Flexible Bytecode for Linking in .NET

Alex Buckley [1] Michelle Murray [2] Susan Eisenbach
Sophia Drossopoulou

Department of Computing, Imperial College London

Abstract

Dynamic linking in modern execution environments like .NET is considerably more sophisticated than in the days of C shared libraries on UNIX. One aspect of this sophistication is that .NET assemblies embed type information about dynamically linked resources.

This type information implicitly represents compile-time assumptions about the resources available at run-time. However, the resources available at run-time may differ from those available at compile-time. For example, the execution environment on a mobile phone might provide fewer, simpler classes than on a desktop PC. As bytecode cannot adapt to its execution environment, component reuse is restricted and development costs are increased.

We have designed and implemented a "flexible" dynamic linking scheme that binds bytecode as late as possible to the assemblies and classes available in a .NET execution environment. We describe the scheme's integration with the .NET linking infrastructure, review important design decisions and report on experiences with the "Rotor" shared source version of .NET.

Key words: Flexible dynamic linking, .NET, CLR, Type variables

1 Introduction

Modern execution environments including Microsoft's Common Language Runtime (CLR) [18] and Sun's Java Virtual Machine (JVM) [17] support dynamic loading and linking of bytecode obtained from local and remote sites. To ensure that code safely interoperates with already running code, verification has become an important step in the linking process. Verification amounts to link-time typechecking, and is assisted by compilers embedding type information into bytecode.

[1] Email: a.buckley@imperial.ac.uk
[2] Email: michelle@mcs.st-and.ac.uk (Work done while at Imperial College London)

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

By embedding type information, a compiler is also implicitly embedding the "shape" of the compilation environment - its classes, their subtyping relationships, their fields and methods. This rules out truly separate compilation because compiling a class requires the availability of all the classes that it references. Furthermore, bytecode is inflexible because it can only run in an environment that duplicates the compilation environment. For example, consider the source code:

```
new A().f.g
```

In a compilation environment where class `A` has a field `f` of type `B`, and class `B` has a field `g` of type `int`, a compiler will produce the following (pseudo-) bytecode:

```
1  new <Class A>
...
5  loadfield <Field B   f>
6  loadfield <Field int g>
```

Suppose we try to run this code in an execution environment where class `A` has a field `f` *of class `C`, not `B`*. Class `C` need not be a subtype of class `B`, or vice versa. In this execution environment, class `C` provides a field `g` of type `int`. The bytecode "ought" to run - `A` provides a field `f`, and the class of that field `f` provides a field `g` of type `int`. Unfortunately, we will get an exception because the bytecode demands a specific class - `B` - that was present in the compilation environment but is not present in the execution environment.

Some scenarios where the compilation and execution environments differ are:

(i) The security libraries provided on a mobile device may be similar to those on a desktop PC, but drop the more complex cryptographic functions.

(ii) A software component executing in a grid architecture should exploit the best distributed data structures available at that installation, which may be different from those available to the component's developer.

(iii) A user could have a vendor-provided implementation of a common API which uses fast but proprietary mechanisms, *e.g.* an ODBC driver from Oracle.

In the first scenario, the mobile libraries will typically provide classes with different names, or in a different namespace, from the desktop libraries. Today, a developer writing a program for mobile and desktop machines must produce multiple builds, each referring to the right set of classes. Conditional compilation via preprocessor directives helps to manage which source fragments use which classes, but not all languages provide such directives. Even with conditional compilation, a programmer is forced to bind early to the set of library classes used in his program.

We argue that a programmer (and a compiler) should not need to know

all the library classes that will be used at run-time. In the second scenario, unlike the first, the names of all classes available on a system are probably not known in advance. The traditional solution to this problem, reflection, can be unwieldy to program and generally loses some degree of type-safety. What really matters in the first and second scenarios is that classes exist in the execution environment that provide the fields and methods referred to by application bytecode. If the OS or VM's linking infrastructure could take such member references into account, then acceptable classes could be identified at run-time without special reflective mechanisms being developed by hand.

In the third scenario, a *service provider interface* (SPI) [21] is commonly used to abstract an application from the exact classes that implement an API. An SPI allows implementations of an API to register and deregister themselves, and be enumerated and bound to by applications. However, each SPI has a slightly different design, registry, and naming conventions, and each API designer essentially re-implements the Factory pattern (the heart of most SPIs). Again, if the OS or VM's linking infrastructure was capable of run-time class selection, then more applications could achieve greater flexibility for lower development cost.

