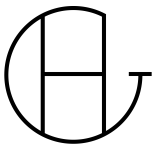

HoGent

Faculteit Bedrijf en Organisatie

Ensuring code quality through diagnostic analyzers on the Roslyn platform

Jeroen Vannevel

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Marc Van Asselberg
Co-promotor:
Schabse Laks

Instelling:

Academiejaar: 2015-2016

1e examenperiode

Faculteit Bedrijf en Organisatie

Ensuring code quality through diagnostic analyzers on the Roslyn platform

Jeroen Vannevel

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Marc Van Asselberg
Co-promotor:
Schabse Laks

Instelling:

Academiejaar: 2015-2016

1e examenperiode

Abstract

Foreword

Contents

1 Introduction **5**
- 1.1 What can Roslyn do? . . . 5
- 1.2 Why was Roslyn built? . . . 6
- 1.3 Positivity of an open compiler . . . 6
- 1.4 Positivity of compiler APIs . . . 7
- 1.5 Introduction to the paper . . . 7

2 Implementing a Diagnostic **8**

3 Implementing a Data Flow based Diagnostic **9**

4 Implementing a SyntaxWalker **10**

5 Roslyn internals **11**
- 5.1 Compiler phases . . . 12
- 5.2 Type structure . . . 12
  - 5.2.1 Syntax Tree . . . 12
  - 5.2.2 Syntax Node . . . 12
  - 5.2.3 Syntax Token . . . 12
  - 5.2.4 Syntax Trivia . . . 12
- 5.3 Red-Green trees . . . 12
  - 5.3.1 Green nodes . . . 12
  - 5.3.2 Red nodes . . . 12
- 5.4 Performance considerations . . . 12
  - 5.4.1 Concurrency . . . 12
  - 5.4.2 Small nodes . . . 12
  - 5.4.3 Object re-use . . . 12
  - 5.4.4 Weak reference . . . 12
  - 5.4.5 Object pooling . . . 12
  - 5.4.6 Specialized collections . . . 12
  - 5.4.7 LINQ . . . 12

5.5 Security Considerations 12
5.5.1 Deterministic builds 12

Chapter 1

Introduction

1.1 What can Roslyn do?

There are three major aspects that can be analyzed: the syntax, the semantic model and the data flow. The syntax, which is really just a collection of tokens that make up our source code, is the very basic level: no interpretation is done, we just make sure that the code that is written is valid syntactically. These validations are performed by confirming that the source code adheres to the rules specified in the C# Language Specification. Note however that there is no analyzing done of the actual meaning of these symbols thus far – we only determined that the textual representation is correct. The next level facilitates this aspect: it interprets the semantic model. This means that so-called 'symbols' are created for each syntactical construct and represent a meaning. For example the following code will be interpreted as a class declaration (through the type 'ClassDeclarationSyntax')

public class MyClass

while in this code, the 'MyClass' will be interpreted as an 'IdentifierNameSyntax':

var obj = new MyClass();

This shows us that the semantic model knows the meaning of each syntactic construct regarding its context. This allows for more extensive analysis since we can now also analyze the relationship between specific symbols. Last but not least: the third aspect of source code that we can analyze is its data flow in a specific region. The 'region' here refers to either a method body or a field initializer. This tells us whether or a variable is read from/written to inside or outside the specified region, what the local variables in this region are, etc. This is a step above "simple" semantic analysis: rather than interpreting the meaning of a symbol in the code, we can now analyze how certain symbols are interacted with without actually executing the code itself. For example consider the following method:

public void MyMethod() int x = 5; int y;

Analyzing this method's data flow and looking at the 'DataFlowAnalysis.AlwaysAssigned' property, would tell us that only 1 local here would be always assigned. However if we now alter the above code to this:

public void MyMethod() int x = 5; int y; if(true) s = ""My string"";

we now receive "2" as a result. This is an extremely powerful (albeit currently fairly limited) feature that allows us to actually interpret a code snippet's data flow without executing it.

