GOOL: A Generic Object-Oriented Language

Anonymous Author(s)

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Given a task, before writing any code a programmer must select a programming language to use. Whatever they may base their choice upon, almost any programming language will work. While a program may be more difficult to express in one language over another, it should at least be possible to write the program in either language. Just as the same sentence can be translated to any spoken language, the same program can be written in any programming language. Though they will accomplish the same tasks semantically, the expressions of a program in different programming languages can appear substantially different due to the unique syntax of each language. Within a single programming language paradigm, such as object-oriented (OO) programming, these differences should not be so extreme. OO programs, no matter the language, share certain structural properties. They are built from variables, methods, classes, and objects. Some OO languages even have very similar syntax. But however similar they may be, no two programming languages are identical.

If a programmer wishes to write a program that will integrate into existing systems written in different languages, they will likely need to write a different version of the program for each. This requires investing the time to learn the idiosyncrasies of each language and give attention to the operational details where languages differ. Ultimately, the code they write will likely be marred by influences of the language they know best. They may consistently use techniques that they are familiar with from one language, while unaware that the language in which they are currently writing offers a better or cleaner way of doing the same task [4, 13]. Besides this likelihood of writing sub-optimal code, repeatedly writing the same program in different languages is entirely inefficient. Languages in the same paradigm have many similarities, and there is an excellent opportunity to take advantage of these similarities to improve the efficiency of writing code. If a program could be written in one language and automatically translated to any other language in the same paradigm, this would greatly facilitate program reuse. Directly translating between existing OO languages will not always be possible because some languages require more information than others. A dynamically typed language like Python, for instance, cannot be straightforwardly translated to a statically typed language like Java, because additional type information would need to be provided. But if there was a language that contained all of the information that any of the other OO languages would need, it could be used as the source language for translation. This source language should also be completely language-agnostic, free of any of the idiosyncratic "noise" required by specific languages. In addition, a translator for such a language would not be subject to the influences of any one target language when translating to a different target language; the code resulting from the translation should be well-suited to the target language.

The similarities between OO programs do not end with syntax and structural components. Additionally, there are tasks and patterns commonly performed by OO programs in any language, from simple tasks like splitting a string or patterns like defining functions on inputs to produce outputs, to higher-level design patterns like those described in [7]. A language that provided abstractions for these tasks and patterns would make the process of writing OO code even easier.

A Domain-Specific Language (DSL) is a high-level programming language with syntax tailored to a specific domain [11]. DSLs allow domain experts to write code without having to concern themselves with the syntactical and operational requirements of general-purpose programming languages. A DSL abstracts over the details of the code, providing notation for a user to specify domain-specific knowledge in a natural manner. DSL code is typically compiled to a more traditional target language. Abstracting over code details and compiling into traditional OO languages is exactly what we want our source OO language to do. The code details to abstract over in this case include both the operational details of using a specific language as well as the higher-level patterns that commonly show up in OO programs. So the source language we are looking for is just a DSL in the domain of OO programming languages!

There are DSLs already for generating code in multiple languages, and these will be discussed further in Section 5, but none of these have the combination of features we require. We should be able to generate OO code for any purpose, rather than being limited to a narrow domain of application. The generated code should be human-readable, so that it can be used in applications where understandability of the code is important. The DSL should generate idiomatic code. That is, code generated in each target language should be expressed naturally given the features and capabilities of

PL'18, January 01–03, 2018, New York, NY, USA
2018.

that language. Finally, the DSL should provide facilities for generating code adhering to common high-level OO patterns.

We have developed a Generic Object-Oriented Language (GOOL), proving that such a language indeed exists. GOOL is a DSL embedded in Haskell that can currently generate OO code in Python, Java, C#, and C++. Theoretically, any OO language could be added as a target language for GOOL. This paper presents GOOL, starting with the syntax of the language in Section 2. Section 3 describes how GOOL is implemented. GOOL provides some higher-level functions for convenient generation of code following commonly used patterns, examples of which are presented in Section 4. We close with a discussion of related work in Section 5, plans for future improvements in Section 6, and conclusions in Section 7.

