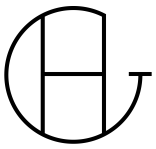

HoGent

Faculteit Bedrijf en Organisatie

# Ensuring code quality through diagnostic analyzers on the Roslyn platform

Jeroen Vannevel

Thesis submitted in partial fulfillment of the requirements for the degree of bachelor applied computer science

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

Thesis submitted in partial fulfillment of the requirements for the degree of bachelor applied computer science

Promotor:
Marc Van Asselberg
Co-promotor:
Schabse Laks

Instelling:

Academiejaar: 2015-2016

1e examenperiode

## Abstract

# Foreword

# Contents

# Chapter 1

# Introduction

## 1.1 What is Roslyn?

## 1.2 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[1]. 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`)

```
1 public class MyClass { }
```

Listing 1.1: Interpretation of a class declaration

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

```
1 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

[1] https://msdn.microsoft.com/en-us/library/ms228593.aspx

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:

```
1    public void MyMethod()
2    {
3        int x = 5;
4        string y;
5    }
```

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:

```
1    public void MyMethod()
2    {
3        int x = 5;
4        string y;
5        if(true)
6            y = "My string";
7    }
```

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.

## 1.3 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[2] and afterwards Github[3], 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 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.

## 1.4 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!


Figure 1.1: Announcement of the first PR by Kevin Pilch-Bisson[4]

There are many more benefits to open-sourcing:

- The amount of scrutiny the code goes through is greater now that the entire developer community has access to it.
- 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.

[2] https://roslyn.codeplex.com/
[3] https://github.com/dotnet/roslyn/
[4] https://twitter.com/pilchie/status/453968834408763392

- It also facilitates advanced users by allowing them to inspect the bare bones of the compiler when something unexpected happens. Compilers are not bug free[5][6] and implementations change over time[7] 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 "Why does Roslyn crash when trying to rewrite this lambda?"[8] where a user noticed that identical code compiles differently on the newer compiler version. Merely hours later, another user had identified the issue, discovered the offending commit in the Roslyn repository and submitted a PR with a fix.

- 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).(Gorohovsky 2014)

## 1.5 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[9]. 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".

## 1.6 Introduction to the paper

[TODO]

[5] http://stackoverflow.com/a/28820861/1864167
[6] http://stackoverflow.com/q/33694904/1864167
[7] http://stackoverflow.com/a/30991931/1864167
[8] http://stackoverflow.com/a/34085687/1864167
[9] IDE: Integrated Development Environment, the editor in which you modify source files

# 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

### 5.1.1 Parsing phase

### 5.1.2 Declaration phase

### 5.1.3 Binding phase

### 5.1.4 Emit phase

## 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:

```
1 var tree = CSharpSyntaxTree.ParseText(@"
2 public class Sample
3 {
```

```
4 }");
```

which will result in the tree as shown in figure 5.1

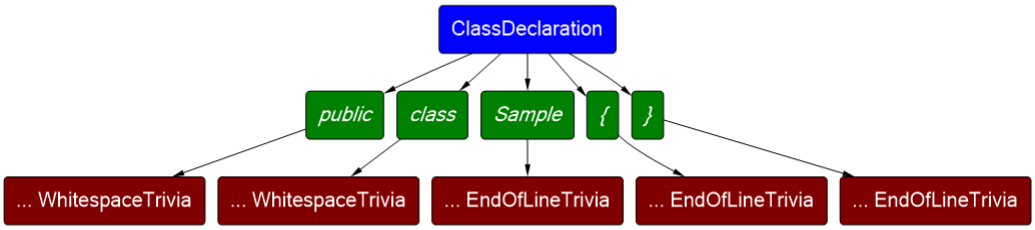


Figure 5.1: Syntax tree representation of a class definition

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. 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 from 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 from 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 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

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

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, 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:

```
1 internal SyntaxNode(GreenNode green,
2                     SyntaxNode parent,
3                     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.

