Masaryk University
Faculty of Informatics

Custom Roslyn Tool for Real-Time Static Code Analysis

Master's Thesis

Zuzana Dankovčíková

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Custom Roslyn Tool for Real-Time Static Code Analysis

Master's Thesis

Zuzana Dankovčíková

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Zuzana Dankovčíková

Advisor: Bruno Rossi PhD

Acknowledgement

TODO: This is the acknowledgement…

Abstract

TODO: This is the abstract ...

Keywords

roslyn, C#, compilers, code review, .NET compiler platform, Kentico, analyzer, code fix..., …

Contents

List of Tables

List of Figures

1 Introduction

[1-2 pages]
TODO...
Ideas:

What is code quality, why is it important, tool that support it.. compilers, diversion ... aaaand here comes Roslyn which provides compiler as a platform.

In the .NET world, the compiler used to be a black box that given the file paths to the source text, produced an executable. In order to do that, compiler has to collect large amount of information about the code it is processing. This knowledge, however, was unavailable to anyone but the compiler itself and it was immediately forgotten once the translated output was produced [**roslyn-overview-github**].

Why is this an issue when for decades this black-boxes served us well? Programmers are increasingly becoming reliant upon the powerful integrated development environments (IDEs). Features like IntelliSense, intelligent rename, refactoring or "Find all references" are key to developers' productivity; and even more so in an enterprise-size systems.

This gave a rise to number of tools that analyze the code for common issues and are able to suggest a refactoring. The problem is that that such tool needs to parse the code first in order to be able to understand and analyze it. As a result companies need to invest fair amount of resources to duplicate the logic that the .NET compiler already possesses. Not only is it possible that the compiler and the tool may disagree on some specific piece of code, but with every new version of C# the tool needs to be updated to handle new language features[**dot-net-development-using-the-compiler-api**].

With roslyn.. etc. etc. .. API for analysis.. use in companies for custom analyzers... etc. etc.... https://github.com/dotnet/roslyn/wiki/Roslyn Overview – motivation Make sure to stress out that ".NET Compiler Platform" and "Roslyn" names will be used interchangably as it is in Roslyn Succinctly on page 11.

2 Code Quality and Static Code Analysis

TODO

3 .NET Compiler Platform

As per [**dragon-book**], compiler is a program that can read a program in a *source* language and translate it into a semantically equivalent program in a *target* language while reporting any errors detected in the translation process.

In the .NET world, the compiler used to be a black box that given the file paths to the source text, produced an executable. This perception was changed in 2015 when Microsoft introduced the .NET Compiler Platform (commonly referred to as "Project Roslyn").

Not only have been compilers for both Visual Basic and C# rewritten into the entirely managed code, they also expose the internals of the compiler pipeline via a public .NET API [1]. This makes them a platform (also known as *compiler-as-as-service*) with rich code analysis APIs that can be leveraged by developers to perform analysis, code generation or dynamic compilation in their own programs [**roslyn-succinctly**]. Those can be then easily integrated into the Visual Studio all without the hard work of duplicating compilers' parsing logic.

This chapter will take a look at how the Roslyn API layers are structured, how the original source code is represented by the compiler and how developers can build tools upon the compiler's API. Lastly it will provide a short overview and evaluation of other tools that were used before Roslyn or have emerged thanks to .NET Compiler Platform.

Note that although Roslyn provides equivalent APIs for both VisualBasic and C#, this thesis will only focus on the latter since only that one is relevant for the practical part of the thesis.

3.1 The Compiler Pipeline

Roslyn compilers expose an API layer that mirrors the traditional compiler pipeline (see 3.1). However, instead of a single process of generating the target program, each compilation step is treated as a separate component [**roslyn-overview**]:

1. Application Programming Interface

- **Parse phase** consists of *lexical analysis* (*scanner*) and *syntactic analysis* (*parser*). First, the lexical analyzer processes the stream of characters from the source program and groups them into meaningful sequences called *lexemes*. Those are subsequently processed by the *syntax analyzer* that creates a tree-like structure of tokens based on the language's grammar [**dragon-book**].
- **Symbols and metadata phase** where named symbols are generated based on the declarations from the source and imported metadata.
- **Bind phase** in which the identifiers from the source code are matched to their respective symbols.
- **Emit phase** where all the gathered information is used to emmit an assembly.

