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 [1, 4]. 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 [2]. 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 [3]. 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

6 Future Work

7 Conclusion

References

[1] 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 on the 12th Koli Calling International Conference on Computing Education Research*. Association for Computing Machinery, Koli, Finland, 151–159.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, Boston, MA, USA.

[3] M. Mernik, J. Heering, and A. M. Sloane. 2003. *When and how to develop domain-specific languages*. Technical Report SEN-E0309. CWI, Amsterdam.

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

A Appendix

Text of appendix …