Automated Transpiler Generation

Structure:

This paper is structured into 6 chapters, as follows:

Chapter 1: Introduction, this chapter serves to describe the motivation behind the paper.

Chapter 2: Metaprogramming, is intended in order to summarily describe the field of Metaprogramming in order to give better context to this paper.

Note: explain C++ template metaprogramming and other concepts

Note: What is metaprogramming

Chapter 3: User Guide is used to shape the design and the capabilities of the presented system, with the purpose of explaining the exposed features of the application, giving the reader the necessary information to practically integrate the application.

Chapter 4: Implementation details contains in-depth information of how the features are implemented, and the motivation behind their implementation.

Chapter 5: The final chapter outlines the limitations of the current system and the extension points for further development.

Abstract

1. Introduction
   1. Motivation and Purpose
   2. Structure
   3. Personal Contributions
2. Metaprogramming
   1. Code Generation
   2. Compiler Generation
3. User Guide
   1. Concepts
   2. Features
   3. Usage
4. Implementation Details
   1. Technologies
   2. Structure
   3. Feature implementations
5. Conclusions, Future Work

Research references:

1. Metacompiler: (computer science) A compiler that is used chiefly to construct compilers for other programming languages. "Sci-Tech Dictionary McGraw-Hill Dictionary of Scientific and Technical Terms, 6th edition". McGraw-Hill Companies.
2. Metaprogramming in .NET (2012), Kevin Hazzard and Jason Bock - https://www.manning.com/books/metaprogramming-in-dot-net
3. Source-to-Source translation and Software Engineering (2013), David A. Plaisted
4. A compiler generator (2012), McKeeman William M, Horning James J. and Wortman David B.
5. Programming Concepts: Type Introspection and Refelction (2014), Krauss Aaron
6. The challenge of metaprogramming (2016), Bicking Ian
7. The Definitive ANTLR 4 Reference (2007), Terence Parr
8. The Definitive ANTLR 4 Reference (2013), Terence Parr
9. C# 9 and .NET 5 – Modern Cross-Platform Development: Build intelligent apps, websites and services with Blazor, ASP.NET Core and Entity Framework Core using Visual Studio Code, 5th edition (2020), Mark J. Price
10. Roslyn Cookbook: Compiler as a Service, Code Analysis, Code Quality and more (2017), Manish Vasani
11. Eloquent Javascript (2011), Marjin Haverbeke
12. Vue.js in Action (2018), Erik Hanchett
13. RESTful.NET (2008), Jon Flander
14. Language Workbenches: The Killer-App for Domain Specific Languages? (2005), Martin Fowler
15. The MPS Language Workbench: Volume I (2014), Fabien Campagne

Other References:

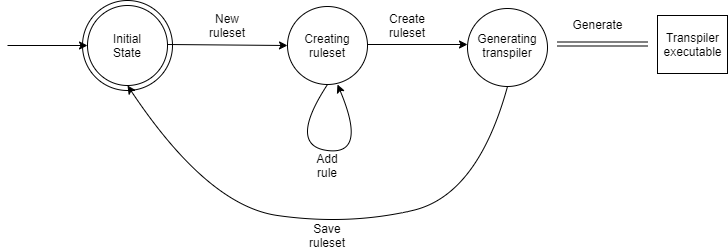
1. <https://en.wikipedia.org/wiki/Compiler-compiler>
2. Metaprogramming in .NET (2012), Kevin Hazzard and Jason Bock - https://www.manning.com/books/metaprogramming-in-dot-net
3. https://levelup.gitconnected.com/four-ways-to-generate-code-in-c-including-source-generators-in-net-5-9e6817db425
4. <https://dotnet.microsoft.com/apps/aspnet/apis>
5. https://github.com/antlr/antlr4/blob/master/doc/index.md
6. <https://fullboarllc.com/using-antlr-4-with-net-core-2-1-and-c-getting-started/>
7. <https://www.willowtreeapps.com/craft/an-introduction-to-language-lexing-and-parsing-with-antlr>
8. <https://channel9.msdn.com/Events/dotnetConf/2020/Create-a-Text-Parser-in-C-with-ANTLR>
9. Source-to-Source Translation and Software Engineering by David A. Plaisted - <https://www.scirp.org/html/5-9301620_30425.htm>
10. SML2Java: A Source to Source Translator - <http://www.litech.org/~vaughan/pdf/dpcool2003.pdf>
11. <https://www.geeksforgeeks.org/source-to-source-compiler/>
12. <https://humbletoolsmith.com/2018/04/05/t4-csharp_s-little-known-code-generator/>
13. Understanding Typescript - <https://www.semanticscholar.org/paper/Understanding-TypeScript-Bierman-Abadi/1469b0cbb109c2a788a346dd0480070de8334dea>
14. Universal-transpiler - <https://jarble.github.io/transpiler/>
15. JetBrains MPS - <https://en.wikipedia.org/wiki/JetBrains_MPS>

