Fachbereich Mathematik und Informatik

Bachelorarbeit

im Studiengang Informatik

Thema: Übersetzen einer universellen Teilmenge von C#5 nach C++11.

eingereicht von: Christoph Husse (christoph.husse@fu-berlin.de)
eingereicht am: 20. August 2012
Betreuerin: Prof. Dr. Elfriede Fehr

Eidesstattliche Erklärung

Hiermit versichere ich an Eides Statt, dass die vorliegende Bachelorarbeit von niemand anderem als meiner Person selbst verfasst wurde und dabei keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 19. August 2012

Christoph Husse

Contents

1 Introduction

During the past decade, virtual machines have seen their golden age and are used virtually everywhere nowadays. Native languages such as C++ [1] have lived sort of a shadow life for specialized low-level tasks or highly demanding applications. But recently there is a notable shift back to native languages [2]. The late demand for high-performance server architecture in particular raises the question if virtual machine can deliver the required effectiveness in the short term. If one looks at high-end server clusters, cloud computing, etc. the power supply and maintenance costs seem to outweigh the programmer costs by far, which are not even a rounding error [3]. Further some application like medical imaging, 3D application, etc. notoriously require highly optimized code to be even run on normal desktop computers. This basically raises the demand for C++ programmers in particular who are usually more expensive and also may need more time to develop software due to the inherent complexities arising when not using the convenient infrastructure that comes with languages such as Java or C# [4]. Even if the C++ program was only 20% faster, the savings in terms of power supply and maintenance would far outweigh the additional programmer costs which could easily be doubled or tripled.

The original question I wanted to solve is*: How do native compilers compare to virtual machines, performance wise?*

There are plenty of examples where either of it is ahead in this or that benchmark, but in the end those benchmarks are pretty much irrelevant. Companies have invested a lot of money in their existing infrastructure and even if C++ would beat C# or Java anytime, there is no way one can expect them to rewrite their entire code-base. Also I don't think that this is the way to go. Managed languages have many advantages over C++ if one doesn't need all the control, most of all simplicity and convenience. Use the right tool for the job! Instead the question is, can the performance of a language like C#, which was meant to be run in a VM from the beginning, be improved by native compilation, and if so, how much faster can we get and where are the exact advantages of a virtual machine. Is C# slower and if so for what applications is it slower. Is it slower because of a non-optimized VM or because it just inherently lacks the expressiveness to let a compiler generate faster code. Unfortunately, all this can only be the motivation behind this thesis, since the process of developing such a C#-to-native compiler is already far exceeding the time-frame of a bachelor thesis.

In this thesis I have set the foundation for a C# to C++ compiler that can compile the entire fundamental Mono class library, referred to as `mscorlib`, containing over 1000 classes and 300k lines of C# code. With a few more months of dedication, this compiler could reach a standard conforming level and most likely run a lot of existing C# applications. Right now, only simple applications are executable, even though much more complicated applications do at least compile properly.

1.1 Why C# and C++?

The reason to use C# as source language is not only because it is my favorite language, besides C++, but also because in contrast to other candidates, like Java or Python, maps exceptionally well to C++11. It would be considerably harder to create such a translator for Java and even more so for dynamic languages under the constraint that it should yield a more efficient translation than using a virtual machine in the first place.

Choosing C++11 as target language is based on the fact that C++ is generally accepted as the language of choice when it comes to performance critical applications. Further it is supported on virtually any platform, is continuously developed and improved, and has the necessary expressiveness to tailor a translation framework that can capture most of C# without compromising performance, thanks to features such as template- and preprocessor-meta-programming. Also, part of the goal, not only for easier development, is to generate human readable C++ code that can possibly be integrated into existing C++ projects and vice-versa. That would not be possible when generating assembler code directly, for instance. Additionally, some C++ compilers support some interesting features such as C++AMP, which can run C++ code on massively parallel architectures, and could be utilized for C# as well without reinventing the wheel, by simply providing similar capabilities through C# attributes and then translating them properly onto such a specialized C++ feature set.

Another supporting factor is Microsoft, which is currently said to develop a C# and C++ compiler with a shared backend, an effort originating from the (failed/abandoned) Phoenix-Compiler-Framework [5]. Acknowledging the fact that C++ is much more expressive than C#, except for garbage collection, mapping C# onto C++ is very similar to having a shared backend. Future version could integrate MCS as frontend into Clang, for instance, which could then skip the C++ intermediate language generation and directly map the C# AST to a C++ AST and achieve the very same shared backend idea, just OpenSource.

Facebook is using its popular HipHop engine to greatly improve the speed of PHP applications. Early attempts were made by translating to C++ which yielded superior speed. Nowadays they are using an optimized VM, which is producing even faster code, but given the dynamic and chaotic nature of PHP applications, this does not make a good case that a todays VM yields faster code for all programming languages.

1.2 Approach

The general approach will be to extend the existing Mono C# compiler (referred to as MCS) with a new backend. The intermediate abstract syntax tree will be used to generate C++ code through a visitor pattern. Mono C# provides an enriched AST, which also has full type-information, is always correct (incorrect programs will be rejected earlier), has useful built-in features such as variable hoisting for lambda expressions, as well as state-machine generation for the C#5 `async` keyword and `yield return` (C# version of Python iterators). Further, it is a proven and widely used compiler, supporting all C# features.

One thing in particular is not part of this thesis and that is garbage collection. For any benchmarks, garbage collection can be disabled in Mono, and does not exist at all on the C++ side. In the past, Microsoft has proven that adding an efficient garbage collector to a subset of C++ is possible (see C++.NET), so I consider that a solved problem falling into the not-invented-here category.

1.3 Extending the Mono C-Sharp Compiler

The MCS compiler does not natively support the custom generation that is attempted in this project. But there are several entry points and some minor skeletons that look like they were meant to support such kind of plugins but never have been completed. So far that is *visit.cs*, which basically represents a half completed AST visitor pattern. I have extended it with all necessary AST nodes, which of course also requires *accept* methods to be added in all respective nodes. Further a lot of data types and properties needed to be made public. The main entry point for AST post-processing is within the compiler driver, located in the *driver.cs* file. The interesting section inside of the *Compile* method is:

```
…
tr.Start (TimeReporter.TimerType.EmitTotal);
assembly.Emit ();
tr.Stop (TimeReporter.TimerType.EmitTotal);
…
```

We have two options for parsing the AST. The first one is before *emit*, and the second is after *emit*. This project hooks in after *emit*, since this will convert lambda expressions, linq expression and a lot of other C# 3.0+ specific constructs back to CS# 2.0 specification. If someone touches the AST before *emit*, then those expressions will remain in their source code form which can be good if one wants them like that. But since C++ has no equivalent constructs, we need the post-processed form in our case.

The hardest part about extending MCS was that there is practically no documentation whatsoever. It was all try and error and debugging, while inspecting the errors and types at runtime and trying to figure out how to map it to C++ properly.

1.4 Overview

Before going into details about the compilation process, let's look at what happens to a C# program when it is run through the compiler.

The compiler gets a directory in which to look for C# files. All matching files are then compiled into a C# assembly, using the regular MCS compiler. Further, during this step all types are analyzed and imported to an internal, enriched format. This information is gathered directly from the generated AST we get from MCS. In the next phase, a C++ header file is generated, containing all declarations for all types, fields and methods. Generic classes go into a separate header file and also contain all definitions, instead of just declarations. The following phase will emit various C++ files for non-template classes. Each file has a size limit, since GCC tools seem to choke themselves on too large C++ object files. Emitting the definitions is mostly just walking through all classes and all methods and properly emitting the method bodies, which themselves are just an AST with blocks, statements and expressions that are traversed in-order. The last phase emits a CMake[1] build system that can compile all generated C++ files into one executable. Now it is possible to build the whole thing using a simple *make* command.

[1] http://www.cmake.org/cmake/help/documentation.html

1.5 The Example Program

In the following I will describe in more detail how to compile a test program from scratch, including how to setup the compilation environment and how to integrate the generated code into a debugger/IDE.

The basis for this chapter is a fresh copy of Debian 7.1 64-Bit. Linux distributions can have an infinite variety of configurations, of which older ones are incompatible or hard to fix, so I am only going to assume this specific system. The exact versions for each program can be retrieved from *www.debian.org*.

The unpacked compiler archive will contain a directory named *build-environment*. This is where C++ compilation happens and where all support code and CMake scripts are stored. The actual C# program to compile is stored in *UnitTest/tests/arrays*. Currently, all paths are hardwired into the compiler (as relative paths), which can easily be changed but doesn't make much sense at the moment. The before mentioned *arrays* directory already contains a single file with our example application, which is used throughout the thesis to illustrate introduced concepts. It is supported to have multiple C# files in there, but only one of them shall have a *Main* method.

