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 address:

- 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[**?**]. 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.[**?**] 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

Minecraft is a computer game that was introduced in 2009. Minecraft opened a new genre - Sandbox. Sanbox genre, effectively, allowed players to manipulate their environaments and shape them visually how they see fit. Game introduced first few 'quests' that introduced player to the game, which then left player to explore world on their own. Minecraft is extremely successful, but not for a straightforward 'because of this and that' reasons. [**?**] Why Minecraft game that has nothing to do with teaching and programming[**?**] (at the very least initially, it was designed for self-entertainment by creating stories using creative tools that the game provided[**?**]) 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.
// write up

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

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

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.

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

[**?**] // write up

5.2 Node-like Shader Editor Design

[**?** **?**] // write up

5.3 Unreal Engine Blueprints inspired Design

// write up

5.4 Stateless Node Editor

[**?**] // write up

6 Implementation

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. Node Editor metadata:

- `bgSizes`
  This instructs Node Editor how zoomed in the code map currently is.
- `stagePos`
  This tells Node Editor the last position the code map.
- `nodeCount`
  This is used for node id-ing, in other words, this value is assigned to new nodes on creation then incremented by 1.

- `dataCount`
  This is the same as nodeCount but used for id-ing data, again, when data is first introduced, it will be assigned this value, then the value will be incremented by 1.

When this metadata is parsed into the Node Editor it will load 'preferences' from the last save of the document. G# IR also has transformer metadata, however, it is used for both Node Editor and transformer. This metadata defines the type that is being parsed out or in to the function. This is later used for transpiler-side validation in case the G# source was modified by hand.

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] The Overwatch language server handles multiple tasks listed below.

- **Type-Safety Enforcer**
  When user tries to connect nodes in the node editor, types of connectors are evaluated through Overwatch server request. Overwatch, being connected to Omnisharp server, will evaluate validity of type parsing and return boolean with TRUE value if parsing type is 'legal' for the connector, otherwise it will return FALSE, which will instruct node editor to highlight connection is 'illegal'. Highlight can either allow connection with red-error cross across illegal connection (can be seen in figure 1) or prevent connection altogether and raise an error through message box (can be seen in figure 2).