2 GOOL Syntax

3 GOOL Implementation

4 Higher-level GOOL functions

5 Related Work

Haxe is a general-purpose multi-paradigm language and cross-platform compiler. It compiles to all of the languages GOOL does, in addition to many others. However, it does not offer the high-level abstractions GOOL provides [3] (better reference?). Also, the generated source code is not very readable as Haxe generates a lot of "noise" and strips comments from the original Haxe source code.

Protokit's 2nd version is a DSL and code generator for Java and C++, where the generator is designed to be capable of producing general-purpose imperative or object-oriented code. The Protokit generator is model-driven and uses a final "output model" from which actual code can be trivially generated. Since the "output model" was so similar to the generated code, it presented challenges with regards to semantic, conventional, and library-related differences between the target language [9]. GOOL's finally-tagless approach and syntax for high-level tasks, on the other hand, helped it overcome differences between target languages.

ThingML [8] is a DSL for model-driven engineering targeting C, C++, Java, and JavaScript. While it can be used in a broad range of application domains, they all fall under the umbrella domain of distributed reactive systems, and so it is not quite a general-purpose DSL, unlike GOOL. ThingML's modelling-related syntax and abstractions are a contrast to GOOL's object-oriented syntax and abstractions. The generated code lacks some of the pretty-printing provided by GOOL, specifically indentation, which detracts from the readability.

IBM developed a DSL for automatic generation of OO code based on design patterns [5]. Their DSL was in the form of a visual user interface rather than a programming language, and could only generate code that followed a design pattern. It could not generate any general-purpose code.

There are many examples of DSLs with multiple OO target languages but for a more restricted domain. Google protocol buffers is a DSL for serializing structured data, which can then be compiled into Java, Python, Objective C, and C++ [2]. Thrift is a Facebook-developed tool for generating code in multiple languages and even multiple paradigms based on language-neutral descriptions of data types and interfaces [14]. Clearwater is an approach for implementing DSLs with multiple target languages for components of distributed systems [15]. The Time Weaver tool uses a multi-language code generator to generate "glue" code for real-time embedded systems [6]. The domain of mobile applications is host to a bevy of DSLs with multiple target languages, of which MobDSL [10] and XIS-Mobile [12] are two examples. Conjure is a DSL for generating APIs. It reads YML descriptions of APIs and can generate code in Java, TypeScript, Python, and Rust [1]. All of these are examples of multi-language code generation, but none of them generate general-purpose code like GOOL does.

6 Future Work

7 Conclusion

References

[1] [n. d.]. Conjure: a code-generator for multi-language HTTP/JSON clients and servers. https://palantir.github.io/conjure/#/ Accessed 2019-09-16.

[2] [n. d.]. Google Protocol Buffers. https://developers.google.com/protocol-buffers/ Accessed 2019-09-16.

[3] [n. d.]. Haxe - The cross-platform toolkit. https://haxe.org Accessed 2019-09-13.

[4] Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2012. The effect of previous programming experience on the learning of scenario-based programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. ACM, 151–159.

[5] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. 1996. Automatic code generation from design patterns. *IBM systems Journal* 35, 2 (1996), 151–171.

[6] Dionisio de Niz and Raj Rajkumar. 2004. Glue code generation: Closing the loophole in model-based development. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004). Workshop on Model-Driven Embedded Systems*. Citeseer.

[7] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[8] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. Thingml: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 125–135.

[9] Gábor Kövesdán and László Lengyel. 2017. Multi-Platform Code Generation Supported by Domain-Specific Modeling. *International Journal of Information Technology and Computer Science* 9, 12 (2017), 11–18.

[10] Dean Kramer, Tony Clark, and Samia Oussena. 2010. MobDSL: A Domain Specific Language for multiple mobile platform deployment. In *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*. IEEE, 1–7.

[11] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[12] André Ribeiro and Alberto Rodrigues da Silva. 2014. Xis-mobile: A dsl for mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1316–1323.

[13] Jean Scholtz and Susan Wiedenbeck. 1990. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction* 2, 1 (1990), 51–72.

[14] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8 (2007).

[15] Galen S Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, and Koichi Moriyama. 2005. Clearwater: extensible, flexible, modular code generation. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 144–153.

A Appendix

Text of appendix ...