To this end, we advocate *flexible bytecode* that mentions *type variables* rather than class names. Type variables are substituted to real class names in the context of an execution environment. Below, a compiler has used type variable `X` at line 5 that must be substituted to a real class name in order to resolve field `f`:

```
1  new <Class A>
...
5  loadfield <Field X   f>
6  loadfield <Field int g>
```

Type variable substitution could happen at a static linking step *after* compilation but *before* execution, or during the dynamic linking process at run-time. The first option would not correspond to the "lazy" nature of linking in modern execution environments. Furthermore, the later a substitution can be made, the more information is available about the running program and available resources, and the better is the chance for optimisation. Therefore, our interest is in enhancing dynamic linking to safely substitute type variables as late as possible. We call this approach *flexible dynamic linking* and formalised it in [5].

This paper describes our experience implementing flexible dynamic linking in .NET, in order to support flexible bytecode. We find the .NET platform an ideal base for our work: it supports typeful bytecode targeted by popular languages like C♯, and provides a rich *shared source* version of its codebase, known as "Rotor", to experiment with.

The rest of this paper is structured as follows. We review the dynamic linking subsystem in .NET in §2 and identify the meaning of flexible bytecode

and flexible dynamic linking in that environment in §3. We describe our implementation in §4 and §5. We conclude in §6 and §7 with related work and suggest improvements to our scheme.

2 Dynamic Linking in .NET

The ECMA Common Language Infrastructure (CLI) [11] defines an execution engine to load and run programs, and discover and link the resources required by a program. Among its services are garbage collection, exception handling and enforcement of security properties. Microsoft use the term *.NET* for their commercial implementation of the CLI, and *Common Language Runtime* (CLR) for the execution engine implementation.

The CLR executes a machine-independent assembly language, Common Intermediate Language (CIL). Like Java bytecode, CIL member accesses (*i.e.* field access and method call) are annotated with the type that defines the member. Unlike Java bytecode, this type includes the assembly where the defining type was found at compile-time. For example, this CIL (to print "Hello World" on screen) refers to the `WriteLine` method in the scope of the `System.Console` class of the `mscorlib` assembly: [3]

```
.method public hidebysig static void main() cil managed
{
  .entrypoint
  .maxstack  1
  IL_0000: ldstr ‘‘Hello World’’
  IL_0005: call void [mscorlib]System.Console::WriteLine(string)
  IL_000a: ret
}
```

CIL code resides in *assemblies*. An assembly is a self-describing, versioned, platform-independent file that contains one or more modules plus a manifest that lists resources provided by an assembly and external resources that it uses. A module contains type definitions and other resources like localisation strings. On the CLR, an assembly can be *private* (used by a single application) or *public* (cryptographically signed and registered in a system-wide *Global Assembly Cache* for all applications to use; different versions of a public assembly can execute "side by side" without "DLL hell" [12].)

The execution engine uses an assembly's metadata - referenced assembly names, class names, and other values - to load assemblies and classes, verify and JIT-compile methods, and resolve references:

Class loading The "Fusion" subsystem locates and loads assemblies, and loads classes from assemblies. It searches for assemblies in several different places, including the application's directory and the Global Assembly Cache.

[3] This assembly contains most of the core .NET classes.

Search policies are configurable with XML files.

Verification The verifier checks that a loaded class's metadata is well formed and that type signatures are used correctly within CIL code, *i.e.* that type safety is preserved. These checks must be satisfied before a method's CIL code can be executed. Verification is optional in the CLI specification, and can be disabled on a per-assembly basis.

JIT-compilation The JIT compiler converts a method's CIL code into native code when the method is invoked. Each method is JIT-compiled at most once within a single program run. Methods that are never used within a program are not JIT-compiled even if other methods of the same class have been.

Resolution A CIL instruction that accesses a class member undergoes resolution as it is JIT-compiled. The assembly and class that scope the member are looked up in the metadata of the assembly being JIT-compiled, and the member's assembly is loaded by Fusion (if not already loaded). Then, the member's class is extracted from the assembly and the offset to the member in the class is calculated. Resolution can fail in many ways: if the target assembly cannot be found in the referencing's assembly's metadata, if an assembly file cannot be loaded, if a class cannot be found in an assembly, or if fields and methods are not found in a class.