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

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 this approach.(Lippert 2012)

## 5.4 Performance considerations

Performance is an integral aspect of any application and as such Roslyn doesn't escape from it either. It's not just about compiling an assembly as fast as possible: that's just one of the usecases in which Roslyn is used. Think about other scenarios such as getting quick intellisense[3] or fluent syntax highlighting of text as you scroll but also affects other performance aspects such as the memory impact.

In this section we will look at a handful of techniques the Roslyn team uses to optimize the platform. These are general approaches that often apply their ideas in several areas of the codebase, sometimes through a common resource. This should indicate that these are often optimizations at a high level rather than very specific single-use optimizations.

### 5.4.1 Concurrency

> A concurrent system is one where a computation can make progress without waiting for all other computations to complete – where more than one computation can make progress at "the same time"
>
> *Abraham Silberschatz*
> *Operating System Concepts 9th edition*

When you think of performance, concurrency is often one of the first aspects that come to mind. As such, it also takes a prominent place in Roslyn's architecture.

[3]Intellisense: context-aware hints as you type

We have already established that one of the characteristics of a syntax tree is its immutability – the inability to make changes to it after it is constructed. This is done with concurrency in mind: if we create a new tree for every concurrent operation, we have the guarantee that changes from operation X does not affect the tree that is being manipulated by operation Y. There are several more areas of concurrency though:

### Source parsing

Every file containing source code is parsed independently of any other files. The file is parsed sequentially (from top to bottom) and multiple files can be parsed at once.(Sadov 2014)

### Symbol binding

When identifiers are bound to symbols (see Chapter 5.1.3) there is no real need to do this sequentially: in the end you basically just lookup the symbol in a table according to an identifier. One remark we have to add here though is the fact that a type's base members should also be bound when that type is being bound.

Consider the following example:

```
1  class BaseType
2  {
3    public virtual void Method() { }
4    public void BaseMethod() { }
5  }
6
7  class SubType : BaseType
8  {
9    public override void Method()
10   {
11     base.BaseMethod();
12   }
13 }
14
15 class AnotherType
16 {
17   public void Method() { }
18 }
19
20 class StartUp
21 {
22   public static void Main(string[] args)
23   {
```

```
24      BaseType a = new SubType();
25      a.Method();
26      AnotherType b = new AnotherType();
27      b.Method();
28    }
29  }
```

Listing 5.2: Why type hierarchy matters for binding

When we bind `SubType` we have to bind `BaseType` as well because, as is indicated, there might be a semantical dependency: we have to know, for example, whether that `override` keyword is appropriate there. This is not a problem when there is no explicit base type involved: we know the base type is just `System.Object` which is likely already bound and as such there are no other types we have to investigate. For this reason we can bind unrelated symbols in no specific order.
We can say there is "implicit partial ordering".(Sadov 2014)

#### Method body compilation

Method bodies are bound and emitted on a type-by-type basis and in lexicographical order (based on the alphabet). This order is important: when compiling identical code multiple types it should emit the same result every time. Sometimes compiling a method creates an additional structure such as a state machine (as is the case with async/await and iterator blocks). This idea of a 'deterministic build' is looked at more closely in Chapter 5.5.1.

Important to note is that method bodies are not emitted if there were any declaration errors. If that is the case, only binding is done for those aspects that can help in diagnosing the issue.(Sadov 2014)

### 5.4.2 Small nodes

Another performance-oriented aspect is the so-called "memory-footprint"[4]. When an application is executed, it uses memory to temporarily store data that makes up the program's workflow. This memory is (unfortunately) not unlimited: every device only has a certain amount available. The more memory is used, the less memory becomes available for other tasks and the sooner you run out. C# and VB.NET are managed languages which means the allocating and freeing of memory is done for the developer. The freeing of memory is done through the "Garbage Collector" (GC), a specialized service that will free unused memory locations when it deems it to be necessary. Discussing the GC is outside of the scope of this paper but it is important to realize that