The compiler is located in one of the *UnitTest/bin* subdirectories and needs to be invoked from there (due to relative paths). The existing executable shipping with this project should work fine and needs to be run via Mono (this is not necessary on Windows, where double-clicking is sufficient):

```
> mono ./UnitTest.exe -dev-mode
```

Once running, it will automatically translate all code located in the *arrays* directory to C++ and store the result, including scripts, inside of the *build-environment* directory. If the process completes successfully, one can generate a variety of build systems using CMake with the *build-environment* folder as source directory, for instance *Unix-Makefiles*. The compiler supports one command line argument which is *-dev-mode*. This will make the compiler generate a special CMake script that will use the hand-written runtime located in *build-environment/cli/dev*. Normally it is recommended to use this argument, since it is currently not possible to execute programs with the native *mscorlib*. This small library provides hand-written implementations for the required linker references normally provided by a compiled *mscorlib*. This does not only speedup the linker stage by orders of magnitudes, but does also allow debugging and execution of self-contained applications. Self-contained meaning that only basic functionality is used, like arrays, console IO, etc. Using any functionality that is not yet implemented will either raise a *NotImplementedException* at runtime or produce a linker error. In both cases it is possible to just provide a custom implementation for this functionality in the *cli/dev* branch.

When using CMake to generate a build system one should enable *Advanced* and pass some flags to the compiler using the *CMAKE_CXX_FLAGS* variable (I recommend using *cmake-gui* instead of *cmake*). Since it is compiler dependent, I didn't add any to the CMake scripts themselves. Instead I will briefly describe the GCC flags which need to be added:

-fpermissive
: Some of the generated code doesn't seem to be standard conforming in some sense, even though Clang is usually very close to the standard. Without this flag, GCC will not compile the code!

-std=c++11

Enables C++11 support. This flag alone is not sufficient if the system used to compile doesn't have a very recent iteration of the C++ standard library installed (in which case compilation won't succeed).

-fmax-errors=10
: When developing the *cli* code branch or adding new code generation features it is recommended to limit error count, since otherwise the compiler may spend minutes reporting errors which is not only time consuming, but also throws the really important initial errors out of the console buffer.

-w
: Disable all warnings. The code is not made for GCC and will produce a lot of warnings. Disabling all of them is the easiest way to get a clean compilation. Compilation passes Clang with most warnings enabled and there shouldn't be important issues left in the code.

When using Clang, one may instead copy&paste the following flags to get compilation done with all important warnings enabled:

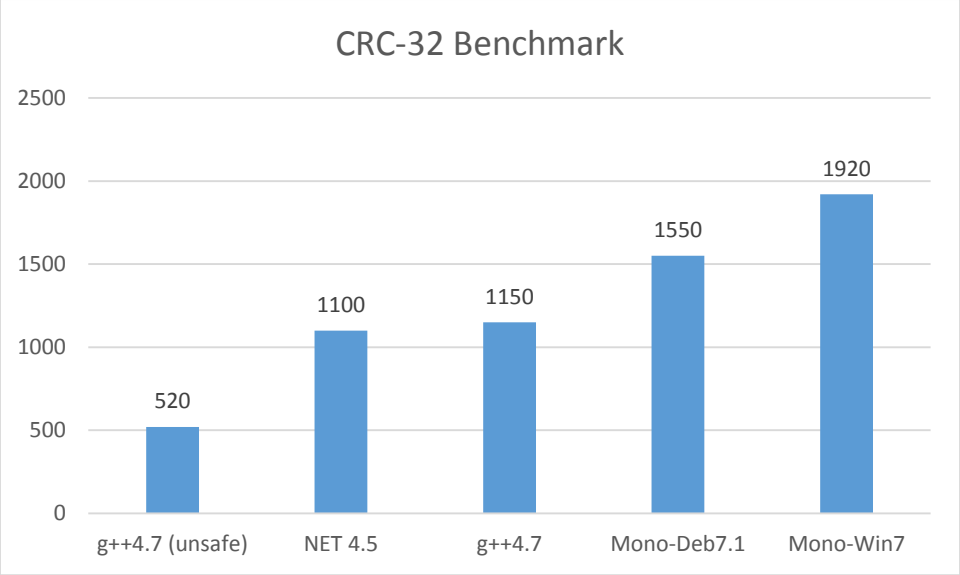
```
-std=c++11 -Werror -Wno-missing-noreturn -Wno-sign-conversion -Wno-conversion -Wno-unused-label -Wno-float-equal -Wno-shadow -Wno-undef -Wno-exit-time-destructors -Wmissing-prototypes -Wmissing-declarations -Wno-parentheses -Wno-switch -Wno-c++98-compat -Wno-unused-variable -Wno-unsequenced -Wno-cast-align -Wno-parentheses-equality -Wno-c++98-compat-pedantic -Wno-padded -Wno-unused-parameter
```

After generating the build system and compiling the C++ code the executable can be found in *build-environment/lib/*.

It is possible to debug the compiled executable using Eclipse, for instance. If one can get CMake to generate debugable Eclipse projects, this would be easy, but I couldn't. Instead I had to setup a fresh C++ project in Eclipse and manually link the source files in the CMake script to it, add the above compiler flags and include directories from the CMake script. Doing this will only take a few minutes and really pays off.

1.6 A Small Benchmark

The previously discussed example application comes with a CRC32 benchmark. By no means shall this benchmark be used to judge about performance for larger common applications, like webservers or games. It basically just measures the integer number crunching performance to some degree. We would expect C++ to stay ahead of C# in especially such applications. The benchmark doesn't give much insight into the performance advantage one might gain through a C++ compilation in comparison to a VM based compilation, but at least it gives a good impression that a C++ compilation can yield faster code, as the following chart illustrates:



Time was measured in milliseconds using the highest precision clocks available on each platform, all code is 64-bit with all available optimizations turned on. NET 4.5 is 111% slower than GCC *without* array boundary checks. GCC with array boundary checks is 4.5% slower than NET 4.5, Mono 2.10 on Debian 7.1 (with ahead of time compilation and all optimizations turned on) is 34% slower than GCC on the same system. Surprisingly Mono 2.10 on Windows 7 is again 23% slower than its brother on Linux, using the same machine, which seems odd. Those values were measured with repeated invocations and rounded to the nearest 10 milliseconds.

Unfortunately, it is currently not possible to run a compiled program on Windows, which leaves a question mark on this whole benchmark. But even without that, we can conclude that C++ compilation certainly gives a major speedup for the Mono implementation, while NET 4.5 seems to be quite optimized and even with other compilers, like Intel C++ Composer 14 (when it is released), I doubt we will see a significant performance improvement (if any) over NET, at least for this simple test-case.

Of course the GCC version without array boundary checks has a major performance advantage and violates the specification, which is why this data point needs to be taken with a grain of salt and cannot be used as reference. It just gives an idea about what is possible by adding a simple attribute based optimization to C# code, like disabling checks for certain parts of the code or even on a per-array basis. Also it shows that required computational resources can potentially be cut in a half by simply removing boundary checks from hotspots.

There was no tweaking involved and only the actual computation (without memory allocation) was instrumented. The optimized C# code for CRC32 was simply copied from *http://dev.khsu.ru/el/crc32/* and compiled to C++. The complete code can be found in the example application and is not very revealing. Performance was measured with high-performance counters in C# and C++ for one gigabyte of continuous data, but due to missing library support, the instrumentation code was added by hand after compiling to C++.

GCC was invoked with the following optimization flags:

```
-O3 -mtune=corei7-avx -march=corei7-avx
```

2 Translation

The translation process was primarily developed using an iterative approach, starting from very basic code samples, fixing all errors until it compiles and going on until a full compilation of the C# core library was possible. While there might be some shortcuts in retrospect, the process is just very convoluted, also due to the hacky nature of MCS and the complexity of both languages. I won't go much into details about all the quirks and hacks that needed to be invented almost everywhere, instead I will focus on the abstract approach that should be sufficient to understand the code later on.

2.1 Object Model

The corner stone of C# are classes or objects. The first question to solve is how to map those to C++. There are a few issues that might pop into mind immediately, like garbage collection, generics, interfaces, method replacement with the new keyword and maybe object construction. In the following we will take an in-depth look at various mapping issues and how they are resolved or ignored.

In general, the primary issue we need to deal with is polymorphism. How do we bring interfaces, methods slots, different assemblies, abstract methods, explicit/implicit interface implementations and all the like together to work as one unit in C++, while still utilizing the existing C++ polymorphism as good as possible. We sure want to use the existing mechanisms, not only for clarity but also for performance. Any emulated polymorphism will not only be not human readable, but also miss out on most compiler specific optimizations regarding polymorphism and maybe even more due to the hacky nature of every hand-written emulation, which makes optimizing such code inherently difficult.

