**MASTER THESIS**

Petr Onderka

# System for extensions of the C# language

Department of Software Engineering

| | |
|---|---|
| Supervisor of the master thesis: | RNDr. Filip Zavoral, Ph.D. |
| Study programme: | Computer Science |
| Study branch: | Software and Data Engineering |

Prague 2018

Dedication.

Title: System for extensions of the C# language

Author: Petr Onderka

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: Abstract.

Keywords: key words

# Contents

# Introduction

Extensibility is an important feature of a programming language and its associated programming environment, because it allows adding new capabilities to the language. This way, a programmer can mold the language to their specific needs.

A common way to extend many languages is through libraries. Libraries let programmers use code written by someone else, which can be powerful, especially when combined with extensibility features built into programming languages, such as virtual functions or lambdas.

But libraries are restricted to the features offered by the used language, which can limit their usefulness.

This work describes a system for extending the C# language beyond what can be accomplished with libraries by user-provided extensions, which perform transformations of C# source code.

These extensions, themselves written in C#, should be easy to create, when compared with existing similar systems, and they should be efficient enough to be usable with code completion in a code editor or an integrated development environment (IDE), such as Microsoft Visual Studio.

To achieve this, the system is composed of two primary parts: the Syntax tree API (Application programming interface) and the Transformations API.

The Syntax tree API is used to represent the original C# source code, examine it, and modify it. The primary goal of this API is to be easy to use and abstract syntax trees fit that requirement well.

The Transformation API enhances the Syntax tree API by adding methods that split source code transformation into smaller parts. The inputs and outputs of each part are then tracked, which means that, after an initial full execution of the transformation, only the parts of the transformation whose inputs changed have to be re-executed. This is done to improve the performance of the system, especially when run from an IDE.

# 1. Background

The .NET ecosystem is composed of programming languages (including C#, F# and Visual Basic .NET), [1] .NET implementations (including .NET Framework, .NET Core and Mono), [2] class libraries, commonly distributed through the NuGet package manager, and tooling, including command-line tools and tools integrated into code editors and IDEs.

What unifies all these components is the Common Language Infrastructure (CLI), [3] which specifies binary file format for "assemblies". These contain compiled .NET code in the form of Intermediate Language (IL) and also metadata associated with this code.

The C# language [4] is an object-oriented programming language which is part of the .NET ecosystem. The C# compiler, code named "Roslyn", [5] compiles C# source code into a .NET assembly. The compiler can also be used as a class library, which exposes types for programmatically manipulating C# source code.

An assembly, produced by the C# compiler or in some other way, can be executed on a .NET implementation. Each .NET implementation contains a runtime, which is responsible for executing code, and a base class library, which contains basic types used by .NET programs.

Runtimes of .NET implementations are usually using a just-in-time (JIT) compiler, which converts the IL for each method into machine code specific for the current instruction set just before executing that method for the first time.

In the .NET ecosystem, class libraries, which are just .NET assemblies, are commonly distributed thorough the NuGet package manager, [6] because it makes using those libraries easier. And while NuGet is primarily used for regular libraries, which are directly used by programmers in their source code, it can also be used for various kinds of special libraries, such as add-ins for general metaprogramming systems like Fody (more on these in section 1.5), or Roslyn analyzers for detecting issues with source code.

The C# language contains some basic extensibility features itself, namely virtual methods and delegates. But for more advanced use cases, it is necessary ot manipulate code in some form and the .NET ecosystem has various approaches to achieve that, including those that manipulate C# code, those that manipulate IL and those that use a custom model for representing code. Some of these approaches will be described in following sections.

## 1.1 Example

To demonstrate various code generation approaches, a running example of generating a simple entity class with a set of properties and having `IEquatable<T>` as an implemented interface will be used. (To make the examples shorter, the `Equals` method required to properly implement the interface will not be included.)

For example, for an entity named `Person` with properties `Name` of type `string` and `Age` of type `int`, the generated code should be similar to the one in Listing 1.

If an approach supports transforming C# code, not just generating it, the example will instead be transforming simple classes containing only fields with

```csharp
using System;

class Person : IEquatable<Person>
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Listing 1: Running example result

```csharp
public static class EntityKinds
{
    // used for generating code
    public static IEnumerable<EntityKind> ToGenerate { get; }

    // used for transforming code
    public static string ToGenerateFromSource { get; }
}

public class EntityKind
{
    public string Name { get; }
    public IReadOnlyList<Property> Properties { get; }
}

public class Property
{
    public string Type { get; }
    public string Name { get; }
    public string LowercaseName { get; }
}
```

Listing 2: Running example data source

the right types and names into the form above. Such transformation is too simple to be useful in practice, but it is sufficient as a demonstration.

All examples will use the `EntityKinds` class and related types as their data source. Their relevant parts are shown in Listing 2.

## 1.2 Manipulating C# source code

This section describes various approaches for generating and transforming C# source code. The resulting code then needs to be compiled by the C# compiler, as usual.

### 1.2.1 T4

Text Template Transformation Toolkit (T4) [7] is a tool for generating text by interspersing snippets of the text to generate with fragments of C# code to control how the text is generated. The resulting text can be in any language, including C#.

```
<#@ assembly name="System.Runtime.dll" #>
<#@ assembly name="$(TargetDir)CSharpE.Samples.Core.dll" #>
<#@ import namespace="CSharpE.Samples.Core" #>
<#@ output extension=".cs" #>
using System;

<# foreach (var entityKind in EntityKinds.ToGenerate) { #>
class <#= entityKind.Name #>
    : IEquatable<<#= entityKind.Name #>>
{
<# foreach (var property in entityKind.Properties) { #>
    public <#= property.Type #> <#= property.Name #> { get; set; }
<# } #>
}
<# } #>
```

Listing 3: T4 example

T4 does not have any special way of accessing other source code, which makes it most suitable for generating code based on external data. Its text-based nature gives it flexibility, but also makes using it fairly hard, since generating C# code effectively requires writing two interleaved programs, without any help from the IDE, because T4 integration into Visual Studio is very limited.

#### Example

The code to generate entities using T4, as required by the running example, can be seen in Listing 3.

The example shows that a T4 file contains text blocks written in the target language (in this case, C#), but also T4 directives and control blocks, enclosed in `<#` and `#>`. T4 directives provide information to the tool, for example which assemblies to reference or what the file extension of the output file should be. Control blocks can contain C# statements, which decide how the text blocks they surround execute, or C# expressions, which parametrize these text blocks.

The example code highlights another issue with T4: indentation. It is hard to keep indentation of both the generated code and the generating code consistent, especially since any whitespace outside of T4 tags will be included in the output.

### 1.2.2 CodeDOM

Code Document Object Model (CodeDOM) [8] is a library for generating source code by using a language-independent object model. It is fairly easy to use, but is limited in what language features it supports, due to its language-independent nature and due to it not being updated since .NET Framework 2.0. Some of these limitations can be worked around by using string-based “snippet” objects, but using them means negating the advantages that CodeDOM has. Some examples of features it does not support are declaring `static` classes, LINQ query expressions or declaring auto-implemented properties.

### Example

The code to generate entities for the running example using CodeDOM is in Listing 4.

As can be seen from the example, generating code using CodeDOM requires first building the object model. Its structure resembles the structure of a C# code file: a namespace (`CodeNamespace`) contains a class (`CodeTypeDeclaration`), which has base types and contains fields (`CodeMemberField`) and properties (`CodeMemberProperty`), which contain `get` and `set` accessors. The `get` accessor contains a `return` statement (`CodeMethodReturnStatement`), which contains an expression referencing a field (`CodeFieldReferenceExpession`). In several cases, object initializers are used to make the code to create these objects more readable. The resulting model is then converted to C# code using `CSharpCodeProvider`.

Especially notice that the code has to manually generate backing fields for properties, because CodeDOM does not support auto-implemented properties.

### 1.2.3 Roslyn

Roslyn [5] is the C# (and Visual Basic .NET) compiler, which can also be used as a library for manipulating C# code, including parsing, transformation and generation. Its object model was primarily designed to be used in the Visual Studio IDE, which is why it is very detailed (so it can accurately represent any source code, including erroneous or incomplete code) and also immutable (so that multiple IDE services can operate on the same model).

Roslyn contains several related Application programming interfaces (APIs) for manipulating source code, each useful in different situations:

- The `SyntaxTree` API forms the basis of Roslyn and represents only syntactic information about code. This means it can be used for just a single source file and makes it very efficient, especially when creating the `SyntaxTree` for code that contains only a small change relative to another `SyntaxTree`.

  On the other hand, no semantic information is available from this API, so for example for the expression `F(A.B)`, it is not possible to determine whether `F` is a method or a delegate, whether `A` refers to a type or a variable, or whether `B` is a field, a property, or a method group.

  The `SyntaxFactory` class can be used to create new nodes for this API.

- The `SemanticModel` class can be used to answer semantic questions about some part of a `SyntaxTree`.

  The disadvantage of using this class is that it requires a full compilation, which includes all files in a project and also all of its dependencies. It is also less efficient, especially when a change is made.

  The `SemanticModel` class surfaces semantic information in two forms:

  - The `ISymbol` API can be used to get semantic information about members declared or referenced by a piece of code.

    For example, for the expression `Console.WriteLine(42)`, this API could return the symbol for the `System.Console.WriteLine(int)`

