GOOL: A Generic Object-Oriented Language

Jacques Carette  
Department of Computing and Software  
McMaster University  
Hamilton, Ontario, Canada  
carette@mcmaster.ca

Brooks MacLachlan  
Department of Computing and Software  
McMaster University  
Hamilton, Ontario, Canada  
maclachb@mcmaster.ca

Spencer Smith  
Department of Computing and Software  
McMaster University  
Hamilton, Ontario, Canada  
smiths@mcmaster.ca

Abstract
We present GOOL, a Generic Object-Oriented Language. GOOL shows that with the right abstractions, a language can capture the essence of object-oriented programs. GOOL generates human-readable, documented and idiomatic code in Python, Java, C#, and C++. In it, we can express common programming idioms and patterns.

CCS Concepts · Software and its engineering → Source code generation; Abstraction, modeling and modularity; Object oriented languages.

Keywords Code Generation, Domain Specific Language, Haskell, Documentation

ACM Reference Format:

1 Introduction
Java or C#? As languages, this is close to a non-question: the two are so similar that only ecosystem issues would be the deciding factor. Unlike the question “C or Prolog?”, which is almost non-sensical, as the kinds of applications where each is well-suited are vastly different. But, given a single paradigm, such as Object-Oriented (OO), would it be possible to write a meta-language that captures the essence of writing OO programs? They generally all contain (mutable) variables, statements, conditionals, loops, methods, classes, etc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PEPM ’20, January 20, 2020, New Orleans, LA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7096-7/20/01 . . . $15.00
https://doi.org/10.1145/3372884.3373159

OO programs written in different languages appear, superficially, quite dissimilar. But this is mostly due to syntactic differences. Are they so different in the utterances that one can make? Are OO programs akin to sentences in Romance languages (French, Spanish, Portuguese, etc.) which, although syntactically different, are structurally very similar?

This is what we set out to explore. One non-solution is to pick a language and implement a translator to the others. It is feasible — say by engineering a multi-language compiler (such as gcc) to de-compile its Intermediate Representation (IR) into most of its input languages. The end-results would be wildly unidiomatic; roughly the equivalent of a novice in a (spoken) language “translating” word-by-word.

So we set out to design a meta-language that embodies the common semantic concepts of OO languages, encoded so that the necessary information for translation is present. This language is agnostic of the eventual target language — and free of the idiosyncratic details of any given language. In fact, with proper care, one can go even further and teach the translator about idiomatic patterns of each target language.

Trying to capture all the subtleties of each language is hopeless — akin to capturing the rhythm, puns, metaphors, similes, and cultural allusions of a sublime poem in translation. But programming languages are most often used for much more prosaic tasks: writing programs for getting things done. This is closer to translating technical textbooks, making sure that all of the meaningful material is preserved.

We want to capture the conceptual meaning of OO programs, so as to fully automate the translation from the “conceptual” to human-readable, idiomatic code, in mainstream languages. At some level, this is not new. Domain-Specific Languages (DSLs) are high-level languages with syntax and semantics tailored to a specific domain [17]. A DSL abstracts over the details of “code”, providing notation to specify domain-specific knowledge in a natural manner. DSL implementations often work via translation to a General Purpose programming Language (GPL) for execution. Some generate human-readable code [5, 12, 18, 24]. This is what we do, for the domain of OO programs.

While designing a generic OO language is a worthwhile endeavour, we had a second motive: we needed a means to do exactly that as part of our Drasil project [22, 23]. The idea of Drasil is to generate all the requirements documentation and code from expert-provided domain knowledge. The
generated code needs to be human readable so that experts can certify that it matches their requirements. We largely rewrote SAGA [5] to create GOOL\(^1\). It is implemented as a Haskell embedded DSL (EDSL) that can currently generate code in Python, Java, C#, and C++. Others can be added, with the implementation effort being commensurate to the (semantic) distance to the languages already supported.

In Section 2, we outline the high-level requirements for GOOL, followed by an outline of the syntax in Section 3, to enable concrete examples. The interesting design details are given in Section 4. How we capture idioms and patterns is in Section 5. We close with a discussion of related work, plans for future improvements and conclusions.