[4]Memory footprint: The amount of memory software uses when running

the more memory we use, the more the GC will have to step in and free up some of the unused memory locations so we can re-allocate these.(Todorov 2013)

Knowing that, we can take a look at the next performance aspect. As we know, syntax nodes are some of the most elemental constructs in our AST[5]. Considering the sheer amount of nodes that a large applications exists of, it would pay off to keep these as small as possible. Doing so would reduce the amount of memory allocated which in turn would reduce the amount of garbage collections. A garbage collection is a relatively expensive operation since you basically put a hold on all active threads (except for the GC thread) so the GC can do its work.(Botelho 2014)

For this reason the Roslyn team decided to store certain information related to syntax nodes in a place that is not associated with that specific syntax node's object.(Sadov 2014) "Meta-information" like diagnostic info and annotations are the two prime examples of this. In any given code base, most syntax nodes won't have any diagnostic information attached to them nor do they have annotations. This means that providing a field on every single syntax node object to store this information would be a waste of memory: in the end, a field with a `null` value will still use some memory due to its pointer. For a 32-bit process this will be a 4-bit pointer while a 64-bit process will use an 8-bit pointer[6]. Multiply this by all the relevant syntax nodes and you reach a sizable amount of wasted memory space.

What Roslyn does instead is store the diagnostics inside the syntax node's associated syntax tree and when the diagnostic information is actually requested, it will look them up inside that tree.
This becomes clear when we take a look at the execution path in the code:

```
1 public new IEnumerable<Diagnostic> GetDiagnostics()
2 {
3   return this.SyntaxTree.GetDiagnostics(this);
4 }
```

Listing 5.3: CSharpSyntaxNode.GetDiagnostics

which passes it on to the following chain of calls inside the syntax tree:

```
1 GetDiagnostics(node.Green, node.Position);
2 EnumerateDiagnostics(greenNode, position);
3 new SyntaxTreeDiagnosticEnumerator(this, node, position);
```

Listing 5.4: CSharpSyntaxTree.GetDiagnostics

We can see a new specific enumerator is created which will traverse the syntax tree's nodes and return any diagnostics it finds. That is not the end, however. Eventually we still have to look up the diagnostic information and we know it's not stored inside the syntax node object. We follow the execution path again and now it becomes clear:

[5]AST: Abstract Syntax Tree
[6]https://msdn.microsoft.com/en-us/library/system.intptr.size(v=vs.110).aspx

```
1 node.GetDiagnostics(); // node is a GreenNode
```

Listing 5.5: SyntaxTreeDiagnosticEnumerator.MoveNext

```
1 if (s_diagnosticsTable.TryGetValue(this, out diags))
```

Listing 5.6: GreenNode.GetDiagnostics

and eventually we reach a statically defined look-up table where we connect given syntax nodes with their respective diagnostic information:

```
1 private static readonly
2   ConditionalWeakTable<GreenNode, DiagnosticInfo[]>
3     s_diagnosticsTable =
4       new ConditionalWeakTable< GreenNode,
5                                 DiagnosticInfo[]>();
```

Listing 5.7: GreenNode.s_diagnosticsTable

This approach comes with the caveat that when there *are* diagnostics or annotations, retrieving them will be more expensive compared to just accessing a field. A trade-off had to be made here between permanent extra memory-usage and occassional extra look-up time and apparently the former was deemed more important.

### 5.4.3 Object re-use

We've already highlighted the importance of memory impact in section 5.4.2 by making the nodes as small as possible. There is another approach we can take to this by making *duplicate* nodes take absolutely no space: we simply re-use existing objects. In section 5.4.3 we will mimic the experiments done by Robin Sedlaczek(Sedlaczek 2015) based on an explanation by Vladimir Sadov(Sadov 2014). In the end we will have shown how we reach a red-tree façade as shown in image 5.2 while having an underlying structure as shown in image 5.3.