2.1.1 Polymorphic Name Extension

The polymorphic name extension, used in following sections, is basically a means to simulate the C# method slots. When talking about polymorphic methods in all of this thesis, I am referring only to methods that are virtual in the sense that they get a *vtable*[2] slot. For usual polymorphism, C# and C++ work the same, except that each explicit interface implementation, each replacing method, interface-reimplementation, etc. gets a new *vtable* slot. This does not map at all to C++, as there we have one slot for each method in the same name. What we need is a way to alter the C# name such that each slot is mapped to a unique name. Further, there is the issue of multiple assemblies. Right now, all dependencies are compiled into one executable. But that may not be a good idea should the project evolve into a practical compiler. So we also have to think about how to make sure that successive, independent compilations always chose the same alternate method name for a particular slot, otherwise we will be spammed with compilation and linker errors later.

To accomplish this, the signature is a hash code over a string composed of several distinctive properties that each method possesses. Compiler generated methods will have a *SpecialName* attribute, which is included in the signature seed, since the user is allowed to use the same method names as compiler generated methods already have, without conflicts. It contains all parameter types including the return

[2] *http://en.wikipedia.org/wiki/Virtual_method_table*

type, since methods with different parameters can have the same name. It also includes the full method path to honor the fact that each explicit interface implementation gets its own method slot. Also it includes the full class path to make signatures unique across different classes, since they can inherit each other without merging their method slots. Currently, the assembly is not included in the signature making it possible for exact same methods in exact same classes in different assemblies to have the same signature. Something that should not be allowed, but sometimes is totally desirable, when only applying minor updates to an assembly. So basically, when multiple assemblies come into play, we get to a case that is not handled well at the moment.

2.1.2 Abstract methods

Abstract methods are simply virtual methods without an implementation. They are translated in exactly the same way as virtual methods, just that no implementation is generated.

```
abstract class ClassD
{
    public abstract void methodA();
}
```

The above class is translated to:

```
struct ClassD {
        virtual void methodA_cb76a80af5d85c1a() = 0;
};
```

In both cases, virtual and abstract, the method name will carry a polymorphic name extension (name signature). If later, someone wants to override this precise method, the compiler will lookup which method is overwritten, and pick its unique name signature to generate the C++ override, which will then override exactly this method, even if there may be hundred other possible methods that could be overwritten, as they will all have different name signatures. This is the principle used in all successive sections.

2.1.3 Replacing Methods

In contrast to C++, C# allows to replace existing virtual methods with new ones. Replacing methods is equivalent to creating a totally new method in the same name just hiding the old one.

```
class ClassA
{
    public virtual void methodA()
    {
        Console.WriteLine("A");
    }
}

class ClassB : ClassA
{
    public new void methodA()
    {
        Console.WriteLine("B");
    }
```

```
}

static void Main(string[] args)
{
    var a = new ClassA();
    var b = new ClassB();

    a.methodA();
    ((ClassA)b).methodA();
    b.methodA();
}
```

The main method will output “AAB”. The translation for the above example looks like this:

```
struct ClassA {
        virtual void methodA_cb76a80af5d85c1a() {
                System::Console::WriteLine11(_T("A"));
        }
};

struct ClassB : public virtual {
        virtual void methodA_d653f90e0a1578bf() {
                System::Console::WriteLine11(_T("B"));
        }
};
```

In C++ a virtual method cannot be replaced, so the obvious solution is to create a whole new virtual method instead, especially since the replaced method is still available for direct access and thus not completely replaced.

2.1.4 Interfaces

Mapping the interface semantics is certainly one of the most complex parts of the translation process. I will step through the emitter in a case by case manner. Some technical details will be omitted, especially when it comes to how to detect those cases based on AST, as it is a hacky process again, also due to the fact that MCS doesn’t need this information to validate C# files but we need it. So it boils down to creating our own algorithms to analyze C# types (which can be found in the source code and are quite involved).

An unfortunate observation is that C++ does not support interfaces at all. It can be emulated with abstract classes and multiple inheritance. But that is not easy, as we will see, since C# interfaces and polymorphism in general have a lot of nasty details that just don’t map to C++ directly. Let’s start with a simple interface definition in C#:

```
interface IfaceA
{
    void methodA();
}

interface IfaceB
{
    void methodA();
}
```

The compiler will generate the following C++ code:

```
struct IfaceA : public virtual System::Object {
        virtual void methodA_cd56d8ae0f0b06b2() = 0;
};

struct IfaceB : public virtual System::Object {
        virtual void methodA_d653f90e0a1578bf() = 0;
};
```

C# supports explicit and implicit interface implementation. If we define a class implementing both above interfaces, then one may already notice the issue we would have in C++, given those two interfaces would just be represented as abstract classes with the same method. In C++, both abstract methods in the same name would be linked to the very same implementation. While it is natural in C# to have the same behavior (the trivial case about which I won't talk here), let's look at using different implementations depending on which interface is used to access the object instance.

```
class ClassA : IfaceA, IfaceB
{

    void IfaceA.methodA()
    {
        Console.WriteLine("A");
    }

    public void methodA()
    {
        Console.WriteLine("B");
    }
}
```

The above code will write "A", if *methodA* is called from `IfaceA`, and it will write "B" otherwise. What happens here is that `IfaceA` is implemented explicitly, meaning that only that particular interface, preceding the method name, will be able to call this implementation. `IfaceB` and the class itself instead will always refer to the public implementation. The compiler generates the following code:

```
struct ClassA : public virtual System::Object, public virtual IfaceA,
                        public virtual IfaceB
{
        void methodA() { System::Console::WriteLine11(_T("A")); }
        void methodA2() { System::Console::WriteLine11(_T("B")); }

        // Automatically generated method stub
        virtual void methodA_cd56d8ae0f0b06b2() override {
                return methodA();
        }

        // Automatically generated method stub
        virtual void methodA_d653f90e0a1578bf() override {
                return methodA2();
        }
};
```

The first two methods are implementations, the last two are so called method proxies, which are inserted specifically to emulate the C# interface semantics. As introduced further above, those two methods have the same name within their respective interface. So when an instance of `ClassA` is cast to `IfaceA`, then calling *methodA* will write "A" and if it is called through `IfaceB`, then it will write "B". Further,

when the class instance itself is used to call *methodA*, then no proxy is used and *methodA2* is invoked directly. This does exactly what we want, but can be quite hard to follow at first.

The main question here is how we know at call-site which of the three choices we need to invoke, when we want to call *methodA*. The answer is simple, as each method call is always preceded by an expression (or implicit this) and the type of that expression determines exactly what method we need to call. We only need to make sure that we can even distinct between those three cases, which would not be possible with plain C++ polymorphism, but it is possible with the polymorphic name extension above.

The following is a case where both interface methods are explicitly implemented, which means if the class didn't have a method in the same name itself, those methods would only be accessible through interfaces, not through a class instance.

```
class ClassC : IfaceA, IfaceB
{
    void IfaceA.methodA()
    {
        Console.WriteLine("A");
    }

    void IfaceB.methodA()
    {
        Console.WriteLine("B");
    }

    public void methodA()
    {
        Console.WriteLine("C");
    }
}
```

This translates in pretty much the same way, and which one of the three methods is called depends on the call site's expression type again.

```
struct ClassC : public virtual System::Object, public virtual IfaceA,
                        public virtual IfaceB {

        void methodA() { System::Console::WriteLine11(_T("A")); }
        void methodA2() { System::Console::WriteLine11(_T("B")); }
        void methodA3() { System::Console::WriteLine11(_T("C")); }

        // Automatically generated method stub
        virtual void methodA_cd56d8ae0f0b06b2() override {
               return methodA();
        }

        // Automatically generated method stub
        virtual void methodA_d653f90e0a1578bf() override {
               return methodA2();
        }
};
```

Interfaces can also be implemented through virtual or abstract methods.

```
abstract class ClassD_Base : IfaceA, IfaceB
{
    void IfaceA.methodA()
    {
        Console.WriteLine("A");
    }
```

```
    public abstract void methodA();
}

class ClassD : ClassD_Base
{
    public override void methodA()
    {
        Console.WriteLine("B");
    }
}
```

Which translates to:

```
struct ClassD_Base : public virtual System::Object, public virtual IfaceA, public virtual
IfaceB {

        void constructor();
        void methodA() { System::Console::WriteLine11(_T("A")); }
        virtual void methodA_cb76a80af5d85c1a() = 0;

        // Automatically generated method stub
        virtual void methodA_cd56d8ae0f0b06b2() override {
                return methodA();
        }

        // Automatically generated method stub
        virtual void methodA_d653f90e0a1578bf() override{
                return methodA_cb76a80af5d85c1a();
        }
};

struct ClassD : public virtual ClassD_Base {

        virtual void methodA_cb76a80af5d85c1a() override{
                System::Console::WriteLine11(_T("B"));
        }
};
```

C# also allows reimplementation of interfaces. This is one of the features that certainly should have never made it into C# as it is so prone to accidental misuse with unpleasant consequences. If one derives from a class that implements `IfaceA` but maybe not know that it does, and then thinks his class should certainly implement that interface too, then we get something like this:

```
class ClassE : ClassD, IfaceA, IfaceB
{
    public void methodA()
    {
        Console.WriteLine("E");
    }
}
```