## 2 Requirements

Our requirements are as follows:

- **mainstream** Generate code in mainstream OO languages.
- **readable** The generated code should be human-readable.
- **idiomatic** The generated code should be idiomatic.
- **documented** The generated code should be documented.
- **patterns** Common OO patterns should be expressible.
- **expressivity** The language should be rich enough to express a set of existing OO programs, which act as test cases for the language.
- **common** Language commonalities should be abstracted.

Targeting OO languages (mainstream) reflects their popularity, thus the most potential users — one reason that the makers of Scala and Kotlin chose to target the JVM to leverage the Java ecosystem, and Typescript for Javascript.

The readable requirement is not as obvious. We aim to write high-level OO code once and have it be available in many GPLs. One use case is to generate libraries of functions for a narrow domain. As needs evolve and language popularity changes, it is useful to have it immediately available in a number of languages. We use it to get extremely well documented code that would be unrealistic to do by hand, as part of Drasil [22, 23]. readable is also a proxy for understandable, which is helpful for debugging.

The same underlying reasons for readable also drive idiomatic and documented, as they contribute to the human-understandability of the generated code. idiomatic is important as readers would otherwise find the code “foreign”. Documentation spans informal one-liners meant for humans to formal, structured comments for generating API documentation with tools like Doxygen, or static analysis. Readability (and thus understandability) are improved when code is pretty-printed [7]. Thus layout, redundant parentheses, well-chosen variable names, and using a common style with lines that are not too long, are just as useful for generated code as for human-written code. GOOL does not prevent bad code, it just simplifies creating readable, idiomatic and documented code in multiple languages.

\(^1\)Available at https://github.com/JacquesCarette/Drasil as a sub-package.

The patterns requirement is typical of DSLs: common idioms can be reified into a linguistic form instead of being informal. Even some of the design patterns of [10] can become part of the language. This makes writing some OO code even easier in GOOL than in GPLs, but it also helps keep GOOL language-agnostic and facilitates generating idiomatic code. expressivity is about GOOL capturing the ideas contained in OO programs. We test GOOL against real-world examples from the Drasil project, such as software for determining whether glass withstands a nearby explosion and software for simulating projectile motion.

The last requirement (common) that language commonalities be abstracted, is internal: we noticed too much code duplication in our initial backends.

## 3 Creating GOOL

To create a “generic” OO language, we chose an incremental abstraction approach: start from programs written in two different languages, and unify them conceptually. We abstract from concrete programs, not just for expressivity, but also because that is our “domain”. Although what can be said in any given OO language is quite broad, what we actually want to say is often much more restricted. And what we need to say is often even more concise. For example, Java offers introspection features, but C++ doesn’t, so abstracting from portable OO will not feature introspection (although generating idiomatic Java may do so). C++ templates are different: while other languages do not necessarily have comparable meta-programming features, it is not only feasible but easy to provide template-like features in GOOL, as well as some partial evaluation. Thus we do not need to generate templates. We are trying to abstract over the fundamental ideas expressed via OO programs, rather than abstracting over the languages. We believe the end result captures the essence of OO programs. Some features, such as static types, which is not a feature of Python but required in Java, will be present as doing full type inference is unrealistic.

Some features of OO programs are not operational: comments and formatting decisions amongst them. To us, programs are a bidirectional means of communication; they must be valid and executable, but also readable and understandable by humans. Generating code for interpretation by machines is well understood, but generating code for human consumption has been given less attention. We paid close attention to readability features — such as when programmers write long methods, they write them as blocks separated by blank lines, often with comments. Thus in GOOL, bodies are not just a sequence of statements, but a list of blocks, to represent the actual structure of OO programs.
The GOOL language is shown in Table 1. We distinguish a variable from its value\(^2\), motivated by semantic considerations; it is beneficial for stricter typing and enables convenient syntax for patterns for more idiomatic code.

