalaLongSignature.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\reflect\ScalaSignature.java

javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\ArrayRuntime.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileBooleanRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileByteRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileCharRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileDoubleRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileFloatRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileIntRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileLongRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileObjectRef.java
javac -d .\classes -cp .\classes Z:\scalaproj\sn2\src\dotnet-full-library\scala\runtime\VolatileShortRef.java


del JavaFilesInScalaLib.jar
jar -cf JavaFilesInScalaLib.jar -C .\classes\ .
del JavaFilesInScalaLib.dll
ikvmc -target:library JavaFilesInScalaLib.jar
```

Listing 2: Command-line to compile the compiler (Sec. 1.2)

```
cd Z:\scalaproj\sn\src

"C:\Program Files\Java\jdk1.6.0_19\bin\java" -Xms512M -Xmx1236M -Xss8M -XX:MaxPermSize=128M
-Xbootclasspath/a:Z:\scalaproj\sn2\build\pack\lib\scala-compiler.jar;Z:\scalaproj\sn2\build\pack\lib\scala-library.jar -classpath
"C:\Program Files\Java\jdk1.6.0_19\jre\lib\alt-rt.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\charsets.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\deploy.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\javaws.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\jce.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\jsse.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\management-agent.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\plugin.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\resources.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\rt.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\ext\dnsns.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\ext\localedata.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\ext\sunjce_provider.jar;C:\Program Files\Java\jdk1.6.0_19\jre\lib\ext\sunmscapi.jar;C:\Program
Files\Java\jdk1.6.0_19\jre\lib\ext\sunpkcs11.jar"
scala.tools.nsc.Main
-target:msil
-Xassem-name compilerRESULT
-Xassem-extdirs Z:\scalaproj\mscor\sn2
-no-specialization -Ystruct-dispatch:no-cache
-d Z:\scalaproj\sn2\out
@Z:\scalaproj\sn2\allCompilerfiles.txt

copy Z:\scalaproj\sn2\out\compilerRESULT.msil Z:\scalaproj\mscor\sn2\scala-compiler.msil

ilasm /QUIET /DEBUG /EXE scala-compiler.msil

cd Z:\scalaproj\sn2\
```

1.3 Trying our luck with `scala-lib`

By now you may have realized that before getting to `ILPrinterVisitor` proper we're checking whether the latest version from trunk compiles the sources that were patched to comply with IKVM.

I'm eliding here some funky errors I was getting for not using an `@` parameter in the launch configuration. In general, there's no easy way to determine if compilation errors are unrelated to Scala.NET (say, by successufully compiling those same sources "for JVM" for comparison). In our case, sources already contain references to .NET (`System.Object` in the snippet).

The changes to the Scala.NET compiler required to bootstrap are minimal, as shown in Figure 1 (that summary should have appeared in the write-up on bootstrapping). `ILPrinterVisitor` is also shown there, but due to differences in formatting only. By the end of this write-up, it will have been updated in preparation for Generics.

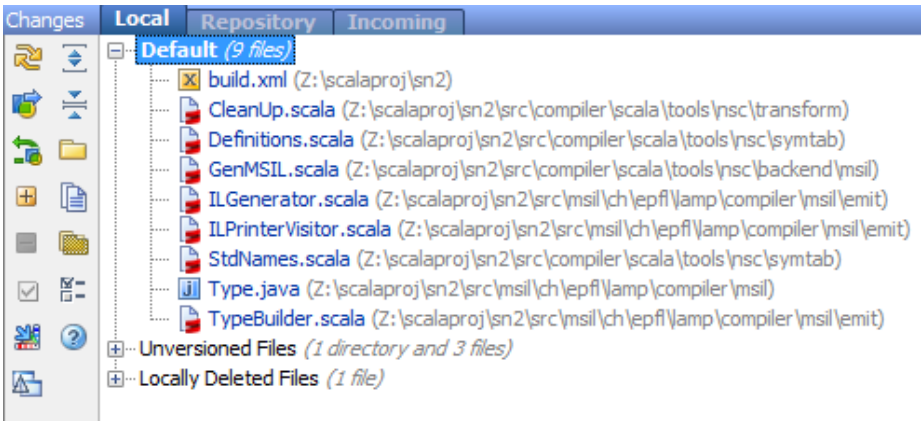


Figure 1: Changes required in Scala.NET to bootstrap

2 What needs to be changed

Each node handler in the `msil.emit.Visitor` trait returns `Unit`, and their realization in `ILPrinterVisitor` chooses the visit order of child nodes (by invoking `print(node)`, which in turn applies the visitor). To realize a multi-pass visitor (CCI likely requires that), a visitor can be defined that visits in some order and delegates node processing proper to subclasses mixing-in this trait.

At the level of `msil.emit`, support for generics comprises:

- definitions introducing type parameters into scope (type, instance constructor, and method definitions). The kinds of constraints (on type arguments) that a type-param may specify are: variance, reference or value type, default constructor, base type, and supported interfaces.
- references to generic types, constructor, or methods. In all cases, these references must be to instantiations i.e., a type argument must be supplied for each parameter. The supplied type argument may be a reference to a type parameter in scope (an instantiation need not denote a concrete type). In ILAsm, these references can appear4:
  - as part of type ascriptions (this includes type-param constraints)
  - in an explicit `.override`
  - as part of an invocation of a generic method

We review next (most of) the grammar productions for the items above.

2.1 Grammar productions

ILAsm grammar productions for type definitions:

```
.class <flags> <dotted_name> < <gen_params> > [extends <class_ref>] [implements <class_refs>] { ... }

<gen_params> ::= <gen_param> [, <gen_param>]*

<gen_param> ::= [<constraint_flags>] [( <constraints> )] <gen_param_name>
```

```
<constraint_flags> ::= + | - | class | valuetype | .ctor

<constraints> ::= <class_ref> [, <class_ref>]*
```

And for method definitions:

```
<gen_method_def> ::=
    .method <flags> <call_conv> <ret_type> <name>< <gen_params> > (<arg_list>) <impl>
    { <method_body> }
```

2.2 Well-Formedness-Rules (WFRs)

The WFRs around generics are formulated in terms of the "logical metadata" format, but their mapping to ILAsm should be apparent. A sample of the WFRs for generic method definitions follows:

- The implementation flags `<impl>` of a generic method definition may be only `cil` or `managed` (the defaults).
- The `<flags>` clause of a generic method definition cannot contain the `pinvokeimpl` flag.
- The `<call_conv>` of a generic method cannot be `vararg`.
- Class constructors (`.cctor`) cannot be generic, because they are not called explicitly, so there is no way to specify a type argument for a `.cctor`. The instance constructors (`.ctor`), which are called explicitly (`newobj` instruction), can be generic [1]

A WFR about references to generic methods:

- *When calling (or otherwise referencing) a generic method instantiation directly, you need to specify its signature as it was defined, not as it became in the instantiation. It is the same rule that applies to calling nongeneric methods of generic types: if a parameter or return type of the method is declared as a type parameter (of class or method) number 0, it should be specified as such at the call site, no matter what type substitutes for the type parameter number 0.* [1] The only text being different are the type arguments, otherwise a method reference resembles the signature of its definition.

TODO It would be great if some of the WFRs around generics were made checkable on `TypeBuilder` and such.

2.3 Looking back at metadata parsing

We added fields to `msil.Type` to hold type-params defs, and these fields are inherited by `TypeBuilder`. Given that `ILPrinterVisitor` serializes all `...Builder`s, we review here those additions.

- `http://lampsvn.epfl.ch/trac/scala/changeset/22150`
- `http://lampsvn.epfl.ch/trac/scala/changeset/22212`

Listing 3: `TMVarUsage`, (Sec. 2.3)

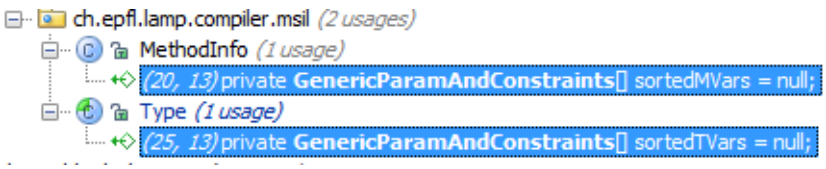
```
static final class TMVarUsage extends Type {

    public final int Number;
    public final boolean isTVar;