* Application lifecycle

The application is composed of two parts:

* A REST API exposing an endpoint for generating a transpiler, referred to as Generator
* A web application which consumes the REST API and exposes an interactive user interface for creating a transpiler, referred to as Frontend
* A runnable application generated on-demand by the Generator which can then be ran from the command line to transpiler an input to an output, referred to as Executable

The life cycle of the applications:

* The Generator is intended to be a functional API, it is stateless, and can be used independently of the Frontend. The Generator has only two easily perceivable states as a web application:
  + Disabled
  + Running
* The Frontend is a web app, partly used to create rulesets used by the Generator, and partly to use those rulesets to consume the Generator. The diagram below depicts the flow of the frontend application. It is stateful, storing sequentially added rules as a ‘ruleset’ used afterwards.
* The executable is designed to be a functional, stateless API, only receiving an input text and producing an output text.
* Feature list
* Generator:
  + REST API endpoint, with input a ruleset and outputting an executable transpiler as a file
  + Consistency check for input Ruleset
  + Error reporting in case of inconsistencies/incompatibility
* Frontend:
  + Web app accessible from the browser
  + Presentation page outlining the purpose of the site
  + Rulesets UX
    - Ruleset entity CRUD
    - Rule subentity CRUD
    - Persistence for rulesets
  + Transpiler Generation
    - Consuming of Generator REST API
    - Web app caching and automatic use of Transpiler for in-browser transpilation
* Executable:
  + Command line execution
  + Functionality to use files for input/output
  + Input test parsing error reporting
* Original contribution

The presented application offers an original contribution through its main feature:

Creating on-demand, customizable text transpilers that are easy to use and integrate with other tools.

This is unique due to the fact that current ‘compiler generation’ frameworks/systems have several limitations, ranging from:

* Lack of customizability
* Can only be used in a certain, constraining context, such as only being usable through the presence of the main codebase of the tool
* Are not designed for integrating with existing programming language development flows
* Very restricting use case, usually by restricting the pool of source/output input
* Cannot be generated on-demand, rather only statically
* Require grasp of advanced concepts to be used efficiently

This application aims to solve or bypass all those limitations by providing a simple to use, simple to use in practice API for on-demand text transpilation.

