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 [5, 18]. 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 [10]. 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 [16]. 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 want. 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)[2], 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++[3]. 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

While we outlined some of our requirements above, here we will give a complete list, along with acronyms (to make referring to them simpler), as well as some reasoning behind each requirement.

mainstream Generate code in mainstream object-oriented languages.
readable The generated code should be human-readable,
idiomatic The generated code should be idiomatic,
documented The generated code should be documented,
patterns The generator should allow one to express common OO patterns.
common Language commonalities should be abstracted.
expressivity The resulting language should be rich enough to express a certain set of test cases, drawn from scientific computation software.

Targetting OO languages (**mainstream**) is primarily because of their popularity, and thus would enjoy the most potential users — in much the same way 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. As DSL users are typically domain experts who are not "programmers", why generate readable code? Few Java programmers ever look at JVM bytecode, and few C++programmers look at assembly. But GOOL's aim is different: to allow writing high-level OO code once, but have it be available in many GPLs. One use case would be to generate libraries of utilities for a narrow domain. As needs evolve and language popularity changes, it is useful to have it immediately available in a number of languages. Another use, which is a core part of our own motivation, is to have *extremely well documented* code, indeed to a level that would be unrealistic to do by hand. But this documentation is crucial in domains where *certification* of code is required.

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 many human readers would find the code "foreign" otherwise, and would not be keen on using it. Note that documentation can span from informal comments meant for humans, to formal, structured comments useful for generating API documentation with tools like Doxygen, or with a variety of static analysis tools. Readability (and thus understandability) are improved when code is pretty-printed[7]. Thus taking care of layout, redundant parentheses, well-chosen variable names, using a common style with lines which are not too long, are just as valid for generated code as for human-written code. GOOL does not prevent users from writing undocumented or complex code, if they choose to do so. It just makes it easy to have **readable**, **idiomatic** and **documented** code in multiple languages.

The **patterns** requirement is typical of DSLs: common programming patterns can be reified into a proper linguistic form instead of being merely informal. In particular some of the *design patterns* of [10] can become part of the language itself. This does make writing some OO code even easier in GOOL than in GPLs, it also helps quite a lot with keeping GOOL language-agnostic and generating idiomatic code. Illustrative examples will be given in Section 5. But we can give an indication now as to why this helps: Consider Python's ability to return multiple values with a single return statement, which is uncommon in other languages. Two choices might be to disallow this feature in GOOL, or throw an error on use when generating code in languages that do not support this feature. In the first case, this would likely mean unidiomatic Python code, or increased complexity in the Python generator to infer that pattern. The second option is worse still: one might have to resort to writing language-specific GOOL, obviating the whole reason for the language! Multiple-value return statements are always used when a function returns multiple outputs; what we can do in GOOL is to support such multiple-output functions, and then generate the idiomatic pattern of implementation in each target language.

The last two requirements, that language commonalities (**common**) be abstracted, and that we can phrase a certain collection of test cases (**expressivity**) are internal requirements: we didn't set out to create GOOL as a primary artifact, but as a side-effect of other work on different methods of creating long-lived scientific software. Part of long-lived means that we need to be flexible about the technology, thus

[2]GOOL is publicly available; the exact link will be given once the paper is no longer anonymous
[3]and is close to generating Lua and Objective-C, but those backends have fallen into disuse

needing to be polymorphic on the underlying language. Regarding commonalities, we noticed a lot of repeated code in our initial backends, something that ought to be distasteful to most programmers. For example, writing a generator for both Java and C♯makes it incredibly clear how similar the two languages are.

3 Creating GOOL

How do we go about creating a "generic" object-oriented language? We chose an incremental abstraction approach: start from two languages, and unify them *conceptually*. In other words, pay very close attention to the *denotational* semantics of the features, some attention to the operational semantics, and ignore syntactic details.

This is most easily done from the core imperative language outwards. Most languages provide similar basic types (variations on integers, floating point numbers, characters, strings, etc) and functions to deal with them. The core expression language tends to be extremely similar cross languages. One then moves up to the statement language — assignments, conditionals, loops, etc. Here we start to encounter variations, and choices can be made, and we'll cover that later.