What happens here is, that we override the implementation of `ClassD`'s interface implementation for `IfaceA` and `IfaceB`. This is really innocently looking, but can cause evil runtime bugs that are hard to track down (the compiler will most likely generate a warning, but still). The translation for this code looks like this:

```
struct ClassE : public virtual ClassD {

        void methodA() { System::Console::WriteLine11(_T("E")); }
```

```
        // Automatically generated method stub
        virtual void methodA_cd56d8ae0f0b06b2() override {
                return methodA();
        }

        // Automatically generated method stub
        virtual void methodA_d653f90e0a1578bf() override {
                return methodA();
        }
};
```

Finally, let's look at a simple test program, exercising all the previously defined classes:

```
class Program
{
    static void Main(string[] args)
    {
        ClassA a = new ClassA();
        ((IfaceA)a).methodA();
        ((IfaceB)a).methodA();
        a.methodA();

        ClassC c = new ClassC();
        ((IfaceA)c).methodA();
        ((IfaceB)c).methodA();
        c.methodA();

        ClassD d = new ClassD();
        ((IfaceA)d).methodA();
        ((IfaceB)d).methodA();
        d.methodA();

        ClassE e = new ClassE();
        ((IfaceA)e).methodA();
        ((IfaceB)e).methodA();
        e.methodA();

        Console.ReadLine();
    }
}
```

This translates to:

```
struct Program : public virtual System::Object {

        static void Main(cli::array<System::String*>* args)
        {
                ClassA *a = cli::gcnew<ClassA>();
                cli::cast<IfaceA*>(a)->methodA_cd56d8ae0f0b06b2();
                cli::cast<IfaceB*>(a)->methodA_d653f90e0a1578bf();
                a->methodA2();

                ClassC *c = cli::gcnew<ClassC>();
                cli::cast<IfaceA*>(c)->methodA_cd56d8ae0f0b06b2();
                cli::cast<IfaceB*>(c)->methodA_d653f90e0a1578bf();
                c->methodA3();

                ClassD *d = cli::gcnew<ClassD>();
                cli::cast<IfaceA*>(d)->methodA_cd56d8ae0f0b06b2();
                cli::cast<IfaceB*>(d)->methodA_d653f90e0a1578bf();
                d->methodA_cb76a80af5d85c1a();

                ClassE *e = cli::gcnew<ClassE>();
                cli::cast<IfaceA*>(e)->methodA_cd56d8ae0f0b06b2();
```

```
            cli::cast<IfaceB*>(e)->methodA_d653f90e0a1578bf();
            e->methodA();

            System::Console::ReadLine();
        }
};
```

Notice how the method name is chosen depending on the expression type. In C# it is hard to tell what method is called without knowing the semantics well, but here it is rather obvious. Both programs give the following output:

```
A B B
A B C
A B B
E E E
```

2.1.5 Nested Classes

C# supports arbitrarily nested classes with arbitrary but non-cyclic dependencies between each other. That means a class defined before another one can depend on it and vice versa at the same time. The following demonstrates this issue:

```
class ClassA
{
    public class NestedA : ClassB.NestedB { }
}

class ClassB : ClassA
{
    public class NestedB { }
}
```

This cannot be mapped to C++ directly, as `ClassA` needs to be defined before `ClassB` can be defined, but `NestedB` and thus `ClassB` needs to be defined before `NestedA` and thus `ClassA` can be defined. This can only be solved by pulling all nested class to top-level and thread them as normal classes. This causes various issues that are mostly easy to deal with. One of them are generic classes, like

```
class ClassA<T1>
{
    public class NestedA<T2> : ClassB.NestedB { }
}
```

which need to be unfolded to

```
class NestedA<T1, T2> : ClassB.NestedB { }
```

, with all the implicit special casing caused by differing from the way MCS handles nested classes.

2.1.6 Type Casting

In C# there are five forms of type casts that are supported by the compiler. All of them can be implemented using C++ dynamic casting, which makes sure that only compatible types and instances can be cast into each other. The full implementation for all cases can be found in *cli/include/casts.hpp*.

In the following I will only talk about boxing, since it is the most interesting kind of type cast. C# distinguishes between classes and structures. Classes are always passed by reference, structs are always passed by value, also called value types. Now there are cases in which structs may implement an interface and a method which might accept that interface. C# allows structs to be passed as interfaces they implement and of course also to be cast into object. Both cases require boxing, a process which creates a version of a value type on the heap. What happens is that C# creates an exact byte-wise copy of the structure on the heap which is then threaded as a boxed structure. If someone changes a boxed instance, the original instance on the stack is not affected by it. This can lead to unexpected behavior, when passing a structure as interface to a method which then changes the structure through an implemented method, as this won't change the parameter passed to the function but only the temporarily created boxed instance that only exists inside the called method.

The whole concept maps well to C++, which basically doesn't have a reference and value-type concept, but threats everything in the same way. Complicated objects with inheritance and plain old data types can both live on the stack and on the heap and they also do frequently. The general concept for translation is that classes are always heap allocated, and structs are always stack allocated, unless they are boxed, in which case they are duplicated on the heap. The compiler emits a special *cli::box* call to initiate this transition.

At some point one usually wants to get back the original value type, which requires a *cli::unbox*, also emitted by the compiler. Unboxing can fail, since it can be applied to any given object instance. At compile time it may not be possible to determine if that cast is valid and since a value type cannot be null, an exception is thrown should the cast be invalid. Boxing can never fail, since it is known at compile time if the cast will succeed or not.

Example Code

The example application also contains the following code section which instruments some of the type casting code:

```
TestStruct a = new TestStruct();
a.a = 9;
ITestFace a_boxed = a;
a.a = 7;
Object a_boxed2 = a;
a.a = 13;
WriteCheck(a.a, 13);
a = (TestStruct)a_boxed;
WriteCheck(a.a, 9);
a = (TestStruct)a_boxed2;
WriteCheck(a.a, 7);

a_boxed = a as ITestFace;
a_boxed2 = a as Object;

object[] arr = {true, (byte)1, (sbyte)2, '3', (short)4, (ushort)5, (int)6, (uint)7,
                                              (long)8, (ulong)9, };
```

```
WriteCastCheck<int>(arr[0], "boxed:bool -> int", false);
WriteCastCheck<bool>(arr[0], true, "boxed:bool -> bool", true);
```

which translates to

```
TestStruct a = cli::ctor<TestStruct>();
a->a = 9;
ITestFace *a_boxed = cli::box<ITestFace*>(a);
a->a = 7;
System::Object *a_boxed2 = cli::box(a);
a->a = 13;
Testing::WriteCheck(a->a, 13);
a = cli::unbox<TestStruct>(a_boxed);
Testing::WriteCheck(a->a, 9);
a = cli::unbox<TestStruct>(a_boxed2);
Testing::WriteCheck(a->a, 7);

a_boxed = cli::cast<ITestFace*>(cli::box<ITestFace*>(a));
a_boxed2 = cli::cast<System::Object*>(cli::box(a));

cli::array<System::Object*> *arr = (new cli::array<System::Object*>({
        cli::box(true), cli::box((System::Byte)1), cli::box((System::SByte)2),
        cli::box(u'3'), cli::box((System::Int16)4), cli::box((System::UInt16)5),
        cli::box((System::Int32)6), cli::box((System::UInt32)7),
        cli::box((System::Int64)8), cli::box((System::UInt64)9)}));

WriteCastCheck_1<int32_t>(arr->at(0), _T("boxed:bool -> int"), false);
WriteCastCheck_12<bool>(arr->at(0), true, _T("boxed:bool -> bool"), true);
```

It's a little bit more verbose than one could hope for and sure can be compacted somehow, but for now this is the outcome.

The cast check is basically a method that validates two other different cast mechanisms and also compares the value against a reference, should the cast succeed.

```
static void WriteCastCheck<T>(object value, string desc, bool expect)
{
    bool res1 = true;
    try
    {
        var test = (T)value;
    }
    catch (InvalidCastException)
    {
        res1 = false;
    }

    bool res2 = value is T;
…
```

This method has a quite pretty representation in C++:

```
template<class T>
void CastTest::WriteCastCheck_1(System::Object* value, System::String* desc,
                      bool expect)
{
    bool res1 = true;
    try {
        T test = cli::unbox<T>(value);
    }
    catch(System::InvalidCastException*) {
        res1 = false;
    }
```

```
    bool res2 = cli::is<T>(value);
…
```

The full code exercises a lot of different type cast scenarios, but certainly not all of them. As with all other tests inside the example application, the most notable thing about them is that one needs more of it.