The timing of dynamic linking in [11] is very flexible. Verification may be performed at any time,[4] and resolution of class members can happen any time after the class is loaded, up to JIT-compilation.[5] Resolution exceptions can be thrown *any time* after the resolution failure, in contrast to Java, which requires that they be flagged as late as possible.[6]

3 Flexible dynamic linking in .NET

The example CIL code in §2 relies on a specific assembly (`mscorlib`) and class (`System.Console`) being available at compile-time *and* run-time. Suppose that `mscorlib` is found on desktop PCs; on a PDA, only an alternate version is available, called `mscorlib-lite`. The example code would fail to link on a PDA because precisely `mscorlib` is required, yet is not available. Flexible dynamic linking allows the CIL code to avoid naming a specific assembly, *i.e.*

```
IL_0005: call void [X]System.Console::WriteLine(string)
```

and also to avoid naming a specific class within an assembly, *i.e.*

```
IL_0005: call void [X]Y::WriteLine(string)
```

[4] [11], Partition II, §3.
[5] [11], Partition I, §12.4.2.3.
[6] [17], §2.17.3.

We use the term *flexible CIL code* for CIL code that features type variables, and call `X` an *assembly type variable* and `Y` a *class type variable*. To allow flexible CIL code and implement flexible dynamic linking in .NET, we need to:

(i) introduce type variables into CIL code, [7]

(ii) build an assembly that includes type variables,

(iii) substitute type variables at run-time.

3.1 Introduction of type variables into CIL

CIL code is easy to obtain from compilers and modify by hand in a text editor. CLI implementations usually provide an assembler to build binary assemblies from CIL files. We allow the use of assembly and class type variables in CIL method call instructions only. We adopt a simple naming convention to help the assembler distinguish type variables from ordinary assemblies and classes: assembly type variables are prefixed by `fdlA` and class type variables by `fdlC`. The same type variable can be used in multiple instructions.

3.2 Substitution of type variables

There are three major opportunities for substitution: when an assembly is built, when an assembly/class is loaded, and when instructions that feature type variables are resolved during JIT-compilation. Since the ethos of dynamic linking in the CLI is that it should be as late as possible, the third option - substitution during JIT-compilation - is most appropriate. We name this approach *JIT-substitution*.

Therefore, when we execute an assembly built from the flexible CIL code above, the JIT compiler will encounter the assembly type variable `X` and ask the assembly loading subsystem to determine an appropriate substitution. After an assembly is chosen, it can be loaded as normal and the class loader can be asked to determine a substitute class for the class type variable `Y`. Providing this assembly/class combination contains a `WriteLine` method with the right signature, JIT-compilation can proceed as normal. If a type variable is used twice in the same assembly, it is assumed to represent the same assembly/class. Multiple type variables can substitute to the same assembly/class.

In this work, we concentrate on the *mechanism* that performs a substitution. We do not automatically infer what a substitution should be. Instead, we ask the end-user to name assemblies and classes that can substitute for assembly and class type variables. Furthermore, we do not check the validity of a user's substitution; for the CIL code above, we would *not* check whether the assembly that substitutes for `X` contains the class that substitutes for `Y`, nor whether `WriteLine` exists in the class that substitutes for `Y`.

[7] We will consider type variables in source languages like C♯ as future work.

3.3 Representation of type variables

The timing of substitution determines the representation of type variables. If we took a static linking approach, the assembler would perform substitutions and there would be no type variables left to represent at run-time. Since we have chosen to substitute type variables at run-time (during JIT-compilation), we need a mechanism to represent them to the execution engine; an assembly type variable should be distinguished from an ordinary assembly, and a class type variable should be distinguished from an ordinary class name.

To represent an assembly type variable, we enhance the metadata that the assembler creates about assemblies referenced in CIL code. Below is a human-readable version of the `AssemblyRef` metadata item created when the ordinary `mscorlib` assembly is referenced in §2:

```
// AssemblyRef #1
// -----------------------------------------------------
//    Token: 0x23000001
//    Public Key or Token: b7 7a 5c 56 19 34 e0 89
//    Name: mscorlib
//    Major Version: 0x00000001
//    Minor Version: 0x00000000
//    Locale: <null>
...
```

In most high-level languages, member accesses are *not* qualified by an assembly. Therefore, the compiler chooses an assembly that contains a class with the desired members, and adds a `.assembly extern` directive to CIL code. This directive specifies the name, version, public key and locale of the chosen assembly. When the assembler parses a CIL instruction that references an external assembly, it looks up the directive describing that external assembly. Then it builds an `AssemblyRef` metadata item, without checking for the external assembly itself.

