FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Stanislav Lukeš

API for C# Code Generation

Department of Applied Mathematics

Supervisor of the bachelor thesis: Mgr. Jan Hubička, Ph.D.
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Title: API for C# Code Generation

Author: Stanislav Lukeš

Department: Department of Applied Mathematics

Supervisor: Mgr. Jan Hubička, Ph.D., Department of Applied Mathematics

Abstract: This work presents a library for implementing robust generators of C# code. Existing code generators often generate invalid code for some inputs. Issues such as name collisions reduce the reliability of code generation. Programmers are then forced to handle these cases manually, which breaks build pipelines and lowers productivity. Our library solves these issues. It automatically avoids name collisions, and keeps the generated code clean and human-readable. We compare our approach to other solutions such as reflection-based metaprogramming, macros, intermediate language rewriting and F# Type Providers.

Keywords: code generation C# ILSpy Coberec

Contents

1. Introduction

Modern software engineering practices often require writing many lines of repetitive code. Such code is not only labour-intensive to write, but also tedious to read. Programmers usually call the repetitive code with little information a **boilerplate code** [1]. While the programming practices are likely going to reduce the maintenance cost as a whole, the boilerplate code often has a high maintenance cost - for example, multiple symbols (i.e. methods, properties or types) [2] might have to be renamed instead of a single one.

Many cures have been invented to the issue with boilerplate code, as it is a common software problem. However, there is no silver bullet and thus a lot of room for further development.

A generic approach is to automatically generate the code that is tedious to write manually. This involves implementing programs that emit the desired code. Often, even a simple script can save a significant amount of time. In more complex scenarios, when the code is generated from complex set of metadata such as OpenAPI specification, the code generators can get quite complex. Build-time code generation usually does not offer very good integration with the remaining code, since the generator does not have the knowledge of the existing symbols. For example, code generation is suitable for API clients, but it would be hard to apply for generating getters and setters for a class.

Building a project is usually not as simple as executing the compiler. A build process may consist of many steps, one of which can be the code generation. In this thesis, we will focus on generators that execute during the build. Our project is a library for code generation of C#, and targets especially the more complex generators. Nevertheless, it is very useful to briefly investigate alternative techniques as well.

1.1 The Coberec Code Generation Library

Code generation is done simple by printing out code fragments. This approach works very well in simple cases, when the output space is fairly limited. For example, a program to pre-generate a list of prime numbers for a hash function could be just a few lines in any reasonable scripting language.

As the space of possible outputs rises, a lot of complications may appear. For example, the code generator will need to sanitize symbol names, since in most languages not all strings can be used as an identifier. Then, the generator has to make sure that there are no name collisions after the sanitization. When referencing existing symbols it has to be extra careful to really reference the expected symbol, and not a different one with the same name. In C#, everything could be referenced by a full name to avoid this problem. In general, being overly explicit prevents issues, but it hampers readability of the generated code.

The goal of our project is to provide an abstraction for source code generators that will help to solve these issues. The user of the API will provide a semantic model of the desired code and the *Coberec* library will translate it to a matching C# code.

Coberec is based on ILSpy [3], an open source C# decompiler. ILSpy provides a reliable backend for emitting C#, and we have built an API on top of it.

As always, the abstraction comes with a cost. We will sacrifice ease of use, since the API will force the user to be overly explicit about which type is used, which method is invoked, etc. We will also sacrifice execution speed, as the library will perform checks and transformations on the model that would be unnecessary if we were emitting the source code directly.

This thesis is organized as follows. In section 2 we discuss existing solutions and practices to reduce boilerplate code. In section 3 we describe the design of our code generation library. Then, we describe how is the library used and how it is implemented.

2. Approaches to Reducing Boilerplate Code

Developers came up with plenty of techniques to avoid writing boilerplate code. In this chapter, we will explore some of them. When programming, it is crucial to keep the alternatives in mind to choose the right one for a given situation – no solution is perfect for every problem. Moreover, we will see many good ideas that we can reuse in our work.

2.1 .NET Reflection

.NET has an API for runtime type introspection [4]. It supports listing the type members and in addition provides means to invoke them. Many .NET libraries rely on custom attributes or naming conventions to automatically discover types, invoke methods or list object properties.

A common use case is automatic dependency injection. An example might be the Scrutor library [5], which automatically registers types into the ASP.NET Core service collection. It scans the specified assembly for classes implementing particular interfaces, based on the type name or a similar pattern. The discovered classes are registered into the service collection, which in turn automatically creates instances of them. The services may have dependencies – they may request other services in its constructor parameters. The dependency injection framework automatically resolves the dependencies; again, by using reflection.

These tools helps to reduce the boilerplate that only initializes the service classes and is straightforward to use. There are, however, several costs involved:

- **Transparency**: We cannot simply look at the code to see why it does something unexpected. And the code being executed is hard to debug because it is too generic and not part of the project.
- **Startup Performance**: Reflection forces runtime to load all metadata for types which impose additional startup costs.
- **Throughput Performance**: Invoking methods via reflection is significantly slower than a standard invocation. For example, using reflection for serializing an object into JSON would be prohibitively expensive, while creating few instances of the services is probably fine.

2.2 Linq.Expressions + Reflection.Emit

In addition to using existing types and methods via reflection, .NET also permits creation of new methods and new implementations of interfaces. This technique is often used to improve performance of code using reflection. Probably all serializers use it to achieve reasonable performance.

We can manually create instruction of of the .NET Intermediate Language, the *IL* [6], but that is quite cumbersome. However, .NET provides an excellent abstraction called Linq Expressions [7]. It is an abstract tree semantically similar to C#, so the API is very accessible to C# developers.

To define new types, implement interfaces and override virtual methods, we can use *Reflection Emit*, a part of reflection allows creation of new symbols. Documentation of AssemblyBuilder class [8] is a good starting point to understand the API. Reflection Emit is sometimes used to automatically declare service decorators for tracing, logging and similar tasks.

Refit, a library that makes writing API clients easier [9], uses Reflection Emit in an interesting way. The user only declares an interface annotated with an URL and Refit automatically implements it with a class that makes the HTTP requests. The following example will run an HTTP GET request to `/users/NAME` when we call `GetUser("NAME")`.

```
public interface IGitHubApi
{
    [Get("/users/{user}")]
    Task<User> GetUser(string user);
}
```

It is still necessary to define the methods and the input and output types – the `User` in this case. Otherwise, the boilerplate is reduced to the bare minimum.

Runtime code generation generally increases throughput while the startup time may rise significantly. Furthermore, it is not very compatible with runtimes which do not use JIT – such as .NET on mobile devices or WebAssembly. This limitation is not a big concern for the .NET community, but is may change in the future, if WebAssembly-based computing gains traction.

Similarly, relying heavily on reflection and runtime code generation afflicts trimming of unused code performed by .NET IL linker [10]. The linker can still be used, but the need to register all types referenced by the reflection makes the usage less streamlined and less efficient.

As we will show in the Design chapter (chapter 3), Linq Expressions from the .NET framework had a very significant influence on the design of our API. Even though we do not generate IL at runtime but C# code before compilation, our expression tree looks very similar.

2.3 F# Type Providers

At the time of writing this thesis, C# does not have any mechanism for making compiler plugins or macros. However, F#, another .NET language, has a support for *type providers*. Type providers allow users to use types parametrized by configuration options – the provider is a plugin that generates the type during compilation. The plugin is implemented in F# and is free to use the options in an arbitrary way. There are some limitations, but almost any type may be produced. Unlike macros in many other languages, this mechanism may generate new API, not only expressions or statements. Type providers are used to create types based on a JSON sample file [11], OpenAPI schema [12], database schema [13] or even result schema of SQL queries [14].

There are two ways to implement a type provider. It may be *erasing*, then the type will not exist in the resulting assembly. The F# compiler will inline all method invocations on the type. The created type may only be used in F# as other compilers do not understand the type provider. Alternatively, the type provider may be *generative* – it creates a real .NET type. In that case, we may use the type from C# or any other .NET language – it is only a matter of referencing the F# project. Unfortunately for the C# developers, not many type providers are generative, so this scenario is not used in practice too much.

F# type providers are a neat technology that inspired similar features in other languages. The type providers are very intuitive for the end users, but we do not see a way to replicate the ease of use in C# without modifying the compiler. On the other hand, implementing a type provider is unfortunately quite cumbersome as the API is not well documented.

2.4 Scala Macros

Scala is a functional programming language with a very capable macro system. Scala macros are similar to the F# type providers – the macros are basically plugins to the Scala compiler. The macros look like functions, but the invocations are replaced by an expression returned by the macro. The macros can also inspect and modify the expression provided in arguments.

For example, ScalaTest, a common test library, contains an `assert` macro [15] that looks like the usual `assert` function. However, it inspects the asserted expression and raises a helpful error message explaining why was the expression false. Clearly, such level of integration with the calling code is not going to be possible with code generation.

Another common macro library is Chimney [16]. Its purpose is to simplify mapping between similar types, similar to the reflection-based AutoMapper [17] library. Both libraries reduce the boilerplate code, while Chimney does not have a runtime cost and mapping errors will arise during compilation, not at runtime.

The Scala macros are written in standard Scala code that runs during compilation. The macro is basically a function that gets its arguments as expression trees and returns another expression tree. To create an expression, we do not even have to use any API – Scala has syntax to produce expressions. String literal prefixed with `q` returns the expression that is in the quotes (e.g. `q"1 + 2"` returns an addition expression of the two constants). As the usual string literals in Scala, the expression literal may be parametrized by other expressions and types. For example, `q"($myExpr * 1.2).asInstanceOf[$expectedType]"` multiplies `myExpr` by 1.2 while performing the type conversion. Note that it is not the same as working with strings – we do not have to worry about parenthesis around `$myExpr`.

The `q` literal makes writing macros very accessible, since the macro developers do not have to learn almost any new API. Also, unlike in many other languages, the macro language is not limited and any Scala code is allowed.

2.5 D mixin

D is a statically typed compiled language with support for static evaluation – the D compiler can interpret almost any code during compilation. By itself, static evaluation may only seem like an optimization. However, in D, it can be used to generate blocks of code and include it anywhere in modules, class declarations, function bodies, etc.