We might eventually give GOOL its own external syntax, but for now it works well as a Haskell EDSL, especially as part of Drasil. We can, with judicious use of smart constructors, somewhat mimic the syntax of OO languages. We also use smart constructors for common idioms, like privMVar to denote a private dynamic state variable, and pubClass for a public class. Note that many of the constructs (see Table 1) have doc versions. We can also generate Makefiles and Doxygen config files.

4 GOOL Implementation

There are three standard methods of encoding EDSLs in Haskell: deep (a set of GADTs), shallow (a set of functions), or “finally tagless” (a set of methods in classes). GOOL uses a “sophisticated” version of tagless [8] involving type families.

Tagless encodes a language as a generalized fold over any representation of the language. Thus what look like GOOL “keywords” are actually methods, typically instantiated to language renderers, but also to static analysis passes. By using type families, each instance can choose different underlying data structures for GOOL’s types. For example the C++ instance stores destructor statements with state variables, which is not needed by the other languages. GOOL is defined by 328 methods across a hierarchy of 43 classes (see Figure 1), grouped by functionality — it is not a small language!

For example, here is part of the class for variables:

```haskell
class (TypeSym repr) => VariableSym repr where
  type Variable repr
```

\(^2\)as befits the use-mention distinction from analytic philosophy

As variables are typed, their representation must be too and thus that capability (the TypeSym class) is a constraint. The associated type Variable repr is a representation-dependent type-level function. Each instance can define its own representation of what a Variable is.

We have defined 300 functions that abstract over commonalities between target languages, making writing new renderers fairly straightforward. GOOL’s Java and C# renderers show this: out of the 328 total methods, 229 are shared. That is 40% more common instances than between Python and Java, for example. A further 37 are partially shared — they call the same common function but with different parameters.

5 Idioms and Patterns

5.1 Idioms

Command line arguments Accessing these differs significantly across languages. Thus we abstract over the details through an argsList that represents the list of arguments, with a dedicated API.

Lists As with command line arguments, list APIs vary considerably. We thus reverse engineer the “useful” API for lists from actual use cases. Lists in OO languages are rarely linked lists (unlike Haskell), but more like dynamically sized vectors. In particular, indexing a list by position, which is a horrifying idea for linked lists, is extremely common. The result is a small set of functions and statements, shown in Table 1 on the line labelled List API.

List slicing (the listSlice statement) is interesting as the code it generates varies a lot by language. For example

```haskell
listSlice someAges (valueOf ages) (Just $ litInt 1)
```
in Python is rendered as:
someAges = ages[1:3:]

while in Java it is
ArrayList<Double> temp = new ArrayList<Double>(0);
for (int i_temp = 1; i_temp < 3; i_temp++) {
    temp.add(ages.get(i_temp));
}
someAges = temp;

Idiomatic code generation is enabled by having appropriate high-level information driving the generation.

**Printing** is also target dependent. Again Python is more “expressive” so that printing a list (via printLn ages) generates print(ages), but in other languages we must generate a loop. There is also similar functionality for reading input.

**Procedures with I/O/B parameters** Our hand-written target codes had methods that used their parameters differently: as inputs, outputs, or both. This is a semantic pattern that is not necessarily obvious in any of the implementations. Once noticed, we created an encoding of that information to generate better, more idiomatic code. Concretely, the following Python code

def applyDiscount(price, discount):
    price = price - discount
    isAffordable = price < 20

    return price, isAffordable

can be captured in GOOL with the inOutFunc idiom:
inOutFunc "applyDiscount" public static_
    [discount] [isAffordable] [price]
    (bodyStatements [
        price &-= valueOf discount,
        isAffordable &-= valueOf price ?< litFloat 20.0])

We can produce the following C#

```csharp
public static void applyDiscount(ref int price, \
    int discount, out Boolean isAffordable) {
    price = price - discount;
    isAffordable = price < 20;
}
```

and C++

```cpp
void applyDiscount(int &price, \
    int discount, bool &isAffordable) {
    price = price - discount;
    isAffordable = price < 20;
}
```
to capture the same idea. The Java version (not shown) is more awkward. A natural task-level "feature" — different kinds of parameters — ends up being rendered differently, but hopefully idiomatically, in each target language. GOOL manages the tedious aspects of generating any needed variable declarations and return statements. To call an inOutFunc function, one must use inOutCall so that GOOL can "line up" all the pieces properly.

