GryphonSharp v1

Anton Sukachev, a.sukachev@lancaster.ac.uk
School of Computing and Communications
Lancaster University

Coordinator: Abe Karnik (a.karnik@lancaster.ac.uk)
Research group: Human Computer Interaction

Contents

Abstract

Write up abstract

1 Introduction

Since the first computers, there has existed a strong need for a high-level programming language that would allow users to quickly and efficiently code applications and systems that they need for their businesses, personal projects or even games.
This research has a few distinctive objectives it attempts to achieve:

- A set of questions in the section Motivation.
- Proposes a design for creating simple programs through G# language.
- Presents implementation of G# VPE.

// Aims of this research

2 Background

Unfortunately, due to the scope of research and novelty of the topic, there is no way to easily pinpoint exact beginning of the programming in general. For this reason, the research will begin to explore programming since the appearance of first languages.

2.1 First abstract language

The first ever abstract language appeared around 1969-1973 and it rapidly replaced assembly language which offered 'one-at-a-time' instruction in a semi-readable for human format. The C Programming Language quickly became a legend among programmers, especially those working on a very first UNIX operating system[2]. The C programming language introduced a wide range of abstract paradigms that programmers could utilize to swiftly develop readable programs where they can orient freely and extend to their needs. Some of the examples of such paradigms include: struct, enum and array indexing with the following syntax:

```
variable[2]
other_variable[0]
```

Listing 1: C array indexing

Programs that would otherwise require hours of revision and coding could be written in a matter of minutes, if not seconds. After C language, a major abstract language at a time, was standardized by ANSI, it became the primary language for most computing systems.

Despite being a standard for most, if not all, computing systems, the language was platform-specific, meaning different platforms would have specific implementations, hence compiled executables could only be run on the platforms for which executables were explicitly compiled. That way executables compiled for Windows would not run on MacOS/Linux Unix systems.
Unlike most, C language offers naked address referencing, meaning any given number could be converted into address in memory and backwards. This low level access to actual computer memory is strongly tied to memory leaks, critical security leaks, and even unwanted overflows that, if left unchecked, could violate kernel space from user space, which in turn could trigger kernel panic.[1] Not only extremely low-level access that the C language provides poses severe vulnerability risk to the entire operating system, but it also requires significantly advanced memory management systems written by the applications' developers. In contrast, languages such as Java or C# have built-in memory management that automatically frees memory when instances no longer accessible. For these reasons C language simply did not offer enough abstraction, security, and flexibility to aid development of complex programs.

2.2 First Visual Programming Environments

Visual Programming Environments (henceforth VPE) was introduced for educational purposes first appearing in 2003 for Intel's Computer Clubhouses where young could learn computer technologies. Original designs of scratch were solely intended for young by adding visual programmable elements that manipulate media on screen. Scratch UI uses sturdy design principles that achieve it's ultimate goal i.e. Single Window with multiple tabs ensures there is always visible primary components such as scene or code.

2.3 Minecraft and Redstone

Why Minecraft game that has nothing to do with teaching and programming (at the very least initially) has a communities which focus heavily on implementing complex computing systems (even entire CPU architectures in Minecraft using nothing else but plain single-thread Redstone)

2.4 Transpilers, an upgrade to existing languages

Transpilers take source code and compile into other source code. This is 'transpilation'.

2.5 Blockly and general-purpose VPE

Blockly was a good take on general-purpose VPE. It uses transpiler principles to interpret visual representation of a code. It support wide range of target languages, however, currently, without specifically designed environment, Blockly, out-of-the-box, only supports few paradigms: functions, variables, arithmetic

operations, flow control (loops, if statements) and a few more basic primitives like lists and color.[1] Blockly, natively, is only a set of theories and findings that build on top of preexisting VPE 'Scratch'. Despite it having rich and extensible documentation, Blockly is just a set of tools for creating VPEs.

3 Motivation

How C achieved it's purpose? Why has it became so popular?
How C represented basic programming constructs (for, while loops, if statements)? When and why it fails?
Why Scratch became so popular? How Scratch aims to achieve their aims?
Scratch is limited by Scratch VM.
Transpilers are interesting take on simplifying development. Any attempts to generate code from Visual Programming Environments - refer to Blockly.

4 System Design

Previously mentioned transpilers were 'the next step' towards higher languages that might provide task-specific abstraction level and paradigms for easy development of said tasks. For example, many game studios that started working in limited language such as Lua had to either rewrite their entire game on a different engine under different programming language. However, since introduction of transpilers that use higher-level or simpler languages to transpile to low-lever or complex languages, one of the most popular examples (in addition to mentioned Typescript-Javascript transpiler) being CSharp.lua[2] which takes c# code and transpilers it into Lua code retaining most, if not all original language features such as reflection, polymorphism, encapsulation etc.
In this project, it was decided to utilize offline transpiler. This decision is motivated by strong presence of high-level heavily optimized languages such as Java, Javascript, C#, Python, C++ and others. In contrast, the other choices were: Compiler and Interpreter. Interpreter is an algorithm which uses calls and translates them one by one into bytecode, all of this 'interpretation' is happening at runtime and therefore creates huge overhead because every instruction the interpreter executes, essentially has to be looked up before it can be executed. Look up of interpreters is *usually* capped at RAM latency, rather than CPU cache latency, meaning look up of instructions happens in RAM in worst case scenario. Compiler both JIT and AOT require significant investment of time to design and implement from scratch. For purposes of this research, creating custom G# to machine code compiler would be infeasible and would simply go outside of this project's scope.

[1] https://developers.google.com/blockly
[2] https://github.com/yanghuan/CSharp.lua

5 Language Design

Design will be split into multiple sections, this is because over the duration of the project this has changed a lot and evolved over the duration of the research addressing one issue at a time, but starting from scratch every time.

5.1 Starcraft II inspired Behaviour Oriented Design

5.2 Node-like Shader Editor Design

5.3 Unreal Engine Blueprints inspired Design

5.4 Stateless Node Editor

6 Implementation

//Small section on how it is achieved, including cross-platform.

6.1 Intermediate GryphonSharp Representation

Intermediate G# Representation (henceforth G# IR) is a JSON-structured file containing code, data and metadata for both Transpiler and Node Editor.

6.2 Incomplete Components

Following subsections will include incomplete or unfinished components of this project. Generally, just like in the proposal of the GryphonSharp, the 3 key components this research was trying to achieve were: Language Server (GryphonSharp-Overwatch), G#-to-C# Transpiler (GryphonSharp-Transpiler), Visual Node Editor (GryphonSharp-vscode). Every component is detailed and documented in the following subsections.

6.2.1 Language Server

Originally, Language Server, codenamed 'Overwatch', was planned to be implemented with VSCode language server protocol.[3]

6.2.2 G#-to-C# Transpiler

6.2.3 Visual Node Editor

7 Example Application

//Mock up examples

[3] https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

8 Discussion

//Why this is cool

9 Appendix

References

[1] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *1984 IEEE Symposium on Security and Privacy*, pages 2–2, 1984.

[2] Dennis Ritchie. The development of the c language.