```csharp
var ns = new CodeNamespace();
ns.Imports.Add(new CodeNamespaceImport("System"));

foreach (var entityKind in EntityKinds.ToGenerate)
{
    var entityType = new CodeTypeDeclaration(entityKind.Name);
    entityType.BaseTypes.Add(new CodeTypeReference(
        "IEquatable", new CodeTypeReference(entityKind.Name)));

    foreach (var property in entityKind.Properties)
    {
        var propertyType = new CodeTypeReference(property.Type);

        entityType.Members.Add(new CodeMemberField
        {
            Name = property.LowercaseName, Type = propertyType
        });

        var fieldReference = new CodeFieldReferenceExpression(
            new CodeThisReferenceExpression(), property.LowercaseName);

        entityType.Members.Add(new CodeMemberProperty
        {
            Attributes = Public | Final,
            Name = property.Name,
            Type = propertyType,
            GetStatements =
            {
                new CodeMethodReturnStatement(fieldReference)
            },
            SetStatements =
            {
                new CodeAssignStatement(fieldReference,
                    new CodePropertySetValueReferenceExpression())
            }
        });
    }

    ns.Types.Add(entityType);
}

var compileUnit = new CodeCompileUnit { Namespaces = { ns } };

using (var writer = new StreamWriter("Entities.cs"))
{
    new CSharpCodeProvider().GenerateCodeFromCompileUnit(
        compileUnit, writer, null);
}
```

Listing 4: CodeDOM example

method. That symbol could then be used to find out more semantic information about that method, like the assembly it is contained in.

  – The `IOperation` API is an alternative representation of statements and expressions as a language-independent abstract syntax tree. It includes semantic information in the form of `ISymbol` objects.

- The `SyntaxGenerator` class offers an alternative, language-independent way of generating Roslyn syntax nodes. It is part of the Workspaces layer of Roslyn, which means that using it on its own requires some additional setup. `SyntaxGenerator` has a more semantic view of code, which can be easier than generating the exact syntax using `SyntaxFactory`.

  Since `SyntaxGenerator` is language-independent, it is using `SyntaxNode` in its API to represent any kind of syntax node, because `SyntaxNode` is the common base class for syntax node types in different languages. This approach makes `SyntaxGenerator` less type-safe when compared with `SyntaxFactory`, which uses specific `SyntaxNode`-derived types in its API.

**Example**

The code to transform entities for the running example using the `SyntaxTree` API can be seen in Listing 5.

The example first parses the input code into a `CompilationUnitSyntax`. Then it replaces each class (`ClassDeclarationSyntax`) with a modified version that has an added base type (`BaseTypeSyntax`) and which replaces each field (`FieldDeclarationSyntax`) with a property (`PropertyDeclarationSyntax`). In the end, whitespace is added to the new nodes before writing the result to a file.

Note that this code is heavily using `SyntaxFactory` to create new syntax nodes, but that type is not visible due to use of `using static`, to make the code more succinct.

Also notice how immutability makes transforming code harder by requiring the use of methods such as `ReplaceNodes` and how the level of detail leads to very verbose code, in some cases requiring even the specification of individual semicolons.

Code to transform entities for the running example using `SyntaxGenerator` can be seen in Listing 6.

Using `SyntaxGenerator` requires some setup. The first step is to create a workspace (`AdhocWorkspace`) containing a project (`Project`) containing a document (`Document`). The next step is to access the root node and semantic model (`SemanticModel`) for the document and compilation (`Compilation`) for the project. The semantic model and compilation will be used to access symbols declared in the document and referenced by the project, respectively.

The overall structure of the rest of the code is similar to the `SyntaxTree` example, except that nodes are created and modified using the `SyntaxGenerator`. Another difference is that `SyntaxGenerator` does not support auto-implemented properties, so the example has to create properties with backing fields.

Also note that `SyntaxGenerator` generates overly verbose code (for example, `global::System.IEquatable<global::Person>`), but this is counterweighted

```
var compilationUnit = ParseCompilationUnit(EntityKinds.ToGenerateFromSource);

compilationUnit = compilationUnit.ReplaceNodes(
    compilationUnit.DescendantNodes().OfType<ClassDeclarationSyntax>(),
    (_, classDeclaration) =>
    {
        classDeclaration = classDeclaration.AddBaseListTypes(
            SimpleBaseType(QualifiedName(IdentifierName("System"),
                GenericName("IEquatable").AddTypeArgumentListArguments(
                    IdentifierName(classDeclaration.Identifier)))));

        classDeclaration = classDeclaration.ReplaceNodes(
            classDeclaration.ChildNodes().OfType<FieldDeclarationSyntax>(),
            (__, fieldDeclaration) =>
            {
                var type = fieldDeclaration.Declaration.Type;
                var name = fieldDeclaration.Declaration.Variables.Single()
                    .Identifier;

                return PropertyDeclaration(type, name)
                    .AddModifiers(Token(PublicKeyword))
                    .AddAccessorListAccessors(
                        AccessorDeclaration(GetAccessorDeclaration)
                            .WithSemicolonToken(Token(SemicolonToken)),
                        AccessorDeclaration(SetAccessorDeclaration)
                            .WithSemicolonToken(Token(SemicolonToken)));
            });

        return classDeclaration;
    });

compilationUnit = compilationUnit.NormalizeWhitespace();

File.WriteAllText("Entities.cs", compilationUnit.ToString());
```

Listing 5: Roslyn `SyntaxTree` example