There would be no point in writing an `.assembly extern` directive for an assembly type variable, since it can have no information beyond the variable's name. Therefore, we modify the assembler to recognise an assembly type variable by our naming convention, and create `AssemblyRef` metadata *without the need for a corresponding directive.* We add an extra field to `AssemblyRef` to indicate to the execution engine (and any tool inspecting the metadata) that an external assembly is actually an assembly type variable. For example, for a reference to `fdlA1` in CIL code, this `AssemblyRef` is created:

```
// AssemblyRef #2
// -----------------------------------------------------
//    Token: 0x23000002
//    Public Key or Token:
//    Name: fdlA1
```

```
//    Major Version: 0x00000000
//    Minor Version: 0x00000000
//    Locale: <null>
...
//    Assembly Variable: yes
```

For referenced classes that are defined in external assemblies, the assembler creates `TypeRef` metadata items. Since the assembler does not check the existence of a class, any class type variable automatically generates a `TypeRef` as would a "real" class. We do not augment a `TypeRef` structure with a field identifying it as a class type variable, and will rely on the `fdlC` naming convention to identify class type variables at run-time.

4 Implementing flexible dynamic linking in .NET

4.1 The Rotor system

Microsoft's *Shared Source Common Language Infrastructure* (SSCLI), known as Rotor, is a minimal implementation of the CLI specification [11] that runs on Windows, MacOS and UNIX. It omits many features of the commercial CLR: there are no Windows-specific APIs, and the JIT compiler and garbage collector are much simpler.

Rotor comprises 1.9 million lines of C++, C♯ and x86 assembler, in four categories:

- The execution engine.
- The Base Class Libraries, *i.e.* frameworks that expose the services of the execution engine, such as reflection and networking.
- The Platform Adaption Layer (PAL), which maps CLI features (*e.g.* threads) to operating system features. [8]
- The compiler and utilities toolchain, including a C♯ compiler, the ILASM assembler and ILDASM disassembler, utilities for viewing the Global Assembly Cache, creating and signing assemblies, *etc.*

Following §3, we have a three-part scheme to implement flexible dynamic linking in Rotor:

(i) Flexible CIL code, understood by ILASM.

(ii) Enhanced metadata, created by ILASM, understood by the execution engine.

(iii) JIT-substitution, silently interceding in the JIT-compilation process.

To implement 1) and 2), we enhance the ILASM assembler that processes CIL directives and code to produce a assembly. CIL files normally contain forward declarations about external assemblies that they reference, yet we

[8] There is one PAL for Windows and one for MacOS/UNIX.

do not require such declarations for assembly type variables. Instead, we identify assembly type variables as they appear in CIL code, and simulate the appropriate declaration. Each assembly type variable receives a first-class reference in the assembled file's metadata.

To implement 3), we enhance the execution engine where it JIT-compiles CIL code. An assembly executes normally until a method call instruction that features an assembly type variable is reached. We offer the user a choice of which assembly to substitute from assemblies registered on their machine. Resolving the class of an invoked method is fairly straightforward compared to assembly resolution, and we silently substitute class type variables. With an instruction's type variables now substituted, JIT-compilation proceeds as if no type variables had been present; the instruction is verifiable and can be translated to native code.

Figure 1 gives an overview of the implementation of 1), 2) and 3).