For ease of experimentation, we chose to make GOOL an embedded domain specic language (EDSL) inside Haskell. Haskell is very well-suited for this task, offering a variety of features (GADTs, type classes, parametric polymorphism, kind polymorphism, etc) which is extremely useful for building languages. Its syntax is also fairly liberal, so that it is possible to create *smart constructors* that somewhat mimic the usual syntax of OO languages.

3.1 GOOL Syntax: Imperative core

As our exposition has been somewhat abstract until now, it is useful to dive in and give some concrete syntax, so as to be able to illustrate our ideas with valid code.

Specifically, 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)` represents a variable called "ages" that is a list of integers. This illustrates a (necessary) design decision: even though we target languages like Python, as we also target Java, types are necessary. As type inference for OO languages is too difficult, we chose to be explicitly typed.

As some constructions are common, it is useful to offer shortcuts for defining them; for example, the above can also be done via `listVar "ages" int`. Typical use would be

```
let ages = listVar "ages" int in
```

so that `ages` can be used directly from then on. Other GOOL syntax for specifying variables is shown in Table 1.

Table 1. Syntax for specifying variables

GOOL Syntax	Semantics
`extVar`	for a variable from an external library
`classVar`	for a variable belonging to a class
`objVar`	for a variable belonging to an object
`$->`	infix operator form of `objVar`
`self`	for referring to an object in the definition of its class

Note that GOOL distinguishes a variable from its value[4]. To get the value of `ages`, one must write `valueOf ages`. The reason for this distinction will be made clear in section **??**, driven by semantic considerations. This is beneficial for stricter typing and enables convenient syntax for **patterns** that translate to more idiomatic code.

Syntax for literal values is shown in Table 2 and for operators on values is shown in Table 3. In GOOL, each operator is prefixed with an additional symbol based on type. Operators that return Booleans are prefixed by a `?`, operators on numeric values are prefixed by `#`, and other operators are prefixed by `$`.

Table 2. Syntax for literal values

GOOL Syntax	Semantics
`litTrue`	literal Boolean true
`litFalse`	literal Boolean false
`litInt i`	literal integer `i`
`litFloat f`	literal float `f`
`litChar c`	literal character `c`
`litString s`	literal string `s`

Syntax for defining values with conditional expressions or function applications is shown in Table 4. `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.

Variable declarations are statements, and take a variable specification as argument. For `foo = var "foo" int`, the corresponding variable declaration would be `varDec foo`, and to also initialize it `varDecDef foo (litInt 5)` can be used.

Assignments are represented by `assign a (litInt 5)`. Convenient infix and postfix operators are also provided, prefixed by `&`: `&=` is a synonym for `assign`, and C-like `&+=`,

[4] as befits the use-mention distinction from analytic philosophy

Table 3. Operators for making expressions

GOOL Syntax	Semantics
?!	Boolean negation
?&&	conjunction
?||	disjunction
?<	less than
?<=	less than or equal
?>	greater than
?>=	greater than or equal
?==	equality
?!=	inequality
#~	numeric negation
#/^	square root
#|	absolute value
#+	addition
#-	subtraction
#*	multiplication
#/	division
#%	modulus
#^	exponentiation

Table 4. Syntax for conditionals and function application

GOOL Syntax	Semantics
inlineIf	conditional expression
funcApp	function application, to a list of parameters
extFuncApp	function application, for external library functions
newObj	for calling an object constructor (extNewObj exists too)
objMethodCall	for calling a method on an object

&++, &-= and &~- (the more intuitive &-- cannot be used as -- starts 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.

Most languages have statement blocks, introduced by block with a list of statements in GOOL. Bodies (body) are composed of a list of blocks, and can be used as a function body, conditional body, loop body, etc. The purpose of blocks as an intermediate 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. Naturally shortcuts are provided for single-block bodies (bodyStatements) and for the common single-statement case, oneLiner.

GOOL has two forms of conditionals: if-then-else via ifCond (which takes a list of pairs of conditions and bodies) and if-then via ifNoElse. For example:

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

GOOL also supports switch statements.