#### Re-using nodes

The first aspect we will look at is re-using our green nodes. As we established in section 5.3.2, green nodes have "vague" information: instead of a specific location in the syntax tree they instead contain the width of themselves. This is good because if it would have a specific location, we would (almost) never be able to re-use it: two distinct nodes in a single tree will be located at a different position. However by using the width instead we *can* consider two similar nodes as identical because we merely look at their width which is the same for both.

We can demonstrate this in the following example where we look at the node that represents 2 + 2.

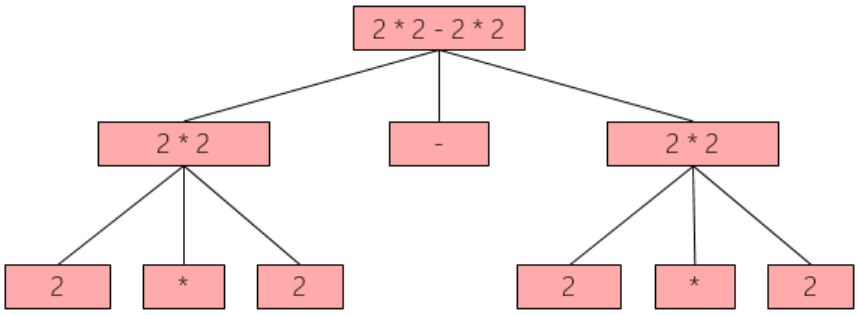


Figure 5.2: Red-tree façade ©Robin Sedlaczek

```
1 int x = 2 + 2; // Width: 5; Position: 9
2 int y = 2 + 2; // Width: 5; Position: 23
```

Listing 5.8: Re-using nodes based on width vs position

Following out of this and the fact that green nodes don't contain any other uniquely identifying information such as a parent, we can conclude that a single green node can represent multiple identical subtrees. In order to see how this is implemented we have to take a look at the `SyntaxNodeCache`[7].

There are three main conclusions that we can take away from this implementation:

- Limited cache size

  Only a limited amount of nodes is stored in this cache. A trade-off has to be made between execution time and memory impact and the Roslyn team has decided to use a cache size of 65536 items (`1 << 16`) as evidenced here:

```
1 private const int CacheSizeBits = 16;
2 private const int CacheSize = 1 << CacheSizeBits;
3 private const int CacheMask = CacheSize - 1;
```

Listing 5.9: Definition of the SyntaxNodeCache's size

- First In, First out

  When a node is added, a hash key will be calculated. As is typical with hash implementations, this should strive to reach a uniform distribution across the data structure (an array in this case) for optimal performance. Based on this hash key and the mask, the to-be-cached node will be inserted at a certain location of

[7] `https://github.com/dotnet/roslyn/blob/b908b05b41d3adc3b5e81f8cf2d0055c13e4a1f6/src/Compilers/CSharp/Portable/Syntax/InternalSyntax/SyntaxNodeCache.cs`

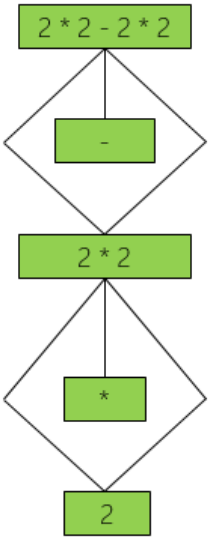


Figure 5.3: Underlying green tree nodes ©Robin Sedlaczek

the array. If an entry already exists at this location it will be overwritten, hence the FIFO (First In, First Out) principle.

```
1 var idx = hash & CacheMask;
2 s_cache[idx] = new Entry(hash, node);
```