```csharp
var project = new AdhocWorkspace().AddProject("MyProject", LanguageNames.CSharp)
    .AddMetadataReference(
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location));
var document =
    project.AddDocument("Entities.cs", EntityKinds.ToGenerateFromSource);
var g = SyntaxGenerator.GetGenerator(document);

var rootNode = await document.GetSyntaxRootAsync();
var model = await document.GetSemanticModelAsync();
var compilation = model.Compilation;

rootNode = rootNode.ReplaceNodes(
    rootNode.DescendantNodes().OfType<ClassDeclarationSyntax>(),
    (_, classDeclaration) =>
    {
        SyntaxNode result = classDeclaration;

        result = g.AddBaseType(result, g.TypeExpression(
            compilation.GetTypeByMetadataName("System.IEquatable`1")
                .Construct(model.GetDeclaredSymbol(classDeclaration))));

        var fields = result.ChildNodes().OfType<FieldDeclarationSyntax>();

        result = result.RemoveNodes(fields, default);

        foreach (var fieldDeclaration in fields)
        {
            var type = fieldDeclaration.Declaration.Type;
            var propertyName = g.GetName(fieldDeclaration);
            var fieldName = propertyName.ToLowerInvariant();

            var field = g.WithName(fieldDeclaration, fieldName);

            var fieldAccess = g.IdentifierName(fieldName);
            var property = g.PropertyDeclaration(
                propertyName, type, Accessibility.Public,
                getAccessorStatements: new[]
                {
                    g.ReturnStatement(fieldAccess)
                },
                setAccessorStatements: new[]
                {
                    g.AssignmentStatement(
                        fieldAccess, g.IdentifierName("value"))
                });

            result = g.AddMembers(result, field, property);
        }

        return result;
    });

document = document.WithSyntaxRoot(rootNode.NormalizeWhitespace());
document = await Simplifier.ReduceAsync(document);

File.WriteAllText(
    "Entities.cs", (await document.GetSyntaxRootAsync()).ToString());
```

Listing 6: Roslyn `SyntaxGenerator` example

by being able to use another Workspace-layer service, `Simplifier`, to make the code simpler.

TODO: More obscure libraries like RoslynDOM

## 1.3 Manipulating IL

This section describes various libraries that can be used for generating and transforming IL code.

Since IL is primarily meant to be produced by compilers and consumed by .NET runtimes, it is not a very convenient language for programmers.

### 1.3.1 System.Reflection.Emit

System.Reflection.Emit [9] is a library that can be used to generate an assembly at the IL level in memory, and then either directly execute code from that assembly or save the assembly to disk.

#### Example

Code to generate entities for the running example using Reflection.Emit can be seen in Listing 7.

Since Reflection.Emit directly manipulates IL code, it does not use documents or compilation units. Instead, the code creates an assembly (`AssemblyBuilder`) containing a module[1] (`ModuleBuilder`), containing types (`TypeBuilder`).

At the IL level, a property is just a named grouping of methods and the code reflects that: to create the equivalent of an auto-generated property, the code creates the backing field (`FieldBuilder`), the property (`PropertyBuilder`) and the `get` and `set` accessors as methods (`MethodBuilder`). The body of each method is defined using instructions for the IL virtual stack machine. For example, the instructions used in the `set` accessor method are:

1. `ldarg.0`: Load the value of the `this` hidden argument on the stack.
2. `ldarg.1`: Load the value of the first actual argument on the stack, which contains the value to set to the property.
3. `stfld` ***`field`***: Store the value at the top of the stack to an instance field ***`field`*** of object just below the top of the stack. Then pop both used values from the stack.
4. `ret`: Return from the method. Since the stack is empty at this point, no value is returned.

[1] Modules exist so that a single assembly could be composed of multiple separately compiled parts. This is rarely used in practice and so the vast majority of assemblies will have exactly one module.

```csharp
var assemblyName = "MyAssembly";
var assembly = AssemblyBuilder.DefineDynamicAssembly(
    new AssemblyName(assemblyName), AssemblyBuilderAccess.Save);
var module =
    assembly.DefineDynamicModule(assemblyName, assemblyName + ".dll");

foreach (var entityKind in EntityKinds.ToGenerate)
{
    var type = module.DefineType(entityKind.Name);

    type.AddInterfaceImplementation(
        typeof(IEquatable<>).MakeGenericType(type));

    foreach (var propertyInfo in entityKind.Properties)
    {
        var propertyType = Type.GetType(propertyInfo.Type);

        var field = type.DefineField(propertyInfo.LowercaseName, propertyType,
            FieldAttributes.Private);

        var property = type.DefineProperty(propertyInfo.Name,
            PropertyAttributes.None, propertyType, new Type[0]);

        var getMethod = type.DefineMethod("get_" + propertyInfo.Name,
            MethodAttributes.Public | MethodAttributes.SpecialName,
            propertyType, new Type[0]);

        var il = getMethod.GetILGenerator();

        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Ldfld, field);
        il.Emit(OpCodes.Ret);

        property.SetGetMethod(getMethod);

        var setMethod = type.DefineMethod("set_" + propertyInfo.Name,
            MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(void), new[] { propertyType });

        il = setMethod.GetILGenerator();

        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Ldarg_1);
        il.Emit(OpCodes.Stfld, field);
        il.Emit(OpCodes.Ret);