2.1.7 Object Construction

Currently, it is only possible to allocate an object, but it will never be released. The following will allocate a new object of type `TObject` and forward all parameters to the object's constructor:

```
template<class TObject, class ... TArgs>
TObject* gcnew(TArgs&&... args)
{
        TObject* res = (TObject*)malloc(sizeof(TObject));
        memset(res, 0, sizeof(TObject));
        new (res) TObject();
        res->constructor(std::forward<TArgs>(args)...);
        return res;
}
```

All objects are initialized with zero, just like in C#. The C++ constructor has a default implementation, unless there is a static constructor (2.1.8 Static Constructors) in which case it will make sure to call it once. Instead, real construction is outsourced to a method, called *`constructor`*. This allows us to web multiple constructors together and establish a call hierarchy between them, something that is not possible in C++, but supported in C#.

2.1.8 Static Constructors

C# supports static constructors that are called before any instance constructors are called. In contrast to C++, static constructors are not called before a particular class is accessed at runtime. In C++, static constructors (which don't actually exist but I am talking about the way one uses to emulate them) are called non-deterministically at application start and thus totally incompatible. Since we are using a special constructor method already, it is simple to add this C# logic to translated code by using the real C++ constructor and implementing the logic in there. This will ensure that the related initialization code is called whenever an instance of a class is created. Further, we would need to call this logic whenever a static method is accessed. There is an issue when it comes to fields, since in C++ it is not possible to intercept field access. Instead one would have to track field access in the compiler and emit initialization code for all field accesses made from outside the class. Currently, the static constructor is only called when constructing an instance of a class.

In the following, the typical initialization code for classes with static constructors is presented:

```
OptimizedCRC::OptimizedCRC()
{
        static volatile bool initialized = false;
        static std::recursive_mutex mutex;
        if(!initialized)
        {
```

```
            std::lock_guard<std::recursive_mutex> lock(mutex);
            if(!initialized)
                static_constructor();
            initialized = true;
        }
}
```

The above initialization code uses double-checked locking to optimize performance. Further, recursive mutexes are used to handle the case in which object instances are created from inside a static constructor. This is discouraged for obvious reasons, but not forbidden in C#. The case in which multiple instances are created from different threads is correctly handled also.

Even if there is no static constructor in C#, one must be generated implicitly when having non-constant static members with in-line initialization. This is not supported in C++ and needs to be emulated by initializing static variables from inside a static constructor, which is also not implemented yet.

2.1.9 Generics to Templates

One corner-stone of supporting the C# object model are generic types. Luckily, they map very well to C++ templates, except for the compile time validation, which is also not necessary as we are only dealing with correct C# code. The compiler just changes the syntax of generic classes to fit C++, but not the semantics. This seems to generate valid code, even though I can't offer more of a proof than that.

That sounds simple enough. Unfortunately, the devil is in the details here. While generic classes are easily emitted as templates, they cause a huge mess inside the compiler itself. The primary issues are that each type parameter can appear everywhere in the code and methods with generic arguments in base classes which are instantiated in subclasses can be overridden by non-generic methods. In short, we need to carry those special type parameters and arguments through all algorithms, from basic method lists up till method equality comparison and hash table algorithms. It is a huge effort to properly implement generic type handling in the compiler, even though once we have the infrastructure to recognize and work with them properly, the mapping to C++ is more or less trivial.

2.1.10 Virtual Generic Methods

A virtual generic method is a method with generic parameters that supports overriding. This is inherently not supported by C++ and would require some sort of emulation. One thing to think about is using the most general types permitted by generic parameters and making a virtual method out of it, emitting type casts where necessary. But since the whole *mscorlib* does not contain one single case like this, there was no reason to even think about implementing this rare design pattern. Also note that generic method does not mean that the method can't have generic arguments, just no generic parameters. So virtual methods with generic arguments are fully supported.

2.1.11 Garbage Collection

C# is based on garbage collection. Of course, C++ doesn't really have one up till now, so this calls for trouble. The compiler currently ignores memory management altogether. Objects that are allocated, will never be released, there is no mechanism for it. There are some hacky GCs for C++, but they will cause a huge performance penalty. C++.NET has shown that a GC is possible, and since the translated code is semantically equivalent to C# without any pointer obfuscation and the like, a GC for this special subset of C++ code should work quite similarly to C++.NET. I consider this more or less a solved problem, in some way something that could technically be done and is not invented here. I will refer from time to time to "assuming GC", which means that a given code sample, idea, solution etc. will work well if there was a GC. An example is throwing exceptions by pointer instead of by value. Further, C++11 has basic GC preparations and it is an ongoing discussion if and how a C++ GC should be implemented [6].

2.2 Hand-crafted Infrastructure

Even though the translation is mostly fully automatic, there are some special cases that need manual crafting of types and infrastructure. This is restricted to the most fundamental things, like builtin-types, enumerations, delegates and the like. Not only is there no code stored in the MCS code base, but also do these types require a lot of complex interactions and handling, that cannot be automatically translated (at least not efficiently) even if there was code for it, as their interaction and dependency with the language and runtime is too involved.

2.2.1 Delegates

This section makes heavy use of C++11 features and it is quite hard to think about how this could have been done with C++03. The general approach is to create a universal delegate wrapper class in C++ that can handle all cases occurring in C#. This way all related C# code sections can be translated smoothly.

C++ Type Definition

Defining a delegate type is everything but easy. It took a lot of try and error and especially hundred thousand lines of C# code to parse and compile until I could say with some confidence that the following just works, in combination with the delegate instantiation introduced later.

```
template<class T> struct delegate;

template<typename Ret, typename... Args> struct delegate<Ret(Args...)> :
	public ::System::MulticastDelegate
{
	typedef Ret signature(Args...);

	bool get_HasSingleTarget() { return true; }

	Ret Invoke(Args...)
	{
		return InternalInvoke(std::make_tuple(args...));
	}

	virtual Ret InternalInvoke(std::tuple<Args...> args)
	{
		return Ret();
	}

	System::IAsyncResult* BeginInvoke(
		Args...,
		System::AsyncCallback* callback,
		System::Object* context)
	{
		return nullptr;
	}

	Ret EndInvoke(System::IAsyncResult* result)
	{
		return Ret();
```

```
        }

        template<class T> T Cast()
        {
                typedef typename std::remove_pointer<T>::type::signature toSignature;

#if _DEBUG
                std::function<signature> from;
                std::function<toSignature> to = from;
#endif

                static_assert(
                        std::is_pointer<T>::value &&
                        std::is_same<
                                delegate<toSignature>,
                                typename std::remove_pointer<T>::type
                        >::value,
                        "Target type shall be a pointer to a cli::delegate.");

                return (T)this;
        }
};
```

So basically it derives from C#'s fundamental delegate type, thus providing the standard interface that is expected by the compiler. Further it implements the compiler generated methods ***Invoke***, ***BeginInvoke*** and ***EndInvoke***. Those are in a fashion also compiler generated on C++ side, since it's a template, and those methods inherit their arguments solely based on the C++ function type the delegate type is instantiated with. Additionally, delegates can be cast into other delegates, as long as the cast honors contravariance of parameters and covariance of return values. This is validated in a special ***Cast*** method which is emitted explicitly for all delegate conversion on C# side. Of course a working compiler should only emit valid casts anyway, but this is just a safety net in case something is broken, as it can lead to ugly errors at runtime.

Delegate instantiation will use derived classes to encapsulate a particular context. ***InternalInvoke*** is introduced to allow derived classes to override the precise way in which a delegate invocation is dispatched. The variadic arguments are packed into a tuple, since I am not even sure if it is possible to override a variadic method in a subclass and if it is it might be much more complicated than it already is. A tuple should not come with any runtime performance costs and can be unpacked within derived classes to call an arbitrary target method.