Listing 5.10: Inserting a node using FIFO

- Up to 3 children

  The last important aspect of this cache is the fact that every node can only have up to three children. If it has any more, the chance of a cache miss is too big(Sadov 2014) so these are not allowed in the first place. This is enforced by making sure there are only overloads available for 1, 2 and 3 nodes.

```
1  private static bool CanBeCached(GreenNode child1)
2  {
3    return  child1 == null || child1.IsCacheable;
4  }
5
6  private static bool CanBeCached(GreenNode child1,
7                                  GreenNode child2)
8  {
9    return  CanBeCached(child1) &&
10           CanBeCached(child2);
11 }
```

```
12
13 private static bool CanBeCached(GreenNode child1,
14                                  GreenNode child2,
15                                  GreenNode child3)
16 {
17   return  CanBeCached(child1) &&
18           CanBeCached(child2) &&
19           CanBeCached(child3);
20 }
```

Listing 5.11: Caching up to three children

### Re-using tokens

Tokens are also cached but take a slightly different approach to doing so. When the source text is being parsed, the `QuickScanner` looks inside the `LexerCache` which maintains caches for both trivia and tokens.

```
1  if (state == QuickScanState.Done)
2  {
3    // this is a good token!
4    var token = _cache.LookupToken(
5        TextWindow.CharacterWindow,
6        TextWindow.LexemeRelativeStart,
7        i - TextWindow.LexemeRelativeStart,
8        hashCode,
9        _createQuickTokenFunction);
10   return token;
11 }
```

Listing 5.12: QuickScanner.QuickScanSyntaxToken

Inside the `LexerCache` the caches for trivia and tokens are stored by way of a `TextKeyedCache` implementation. A `TextKeyedCache` maintains two levels of caching: a local one and a shared one, each with their own characteristics.
L1 Cache:

- Small cache size ($2^{11}$ items)
- Fast
- Thread-unsafe
- Local to the parsing session

L2 Cache:

- Larger cache size ($2^{16}$ items)
- Slower
- Thread-safe
- Shared between all parsing sessions

When an item is searched for in the cache, the `TextKeyedCache` will first attempt to find it in the local cache (`_localTable`) and if it wasn't found there, look in the shared cache (`s_sharedTable`).

```
internal T FindItem(char[] chars,
                    int start,
                    int len,
                    int hashCode)
{
  var idx = LocalIdxFromHash(hashCode);
  var text = _localTable[idx].Text;

  if (text != null && arr[idx].HashCode == hashCode)
  {
    if (StringTable.TextEquals( text,
                                chars,
                                start,
                                len))
    {
      // Return from local cache
    }
  }

  SharedEntryValue e = FindSharedEntry( chars,
                                        start,
                                        len,
                                        hashCode);
  if (e != null)
  {
    // Return from shared cache
  }
}
```

Listing 5.13: TextKeyedCache.FindItem

### Measurements

Some really awesome stuff here. Wait and see.

### 5.4.4 Weak references

> A weak reference, simply put, is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself.
>
> *Ethan Nicholas*
> *Understanding Weak References*[8]

As we talked about before in section 5.4.2, C# is a managed language that manages its memory usage with the GC. In essence it boils down to this: if an object has no references it is eligible for Garbage Collection and the GC will mark it as such so it can be cleaned up for the next iteration. You might have guessed by the title of the section that the above explanation should actually specify it as *strong references* rather than just *references*. A "strong reference" is the default reference type we typically use when referencing another object.

A `WeakReference` is a construct that allows us to reference an object but if the GC decides to clean up that reference it can do so without a problem. You notice from this description that it is perfectly suited for a caching mechanism: that's basically the same result as when we create a cache with a certain size – the size is supposed to prevent us from reaching memory issues.

In section 5.3.2 we already talked shortly about syntax highlighting in the IDE: scrolling through the editor will cause the syntax tree to become materialized which means creating new objects to represent those syntactic constructs. However once a part of the tree leaves the window of the user there is no point in keeping it materialized anymore since the user can't see it anyway. At this point you want those unnecessary materialized nodes to be garbage collected but there is one major problem: nodes are referenced by their parent and its children. If we would use traditional strong references we would never be able to collect any of these objects.