        property.SetSetMethod(setMethod);
    }

    type.CreateType();
}

assembly.Save(assemblyName + ".dll");
```

Listing 7: System.Reflection.Emit example

### 1.3.2 Mono.Cecil

Mono.Cecil [10] is a library that can be used for generating and transforming assemblies. It was written as part of the Mono project.

The main difference between Cecil and Reflection.Emit is that Cecil can be used to read assemblies, including their IL, and to modify them, while Reflection.Emit can only be used to create brand new assemblies. Another difference is that Cecil has its own type system, independent from the type system of the .NET runtime. This means that it can be used for example to work with assemblies that target a newer version of .NET than the currently executing one, or to work with assemblies that target an incompatible .NET implementation.

Cecil does not have an example of its usage shown here. While its APIs is different from Reflection.Emit, those differences are not relevant to this work.

## 1.4 Other approaches

While with the approaches mentioned in previous sections, then generated code (either C# or IL) fairly closely corresponds to the generating code, other options are possible. This section describes one such library.

### 1.4.1 Expression trees

Expression trees, [11] contained in the System.Linq.Expressions namespace, offer another representation of code.

Expression trees were first introduced in .NET Framework 3.5, to support translating of Language Integrated Query (LINQ) queries to existing query languages, such as Structured Query Language (SQL). This initial version included only expression-like constructs and the C# language supported compiling of expression lambdas to code that creates the corresponding expression tree.

In .NET Framework 4.0, expression trees were expanded with statement-like constructs, such as blocks, assignments or loops, to support the Dynamic Language Runtime (DLR). The result is a “language” that is still expression-based, which means that even constructs such as blocks or loops can have a result. This is somewhat similar to functional languages such as F#, which also do not differentiate between statements and expressions. The C# language was not updated to support the new constructs when translating lambdas to expression trees.

An expression tree can be inspected, often in order to be translated to some query language, or it can be executed. Depending on circumstances, executing expression trees is either done by using an interpreter, or by compiling them to IL using Reflection.Emit and then executing the result.

Expression trees can only represent expressions and statements, not types or their members, which limits their usefulness when generating code. This also means the running example is not applicable to expression trees.

## 1.5 Metaprogramming systems

TODO: More obscure tools, especially those that work with C#

### 1.5.1 PostSharp

PostSharp [12] is a commercial tool for transforming built assemblies at the IL level. It focuses on aspect-oriented programming (AOP), which is the idea that cross-cutting concerns (related pieces of code that are spread over the program, such as logging or code related to thread safety) should be specified separately from the rest of the code.

An aspect is applied to the target program element, such as a method, by attaching a specific attribute to it. The attribute can also be attached to a container, such as a type or an assembly, which applies the aspect to all relevant program elements in that container. This is called “attribute multicasting”.

PostSharp includes many built-in aspects and also allows specifying custom aspects. Custom aspects work by calling a user-defined method at a specific point, such as at the start of a method or before another method is called. The user-defined method can also be provided additional information about the modified method, such as its name or arguments.

This approach makes writing custom aspects easy, but it also means they are limited in what they can do and can have some performance penalty (due to allocation of the object that contains information about the modified method).

Like other tools that work at the IL level, PostSharp is also limited when it comes to changes to the shape of types, because any changes it makes will not be visible at compile time, only at runtime.

### 1.5.2 Fody

Fody [13] is an open-source tool for transforming the IL of assemblies. There are many published “add-ins” for Fody, usually distributed through NuGet, and custom add-ins can be written by modifying the assembly using the Mono.Cecil API (see Section 1.3.2). This makes custom add-ins hard to write, but it means they can perform any transformation. Though the limitations of IL-based tools still apply: changes to the shape of type will not be visible at compile time.

### 1.5.3 F# type providers

The F# language contains a feature called “type providers”, [14] meant for easier access to data sources. Type providers generate types at design-time (i.e. while editing code in an IDE) based on their input parameters and on usage of the generated types. These types can be either regular types that still exist after compilation (type providers using this approach are called “generative”) or their use can be transformed into some other code (“erased type providers”).

Type providers use “code quotations” to express code to execute. Code quotations serve a similar purpose in F# as expression trees do in C#.

## 1.6 Summary

This chapter described various existing approaches for manipulating code in .NET and also several metaprogramming systems.

All of the described metaprogramming systems have severe limitations: The IL-based systems don't work well when changing the shape of a type is desired. F# type providers are a special-purpose system for accessing data sources, and also can't be directly used from C#. Because of that, a general metaprogramming system for C# that would not have these limitations would be a useful addition to the .NET ecosystem.

As for manipulating code, Section 2.1 describes in detail how an ideal approach for such a system would look like. And even if it turns out that the existing approaches are not a good fit, they can still be used for inspiration, or as part of the implementation.

# 2. Analysis

The goal of this work is to create a system for extending the C# language. It should be possible to use it create a wide variety of extensions, including:

- Extension similar to an F# type providers.
- Extension similar to a PostSharp aspects.
- Extension for entity types, which can generate constructors, members required for comparison and any other common boilerplate code.
- Extension that modifies how existing language feature operates, for example, improving the time complexity of recursive iterators from quadratic to linear.
- Extension that can be used to write a single method that performs numeric computation with any numeric type. This is easy to achieve with C++ templates, but impossible with C# generics, because they don't have a way of specifying operators required by the method.
- Extension that optimizes LINQ to Objects queries into efficient imperative code.
- Extension that converts an array of structures (AoS) into a structure of arrays (SoA), to improve performance of memory accesses in some cases.

TODO: More use cases from existing code generators?

Writing these extensions should be fairly easy and using extensions should not cause performance issues at design-time, build-time or run-time.

Such a system will require two major parts: an API for representing and modifying code used by extensions; and a component that drives extensions by applying their transformations at the appropriate time.

## 2.1 Representing code

The basic choices for representing C# code in an API are: as C# code, as IL, as some other form.

IL can be ruled out, because it is hard to use due to its low-level nature, especially when it comes to features like `async-await` (the C# compiler transforms `async` methods into state machines at the IL level).

Using a custom form would effectively require creating a new programming language (though one that does not necessarily have a textual form). The main disadvantage of doing that is that users would have to learn the new language and so it is generally the right choice only when no existing language is suitable.

This leaves the last option: using C#. This approach ensures that extension authors are already familiar with the used language, they only need to learn the API used to represent it. It also means that the output of the system will be in

C#, so existing tools for C# can be used. One disadvantage is that extensions can only use features available from C#. For example, the `calli` IL opcode is out of reach.

Putting this all together: C# is the best choice for forming the basis of the API for representing code for this system.

Now that we know that the API will represent C# code, we need to decide how exactly it should look like:

- The API should be a .NET library.

  It would be possible to use an existing transformation language (such as Extensible Stylesheet Language Transformations (XSLT)) or create a new one. An existing language might not suit well the needs of transforming C# code and it would also require fitting C# code into the language's model (Extensible Markup Language (XML) in the case of XSLT). A new language would be unfamiliar to users and would lack tooling, at least initially. This means it would have to provide significant benefits to make creating it worth it.

  The C# language has sufficient capabilities to express the transformations required to implement C# extensions and it is guaranteed to be familiar to the target group of this system: C# programmers. This means making the API a .NET library usable from C# is the best option.

- The API should be mutable.

  Immutable persistent APIs (such as the one used by Roslyn) are useful when multiple versions of the same object need to be preserved (for example, for the "Undo" button in a code editor) or when multiple threads need access to the same object. Their disadvantage is that they make modifying objects harder: any change to a leaf of an object tree needs to be propagated to the root of the tree, creating new objects along the way.

  This system does not need to keep multiple versions of objects, but it might be useful to parallelize it. For example, when two extensions modify different sections of code, it could be advantageous to execute them in parallel. Nevertheless, because of the focus on ease of use, a mutable API is the better choice.

- The API should not reflect the syntax of C# too closely.

  In contrast with Roslyn, this API does not need to be able to represent, preserve and generate every syntactic nuance of C#, though it has to ensure that semantics of code is not changed. The basic examples of this are whitespace and comments.

  A more advanced example is the difference between declaring variables together (`int i, j;`) or separately (`int i; int j;`). The API could represent both syntactic forms the same: as a sequence of two variable declarations.

  Another example are parentheses in expressions. They are useful in the (infix) textual representation to change or emphasize the order of operations. But when representing an expression as a tree of objects, the order of operations is clear from the structure of the tree, so parentheses are not necessary.

- The API should respect the syntactic structure of C#.

  In contrast with the previous point, the API should not be completely divorced from the syntax of C#. For example, the general structure of a simple method declaration in C# is: modifiers, return type, method name, parameters, method body. If possible, the API should follow the same order.

- The API should make common code simple.