This is doable using the `mixin` keyword [18] – it looks like a function, and it works like `eval` from Javascript. It has same syntax as function call and works similarly to the `eval` function from Javascript. Since D is compiled language, `mixin` only works at compile time. The compiler statically evaluates the argument and replaces `mixin` with the string value. As a simple demonstration of the concept, this is a mixin Hello World:

```
void main()
{
    import std.stdio;
    mixin(hello());
}

string hello()
{
    return "writeln(\"hello world\");";
}
```

Unlike in Scala, we are manipulating strings, not a symbolic model of the code. Note how we are referencing the `writeln` function in `hello`, but it is imported in `main`. In such simple example, it looks like a subtle difference, but the `mixin` may use anything from the local scope, and there may be accidental collisions of identifiers.

Since the code generator is just a function returning a string, there is almost no way it could be more straightforward. While it can not integrate with the calling code so tightly as in Scala, combined with D compile-time reflection, it can still offer much tighter integration than a separate code generation process.

2.6 Dynamic Languages

So far, we have only discussed statically typed languages. However, dynamically typed platforms have a lot to say to the problem of a boilerplate code.

The advantage of most dynamically typed platforms is that there is no problem to create a new type at runtime. In Javascript, the {} expression creates an empty object and assigning to `obj.myField` creates a new field if it does not exist already. It is possible to assign a field using `obj["myField"] = value` notation, and any expression may be used instead of `"myField"`. Unlike in C#, where it is necessary to declare the types in code, we just create any objects at runtime. This means that there is no need to declare an expected type when we want to deserialize an object from JSON.

Using a Proxy object [19], we can even intercept fundamental operations on an object, like a field access. Instead of generating API clients, this powerful mechanism may be used to create an object that translates method calls into API calls. A simple demonstration of such API client (and more) may be found in Alberto Gimeno's article on JS Proxies [20].

A slight disadvantage of such approach may be runtime performance. However, in cases when Javascript is used, the runtime overhead of an API invocation does not matter too much. On the other hand, the code size is critical for the client-side applications, so it a great advantage when the API client is only about 20 lines of code.

> Other languages than Javascript often have similar concepts, but we do not have the space here to go through all of them. For brevity, let us just focus on Javascript in this section.

The obvious disadvantage is the lack of guidance before running the program – there are no types to guide the compiler or an IDE. Moreover, the objects lack runtime validation. For example, if the server sent a string instead of a number in the JSON object, we will be unlikely to notice until the specific value is used. On the contrary, such misbehaviour will lead to a crash during deserialization on a strongly typed platform like .NET.

> Interesting compromise could be to generate a TypeScript type definitions from a schema (such as GraphQL or OpenAPI). At runtime we would use the dynamic API clients, or at least dynamic object created from JSON deserialization. To achieve a relative type safety, the usages would be checked by TypeScript at build.

Note that C# has a `dynamic` keyword, which allows programmers to use C# as a dynamically typed language. It supports almost anything that Javascript Proxy objects can do. Yet, in our experience, it not used very widely, since C# programmers do not want to lose their type safety. However, many libraries, including one of the most popular JSON (de)serializer, Newtonsoft.Json, supports it – we can simply declare `dynamic myObject = JObject.Parse(myJson)` and use the object as we would in JavaScript (see the documentation for more details [21])

Note that other dynamically typed languages provide similar concepts. For brevity, we discuss in detail only Javascript and C#.

2.7 C# 9 Source Generators

C# is going to have an API for integrating source code generators into the compiler. It is going to be a way to write plugins that add symbols to the compilation during the process. Note that at the time of writing this work, C# source generators are still in preview, so we have not used them.

The API is quite straightforward – the plugin simply adds new source files to the C# compilation object. The difference from a separate code generation process is that the source generator has access to the metadata about the other compiled types. It remains to be seen what will turn up in the .NET ecosystem. It seems, however, that it should be possible to generate code that depends on the metadata of other types – possibly replacing serializers, dependency injection frameworks and many more libraries that are currently reflection-based.

Let us show the code example from the Source Generators announcement [22]:

```
[Generator]
public class HelloWorldGenerator : ISourceGenerator
{
    public void Execute(SourceGeneratorContext context)
    {
        // begin creating the source we will
        // inject into the users compilation
        var sourceBuilder = new StringBuilder(@"
using System;
namespace HelloWorldGenerated
{
public static class HelloWorld
{
    public static void SayHello()
    {
        Console.WriteLine(""Hello from generated code!"");
        Console.WriteLine(
            ""The following syntax trees existed"" +
            ""in the compilation that created this program:"");
");

        // using the context, get a list of syntax trees
        // in the users compilation
        var syntaxTrees = context.Compilation.SyntaxTrees;

        // add the path of each tree
        foreach (SyntaxTree tree in syntaxTrees)
        {
            sourceBuilder.AppendLine(
                $@"Console.WriteLine(
                    @"" - {tree.FilePath}"");");
        }

        // finish creating the source to inject
        sourceBuilder.Append(@"
    }
}
}");

        // inject the created source into the users compilation
        context.AddSource("helloWorldGenerator",
            SourceText.From(
                sourceBuilder.ToString(),
                Encoding.UTF8
            )
        );
    }
}
```

The generated code is just a string – there is no attempt to provide an API for building the code. That is probably a good thing, since the API may be done by different projects (like our work) and it simplifies the core C# API that will have to be maintained for a long time. At the time of writing, our project has no support for the Roslyn symbol model, but it seems like we should add support when C# code generators become stable.

> Note that even in this simple example, there is a bug in the code generator. The generated code prints a list of filenames in the current project. The filename is simply included in the string, so it will break if any file contains a quotation mark (`"`), and the compilation will fail.

It will be very interesting to observe how this feature will affect the .NET community. While code generation is currently avoided by many project for its unreliability and clumsiness, we can already see a massive interest in Source Generators on social media.

Unfortunately, it is probably not going to be possible to migrate existing libraries to benefit from code generation. Most reflection-based libraries are configured at runtime by initializing it with an "options object". When the logic is dynamic, or the code is generated at runtime, this is reasonable – creating the options in C# is more approachable than learning a specific configuration DSL. With compile-time code generation, it is not possible to know what will come at the runtime. So, either the generated code will have to be very generic to cover all the possible options, or we will see a revival of separate configuration files.

2.8 IL Rewriting

Rewriting of the intermediate language is a powerful technique used by many .NET projects. The point is to generate the boilerplate after the project is compiled. In principle, the transformation may do anything to the compiled binary. Although the power of IL rewriting is nearly unlimited, the transformation should be reasonable to keep the code understandable. Usually, the transformation only does the following:

- Overrides optional methods like ToString, GetHashCode and Equals. For example, Fody.Equals [23] implements structural equality boilerplate.
- Instruments methods with logging, timing or exception handling. For example, Fody.PropertyChanged [24] implements the INotifyPropertyChanged interface and instruments methods with calls of PropertyChanged; Tracer [25] adds tracing; and Fody.MethodTimer [26] uses Stopwatch class to measure execution time of a method.
- Replaces basic constructs. It may redirect method calls to a different method. For example, Fody.Caseless [27] makes all string comparisons case insensitive. Alternatively, LoggerIsEnabled.Fody [28] adds a condition around logging statements.

- Replaces empty method bodies. For example With.Fody [29] automatically implements a With methods for immutable objects.

All the mentioned examples are from the .NET ecosystem and use a common abstraction Fody [30]. As the Fody authors claim in the project description, it greatly reduces the effort needed to integrate the rewriting step into .NET build.

> Manipulating the IL of an assembly as part of a build requires a significant amount of plumbing code. This plumbing code involves knowledge of both the MSBuild and Visual Studio APIs. Fody attempts to eliminate that plumbing code through an extensible add-in model.

Some other platforms also have projects that do source code or intermediate language rewriting. In the Javascript ecosystem, it is usual to have multiple rewriting steps involved in the build:

- To replace new language features with the ones that even older browsers can run.
- To bundle multiple modules together to make the distribution over the internet more efficient.
- To minify the code to make the bundle smaller in size.

The concerns about older versions of the virtual machine and application size are much stronger on the web. Still, even in the .NET ecosystem, there is a .NET IL Linker project that does tree shaking to reduce the binary size. Such transformations are however a bit out of scope of this work. We are slowly getting into the area of compiler optimizations, which may sometimes also help reducing boilerplate, but it is not the main point.

The limitation of IL rewriting is that introducing new symbols is quite dubious. The problem is that the project compilation runs before the rewriting step. If we are adding symbols after the project is compiled, the project itself must not reference them. Alternatively, there must be a dummy symbol that the C# compiler may use, and then the IL rewriter will redirect it to the new symbol.

The already mentioned With.Fody [29] library has this exact issue. It implements a `With` method to create a modified clone of an immutable object. Given an object:

```
public class MyClass
{
    public MyClass(int a, string b)
    {
        this.A = a;
        this.B = b;
    }
```

```
    public int A { get; }

    public string B { get; }
}
```

The Fody add-in will introduce a method for each property:

```
public MyClass With(int value)
{
    return new MyClass(value, this.B);
}

public MyClass With(string value)
{
    return new MyClass(this.A, value);
}
```

This by itself would work well, but only if the project was a class library that itself never calls the `With` method. Nevertheless, there is a way – we introduce a dummy symbol. We will add a generic `With<T>(T value)` method to the class, which the C# compiler will use in that project. As the author claims in the blog post [29]:

> It is safe to call With methods in the same assembly where the class is defined: the calls get adapted to the real implementation automatically.

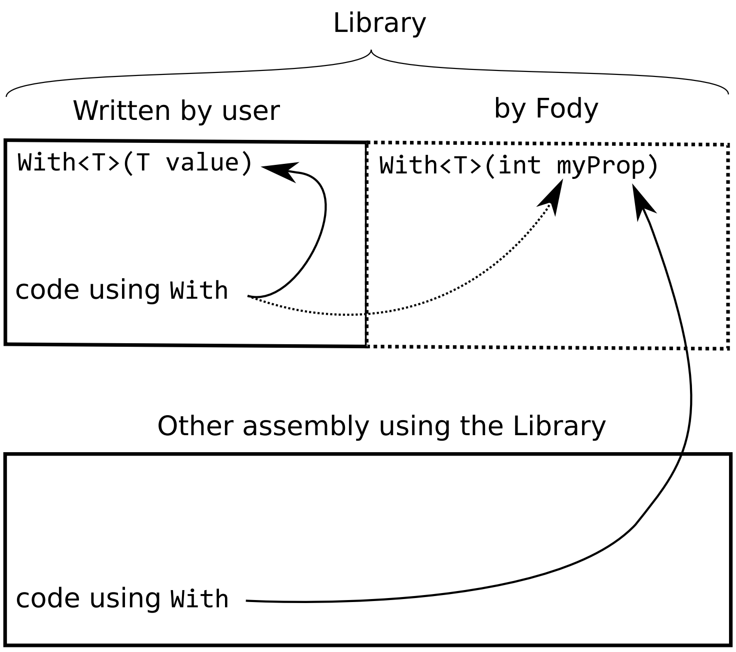

Figure 2.1: Illustration of the symbol references when using Fody.With. Dashed lines are introduced in the rewriting process by Fody.