Getters and setters are common in OO programs (regardless of whether these actually achieve encapsulation), and mechanical to write. In GOOL, getMethod "FooClass" foo and setMethod "FooClass" foo are sufficient to generate that code, which can be called with get and set, and yet abstracts over the idiosyncracies of each target language.

Design Patterns GOOL currently handles three design patterns: Observer, State, and Strategy [10].

For Strategy, we ensure that the set of strategies that will be used are statically known at generation time, and then generate code only for those that will be used. runStrategy is the user-facing function.

For Observer, initObserverList generates an observer, given a list of initial values; it generates a declaration of an observer list variable, initially containing the given values. addObserver can be used to add a value to the observer list, and notifyObservers will call a method on each of the observers. Currently, the name of the observer list variable is fixed, so there can only be one observer list in a given scope.

The State pattern is specialized to Finite State Machines with fairly general transition functions. Transitions happen on checking, not on changing the state. initState takes a name and a state label and generates a declaration of a variable. changeState changes the state of the variable. checkState is more complex: it takes the name of the state variable, a list of value-body pairs, and a fallback body; it generates a conditional (usually a switch statement) that checks the state and runs the corresponding body, or the fallback body, if none of the states match.

The design patterns could have been coded in GOOL, but having these as language features is useful for two reasons: 1) the GOOL-level code is clearer in its intent (and more concise), and 2) the resulting code can be more idiomatic.

Figure 2 shows a larger example. The recommended style is to name all strings (to avoid hard-to-debug typos) and variables, then write the code proper.

6 Related Work

6.1 General-Purpose Code Generation

Haxe [3] is a general-purpose multi-paradigm language and cross-platform compiler. It compiles to all of the languages GOOL does, and many others. However, it is designed as a more traditional programming language, and does not offer the high-level abstractions that GOOL provides. Furthermore

```plaintext
patternTest :: (MethodSym repr) =>
  repr (Method repr)
patternTest = let
doPattern = "DoSomething"
offState = "Off" onState = "On"
noState = "Neither" obsName = "Observer"
obs1Name = "obs1" obs2Name = "obs2"

// Calculate the total
printTotal = "printTotal" nName = "n"
obsType = obj obsName
n = var n int
obs1 = var obs1Name obsType
obs2 = var obs2Name obsType
newObs = extNewObj obsName obsType []
in
mainFunction (body [block [
  varDec n,
  initState fsmName offState,
  changeState fsmName onState,
  checkState fsmName
  (litString offState ,
    oneLiner $ printStrLn offState ),
  (litString onState ,
    oneLiner $ printStrLn onState ))
  (oneLiner $ printStrLn noState )],
  block [varDecDef obs1 newObs, varDecDef obs2 newObs],
  block [initObserverList obsType [valueOf obs1],
    addObserver $ valueOf obs2,
    notifyObservers (func printNum void [[]] obsType)]
```

Figure 2. GOOL sample code

Haxe strips comments and generates source code around a custom framework; the effort of learning this framework and the lack of comments makes the generated code not particularly readable. The internal organization of Haxe does not seem to be well documented.

Protokit [14] is a DSL and code generator for Java and C++, where the generator is designed to produce 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 generated. Since the "output model" is quite similar to the generated code, it presented challenges with regards to semantic, conventional, and library-related differences between the target languages.
A number of languages for modeling design patterns have (UML) but designed specifically to overcome UML’s shortcomings. DPML [16] is similar to the Unified Modeling Language (UML) in that it is an object-oriented metamodeling language. However, DPML focuses on modeling design patterns rather than general-purpose modeling. DPML is designed to overcome some of the limitations of UML in modeling design patterns.

There are code generators which multiple target OO languages, but all are domain-specific.