  In the previous point, the list of elements of a method declaration was not complete: method declarations can also have attributes, type parameters and constraints. But many methods will not have these optional elements, and it should not be required to explicitly specify that a method does not have some of these elements.

  Another example are method arguments. Method argument is commonly just an expression, but it can also have a modifier like `ref` or a name. But generating a method call without these optional elements is likely going to be the most common case, so it should be simple and not burdened by the requirements of more complex cases.

- The API should be succinct.

  The structure of real code is often complex, so the API should handle generating such code. Because of this, it should avoid any unnecessary repetition, such as CodeDOM's `Code` prefix, Roslyn's `Expression` suffix or even repeated use of the `new` operator. Roslyn's `SyntaxFactory` with its `static` methods serves as a fairly good model here: when combined with `using static`, it leads to code that does not repeat itself much.

  At the same time, the API should not be too succinct by abbreviating names, for example the way the C function `strpbrk` is named. This leads to names that are hard to understand and that also violate Framework Design Guidelines. [15]

- The API should seamlessly include semantic information.

  Semantic information can be very useful, so it should be easily accessible. There shouldn't be a barrier similar to Roslyn's `SemanticModel`, where syntactic information is included in a syntax tree, but semantic information has to be accessed separately.

  On the other hand, accessing semantic information is more expensive in terms of performance than information based purely on syntax, so it might make sense to somehow discourage their use.

- The API has to be capable of handling invalid code.

  For some extensions, it will be useful if its users can write code that is not valid C#, which will then be transformed by the extension to make it valid. For example, an extension that automatically implements an interface could require that its users specify that interface in the list of interfaces a class implements, but then omit any implementation. That is not valid C# code, but it will be filled out by the extension.

  Another reason is that extensions need to work at design-time, while the code is being edited, so that auto-completion can include members produced by extensions. This is especially important for extensions similar to F# type providers, where generating new members is the reason why they exist.

  Note that not all invalid code has to be handled equally well. Specifically, it is not clear how to parse or represent code that is not syntactically valid C#, such as code that attempts to use an operator that does not exist in C#. Such code still has to be handled by the API, but not necessarily consistently and it could include error nodes, or some similar representation for errors.

  On the other hand, for code that is syntactically valid, but semantically invalid, such as the example of interface with no implementation mentioned above, it is clear how to parse such code and so its representation should be consistent and should not include any errors.

- The API should not be language-independent.

  Several existing APIs for representing C# code are language-independent, at least to some degree. But the goal of this work is only to make the C# language extensible, so doing this is outside its scope.

Since none of the existing APIs satisfy these criteria well, it will be necessary to create a custom API for this system.

An API that follows the principles explained above will work the best for its designed purpose: writing extensions for this system. But it could be also used for others purposes, with some limitations. For example, if such API was used to write a Roslyn analyzer, it should be able to detect semantic issues ("Was a disposable object correctly disposed?") but would likely have problems detecting syntactic or stylistic issues ("Does the code use `int` and not `Int32`?").

## 2.2 The system

The next step is to consider how the overall system of executing extensions and applying their transformations should work.

- The system has to have a design-time component.

  The primary purpose of some of the possible extensions mentioned at the start of this chapter is to generate code that is meant to be directly accessed by the extension's user, often generated in response to other parts of the user's code. This means that the generation has to be performed while the user is editing their code, in other words, at design-time.

- The system has to have a build-time component, which should be separate from the design-time component.

  All of the mentioned possible extensions need to modify the build output, so a build-time component is clearly necessary.

  And since the design-time component is likely going to be tied to a specific IDE or code editor, while the build-time component should work in a variety of situations, like building from the command line or on a build server, the two components should be separate.

- The system should support extensions with different design-time generation requirements.

  Possible extensions have various requirements on code generated at design-time and at build-time, and the system should handle all of them. These include:

  – An extension that generates different code at design-time and at build-time.

    An example of such extension is one that is similar to an erased type provider: At design-time, it generates members with no implementation, which are then accessed by the user. At build-time, the members generated at design-time don't exist and user code that uses them is transformed into some other form.

    This effectively requires writing two different transformations, one for each stage.

  – An extension that generates the same code at design-time and at build-time.

    An example is an extension similar to a generative type provider: Members are generated at design time and the same members are still used at build time.

    There is still a difference between the two stages: it is not necessary to generate implementation of generated members at design-time, which is especially useful since design-time transformations are more time-sensitive. But it shouldn't have be required to write two similar transformations for this.

  – An extension that generates no code at design-time, only at build-time.

    An example is an extension similar to an aspect: The extension is activated by attaching an attribute to a code element. The attribute doe not change, so it does not have to be generated and can come from a regular library. This means that no code has to be generated at design-time. At build-time, the relevant code is then transformed based on what the aspect does.

- The system should regenerate only code that depends on modified code at design-time.

  Performance of the design-time component is important, because it directly affects user experience when editing code that uses extensions, especially when it comes to auto-completion.

  To help with that, we can take advantage of the fact that when the user is editing their code, they usually only change one piece of it at a time. And since parts of an extension's transformation usually only depend on specific pieces of user code, it should be possible to execute only the parts of the transformation that depend on changed pieces of user code.

  Another reason why this should be done is that extensions can affect performance of the whole system in unpredictable ways and limiting how much extension code runs also limits that unpredictability.

  Doing this might require extending the API of the system.

- The system should make experimenting when creating extensions easy.

  For example, if the system included the API suggested in the previous point, using it makes the extension more efficient but also more complicated. Because of that, use of this API should be recommended, but optional. This way, experimenting with writing extensions or creating personal extensions is simple thanks to the simple API, while production extensions can be efficient thanks to the complex API.

- The system should allow distributing extensions through NuGet.

  NuGet is an established distribution channel for regular .NET libraries, and also for other kinds of libraries, like Roslyn analyzers or Fody add-ins. This makes NuGet a good fit for distributing extensions for this system.

  This also effectively implies that extensions should be distributed as .NET libraries. While NuGet can work for other kinds of artifacts, it works best for .NET libraries.

- The system should allow extensions to report errors and warnings about code.

  Many extensions are likely going to have ways of using their API (generated or not) in ways that are suspicious or outright incorrect. The system should let extensions report these issues to the user, including identifying the problematic part of their code.

  As a side-effect, it would be possible to write an extension that does not perform any transformations, it only reports errors or warnings. Such extension would serve the same purpose as a Roslyn analyzer and would have similar limitations than those mentioned at the end of the previous Section.

# 3. Design

Before starting actual design of the system, its name should be decided. This name would be used in names of namespaces, assemblies, NuGet packages and so on, so it should fit well with their requirements and conventions. The name should also be reasonably unique, easy to remember and not too long. The name chosen based on these principles is "CSharpE", meaning "C#, extensible".

As explained in the previous chapter, this system has two main tasks: representing code and transforming code. This means it is natural to split the project into two main parts: `CSharpE.Syntax` and `CSharpE.Transform`, respectively.

## 3.1 CSharpE.Syntax

The `CSharpE.Syntax` namespace contains all the types necessary for representing and modifying C# code starting from the project level and going down all the way to the expression level. There is no representation for solutions, because extensions work at the project level, which means solutions are not necessary.

### 3.1.1 Example

Before explaining details about the design of this namespace, it might be useful to see an example of its usage. Listing 8 shows how the running example from Section 1.1 could be implemented using this API. Notice how the code is very short, when compared with the previously shown examples from Chapter 1, without resorting to a custom language that mixes two programs, like T4 does. This is thanks to following the guidelines from Section 2.1.

Specifically, what the code does is to create a project (`Project`) containing one source file (`SourceFile`) that contains the initial code. Then it finds classes (`ClassDefinition`) in the project and for each of them, adds `IEquatable<T>` to the list of base types and also changes all its fields (`FieldDefinition`) to auto-implemented properties (`PropertyDefinition`).

The rest of this chapter describes the API in detail.

### 3.1.2 General principles

There are some rules that apply though all levels of this API:

- Types in this API are regular mutable classes.
- Types that represent nodes in the C# syntax tree inherit from the common base class `SyntaxNode`. This includes most types from the source file level down.

  A syntax node can have only one parent node, to make sure the syntax tree is actually a tree and so that mutating a node does not affect another, seemingly unrelated part of the syntax tree. But this parent node is not exposed in the API, for reasons explained in Section 3.2.