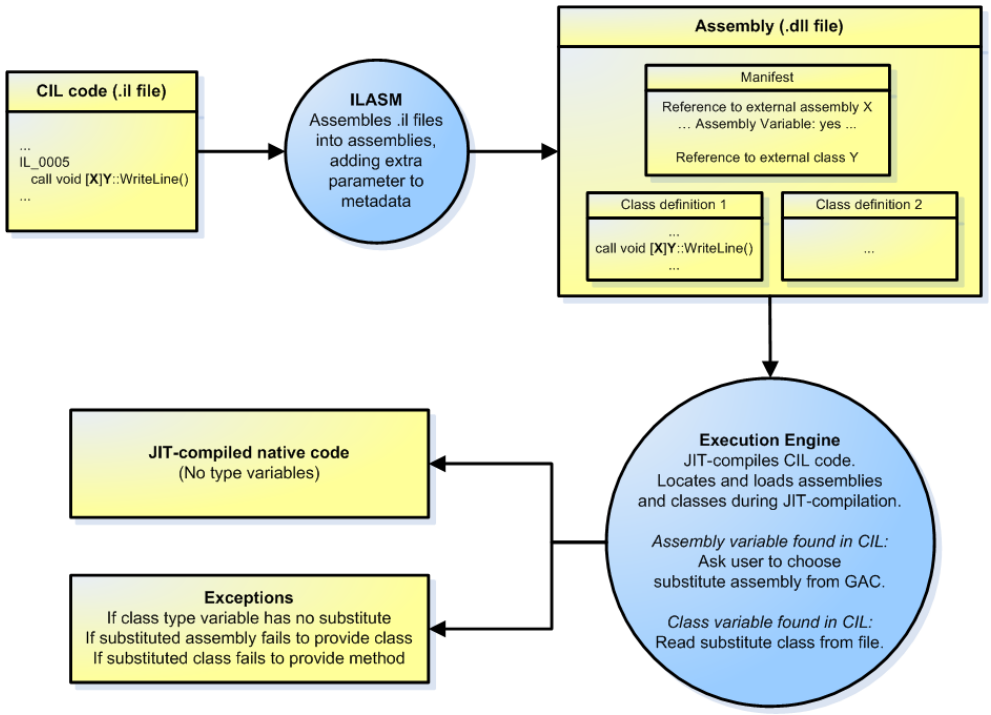

Fig. 1. Overview of flexible dynamic linking in Rotor

4.2 Enhancing the ILASM assembler

ILASM's first task in assembling a CIL file is to process directives, which are preceded by a period and indicate that the parser is about to encounter a major piece of data that will derive metadata. The directive that declares an external assembly, `.assembly extern`, plays a key role for us. Instances of this directive are normally inserted by a source language's compiler to identify which assemblies provide the classes used in the CIL code. Each external

assembly directive receives a corresponding `AssemblyRef` metadata item in the assembly file created by ILASM.

For each instance of `.assembly extern`, the `AsmMan` class (responsible for building an assembly's manifest) creates an `AsmManAssembly` object to encapsulate the name, version, public key and locale of the external assembly, and adds it to a list of external assemblies. Then, when a CIL method invocation, *e.g.*

```
IL_0005:  call void [fdlA1]fdlC1::WriteLine(string)
```

is parsed, `AsmMan` searches the list of external assemblies for an appropriate `AsmManAssembly` object. An assembly type variable like `fdlA1` is seen for the first time in a method invocation, as it has no corresponding `.assembly extern` directive. Therefore, it has no corresponding `AsmManAssembly` object in the list of external assemblies, so ILASM stops.

To allow ILASM to continue, we modify `AsmMan` so that if it meets an assembly type variable, it creates an `AsmManAssembly` object exactly as if there had been a `.assembly extern` directive. The `AsmManAssembly` object is added to the list of external assemblies, so that later CIL instructions understand the assembly type variable as an external assembly.

When all CIL code has been parsed, the assembler calls on `AsmMan` to emit an `AssemblyRef` metadata item for each `AsmManAssembly` object, including those representing assembly type variables. In addition, a `TypeRef` metadata item is emitted for the class referenced by the CIL instruction, whether it is a class type variable or not.

To help us, we modify `AsmMan`'s `StartAssembly` method in two ways: it receives an extra parameter, indicating an assembly type variable, and updates a new field `isVar` in the `AsmManAssembly` object with this parameter. Thus, when `AssemblyRef` metadata is created from an `AsmManAssembly` object, the metadata includes the "Assembly Variable:" flag shown earlier. Only our code, when recognising an assembly type variable, calls `AsmMan::StartAssembly` with the new parameter set to *true*. Because it has a default value of *false*, all other calls will emit metadata that does not flag an assembly type variable.

4.3 Enhancing the execution engine

A program whose assemblies feature type variables is started as normal, with the `clix` command-line tool. This tool initialises Rotor's execution engine and executes the `main()` method of the user's program. From then on, the heart of execution is the `FJit` class, which performs one-pass verification and JIT-compilation for each CIL instruction.

For a method call instruction, `FJit::compileCEE_CALL` is dispatched. This retrieves the method to be called from the CIL code, and passes it to `CEEInfo::findMethod` to load the right assembly (via the `Assembly` class) and extract the class and method definition (via the `ClassLoader` class).

`FJit` then verifies the method call instruction according to ECMA re-

quirements, *e.g.* checking that the called method is not abstract. Provided verification is successful, `FJit` emits native code and moves on to the next CIL instruction. The relationship among these classes is sketched in fig. 2.

