Common Compiler Infrastructure

A Quick Introduction

Contents

[Introduction 3](#_Toc227515960)

[What is CCI? 3](#_Toc227515961)

[Why CCI? 3](#_Toc227515962)

[Possible usage scenarios 4](#_Toc227515963)

[Uses of CCI in other projects 4](#_Toc227515964)

[Overview of CCI components 5](#_Toc227515965)

[Metadata model 5](#_Toc227515966)

[Code model 5](#_Toc227515967)

[A simple metadata example 7](#_Toc227515968)

[Host environment 7](#_Toc227515969)

[Assemblies and modules 7](#_Toc227515970)

[Loading an assembly 8](#_Toc227515971)

[Traversing an assembly 9](#_Toc227515972)

[The Main() method 9](#_Toc227515973)

[A sample output 10](#_Toc227515974)

Chapter 1

Introduction

# What is CCI?

The Common Compiler Infrastructure (CCI) is an integrated set of components encapsulating the common code comprising compiler front end and post-compilation tools operating on Common Language Runtime (CLR) assemblies. CCI subsumes several technologies currently used by other compilers and development tools for the .NET platform:

* Metadata APIs (e.g., IMetaDataEmit), except for unmanaged code support;
* System.Reflection and System.Reflection.Emit;
* System.CodeDom;
* Tools ilasm and ildasm.

CCI augments thosewith a unified framework for static analysis and (re)writing of assembly metadata and intermediate language (IL). It also allows the IL do be decompiled into source code, as well turn the source code back into IL.

# Why CCI?

Users of current language tools are presented with a bewildering set of interfaces to implement and no easy way to get up and run them quickly. CCI provides a set of base classes that give default implementations of all interfaces, and that users can grasp on quickly. When customization calls for, users need to override only what they require different and can do this in incremental fashion.

Apart from providing a lot of common boiler plate code, other important CCI features include efficient consumption and production of metadata and IL, standard but extensible intermediate representation, as well as standard but extensible visitor classes.

# Possible usage scenarios

Here is a list of some possible CCI usage scenarios:

* Writing a custom static analyzer operating on assembly metadata or IL;
* Rewriting assembly metadata or IL;
* Generating IL and metadata;
* Using CCI as a managed replacement for the IMetadata interfaces.

# Uses of CCI in other projects

CCI is used in a number of projects in Microsoft, such as:

* Code Contracts;
* FxCop;
* ILMerge;
* Sandcastle;
* Spec #;
* SpecExplorer.

Chapter 2

Overview of CCI components

CCI defines a set of highly factored classes that provide a faithful object model of CLR metadata and IL called the metadata model, and of common source elements and contracts called the code model. Both models are interface-based and by default provide immutable objects implementation. For each immutable object, however, a mutable copy can always be obtained, as well as mutable versions of objects can be used in the first place. Among other features, there is a good support for generics, modular analysis, concurrency and incremental changes.

# Metadata model

The major goal of the metadata model is to provide a unified view of CLR metadata and IL regardless of whether the purpose is to read an assembly or to generate one. The metadata model is defined in terms of interfaces for CLR metadata and IL structures, providing for each of them, along with a dummy one,a mutable and an immutable implementation. The metadata model comes packaged in the following assemblies:

* Microsoft.Cci.MetadataModel.dll
* Microsoft.Cci.MetadataHelper.dll
* Microsoft.Cci.MutableMetadataModel.dll

# Code model

Once a model has been created, it can be either be analyzed to determine if certain properties hold, transformed to another model, or written out as text or in some binary format such as a portable executable (PE) or a program database (PDB) file.

Analyzers typically traverse the model by means of a visitor that extends one of the standard base classes:

* BaseMetadataVisitor
* BaseMetadataTraverser
* BaseCodeVisitor
* BaseCodeTraverser
* BaseCodeAndContractTraverser

Transforming a model into another kind of model can also be done by means of a visitor. CCI includes visitors that transform a code model into a simplified code model and the simplified code model into a metadata model. The corresponding classes are:

* CodeModelNormalizer
* CodeModelToIL

Visitors, such as the CodeModelNormalizer, that produce a model that is largely the same as the original model are called mutators and extend one of the following base classes:

* MetadataMutator
* CodeMutator
* CodeAndContractMutator

A metadata model can be written out as a CLR module or assembly using PeWriter. If a corresponding source model is available it can be written out as PDB file using PdbWriter. Likewise, PeReader provides methods for reading metadata and IL from a PE file, while PdbReader maps offsets in the IL back to source locations.

Chapter 3

A simple metadata example

In order to illustrate usage of CCI metadata API, let us have a look at a simple custom static analyzer of .NET assemblies. The goal is to search an assembly for a generic method that lies within a generic type by using the CCI's API for reading metadata. Let the application take names of assemblies to search for from arguments passed on the command line.

# Host environment

CCI is designed in such a way that it abstracts over many standard operations and allows the consumer of the library to specify how to treat particular cases. For instance, by abstracting the file system, CCI uniformly deals with a PE file regardless of whether it comes from a file system/network or is just created in the memory. Further, it also abstracts away a unification policy by which one can control how the assemblies are unified (if they are unified at all), where the files are found and so on.

