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

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 { }
```

Listing 1.1: Interpretation of a class declaration

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

```
var obj = new MyClass();
```

Listing 1.2: Interpretation of an object instantiations

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;
    string y;
}
```

Listing 1.3: A method with definite assignments

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;
    string y;
    if(true)
        y = "My␣string";
}
```

Listing 1.4: A method with conditional assignments

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 out of your reach. By open-sourcing Roslyn and placing it on CodePlex[1] and afterwards Github[2], suddenly this box was opened and everybody could look at the internals of how a high-end compiler is written.

[1] https://roslyn.codeplex.com/

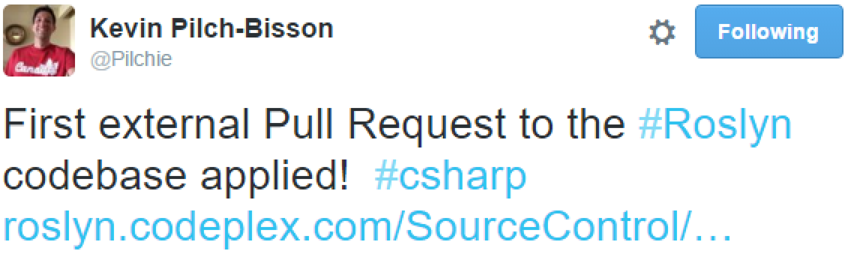
[2] https://github.com/dotnet/roslyn/

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 code base 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.

It should be noted here that Roslyn is also VB.NET its new compiler and as such, VB.NET-specific parts are written in VB.NET. You will also notice that every unit test that applies to both C# and VB.NET is duplicated in both languages. In this paper, however, we will focus on the C# aspects.

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!



Kevin Pilch-Bisson
@Pilchie

First external Pull Request to the #Roslyn codebase applied! #csharp roslyn.codeplex.com/SourceControl/…

Figure 1.1: Announcement of the first PR by Kevin Pilch-Bisson[3]

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 bare bones of the compiler when something unexpected happens. Compilers are not bug

[3] https://twitter.com/pilchie/status/453968834408763392

free[4][5] and implementations change over time[6] 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

[TODO]

[4] http://stackoverflow.com/a/28820861/1864167

[5] http://stackoverflow.com/q/33694904/1864167

[6] http://stackoverflow.com/a/30991931/1864167

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

At its core, Roslyn creates tree representations from any given piece of code. Whether this concerns a single statement or an entire class doesn't matter – in the end, everything is reduced to a tree data structure. We can distinguish 2 different kinds of elements that make up the tree and a 3rd element that augments additional information.

5.2.1 Syntax Tree

The syntax tree in itself is the starting point from which a lot of aspects start or end: in a CodeFix you can end up replacing the original tree with the modified one while it might as well be the starting point with which you interpret plain text as something you can inspect.

Creating a syntax tree can be as easy as parsing plain text:

```
var tree = CSharpSyntaxTree.ParseText(@"
public␣class␣Sample
{
}");
```

which will result in the tree as shown in figure 5.1

A syntax tree represents at the highest level a single top-level type (class or struct) with all its containing members – fields, methods and nested types. Multiple syntax trees can be grouped in a compilation. It is however not a requirement to represent a type when parsing a syntax tree: any kind of statement or expression can be parsed and represented in a syntax tree. In the end, every syntax node represents a tree of itself.

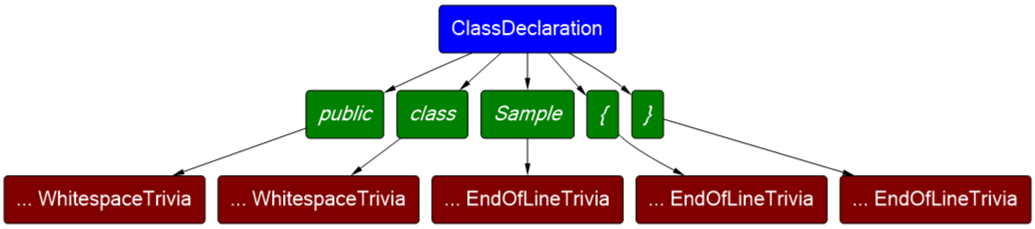


Figure 5.1: Syntax tree representation of a class definition

This idea of trees comprised of other trees is what enables interesting performance optimizations (more on that later).

Syntax trees have a few very important properties:

- Full fidelity – The guarantee that every character, whitespace, comment, line ending, etc from the source code is represented in the syntax tree
- Round-trippable – Following out of the above: translating the source code to a syntax tree and back results in exactly the same code
- Code as written, not as executed – When you use the 'async' and 'await' keywords it will represent these keywords in the syntax tree and not the state machine that the compiler generates out of it
- Immutable – Every syntax tree is immutable. This allows for thread-safe operations since everything handles its own "copy" of the syntax tree anyway

5.2.2 Syntax Node

A syntax node is an element of the tree that is always the parent to other nodes or to tokens (also known as 'non-terminal') and represents an expression or a statement.

Syntax nodes are defined in the Syntax.xml[1] file which generates classes that represent the relationship between all kinds of syntax nodes.

[More info on syntax.xml]

5.2.3 Syntax Token

Syntax tokens are what make up the individual, small pieces of the code. Whereas syntax nodes mainly represent the syntactic constructs of the language, syntax tokens

[1] https://github.com/dotnet/roslyn/blob/master/src/Compilers/CSharp/Portable/Syntax/Syntax.xml

are the building blocks that these syntactic constructs consist of. A syntax token can be a keyword, identifier, literal and punctuation and only exists as a leaf in the tree: it will never be the parent of another node or token.

Contrary to syntax nodes, syntax tokens are not represented by separate classes. Instead, a single struct exists which can be differentiated based on its 'Kind' property.

[Look it up]

5.2.4 Syntax Trivia

The last aspect of the tree is the syntax trivia. This is slightly misleading because it's not actually a part of the tree: it is stored as a property on a syntax token. This stems from the idea that syntax tokens could not have child nodes and as such it is implemented by adding it to a token's 'LeadingTrivia' and 'TrailingTrivia' properties.

Syntax trivia is the rest of the source code with no actual semantic meaning: comments, whitespace, newlines, etc. Important to note are preprocessing directives: these are also considered trivia.

It is important to represent the trivia inside the syntax tree in order to ensure full-fidelity: if we wouldn't, we would have a differently layed out source code after translating our source to a syntax tree and back.

5.3 Red-Green trees

5.3.1 The problem

We already know that we can use the syntax tree to traverse a node's descendants and we have also seen that we can look at a node's ancestors. Additionally we also know that every node is immutable: once created it cannot be changed. This introduces an interesting dilemma: how do you create a node that has knowledge of its parent *and* its children at the moment it's created? As we just established we can't create child- or parent-nodes afterwards and then modify the node since that would violate the immutability principle.

Another consideration stemming from this is performance related: node objects will never be able to be re-used because you can't change certain aspects such as the parent it refers to (again coming from that immutability requirement).

5.3.2 The solution

The solution to these problems came in the form of the so-called red-green trees. The basic idea is straightforward: Roslyn will create two trees, a red and a green one. The green nodes will contain "vague" information such as their width instead of their start-

and end-position and they won't keep track of their parent. This tree is built from the bottom-up: this means that when a node is built, it knows about its children. When an edit occurs two things happen: a green node is created alongside a trivial red node (a node without a position or parent). The key here is that this red node also contains a reference to the underlying green node[2] which will allow us to construct a red node at *some point in time* using this green node's data. The emphasis is very important here: a red node is materialized lazily – this means that the red tree itself is only created when it's actually requested. For example when you browse through code in the Visual Studio IDE[3], it will create these red trees for each part of the code that the user can actually see and which needs highlighting. Code that the user cannot see doesn't need to be highlighted and as such does not need to be materialized into a red tree.

When the red tree is built, Roslyn works with a top-down approach. This allows us to discover the exact location of every node by starting from position 0 and incrementing the position by the width of each subsequent node which we store in every red node's underlying green node. We also know the underlying green node's children because we built that green node from the bottom-up. This provides us with all the information we need to create a new red node at a certain position with a given parent and the necessary contextual information. Indeed, when we look at the constructor of a 'SyntaxNode' (which is what we consider a 'red node') we can see it has the following definition:

```
internal SyntaxNode(GreenNode green,
                    SyntaxNode parent,
                    int position)
```

Listing 5.1: Constructing a (red) SyntaxNode

This perfectly displays the principle we talked about of creating a red node based on its parent, the position (which is calculated from the cumulative width of elements preceding it) and the associated green node.

A performance consideration worth mentioning here is the fact that generation of the red tree is cancelled when an edit occurs. Any analysis performed until that moment is outdated (or possible to be outdated) so there is no point in finishing it up. The red tree at this point is discarded and a new green tree is generated based on the newly typed characters. In reality this green tree will be resolved into a red tree *almost* immediately – there is a slight delay between typing and generation of the red tree so there won't be a constant stream of newly created and aborted trees while the user is typing rapidly.

The colours "red" and "green" have no particular meaning – they were simply the two colours used to describe the idea on the Roslyn team's blackboard when designing

[2] http://source.roslyn.codeplex.com/#Microsoft.CodeAnalysis/Syntax/SyntaxNode.cs,41

[3] IDE: Integrated Development Environment, the editor in which you modify source files

this approach.

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

5.4.7 LINQ

5.5 Security Considerations

5.5.1 Deterministic builds

List of Figures

List of Tables