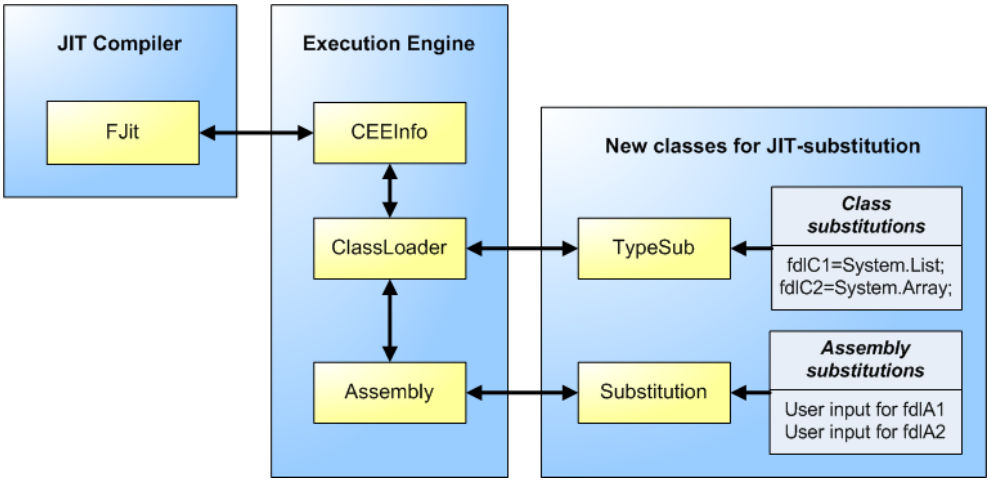

Fig. 2. Major classes supporting flexible dynamic linking in Rotor

4.3.1 Assembly resolution

`CEEInfo::findMethod` checks if the invoked method's class is already loaded, and if not, delegates the task of loading it to the classloader of the assembly currently being JIT-compiled. The `Classloader` class checks if the invoked class is defined in the same module (of the current assembly) as the class being JIT-compiled. If not, the `Assembly` class inspects the current assembly's metadata, hoping to find a `TypeRef` item that refers to the invoked class. If found, it indicates whether the class resides in a different module of the *same* assembly, or in an external assembly.

A class invoked in the scope of an assembly type variable (*e.g.* `call ... [fdlA1]...`) will always have a `TypeRef` item that indicates an external assembly. `Assembly` will look up the external assembly in the current assembly's metadata, and try to load it. This is the point where we can make a substitution for an assembly type variable. After a substitution is made, `Assembly` loads an assembly file from disk as usual.

4.3.2 Assembly substitution

The metadata returned to `Assembly` about an external assembly includes its name, public key and version number. An assembly type variable has no public key value; version numbers of the form `0000:0:0:0000`; and a name that does not, of course, identify a real assembly file. `Assembly` checks the `isVar` value in the metadata, and if it indicates a type variable, we choose and apply a substitution using our new class, `Substitution`.

`Substitution` contains a singleton array pointing to objects that store the name, public key, and version of each GAC assembly. To obtain infor-

mation about GAC-installed assemblies, we would like to use the "managed" `Reflection` API but cannot do so because we are in "unmanaged" C++ code inside the execution engine. We therefore borrow code from the `gacutil.exe` tool to call the assembly enumeration interface of the Fusion DLL. (Recall that Fusion is the assembly loading subsystem.)

To determine a substitution, the `Assembly` class passes the name, version and public key of the assembly type variable to `Substitution`. It presents the user with a list of possible substitute assemblies, *e.g.*

```
A substitution needs to be made for the assembly variable fdlA1.
The current assemblies available in the GAC are:
      1.    ISymWrapper
      2.    System
      3.    System.Xml
      4.    System.Runtime.Serialization.Formatters.Soap
      5.    System.Runtime.Remoting
      6.    Microsoft.Vsa
      7.    Microsoft.JScript
      8.    mscorlib
Please choose an assembly by entering 1-8:
```

The user enters a number and the corresponding object in `Substitution`'s array is returned to `Assembly`, which can now load an assembly from disk as if no substitution had occured.

4.3.3 Class resolution and substitution

Once the current assembly's classloader has successfully loaded an assembly, the desired class must be extracted. The classloader *for the newly loaded assembly* tries to do this in `ClassLoader::FindClassModule`. Each classloader has a hashtable of classes physically located within its assembly, set up when the classloader is initialised. This hashtable is checked; if the class is not found, a resolution exception `System.TypeLoadException` is thrown.