Another problem with IL Rewriting is that the .NET tooling is not well-prepared for it and some advanced development features might stop working. For example, Edit & Continue feature does not work with Fody [31] nor with PostSharp [32], a proprietary tool doing IL Rewriting.

2.9 Summary

We have discussed many methods used by programers to avoid writing boilerplate code. With this background we can identify the following advantages of the code generation approach used by Coberec:

Transparency – The generated code can be easily read and debugged. Naturally, this point depends on the output quality of the generator and the complexity of the generated program. We consider this one of the main strengths, so we put a lot of focus on the quality of the generated code. The alternatives usually do not come even close to the transparency we can offer with source generation. Even with D `mixin` and presumably C# 9 Source Generators, one can not directly open the generated source file in debugger.

Runtime Performance – Source generation does not need any runtime initialization and we can generate fairly specialized and optimized code for the specific task. We expect both startup and throughput performance similar to hand written code, even when Coberec is used in large projects. Macros and IL rewriting also achieve similar performance.

API Creation – Code generation is nearly the only discussed approach that can introduce new API symbols. While dynamic languages can introduce new symbols at runtime, they do not offer type safety and IDEs can not guide the user through autocompletion. However, as soon as we start generating large APIs, we hit a range of edge cases – the symbols must get valid names, there must not be any collisions, we must reference other symbols correctly, etc.

Portable Principle – In principle, we can generate any text-based programming language. In this work, we focus on C#, but we could have a similar project for any language. By contract, macros and reflection-based tricks are mostly focused on a single platform – there is no way to create a similar API for C# and for, let us say, Python. However, writing a robust code generator is not as easy, so we only claim that the principle is portable. Most of the work done on a code generator is unfortunately not going to be portable. In theory, we could design a common abstraction, but there would be enormous amount of compromises to make.

Compared to the alternatives discussed above, we are aware of the following limitations:

Code Size – Generating a lot of boilerplate code is going to affect the application size, which may sometimes degrade performance. It is most problematic for client-side web applications, where the size of the code is crucial for reasonable load time.

Build Time – Another step in the build process is going to affect the time it takes to compile the project. Not only does it make it slower, it also complicates the process and makes it less reliable.

Reliability – Especially when we generate a large API, the number of edge cases the code generator may hit is vast. This is the issue we want to tackle by the Coberec library – create an abstraction that will make it possible to write a reliable code generator.

3. Design

In this chapter, we will go through the design decisions made during the project development and explore some alternatives.

3.1 `System.Linq.Expressions` Influence

We consider the Linq Expressions API [7] a very reliable API for emitting the intermediate language at runtime. We have designed our API to be similar and thus easy to use for developers who are already familiar with Linq Expressions.

In combination with Reflection Emit, it is possible to create a .NET assembly using this API. However, we find Reflection Emit harder to use and it has several additional problems. In particular, the API depends on reflection, so the referenced symbols of the output assembly must be loaded into the code generator. These references may collide with dependencies of the generator itself and cause reliability issues.

We originally intended to reimplement the Linq Expression API for code generation. However, to avoid the dependency on .NET reflection a new method of representing metadata had to be created (3.7).

3.2 Expression Based Model

Major similarity to Linq Expressions is that every code fragment is an expression; there are no statements. For example, we do not distinguish between `if` statement or `if` expression (the `a ? b : c` ternary operator in C#), there is only a single ConditionalExpression. The Coberec library will expand the expressions into a sensible C# form; the generated code will not be necessarily a single expression.

C# distinguishes between expressions and statements, expressions always return a value (i.e. cannot return `void`) while statements do not. To simplify the concept, we represent every code fragment as a generalized expression. Our expressions do not have to return anything – we allow returning `void` and we will allow variable declarations and inline blocks in the expressions. From that perspective, our code model is similar to F#, Scala or other expression-based languages.

3.3 Blocks and Variables

We represent blocks and variable definitions a bit differently than Linq Expressions do, to more closely follow the single responsibility principle [33] and thus make the types simpler. Linq Expressions use a single node type (BlockExpression [34]) to declare variables and chain multiple expressions. We have separated the two responsibilities into BlockExpression and LetInExpression.

The block expression executes a list of expressions ignoring their results and then returns the `result`. The `result` may be `default(void)` expression, if we do not want to return anything:

```
type BlockExpression {
    expressions: [Expression]
    result: Expression
}
```

The name of the "variable declaration" is inspired by the `let variable = value in target` syntax from F#, since the expression has the same semantics. First, the variable is initialized by the `value` and then `target` expression is evaluated with the new variable in scope.

```
type LetInExpression {
    variable: ParameterExpression
    value: Expression
    target: Expression
}
```

3.4 Control Flow

Same as in Linq Expressions, we have a ConditionalExpression representing an `if` statement or the `a ? b : c` ternary operator. Other control flow statements (loops and jumps) are a bit more different to C# in our expression-based model.

Linq Expressions support jumps with `goto` and labels. The behaviour is a bit unintuitive, however. Jumping into the middle of an expression does not make sense since that could evaluate only a part of the expression, and the other parts would have an undefined result. For example, we would jump to an expression in a method call. The single method argument would get evaluated, but the method can not be invoked, since the other arguments would not be computed. For this reason, Linq Expressions do not support jumps to arbitrary locations, but the limitations are hard to understand.

WebAssembly is another expression-based system which handles control flow in a slightly different way. It has an `if` instruction [35], which is very similar to our ConditionalExpression. WebAssembly does not have a generic `goto` instruction and only supports an infinite loop (`loop` instruction) and break to a label (`br` and its shortcuts). The lack of the `goto` instruction ensures structured control flow, which fits nicely into the concept of expression-based code.

WebAssembly `br` instruction links to the target by its index. That works well for a binary format, but would not be a pleasure to use it in an API while building the expression. Apart from BreakExpression, we also introduce BreakableExpression, which is a possible target of the `break`. It is referenced by a unique ID (inside of the LabelTarget type).

```
type BreakableExpression {
    expression: Expression
    label: LabelTarget
}
```

We also allow the BreakableExpression to return a value. This value either comes from the inner `expression` or must be passed as `value` to the BreakExpression that interrupts the execution:

```
type BreakExpression
    @format(t: "break {value} -> {target}") {
    value: Expression
    target: LabelTarget
}
```

> We do not have any explicit support for early return (i.e. return that is not at the end of a method body). Each method returns a single expression. However, this mechanism corresponds to early returns very closely and may be used to implement it. It is more generic than the C# `break` or `return`, so, in some cases, it may emit code containing `goto` statements.

3.5 References

As of version 7.0, C# allows working with references to variables and fields. Having a field, it is possible:

- to read the value of the field;
- to write a value into the field;
- to create a reference pointing to the field,

Having a reference to a field, it is possible to both write to and read from the reference. This means that it is only necessary to support creation of references to fields and variables instead of having three different expression types for read, write and reference.

We will only have a single FieldAccessExpression that returns a reference to the field. Similarly, instead of a variable assignment, we will have a VariableReferenceExpression that creates a reference to a variable or a parameter.

To read a value from a field, we need to combine the FieldAccessExpression with DereferenceExpression. To write a value into a field, we need to combine it with ReferenceAssignExpression.

> This keeps the core model simpler. However, for ease of use, we provide extension methods like `AssignField` that create the whole subexpression.

It is possible to store the references in variables, pass them as arguments and return them from methods. However it is not allowed to store them in a field – C# does not allow references inside of objects; references can not be stored on the garbage collected heap. That also means that reference variables can not be captured in a closure of a local function nor a lambda function.

3.6 ILSpy as a Backend

We use ILSpy [3] as a code generation backend. ILSpy is a C# decompiler witch can serve as a reliable abstraction of several edge cases of the C# language. C# is a complex language with many edge cases, so emitting code that corresponds to the exact symbols specified by the expression is very tricky.

To illustrate that, consider a few examples. Let us assume that we want to call the `Uri.EscapeDataString` method. In our expression tree, it is represented by a MethodCallExpression pointing to the descriptor of the static method. Normally, the method is invoked by the code `Uri.EscapeDataString(argument)`. However, that is not the case, if there is another symbol called `Uri` in scope – for example a property:

```
public string Uri { get; }
public void M() {
    var x = Uri.EscapeDataString(Uri);
}
```

That code would not compile as C# would try to invoke the method on the property `Uri`. To resolve the issue, we would have to use the full name:

```
public string Uri { get; }
public void M() {
    var x = System.Uri.EscapeDataString(Uri);
}
```

However, using full names everywhere would make all generated code longer. Additionally, it does not actually solve our problem. `System` corresponds to the namespace only until there is another symbol called `System`:

```
public string Uri { get; }
public bool System { get; }
public void M() {
    var x = System.Uri.EscapeDataString(Uri);
}
```

We can be even more explicit and prefix the fully qualified name with `global::` to instruct the C# compiler that it should look for global symbols (i.e. namespaces).

There are many more similar cases – another example might be an implicit conversion. One could think that no code has to be emitted when we want to use an implicit conversion, as the compiler will perform it automatically. However, this is not always the case, specifically due to method overloading. Let us say we have a class with an implicit conversion to `double`:

```
public class MyNumber {
    public static implicit operator double(MyNumber @this) => ...;
    public override string ToString() => ...;
}
```

Then, we use `Console.WriteLine(object)` to print a `MyNumber` instance to stdout (which will call the `ToString` method). Conversion to `object`, the expected argument of `WriteLine` method is implicit, but without an explicit conversion, the `Console.WriteLine(double)` overload will be called instead. In practice, this could lead to reduced precision of the output or wrong output format; most importantly, it is a violation of the promise that Coberec always references the exact symbols user has specified in the tree.

Being explicit in every aspect is a way chosen by many code generators. An example might be the code generator for resource (`.resx`) files (see an example file[1] or DotVVM view compiler (generator implementation[2]). However, the explicitness heavily afflicts readability of the produced code. We think that that transparency and "debuggability" is a crucial advantage of code generation compared to the other approaches, so we would like to get the best of both worlds by having a smart code generator.