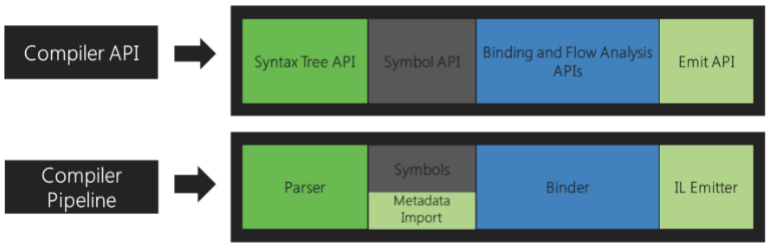

Figure 3.1: Compiler pipeline [**roslyn-overview**]

In each phase, the .NET Compiler Platform creates an object model containing gathered information and exposes it through the API in form of .NET objects. These object are also used internally by Visual Studio [2] to support basic IDE functionality. For instance **syntax tree**, that is the result of the parse phase, is used to support formatting and colorizing the code in the editor. The result of the second phase – **hierarchical symbol table**, is the basis for *Object browser* and *Navigate to* functionality. Binding phase is represented as an **object model that exposes the result of the semantic analysis** and is utilized in

2. The new generation of Visual Stuio leveraging from the Roslyn compiler are called vNext and first one was VS 2015.

Find all references or *Go to definition*. Finally, the Emit phase produces the Intermediate Language (IL) byte codes and is also used for *Edit and Continue* feature [**roslyn-overview**].

3.2 The .NET Compiler Platform's Architecture

The Roslyn's architecture consists of two main layers - Compiler and Workspaces APIs, and one secondary layer - Features API, as seen on Figure 3.2.

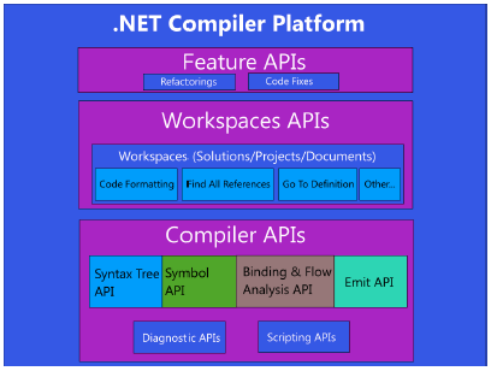

Figure 3.2: .NET Compiler Platform Architecture [**roslyn-succincly**]

One of the key concepts of .NET Compiler Platform is immutability. The compiler exposes hundreds of types that represent all information about source code from `Project` and `Document` to `SyntaxTrees` with almost all of those types being immutable. This means, that once created, the object cannot change. In order to alter it in any way, new instance must be created, either manually, or from an existing instance by applying one of many `With()` methods that the API provides.

The immutability enables the compiler to perform parallel work without need to create duplicate objects or apply any locks on them. This concept is useful for the command line compiler but it is considered extremely important for IDEs where it enables for one document to be handled by multiple analyzers in parallel.

3.2.1 The Compiler APIs

As discussed in the previous section, the Compiler APIs offer an object model representing the results of syntactic and semantic analysis produced by the respective phases of the compiler pipeline. Moreover, it also includes an immutable snapshot of a single compiler invocation, along with assembly references, compiler options, and source files. This layer is agnostic of any Visual Studio components and as such can be used in stand-alone applications as well. There are two separate, though very similar, APIs for Visual Basic and C#, each providing functionality tailored for specific language nuances.