1.2 Why was Roslyn built?

Now that we know what Roslyn does it's important to realize what gap is being closed. After all, something has to be special about Roslyn if it is to receive such widespread attention. Roslyn introduced two very big changes to the community: the C# compiler's language changed and it has now become open-source. Prior to Roslyn, the C# compiler was like a black box: everybody had access to the language specification so you could know what the rules of language were but the actual process of verifying your code against these rules was a black box. By open-sourcing Roslyn and placing it on CodePlex and afterwards Github, suddenly this box was opened and everybody could look at the internals of how a high-end compiler is written. This brings us to the other big change: because Roslyn is written in C# rather than C++, the C# compiler is now able to bootstrap itself. This means that Roslyn can compile itself in C# to compile the next versions of the Roslyn compiler! The process is simple: Roslyn is compiled using the old C++ compiler after which that newly created compiler re-compiles the Roslyn codebase and suddenly you have a C# compiler that is written in C#. An added benefit of rewriting the compiler is the fact you can "discard" the older one. [Fleshing out needed]

1.3 Positivity of an open compiler

Opening the compiler to the public brought a lot of good things, not in the least external contributions! Only 6 days after the Roslyn source was released, the first PR (Pull Request) was already integrated in the system!

[Add image of PR tweet]

There are many more benefits to open-sourcing: ▪ The amount of scrutiny the code goes through is multiplied now that the entire developer community has access to it. [Needs citation] ▪ Developers themselves can now more easily learn how compilers work in a language they're familiar with. Writing compilers has become more and more a "far-from-my-bed-show" with the rise of high-level languages – this might (re-)introduce some interest into such low-level concepts. ▪ It also facilitates advanced

users by allowing them to inspect the barebones of the compiler when something unexpected happens. Compilers are not bug free[http://stackoverflow.com/a/28820861/1864167] [http://stackoverflow.com/q/33694904/1864167] and implementations change over time[http://stackoverflow.com/a/30991931/1864167] which may lead to small but distinct differences in behaviour. An example that comes to mind is laid out in this Stack overflow question titled "Enums in lambda expressions are compiled differently; consequence of improvements?" where a user noticed that identical code compiles differently on the newer compiler version. ▪ Last but not least it simply provides a very robust and efficient parser that external parties now won't have to develop themselves. This doesn't necessarily mean that those tools should switch to the Roslyn platform but they might incorporate some ideas of it in their own parser (as is the case with ReSharper, a popular code-refactoring tool[https://blog.jetbrains.com/dotnet/2014/04/10/resharper-and-roslyn-qa/]).

1.4 Positivity of compiler APIs

Separately, providing APIs to your compiler brings benefits with it as well. Because of the broad support of APIs for all kinds of aspects of a compilation's lifetime, a very convenient service is created where external parties can easily hook into and create their own diagnostics using the information that comes straight from the compiler. This encourages new tools aimed at the platform since the hassle of writing your own parser is removed – developers can go straight to implementing business logic. On top of that, the "native" support means that there is a seamless integration with Visual Studio, Microsoft's own IDE (Integrated Development Environment). This integration allows you to use the results of your diagnostics to be displayed and used directly inside of Visual Studio rather than needing to build a third-party tool or a plugin. This is the essence of why Roslyn is regarded as "Compiler as a Service".

1.5 Introduction to the paper

Chapter 2

Implementing a Diagnostic

Chapter 3

Implementing a Data Flow based Diagnostic

Chapter 4

Implementing a SyntaxWalker

Chapter 5

Roslyn internals

5.1 Compiler phases

5.2 Type structure

5.2.1 Syntax Tree

5.2.2 Syntax Node

5.2.3 Syntax Token

5.2.4 Syntax Trivia

5.3 Red-Green trees

5.3.1 Green nodes

5.3.2 Red nodes

5.4 Performance considerations

5.4.1 Concurrency

5.4.2 Small nodes

5.4.3 Object re-use

5.4.4 Weak reference

5.4.5 Object pooling

5.4.6 Specialized collections [12]

5.4.7 LINQ

5.5 Security Considerations

5.5.1 Deterministic builds

List of Figures

List of Tables