One major area in which this technique is used is when it comes to a method body. Method bodies are in essence a group of statements. While it can be very useful to have information such as the method's definition available while programming, the exact contents of that method are less important – this makes it a perfect target to use weak references.

This is implemented in the `SyntaxList.WithManyWeakChildren` type. It works very straightforward: by storing an array with as type a `WeakReference<SyntaxNode>`

[8] http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html

we are already done: now these syntax nodes will be able to be GC'd if the GC marks them and no strong references exist to those particular nodes.

```
private readonly
  ArrayElement<WeakReference<SyntaxNode>>[] _children;
```

Listing 5.14: SyntaxList.WithManyWeakChildren._children

In the base type `SyntaxList.WithManyChildrenBase` we can see how the creation of the red node is delegated:

```
internal override SyntaxNode CreateRed(SyntaxNode parent,
                                       int position)
{
  var p = parent;
  if (p != null && p is CSharp.Syntax.BlockSyntax)
  {
    var gp = p.Parent;
    if (gp != null &&
        (gp is CSharp.Syntax.MemberDeclarationSyntax ||
         gp is CSharp.Syntax.AccessorDeclarationSyntax))
    {
      return new
        SyntaxList.WithManyWeakChildren(this,
                                        parent,
                                        position);
    }
  }

  if (this.SlotCount > 1 && HasNodeTokenPattern())
  {
    return new
      SyntaxList.SeparatedWithManyChildren(this,
                                           parent,
                                           position);
  }
  else
  {
    return new
      SyntaxList.WithManyChildren(this,
                                  parent,
                                  position);
  }
}
```

Listing 5.15: SyntaxList.WithManyWeakChildrenBase.CreateRed

### 5.4.5 Object pooling

In chapter 5.4.3 we've discussed a specific approach at minimizing the amount of objects created. In this section we will take a look at a more general implementation of this idea. While nodes and tokens are at the very core of the Roslyn code base, they're not the only objects we'll create.

When creating a lexer, one of the major tasks you're performing is reading source code and creating strings. If you would intermittently concatenate strings, this would increase memory pressure by a lot: every concatenation using + creates a new string but you might only be interested in the result[9]. However creating a new `StringBuilder` object every time you want to concatenate strings is an expensive operation as well: since a lot of string concatenation is done, a lot of distinct `StringBuilder` instances will be created and you again have the issue as described before. The solution here is to take pooling one step further: every type can be pooled through the `ObjectPool<T>` class.(Warren 2014) This provides a generic implementation of the object pooling pattern and will be used through intermediate classes.

An accurate description of this class' workings can be found in its documentation[10]:

> Generic implementation of object pooling pattern with predefined pool size limit. The main purpose is that limited number of frequently used objects can be kept in the pool for further recycling.
>
> Notes:
>
> 1) it is not the goal to keep all returned objects. Pool is not meant for storage. If there is no space in the pool, extra returned objects will be dropped.
>
> 2) it is implied that if object was obtained from a pool, the caller will return it back in a relatively short time. Keeping checked out objects for long durations is ok, but reduces usefulness of pooling. Just new up your own.
>
> Not returning objects to the pool in not detrimental to the pool's work, but is a bad practice. Rationale: If there is no intent for reusing the object, do not use pool - just use "new".

An interesting addition here is the implementation of `SharedPools`. As the name indicates, this is a class that manages the sharing of object pools. There are two ways to use this class:

- As a single, standalone pool

[9] https://msdn.microsoft.com/en-us/library/2839d5h5(v=vs.110).aspx

[10] `https://github.com/dotnet/roslyn/blob/b908b05b41d3adc3b5e81f8cf2d0055c13e4a1f6/src/Compilers/Core/SharedCollections/ObjectPool%601.cs`

This does not share the pool but instead just makes it easier to create your object pools. For this purpose you can choose between a small pool (20 elements) and a big pool (100 elements). For example in the case of a small pool you call

```
public static ObjectPool<T> Default<T>()
                                    where T : class,
                                              new()
{
  return DefaultNormalPool<T>.Instance;
}
```