Instantiating A Delegate

Instantiating a delegate is the process of creating an object instance of a specified delegate type that encapsulates a certain method call in this abstract delegate interface. Currently, there are three types of encapsulation available. The first one binds to a regular C++ method, the second one to a C++ functional/lambda expression, and the third one to a hoisted C# lambda expression or instance member method. In the following, the templates for encapsulating each of those types are prototyped:

```
template<class T>
delegate<typename get_signature<T>::type>* bind(T func);

template<class T>
delegate<typename get_signature<decltype(&T::operator())>::type>* bind(T func);
```

```
template<class TStorey, class TFunc>
delegate<typename get_signature<TFunc>::type>* bind(TStorey storey, TFunc storeyFunc);
```

They automatically derive the delegate type from given parameters and use S.F.I.N.A.E.[3] to find the correct overload. The implementations can be found in *cli/include/Delegates.hpp*. In the following I will describe one of them.

```
template<int ...> struct template_seq {};
template<int N, int ...S> struct template_gens : template_gens<N-1, N-1, S...> {};
template<int ...S> struct template_gens<0, S...>{ typedef template_seq<S...> type; };

template<typename TFunc, class TStorey, class TStoreyFunc>
struct storey_delegate_binder : public delegate<TFunc>
{
	typedef get_signature<TFunc> signature;

	TStorey storey;
	TStoreyFunc storeyFunc;

	storey_delegate_binder(TStorey storey, TStoreyFunc storeyFunc) :
		storey(storey), storeyFunc(storeyFunc) { }

	virtual typename signature::return_type InternalInvoke(typename signature::args
args)
	{
		return Dispatcher(
			typename template_gens<
				std::tuple_size<typename signature::args>::value
			>::type(), args);
	}

private:
	template<int ...S>
	typename signature::return_type Dispatcher(
		template_seq<S...>, typename signature::args args)
	{
		return (storey->*storeyFunc)(std::get<S>(args)...);
	}
};

template<class TStorey, class TFunc>
delegate<typename get_signature<TFunc>::type>* bind(TStorey storey, TFunc storeyFunc)
{
	return new storey_delegate_binder<
		typename get_signature<TFunc>::type,
		TStorey,
		TFunc
	>(storey, storeyFunc);
}
```

The first three templates were copied from StackOverflow[4]. They are used to unpack a given tuple into a list of actual method parameters. *InternalInvoke* measures the size *N* of this tuple, which is the amount of parameters of the delegate and also generates a list of integers, from zero to *N-1*. Those parameters are passed to a private *Dispatcher*. It uses variadic template pattern matching to transform the list of integers into a variadic parameter pack. The member function given by the instance *storey* and the member function pointer *storeyFunc* is then called with its original argument list by unpacking

[3] http://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error

[4] http://stackoverflow.com/questions/7858817/unpacking-a-tuple-to-call-a-matching-function-pointer

the delegate's invocation parameters, which we received as a tuple, into actual parameters. This is achieved by using the variadic unpack operator *"…"* on the tuple, which yields something like this as a compiler's internal state:

```
return (storey->*storeyFunc)(std::get<0>(args), std::get<1>(args), std::get<2>(args),
etc.);
```

And thus causes the tuple to be unpacked into a parameter list.

The meta-template *get_signature* returns the cleaned up function type for a given function type, which consists only of the return value and its parameters, using pattern matching again.

```
template<typename T> struct get_signature;

template<typename Mem, typename Ret, typename... Args> struct get_signa-
ture<Ret(Mem::*)(Args...) const> {
        typedef Ret type(Args...);
        typedef Ret return_type;
        typedef Mem member_type;
        typedef std::tuple<Args...> args;
};
```

There are more overloads of this template located in *Delegates.h*.

Deriving Custom Delegates

A custom C# delegate is then mapped to a C++ template alias. It supports generic delegates by using an alias with its own template parameters, for instance:

```
template<class T> using Action = cli::delegate<void (T obj)>;
```

This alias represents the C# delegate:

```
delegate void Action<in T>(T obj);
```

That is all there is to it, thanks to the previously developed C++ magic.

2.2.2 Enumerations

Enumerations have no equivalent in C++ and need special attention. They are also a very common language construct in C#, so a good mapping is desirable. The primary properties that do not map to C++ are reflection and string conversions. All C# enumerations derive from `System::Enum` and we start by simply reproducing this inheritance. Further, each mapped enumeration needs to provide all its members in string and value form. In the following there is a basic skeleton for such a base class that also shows a basic *ToString* method, that doesn't account for flags (OR-ing enum members) yet.

```
template<class T>
struct enumeration : public System::Enum
{
        T value;

        virtual const std::vector<T>& GetValues() const = 0;
```

```
        virtual const std::vector<::System::String*>& GetStrings() const = 0;

        virtual ::System::String* ToString_1636a0751cb9ac11() override
        {
                for(int i = 0; i < GetValues().size(); i++)
                {
                        if(GetValues()[i] == value)
                                return GetStrings()[i];
                }

                return _T("[Unknown]");
        }
};
```

Here is example of how to map a simple C# enumeration to a C++ type:

```
enum class TaskStatus
{
    Created = 0,
};

struct TaskStatus_Impl : public cli::enumeration<TaskStatus>
{
    TaskStatus_Impl(const TaskStatus& val) : cli::enumeration<TaskStatus>(val) { }
    operator TaskStatus() { return value; }

    virtual const std::vector<TaskStatus>& GetValues() const override
    {
        static std::vector<TaskStatus> res =
        {
            TaskStatus::Created,
        };
        return res;
    }

    virtual const std::vector<::System::String*>& GetStrings() const override
    {
        static std::vector<::System::String*> res =
        {
            _T("Created"),
        };
        return res;
    }
};
```

This implementation avoids unnecessary memory allocation by using the singleton pattern for returned vectors. Each enumeration comes in two flavors, the actual C++11 version of this enumeration and the C# object form. The compiler takes care of the conversion and will only use the costly object form when its member methods need to be accessed. This is also the standard procedure for other fundamental types.

The actual code will later only get the enumeration type, as in ***`TaskStatus`***. It needs a means to translate this type back into the implementation type ***`TaskStatus_Impl`***. To make this possible, each enumeration enters a template specialization into the ***`cli`*** namespace:

```
template<> struct enum_to_impl<System::Threading::Tasks::TaskStatus>
{
        typedef System::Threading::Tasks::TaskStatus_Impl type;
};
```

This meta-template allows arbitrary code to map enumerations to their implementation at compile time.

2.2.3 Builtin Types

Fundamental data types in C#, like `int`, `long`, `float`, etc. do provide member methods. To maintain maximum speed, the compiler only switches to object form when needed. Otherwise, the native C++ types are chosen instead. This works by intercepting member access on fundamental types and then emitting a very simple C++ wrapper *`cli::import()`*. This will map a fundamental type given in its native form into its object form. There is no heap-based memory allocation involved, but the usual virtual inheritance/object overhead.

2.3 Translation Schemes

The translation process is a fine-tuned process of many interacting small code emitters. Most of those emitters alone are pretty boring to look at, like the for-loop, as nothing intriguing happens in there. A few of them are more complex but so convoluted or specialized that it would basically cause a huge hiccup trying to describe them here. In the following I will focus on more complex emitters that can be explained well without digging too deep into the inner working of the code. But it is by no means a complete coverage, rather the top of the ice-berg.

2.3.1 Method Emitter

The method emitter is responsible for emitting both, declaration and definition. A method declaration is always inside a class and has the following general structure:

```
[template<…>]
[static|virtual] void|typename MethodName(…) ["=0"|override|final];
```

Some real life examples include:

```
virtual void Decode_cefcb854d4bf305c() override;

void OnProcessingInstruction(System::String* name, System::String* text);

virtual T Creator_c074cf904cf11d1e() = 0;

template<class TResult>
Task2<TResult>* StartNew_1(System::Func<TResult>* function);
```

C++11 supports both, the `override` and `final` modifier and applies the associated correctness checks with them. This is useful to detect bugs in the compiler.

Mapping the parameter types can be a quite convoluted process, thanks to a lot of special casing and recursion, but in general it maps classes by pointer and structures by value. There is a special case in which we need to handle `__arglist`, which is a rare C# keyword to accept C-style variable argument lists in interop-code. This maps just to the latter in C++ too. The C# `params` keyword is resolved by MCS and converted into an array, into which all associated parameters are written to. So no special handling is required here. For instance,

```
string GetText (string fmt, params object [] args)
```

maps to:

```
System::String* GetText2(System::String* fmt, cli::array<System::Object*>* args);
```

A method definition has a similar structure, except that there are no inheritance modifiers and no static or virtual keyword. Also, since a definition is always emitted outside of a class body, the method name is preceded by its fully qualified class name and if the class is generic, a method definition can have up to two template clauses preceding it:

```
template<class TResult>
template<class TAntecedentResult>
Task2<TResult>* TaskFactory2<TResult>::ContinueWhenAll_1(…) { … }
```

In that case the first template clause refers to the class and the second to the method.

The method body emitter is simply a recursive AST traversal that is calling the emitters for each tree node.

2.3.2 Switch-Statement

In C# the switch statement is more complicated than one would expect and needs special handling to be mapped to C++. Here is an example of a more advanced switch-statement:

```
switch(v)
{
    case -1:
    case 0:
        int a = -1;
        goto case 2;
    case 1:
        a = 1;
        goto default;
    case 2:
        break;
    case 3:
        if(v == 3)
            break;
        a = 2;
        break;
    default:
        goto case 0;
}
```

C++ has no built-in mechanism to map the `goto`s as well as the variable declaration of *a* which is implicitly available in all successive case branches. To map the `goto`s, all branches are assigned a unique label, to which we can then jump via C++ `goto`. The variable *a* causes more trouble. It would technically be possible to detect such usages and pull them into the enclosing scope automatically, thereby potentially breaking the code. But I chose not to, as in my view, the mere usage of this pattern is already bad coding practice and occurs only one single time in the 300k lines of *mscorlib*. Also the effort to pull this off reliably is substantial. Currently, to translate such code, one will need to manually fix the variable declaration in the C# source code prior to compilation.

