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, the expressions of a program in different programming languages can appear substantially different due to the unique syntax of each language. Within a single language paradigm, such as object-oriented (OO), these differences should not be as extreme – at least the global structuring mechanisms and the local idioms will be shared. Mainstream OO languages generally contain (mutable) variables, methods, classes, objects and a core imperative set of primitives. Some OO languages even have very similar syntax (such as Java and C# say).

When faced with the task to write a program meant to fit into multiple existing infrastructure, which might be written in different languages, frequently that entails writing different versions of the program, one for each. While not necessarily difficult, it nevertheless requires investing the time to learn the idiosyncrasies of each language and pay attention to the operational details where languages differ. Ultimately, the code will likely be marred by influences of the language the programmer knows 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, both as an up-front development cost, and even more so for maintenance.

Since languages from the same paradigm share many semantic similarities, it is tempting to try to leverage this; perhaps the program could be written in one language and automatically translated to the others? But a direct translation is often difficult, as different languages require the programmer to provide different levels of information, even to achieve the same taks. For example, a dynamically typed language like Python cannot be straightforwardly translated to a statically typed language like Java, as additional type information generally needs to be provided[1].

What if, instead, there was a single meta-language which was designed to contain the common semantic concepts of a number of OO languages, encoded in such a way that all the necessary information for translation was always present? This source language could be made to be agnostic about what eventual target language was used – free of the idiosyncratic details of any given language. This would be quite the boon for the translator. In fact, we could try to go even further, and attempt to teach the translator about idiomatic patterns of each target language.

Why would this even be possible? There are commonly performed tasks and patterns of OO solutions, from idioms to architecture patterns, as outlined in [7]. A meta-language that provided abstractions for these tasks and patterns would make the process of writing OO code even easier.

But is this even feasible? In some sense, this is already old hat: most modern compilers have a single internal Intermediate Representation (IR) which is used to target multiple processors. Compilers can generate human-readable symbolic assembly code for a large family of CPUs. But this is not quite the same as generating human-readable, idiomatic high-level languages.

There is another area where something like this has been looked at: the production of high-level code from Domain-Specific Languages (DSL). A DSL is a high-level programming language with syntax and semantics tailored to a specific domain [11]. DSLs allow domain experts to write code without having to concern themselves with the details of General-Purpose programming Languages (GPL). A DSL abstracts over the details of the code, providing notation for a user to specify domain-specific knowledge in a natural manner. Such DSL code is typically translated to a GPL for execution. Abstracting over code details and compiling into traditional OO languages is exactly what we want to do! The details to abstract over include both syntactic and operational details of any specific language, but also higher-level idioms in common use. Thus the language we are looking for is just a DSL in the domain of OO programming languages!

There are some DSLs that already generate code in multiple languages, to be further discussed in Section 6, but none of them have the combination of features we require. We are indeed trying to do something odd: writing a "DSL" for what is essentially the domain of OO GPLs. Furthermore, we have additional requirements:

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

[1]Type inference for Python notwithstanding

1. The generated code should be human-readable,
2. The generated code should be idiomatic,
3. The generated code should be documented,
4. The generator should allow one to express common OO patterns.

We have developed a Generic Object-Oriented Language (GOOL), demonstrating that all these requirements can be met. GOOL is a DSL embedded in Haskell that can currently generate code in Python, Java, C#, and C++[2]. Others could be added, with the implementation effort being commensurate to their (semantic) distance to the languages already supported.

First we present the high-level requirements for such an endeavour, in Section 2. To be able to give illustrated examples, we next show the syntax of GOOL in Section 3. The details of the implementations, namely the internal representation and the family of pretty-printers, is in Section 4. Common patterns are illustrated in Section 5. We close with a discussion of related work in Section 6, plans for future improvements in Section 7, and conclusions in Section 8.

2 Requirements

3 GOOL Syntax

GOOL's syntax was designed to be similar to OO languages while also providing useful abstractions. Basic types in GOOL are `bool` for Booleans, `int` for integers, `float` for doubles, `char` for characters, `string` for strings, `infile` for a file in read mode, and `outfile` for a file in write mode. Lists can be specified with `listType`. For example, `listType int` specifies a list of integers. Types of objects are specified using `obj` followed by the class name, so `obj "FooClass"` is the type of an object of a class called "FooClass".

Variables are specified with `var` followed by the variable name and type. For example, `var "ages" (listType int)` defines a variable called "ages" that is a list of integers. GOOL offers a shortcut for defining list-type variables with `listVar`, so the "ages" variable could alternatively be specified by `listVar "ages" int`. Since GOOL is embedded in Haskell, such a variable definition can be assigned as the value of a Haskell function:

```
ages = listVar "ages" int
```

Then, in future code the variable can be referred to simply by `ages`. Other keywords for specifying variables include `extVar` for variables from external libraries, `classVar` for accessing variables from a class, `objVar` for accessing variables from an object, and `self` for referring to an object in its own class definition, equivalent to `self` in Python or `this` in Java. The infix operator `$->` is an alternative to the `objVar` keyword.