Diagnostic APIs

Apart from parsing code and producing an assembly, the compiler is also capable of raising diagnostics, covering everything from syntax to semantics, and report them as errors, warnings or information messages [**roslyn-succinctly**]. This is achieved through the compilers' Diagnostics APIs that allow developers to effectively plug-in to compiler pipeline, analyze the source code using the exposed object models, and surface custom diagnostics along with those defined by the compiler itself. These APIs are integrated to both MSBuild [3] and Visual Studio (2015 and newer). providing seamless developer experience. The practical part of this thesis relies on Diagnostic APIs to provide custom diagnostics and the details will be discussed in Chapter 5.

Scripting APIs

As a part of the compiler layer, Microsoft team has introduced new Scripting APIs that can be used for executing code snippets. These APIs were not shipped with .NET Compiler Platform 1.0 and are part of v2.0.0 RC3[4].

3. The Microsoft Build Engine https://github.com/Microsoft/msbuild
4. Release candidate 3, as per https://github.com/dotnet/roslyn/wiki/Scripting-API-Samples [26-02-2017].

3.2.2 Workspaces APIs

Workspace represents a collection of solutions, projects, and documents. It provides a single object model containing information about the projects in a solution and their respective documents; exposes all configuration options, assembly and inter-project dependencies, and provides access to syntax trees and semantic models. It is a starting point for performing code analysis and refactorings over entire solutions.

Although it is possible to use the `Workspace` outside of any host environment, the most common use case is an IDE providing an instance of `Workspace` that corresponds to the open solution. Since the instances of `Solution` are immutable, the host environment must react to every event (such as user key stroke) with an update of the `CurrentSolution` property of the `Workspace`.

3.2.3 Feature APIs

This layer relies on both compiler and workspaces layers and is designed to provide API for offering code fixes and refactorings. This layer was also used while working on the practical part of this thesis.

3.3 Syntax Tree

As mentioned in the previous sections, the product of the syntactic analysis is a syntax tree. It enables developers to work with the code in a managed way instead of working against plain text. Syntax trees are used for both analysis and refactorings, where the new code is generated either manually or as a modified version of the existing tree. While being immutable, syntax trees are thread-safe and analysis can be done in parallel.

It is important to point out, that in a same way the compiler constructs a syntax tree from the source text, it is also possible to round-trip back to the text representation. Thus, the source information is always preserved in full fidelity. This means that every piece of information from source must be stored somewhere within the tree, including comments, whitespaces or end-of-line characters.

Figure 3.3 shows a syntax tree of an invocation expression as obtained from Syntax Visualizer[5] extension available in Visual Studio. This tool is useful for understanding how Roslyn represents particular language constructs and is widely utilized whenever one needs to analyze the code. Following sections explain what are the main building blocks of such syntax tree and will refer to this figure:

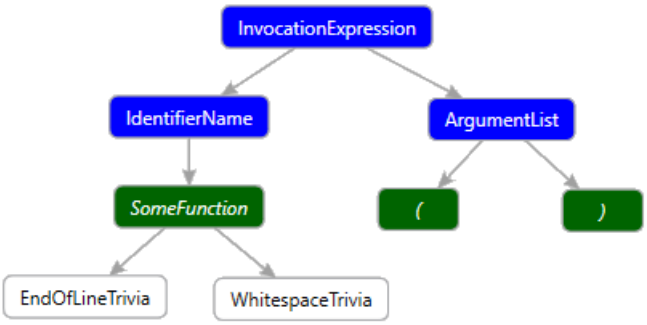

Figure 3.3: Syntax tree of an invocation expression

Syntax Nodes

Syntax nodes (blue color) are non-terminal nodes of a syntax tree, meaning they always have at least one other node or token as a child. Nodes represent syntactic constructs of a language such as statements, clauses or declarations. Each type of node is represented by a single class deriving from `SyntaxNode`. Apart from common properties `Parent`, `ChildNodes` and utility methods like `DescendantNodes`, `DescendantTokens`, or `DescendantTrivia`, each subclass exposes specific methods and properties. As shown in Figure 3.3, `InvocationExpression` has two properties, `IdentifierName` and `ArgumentList` both of which are `SyntaxNodes` themselves.