Implementing such a smart code generator would be very demanding, so we are not aware of anyone doing that in C# for code generation. However, the ILSpy project shares the same problem, and it already has a very reliable C# emitter. ILSpy is a decompiler for .NET assemblies – a program that converts .NET intermediate language (IL) into C#. IL has exact symbol references and no space for implicitness, similar to our expression tree. The ILSpy authors take correctness and precision very seriously – in the end, we were only able to find two bugs, and one of them was already fixed in a newer version.

In the first step of the decompilation, ILSpy parses IL into an internal abstract tree – the ILAst tree [36]. After that, many transformations run on the tree and then it is translated into a C# syntax tree. Few other transformation run on the C# syntax tree and then it is formatted into a text form.

To avoid too noisy output and too complicated implementation, we will translate our expression into ILSpy's internal ILAst structure and then let ILSpy produce the code.

[1] `https://github.com/dotnet/runtime/blob/3bb5f14/src/libraries/Common/tests/Resources/Strings.Designer.cs`

[2] `https://github.com/riganti/dotvvm/blob/61ee3fd/src/DotVVM.Framework/Compilation/DefaultViewCompilerCodeEmitter.cs#L702`

3.7 Metadata

Since we can not depend on the .NET reflection, we need another way to represent references to symbols [2] – types, methods, properties, etc. We also need to allow users to define new symbols, so our model must account for that.

3.7.1 ILSpy Type System

ILSpy has its own type system built on top of System.Reflection.Metadata library [37]. System.Reflection.Metadata only allows reading .NET assemblies; it is not possible to define new symbols on the fly. However, ILSpy type system is quite extensible, so it is possible to create new symbols.

> Note that System.Reflection.Metadata is very different to the standard reflection in .NET framework. It does not use the internal .NET mechanisms to load types, it has its own assembly reading implementation in C#. Loading types using this library does not have the problem with version collisions, like loading it with reflection has.

We started with using the ILSpy type system directly in our expression trees, but it has several deficiencies. In ILSpy, all the symbols are known in advance – there are not any new symbols emerging after the assemblies are loaded. That is not the case in our project, where the entire point is to declare new symbols and then create code from them. Types in ILSpy system contain references to all other related symbols like the type members and the declaring type. That is convenient when we inspect the types, but complicates the construction of new types. The references between symbols in ILSpy type system are cyclic, so we have to make some of the properties mutable to make the creation possible. At some places, however, ILSpy assumes that symbols do not change to cache information, so we have to be quite careful about what and when we can mutate.

Having mutable types also complicates the design for us. While the expression tree is created, we would like to validate basic constraints (if types fit, if the correct number of arguments was specified, etc.). The validation however stops making sense, when some of the validated properties may change. Having an immutable model of metadata would be very beneficial for this purpose.

Also, by using ILSpy type system, we would lock ourselves to using ILSpy as a backend. At the time publishing of this work, there is no support for any other. However, it could make sense to add support for Reflection Emit and Linq Expression backend. That would provide a common API for generating code compile-time and code at runtime.

3.7.2 Symbol Signatures

We cannot simply copy the ILSpy system or reflection and allow adding new symbols, since we want to have the metadata immutable. Immutability is a promise that whenever a function gets information about a symbol, it will not change.

We need to allow adding new symbols even when there are cyclic references (like recursive methods and recursive types). This means that the information about a type can not contain its members, and the information about a method can not contain the body. We have split the responsibilities – type or method **definition** contains contain all information about its contents and a **signature**, that only contains the most basic information.

To define a type definition, the user will need all members of the type already defined. On the other hand, creating a signature is simple – they will just need to know the full name and basic information like parameters, accessibility, etc.. To reference a symbol from the expression (such as to call a method), only the signature is needed. So, symbols can be referenced before they are defined with all of their contents. Moreover, validating usage should be sound, since all properties of the type, method, field or property signature are immutable. Obviously, is not possible to perform validation based on the contents of the type – for example, there is not a way to check that a referenced field exists on the type.

Type signatures contain information about its full name, accessibility, kind of the type (interface, struct, class, ...), if it is abstract or sealed and the generic type parameters. Type members contain a signature of the declaring type, name of the member and other basic information (arguments, result type, if it is static, virtual or abstract).

> For more details, see the definition of TypeSignature, MethodSignature, FieldSignature and PropertySignature.

3.7.3 Symbol References

.NET has support for generics – types and methods may be parametrized by a type argument. Since we want to support the concept, every time we will be referencing a symbol from the expression, we will provide a list of type arguments for that symbol.

It is important to make a distinction between a symbol signature and a symbol reference. **Signature** is a generic version of the symbol – as it is declared in the code; the type parameters are unassigned. **Reference** is a specialized version of the symbol – the type parameters have their arguments filled in. When declaring symbols, we use the signatures – we only know the definition of the type parameter. On the other hand, in code, we almost always need to know the type including its type argument. It would not make any sense to create an instance of `List<?>` without knowing the type parameter, for example.

When the symbol is not generic, there are no generic arguments to be filled, so the reference becomes basically equivalent to a signature. Because in practice, most types and methods are not generic, and we will have an implicit conversion from signature to reference. The conversion will throw an exception if there would be missing type arguments, which is not considered a good practice in general.

However, it significantly reduces noises of the user code, so we find it helpful regardless.

> Note that a type parameter may end up filled by another type parameter. When we are declaring a generic type or method, we will be using its generic parameters (instances of GenericParameter) in the symbol references. We can then use the generic parameters in the arguments of other types. The difference between a signature and a reference is similar to the difference between `typeof(List<>)` and `typeof(List<TParam>)` in .NET reflection. In the second case (similar to reference), the List is not specialized by another parameter `TParam`, not its own parameter `T`.

3.8 Functions and Delegates

C# has support for functions declared inside of methods – either as lambda functions [38] or local functions [39]. The concept of delegates allows us to work with a function as with any object – i.e. store them in variables, fields, parameters and return them from methods. Both local function and lambda function can capture variables from the scope where they are declared.

Since local function is almost equivalent to a lambda assigned to a local variable, we have decided to simplify the concept and only support lambda function. When the lambda function is immediately assigned to a variable, Coberec will translate it into a C# local function, which arguably looks nicer. However, to declare a function variable in C#, we would need a delegate matching the function signature. Finding a matching delegate is usually not a problem – unless our function has `ref` parameters or very many arguments, we can usually use `Action<...>` or `Func<...>` delegates from the standard library.

The lack of a matching delegates would make it impossible to declare local functions taking parameters by reference. That could be quite limiting. However, since we own and design the type system, we could add a special function type – an inline delegate. This is nothing new under the sun – most functional programming languages have a "function type".

There is one more problem with delegates that we can solve using our function types. Delegates are basically special objects with Invoke method matching the signature of the delegate, so our type signature is not going to contain the actual arguments and return type. Not only we would not be able to validate the arguments of an invocation – we would not be able to determine result type of the invocation expression. For all other expressions, we can do that, and most of the validation logic depends on the `Expression.Type()` method.

Including the delegate arguments and return type in the type signature is not possible – delegates that return themselves (`delegate A A()`) are perfectly valid in C#. Since our types are immutable, cyclic references are forbidden, and it would not be possible to construct such a delegate. More importantly, we could not even load assemblies with such types into our type system.

```
type FunctionType {
    params: [MethodParameter]
    resultType: TypeReference
}
```

Our function type simply contains parameters and the return type. We only allow invocations when the target is an expression of FunctionType. Every FunctionExpression returns the FunctionType and to convert between delegates and function, we are providing FunctionConversionExpression.

> Apparently, it is not possible to return itself from the FunctionType as such type would have to return itself. However, we can still return a delegate from the function. Afterwards, the delegate can be freely converted into a function type, and invoked again.

> Note that FunctionConversionExpression can also convert between FunctionTypes, when the parameters and return types are compatible. For consistency, it can also convert between different delegate types, which is a bit problematic in C# itself, so it will result in a lambda function being emitted.

3.9 Metadata Context

Using the Expression class, we can define the code that we can put into metadata definitions. Then, we need to collect the type definitions, connect them with references libraries and emit the C# code. We will introduce a MetadataContext class that will have the following responsibilities:

1. Load symbols from the referenced assemblies.
2. Allow the user to add new types.
3. Allow the user to explore the registered and referenced symbols.
4. Emit the added types into C# code – either a single string or multiple files.

Since the Coberec API is build on top of ILSpy, MetadataContext is a wrapper of the ILSpy ICompilation object, which has a similar role of holding all the type information together.

In principle, the API workflow looks like:

- Create MetadataContext:

  ```
  context = MetadataContext.Create(
      references: listOfReferencedAssemblies);
  ```

- Explore referenced symbols:

```
type = context.GetNamespaceMembers(namespace).First();
members = context.GetMembers(type);
```

- Add new type definitions:

```
context.AddType(typeDef)
```

- Build the result – generate the C# code:

```
var output = context.EmitToString();
context.EmitToDirectory(outputDirectory);
```

Likely, significant parts of the code working with the MetadataContext are only going to use it only for symbol exploration and not to add new types. So, it could make sense to split the context into a type that only allows exploration and type that also allows adding new types. However, listing types depends on the AddType method, as it also lists the added types. Having a read-only context could create a false sense of immutability, and we would have multiple objects that are interconnected by mutations. We choose against this division of responsibilities; this way, it is simpler and will hopefully be less confusing.

3.10 Symbol Renaming

We have already discussed that naming of the generated symbols is a significant problem for code generators. Most existing code generators handle it somehow heuristically, and it is quite easy to come up with an input that forces it to produce invalid code. It is a topic for a different discussion on how big of a problem this is – it depends strongly on the specific use-case. However, our abstraction mostly solves this problem once for all.

We let the API user register symbols with any names – even those completely invalid in C#. When we get to the point of generating C# code, we have to transform the registered metadata into ILSpy type system. It is in this step that we do the symbol naming.

First, we sanitize each name of invalid characters. Optionally, we transform the casing into the C# recommended PascalCase. Then, we assign names to types, one by one. We can not name a type the same as another type in the namespace or any of its members. From principle, we refuse to name types as .NET special methods like `Finalize`, `GetHashCode`, etc. Even when the type does not contain the method, it would be impossible to add it later.

When naming members, we try to prioritize them based on the damage done by giving them a different name. The absolute priority is a overridden method or property – these must have the same name as they have in the base type. Fortunately, C# does not support multiple class inheritance, so we can not have a name collision. Interface implementations are not as critical; we can rename the public method and add an explicit interface implementation [40]. Then we try to prioritize public members over private or internal ones.

C# supports method overloading, so we try not to rename methods. However, when there is a method collision with a higher priority property or when we have two methods with the same parameters, we will do the rename regardless.