```
var project = new Project(new SourceFile(
    "Entities.cs", EntityKinds.ToGenerateFromSource));

foreach (var classDefinition in project.GetClasses())
{
    classDefinition.BaseTypes.Add(
        TypeReference(typeof(IEquatable<>), classDefinition));

    foreach (var field in classDefinition.Fields)
    {
        classDefinition.AddAutoProperty(Public, field.Type, field.Name);
    }

    classDefinition.Fields.Clear();
}

File.WriteAllText("Entities.cs", project.SourceFiles.Single().GetText());
```

Listing 8: CSharpE.Syntax example

Syntax nodes can be deep cloned by calling the `Clone` method, which is a generic extension method. It is that way to avoid having to cast its result to the correct type, which would be necessary if `Clone` was a simple instance method on `SyntaxNode`. Syntax nodes are also cloned instead of assigning a new parent node, to maintain the tree shape.

- Collections of nodes are usually exposed as properties of type `IList<T>`. This interface is the most flexible out of the commonly used collection interfaces in .NET. Using an interface means that the user is shielded from the implementation detail of which specific collection type is used.

  Using this interface, which is implemented by commonly used types like arrays or `List<T>`, also means that collection properties can have easily usable setters. For example, to set the body of a method to a specific sequence of statements, one could use code similar to the following: `method.Body.Statements = new Statement[] { ... };`.

  For convenience, some collections are also exposed on higher levels as methods. For example, `TypeDefintion` contains the `Methods` property, while `SourceFile` and `Project` contain the `GetMethods` method, which returns methods from the whole file or project, respectively. Such methods return `IEnumerable<T>`, because it does not make sense to for example add a method directly to a file or a project, it has to be added to a specific type.

- The API includes implicit conversion operators when appropriate. Specifically, they can be used to convert from a definition to a reference (e.g. from `TypeDefinition` to `NamedTypeReference`) or from a node to a simple wrapper for that node (e.g. from `Expression` to `ExpressionStatement`).

  Using implicit conversions makes code shorter, but also harder to understand, because it is an operation that is not visible in the code. For this reason, all implicit conversion operators have an alternative form, usually a constructor of the target type.

```
├── Project
├── NamespaceOrTypeDefinition
└── SyntaxNode
    ├── SourceFile
    ├── NamespaceDefition
    ├── MemberDefinition
    │   ├── BaseTypeDefinition
    │   │   ├── TypeDefinition
    │   │   │   ├── ClassDefinition
    │   │   │   ├── StructDefinition
    │   │   │   └── InterfaceDefinition
    │   │   ├── EnumDefinition
    │   │   └── DelegateDefinition
    │   ├── BaseMethodDefinition
    │   │   ├── MethodDefinition
    │   │   ├── ConstructorDefinition
    │   │   ├── DestructorDefinition
    │   │   └── OperatorDefinition
    │   ├── BasePropertyDefinition
    │   │   ├── PropertyDefinition
    │   │   └── IndexerDefinition
    │   ├── FieldDefinition
    │   └── EventDefinition
    ├── Statement
    └── Expression
```

Listing 9: Inheritance hierarchy of commonly used types in the `CSharpE.Syntax` namespace

- There is a `SyntaxFactory` type, which exists to make creating syntax nodes more succinct, when combined with `using static`. For example, it means that to use the `this` keyword, it is possible to write just `This()`, instead of `new ThisExpression()`.

Listing 9 shows inheritance hierarchy of the `CSharpE.Syntax` namespace.

### 3.1.3 Projects

At the top of this API is the `Project` class, which represents a collection of C# source files and library references. It also contains helper methods for accessing types from all files within a project, for extensions that do not care about which file contains which type.

The `Project` class is also point of interoperation between this API and Roslyn: `Project` can be constructed from a `CSharpCompilation` and it also exposes a `CSharpCompilation` as a property.

### 3.1.4 Source files

Each C# source file in the project is represented as an instance of the `SourceFile` class. A source file has name, text and a list of members, which are namespace and type definitions.

The obvious object-oriented way to model this list would be to have a common base class shared by the namespace definition class and the type definition class. The problem with this is that types can be members of other types, while namespaces cannot. So, we would want type definition to inherit from the member definition class (along with other kinds of type members), and namespace definition to not inherit from that class. But this is not possible, because .NET does not allow multiple inheritance of classes.

There are several options of how to resolve this:

- Make the common base class of namespace definition and type definition into an interface, since multiple inheritance of interfaces is allowed. This interface would not be very useful on its own, since namespace and type definitions are not very similar and generally require different processing.

- Make namespace definition inherit from the member definition class. This would allow adding namespace definitions as members of types, which is not valid C#, so it is undesirable.

- Don't have namespace definitions in the syntax tree at all, instead the namespace of a type definition is surfaced as a property. This is how namespaces are represented in IL and Reflection. The problem with this is that deviates from the syntax of C# too much, which could be confusing to users.

- Use a different class for representing nested type definitions. This way, there is no issue with multiple inheritance, because each base class would be inherited by a different class. The problem here is that users will probably expect that nested types behave the same as regular types, since both have the same syntax in C#.

- Instead of a common base type, use a discriminated union. The biggest issue with this approach is that it is not commonly used in object-oriented design and so it might be unfamiliar to C# programmers and it also would not fit well with the rest of this object-oriented API.

The option chosen from the above was to use discriminated union `struct` called `NamespaceOrTypeDefinition`, because it fits the best with the expected usage pattern of these types, where nested types and regular types are handled the same, while namespaces and types are handled in separate code paths.

Notice that `using` directives are not exposed in the API at all, because they are managed automatically. This makes the API easier to use, at the cost of preventing users from choosing which syntax should be used, which is consistent with the principles outlined in the previous chapter.

### 3.1.5 Types

The types that can be defined in C# are classes, structs, interfaces, enums and delegates. Classes, structs and interfaces are very similar in that all three can be generic, have a list of base types and have various members. Also, especially when it comes to classes and structs, it could make sense fairly often to manipulate them in the same way.

This means that a reasonable design would be to have the types for classes, structs and interfaces inherit from a common base class and this base class, along with the types for enums and delegates, should inherit from another base class. The problem with this design is naming: there is no established name that would apply to classes, structs and interfaces, but not to enums and delegates. For this reason, the names chosen for the two base classes are `TypeDefinition` and `BaseTypeDefinition`.

For a `TypeDefinition`, all its members are contained in the `Members` list, while various kinds of members like fields or methods are also contained in their own lists (for example, `Fields` or `Methods`). These smaller lists are kept synchronized with the main list of members. This means these lists effectively act as filtered views on the main list.

To make adding new members easier, `TypeDefinition` also contains methods like `AddFields`. For example, consider the following code:

```
var field = new FieldDefinition(...);
type.Fields.Add(field);
```

By using the `AddFields` method, that code can be simplified to:

```
var field = type.AddField(...);
```

### 3.1.6 Members

Class, struct and interface definitions can contain various kinds of members, namely fields, methods, properties, events, indexers, operators, constructors, destructors (also known as finalizers) and nested type definitions. Not all kinds of members are valid for all three kinds of types, but since invalid members might be useful to some extensions, the API still allows them.

All kinds of members can have modifiers. While in source, member modifiers can be defined in different order (for example, both `public static` and `static public` are valid modifiers for a method), this order does not make a difference. For this reason, this API represents all modifiers using a single flags enum, `MemberModifiers`.

Because manipulating flags enums using bitwise operators can be cumbersome, the API also includes helper read-write properties for most valid modifiers for each kind of member. The exception to this are access modifiers. Because a member can have only one kind of declared accessibility (which includes all the access modifiers on their own and also the combinations `protected internal` and `private protected`), access modifiers are surfaced as a read-write property named `Accessibility`, along with a read-only property for each kind of declared accessibility.

Even though the list of kinds of members above might make it seem like the mapping between syntactic forms of members from the C# specification and types in this API should be obvious, it is not. The cases that are not obvious are:

- Constants, fields and field-like events can define more than one member with each declaration, for example, `private int x, y;`. With considered designs, this ability makes processing all declarations of these kinds significantly harder, even those that declare just one member. For this reason, each member defined in this way is actually represented as a distinct instance of the appropriate type. For example, the previously shown declaration would be represented as two instances of the `FieldDefinition` type.

- The specification lists constants separately from fields. But since constants are very similar to fields with the `const` modifier, that is how they are represented: both are `FieldDefinition`, the difference is only in modifiers. This also makes changing a field to a constant or vice versa easier.

- The specification lists two syntactic forms of events: field-like events and events with accessors. Since both define the same kind of member, they are both represented as `EventDefinition`.

  This is made easier by splitting a single declaration with multiple definitions into separate instances, as mentioned before, because otherwise the same type would have to represent quite distinct values: a field-like event declaration that has no accessors, but could define more than one event; and a declaration of an event with accessors, which always defines exactly one event.

- The specification lists three syntactic forms of operators: unary operators, binary operators and conversion operators. It would be possible to have a separate type for each form, but a simpler solution is to have just one type, `OperatorDefinition`, which represents all three forms. An `OperatorDefinition` then has a kind, which is `Implicit` or `Explicit` for conversion operators, or the name of one of the overloadable unary or binary operators (for example, `Addition` or `Xor`).

- The specification lists instance constructors and static constructors separately. But since `Static` already exists as a member modifier for other kinds of members, it could be confusing if it did not work for constructors. For this reason, both instance and static constructors are represented as `ConstructorDefinition`.

Another set of decisions to make are about the inheritance hierarchy of member definitions:

- Classes for all kinds of member definitions derive from a common base class, `MemberDefinition`. This includes type definitions, to support nested types, as has been described in previous sections.

- Methods, constructors, destructors and operators all have parameters and a body, so they inherit from the `BaseMethodDefinition` class. This could make processing method-like definitions easier.

- Properties and indexers both have a type and also `get` and `set` accessors, so they inherit from `BasePropertyDefinition`. Events also have a type and accessors, but they are a different set of accessors (`add` and `remove`), so they are not included.

TODO: Bodies

### 3.1.7 Statements

### 3.1.8 Expressions

### 3.1.9 References

## 3.2 CSharpE.Transform

TODO: subsections

Now that the API for representing and modifying code has been designed, it is time to decide how is the system going to transform a project for some extension. Per Section 2.2, extensions are distributed as .NET libraries. This means the system will need some way to recognize what transformations an extension wants to perform and then execute them.

A reasonable way to do this is to have an interface that is implemented by each transformation. The system would then create an instance of the type (or types) that implements this interface and call a method on it whenever the transformation needs to be performed.

As has been explained before, the transformation that need to be performed can be different at design time and at build time, so the interface should account for that. Also, different extensions will have different relationships between the two transformation, so the API should account for that. The solutions to these two requirements are that the interface will include a parameter that specifies which transformation should be performed and there will also be abstract classes that hide this parameter for transformations that do not need it. This way, full flexibility of using an interface is maintained, while recommended patterns are communicated using the abstract classes.

Listing 10 shows how the running example from Section 1.1 could be implemented as a transformation. Notice that it uses one of the aforementioned abstract classes and that the body of the method is the same as the main part of Listing 8.

A much more complicate question is: how to make design-time transformations efficient, by regenerating only code that depends on modified code? As can be seen from the previous example, transformations are often structured around `foreach` loops over syntax nodes: the transformation applies for each class, or for each type with an attribute, or for each method. And this includes `foreach`

```
public class EntityTransformation : SimpleTransformation
{
    protected override void Process(Project project)
    {
        foreach (var classDefinition in project.GetClasses())
        {
            classDefinition.BaseTypes.Add(
                TypeReference(typeof(IEquatable<>), classDefinition));

            foreach (var field in classDefinition.Fields)
            {
                classDefinition.AddAutoProperty(
                    Public, field.Type, field.Name);
            }

            classDefinition.Fields.Clear();
        }
    }
}
```

Listing 10: CSharpE.Transform simple example

loops that are hidden by convenience methods from `CSharpE.Syntax`. For example, consider the following code iterating over all methods in a project:

```
foreach (var method in project.Methods())
{
   ...
}
```

The above code is effectively the same as the following code, which does not use convenience methods:

```
foreach (var file in project.SourceFiles)
{
    foreach (var type in file.GetTypes())
    {
        foreach (var method in type.Methods)
        {
            ...
        }
    }
}
```

This, combined with the fact that a programmer editing their code usually focuses at a single higher-level syntax node (like a method or a class) at a time, makes `foreach` loops the ideal point where to decide whether transformation code should be executed again.

But the regular `foreach` loop does not offer sufficient flexibility for the kind of operations that will be required to implement this "smart" `foreach` loop. Instead, a method with a lambda parameter will be used, similar to the `Parallel.ForEach` method in .NET. [16]

To make this more concrete, Listing 11 shows how the "smart" `foreach` loop could be used in the running example. Notice that the outer `foreach` loop from

```
public class EntityTransformation : SimpleTransformation
{
    protected override void Process(Project project)
    {
        Smart.ForEach(project.GetClasses(), classDefinition =>
        {
            classDefinition.BaseTypes.Add(
                TypeReference(typeof(IEquatable<>), classDefinition));

            foreach (var field in classDefinition.Fields)
            {
                classDefinition.AddAutoProperty(
                    Public, field.Type, field.Name);
            }

            classDefinition.Fields.Clear();
        });
    }
}
```

Listing 11: Example of CSharpE.Transform smart `foreach` loop

Listing 10 was replaced with a call to the `Smart.ForEach` method, with a lambda serving as the loop body. Why the inner loop was not similarly changed will be explained at a later point.

In general, how this works is that each loop iteration has its own inputs and outputs. Inputs are the syntax nodes and other data that the iteration has access to. Outputs are any changes made to those syntax nodes and also any data it returns (while a regular `foreach` loop cannot return anything, a `foreach`-like method can).

When the loop is executed for the first time, its body is invoked as if it was a regular `foreach` loop, but at the same time, its inputs and outputs are recorded. When it is executed again, the new inputs are compared with the recorded one. If the new inputs match the recorded ones, instead of invoking the loop body, the recorded outputs are used. This way, if only a small part of the input changes, there is a good chance that only small part of the transformation will have to be executed again.

As an example, consider a project with two files, each of which contains two classes and a transformation that uses a smart `foreach` loop over the project's classes. Since the loop over classes in a project includes a hidden loop over files in the project, the first time this transformation is ran, an iteration of the first loop will be executed for each of the two files and an iteration of the second loop will be executed for each of the four classes.

If the second class in the second file is modified and the transformation is ran again, the first iteration of the first loop will not be executed, because nothing in the first file changed. The previously recorded outputs will be used for this iteration. The second iteration will be executed, because the second file changed. The iteration for the first class in the second file will not be executed, because the code of this class did not change. The only class for which any code from the transformation will be executed is the one that changed: the second class of the

```
(TransformProject, , transform)
(SourceFile, File1.cs, transform)
(ClassDefinition, Class1A, transform)
(ClassDefinition, Class1B, transform)
(SourceFile, File2.cs, transform)
(ClassDefinition, Class2A, transform)
(ClassDefinition, Class2B, transform)

