NINNES

dotNet and Roslyn behind the covers

Germán Valencia - @XMachinarius

November 8, 2018

Growth Acceleration Partners

Table of contents

Introduction

About this talk

Roslyn is the unsung hero of the .Net world behind most, if not all, C# and Visual Basic compilation processes. This talk aims to shine a light in the backstage of dotNet and, specifically on Roslyn as the fundamental tool of every single dotNet developer.

How it understands code, how it compiles said code into a CLR-compatible binary, and more appliccably to the real world, how all of this can be customized for specific use cases to build Domain-Specific Languages based on C# with an imperative API. With a small focus on NES game programming, because... Why not?.

About the NINNES project

The NINNES project has the ultimate goal of allowing C# programmers to hit F5 on Visual Studio and see their code running on a NES emulator of their choice (not provided by the project itself) and optionally deploy to NES flash carts for execution on real hardware.

To ground such a lofty ideal the project is structured with 3 pillars: C# Code restrictions, NES dotNet Shims Platform, and C# Compilation to the NES CPU.

C# Code Subsetting

In order to program any hardware at all the first step is to know and comply with the restrictions of said hardware's ISA (Instruction Set Architecture), memory and I/O capabilities. To ease the casual C# programmer into the limitations (particularly ISA) of the NES CPU the Roslyn C# compiler must be told to invalidate correct C# code that would make no sense or be impossible to implement in hardware within the limits of said CPU, thus a collection of Roslyn analyzers and code-fixes must be crafted to tailor the language. Visual Basic and other dotNet-compatible languages are outside of the scope of this project.

This pillar is the subject of the talk.

NES dotNet Shims Platform

The NES has its own particular way of doing things which must be mapped to the dotNet realm so a NuGet package providing a dotNet-friendly API to elements like the CPU instructions, the PPU (Picture Processing Unit) and the APU (Audio Processing Unit) of the NES is essential to the success of the main goal.

A mechanism to run the C# code on the standard desktop CLR with display, audio and input capabilities, similar to how XNA used to enable simple game programming, must also be included in the NuGet package so the code may be easily debugged. Direct2D, OpenGL and WebGL with WebAssembly are possible backend targets for this.

C# Compilation to the NES CPU

Actually outputting a valid ROM file that may run on the NES hardware from the C# code, with some possible avenues for this including:

- Code transpilation from C#
  - To C for compilation with CC65, a full featured NES-compatible compiler
  - Directly outputting NES-compatible assembly code from the Roslyn code model
- MSIL recompilation with LLILC and a Clang MOS6502 backend
- Researching the work of the .Net Micro Framework team

Why?

The main objective of this project is to showcase Roslyn's flexibility for custom scenarios with an extreme case study relative to the expected use cases on the usual day to day development, so as to produce a library of Roslyn customization examples covering a wide range of scenarios.

And for 1337 internet points, of course.

From Visual Studio to F5, dotNet behind the curtains

C#, the Idiom and Culture

The programming language we all know and love.

BCL, the Acting Academy

The Base Class Library is the basic set of classes and methods that Microsoft publishes with every dotNet release. This includes the built-in types like int, Integer, string, byte, Byte and methods ranging from Console.WriteLine to Task.Run and the P/Invoke infrastructure.

Common extensions to the BCL range from essentially Win32 API wrappers like Windows Forms to wholly dotNet APIs based on Win32 calls like Windows Presentation Foundation and Windows Communication Foundation.

Roslyn, Costumer Extraordinaire

The dotNet Compiler Platform, or Roslyn as the original codename and the community calls it, was born as a replacement for the ancient C# compiler that was originally written in C++ both as a way to show how the language had been evolving and ”strengthening the ecosystem and becoming the best tooled language on the planet”, as Mads Torgensen puts it.

Roslyn, Costumer Extraordinaire

Roslyn is not only a compiler though, it is an SDK (that Microsoft dogfoods on for Visual Studio on its 3 different flavors) to implement functionality around the C# and Visual Basic code parsing, analysis and compilation processes, with the express purpose of letting people hook into that information and, if so desired, influence it and modify it to suit diverse needs, even if that means strengthening the value proposition for Microsoft's competitors in the C# tooling space.

Roslyn, Costumer Extraordinaire

These are some of the ways Microsoft and the community use Roslyn:

- Syntax Highlighting
- Code Refactoring
- Code Security Analysis
- Code Quality Analysis
- Code Generation

MSIL, the Script

One of the original goals of dotNet was giving Microsoft a hardware platform-neutral and managed way to program Windows, specially for servers of different hardware architectures. To that end, an intermediate language was created as the language of choice for dotNet execution to be translated into native CPU code with a JIT -Just In Time- model.