Listing 5.16: SharedPools.Default<T>

This method, in turn, will lazily create a new pool the first time a pool of that type is requested. This pool will defer to the beforementioned `ObjectPool` implementation:

```
private static class DefaultNormalPool<T>
                                    where T : class,
                                              new()
{
  public static readonly ObjectPool<T> Instance =
    new ObjectPool<T>(() => new T(), 20);
}
```

Listing 5.17: SharedPools.DefaultNormalPool<T>

- As a shared pool

  Taking the entire idea another step further, you can also share the pools themselves. This is useful to share data inside the pools: if in one parsing session you create a string "using" then it would be nice to re-use that in another parsing session. This follows the same idea of the statically shared `TextKeyedCache` pool as we talked about in section 5.4.3 but expands this usage across all types due to its generic nature. Here again we see two separate usages:

  As a pre-defined pool

  Inside the `SharedPools` we have a few pre-defined pools for general use cases. One of those is `StringIgnoreCaseHashSet` which provides a hashset pool that compares its elements case-insensitively.

```
public static readonly
  ObjectPool<HashSet<string>> StringIgnoreCaseHashSet =
  new ObjectPool<HashSet<string>>(
    () => new HashSet<string>(
```

```
5 |                 StringComparer.OrdinalIgnoreCase), 20);
```

Listing 5.18: SharedPools.StringIgnoreCaseHashSet

As a pre-defined type

There are also specific types that provide some type-specific behaviour. An example of this is the `StringBuilderPool`:

```
1  internal static class StringBuilderPool
2  {
3    public static StringBuilder Allocate()
4    {
5      return SharedPools.Default<StringBuilder>()
6                        .AllocateAndClear();
7    }
8
9    public static void Free(StringBuilder builder)
10   {
11     SharedPools.Default<StringBuilder>()
12                .ClearAndFree(builder);
13   }
14
15   public static string ReturnAndFree(
16                             StringBuilder builder)
17   {
18     SharedPools.Default<StringBuilder>()
19                .ForgetTrackedObject(builder);
20     return builder.ToString();
21   }
22 }
```

Listing 5.19: StringBuilderPool

### 5.4.6 Specialized collections

### 5.4.7 LINQ

## 5.5 Security Considerations

### 5.5.1 Deterministic builds

# Bibliography


Botelho, Levi (2014). *[C#] How does the garbage collector work?* URL: http://www.levibotelho.com/development/how-does-the-garbage-collector-work/ (visited on 12/06/2015).

Gorohovsky, Jura (2014). *ReSharper and Roslyn: QA*. URL: https://blog.jetbrains.com/dotnet/2014/04/10/resharper-and-roslyn-qa/ (visited on 12/03/2015).

Lippert, Eric (2012). *Persistence, Facades and Roslyn's Red-Green Trees*. URL: http://blogs.msdn.com/b/ericlippert/archive/2012/06/08/persistence-facades-and-roslyn-s-red-green-trees.aspx (visited on 12/03/2015).

Sadov, Vladimir (2014). *Roslyn's performance*. URL: https://roslyn.codeplex.com/discussions/541953 (visited on 12/05/2015).

Sedlaczek, Robin (2015). *Inside the .NET compiler platform – performance considerations during syntax analysis*. URL: http://robinsedlaczek.com/2015/04/29/inside-the-net-compiler-platform-performance-considerations-during-syntax-analysis-speakroslyn/ (visited on 12/08/2015).

Todorov, Tsvetomir Y. (2013). *Understanding .NET Garbage Collection*. URL: http://www.telerik.com/blogs/understanding-net-garbage-collection (visited on 12/06/2015).

Warren, Matt (2014). *Roslyn code base – performance lessons (part 2)*. URL: http://mattwarren.org/2014/06/10/roslyn-code-base-performance-lessons-part-2/ (visited on 12/16/2015).

# List of Figures

# List of Tables