There are a variety of loops: for-loops (for), which are parametrized by a statement to initialize the loop variable, a condition, a statement to update the loop variable, and a body; forRange loops, which are given a starting value, ending value, and step size; as well as forEach loops. For example:

```
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 (while) are parametrized by a condition and a body. Finally, try-catch statements (tryCatch) are parametrized by two bodies.

3.2 GOOL Syntax: OO features

A function declaration is followed by the function name, scope, binding type (static or dynamic), type, list of parameters, and body. Methods (method) are defined similarly, with the addition of the specification of the containing class' name. Parameters are built from variables, using param or pointerParam. For example, assuming variables "num1" and "num2" have been defined, one can define an add function as follows:

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

The pubMethod and privMethod shortcuts are useful for public dynamic and private dynamic methods, respectively. mainFunction followed by a body defines the main function of a program. docFunc generates a documented function from a function description and a list of parameter descriptions, an optional description of the return value, and the function itself. This generates Doxygen-style comments.

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. For example:

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

`Nothing` here indicates that this class does not have a parent, `privClass` and `pubClass` are shortcuts for private and public classes, respectively. `docClass` serves a similar purpose as `docFunc`.

3.3 GOOL syntax: modules and programs

Akin to Java packages and other similar constructs, GOOL has modules (`buildModule`) consisting of a module name, a list of libraries to import, a list of functions, and a list of classes. Module-level comments are done with `docMod`.

Finally, at the top of the GOOL hierarchy are programs, auxiliary files, and packages. A program (`prog`) has a name and a list of files. A `package` is a program and a list of auxiliary files. These files are non code files that augment the program. Examples are a Doxygen configuration file (`doxConfig`), and a makefile (`makefile`). One of the parameters of `makefile` toggles generation of a `make doc` rule, which will compile the Doxygen documentation with the generated Doxygen configuration file.

4 GOOL Implementation

There are two "obvious" means of dealing with large embedded DSLs in Haskell: either as a set of Generalized Algebraic Data Types (GADTs), or using a set of classes, in the "finally tagless" style [8] (we will refer to it as simply *tagless* from now on). The current implementation uses a "sophisticated" version of tagless. A first implementation [5] used a straightforward version of tagless which did not allow for enough generic routines to be properly implemented. This was replaced by a version based on GADTs, which fixed that problem, but did not allow for *patterns* to be easily encoded. Thus the current version has gone back to tagless, but also uses *type families* in a crucial way.

It is worth recalling that in tagless, the means of encoding a language, through methods from a set of classes, really encodes a generalized *fold* over any *representation* of the language. Thus what looks like GOOL "keywords" are either class methods or generic functions that await the specification of a dictionary to decide on the final interpretation of the representation. We typically instantiate these to language renderers, but we're also free to do various analysis passes if we wish.

Because tagless representations give an embedded syntax to a DSL while being polymorphic on the eventual semantic interpretation of the terms, [8] dubs the resulting classes "symantic". Our language is defined by a hierarchy of 43 of these symantic classes, grouped by functionality. For example, there are classes for programs, bodies, control blocks, types, unary operators, variables, values, selectors, statements, control statements, blocks, scopes, classes, modules, and so on. These define 328 different methods — GOOL is not a small language!

Perhaps a graph of all 43 classes as a picture would be a nice illustration?

For example, here is how variables are defined:

```
class (TypeSym repr)
  => VariableSym repr where
    type Variable repr
    var :: Label -> repr (Type repr)
                 -> repr (Variable repr)
```

As variables are typed, a representation of variables much also know how to represent types, thus we constrain our representation with that capability, here the `TypeSym` class. We also notice the use of an *associated type* `type Variable repr`. This is a type-level function which is representation-dependent. Each instance of this class is free to define its own internal representation of what a `Variable` is. `var` is then a constructor for variables, which takes a `Label` and a representation of a type, returning a representation of a variable. Specifically, `repr` has kind `* -> *`, and thus `Variable` has kind `(* -> *) -> *`. In `repr (X repr)`, the type variable `repr` appears twice because there are two layers of abstraction: over the target language, handled by the outer `repr`, and over the underlying types to which GOOL's types map, represented by the inner `repr`.

The principal use we make of the flexibility of type families on a per-target-language basis is to record more (or less) information for successful code generation. For example, the internal representation for a state variable in C++stores the corresponding destructor code for the variable, but in the other languages destructors are not needed so the internal representation of a state variable is just a `Doc`.

For example, for Java, we instantiate the class as follows:

```
instance VariableSym JavaCode where
  type Variable JavaCode = VarData
  var = varD
```

where `JavaCode` is essentially the `Identity` monad by another name:

```
newtype JavaCode a = JC {unJC :: a}
```

The `unJC` record field is useful for type inference: when applied to an otherwise generic term, it lets Haskell infer that we're then wishing to only consider the `JavaCode` instances. `VarData` is defined as

```
data VarData = VarD {
  varBind :: Binding,
```

[5] citation omitted for anonymization

```
  varName :: String,
  varType :: TypeData,
  varDoc :: Doc}
```

In other words, for every (Java) variable, we store its binding time, either `Static` or `Dynamic`, the name of the variable as a `String`, and its type as a `TypeData`, which is the representation for `Types` for Java, and finally how the variable should appear in the generated code, represented as a `Doc`. `Doc` comes from the package `Text.PrettyPrint.HughesPJ` and represents formatted text.

All representing structures contain at least a `Doc`. It can be considered to be our *dynamic* representation of code, from a partial-evaluation perspective. The other fields are generally *static* information used to optimize the code generation.

Generally, GOOL prefers to work generically. So there is as little code as possible that works on `VarData` directly. Instead, there are methods like `variableDoc`, part of the `VariableSym` type class, with signature:

```
variableDoc :: repr (Variable repr)
  -> Doc
```

which acts as an accessor. For `JavaCode`, its instance is straightforward:

```
variableDoc = varDoc . unJC
```

Here are a few more examples of the kinds of additional information stored by each representation. `Statement` stores a `Terminator` which is how that language indicates how a statement is to be terminated (frequently this is a semicolon). For `Method`, a Boolean indicates whether it is the main method. For `Value`, `UnaryOp` and `BinaryOp`, precendence information is stored so that printing can elide parentheses whenever possible, leading to more readable code.

Note that the `JavaCode` instance of `VariableSym` defines the `var` function via the `varD` function:

```
varD :: (RenderSym repr) => Label ->
  repr (Type repr) -> repr (Variable repr)
varD n t = varFromData Dynamic n t
  (varDocD n)


varDocD :: Label -> Doc
varDocD = text
```

`varD` is generic, i.e. works for all instances, via dispatching to other generic functions, such as `varFromData`:

```
varFromData :: Binding -> String ->
  repr (Type repr) -> Doc ->
  repr (Variable repr)
```

This method is in a type class `InternalVariable`. Several of these "internal" classes exist, none of which are exported from GOOL's interface. They however contain functions useful for the various language renderers, but not meant to be used to construct code representations, as they reveal too much of the internals (and are rather tedious to use too). One important example is the `cast` method, which is never needed by user-level code, but frequently used by higher-level functions.

`varDocD` can simply be `text` as `Label` is simply an alias for a `String` – and Java variables are simply their names, which is indeed the case for most OO languages. Exceptions can use the class mechanism to override this in their specific case.

This genericity makes writing new renderers for new languages fairly straightforward. GOOL's Java and C♯renderers demonstrate this fact well. Out of 328 methods across all of GOOL's type classes, the instances of 228 of them are shared between the Java and C♯renderers, in that they are just calls to the same common function. A further 37 are partially shared, for example they call the same common function but with different parameters. 143 methods are actually the same between all 4 languages GOOL currently targets. This might indicate that some should be generic functions rather than class methods, but we have not investigated this in detail yet.

Examples from Python and C♯are not shown here because they both work very similarly to the Java renderer. There are `PythonCode` and `CSharpCode` analogs to `JavaCode`, the underlying types are all the same, and the methods are defined by calling common functions where possible or by constructing the GOOL value directly in the instance definition, if the definition is unique to that language.

C++is different since most modules are split between a source and header file. To generate C++, we traverse the code twice, once to generate the header file and a second time to generate the source file corresponding to the same module. This is done via two instances of the classes, for two different types: `CppSrcCode` for source code and `CppHdrCode` for header code. Since a main function does not require a header file, the `CppHdrCode` instance for a module containing only a main function is empty. The renderer optimizes empty modules/files away — for all renderers.

As C++source and header should always be generated together, a third type, `CppCode` achieves this:

```
data CppCode x y a =
  CPPC {src :: x a, hdr :: y a}
```

The type variables `x` and `y` are intended to be instantiated with `CppSrcCode` and `CppHdrCode`, but they are left generic so that we may use an even more generic `Pair` class:

```
class Pair (p :: (* -> *) -> (* -> *)
  -> (* -> *)) where
  pfst :: p x y a -> x a
  psnd :: p x y b -> y b
  pair :: x a -> y a -> p x y a
```

```
instance Pair CppCode where
  pfst (CPPC xa _) = xa
  psnd (CPPC _ yb) = yb
  pair = CPPC
```

`Pair` is a *type constructor* pairing, one level up from Haskell's own `(,) :: * -> * -> *`. It is given by one constructor and two destructors, much as the Church-encoding of pairs into the λ-calculus.

To understand how this works, here is the instance of `VariableSym` but for C++:

```
instance (Pair p) => VariableSym
  (p CppSrcCode CppHdrCode) where
  type Variable
    (p CppSrcCode CppHdrCode) = VarData
  var n t = pair
    (var n $ pfst t) (var n $ psnd t)
```

The instance is generic in the pair representation `p` but otherwise concrete, because `VarData` is concrete. The actual instance code is straightforward, as it just dispatches to the underlying instances, using the generic wrapping/unwrapping methods from `Pair`. This pattern is used for all instances, so adapting it to any other language with two (or more) files per module is straightforward.

At the program level, the difference between source and header is no longer relevant, so they are joined together into a single component. For technical reasons, currently `Pair` is still used, and we arbitrarily choose to put the results in the first component.

While "old" features of OO languages — basically features that were already present in ancestor procedural languages like Algol — have fairly similar renderings, more recent (to OO languages) features such as for-each loops show more variations. More precisely, the first line of a for-each loop in Python, Java, C♯ and C++ are (respectively):

```
for age in ages:
```

```
for (int age : ages) {
```

```
foreach (int age in ages) {
```

```
for (std::vector<int>::iterator age \
  = ages.begin(); age != ages.end(); \
  age++) {
```

By providing `forEach`, GOOL abstracts over these differences.

5 Encoding Patterns

There are various levels of "patterns" to encode. The previous section documented how to encode the programming language aspects. Now we move on to other patterns, from simple library-level functions, to simple tasks (command-line arguments, list processing, printing), on to more complex patterns such as methods with a mixture of input, output and in-out parameters, and finally on to design patterns.

Consider the simple trigonometric sine function, called `sin` in GOOL. It is common enough to warrant its own name, even though in most languages it is part of a library. A GOOL expression `sin foo` can then be seamlessly translated to yield `math.sin(foo)` in Python, `Math.sin(foo)` in Java, `Math.Sin(foo)` in C♯, and `sin(foo)` in C++. Other functions are handled similarly. This part is easily extensible, but does require adding to GOOL classes.

A slightly more complex task is accessing arguments passed on the command line. This tends to differ more significantly accross languages. GOOL offers an abstraction of these mechanisms, through an `argList` function that represents the list of arguments, as well as convenience functions for common tasks such as indexing into `argList` and checking if an argument at a particular position exists.

Variations on lists are frequently used in OO code. But the actual API in each language tends to vary quite a lot, so we need to provide a single abstraction that provides sufficient functionality to do useful list computations. Rather than abstracting from the functionality provided in the libraries of each language to find some common ground, we instead reverse engineer the "useful" API from actual use cases in scientific code.

One thing we immediately notice from such an exercise is that lists in OO languages are rarely *linked lists* (unlike in Haskell, our host language), but rather more like a dynamically sized vector. In particular, indexing a list by position, which is a horrifying idea for linked lists, is extremely common.

This narrows things down to a small set of functions and statements: For example, `listAccess (valueOf ages) (litInt 1)` will generate `ages[1]` in Python and C♯, `ages.get(1)` in Java, and `ages.at(1)` in C++. List slicing is a very convenient higher-level primitive. The `listSlice` *statement* gets a variable for the rest, a list to slice, and three values representing the starting and ending indices for the slice and

Table 5. List functions

GOOL Syntax	Semantics
`listAccess`	access a list element at a given index
`listSet`	set a list element at a given index to a given value
`at`	same as `listAccess`
`listSize`	get the size of a list
`listAppend`	append a value to the end of a list
`listIndexExists`	check whether the list has a value at a given index
`indexOf`	get the index of a given value in a list

the step size. These last three values are all optional (we use Haskell's `Maybe` for this) and default to the start of the list, end of the list and 1 respectively. To take elements from index 1 to 2 of `ages` and assign the result to `someAges`, we can use