The manually fixed and then translated code will look somewhat like this:

```
int a = -1;
switch (v)
{
        case -1: case 0: case_0: {
                goto case_2;
                break;
        }
        case 1: case_1: {
                a = 1;
                goto case_4;
                break;
        }
        case 2: case_2: {
                break;
        }
```

```
        case 2: case_3: {
                if(v == 3)
                        break;
                a = 2;
                break;
        }
        default: case_4: {
                goto case_0;
                break;
        }
}
```

Besides that, C# also supports `switch` on strings. This is handled by introducing a temporary variable and assigning a match number to it. Each `case`-string is then compared against the `switch`-variable and if they are equal, the associated match number is assigned to the temporary variable. Then a normal `switch` block is emitted, just that this time we don't `switch` on the string but on the match number.

2.3.3 Strings and Decimals

In C#, strings and decimals can be compile time constants. As such they cause no additional memory allocation when returned from a function, for instance. Since we don't want to generate slower code than C#, it's a good idea to imitate this behavior, instead of always associating strings and decimals with memory allocation. But since both derive from `System::Object` the solution is not obvious at first. C++ does not support compile time constants with a more or less unrestricted object model. There is a new feature called `constexpr` which partly allows for objects to be immutable and allocated as compile time constant, but due to the complexity of strings and decimals that is no solution in this case. Instead we utilize a template and preprocessor trick, which works as follows:

```
template<int T, int line> ::System::String* string(const char16_t* str)
{
        static ::System::String* value = new ::System::String();
        return value;
}

#define _T(str) ::cli::string<__COUNTER__, __LINE__>(u##str)
```

Now when translating a C# string to C++, we simply wrap it in the above *_T* macro. This won't give us a constant expression, but it will give us a singleton instance of the string. That is, no matter how often the code line in which we called *_T* is executed, we always get the very same object instance. This works thanks to two tricks. First we obtain a unique number using the preprocessor macro *__COUNTER__* which is auto-incremented on every occurrence and pass it as template parameter. Further we also need to pass the line number in which the string was created, because *__COUNTER__* is only guaranteed to be unique within the same compilation unit. Of course, this is still not guaranteed to be unique, but the probability that two string are mapped to different compilation units with the same line and counter is extremely low. Second, we abuse the fact that each unique template instantiation will receive its own set of static local variables. Thus we have our unique singleton variables.

Other than that, strings and decimals are fully implemented in C# and need no further handling so far. There are a few support routines to make it easier to work with strings manually in C++, but they are

not an integral part of the translation process and not described here. The full implementation can be found in *cli/string.hpp*.

2.3.4 Exception handling

Exception handling is rather straightforward. All exceptions are thrown and caught by pointer, to maximize compatibility with the rest of the generated code, which works well assuming GC. C++ provides compatible exception handling except for the `finally` clause. Since C++11 this can easily be handled through a finally-guard, for instance:

```
{
    cli::finally_guard([&]() {
        Finalize_b946fbc32e26afd6();
    });

    System::Array::Clear2(this->block, 0, this->blockSize);
}
```

Which is the generated code for:

```
try
{
        System.Array.Clear(block, 0, blockSize);
}
finally
{
        Finalize();
}
```

In case of additional `catch` clauses, the above code is just wrapped in `try-catch` with a separate C++ `catch` clause for each C# `catch` clause. The finally guard itself is a simple RAII[5] construct which basically just calls its lambda expression on deletion.

2.3.5 Lambda Expressions

MCS uses the AST node `AnonymousMethodBody` to specify lambda methods. In the following, the expression defined by *e* and *x* are C# lambda functions:

```
ProcessTypeGroupsByNamespace(Metadata.InterfacesByInheritence,
    e =>
    {
        if (list.All(x => x != e))
            WriteTypeDefinition(e);
    });
```

If there is a non-null *Storey* member inside `AnonymousMethodBody`, then this lambda expression does depend on enclosing state, which means that methods variables and parameters are accessed in some way. In the above example, both lambda expression would have a *Storey* member, as *e* accesses *list* and *x* accesses *e*. If that happens, the expression is hoisted into a separate object. This is done by the

[5] http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

MCS compiler and those objects are translated just as any other C# type is, with the difference that their definitions are emitted in correct order and types are not pre-declared.

In case the lambda expression does not depend on enclosing state, we can simply use a C++ lambda.

```
auto myLambda = cli::bind([&] (System::String* dv) -> int32_t
{
	return System::Int32::Parse3(
		dv, System::Globalization::NumberStyles::HexNumber);
})
```

The *cli::bind* function introduced above (2.2.1 Delegates) is called for a C++ functional. The body of this functional is generated with the very same code emitter that is used for anything else. Since it is guaranteed that the emitted code does not depend on enclosing state, the semantics of C# and C++ lambdas are equivalent under GC assumption.

In case MCS has generated a *Storey* for this expression, we need an instantiated S*torey* for each invocation of the lambda. That is, MCS has not emitted the allocation and initialization of this *Storey* for us and we need to explicitly emit initialization code before usage. This is done at two occasions. First, if the method containing our lambda expression provides any parameter that is used by the lambda, then we initialize the *Storey* right after method entry and fill its variables with the parameters that are accessed by the lambda. Second, whenever a new variable is initialized that is later accessed by the lambda, we instanciate a new *Storey*, unless there already exists one, and save the initial value in the *Storey*. The fundamental concept here is that all state that is contained inside a *Storey* is the only place where this state is actually stored. For instance, if we declare a variable that is later accessed inside a lambda, but then want to read/write to that variable somewhere on the way, the compiler will always read/write from the *Storey*. The variable declared in C# code does not exist anymore, only inside of the *Storey*. For method parameters it is a bit different. They provide the initial values but then again, all read/writes to that parameter are performed through a *Storey*, even outside of lambda expressions. This way iterative code, where a lambda expression modifies enclosing state and then the enclosing state works with that modified value and calls another lambda, etc. will work just fine, as there is only one place where each variable is stored. Either where it was declared or inside a *Storey*, but never in two places.

All we have to do now is to *cli::bind* to the *Storey* instance, which works by providing the instance as first argument and a method pointer to the *Storey*'s invocation member function as second parameter. The resulting delegate can then be used to call the lambda for the enclosing state defined by our *Storey*.

As an example, the following code snipped

```
public static void Run ()
{
	Console.WriteLine ();
	Console.WriteLine ("Running delegate test...");

	// Lambda test
	Func<int, int> lambdaA = (x) => x * 4;
	int a = lambdaA(4);
	WriteCheck (a, 16);

	// Story test
	a = 1;
	Func<int, int> funcA = (x) => x * a;

	a = funcA (3);
```

```
a = funcA (3);
WriteCheck (a, 9);

a = 1;
for (int i = 1; i <= 3; i++)
{
	a += funcA(i);
}
```

is translated to

```
void DelegateTest::Run()
{
	System::Console::WriteLine();
	System::Console::WriteLine11(_T("Running delegate test..."));
	System::Func2<int32_t, int32_t> *lambdaA = cli::bind(
		[&] (int32_t x) -> int32_t { return (x * 4); });

	auto storey_2 = cli::gcnew<_Run_c__AnonStorey40>();
	storey_2->a = lambdaA->Invoke(4);
	Testing::WriteCheck(storey_2->a, 16);

	storey_2->a = 1;
	System::Func2<int32_t, int32_t> *funcA =
		cli::bind(storey_2, &_Run_c__AnonStorey40::__m__5D);
	storey_2->a = funcA->Invoke(3);
	storey_2->a = funcA->Invoke(3);
	Testing::WriteCheck(storey_2->a, 9);

	storey_2->a = 1;
	for(int32_t i = 1; (i <= 3); i++){
	    (storey_2->a = storey_2->a + funcA->Invoke(i));
	}
```

It illustrates both cases discussed above. The variable *a* is no longer a method variable, but a storey member. If the lambda expression does not access enclosing state, a C++ lambda method is generated, just like for *LambdaA*. For lambda expressions who access enclosing state, a storey class is generated and the lambda method becomes a class member of this storey, for instance:

```
int32_t _Run_c__AnonStorey40::__m__5D(int32_t x) { return (x * a); }
```

3 Utilities

Besides the translation process, there are other important mechanisms that need to be implemented. In the following I will introduce a few different features that fall into the utility category.

3.1 CMake

Just modifying MCS to generate C++ code is not enough to get a convenient framework for translating C# code. Instead we also want it to generate a build system for us.

The translator generates a directory structure with CMake scripts to provide a one-click build system. The directory structure of a standard compilation environment looks as follows. All compiled C# assemblies (the real MCS output, not the C++ code) will land in the *lib* directory. *include* contains all generated pre-declarations for each assembly in **.hpp* files and all template definitions in **.template.hpp* files. The *cli* folder is static and has no automatically generated files in it. Compiled C++ executables land in *lib* too. *src* contains all generated non-template C++ definitions.