(TransformProject, , transform)
(SourceFile, File1.cs, cached)
(SourceFile, File2.cs, transform)
(ClassDefinition, Class2A, cached)
(ClassDefinition, Class2B, transform)
```

Listing 12: Log output for running a transformation twice, with a change between the runs

second file.

Listing 12 shows the output of internal logging for this scenario. Each line represents a section of code to be executed. The first entry on each line is the name of the type of the object that is being processed. The second entry is the name of the object (empty in the case of a project). The last entry indicates whether the section of code was actually executed: `transform` means that it was executed, `cached` means that it was not executed and that its recorded outputs were used.

When it comes to designing how exactly the “smart” `foreach` loop should look like, there are several considerations:

- The system assumes that code in a loop iteration body is deterministic and free of unrecognized side-effects.

  For smart loops to work correctly, the output of each loop iteration has to depend only on its inputs and all its outputs have to be recognized by the system. Otherwise, when the recognized inputs stay the same, the iteration is not re-executed, and the end result is different than what it would be without a smart loop.

  This does not mean non-determinism or side-effects are completely forbidden in transformations, but it does mean they are severely limited. Some examples:

  - If an iteration needs access to the current date, it should not read the `DateTime.Today` property itself. Instead, its value can be passed as an input.
  - If an iteration logs what it does to a file, it might be acceptable that the log file is only updated when the inputs change.
  - If an iteration updates some global cache, then that is technically a side-effect. But if that cache is only used to make the code more efficient and no code relies on it for correctness, then this side-effect does not cause any issues.

  - If an iteration uses a randomized algorithm, the fact that the output of the algorithm could change with each execution is generally not a positive feature. In this case, skipping re-execution should be acceptable.
  - Mutating the syntax node that is being processed by the iteration is explicitly allowed. This is because any changes to that node are recognized by the system and are re-applied even if execution of the iteration is skipped.

  Because determinism and unrecognized side-effects can sometimes be desirable, and because .NET does not have a good way of recognizing them, the system does not have any mechanism of preventing them.

  Also note that non-determinism might be undesirable in a transformation, even when ignoring its effects on smart loops. This is because a transformation can decide what code is valid and how the code behaves when executed. So, code that compiles and works correctly now could stop compiling or it could change its behavior in the future, which is generally undesirable.

- Access to syntax nodes has to be controlled.

  When some part of some syntax node is accessed by a loop iteration, and this part of the syntax node is changed between two executions of that iteration, the iteration has to be re-executed, because that change could alter its output. While it would be possible to track which parts of which syntax nodes are accessed by an iteration, the system described in this work does not attempt to do so, on the assumption that it is not necessary to achieve sufficient performance for design-time transformations.

  Instead, whenever any syntax node accessible by a loop iteration changes, that iteration is re-executed. As a consequence of this, syntax node objects do not have parent references, as has been pointed out in Section 3.1.2. If such references did exist, the whole file or even the whole project would be accessible from any syntax node, which means even a small change would cause re-execution of large amounts of code.

  Instead of providing no parent references whatsoever, another option would be to have parent references on all node types, but prohibit accessing "dangerous" parents at runtime (attempting to do so could result in throwing an exception or returning `null`). But because this could be confusing to users and because parent references are not necessary, this option was not chosen.

  Another consequence of these node accessibility rules is that care has to be taken about which other nodes, apart from the one being processed by the loop iteration, are accessible. This is considered along with other kinds of input in the next point.

- All data that is passed in has to be tracked.

  The system has to understand all inputs of a loop iteration. This includes the syntax node that is being processed by the iteration, but also any other data. For that data, the system has to:

  - understand when they change, so that it can correctly decide when to re-execute the loop,
  - ensure loop iterations do not mutate them, which would be considered an unrecognized output,
  - deep clone them, so that further mutations after the smart loop do not affect following executions.

  For these reasons, the system strictly controls which data types are allowed as additional inputs. The allowed types are: primitive types (such as `int` and `bool`), `string`, delegates without closures, syntax nodes, and also tuples, arrays and lists of allowed types. This set of types should be sufficient for most use cases, but if it turns out it is not, it can be easily extended in the future.

  The natural way of passing additional data into a lambda is by directly accessing variables from the outer scope. When such lambda is compiled, any variables from the outer scope that it accesses are stored in a closure object, that then becomes part of the delegate that represents the lambda.

  TODO: explain why I'm not actually using closures: closure could include too many variables, is not explicit and depends on compiler implementation details

- passing data out
- extension method
- understanding MLM (and the combinatorial explosion)

TODO: mention that I considered transformation transformation

TODO: Not just API!

# 4. Implementation

## 4.1 Syntax trees

## 4.2 Transformations

## 4.3 IDE integration

## 4.4 Example extensions

# 5. Comparison with existing tools

## 5.1 Reading and writing code

## 5.2 Transforming code

# 6. Future work

# Conclusion

# Bibliography

[1] Phillip Carter. *Tour of .NET. Microsoft Docs .NET Guide.* 22 May 2017. URL: `https://docs.microsoft.com/en-us/dotnet/standard/tour` (visited on 18 June 2018).

[2] Phillip Carter. *.NET architectural components. Microsoft Docs .NET Guide.* 23 Aug. 2017. URL: `https://docs.microsoft.com/en-us/dotnet/standard/components` (visited on 17 June 2018).

[3] Ecma International. *Standard ECMA-335: Common Language Infrastructure (CLI).* June 2012. URL: `https://www.ecma-international.org/publications/standards/Ecma-335.htm` (visited on 17 June 2018).

[4] *C# 6.0 draft language specification. Microsoft Docs C# Guide.* 22 May 2018. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/` (visited on 17 June 2018).

[5] *The .NET Compiler Platform. GitHub.* URL: `https://github.com/dotnet/roslyn` (visited on 17 June 2018).

[6] *NuGet Gallery.* URL: `https://www.nuget.org/` (visited on 30 July 2018).

[7] Genevieve Warren. *Code Generation and T4 Text Templates. Microsoft Docs Visual Studio documentation.* 4 Nov. 2016. URL: `https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates` (visited on 9 Aug. 2018).

[8] Ron Petrusha. *Dynamic Source Code Generation and Compilation. Microsoft Docs .NET Framework Guide.* 30 Mar. 2017. URL: `https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-source-code-generation-and-compilation` (visited on 9 Aug. 2018).

[9] Ron Petrusha. *Emitting Dynamic Methods and Assemblies. Microsoft Docs .NET Framework Guide.* 30 Aug. 2017. URL: `https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/emitting-dynamic-methods-and-assemblies` (visited on 9 Aug. 2018).

[10] *Mono.Cecil. Mono documentation.* URL: `https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/` (visited on 9 Aug. 2018).

[11] Bill Wagner. *Expression Trees. Microsoft Docs C# Guide.* 20 July 2015. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/` (visited on 10 Aug. 2018).

[12] *PostSharp.* URL: `https://www.postsharp.net/` (visited on 13 Aug. 2018).

[13] *Fody. GitHub.* URL: `https://github.com/Fody/Fody` (visited on 14 Aug. 2018).

[14] Phillip Carter. *Type Providers. Microsoft Docs F# Guide.* 2 Apr. 2018. URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/` (visited on 14 Aug. 2018).

[15] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines. General Naming Conventions. Microsoft Docs .NET Guide*. 22 Oct. 2008. URL: `https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions` (visited on 17 Aug. 2018).

[16] Ron Petrusha. *How to: Write a Simple Parallel.ForEach Loop. Microsoft Docs .NET Guide*. 12 Sept. 2018. URL: `https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-simple-parallel-foreach-loop` (visited on 20 Sept. 2018).

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **AOP** | aspect-oriented programming |
| **AoS** | array of structures |
| **API** | Application programming interface |
| **CLI** | Common Language Infrastructure |
| **CodeDOM** | Code Document Object Model |
| **DLR** | Dynamic Language Runtime |
| **IDE** | integrated development environment |
| **IL** | Intermediate Language |
| **JIT** | just-in-time |
| **LINQ** | Language Integrated Query |
| **SoA** | structure of arrays |
| **SQL** | Structured Query Language |
| **T4** | Text Template Transformation Toolkit |
| **VB.NET** | Visual Basic .NET |
| **XML** | Extensible Markup Language |
| **XSLT** | Extensible Stylesheet Language Transformations |

# A. Attachments

## A.1 First Attachment