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

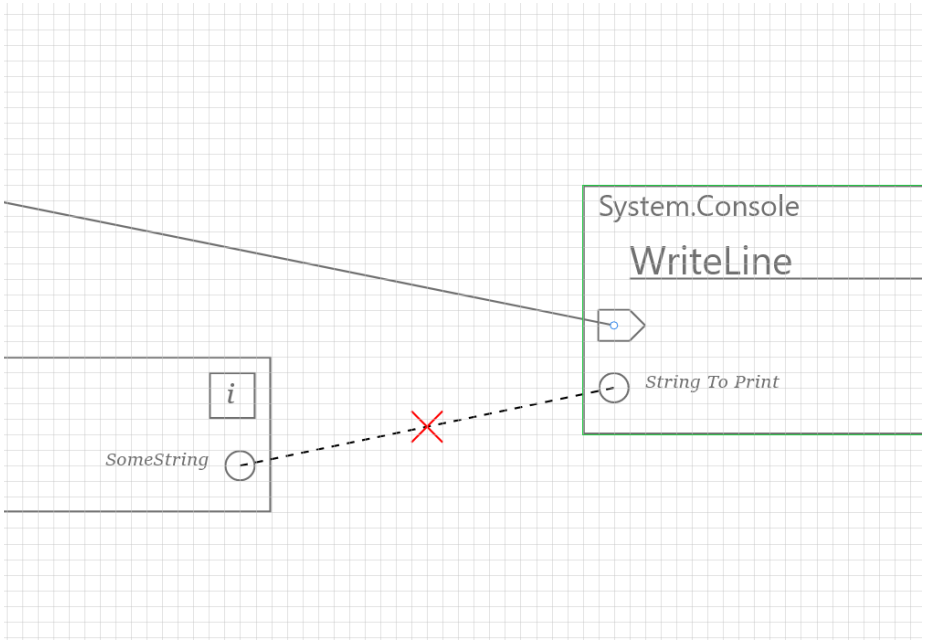

Figure 1: Illegal Connection

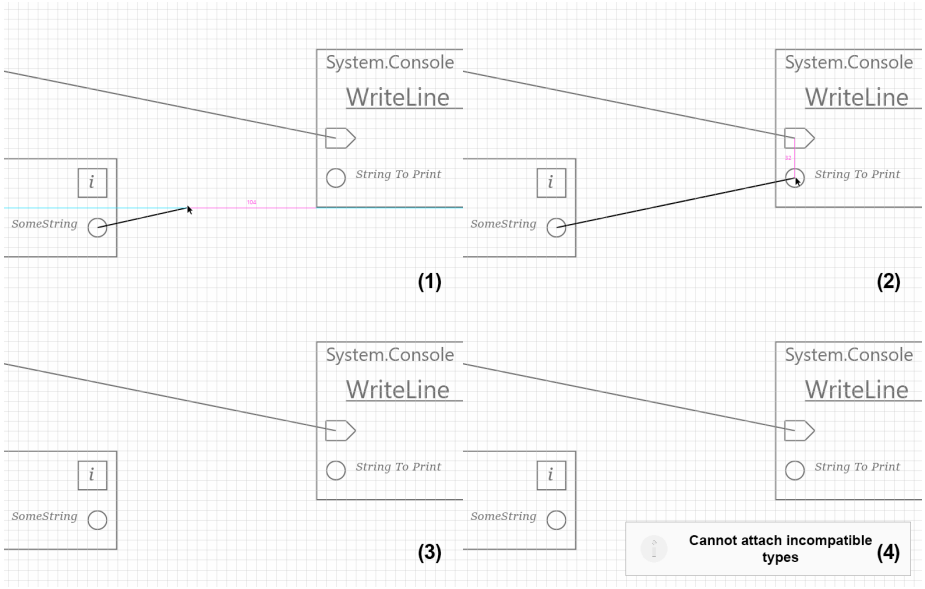

Figure 2: Illegal Connection

- **Type Discovery**
  When user attempts to parse a type into a empty space in the node editor, this will prompt user with function suggestions that match the parsed type.

- **Logic Evaluation**
  When user constructs loops without exist clauses, such as while(true) loops or implement strange if statements, such as if(false), this will do a basic evaluation of the logic through existing tools in Omnisharp server.

- **Autocompletion**
  When user types in a name of a function, variable, class or any other language construct, this will ask Omnisharp for Intellisense suggestions, which are then parsed into Node Editor for display.

- **Source Watcher**
  When user makes changes to any of the files, Node Editor will notify VSCode of any workbench changes. These workbench changes are also parsed into language server. When the changes are made to source files or project files, language server will invoke G#-to-C# transpiler task.

6.2.2 G#-to-C# Transpiler

This is the actual transpiler that will convert G# IR into C# source code by the means of CodeDOM Microsoft package. CodeDOM allows construction of source code in readable format and supports all features of C# language that are available to developers. CodeDOM is a powerful execution graph builder that supports C# and IronPython transpilation. However, it can be defined, through a transformer factory, to translate into any other language according to the language's rules.