    /** Non-defining reference to either a TVar or an MVar */
    public TMVarUsage(int Number, boolean isTVar) {
        super(null, 0, ((isTVar ? "!" : "!!") + Number), null, null, null, 0, null);
        this.Number = Number;
        this.isTVar = isTVar;
    }

    public String toString() {
        return (isTVar ? "!" : "!!") + Number;
    }
}
```

Basically, as shown in Listing 3 and Listing 4, which in turn are used in:



`msil.Type` sports a `getSortedTVars()` getter, while `MethodInfo` has `getSortedMVars()`. Both return an array of `GenericParamAndConstraints` and perform sorting on demand. Whenever a type-param definition is added (by invoking `addTVar` or `addMVar`) the sorting is invalidated (`sortedTVars = null` resp. `sortedMVars = null`) and recomputed upon the getter's invocation.

Listing 4: `GenericParamAndConstraints`, (Sec. 2.3)

```
public class GenericParamAndConstraints {

    public GenericParamAndConstraints(int Number, String Name, Type[] Constraints,
                                      boolean isInvariant, boolean isCovariant, boolean isContravariant)
    { // TODO representation for the class and new() constraints missing
      . . .
    }

    public final int Number;
    public final String Name; // can be null
    public final Type[] Constraints; // can be empty array
    public final boolean isInvariant; // only relevant for TVars, not for an MVar
    public final boolean isCovariant; // only relevant for TVars, not for an MVar
    public final boolean isContravariant; // only relevant for TVars, not for an MVar

    public String toString() { . . . }

}
```

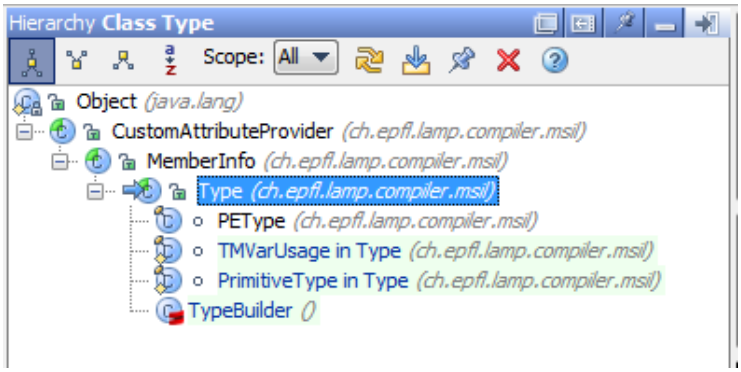


Figure 2: `msil.Type`

3 `caseTypeBuilder()` and `caseMethodBuilder()` do generics

The latest additions to `msil.Type` (e.g., `sortedTVars` for parsing metadata, Sec. 2.3) don't follow the `TypeBuilder` idioms to build type representations (`DefineField()`, `DefineMethod()`, `DefineConstructor()`, and `DefineNestedType()`). Instead, type parameters are defined by invoking:

```
public void addTVar(GenericParamAndConstraints tvarAndConstraints) {
    sortedTVars = null;
    tVars.add(tvarAndConstraints);
}
```

on a type. Actually, invoking `addTVar()` on `PEType` and `TypeBuilder` make sense, unlike for the two other subclasses of the `msil.Type` abstract class (`TMVarUsage` and `PrimitiveType`). For similar reasons, a `TMVarUsage` shouldn't be stored in the `Type.types` map from a type's fullname to its representation, i.e. don't invoke `Type.addType(t)` with `t` a `TMVarUsage` (there's an `assert` guarding against this, BTW).

In ILAsm notation, lists of type params (enclosed in angle brackets) follow the same grammar production for both types and methods, and thus the following helper (in `ILPrinterVisitor`) unparses them (those lists differ in WFRs however, e.g. variance can't be specified for method-scoped type params).

```
def printTypeParams(sortedTVars : Array[GenericParamAndConstraints])
```

Actually, we need to unparse type-param names only for types we define ourselves, as the following comment in `ILPrinterVisitor` shows:

```
def tparamName(tVar : GenericParamAndConstraints) = {
 /* TODO Type-params in referenced assemblies may lack a name
     (those in a TypeBuilder or MethodBuilder shouldn't).