```
listSlice someAges (valueOf ages)
  (Just $ litInt 1) (Just $ litInt 3)
  Nothing
```

List slicing is of particular note because the generated Python is particularly simple, unlike in other languages; the Python:

```
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;
```

where we use backslashes in generated code to indicate manually inserted line breaks so that the code fits in this paper's narrow column margins. This demonstrates GOOL's idiomatic code generation, enabled by having the appropriate high-level information to drive the generation process.

Printing is another such important feature, which generates quite different code depending on the target language. Here again Python is more "expressive" so that printing a list (via `printLn ages`) generates `print(ages)`, but in other languages must generate a loop; for example, in Java:

Can't better code be generated in Java for this? Not that I want GOOL changed now, what I mean is perhaps using C++ as an example, where this is not going to be a feature (I think), the loop is indeed needed.

```
System.out.print("[");
for (int list_i1 = 0; \
  list_i1 < ages.size() - 1; \
  list_i1++) {
    System.out.print(ages.get(list_i1));
    System.out.print(", ");
}
if (ages.size() > 0) {
    System.out.print(ages.get( \
      ages.size() - 1));
}
System.out.println("]");
```

In addition to printing, there is also functionality for reading input.

Moving to larger-scale patterns, we noticed that our codes had methods that used its parameters differently: some were used as inputs, some as outputs and some for both purposes. This was a *semantic* pattern that was not necessarily obvious in any of the implementations. But once we noticed it, we could use that information to generate better, more idiomatic code in each language, while still capturing the higher-level semantics of the functionality we were trying to implement. More concretely, consider a function `applyDiscount` that takes a price and a discount, subtracts the discount from the price, and returns both the new price and a Boolean for whether the price is below 20. In GOOL, using `inOutFunc`, assuming all variables mentioned have been defined:

```
inOutFunc "applyDiscount" public static_
  [discount] [isAffordable] [price]
  (bodyStatements [
    price &-= valueOf discount,
    isAffordable &=
      valueOf price ?< litFloat 20.0])
```

`inOutFunc` takes three lists of parameters, the input, output and input-output respectively.

edited until here

This function has multiple outputs—`price` and `isAffordable`—and each of GOOL's target languages handles functions with multiple outputs differently. In Python, return statement with multiple values is used:

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

    return price, isAffordable
```

In Java, the outputs are returned in an array of `Objects`:

```
public static Object[] applyDiscount( \
  int price, int discount) \
  throws Exception {
    Boolean isAffordable;

    price = price - discount;
    isAffordable = price < 20;

    Object[] outputs = new Object[2];
    outputs[0] = price;
    outputs[1] = isAffordable;
    return outputs;
}
```

In C♯, the outputs are passed as parameters, using the `out` keyword if it is only an output or the `ref` keyword if it is both an input and an output:

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

```
    isAffordable = price < 20;
}
```

And in C++, the outputs are passed as pointer parameters:

```
void applyDiscount(int &price, \
  int discount, bool &isAffordable) {
    price = price - discount;
    isAffordable = price < 20;
}
```

The structure of the function in each language is different, from the parameters to the function body to the return type. But each uses the language's most natural way of defining a function with multiple outputs. GOOL generates any needed variable declarations and return statements automatically, so the GOOL user is saved from typing these lines out manually. Functions defined with `inOutFunc` can be called with `inOutCall`, which again accepts the three lists of inputs, outputs, and those that are both. Through `inOutCall`, GOOL will again automatically generate any needed variable declarations and assignments, such as declaring the `outputs` array and then assigning its elements to the appropriate variables in Java.