The CMake scripts themselves are rather simple and just make an executable out of the C++ files contained in the *src* directory.

3.2 Multiple Assemblies and Dependencies

Normally even the simplest C# programs have various dependencies on other assemblies. Dynamically linking translated C++ assemblies is possible, but would require some changes to fix any linker issues that might popup. It would also cause much bigger files due to duplicated template instantiations.

It was tricky to handle multiple assemblies, since MCS is using things like `ImportedTypeDefinition`s to represent imported types, which give up to no insight into what MCS is referring to. For types, this can be resolved without too much trouble. It gets harder for class members, for which I was not able to create a correct mapping, even though it might be possible somehow. Instead, all dependencies are translated as a whole into one executable program, which completely negates the issues arising from those imported member references. The downside is that if multiple assemblies define a class with the same name in the same namespace, MCS will refuse to compile the assembly due to name clashes. There is no solution for a case like this at the moment.

3.3 Resolving *InternalCall* and *DllImport*

Mono uses a lot of internal calls to handle OS specific, fundamental implementations, like file access. Currently, those method declarations are just mapped to an empty method stub. For instance

```
[MethodImplAttribute (MethodImplOptions.InternalCall)]
public extern static double Exp (double d);
```

is mapped to:

```
double Math::Exp(double d) {
    // TODO: "Unimplemented external method!"
    throw cli::gcnew<System::NotImplementedException>();
}
```

As for now this is not of much help, except for getting compilation done. In practice, one would want to have declarations only and an extra code directory only for external calls, which are then not overwritten on successive translations, as they are now. For `DllImport` basically the same applies, just that the attribute does look differently. There is no marshalling implemented at the moment.

3.4 Debugging Code Generation

One of the primary tools to develop the compiler is a feature I would call "break on generated code". For instance, say the compiler generates a faulty sequence of C++ code and we want to know what's going on. This could be close to impossible to debug normally. The compiler uses a special code writer, which keeps track of all emitted code so far and runs a scanner over the current end of the document. We may just enter a sequence of code we want to break on and the compiler will do so. Then we can inspect the AST using a debugger and follow the code generation as it happens. Without this utility, the development would have been impossible. The code lines implementing this behavior can be found in the method *`CheckBreakpoints`* located in *`CppSourceGenerator.cs`*.

4 Remaining Issues

The compiler could not be finished in the short timeframe set by a bachelor thesis. It can translate the code but will often generate faulty code that can be compiled but does not run correctly. Unfortunately, it is hard to specify exactly what is missing/broken, as this can't be predicted until starting to compile and execute more complex programs.

4.1 Linker

The biggest problem with CMake or C++ compilation in general is speed. I couldn't test compilation with GOLD, the parallel GNU linker, but even with that the linking phase will still be a major bottleneck. Right now we are talking about five minutes for a simple program. I am not sure why this is taking so long, but it is real showstopper during development, since it is hard to iterate and debug over *cli* code when we need to make a coffee break whenever we want to test a new implementation. If the compiler would work already, the linking/compilation wouldn't matter that much, as it is the last step in product development (assuming a conforming implementation) and not a development tool. Also I haven't found a debugger yet that could provide any meaningful feedback for a project of this size. Unfortunately the choices are reduced tremendously by being bound to the latest Clang/GCC version. Microsoft C++ and Intel C++ officially do not support all the C++ features required to compile the generated code yet which makes any attempt on Windows and Visual Studio futile. MinGW as well can only compile but fails to execute or debug the example application for some reason.

For now, I added a special generator that allows one to generate a build system only compiling the user supplied assembly, and not the *mscorlib*. This means that all references to methods and types inside *mscorlib* need manual implementations. The *mscorlib* header will still be used, but no implementations. A few of those manual implementations have been done in the *cli/dev* directory to run basic example code. This allows decent compilation and debugging speed (like only a few seconds to compile and link everything from scratch), providing the usual iteration cycle and thus allowing the core features to be tested and validated until a compilation of the real *mscorlib* becomes feasible. The idea is that if the compiler is standard conforming at some later stage, the *mscorlib* should compile without too much issues and the time consuming iteration cycle on the full code base could be greatly reduced by only needing few iterations.

4.2 TODO

There are a few issues that are known and need attention, of which a lot have already been mentioned throughout the thesis. Another one is code like the following:

```
int[] arr = new int[8];
for(int i = 0; i < 8; ) arr[i++] += 3;
```

Currently, such code is translated like this:

```
cli::array<int>* arr = cli::gcnew<cli::array<int>>(8);
for(int i = 0; i < 8; ) arr->set(i++, arr->get(i++) + 3);
```

Obviously, this is wrong, as `i` is incremented two times instead of just once. This would require recursive unfolding and pre-computation of property and array indices. There will most likely be more details like that, which show up when trying to run through extensive unit-tests.

The `async` keyword and `yield return` are not translated properly at the moment. Further I expect issues with type casts, object construction and static constructors. The `checked`/`unchecked` keywords cannot be trivially implemented either. Also the more common `dynamic` keyword may require considerable effort to be implemented correctly.

I already fixed a lot of runtime errors arising during the development and testing of the example application and the contained benchmark. This was a relatively simple application and the amount of errors to fix was quite high.

Next step would be to start going through the UnitTests coming with Mono itself and porting them to the hand-written *`cli/dev`* framework. When most of those tests pass, it would be time to start compiling & executing the real *`mscorlib`*. To do this in a reasonable manner, it would be necessary to do some kind of dependency walk to only include C++ code that is actually needed, otherwise the linker phase will take minutes to complete.

5 Conclusions

Designing this compiler was very time consuming but also quite revealing. I would say I possessed fairly advanced knowledge in both C# and C++ but during development I gained considerably more knowledge especially in the details one normally doesn't look at, but just takes for granted. Besides the learning experience this project also lays a solid foundation for future work on C# to C++ compilation, which could also be used for a shared C# and C++ backend. Unfortunately, the time-frame for a bachelor thesis, which is only three months, didn't allow to complete the compiler and leaves a lot of work left.

The linker performance leads to an intriguing question as to why linking consumes so much time even on trivial programs, just by the sole size and symbol count of the object files, even though 99% of them are not used in the whole program. This could be an area of future investigation, as it makes no sense how it is right now.

A side note can be made about C++11 support. In theory, GCC should compile it, but given the generally broken state of C++11 standard libraries coming with this compiler, success hugely depends on the specific system an attempt is made on. I could compile the generated code on Debian 7.1 with g++4.7, but it failed on older versions as well as on MinGW. For now, only Clang 3.3 with its own standard library is able to compile the code reliably, regardless of the platform. All other C++11 compilers lack several important features, which further complicates development, since Clang currently lacks a good IDE & Debugger. The Intel C++ Composer 14 will most likely be able to compile it too, at least it is feature complete on paper, unfortunately by the time of this writing only 13.x is available.

The compiler provides a lot of anchors for future research. Besides leaving left quite some work until being standard conform, it also allows completely new approaches and technologies to be incorporated and tested inside of C#. Instead of having to test out new ideas in a VM, which might make it a lot more complex to integrate new technology successfully, one can now simply add new C++ generators and re-use existing technology. For instance, if someone would want to add GPGPU capabilities to C#, then it is now possible to add/modify a C++ generator to emit C++AMP or Cuda++ code directly, instead of having to fiddle those technologies into an existing VM. Also it is much easier to analyze potential optimizations by using C# annotations which could guide the C++ translation. Thinking of disabling array boundary check on a case-by-case basis, pure-functions, const-correctness, pointer aliasing, alignment, MMX, SSE, AVX intrinsics etc. Also it provides a good playground for testing future C++ garbage collectors, by being able to use real world applications that don't care about memory management at all and testing a new GC on them. There are most likely tons of additional scenarios in which this compiler may be useful. Some of them include demanding applications like 3D game logic, which today often has performance critical algorithms and relies on a native backend, usually a 3D engine. In contrast to normal application development, using C++ for game logic is not desirable, no matter the costs, since game logic is often written by game designers who may just not have the knowledge to write decent C++ code (even Unreal Engine 4, which brings C++-only scripting to the table, still has a highly sophisticated visual scripting language which can be used by artists). This is one of the many practical cases where a compiler generating high performance native code out of managed programming languages would be the best solution. And since C++ is often the programming language behind script languages, it is very convenient just to convert a script language to its native host, as this also eliminates any marshalling costs and makes debugging optimized code easy as well, since everything is in the same language.

Bibliography

[1] C++ ISO Standard, "http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf," 2011.

[2] H. Sutter, "C++ and Beyond 2011: Herb Sutter - Why C++?," 2011.

[3] J. Hamilton, "http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx".

[4] C# Specification, "http://www.microsoft.com/en-us/download/details.aspx?id=7029".

[5] Microsoft Research, "http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx".

[6] H. Sutter, "http://herbsutter.com/2011/10/25/garbage-collection-synopsis-and-c/".