> Note that we do not rename method unless they have exactly the same parameters. This may still create problems when relying on implicit conversions. Hopefully, this problem is quite rare and will not lead to many problems.

Because all references to the renamed symbols are symbolic, and nothing in the expression is referenced by name, we rename all usages automatically.

3.11 External References

The ILSpy code emitter must know everything about the symbols that are being used in the code. If we want to use external libraries, we must explicitly include them in the project. When creating a new MetadataContext, we can use the `references` parameter with a list of paths to referenced libraries.

Sometimes it is useful to generate code into an existing project that already contains some code. It is not possible to add an assembly reference to the same project we are just building – for that case, we need an External Symbol API. That will allow us to declare the types we expect to be in the project and register them in the MetadataContext.

Since we already have the broad API for defining types, the simplest option is to reuse it. We have added an `isExternal` parameter to the MetadataContext.AddType method. When set to true, the added type will not be included in the output, but the emitter will know that it exists.

> The API requires us to specify even the bodies of the declared methods, the same as it does for standard definitions. It does not make much sense, but it is easier this way. Providing an empty method body is very easy for the user – create a DefaultExpression of the result type.

This simple addition breaks the fundamental barrier that we could not call the handwritten code from the generated code. However, unlike in macros or reflection-based approaches, this connection is still far from being easy to use. We can not inspect the handwritten code, and we even have to redeclare it.

It may be quite a significant issue of code generation, and resolving it could be a direction for future extension of the project. One idea would be to parse the existing C# code to extract the declared symbols from it. However, that could be a significant performance cost. With version 9, C# should get the Source Generators feature, as we discussed in the previous chapter (2.7). The Source Generators run in the C# compiler, so we could use the metadata from the compilation.

3.12 ILSpy Fallback

C# has a rich set of language features, and it is not possible to express everything in our simplified expression model. We hope that we will be able to fill in the essential missing bits, but it will probably never be complete. For this reason, we provide a simple fallback API that allows users to build the ILAst directly or do any post-processing to the ILSpy type system entities.

We have introduced a RegisterTypeMod method on the MetadataContext. It registers a function to a specified type signature which will be executed after the specified type is created. The function will get a VirtualType instance and may do any modification – add new members, modify the existing members, remove or hide them, etc. However, there is the risk of breaking the automatic symbol renaming by adding more symbols that were not expected. It is thus recommended to add dummy symbols and then replace them.

Another option is to use a special ILSpyMethodBody expression as a method body. It contains a function that returns an ILFunction – an ILAst node that represents the whole function. This option is lighter compared to the RegisterTypeMod and also less risky in terms of symbol renaming.

4. API Overview

In this chapter, we will briefly show how is the library used. We want to familiarize the reader with the usage before we discuss more implementation details. For more complete guides and examples, please see the documentation[1].

4.1 Hello World

First, we will show a simple example, that creates the "Hello world!" program. Even though the result is elementary, the example shows the boilerplate that is needed to initialize the Coberec library.

```
// First, declare symbol signatures:

// namespace MyApp.HelloWorld {
var ns = NamespaceSignature.Parse("MyApp.HelloWorld");
var @public = Accessibility.APublic;
// public class Program {
var programType = TypeSignature.Class("Program", ns, @public);
// public static int Main() {
var mainMethod = MethodSignature.Static(
    "Main", programType, @public,
    returnType: TypeSignature.Int32);

// get the Console.WriteLine reference
var writeLineRef = MethodReference.FromLambda(
    () => Console.WriteLine(""));

// then, build the actual expression tree
var body = Expression.Block(
    new [] {
        // invoke the WriteLine method
        Expression.StaticMethodCall(
            writeLineRef,
            Expression.Constant("Hello World!")
        )
    },
    // and return 0
    result: Expression.Constant(0)
);

// after all, just add the method with the body into the type
var type = TypeDef.Empty(programType).AddMember(
    MethodDef.Create(mainMethod, body)
```

[1] https://github.com/exyi/coberec/

```
);

// create a default context
var cx = MetadataContext.Create();
cx.AddType(type);
// and produce a string with the output.
var csharp = cx.EmitToString();
```

First, it is necessary to create the signatures of the symbols we plan to declare, and we need to create a MethodReference to the `Console.WriteLine` method. For convenience, we can create the reference from a C# lambda function – otherwise, we could create it from a MethodInfo (from reflection) or find the method in the MetadataContext. The method body is fairly simple, we only call the method and return 0, so we create a block with the two expressions.

To format it into a string, we need a MetadataContext (3.9) – the class that holds information about all symbols in the referenced assemblies and the symbols we have declared. We do not need any references beyond the standard library, so we do not have to specify any parameters; otherwise, we could pass an array of references into the Create method. After all, we call the `EmitToString` method, which finally invokes the entire machinery including ILSpy and produces the code:

```
using System;

namespace MyApp.HelloWorld
{
    public class Program
    {
        public static int Main()
        {
            Console.WriteLine("Hello world!");
            return 0;
        }
    }
}
```

4.2 Expression API

The core of our API is the `Expression` class. It incorporates the core model and some helper methods to construct the model more conveniently. We discussed how we are going to represent more complex code figures in the Design chapter. For completeness and clarity, let us skim over the basic C# constructs. The table below shows how the Expression represents various C# code fragments.

C#	ExprCS Expression
`1`	`Expression.Constant(1)`
`"abc"`	`Expression.Constant("abc")`

C#	ExprCS Expression
`a + b`	`Expression.Binary("+", a, b)`
`a * b`	`Expression.Binary("*", a, b)`
`a >> b`	`Expression.Binary(">>", a, b)`
`a & b`	`Expression.Binary("&", a, b)`
`a && b`	`Expression.And(a, b)`
a || b	`Expression.Or(a, b)`
`!a`	`a.Not()`
`a ? b : c`	`Expression.Conditional(a, b, c)`
`if (a) { ...}`	`Expression.IfThen(a, ...)`
`a ?? b`	`a.NullCoalesce(b)`
`a is null`	`a.IsNull()`
`default(T)`	`Expression.Default(typeT)`
`MyClass.Method(a)`	`Expression.StaticMethodCall( myMethodReference, a)`
`a.Method(b)`	`a.CallMethod(myMethodReference, b)`
`a.MyProperty`	`a.ReadProperty( myPropertyReference)`
`MyClass.MyProperty`	`Expression.ReadStaticProperty( myPropertyReference)`
`a.MyField` (read reference)	`a.AccessField(myFieldReference)`
`a.MyField` (read value)	`a.ReadField(myFieldReference)`
`a.MyField = b`	`a.AssignField( myPropertyReference, b)`
`new Abc(a)`	`Expression.NewObject( construtorReference, a)`
`new [] { a, b, c }`	`ExpressionFactory.MakeArray(a, b, c)`
`new T[x]`	`Expression.NewArray( elementType, x)`
`a[x]` (read reference)	`Expression.ArrayIndex(a, x)`
`{ A1; A2; ... return X }` (block)	`Expression.Block(A, result: X)`
`{ A1; A2; ... return X }` (block)	`Expression.Block(A, result: X)`
`new T?(x)`	`ExpressionFactory. Nullable_Create(x)`
`x.Value` (when `x: Nullable<T>`)	`ExpressionFactory. Nullable_Value(x)`
`"abc" + x`	`ExpressionFactory. String_Concat( Expression.Constant("abc"), x)`
`Type a = value; rest...`	`Expression.LetIn( ParameterExpression.Create( Type, "a"), value, rest)`
`myFunction(a)` (for functions)	`myFunction.Invoke(a)`
`a => a` (create function)	`Expression.Function( aParameter.Read(), aParameter)`

C#	ExprCS Expression
`(Func<int, int>)(a => a)`	`theLambda.FunctionConvert( TypeReference.FromType(typeof( Func<int, int>)))`
`a += b`	`a.Ref(). ReferenceCompoundAssign("+", b)`
`a.Property += b`	`a.PropertyCompoundAssign( property, "+", b)`
`a.field += b`	`a.FieldCompoundAssign(field, "+", b)`

> By far, not all C# constructs are in the table. It is possible to express some using these basic components. Unfortunately, some constructs can not be expressed this API. In such case, users may fall back to using the ILSpy tree directly, as we have shown in the ILSpy Fallback chapter (3.12).

More detailed documentation is available as an attachment or on the web[2].

4.3 Metadata Definitions

The second aspect of C# code is the types, methods, properties and fields. We have a broad API for defining symbols in the rich .NET type system.

To declare a namespace, we call `NamespaceSignature.Parse("MyNamespace")`.

4.3.1 Types

First, we declare a few variables for brevity.

```
var ns = NamespaceSignature.Parse("MyNamespace");
// also public accessibility, for brevity
var @public = Accessibility.APublic;
```

C#	Coberec.ExprCS
`internal class C`	`TypeSignature.Class("C", ns, Accessibility.AInternal)`
`public class C`	`TypeSignature.Class("C", ns, @public)`
`public abstract class C`	`TypeSignature.Class("C", ns, @public, isAbstract: true)`

[2] https://github.com/exyi/coberec/blob/master/docs/csharp-features/cheatsheet.md

C#	Coberec.ExprCS
`public sealed class C`	`TypeSignature.SealedClass("C", ns, @public)`
`public static class C`	`TypeSignature.StaticClass("C", ns, @public)`
`public struct C`	`TypeSignature.Struct("C", ns, @public)`
`public interface C`	`TypeSignature.Interface("C", ns, @public)`

All of these helper methods have an additional argument for a list of generic type parameter. A class `public class GenericClass`<T> may be declared as follows.

```
var paramT = GenericParameter.Create("T");
var genericClass = TypeSignature.Class(
    "GenericClass", ns, @public,
    genericParameters: new [] { paramT })
```

Interface implementations and base types are not specified in the signature, but in the `TypeDef`. The following example shows how to declare a derived class and an interface.

```
var derivedClass =
    TypeDef.Empty(
        TypeSignature.Class("Derived", ns, @public),
        extends: justClass.Specialize()
    );

var classImplementingInterface =
    TypeDef.Empty(TypeSignature.Class("Implements", ns, @public))
            .AddImplements(myInterface.Specialize());
```

To add contents to the TypeDef, we may use the AddMember method.