In OO programming, it is common to write getter and setter methods in a class, so GOOL abstracts over these patterns as well. `getMethod` can be used to define a getter, and `setMethod` for a setter. Each needs only be provided the name of the class to which the method will belong, and the variable to be get or set. So, assuming the class `FooClass` has a variable `foo`, a getter for `foo` is simply `getMethod "FooClass" foo` and a setter is simply `setMethod "FooClass" foo`. The generated set method in Python looks like:

```
def setFoo(self, foo):
    self.foo = foo
```

In Java:

```
public void setFoo(int foo) \
  throws Exception {
    this.foo = foo;
}
```

In C♯:

```
public void setFoo(int foo) {
    this.foo = foo;
}
```

And in C++:

```
void FooClass::setFoo(int foo) {
    this->foo = foo;
}
```

We show this example not only to demonstrate how a single, small line of GOOL code can generate much more code in the target language, but also to visualize some of the idiosyncracies present in the target languages. Even for such a simple function, there are subtle differences in each language that would be difficult to keep track of if someone were trying to write these programs manually, and GOOL saves the programmer from this tedium. For calling methods defined with `getMethod` or `setMethod`, GOOL also provides `get` and `set` functions, which must be passed the value of the object on which the method should be called, the variable to get or set, and, in the case of `set`, the value to set it to.

The final category of higher-order functions we will discuss are those that abstract over the design patterns described in [10]. GOOL currently has syntax for defining instances of three design patterns: Observer, State, and Strategy. The versions of these design patterns GOOL generates are simplified and small in scale. In the case of the Strategy pattern, Haskell does the work of storing and checking which strategy to use, only actually generating code for a strategy when it is used. `runStrategy` is the user-facing function for using the Strategy pattern in GOOL. It must be passed the name of the strategy to use, a list of pairs of strategy names and bodies, and `Maybe` a variable and value to assign after running the strategy. Haskell will check if the given strategy name is in the list, and simply generate the corresponding body if it is.

For the Observer pattern, `initObserverList` can be passed a list of values and will generate a declaration of an observer list variable initially containing the given values. `addObserver` can then be used to add a given value to the observer list, and `notifyObservers` will call a method on each of the observers. The name of the observer list variable is fixed, so there can only be one observer list in a given scope.

For the State pattern, `initState` will take a name and a state label and generate a declaration of a variable with the given name and assign it the given state. The states are just literal strings. `changeState` takes the variable name and a new state, and changes the state of that variable to the new state. And `checkState` takes the name of the state variable, a list of value-body pairs, and a fallback body, and 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 functionality granted by these high-level design pattern functions was already possible with GOOL's other functions. But they are useful because they are tailored to specific design patterns, so they are concise, for example by not requiring the user to manually define a loop for notifying observers, and their syntax should be easily understood by those familiar with OO programming.

6 Related Work

We divide the Related Work into the following categories

- General-purpose code generation

- Multi-language OO code generation
- Design pattern modeling and code generation

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, Haxe strips comments and generates source code around a custom framework which users would not be familiar with, so the generated code is not very readable.

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 [13]. GOOL's finally-tagless approach and syntax for high-level tasks, on the other hand, helped it overcome differences between target languages.

ThingML [11] 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.

Moving on to OO-specific code generators with multiple target languages, there are many examples of such DSLs, but for more restricted domains than GOOL. 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 [19]. Clearwater is an approach for implementing DSLs with multiple target languages for components of distributed systems [20]. The Time Weaver tool uses a multi-language code generator to generate "glue" code for real-time embedded systems [9]. The domain of mobile applications is host to a bevy of DSLs with multiple target languages, of which MobDSL [14] and XIS-Mobile [17] 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] (include this?). All of these are examples of multi-language code generation, but none of them generate general-purpose code like GOOL does.

A number of languages for modeling design patterns have been developed. The Design Pattern Modeling Language (DPML) [15] is similar to the Unified Modeling Language (UML) but designed specifically to overcome UML's shortcomings to be able to model all design patterns. DPML consists of both specification diagrams and instance diagrams for instantiations of design patterns, but does not attempt to generate actual source code from the models. The Role-Based Metamodeling Language [12] 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 behaviour in patterns. Again, source code generation is not attempted. Another metamodel for design patterns includes generation of Java code [4], and IBM developed a DSL for generation of OO code based on design patterns [6]. IBM's DSL was in the form of a visual user interface rather than a programming or modeling language. The languages that generate code do so only for design patterns, not for any general-purpose code like GOOL does.