Lab 5 – 8

1. Technologies

* Generator
* **C#:** The backend of the application, the API, the transpiler generator is written using C#, which is a modern, object-oriented and type-safe high-level programming language created by Microsoft. I have chosen it due to the plethora of features that make robust application development faster than other languages. I have chosen it in detriment to other similar languages like Java and Kotlin due to the .NET ecosystem containing very good support for metaprogramming and dynamic execution of outside tools.
* **.NET:** is a development platform and a collection of tools used for developing applications using C#, F# or VB.NET. Additionally, it is surrounded by a library ecosystem powered by NuGet. I have chosen .NET 5 for realizing the application as it is the latest stable version, provides the best support for Web Applications and it contains officially supported packages for metaprogramming, such as the officially supported Roslyn compiler and its complementary toolset for dynamically creating applications, which is a tool lacking equivalent for Java/Kotlin’s MVN repository ecosystem.
* **ANTLR4:** Antlr4 (Another Tool for Language Recognition) written in java is a powerful parser generator for reading, processing executing or translating structured text. It is widely used for building languages, tools and frameworks due to automating the parsing process. Antlr4 is the most widely used parser generator and as such has the best support. Alternatives include Antlr3, JavaCC, bison, however they are either outdated, lack support or do not target C#
* Frontend
* **Javascript, Vue.js:** The frontend is built using Javascript and the Vue.js framework. Javascript is a modern high-level dynamically typed programming language designed and optimized for web development. Vue.js is a framework written in Javascript entitled as the most progressive javascript framework. It is used to facilitate complex application building using javascript. Alternatives to this stack would be Blazor using WebAssembly, and for Vue.js, alternatives are either React, or Angular. I have chosen Vue as React is designed for stateless, functional UIs and Angular is a very bulky framework with worse performance. Vue combines both good performance and support for creating inbrowser stateful user experiences.
* Executable
* **C# & .NET 5:** The executable application which represents a transpiler is a .NET Core Console application written using C# and targeting .NET 5. This choice was made in order to make the transpiler applications run cross-platform, with the only major dependency being the .NET 5 sdk, as well as being able to be dynamically constructed using the Roslyn compiler by the Generator.

1. Feature subset - Reproductibility conditions
   1. Feature subset
      * Transpiler Generation

The core feature of the Generator, the REST API endpoint, receiving a ruleset based on which it generates an executable transpiler is implemented by the application: <https://github.com/Horatzio/LangBuilder/blob/main/LangBuilder/Source/Controllers/GenerateApiController.cs>.

The way this feature works is through three individual steps:

* Generating the Antlr4 grammar – using the input ruleset which is checked for consistency and Razor, a .g4 grammar file is compiled and saved using the Antlr4 comprehensible syntax for grammar files
* Generate Antlr files – Once the grammar has been composed, the Antlr4 executable is run in order to generate the parser based on the grammar file. These files are created within the ‘Transpiler’ project inside the LangBuilder solution.
* Generate executable – The syntax of the Antlr generated files is loaded by the Roslyn compiler. The dynamically generated code necessary for effectuating the transpilation is inserted into the syntax, and an executable is Dynamically Generated.
  1. Reproductibility conditions
     + Dockerization

The Generator and the Frontend are dockerized and as such designed to run within a container which can be run on any hardware to get the application running.

* + - Unit testing

Unit testing is used to test the main feature of the Generator and confirm it is operating as expected.

1. Comparison with similar applications – Deployment plan
   1. Comparison

A similar, fully developed application is Jetbrain’s MPS <https://en.wikipedia.org/wiki/JetBrains_MPS>

* 1. Deployment plan

The automated transpiler generator is implemented as a web application, and as such the goal is to expose it as a publicly available website. The intent is to expose both the frontend application as well as the REST API to be consumed.

This is to be achieved by deploying the complete application into the cloud, onto the platform of a cloud provider, such as Google Cloud, AWS or Azure. Once deployed to the cloud provider the application is intended to run constantly.

As the application is dockerized and lacks state persistence outside the browser, it will be deployed into two containers, one for the frontend and one for the Api. Container services are readily available through all cloud providers, e.g. Azure Web App for Containers, Amazon Elastic Container Instance. Additionally, the API can be run independently of the frontend.

Furthermore, a domain name shall be purchased to publicly expose the web application and secure it using SSL. An example domain name: langbuilder.com. The frontend will be exposed at an usual subdomain, e.g. [www.langbuilder.com](http://www.langbuilder.com) while the API will be exposed at another, such as api.langbuilder.com. Once the DNS change propagates and the application is confirmed to be up and running uneventfully, it will be publicly exposed for users to either consume the API or use the UI to generate their transpilers.

The transpiler executables will be exposed as a zip package containing all their dependencies, except the main dependency – the .NET 5 SDK. Once unpacked, it can be run locally by clients.

1. Soure code, History

Github repository: <https://github.com/Horatzio/LangBuilder>

Commits: <https://github.com/Horatzio/LangBuilder/commits/net-core>