[2] and is close to generating Lua and Objective-C, but those backends have fallen into disuse

Unlike in most OO languages, the syntax of GOOL distinguishes a variable from its value. To actually use the value of the `ages` variable defined above, one must write `valueOf ages`. This highlights another goal in developing the syntax of GOOL. We did not want to simply look at the syntax of existing OO languages and develop a parallel syntax for GOOL. Instead, we considered semantics, and aimed to provide different syntax for semantically different tasks. In most languages, writing a variable's name is the syntax both for referring to the variable as a variable and referring to the variable's value, but since these are semantically different tasks, GOOL provides different syntax for each. Having distinguishing syntax for semantically distinguishable tasks enables stricter typing and higher-level syntax that translates to more idiomatic code, which will become apparent in future sections of this paper.

Literal values can be referred to by `litTrue`, `litFalse`, `litInt`, `litFloat`, `litChar`, and `litString`. Similar to those seen in most programming languages, GOOL provides many unary prefix operators and binary infix operators for defining expressions. In GOOL, each operator is prefixed with an additional symbol based on type. Operators that return Booleans are prefixed by a ?, for example `?!` is used for negation, ?&& for conjunction, ?|| for disjunction, and ?== for equality. Operators on numeric values are prefixed by #, such as `#~` for negation, `#/^` for square root, #| for absolute value, #% for modulus, and `#^` for exponentiation. Any other operators are prefixed by $, such as the previously mentioned `$->` operator for accessing a variable from an object.

A conditional value can be specified with the keyword `inlineIf` followed by the condition, the value if the condition is true, and the value if the condition is false. For example, given a Boolean-type variable `a`, the following conditional value can be constructed:

```
inlineIf (valueOf a)
  (litString "a is true")
  (litString "a is false")
```

Note that since this is really just Haskell, values can extend across multiple lines as long as subsequent lines are indented.

Function application can be done with the keyword `funcApp` followed by the function name, return type, and values to pass as parameters. Assuming one has defined a function "add" for adding two integers, it could be called like so:

```
funcApp "add" int [litInt 5, litInt 4]
```

Other keywords for function application are `extFuncApp` for when the function comes from an external library, `newObj` and `extNewObj` for calling an object constructor, and `objMethodCall` for calling a method on an object. `selfFuncApp` and `objMethodCallNoParams` are two shortcuts for the common cases when a method is being called on `self` or when the method takes no parameters.

If Haskell function `foo` has been defined as GOOL variable `var "foo" int`, a GOOL statement declaring the variable would be `varDec foo`. To declare and define the variable at the same time, `varDecDef foo (litInt 5)` can be used. Or to define the variable when it has already been declared, `assign foo (litInt 5)` can be used. Other GOOL keywords for declarations and definitions are `constDecDef` for declaring and defining constants, `listDec` and `listDecDef` for declaring and defining lists, and `objDecDef` for declaring and defining objects. `objDecNew` is a shortcut for the common case where an object constructor is called to define an object, and `objDecNewNoParams` is a further shortcut for when the constructor takes no parameters.

Infix and suffix operators for assignments, as seen in most programming languages, are also offered by GOOL. These are all prefixed with &. Instead of `assign a (litInt 5)`, an assignment can be written as `a &= litInt 5`, which looks more like traditional programming languages. The other assignment operators are `&+=` for adding a value to a variable, `&++` for adding 1 to a variable, `&-=` for subtracting a value from a variable, and `&~-` for subtracting 1 from a variable (Unfortunately, the more intuitive `&--` could not be used because `--` initiates a comment in Haskell).

Other simple statements in GOOL include `break` and `continue`, `returnState` followed by a value to return, `throw` followed by an error message to throw, `free` followed by a variable to free from memory, and `comment` followed by a string to be displayed as a single-line comment.

GOOL statements can be grouped together into blocks by specifying a list of statements after the `block` keyword. A GOOL body can then be made by specifying a list of blocks after the `body` keyword. A body can be used as a function body, conditional body, loop body, or similar. The purpose of blocks as an intermediate structure between statement and body is to allow for more organized, readable generated code. For example, the generator can choose to insert a blank line between blocks so lines of code related to the same task are visually grouped together. However, for the case where there is no need for distinct blocks in a body, the body can be made directly from a list of statements with `bodyFromStatements` or even from a single statement with `oneLiner`.