7 Future Work

Currently GOOL code is typed based on what kind of code it represents: variable, value, type, or method, for example. Code that represents a variable or value is not further typed based on what the type of the variable or value would be in a traditional programming language: Boolean or integer, for example. There is nothing to stop a user from passing a non-list value to a function specifically intended for lists, like `listSize`, or from passing string values to numeric operators like `#+`. We plan on adding this additional layer of typing, making GOOL a statically typed programming language. We have started work to this end by making the underlying types for GOOL's `Variables` and `Values` Generalized Algebraic Data Types (GADTs), such as this one for `Variables`:

```
data TypedVar a where
  BVr :: VarData -> TypedVar Boolean
  IVr :: VarData -> TypedVar Integer
  ...
```

Members of the type family `Variable repr` would now map to `TypedVar` and the type for a `Variable` in the generic `repr` context would now be something like `repr (Variable repr Boolean)` instead of just `repr (Variable repr)`. All instances of `TypedVar` are built from the same `VarData` structure, but variables built with the different GADT constructors will have different types and Haskell's compiler will throw errors if a wrongly-typed variable is passed to a function.

There are many improvements we plan on making to the generated code, especially with regards to not generating code that is not necessary. For example, we currently always generate import statements for certain libraries, like math and IO-related libraries, but we'd instead like to detect when a library is used and only import those that are actually used.

We plan on using Haskell's `State` monad to have a list of libraries that gets updated whenever a new library is used, and can then be read by the `buildModule` function to generate only the necessary imports. This technique of using the `State` monad will be useful for other improvements as well, such as to only generate `throws Exception` in the header of a Java method that actually may throw an exception, rather than in every method as is done currently.

We plan on adding syntax to interface with more kinds of external libraries, similar to how we currently interface with math libraries with functions like `sin`. We would like to be able to interface with libraries for solving Ordinary Differential Equations (ODEs), for instance, with different functions available for using different ODE-solving algorithms. Since interfacing with ODE-solving libraries can have major impact on the structure of a program, implementing these high-level functions may require having multiple passes over a GOOL program: an initial pass to collect information, such as the information that an ODE library is used, and then a second pass to generate the code.

Currently GOOL forces certain design decisions on the user, but we plan on providing configuration options to give the user more control over the code they generate. For example, GOOL uses `ArrayLists` to represent lists in Java, but there are many other list implementations that could be used, such as `Vector`. This is something we would like the user to have control over. Another example, building on the aforementioned plan to provide functions to interface with more external libraries, is to allow the user to choose which specific external library they want to be used. These kinds of choices could be accomplished by having the user pass a configuration (such as a Haskell record) to GOOL, which GOOL will then read to decide how the code should be generated.

GOOL's current set of high-level functions for generating common OO patterns is by no means exhaustive. We plan to continue to identify patterns for which GOOL can provide abstractions, and to implement those as more high-level functions, with the aim of writing OO programs in GOOL as efficiently as possible without losing expressivity.

8 Conclusion

OO programming languages are similar enough that it is feasible to have a single OO language that can be compiled to any other OO language. GOOL is a DSL with multiple target languages for the domain of OO GPLs. Like most DSLs, GOOL provides abstractions suited to its domain, in this case abstractions of common OO patterns. Unlike most DSLs, GOOL considers the code it generates to be a product for human consumption in addition to computer consumption, so it focuses on generating idiomatic, human-readable, and documented code. The goal of generating idiomatic code is helped by the abstractions: GOOL provides syntax for expressing OO patterns naturally and efficiently, and these high-level functions afford each target language renderer the freedom to generate code following its own idioms. The goal of generating documented code is realized by GOOL's syntax for generating informal documentation in the form of code comments, or more formal documentation in the form of Doxygen-style structured comments for functions, classes, and modules. GOOL can even generate Doxygen configuration files and makefiles to facilitate compiling the documentation into PDF or HTML formats. The generated code is pretty-printed so that it is readable, and GOOL allows organization of code related to the same task into blocks to further increase readability. The idiomaticity and presence of documentation in the generated code also contribute to readability.

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] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. 2001. Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*.

[5] 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.

[6] 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.

[7] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2009), 546–558.

[8] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

[9] 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.

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

[11] 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.

[12] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. 2003. A uml-based metamodeling language to specify design patterns. In *Proceedings of Workshop on Software Model Engineering (WiSME), at UML 2003*. Citeseer.

[13] 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.

[14] 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.

[15] David Mapelsden, John Hosking, and John Grundy. 2002. Design pattern modelling and instantiation using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc., 3–11.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.