6.2.3 Visual Node Editor

Node Editor (in code is referred to as NE) is the primary focus of this research. Visual design of the editor is explained in-depth in section Language Design. In this section, however, only the technical side of editor will be discussed. Visual Node editor is an extension for a popular Electron app "Visual Studio Code"[4]. Extension is written in typescript ES2015 and has 2 primary parts within the extension: Editor and Extension (sometimes referred to as simply frontend for Editor and backend for Extension).
Frontend of NE is an html page covered by a Konva canvas. Konvajs canvas is a high-performance canvas for drawing nodes. It has powerful API and has some very promising demos, some of which are: fully functional Google-like spreadsheets, react-konvajs which is used by some of high-end applications such as Polotno.[5] User can manipulate nodes in the editor which will be written into

[4] https://code.visualstudio.com/
[5] https://polotno.dev/

a `*.gs` file upon saving. G# (or `*.gs`) files contain JSON intermediate representation (IR) of G#. This JSON contains a lot of metadata for both NE and transpiler. Furthermore, JSON is plaintext serializable data structure, meaning it can be used seamlessly with code transformation tools, version control tools, and even modified by hand. Frontend of NE is written in Typescript, unlike extension it has slightly different compilation and deployment pipelines. Frontend has to be compiled 'for browser', since, as stated above, VSCode is only a chromium that displays webpage, but has a backend that allows it to interact with operating system. Unfortunately, TypeScript doesn't support compilation for Browser runtime, which means the code will have to be bundled. This projects takes advantage of Rollup bundler, which compiles Typescript code into ES6, but the transforms it through iteration of dependency tree. Source files, along with any referenced NodeJS libraries, bundled into a single `*.js` file that can be instantly executed by NE frontend. The second part of the NE is backend. Backend is written in Typescript as well and can be natively run by NodeJS runtime. Due to limitation and old technology used in VSCode, backend code is still compiled into deprecated CommonJS code. Extension code is then injected into a developer instance of VSCode. For release versions, code is bundled and executed as a module when added.

7 Discussion

This project was heavily time-constrained and therefore was left in a work-in-progress state.

8 Appendix

References

[1] Youngkyun Baek, Ellen Min, and Seongchul Yun. Mining educational implications of minecraft. *Computers in the Schools*, 37:1–16, 01 2020.

[2] Grady Booch. From minecraft to minds. *IEEE Software*, 30:11–13, 03 2013.

[3] Jean Bresson and Carlos Agon. Musical representation of sound in computer-aided composition: A visual programming framework. *Journal of New Music Research*, 36:251–266, 12 2007.

[4] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation, 11 1998.

[5] Enrique Coronado, Fulvio Mastrogiovanni, Bipin Indurkhya, and Gentiane Venture. Visual programming environments for end-user development of intelligent and social robots, a systematic review. *Journal of Computer Languages*, 58:100970, 06 2020.

[6] Mustafa Can DEMİRKIRAN and Fatma TANSU HOCANIN. An investigation on primary school students' dispositions towards programming with game-based learning. *Education and Information Technologies*, 02 2021.

[7] Neil Fraser. Ten things we've learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50, 2015.

[8] Swen Gaudl, Simon Davies, and Joanna J. Bryson. Behaviour oriented design for real-time-strategy games: An approach on iterative development for starcraft ai. In *Foundations of Digital Games Conference 2013 (FDG 2013)*, pages 198–205. Foundations of Digital Games, May 2013. Foundations of Digital Games 2013 ; Conference date: 14-05-2013 Through 17-05-2013.

[9] V. V. Gribova and A. S. Kleschev. Ontology paradigm of programming. *Automatic Documentation and Mathematical Linguistics*, 47:180–187, 09 2013.

[10] 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.

[11] JAMES D. KIPER, ELIZABETH HOWARD, and CHUCK AMES. Criteria for evaluation of visual programming languages. *Journal of Visual Languages Computing*, 8:175–192, 04 1997.

[12] Addie Matteson. A diy computer, plus minecraft. *School Library Journal*, 63(10):20, 2017.

[13] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 21–24, 2017.

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

[15] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2:247–261, 09 1977.

[16] Kirsten N. Whitley, Laura R. Novick, and Doug Fisher. Evidence in favor of visual representation for the dataflow paradigm: An experiment testing labview's comprehensibility. *International Journal of Human-Computer Studies*, 64:281–303, 04 2006.