As alluded to before, bodies can be used in more complex statements like conditionals and loops. An if-then-else statement can be written in GOOL as `ifCond` followed by a list of condition-body pairs and then a final body for the else. An example is shown below.

```
ifCond [
  (foo ?> litInt 0, oneLiner (
    printStrLn "foo is positive")),
  (foo ?< litInt 0, oneLiner (
    printStrLn "foo is negative"))]
  (oneLiner $ printStrLn "foo is zero")
```

`ifNoElse` can be used in place of `ifCond` for when there is no else condition. GOOL also supports `switch` statements. A for-loop can be written in GOOL as `for` followed by a statement to initialize the loop variable, a condition, a statement to update the loop variable, and a body. GOOL also offers `forRange` loops, which are given a starting value, ending value, and step size, as well as `forEach` loops. The following examples assume variable `age` and `ages` and body `loopBody` have already been defined:

```
for (varDecDef age (litInt 0))
  (age < litInt 10) (age &++) loopBody
forRange age (litInt 0) (litInt 9)
  (litInt 1) loopBody
forEach age ages loopBody
```

While-loops are also available, using keyword `while` followed by the condition and body. Finally, GOOL offers try-catch statements, which are written as `tryCatch` followed by a body to try and a body for when an exception is caught.

A function in GOOL is specified by `function` followed by the function name, scope, either static or dynamic binding, type, list of parameters, and body. Methods are defined similarly, using the `method` keyword, with the only other difference from functions being that the name of the class containing the method must be specified. Parameters are built from variables, using `stateParam` or `pointerParam`. The "add" function called in an earlier example can thus be defined like so, assuming variables "num1" and "num2" have been defined:

```
function "add" public dynamic_ int
  [stateParam num1, stateParam num2]
  (oneLiner (returnState (num1 #+ num2)))
```

The `pubMethod` and `privMethod` shortcuts can be used for public dynamic methods and private dynamic methods, respectively. `mainFunction` followed by a body defines the main function of a program. A documented function can be generated with `docFunc` followed by a brief function description, a list of parameter descriptions, a description of what is returned (if applicable), and lastly the function itself. A documented function will have comments in Doxygen-style, and GOOL offers further support for compiling Doxygen documentation, to be discussed at the end of this section.

Classes are defined with `buildClass` followed by the class name, name of the parent class (if applicable), scope, list of state variables, and list of methods. State variables can be built by `stateVar` followed by an integer, scope, static or dynamic binding, and the variable itself. The integer is a measure of delete priority. `constVar` can be used for constant state variables. Shortcuts for state variables include `privMVar` for private dynamic, `pubMVar` for public dynamic, and `pubGVar` for public static variables. Assuming variables "var1" and "var2" and methods "mth1" and "mth2" have been defined, a class containing them all can be defined as:

```
buildClass "FooClass" Nothing public
  [pubMVar 0 var1, privMVar 0 var2]
  [mth1, mth2]
```

"Nothing" indicates that this class does not have a parent. `privClass` and `pubClass` are shortcuts for private and public classes, respectively. Classes can be documented using `docClass` followed by a description of the class then the class itself. Like with functions, the documentation will be in Doxygen-style.

Functions and classes can be grouped together into a GOOL module using `buildModule`, followed by the module name, a list of libraries to import, the list of functions, and the list of classes. Passing a module to `fileDoc` will put the finishing touches on the generated file, such as imports of standard libraries and preprocessor guards for C++ header files.

```
fileDoc $ buildModule "mod1" ["library1"]
  [func1, func2] [class1, class2]
```

Documenting a file with a Doxygen-style file header can be done with `docMod` followed by a file description, list of author names, date string, and then the file, specified in GOOL, to document.

Finally, at the top of the GOOL hierarchy are programs, auxiliary files, and packages. A program is constructed by passing a program name and list of files to `prog`. Then a program and a list of auxiliary files can be passed to `package` to create a complete package, specified in GOOL. Auxiliary files are non-OO code files that augment the OO program. For example, a Doxygen configuration file is an auxiliary file that can be specified using the `doxConfig` keyword, and a makefile is another auxiliary file that can be specified with the `makefile` keyword. One of the parameters to the `makefile` keyword will toggle generation of a `make doc` rule, which will compile the Doxygen documentation with the generated Doxygen configuration file.

Syntax for higher-level patterns will be discussed in Section 5.

4 GOOL Implementation

5 Higher-level GOOL functions

- simple examples: log, sin, etc. , args
- bigger examples: print, listAppend, listSize
- even bigger examples: listSlice, inOutFunc, inOutCall, getMethod, setMethod
- design patterns: runStrategy, checkState, addObserver, initObserverList
- auxiliary files

6 Related Work

We divide the Related Work into the following categories

- cat 1
- cat 2
- cat 3

which we present in turn.

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.

7 Future Work

8 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 …