4.3.2 Methods

For brevity, we will have a variable with declaring type and public accessibility.

```
var @public = Accessibility.APublic;
var declType = TypeSignature.Class(
    "MyClass",
    NamespaceSignature.Parse("MyNamespace"),
    @public,
    isAbstract: true);
```

C#	Coberec.ExprCS
`public void M()`	`MethodSignature.Instance("M", declType, @public, returnType: TypeSignature.Void)`
`public static void M()`	`MethodSignature.Static("M", declType, @public, returnType: TypeSignature.Void)`
`public abstract void M()`	`MethodSignature.Abstract("M", declType, @public, returnType: TypeSignature.Void)`
`public virtual void M()`	`MethodSignature.Virtual("M", declType, @public, returnType: TypeSignature.Void)`
`public override void M()`	`MethodSignature.Override( declType, overridenMethod)`
`public override string ToString()`	`MethodSignature.Override( declaringType, MethodSignature. Object_ToString)`

All method may have parameters and generic type parameters:

```
// public void MethodWithParams(
//    string p1,
//    ref double byReferenceParameter,
//    int withDefaultValue = 0)
var parameters = new [] {
    new MethodParameter(TypeSignature.String, "p1"),
    new MethodParameter(
        TypeReference.ByReferenceType(TypeSignature.Double),
        "byReferenceParameter"),
    new MethodParameter(
        TypeSignature.Int32,
        "withDefaultValue")
        .WithDefault(0)
};
var methodWithParameters =
    MethodSignature.Instance(
        "MethodWithParams", declType, @public,
        returnType: TypeSignature.Void,
        parameters);

// public T GenericMethod<T>()
var paramT = GenericParameter.Create("T");
var genericMethod =
    MethodSignature.Static(
        "GenericMethod", declType, @public,
        returnType: paramT,
```

```
        new [] { paramT });
```

After we declare the method signature, we will need to add a body and create a MethodDef. The body must have a way to reference the method parameters through the ParameterExpression. For this reason, MethodDef contains the list of ParameterExpression, although it is a bit redundant with the list of parameters in the signature. The ParameterExpression assign every parameter an id by which we can reference it from the body expression.

```
type MethodDef {
    signature: MethodSignature
    argumentParams: [ParameterExpression]
    body: Expression?
    implements: [MethodReference]
}
```

> Note that the MethodDef also contains a list of implemented methods. The `implements` field is there to enable explicit interface implementations.

To implement an interface method or override a method from the base type, it would not make sense to specify all the metadata again. We have added a helper method MethodDef.Override that copies the return type, argument and name from the base method.

Since it would be a bit noisy to declare the parameters twice, we have a set of helper methods that make the method declarations less cumbersome. The method `MethodDef.CreateWithArray` gets a signature, creates the ParameterExpression list and calls a lambda to create the body expression. In instance (not static) methods, the first argument is `this` – the object on which we call the method. In the following example, we call another instance method while passing it all parameters as we got them:

```
var definition = MethodDef.CreateWithArray(
    methodWithParameters,
    args => args[0].Read()
        .CallMethod(anotherInstanceMethod, args.Skip(1))
)
```

In case we have a fixed number of parameters, we can use an overload of the Create method. Instead of having the arguments in an array, we get them in separate arguments to the lambda function. For example, we could declare a static method for number addition in the following way:

```
var definition = MethodDef.CreateWithArray(
    methodWithParameters,
    (argA, argB) => Expression.Binary("+", argA, argB)
)
```

4.3.3 Fields

Declaring fields is relatively simple, fields are mostly a pair of the name and a type.

C#	Coberec.ExprCS
`public readonly int F`	`FieldSignature.Instance("F", declType, @public, TypeSignature.Int32)`
`public int F`	`FieldSignature.Instance("F", declType, @public, TypeSignature.Int32, isReadonly: false)`
`public static readonly int F`	`FieldSignature.Static("F", declType, @public, TypeSignature.Int32)`
`public static int F`	`FieldSignature.Static("F", declType, @public, TypeSignature.Int32, isReadonly: false)`

Field definition does not have any other info about the type, except for a documentation comment. It is created simply by calling the FieldDef constructor.

4.3.4 Properties

.NET property is basically a pair of methods – the getter and the setter. Both of these are optional, properties without a setter are quite common while the ones without a getter are rare. Property signatures and definitions are created from the two methods, but we have a helper method prepared, that declares the method and property at the same time.

C#	Coberec.ExprCS
`public int P { get { } }`	`PropertySignature.Instance("P", declType, TypeSignature.Int32, getter: @public, setter: null)`
`public int P { set { } }`	`PropertySignature.Instance("P", declType, TypeSignature.Int32, getter: null, setter: @public)`
`public int P { get { } private set { } }`	`PropertySignature.Instance("P", declType, TypeSignature.Int32, getter: @public, setter: Accessibility.APrivate)`
`public static int P { get { } }`	`PropertySignature.Static("P", declType, TypeSignature.Int32, getter: @public, setter: null)`

C#	Coberec.ExprCS
`public static int P { get { } set { } }`	`PropertySignature.Static("P", declType, TypeSignature.Int32, getter: @public, setter: @public)`
`public abstract int P { get { } }`	`PropertySignature.Abstract("P", declType, TypeSignature.Int32, getter: @public)`
`public override int P { ... }`	`PropertySignature.Override( declType, overriddenPropertySignature)`

Many properties in C# programs are the automatically defined properties [41]. We have a helper that defines it – it defines the backing field and the property and returns them in a tuple. Both have to be added to the TypeDef. For getter-only properties, the field is used to assign a value in the constructor.

C#	Coberec.ExprCS
`public int P { get; }`	`PropertyBuilders. CreateAutoProperty(declType, "P1", TypeSignature.Int32)`
`protected int P { get; }`	`PropertyBuilders. CreateAutoProperty(declType, "P2", TypeSignature.Int32, accessibility: Accessibility. AProtected)`
`public int P { get; set; }`	`PropertyBuilders. CreateAutoProperty(declType, "P3", TypeSignature.Int32, isReadOnly: false)`
`public static int { get; }`	`PropertyBuilders. CreateAutoProperty(declType, "P4", TypeSignature.Int32, isStatic: true)`

4.3.5 Documentation Comments

All member definitions have a `doccomment` field. In C#, documentation comments must be valid XML. We can either set the field while we create the definition, or use the `With(doccomment: ...)` method.

```
var type =
    TypeSignature.Class("MyType", ns, Accessibility.APublic);
var td =
    TypeDef.Empty(type).With(doccomment:
```

```
        new XmlComment("<summary> My type </summary>"));

var fieldSgn = FieldSignature.Instance(
    "Field", type, Accessibility.AInternal, TypeSignature.Int32);
td = td.AddMember(
    new FieldDef(
        fieldSgn,
        new XmlComment("<summary> My field </summary>"))
);
```

More detailed documentation of the API may be found in Github project[3].

[3]https://github.com/exyi/coberec/#complete-exprcs-documentation

5. C# from GraphQL Schema Generator

To demonstrate the functionality of our code generation library, we have reimplemented a C# code generator using the Coberec library. It translates a domain model written in GraphQL Schema language into immutable classes. The model in GraphQL is very concise while the generator produces quite rich classes with equality, ToString implementation and methods for easier modification of the immutable objects. The point of this chapter is not to explain its usage in depth, as there is more detailed documentation[1] on the GraphQL Schema compiler.

Since we use this code generator to create parts of the Expression and metadata API, we will briefly explain how it works. The Expression and metadata are quite broad types, and we want them to be immutable, so we are using this code generator to reduce the amount of boilerplate we had to write. On top of the basic generated API, we still provide helper methods to make it easier to create the objects.

To illustrate how the generated API looks, let us show simple GraphQL Schema examples and how they compile to C#. Fist, a simple type with properties `a` and `b`, the `b` is an array and must contain at least one element:

```
type T {
    a: Int
    b: [String] @validateNotEmpty
}
```

The entire class is in the CodeGeneratorTests.ThesisExample.cs file. The important features of the class are:

- The properties `A` and `B` of types `int` and `ImmutableArray<string>`
- A constructor `T(int a, ImmutableArray<string> b)`.
- A more general constructor `T(int a, IEnumerable<string> b)`.
- Implemented Equals method, GetHashCode method and the `==` and `!=` operators. The types are equal when all properties of their are equal.
- Implemented ToString method.
- `With(...)` method that creates a new instance with modified properties. The arguments are all optional, so `x.With(a: 1)` sets `A`, `x.With(b: ...)` sets `B`, and `x.With(a: 1, b: ...)` sets both.

The type validates the constraints when it is created and throws a ValidationErrorException when it is invalid. The exception contains a ValidationErrors instance with information on which fields were invalid with proper error messages. When there is more than one validation error, all should be present in the

[1] `https://github.com/exyi/coberec/blob/master/docs/graphql-gen.md`

list. This is convenient in the case of the Expression class as it is often validated with many rules.

The types also contain Create method that returns a `ValidationResult<T>`. Instead of throwing the exception, it always returns an object which contains either the created instance or the validation errors. In cases when we are not sure about the validity, the `Create` method might be more convenient. The `ValidationResult<T>` is a monad implementing `Select` and `SelectMany` methods which allow usage of the C# query syntax [42]; similarly as shown on Mark Seemann's blog [41].

In C#, default argument values are quite limited. The default may only be a primitive type, string or `default(T)` (null for classes, "zeros" for structs). This a complication for the `With` method – we would like to have the current property value as the default; the intended signature would be `With(int a = this.A, ImmutableArray<string> b = this.B)`, but this is not possible in C#. As a workaround, we have introduced a `OptParam<T>` type. It recognizes two states – either it has a value of type T, or it has no value; somewhat like the Option or Maybe known from other languages. Moreover, it has an implicit conversion from T, so it is constructed automatically when the parameter is used. This is the simplified implementation:

```
public readonly struct OptParam<T>
{
    public readonly T Value;
    public readonly bool HasValue;

    public OptParam(T v)
    {
        this.Value = v;
        this.HasValue = true;
    }

    public static implicit operator OptParam<T>(T val) =>
        new OptParam<T>(val);
}
```

The `default(OptParam<T>)` has `HasValue = false` and the implicitly constructed instance has `HasValue = true`. The With method will have the following signature with both parameters optional:

```
public ValidationResult<T> With(
    OptParam<int> a = default,
    OptParam<ImmutableArray<string>> b = default
)
```

> It may seem that we could simply use null as the default value for our properties in the With method. However, that would make setting a field to null impossible. Furthermore, it would add an edge case

where With and constructor behave differently, which could lead to unexpected behaviour.

We also have support for GraphQL unions [43] which are less straightforward to represent in C#. We are using inheritance to represent one option of many, but the union types also have helper methods to make creating and processing the instances easier. In the Coberec API, Expression and TypeReference are unions. A simpler example of a generated union may found in the CodeGeneratorTests.SimpleUnionType.cs file.