**Google protocol buffers** [2] is a DSL for serializing structured data, which can be compiled into Java, Python, Objective C, and C++. **Thrift** [20] 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. **Clearwater** [21] is an approach for implementing DSLs with multiple target languages for components of distributed systems. The **Time Weaver** tool [9] uses a multi-language code generator to generate “glue” code for real-time embedded systems. The domain of mobile applications is host to a bevy of DSLs with multiple target languages, of which **MobDSL** [15] and **XIS-Mobile** [19] are two examples. **Conjure** [1] is a DSL for generating APIs. It reads YAML descriptions of APIs and can generate code in Java, TypeScript, Python, and Rust.

**6.2 Object-Oriented Generators**

There are code generators which multiple target OO languages, but all are domain-specific.

**Google protocol buffers** [2] is a DSL for serializing structured data, which can be compiled into Java, Python, Objective C, and C++. **Thrift** [20] 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. **Clearwater** [21] is an approach for implementing DSLs with multiple target languages for components of distributed systems. The **Time Weaver** tool [9] uses a multi-language code generator to generate “glue” code for real-time embedded systems. The domain of mobile applications is host to a bevy of DSLs with multiple target languages, of which **MobDSL** [15] and **XIS-Mobile** [19] are two examples. **Conjure** [1] is a DSL for generating APIs. It reads YAML descriptions of APIs and can generate code in Java, TypeScript, Python, and Rust.

**6.3 Design Patterns**

A number of languages for modeling design patterns have been developed. The **Design Pattern Modeling Language** (DPML) [16] is similar to the Unified Modeling Language (UML) but designed specifically to overcome UML’s shortcomings. DPML consists of both specification diagrams and instance diagrams for instantiations of design patterns, but does not attempt to generate actual source code. The **Role-Based Metamodeling Language** [13] is also based on UML but with changes to allow for better models of design patterns, with specifications for the structure, interactions, and state-based behavior in patterns. Again, source code generation is not attempted. Another metamodel for design patterns includes generation of Java code [4]. IBM developed a DSL in the form of a visual interface for generation of OO code based on design patterns [6]. The languages that generate code do so only for design patterns, not for any general-purpose code, as GOOL does.

**7 Future Work**

Currently GOOL code is typed based on what it represents: variable, value, type, or method, for example. The type system does not go “deeper”, so that values (such as booleans and strings) are simply “values”. This is sufficient to allow us to generate well-formed code, but not to ensure that it is well-typed. We have started to statically type GOOL.

We also want to improve the generated import statements, via tracking actual dependencies on features used. In general, we can do various kinds of static analyses to enhance the code generation quality, such as being more precise about throws. Exception in Java.

We also want to interface with external libraries, such as a variety of ODE solvers, since Drasil currently focuses on scientific applications. The API for available solvers varies considerably, so we will need to change the “shape” of generated code depending on the user’s choice.

Some implementation decisions, such as representing lists in Java as ArrayList, are hard-coded. But we could have used Vector instead. We would like such a choice to be user-controlled. Another such decision point is to allow users to choose which specific external library to use.

**8 Conclusion**

We currently successfully use GOOL to simultaneously generate code in all of our target languages for the glass and projectile programs described in Section 2.

Conceptually, mainstream object-oriented languages are similar enough that it is indeed feasible to create a single “generic” object-oriented language that can be “compiled” to them. Of course, these languages are syntactically quite different in places, and each contains some unique ideas as well. In other words, there exists a “conceptual” object-oriented language that is more than just “pseudocode”: it is a full-fledged executable language that captures the common essence of mainstream OO languages.

GOOL is an unusual DSL, as its “domain” is actually that of object-oriented programs. More precisely, of conceptual programs that can be easily written in languages containing a procedural core with an object-oriented layer on top — which is what Java, Python, C++, and C# are.

Since we are capturing conceptual programs, we can achieve several things that we believe are together new:

- generation of idiomatic code for each target language,
- turning coding patterns into language idioms,
- generation of human-readable, well-documented code.

We must also re-emphasize this last point: that for GOOL, the generated code is meant for human consumption as well as for computer consumption. This is why semantically meaningless concepts such as “blocks” exist: to be able to chunk code into pieces meaningful for the human reader, and provide documentation at that level as well.
References