    Given that we need not list (in ilasm syntax) the original type-params' names when
     providing type arguments to it, the only type-param-names we'll serialize into a .msil file
     are those for type-params in a TypeBuilder or MethodBuilder. Still, more details on this
     appear in Sec. 4.5 "Faulty metadata in XMLReaderFactory" of
     http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/Libs4Lib.pdf
```

```
    To avoid name clashes when choosing a param name,
    first collect all existing tparam-names from a type (and its nested types).
    Not that those names are needed (ordinal positions can be used instead)
    but will look better when disassembling with ildasm. */

  assert(tVar.Name != null)
  tVar.Name
}
```

3.1 References to type-params in type ascriptions

There's another low-hanging fruit besides those described in the previous subsection. Given that `TMVarUsage` subclasses `msil.Type`, existing unparsing code will produce the correct ILAsm syntax whenever a type-param is addressed. `ILPrinterVisitor.printReference()` serializes the `TMVarUsage`'s `Name` as an ordinal reference (due to the assignment shown in Listing 3).

A type-param reference (ordinal or name-based) always refers to a type param defined in the immediately enclosing scope (type for `!`-style usages, method or instance constructor for `!!`-style usages) (EXCEPT when referring to the signature of a generic method, as part of invocations or explicit overrides, where type-param usages refer to the type vars in scope in the method declaration). Unlike in Java, in a class `N` nested in `C`, the type params (if any) of `C` are not in scope in `N`.

A `TMVarUsage` neither points nor is pointed to by other nodes (as already mentioned, the trees rooted at `AssemblyBuilder` are CSTs rather than ASTs).

3.2 Unparsing (references to) constructed types

In keeping with .NET speak, appling a type constructor to type-args results in a *constructed type*:

```
.field private class List`1<string> nameList
.field private class List`1<[mscorlib]System.Type> typeList
```

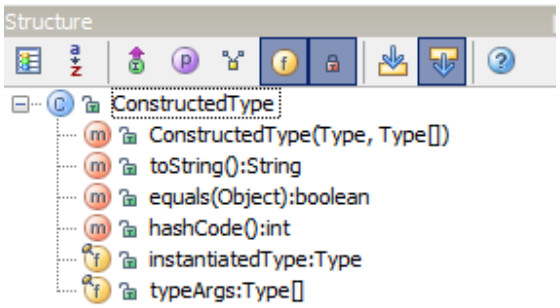
Note: Please notice the `class` keyword in the snippet above (in a type reference) hinting `ilasm` to encode the thing as a row in the TypeSpec table (and not in the TypeRef table). Quoting from [1, p. 232]: "*This is a general rule of ILAsm, not specific to the generic type instantiations. For example, the notation* `[mscorlib]System.Type` *translates into a* `TypeRef`*, while the notation* `class [mscorlib]System.Type` *translates into a* `TypeSpec` *with the signature {*`E_T_CLASS, <token>`*}, where* `<token>` *is a* `TypeRef` *token of* `[mscorlib]System.Type`*.*"

4 To be Continued

At this point we noticed that no more changes to `ILPrinterVisitor` make sense until the representation of generics in terms of `msil.Type` has been decided, and for that we need to extend `TypeParser`. That's the topic of the next write-up. We'll start there with a newly added class for this purpose:

Listing 5: Quiet `NaN` vs. Signaling `NaN` (Sec. ??)

```
. . .
  } else if (value.isInstanceOf[Float]) {
    // TODO !!! check if encoding is correct
    val bits = java.lang.Float.floatToRawIntBits((value.asInstanceOf[Float]).floatValue())
    // float32(float32(...)) != float32(...)
    print("float32 (float32 (")
    /* see p. 170 in Lidin's book Expert .NET 2.0 IL Assembler */
    val valFlo = value.asInstanceOf[Float]
    if (java.lang.Float.NaN == valFlo) print("0xFFC00000 /* NaN */ ") /*- TODO this is 'quiet NaN',
        http://www.savrola.com/resources/NaN.html , what's the difference with a 'signaling NaN' ?? */
    else if (java.lang.Float.NEGATIVE_INFINITY == valFlo) print("0xFF800000 /* NEGATIVE_INFINITY */ ")
    else if (java.lang.Float.POSITIVE_INFINITY == valFlo) print("0x7F800000 /* POSITIVE_INFINITY */ ")
    else print(bits)
    print("))")
  } else if (value.isInstanceOf[Double]) {
. . .
```



5 `NaN`, `Infinity`, and `-Infinity` in ILAsm

Quoting from [1]:

> *Floating-point numbers have special cases, such as positive infinity, negative infinity, and not-a-number (NaN), that cannot be presented textually in simple floating-point format. In these special cases, the floating-point constants can alternatively be represented as integer values with a matching byte count . . .*
>
> ```
> .field public float32 fPosInf = float32(0x7F800000)
> .field public float32 fNegInf = float32(0xFF800000)
> .field public float32 fNAN = float32(0xFFC00000)
> ```

The current solution appears in Listing 5.

TODO References to type-params in CIL instructions

References

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