Apart from the basics like constructors, equality and ToString, we automatically implement some helpers. Each case of the union has a factory method on the type, for example, we can use `Expression.Constant(...)` instead of `new Expression.ConstantCase(...)`. We have a method called Match that helps with exhaustive matching. It takes a lambda function for each case of the union, invokes one of them and returns the result value. The principle is similar to the C# switch expression, but Match enforces that all cases are covered. For non-exhaustive matching, we recommend using the switch expression [44].

The GraphQL Schema compiler existed before the Expression API, which is the reason why the Expression API may be generated from GraphQL. Initially, the project was based purely on the ILSpy decompiler. By now, it is mostly migrated to the Expression API while some parts still use the ILSpy Fallback API.

The code generator is published as `Coberec.CLI` NuGet package[2]. Since the GraphQL generator currently depends on the Expression API, and the Expression API is built from GraphQL Schema, we use the code generator NuGet package. The project is not hard to bootstrap from historical source codes, but the cyclic dependency is easier to break using the published binary package.

We could also do it vice-versa and reference the Expression API library NuGet package to build the code generator. However, the code generator was much more stable than the Expression API when we were developing the project.

[2]`https://www.nuget.org/packages/Coberec.CLI/`

6. Implementation of the API

We have already described how the API was designed and briefly shown how it is used. In this chapter, it remains to describe how we implement the translation to C#.

6.1 ExprCS API

As we have mentioned in the previous chapter, the model is declared in the GraphQL Schema. The schema is in these three files:

- `metadata.gql`: Contains the symbol signatures and references (like MethodSignature and MethodReference).
- `expression.gql`: Contains the declaration of the Expression type and all the expression subtypes.
- `composition.gql`: Contains the symbol definitions; the types that compose metadata with expressions.

The GraphQL files are translated into partial C# classes by the Coberec.CLI. Additional helper methods are declared on these classes manually (these are in the src/Coberec.ExprCS/ModelExtensions directory).

The user will create type definitions and registers them into the MetadataContext. At this point, not much is done; the type is only registered and stays in a "Waiting for commit" stage until the user requests the output code. When the code is requested, we invoke CommitWaitingTypes method. At this point, the types and their content will be converted into ILSpy internal representation. When the translation is performed, most of the tree will be type-checked – checked that the referenced symbols exist and have the expected properties.

6.2 Metadata Translation

The reason why type definitions are translated and checked in bulk in the CommitWaitingTypes method is that the types may be referencing each other. If the types were translated and registered one after another, the backward references would be invalid and cause the registration to fail.

The commit process is performed by the CommitWaitingTypes method and has these steps:

- Types are ordered topologically in the order of inheritance. The base types are part of the core type definition, so all base types and implemented interfaces must be declared beforehand.
- Type names are assigned and the types are declared without their members. We have already described how we rename symbols (3.10).

- The type members are declared: Member declaration follows a similar pattern:
  - First, all the names are assigned.
  - Signatures are declared.
  - After all signatures are declared, the method and property bodies are added. This makes sure that cyclic references between symbols work correctly.

The translation from the metadata classes into ILSpy type system is handled by the MetadataContext class.

6.3 Expression Translation

The Coberec expressions are converted into ILSpy internal abstract tree called ILAst [36]. The CodeTranslator class performs this transformation.

The ILAst (`ILInstruction` nodes in code) is a tree made mainly of IL instructions. It is specific with semantic information (types, methods and other symbols). ILSpy then uses this tree for transformations on the code, before it is converted into a C# syntax tree. The semantics of ILAst are closer to IL than C#, but not in every aspect – for example, it contains a node for `async/await` and for `yield return`, which are purely C# concepts. The tree is not designed to be easy to build – it performs almost no validation on construction, the API generally not very intuitive to use, and most mistakes result in a strange behaviour. Another major complication is that it does not have information about the result type of a given expression – this is because the type information is unclear only from reading the IL.

The advantage of using ILAst for code generation is that the tree does not describe the syntax at all. That means that `using` statements, implicit/explicit conversions, choosing the right syntax to call a specific method overload are handled for us by ILSpy.

The CodeTranslator recursively maps the Coberec expressions into a StatementBlock block structure. StatementBlock is a thin wrapper around the ILSpy ILInstruction node. It contains:

- list of the nodes that are the effect of that expression – list of ILInstruction or Block.
- result value – one ILInstruction, or nothing
- type of the result value, since ILInstruction does not contain that information

Some translations are fairly obvious and simple; for example, the TranslateNot method translates the NotExpression:

```
StatementBlock TranslateNot(NotExpression item)
```

```
{
    // translate the argument recursively
    StatementBlock b = this.TranslateExpression(item.Expr);
    // create ILAst negation node
    ILInstruction expr = Comp.LogicNot(b.Instr());
    return StatementBlock.Concat(
        // first, we perform the effects of the inner expression
        b,
        // we return the negated value
        StatementBlock.Expression(b.Type, expr)
    );
}
```

Other expressions are usually a bit more involved. For example, control flow (conditions, loops and breaks) is represented by basic blocks and jumps in the ILAst. The ILAst has a concept of BlockContainer, which is an ILInstruction that contains a list of basic blocks that form a small control flow graph. The blocks are connected by (conditional) branches and a Leave instruction is used to exit the entire BlockContainer. The CodeTranslator creates a BlockContainer for ConditionalExpression, LoopExpression and BreakableExpression. Since the block container can not return a value while conditional and breakable expressions may, we introduce a helper variable that stores the result. In simpler cases, this variable will get elided by ILSpy in the following ILAst transformations.

The translation of anonymous functions is quite tricky. When the function is immediately assigned into a variable, we translate it into a local function; otherwise, we translate it to a lambda function. Creating lambda or local function is quite similar in the ILAst model, and it is handled by the TranslateFunction and TranslateLocalFunction methods. The functions are translated as separate methods, we only keep the callee context when it comes to local variables. Each lambda function also need a matching .NET delegate assigned. As discussed in the Functions and Delegates section (3.8), functions have a special FunctionType which does not have a direct .NET mapping. If a FunctionConversionExpression immediately converts the function to a delegate, we use the delegate from there. Otherwise we pick any matching delegate like `Func<...>` or `Action<...>`.

> Note that looking for a matching delegate might sometimes fail. Functions that take too many arguments or take them by reference might not have a matching delegate. In such case, the translation will fail, and the user will have to declare the delegate explicitly.

6.4 API Usability

The expression and metadata API is made to be easy to use and debug.

When we raise errors early, it takes less effort to debug them. We have implemented many checks that may raise errors when the expressions or signatures are constructed. Most importantly, we always check that the expression types match. We also check other invariants, such that an abstract type is not instantiated, etc. Since the types are immutable, we can check the invariants in the constructor. When the With method is invoked, we revalidate the new object. Sometimes, it is essential to assign two properties in one With call not to break any invariants. For example, changing a static method to an instance method in a MethodCallExpression must be done together with assigning null to the `target`.

However, there is a limit on what we can do during the initialization; we can only work with the information provided in the symbol signatures. For example, we can not validate that a reference conversion is valid as we do not have enough information. The type reference does not contain any information about base types, so we can not know if the cast is permitted. However, we can still perform some checks on a best effort basis – for example, a sealed type can not be converted into a different sealed type.

Often, the expression will be allowed by the validation in constructors, but the translation will fail later. We did our best to throw a meaningful error message even in these cases. The error message will usually contain the stringified expression that caused the problem.

All expressions and metadata types have a ToString method that produces a string representation. Not only does it help with error messages, debuggers will invoke the ToString and show it as a variable tooltip. Also, for printf-style debugging (or, WriteLine debugging in .NET), the ToString is very useful. We try to keep the representation reasonably concise while being clear about types. That is a compromise – for example, constants are formatted as `{value}:{type}` (`100:int` for integers). Often the details are not necessary, but since the string representation is mostly going to be used for debugging the problematic cases, it seems better to be explicit.

There is still a number of things that can be done to improve error messages and the string representation of the expressions. We can already format the expressions into a structure where we can track which expression produced which part of the string. That could allow us to implement a formatter that will highlight the problematic places based on errors from the translator. Apart from fancy features, we should simply collect cases when the translation fails in ILSpy or on internal asserts and handle them correctly.

> We tried tracking the creation location of each expression using C# caller information attributes [45]. It could provide the user with very valuable information for finding the root issue; however, it is quite problematic. Most of the expressions are created through helper methods, not the constructors generated from the GraphQL Schema. Either we would have to pass the caller information from every helper method, or we would only see information about the helper used. We could try to collect entire stack traces, but that would have a significant performance impact.

6.5 Testing

The main goal of our project is to provide an abstraction that helps with the correctness of code generators. To accomplish that, we certainly should not cause even more problems by bugs in our code. No testing can catch all issues, and there is still work in testing that could be done. However, we have a number of different tests; they at least ensure that we do not break what already works.

We have many tests where we generate a piece of code, and we manually accept that the result corresponds to the tested Expression. For asserting that the code is as expected, we use the CheckTestOutput[1] library. For each test, there is a file with the expected output. If the code generator produces the same output as the contents of the file, the test passes. Otherwise, it prints out a diff, and the test fails. In the likely case that the output is different but still correct, we can simply stage the new output version in git which marks the new version as correct.

We have a custom model of metadata, and we offer a function to convert reflection symbols into our symbols. Since there were several bugs in these helpers, we have implemented tests that check every symbol in the .NET standard library. However, these tests are disabled by default as there were stability issues with different .NET framework versions, and the tests require significant time to run.

The GraphQL Schema transpiler is implemented on top of the Expression API, so this project is a complete end to end test of the library. The generated code is sometimes quite complex, so most bugs we discovered now also have a corresponding simpler test which runs on the Expression API directly. We also use the CheckTestOutput library for the GraphQL project, but we do not have as many tests here. However, the simplicity of GraphQL allows us to generate random schemas using FsCheck [46]. Large random GraphQL types are good for uncovering edge cases in the symbol renaming logic and also problems with reference cycles. To run into the problematic cases faster, we seed FsCheck with the Big List of Naughty Strings [47].

> We do have a lot of small and isolated test cases, but we are still missing tests for more complex method bodies. The GraphQL Schema only generates quite simple methods, although there is a lot of them.

[1] `https://github.com/exyi/CheckTestOutput`

7. Conclusion

We have implemented a library for C# code generators. It offers an expression based API for modeling the resulting code (chapter 4), somewhat similar to Linq Expressions from the .NET framework.