Just before the exception is thrown, we check if the class cannot be found because it is a class type variable, *i.e.* its name starts `fdlC`. If so, our `TypeSub` class reads a substitution from a text file in the same directory as the executing program; an example is:

```
fdlC1=System.Collections.Hashtable;
fdlC2=System.Collections.SortedList;
END
```

Assuming this file is present, correct and contains the necessary class type variable, we know what class should substitute for it. We manipulate the object that represents the class to be resolved, by changing the class to be found from a class type variable to its substitute class name - as if there had never been a type variable. For example, with the file above, a `NameHandle` object with an empty namespace and a class name `fdlC2` is updated to have

namespace `System.Collections` and class name `SortedList`. We check if the substitute class name is in the hashtable, meaning that the substitute class is in the assembly. If it is, then `ClassLoader::FindClassModule` succeeds.

Unwinding the call stack back to the JIT compiler in §4.3, `CEEInfo::findMethod` will succeed and `FJit::compileCEE_CALL` will finally be able to verify properties of the invoked method and emit native code.

5 Discussion

5.1 Observations on the Rotor codebase

The Rotor codebase is very large. The source directory for the CLI implementation comprises 2048 files, of which just over half are the execution engine and associated tools in C++. (The other half is the Base Class Library, written in C♯, and resource and make files.) This C++ code, some 750,000 lines, is the codebase we modified.

Because our changes are at such a low level, we only needed small amounts of code to affect all loading and linking operations. For the assembler, we modified under 20 lines of code and wrote under 50 new lines. This is in a codebase of 78,000 lines spread over three major directories. For the execution engine, we wrote under 400 new lines, in a codebase spread over five directories that comprises 334,000 lines. We benefited from the simpler JIT-compiler in Rotor (compared to the full CLR), which at 21,000 lines was much easier to understand than the 220,000-line virtual machine.

We note very different code paths for loading assemblies versus loading classes. This reflects the fact that the entire Fusion subsystem is needed to locate assemblies, validate them and extract their metadata. In contrast, once an assembly is loaded, it is easy to inspect a particular class within it.

Of the five "partitions" in the CLI specification [11], the partition about metadata is the largest, at 174 pages. By contrast, the architecture partition that defines the Common Type System and execution system is only 105 pages. Such size helps explain the massive complexity and size (60,000 lines) of the code to manage assembly metadata. This particular codebase needs a spring clean; we found that some functionality for reading metadata was duplicated by similar methods in different classes, which themselves implemented different but similar interfaces.

5.2 Representing type variables

Custom attributes allow metadata to be added to classes and methods in a non-intrusive way, then retrieved through the `Reflection` API available in C♯. They are usually thought of as source-level entities: they are defined as subclasses of `Attribute` and are applied to a class definition with the `[<attribute name>(<attribute parameters>)]` notation. In fact, they have a CIL representation and are encoded into the `TypeRef` metadata for a

class, held as part of the assembly it belongs to. The CIL representation of a custom attribute `MyAttr` applied to class `A` is:

```
.class A {
    .custom instance void Attribs.MyAttr ...
}
```

When ILASM meets an instruction with a type variable, it could create a custom attribute to identify the type variable. The custom attribute could be represented by adding a `.custom instance` directive to the class's CIL code, or by augmenting metadata items directly. Since we have avoided rewriting CIL code throughout, we would prefer to create custom attribute items directly in an assembly's metadata. Then, during compilation of method call instructions, the JIT compiler would perform reflection on custom attributes to identify which assembly/class type variables need to be substituted.

6 Related work

The use of type variables for abstracting over data types is well-known in the functional [22] and object-oriented [15,4] worlds. Recently, the introduction of type variables into bytecode at compile-time has been exploited to provide true separate compilation for Java-like languages [1].

Dynamic linking [13] is often formalised [7,14,8] because of its perceived complexity [9]. Most studies have targeted the JVM because of its greater familiarity and tenure, though [10] models both the JVM's and the CLR's dynamic linking strategies. We unified the use of type variables in bytecode with a model of dynamic linking in [5]. This paper reports the first implementation on any platform of a dynamic linker that supports type variables.

An aspect-oriented approach would help us separate the policy of choosing substitutions from the dynamic linking code that needs them. There are numerous AOP tools [16,6,23] to intercede in Java bytecode at fine granularities, both at compile-time and run-time. The .NET community prefers compile-time meta-programming that exploits custom attributes [20,2,3], though run-time CIL modification is possible [19].

7 Conclusion and Further work

This paper has presented the design and implementation of a scheme for very late binding of components in the CLI execution engine. This supports the execution of code in environments different from their compilation environments, a situation we expect to see more and more as code runs on a greater variety of devices. Type variables originate in CIL code (so are independent of the many source languages that run on the CLI) and are embedded into assembly metadata (so may be manipulated by a wide range of tools). At run-time, they guide the execution engine's dynamic linking subsystem to of-

fer the user a choice about which assembly/class to link. Our implementation is relatively straightforward, and platform-independent.

The obvious improvement to our scheme is to determine valid substitutions automatically. We would exploit techniques from compositional compilation [1] and refactoring [24], by equipping bytecode with constraints that describe the bytecode's dependencies on other assemblies and classes. For example, a constraint for some class might be that the assembly that substitutes for `fdlA1` must define a class `List` with method `add` of a certain signature. Either a compiler could generate constraints and store them in metadata, or the execution engine could generate them by scanning an assembly at load-time to determine how it uses (classes in) other assemblies. Then, the execution engine would separately scan the assemblies available in its execution environment, and at JIT-substitution, match constraints against the structure of those assemblies (and their classes, and those classes' members). We have already implemented this structural matching as part of a Java custom classloader that is aware of type variables.

We would like to make alternative uses of metadata, a compelling feature of the CLI. Our extra metadata required changing more code than we had expected, yet we barely scratched the surface of the metadata subsystem. In addition to using custom attributes to represent type variables, as in §5, we are investigating how a programmer can use custom attributes to control flexible dynamic linking. We plan to modify the C♯ compiler to process source-level attributes that scope either an entire assembly, a class or an individual field or method. Within its scope, an attribute specifies how referenced assemblies and classes should be substituted. This scheme also expands the set of "flexible" CIL instructions to include field access and update.

Acknowledgements

We thank the Imperial SLURP group for helpful discussions, and the anonymous reviewers for their comments.

References

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, CA, USA, January 2005.

[2] Giuseppe Attardi, Antonio Cisternino, and Diego Colombo. CIL + Metadata gt Executable Program. *Journal of Object Technology, Special issue: .NET: The Programmers Perspective: ECOOP Workshop 2003*, 3(2):19–26, February 2004.

[3] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Code Bricks: Code Fragments as Building Blocks. In *Proceedings of the 2003 SIGPLAN*

Workshop on Partial Evaluation and Semantic-based Program Manipulation (PEPM 2003), San Diego, CA, USA, June 2003.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'98)*, Vancouver, BC, Canada, October 1998.

[5] Alex Buckley and Sophia Drossopoulou. Flexible Dynamic Linking. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2004)*, Oslo, Norway, June 2004.

[6] Shigeru Chiba. Load-time Structural Reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *LNCS*, pages 313–336, Cannes, France, June 2000. Springer Verlag.

[7] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[8] Sophia Drossopoulou. An Abstract Model of Java Dynamic Linking and Loading. In Robert Harper, editor, *Proceedings of the Third International Workshop on Types in Compilation (TIC 2000)*, volume 2071 of *LNCS*, pages 53–84. Springer-Verlag, 2000.

[9] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Dynamic Linking. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002.

[10] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.

[11] ECMA. *Standard ECMA-335: Common Language Infrastructure.* ECMA International, December 2002.

[12] S. Eisenbach, V. Jurisic, and C. Sadler. Feeling the way through DLL Hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002.

[13] Michael Franz. Dynamic Linking of Software Components. *IEEE Computer*, pages 74–81, March 1997.

[14] T. Jensen, D. Le Metayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 4–15, Chicago, IL, USA, 1998.

[15] Andrew Kennedy and Don Syme. The Design and Implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001), Snowbird, UT, USA, June 2001.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, Budapest, Hungary, June 2001.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1999.

[18] Eric Meijer and John Gough. *Technical Overview of the Common Language Runtime*. Microsoft, 2000.

[19] Aleksandr Mikunov. Rewrite MSIL Code on the Fly with the .NET Framework Profiling API. *MSDN Magazine*, September 2003.

[20] Frank Piessens, Bart Jacobs, Eddy Truyen, and Wouter Joosen. Support for Metadata-driven Selection of Run-time Services in .NET is Promising but Immature. *Journal of Object Technology, Special issue: .NET: The Programmers Perspective: ECOOP Workshop 2003*, 3(2):27–35, February 2004.

[21] Robert Seacord and Lutz Wrage. Replaceable Components and the Service Provider Interface. Technical Note CMU/SEI-2002-TN-009, Carnegie Mellon/SEI, July 2002.

[22] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'93)*, pages 439–450, Charleston, SC, USA, 1993.

[23] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java Semantics via Bytecode Manipulation. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *LNCS*, pages 283–298, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[24] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2003)*, Anaheim, CA, USA, October 2003.