MSBuild, the Playwright

The build system of choice for standard dotNet projects is MSBuild, a tool that reads project files (.csproj, .vbproj and so on) and invokes Roslyn and other associated tools on the input files to finally produce the MSIL .exe file.

RyuJIT, the Director

RyuJIT falls in the Just In Time model of dotNet execution as the translator from MSIL to native CPU code.

CLI, the Stage

The Common Language Infrastructure is the open ECMA standard that any platform that wishes to run dotNet code must implement, with CLR -Common Language Runtime- being Microsoft's official implementation. The CLI dictates how MSIL code must be run, including how a JIT (RyuJIT in the CLR) is to be used.

DLR, the Improv Guide

The Dynamic Language Runtime is a relatively new addition to the dotNet ensemble, primarily to enable support of dynamic languages like Python, Ruby, and Dynamically-Typed C# to be compiled into MSIL and hence executable wherever a capable CLI implementation exists.

dotNet Native, the Cameraman

dotNet native ¨records¨ RyuJIT in action and generates a completely native (yet still BCL and CLI dependent) binary for any target CPU architecture. This is usually done for extremely performance-sensitive workloads, specially so in the case of those sensitive to startup delays.

From Visual Studio to the NES

The Limits of the NES

The NES runs on a MOS6502 8-bit CPU implementation from Ricoh, albeit without the decimal/floating point mode enabled. This brings forth several limitations:

- No floating point calculations
- No multiplication or division implemented in hardware
- No Operating System
- Limited memory access capabilities

Removing dotNet from C#, 8 bits at a time

To be able to fit a C# program inside the NES some concessions must be made:

- No Garbage Collection
- No Threading
- No Operating System services
- No DLR
- No Reflection
- No Sockets/Communication
- And much more NOs!

MSIL and 8-bit CPUs don’t mix

Bringing C# compilation output to a MOS6502-compatible format may entail

- Assembler compilation from Roslyn ASTs with a C# Transpiler
- MSIL recompilation with LLILC
- Researching the work of the .Net Micro Framework

Roslyn is a Friend, not an Enemy

General Roslyn Services

- Workspace, Solution and Project management
- Syntax parsing and analysis
- Semantic code analysis
  - Data Flow
  - Control Flow
- Code generation
- Syntax tree manipulation

Compiler basics - What is code?

Code is text that is formatted in a specific way, with defined syntax constructs to instruct the machine (via the compiler) to perform the tasks the programmer desires, and is usually read and executed exactly as English: from top to bottom and left to right. This all conforms to the usual expectations of us programmers, except it's far, far away from how compilers understand code.

Compiler basics - Code Sample

```
x=1
y=2
3*(x+y)
```

Source for this example:
`https://stackoverflow.com/a/9864571`

Compiler basics - Parse Tree

`https://i.stack.imgur.com/SyonV.png`

Compilers do read code once as we programmers do, but they quickly build a wholly different model from the text representing the original text as a tree with the outer leaves being simple lexemes (sequences of characters that make sense in the syntax of the language) and nodes with higher complexity semantics as you close in on the root. This construction primarily obeys performance and simplicity reasons, as raw text manipulation and re-interpretation are expensive and complex operations when repeated enough.

Compiler basics - Abstract Syntax Tree

https://i.stack.imgur.com/dhd3v.png

Further down the pipeline compilers take the parse tree and discard semantically meaningless information like whitespace characters to build the ultimate representation of code: The Abstract Syntax Tree. This tree contains the essential information (and little more) that is required to build further models and compilation.

Roslyn gives us the capability to inject code to reason about and manipulate the AST.

Roslyn Code Model

Roslyn has always had two extremely important non-functional requirement: Speed and Memory Efficiency. As the main use case for Roslyn is reasoning about code that is being written live to serve as the foundation for IDE tooling (With Visual Studio at the forefront, but not the only citizen). Failing to comply with these tenets would lead to very unpleasant developer experiences.

It stands to reason then to implement the basic code representation (AST) model with a focus on re-usability and immutability to make multi-threading easier, It is very important to have this in mind, as it will affect the API and programming model significantly.

Roslyn Basics - Code Diagnostics

One of the entry-points of the Roslyn API for code modification is the Code Diagnostics API. This API exposes a mechanism to integrate into the usual code analysis flows that IDEs and other tooling ask of Roslyn in order to augment them with additional custom warnings, recommendations and refactorings that are reflected in the IDE.

DEMO