The tool is aiming at generating *COrrect, REadable and BEautiful Code*, with priority on the correctness – thus the name Coberec. The C# code emitter is based on the ILSpy decompiler, which makes sure that it produces readable code that always represents what was specified in the API. C# is a complex language, and it would be tough to accomplish our goals without using ILSpy's backend.

We demonstrated usage of the Coberec code generation library by implementing the GraphQL Schema compiler (chapter 5). This compiler is used in the project itself and thus well tested. We are confident that the library works well and that the API is not painful to use.

The GraphQL Schema compiler creates a significant number of symbols that may collide with each other. It itself also does not contain any logic to prevent the collisions, it depends solely on the logic provided by the abstraction (3.10). We also have quite an extensive test suite for the name collisions, so this part of the project seems to be quite dependable. On the other hand, the GraphQL compiler does not stress the expression translation too much – all generated methods are fairly simple.

7.1 Future Work

The library is certainly not feature-complete, there is still a number of C# features that can not be accessed by code generators using the Expression tree. We have a near-complete list of them in our GitHub project[1]. These features can be however accessed through the ILSpy Fallback API (3.12), so none of these issues should be a critical blocker.

We would like to adopt Coberec to work with the C# 9 Source Generators, as we have mentioned. It will likely only mean that we will implement ILSpy type system for the metadata exposed from the C# compiler.

It could be interesting to add support for translating our Expression tree into Linq Expressions. It would allow both runtime and build-time code generation using a single API. There are real use cases where having a single code generator with both backends would be an advantage. For example, in DotVVM web framework, web pages are translated into a C# form at runtime. It would be good to add support for build-time page compilation while keeping the support for runtime compilation. Both have their advantages – build-time compilation would be more transparent and lead to a lower startup time, while runtime code generation is easier to setup and allows to recompile a web page without restarting the application.

[1] `https://github.com/exyi/coberec/issues`

References

[1] Wikipedia contributors. Boilerplate code — Wikipedia, the free encyclopedia. Available at https://en.wikipedia.org/wiki/Boilerplate_code, 2020.

[2] Wikipedia contributors. Symbol (programming) — Wikipedia, the free encyclopedia. Available at https://en.wikipedia.org/wiki/Symbol_%28programming%29, 2020.

[3] Siegfried Pammer, Daniel Grunwald, David Srbecký, Eusebiu Marcu, Christoph Wille, and other contributors. ILSpy - open-source .NET assembly browser and decompiler. Available at https://github.com/icsharpcode/ILSpy, 2020.

[4] .NET Docs contributors. Reflection in .NET. Available at https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection, 2017.

[5] Kristian Hellang and other contributors. Scrutor - Assembly scanning and decoration extensions for Microsoft.Extensions.DependencyInjection. Available at https://github.com/khellang/Scrutor, 2018.

[6] Wikipedia contributors. Common Intermediate Language — Wikipedia, the free encyclopedia. Available at https://en.wikipedia.org/wiki/Common_Intermediate_Language, 2020.

[7] Microsoft. System.Linq.Expressions.Expression Class. Available at https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.expression?view=netframework-4.7.2, 2020.

[8] Microsoft. System.Reflection.Emit.AssemblyBuilder Class. Available at https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.assemblybuilder?view=netcore-3.1, 2020.

[9] Anaïs Betts, Bennor McCarthy, Sergio Pedri, Claire Novotny, and other contributors. Refit - The automatic type-safe REST library for .net. Available at https://github.com/reactiveui/refit, 2020.

[10] Mono team. ILLink.Tasks - IL Linker. Available at https://github.com/mono/linker/blob/master/docs/illink-tasks.md, 2020.

[11] Gustavo Guerra, Tomas Petricek, and Colin Bull. JSON Type Provider. Available at https://fsharp.github.io/FSharp.Data/library/JsonProvider.html, 2020.

[12] Roman Provazník. OpenAPI Type Provider. Available at https://github.com/fsprojects/OpenAPITypeProvider, 2020.

[13] Ross McKinlay, Colin Bull, and Tuomas Hietanen. SQLProvider. Available at https://fsprojects.github.io/SQLProvider/, 2020.

[14] Dmitry Morozov and Max Grebeniuk. FSharp.Data.Npgsql. Available at https://github.com/demetrixbio/FSharp.Data.Npgsql, 2020.

[15] Bill Venners and Artima. ScalaTest: Using assertions. Available at https://www.scalatest.org/user_guide/using_assertions, 2020.

[16] Piotr Krzemiński and other contributors. Chimney - Scala library for boilerplate-free, type-safe data transformations. Available at https://github.com/scalalandio/chimney, 2020.

[17] Jimmy Bogard and other contributors. Automapper - A convention-based object-object mapper. Available at https://automapper.org/, 2020.

[18] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley, Upper Saddle River, N.J, 1 edition, 2009.

[19] Kyle Simpson. *You Don't Know JS: ES6 and Beyond*. O'Reilly Media, Sebastopol, CA, 2016.

[20] Alberto Gimeno. How to use JavaScript Proxies for Fun and Profit. Available at https://medium.com/dailyjs/how-to-use-javascript-proxies-for-fun-and-profit-365579d4a9f8, Jan 2018.

[21] James Newton-King. Querying JSON with dynamic. Available at https://www.newtonsoft.com/json/help/html/QueryJsonDynamic.htm, 2020.

[22] Phillip Carter. Introducing C# Source Generators. Available at https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/, Apr 2020.

[23] Simon Cropp and other contributors. Fody/Equals - Generate Equals, GetHashCode and operators methods from properties. Available at https://github.com/Fody/Equals, 2020.

[24] Simon Cropp, Rafał Jasica, and other contributors. Fody/PropertyChanged - Injects INotifyPropertyChanged code into properties at compile time. Available at https://github.com/Fody/PropertyChanged, 2020.

[25] Csaba Nemes and other contributors. Tracer - Tracing rewriter using Fody. Available at https://github.com/csnemes/tracer, 2019.

[26] Simon Cropp, Geert van Horrik, and other contributors. Fody/MethodTimer - Injects some very basic method timing code. Available at https://github.com/Fody/MethodTimer, 2020.

[27] Simon Cropp and other contributors. Fody/Caseless - Change string comparisons to be case insensitive. Available at https://github.com/Fody/Caseless, 2020.

[28] Wazowski. LoggerIsEnabled.fody - Add-in for Fody, ads ILogger.IsEnabled check to your logging statements. Available at https://github.com/wazowsk1/LoggerIsEnabled.Fody, 2018.

[29] Mikhail Shilkov. Tweaking immutable objects with C# and Fody. Available at https://mikhail.io/2016/05/tweaking-immutable-objects-with-csharp-and-fody/, 2016.

[30] Simon Cropp and other contributors. Fody - Extensible tool for weaving .net assemblies. Available at https://github.com/Fody/Fody, 2020.

[31] Fody contributors. Fody: Edit and continue. Available at https://github.com/Fody/Fody/tree/c31ea96b5be6ce66b992614dda2af2c0a9bb91d2#edit-and-continue, 2018.

[32] PostSharp Technologies. PostSharp: Known incompatibilities. Available at https://doc.postsharp.net/requirements#incompatibilities, 2018.

[33] Wikipedia contributors. Common Intermediate Language — Wikipedia, the free encyclopedia. Available at https://en.wikipedia.org/wiki/Single-responsibility_principle, 2020.

[34] .NET Docs contributors. System.Linq.Expressions.BlockExpression Class. Available at https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.blockexpression?view=netcore-3.1, 2020.

[35] Andreas Rossberg. Webassembly core specification: Control instructions. Available at https://www.w3.org/TR/wasm-core-1/#control-instructions%E2%91%A8, 2019.

[36] Daniel Grunwald. ILSpy/doc/ILAst.txt. Available at https://github.com/icsharpcode/ILSpy/blob/master/doc/ILAst.txt, 2016.

[37] .NET Docs contributors. System.Reflection.Metadata Namespace. Available at https://docs.microsoft.com/en-us/dotnet/api/system.reflection.metadata?view=netcore-3.1, 2020.

[38] .NET Docs contributors. Lambda expressions (C# Programming Guide). Available at https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions, 2019.

[39] .NET Docs contributors. Local functions (C# Programming Guide). Available at https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions, 2017.

[40] .NET Docs contributors. Explicit Interface Implementation (C# Programming Guide). Available at https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation, 2020.

[41] .NET Docs contributors. Auto-Implemented Properties (C# Programming Guide). Available at https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties, 2020.

[42] .NET Docs contributors. from clause (C# Reference). Available at https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/from-clause, 2015.

[43] The GraphQL Foundation. GraphQL Schemas and Types: Union types. Available at https://graphql.org/learn/schema/#union-types, 2015.

[44] .NET Docs contributors. switch expression (C# Reference). Available at https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/switch-expression, 2020.

[45] .NET Docs contributors. Reserved attributes: Determine caller information. Available at https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/caller-information, 2020.

[46] Kurt Schelfthout, Mark Seemann, and FSCheck contributors. FSCheck: Random Testing for .NET. Available at https://github.com/fscheck/FsCheck, 2020.

[47] Max Woolf and other contributors. Big List of Naughty Strings. Available at https://github.com/minimaxir/big-list-of-naughty-strings/, 2020.

A. Attachments

File name	Description
src/	Source code of the Coberec library
deps/	Cloned dependencies of Coberec – ILSpy, GraphQL parser and the Big List of Naughty Strings
docs/ and README.md	Project documentation in Markdown format
doxygen/html	API reference generated by Doxygen in HTML format

The project may be also found on GitHub: `https://github.com/exyi/coberec`. The thesis is also available one the web: `https://github.com/exyi/coberec/blob/master/docs/thesis/intro.md`

A.1 Build Instructions

In order to compile and run the attached source code, .NET Core SDK is needed. The easiest way to break the dependency cycle is to download the `Coberec.CLI` package:

```
dotnet tool install -g Coberec.CLI
```

To get more up to date source codes, instead of using the attachments, it should be possible to clone our repository.

```
git clone git@github.com:exyi/coberec.git --recursive
```

The `dotnet test` command will run the unit tests. For trying out the Expression API, we can recommend to add a new test case into the `Coberec.ExprCS.Tests` project. To use the API in another project, a reference must be added to the MSBuild file (`.csproj`, `.fsproj`, ...):

```
<ProjectReference
    Include="./src/Coberec.ExprCS/Coberec.ExprCS.csproj" />
```