Syntax Tokens

As opposed to nodes, syntax token (green color) represent terminals of the language grammar, such as keywords, punctuation, literals and

5. https://roslyn.codeplex.com/wikipage?title=Syntax%20Visualizer

identifiers. For the sake of efficiency, `SyntaxToken` is implemented as a value type (C# structure) and there is only one for all kinds of tokens. To be able to tell them apart, tokens have `Kind` property. For example, `SomeFunction` is of kind `IdentifierName`, whereas `"("` character is `OpenParenToken`.

Syntax Trivia

In order to enable refactoring features, syntax trees must also store information about whitespaces, comments and preprocessor directives that are insignificant for compilation process itself. This information is represented by another value type – `SyntaxTrivia` (white color). Trivia are not really parts of the tree itself, rather they are properties of tokens accessible by their `LeadingTrivia` and `TrailingTrivia` collections.

3.4 Semantics of the Program

Even though syntax trees are enough to describe proper form of the program (compliance to the language grammar), they cannot enforce all language rules, for example, type checking. In order to tell whether a method is called with the right number of arguments, or operator is applied to operands or right type, it's inevitable to introduce semantics.

As described in [**dragon-book**], one of the core responsibilities of a compiler is to collect information about all elements and their properties from the source program. These are attributes such as identifier name, type, allocated storage, scope or for method names the number and types of arguments and their return values.

All this data is being incrementally collected when parsing the source code (analysis phase) and is stored in *symbol tables*. These are later used in synthesis where intermediate language representation is produced.

Symbols

In .NET Compiler Platform, a single entry of a symbol table is represented by a class deriving from `ISymbol`. The symbol represents every distinct element (namespace, type, field, property, event, method or

parameter) either declared in the source code or imported as metadata from a referenced assembly. Each specific symbol has its own methods and properties often directly referring to other symbols. For example `IMethodSymbol` has a `ReturnType` property specifying what is the type symbol the method returns.

Compilation

An important immutable type, that represents everything needed to compile a C# (or Visual Basic) program is a `Compilation`. It contains all source files, compiler options and assembly references. Compilation provides convenient ways to access any discovered symbol. For instance, it is possible to access the entire hierarchical symbol table rooted by global namespace or look up type symbols by their common metadata names.

Semantic Model

When analyzing a single source file of a compilation, all its semantic information is available through a *semantic model*. The `SmeanticModel` object can answer many questions such as:

- What symbol is declared at the specific location in the source?
- What is the result type of an expression?
- What symbols are visible from this location?
- What diagnostics are reported in the document?

This makes semantic model very useful when performing static code analysis concerned with more than just syntax.

3.5 Analysers and Code Refactorings

[3 pages?]

3.6 Other Tools

[2-3 pages?] TODO: Should this be a separate chapter????
3. Other tools for code analysis available for C#
- new DotNetAnalyzers available thanks to Roslyn
- Resharper,
- FxCop
- StyleCop

4 Kentico CMS Internal Guidelines

[3-4 pages]
What is Kentico CSM?
Current situation & motivation
- how code reviews are done at kentico
- tools that aid code reviews (??)
- original BugHunter
- use of FxCop and ReSharper

5 Implementation of Custom Analyzers

[5-7 pages]
How was the tool implemented,
Project structure,
What it contains
Concerns about Performance

6 Measuring and Optimizing the Performance

[up to 7 pages?]

Why tool needs to be super-fast (refer to chapter 4 where this should have been said)

Talk about /ReportAnalyzer switch of MSBuild process (csc.exe)

How the performance of the slowest analyzers (SystemIO, BaseChecks) was improved

Talk about how analyzers deployment influenced the build time

Questionares sent to development team, feedback from senior developers

7 Deployment and Versioning

[2 pages]

8 Conclusion

[1-2 pages]

A Questionnaires

TODO...