An application that hosts components providing or consuming objects from the CCI metadata model has to create a host environment that will abstract over particular standard operations. Since in our example we want to read metadata, the application will host the metadata reader. This will be accomplished by providing a class inheriting from MetadataReaderHost in which one specifies how the assemblies are being accessed.

# Assemblies and modules

In CCI, both assemblies and modules are put under a common notion of a unit that is represented via the IUnit interface. This interface represents a unit of metadata stored as a single artifact and potentially produced and revised independently from other units. The IModule interface (extending IUnit interface) represents a .NET module, while the IAssembly interface extending (IModule interface) represents a .NET assembly.

.NET assemblies and modules come persisted as files in PE format. CCI provides the PeReader class for reading metadata and IL from a PE file. It gives access to all information that exists in a PE file by populating the corresponding CCI interfaces. In particular, PeReader's OpenModule() method loads the module and returns an IModule object corresponding to the opened module (in case of an assembly, returned is an IAssembly object). The IModule interface gives access to properties of the module such as the assembly containing the module, a list of referenced assemblies and modules, type definitions, entry method, various flags, etc. The IAssembly interface in addition provides access to the assembly manifest (e.g., public key, security attributes, etc).

# Loading an assembly

Let us now show the HostEnvironment class extending MetadataReaderHost, a base class for an object to be employed by the application hosting the metadata reader. In this class, we have to define a method LoadUnitFrom() that returns a unit stored at the given location.

internal class HostEnvironment : MetadataReaderHost

{

PeReader peReader;

internal HostEnvironment()

: base(new NameTable(), 4)

{

this.peReader = new PeReader(this);

}

public override IUnit LoadUnitFrom(string location)

{

IUnit result = this.peReader.OpenModule(

BinaryDocument.GetBinaryDocumentForFile(location, this));

this.RegisterAsLatest(result);

return result;

}

}

We see that the method uses PeReader's OpenModule() to load the assembly by reading it as a binary file. It also registers the loaded assembly as the latest unit associated with its location so that it can be later discovered by clients. The MetadataReaderHost's constructor uses two parameters to construct an object that provides an abstraction over the application hosting compilers based on the framework: NameTable is a reusable implementation of the name table containing names that are commonly used during compilation, and 4 stands for the pointer size.

The assembly can now be loaded in our application by using an instance of the HostEnvironment class:

# Traversing an assembly

Once an IAssembly object representing the opened assembly is obtained, we can easily traverse it in order to find generic types and within them generic methods. Calling the GetAllTypes() method on this object returns all of the types defined in the assembly. The returned object is an enumerator over named type definitions, i.e., instances of INamedTypeDefinition. One can inspect whether a type is parameterized by checking its IsGeneric property.

In addition, the object representing a type contains a property Methods that returns an enumerator over methods defined by the type. A method is represented via an IMethodDefinition object which models the metadata representation of the method. One can see if the method has generic parameters by looking at its IsGeneric property.

In order to obtain a default C#-like string representation of an IMethodDefinition object one can just call its ToString() method. However, if one would like to have a finer-grained control over the formatting, a helper static method MemberHelper.GetMemberSignature() can be employed which allows a number of formatting options to be used.

The resulting code for traversing the assembly and printing out its generic methods contained within generic types would therefore look as follows.

foreach (INamedTypeDefinition type in assembly.GetAllTypes()) {

if (type.IsGeneric) {

foreach (IMethodDefinition methodDefinition in type.Methods) {

if (methodDefinition.IsGeneric) {

Console.WriteLine(MemberHelper.GetMemberSignature(

methodDefinition,

NameFormattingOptions.Signature |

NameFormattingOptions.TypeParameters |

NameFormattingOptions.TypeConstraints |

NameFormattingOptions.ParameterName

));

}

}

}

}

# The Main() method

Combining all pieces together, here is shown the complete source code of the Main() method.

static int Main(string[] args)

{

HostEnvironment host = new HostEnvironment();

foreach (string assemblyName in args) {

IAssembly/\*?\*/ assembly = host.LoadUnitFrom(assemblyName)

as IAssembly;

if (assembly == null || assembly == Dummy.Assembly) {

continue;

}

else {

Console.WriteLine("Generic Methods in generic types from '"

+ assembly.Name.Value + "':");

}

foreach (INamedTypeDefinition type in assembly.GetAllTypes()) {

if (type.IsGeneric) {

foreach (IMethodDefinition methodDefinition in type.Methods) {

if (methodDefinition.IsGeneric) {

Console.WriteLine(MemberHelper.GetMemberSignature(

methodDefinition,

NameFormattingOptions.Signature |

NameFormattingOptions.TypeParameters |

NameFormattingOptions.TypeConstraints |

NameFormattingOptions.ParameterName

));

}

}

}

}

}

return 0;

}

# A sample output

When run on the mscorlib assembly, the application produces the following output:

